Politecnico di Torino

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Automated driving for an UGV for agriculture 4.0

Supervisors:

Dr. Fabrizio Dabbene
Prof. Davide Ricauda Aimonino
Dr. Martina Mammarella

Candidate:

Mohammadreza Beygifard

July 2020

Acknowledgment

I would like to express my special thanks of gratitude to my supervisors, Dr. Fabrizio Dabbene, Prof. Davide Ricauda Aimonino, and Dr. Martina Mammarella who gave me the golden opportunity to do this wonderful thesis on the topic, Automated driving for a UGV for agriculture 4.0, which also helped me in doing a lot of research and I came to know about so many new things I am really thankful to them.

Secondly, I would also like to thank Dr. Arsham Ghavasieh who helped me a lot in this thesis and provided intelligent ideas for the thesis.

I appreciate the efforts my family did to make it possible for me to study abroad and helped me economically and emotionally during the course and the thesis.

And finally, I would like to send my special regards to EDISU Piemonte that provided me a generous scholarship with which I could continue living and studying in Italy.

Abstract

This thesis report presents an approach to create autonomy for a mobile robot and then focuses on the planning module of the autonomous four-wheel steering (4WS) mobile robot. The planning module is composed of a global path planner and a local path planner. The global path planner, working with Rapid Random exploring Tree (RRT), provides local goals to a local path planner, which is implemented with the Dynamic-Window approach (DWA). DWA is a receding horizon technique that can produce a smooth and optimized path. The path is later given as a reference signal to a motion controller that another student worked on.

Abbreviations

Abbreviation	full name
ASIC	Application Specific Integrated Chip
Agribot	Agricultural robot
CAGD	Computer Aided Geometric Design
CSPTP	Constrained shortest path tour problem with a time window
CNR	Consiglio Nazionale delle Ricerche
DISAFA	Dipartimento di Scienze Agrarie, Forestali e Alimentari
DWA	Dynamic Window Approach
FPGA	Field Programmable Gate Array
GPS	Global Positioning System
GPU	Graphics Processing Unit
IRC	Instantaneous Rotation Center
LIDAR	Light Detection and Ranging
PFM	Potential Field Method
RF	Reference Frame
ROS	Robotic Operating System
RRT	Rapid Random exploring Tree
SAE	Society of Automotive Engineers
UGV	Unmanned Ground Vehicle
4WS	Four Wheel Stearing

Contents

1 Introduction **2**
- 1.1 Robots in agriculture . . . 2
- 1.2 Research focus . . . 3
  - 1.2.1 Our robot . . . 3
- 1.3 Autonomy requirements . . . 4
  - 1.3.1 Levels of automation . . . 5
  - 1.3.2 Hardware . . . 6
  - 1.3.3 Software . . . 7

2 Planning **8**
- 2.1 Introduction . . . 8
- 2.2 Optimization . . . 10
  - 2.2.1 Parametric Optimization . . . 10
- 2.3 Path planning technique . . . 13
  - 2.3.1 State of the art . . . 13
  - 2.3.2 Dynamic Window Approach . . . 15
- 2.4 Final Path Planning Hierarchy . . . 16

3 Bicycle model **18**
- 3.1 Kinematic constraints . . . 18
  - 3.1.1 Non-holonomic constraints . . . 18
- 3.2 Kinematic model . . . 20
- 3.3 Bicycle model . . . 20
  - 3.3.1 Ackermann steering system . . . 20
  - 3.3.2 Four wheel steering system . . . 22
- 3.4 Numerical integration . . . 23

4 Global path planner **24**
- 4.1 Introduction . . . 24
- 4.2 RRT . . . 25
- 4.3 RRT* . . . 27

5 Local path planner **29**
- 5.1 Introduction . . . 29
- 5.2 Dynamic-Window Approach . . . 29
  - 5.2.1 Velocities and steering angles . . . 30
  - 5.2.2 Collision Checking . . . 32
  - 5.2.3 Cost Function . . . 33

6 Results and discussion **38**

7 Conclusion **46**

List of Figures

1.1 Our mobile robot. 4
1.2 Schematic of our robot wheel base. 4
1.3 Autonomy requirements hierarchy [1]. 5
1.4 SAE J3016 Levels of Driving Automation [2]. 6

2.1 The Desired Path. 8
2.2 The curvature of a curve [3] 9
2.3 The top view of the wheel. 9
2.4 Robot states representation. 11
2.5 A comparison between A* (left) and D* (center and right) [4]. D* can generate a smoother path. 13
2.6 A comparison between RRT* (left) and RRT (right) [5]. RRT generates more branches and is computationally heavier, while RRT* is faster and generates less branches. 14
2.7 Dynamic Window Approach. Several paths are generated and then through a cost function the collision is checked and the best path is chosen. 15
2.8 Dynamic Window Approach. This method can get stocked in the local minimum. 16
2.9 Inserting local goal manually. 17
2.10 Our Path Planning Hierarchy. 17

3.1 Generalized coordinates for a disk rolling on a plane [6]. 19
3.2 Bicycle model with Ackermann steering system. 21
3.3 Bicycle model with four wheel steering system (4WS) [7]. 22

4.1 DWA is a branch and bound method. 24
4.2 If the planner gets stocked in a local minimum the robot is stocked as it does not know any other path to follow. 25
4.3 First attempt of RRT. 26
4.4 RRT can take so time consuming if the map has curvy narrow corridors. 27
4.5 RRT* is much faster than RRT. 28

5.1 Sub-Trajectories generated by the chosen configuration. Notice that for them each Sub-Trajectory is according to a couple of linear and angular velocities, but in our case, each Sub-Trajectory is according to a set of four inputs. The inputs are V_f, V_r, δ_f, δ_r. 33
5.2 Schematic figures of cost function terms introduced by [8] 34

5.3 Comparison between the path generated with two cost functions on a small map. 35
5.4 Comparison between the path generated with two cost functions on a large map. 36
5.5 Comparison between DWA with and without receding horizon technique 37

6.1 Three main paths. 38
6.2 Map 1 paths. 39
6.3 cost function behaviour. 40
6.4 Map 2 paths with the initial heading of $3\pi/4$. 41
6.5 Map 2 paths with the initial heading of π. 42
6.6 Map 2 paths with the initial heading of π, with high resolution global path planner. 43
6.7 Map 3 paths. 44
6.8 The results of the controller on following the provided path with long and narrow corridors [9]. 44
6.9 The results of the controller on following the provided path on the farm [9]. 45

Chapter 1

Introduction

1.1 Robots in agriculture

The Robotic Industries Association (RIA) defines a robot as: "A robot is a re-programmable, multi-functional manipulator designed to move material, parts, tools or specialized devices through variable programmed motions for the performance of a variety of tasks." [10] Robots have served humans in several industrial applications, like material handling, material transfer, automobile manufacturing, inspection, and quality control. There are many successful stories of robotic applications in the industry; as an example, Sara Bragança et al. [11] have done a brief overview of the opportunities that collaborative robots have brought to industry 4.0 and how these robots can support workers.

In recent years there was a noticeable trend in applying robotics in agriculture. There are several reasons for employing robots in this field, like increasing food productivity and improving its quality, decreasing the labor cost, and the needed time to produce an agricultural product. As N.Vamshidhar Reddy et al. [12] discussed, there is also another significant motivation for using agricultural robots, which is a lack of sufficient skilled workers in this sector. Developing countries like India, which were studied by the mentioned author, are the most influenced nations by this defect. Therefore if agricultural robots empower farmers, there can be a significant improvement in food production for such countries.

The agricultural robot or "*Agribot*" is a robot used for agricultural purposes. As mentioned before, using such robots can reduce the required time and cost for agriculture productions, and raise the quality and quantity of such products. As N.Vamshidhar Reddy et al. [12] reports, there are several examples of successful applications of Agribots in seeding, harvesting, weed control, grove supervision, chemical applications, etc. The mentioned author suggests that the primary use of Agribots is at the harvesting stage; however, various research societies have applied different automation solutions (e.g., sensor networks, manipulators, ground vehicles, and aerial robots) to diverse agricultural missions. J.J. Roldán et al. [13] have studied many examples and discussed them in detail. But there are several challenges for industrialization of agriculture that is reported by most of these groups. On the one hand, the agricultural environments are not well structured and controlled as the industrial environments are. On the other side, the industrial tasks are modular, and different robots would perform each module. In contrast, the complexity of agricultural tasks does not permit of such a *luxury* for Agribots. We think that

there is another challenge besides all of the mentioned ones. In the industry, we have operators that are usually well trained to cope with robots, which are not the case in agriculture, especially in developing countries.

To sum up, we would say working on Agribots is attractive, especially for those ready to admit challenges and deal with them. The result of such researches is crucial for humanity to deal with the increasing need for food production.

1.2 Research focus

In this study we focus on creating autonomy for a four-wheel steering (4WS) mobile robot that Dipartimento di Scienze Agrarie, Forestali e Alimentari, DISAFA, of Università di Torino has devloped.

At first, this study aimed to understand the needed modules to create autonomy for a mobile robot. Then we focus on *path planning*. Our path planning aimed to generate a $feasible$ and *optimal* path for the robot and it contains a *global path planner* and a *local Path planner*. we choose the Dynamic-Window approach (DWA) for our local path planner, which is a receding horizon technique, and it can generate a smooth and optimized path for the robot through its cost function. However, the DWA technique can get stocked in the local minimum during path optimization. To solve this problem, we have a global path planner that generates local goals and saves DWA from getting stocked. As the Rapid Random exploring Tree (RRT) showed fast and reliable results in the previous studies at the literature, we decided to choose this technique as our global path planner.

To generate a feasible path, a path planner should respect the kinematic constraints of the robot. The so-called kinematic model is a mathematical description of those constraints, and both local path planner and global path planner uses this model to generate paths.

Both path planners have cost functions that we tuned through several tests. Finally, we provided an efficient, fast, and optimal planner module for our mobile robot and any other mobile robot with a 4WS wheel based locomotion.

While we developed the path planner, Ing. Simone Scivoli [9] was developing the *motion planner*, a controller that guides the robot. The path generated by us is provided as a reference signal to the controller. We cooperated to develop two comprehensive modules, and we tuned our cost functions and control gains together.

1.2.1 Our robot

Our robot, which is shown in figure 1.1 works in the agricultural field, so it is an agribot. As it uses wheels for movement and direction control, it is a wheel-based mobile robot. Since both front and rear wheels can turn to aid it during sharp turns, it is considered a 4WS mobile robot.

As shown in figure 1.2 the distance between each wheel in each axis, $Track$, is 1000 mm the distance between the front and the rear axis, $Wheel - base$ is 1500 mm and the center of gravity of the robot is located almost at the mid-length of Track and wheel-base.

Its maximum turning angle for each axis is 23°, which yields a total maximum turning angle of 46° if the front and rear axis turn by opposite angles. However, in designing the path planner, we assumed the maximum turning angles as 15° and

Figure 1.1: Our mobile robot.

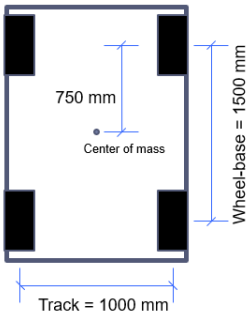


Figure 1.2: Schematic of our robot wheel base.

30° respectively to generate a smoother path and leave a safety bias and not put too much load on the robot axis.

1.3 Autonomy requirements

As we discussed in the previous section, our robot is a mobile robot. As Bruno Siciliano et al. [6] discussed in their book, mobile robots are chosen whenever we need a moving base to perform a service in the environment autonomously. They continued that a mobile robot contains one or more rigid bodies equipped with a locomotion system from a mechanical point of view. The locomotion system can be wheel based or leg based.

To drive a mobile robot autonomously, we need some hardware and software requirements. Roland Seigwart et al. [1] in their book, shared a hierarchy that summarizes all of the elements in one figure (figure 1.3).

One by looking at this hierarchy can easily understand how vast is the area of autonomous mobile robots, and studying each part of the hierarchy needs a considerable amount of time.

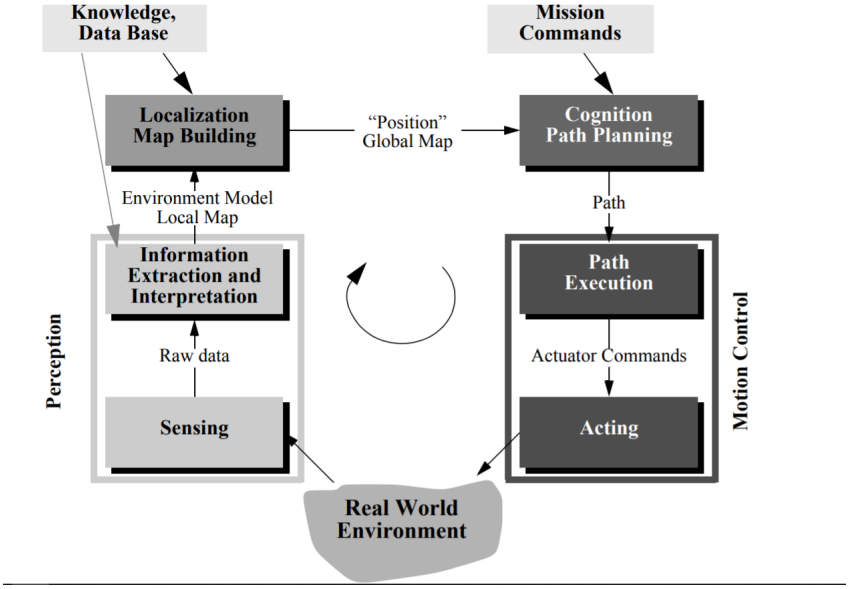


Figure 1.3: Autonomy requirements hierarchy [1].

To summarize the hierarchy, the robot needs a system to create a map for the robot itself. The mentioned system should also be able to detect the position of such a robot on the map. This system is named *Localization Map Building*. Later we need to define a mission for the robot, which can be a simple movement from one point of the map to the other point, which is defined as *Mission Commands*. These commands are given to a system, named *Cognition Path Planning*, which is responsible to define a $safe$ and *admissible path* for the robot to follow. This path is a reference signal to a controller, which is named *Path Execution*, controlling the behavior of the robot. This controller takes the path and generates *"Actuator Commands"* to govern the locomotion system of the robot. The locomotion system can have several low-level controllers, which are completely out of the scope of this study, and finally, the robot will change his state through *Acting*, which can be its location or just its orientation. The new state and possibly the new changes in the *Real World Environment* are sensed by the sensors through the *Sensing* process. The *Raw data* of sensors is given to $Information\ Extraction\ and\ Interpretation$ system, in which we can have some machine learning process. Finally the extracted *Environment Model Local Map* is returned to the *Localization Map Building* system. The robot repeats all of these processes until it reaches the goal of our mission.

As each of the mentioned systems can be challenging to obtain, we define different levels of automation for our robot. We usually do this procedure because of technological or economic limits. In the following sections, we discuss different automation levels and some of the required hardware and software.

1.3.1 Levels of automation

As we mentioned before, to optimize the final cost of the product and consider the technology limits, companies provide different automation levels for their robots. In the automotive sector, the Society of Automotive Engineers (SAE) introduced an informative standard, named SAE J3016 (figure 1.4), to describe the different

levels of automation in the vehicles. We find this standard so convenient for our application because our robot is a four-wheel mobile robot.

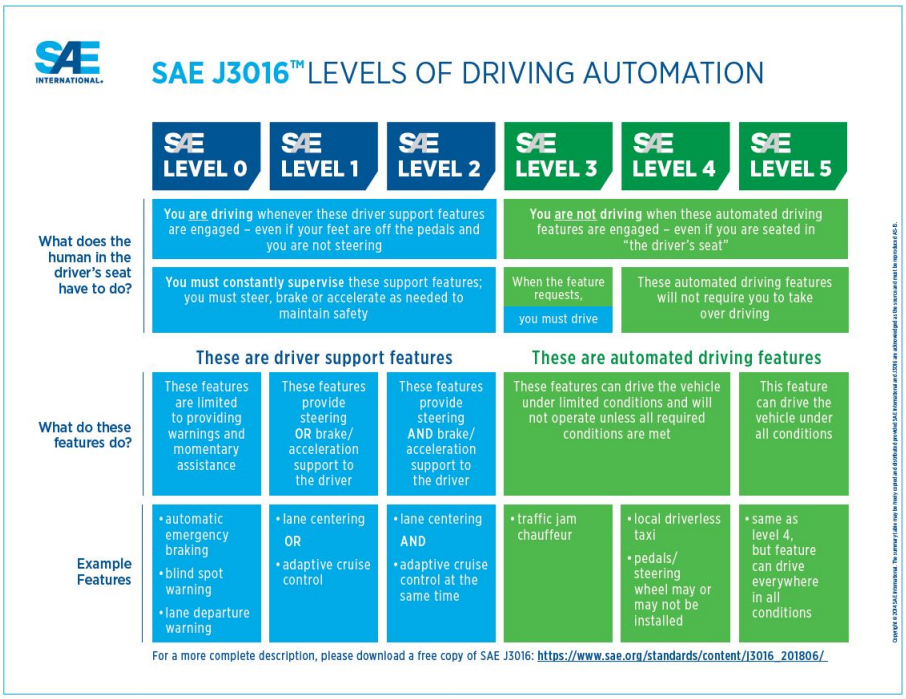


Figure 1.4: SAE J3016 Levels of Driving Automation [2].

As the standard describes at *SAE Level* 4 and *Level* 5, we can call our vehicle autonomous, as there is no need for the driver, or in our application operator, to drive, or operate, the vehicle. Our project aims to reach an autonomous level of 4 or 5 for our robot, so the related studies should inspect the hardware and software needed for these technology levels.

In the following section, we discuss the required hardware and software for obtaining autonomy for a mobile robot.

1.3.2 Hardware

As one can see in figure 1.3, we need some hardware, named sensors, to perform *Sensing*, which is gathering raw information from the environment. For an autonomous vehicle, we can use cameras, radars, Light Detection and Ranging (LIDAR), and Global Positioning System (GPS) to sense the location, obstacles, and any further needed information, like the position of grapes.

Another essential piece of hardware in autonomous vehicles is computing hardware. The computing hardware is like the robot's brain, and the robot can perform more sophisticated tasks with better hardware. This hardware takes the raw information from the sensor data and provides output commands to the vehicle. For this purpose, we can employ Graphics Processing Unit (GPUs), Field Programmable Gate Array (FPGAs), and Application Specific Integrated Chip (ASICs).

At the time of writing this thesis, we did not decide about the hardware we might need to use because we preferred to complete the software part of the project and then choose the hardware considering the requirements of our software.

1.3.3 Software

The software requirements can be obtained trivially by looking at figure 1.3. The first software module is named *Environment Perception Module*. The environment perception module has two main duties, first, identifying the current location of the robot in the environment, and second, classifying and locating important features of the environment for the driving task. For instance, the grape bushes or other farmers or robots are the environment's important features. *Sensing* and *Information Extraction and Interpretation* are included in this module.

The *Localization Map Building Module*, as it is clear from its name, is responsible for creating maps for locating objects around the vehicle and locating the vehicle inside the *global* map. The global map is the main map that we are doing our tasks inside, and for our application, the global map is the grape farm located in Grugliasco. For some projects, the global map is not available, and the robot itself is responsible for building such a map based on environment exploration.

Motion Control Module is responsible for taking the mission points, in our study starting and endpoint, and creates the optimum path and the required control inputs, like, velocities, and steering angles, to perform such a path. This module in our study consists of *Global Path Planner*, *Local Path Planner* and *Motion Controller*.

The software modules can be different from what we discussed, for example, in autonomous cars, we have another module, usually named as *System Supervisor*, which is responsible for checking the safety of driving decisions made by the motion control module. At the time of writing this thesis, we decided to keep the first three software modules. However, any modification of this structure is possible as the project goes forward.

The next chapter is about planning problems. There we will discuss what the challenges related to planning are and how we can overcome these problems.

Chapter 2

Planning

2.1 Introduction

We discussed the hierarchy of autonomy in the previous chapter. As figure 1.2 demonstrates, we need to define a mission for our robot, and then we let the robot perform the task autonomously. In our study, the mission is moving from a starting point and reach a goal.

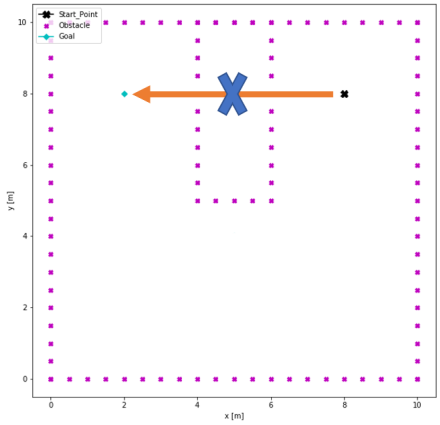


Figure 2.1: The Desired Path.

As we showed in figure 2.1, the desired path between two points is a straight line. One can quickly obtain from geometry that the shortest path between two points is a straight line that connects them. So the orange line is the shortest possible path between these two points. Another reason that we desire such a path is that the curvature of that path is the least possible curvature, which is zero. Such a characteristic lets us drive the robot as fast as possible. The combination of the mentioned reasons will lead our robot in the minimum time to its goal.

To make the previous explanation clearer, we focus on the definition of curvature and its relationship with the maximum possible velocity of the vehicle.

The curvature at a point of a differentiable curve is the inverse of the radius of the circle that is the best approximate of the curve near that point. The center of the mentioned circle is the center of curvature, or in our case, the instantaneous center of rotation. Finally, by convention, the curvature of a straight line is zero, or the radius of curvature of a straight line is infinity. Figure 2.1 provides a visual definition of curvature.

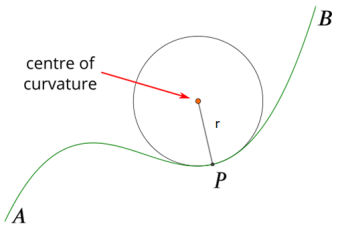


Figure 2.2: The curvature of a curve [3]

The other important concept here is the *Centripetal Force*, a lateral force that acts on the robot and keeps the robot in the path while he is turning. Chris Carter in his book [14] describes the magnitude of the centripetal force on an object of mass m moving at tangential speed v along a path with a radius of curvature r is:

$$F_c = ma_c = \frac{mv^2}{r} \tag{2.1}$$

In mobile robots, wheels are responsible for providing such a force, as figure 2.3 shows. Due to limited friction force, wheels can provide limited centripetal force at each instance. So at each instance, with any radius of curvature, the velocity of the robot is limited to keep the robot at the path. Otherwise, the wheels can not keep the robot on the path, and the robot will lose its trajectory. However, since the radius of curvature for a line is infinity, there is no velocity limit, concerning the lateral dynamics of the robot (equation 2.1), and we can drive the robot with its maximum possible velocity.

By having the shortest possible path and maximum possible velocity, we can reach the goal point at the shortest possible time, which is our objective.

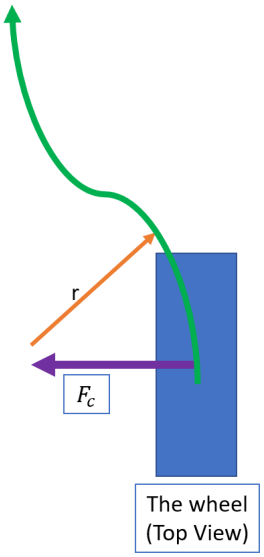


Figure 2.3: The top view of the wheel.

Unfortunately, the task of path planning for a robot is not an easy job. Difficulties usually come from the obstacles present on the road. For instance, figure 2.1 demonstrates an obstacle on our straight path. So the robot cannot reach its goal

directly. For this reason, we put an X on the path in the mentioned figure, and we have to generate another path for our robot. The generated path should have two principal characteristics. First, it should be feasible for the robot, which means the robot should follow that path considering his kinematic constraints. Second, the path should be *Optimal*. We discussed the first problem in the third chapter, The bicycle model, and the second problem in the next section of the current chapter, Optimization.

2.2 Optimization

As we discussed before, the generated path should be optimal. Nevertheless, for generating something optimal, we should define the optimum at first, and then we should try to rich the optimum through optimization. In our problem, which is to reach the target point from the starting point in the minimum possible amount of time, the optimum path is the straight line. Consequently, the generated path should be the closest one to the straight line. Since a straight line has the minimum possible length between two points and it has the minimum possible curvature that is zero, we can pass it with maximum velocity in the shortest possible time.

However, there is a problem with our optimization. Path length and curvature work opposite of each other some times. For example, in figure 2.1, we can have a long path with low curvature or a short path with significant curvature. In both cases, we might not be in the optimum situation because we have a long way to pass on the first one, and it can take an incredible amount of time, even though we can travel fast. On the second one, instead, we have a short way to pass, but due to the high curvature, we can not move fast. So, to do optimization, we shall define a cost function, and we need to have both parameters in our cost function to get sure we can get to the optimum.

2.2.1 Parametric Optimization

One convenient way to perform an optimization might be to parameterize the path and then perform the optimization. Alonzo Kelly et al. [15] performed such a parametrization through *Cubic Spirals* technique.

At first, they defined each point of the path, considering the states of the robot as:

$$X = (x, y, \theta, \kappa) \tag{2.2}$$

x and y are the coordinates of that point concerning the reference frame (RF). θ is the heading of the robot, and κ is the curvature of the path at the mentioned point. We represent the states at figure 2.4.

Then they parametrized the curve with the path length as follows:

$$x_{(s)} = \int_0^s cos\theta_{(s)}\, ds \tag{2.3}$$

$$y_{(s)} = \int_0^s sin\theta_{(s)}\, ds \tag{2.4}$$

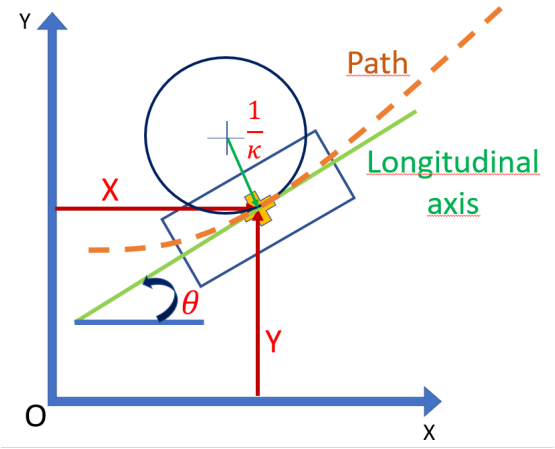


Figure 2.4: Robot states representation.

$$\theta_{(s)} = \int_0^s \kappa_{(s)}\, ds \tag{2.5}$$

$$\kappa_{(s)} = a + b.s + c.s^2 + d.s^3 \tag{2.6}$$

In the last equation, equation 2.6, they used a technique to overcome the mentioned problem of our optimization. Our curvature has parameters as a, b, c, and d which can be optimized through an optimization approach along with the path length s. In this way, we are optimizing both curvature and path length at the same time. We can optimize a previously generated path, usually through a random path planning generation approach, and after parametrizing the curve, we can start the optimization.

For such optimization, we define a cost function to quantify the quality of the path. By decreasing the cost, we can increase quality.

Cost function

For different applications, we can have different cost functions, for example, we might want our path to be near a global path that we have generated before, so we introduce a cost function describing the variance of our path and the global one, then by decreasing the cost, we get a path that is nearer to our desired path. For our purpose, which is reaching the goal in the minimum possible time, we need to recognize the path length and the curvature. Alonzo Kelly et al. [15] used a cost function that is defined as follows:

$$J = \int_0^{S_f} (\kappa_s)^2\, ds \tag{2.7}$$

S_f is the final path length, and s is the path length at each point, which means the length of the path from starting point to the mentioning point, which is the integration parameter. Curvature κ can be negative or positive, depending on defining the reference frame and the direction of curvature in the RF. When we talk about minimizing the curvature, we are willing to minimize the absolute value, as working with absolute value function can generate different numerical problems in coding, it

is preferred to use the squared of curvature to avoid its sign. For our optimization, we need to minimize J. Since integral is a strictly increasing function, decreasing its total value is possible by decreasing each of the parameters' values. So the optimization will decrease the value of the final length and the curvature at the same time.

Constraints

After defining the cost function, we should consider our boundary conditions, which are initial and final poses, $(x_0, y_0, \theta_0, \kappa_0)$ and $(x_f, y_f, \theta_f, \kappa_f)$. We also shall consider obstacles as forbidden zones, since optimization solver might generate a path that crosses them. Sometimes, the solver might give us a path with a short length, but high curvature that our robot might not be able to turn with such a curvature. To solve this problem, we should force our solver to bound our absolute value of the curvature below the maximum affordable curvature. Finally, our optimization will be like this:

$$min\ J(S_f, \kappa_s)\ s.t. \begin{cases} |\kappa_{(S)}| < \kappa_{Max} \\ r_{(S)} \neq \text{Obstacles Coordinates}, \forall S \in [0, S_f] \\ x_{S0} = x_0, \quad x_{S(S_f)} = x_f \\ y_{S0} = y_0, \quad y_{S(S_f)} = y_f \\ \theta_{S0} = \theta_0, \quad \theta_{S(S_f)} = \theta_f \\ \kappa_{S0} = \kappa_0, \quad \kappa_{S(S_f)} = \kappa_f \end{cases} \tag{2.8}$$

NP-Hard problem

In literature, such optimizations have a specific name, Constrained shortest path tour problem with a time window (CSPTP), and as stated in [16], such optimization is an NP-Hard problem, and we should solve it by heuristic methods. It means if we want to plan a path for a long way, we need to break our problem to several local paths and perform the optimization for each of them. Such an activity might not result in the optimum global path; however, this is the only way to perform these optimizations with our current computational hardware.

We summarize The approach that Alonzo Kelly et al. [15] introduced as the following. At first, we need to generate a path with a path planning method without considering optimality. For this purpose, they used a clothoid generated path. We optimize the generated path through the mentioned approach. Finally, through a receding horizon technique, which they explained in their paper in detail, we can generate the control input for our robot.

Alonzo Kelly et al. [15] work provided a comprehensive point of view for us and was an essential part of our study. However, We used a different technique, as theirs can generate a path that has a collision with the obstacle. B. Siciliano et al. [6] explain in their book that it is impossible to write the obstacle avoidance term in the constraints described in 2.8. Consequently, we study all the paths to be collision-free by checking the euclidean distance of equally spaced points on the path with obstacle points. Then we optimize the paths that all of their points are in a safe distance from the obstacles. Second, we preferred to detach control input generator (path execution) and path generating problems to be able to implement each module

on a different processor in the future, and also one module that performs two tasks can be computationally heavy.

In the following section, we discuss different path planning techniques available in the literature and the techniques that we chose for our study.

2.3 Path planning technique

2.3.1 State of the art

Lots of researchers studied Path planning in mobile robotics in the previous decades. David González et al. [17] did an extensive review of different researches, and divided the path planning techniques into four subgroups: graph search, sampling, interpolating, and numerical optimization.

1. Graph search based planners

In automated driving, the basic idea is to get from point A to point B by traveling along with the state space. An occupancy grid represents this state space and demonstrates where each object is. From the planning point of view, graph searching algorithms implement a path that visits the different states in the grid, giving a solution to the path planning problem. There are several algorithms to perform such an action.

Dijkstra algorithm : It is a graph searching algorithm that finds a single source shortest path in the graph. A discrete cell-grid space approximates the configuration space, and then a search method is implemented to find the shortest path considering these cells.

Implementing some heuristics on Dijkstra can improve its performance, which is the base of the other algorithm, A*.

$A-Star\ algorithm\ (A^*)$: It is a graph searching algorithm that enables a fast node search due to the implementation of the heuristic method and a cost function on the Dijkstra algorithm. This method is suitable for searching spaces which are known a priori. Implementing the robot's dynamic model in such a method can increase its performance, which is known as the D* technique.

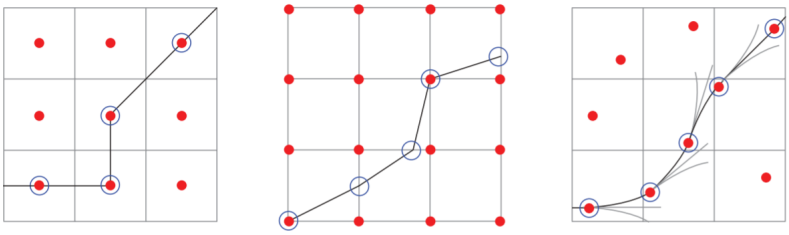


Figure 2.5: A comparison between A* (left) and D* (center and right) [4]. D* can generate a smoother path.

2. Sampling based planners

These methods try to meet strict timing requirements, and consist of randomly sampling the configuration space or the state space and look for connectivity inside

it. Here we described the technique from [17], which we used for our global path planner.

Rapid random exploring tree (*RRT*) : It belongs to the sampling-based algorithms applicable to on-line path planning. It allows fast planning in semi-structured spaces by executing a random search through the navigation area. This method can consider the robot's non-holonomic constraints, for example, by employing the bicycle model of the robot and generating new searching points according to the model. We explained non-holonomic constraints and bicycle model in detail in Chapter 3. Applying some heuristic methods, such as searching the nearest points to the goal at first, can introduce RRT*.

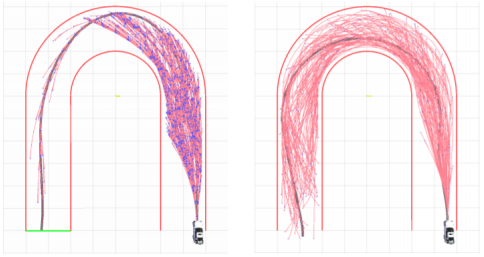


Figure 2.6: A comparison between RRT* (left) and RRT (right) [5]. RRT generates more branches and is computationally heavier, while RRT* is faster and generates less branches.

3. Interpolating curve planners

Techniques as Computer-Aided Geometric Design (CAGD) implement path smoothing solutions for a given set of way-points, usually generated with sampling-based planners. These techniques allow the path planner to increase the comfort of the generated path or reduce the needed control effort. They are suitable for offline path planning, but they are not usually suitable for online path planning in real-time due to their computational load. Clothoid Curves, Polynomial Curves, Bézier curves, and spline curves are their examples. One can find a suitable description for each of them in [17]. We did not study them in detail due to their fundamental problem of implementing real-time path planning.

4. Numerical Optimization

These methods are useful for optimizing the previously generated path through other planning techniques, and Alonzo Kelly et al. [15] are using these techniques in their work.

Here we aim to maximize or minimize a cost function subject to different constrained variables. there are two methods among them that we found interesting.

One is named *Potential Field Method* (PFM). In 1997 Dieter Fox et al. [18] studied this method. In PFM, we assume that obstacles assert negative forces on the robot, and the target location affirms a positive one. PFM is extremely fast; however, it can study only a subset of obstacles near the robot. Consequently, with the PFM technique, we can solve the problem of obstacle avoidance presented in Alonzo Kelly et al. [15] paper. However, D.Fox et al. [18] continue that PFM

is not able to find a path in narrow corridors, and also, there is a chance that this approach generates a chattering behavior for the robot. Consequently, they introduced another method, which is named *Dynamic Window Approach*.

2.3.2 Dynamic Window Approach

This approach tries to implement the way-point generation and numerical optimization simultaneously. D.Fox et al. [18] explain that their approach generates several paths considering the kinematic model of the robot, so all of the paths are $affordable$ for the robot, and then by implementing a cost function they choose between all of the generated paths. Applying *Admissible Velocity* and *Steering Angles* to the kinematic model, we generate paths, so the kinematic and dynamic constraints of the robot are at the same time met.

The kinematic model of the robot is a mathematical description of the kinematic constraint of the robot. For example, we all know that a car can not move normal to its longitudinal axis; this is a non-holonomic kinematic constraint. It is non-holonomic since it does not change during the time, and it is a kinematic constraint since it puts limits on velocities of the car. We explained the kinematic model and non-holonomic constraints in Chapter 3 and the Dynamic-Window approach in detail in Chapter 5.

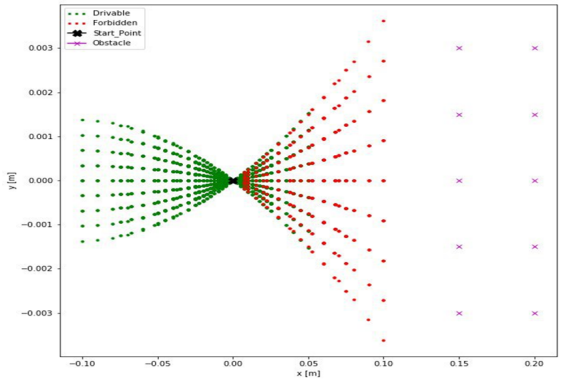


Figure 2.7: Dynamic Window Approach. Several paths are generated and then through a cost function the collision is checked and the best path is chosen.

The problem of local minimum

We chose DWA for our path planner. However, this method has a problem that one needs to consider. This method has an inherent problem of getting stocked in a local minimum. As figure 2.8 shows, to reach the goal, the robot needs to get far from it at first, and then after turning to the right, it can get near to the goal again. There is a term for distance from the goal in the cost function (equation 5.9 in chapter 5) we are using for DWA. Getting far from the goal will increase that term and consequently, the cost function, so the DWA approach prefers to stay at the same place rather than moving. To solve this problem, we can define some local goals. they can be defined manually as figure 2.9, or we can provide them by a *Global Path Planner*.

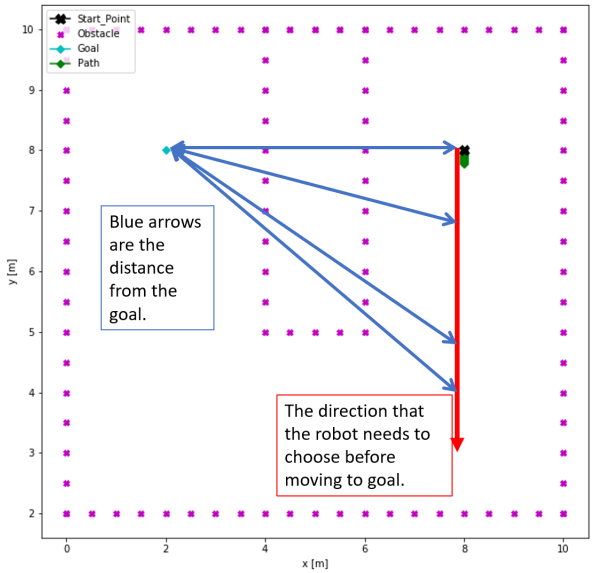


Figure 2.8: Dynamic Window Approach. This method can get stocked in the local minimum.

2.4 Final Path Planning Hierarchy

Figure 2.10 represents the final *Path Planning Hierarchy* that we chose for our study. In this study, we define map as some objects, Starting point, Goal Point, and Obstacles, in Python code and mission is reaching from the starting point to the goal point. They are passed to our global path planner, which is RRT* path planner, and to our local path planner, which is a DWA path planner. In a further development, we can use the modules we developed as software packages of ROS and implement them on the robot.

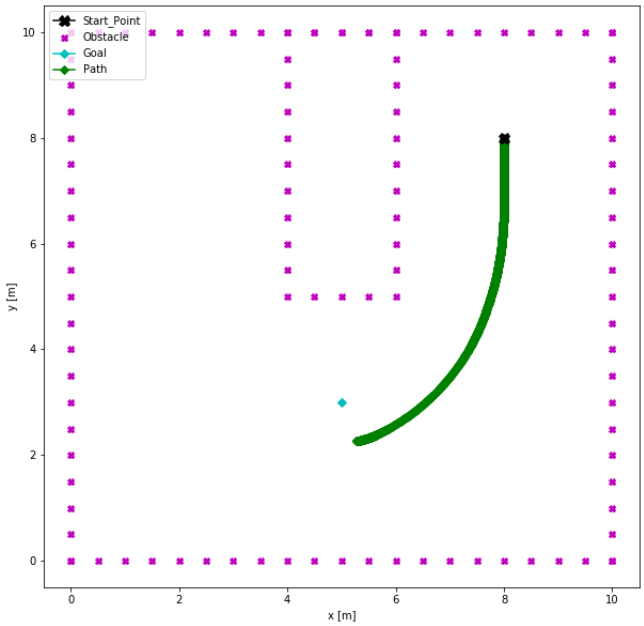


Figure 2.9: Inserting local goal manually.

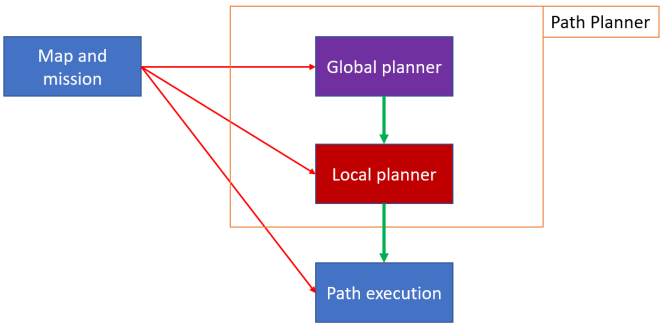


Figure 2.10: Our Path Planning Hierarchy.

Chapter 3

Bicycle model

The previous chapter tried to explain the problem of path planning and different techniques to solve it. We mentioned the kinematic model and the fact that all the techniques, somehow, need to consider this model to generate the path. Here, we want to introduce the kinematic model and explain why path planning techniques need to respect this model.

3.1 Kinematic constraints

As [6] explain, wheels are the most common form of locomotion system in mobile robots. Although any vehicle that moves with wheels is subjected to some kinematic constraints, it is still possible for that vehicle to admit any arbitrary pose in the environment. For example, any driver knows that it is impossible to drive a car perpendicular to its heading, but still, it is possible to park the car in any given direction by performing a suitable maneuver. However, this characteristic is not common between all kinds of kinematic constraints. In general, any mechanical connection can introduce some kinematic constraints. Sometimes, these constraints allow us to reach every arbitrary $configuration$ of our system by performing suitable maneuvers. However, there are some situations that these kinematic constraints *block* some configurations completely, and with no maneuver, we can reach the blocked configuration. The kinematic constraints applied on the mobile robot are from the first example, which we call them non-holonomic (not integrable), and some of the kinematic constraints applied on *Anthropomorphic* robots are like the second example which is called holonomic constraints. One can find the mathematical details and proofs in detail in the mentioned book [6]. Both kinematic constraints can be represented in the so-called $Pfaffian\ form$:

$$A^T(q)\dot{q} = 0 \tag{3.1}$$

in which q is our configuration vector, and A is a matrix to describe the constraints. In the next sub-section, we clarify this mathematical form by discussing an example.

3.1.1 Non-holonomic constraints

As already anticipated, Non- holonomic constraints do not block any configuration, so they do not reduce the $configuration\ space$. Instead, they put limits on

generalized velocities, so they limit different velocities that the robot can accept. For clarification, we recall a classic example from [6].

Pure rolling constraint

Let's consider a disk (figure 3.1 [6]) that rolls without slipping on the horizontal plane, while keeping its *sagittal plane* (i.e. the plane that contains the disc). Three generalized coordinates can describe its configuration: the Cartesian coordinates (x, y) of the contact point with the ground, measured in a fixed reference frame, and the angle θ characterizing the orientation of the disk to the x axis. The configuration vector is therefore $q = [x, y, \theta]^T$.

The *pure rolling* constraint for the disk is expressed in the Pfaffian form as:

$$\dot{x}sin(\theta) - \dot{y}cos(\theta) = [sin(\theta) \quad -cos(\theta) \quad 0]\dot{q} = 0 \tag{3.2}$$

Here the matrix $[sin(\theta) \quad -cos(\theta) \quad 0]$ is our matrix A which we introduced in Pfaffian form of kinematic constraints and q is our configuration vector.This constraint shows that in the absence of slipping, the contact point's velocity has zero components in the orthogonal direction to the sagittal plane. So the generalized velocities are limited. However, the mentioned disk still can go from any arbitrary configuration to any other arbitrary configuration by taking the right maneuver.

To prove that let's consider that the disk is in initial configuration $q_i = [x_i, y_i, \theta_i]^T$ and wants to go to any arbitrary final configuration $q_f = [x_f, y_f, \theta_f]^T$. We can perform such a maneuver through the following sequence of movements satisfying the constraints

1. rotate the disk around its virtual axis so as to reach the orientation θ_v for which the *sagittal axis* (i.e. the intersection of the sagittal plane and the horizontal plane) goes through the final contact point (s_f, y_f);

2. roll the disk on the plane at a constant orientation θ_v until the contact point reaches its final position (x_f, y_f);

3. rotate again the disk around its virtual axis to change the orientation from θ_v to θ_f

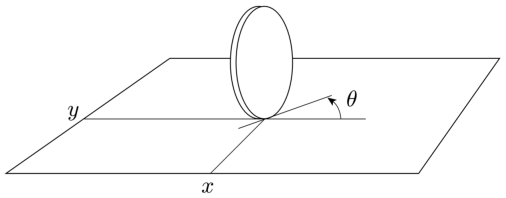


Figure 3.1: Generalized coordinates for a disk rolling on a plane [6].

3.2 Kinematic model

As we discussed, non-holonomic constraints will let us take any arbitrary configuration in the working space, but they limit us in the admissible velocities and, as a result, in the trajectories that our locomotion system can follow. The admissible trajectories of a system with k Pfaffian constraints that entails the permissible generalized velocity can be obtained by solving the following nonlinear dynamic system [6]:

$$\dot{\mathbf{q}} = \sum_{j=1}^{m} g_j(\mathbf{q}) u_j = \mathbf{G}(\mathbf{q})\mathbf{u} \tag{3.3}$$

in which $\mathbf{q}$ is our state vector (configuration vector), $\mathbf{u} = [u_1, u_2, ..., u_m]^T$ is the input vector, G is the *input vector fields matrix* and this system is called $driftless$, because, we can get $\dot{q} = 0$ only if we have a zero input. We should also consider that the choice of G is not unique, consequently, we can have different types of inputs u, however Bruno Siciliano et.al. [6] discussed in detail that $\mathbf{u}$ is not directly related to the actual control inputs, which are forces and torques, so we can call such a system as *kinematic model*.

Whenever we have such a system, we can check if these constraints are holonomic or non-holonomic and then decide about our system. Bruno Siciliano et al. [6] introduce a methodology to study this fact, and since we already know in path planning of wheel-based mobile robots, we are facing non-holonomic constraints, we skip this methodology. The only point that is so important for us is that we can find the admissible trajectories by solving the mentioned kinematic model (equation 3.3). As trajectories are paths with timing law, if a path planner wants to generate an *admissible* path, which is a path that the robot can follow, it should create the path by solving the kinematic model of the robot. Usually, there is no analytical method to solve such models, and numerical methods are used to solve such equations.

In the following, we discuss two kinematic models that are relevant to our project. First, we discuss a *bicycle* with *Akcermann steering system*, and then we discuss a bicycle model with a four-wheel steering system (later called 4WS vehicles).

3.3 Bicycle model

Under the underlying assumptions of planar motion, rigid body and non-slippage of the tire, Singletrack model, or Bicycle model, can be used as a satisfactory approximation for describing a four-wheel mobile robot. Considering which type of steering system we use, we can have a different Bicycle model. Here we discuss the two models that we have used in our path planners, i.e., Ackermann steering and 4WS.

3.3.1 Ackermann steering system

Consider a bicycle that has an orientable wheel, and a fixed wheel as shown in figure 3.2.

A possible choice for generalized coordinates (State vector) can be $q = [x, y, \theta, \phi]^T$, where (x, y) are the Cartesian coordinates of the contact point between the rear

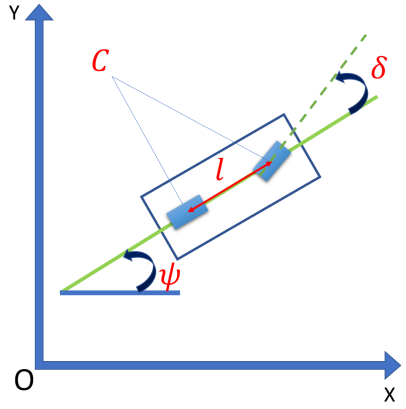


Figure 3.2: Bicycle model with Ackermann steering system.

wheel and the ground (i.e., of the rear wheel center), θ is the orientation of the vehicle considering the x-axis, and ϕ is the steering angle of the front wheel with respect to the vehicle longitudinal axis.

The motion of the vehicle is subject to two pure rolling constraints, one for each wheel:

$$\dot{x}_f sin(\psi + \delta) - \dot{y}_f cos(\psi + \delta) = 0 \tag{3.4}$$

$$\dot{x} sin(\psi) - \dot{y} cos(\psi) = 0 \tag{3.5}$$

where (x_f, y_f) is the Cartesian position of the center of the front wheel, the geometric meaning of these constraints is obvious: the velocity of the center of the front wheel is zero in the direction orthogonal to the wheel itself, while the velocity of the center of the rear wheel is zero in the direction orthogonal to the sagittal axis of the vehicle. Using the rigid body constraint we obtain:

$$x_f = x + l cos(\psi) \tag{3.6}$$

$$y_f = y + l sin(\psi) \tag{3.7}$$

where l is the distance between the wheels. So equation 3.4 can be rewritten as

$$\dot{x} sin(\psi + \delta) - \dot{y} cos(\psi + \delta) - l\dot{\psi} cos(\delta) = 0 \tag{3.8}$$

We can write the matrix A associate with the Pfaffian constraints. For the equations, you need to recall them using, e.g., "Eq. (3.8) and Eq. (3.5)":

$$A^T(q) = \begin{bmatrix} sin(\psi) & -cos(\psi) & 0 & 0 \\ sin(\psi + \delta) & -cos(\psi + \delta) & -lcos(\delta) & 0 \end{bmatrix}$$

Finally, by some algebraic manipulation, we can get the bicycle model for Ackermann steering vehicle as:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\psi} \\ \dot{\delta} \end{bmatrix} = \begin{bmatrix} cos(\psi) \\ sin(\psi) \\ tan(\delta)/l \\ 0 \end{bmatrix} . v + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} . \omega \tag{3.9}$$

3.3.2 Four wheel steering system

The previous kinematic model, for an Ackermann steering vehicle, is not sufficient for our robot. Since our robot has a four-wheel steering system, we have to develop its relative kinematic model. D. Wang et al. [7] have developed such a model for a 4WS robot.

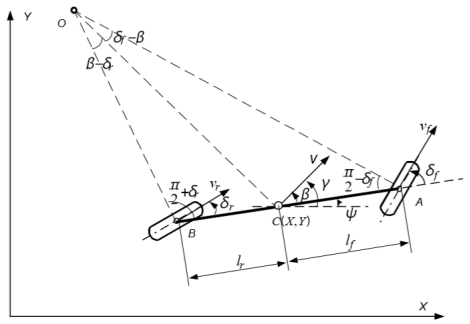


Figure 3.3: Bicycle model with four wheel steering system (4WS) [7].

The motion status of the vehicle at an arbitrary moment can be described as shown in figure 3.3, where *reference point* C is taken at the center of gravity of the vehicle body. Coordinates (x, y) represent the position of the vehicle. *vehicle velocity* ν is defined at the reference point C; *heading angle* ψ is the angle from the X-axis to the longitudinal axis of the vehicle body AB; *course angle* γ is the angle from the X-axis to the direction of the vehicle velocity, ν; $side - slip\ angle\ \beta$ is the angle from the longitudinal axis of the vehicle body AB to the direction of vehicle velocity ν; *turning radius* r is the distance between the reference point C and the *Instant Rotating Center (IRC)* O; *front wheel velocity* ν_f is the velocity defined at the intersection of the mid-plane of the virtual front wheel and the front wheel axle, A; *rear wheel Velocity* ν_r is the velocity defined at the intersection of the mid-plane of the virtual rear wheel and the rear wheel axle, B; *front wheel steering angle* δ_f is the angle from the longitudinal axis of the vehicle body AB to the direction of ν_f; *rear wheel steering angle* δ_r is the angle from the longitudinal axis of the vehicle body AB to the direction of ν_r.

Referring to the figure and the explanations we can obtain:

$$\dot{X} = Vcos(\psi + \beta) \tag{3.10}$$

$$\dot{Y} = Vsin(\psi + \beta) \tag{3.11}$$

$$\dot{\psi} = \frac{Vcos(\beta)(tan(\delta_f) + tan(\delta_r))}{l_f + l_r} \tag{3.12}$$

$$\beta = arctan(\frac{l_f tan(\delta_r) + l_r tan(\delta_f)}{l_f + l_r}) \tag{3.13}$$

$$V = \frac{V_f cos(\delta_f) + V_r cos(\delta_r)}{l_f + l_r} \tag{3.14}$$

In this model, there are two steering angels, δ_f and δ_r and two wheel-velocities, V_f and V_r. The state variables of kinematic motion describe the robot configuration (X, Y, ψ).

3.4 Numerical integration

For solving the kinematic model, we need to integrate on our differential equations. However, since they are not decoupled, there is usually no analytical method to perform such integration, and we have to perform numerical integration. To perform such a numerical integration, we used a *forward euler* technique.

In this method, we consider obtaining the shape of an unknown curve, which starts at time 0 and satisfies a given differential equation. We can assume the differential equation as a formula in which the slope of the tangent line to the curve can be computed at any point on the curve. Consider a differential equation, $\dot{y} = f(t, y(t))$, if we know the initial value of y which is y_0 we can calculate the slope of the curve at y_0 as $f(t_0, y_0)$ and we can obtain the next step value y_1 by solving the following equation:

$$[\dot{y} = f(x)]_{ZOH} : y_i = y_{i-1} + f(t_{i-1}, y_{i-1}).dt \ \ \forall i = 1, n \tag{3.15}$$

where i is the index of the step that we want to calculate y_i. $y_(i-1)$ is the value of y at the previous step (the initial value), $f(t_{i-1}, y_{i-1})$ is the slope of the curve at the initial point and dt is the length of the step that we want to take, $dt = t_i - t_i - 1$. We can divide the *time-window* between $[0, t]$ n steps, and if we choose the steps small enough according to this method we can regenerate the curve with small error.

Now we can apply the method to our kinematic model, and we get:

$$X_n = X_{n-1} + V_{n-1}cos(\psi_{n-1} + \beta_{n-1}).dt \tag{3.16}$$

$$Y_n = Y_{n-1} + V_{n-1}sin(\psi_{n-1} + \beta_{n-1}).dt \tag{3.17}$$

$$\psi_n = \psi_{n-1} + \frac{V_{n-1}cos(\beta_{n-1})(tan(\delta_{fn-1}) + tan(\delta_{rn-1}))}{l_f + l_r}.dt \tag{3.18}$$

$$\beta_{n-1} = arctan(\frac{l_f tan(\delta_{rn-1}) + l_r tan(\delta_{fn-1})}{l_f + l_r}) \tag{3.19}$$

$$V_{n-1} = \frac{V_{fn-1}cos(\delta_{fn-1}) + V_{rn-1}cos(\delta_{rn-1})}{l_f + l_r} \tag{3.20}$$

Till now, we have obtained the *discretized* model of our robot. However, since implementing this model is computationally expensive for our global path planner, we prefer to discretize the bicycle model with the Ackermann steering system and apply it to our global path planner. The details of the global path planner and the local path planner and their communication are described in the next chapters. The discretized bicycle model with Ackermann steering system is as follows:

$$x_n = x_{n-1} + \nu_{n-1}cos(\psi_{n-1}).dt \tag{3.21}$$

$$y_n = y_{n-1} + \nu_{n-1}sin(\psi_{n-1}).dt \tag{3.22}$$

$$\psi_n = \psi_{n-1} + \nu_{n-1}tan(\delta_{n-1})/l.dt \tag{3.23}$$

In the two following chapters, we discuss the implementation of global and local path planners and how these planners communicate with each other.

Chapter 4

Global path planner

4.1 Introduction

As we discussed in chapters 1 and 2, the whole path planning procedure is an optimization problem, and it is an NP-hard optimization, so we have to solve it heuristically. We have decided to use DWA to allows us to produce an *admissible* path for the robot. The path generated by DWA is also smoother than other methods as DWA gives different velocities and steering angles to the kinematic model and then choose the best path at each step.

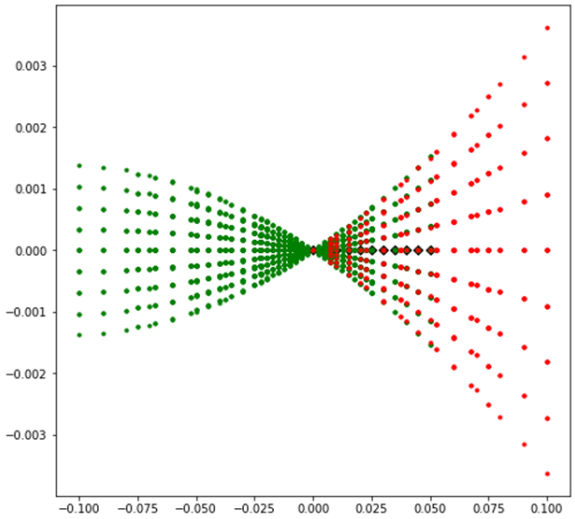


Figure 4.1: DWA is a branch and bound method.

As figure 4.1 demonstrates, DWA generates different branches and then chooses the best one considering its cost function, nodes illustrated as black diamonds belonging to the chosen branch. The branches that are not chosen are deleted permanently due to memory limitations, so if the chosen branches get stocked in a local minimum, the robot is stocked as it does not know any other path to follow. In figure 4.2 the planner wanted to choose the shortest and the most smooth path to the goal, but there is an obstacle in front of it, so the planner is stocked in a local minimum, as by choosing any other path the length and the curvature of the path will become larger than passing through the red line.

For this reason, we need another planner that is computationally affordable to

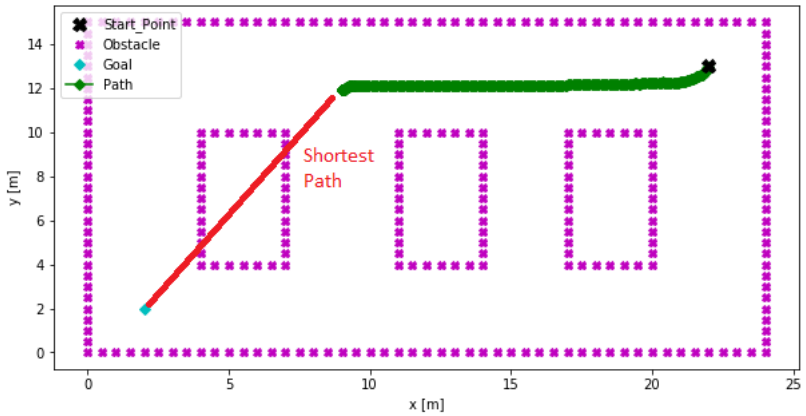


Figure 4.2: If the planner gets stocked in a local minimum the robot is stocked as it does not know any other path to follow.

have a global look at the map and guide the robot through the global map, then the result of the global planner is given to the local path planner to generate a smooth path related to the local information.

In chapter two, we did a literature review on different path planning techniques. Among all the mentioned techniques in the literature, A*, D*, RRT, and RRT* are less computationally heavy. We have chosen RRT* as we can easily use the robot's kinematic model inside it and generate a feasible *global* path for our robot.

4.2 RRT

J. H. Jeon et al. [5] used an RRT path planner to introduce an *Optimal motion planning*. The planner uses the Akermann steering angle bicycle model, although we discussed the model in chapter three, we write all of its discretized equation here to follow this part quickly. We also need to add that we decided to use the Akermann steering angle bicycle model for the global planner as it is computationally lighter than the 4WS model.

$$x_n = x_{n-1} + \nu_{n-1} cos(\psi_{n-1}).dt \tag{4.1}$$

$$y_n = y_{n-1} + \nu_{n-1} sin(\psi_{n-1}).dt \tag{4.2}$$

$$\psi_n = \psi_{n-1} + \nu_{n-1} tan(\delta_{n-1})/l.dt \tag{4.3}$$

Our planner, which has been inspired by [5], plans for each second of the robot's movement, so $dt = 1$. It is based on the multi-step procedure:

1. It generates a search tree for discovering the map and finding a path based on the robot's kinematic model.

2. It considers a constant speed for the robot as 1 m/s, so $\nu_{n-1} = 1$ for all the steps;

3. It gives three choices of steering angles, δ, to generate three branches for the search tree. In our case they were $\pi/6$, 0 and $-\pi/6$ radian;

4. By giving these data to equations 3.21, 3.22, and 3.23 for every initial pose $(x_{n.1}, y_{n-q}, \psi_{n-1})$, the generator will generate three new poses;

5. The new points are checked to be collision-free, and we calculate each new point's distance from the obstacle points;

6. If the new points were collision-free, they are added to a list of initial poses.

7. At each step, the planner looks at the mentioned list and randomly chooses an initial point to generate new points, whereas the points used for generating trees as initial point are removed from the list;

8. These steps are repeated until the planner reaches the goal or finds an empty list of initial points that means the planner can not find a feasible path. Then the planner connects the point that touched the goal to its initial pose and continues this connecting activity to reach the robot's initial position. The connected points yield the guiding way-points for the local path planner.

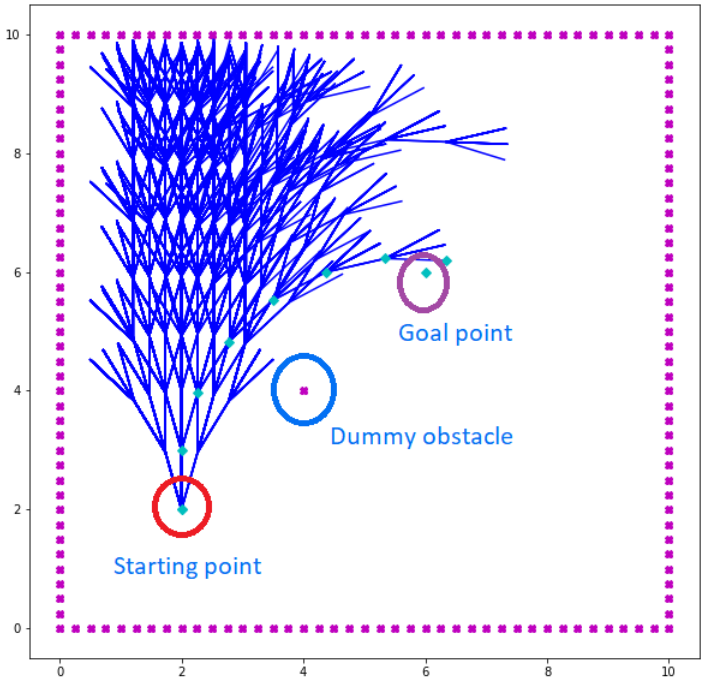


Figure 4.3: First attempt of RRT.

Figure 4.3 shows our first attempt for implementing RRT. The planner starts from the starting point as the initial point and starts to generate the tree shape paths, which are all the planner's attempts to reach the goal. As in this planner, we reserve all the nodes unless they have collision, we have a full global view on the map, and we can guide the robot globally through the map. In this attempt, we had the map border and a dummy obstacle as our obstacles to test the collision checking mechanism of the global path planner. After one node of the planner touched the goal, the planner started to connect the nodes to make the path that he found.

This planner is fast and provides a good guide for the local path planner; however, the process can still be so time-consuming if we have a big map or a map with curvy

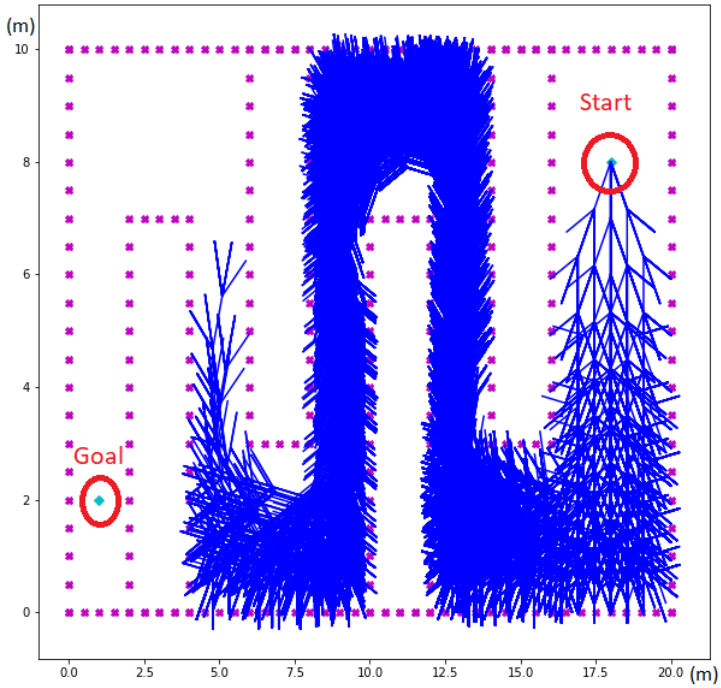


Figure 4.4: RRT can take so time consuming if the map has curvy narrow corridors.

and narrow corridors. Figure 4.4 shows one of our attempts. In this attempt, the global path planner was running for half a day, and still, it continued running. We decided to add another heuristic measure to our RRT to make it suitable for such missions. As we mentioned in chapter two, we introduce the RRT* algorithm by adding a heuristic logic to the RRT.

4.3 RRT*

To make the path planner more computationally efficient, we decided to add a heuristic rule. The rule is simple; the planner should always choose from the new points that are nearer to the goal. To implement this at first, we calculate every new point's distance to the goal point as:

$$d_i = \sqrt{(x_i - x_{goal})^2 + (y_i - y_{goal})^2} \tag{4.4}$$

then we sort, in increasing order, all the calculated distances in a set called distance set:

$$distance\_set = \{d_1, d_2, ..., d_n\} \tag{4.5}$$

then calculate the set's quartiles, and we put four different sets for recently generated points as:

$$from\ 0\_25\% = \{d_1, d_2, ..., d_i\} \tag{4.6}$$

$$from\ 25\_50\% = \{d_i, d_{i+1}, ..., d_j\} \tag{4.7}$$

$$from\ 50\_75\% = \{d_j, d_{j+1}, ..., d_k\} \tag{4.8}$$

$$from\ 75\_100\% = \{d_k, d_{k+1}, ..., d_n\} \tag{4.9}$$

The first quartile points are added to the first set, the points in the second quartile to the second set, and so on. The path planner will choose the initial points to generate a new tree, from the first set and evaluate the lately generated points. If the first list is empty, the planner will take a look at the second set, if it was blank planner checks the third set and finally the fourth set. With this technique, the points nearer to the goal have a higher chance to be chosen as the root of the new tree and planner acts faster. We need to mention that if all the sets were empty, all the newly generated points had a collision, and the planner declares there is no way to reach the goal.

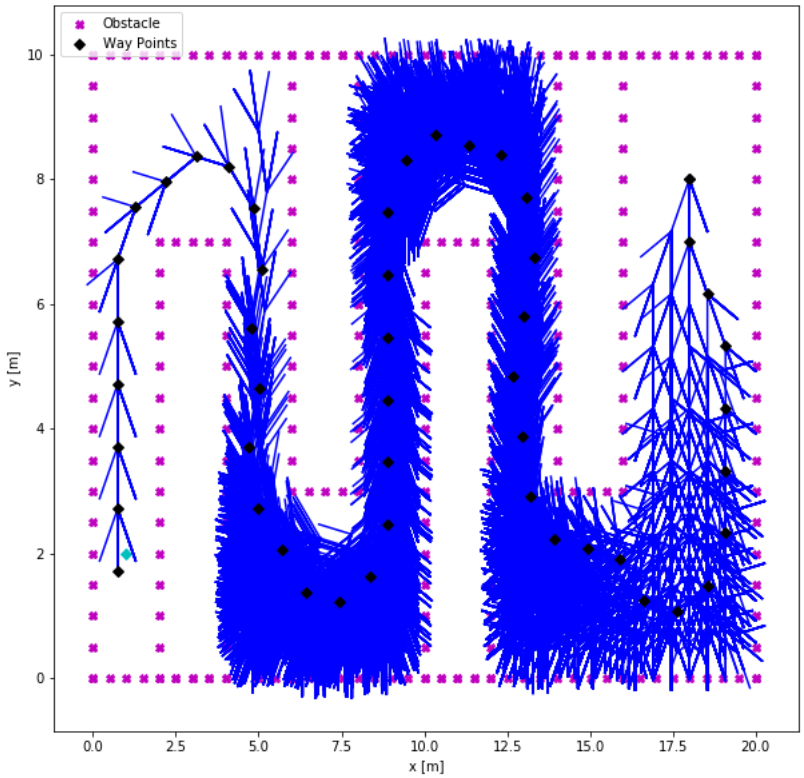


Figure 4.5: RRT* is much faster than RRT.

Figure 4.5 shows the result of our attempt by RRT*. The new path is generated in two minutes!

The generated way-points are given to the next level of planner that is our local planner to generate a smooth path for the robot. The local planner, which uses the DWA algorithm, is explained in the next chapter.

Chapter 5

Local path planner

5.1 Introduction

In the previous chapter, we discussed that the RRT* path planner generates a path. However, we can not use the path directly to control the robot. Since the generated path will is the reference signal for the robot's motion controller, smoothness of the path is vital to guarantee an excellent control performance, whereas, RRT* path planner generates a low-quality path, and it is not a useful reference signal for our controller.

Thus, we need to use another path planner called Local path planner to use the guide provided by the global path planner and generate a smooth path that we can provide to the motion controller as a reference signal.

To generate a smooth and affordable path, we have to strictly follow the robot's non-holonomic constraints, which are illustrated as the kinematic model of the robot. We used the Ackermann steering system bicycle model for our global path planner as we wanted to introduce a fast planner. However, for the local planner, since it will study smaller parts of the map each time and it can perform a more complicated calculation requiring the same computational time, we can use a computational more massive model, which is the 4WS bicycle model.

As the Dynamic-Window approach provides better performance than its rivals, which we discussed in chapter 2, we chose DWA as our local path planner, and we used the 4WS bicycle model to generate the path.

Dynamic-Window approach gives different inputs, V_f, V_r, δ_f, δ_r to equations 3.16, 3.17, 3.18, 3.19, and 3.20 at each time instances and generates some sub trajectories and then evaluate those sub-trajectories to choose the optimized path. This procedure is explained in detail in the next section.

5.2 Dynamic-Window Approach

To generate the $sub - trajectories$, we can give different velocities and steering angles to the model for a time instance and then record the *generated path*, related *velocities*, and *steering angles*. We can then check each sub-trajectory for collisions, rate collision-free sub-trajectories with our cost function, and record the *chosen*, path, velocity, and steering angles. Repeating these steps till the end of our mission(reaching the goal point), we will generate both paths, velocity, and steering angles profile, which we call trajectory.

The task described here consists of defining velocity and steering angle domains. One should notice that these domains are bounded, which means that we have limits on the maximum velocity that our robot can reach and the maximum steering angles at both directions that the robot can produce. The other important point is that when the robot has a specific velocity or steering angle, due to its finite accelerations, at each time step, it can reach a limited velocity or steering angles in the domain. Furthermore, last is that we should always drive safely, which means that even if a high-speed is achievable for us, we should check its *admissibility*. Admissible velocities and possible steering angles are described in detail in the next sub-section.

5.2.1 Velocities and steering angles

In this subsection, we discuss velocities that are reachable for us within each *time window* and the *admissible* (maximum) velocity that we can take at each instance. The first part belongs to $dynamic - window$, which discuses which velocities and steering angles are possible to achieve considering the accelerations, and the second part belongs to *admissible velocity*, which means which velocity is safe.

Dynamic-Window approach

One reality about every motor is that the motor can generate a limited torque, and since every mechanical component has its inertia, so with any motor, we can reach a limited acceleration.

Let us consider the classical equation of motion-defined as $V = V_0 + \dot{V}t$:

$$V = V_0 + \dot{V}.t \tag{5.1}$$

where V_0 is our initial velocity, and $\dot{V}$ is our acceleration, and t is the time we need to get from V_0 to V by the mentioned acceleration. This equation of motion is correct for both acceleration and deceleration. As a result, at each time window, we can access the velocities that belong to the $Dynamic - Window$ set or V_d is defined as:

$$V_d = \{v \mid v \in [V_0 - \dot{V}.t, V_0 + \dot{V}.t]\} \tag{5.2}$$

Fox et al. [18] also considered *angular velocity* in the Dynamic-Window because they wanted to write the algorithm for a synchro-drive robot which has the angular velocity term in its kinematic model, so for them, $V_d = (v, \omega)$. F. Zhang et al. [8] and M. Suchi et al. [19] did the same because they developed this approach for a differential-drive mobile robot. However, in our case, which is a 4WS mobile robot, we have two steering angle terms in our kinematic equation, and we should give the steering angles as a command to the robot. Consequently, we have to modify the Dynamic-Window approach to make it suitable for our application.

Almost the same idea holds for steering angle, but we should notice that the steering angle is an angular displacement, so the motion equation that we should consider is different. We discuss the steering angles Dynamic-Window in the next part.

Steering Angles

The same limitations, which we have for motors that give us the linear velocities, hold for Steering-Angle motors. So, we should also consider their limitation when

we are generating steering angles. However, as we mentioned in the previous part, the steering angle is an angular displacement, so the equation of motion that we consider is:

$$\delta = \delta_0 + \dot{\delta}_0.t + \frac{1}{2}\ddot{\delta}.t^2 \tag{5.3}$$

where $\dot{\delta}_0$ is the initial angular velocity of the motor responsible for steering angle. $\ddot{\delta}$ is the angular acceleration of the same motor, and δ_0 is the initial steering angle at the beginning of the time-window. $\dot{\delta}_0$ is always 0 at the start of the motor activation, so, we do not consider this term.

Finally, the Dynamic-Window for the steering angles will be:

$$\delta_d = \{\delta \mid \delta \in [\delta_0 - 0.5.\ddot{\delta}.t^2, \delta_0 + 0.5.\ddot{\delta}.t^2]\} \tag{5.4}$$

The last thing we should notice is that in our kinematic model, we have two velocities and two steering angles. Since the exploited motors for both front and rear wheels and traction and steering are the same, the accelerations are considered the same for both rear and front wheels. What is new in our approach, concerning others already mentioned, is that we have four search spaces for our optimization, which is forced by our kinematic model, and two of those variables are displacement so we should use a different equation of motion for defining their dynamic window as we did.

In the next part, we discuss the *admissible velocity*, and then we will have a summary of the modified Dynamic-Window approach that we are using.

Admissible Velocities

The first rule in driving a car might be to drive safely. The same rule applies with higher priority when we talk about autonomous vehicles as the 4WS Robot considered.

Drive safely means that we should always consider the necessary braking distance for our vehicle, considering its current velocity. However, the distance is forced by the environment, and usually, we can not change it. Moreover, braking deceleration is a limitation of our system, and it has a maximum value. So, the only parameter that we can control is the velocity.

To sum up, we should always consider the distance of our robot with the obstacles and limit the velocity so that the robot can brake effectively in case of a possible collision.

One might ask how it is possible to have a collision while checking the sub-trajectories for being collision-free, and we have a static environment in our case. We should always consider the probable failures that might happen during the operation, and we should make sure we have our system in a condition that can stop it effectively in emergencies.

Fox et al. [18] considered the same reasoning when they introduced the admissible velocity for their model. We use the same equation of motion and develop an admissible velocity for our model.

The mentioned equation of motion is the following:

$$V^2 = V_0^2 + 2.b.(r - r_0) \tag{5.5}$$

V and V_0 are the final and initial speed, respectively, b is the braking acceleration and $(r - r_0)$ is the distance to the obstacle. By putting V equal to 0, since we want to stop at most on the border of the obstacle and do some algebra we get:

$$V_a = \sqrt{2.dist(v, \delta).b} \tag{5.6}$$

where V_a is the *Admissible Velocity*, which means at each step this is the maximum velocity that we are allowed to impose to the robot. $dist(v, \delta)$ is the distance between the endpoint of the sub-trajectory generated by couples of (v, δ), and the nearest obstacle and b is the braking deceleration.

However, there is a problem. Recalling equation 3.20, one should consider that we are giving two velocities and two steering angles to the robot, which are front wheel and rear wheel velocities and steering angles. Nevertheless, the admissible velocity is the velocity of the robot. Our search space is on the front and rear wheel velocities (general coordinate space), but we have found the limitation on the overall robot velocity (configuration space). To map this limitation, we should use our robot's inverse kinematics, which is not an easy job. To solve this problem, we decided to create all possible overall velocities of each set of $\{V_f, V_r, \delta_f, \delta_r\}$, which are accessible in the dynamic window, and then omit those sets that generate an overall velocity bigger than the admissible one.

Summary

Here we summarize all the mentioned points and state how we implemented the Dynamic-Window approach.

Our approach is based on a step by the step optimization procedure. At each step, we should consider a dynamic window around our Velocities and Steering angles, as we mentioned, we should calculate the admissible velocity considering the distance of the robot form the nearest obstacles. After doing so, we should check all the possible generated *overall velocity* by the configuration of two input velocities and two input steering angles and omit those configurations which generate a velocity more than the admissible one.

Last, we give the selected configuration of velocities and steering angles to the kinematic model we have developed, and we generate what we call $sub-trajectories$ as shown in Figure 5.1.

The next step is to check which of these sub-trajectories are *collision free*, as discussed in the next subsection.

5.2.2 Collision Checking

As we want to generate a trajectory that the robot can follow, we should get sure that the robot will not collide with any obstacle in its path. Thus, we should check each of the paths generated by sets of velocities and steering angles to be collision-free.

The first step in collision checking is to define a safe zone around our robot. For doing so, we got a circle around the center of gravity of our robot, and then we checked if any point of the Sub-Trajectories are inside this circle or not. In finding obstacle points inside the circle, we detected its velocities and steering angle configuration as forbidden.

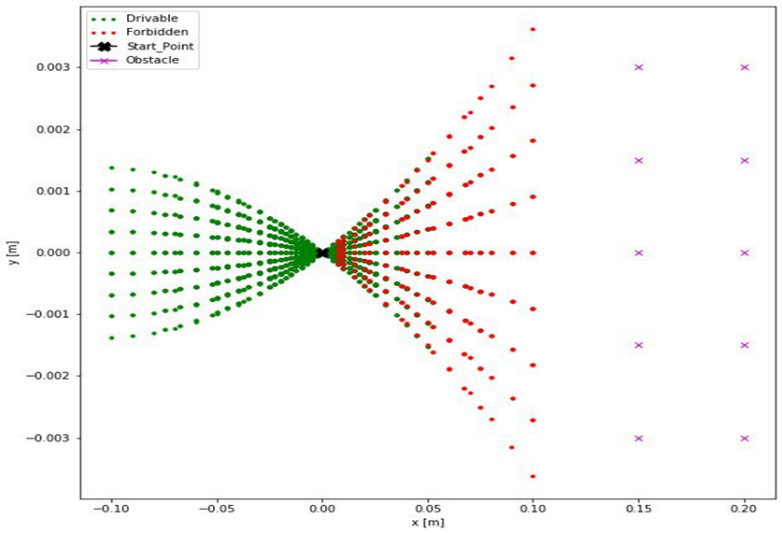


Figure 5.1: Sub-Trajectories generated by the chosen configuration. Notice that for them each Sub-Trajectory is according to a couple of linear and angular velocities, but in our case, each Sub-Trajectory is according to a set of four inputs. The inputs are V_f, V_r, δ_f, δ_r.

The next step is to choose one of the sub-trajectories as the optimal one and let the robot follow that to reach the next optimization step. We defined a cost function based on what F. Zhang et al. [8] introduced, which is explained in the next subsection.

5.2.3 Cost Function

State of the art

In 1997 Dieter Fox et al. [18] used this approach for the first time. They used this approach to plan a path for a synchro-drive mobile robot, and the cost function they used had three terms:

1. Target heading, which calculates the distance of the robot from the goal point
2. Clearance, which is the distance to the closest obstacle
3. Velocity, which is the velocity of the robot at each instance.

They wrote the equation for their cost function as:

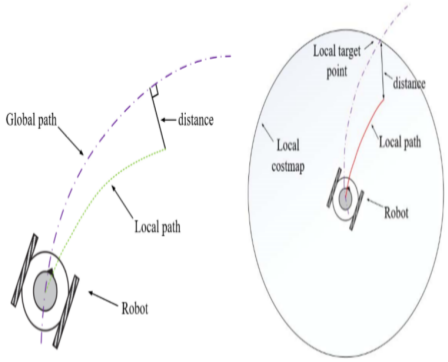
$$G(\nu, \omega) = \sigma(\alpha.heading(\nu, \omega) + \beta.dist(\nu, \omega) + \gamma.vel(\nu, \omega)) \tag{5.7}$$

They stated that this cost function is suitable when we have low speed.

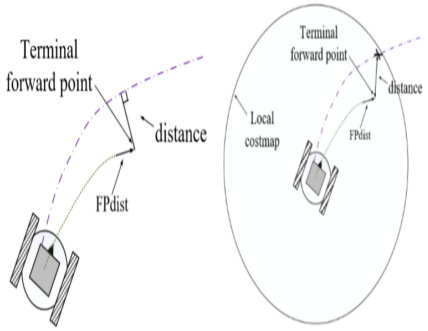
F. Zhang et al. [8] introduced a cost function with lots of terms to make sure they get the most optimized path in high velocity. Their cost function is as follows:

$$\begin{aligned} CostFunction(v, \delta) = \alpha.Obs(\nu, \omega) + \beta.Pdist(\nu, \omega) + \\ \gamma.Gdist(\nu, \omega) + \lambda.DirPath(\nu, \omega) \\ +\delta.DirGoal(\nu, \omega) + \eta.SVel(\nu, \omega) + \mu.MVel(\nu, \omega) \end{aligned} \tag{5.8}$$

where $Obs(\nu, \omega)$ is the distance evaluation sub-function of the robot from the obstacle, $Pdist(\nu, \omega)$ is the distance evaluation sub-function of the robot's sub-trajectory endpoint from the global path. $Gdist(\nu, \omega)$ is the distance evaluation sub-function of the robot's sub-trajectory endpoint from the local target point on the global path. $DirPath(\nu, \omega)$ is the distance evaluation sub-function of the robot's sub-trajectory endpoint from the global path's forward point. $DirGoal(\nu, \omega)$ is the distance evaluation sub-function between the forward point of the endpoint of the robot's sub-trajectory and the local target point on the global path. $SVel(\nu, \omega)$ is the smooth robot speed evaluation sub-function and $MVel(\nu, \omega)$ is the coefficient of each evaluation sub-function to improve the robot speed evaluation sub-function. α, β, γ, γ, λ, δ, η and μ are the weighting parameters of the cost-function.



(a) Schematic figure of the pad evaluation.



(b) schematic figure of path evaluation sub-function.

Figure 5.2: Schematic figures of cost function terms introduced by [8]

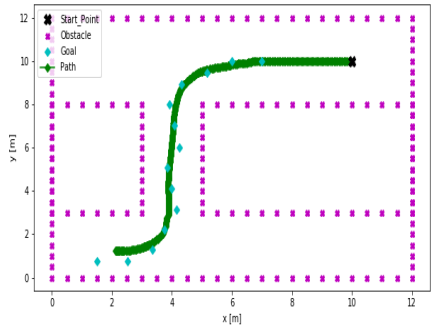
Such a cost function provides excellent performance for high velocity. However, as our application is in low velocity, we did not find it necessary to use such a vast cost function.

Our cost function

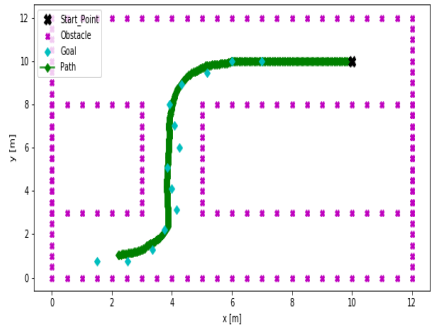
The first important difference between our case and the mentioned studies is that in our robot, each sub-trajectory is defined according to a set of four inputs, which are V_f, V_r, δ_f, δ_r. However, for being brief in notation, we have written each term as (v, δ). And then by some trial and errors, we find the following appropriate cost function:

$$Cost(v, \delta) = \alpha.ObsDist(v, \delta) + \beta.GoalDist(v, \delta) + \gamma.Heading(v, \delta) \tag{5.9}$$

where $ObsDist(v, \delta)$ is the distance between the final node of the sub-trajectory and the nearest obstacle, $GoalDist(v, \delta)$ is the distance between the final node of the sub-trajectory and the goal, $Heading(v, \delta)$ is the difference between the heading of the vehicle and the goal pose. and α, β, and γ are the weighting parameters that we tuned for our application. We also tried this cost function with only two terms, $ObsDist(v, \delta)$ and $GoalDist(v, \delta)$.



(a) Local path on a small map with considering $Heading(v, \delta)$ term in the cost function.

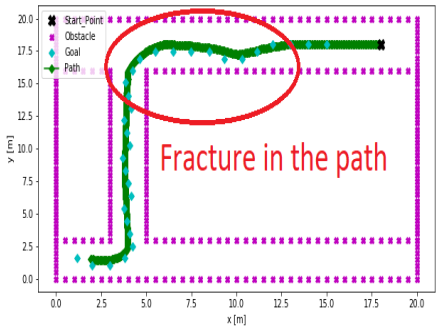


(b) Local path on a small map without $Heading(v, \delta)$ term in the cost function.

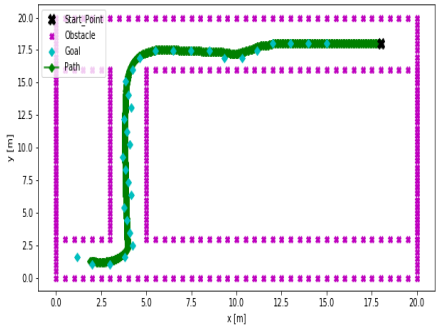
Figure 5.3: Comparison between the path generated with two cost functions on a small map.

Figure 5.3 shows that in a small map, there is no significant change in the quality of the path. Then we decided to test these cost functions on a larger map.

Figure 5.4 shows that on a large map presence of the heading term can cause some issues. Figure 5.4 (b) shows the path generated without that term. We can notice that the robot will get near to the obstacle after some time to make a turn with a smaller curvature radius. However, in figure 5.4 (a), the robot gets far again from the obstacle and generate a fracture in the path for turning it makes a larger radius of curvature, which we should avoid in agricultural fields as we want to use the minimum possible space for robot applications. The fracture happens since the local goal headings will force the robot to follow them instead of just guiding the



(a) Local path on a large map with considering $Heading(v, \delta)$ term in the cost function.



(b) Local path on a large map without $Heading(v, \delta)$ term in the cost function.

Figure 5.4: Comparison between the path generated with two cost functions on a large map.

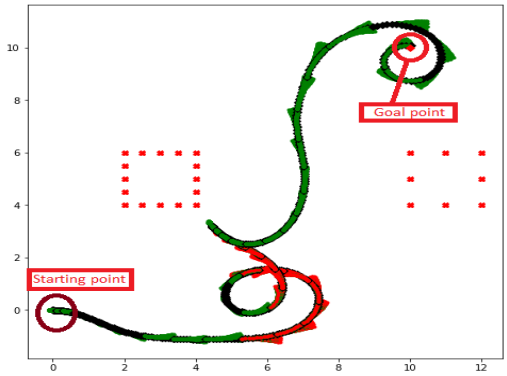
robot in the right direction, and as we discussed, these local goals are generated with a random technique, RRT*, which is fast but not precise. As a result, we decided to use a cost function with only two terms as:

$$Cost(v, \delta) = \alpha.ObsDist(v, \delta) + \beta.GoalDist(v, \delta) \tag{5.10}$$

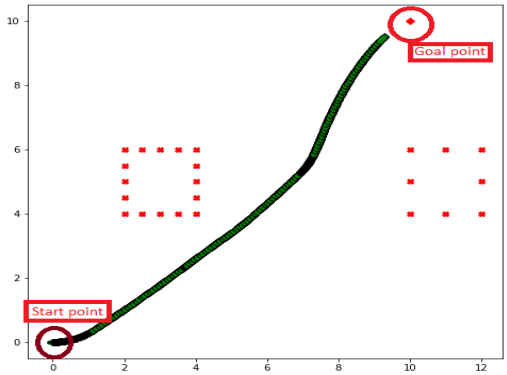
After calculating the cost of each sub-trajectory, we choose the sub-trajectory with the lowest cost. Then we perform the first point of the sub-trajectory, and we repeat all the steps. In this manner, DWA is used combined with a receding horizon technique and generates smooth paths.

We performed several tests to evaluate the path planner's performance, with and without the receding horizon technique.

As figure 5.5 illustrates, without receding horizon technique, DWA can generate a curvy path, and for every new map, the user needs to tune the cost function. On the other hand, we achieve a smooth and efficient path by using the receding horizon technique.



(a) DWA without receding horizon technique.



(b) DWA withreceding horizon technique.

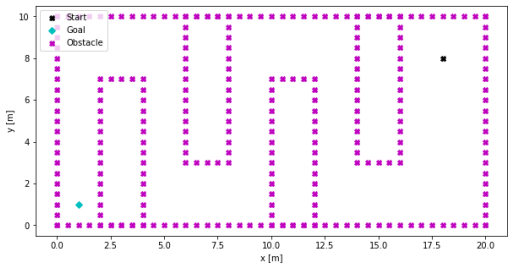
Figure 5.5: Comparison between DWA with and without receding horizon technique

Chapter 6

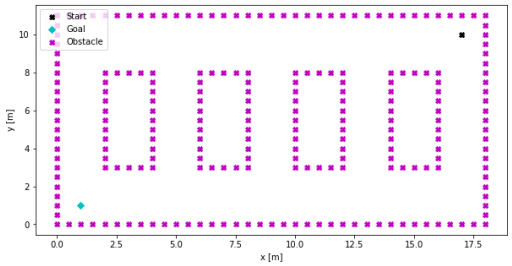
Results and discussion

After setting up the local path planner, we can give our planner module different maps to evaluate its behavior.

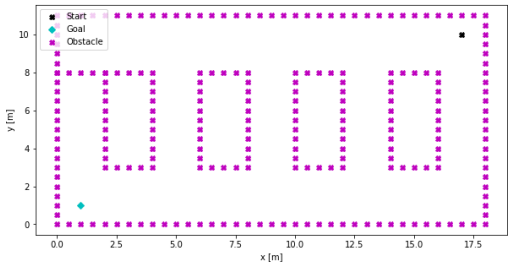
A map for our planner module is a set of nodes that some obstacles and the others are goal points. The two main maps that we provided to the planner are the following.



(a) Map 1, the hardest scenario for the planner.



(b) Map 2, a model of our farm.



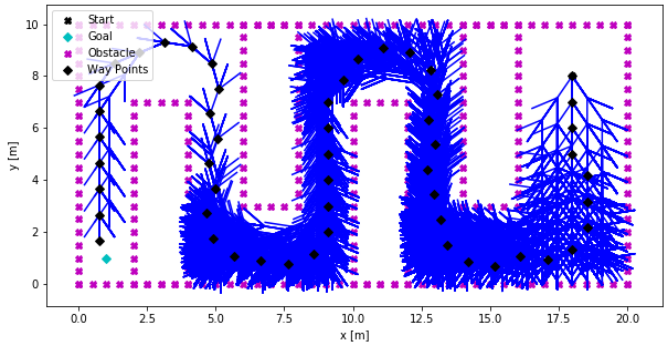
(c) Map 3, a model of our farm with a blocked passage.

Figure 6.1: Three main paths.

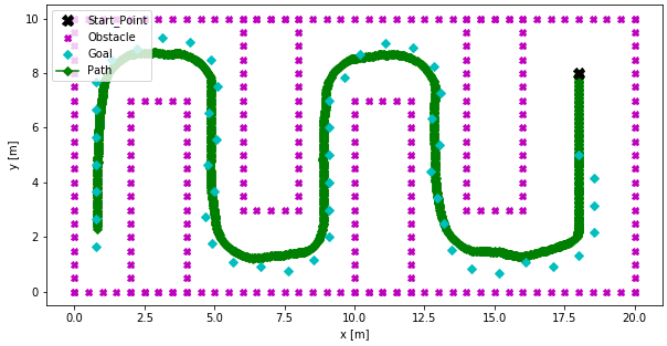
As it is mentioned in the legend, the node identifies with a black cross is our starting node. The purple nodes are obstacles, and the node with cyan color is the global goal. These maps are provided at first to our global path planner, and it generates the waypoints. The new map with the waypoints is provided to the local path planner, and it generates a smoother path among the waypoints.

Long, narrowed corridors characterize the first map, and, for this reason, it represents the most complicated scenario. This map shows the difference between RRT and RRT*, as discussed in chapter 4. We believed that if our planner can plan a path for this map, it can generate a path for every other path.

The results are the following.



(a) Global path on map 1.



(b) Local path on map1

Figure 6.2: Map 1 paths.

Figure 6.2(a) shows the results of the global path planner. The map size is 20 * 10 m. The robot initial pose parameters are $x_0 = 18m$, $y_0 = 8m$,and $\psi_0 = 3 * \pi/2$. The goal pose parameters are $x_f = 1m$, $y_f = 2m$, and $\psi_f = 3 * \pi/2$. The weighting parameters of the local path planner cost function, represented with equation 5.9, are $\alpha = 0.5$ and $\beta = -0.8$ $\gamma = 0$. as we discussed before we do not consider the final heading in our cost function to avoid strange behaviours in our path planning and to give the freedom to the planner to decide the best final heading. In case of a need to reach a special final heading, we can easily put the respective weight, γ, more than zero and the planner will take care of that situation.

The global path planner tries to find a path by generating a search tree and growing its branches. As this procedure is based on a fast and random approach, the generated path has low quality. However, as we discussed in the global path

planner chapter, thanks to this random approach, it is guaranteed that if there is a feasible way, the global planner can find it, as it did in this example.

The map containing the global path is then provided to the local planner module to make a smoother path. As Figure 6.2(b) shows, the second planner can generate a smoother path thanks to its optimization technique and its implemented cost function. This final path is then provided to a controller module.

After the first successful test, we provided a model of our farm to our planner module, indicated as map2 (figure 6.1 (b)). The local planner cost function parameters are kept the same for all the maps till the end. The robot initial pose parameters are $x_0 = 17m$, $y_0 = 10m$,and $\psi_0 = 3 * \pi/2$. The goal pose parameters are $x_f = 1m$, $y_f = 1m$, and $\psi_f = \pi$.

Since there can be multiple ways toward the global goal on this map, we studied the effect of the initial state on the robot's generated path.

Looking at the cost function behavior along the path clarifies why we should always use a global path planner and a DWA local path planner.

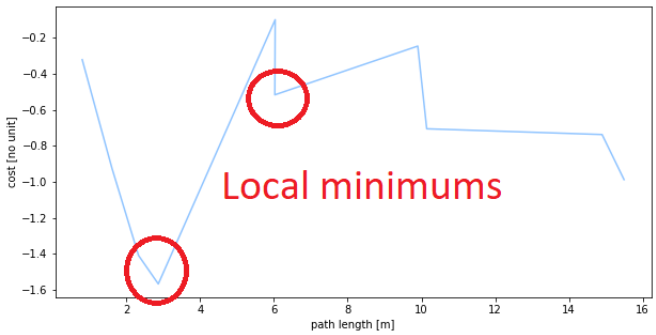


Figure 6.3: cost function behaviour.

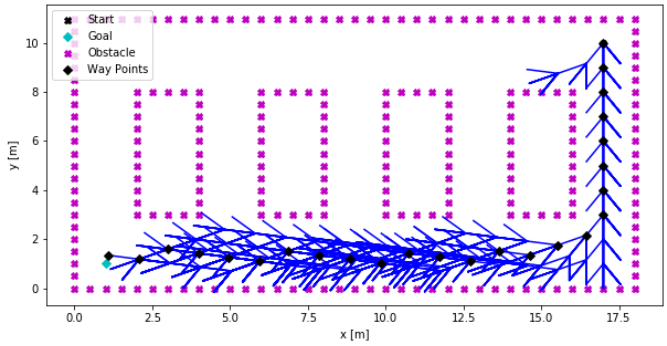
Figure 6.3 illustrates that during the path, the cost function has several local minimums, and without a global path planner, it can get stocked in these local minimums.

Figure 6.4 shows the generated paths for a model of our farm (map 2). As expected, the planner module was able to find the shortest path toward the global goal.

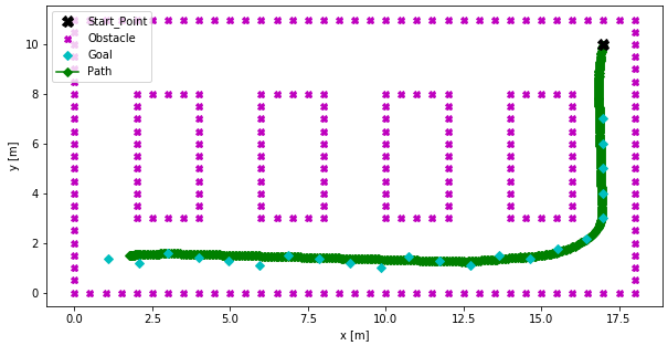
Then we decided to test the planner by new initial heading π. Figure 6.5 shows the results.

As Figure 6.5 shows, the direction of the path is changed, and the planner decided a completely different path for the robot. It is because the planner module uses the robot's kinematic model to generate the path and generate and optimize the path for that kinematic model. As the robot has a different heading in this example, the cost function has noticed it is more energy-efficient for the robot to continue on a line with the same heading of the initial heading of the robot, although it could choose the previous path and both of them has the same length and curvature. This example showed us that our cost could also take care of energy consumption for the robot.

After this example, we decided to increase the nodes generated by the global path planner to see its effect on the local path planner. One should notice that increasing the number of nodes can increase the time consumption of the global



(a) Global path on map 2 with the initial heading of $3\pi/2$.



(b) Local path on map 2 with the initial heading of $3\pi/2$.

Figure 6.4: Map 2 paths with the initial heading of $3\pi/4$.

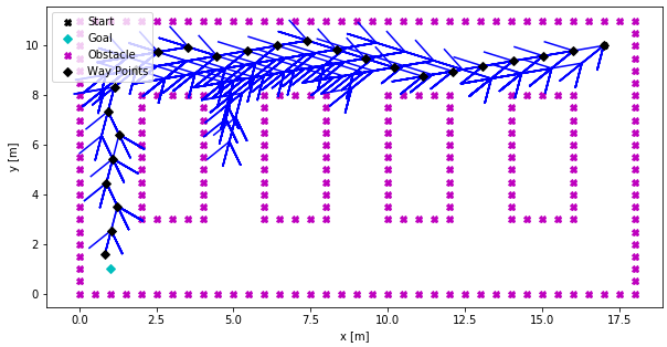
path planner significantly and should be done only in case of a significant change in the global path planner results.

Figure 6.6 showed us that there is no significant change in the quality of the generated local path by providing a high-resolution global path. Consequently, it is not advisable to increase the number of generated nodes by the global path planner.

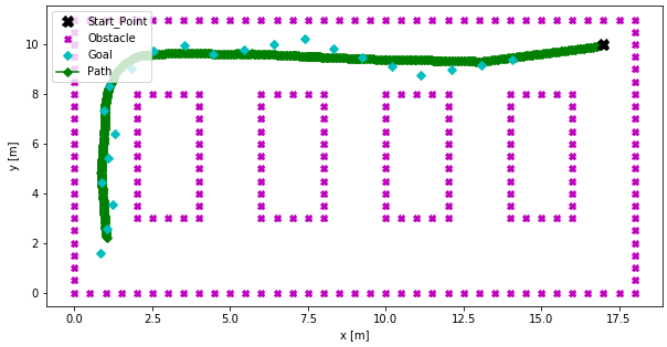
Later, we decided to study the path planner's behavior if we put an obstacle in the corridor that he has already chosen. For this reason, we provided map3, figure 6.1(c).

Figure 6.7 shows the results of the last test. The global path planner could successfully find another path as soon as it reached the obstacle on the shortest path.

Providing the paths as a reference signal to the motion controller that [9] developed. The results are presented in figures 6.8 and 6.9. These figures illustrate that we have reached our goal, providing a useful reference to the robot's motion controller.

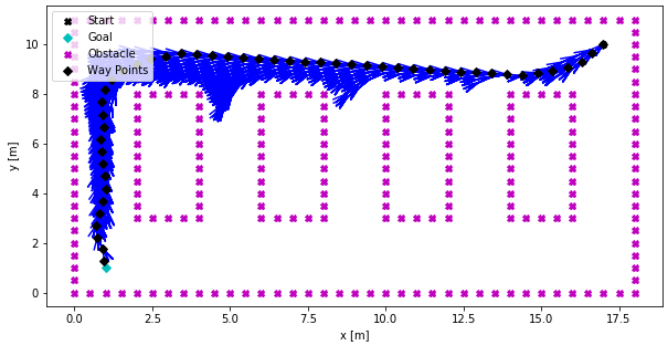


(a) Global path on map 2 with the initial heading of π.

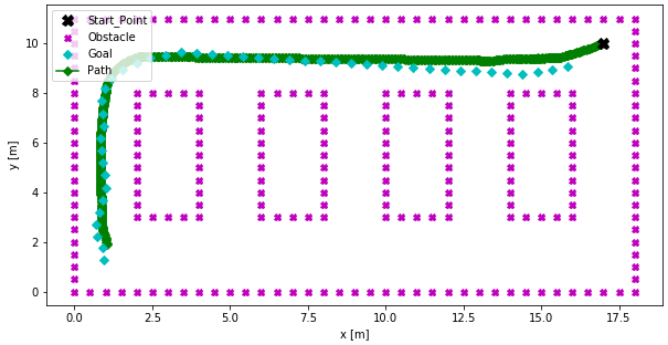


(b) Local path on map 2 with the initial heading of π.

Figure 6.5: Map 2 paths with the initial heading of π.

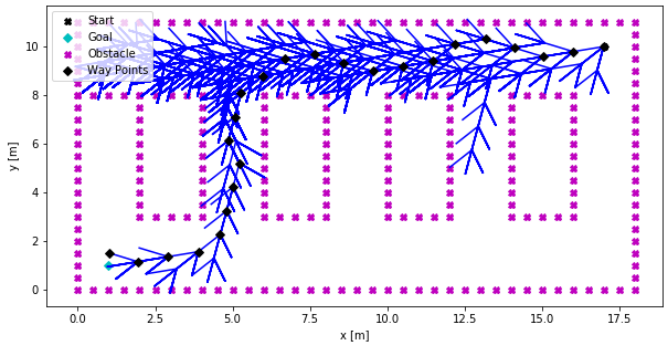


(a) Global path on map 2 with the initial heading of π, with extra nodes.

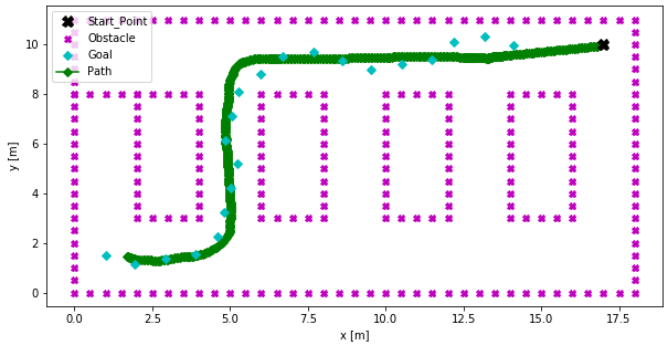


(b) Local path on map 2 with the initial heading of π, with high resolution global path planner.

Figure 6.6: Map 2 paths with the initial heading of π, with high resolution global path planner.



(a) Global path on map 3.



(b) Local path on map 3.

Figure 6.7: Map 3 paths.

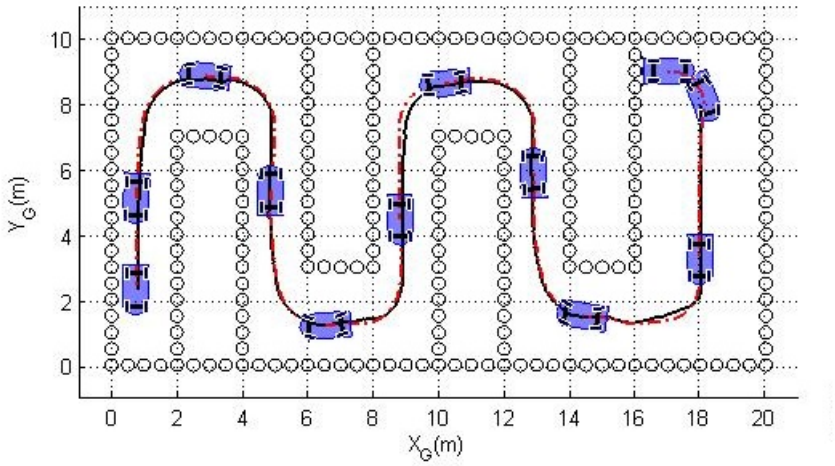


Figure 6.8: The results of the controller on following the provided path with long and narrow corridors [9].

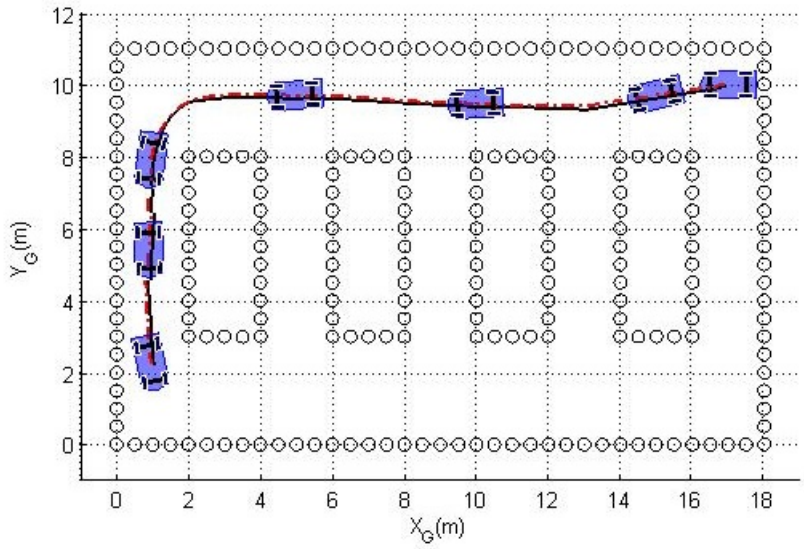


Figure 6.9: The results of the controller on following the provided path on the farm [9].

Chapter 7

Conclusion

In this thesis report, we discussed our study's results on creating a planning module for a 4WS mobile robot. At first, this study aimed to understand the needed modules to create autonomy for a mobile robot that we discussed in chapter1. After choosing our autonomy hierarchy, we aimed to work on the motion planning module.

Our motion planning aimed to generate a $feasible$ and *optimal* path for the robot. After studying the literature, we decided to choose the DWA technique for our path planner as it is a receding horizon technique and can generate a path that is a useful reference for the robot's motion control module.

As the DWA technique has an intrinsic possibility of getting stocked in the local minimum, we studied the planning module's division to global path planner and local path planner. As the RRT* technique showed fast and reliable results in the previous studies at the literature, we decided to choose this technique as our global path planner to guide our local path planner and to avoid getting stocked in the local minimum.

After passing the first tests that were discussed in detail in chapters 4 and 5 and tuning the cost function of both planners, we planned final tests for our planner module to study its robustness and performance in different scenarios.

Chapter 6 discussed those scenarios that were challenging for a mobile robot to plan a path in them. We also studied different computation loads for our planner to test its performance in different modes.

Finally, we provided an efficient, fast, and optimal planner module that can be used for our mobile robot and any other mobile robot which has a 4WS wheel based locomotion.

The planner module that we have developed can easily be used as software packages for being implemented in ROS and can also be used for robots with a different kinematic module. The autonomy engineer changes the implemented kinematic model in each planner.

This study was an effort to pave the path for future works on autonomous mobile robots in agriculture, and we tried to provide a planner module that shows a robust and efficient performance for agricultural fields.

Bibliography

[1] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[2] Society of Automotive Engineers. Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles j3016-201806, url: https://www.sae.org/binaries/content/gallery/cm/articles/news/2018/12/j3016-levels-of-driving-automation-12-10.jpg.

[3] Stack Exchange. Url: https://math.stackexchange.com/questions/2404946/defining-gaussian-and-mean-curvature-for-an-architect-non-mathematician/2405027.

[4] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, et al. Junior: The stanford entry in the urban challenge. *Journal of field Robotics*, 25(9):569–597, 2008.

[5] Jeong hwan Jeon, Raghvendra V Cowlagi, Steven C Peters, Sertac Karaman, Emilio Frazzoli, Panagiotis Tsiotras, and Karl Iagnemma. Optimal motion planning with the half-car dynamical model for autonomous high-speed driving. In *2013 American control conference*, pages 188–193. IEEE, 2013.

[6] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Robotics: modelling, planning and control*. Springer Science & Business Media, 2010.

[7] Danwei Wang and Feng Qi. Trajectory planning for a four-wheel-steering vehicle. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164)*, volume 4, pages 3320–3325. IEEE, 2001.

[8] Fuhai Zhang, Ning Li, Tiegang Xue, Ying Zhu, Rupeng Yuan, and Yili Fu. An improved dynamic window approach integrated global path planning. In *2019 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 2873–2878. IEEE, 2019.

[9] Simone Scivoli. Modelling, control and simulation of an unmanned ground vehicle for agriculture 4.0. Master's thesis, Politecnico di Torino, 2020.

[10] Mikell P Groover, Mitchell Weiss, and Roger N Nagel. *Industrial robotics: technology, programming and application*. McGraw-Hill Higher Education, 1986.

[11] Sara Bragança, Eric Costa, Ignacio Castellucci, and Pedro M Arezes. A brief overview of the use of collaborative robots in industry 4.0: human role and

safety. In *Occupational and Environmental Safety and Health*, pages 641–650. Springer, 2019.

[12] N Vamshidhar Reddy, AV Vishnu Vardhan Reddy, S Pranavadithya, and J Jagadesh Kumar. A critical review on agricultural robots. *International Journal of Mechanical Engineering and Technology*, 7(4):183–188, 2016.

[13] Juan Jesús Roldán, Jaime del Cerro, David Garzón-Ramos, Pablo Garcia-Aunon, Mario Garzón, Jorge de León, and Antonio Barrientos. Robots in agriculture: State of art and practical experiences. *Service Robots*, 2018.

[14] Chris Carter. *Facts and Practice for A-level: Physics*. Oxford University Press, 2001.

[15] Alonzo Kelly and Bryan Nagy. Reactive nonholonomic trajectory generation via parametric optimal control. *The International Journal of Robotics Research*, 22(7-8):583–601, 2003.

[16] Luigi Di Puglia Pugliese, Daniele Ferone, Paola Festa, and Francesca Guerriero. Shortest path tour problem with time windows. *European Journal of Operational Research*, 282(1):334–344, 2020.

[17] David González, Joshué Pérez, Vicente Milanés, and Fawzi Nashashibi. A review of motion planning techniques for automated vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 17(4):1135–1145, 2015.

[18] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1):23–33, 1997.

[19] MB Markus Suchi and M Vincze. Meta-heuristic search strategies for local path-planning to find collision free trajectories. In *Proceedings of the Austrian Robotics Workshop (ARW-14)*, pages 36–41, 2014.