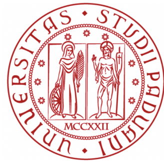


Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in
Ingegneria dell'Automazione

Design and Implementation of a Model Predictive Control Algorithm for Ackermann Steering Vehicles

Relatore:
Prof. Alberto Pretto
Correlatore:
Daniele Evangelista

Laureando:
Francesco Marchetti
1150102

Anno Accademico 2021/2022

Abstract

In this century one of the main objective in the automotive branch, is surely the continuous improvement of the autonomous driving technology until they reach the complete control of the vehicle. The purpose of this thesis is study and implement an MPC non-linear control algorithm that, once the target is decided, computes the real-time ideal path for an autonomous driving vehicle and is able to avoid possible obstacles while driving.
The MPC has also be designed to be robust and usable in different scenarios. This algorithm has been then tested by implementing some graphic simulations with Gazebo in the ROS framework.
The obtained MPC has also been tested on an RC car opportunely equipped with external control boards and sensors that can map the surrounding environment and recognize eventual obstacles.

Contents

1 Introduction **1**

1.1 Autonomous Driving . . . 1

1.1.1 Classification of autonomous vehicles . . . 3

1.1.2 History of the first autonomous veichles . . . 5

1.2 Techmo . . . 11

2 Theoretical Background **13**

2.1 MPC Algorithm . . . 14

2.1.1 Non Linear MPC . . . 18

2.1.2 IPOPT Solver . . . 24

2.2 Ackermann Model Description . . . 25

3 Hardware Architecture **31**

3.1 Arduino Uno . . . 32

3.2 Raspberry Pi 3 . . . 33

3.3 RC Car . . . 37

3.3.1 Choose the RC car . . . 37

3.3.2 3D Printed Parts . . . 38

3.3.3 Assembling . . . 39

3.3.4 Working Overview . . . 40

4 Tracking **43**

4.1 Software Tools . . . 43

4.1.1 Arduino IDE . . . 44

4.1.2 Robot Operative System . . . 45

4.1.3 Gazebo Simulator . . . 48

4.2 Worlds Creation . . . 50

4.2.1 World Description . . . 50

4.2.2 Mapping . . . 52

4.3 Path Tracking 55
4.3.1 Obstable Avoidance 56
4.3.2 TEB Local Planner 57

5 Experimental Setup 61
5.1 Connection Setup 61
5.1.1 Raspberry Image 61
5.1.2 Connection Between Raspberry and Laptop 62
5.1.3 Hardware Implementation 64
5.1.4 Raspberry and Arduino talk to each other 65
5.1.5 Sensors 67
5.1.6 Hector-SLAM mapping 68
5.1.7 Localization and Navigation 70
5.2 Coordinate Frames 70

6 Graphical Simulations 73
6.1 Simulations 73
6.1.1 Straight World 74
6.1.2 Square World 74
6.1.3 Turtle World 75
6.1.4 Custom Map 75
6.1.5 Small House Map 77
6.2 ROS Navigation Turning 78
6.2.1 Velocity and Acceleration 78
6.2.2 Global Planner 78
6.2.3 Costmap 79

7 Practical Simulations 81
7.1 Tests of Mobile Robot Motion 81
7.1.1 Point to Point Motion 81
7.1.2 Test of the RC Car 83

8 Conclusions and Future Works 85
8.1 Conclusions 85
8.2 Future Works 86

Bibliography 87

Chapter 1

Introduction

Reaching Level 5 of autonomous driving means that the vehicle is able to drive without the presence of anyone or any external control. To do this the vehicle must be equipped with sensors that can analyze the environment in an appropriate neighborhood in order to find the best and secure path possible that achieve the planned task.
Particularly in the autonomous driving for the people transport, but also on every other application, the safety and the robustness are the most important factors since people's lives are being put into the hands of a machine.
Handling accuracy depends on the type of application and vehicle on which you are working.

1.1 Autonomous Driving

According to a Morgan Stanley study [1], autonomous cars could save US $1.3 trillion a year. These earnings would be split between fuel savings from better traffic management, the lowered health care expenses due to the reduction of the number of road kills, and the productivity gains from spending less time driving. On the welfare side, autonomous cars could allow elderly and disabled people to gain back some mobility and they can reduce the need for parking space in cities as well. A 2016 study by the American Automobile Association Foundation for Traffic Safety [3] estimated that the average American motorist was driving around 50 km a day. In a week, this can represent several hours that cannot be used for leisure purposes. By delegating the driving task to an autonomous

vehicle, these hours would be freed, resulting in a higher quality of life.

Another benefit of autonomous vehicle hitting the mainstream market is inter-vehicle communication(Figure 1.1). Nowadays, one human driver can only communicate with its peers using simple tools (directional signals, braking signals, etc). Autonomous vehicles could take advantage of a wireless communication system using a frequency band and use traffic optimization algorithms to reduce traffic jams. This implies a 33% reduction of total travel times and a significant reduction of traffic jams, which would have a positive environmental impact since it has been estimated that reducing them could lower the carbon dioxide emissions by 20% [4].

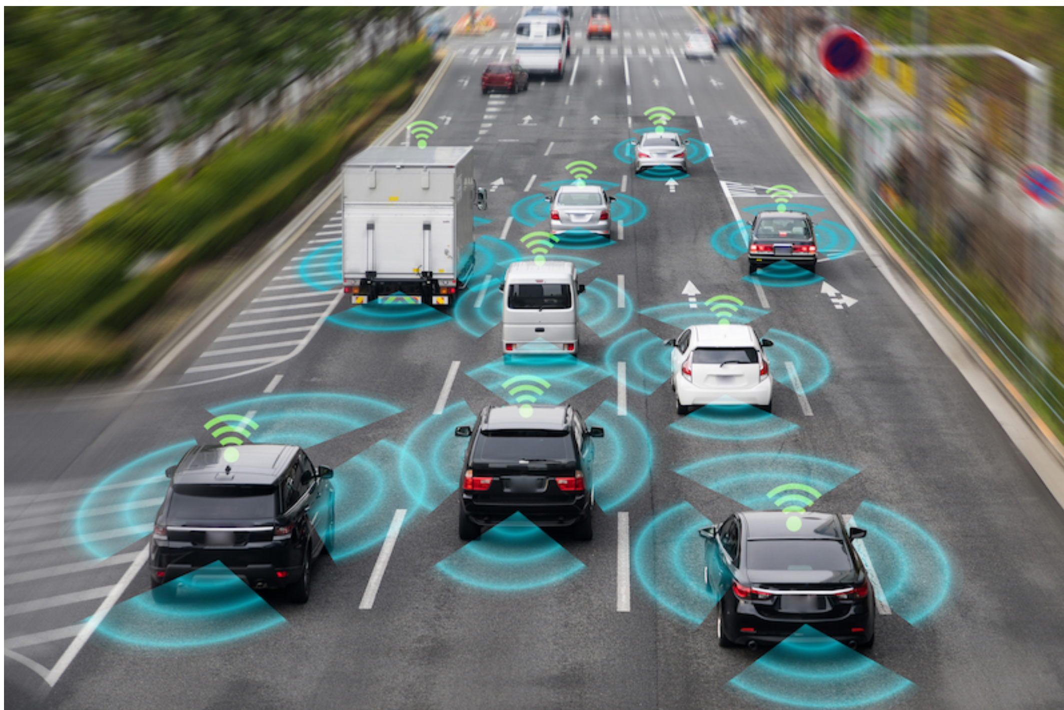


Figure 1.1: A potential communication between different vehicles

A complementary approach to designing cooperatively driving vehicles is to enable individual car's steering controllers to handle the car at its limits of adherence on the road. It can be imagined that these scenarios include high speed maneuvering control, and a racing context would be very adapted to provide such constraints.

1.1.1 Classification of autonomous vehicles

The current progress of autonomous driving technologies is classified using the SAE levels. Since its initial launch in 2014, as quoted in [2]: " *SAE J3016™ Recommended Practice: Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*", commonly referenced as the SAE Levels of Driving Automation™, has been the industry's most-cited source for driving automation. With a taxonomy for SAE's six levels of driving automation, SAE J3016 defines the SAE Levels:

- **Level 0**: No automation → the human driver must have maximum attention and total control of the vehicle. Almost all cars ever built from 1885 to 2010 have lacked self-driving capacities. With this level of automation the cars may have simple warning or intervention systems, but they all require the driver to brake, steer, park, maneuver and accelerate manually.

- **Level 1**: Driver assistance → The system can handle either steering or acceleration/deceleration using information about the driving environment, and the human driver is expected to perform all remaining aspects of the dynamic driving task.

- **Level 2**: Partial automation → The vehicle can now handle both steering and acceleration/deceleration using information about the driving environment. Actually Mercedes™ has developed the *Intelligent Drive*, an assistance and security system that enhance the comfort and the driving experience. In parallel, Toyota™ presented *Safety Sense* that consists in a few of security assistance, in particular: a pre-collision system, a warning if you go on the other lane, a recognition system of road signs and finally an adaptive cruise control.

- **Level 3**: Conditional automation → The vehicle handles the full driving task but may request the human driver to take control in some specific cases and expect an appropriate response. The first system that can reach the level 3 is called *AI traffic jam pilot* it can be used until $60 km/h$ are reached and it was introduced in Audi A8™.

- **Level 4**: High automation → The car may still request an human intervention in certain cases but now it should not assume an appropriate decision from the driver.

- **Level 5**: Full automation: The vehicle handles the full driving task.

For legal and above all ethical problem due to the responsibility for accidents, the highest level reached for the big distribution cars is Level 3. In Italy, for example, unlike other countries such as the United States, the car company can sell only cars with ADAS that reach the Level 2 of Autonomous driving since the law can not permit that the driver does not intervene at all in the vehicle. It would appear that Italy will also be ready for the transition in the current year Elon Musk, the famous CEO of Tesla Motors™, announced that in the next 5 years his vehicles will be equipped with a Level 5 automation.

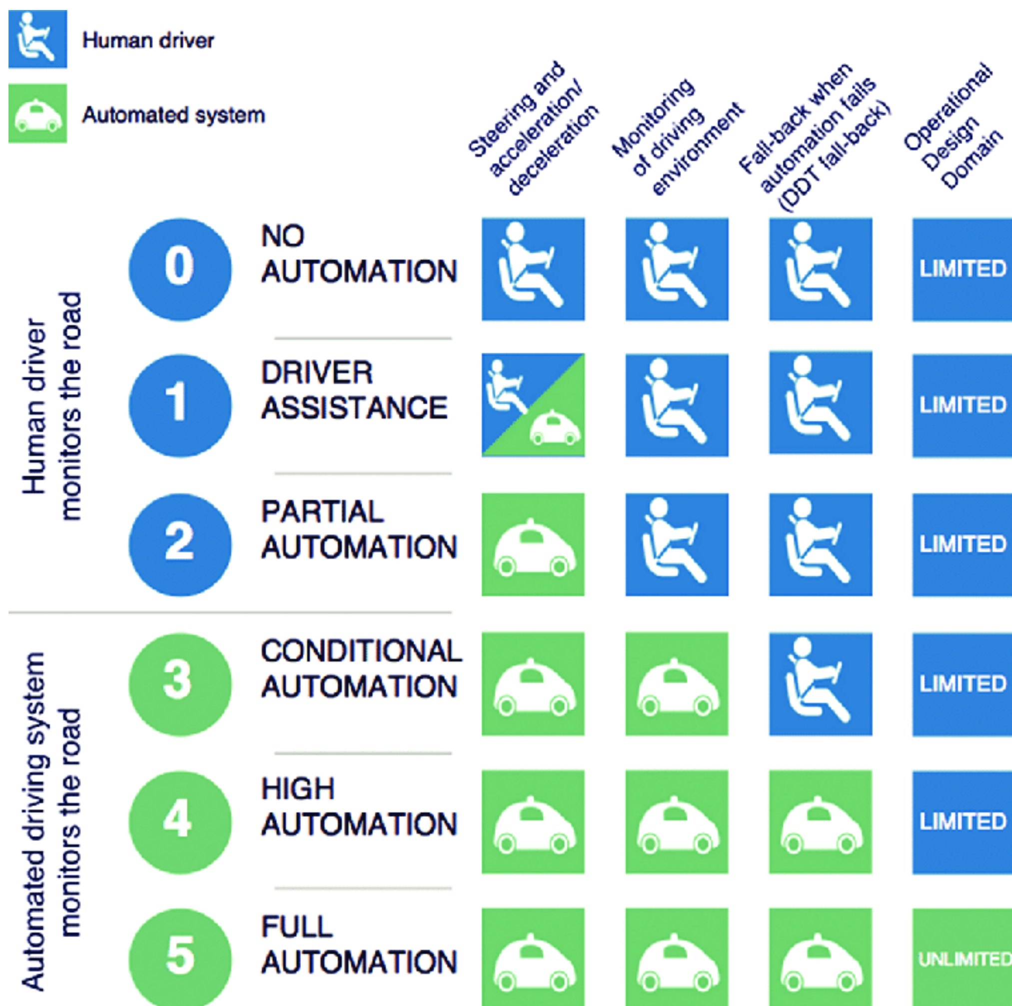


Figure 1.2: SAE J3016 levels of driving automation

1.1.2 History of the first autonomous veichles

It is interesting to analyze the evolution of the automated vehicles: the first "autonomous" cars were more like robots than like real vehicles until the arrival of more solutions such as the revolutionary *"Pop-Up"* designed by Italdesign™, Audi™ and Airbus™.

Shakey

From 1966 to 1972 at SRI international in the USA *Shakey*(Figure 1.3) was designed and implemented the first Robot having an autonomous behaviour. The Robot was composed of an oriented camera, ultrasound that was able to compute the distance and touch sensors. Shakey, indeed, was connected to a computer with a Radio Frequency(RF) connection that managed the navigation and the exploration.

Figure 1.3: A photo of Shakey robot

ALV: Autonomous Land Vehicle

From 1985 to 1988 in the Strategic Computing Program, in the United States the *DARPA Autonomous Land Vehicle*(ALV)(Figure 1.4) was developed, built from a standard 8 wheels vehicle that could reach $72km/h$ in normal streets and $29km/h$ in off-road streets. The ALV could load up to 6 pallets of electronic devices using a power of $12kW$ Diesel engine. On the ALV a sensor system

was also fitted including a color camera and a laser scanner that gave back data every 1-2 seconds. In the first demonstration(in 1985) of this particular vehicle, the ALV followed the road for $1km$ with constant velocity of $3km/h$. In the subsequent years another demonstration was done in which it followed a path, with different curves and in different types of terrain, for $4.5km$ with a velocity of $10km/h$. Finally, in the 1987, a demonstration took place with an average velocity of $15.5km/h$ for $4.5km$ during which it followed a path with different types of terrain, with different road widths and with some obstacles that the ALV had to avoid.

Figure 1.4: Autonomous Land Veichle

VaMP

From 1933 to 1955 was developed *VaMP*(Figure 1.5), in Germany at the Munchen Bundeswhr University, one of the first real autonomous driving cars, based on a Mercedes 500SEL™. It could control the breaks, the throttle and the steering with the help of a computer. This car was able to drive for long distances, also in traffic condition, without the human help. This was possible thanks to artificial vision that could detect and avoid the moving obstacles with only 4 cameras: 2 in the front and 2 in the back. In 1995 it was tested in a really long travel from Munich to Odense(Danimank); the car drove for $1600km$, 95% of which in a completely autonomous way.

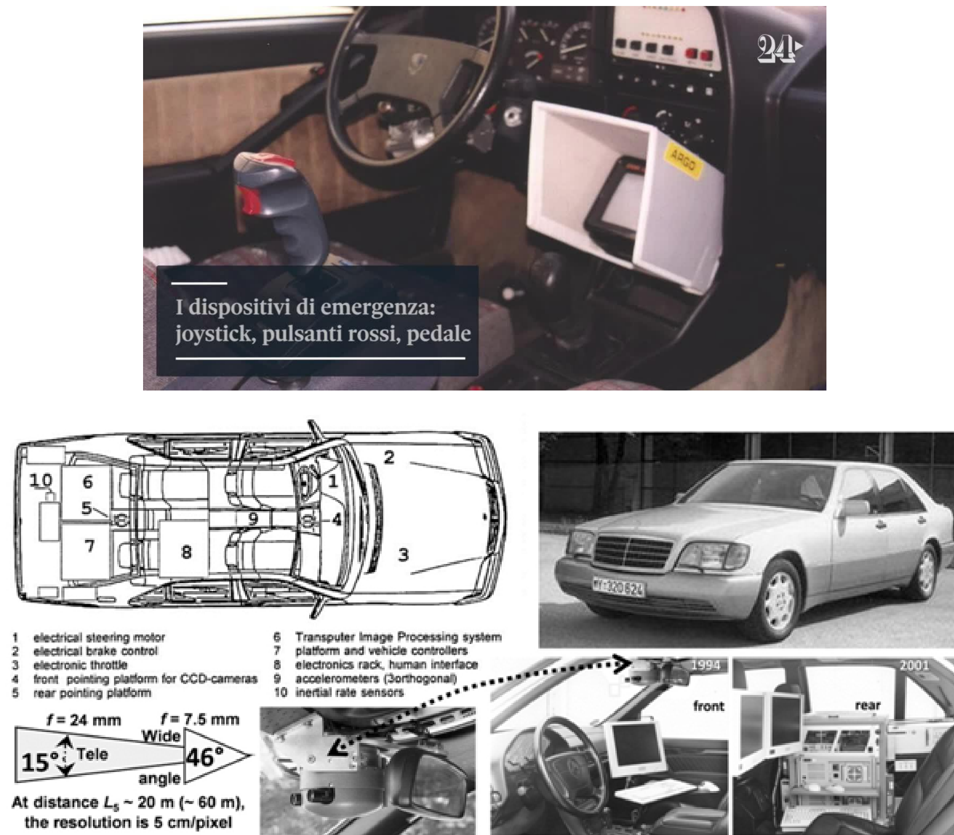


Figure 1.5: The Italian ARGO car above and the Mercedes VaMP below

ARGO

Italy landed in the world of autonomous driving in 1998 with *ARGO* that has been developed at the Information Engineering department in University of Parma. ARGO was based on a Lancia Thema 2000™ that mounted an electric motor directly on the steering column which allowed the vehicle to steer. The car also has passive sensors like a stereoscopic vision system which allowed the car to extract information about the surrounding environment so that it could follow the lane, locate obstacles and possibly make lane changes. It was subjected to a long test in which 2000km were traveled on Italian roads. Argo was able to drive 94 % of the route in total autonomy.

Waymo

A more recent and very avant-garde result has been made from a Google™ company called Waymo™. *Waymo LLC* is an American autonomous driving technology development company. It is a subsidiary of *Alphabet Inc.*, the parent company of

Google™. Waymo™ operates a commercial self-driving taxi service in the greater Phoenix area called "Waymo One", with Arizona fully mapped. In October 2020, the company expanded the service to the public, and this meant that it was the only self-driving commercial service that operated without safety backup drivers in the vehicle at that time. Waymo™ also developed driving technology for use in other vehicles, including delivery vans and Class 8 tractor-trailers for delivery and logistics.

Waymo is run by co-CEOs Tekedra Mawakana and Dmitri Dolgov. The company has raised $5.5 billion in multiple outside funding rounds. it has numerous partnerships with leading automotive manufacturers (including Nissan-Renault™, Stellantis™, Jaguar Land Rover™, and Volvo™) to fully integrate Waymo's technology into their vehicles.

It is interesting to briefly look at the key historical steps of the company: Waymo began from Google's development of self-driving technology in 2009 (Figure 1.6), when they set out on a challenge to drive autonomously over ten uninterrupted 100-miles routes in a Toyota Prius™ vehicle.

Figure 1.6: Self driving Toyota Prius™

In 2015, Google™ explored what fully autonomous cars could be with the Firefly(Figure 1.7), the car that Google™ entirely built itself that looked like no other car on the roads. The Firefly was a vision of the science-fiction future that self-driving cars could provide for the costumers: They had no steering wheel and no dashboards just two seats and some buttons, letting people sit in their pods as they drifted peacefully across town, checking their phones, and not having

to worry about traffic or weather conditions besetting the world outside this prototype. The cars evoked a friendlier world, where cars were not things to be feared, nor the harbingers of thousands of deaths across the world. That year, Steve Mahan, who's legally blind, took the world's first fully autonomous ride on public roads in Austin.

Figure 1.7: Firefly, the car built by Google™

In December 2016, the self-driving project was renamed Waymo and became a new start-up company that is part of Alphabet Inc.™ (is a holding company that gives ambitious projects the resources, freedom, and focus to make their ideas happen). The name Waymo was derived from its mission, *"a new way forward in mobility."*.

In 2017, they partnered with FCA to introduce a modified version of the 2017 Chrysler Pacifica Hybrid minivan™ (Figure 1.8) to their fleet: their first vehicle was built on a mass-production platform designed for and integrated with the Waymo Driver. In the same year they invited residents in Phoenix to join the first public trial of autonomously driven vehicles. Their feedback has been integral in helping shape their technology, service, and customer experience.

In 2018 Waymo launched the world's first commercial autonomous ride service: the *Waymo One*. The service offers fully autonomous rides in the Phoenix Metropolitan Area, for the first 400 users that singed up in the website, to try out a test edition of Waymo's transportation service.
In 2020 Waymo expanded its area of interest with *Waymo Via* (Figure 1.9), a trucking and delivery pilot programs, exploring how logistics partners could move goods more safely and efficiently with the Waymo Driver. In the same years they

Figure 1.8: Chrysler Pacifica Hybrid Minivan™

released an app in the United States that every one could download and instantly hail a fully autonomous ride in their restricted service territory.

Figure 1.9: Waymo Via

1.2 Techmo

Techmo Car S.p.A. is a world leader in the engineering and production of high-end "tailor made" mobile and stationary equipment for the aluminium industry.

The company was founded in 1961 by dr. Franco Zannini and since the beginning it was focused on providing original and state-of-the-art solutions to problems related to metal production. Techmo was the first company in the world to conceive and fabricate specialized vehicles for the aluminium electrolysis in the 60's of the past century.

Nowadays Techmo products are appreciated by the most demanding aluminium producers, adopting every kind of smelter reduction technologies, located in more than 40 countries.

In order to keep the quality of its products high, its research and development programs are focusing on multiple objectives, such as the increase of the durability of its products in the years and the improvement of the efficiency of the production process while reducing production, running and maintenance costs.

Figure 1.10: Techmo:Transport and Material handling

An other important goal for Techmo is the improvement of the environmental aspects. This can be obtained by reducing the carbon footprint of its equipment, increasing the recyclability of the used components and making the comfort and safety conditions for the operators better.

With these aims, nowadays Techmo is collaborating with the University of Padova to try to make each vehicle autonomous and able to move and perform

its tasks without the need for a human to drive, this will allow greater efficiency and safety. Naturally it is not only necessary to make the vehicle self-driving but, depending on the type of vehicle and its task, it will also be necessary to study a way to make it perform it autonomously and obviously this has nothing to do with autonomous driving.

Chapter 2

Theoretical Background

Model Based Predictive Control(MPC) originated in the late seventies and has developed since then. The term Model Predictive Control does not designate a specific control strategy but a very ample range of control methods which make an explicit use of a model of the process to obtain the control signal by minimizing an objective function. These design methods lead to linear controllers which have practically the same structure and present adequate degrees of freedom. The ideas appearing in greater or lesser degree in all the predictive control families are basically:

- *Explicit use of a model to predict the process output at future time horizon*
- *Calculation of a control sequence minimizing an objective function*
- *Receding strategy, so that at each instant the horizon is displaced towards the future, which involves the application of the first control signal of the sequence calculated at each step.*

The various MPC algorithms only differ amongst themselves in the model used to represent the process, the noises and the cost function to be minimized.

2.1 MPC Algorithm

Model Predictive Control (MPC) is a widely spread technology in industry for control design of highly complex multivariable processes.

The idea behind MPC is to start with a model of the open-loop process that explains the dynamical relations among system's variables (command inputs, internal states and measured outputs). Then, constraint specifications on system variables are added, such as input limitations (typically due to actuator saturation) and desired ranges where states and outputs should remain. Desired performance specifications complete the control problem setup and are expressed through different weights on tracking errors and actuator efforts (as in classical linear quadratic regulation). The rest of the MPC design is automatic.

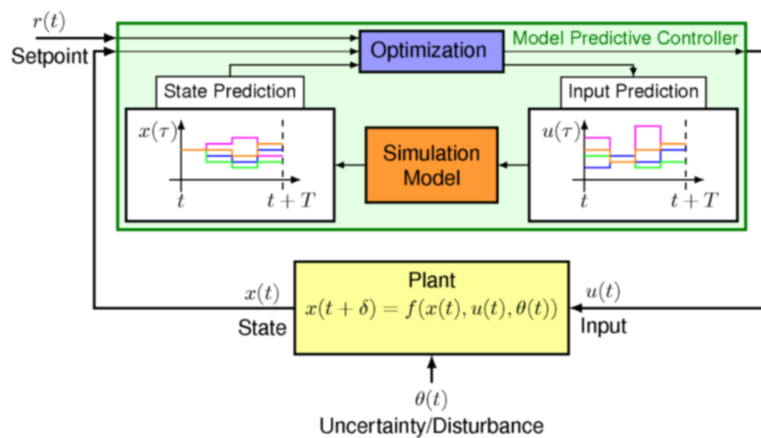


Figure 2.1: Block diagram of a model predictive controller in a feedback loop with a plant.

First, an optimal control problem (based on the given model, constraints, and weights) is constructed and translated into an equivalent optimization problem, which depends on the initial state and reference signals. Then, at each sampling time, the optimization problem is solved by taking the current state as initial state of the optimal control problem. For this reason the approach is called *"predictive"*. The optimal control problem is formulated over a time-interval that starts at the current time up to a certain time in the future.

The result of the optimization is an optimal sequence of future control moves and only the first sample of such a sequence is actually applied to the process; the remaining moves are discarded. At the next time step, a new optimal control problem based on new measurements is solved over a shifted prediction horizon. For this reason this approach is also called *"rolling-horizon"* control.

MPC provides near-optimal performance. However, there exists a trade-off

between model accuracy and complexity of the optimization: the simpler the model, the easier is solving the optimization and vice versa. Henceforth, while in building simulation models the most accurate model is sought to numerically reproduce the behavior of the process as faithfully as possible, prediction models used in MPC are usually very simple, yet representative enough to capture the main dynamical relations.

Summing up, MPC directly embodies technical specifications like abstract model, performance and limits into the control algorithm and, in particular, no a-posteriori patches are required to take into account limitations on system's variables. In this respect, MPC is a systematic design flow, being independent from the chosen model and performance/constraint specifications.

All the MPC algorithms possessing common elements and different options can be chosen for each one of these elements giving rise to different algorithms:

- Prediction Model
- Objective Function
- Obtaining the Control Law.

It is possible to distinguish different MPC types. For example, in the case of a linear model, a quadratic cost function with respect to the input sequence $\mathbf{u}$ and no constraints differ from the one given by the system description. The so called **Linear MPC** solves, at each time step, a Linear Quadratic Regulation(LQR) problem, that has a closed form solution described by a state-feedback input. The addition of linear constraints, that could also be expressed through inequalities, makes the problem a Quadratic programming task that is still efficiently solvable using the linear MPC since the cost function is convex.

In the case of non linear systems the **Adaptive MPC** can be applied. It linearizes the considered model at the current state for each time instant such that, as before, the problem can be traced back to a Quadratic Problem. In a more general scenario in which the cost function is no more convex a **Non Linear MPC** can be used paying attention to the high computational burden required in this case. Other possible MPC approaches, such as the **Explicit MPC** and the **Robust MPC**, try to reduce the computational burden required and to improve robustness to model disturbances. In this work a path tracking algorithm was implemented for mobile robot using Nonlinear Model Predictive Control (NMPC), that is one of the optimal controllers with the help of IPOPT solver

(more details on section 2.1.2) for managing the computational burden. The MPC can provide powerful performance among existing optimal controllers since it can have good prediction capabilities and it makes the constraints regulation easy.

Mathematical background

For numerical solutions to the open-loop optimal control problem in the general setup it is often necessary to parameterize the input in an appropriate way. This is normally done by using a finite number of basis functions, i.e. the input can be approximated as a piece wise constant over the sampling time δ. As will be shown, the calculation of the applied input based on the predicted system behavior allows the inclusion of constraints on states and inputs as well as the optimization of a given cost function. However, since in general the predicted system behavior will differ from the closed-loop one, precautions must be taken to achieve closed-loop stability.

Several formulations and variations of MPC algorithms have been proposed in the literature and have been implemented in various tool. Consider a generic discrete-time model defined by the equations for $k \in \mathbb{R}$:

$$\begin{cases} \mathbf{x}(k+1) = \mathbf{f}(\mathbf{x}(k), \mathbf{u}(k)) \\ \mathbf{y}(k) = \mathbf{g}(\mathbf{x}(k), \mathbf{u}(k)) \\ \mathbf{x}(0) = x_0 \end{cases} \tag{2.1}$$

As always in controls, $\mathbf{x}(\cdot), \mathbf{y}(\cdot), \mathbf{u}(\cdot)$ are respectively the state, the output and the input. x_0 the initial state and k is the present time.

The aforementioned cost function is usually denoted with $\mathcal{J}(\cdot, \cdot, \cdot)$ and it is not unique. In theory there always exists a surrogate cost function that allows MPC to recover the globally optimal trajectory. However, while an optimal surrogate cost function may exist, it would be impractical to use as it would require prior knowledge of the globally optimal controls.

In order to control the system (2.1) the MPC searches for a control sequence $\hat{\mathbf{u}} = [\hat{\mathbf{u}}(k), \ldots, \hat{\mathbf{u}}(k+N-1)]$ that minimizes the quadratic cost function:

$$\mathcal{J}(\mathbf{x}(k), \mathbf{u}(\cdot), k) = \sum_{i=0}^{T-1} (\|\mathbf{x}(k+i)\|^2 + \|\mathbf{u}(k+i)\|^2) + \|\mathbf{x}(k+T)\|^2 \tag{2.2}$$

So the MPC searches $\hat{\mathbf{u}}$ such that:

$$\hat{\mathbf{u}} = \arg\min_{u} \mathcal{J}(\mathbf{x}_k, \mathbf{u}(\cdot), k) \tag{2.3}$$

$$\text{subject to} \quad \begin{cases} \mathbf{x}(k+i+1) = \mathbf{f}(\mathbf{x}(k+i)), \mathbf{u}(k+i) \\ \mathbf{y}(k+i) = \mathbf{g}(\mathbf{x}(k+i), \mathbf{u}(k+i)) \\ \mathbf{x}(k) = \mathbf{x}_k \\ \mathbf{x}(k+i) \in \mathcal{X} \\ \mathbf{y}(k+i) \in \mathcal{Y} \\ \mathbf{x}(k+N) \in \mathcal{X}_{fin} \\ \mathbf{y}(k+N) \in \mathcal{Y}_{fin} \end{cases} \quad \text{for every } i = 0, \ldots, N-1$$

In (2.3) the state, the input, the output, the final state and the final output domains are respectively $\mathcal{X} \in \mathbb{R}^m$, $\mathcal{U} \in \mathbb{R}^p$, $\mathcal{X}_{fin} \subseteq \mathcal{X}$, $\mathcal{Y}_{fin} \subseteq \mathcal{Y}$ where the current state $\mathbf{x}(k) = \mathbf{x}_k$ defines the initial condition of the optimization problem. Furthermore, all these quantities are defined according some constraints. N is the number of sampling instants in the so called *Prediction Horizon* (see Figure 2.2) that represents the temporal window that starts at k (in the Figure 2.2 it corresponds to t since it refers to a continuous time system while the system under study is discrete time. This is explained more precisely in section 2.1.1) and is considered in the current optimization process; the higher is N the larger is the temporal window.

According to the MPC approach, the control sequence $\hat{\mathbf{u}}(\cdot)$ described above is used as the current input for the controlled system as shown in Figure 2.1.

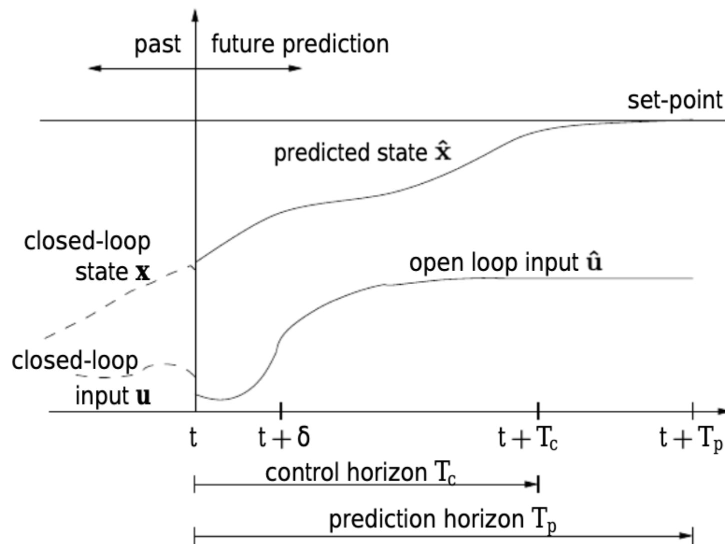


Figure 2.2: Basic principle of model predictive control as introduced in the previous chapter

2.1.1 Non Linear MPC

In mathematics, a nonlinear algorithm is the process of solving an optimization problem where some of the constraints or the objective function are nonlinear. An optimization problem is the problem of finding the best solution from all feasible solutions. The standard form of an optimization problem consists in maximizing or minimizing some function relative to some set and representing a range of choices available in a certain situation. The function allows comparison of the different choices for determining which might be best.

An optimization problem is the subfield of mathematical optimization that deals with problems that are not linear.

In general, the model predictive control problem is formulated for solving on-line a finite horizon open-loop optimal control problem subject to system dynamics and constraints involving states and controls (see Figure 2.2). Based on measurements obtained at time t, the MPC controller predicts the future dynamic behavior of the system over a prediction horizon T_P and determines, over a control horizon $T_C \leq T_P$, the input such that a predetermined open-loop performance objective functional is optimized. If there were no disturbances and if the optimization problem could be solved for an infinite time horizon, then one could apply the input function found at time $t = 0$ to the system for all times $t \geq 0$.

However this is not possible: in general the true system behavior is different from the predicted behavior because of the disturbances and the model-plant

mismatch. In order to incorporate some feedback mechanisms, the obtained open-loop manipulated input function will be implemented only until the next measurement becomes available. The time difference between measurements can vary, however often it is assumed to be fixed (the measurement will take place every δ). Using the new measurement at time $t + \delta$, the procedure of prediction and optimization is repeated to find a new input function with the control and prediction horizons moving forward.

In many situations the process will be operating in the neighborhood of a steady state and a linear representation will be adequate. There are some very important situations where this does not occur. Sometimes non-linearities are so rigorous and so crucial to the closed loop stability that a linear model is not sufficient. In other scenarios there are some processes that experience continuous transitions and they spend a great deal of time away from a steady state operating region or even processes which are never in steady state operation.

For these processes a linear control law will not be very effective so nonlinear controllers will be essential for improving the performance.

The extension of MPC general formulation to the nonlinear process is straight forward the same, at least conceptually. However some problems arise which need to be taken into account, such as:

- The availability of nonlinear models due to the lack of identification techniques for nonlinear processes
- The computational complexities for solving the model predictive control of nonlinear processes
- The lack of stability and robustness results for the case of nonlinear systems.

Developing nonlinear empirical models may be very difficult and there is no model form that is clearly suitable to represent general nonlinear processes.

The bigger mathematical obstacle to a complete theory of nonlinear processes is the lack of a superposition principle for nonlinear systems. This implies difficulty for the determination of models from process input/output data. The amount of plant tests required to identify nonlinear plant is higher than the linear case.

In the ideal case of linearity of the plant only a step test has to be performed in order to know the step response of the plant and due to the superposition principle, the response to different size step can be obtained by multiplying the response to the step test by the ratio of both step sizes.

This is not the case of nonlinear processes, where tests with many different size steps must be performed in order to get the step response of the nonlinear plant. If the process is multi-variable, the difference in the number of tests required is even higher.

If the deviation from linearity is not too large, some approximations can be made assuming linearity in the neighborhood of a specific operating point.

Mathematical background

To use the Model Predictive Control a dynamic model would be needed that also takes into account the force applied to the vehicle body. Here we will simplify it to a kinematic model (the Ackermann Model is described in detail in section 2.2).

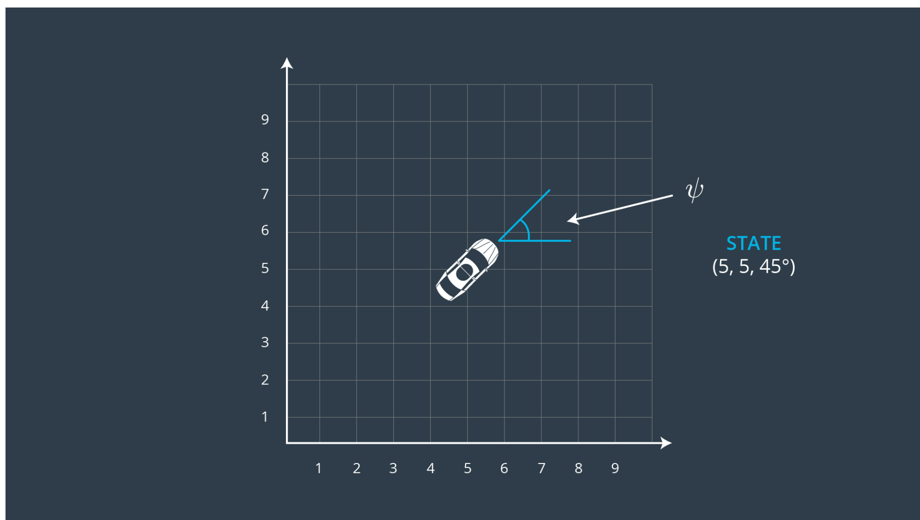


Figure 2.3: Kinematic Model

The following equation can be derived for the state of the car:

$$\begin{cases} x_{K+1} = x_K + v_K \cos \psi_K dt \\ y_{K+1} = y_K + v_K \sin \psi_K dt \\ v_{K+1} = v_K + a_K dt \\ \psi_{K+1} = \psi_K + \frac{v_K}{L_f} \delta_K dt \end{cases} \tag{2.4}$$

Where (x, y, ψ, v) are the position and the orientation (see Figure 2.3) and the vector input is (δ, a), where δ represents the steering wheel angle and a represents

the acceleration. Finally, L_f is the distance from the Center of Gravity to the tip of the car.

Furthermore, the Cross Tracking Error (CTE) is the distance between the reference point on the vehicle and the closest point on the desired path (see Figure 2.4). It is the principal measure of how close the vehicle is to the desired position along the path. The evolution over the time of the CTE can be calculated by extracting the lateral component of the forward velocity. It can be notice that as the velocity increases the CTE changes more quickly, this meaning that smaller steering angles are needed to correct same size errors.

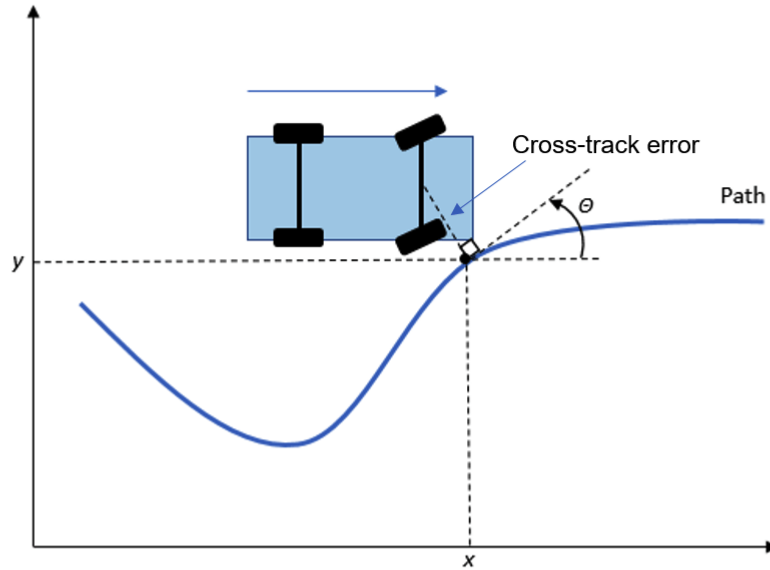


Figure 2.4: CTE: the distance between the reference point on the vehicle and the closest point on the desired path

It is introduced also the Heading Error $e\psi$ that is equal to the difference between path and vehicle headings at the reference point along the path. It is a principal measure of how well the vehicle is aligned with and moving in the direction of the desired path (see Figure 2.5 where the orange straight line is the reference path and the dotted line from the vehicle reference point to the path reference point is perpendicular to the path).

Both heading error and CTE must converge to zero for the vehicle to be properly tracking the desired path.

These quantities come from:

$$\begin{cases} CTE_{K+1} = CTE_K + v_K \sin(e\psi_K)dt \\ e\psi_{K+1} = e\psi_K + \frac{v_K}{L_f}\delta_K dt \end{cases} \tag{2.5}$$

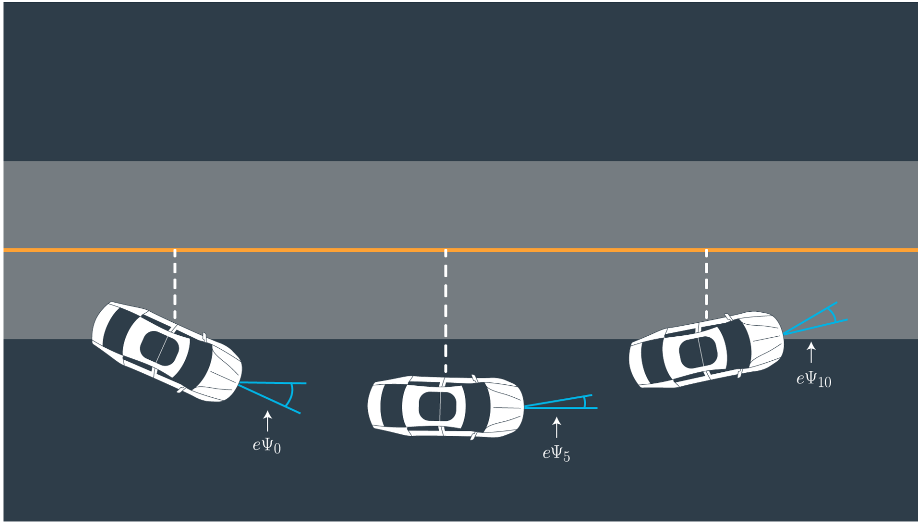


Figure 2.5

Control system design rarely requires only reaching a given target state and in many cases the process of reaching that target state is also given to the design specifications.

In this work the CTE is reduced to avoid, for example, sharp turns or sudden accelerations in order to reproduce the real behavior as if there was a human driver.

The problem of designing a control system that minimizes the cost function by evaluating the degree of achievement of the "humanity" behavior is called the **optimal control problem**. To solve this problem it is necessary to set the cost function (as described in the previous section) and in this specific case the following can be considered as necessary cost functions:

- CTE
- Angle difference between the CTE reference line and the angle between the current car:

$$\mathcal{J} = \sum_{K=1}^{N} (CTE_K - CTE_{ref})^2 + (e\psi_K - e\psi_{ref})^2 \tag{2.6}$$

As can be seen in the Figure 2.6, only the second cost function is considered. When the target trajectory is reached the cost becomes 0, so the vehicle does not need to move and stops the ride.

In order to avoid this problem it is necessary to determine the target velocity and include it in the cost function.

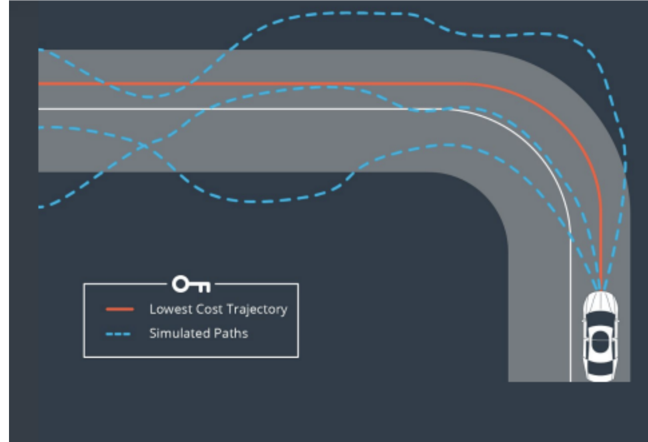


Figure 2.6: The chosen trajectory is the one that minimize the cost

Furthermore, when trying to reproduce the trajectory driven by humans, sudden changes in the steering wheel and acceleration amount are not desired. Therefore, this should also be included in the cost function.

Bringing together the above considerations, the following cost function has been formulated:

$$\mathcal{J} = \int_0^T (a \times CTE_K^2 + b \times e\psi_K^2 + c \times (v_K - v_{ref})^2 + d \times (\delta_k - \delta_{K-1})^2 + \\ + e \times \delta_K^2 + f \times (a_K - a_{K-1})^2 + g \times a_K^2)dt$$

$$\text{subject to} \quad \begin{cases} x_{K+1} = x_K + v_K \cos\psi_K dt \\ y_{K+1} = y_K + v_K \sin\psi_K dt \\ v_{K+1} = v_K + a_K dt \\ \psi_{K+1} = \psi_K + \frac{v_K}{L_f}\delta_K dt \\ CTE_{K+1} = CTE_K + v_K \sin(e\psi_K)dt \\ e\psi_{K+1} = e\psi_K + \frac{v_K}{L_f}\delta_K dt \end{cases} \tag{2.7}$$

Here v_{ref} is the target speed, a, b, c, d, e, f, g represent weights and can be set freely. For example, if you put a large value in the weight of CTE, you can get an operation input that reduces CTE quickly. Furthermore, since each cost unit is different, the weight must be determined carefully.

In order to solve this optimal control problem, it is necessary to solve a

partial differential equation called the Hamilton-Jacobi equation. This is omitted because the theory is complicated.

IPOPT Solver (Interior Point OPTimizer, a brief description can be found in section 2.1.2) will help to reduce the computational burden, in fact it implements an interior-point line-search filter method assuming that the Jacobian matrix of constraint function is sparse (in numerical analysis and scientific computing, a sparse matrix is a matrix in which most of the elements are zero). The optimal control problem is transcribed to a Nonlinear Programming Problem (NLP) then the NLP is solved using the solver IPOPT.

2.1.2 IPOPT Solver

The rigorous definition of IPOPT Solver is set out below:

> ≪*A software package for large-scale nonlinear optimization. It is designed to find local solutions of mathematical optimization problems*≫

IPOPT has been developed by Andreas Wächter and Carl Laird and it can be used on Linux, Unix, Mac OS X and Microsoft Windows operating systems. IPOPT is written in C++ and it is released as open source code under the Eclipse Public License. IPOPT can be used as a library that can be linked to C++, C or Fortran code, as well as a solver executable for the AMPL modelling environment.

IPOPT implements an interior-point line-search filter method and this approach makes IPOPT particularly suitable for large problems with up-to millions of variables and constraints, assuming that the Jacobian matrix of constraint function is sparse but also small and dense problems can be solved efficiently.

It can be used to solve general nonlinear programming problems of the form:

$$\arg\min_{x \in \mathbb{R}^n} \mathbf{f}(x) \quad \text{s.t.} \quad \mathbf{g}^L \leq \mathbf{g}(x) \leq \mathbf{g}^U \quad \text{and} \quad x^L \leq x \leq x^U$$

Where $x \in \mathbb{R}^n$ are the optimization variables, $\mathbf{f}(\cdot) : \mathbb{R}^n \to \mathbb{R}$ is the objective function and $\mathbf{g}(\cdot) : \mathbb{R}^n \to \mathbb{R}^m$ is the function that describes general nonlinear constraints. $\mathbf{f}(x)$ and $\mathbf{g}(x)$ can be linear or nonlinar and convex or non-convex but should be twice continuously differentiable.

In this thesis the IPOPT Solver has been used to optimize the Nonlinear Model Predictive Control in order to reduce the computational burden of the algorithm.

2.2 Ackermann Model Description

The ackermann steering is used in car-like vehicles. The basic idea consists of rotating the inner wheel slightly sharper than the outer to reduce tire slippage.

The Ackermann level in a vehicle steering geometry is represented as a percentage, where 100% Ackermann means the difference in steer angle between the inside and outside tyre matches the geometric low-speed turn centre.

When a vehicle is turning, the inner front wheel needs to turn at a different angle to the outer because they are turning on different radii. The Ackermann steering mechanism is a geometric arrangement from the bicycle model where linkages in the steering of a vehicle are designed to turn the inner and outer wheels at the appropriate angles. This model is fully parameterized, allowing customization and component sizing.

Consider a low-speed cornering manoeuvre, where all tyres are in pure rolling condition, and there is no vehicle sliding. As the vehicle travels along a curved path, all the tyres follow unique trajectories around a shared turn centre, as it can be seen in Figure 2.7.

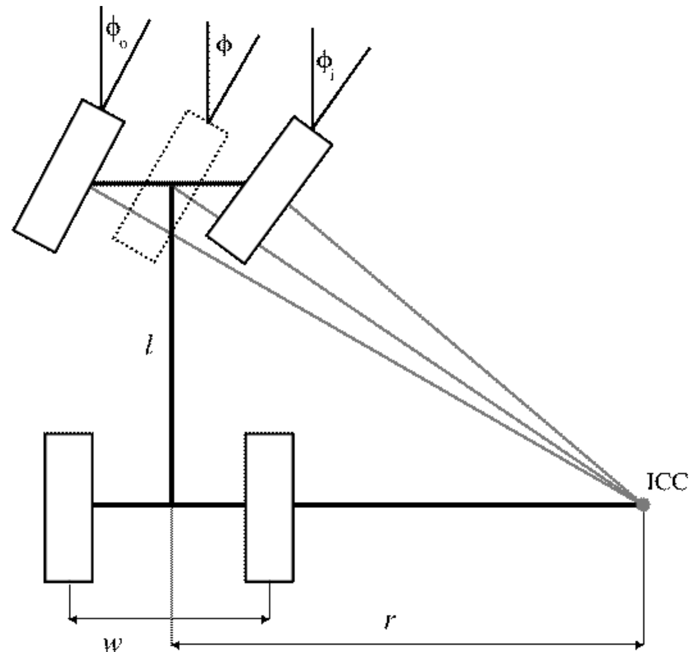


Figure 2.7: Description of Ackermann Steering

Different curvature radii means that, to avoid sliding, the steering geometry must steer the inside front tyre at a larger angle than the outside front wheel. Ackermann Steering refers to the geometric configuration that allows both front wheels to be steered at the appropriate angle to avoid tyre sliding.

Kinematic Model

To understand the next equations, some quantities need to be defined:

- w: The track width or the lateral wheel separation
- l: The wheel base or the longitudinal wheel separation
- ϕ_i: The relative steering angle of the inner wheel
- ϕ_o: The relative steering angle of the outer wheel
- r: The distance between instantaneous centre of curvature and the centre of the vehicle.

The Ackermann steering equation (which is not the Ackermann model) can then be derived quite easily with trigonometry by considering the three triangles (again see Figure 2.7) formed by the vertical side l and the side r plus or minus $\frac{w}{2}$. So we get:

$$\phi_o = \tan^{-1}\Big(\frac{l}{r+\frac{w}{2}}\Big) \quad \phi = \tan^{-1}\Big(\frac{l}{r}\Big) \quad \phi_i = \tan^{-1}\Big(\frac{l}{r-\frac{w}{2}}\Big)$$

The forward kinematic for a car-like vehicle, which is the prediction of the future state given the configuration, can be formulated from the bicycle model defined as follows (see also Figure 2.8):

$$\begin{cases} \dot{x} = v\cos(\psi+\beta) \\ \dot{y} = v\sin(\psi+\beta) \\ \dot{\psi} = \frac{v}{l_r}\sin(\beta) \\ \dot{v} = a \\ \beta = \tan^{-1}\Big(\frac{l_r}{l_f+l_r}\tan(\delta_f)\Big) \end{cases} \tag{2.8}$$

where (x, y) are the coordinates of the center of mass, ψ is the inertial heading, l_f and l_r represent the distances from the center of the mass of the vehicle respectively to the front and to the rear axes, β is the angle of the current velocity of the center of mass with respect to the longitudinal axes of the car, a is the acceleration of the center of mass in the same direction as the velocity. Finally, the control inputs are the acceleration a and the front and rear steering angles δ_f. Since in this model (and either in the most vehicles) the rear wheels can not steer, it is assumed $\delta_r = 0$ for simplicity.

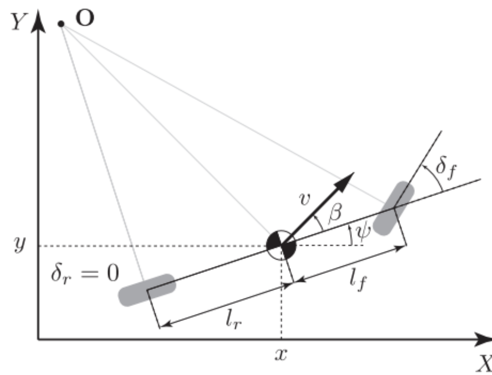


Figure 2.8: Kinematic Bicycle Model

Compared to higher fidelity vehicle models, the system identification on the kinematic bicycle model is easier because there are only two parameters to identify: l_f and l_r.

This makes it simpler to port the same controller or path planner to other vehicles with differently sized wheelbases.

The kinematic model focuses on the geometrical properties of the vehicle and, by definition, does not take into account the dynamics of the vehicle (tire forces, yaw rate, ...). It assumes that the car adheres perfectly to the road (no slip) in fact this model gives good results when the car runs at low speed.

The kinematic model (2.8) can be further simplified by assuming constant the speed v and taking into account that the Ackermann steering has the rotational center (or the origin) not in the middle of the car but between the back wheels.

Assuming that $l = l_r + l_f$ denotes the wheelbase of the vehicle the simplified model is:

$$\begin{cases} \dot{x} = v \cos\theta \\ \dot{y} = v \sin\theta \\ \dot{\theta} = \frac{v}{l} \tan\phi \end{cases}$$

The difference in front-wheel steer angle as a function of the input steer angle is known as Dynamic Toe. If the vehicle dimensions are known, it is possible to construct a curve of the desired toe change for the full range of expected turn radii, such as the example in Figure 2.10.

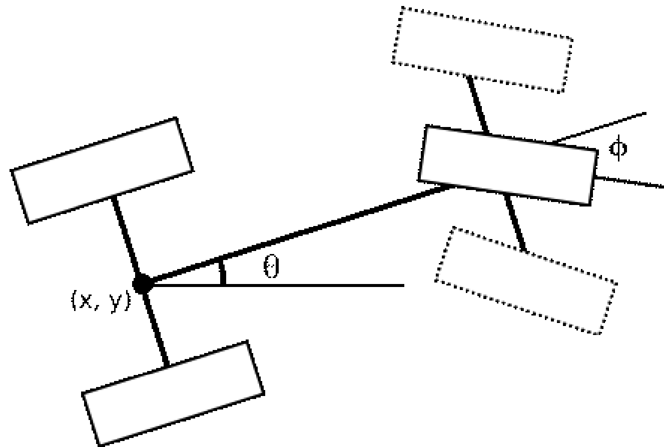


Figure 2.9: Ackermann Steering configuration for a simplified model

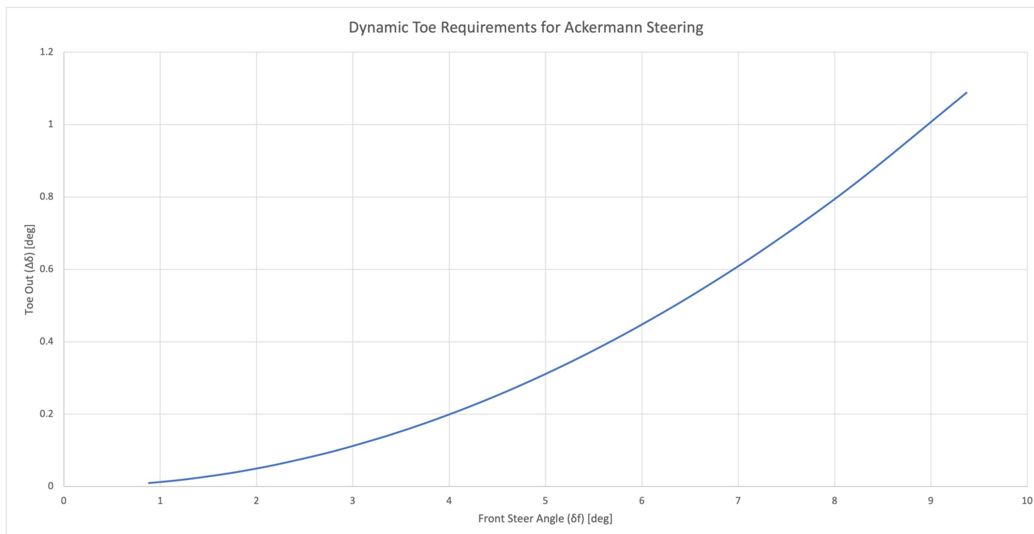


Figure 2.10: Ackermann Steering configuration for a sample vehicle

The tighter the desired vehicle turn radius, the larger the difference in steer angles required. The picture gets much more complicated once the vehicle is at high speed.

Dynamic Model

Even if in this thesis only the kinematic model will be used, for completeness in this section the bicycle dynamic model is also briefly described.

The inertial position coordinates and heading angle in the dynamic bicycle model are defined in the same manner as those in the kinematic bicycle model. The differential equations in this case are given by:

$$\begin{cases} \ddot{x} = \dot{\psi}\dot{y} + a_x \\ \ddot{y} = -\dot{\psi}\dot{x} + \frac{2}{m}(F_{c,f}\cos\delta_f + F_{c,r}) \\ \ddot{\psi} = \frac{2}{I_z}(l_f F_{c,f} - l_r F_{c,r}) \\ \dot{X} = \dot{x}\cos\psi - \dot{y}\sin\psi \\ \dot{Y} = \dot{x}\sin\psi - \dot{y}\cos\psi \end{cases} \tag{2.9}$$

where $\dot{x}$ and $\dot{y}$ denote the longitudinal and lateral speeds in the body frame respectively, $\dot{\psi}$ denotes the yaw rate, m and I_z denote the vehicle's mass and yaw inertia, respectively. $F_{c,f}$ and $F_{c,r}$ denote the lateral tire forces at the front and rear wheels, respectively, in a coordinate frames aligned with the wheels.

For the linear tire model, $F_{c,i}$ is defined as:

$$F_{c,i} = -C_{\alpha_i}\alpha_i \tag{2.10}$$

where $i \in (f, r)$, α_i is the tire slip angle and C_{α_i} is the tire cornering stiffness.

Chapter 3

Hardware Architecture

The implementation of the Non linear MPC described in the previous chapter was made on an RC car suitably modified to mount various controllers: a computer board, a LiDAR and a low-definition camera (which will not be used in this work but has been included for completeness and for any future work).
The shell has been removed and in its place a 3D-printed base has been insert in which an Arduino Uno has been inserted.
Since the LiDAR cannot have obstacles in his field of work, a riser has also been 3D-printed above which the LiDAR has been placed in such a way as to be in the highest part of the structure. The pillars of the riser were used to house a Raspberry Pi3.
The control system of the autonomous car is subdivided into master and slave control unit. The master control unit is the Raspberry Pi 3 Model B microprocessor and the slave control unit is based on the Arduino Uno development platform. The two control units are separated and the tasks are distributed and allocated to them due to the fact that it was possible to achieve higher responsiveness and better performance of the system.

3.1 Arduino Uno

Arduino Uno is a basic microcontroller board based on the ATmega328P processor. It has 14 digital input/output pins (6 of which can be used as Pulse Width Modulation outputs), 6 analog inputs, a 16 MHz ceramic resonator (CSTCE16M0V53-R0), a USB connection, a power jack, an ICSP header and a reset button (Figure 3.1 shows the architecture of Arduino Uno). It contains everything needed to support the microcontroller; simply connect it to a computer with a USB cable or power it with an AC-to-DC adapter or battery to get started. You can tinker with your Uno without worrying too much about doing something wrong, worst case scenario you can replace the chip for a few euros and start over again.

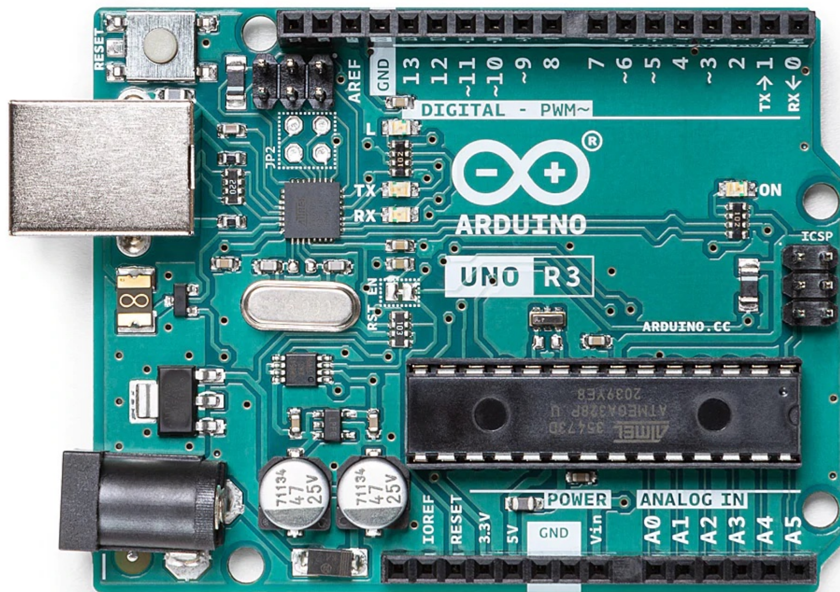
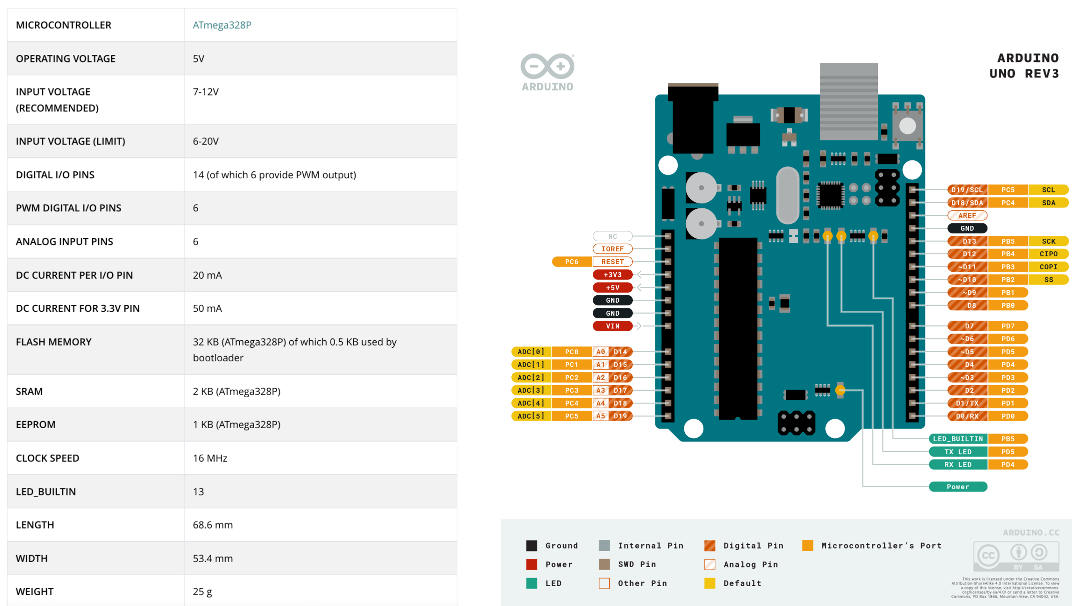


Figure 3.1: Upper view Arduino Uno

"Uno" means one in Italian and was chosen to mark the release of Arduino Software (IDE) 1.0. The Uno board and version 1.0 of Arduino Software (IDE) were the reference versions of Arduino, now evolved to newer releases. The Uno board is the first in a series of USB Arduino boards and the reference model for the Arduino platform; for an extensive list of current, past or outdated boards see the Arduino index of boards in `Arduino.cc` website. In this work Arduino Uno was chosen for its simplicity of use and for its low cost. It was chosen instead of other microcontrollers of the same brand because it has a sufficient number of input pins.



MICROCONTROLLER	ATmega328P
OPERATING VOLTAGE	5V
INPUT VOLTAGE (RECOMMENDED)	7-12V
INPUT VOLTAGE (LIMIT)	6-20V
DIGITAL I/O PINS	14 (of which 6 provide PWM output)
PWM DIGITAL I/O PINS	6
ANALOG INPUT PINS	6
DC CURRENT PER I/O PIN	20 mA
DC CURRENT FOR 3.3V PIN	50 mA
FLASH MEMORY	32 KB (ATmega328P) of which 0.5 KB used by bootloader
SRAM	2 KB (ATmega328P)
EEPROM	1 KB (ATmega328P)
CLOCK SPEED	16 MHz
LED_BUILTIN	13
LENGTH	68.6 mm
WIDTH	53.4 mm
WEIGHT	25 g

Figure 3.2: Full Arduino Uno technical specifications and Pinout diagram

In this work the Arduino Uno deals with controlling a servomotor (Figure 3.4), which will take care of controlling the steering, and a $12V$ DC motor (Figure 3.3 on the left) that takes care of rear-wheel drive. Since a precise control over the *rpm* of the motor is required to control the velocity of the car and also the ability to reverse the rotation of the motor (both to brake and to retreat), a driver was also added a (precisely the chip L293D, see Figure 3.3 on the right) which allows the previously described control on the DC motor. This is a very useful chip and can also be used to control two motors independently. This is the reason for which most of the pins on the right hand side of the chip are not used since they are used for controlling a second motor. This is not the unique way to control the motor but it is the easiest.

3.2 Raspberry Pi 3

Raspberry Pi is a series of small single-board computers developed in the United Kingdom by the Raspberry Pi Foundation in association with Broadcom. The Raspberry Pi project originally leaned towards the promotion of teaching basic computer science in schools and in developing countries. The original model became very popular selling outside its target market for uses such as robotics. It is widely used in many areas because of its low cost, modularity, and open design. It is typically used by computer and electronic hobbyists, due to its adoption of

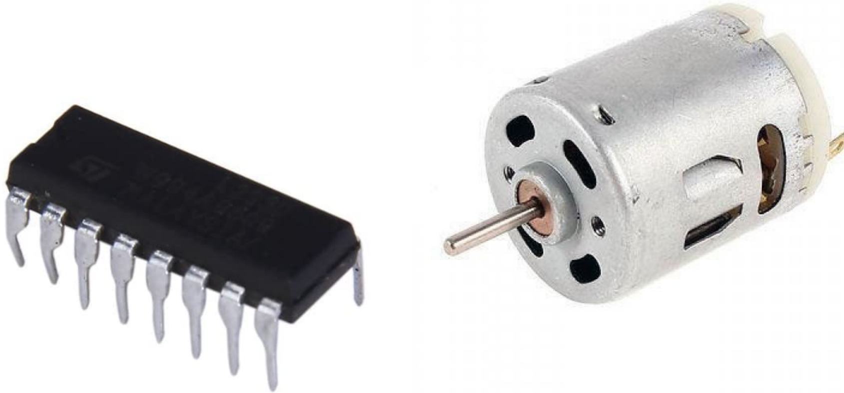


Figure 3.3: Step motor driver on the left and DC Motor on the right

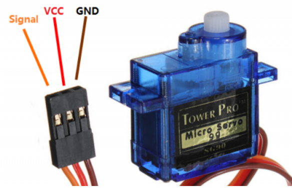


Figure 3.4: Servomotor

HDMI and USB devices.

After the release of the second board type, the Raspberry Pi Foundation set up a new entity, named Raspberry Pi Trading and the foundation was rededicated as an educational charity for promoting the teaching of basic computer science in schools and developing countries. Most Raspberrys are made in a Sony factory in Pencoed, Wales, while others are made in China and Japan.

Figure 3.5: Raspberry Pi 3 Model B - A single-board computer with wireless LAN and Bluetooth connectivity

The Raspberry Pi hardware has evolved through several versions that feature variations in the type of the central processing unit, amount of memory capacity,

networking support and peripheral-device support.

Raspberry Pi removes the high entry cost to computing for people across all demographics: while children can benefit from a computing education that previously was not open to them, many adults have also historically been priced out of using computers for enterprise, entertainment and creativity. Raspberry Pi eliminates those barriers.

In this work was used a Raspberry Pi 3 Model B (Figure 3.5) that has the following specifics:

- Quad Core 1.2GHz Broadcom BCM2837 64bit CPU
- 1GB RAM
- BCM43438 wireless LAN and Bluetooth Low Energy (BLE) on board
- 100 Base Ethernet
- 40-pin extended GPIO
- 4 USB 2 ports
- 4 Pole stereo output and composite video port
- Full size HDMI
- CSI camera port for connecting a Raspberry Pi camera
- DSI display port for connecting a Raspberry Pi touchscreen display
- Micro SD port for loading your operating system and storing data
- Micro USB power source up to 2.5A

To the Raspberry was connected a Raspberry Pi Camera (Figure 3.6) that is not used but mounted on it only for completeness and eventually future works.

A laser scanner LiDAR(Light Detection And Ranging) was also connected to the board.

The RPLiDAR A3 from SLAMTEC is a 360° Laser range scanner for indoor and outdoor application (Figure 3.7). Thanks to a sample rate of 25600 $times/s$ the robot can build map quickly and accurately. RPLiDAR A3 also improves the internal optical design and the optimization algorithm with respect to the

Figure 3.6: Raspberry Pi Camera

previous versions. With a ranging radius of 25 meters, is much better than most competitors like the G4 from YDLiDAR or the M10 from Smartfly.

In enhanced mode, RPLiDAR A3 works with the maximum ranging radius and sampling rate to realize an optimistic mapping performance in indoor environments. While in outdoor mode, it works with a more reliable resistance to daylight interference. Finally it keeps its performance stable when detecting white and black objects.

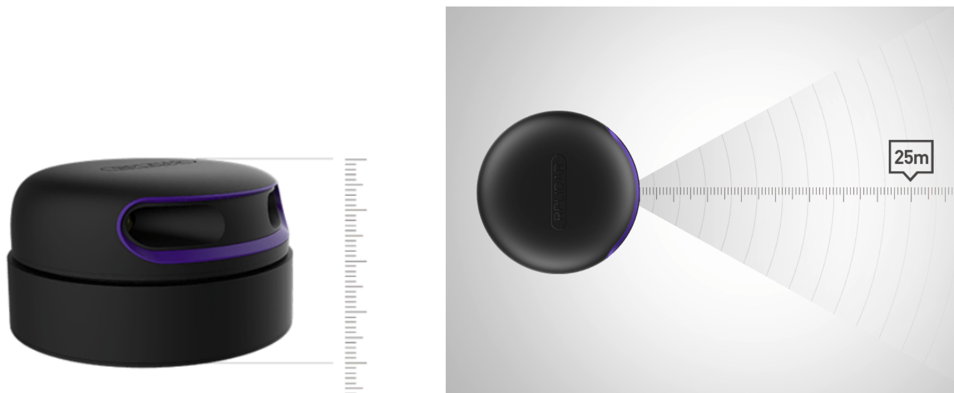


Figure 3.7: RPLIDAR A3 from SLAMTEC side and up view

However, the main purpose of the Raspberry is to be the brain of the implementation part of the thesis. In fact on the Raspberry computer was installed ROS (Robot Operative system) that permits the communication between all the hardware installed on the RC car and it's able to implement the Model Predictive control described in Chapter 2, to map the environment (in the Mapping mode) and to follow the desired path of the car.

3.3 RC Car

The whole process of assembling and setting the Radio Controlled Car (RC Car) in its autonomous version is described in this section:

3.3.1 Choose the RC car

There are various types of RC cars on the market, the choice fell on an economical model, easy to disassemble and with enough space to mount the hardware necessary for the purpose. The shell was suddenly removed since is not needed for this work (Figure 3.8).

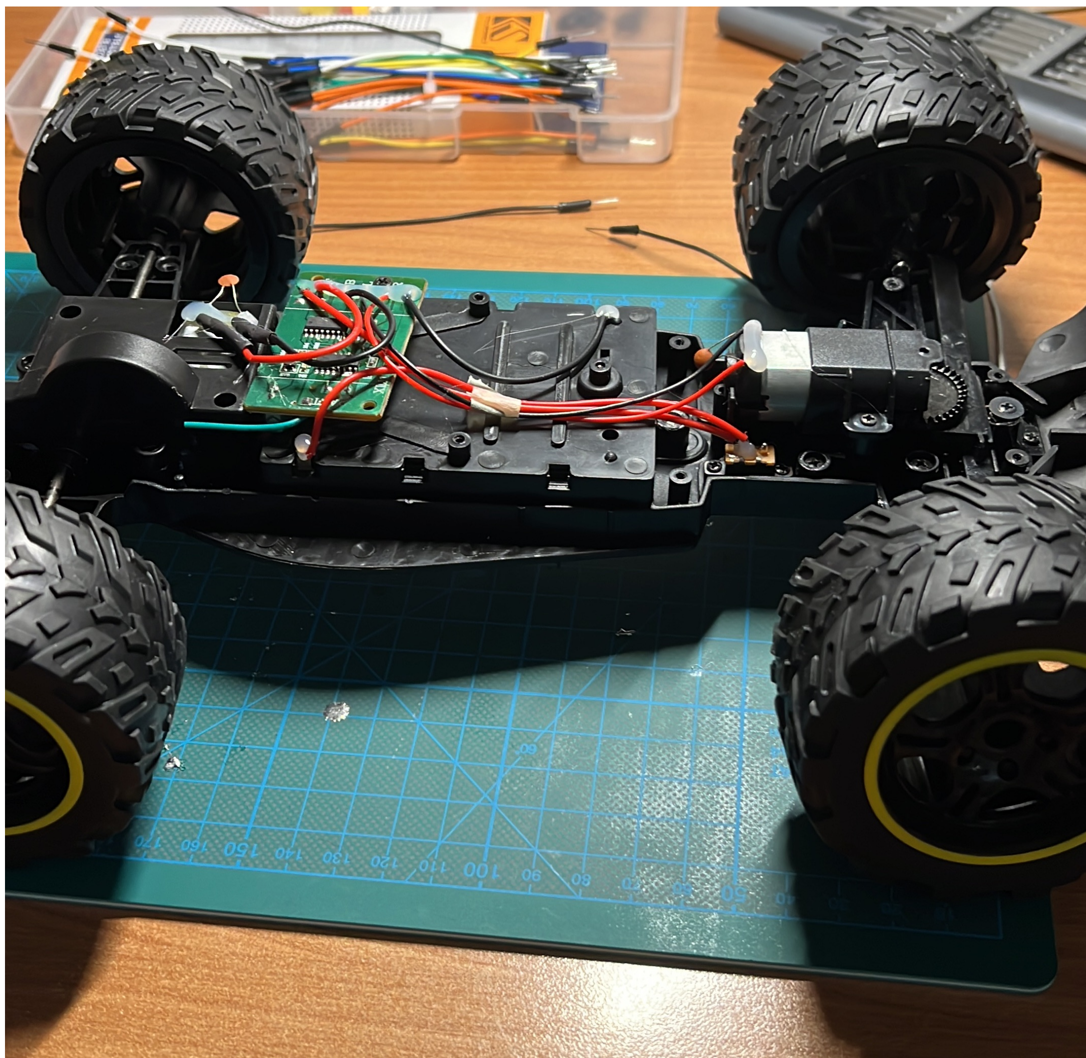


Figure 3.8: The original state of the RC Car

The main difficulty in choosing the machine is that all the economical RC cars available on the market had the steering controlled by a small $12V$ DC motor

and therefore the steering signal was like ON/OFF which did not allow precise steering control. To overcome this problem, the steering mechanism had to be modified: the original steer engine was removed, a hole was made in the vehicle frame and a piece was 3D printed (Figure 3.9) which, once connected to the steering bar, allowed the installation of a servomotor that will be controlled by Arduino.

Finally, the RC module was removed (which allowed to control the machine with the radio control) as the purpose is be able to send commands to the machine via a laptop connected to the same WiFi network as the Raspberry.

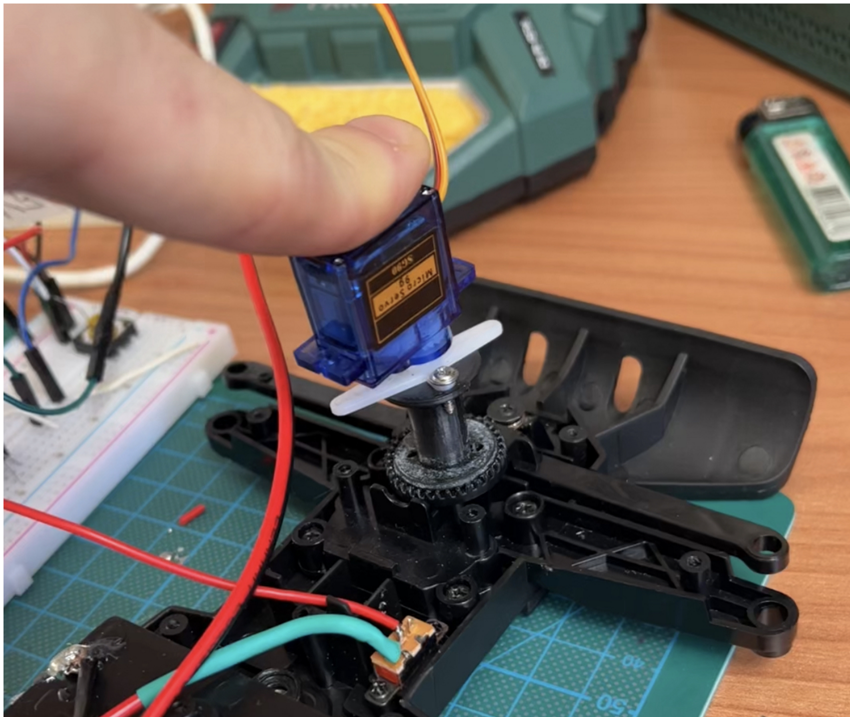


Figure 3.9: The new mechanism of the steering

3.3.2 3D Printed Parts

To be able to assemble all the components in a stable way, pieces were 3D printed. In particular the base in which the Arduino will be housed has been printed, the base keep also the switch and some electrical connections.

In the base printing process some bolts have been incorporated and melted into the print that will be used to fix the Arduino. Also a structure capable of carrying both the Raspberry and the RPLiDAR has been printed in 3 parts since printing a single piece was a bit more complicated. The latter was made high

enough to allow the lidar not to have obstacles due to the components during the scans.

Finally, a small support was also printed that can keep the powerbank in the bottom part of the base.

Figure 3.10: The .stl file of the 3D printed base

3.3.3 Assembling

The assembly of the various components was quite simple. The Arduino takes care of the steering and acceleration while the Raspberry calculates the trajectory and sends steering and velocity commands to the Arduino. To have a more accurate control of the acceleration or to possibly reverse the direction of the engine, a L293D chip was mounted. L293D is a very useful chip, it can actually control two motors independently but for this work it is just used half the chip. Most of the pins on the right hand side are for controlling a second motor and are unplugged.

The L293D are quadruple high-current half-H drivers, are designed to provide bidirectional drive currents up to 600mA at voltages from 4.5 V to 36 V. They are designed to drive inductive loads such as relays, solenoids, dc and bipolar stepping motors, as well as other high-current/high-voltage loads in positive-supply applications.

On the top of the structure was fixed the lidar with some M3 screw. Finally, for convenience, a general switch has been inserted that sends the voltage to the motor that takes care of the traction. There was already a switch that performed

this task but it was under the frame, so a little uncomfortable. A $3V$ led was also mounted in a side of the switch to check the status of the batteries.

Instead for the power supply of the controllers, a $5V$ and $5000mAh$ powerbank was inserted on the 3D printed support under the base.

3.3.4 Working Overview

The scope is to make a subscriber ROS node that would run on the Arduino and listen the ROS topic *\cmd_vel* (a description of ROS can be found in Section 4.1. Once the MPC algorithm has determined the direction in which the car has to move, the master controller (Raspberry) will transmit the topic via serial communication to the slave controller (Arduino Uno).

The *\cmd_vel* topic publishes the ROS message *\geometry_msgs\Twist* that have the structure show in Figure 3.11.

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Figure 3.11: The structure of *geometry_msgs/Twist* ROS message

The non-zero components are *message.linear.x* and *message.angular.z* which correspond respectively to the linear velocity command and the angular velocity command. The other components are zeros because the RC Car will move on them 2D plane.

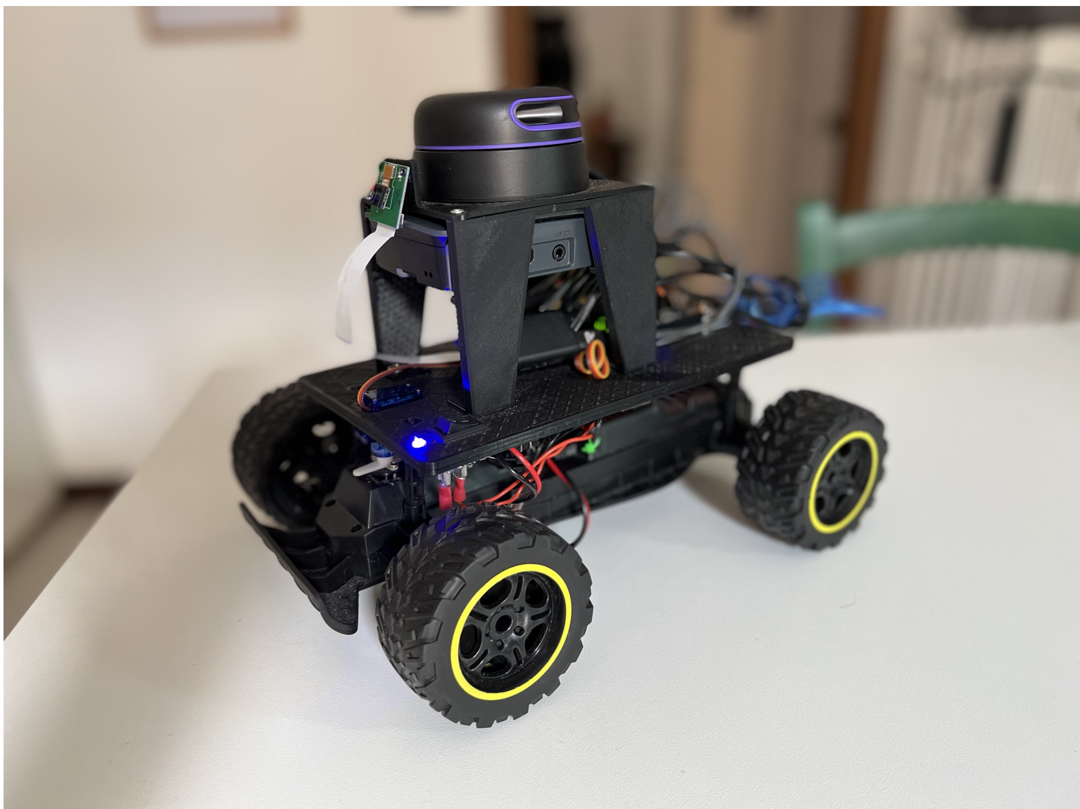


Figure 3.12: The final state of the RC Car

Chapter 4

Tracking

For this work different software has been used, all of these open source and therefore allowing a wide customization of the desired results. For the Arduino has been used the Arduino IDE, in the Raspberry has been installed Ubuntu 20.04 as operative system and ROS to build the robot applications. Finally Gazebo, that is a 3D Robotic open source simulator, is used to visualize the results.
In the simulated part different paths has been designed and implemented, each path has been mapped and then, once the goal point has been chosen, the MPC will be able to calculate an ideal path to follow as best as it can. The simulation gives good results even if unexpected obstacles are suddenly inserted in the path while the simulation has already started.
The visualization of the simulations takes place in parallel both in Gazebo and in RViz, that is a 3D visualization tool for ROS. It displays the desired topics like the maps, the robot model, the desired path.

4.1 Software Tools

This section is dedicated to explain the software used in this thesis. For each software there is a brief description and some hints of history, for more detailed information please refer to the respective sites.

4.1.1 Arduino IDE

A program for Arduino hardware can be written in any programming language with compilers that produce binary machine code for the target processor. *Atmel Studio* provides a development environment for their 8-bit *AVR* and 32-bit ARM Cortex-M based microcontrollers: *AVR* Studio and *Atmel Studio*.

The Arduino integrated development environment (IDE) is a cross-platform application (for Microsoft Windows, macOS, and Linux) that is written in the Java programming language. It originated from the IDE for the languages Processing and Wiring. The source code for the IDE is released under the GNU General Public License.

The Arduino IDE supports the languages C and C++ using special rules of code structuring. It supplies a software library from the Wiring project, which provides many common input and output procedures. User-written code only requires two basic functions, for starting the sketch and the main program loop, that are compiled and linked with a program stub main() into an executable cyclic executive program with the *GNU* toolchain, also included with the IDE distribution. The Arduino IDE employs the program *avrdude* to convert the executable code into a text file in hexadecimal encoding that is loaded into the Arduino board by a loader program in the board's firmware.

The Robot Operating System (ROS) is an open source framework used to build advanced robot applications. Robots can have many operating systems, such as Microsoft Robotics Studio, Webots, Android and so on. ROS includes a set of software libraries and tools that help building robot applications. From drivers to state-of-the-art algorithms, and with powerful developer tools, it is a very useful open source software development kit for robotics applications. It offers a standard software platform to developers across industries that will carry them from research and prototyping all the way through to deployment and production. The emergence of ROS greatly reduces the difficulty of robot development, and promotes the development of robot design. In particular it is an open-source, meta-operating system for robots. It provides the services expected from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing

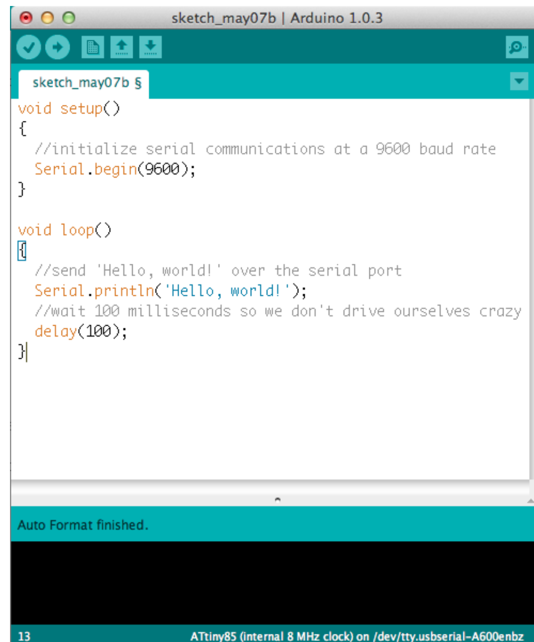


Figure 4.1: An example of Arduino IDE showing a very simple program

ROS

4.1.2 Robot Operative System

between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

Due to the fact that it is open-source, there is the flexibility to decide where and how to use ROS, as well as the freedom to customize it for the needs. Furthermore, ROS is not exclusive, it is not necessary to choose between ROS or some other software stack; ROS easily integrates with the existing software to bring its tools to the problem solution.

The ROS runtime graph (Figure 4.2) is a peer-to-peer network of processes (potentially distributed across machines) that are loosely coupled using the ROS communication infrastructure. It implements several different styles of communication, including synchronous Remote Procedure Call-style communication over services, asynchronous streaming of data over topics, and storage of data on a Parameter Server.

ROS is not a real-time framework, though it is possible to integrate ROS with real-time code. It also has seamless integration with the Orocos Real-time Toolkit.

ROS is designed to be as thin as possible so the code written for ROS can be used with other robot software frameworks. A corollary to this is that it is easy

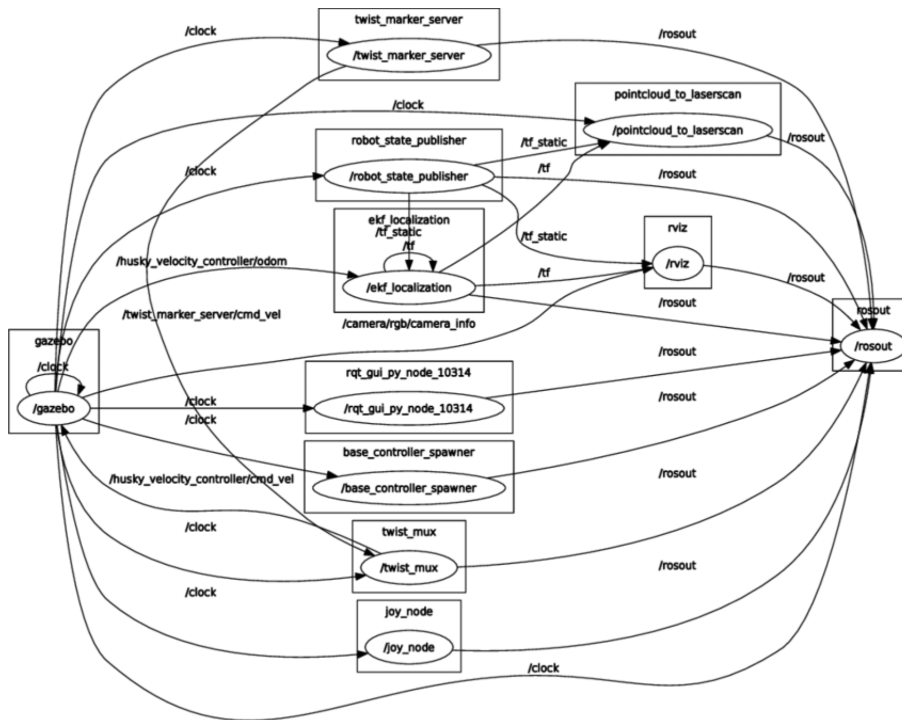


Figure 4.2: An example of a ROS topics publisher/subscriber architecture. It is a graph where a node is a process providing or obtaining data from other processes. Related processes are linked by edges and denote interaction based on message passing.

to integrate with other robot software frameworks: it can already been integrated with *OpenRAVE*, *Orocos*, and *Player*.

The ROS framework is easy to implement in any modern programming language. It is already implemented in Python, C++, and Lisp, and it has experimental libraries in Java and Lua.

ROS is the communication-based programs. To provide a service, many programs are written in the same framework, ranging from the hardware drive of sensors or actuators to all kinds of programs such as sensing, recognition and action. According to the purpose of each processor, it is divided into smaller parts. According to the different platforms, it is called component or node. Data is sent and received between nodes divided into minimum execution units, and the platform has all general information about the data communication. This is consistent with the notion that the smallest unit processes are connected to the Internet of Things (IoT), and programs divided into smallest execution units can be tested in small units to help identify errors and save time.

There are three communication protocols that the nodes use to communicate in ROS:

- **Topics**: These are data buses that are used by nodes to exchange messages among them.
  - *publishers*: Generate data of one topic, for example the nodes that corresponds to the encapsulated drivers of the sensors are publishers that publish messages that contain the sensed values.
  - *subscribers*: Nodes that are subscribed to the topics published by other nodes. All the nodes can be at the same time publishers and subscribers of different topics: multiple publishers and subscribers of the same topic are allowed. The nodes can be subscribed or can publish topics in anonymous form, therefore, the production of information is independent of its use. In general, nodes do not know with whom they are communicating. The unit that performs communication and knows whether nodes are published or subscribed to a topic is the *roscore*.
- **Services**: These allows communication between nodes that have requests and answers. One service is defined by two types of messages, one for the request and the other for the answers. In these situations one node takes up the role of the client: it sends the request to obtain a service and it waits until the server node sends the answer.
- **Actions**: Communication on action is used when a requested goal takes a long time to be completed, therefore progress feedback is necessary. This is very similar to the service where ”goals” and ”results” correspond to ”requests” and ”responses” respectively. In addition, the ”feedback” is added to report feedbacks to the client periodically when intermediate values are needed. The message transmission method is the same as the asynchronous topic. The feedback transmits an asynchronous bidirectional message between the action client which sets the goal of the action and an action server that performs the action and sends the feedback to the action client. Unlike the service, the action is often used to command complex robot tasks such as canceling transmitted goal while the operation is in progress.

To complete the ROS interface, let’s define the concept of **rosbags**: they are data files where the publisher messages are stored. In these files only the information of the publisher nodes that are needed to replay the robot (real or simulated) experiment offline in the computer are saved. This data can be stored

in topics of one node that performs a complex behaviour of the robot or can be only nodes that encapsulate one driver of one sensor and return the sensed data. These files allow to recreate real situations offline and provide the option to verify or improve the created algorithms. Therefore, the rosbags may be designed for two reasons: use the selected node in the environment Gazebo of the computer to finish the debug of some errors or record the experiments on the robot to analyze them further.

To complete the ROS interface, let's define the concept of **Rosbags**: Rosbag is a program that creates, plays, and compresses bags, and has various functions. The needed data from the ROS publisher messages can be recorded. In ROS, bag can be used to record messages and play them back when necessary to reproduce the environment when messages are recorded. For example, when performing a robot experiment using a sensor, sensor values are stored in the message form using the bag. This recorded message can be repeatedly loaded without performing the same test by playing the saved bag file. Record and play functions of rosbag are especially useful when developing an algorithm with frequent program modifications.

In this thesis bags were very useful for example in the mapping, in the scanning phase the data were saved in a rosbag and subsequently used to create the maps.

4.1.3 Gazebo Simulator

Gazebo brings a fresh approach to simulation with a complete toolbox of development libraries and cloud services to make simulation easy. Iterate fast in realistic environments with high fidelity sensors streams. Test control strategies in safety, and take advantage of simulation in continuous integration tests.

Gazebo is a 3D simulator that provides robots, sensors, environment models for 3D simulation required for robot development, and offers realistic simulation with its physics engine. Gazebo is one of the most popular simulators for open source in recent years, and has been selected as the official simulator of the DARPA Robotics Challenge [11] in the US. It is a very popular simulator in the field of robotics because of its high performance even though it is open source.

Figure 4.3: One possible very complex simulation in Gazebo

Moreover, Gazebo is developed and distributed by Open Robotics which is in charge of ROS and its community, so it is very compatible with ROS.

Gazebo has the following characteristics:

- **Dynamics Simulation**: In the early days of development, only *ODE* (Open Dynamics Engine) was supported. However, since version 3.0, various physical engines such as *Bullet*, *Simbody*, and *DART* (Dynamic Animation and Robotics Toolkit) are used to meet the needs of various users

- **3D Graphics**: Gazebo uses *OGRE* (Open-source Graphics Rendering Engines), which is often used in games, not only the robot model but also the light, shadow and texture can be realistically drawn on the screen (as show in figure 4.3)

- **Sensors and Noise Simulation**: Laser range finder (LRF), 2D/3D camera, depth camera, contact sensor, force-torque sensor and much more are supported and noise can be applied to the sensor data similar to the actual environment

- **Plug-ins**: APIs are provided to enable users to create robots, sensors, environment control as a plug-in

- **Robot Model**: *PR2*, *Pioneer2 DX*, *iRobot Create*, and *TurtleBot* are already supported in the form of SDF, a Gazebo model file, and users can add their own robots with an SDF file

- **TCP/IP Data Transmission**: The simulation can be run on a remote server and *Google's Protobufs*, a socket-based message passing is used
- **Cloud Simulation**: Gazebo provides cloud simulation CloudSim environment for use in cloud environments such as *Amazon*, *Softlayer* and *OpenStack*
- **Command Line Tool**: Both GUI and CUI tools are supported to verify and control the simulation status.

4.2 Worlds Creation

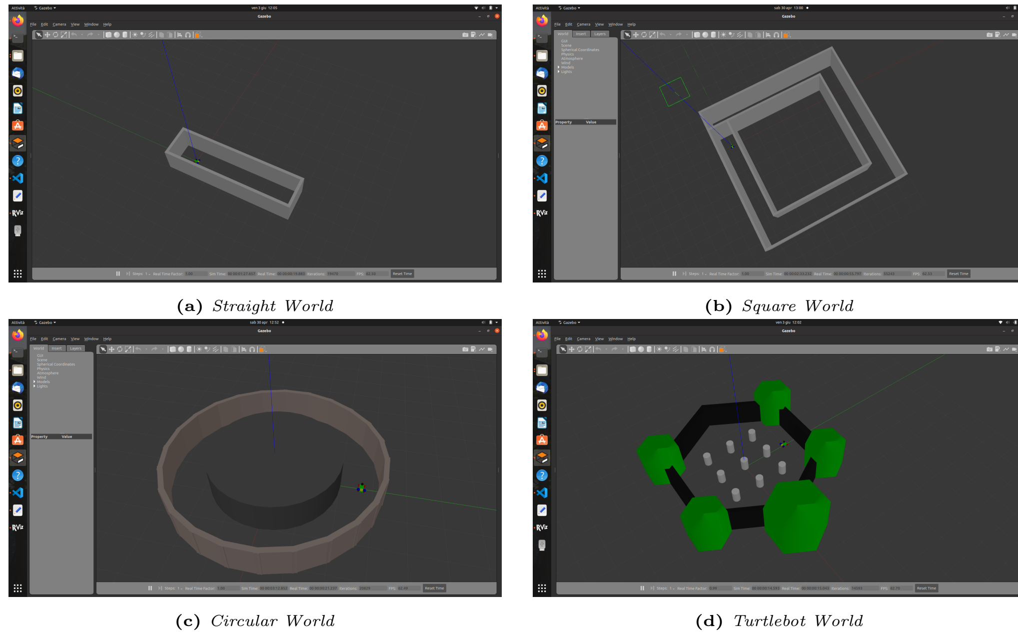
As mentioned in the introduction of this chapter, many worlds has been created with Gazebo simulator. Gazebo offers the possibility to build simple worlds by yourself but, for more complex worlds, it becomes complicated to use the integrated *Building Editor*. An alternative to build more complex worlds can be to use Sketchup and Blander (which are both 3D graphic modeling software) and then insert the SDF file into Gazebo.

4.2.1 World Description

Some of the created worlds are the following:

- **Straight World**: an elementary world, it simply denotes a straight path. The width of the path has been set at 1.75 meters
- **Square World**: a square-shaped world, the critical points are the corners
- **Circular World** : a circular world, pretty simple path
- **Turtlebot World**: an already created world taken from the turtlebot repository
- **Custom World**: a more complex world created by me that permits to explore the power of the algorithm.
- **Small House**: a really complex and realistic world found on internet at [12] used to better test the algorithm

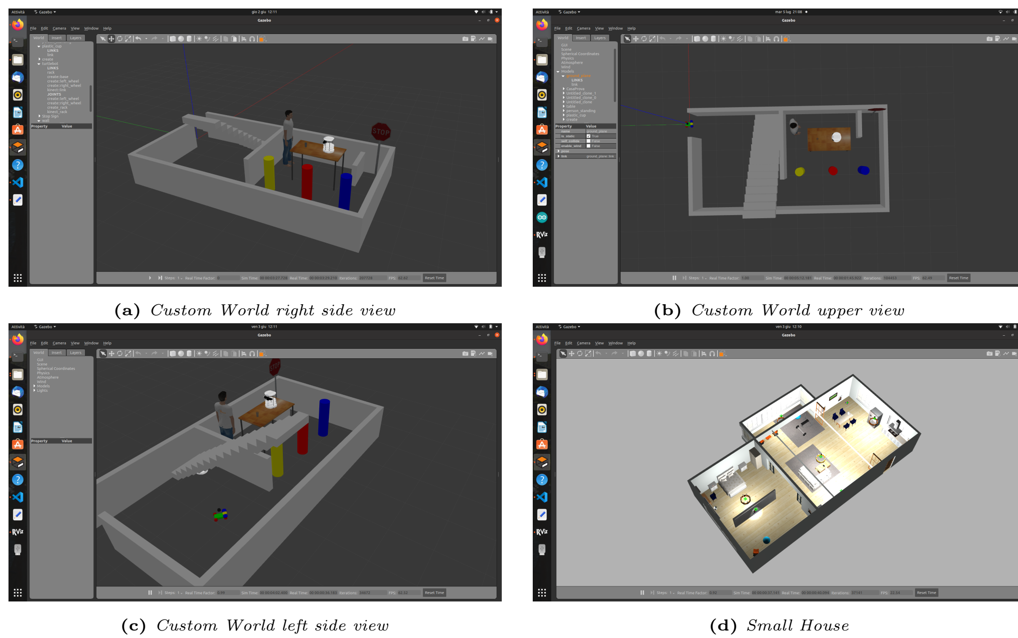
The choice of these worlds was made in order to show an increasing difficulty and, in the last case, to test the algorithm in scenarios that can put it more



(a) *Straight World* **(b)** *Square World*

(c) *Circular World* **(d)** *Turtlebot World*

Figure 4.4: Some examples of the created worlds



(a) *Custom World right side view* **(b)** *Custom World upper view*

(c) *Custom World left side view* **(d)** *Small House*

Figure 4.5: Some examples of the created worlds

in difficulty. They were not the only worlds created and tested but, not to be redundant in this pages only these are shown.

The last world named *Small House*, was found on the internet and it was insert in this thesis because it is very well done, it is very realistic and it has been used to test the algorithm even in such confusing environments.

The worlds are not sufficient to perform the tracking since they are used only in the 3D simulation on Gazebo, to use them with tracking algorithm it was necessary to create a map associated to the world, a 2D projection of it.

This map is a 2D representation extracted from the 3D one but obtained from logged transform and laser scan data. The world are loaded in the *launch* file from the package TurtleBot. TurtleBot is a ROS standard platform robot, it is a low-cost, personal robot kit with open-source software and there are diffent versions of the TurtleBot series (Figure 4.6).

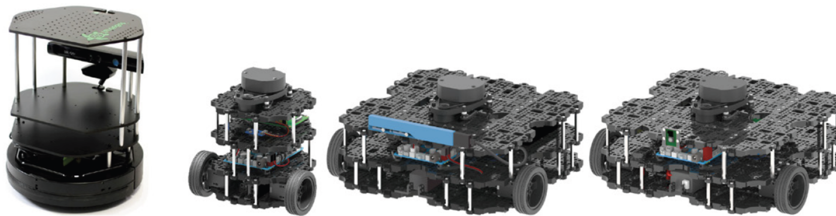


Figure 4.6: Different models of TurtleBot from Robotis

4.2.2 Mapping

All the environment are manually mapped, using both Gazebo and RViz, turning around for the world with keyboard using the ROS package *Turtlebot_teleop* (that permits to command the velocity and the steering using the keyboard of the PC).

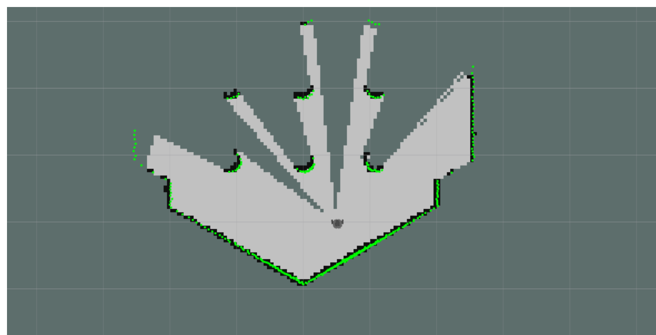


Figure 4.7: view of RViz while GMapping is scanning with turtlebot

For this purpose it was helpful the use of RViz, that is a 3D visualizer for the

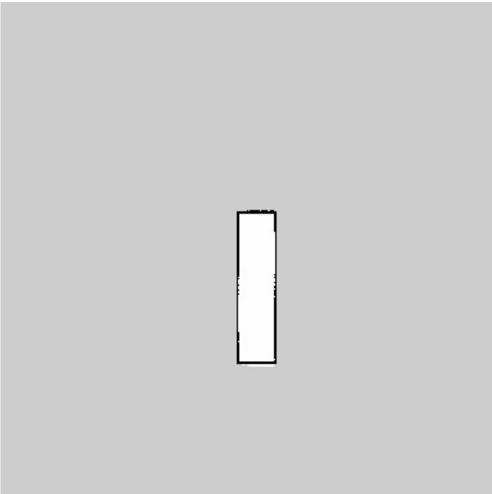
Robot Operating System (ROS) framework, where you can see the map appearing while you are scanning the walls and obstacles (Figura 4.7).

In order to do the mapping it was used the GMapping algorithm: The GMapping algorithm is a laser-based SLAM (Simultaneous Localization And Mapping) algorithm for grid mapping. This is probably the most used SLAM algorithm, currently the standard algorithm on the Willow Garage PR2 (Personal Robot 2 that is a very popular mobile manipulation platform, see Figure 4.8) with implementation available on `openslam.org`.

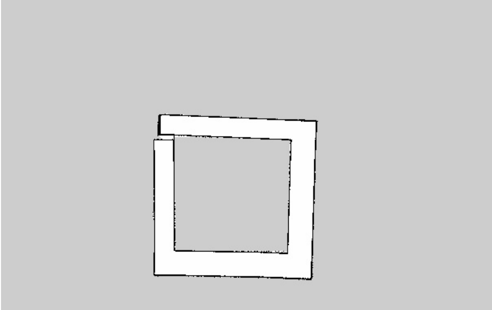
Figure 4.8: The Willow Garage Personal Robot 2

The algorithm was initially proposed in the Conference on Uncertainty in Artificial Intelligence [15], and the main idea is to use *Rao–Blackwellized* particle filters (RBPFs) to predict the state transition function. The algorithm is also known as the RBPF SLAM algorithm. In subsequent years, two major improvements were made by optimizing the proposal distributions and introducing adaptive resampling, making the algorithm much more suitable for practical applications. It is then dubbed GMapping (G for Grid) due to the use of grid maps.

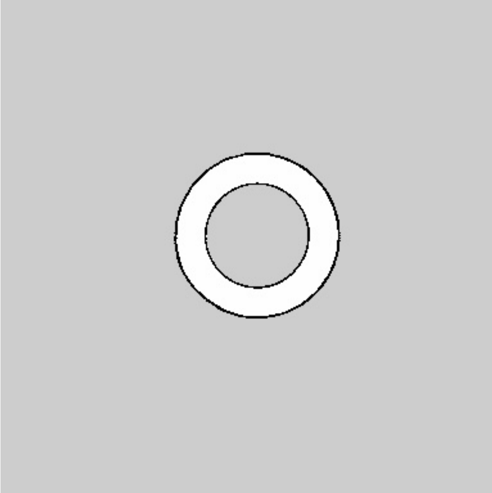
In ROS, the GMapping package provides laser-based SLAM as a ROS node called slam_gmapping. A limit of all the mapping packages is the long straight paths where the mapping can be confused since, if the straight path is too long, the algorithm scans a set of all equal points, leading to incorrectly results.



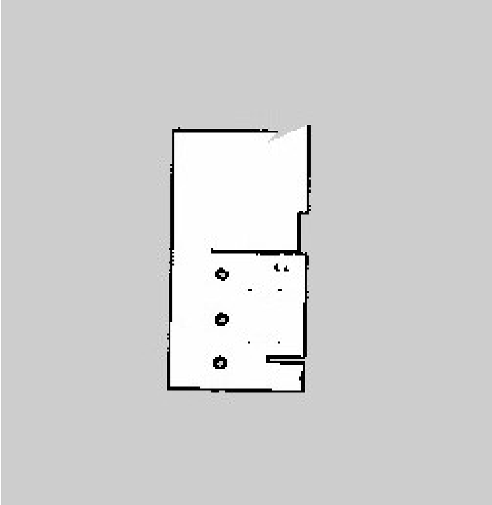
(a) *Straight World*



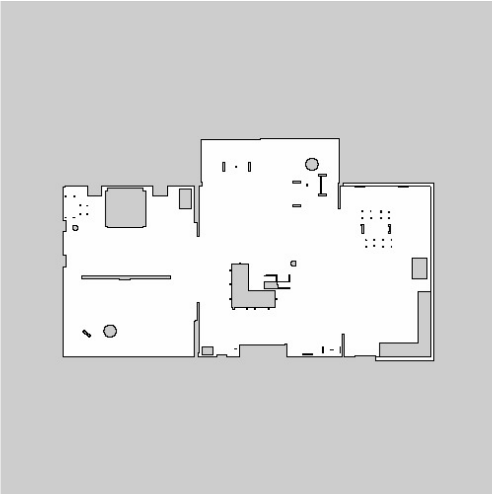
(b) *Square World*



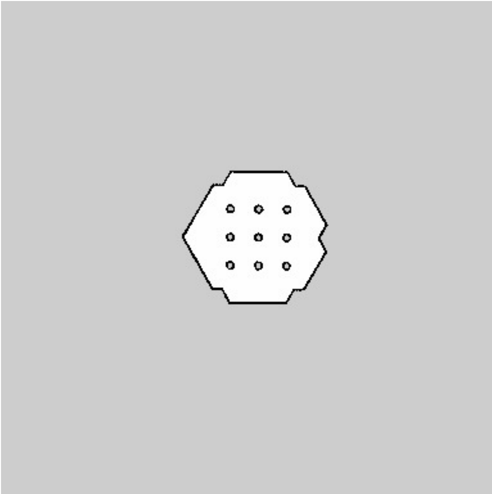
(c) *Circular World*



(d) *Custom World*



(e) *Small House*



(f) *Turtlebot World*

Figure 4.9: Some of the created maps

4.3 Path Tracking

The path tracking ability of an autonomous vehicle is crucially linked to its dynamic behaviour, especially when considering high speeds and low surface friction values, because the lateral dynamics are highly nonlinear and the vehicle response to maneuvers can exhibit oscillations and diverge. In this thesis, an advanced gain scheduling control strategy, capable of adapting to vehicle speed variations, will be implemented for the lateral dynamics stabilization and to enhance the damping capacity of the system.

Thus, a predictive control algorithm will be adopted for the path tracking purpose, exploiting its predictive capabilities and the possibility to define constraints on the input variables. Finally, the control strategy will be validated by simulations on a nonlinear model of the system.

For an autonomous driving vehicle it is essential to find the best path to reach a given goal in a reasonable time and with an appropriate speed, being careful not to hit any obstacles.

Thus, it is needed to know the geometry of the work environment, of the walls, of the obstacles, of the robot, his kinematic and the initial conditions. With the geometries, using the implemented algorithm, a desired trajectory was found as a continuous sequence of collision free configurations between initial known position and desired known final position.

There are several path planning techniques, which can be divided into two macro approaches: a local planner and a global planner.

The local one tries to solve the path planning problem in a general mode, using only the information available to the robot neighborhood to determine the next movement command.
The global planning instead, needs a complete knowledge of the environment. Once the ideal trajectory was defined, the next step is the path following problem, which consists in elaborating a control logic that allows the vehicle to move along the path as precisely as possible. This problem can led back to the determination of the commands on the speed and the direction to be imposed on the robot, compatibly with its kinematic limits and also its dynamic.

The approach that was used with the ackermann model is to convert a velocity signal into a torque command to be imposed on the wheels of the vehicle. Another approach, if differential drive is used, is to imposing different velocity to the wheel on the same axle in order to allow the vehicle to rotate and follow the desired direction.

4.3.1 Obstable Avoidance

The path following do not include the obstacle avoidance strategy. This strategy must work real-time in parallel with the path following. Its purpose is to allow the vehicle to manage the presence of any obstacles that may suddenly arise along the route and therefore modifying in real time the previously planned route in order to adjust the velocity and the direction.

To better explain how this algorithm works an example was shown that consists of a mobile robot which approaches polygon-shaped obstacle while traveling to the current goal. The planned trajectory is composed of multiple robot poses. The planner aims to arrange each two consecutive poses according to the desired temporal resolution.

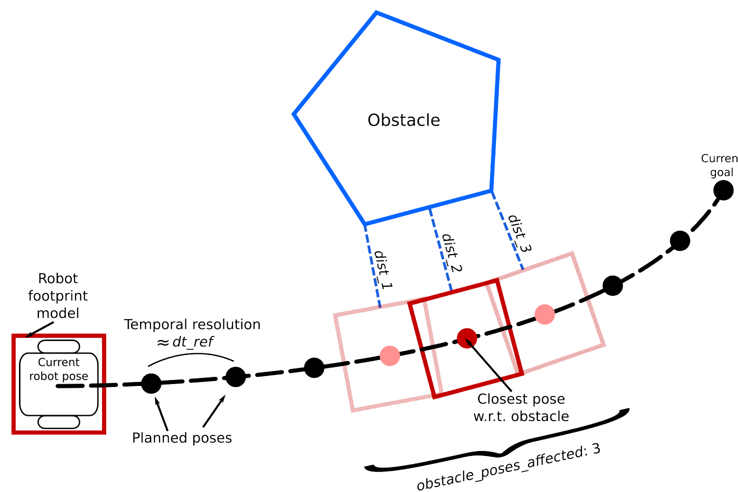


Figure 4.10: Obstacle avoidance scenario

For obstacle avoidance, the distance between a planned pose and the obstacle has to be bounded from below. The example trajectory in the Figure 4.10 consists of 8 changeable poses since start and goal poses are fixed. Many distance calculations are required in order to achieve a collision free trajectory, the optimizer invokes the computation of cost function values many times. According to the value of local planner parameter *obstacle_poses_affected* (the value of this parameter slightly influences the smoothness of the trajectory around obstacles) neighbors of the closest pose are taken into account as well. Only this selected subset of poses is taken into account in the subsequent optimization step, in the shown scenario there are 3 poses and therefore 3 penalty terms. Bigger obstacles require more connected poses to avoid not admissible cutoff. You may also choose a high value in order to connect all poses with each obstacle.

Note that the robot footprint model (see section 6.2.3 for details) is taken

into account for distance calculations and is therefore crucial for the required computational resources.

4.3.2 TEB Local Planner

In this work the Time Elastic Band (TEB) local planner was used as the local planner.

The teb_local_planner package implements a plugin to the base_local_planner of the 2D navigation stack.

In the context of mobile robot navigation trajectory planning and control it constitutes a fundamental task in applications such as service robotics and autonomous transportation systems. Online planning is preferred over offline solutions since the former integrates planning with state feedback and responds to dynamic environments and perturbations at runtime.

Most planners only consider non-holonomic constraints of a differential-drive robot, which however does not capture the minimum turning radius of a car-like robot.

The Elastic Band (EB) approach is well known for online path deformation [18]. Predefined internal forces contract the path while external forces maintain a separation from obstacles. However, conventional path planning does not explicitly incorporate temporal and dynamic constraints. Discrete trajectory way-points are repelled from obstacles. Their connectedness w.r.t. a dynamic motion model is restored afterward.

Online trajectory optimization based approaches are often limited by the computational burden to converge to a feasible and optimal solution under real-time constraints.

The TEB approach provides a real-time capable online trajectory planner for differential-drive robots [20]. It is inspired by the idea of the EB method but reformulates planning of trajectory and controls as a sparse optimization problem. The TEB mimics a predictive controller and efficiently optimizes the trajectory with respect to dynamic constraints while explicitly incorporating temporal information to achieve a time-optimal solution.

The TEB is intended as a local planner in a hierarchical navigation architecture. The planning horizon coincides with the robot perceptual range, the remote path based on a static map is entrusted to the global planner.

The *teb_local_planner* ROS package implements a plugin to the *bas_local_planner* of the 2D navigation stack. The underlying method locally optimizes the robot's

trajectory with respect to trajectory execution time, separation from obstacles and compliance with kinodynamic constraints at runtime.

This package implements an online optimal local trajectory planner for navigation and control of mobile robots as a plugin for the ROS navigation package. The initial trajectory generated by a global planner is optimized during run-time with respect to minimizing the trajectory execution time, separation from obstacles and compliance with dynamic constraints such as satisfying maximum velocities and accelerations.

The optimal trajectory is efficiently obtained by solving a sparse scalarized multi-objective optimization problem. The user can provide weights to the optimization problem in order to specify the behavior in case of conflicting objectives.

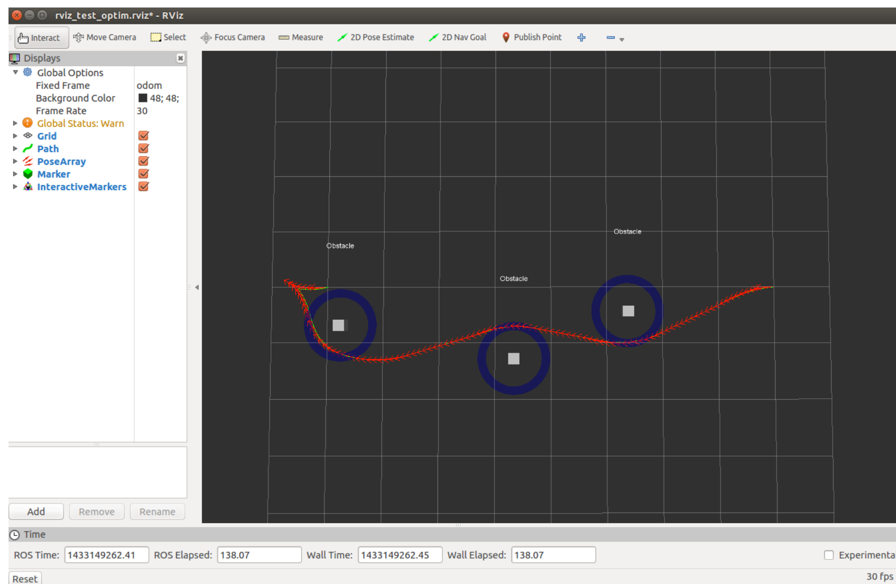


Figure 4.11: rviz window that shown optimizing procedure

Since local planners such as the Timed-Elastic-Band get often stuck in a locally optimal trajectory an extension is implemented as they are unable to transit across obstacles. A subset of admissible trajectories of distinctive topology is optimized in parallel. The local planner is able to switch to the current globally optimal trajectory among the candidate set. Distinctive topologies are obtained by utilizing the concept of homology and homotopy classes.

The package includes a simple test node *test_optim_node* that optimizes a trajectory between a fixed start and goal pose which do not coincide. For configuring the planning of a single trajectory between start and goal, three point obstacles are included. They are represented as an interactive markers type and therefore the obstacle configuration can be changed by clicking and holding the

blue circle around each individual obstacle.

Since the Timed-Elastic-Band utilizes a local optimization scheme, the trajectory cannot transit across obstacles.

It is possible to customize the optimization just by running the package rqt_reconfigure, trying to customize the optimization according to the desires. It is advised to modify the parameters only slightly and one by one, since some parameter sets could lead to undesired convergence behavior or to a bad performance, especially by changing the optimization parameters.

Chapter 5

Experimental Setup

In this chapter the main steps that bring together what has been seen in the previous chapters will be described. In particular, it describes how to send commands from the laptop to the raspberry, the implementation of the various hardware on board the RC car, and the mapping mode used. The idea is to map a small room (with a L-shape) and then to be able to navigate the vehicle inside it. The Lidar in this phase was fundamental since it allows the instant location of the robot in the environment in which it is located.

5.1 Connection Setup

Before the simulations with the real RC Car and after the assembly of all the parts of the car, it is necessary to build a sort of connection between the various hardware components.

5.1.1 Raspberry Image

Ubuntu 20.04 image with pre-installed ROS Noetic from the Ubiquity Robotics site (`https://www.ubiquityrobotics.com/`) was downloaded and flashed on the Raspberry. This image is perfect for building a Raspberry Pi robot, in fact, they produce a robot named *Magni* (Figure 5.1) that is also available on their site, is a quite expensive hardware solution for this work but cheap respect other commercial robots (it costs about 2500$).

In the first step of this thesis the Ubuntu 16.04 Xenial operating system was chosen to be installed, with ROS Kinetic on the Raspberry. Anyway, this was

an obsolete version of the operative system and, in fact, many incompatibility problems have arisen. Fortunately, a few days later Ubiquity released a new and updated version of its operating system that is fully compatible with the work done. You could also install a normal version of the operating system and install ROS and the packages they needed separately, but this version was already complete and very convenient to use because with a single line of code you could completely detach from the dependencies on their native robot.

Figure 5.1: The Magni Robot from Ubiquity Robotics

Magni has an integrated Raspberry Pi4 running Ubuntu and ROS, enabling powerful computing power while keeping the price tag low. For how it is done, the computing unit can be easily upgraded if the application needs more performance. It is an hardware platform in the form of a mobile base, it can handle vision localization, communication and mobility.

5.1.2 Connection Between Raspberry and Laptop

To make the Robot and a laptop communicate with each other, the installed software in the Raspberry has simplified things and, with a couple of terminal instructions, you are able to access the ROS communication messages from the laptop. This also allows for a more complete and simple visualization with RViz.

The final scope of this step is that both machines (the laptop and the Raspberry) have internet and can communicate over a Secure Shell Protocol (**SSH**) session, that is a cryptographic network protocol for operating network services securely over an eventually unsecure network.

ROS nodes exchange topic and service data peer-to-peer; this means that all nodes need to be able to communicate over the network with all other nodes.

Each node has an URI (Uniform Resource Identifier) that it sends to the ROS master, along with its list of topics and services. When other nodes want to use a topic or service, they ask the ROS master for the URIs of the nodes on those topics, the ROS master gives them those URIs. The other nodes then use the node's URI to establish topic and service connections.

By default, a node creates its URI from the hostname of the computer that it is running on. Setting ROS_IP overrides this, and causes the node to use the specified IP address.

The ROS_IP should be set to the IP address of the computer that the ROS node is running on. If you want your node to be reachable on several different networks, you can either set up routing so that the node's IP is reachable from all networks. This way, hosts on one network will resolve one IP address for the node, and hosts on the other network will resolve the other IP address for the node.

When ROS nodes running on multiple machines, only one master process needs to be started. The ROS_MASTER_URI will have the address of the machine on which you will run the master process (in this scenario the master process is running on the Raspberry). This value should be the same on all machines.

To be able to communicate between the laptop and the Raspberry (which are in the same network) every time a terminal is opened it is necessary to set the ROS_IP and the ROS_MASTER_URI in both the laptop and the robot (to avoid repeating it every time the simulation is run, is sufficient to add a couple of commands lines in the **/.bashrc** of the laptop).

Now, on the master (i.e. the robot), **roscore** is running and so the observer (i.e. the external laptop) has access to all the messages and topics that are on the master.

For the step above, the Raspberry was used like a WiFi network access. The robot, once the operative system was flashed, will boot up in WiFi Access Point mode and it will not connect to internet, it is only providing its own network where the laptop can be connected. With the access point the user is enabled to connect directly to the robot (with WiFi or Ethernet connection), for example it can be driven directly from a workstation running ROS commands such as keyboard teleoperation. However, with this connectivity, the robot cannot access the Internet. To fix that the instruction found in [16] have been followed.

5.1.3 Hardware Implementation

The next step was testing the RPLiDAR, in particular the **RPLiDAR A3** is used for this build (as described in Section 3.2). This was easily done with the following way: once the drivers corresponding to this LiDAR model were founded and installed (for the RPLiDAR family they can be found in the following github repository `https://github.com/Slamtec/rpLiDAR_ros`), the RPLiDAR was connected to the power supply and to the Raspberry in order to test it. The launch file for the test of the laser was founded in the repository package defined above, once the test has been launched what was seen by the laser is automatically displayed in an *RViz* window as shown in Figure 5.2.

The github repository includes the drivers and launch files of the whole RPLiDAR family, it is essential to select the right one with a baudrate of $256000bps$ as the others have a different value of communication speed and you will not be able to view anything.

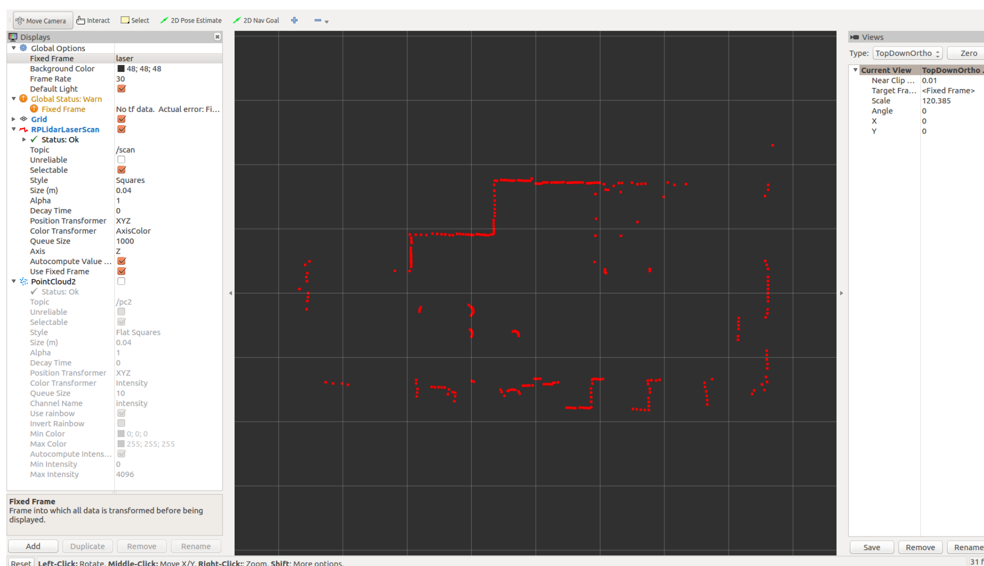


Figure 5.2: The RViz visualization of the /scan topic

This test can also be used to set the display parameters such as the resolution of detected object that by default is set to $0.03m$ (precisely this parameter indicates the size of the red squares in the Figure 5.2) but for better visualization is set to $0.08m$.

In this work was used a RPLiDAR A3 but it can also work with other types of LiDAR, just search for the respective drivers and proceed in the same way as described above.

5.1.4 Raspberry and Arduino talk to each other

At this point it was needed to get ROS and Arduino to talk to each other. The goal is to get commands from the Raspberry to the Arduino with the scope of having the Arduino to tell the motors how to move. The package *rosserial* was installed on the ROS environment. This is a ROS module that enables Arduino-ROS communication. The *rosserial_lib* library is added to the Arduino IDE in the Laptop where the script to control Arduino is created.

Rosserial

With the use of *rosserial_arduino* package, ROS can be used directly with the Arduino IDE. It provides a ROS communication protocol that works over the Arduino's UART (Universal Asynchronous Reception and Transmission, a simple serial communication protocol that allows the Arduino to communicate with serial devices). It allows the Arduino to be a full-fledged ROS node which can directly publish and subscribe to ROS messages, publish transforms and get the ROS system time. Every time the serial communication is opened, it is necessary to specify in which port the USB port of the device is connected.

The ROS bindings are implemented as an Arduino library. Like all Arduino libraries, *ros_lib* works by putting its library implementation into the libraries folder of the sketch in an iterative way by continuously fusing various measurements from the onboard sensors.

To create the ROS node on Arduino that would allow ROS communication to move the robot in the world, a few lines of code have been written directly in the Arduino IDE. In Figure 5.3 only the callback function is highlighted, which allows to read the geometry_msgs/Twist related to the cmd_vel topic and convert it into a PWM signal to be sent to the DC motor. The goal is to map values in percentages to the range $[0, 255]$ that the motor controller understands. This function also converts the steering angle from radians to degrees.

```
void messageCb( const geometry_msgs::Twist& msg){
  speed_ang = msg.angular.z;
  speed_lin = msg.linear.x;


  linSpeed_PWM = 1446.1 * speed_lin * speed_lin * speed_lin - 1848.9 * speed_lin * speed_lin + 1080.5 * speed_lin  - 44.451;


  angle = speed_ang * 180 / 3.141592653589793;

}
```

Figure 5.3: Arduino code that shows the implementation of the conversion function

Calibration

In order to find a function that maps the linear speed of the installed motor to PWM is necessary to do some practical tests:

- create a small Arduino program (Figure 5.3) that starts with PWM at 0 and slowly increase it until the robot starts moving (as will also be explained in detail in the section 6.2.1). It's moving at PWM = 100, that is the minimum speed.
- with the help of a stopwatch, the time needed to travel a meter with a variable PWM (25% PWM, then 50%, then 75%, then 100%) is measured.
- find the equation that converts PWM to speed with Excel which finds the function from experimental points.

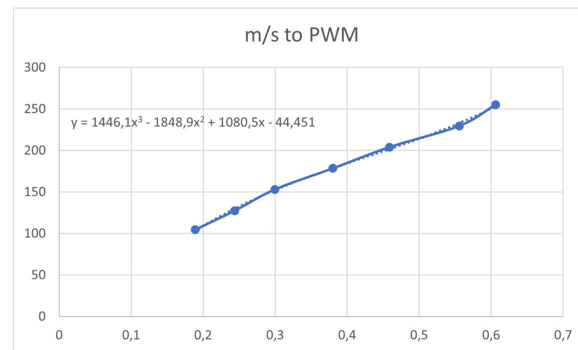


Figure 5.4: Function generated with Excel from the point cloud taken from the experiment

The car moves with a fairly high PWM as the engine of the car is not very powerful and, with the modifications made, the vehicle has become quite heavy.

The point cloud in the test described above leads to the function also in Figure 5.4:

$$y = 1446.1x^3 - 1848.9x^2 + 1080.5x - 44.451 \tag{5.1}$$

To find this function, each measurement was taken 4 times and averaged.

For the steering angle the procedure used is slightly different. The idea is to write simple code (Figure 5.5) that converts angular velocity to steering angle, note that the angle that this function returns, is in radians so it is also necessary to transform it into degrees for a better understanding of the angle.

```
7  def convert_trans_rot_vel_to_steering_angle(v, omega, wheelbase):
8    if omega == 0 or v == 0:
9      return 0
10
11   radius = v / omega
12   return math.atan(wheelbase / radius)
13
```

Figure 5.5: This code will convert angular velocity to steering angle

In Figure 5.5 *radius* is turning radius, v is the linear velocity, *omega* the angular velocity and *wheelbase* is the distance between the axes (calculated in $[m]$).

Finally, it should be taken into account that the working angles of the servomotor require an additional transformation of the received angle into degrees.

After a simple measurement, the steering angle of the real car can be approximated in the range of $-30°$ to $+30°$ (ϕ angle in Figure 5.6), the servo motor instead has a steering angle ranging from 0 to 180 degrees. Therefore it is necessary to map the real calculated steering angles with those of the servomotor.

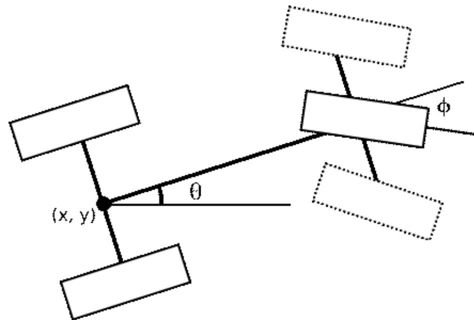


Figure 5.6: the approximated steering angle in the real car

If the angle given in input is out of this range, problems may arise. There is a risk that the servomotor forces the structure and breaks some component, therefore in the code it was necessary to add a couple of lines to overcome this problem and to ensure that the angle of the servo never goes out of its range in case of communication errors.

5.1.5 Sensors

The sensor setup includes LiDAR and depth camera to collect environmental information as well as internal measurements from the IMU (Inertial Measurement Unit). A 2D map is generated after the processing done by a mapping algorithm. Depending on the purpose of the map, different SLAM algorithms are available. In this thesis the GMapping algorithm is already being used for the mapping in

the simulation. Hence, for this section, it was used the Hector-SLAM algorithm. The main difference is that GMapping is based on a particle filter pairing algorithm, while Hector-SLAM is based on a scan matching algorithm.

The Hector-SLAM algorithm differs from other grid-based mapping algorithms as it does not require any odometry information, but it needs laser data and an a-priori map. In order to use *rosserial* libraries in the code, it must be first included *ros.h* header prior to including any other header files, otherwise the Arduino IDE will not be able to locate them.

5.1.6 Hector-SLAM mapping

For a mobile robot, SLAM involves both localization and mapping. It does not require odometry information, but it needs only laser data.

In ROS environment, *hector_slam* package (an example in Figure 5.7) uses the *hector_mapping* node for learning a map of the environment and simultaneously estimating platform's 2D pose at laser scanner frame rate. The frame nodes and options for *hector_mapping* have to be set correctly.

From Figure 5.7 can be seen that, the result is practically the same respect the one described in section 4.2.2.

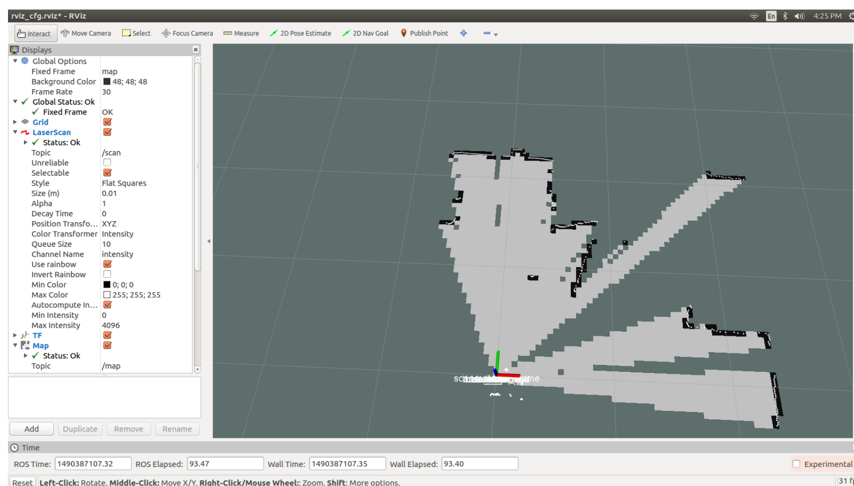


Figure 5.7: Example of the hector algorithm scanning the target room

The general relationship between *map*, *odom* and *base_link* frames is already described in Section 5.2 and shown in Figure 5.9. Two frames are used in between *odom* and *base_link*: the *base_footprint* frame provides no height information and represents the 2D pose (position and orientation) of the robot; the *base_stabilized* frame adds information about the robot height relative to the *map*/*odom* layer the

base_link frame is rigidly attached to the robot and adds the roll and pitch angles compared to the *base_stabilized* frame. For a platform not exhibiting roll/pitch motion, the *base_stabilized* and *base_link* frames are equal. This step will add mapping functionality to the robot.

The next step has been creating a ROS package that would allow ROS communication to move the robot in the world. Basically, in this step Arduino comes in and, what was said in the previous section, will be exploited here. The goal was to make a subscriber node that would run on the Arduino and listen to the topic *cmd_vel*. This lower level robot control is useful since it allows to send commands from the laptop to the robot in order to navigate into the environment to create a map.

Now the procedure to create the map is pretty the same as in Section 4.2, the main difference is that the files that can be launched are different and they are referred directly to the real robot and not to the Turtlebot.

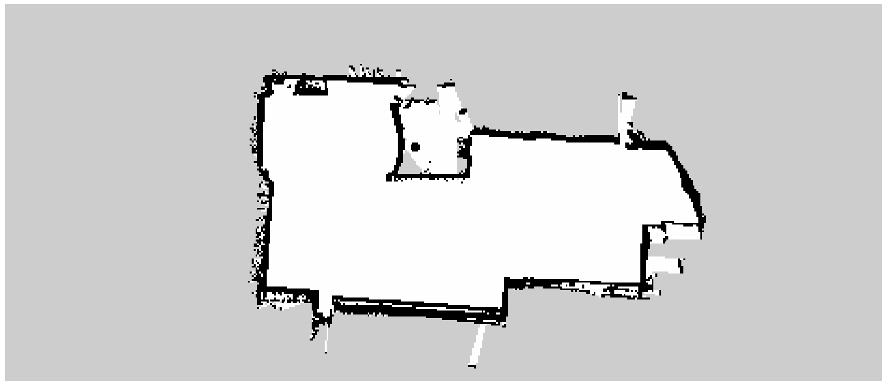


Figure 5.8: The Map generated with the Hector-SLAM algorithm

Once the car is driven slowly through the environment, the map is built. The Figure 5.8 shows the results of the mapping, the resolution of it depends mainly on how critical points and the speed at which the room was scanned.

The inaccuracies that can be seen from the figure are due to the fact that in the room there were objects or small holes between the furniture that could not be removed and therefore led to this result. This map can be used for both the localization and the navigation of the robot.

5.1.7 Localization and Navigation

The purpose of this step is to make the robot locate itself in the known map recorded in section 5.1.6 and navigate through it to a specified pose.

The main components of a probabilistic localization system for a robot moving in 2D are:

- **Laser scanner** : The */scan* topic can be read from the laser scanner in real-time
- **Odometry** : The odometry will be used to understand how much the robot has moved at that moment. It helps to determine how the robot should change its previous localization estimate to accommodate the motion
- **Saved map** : It is possible to compare the robot's LiDAR observations in real-time with observations from different points on the map and say that the point on the map is the position of the robot.

Odometry's data are usually extracted from accelerometers (Inertial Measurement Units (IMUs)), wheel encoders, GPS, or cameras but in this configuration there is only the LiDAR scanner. Therefore, to get odometry data it was necessary to exploit an undocumented feature of Hector-SLAM that provides the displacement of the laser scanner over time. To do this is used the content of the repository in [21] adapting it to this problem.

To localize the RC car in a known map, information are feeded from the recorded maps and the Hector SLAM odometry to AMCL (Adaptive Monte Carlo Localization described in [19], that is a probabilistic localization system for a robot moving in 2D).

5.2 Coordinate Frames

Developers of drivers, models, and libraries need a share convention for coordinate frames in order to better integrate and re-usable software components. Shared conventions for coordinate frames provides a specification for developers creating drivers and models for mobile bases. Similarly, developers creating libraries and applications can more easily use their software with a variety of mobile bases that are compatible with this specification. It specifies frames that can be used to refer to the mobile base of a robot.

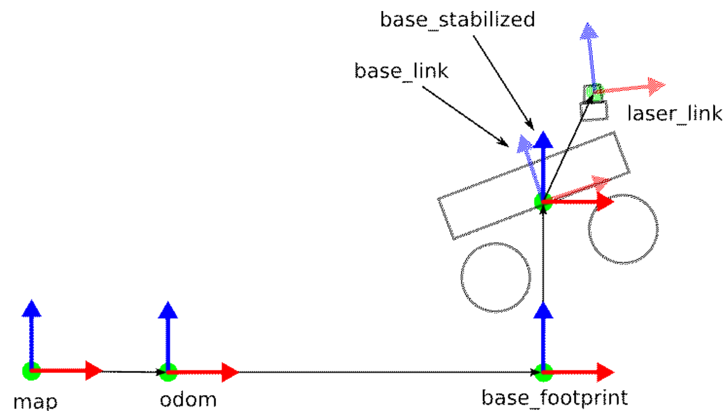


Figure 5.9: The image above shows all potential frames of interest in a simplified 2D view of a robot travelling through rough terrain, leading to roll and pitch motion of the platform

The convention for coordinate frames are:

base_link

The coordinate frame called *base_link* is rigidly attached to the mobile robot base. The *base_link* can be attached to the base in any arbitrary position or orientation; for every hardware platform there will be a different place on the base that provides an obvious point of reference.

odom

The coordinate frame called *odom* has its origin at the point where the robot is initialized and is fixed in the world. The pose of a mobile platform in the *odom* frame can drift over time, without any bounds. This drift makes the *odom* frame useless as a long-term global reference. However, the pose of a robot in the *odom* frame is guaranteed to be continuous, meaning that the pose of a mobile platform in the *odom* frame always evolves in a smooth way, without discrete jumps. In a typical setup the *odom* frame is computed based on an odometry source, such as wheel odometry, visual odometry or an inertial measurement unit. The *odom* frame is useful as an accurate, short-term local reference, but drift makes it a poor frame for a long-term reference.

map

The coordinate frame called *map* is a world-fixed frame, with its Z-axis pointing upwards. The pose of a mobile platform, relative to the *map* frame, should not significantly drift over time. The *map* frame is not continuous, meaning the pose of a mobile platform in the *map* frame can change in discrete jumps at any time. In a typical setup, a localization component constantly re-computes the robot pose in the *map* frame based on sensor observations, therefore

eliminating drift, but causing discrete jumps when new sensor information arrives. The *map* frame is useful as a long-term global reference, but discrete jumps in position estimators make it a poor reference frame for local sensing and acting.

Chapter 6

Graphical Simulations

This chapter presents the experimental results on path planning obtained with both ROS and Gazebo software. The attention is directed to the generation of the path: the vehicle is approximated as a 2D material point, thus leaving out all the kinematic and dynamic aspects of the vehicle. The scenario has a key role in path planning, therefore some scenarios have been chosen to evaluate the simulations: some purely ideal ones and an almost real one.

6.1 Simulations

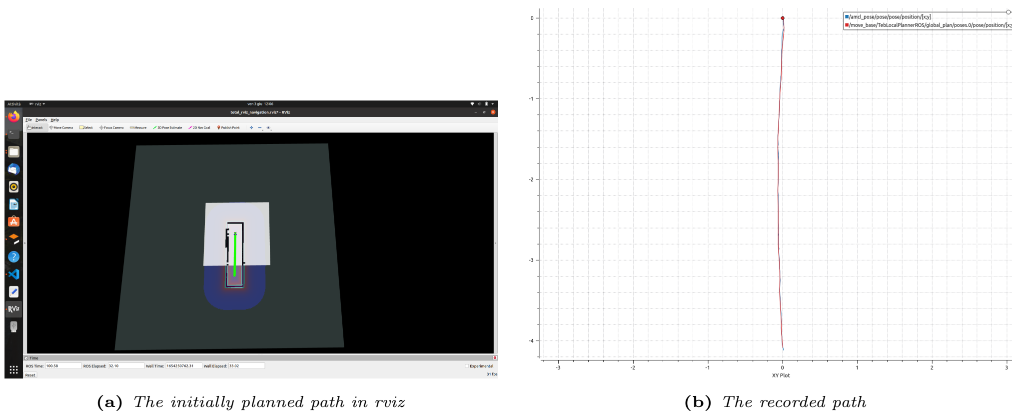
The simulations are performed separately on each created world and the performance of the simulations are evaluated by comparing the real path with the planned path. In some worlds the path is unique in the sense that it naturally has a beginning and an end so the simulation has a start point and a stop point. In other simulations the world is more general, so the starting and ending points are chosen assuming to test the planner in most insightful ways.

In the *rviz* visualization many topics can be shown, like the **tf**, that lets the user to keep track of multiple coordinate frames over time, the **laser_scan**, that outputs the obstacles views by the scanner, and the **odom**, that represents more then just the pose of the mobile robot (it can be seen that it also contains the current velocity, as well as the respective uncertainties). These topics and many more were not shown in *RViz* view to keep the simulation cleaner but sometimes they were useful to verify that everything was working correctly.

6.1.1 Straight World

This world is the easiest possible and it was one of the first tested. It has been inserted only because in the various simulations it has been noted that about $4m$ is the maximum length of the straight path that can be mapped. For longer distances misleading results are obtained, the robot ”sees” an environment that is always the same. Therefore it becomes disoriented and is no longer able to understand where it is in the straight path, as a consequence it will think the goal point was reached.

In this map it is clear where the start and the end points are.



(a) *The initially planned path in rviz* **(b)** *The recorded path*

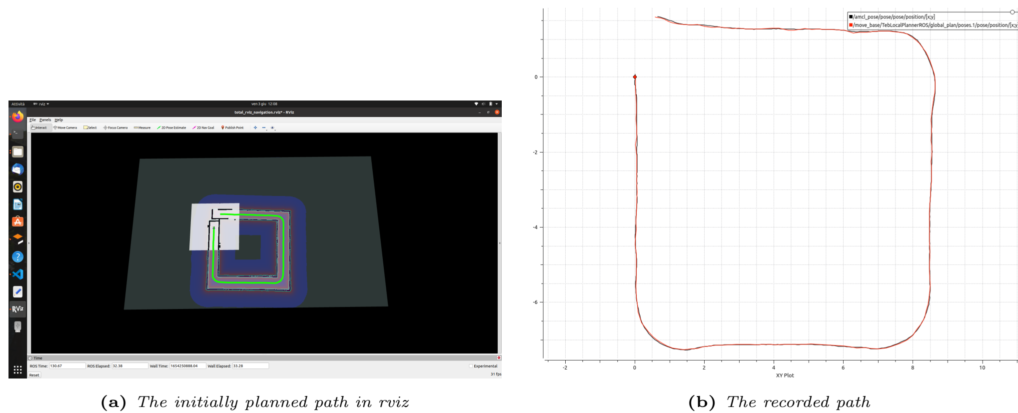
Figure 6.1: Straight World simulation

As you can see from the Figure 6.1, the route is very simple and easy to plan. There were no problems of any kind in this scenario.

6.1.2 Square World

This is a more challenging world where some problems arose, in particular the sharp curves which on many occasions made the robot stop and could mislead the path. These problems were solved with a precise tuning of the parameters (see section 6.2), in particular of the footprint size: the robot thought it could not make the turn as it was too big when it is evident that it should have made the curve without problems. The straight sections of the route have never created problems. Also in this simulation the starting and the ending points were clear.

From the images it is evident that the path followed is very faithful to that planned at the beginning of the simulation.



(a) *The initially planned path in rviz* (b) *The recorded path*

Figure 6.2: Square World simulation

6.1.3 Turtle World

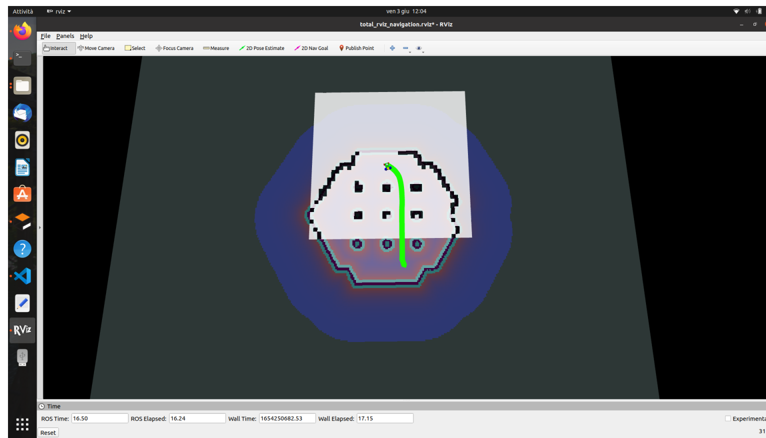
This world was included because there are several obstacles quite close together and the environment has very repetitive shapes. This may be a good way to see how the implemented algorithm behaves in these situations.

In this world several simulations were made. Since they are all more or less similar only one path is shown. The vehicle manages to juggle through the various cylinders present in the path and it reaches the goal point without problems even if the distance between them is quite small.
Obviously the algorithm, calculating the shortest path, will never make the path "zigzag" so it is not a very demanding world.

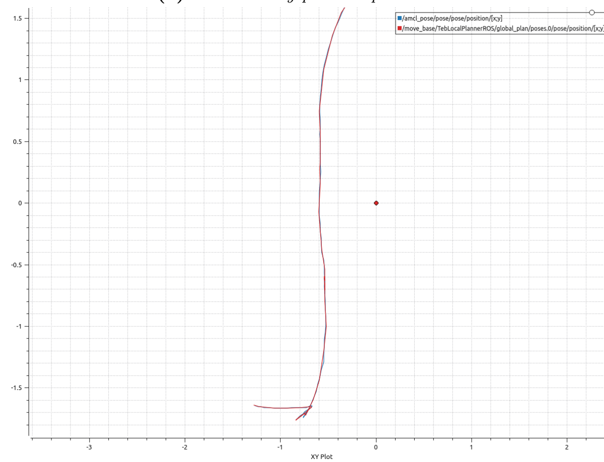
6.1.4 Custom Map

This world was purposely created to be both structured and a quite realistic situation. As can be seen, it tries to reproduce a couple of rooms on the ground floor of a house with some classic obstacles, an human, a table and some extra obstacles (the colored cylinders) just to complicate the path. Finally a small wall was added with the idea of making the hollow a sort of docking for the robot, and the stop signal has been inserted to indicate the goal point.

The planning works without problems. Obviously, due to the fact that the passage is not very wide, to pass the series of cylinders it does a few maneuvers because it cannot take the curve wide enough to be able to go straight into the hole and also in the docking proceeds slowly as space is limited. If the cylinders are removed the simulation goes more smoothly.

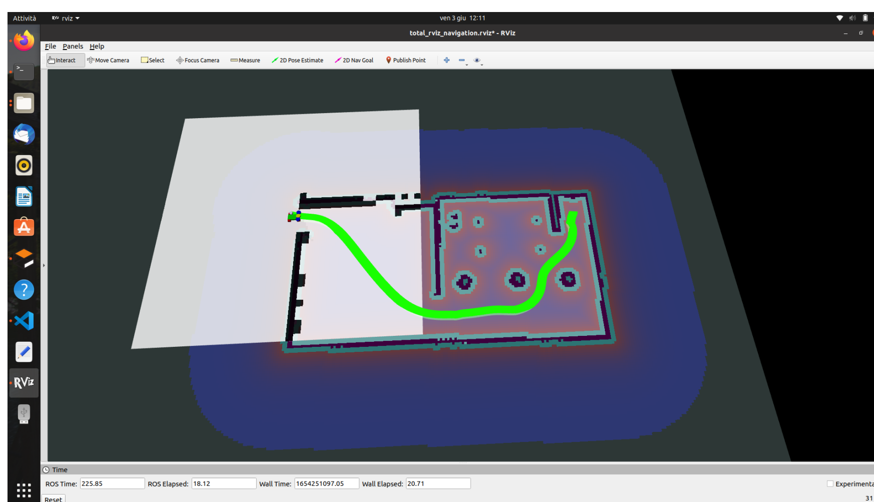


(a) *The initially planned path in rviz*



(b) *The recorded path*

Figure 6.3: Turtle World simulation

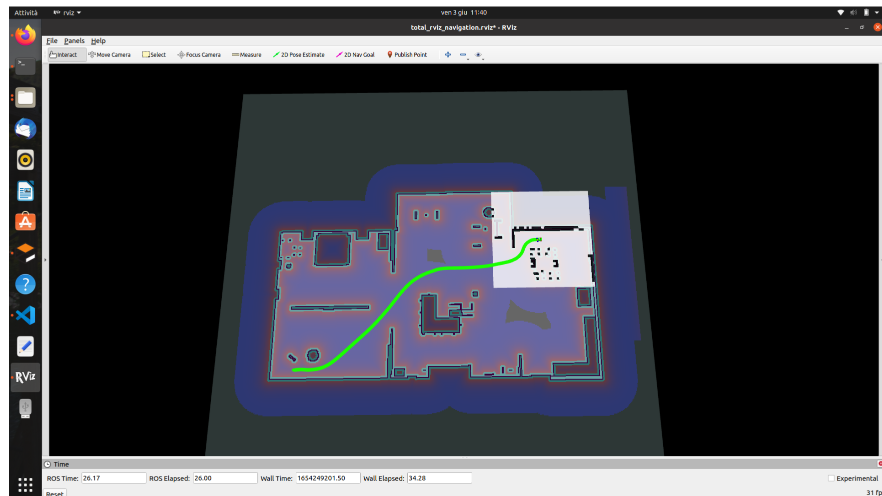


(a) *The initially planned path in rviz*

Figure 6.4: Custom World simulation

6.1.5 Small House Map

Among the various researches for this thesis work, I came across this world already done. It is not a very difficult world, it is just a little confusing, but it is very realistic and full of little obstacles that challenge the planning.



(a) *The initially planned path in rviz*

Figure 6.5: Small House simulation

Also in this case the planning performs very well. Various routes can be planned in this map and no one causes problems. From the *.launch* file the initial point can be modified in order to start the simulation in the desired point. The goal point is decided every time directly on *RViz* by simply clicking in the goal point and specifying the final desired orientation.

With the positive results with these maps it is clear that the algorithm works very well in different environments. In the next section it is shown the implementation of the algorithm in a real problem by charging it in the Raspberry mounted in the RC Car.

In all these simulations, the situations that have been most critical are especially the sharp curves or ways where the robot gets very close to the wall or obstacles. The algorithm knows the footprint of the robot and this has been set slightly larger than its actual size for safety reasons. It is with this simulation that, after several tests, it was understood that the dimensions of the robot should be slightly oversized.

6.2 ROS Navigation Turning

The ROS navigation stack is powerful for mobile robots to move from place to place reliably. The job of navigation stack is to produce a safe path for the robot to execute, by processing data from odometry, sensors and environment map. Maximizing the performance of this navigation stack requires some fine tuning of parameters.

6.2.1 Velocity and Acceleration

The velocity and acceleration of the robot is essential for local planners including the TEB local planner used for this thesis. In ROS navigation stack, local planner takes in odometry messages (”odom” topic) and outputs velocity commands (”cmd_vel” topic) that controls the robot’s motion.

Maximum and minimum velocity and accelerations are basic parameters for the mobile base, setting them correctly is very helpful for an optimal local planner behavior.

To obtain the robot’s maximum velocity is common to refer to the robot’s manual. A rough but efficient method, to use in the case that the manual is not available, is to find the maximum velocity by subscribing to the *odom* topic and reading odometry data while the robot is manually run until it reaches the maximum constant velocity.

To set the minimum speed, two main factors must be taken into account: first that the robot can also go in reverse, so the minimum speed will be negative. Second, on the other hand, is that to find the minimum absolute value of the linear speed, the minimum PWM (Pulse Width Modulation) signal sent to the DC motor must be taken into account, to be able to move the machine. This problem arises only in the case of the real simulation, the results obtained in the virtual one are not accurate although in the Gazebo simulator as many physical values are set.

Also the minimum rotational velocity must be set to a negative value so that the robot can rotate in both directions.

6.2.2 Global Planner

To use the *move_base* node in navigation stack it is necessary to have both a global planner and a local planner. The most used global planner is the *global_planner*.

The *global_planner* node basically has three parameters that actually determine the quality of the planned global path and they can be modified according to the type of route that will be followed. This parameters are *cost_factor*, *neutral_cost* and *lethal_cost*.

Setting *cost_factor* or *neutral_cost* too low or too high lowers the quality of the path, these paths do not go through the middle of obstacles on each side and have relatively flat curvature. On average these parameters are set to *cost_factor=0.55* and *neutral_cost=66*.

For *lethal_cost*, setting it to a low value may result in failure to produce any path, even when a feasible path is obvious. This parameter is usually set to *lethal_cost=253*.

6.2.3 Costmap

Costmap parameters tuning is essential for the success of local planners. In ROS, costmap is composed of static map layer, obstacle map layer and inflation layer: static map layer directly interprets the given static SLAM map provided to the navigation stack, obstacle map layer includes 2D and 3D obstacles. Inflation layer is where obstacles are inflated to calculate cost for each 2D map cell. Global costmap is generated by inflating the obstacles on the map provided to the navigation stack, local costmap is generated by inflating obstacles detected by the robot's sensors in real time.

In this area there are some important parameters that should be set as good as possible:

- *footprint*, that is the contour of the mobile base, in ROS it is represented by a two dimensional array where the first coordinate do not need to be repeated. This footprint will be used to compute the radius of the inscribed circle and the circumscribed circle which are used to inflate obstacles in a way that fits this robot. For safety, the footprint has to be slightly larger than the robot's real contour

- *inflation*, it consists of cells with costs ranging from 0 to 255. Each cell is either occupied, free of obstacles or unknown.
  inflation_radius and *cost_scaling_factor* are the parameters that determine the inflation: *inflation_radius* controls how far away the zero cost point is from the obstacle, *cost_scaling_factor* is inversely proportional to the cost of a cell, setting it higher will make the decay curve more steep.

These parameters are optimized by setting *inflation_radius* = 0.55 and *cost_scaling_factor* = 5.0 if a steep inflation curve is needed, *inflation_radius* = 1.75 and *cost_scaling_factor* = 2.58 if a gentle inflation curve is needed.

There would be many other parameters that could have been set but all have much less evident effects, and this affect the performance of these much less.

Chapter 7

Practical Simulations

In this chapter the implementation part will be treated, in particular the results obtained from the practical simulations of the RC Car in some simple paths.
Unfortunately we cannot expect all the positive results obtained from the planning of the section 6.1 in complicated paths, this because a lot of factors and difficulties come into play that are not present in simulation software.

7.1 Tests of Mobile Robot Motion

To begin the maneuverability test of the mobile robot, the first step is moving the robot from point to point so the first simulations made have also the purpose of verifying if the calibrations made in section 5.1.4 were correct.

The second step was to test the goodness of the algorithm on a slightly more complicated path.

7.1.1 Point to Point Motion

With the help of a meter, a starting and an arrival points were marked on the floor at a distance of one meter from each other. Then, with a simple program, the command to travel one linear meter was given to the car and the final frame was extracted once the car stopped. As can be seen in Figure 7.2 the result is accurate enough, the car only exceeded the route by a few centimeters. To obtain an even more precise result, it would be necessary to measure the number of pulse generated by the motor every revolution.

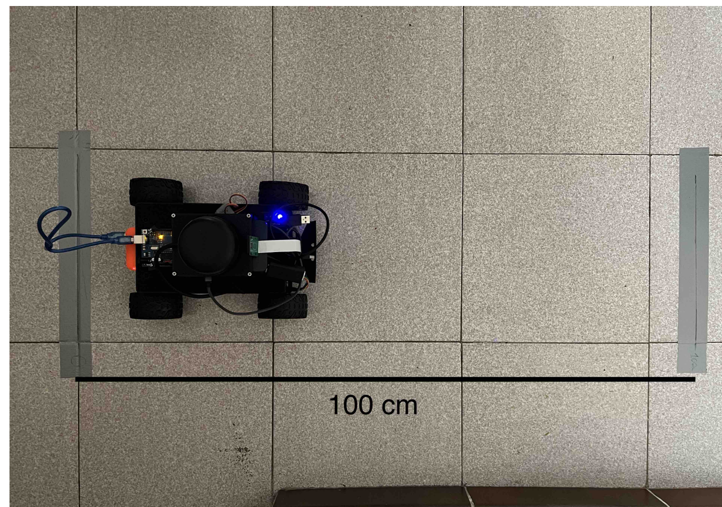


Figure 7.1: A frame of the setup taken from the calibration tests

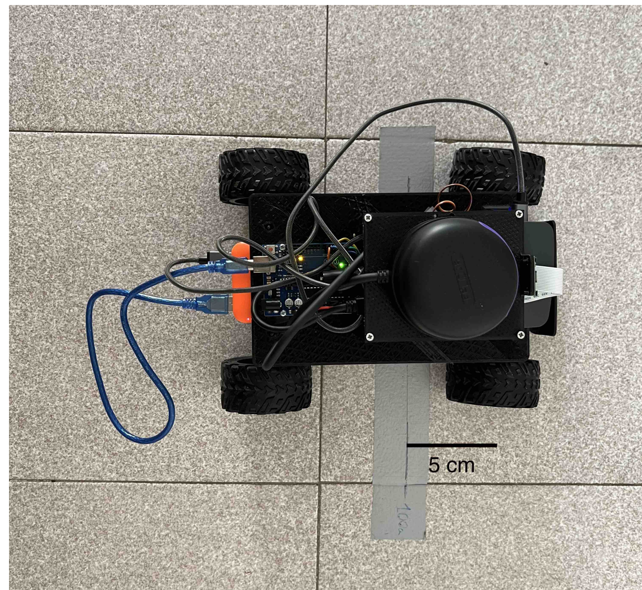


Figure 7.2: The precision of the calibration on the real map

Data from wheel encoders helps to determine how much each wheel has rotated. Since the radius of the wheels are known, the pulses data and the wheel radius data can be used to determine the distance traveled by each wheel. For this job, however, the precision achieved is sufficient.

7.1.2 Test of the RC Car

To test the maneuverability of the mobile robot and to check its controller, a simple path with a couple of curves is tested.

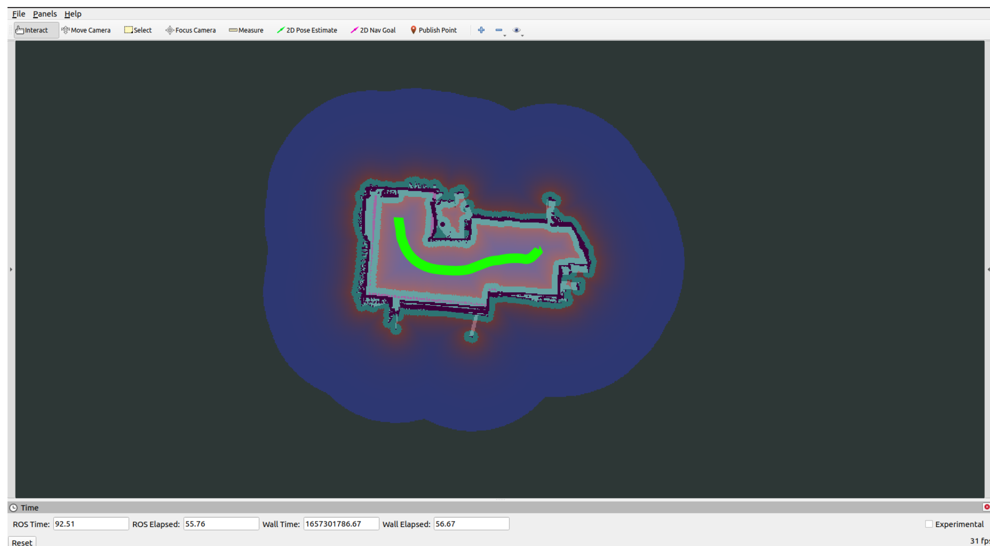


Figure 7.3: The planned path on the real map

As can be seen from the *RViz* simulation shown in Figure 7.3, a good result has been achieved. Once the goal point has been set (the starting point is automatically recognized as described in section 5.1.7) the car follows the planned path, reaching the expected point.

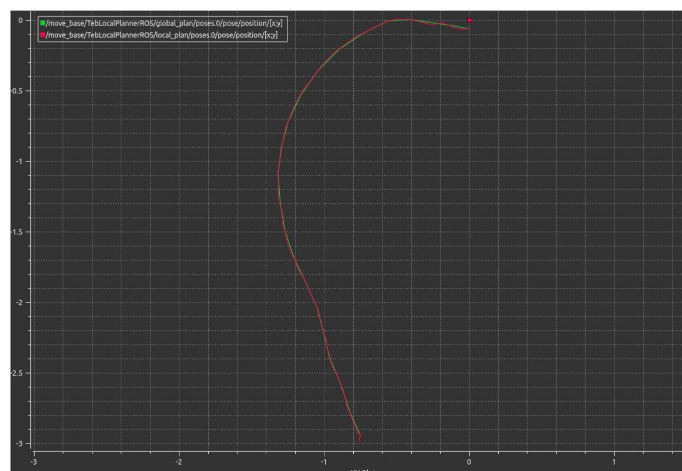


Figure 7.4: The planned path on the real map

Figure 7.5 show the execution of the algorithm, a video would have made it clearer but it has been tried to make it understood more faithfully with some frames extracted from a video. To make the figure more understandable it was

necessary to shorten the planned path (only for this figure), the longest possible path was taken so that the camera placed at the top could shoot in a static way.

The path planning is done correctly as in the simulations but, due to the fact that the linear travel speed of the path is higher than the simulation (otherwise the real car could not move as described in detail in the 5.1.4 section), the same path following quality is not obtained. However, considering the fact that it can only be oriented via scan topic, the results obtained are considered actually good.

Ideally this algorithm would also have the potential to avoid unexpected obstacles on the path, in the simulator part this feature was verified while in the real one was not tested because the space available for testing was limited. Figure 7.4 shows the recorded data of the real and the planned path. This approach shows more clearly the fidelity of the algorithm in following the planned trajectory.

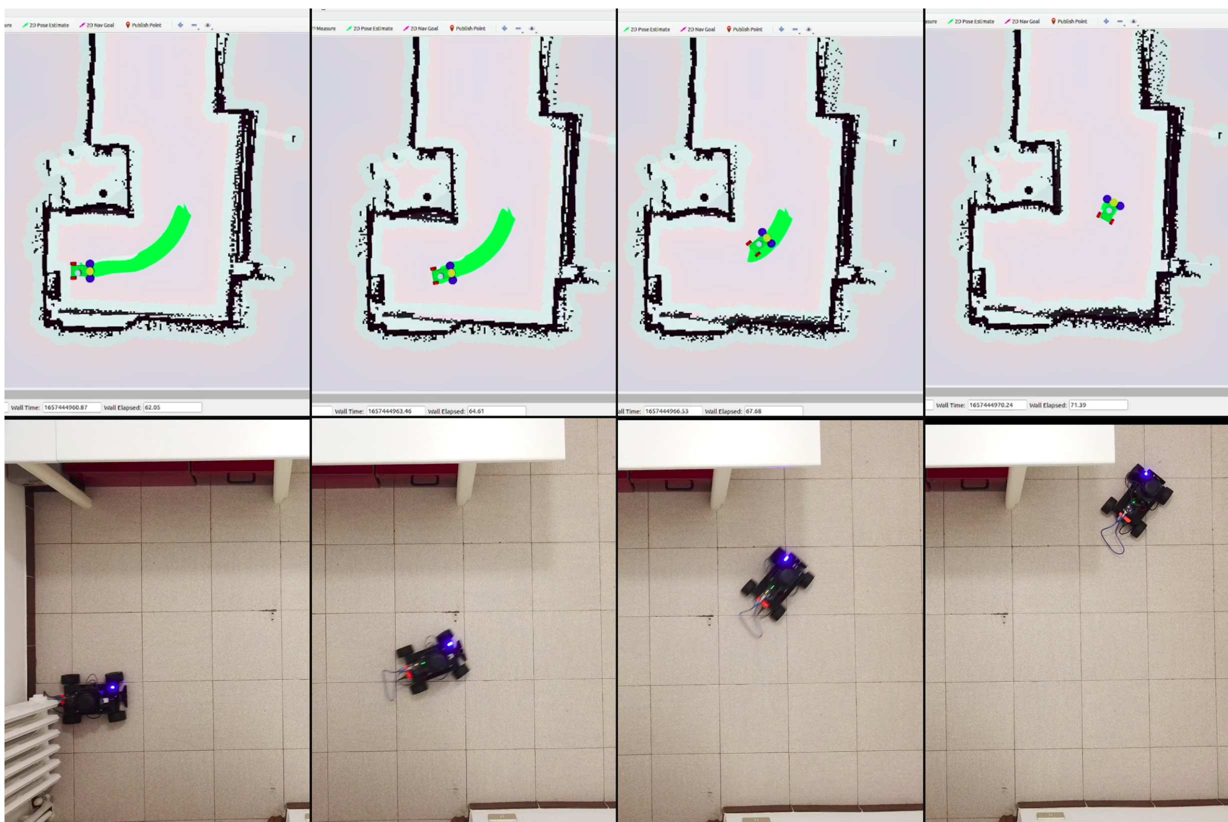


Figure 7.5: The planned path on the real map

Chapter 8

Conclusions and Future Works

8.1 Conclusions

A Model Predictive Control algorithm for path planning and control of a car-a-like robot has been studied and implemented in this thesis. A general overview of the various MPC controllers has been done in particular regarding the Non Linear one used in this thesis.

It has been shown how the algorithm behaved in simulation tests where, all the maps and environments, achieved good results. The Gazebo simulator proved to be an excellent starting point for testing the MPC algorithm as it integrates very well with ROS. With these simulations, a clear idea of the behavior that can be obtained in reality was derived.

Subsequently, it has been explained in detail how the real setup was developed and how the algorithm was implemented in the Raspberry on an RC car.

Finally, the implementation of the algorithm on the hardware, the real simulations and their performances were shown. The results of the real simulation can be considered good but not at the same level as those obtained in the PC simulations. In the reality, the algorithm needs some optimal speed to perform at its best but unfortunately the car has a rather high minimum speed due to the weight achieved by adding the various elements to it.

8.2 Future Works

The first possible improvement in this work could be to replace the motor with a more powerful one to solve the minimum speed problem. Replacing the DC motor also enables a more reactive system that better responds to the speed references set from the controller. Achieving improved overall behavior and the ability to better parameterize the controller itself are also expected changes.

Another improvement could be the usage of a camera. This would allow the vehicle to carry out more complex paths by adding more information about navigation to the MPC. Future researches could aim to improve the motion planning algorithm and to deal with more complex driving environments. These improvements will be possible by considering dynamic obstacles. In the configuration of this thesis these obstacles are actually experimentally handled by the local planner.

Another future development could be to assemble other vehicles with the same configuration and MPC controller and having them to communicate with each other. By doing so, it is be possible to imagine a real-time network that allows the vehicles that know where the others are and to move accordingly without colliding.

Autonomous driving will definitely be a key part of our future and will completely change our concept of driving.

Bibliography

[1] A. Herrmann, W. Brenner, R. Stadler "*Economics, Autonomous Driving*", Emerald Publishing Limited, Bingley, 2021.

[2] SAE International "*Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles* $J3016_2 02104" BusinessInsider$, 2014.

[3] S. Rosenbloom, B. Tefft, T. Triplett, R. Santos. "*American driving survey: 2014 – 2015*", AAA foundation for traffic safety, 2016.

[4] M. Barth. "*Real-World CO2 Impacts of Traffic Congestion*". University of California Transportation Center University of California, 2008.

[5] C. Vianello, "*MPC Approach for AGVs Driving in the Primary Aluminum Industry*", Master's thesis, University of Padova, 2019.

[6] A. Carvalhoa, G. Schildbacha, J. Konga, F. Borrellia, "*Automated Driving: The Role of Forecasts and Uncertainty - A Control Perspective*", European Journal of Control, 2015.

[7] A. Bemporad, "*Model Predictive Control Design: New Trends and Tools*", Proceedings of the 45th IEEE Conference on Decision Control, 2006.

[8] R. Findeisen, F. Allgower, "*An Introduction to Nonlinear Model Predictive Control*", Institute for Systems Theory in Engineering, University of Stuttgart, Germany, 2002.

[9] F. Curinga, "*Autonomous racing using model predictive control*", Degree project, Information and Communication in Technology, Sweden, 2018.

[10] E. F. Camacho, C. Bordons "*Model Predicive Control*", Springer, 2000.

[11] Defence Advanced Research Projects Agency, "*DARPA Robotics Challenge*", [Online], http://www.darpa.mil/program/darpa-robotics-challenge, 2015.

[12] GitHub Repository, "*Aws Robomaker*" [Online], https://github.com/aws-robotics/aws-robomaker-small-house-world.

[13] H. Lin, *HyphaROS MiniCar (1/30 Scale MPC Racing Car)*, [Online], https://github.com/Hypha-ROS/hypharos_minicar, 2016.

[14] J. Kong, M. Pfeiffer, G. Schildbach, and F. Borrelli. *Kinematic and Dynamic Vehicle Models for Autonomous Driving Control Design*, IEEE Intelligent Vehicles Symposium, 2015

[15] A. Doucet, J. F. G. de Freitas, K. Murphy, and S. Russel, "*Rao-Blackwellized particle filtering for dynamic bayesian networks*", Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI), pp. 176–183, Stanford, 2000.

[16] "*Ubiquity Robotics*", [Online], https://learn.ubiquityrobotics.com/noetic_quick_start_connecting

[17] K.Zheng "*ROS Navigation Tuning Guide*", Article, 2016.

[18] S. Quinlan, "*Real-Time Modification of Collision-Free Paths.*" Stanford, CA, USA: Stanford University, 1995.

[19] ROS Documentation, "*HMMWV*", "*amcl*", [Online], http://wiki.ros.org/amcl

[20] C. Rosmann, W. Feiten, T. W¨osch, F. Hoffmann, and T. Bertram, "*Trajectory modification considering dynamic constraints of autonomous robots*" in 7th German Conference on Robotics, pp. 74–79, 2012.

[21] GitHub Repository, [Online], https://github.com/ne0h/hmmwv/tree/master