Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Collision detection and avoidance during trajectory tracking for F1/10 autonomous car model

Tomáš Nagy

Supervisor: Ing. Jaroslav Klapálek
May 2021

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Nagy Tomáš** Personal ID number: **483574**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Cybernetics and Robotics**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Collision detection and avoidance during trajectory tracking for F1/10 autonomous car model

Bachelor's thesis title in Czech:

Detekce a předcházení kolizím při sledování trajektorie pro model autonomního auta F1/10

Guidelines:

1. Get familiar with Robot Operating System (ROS) and project F1/10.
2. Perform an analysis of car trajectory tracking algorithms.
3. Implement at least three different trajectory tracking approaches and compare their performance on the F1/10 platform. Focus your comparison on an evaluation of tracking error.
4. Extend the functionality of the F1/10 model by detection of obstacles on a tracked trajectory.
5. Implement a program for collision avoidance. This program should support there two modes:
6. Switching to reactive control (e.g., Follow The Gap). When the program detects that it is safe to continue with the trajectory tracking, reactive control is turned off.
a. Local replanning of the trajectory (i.e., temporary substitution of a part of the trajectory that leads to the collision).
b. Extend the algorithm comparison (from point 3) with situations when the implemented program is active. Focus on tracking error evaluation when the mode switching occurs.
7. Evaluate your results and document everything thoroughly.

Bibliography / sources:

[1] M. Quigley et al., 'ROS: an open-source Robot Operating System', presented at the ICRA Workshop on Open Source Software, 2009, vol. 3.
[2]S. Dominguez, A. Ali, G. Garcia and P. Martinet, "Comparison of lateral controllers for autonomous vehicle: Experimental results," 2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC), Rio de Janeiro, 2016, pp. 1418-1423, doi: 10.1109/ITSC.2016.7795743.

Name and workplace of bachelor's thesis supervisor:

Ing. Jaroslav Klapálek, Department of Control Engineering, FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

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

Date of bachelor's thesis assignment: **26.01.2021** Deadline for bachelor thesis submission: **21.05.2021**

Assignment valid until:
by the end of summer semester 2021/2022

Ing. Jaroslav Klapálek Supervisor's signature	prof. Ing. Michael Šebek, DrSc. Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor'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 bachelor'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 gratitude to my supervisor Ing. Jaroslav Klapálek, for his valuable advice during the writing of this thesis. I would like to also thank my family, friends, and colleagues for their support during my studies.

Declaration

I hereby declare that I worked on this thesis individually and listed all of the used information sources according to Methodical Guideline on Ethical Principles for College Final Work Preparation.

In Prague, 21st May, 2021.

Abstract

This bachelor thesis is devoted to the problem of trajectory tracking, obstacle detection, and obstacle avoidance. In the first section of this thesis, an analysis of obstacle avoidance algorithms was performed. In the next part, three trajectory tracking algorithms, an obstacle detection algorithm, and two obstacle avoidance algorithms were theoretically discussed and implemented. In the last section, all trajectory tracking algorithms, as well as all of the possible combinations of trajectory tracking and obstacle avoidance algorithm, were tested and compared.

Keywords: F1/10 car model, ROS, obstacle detection, obstacle avoidance

Supervisor: Ing. Jaroslav Klapálek

Abstrakt

Táto bakalárska práca sa zaoberá problémom sledovania trajektórie, detekcie prekážok na trajektórií a predchádzaniu kolíziam pre model autonomného auta F1/10. V prvej časti tejto práce bola vypracovaná analýza algoritmov sledovania trajektórie. V ďalšej časti boli teoreticky popísané a implementované algoritmy Pure Pursuit, Stanley and Lateral speed controller na sledovanie trajektórie, algoritmus detekcie prekážok na trajektórií a dva rôzne algoritmy na vyhýbanie sa prekážkam. V poslednej časti boli otestované a porovnané všetky sledovacie algoritmy ako aj všetky možné kombinácie sledovacieho a vyhýbacieho algoritmu.

Kľúčové slová: F1/10 model auta, ROS, detekcia prekážok, vyhýbanie sa prekážkam

Preklad názvu: Detekcia a predchádzanie kolíziam pri sledovaní trajektórie pre model autonomného auta F1/10

Contents

1 Introduction **1**

2 Literature review **3**
2.1 Algorithms for trajectory tracking 3
2.2 Algorithms for obstacle avoidance 4
2.2.1 Switching to reactive algorithm 4
2.2.2 Trajectory planning algorithms 5

3 Theoretical background **7**
3.1 Definitions 7
3.2 Kinematic vehicle model 8
3.3 Trajectory following 9
3.3.1 Pure Pursuit 9
3.3.2 Stanley method 11
3.3.3 Lateral speed controller 12
3.4 Obstacle avoidance 13
3.4.1 Switching to reactive algorithm 13
3.4.2 Planning new path 13
3.4.3 Rapidly exploring random trees - RRT* 13
3.5 Bezier curves 15

4 F1/10 platform **19**
4.1 Hardware 19
4.1.1 Lidar 19
4.1.2 VESC 20
4.2 Software 20
4.2.1 ROS 20

5 Implementation **21**
5.1 Longitudinal control 21
5.1.1 Trajectory representation in trajectory tracking algorithms . . . 22
5.1.2 Pure pursuit 23
5.1.3 Stanley 23
5.2 Obstacle detection 23
5.3 Obstacle avoidance 24
5.3.1 Switcher to FTG 24
5.3.2 RRT* planner 25

6 Experiments **31**
6.1 Scenarios description 31
6.1.1 Trajectory tracking scenarios 31
6.1.2 Obstacle avoidance scenarios 32
6.2 Longitudinal control results 34
6.3 Trajectory following 34
6.4 Obstacle avoidance 37

7 Future work **43**

8 Conclusion **45**

Bibliography **47**

Figures

3.1 Kinematic bicycle model 8
3.2 Pure Pursuit 10
3.3 Stanley method 11
3.4 Lateral velocity controller 12
3.5 Comparison of RRT and RRT* path creation 15
3.6 Comparison of a cubic and a two-segment quadratic Bezier curve 16
3.7 Creation of the path between two poses using two quadratic Bezier curves 16

4.1 F1/10 platform 20

5.1 Bisection method used on trajectory 22
5.2 Example of the occupancy grid . 24
5.3 Simplified switcher ROS node diagram 25
5.4 Example of creating a return path by FTG switcher 25

6.1 Photos of the track used for trajectory tracking experiments . . . 31
6.2 Scenario A 32
6.3 Scenario B 32
6.4 Photo of the track used for obstacle avoidance scenarios 33
6.5 Simple obstacle on trajectory . . . 33
6.6 Simple obstacle on trajectory . . . 33
6.7 Speed tracking results 34
6.8 Trajectory tracking results of the experiment A 35
6.9 Trajectory tracking results of the experiment B 36
6.10 Velocity tracking results of the experiment B 37
6.11 Result of obstacle avoidance on Scenario C using Pure Pursuit and RRT* planner 38
6.12 Result of obstacle avoidance on Scenario C using Stanley method and RRT* planner 38
6.13 Result of obstacle avoidance on Scenario C using LSC and RRT* planner 39
6.14 Result of obstacle avoidance on Scenario C using Pure Pursuit and FTG switcher 39
6.15 Result of obstacle avoidance on Scenario C using Stanley method and FTG switcher 39
6.16 Result of obstacle avoidance on Scenario C using LSC and FTG switcher 39
6.17 Result of obstacle avoidance on Scenario D using Pure Pursuit and RRT* planner 40
6.18 Result of obstacle avoidance on Scenario D using Stanley method and RRT* planner 40
6.19 Result of obstacle avoidance on Scenario D using Pure Pursuit and FTG switcher 41
6.20 Result of obstacle avoidance on Scenario D using Stanley method and FTG switcher 41
6.21 Result of obstacle avoidance on Scenario D using LSC and FTG switcher 41

Tables

3.1 Parameters used in kinematic bicycle model 8

4.1 F1/10 car component list 19
4.2 Parameters of lidar Hokuyo UST-10LX 20

5.1 Parameters of used occupancy grid 24

Chapter 1

Introduction

As the popularity of self-driving cars is slowly on the rise over the past few years, this area of study is becoming more and more critical. Even competitions with full-scale autonomous cars are starting to emerge like Indy autonomous challenge [1] and Roborace [2].

There are also competitions that are using a scaled model of the autonomous car. The advantages of the scaled car are cheaper parts, easier algorithm testing with the possibility to scale algorithms for a full-size car, and crashes are less problematic, etc.

Our school has a team that competes in one of the autonomous racing competitions with a scaled model. This competition is called F1/10 Autonomous Racing Competition [3]. The F1/10 competition is a worldwide competition mainly between university students. It is based upon four pillars: build, learn, race, research. It was originally founded at the University of Pennsylvania in 2016. In the beginning, there was only one race category, in which the competitors have to build an autonomous F1/10 car that should complete a simple course without obstacles in the best time possible. Our team was successful in this competition with the reactive Follow The Gap method. But the goals and rules of the competitions are constantly changing, and new types of categories were introduced in the last few years. One of which is a course with static obstacles, to which this thesis is dedicated to.

We assume that we have an available reference trajectory on the course. In order for F1/10 platform to complete the course without crashing, we need to follow reference trajectory with good precision, detect obstacles on the trajectory and avoid them. Therefore the goal of this thesis was to develop a trajectory following and obstacle avoidance algorithm to have the chance to finish in the best place possible in the competition.

Chapter 2

Literature review

In this section we review state of the art approaches and different methods for problems of trajectory tracking and obstacle avoidance.

2.1 Algorithms for trajectory tracking

Since trajectory tracking is base for many ground robotic systems, a lot of research has been already done in the area.

Firstly. we mention the Pure Pursuit algorithm [4]. The first application of this method came with the Terragator, a six-wheeled skid-steered robot that was used for outdoor vision experimentation in the early '80s. As stated in [5], Pure Pursuit is a geometric algorithm that calculates the arc that will move a vehicle from its current position to some goal position. The position of the goal point is in some distance on the trajectory in front of the car. This distance is usually chosen proportional to the current speed. The name Pure Pursuit comes from the analogy that we use to describe the method. We tend to think that the vehicle is chasing a point on the path some distance ahead of it. That analogy is often used to compare this method to the way humans drive. The Pure Pursuit works well and is robust to large errors and discontinuous paths. However, at higher speeds, it starts to cut corners because of a big look-forward distance [6].

Another algorithm we can use is called Vector Pursuit [7]. This algorithm is based on the theory of screws [8], which was developed by Sir Robert Ball in 1876. The Vector Pursuit is very similar to the Pure Pursuit. The difference being that the Vector Pursuit also uses desired orientation of the vehicle in the goal position. The advantage of this algorithm with respect to the Pure Pursuit is that it is more robust with respect to the sensitivity of the look-ahead distance, and the ability to handle sudden large position and heading errors, but when the Pure Pursuit is tuned correctly, it can track trajectory with slightly better precision [7].

Stanley method was developed by Stanford University to compete in the DARPA Grand Challenge in 2005 [9]. The name Stanley comes from the name of their Volkswagen Touareg, which they entered the competition with. The Stanley method is also a nonlinear proportional controller like the Pure Pursuit but works on a different principle. This method chooses the goal

point as the closest point on the trajectory to the center of the front axle of the car. Then, it calculates the distance between those points, compares the orientation of the car to the desired orientation in the goal point, and determines the control output accordingly. As stated in [6], Even with this simple approach, Stanley can perform well. In contrast to the Pure Pursuit, a well-tuned Stanley tracker will not "cut corners", but rather overshoot turns because it does not have a look-ahead distance. The Stanley method is not very robust to large errors and non-smooth paths.

The Kinematic Lateral Speed Controller, further denoted as LSC, was developed to control the rear lateral distance and the orientation error by controlling the lateral speed, which is in turn controlled by the angular speed of the car through the steering angle [10]. The reason to develop the control method based on this principle was to maximize the comfort of passengers while still keeping reasonable precision and good stability. With lateral speed under control, this controller is able to turn smoother. In [10], LSC was this tracking method compared to the Pure Pursuit and Stanley method. Experiments were performed on the real car on the same track. Pure Pursuit had maximum error of 36 cm with 75 % of error measurements under 11 cm, Stanley method had maximum error of 40 cm with 75 % of error measurements under 9 cm, and finally lateral velocity controller had maximum error of 30 cm with 75 % of error measurements under 7 cm. In their testing, LSC performed well with reasonable tracking error and without sudden lateral movements, therefore producing a feeling of safety.

Model Predictive Control (MPC) is a large group of controllers based on the same principle [11]. MPC algorithms use kinematics and dynamics of the vehicle. Using these models and a few previous measured vehicle states, MPC tries to calculate optimal steering angle. One of the algorithms used by this group is the Linear Quadratic Regulator (LQR). This method was compared to the Pure Pursuit and Stanley method in the following study [6]. Algorithms in this paper were compared only in the simulation. The result was that LQR does not perform considerably better than kinematic methods. It had the smallest steady-state error but had poor robustness to disturbances. Also, the most significant disadvantage with respect to kinematic methods is that LQR is hard to tune properly.

2.2 Algorithms for obstacle avoidance

After detecting some obstacles on the trajectory, we have two possible methods of approaching this problem.

The first method is to switch to some reactive algorithm. The second one is to calculate a new trajectory.

2.2.1 Switching to reactive algorithm

This approach is one of the more straightforward methods to avoid obstacles. When there is some obstacle detected on a trajectory, the method switches

to reactive control and checks if the path back to the original trajectory is available. There are few reactive algorithms we can use.

The first reactive algorithm we can use is a simple following of the left or the right wall. This method can be used in a simple environment, but it has a problem with more complex obstacles.

Next algorithm is called Follow the Gap [12] (FTG). It tries to find a sudden rise of the distance between neighboring angles in the lidar scan, which can indicate the start or the end of a gap between obstacles. Then it tries to find a gap based on parameters like width or depth of it. Based on this, it calculates the steering angle to navigate the vehicle to that area.

2.2.2 Trajectory planning algorithms

One of the methods we can use to solve this problem is the potential field method [13]. In this algorithm, we create a potential function in which the start point is in an area of higher potential than a goal point, and also obstacles are in areas with a higher potential. After determining the potential function, we use the gradient descent method to find the goal point. This method can always find a solution if some exist, but it is computationally costly.

Next method is called RRT [14]. This method belongs to the category of sampling graph-based methods. It is an iterative algorithm where we create a random node somewhere in space and then connect it to the closest node in a tree structure. This method can always find a solution, but an optimal solution is almost never achieved. Its biggest advantage is that it can explore areas quickly, and after the tree is created, we follow parent nodes from goal to find a solution.

RRT* [15] is based on previous method. In this algorithm, we also optimize existing paths in the tree with newly created nodes to achieve asymptotic optimality.

Another sampling graph-based method we can use is the probabilistic roadmap method [16]. This algorithm firstly generates a number of random nodes in the environment. Then, it tries to connect them while avoiding obstacles. The advantage of this algorithm is that we can run it once, and we obtain a path from every point to every point in the environment. This method can achieve the optimal solution as the number of nodes approaches infinity is computationally expensive. We still need to find a path within the created graph, unlike in RRT.

To find the path in a discretized environment, we can also use search graph-based methods. To this group belongs algorithms like Dijkstra [17] and A* [18]. Dijkstra algorithm was created by Dijkstra (1959). It finds the shortest path to a goal and the shortest paths from a start to all other vertices of graph [19] using the cost function. A* algorithm is based on the Dijkstra algorithm, and apart from cost between different nodes, it also uses some heuristic function, usually Euclidean distance to the goal. This helps to A* to reach the goal in less time.

Chapter 3

Theoretical background

In this section we discuss chosen methods for the trajectory tracking, obstacle detection, and obstacle avoidance. We also discuss reasons why a specific methods were chosen and their mathematical principles.

3.1 Definitions

State of the robot. State of the robot is a tuple (x, y, φ, a, v, ϕ), where x and y is the current position, φ is and orientation in the environment, a is an acceleration, v is a velocity, and ϕ is a steer angle.

Pose. Pose of the robot is a tuple (x, y, φ), where x and y is the current position, and φ is and orientation in the environment.

Path. Path is an array of path poses.

Path pose. Path pose is the pose that the robot should have at a particular point on the path.

Trajectory. The trajectory is array of trajectory poses.

Trajectory pose. The trajectory pose is the state that the robot should have at a particular point on the trajectory. The state of the robot on the trajectory is a tuple (x, y, φ, a, v, κ), where x and y is the position, φ is the orientation, a in the acceleration, v is the velocity, and κ is the trajectory curvature.

Curvilinear coordinate system. This thesis will refer to this as a coordinate system where one axis is defined by the reference path and the second axis is defined by the norm to every point on the path.

Cross track error. Cross track error is a trajectory tracking error defined as a distance from the car position to the closest point on the trajectory.

Frame. Frame defines a coordinate system. It is defined by a tuple [O, X, Y], where O is an origin, X is first axis, and Y is a second axis.

Path resolution. Path resolution defines how dense are path poses sampled.

3.2 Kinematic vehicle model

As a model of the car is used a simple kinematic bicycle model [10]. This model is accurate only in lower speeds on non-slippery road [6], but the biggest advantage of this model is simpler computation. This played a major role in choosing this algorithm because we have computation power limited by competition rules as described in Chapter 4.

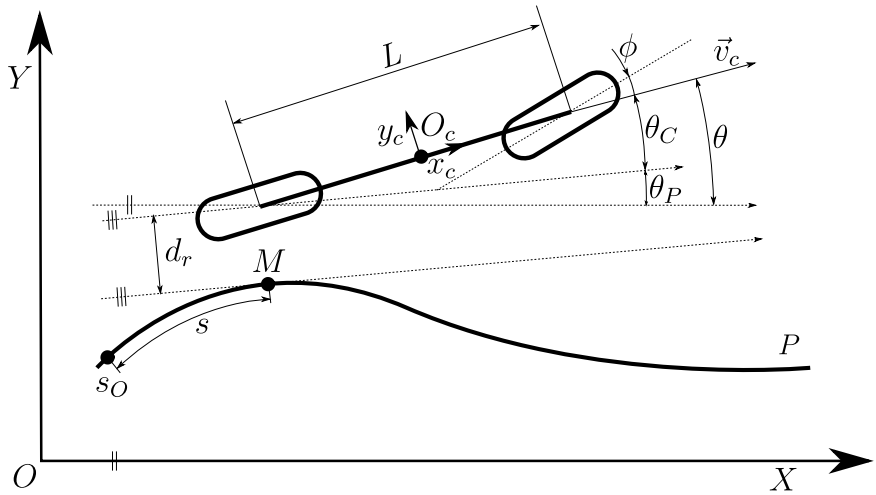


Figure 3.1: Kinematic bicycle model. Parameters are defined in Table 3.1.

Parameter	Explanation
$[O, X, Y]$	map frame
$[O_c, x_c, y_c]$	car frame
P	reference path
M	point on the reference path P closest to the center of the rear axle
s_o	origin of the curvilinear coordinate system defined by P
s	curvilinear abscissa which defines point M in s_O
d_r	distance between M and the center of the rear wheel
L	length of the wheelbase
θ_p	angle between the orientation of the P in M and the map frame
θ_C	angle between the orientation of the car and P in M
θ	angle between the orientation of the car and the map frame
v_c	speed of the car
ϕ	steering angle

Table 3.1: Parameters used in kinematic bicycle model

This system can be mathematically written with respect to the trajectory [10] as

$$\begin{cases} \dot{s} = \dfrac{\cos(\theta_c)}{1 - c(s)d_r}\|\vec{v}_c\| \\ \dot{d}_r = \sin(\theta_c)\|\vec{v}_c\| \\ \dot{\theta}_c = \left(\dfrac{\tan(\phi)}{L} - c(s)\dfrac{\cos(\theta_c)}{1 - c(s)d_r}\right)\|\vec{v}_c\| \end{cases}, \tag{3.1}$$

where $c(s)$ is curvature of P in M and v_u, ϕ are inputs into the system.

To get steering angle ϕ from Eq. (3.1), the model is linearized using exact linearization method [20], by defining a new input W_1 to solve Eq. (3.1). Finally the steering angle can be computed as

$$\phi = \arctan\left(L\left(\frac{W_1}{\|\vec{v}_c\|} + c(s)\frac{\cos(\theta_p)}{1 - c(s)d_r}\right)\right), \quad \|\vec{v}_c\| \neq 0, \tag{3.2}$$

where W_1 is the new linearized input to the system, where

$$W_1 = \dot{\theta}_c, \tag{3.3}$$

where $\dot{\theta}_c$ is an angular velocity.

3.3 Trajectory following

From review Section 2.1, we selected three different algorithms, which we theoretically discuss in this section.

3.3.1 Pure Pursuit

The Pure Pursuit [21] is one of the most common trajectory following algorithms [6]. The advantage of this algorithm is that it does not need a high resolution path. But this also means that it is stable during trajectory change.

As shown in Fig. 3.2, this algorithm finds a point, also called the goal point G, on a reference path, which

$$d(G, c_r) = l_d, \tag{3.4}$$

where $d(a, b)$ is distance between a and b and c_r is a center of the rear axle. Then it tries to steer towards that point, using the path defined as a circular arc with the radius R, calculated by Eq. (3.6), between the centre of the rear axle and the goal point.

Therefore firstly, we need to find the radius of this circular arc. We begin with the equation

$$\frac{l_d}{\sin(2\alpha)} = \frac{R}{sin(\frac{\pi}{2} - \alpha)}, \tag{3.5}$$

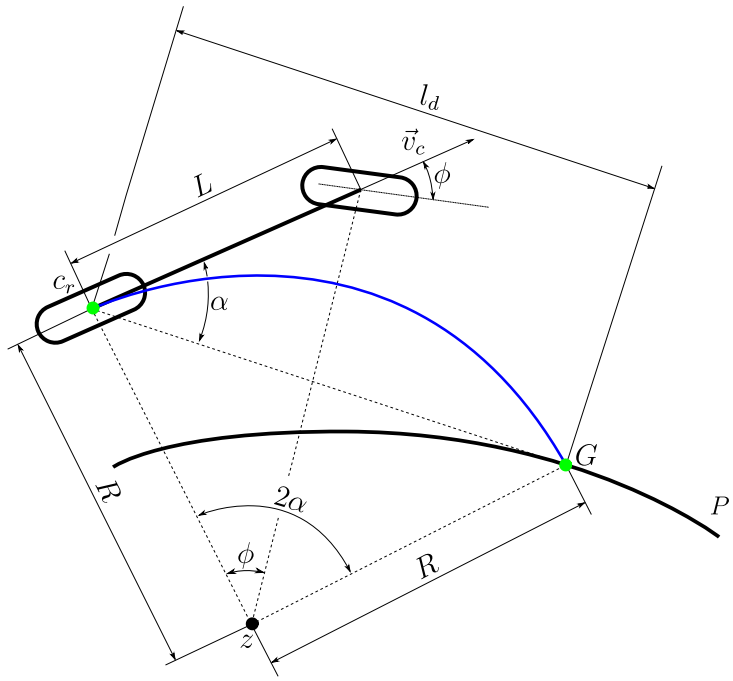


Figure 3.2: Pure Pursuit

which we can easily obtain from Fig. 3.2 using the law of sines. Then, we derive R, which gives us

$$R = \frac{l_d}{2\sin(\alpha)}. \tag{3.6}$$

Then, we can write the steering angle from Fig. 3.2 as

$$\phi = \arctan\left(\frac{L}{R}\right). \tag{3.7}$$

Finally, combining Eq. (3.6) and Eq. (3.7) gives us the Pure Pursuit control law as

$$\phi(t) = \arctan\left(\frac{2L\sin(\alpha(t))}{l_d}\right), \tag{3.8}$$

where the look ahead distance which is usually proportional to the vehicle speed, therefore we can write

$$l_d = k\ \|\vec{v}_c(t)\|, \tag{3.9}$$

where k is gain of the look-ahead distance.

3.3.2 Stanley method

The basic Stanley control law consists of two parts

$$\phi(t) = P_1(t) + P_2(t), \tag{3.10}$$

$$P_1(t) = \psi(t), \tag{3.11}$$

$$P_2(t) = \arctan\left(\frac{k\ e_r(t)}{\|\vec{v}_c(t)\|}\right) \tag{3.12}$$

where ϕ is the steering angle, ψ is the angle between the orientation of the car and the path P at the point that is closest to the center of the front axle, e_r is the distance between the center of the front axle and the closest point on the path P, v_c is the speed of the car, and k is the control constant. The first part P_1 only aligns the orientation of the front wheels with the orientation of the path and the second part P_2 steer towards the path.

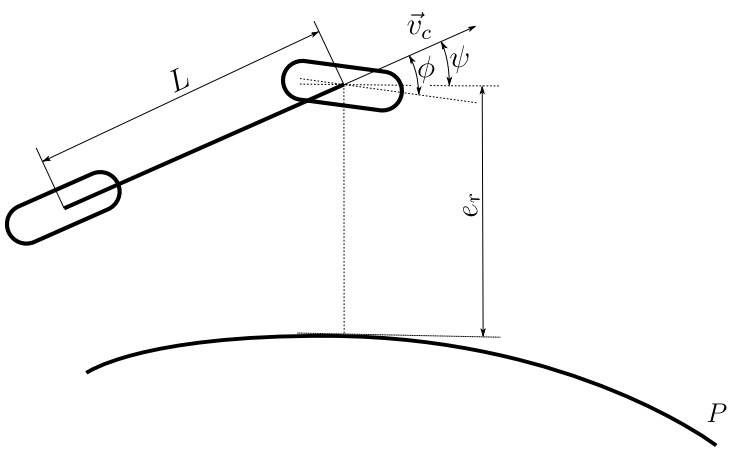


Figure 3.3: Stanley method

Disadvantage of this algorithm is that it needs a high resolution reference path, unlike the Pure Pursuit.

3.3.3 Lateral speed controller

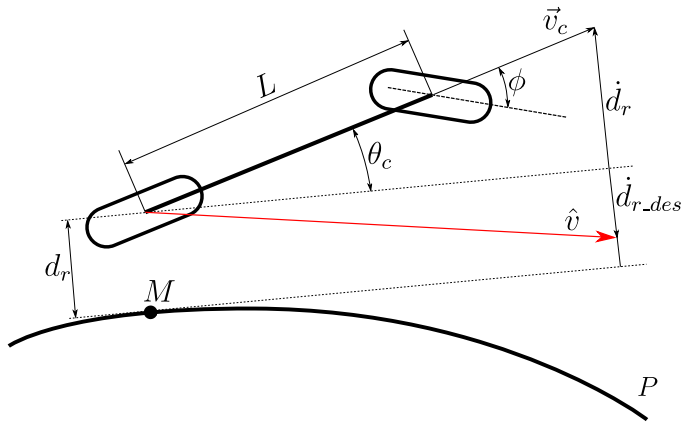


Figure 3.4: Lateral velocity controller

As mentioned in Section 2.1, LSC works by controlling the lateral velocity of the vehicle. To reduce the lateral error, the controller must steer the car towards the direction of the vector $\hat{v}$ as shown in Fig. 3.4.

When car is far from path, we want it to approach path more quickly then when it is close. So we can define desired lateral velocity proportional to distance from center of rear axle to closest point on path. Therefore

$$\dot{d}_{r\_des} = -k_{lat} d_r, \tag{3.13}$$

where d_r is the distance from the center of the rear axle to the closest point on a trajectory to the rear axle and k_{lat} is a constant defining how quickly should the car approach the path. From Eq. (3.1) we get the real lateral velocity of the car.

$$\dot{d}_r = \sin(\theta_c)\|\vec{v}_c\| \tag{3.14}$$

So we can write the lateral speed error $\dot{d}_{err}$ as

$$\dot{d}_{err} = \dot{d}_r - \dot{d}_{r\_des}, \tag{3.15}$$

$$\dot{d}_{err} = \sin(\theta_c)\|\vec{v}_c\| + k_{lat} d_r. \tag{3.16}$$

The control variable in Eq. (3.2) is proportional to the lateral error, so we can write

$$W_1 = K_\theta(\sin(\theta_c)\|\vec{v}_c\| + k_{lat} d_r), \tag{3.17}$$

where K_θ is the control gain. Finally, we get the lateral speed control law as

$$\phi = \arctan\left(L\left(K_\theta(\sin(\theta_c)) + \frac{K_\theta(k_{lat}d_r)}{\|\vec{v}_c(t)\|} + c(s)\frac{\cos(\theta_p)}{1 - c(s)d_r}\right)\right) \tag{3.18}$$

$\|\vec{v}_c(t)\| \neq 0.$

3.4 Obstacle avoidance

Based on the assignment, we have to choose two obstacle avoidance algorithms. The first algorithm is based on switching to some reactive algorithm; the second algorithm is based on computing a new path.

3.4.1 Switching to reactive algorithm

From reactive algorithms described in Section 2.2.1 we have chosen FTG because it is a nice compromise between the complexity of the algorithm and its ability to avoid obstacles. This algorithm was also already implemented in our F1/10 car.

3.4.2 Planning new path

In this thesis, we discuss path planning as we use an algorithm called trajectory profiler, which computes states for the path, connecting it into Trajectory. This algorithm is described in [22].

Based on the literature review, we have selected RRT* because of its ability to explore large areas and narrow, curved corridors very fast. Its description is in the next section.

3.4.3 Rapidly exploring random trees - RRT*

This algorithm starts with the start point and endpoint, and it tries to plan a path between these two points while avoiding obstacles. At first, we describe RRT [14] algorithm, which is the base of RRT* [15].

Pseudocode to RRT algorithm is shown in Algorithm 1. Inputs to this algorithm are: start point p_{start}, endpoint p_{end}, and the number of algorithm cycles k. Output is the path to the goal point. N is an array of nodes, where every node n has position, reference to a parent, reference to children nodes,

the cost to the parent, and cost to the root node.

Algorithm 1: RRT

```
Input  : p_start, p_end, k
Output : path
n_start ← init(p_start)                    ▷ Create start node from start pose
N ← n_start                                ▷ Init the tree N with start node
i ← 0
for i < k do
    p_rand ← samplePoint()
    n_par, p_rand ← chooseParent(p_rand)
    if isFree(n_par.point, p_rand) then
        N, n_rand ← addToTree(N, p_rand, n_par)
        N ← rewire(N, n_rand)              ▷ RRT* only
    end
    i ← i + 1
end
path ← findPath(N, p_end)
```

samplePoint(). The function samplePoint() creates a random point somewhere in the environment. With defined probability, it samples point precisely in the goal position. Therefore the tree is biased toward the goal.

chooseParent(p_{rand}). The function chooseParent(p_{rand}) finds the closest node in the tree to a randomly sampled point, which is chosen to be the parent node. If the randomly sampled point is closer to the parent node than the step distance, it does not change the position of the random point. If the random point is further away from the parent node, it changes the position of this random point to be in the same direction but in the step distance from the parent node. This functions returns corrected position of the randomly sampled point p_{rand} and chosen parent node n_{par}.

isFree(p_1, p_2). The function isFree(p_1, p_2) checks if there is an obstacle on the straight line between points p_1 and p_2.

addToTree(N, p, n_{par}). The function addToTree(N, p, n_{par}) creates a new node n with position p and adds it into the tree N with n_{par} as the parent node. This function returns created node n and updated tree N.

findPath(N, p_{end}). The function find path(N, p_{end}) finds node in the tree N that is in the position p_{end}. If such a node does not exist, the function returns an empty path. If the node is found, the function creates a path using all parent nodes up to the start node.

The algorithm RRT* works on the same principle as the RRT. The only difference is that function rewire() is added. This function tries to optimize the tree with a newly created node in order to create more optimal paths, as we can see in Fig. 3.5. Pseudocode to RRT* is shown in Algorithm 1.

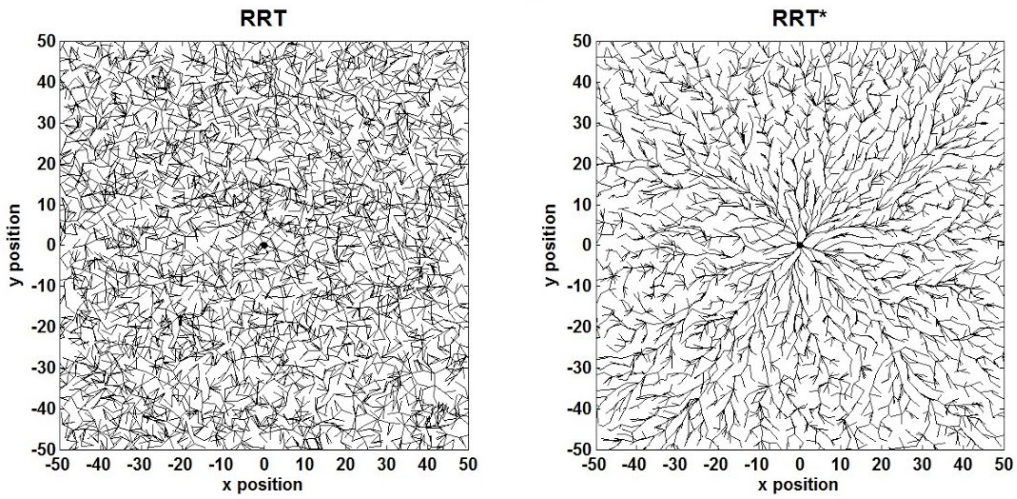


Figure 3.5: Comparison of RRT and RRT* path creation [23]. In both scenarios 4999 nodes are created.

3.5 Bezier curves

As discussed in Section 3.4.3, basic RRT* implementation checks for an obstacle between two nodes using a straight line. However, it is impossible for the car to follow the path created using straight lines correctly; therefore, we need to connect two nodes in the RRT* tree using a curve. We chose the Bezier curve because it has an easy mathematical representation and good differentiability [24]. The following theory is needed to understand our RRT* implementation in Section 5.3.2.

Calculation of the path from a pose to a point

To calculate a path from a start pose q with position p_q and orientation φ_q to an end point p_e, we use the quadratic Bezier curve with the following equation

$$B(t) = (1-t)^2 p_1 + 2(1-t)tp_2 + t^2 p_3,\ 0 \leq t \leq 1, \tag{3.19}$$

where p_1, p_2, and p_3 are control points that define the curve. Variation of the parameter t over the interval is used to calculate points of the curve. Control point p_1 defines start of the curve, therefore $p_1 = p_q$. Control point p_3 defines the end of the curve, so $p_3 = p_e$. Finally we calculate control point p_2 as:

$$p_2 = \begin{bmatrix} x \\ y \end{bmatrix} = p_q + d_k \begin{bmatrix} \cos(\varphi_q) \\ \sin(\varphi_q) \end{bmatrix}, \tag{3.20}$$

where d_k is proportional to the distance between points p_q and p_e, so

$$d_k = k_d ||p_q p_e||, \tag{3.21}$$

where k_d is a control constant. Constant k_d was experimentally set to $k_d = 0.5$ in order to create paths with an acceptable curvature gradient.

Calculation of the path from a pose to a pose

To create a path between a start pose and an end pose, we use two quadratic Bezier curves. It is also possible to use a cubic Bezier curve, but as we can see in Fig. 3.6, the path that was created using two quadratic curves has a smoother curvature gradient than the one that was created by a cubic Bezier curve using the same control points.

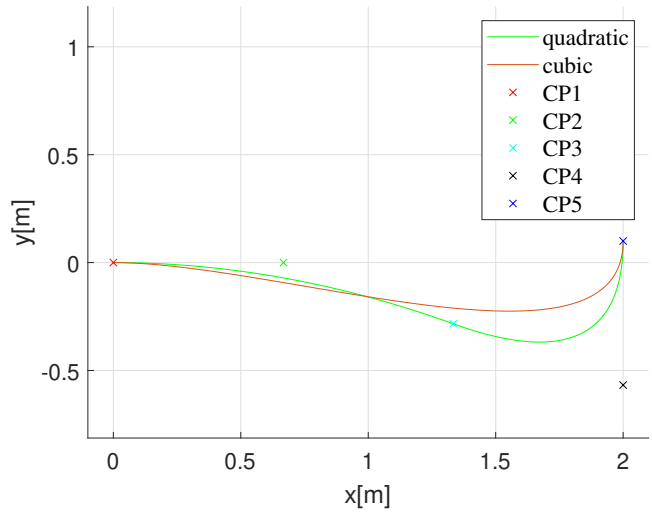


Figure 3.6: Comparison of a path created by a cubic Bezier curve and a path created using two quadratic Bezier curves. For a cubic bezier curve are used control points (CP) CP1, CP2, CP4, and CP5. For the first quadratic bezier curve are used control points CP1, CP2, and CP3 and for the second quadratic bezier curve are used control points CP3, CP4, and CP5.

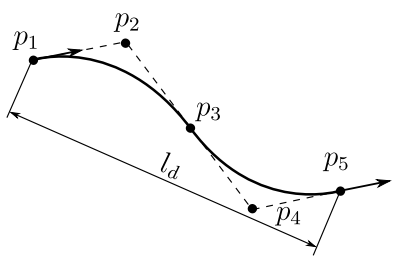


Figure 3.7: Creation of the path between two poses using two quadratic Bezier curves

As shown in Fig. 3.7, in order to create a path between two poses using two quadratic Bezier curves we need five control points. The first quadratic Bezier curve (Eq. (3.19)) consists of control points p_1, p_2, p_3 and the second quadratic Bezier curve is defined by control points p_3, p_4, and p_5. Point p_1 is a start point and p_5 is an end point. To achieve a path that has acceptable curvature gradient; control point p_2 is chosen in the direction of the start pose at a distance of $l_d/3$ and control point p_4 is chosen in the opposite direction

of the end pose at the distance of $l_d/3$. Finally the control point p_3 is created in the middle between points p_2 and p_4.

Calculation of the maximum curvature of a quadratic Bezier curve

When a path is created using a quadratic Bezier curves, we need to make sure that the car is able to follow it, because the car has limited turning radius. Therefore we need to find the maximum curvature of the quadratic Bezier curve. To achieve this we use following theorem[25].

Theorem 1. *We have quadratic Bezier curve* $B(t) = (1-t)^2 p_1 + 2(1-t)tp_2 + t^2 p_3$. *Let* A *be the area of the triangle* $p_1p_2p_3$ *and* m *be the midpoint of the segment* p_0p_2. *The maximum curvature of* $\boldsymbol{B}$ *is either equal to* $\frac{||p_2m||^3}{A^2}$ *if* p_2 *lies strictly outside the two disks of diameter* p_1m *and* mp_3, *or is equal to* $\max(\kappa_0, \kappa_1)$ *where* $\kappa_0 = \frac{A}{||p_1p_2||^3}$ *and* $\kappa_1 = \frac{A}{||p_2p_3||^3}$ *are the curvature of* $\boldsymbol{B}(t)$ *at the endpoints* $\boldsymbol{B}(0)$ *and* $\boldsymbol{B}(1)$.

Chapter 4

F1/10 platform

F1/10 car is a scaled-down model of a real car in the scale of 1:10. F1/10 cars are mainly developed for the F1/10 Autonomous Racing Competition [3], therefore they are built according to the F1/10 Autonomous Racing Competition. F1/10 competition forbids the use of any external computation and localization, so all sensors and a computer must be on the F1/10 car.

However, attending the competition is not the only purpose of this platform. This platform is also useful for the testing of algorithms that are developed for full-scale cars. The main advantages of testing the algorithms on a scaled model are: much safer environment, a controllable environment, and lower cost.

4.1 Hardware

The car is built upon Traxxas Slash 1:10 4WD, RC car chassis mainly used in RC hobby in scale 1:10 to a normal car. Rest of components are listed in Table 4.1.

Onboard computer	NVidia Jetson TX2
Lidar	Hokuyo UST-10LX
Engine	Velineon 3500
Engine controller	VESC
Servo	Traxxas 2075R
IMU	SparkFun 9DoF Razor IMU

Table 4.1: F1/10 car component list

4.1.1 Lidar

Lidar is one of the basic sensors which can detect the environment around the car. We use Hokuyo UST-10LX, which is a lightweight 2D LiDAR sensor. It is equipped with an Ethernet for high-speed measurement data. It also has low power consumption [26], therefore it is ideal for battery-powered robots. Parameters are shown in Table 4.2.

Horizonal angle	270°
Scanning frequency	40Hz
Angular resolution	0.25°
Working range	0.06m - 10m
Scanning range	0.6m - 4m
Connection	Ethernet

Table 4.2: Parameters of lidar Hokuyo UST-10LX

4.1.2 VESC

The VESC, Vedder Electronic Speed Controller[27], is named after its creator Benjamin Vedder. It is mainly used in electronic skateboards. The VESC is a speed controller for BLDC motors that has a build-in close-loop RPM regulator based on measuring the back EMF [28] of the BLDC motor. VESC also includes motor and battery protection, regenerative braking, and programming options, e.g., acceleration and deceleration curves.

4.2 Software

Software architecture is based on ROS1 Kinetic, which runs on Ubuntu 16. For localization and mapping, we use the algorithm Cartographer SLAM [29] which uses sensor fusion of different sources (IMU, lidar) to compute the probability of obstacles in the environment and position and orientation of the car within.

4.2.1 ROS

ROS, Robot Operating System, provides open-source libraries and tools to help software developers create robot applications [30]. Different processes (programs) in the ROS are called nodes. Nodes communicate through different topics via messages. Nodes can be written in Python or C++.

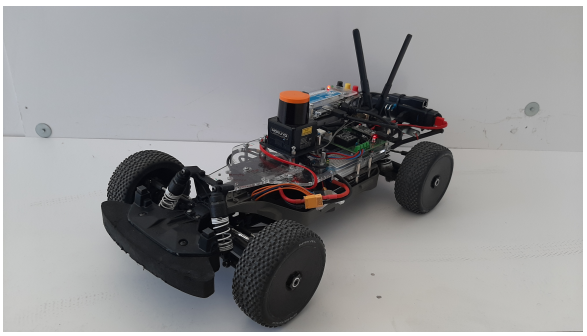


Figure 4.1: F1/10 platform

Chapter 5

Implementation

In this chapter we discuss implementation of the algorithms described in Chapter 3.

5.1 Longitudinal control

In order to achieve good lateral control over the vehicle, we need to have good longitudinal control. We have an available reference speed curve on the track from trajectory profiler (Section 3.4.2), and VESC can already keep it even with disturbances, as stated in Section 4.1.2. Therefore we implemented a simple integration ramp based on maximum acceleration and deceleration mainly to have better control over the vehicle during start. We control the car with velocities from the reference trajectory only if the car is closer than 40 cm to the reference trajectory. If the car is further away, it starts to slow proportionally to the distance, therefore

$$v_{des}(t) = \begin{cases} v_{ref}(t) & \text{if } c_e(t) < 0.4\,\text{m} \\ \frac{v_{ref}(t)}{\max(\min(1+0.1(c_e(t)-0.4)),1.1)} & \text{if } c_e(t) \geq 0.4\,\text{m} \end{cases} \tag{5.1}$$

where v_{ref} is reference velocity, v_{des} is desired velocity, and c_e is a cross track error. Function in Eq. (5.1) was designed in order for F1/10 platform to slow down when c_e is too high to regain control over the car. This works as a fail-save if the car happens to be unstable during testing.

Integration ramp is implemented as follows. First we calculate velocity error $v_{err} = v_{des} - v_{act}$, where v_{des} is desired velocity and v_{act} is current set speed. Then acceleration is calculated as $a_{act} = v_{diff}/t_{cycle}$, where t_{cycle} is time from last program cycle. Then we check saturation of accelerating to max and min value. Finally we calculate change of velocity and add it to current set speed as $v_{act} = v_{act} + a_{act}t_{cycle}$. The F1/10 platform has maximum speed set to 4.5 m/s, maximum acceleration to 0.9 m/s^2 and maximum deceleration to 4.5 m/s^2.

In FTG mode F1/10 platform has three available speeds based on steer

angle Θ, therfore

$$v_{act} = \begin{cases} 2.3 & \text{if } |\Theta| \leq 3\,\text{deg} \\ 1.9 & \text{if } |\Theta| \leq 10\,\text{deg} \\ 1.5 & \text{otherwise} \end{cases} \tag{5.2}$$

.

5.1.1 Trajectory representation in trajectory tracking algorithms

Trajectory poses are sampled 5 cm apart because if poses were sampled denser, e.g., 1 mm, it would be a problem on bigger maps because it would take longer to compute closest trajectory point to a car and also it would take longer to transfer bigger message between ROS nodes [31]. The 5 cm gap between trajectory poses is not an issue for the Pure Pursuit because it does not need a high-resolution trajectory; however, it poses a big problem for the Stanley method and LSC. To obtain intermediate points, we linearly interpolate three closest points on the reference trajectory and find the closest point using the bisection method. We also interpolate velocities, accelerations, curvatures, and orientations.

Bisection method

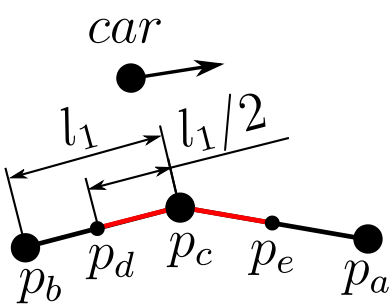


Figure 5.1: Bisection method used on trajectory

The bisection method works as follows. First, we find the closest point p_c on the reference trajectory. As shown in Fig. 5.1 p_b is the previous point to p_c on the reference trajectory, and p_a is the next point of the reference trajectory. This gives us the starting interval defined by points p_b, p_c, and p_a. Because p_c is the closest point, we can reduce this interval by creating two new points. We create a point p_d in the middle between p_b and p_c and the second point p_e in the middle between points p_c and p_a. Now we have five points. We find the closest point to the car from these five points and repeat this algorithm until we have reasonable precision. We are using five iterations of this method, because it is able to reduce error from 5 cm to approximately 0.15 cm.

5.1.2 Pure pursuit

Since Pure Pursuit oscillates at low speeds and cuts corners for higher speeds, we have constrained the look-ahead distance by minimal and maximal value. Therefore Pure Pursuit is more stable. These limits were set by experimental method to 0.4 m for a minimal look-ahead distance and 2.2 m for maximal.

5.1.3 Stanley

As mentioned in Section 3.3.2, the basic Stanley control law is defined as

$$\phi(t) = \psi(t) + \arctan\left(\frac{k\ e_r(t)}{\|\vec{v}_c(t)\|}\right). \tag{5.3}$$

Basic Stanley control law is defined by equation Eq. (3.12). However, the part $\psi(t)$ of this equation which should keep wheels parallel with trajectory, suffered from great oscillations. Therefore we added a parameter to this term to lower its influence on the steering angle, so it does not keep wheels parallel with trajectory anymore but only helps to steer towards it. To have even better performance, we added another term, which is a feed-forward based on curvature. Finally new control law is defined as

$$\phi(t) = k_1\psi(t) + \arctan\left(\frac{k\ e_r(t)}{\|\vec{v}_c(t)\|}\right) + k_2 \arctan(L\kappa(t)), \tag{5.4}$$

where L is is length of wheelbase, κ is curvature of the nearest trajectory point, and k_1, k_2 are control constants. These constants were set experimentally to $k_1 = 0.42$ and $k_2 = 0.61$.

5.2 Obstacle detection

Obstacle detection is implemented in C++. In this thesis, our goal is to avoid static obstacles on the reference trajectory with the F1/10 platform. To process the obstacles, we discretize the environment into an occupancy grid. The occupancy grid is a 2d binary map of the environment where the value indicates the presence of an obstacle in a grid cell. This grid is created in the car frame (Fig. 5.2), because it is computationally less expensive to convert few trajectory points from the map frame to the car frame.

To reduce the computation time required to detect if the F1/10 platform can be in some position, we inflate obstacles in the occupancy grid by half of the car's width. Therefore the F1/10 platform can be represented as a single cell in the occupancy grid. In our implementation the occupancy grid has parameters shown in Table 5.1.

Dimension of one grid cell	3×3 cm
Number of cells in X direction	180
Number of cells in Y direction	135
Distance in X	5.4 m
Distance in Y	4.05 m
Obstacle inflation	19.5 cm

Table 5.1: Parameters of used occupancy grid. In summary the car can see 5.4 m forward and 2.025 m to sides.

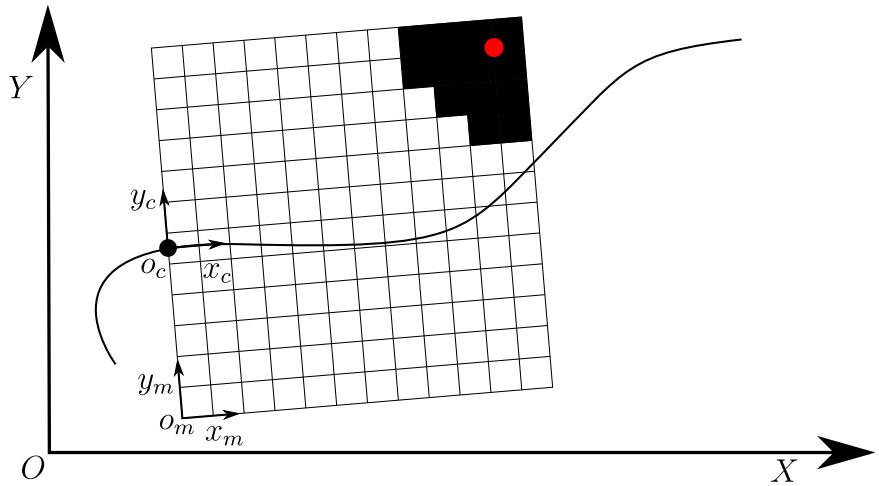


Figure 5.2: Example of the occupancy grid. $[O, X, Y]$ is a map frame, $[o_c, x_c, y_c]$ is a car frame and $[o_m, x_m, y_m]$ is an occupancy grid frame. White grid cells are free, and black cells are occupied. The red dot is a representation of some obstacle found by lidar. As we can see none of the occupied cells are on path, therefore obstacle is far enough from path and car can stay on the path.

5.3 Obstacle avoidance

Trajectory avoidance algorithms described in this section are implemented in C++ for the best performance.

5.3.1 Switcher to FTG

As was already mentioned in Section 2.2.1, when an obstacle is detected on the path, the program FTG switcher switches to the FTG algorithm.

The problem of this method is getting the car back on the reference trajectory after avoiding an obstacle. We solved this by creating a new path while in FTG until some path is free of obstacles. Then the path is sent through trajectory profiler (Section 3.4.2) to the trajectory follower algorithm, and FTG is turned off, as shown in Fig. 5.3. After the FTG switcher detected that the car is close enough to the reference trajectory, it sends the reference trajectory to the trajectory follower algorithm, and finally, the car is back on the reference trajectory.

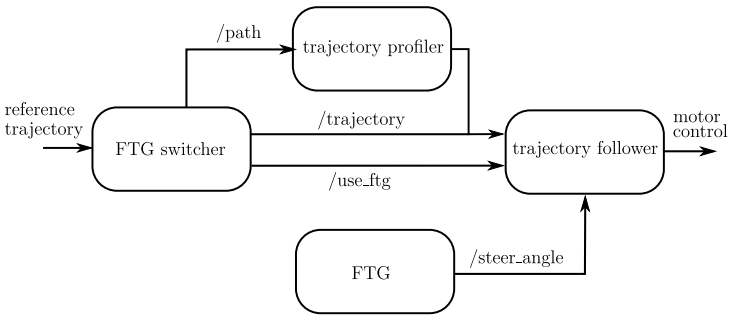


Figure 5.3: Simplified switcher ROS node diagram

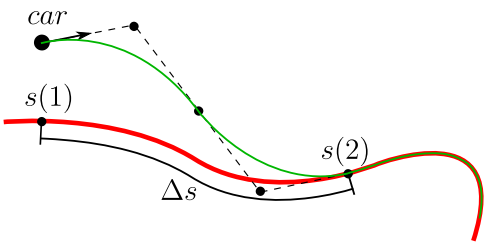


Figure 5.4: Example of creating a return path. The reference path is red, and the newly created path is green. To plan a return path, we first find a pose $s(1)$ on the reference path that is closest to the car. Then we find second pose $s(2)$ on the reference path that is 3 m forward from the closest pose in curvilinear coordinates, therefore $\Delta s = 3\,\mathrm{m}$. The $s(2)$ is our return pose. Next, we plan the path from the car pose to the return pose using the method described in Section 3.5. Finally, to have a smooth connection to the original path, we append the following 4 m of the reference path to our new path.

The path is sometimes created with high curvature, so the car is not able to follow it. We check this using method described in Section 3.5. If the car is not able to follow it, the path is thrown away and created again in the next program cycle.

5.3.2 RRT* planner

After an is detected on the trajectory, this method creates a new trajectory using the RRT* method. In this section, we describe issues encountered during the implementation and provide ways of solving them.

Path generation between nodes in the RRT*. The biggest problem of the RRT* algorithm (as described in 3.4.3) is that it creates paths that consist only of points and straight lines. In order for the car to be able to follow the path, we need to create the path using curves. Therefore we use Bezier curves described in Section 3.5. This way, our RRT* implementation does not work with positions but with path poses.

Step of the RRT*. In RRT* implementation (as described in 3.4.3) is defined maximum distance between a child node and a parent node by a step size. In our implementation of RRT*, we define two step constants. The first

constant is called step size. It is similar to the original RRT* implementation and is used to limit the maximum distance between the new node and closest node in the RRT* tree. The second constant is called neighborhood size and is used to define an area from which RRT* chooses a parent node for the new node and also to define an area that is optimized by a new node in the rewire() function. By defining two constants with different sizes, the RRT* has more parent options for a new node, and the area of the function rewire() can be different from the step size.

Sampling area. Another problem we encountered was when an obstacle was too big; it was impossible for the RRT* to find a path to the goal point. Therefore we create the sampling environment of the RRT* slightly bigger than the dimensions of the occupancy grid. This allows the RRT* to always find the goal point even when it is behind a large obstacle.

Biased node generation toward the last created path. During the implementation on the F1/10 platform, we found out that it had a problem with obstacles that were possible to avoid from both sides. The problem was that in the one iteration of the algorithm, the path was created correctly on the left side of an obstacle. But because the path was created close to the obstacle and because of the noise in the lidar data, the path was in the next program cycle evaluated as impassable. This time was a new path created on the right side. This sometimes happened few times in a row, causing the car to go straight and hitting the obstacle. We solved this by biasing node creation towards the last created path. The RRT* algorithm always saves the last created path, and during the creation of the next path, it samples nodes more often into the area defined by an inflation of the first half of the last created path and effectively stops the oscillation of the path.

Curved paths. One of the last problems of the RRT* algorithm is that because it is using random trees, a path that it creates usually has a lot of small curves. We tried to smooth them out by choosing few points on a new path and interpolating them again using bezier curves. However, this almost never succeeded because smoothed path usually passed through some obstacle or had some part that was unfollowable by the car. Nevertheless, we found a partial solution for paths that should bring the car back to the reference trajectory. Before running an RRT*, we use path creation from FTG switcher from Section 5.3.1. Then we run RRT* only if this path creation fails.

Failsave. When the car is in front of some obstacle close enough that the car is not able to avoid it anymore, RRT* creates only a few nodes because of the obstacle. Our program can detect this and stop the car, therefore avoiding a collision.

Changes to the algorithm stated above are shown in the implementation itself in the next section.

Our RRT* implementation

This implementation is based on lightweight, optimized C++ implementation from [32]. Its input arguments are: a start pose q_s, an end pose q_e, a run time t_{end}, and a last created path P_{last}. Output is a path P_{out}, as shown in Algorithm 2.

samplePoint(N, P_{last}). Inputs to this function are: node tree N, and the last created path P_{last}. Output of this function is a random pose q_{rand}. This function first generates random number m from a uniform distribution on interval <0; 1>. This number m determines from which area we are going to randomly choose points.

- If $m \in$<0; 0.1)
  The goal pose q_e is selected as a random pose q_{rand}.
- If $m \in$<0.1; 0.2)
  Random pose q_{rand} is generated from second area. This area is defined by an inflation of the last generated path P_{last} by 20 cm to the sides. In this case the orientation of the random pose q_{rand} is not fixed, so it is set to None and determined later in the algorithm.
- If $m \in$<0.2; 1>
  Apart from the RRT* implementation (as described in 3.4.3), this function also finds closest node in the tree N and changes the position of the random point q_{rand} to be in the step size distance from closest node.

isFree(q). The function isFree(q) checks whether the position of the pose q is not in the obstacle. This function returns True/False based on the occupancy of the position.

growTree(N, q). This function chooses a parent for a pose q from nodes in the area defined by distance of *neighborhood_size* from the pose q and adds in into the tree N, as shown in Algorithm 4. Function returns updated tree N and node n created from the pose q. If the creation of the node n was not successful the function returns an unchanged tree N and node $n =$ None.

steer(q_1, q_2). The function steer(q_1, q_2) creates a curve from *pose*1 to *pose*2 using bezier curve desribed in Section 3.5. It determines whether the orientation in q_2 is fixed and creates a curve between the pose q_1 and a position of the pose q_2 (Section 3.5) or between poses q_1 and q_2 (Section 3.5). It also determine if the car is able to follow the curve due to its limited turning radius using method desribed in (Section 3.5) and checks for obstacles on the curve. This function returns True if the curve creation was a success.

rewire(N, n). This function works on the same principle as the function rewire() in Section 3.4.3. It tries to optimize the tree N using the node n. The function rewire() finds all nodes in the tree in the distance called *neighborhood_size*. This function rewires the path using steer() function

described above, as shown in Algorithm 3. Output of this function is updated tree N.

updateCost(n, $cost$). This function updates costs in all children nodes of the node n using the new $cost$ of the node n.

findPath(N, q_e). Because sometimes the RRT* is not able to find a path to the goal pose q_e exactly, this function finds the closes node to the q_e, then finds all parent nodes up to start node, and finally samples all points between nodes using steer() function. It returns the created path P_{out}.

Algorithm 2: RRT*

```
Input  : q_s, q_e, t_end, P_last
Output : P_out
n_s ← init(q_s)                    ▷ Create start node n from start pose q_s
N ← n_s                            ▷ Init the tree N with the start node n_s
while timeNow() < t_end do
    q_rand ← samplePoint(N, P_last)
    if isFree(q_rand) then
        N, n_rand ← growTree(N, q_rand)
        if n_rand != None then
            N ← rewire(N, n_rand)
        end
    end
end
P_out ← findPath(N, q_e)
```

Algorithm 3: rewire()

```
input  : N, n_rand
output : N
                                   ▷ For every node n in the tree N
for n in N do
    segment_cost ← dist(n_rand.pose, n.pose)
    if segment_cost < neighborhood_size then
        if steer(n_rand.pose, n.pose) then
            cost ← n.root_cost + segment_cost
            if cost < cost_min then
                n.parrent ← n_rand
                updateCost(n, cost)
            end
        end
    end
end
```

Algorithm 4: growTree()

```
input  : N, q_rand
output : N, n_rand
cost_min ← ∞
n_par ← None
                                   ▷ For every node n in the tree N
for n in N do
    segment_cost ← dist(q_rand, n.pose)
    if segment_cost < neighborhood_size then
        if steer(n.pose, q_rand) then
            cost ← n.root_cost + segment_cost
            if cost < cost_min then
                cost_min ← cost
                n_par ← n
            end
        end
    end
end
if n_par != None then
    N, n_rand ← addToTree(N, p_rand, n_par)
end
```

Chapter 6

Experiments

In this section, we discuss the design of the experiments and their results of trajectory tracking and obstacle avoidance algorithms described in Chapters Chapter 3 and 5. Algorithms are tested on the F1/10 platform (Chapter 4) on various types of tracks and obstacles as described in Section 6.1.

To be able to compare fairly all of the algorithms, we have to note that the localization method we used has a maximum error up to 10 cm during fast driving and up to 3 cm during slow driving and stationary position [33].

6.1 Scenarios description

All testing tracks and scenarios described in this section were built in the Czech Institute of Informatics, Robotics, and Cybernetics (CIIRC) on the ground floor of building A. These experiments are designed to simulate possible scenarios during competition and to evaluate all algorithms fairly.

6.1.1 Trajectory tracking scenarios

Track shown in Fig. 6.1 is used for all trajectory tracking experiments. This track has a cross-section in the middle that allows us to test multiple trajectories without changing the whole track.

Figure 6.1: Photos of the track used for trajectory tracking experiments

Scenario A

The first scenario is a track shown in Fig. 6.2 that is designed to test the performance of tracking algorithms at high speeds.

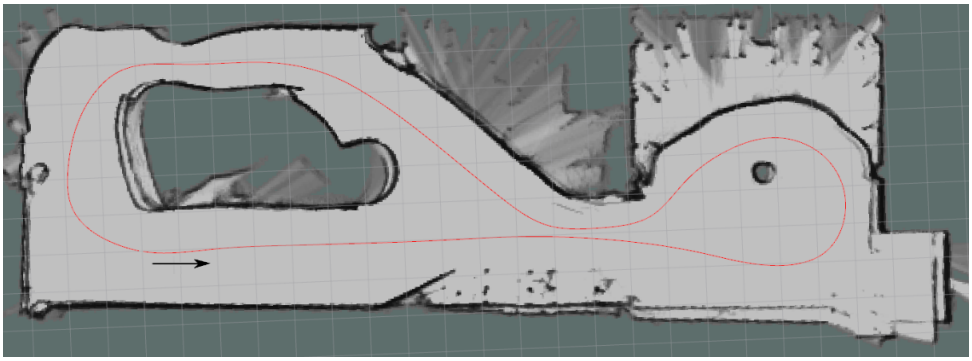


Figure 6.2: Scenario A

Scenario B

The second scenario shown in Fig. 6.3 is a small circuit with two 180 degree left-hand turns followed by a right-hand turn of small curvature. This track with constantly changing curvature without straight sections is suitable for stability testing of tracking algorithms.

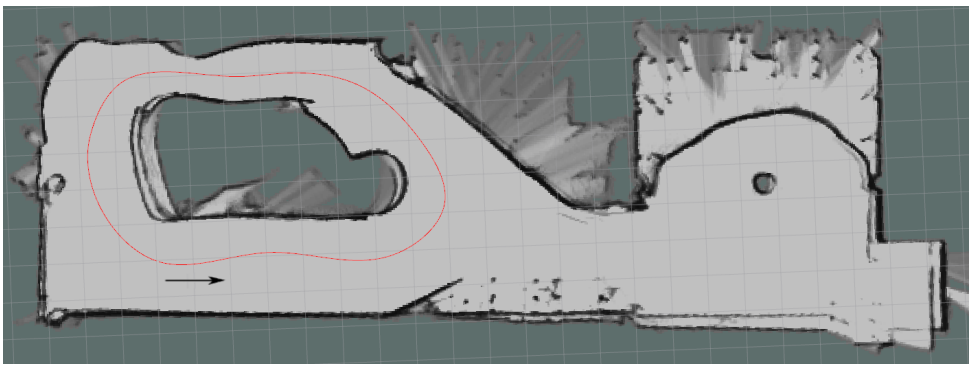
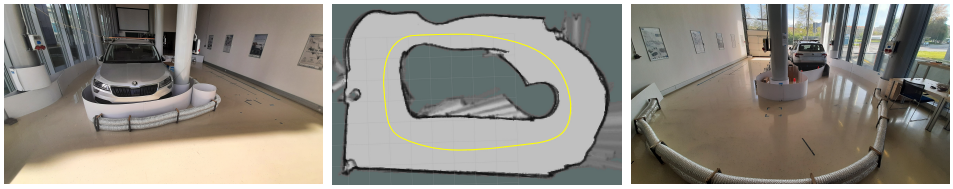


Figure 6.3: Scenario B

6.1.2 Obstacle avoidance scenarios

Track shown in Fig. 6.4 is used for all obstacle avoidance experiments. It is a track with a simple round reference trajectory, as shown in Fig. 6.4b.

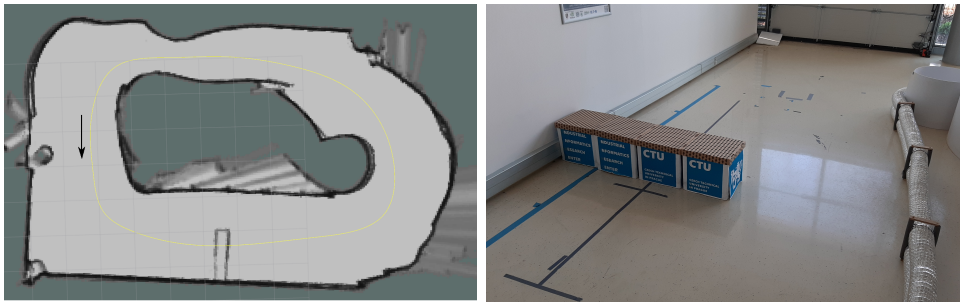


(a) : Photo of the track **(b) :** Reference path **(c) :** Photo of the track

Figure 6.4: Photo of the track used for obstacle avoidance scenarios

Scenario C

The first scenario is a simple obstacle with a 1.2 m gap for maneuvers as shown in Fig. 6.5. This obstacle should be easy to avoid for every obstacle avoidance algorithm.

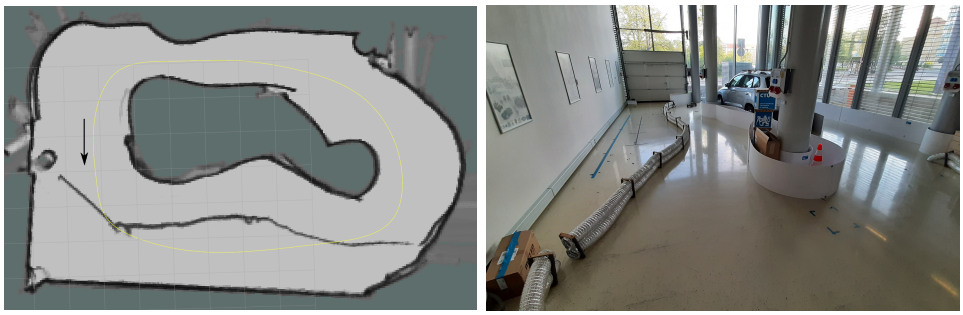


(a) : Map of the obstacle on a path **(b) :** Photo of the obstacle on a track

Figure 6.5: Simple obstacle on trajectory

Scenario D

The second scenario is a large obstacle that covers almost 7 m of reference trajectory as shown in Fig. 6.6. This is a challenging obstacle avoidance problem because of the large obstacle; the trajectory for avoidance can not be computed only once but needs to be constantly updated.

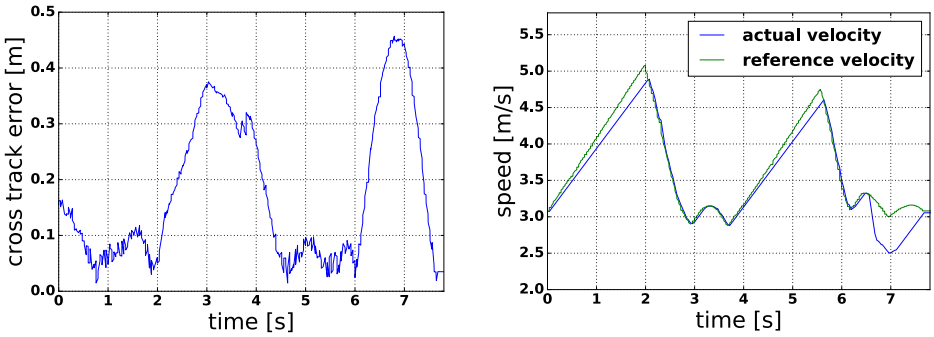


(a) : Map of the obstacle on a path **(b) :** Photo of the obstacle on a track

Figure 6.6: Simple obstacle on trajectory

6.2 Longitudinal control results

This experiment was designed to show that longitudinal control works as described in Section 5.1. We set the maximum acceleration of an F1/10 platform to 0.9 m·s^{-2}, although the acceleration limit of the reference trajectory was set to 1 m·s^{-2} to show that the acceleration limit works properly. Velocity on the trajectory was set to be too high to make the F1/10 platform unstable in turns, causing a cross track error to be higher than 0.4 m.



(a) : Cross track error **(b) :** Speed tracking

Figure 6.7: Speed tracking results

As shown in Fig. 6.7b, in the parts with long acceleration, longitudinal controller accelerates 0.1 m·s^{-2} slower then reference acceleration. As shown in Fig. 6.7a in the time of 6.5 s the F1/10 platform crosses cross track error limit causing F1/10 platform to slow down as shown in Fig. 6.7b.

Therefore the results described in the previous paragraph show that the longitudinal controller works as expected.

6.3 Trajectory following

In this section, we present experimental results of tracking algorithms on scenarios described in Section 6.1.1. Trajectory tracking algorithms are compared based on two metrics. The first metric is a maximum cross track error. The second metric is a value defined by a maximum cross track error of 75 % of the lowest values. The second metric is used to show the precision of algorithms without sudden big spikes.

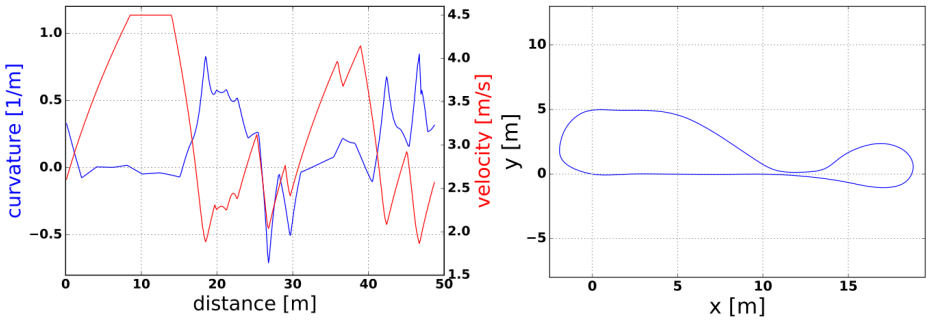
Experiment A

Experiment was conducted on Scenario A Section 6.1.1. Reference speed and curvature are shown in Fig. 6.8a.

In Fig. 6.8c is shown that the maximum tracking error of the Pure Pursuit algorithm is 0.122 m, with 75 % of values under 0.072 m. Stanley method has

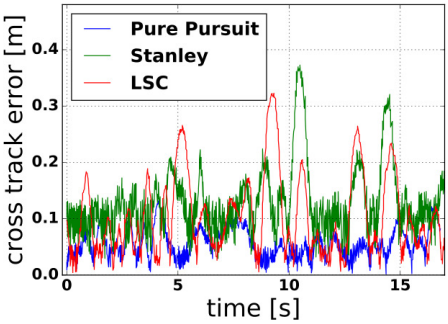
maximum tracking error of 0.357 m, with 75 % of values under 0.151 m. LSC has maximum tracking error of 0.316 m, with 75 % of values under 0.138 m.

From the results described above and shown in Fig. 6.8 we can see that the Pure Pursuit has the best precision and the lowest value of the maximum cross track error. In Fig. 6.8c in the first 2.4 s is shown that the Stanley method is stable and has steady-state error of 0.09 m. Therefore we can see that the Pure Pursuit performed almost two times better than Stanley and LSC. Results of Stanley and LSC were similar, with slightly better performance of the LSC.

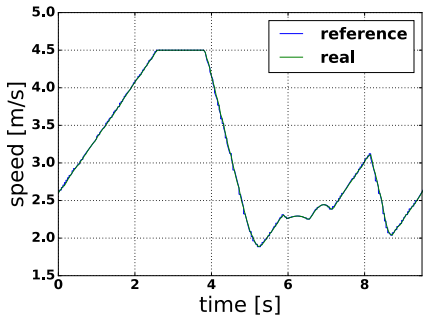


(a) : Reference curvature and velocity

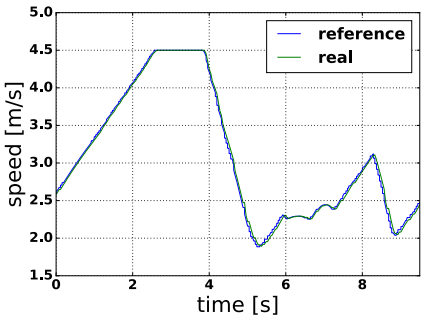
(b) : Visualization of the trajectory



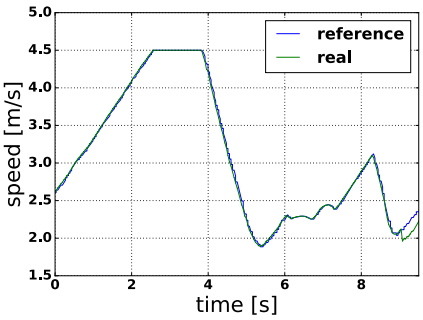
(c) : Cross track error



(d) : Velocity tracking using Pure Pursuit



(e) : Velocity tracking using Stanley



(f) : Velocity tracking using LSC

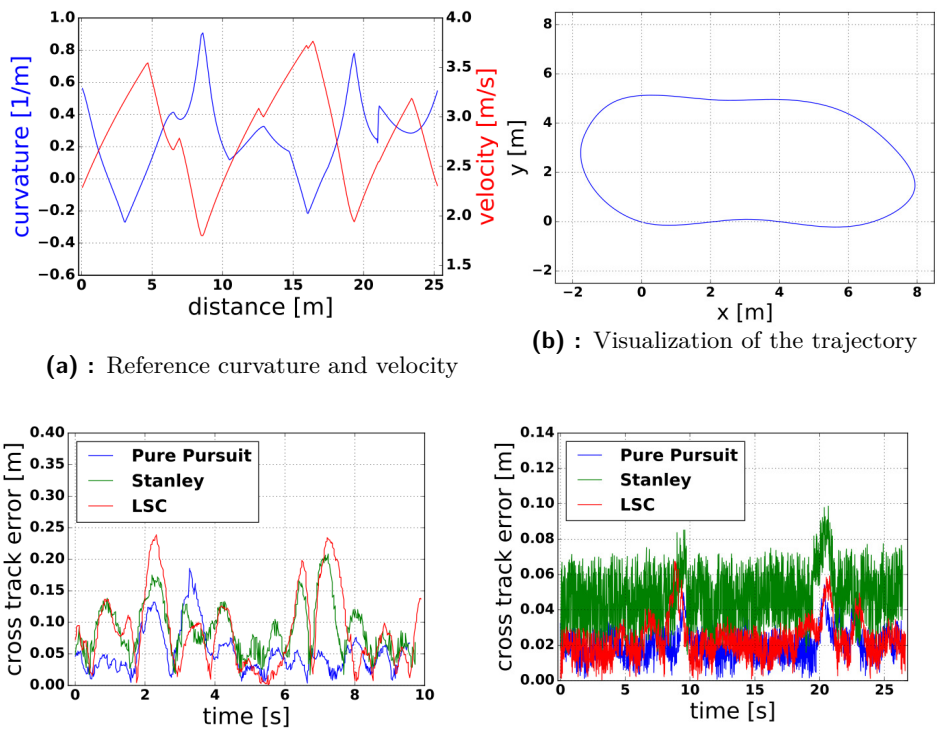
Figure 6.8: Trajectory tracking results of the experiment A

Experiment B

Experiment was conducted on Scenario B Section 6.1.1. Reference velocity and curvature are shown in Fig. 6.9a.

As shown in Fig. 6.9c the Pure Pursuit has maximum tracking error of 0.168 m, with 75 % of values under 0.062 m. Stanley has maximum tracking error of 0.201 m, with 75 % of values under 0.116 m. LSC has the biggest tracking error of 0.232 m, with 75 % of values under 0.121 m.

From the results above and in Fig. 6.9c we can see that the Pure Pursuit performed better than the Stanley method and LSC. The Stanley method has the worst maximum tracking error and also the worst precision. Also, we have to note that the maximum tracking error of the Pure Pursuit is caused by a sudden jump in localization as shown in Fig. 6.9c in 3.3 s. The error suddenly jumped from error value of 14 cm to 18 cm. The Pure Pursuit was able to get back on the reference trajectory in 0.5 s an is an acceptable result.



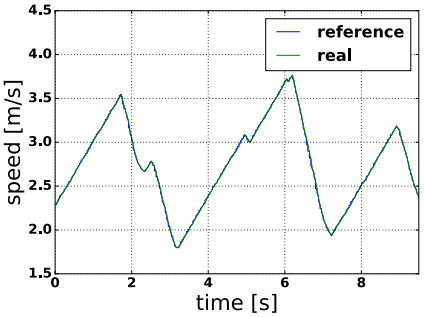
(a) : Reference curvature and velocity

(b) : Visualization of the trajectory

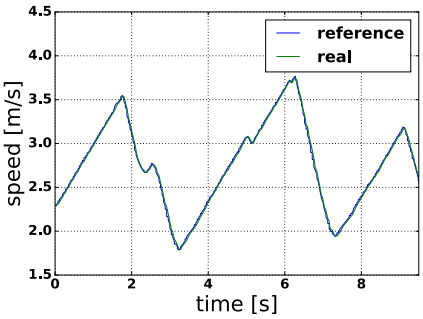
(c) : Cross track error with reference speed

(d) : Cross track error with speed of $1\,\mathrm{ms}^{-1}$

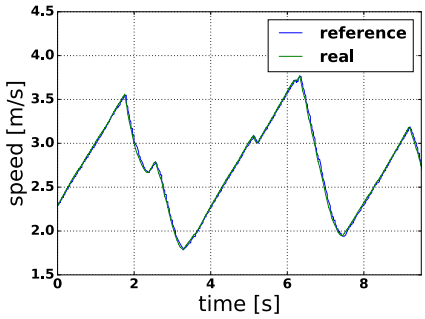
Figure 6.9: Trajectory tracking results of the experiment B



(a) : Velocity tracking using Pure Pursuit



(b) : Velocity tracking using Stanley



(c) : Velocity tracking using LSC

Figure 6.10: Velocity tracking results of the experiment B

This experiment was also repeated with a constant speed of 1 ms^{-1} to compare cross track error of algorithms in respect to the speed of the F1/10 platform.

Results of this experiment are shown in Fig. 6.9d. Pure Pursuit has maximum cross track error of 0.045 m, with 75 % of values under 0.027 m. Stanley method has maximum cross track error of 0.088 m, with 75 % of values under 0.061 m. And LSC has maximum cross track error of 0.058 m, with 75 % of values under 0.029 m.

From the results described above and shown in Fig. 6.9d we can see that the cross track error of all algorithms has improved. This big difference in precision is caused by latency in our system, which dramatically affects performance at higher speeds, mainly in the Stanley method and LSC. In this last experiment, we can also better see the noise caused by localization which is ±3 cm in this speed, as shown in Fig. 6.9d.

6.4 Obstacle avoidance

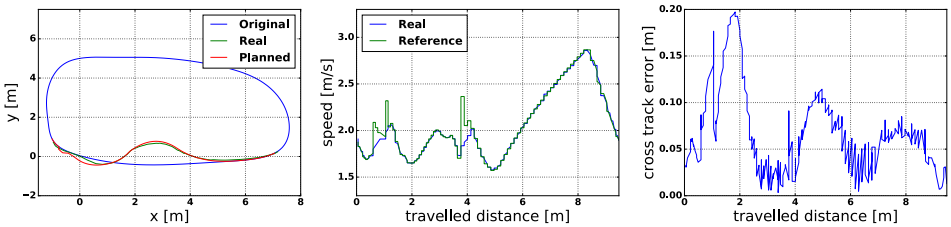
In this section, we compare the performance of every combination of the tracking and avoidance algorithms presented in this thesis on the scenarios shown in Section 6.1.2.

Experiment C

Experiment was conducted on Scenario C Section 6.1.2.

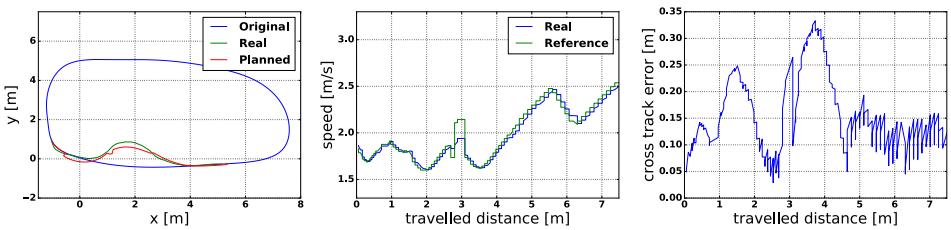
In every graph of the cross track error of FTG switcher (Fig. 6.14c, Fig. 6.15c and Fig. 6.16c), we can see a gap in data. This gap is caused by switching to the FTG algorithm because when F1/10 platform uses FTG, we can not measure the distance from the reference trajectory.

All of the methods avoided this simple obstacle successfully, as we can see on the graphs below. The least amount of cross track error has a combination of FTG switcher and Pure Pursuit, as shown in Fig. 6.14c. We can see in Fig. 6.13c in the travelled distance of 1.4 m that RRT* plans a new trajectory in order to reduce cross track error. This does not always work perfectly as shown in Fig. 6.12c in travelled distance of 3.1 m. The connection back to the reference trajectory from the FTG algorithm is always smooth (Fig. 6.14c, Fig. 6.15c, Fig. 6.16c).



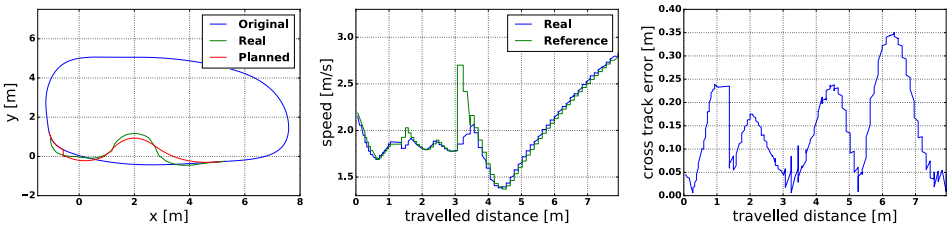
(a) : Paths **(b) :** Speed **(c) :** Tracking error

Figure 6.11: Result of obstacle avoidance on Scenario C using Pure Pursuit and RRT* planner



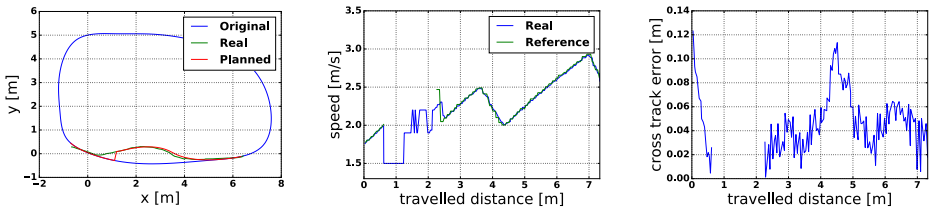
(a) : Paths **(b) :** Speed **(c) :** Tracking error

Figure 6.12: Result of obstacle avoidance on Scenario C using Stanley method and RRT* planner



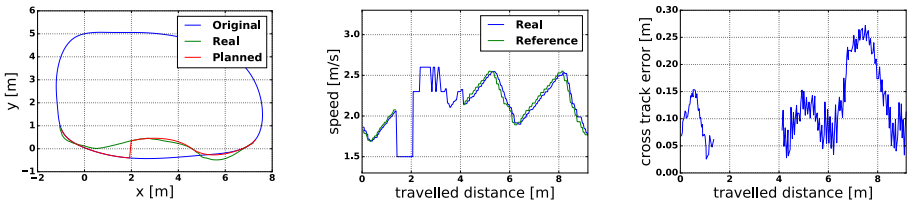
(a) : Paths **(b) :** Speed **(c) :** Tracking error

Figure 6.13: Result of obstacle avoidance on Scenario C using LSC and RRT* planner



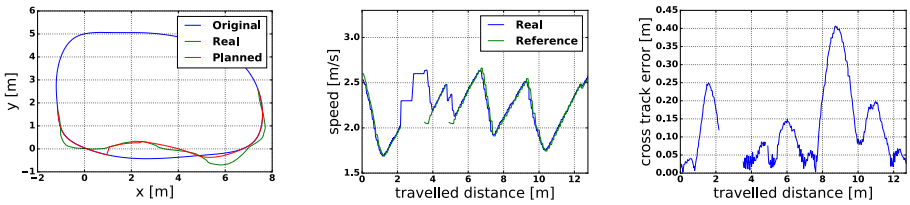
(a) : Paths **(b) :** Speed **(c) :** Tracking error

Figure 6.14: Result of obstacle avoidance on Scenario C using Pure Pursuit and FTG switcher



(a) : Paths **(b) :** Speed **(c) :** Tracking error

Figure 6.15: Result of obstacle avoidance on Scenario C using Stanley method and FTG switcher



(a) : Paths **(b) :** Speed **(c) :** Tracking error

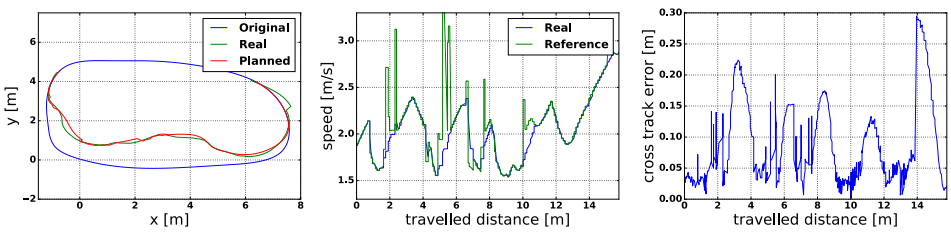
Figure 6.16: Result of obstacle avoidance on Scenario C using LSC and FTG switcher

Experiment D

Experiment was conducted on Scenario D Section 6.1.2.

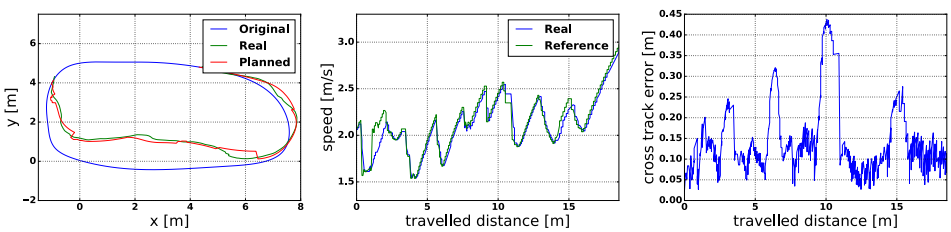
In the experiment results below, we did not include a combination of the RRT* planner and LSC because it collided with an obstacle. All other combinations avoided the obstacle successfully. As shown in Fig. 6.17 and Fig. 6.18 in this scenario RRT* algorithm replanned trajectory quite often. As already mentioned in the results of the previous experiment, RRT* plans trajectory in the best way possible to reduce tracking error. The reason this does not always work is that RRT* has a runtime of 60 ms. When RRT* starts to compute a new trajectory, the position of the F1/10 platform differs from the position that RRT* works with. This can sometimes cause instability in trajectory tracking during trajectory switching. Another disadvantage of the RRT* planner is that it plans trajectories that often have a sudden change in curvature. All of the methods that use FTG switcher have a smooth connection to the trajectory when switching back from the FTG (Fig. 6.19c, Fig. 6.20c, Fig. 6.21c).

The combination that produced the least amount of tracking error is FTG switcher and Pure Pursuit (Fig. 6.19). Second best is RRT* planner with Pure Pursuit.



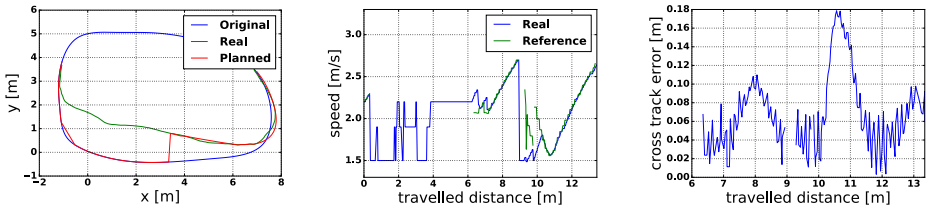
(a) : Paths **(b) :** Speed **(c) :** Tracking error

Figure 6.17: Result of obstacle avoidance on Scenario D using Pure Pursuit and RRT* planner



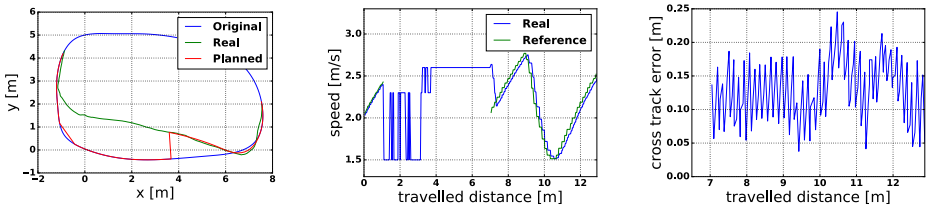
(a) : Paths **(b) :** Speed **(c) :** Tracking error

Figure 6.18: Result of obstacle avoidance on Scenario D using Stanley method and RRT* planner



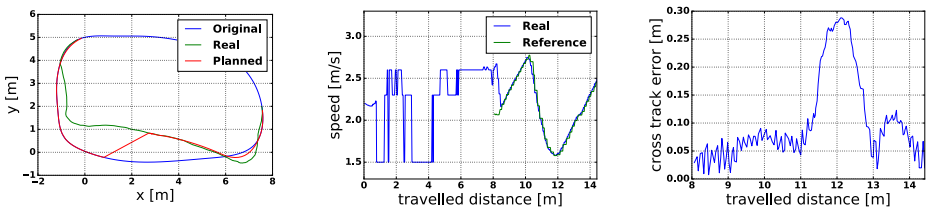
(a) : Paths **(b) :** Speed **(c) :** Tracking error

Figure 6.19: Result of obstacle avoidance on Scenario D using Pure Pursuit and FTG switcher



(a) : Paths **(b) :** Speed **(c) :** Tracking error

Figure 6.20: Result of obstacle avoidance on Scenario D using Stanley method and FTG switcher



(a) : Paths **(b) :** Speed **(c) :** Tracking error

Figure 6.21: Result of obstacle avoidance on Scenario D using LSC and FTG switcher

Chapter 7

Future work

In this chapter, we present a possible solution to the problem found in the algorithms.

C++ implementation of trajectory tracking algorithms

Trajectory tracking algorithms in this thesis are implemented in Python. Python applications are generally slower compared to, e.g., C++. Therefore, as a topic for future research, algorithms should be rewritten to C++ to increase their performance and compare them to prior Python implementation.

Upgrade to ROS2

As already mentioned earlier (Section 4.2.1), the F1/10 platform currently runs on ROS1. However, the team already conducted few experiments with ROS2, which seems to have better performance. Therefore it would be beneficial to port all of the algorithms to ROS2 and compare their differences in performance between ROS1 and ROS2.

Implement different planner algorithm

As found out in the experiments, the 60 ms runtime of computation of RRT* planner is sometimes not fast enough, especially at higher speeds. Also, another problem of the RRT* is that it often plans trajectories too curvy. It would be interesting to implement another planner, this time based on some graph search method like A*, because we expect it to plan more straight paths. So it would be easier for trajectory tracking algorithms to follow it.

Chapter 8

Conclusion

In the first part of this thesis, we performed an analysis of trajectory tracking and obstacle avoidance methods, from which we have chosen algorithms based on expected performance based on other studies.

We have implemented, tested, and evaluated Pure Pursuit, Stanley method, and Lateral velocity controller as algorithms for trajectory tracking on the F1/10 platform. Based on the experiments, we have found out that the Pure Pursuit is the most precise algorithm on our F1/10 platform, mainly because of the latency in our system.

F1/10 platform functionality was successfully extended to find obstacles on a tracked trajectory.

We have created two obstacle avoidance methods, switcher to the FTG algorithm and RRT* based planner, tested them and evaluated them in combination with every trajectory tracking algorithm described in the previous paragraph.

Based on the experiments, we have found out that the best results were achieved by a combination of FTG switcher and Pure Pursuit. However, RRT* planer with Pure Pursuit tracking method worked almost as well. The most significant advantage of the RRT* planner is that we have full control over the trajectory at any time during the drive.

Ideas for future work were presented in the previous chapter.

Bibliography

[1] "Indy autonomous challenge." https://www.indyautonomouschallenge.com.

[2] "World's first extreme competition of teams developing self-driving AI." https://roborace.com.

[3] "F1tenth." `https://f1tenth.org/`.

[4] O. Amidi and C. E. Thorpe, "Integrated mobile robot control," in *Mobile Robots V*, vol. 1388, pp. 504–523, International Society for Optics and Photonics.

[5] C. R. Craig, "Implementation of the pure pursuit path tracking algorithm," 1992.

[6] J. Snider, "Automatic steering methods for autonomous automobile path tracking," 2011.

[7] J. Wit, C. D. Crane, and D. Armstrong, "Autonomous ground vehicle path tracking," vol. 21, no. 8, pp. 439–449. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rob.20031.

[8] Robert Stawell Ball, *The Theory of Screws: A Study in the Dynamics of a Rigid Body*. Hodges, Foster.

[9] G. M. Hoffmann, C. J. Tomlin, M. Montemerlo, and S. Thrun, "Autonomous automobile trajectory tracking for off-road driving: Controller design, experimental validation and racing," in *2007 American Control Conference*, pp. 2296–2301. ISSN: 2378-5861.

[10] S. Dominguez, A. Ali, G. Garcia, and P. Martinet, "Comparison of lateral controllers for autonomous vehicle: Experimental results," in *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, pp. 1418–1423. ISSN: 2153-0017.

[11] C. E. García, D. M. Prett, and M. Morari, "Model predictive control: Theory and practice—a survey," vol. 25, no. 3, pp. 335–348.

[12] V. Sezer and M. Gokasan, "A novel obstacle avoidance algorithm: "follow the gap method"," vol. 60, no. 9, pp. 1123–1134.

[13] S. elia nadira, R. Omar, and C. K. N. Hailma, "Potential field methods and their inherent approaches for path planning," vol. 11, pp. 10801–10805.

[14] S. M. Lavalle, "Rapidly-exploring random trees: A new tool for path planning."

[15] S. Karaman and E. Frazzoli, "Incremental sampling-based algorithms for optimal motion planning,"

[16] L. Kavraki, M. Kolountzakis, and J.-C. Latombe, "Analysis of probabilistic roadmaps for path planning," vol. 14, no. 1, pp. 166–171. Conference Name: IEEE Transactions on Robotics and Automation.

[17] E. W. Dijkstra, "A note on two problems in connexion with graphs," vol. 1, no. 1, pp. 269–271.

[18] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," vol. 4, no. 2, pp. 100–107. Conference Name: IEEE Transactions on Systems Science and Cybernetics.

[19] J. A. Bondy and U. S. R. Murty, *Graph Theory With Applications*. Elsevier Science Ltd/North-Holland.

[20] H. Nijmeijer and A. van der Schaft, "Feedback linearization of nonlinear systems," in *Nonlinear Dynamical Control Systems* (H. Nijmeijer and A. van der Schaft, eds.), pp. 161–192, Springer.

[21] R. Wallace, A. T. Stentz, C. Thorpe, H. Moravec, W. R. L. Whittaker, and T. Kanade, "First results in robot road-following," in *Proceedings of 9th International Joint Conference on Artificial Intelligence (IJCAI '85)*, vol. 2, pp. 1089 – 1095, August 1985.

[22] D. Kopecký, "Localization and advanced control for autonomous model cars," 2019.

[23] Yiqun Dong, "What's the difference between RRT and RRT* and which one should we use."

[24] "Derivatives of a bézier curve." `https://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/Bezier/bezier-der.html`.

[25] H. Deddi, H. Everett, and S. Lazard, "Interpolation problem with curvature constraints," Publisher: Vanderbilt University press.

[26] "Ust-10lx." https://hokuyo-usa.com/products/lidar-obstacle-detection/ust-10lx.

[27] V. Benjamin, "VESC – open source ESC | benjamin's robotics." `http://vedder.se/2015/01/vesc-open-source-esc/`.

[28] C. Kiree, D. Kumpanya, S. Tunyasrirut, and D. Puangdownreong, "PSO-based optimal PI(d) controller design for brushless DC motor speed control with back EMF detection," vol. 11, no. 3, pp. 715–723. Publisher: The Korean Institute of Electrical Engineers.

[29] W. Hess, D. Kohler, H. Rapp, and D. Andor, "Real-time loop closure in 2d LIDAR SLAM," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1271–1278, IEEE.

[30] "kinetic - ROS wiki." `http://wiki.ros.org/kinetic`.

[31] X. Hu, "shm_transport/latency vs message size." `https://github.com/Jrdevil-Wang/shm_transport`.

[32] T. Henderson, "sportdeath/motion_planning." original-date: 2018-07-19T00:12:12Z.

[33] D. Zahrádka, "Optimization-based control of the f1/10 autonomous racing car." Accepted: 2020-09-04T13:58:32Z Publisher: České vysoké učení technické v Praze. Vypočetní a informační centrum.

[34] M. Samuel, M. Hussein, and M. B. Mohamad, "A review of some pure-pursuit based path tracking techniques for control of autonomous vehicle," vol. 135, no. 1, pp. 35–38. Publisher: Foundation of Computer Science (FCS), NY, USA.

[35] A. Patnaik, M. Patel, V. Mohta, H. Shah, S. Agrawal, A. Rathore, R. Malik, D. Chakravarty, and R. Bhattacharya, "Design and implementation of path trackers for ackermann drive based vehicles,"

[36] L. L. Scharf, W. P. Harthill, and P. H. Moose, "A comparison of expected flight times for intercept and pure pursuit missiles," vol. AES-5, no. 4, pp. 672–673. Conference Name: IEEE Transactions on Aerospace and Electronic Systems.

[37] A. Krause, "First results in robot road-following," in *The Robotics Institute Carnegie Mellon University*.

[38] J. L. Blanco, M. Bellone, and A. Gimenez-Fernandez, "TP-space RRT – kinematic path planning of non-holonomic any-shape vehicles," vol. 12, no. 5, p. 55. Publisher: SAGE Publications.

[39] S. Karaman, M. Walter, A. Perez, E. Frazzoli, and S. Teller, "Anytime motion planning using the RRT*," pp. 1478–1483.

[40] Aaron Becker, "RRT, RRT* & random trees."

[41] MATLAB, "Autonomous navigation, part 4: Path planning with a* and RRT."

[42] MATLAB, "Autonomous navigation, part 1: What is autonomous navigation."

[43] Y. Kim and H. Bang, *Introduction to Kalman Filter and Its Applications*. IntechOpen. Publication Title: Introduction and Implementations of the Kalman Filter.