POLITECNICO DI TORINO

MASTER's Degree in MECHATRONIC ENGINEERING

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

MASTER's Degree Thesis

Planar target tracking in a moving multi-obstacles scenario

Supervisors

Prof. Marcello CHIABERGE

Prof. Aude BILLARD

Prof. Francesco BRAGHIN

Dr. Diego PAEZ

Candidate

GABRIELE COPPOLA

December 2020

"Success is 1% *inspiration,* 98% *perspiration, and* 2% *attention to detail."*

— Phil Dunphy

Abstract

The increasing interest in the collaboration between humans and machines has led to the development of collaborative robots, called cobots, both in manufacturing plants, to help the workers with their tasks, and in the medical sector, to improve the effectiveness of physical therapy. Most of these robots have to interact and move in unknown environments that are often crowded. Such operations are non-trivial, because the knowledge that the robot has about its surroundings is often limited; moreover, they require complex control strategies that work only in very specific conditions.

This thesis tackles this problem with an innovative approach. A controller for a differential drive robot is designed to track and follow a target in a moving multiple-obstacles environment. The control structure is composed by three layers:

1. Machine learning-based identification of obstacles and target;
2. DS-based motion planner for target tracking;
3. Model Predictive Control (MPC) low-level controller.

To continuously map points cloud data (LIDAR) to a probability distribution of the obstacles over the 2D motion space, two algorithms have been implemented: a first one using a Gaussian Mixture Model, and a second one using a K-means clustering algorithm. A DS-based motion planner, relying on the modeling of the obstacles distributions, generates feasible trajectories for reaching and following the target while avoiding obstacles. The MPC-based low-level controller is in charge of generating the control law to follow the desired trajectories.

Evaluation of the whole system architecture and control has been performed in simulation using Python and with raw data of moving people. Results show that the proposed approach is feasible for real-time control implementation, with the main limitation being the computational cost for the non-linear MPC.

From this study it is possible to conclude that the control architecture presented in this work has the following advantages:

- the implementation of Machine Learning algorithms for the obstacles detection guarantees high performances, responsiveness and stability;
- the DS-based motion planner provides a reactive and robust to perturbation behaviour with instantaneous trajectory re-planning based on the robot spatial location;

- MPC-based controller guarantees optimality of the proposed control-law for trajectory following;
- the modular structure allows for fast adaptation to different robotic systems.

The main disadvantage of this solution consists in the computational timing, because, in order to have consistent control inputs from the MPC, the prediction horizon has to be large enough, which causes the computational cost to be quite high.

Acknowledgements

First of all, I would like to devote a word of gratitude to all the members of the Learning Algorithms and Systems Laboratory (LASA), and in particular to my supervisors Prof. Aude Billard, for her guidance and to Dr. Diego Paez, who proposed me this project and helped me overcoming all the difficulties encountered. I would like to thank also Dr. Bernardo Fichera, who was always present when I needed explanations and inspired me to think out of the box and to look for unconventional solutions. A vote of thanks is dedicated also to my supervisor from Politecnico di Torino, Prof. Marcello Chiaberge who assisted me during the Erasmus and shared with me his experience; and to my supervisor from Politecnico di Milano, Prof. Francesco Braghin, who serves as co-supervisor of this thesis within program of double degree of Alta Scuola Politecnica, for his constant and useful advice and for the passion he puts in his work.

A special word of gratitude and appreciation is spent for my family, who stood by my side all my life and supported me and all my choices. In particular to my father and his experience of the bureaucracy, without which I would have not graduated. To my mother, who constantly gave me moral support and love; who always believed in me and spur me to achieve better results. To my brother, to my grandparents and to my uncle who were always proud of me and supported me (especially from a financial point of view).

A special thank to all my friends *"di giù"* Giorgia, Amanda, De Simei and Colelli.

I would like to thank also all the people that I encounter during my university career, that I am happy to call friends. Firstly the group of Electronic Engineering, with whom I shared both amazing moments and struggling periods. Then, all the Mechatronic friends that really made my days during the first Master year. A

special thank must be dedicated also to Dr. Basciani, with whom I shared the Erasmus experience.

A big thanks to all the other special friends that I met outside the University like Paola, Sara and the Beautiful Luca.

Finally, a really heartfelt thanks to my second family at Collegio Einaudi: starting from my mentor Gian Franco, who always inspired me, and my Einaudi mother Adelina. To Punzo, who is always very kind, and to Anna, for her help and wisdom. A very big world of gratitude is dedicated to the so called "Pinguini", who were always there for me and shared with me pleasures and dramas; I'll keep them in my heart forever. To conclude, a sincere thank to Tina, who helped me to survive the solitude during the quarantine and for her true friendship.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Project description

In the last years we have observed an increasing interest in the field of the collaboration between humans and machines. In particular, new kinds of cobots [1], i.e. collaborative robots, have been introduced firstly in the manufacturing plants to help the workers with their task; then in the medical sector to improve the effectiveness of the physical therapy.

Most of these robots have to interact and move in unknown environments that are often crowded. The appeal of this challenge and the possibility of profitable innovations have lead different organizations to play in advance and to work on possible solutions. The CrowdBot project [2] is an example of this push and the Qolo robot [3] is the perfect representation of a cobot that has to be used in a crowed scenario.

The project **Planar target tracking in a moving multi-obstacles scenario** was proposed by the Learning Algorithms and Systems Laboratory (LASA)at the École polytechnique fédérale de Lausanne - EPFL inside the International Mobility Program of 2019/2020.

The main problem that the Master Project aims to tackle is the **target tracking** while avoiding **multiple moving obstacles**. The non-triviality of this question requires complex control techniques: non-linear control techniques present in the literature showed interesting results in person following and static obstacle avoidance. The project wants to deal with real-world scenarios that encompass moving obstacles and all the issues related to them: they might or might not pass very close to the desired tracked target. In this situation the controlled agent has to react fast to incoming obstacles or generic perturbation from the external environment.

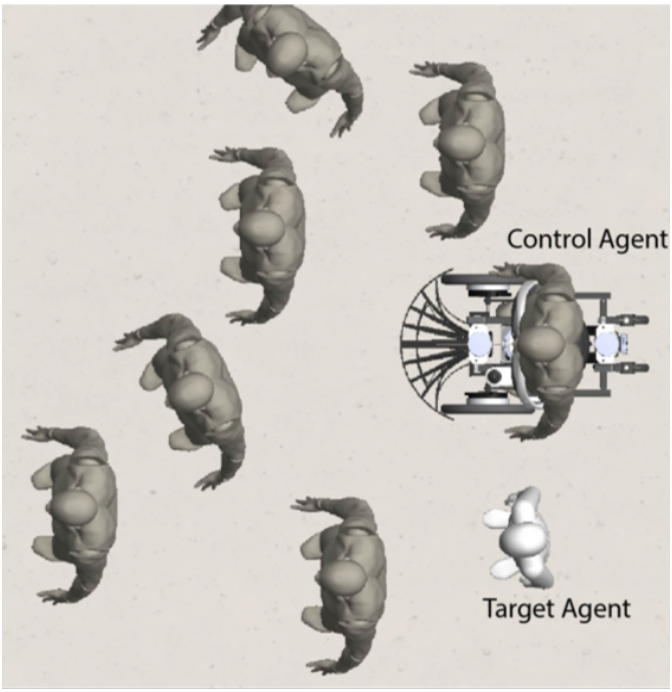


Figure 1.1: Crowded scenario

The goal of the project is the design of a controller, using machine learning algorithms, for a mobile robot platform for safe navigation in a crowded scenario while following a predefined moving target. This involves the capability of avoiding moving obstacles; in addition the robot has to be capable of following the target keeping a constant side offset and align its velocity in magnitude and direction with the target upward orientation. The controller has to deal also with the robot's constraints that limit some of its movements.

The solution proposed hereby implements different machine learning techniques and non-linear control algorithms to identify the obstacles in the environment and to produce suitable control inputs.

The use of DS-based control techniques offers a useful tool to acquire stable and robust behavior: the adoption of DS learning technique to shape the high-level robot policy is one of the fundamental pillars of this project. Deformation of DS streamlines in order to account approaching obstacles can be obtained by applying proper modulations as proposed in LASA **inserire reference**. Stability guarantees can be achieved by leveraging on a differential approach known as contraction theory.

Nonetheless, given the complexity of the scenario due the partially observability

and controllability of the control problem other approaches such as Model Predictive Control have been implemented, in order to generate the low-level robot policy: the non-linearity of the dynamical system and the constraints imposed by the robot's structure can not be tackled with the standard control algorithms.

Furthermore, machine learning techniques were used to model the obstacles and the environment, starting from raw data obtained by the robot's sensors. They are very useful to remove the outliers in the data caused by noise (implementing a noise filter) and to cluster the data-points to identify the obstacles.

Chapter 2

Work organization

In this chapter is present a brief description of the universities involved in this project; there will be also an introduction to the algorithm structure, linked to the workload distribution during the last months.

2.1 Institutions involved

The institutions involved in this Master Project are mainly three, all of them are academic organization and contributed in different way to the project development.

- **Politecnico di Torino**: as home university, it provided mainly bureaucratic support and technical knowledge. It was represented by prof. Marcello Chiaberge, who acted as main supervisor.

- **École polytechnique fédérale de Lausanne - EPFL**: the project was proposed in this university by the Learning Algorithms and Systems Laboratory (LASA). EPFL is, therefore, the host university and thanks to the supervisors prof. Aude Billard and dr. Diego Paezand to the PhD. student Bernardo Fichera, it provided cognitive resources and contributed in a significant way to develop all the technical aspects of the project. In addition, LASA provided the instruments and the places to work safely, even during the pandemic.

- **Politecnico di Milano**: it is involved thanks to the Alta Scuola Politecnica program, a double degree project that interests both the Politecnico di Torinoand the Politecnico di Milano. Prof. Francesco Braghinwas the supervisor from this university; he showed great interest in the project and gave useful suggestions and supplied technical knoledge.

2.2 Work schedule

This project started after the end of the exam session at École polytechnique fédérale de Lausanne - EPFL; the first month, i.e. February, was dedicated to the literature review. In particular the main aspects that were analyzed in this period were:

- control algorithms for target tracking in static and dynamic environments;
- control algorithm for obstacle avoidance in static and dynamic environments;
- machine learning algorithms developed at Learning Algorithms and Systems Laboratory (LASA), in particular the DS-based algorithms, both for learning a trajectory and for producing control inputs;
- reinforcement learning algorithms, used to learn a behaviour starting from a dataset obtained simulating the environments or recording in reality a desired conduct.

In the meanwhile, using MATLAB [4], some simulation environments were created, in order to become more familiar with the robot's dynamics and try out the DS-based algorithms already implemented and available on LASA website.

The activities mentioned before were conducted in the LASA's laboratory and offices, but, after the pandemic outbreak, all the work was carried in smart working from March until June, when it was possible to go back to the university.

During this period, a deep analysis of the paper **Avoidance of Convex and Concave Obstacles with Convergence ensured through Contraction** [5] was conducted and, using MATLAB, all the main steps of the algorithm were re-implemented. In such a way, it was possible to better understand the potentiality of DS-based control; moreover, different changes to the algorithm were tried in order to adapt it to a situation in which the target is not static any more, but it is moving. This first part of work focused especially on an high-level type of control, for this reason, the kinematic of the robot and its volume were often neglected, preferring a point mass idealization. In addition, at the beginning, a complete knowledge of the environment and of the obstacles was supposed.

After some months of work, it was implemented in MATLAB a LIDAR simulator, in order to be more accurate to the real implementation. Using the simulated datapoints obtained by the LIDAR, some machine learning algorithms were used to model the environment around the robot.

As said before, the work done until that moment didn't consider the kinematics of the robot. To compensate this issue, a low-level controller was studied, that had

the task to translate the high-level commands to control inputs suitable for the robot and, at the same time, had to consider the constraints caused by the robot's structure.

Different approaches were tried, from the simplest ones to the more complex. The best results were obtained using non-linear Model Predictive Control, which is able to take into account both the system's dynamics and the system's constraints.

At this point, there was enough material to try everything on the robot, for this reason in was necessary to move all the code in Python [6]. It was possible to use and to modify the code already developed by Lukas Huber [7], that deployed several optimized features that weren't implemented in MATLAB.

When it was possible to go back to the university (June 2020), some tests were done on the robot (between June and July), and several problems related to the timing of the algorithms emerged: the SLAM algorithm of the robot, the high-level controller and the low-level one were too slow and it was not possible to control the robot in a good way.

At the end of July it was necessary to use again the smart working solution, since the Mobility period ended. In August and in the first weeks of September other low-level controllers were developed, in order to respect the time constraints of the robot; the simulated results were promising, but real tests on the robot have to be done.

In September and in October the work focused again on the machine learning implementation that has to deal with the raw datapoints obtained by the LIDAR and the camera. The main methods used for this purpose were the Gaussian Mixture Model [8] and the K-means [9].

Chapter 3

QOLO robot [10] [2] [11]

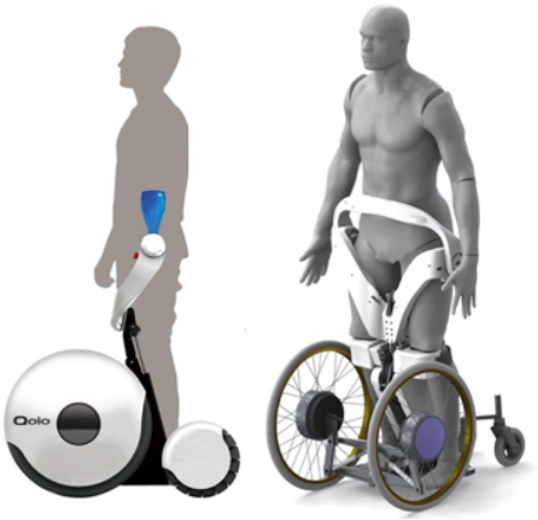


Figure 3.1: QOLO robot [2] [12]

The project revolves around the QOLO (Quality of Life with Locomotion) robot [11] [3], shown in 3.1. The robot is part of the CrowdBot project [2], a collaboration between 7 organizations from 4 European countries. The goals of this project is to combine technical knowledge with ethics, in order to obtain robots that are able to move in a crowded scenario while sensing and predicting the specific dynamics of the pedestrians around the robot.

Qolo robot is a device able to combine active powered wheels and passive exoskeleton, resulting in a wearable robot for wheelchair user. The great revolution of this robot consists of a standing mobility that does not impact on the compactness and on the portability, thanks also to its light weight.

A Qolo's user is helped by the passive exoskeleton to sit and to stand up, using easy body posture changes. Moreover, it is possible to use the robot without using the hand, while standing, thanks to its control interface [13].

The main aspects of the robot that will be used for this project are:

1. Sensing - how to perceive the obstacles around the robot;
2. Movement - the dynamic model of the robot;
3. Control and interface - the software part of the system.

3.1 Sensing

In a crowded scenario, a crucial aspect for a good behaviour of the whole system is how the robot sense and perceive the environment around it. Qolo uses different instruments that allow it to have a good knowledge of its surroundings in real-time. First of all, it uses a front and a rear lidar to sense the environment at medium distances. A lidar is a device that illuminates the space with laser light and measures the reflection, doing so it is able to compute the distances of the objects around it. Using different wavelengths and return times of the laser, it is possible to obtain a 3D representation of the surroundings. An example of the device used on the robot is shown in Figure 3.2.

Figure 3.2: Lidar [14]

In addition to the lidars, the robot has three depth cameras: they are *Intel RealSense™ Depth Camera D435* [15], shown in Figure 3.3. This kind of camera is

an optimal solution because its cost is relatively low and with a single device it is possible to measure near distances and to record.

Combining the lidars and the depth cameras allows the robot to have information both on the depth and on the kind of obstacles around it. Using these data, an already implemented algorithm, is able to distinguish the people near the robot from other obstacles. Unfortunately this algorithm can work only at 10Hz and it often produces false positives: the algorithm sees people even if they are not present in reality. This Master project wants to tackle also this problem, using machine learning algorithms able to filter the noise and produce more reliable information regarding the obstacles and the environment.

Figure 3.3: Intel RealSense™ Depth Camera D435i [16]

3.2 Movement

Qolo's movement structure is composed of two differential motored wheels, that propels the robot and allow the movement; and two castor wheels that are necessary for stability reasons. This typology of drive mechanism can be modeled as a **unicycle** and it is called **differential drive**: the motored wheels can work independently one from the other, but they have the same stearing angle; this causes the system to be **nonholonomic**.

3.2.1 Holonomic and Nonholonomic system

The configuration of a robot can be described by a vector $\boldsymbol{q} \in \mathbb{R}^n$, called generalized coordinates. Assuming that the set of all possible configurations, called

configuration space $\mathbb{C}$, coincides with $\mathbb{R}^n$, the robot motion can be described by $\boldsymbol{q}(t)$, that can be subject to different constraints.

In literature, we distinguish the constraints in **holonomic** and **nonholonomic**. The first typology of constraints causes the reduction of the accessible configurations from $\mathbb{C}^n$ to $\mathbb{C}^{n-k}$, where k is the number of holonomic constraints. This implies that k generalized coordinates become a function of the other $n-k$. The holonomic constraints are generally represented as:

$$f_i(\boldsymbol{q}, t) = 0, \quad with\ i \in [1, k] \tag{3.1}$$

A system is **nonholonomic** when its state depends on the path taken in order to achieve it [17]. Differently from the holonomic constraints, these do not imply a loss in the accessibility of $\mathbb{C}$, but they constrain the velocities to a subspace of dimension $n - k$, where k is the number of nonholonomic constraints.

3.2.2 Unicycle Model

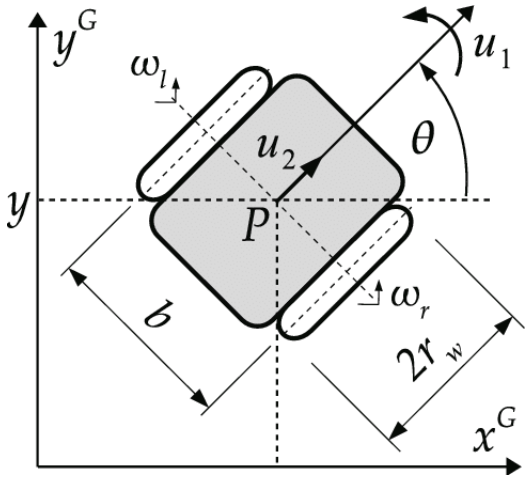


Figure 3.4: Unicycle model [18]

The Qolo robot's model on an horizontal plane is called **Unicycle Model**. Assuming the generalized coordinates vector $\boldsymbol{q} = [x, y, \theta]^T$, the robot's kinematic can be described by the non-linear system:

$$\begin{cases} \dot{x} = \frac{\omega_R + \omega_L}{2} \cdot r \cdot cos(\theta) \\ \dot{y} = \frac{\omega_R + \omega_L}{2} \cdot r \cdot sin(\theta) \\ \dot{\theta} = \frac{\omega_R - \omega_L}{b} \cdot r \end{cases} \quad \Rightarrow \quad \begin{cases} \dot{x} = v \cdot cos(\theta) \\ \dot{y} = v \cdot sin(\theta) \\ \dot{\theta} = \omega \end{cases} \tag{3.2}$$

Where x, y and θ are the position and the orientation of the robot in a fixed reference frame; r is the radius of the wheels, d is the distance between the two

wheels and ω_R and ω_L are the angular velocity of the right and left wheels.
$v = \frac{\omega_R + \omega_L}{2} \cdot r$ and $\omega = \frac{\omega_R - \omega_L}{b}$ (in Figure 3.4 u_2 and u_1 respectively) are the the control inputs for the system.
As a matter of fact it is possible to rewrite the dynamics of the system as:

$$\dot{\boldsymbol{q}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = G(\boldsymbol{q}) \cdot u = \begin{bmatrix} cos(\theta) & 0 \\ sin(\theta) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{3.3}$$

The nonholonomicity of this model is caused by the pure rolling constraint: theoretically, in normal conditions, the wheels of the robot should roll without slip and, as a consequence, the velocity component along the normal direction to the wheels plane is always non-zero. This constraint means that the robot can not move side to side, therefore appropriate control strategies have to be implemented in order to drive the robot to the desired position.

3.3 Control and interface

The robot is controlled using ROS, a robotics middleware [19] [20], that allows the robot and software designer to easily interface the different hardware parts with the software unit, dedicated to the control.

As a consequence, the control algorithms have been developed in Python [6] [21], in order to guarantee the compatibility with the existing control structure. Nevertheless, a part of the control algorithms has been firstly implemented in MATLAB [4][22] and then converted in Python, in order to understand better the algorithm structure and because of the easiness of the debugging.

Chapter 4

Algorithm Structure

The structure of the algorithm can be divided into three main components, that combined all together provide a complete control algorithm for the target tracking and the obstacle detection and avoidance.

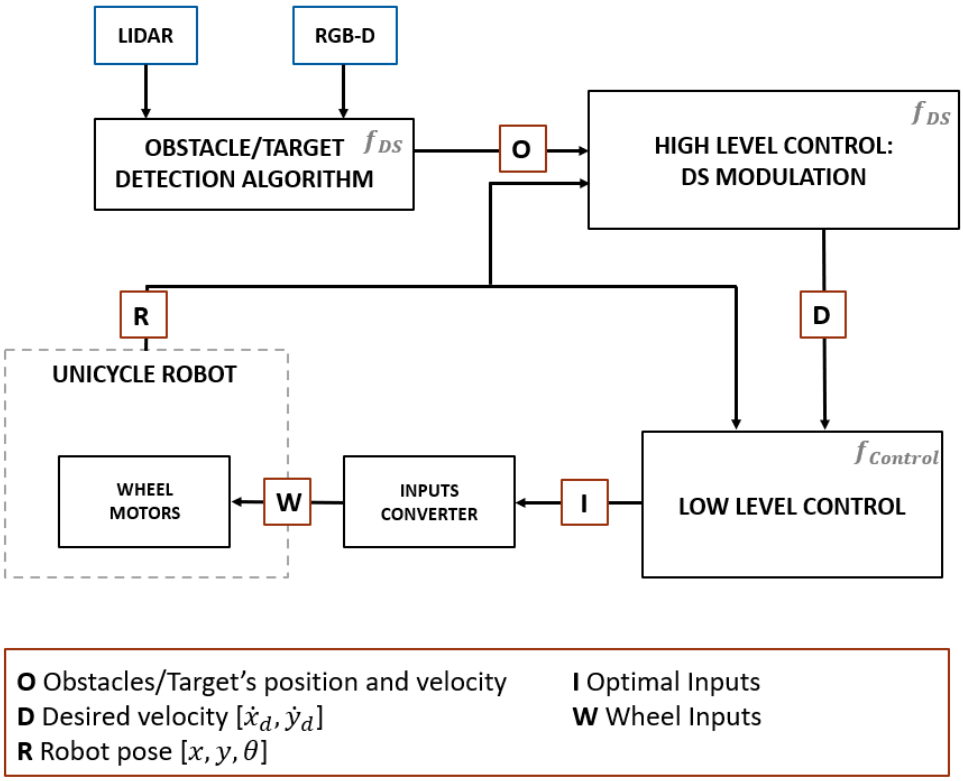


Figure 4.1: Algorithm Scheme

The first component is the **obstacles and target detection algorithm**; it has to analyse the data obtained by the LIDAR and the camera and produce as output the position of the obstacles, with their velocity and the volume occupied, and the position of the target with its velocity and volume. The latest task is the most difficult one, since it is necessary to extract from the information available a sufficiently stable knowledge of the target, otherwise the control inputs produced would be irregular and unpleasant.

The methods adopted for this part are mainly machine learning algorithms and will be explained in detail in Chapter 7 with some anticipations in Chapter 5.

The **High Level Control** is the part of algorithm that takes as inputs the position and the orientation of the robot and all the possible information about the environment and, using a dynamical system (DS) approach produces locally the desired velocity vector that the mass point robot should have in order to reach the target and avoid the obstacles.

Depending on the implementation of the three algorithm together, the output of this subsystem can be the desired velocity or directly the desired next position, integrating the system for one time-step.

In Chapter 5 there will be an exhaustive description of the methods used and of the trials and errors.

The last component is the **Low Level Control** and has the purpose of converting the results obtained with the High Level controller into control inputs for the robot. It has to deal with the kinematic constraints of the unicycle, with the limitations imposed by the actuators on the robot (linear velocity and angular velocity) and with the limits dictated by the acceleration and the jerking that the human body can withstand.

Many typologies of controller have been tried, but just few of them have shown good results; the Chapter 6 will display the different attempts.

To conclude, it is worth to mention that the sine qua non condition for the combination of the three sub-algorithms to work properly is that the timing constraints are respected: the algorithms have to run at a frequency that is fast enough to compute, at every time-step, all the necessary outputs. If this is not respected, for example, if the Low Level Control is too slow, the control inputs of the robot would not be up to date and would control a situation that is not the real one, producing a behaviour that could be potentially dangerous for the user and for the people around him or her.

Algorithm 1 Algorithm Structure

▷ **Execution**
while Robot is working **do**
    ▷ **Obstacle and Target Detection**
    ▷ Get data from LIDARS
    $Raw\_data \leftarrow$ LIDARs
    ▷ Get information about robot's position in the environment
    $[x, y, \theta]^T \leftarrow$ SLAM algorithm
    ▷ Run the algorithm for the people detection (already existing)
    $People \leftarrow$ People detection
    ▷ Use Raw_data to model the obstacles
    $[Obstacle, \dot{Obstacle}] \leftarrow MLalg(Raw\_data)$

        ▷ $MLalg()$ identifies a Machine Learning algorithm developed to find the obstacle position, shape and velocity

    ▷ Use $People$ to identify the target
    $Target \leftarrow TargetDetection(Obstacle, People)$

        ▷ $TargetDetection()$ identifies an algorithm than choose the target among the obstacles

    ▷ **High Level Controller**
    ▷ Use the information on $Obstacle$, $\dot{Obstacle}$ to model the environment
    ▷ Use the information on $Target$ and $\dot{Target}$ to define the attractor
    ▷ Apply the modulating DS algorithm to obtain the desired velocity vector of the robot
    $\dot{robot}_{des} \leftarrow Mod\_DS()$

        ▷ $Mod\_DS()$ identifies the modulating algorithm

    ▷ **Low Level Controller**
    ▷ Use the information on $\dot{robot}_{des}$ to compute the control inputs
    ▷ Optimize the control inputs to be compliant to the boundaries
    $Inputs \leftarrow InputGen(\dot{robot}_{des})$

        ▷ $InputGen()$ identifies an algorithm that generates the inputs, taking into account the robot's contraints

    ▷ Apply the control inputs to the robot
end while

Chapter 5

High Level Control

5.1 Dynamical System based control

The mathematical definition of a dynamical system is straightforward: it is a system described by a function that relates the time dependence of a point to a geometrical space. [23]

This definition can be adapted to the physical world describing "a particle or ensemble of particles whose state varies over time and thus obeys differential equations involving time derivatives." [24]

The main characteristic of a dynamical system, at a time t, is its state $\xi(t)$, a vector of real numbers with which it is possible to identify a point in a geometrical manifold called state space. The differential equations define an evolution rule that traces the future states, starting from a current state. There are two typologies of dynamical systems' functions:

- Deterministic functions - the evolution from a certain state is completely known and only one state can be the next one;
- Stochastic functions - the evolution is characterized by randomness, as a consequence, probabilistic events determine the future states.

The knowledge of the function allow us to predict the behaviour of the system using an analytical solution or, in the more complex cases, through numerical integration.

The general formula of a differential equation for a dynamical system is the Equation 5.1, in which the time variance is explicit and the state evolution might depend also on other variables identified with u.

If the system is autonomous and time invariant the formula that describes such system is illustrated in 5.2. This typology of dynamical system is very common in

control theory and it will be fully exploited in this project.

$$\dot{\xi} = f(\xi, u, t) \tag{5.1}$$

$$\dot{\xi} = f(\xi) \tag{5.2}$$

A dynamical system (DS) based control differs from the classical path planning in the planning itself: the DS-based presents a control law that is in closed form and, as a consequence, does not need a re-planning.

That allows the system to be very reactive, hence, this kind of controller presents optimal features for an obstacle avoidance control algorithm, because in such a situation the responsiveness of the robot is a must

5.2 Avoidance of Convex and Concave Obstacles

The first part of the work of the project consisted in the understanding, reproduction and modification of the control algorithm for the avoidance of multiple concave obstacles. The paper used to set the algorithm is "Avoidance of Convex and Concave Obstacles with Convergence ensured through Contraction" [5] and shows a method that guarantees asymptotic stability.

In this section a detailed explanation of the original algorithm will be provided, while in the following ones there will be the different tests done to reproduce.

First of all, it is necessary to define a state $\xi \in \mathbb{R}^d$, where d is the dimension of the state; in our case $d = 2$ because the model taken into account is the unicycle one, so working in a plane is sufficient; but this method can be applied also for three-dimensional spaces. Then, an attractor $\xi_a \in \mathbb{R}^d$ has to be defined. Since it represents the desired final state, a dynamic system is needed to impose a control law; for this purpose, an autonomous, time invariant, linear system has been chosen, of the form written in Equation 5.3.

$$\dot{\xi} = \boldsymbol{f}(\xi) = -(\xi - \xi_a) \tag{5.3}$$

With the dynamics above mentioned the point mass is not able to avoid obstacles that are in its trajectory. In order to solve this problem, a modulation matrix $M(\xi)$ is introduced:

$$\dot{\xi} = M(\xi) \cdot \boldsymbol{f}(\xi) \tag{5.4}$$

This matrix does not change the existing minimum produced by the attractor and, as long as the matrix has full rank, no other extrema (maxima or minima)

are introduced. $M(\xi)$ is obtained multiplying specific matrices:

$$M(\xi) = E(\xi)D(\xi)E(\xi)^{-1} \tag{5.5}$$

Where $E(\xi)$ is a basis matrix composed by:

$$E(\xi) = [\boldsymbol{r}(\xi), \boldsymbol{e}_1(\xi), ..., \boldsymbol{e}_{d-1}(\xi)] \tag{5.6}$$

remembering that d is the dimension of the state and, hence, of the system.

$$\boldsymbol{r}(\xi) = \frac{\xi - \xi^r}{||\xi - \xi^r||} \tag{5.7}$$

Where ξ^r is a reference point inside the obstacle. e_i with $i \in [1, d-1]$ are tangents to the surface of the obstacle, in the point where the border of the obstacle meets the line that joins ξ with ξ^r.

$D(\xi)$ is the associated eigenvalue matrix that is able to stretch and compress the dynamics along the directions $e - r$-system.

$$D(\xi) = diag(\lambda_r(\xi), \lambda_{e_1}(\xi), ..., \lambda_{e_{d-1}}(\xi)) \tag{5.8}$$

The eigenvalues λ determine the amount of stretching in each direction. A necessary condition that ensures that the point mass does not penetrate the obstacle is to have $\lambda_r = 0$ on the edge of the obstacle; in this way we cancel the flow in the direction of the obstacle. While, the effect of the eigenvalues of the tangent directions is to increase the speed along that directions.

In order to ensure that the magnitude of the velocity is preserved in certain direction we want that:

$$0 \leq \lambda_r(\xi) \leq 1 \quad and \quad \lambda_e(\xi) \geq 1 \tag{5.9}$$

A possible solution for these eigenvalues is the following:

$$\lambda_r(\xi) = 1 - \frac{1}{\Gamma(\xi)} \quad and \quad \lambda_e(\xi) = 1 + \frac{1}{\Gamma(\xi)} \tag{5.10}$$

where $\Gamma(\xi)$ is a distance function defined as

$$\Gamma(\xi) = \left(\frac{||\xi - \xi^r||}{R(\xi)}\right)^{2h} \tag{5.11}$$

With $R(\xi)$ distance from the reference point ξ^r within the obstacle to the surface of the obstacle itself, in direction $r(\xi)$ and $h \in \mathbb{N}^+$.
In this way $\Gamma(\xi) > 1$ for all the points outside the obstacles and $\Gamma(\xi) = 1$ for all

the points on the boundary of the obstacle.

$E(\xi)$ has to be invertible, which means it has to have full rank; this condition is meet if the reference direction of $\boldsymbol{r}(\xi)$ can not be written as a linear combination of the tangent directions $\boldsymbol{e}_i(\xi)$. In order to obtain this situation, the reference point ξ^r has to be inside a **convex** obstacle. Depending on the position of the point, different results might be obtained, as shown in Figure 5.1

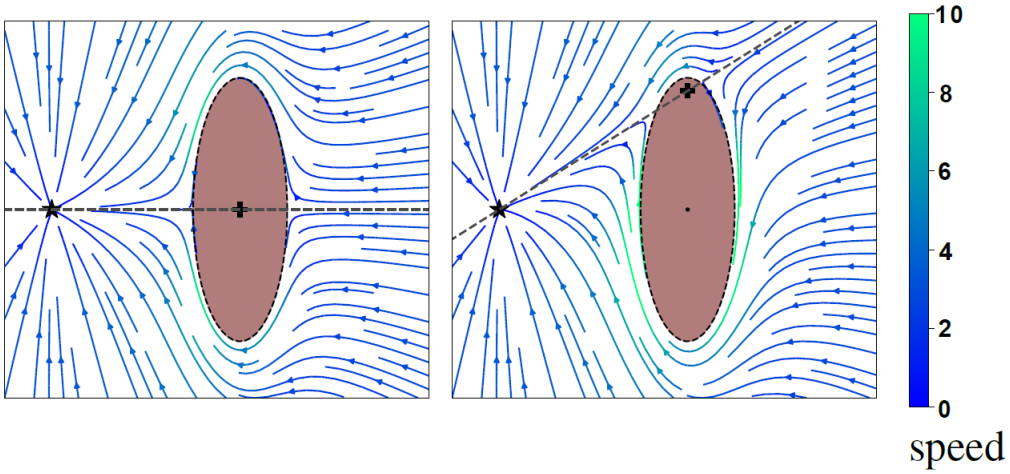


Figure 5.1: LEFT: Reference in the Center
Right: Reference to the Top
[5]

However, having only convex obstacles can be very limiting in real life scenarios; a smart way to solve this issue is to use the so called **star-shaped** obstacles. They are basically obstacles composed by two or more convex obstacles that share the same reference point.

If an obstacle in moving, the modulation can be performed in the reference frame of the obstacle and then it can be brought back in the inertial frame. Considering the linear velocity $\dot{\xi}^{L,o}$ and an angular velocity $\dot{\xi}^{R,o}$ of the obstacle with respect to its center point, the relative position of the point mass with respect to the obstacle is:

$$\tilde{\xi} = \xi - \xi^c \tag{5.12}$$

The modulation in this condition is

$$\dot{\xi} = M(\xi)(\boldsymbol{f}(\xi) - \dot{\tilde{\xi}}) + \dot{\tilde{\xi}} \tag{5.13}$$

with

$$\dot{\tilde{\xi}} = \dot{\xi}^{L,o} + \dot{\xi}^{R,o} \times \tilde{\xi} \tag{5.14}$$

$\times$ is the cross product, so a fast way to compute $\dot{\xi}^{R,o} \times \tilde{\xi}$ in the 2D case is to multiply the angular velocity $\dot{\xi}^{R,o}$ with $[\tilde{\xi}(2), -\tilde{\xi}(1)]^T$, where $\tilde{\xi}(1)$ is the first component of the vector $\tilde{\xi}$ and $\tilde{\xi}(2)$ the second one.

When multiple obstacles are present in the scene, the modulation approach can not be used in a straightforward manner: some changes in the algorithm are needed. First of all, it is necessary to compute the modulated DS for each obstacle $\dot{\xi}^o$ with $o \in [1, N^o]$ and N^o number of obstacles. Every modulated DS can be decomposed in magnitude and direction:

$$\dot{\xi}^o = ||\dot{\xi}^o|| \boldsymbol{n}^{\dot{\xi}^o} \tag{5.15}$$

We will use these components separately to compute the total modulated DS $\dot{\bar{\xi}}$ as a weighted mean, using the weights w^o. In order to have a balanced effect of the obstacles and to avoid that the effect of an obstacle is perceived on the border of the other ones, the weights are constrained so that:

$$\sum_{k=1}^{N^o} w^o(\xi) = 1 \tag{5.16}$$

The weight $w^o(\xi)$ is chosen to be inversely proportional to the distance measure $\Gamma^o(\xi) - 1$, using the formula in Equation 5.17

$$w^o(\xi) = \frac{\prod_{i \neq o}^{N^o} (\Gamma^i(\xi) - 1)}{\sum_{k=1}^{N^o} \prod_{i \neq k}^{N^o} (\Gamma^i(\xi) - 1)} \tag{5.17}$$

The magnitude $||\dot{\bar{\xi}}||$ is computed using the formula:

$$||\dot{\bar{\xi}}|| = \sum_{o=1}^{N^o} w^o ||\dot{\xi}^o|| \tag{5.18}$$

As for the direction $\bar{\boldsymbol{n}}(\xi)$, it is computed as the change in direction from the original DS, compared to the versor $\boldsymbol{n}^f(\xi)$ that represents the the original DS' direction.

The function

$$\boldsymbol{\kappa}(\dot{\xi}^o, \xi) = arccos(\hat{\boldsymbol{n}}_1^{\dot{\xi}^o}) \frac{[\hat{\boldsymbol{n}}_2^{\dot{\xi}^o}, ..., \hat{\boldsymbol{n}}_d^{\dot{\xi}^o}]'}{\sum_{i=2}^{d} \hat{\boldsymbol{n}}_i^{\dot{\xi}^o}} \tag{5.19}$$

projects the modulated DS of every obstacle onto a (d-1)-dimensional hyper-sphere, whish ad radius equal to π.

$$\hat{\boldsymbol{n}}_i^{\dot{\xi}^o} = R_f^T \boldsymbol{n}_i^{\dot{\xi}^o} \tag{5.20}$$

With $R_f = [n^f(\xi), e_1^f(\xi), ..., e_{d-1}^f(\xi)]$ and $e_i^f(\xi)$ chosen to form an orthonormal basis.
Since we are working in a 2D environment, $d = 2$ and, as consequence

$$\boldsymbol{\kappa}(\dot{\xi}^o, p) = arccos(\hat{\boldsymbol{n}}_1^{\dot{\xi}^o}) sign(\hat{\boldsymbol{n}}_2^{\dot{\xi}^o}) \tag{5.21}$$

Evaluating the weighted mean in the κ-space we have

$$\bar{\boldsymbol{\kappa}}(\xi) = \sum_{o=1}^{N^o} w^o(\xi) \boldsymbol{\kappa}^o(\dot{\xi}, \xi) \tag{5.22}$$

Now it is necessary to bring back in the original space the modulated DS $\dot{\bar{\xi}}$, using the formula

$$\bar{\boldsymbol{n}}(\xi) = R_f(\xi) \left[cos(||\bar{\boldsymbol{\kappa}}(\xi)||), sin(||\bar{\boldsymbol{\kappa}}(\xi)||) \cdot sign(\bar{\boldsymbol{\kappa}}(\xi)) \right]' \tag{5.23}$$

Having both the new direction $\bar{\boldsymbol{n}}(\xi)$ and the magnitude $\dot{\bar{\xi}}$, the final velocity is

$$\dot{\bar{\xi}} = \bar{\boldsymbol{n}}(\xi) \cdot ||\dot{\bar{\xi}}|| \tag{5.24}$$

A possible idea to enforce the attraction to the reference point, could be to consider $N^o + 1$ obstacles: the additional obstacle is the attractor, but in this case we do not modulate the DS ($M^s(\xi) = I$) and the consequence is that the final velocity $\dot{\bar{\xi}}$ tends to be more aligned with the original DS direction $\boldsymbol{n}^f(\xi)$.
This may cause some problem if the target goes on an obstacle, but we will analyse this case in the next sections.

Another problem that emerges from this algorithm is that the point mass is subject to the influence of all the obstacle, also the ones that are unlikely to collide with it because they are quite far from the point. A plausible solution could be to take into account only the the obstacles that are in a circumference centered in the point mass and with a radius that can be defined a priori or that can depend on the dimension of all the obstacles.

5.3 Dynamical system analysis

As illustrated in the previous sections, the first thing to do to understand the modulated DS, is to analyse the DS without modulation. The Equation 5.3 describes a dynamical system which is attracted by the point ξ_a. A more general formula is:

$$\dot{\xi} = \boldsymbol{f}(\xi) = -A \cdot (\xi - \xi_a) \tag{5.25}$$

With $A > 0$ to guarantee the attraction towards ξ_a. It is obtained starting from the definition of potential energy U:

$$U = \frac{1}{2}(\xi - \xi_a)^T A(\xi - \xi_a) \tag{5.26}$$

Computing the gradient of the potential energy U, we obtain $\boldsymbol{f}(\xi)$:

$$\boldsymbol{f}(\xi) = \dot{\xi} = -\nabla \cdot U \tag{5.27}$$

Some graphical examples can be obtained, firstly considering $A = 1$ with $\xi_a = [0,0]^T$ in Figure 5.2 and then $\xi_a = [3,3]^T$ in Figure 5.3.

Other examples can be obtained changing the value of A; for instance, using $\xi_a = [0,0]^T$, we can show the different behaviour with $A = 3$ in Figure 5.4 (a) and $A = 10$ in Figure 5.4 (b).

What immediately stands out is the possibility to change easily the attractor point ξ_a and the attraction strength varying A: higher is the value of A, higher will be the attractive force. Moreover, this typology of dynamical system present many advantages because of its time invariance and linearity.

5.4 Dynamical System with moving attractor

Using the formula in the Equation 5.25, we describe the dynamic of a mass point which is drawn to a static attractor. Since this project has the goal to follow a moving attractor, a new DS law has to be designed.

Assuming to know the velocity vector of the attractor, identified as $\dot{\xi}_a$, a potential function that guarantees the stability of the system is:

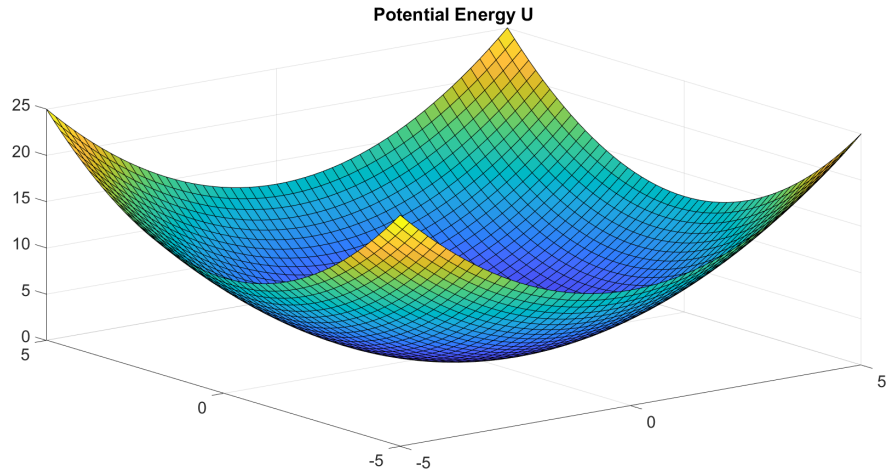
$$U = \frac{1}{2}(\xi - \xi_a)^T A(\xi - \xi_a) - (\xi - \xi_a)^T \cdot \dot{\xi}_a \tag{5.28}$$

Applying the gradient, the new dynamic become:

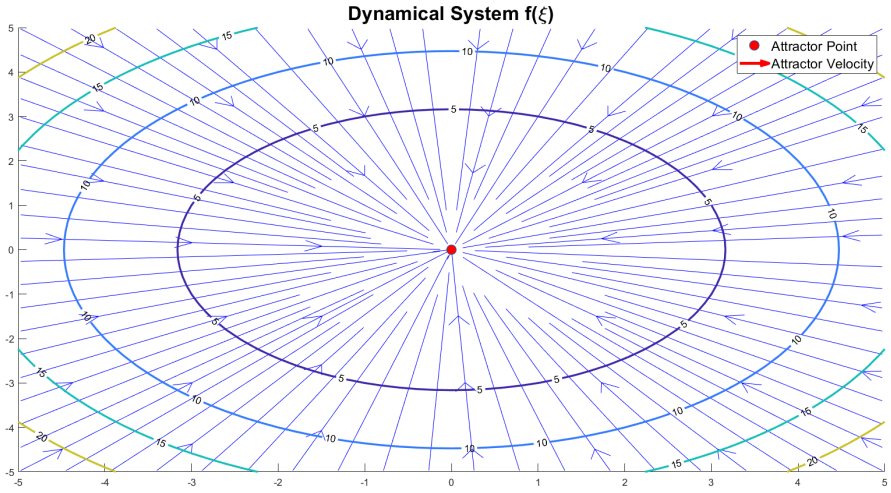
$$\boldsymbol{f}(\xi) = \dot{\xi} = -\nabla \cdot U = -A \cdot (\xi - \xi_a) + \dot{\xi}_a \tag{5.29}$$

Setting the attractor as $\xi_a = [0,0]^T$, $A = 1$ and the velocity of the attractor as $\dot{\xi}_a = [1,0]^T$, the result obtained is shown in figure 5.5.

Even if the attractor is in the origin, the state of the mass point tends to go to another point, which is the minimum of the potential function U. This behaviour is very useful for our purpose because we take into account the dynamic of the attractor, hence the mass point predicts the future behaviour (or at least the expected one) of the attractor and compensate for it, going towards the future positions of the attractor.



(a) Potential Energy



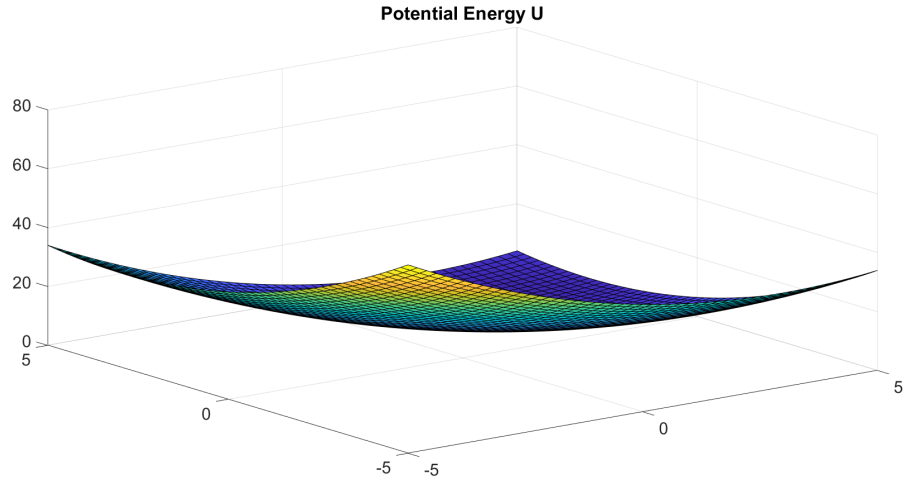
(b) Dynamical System

Figure 5.2: DS example 1: $A = 1$ and $\xi_a = [0,0]^T$

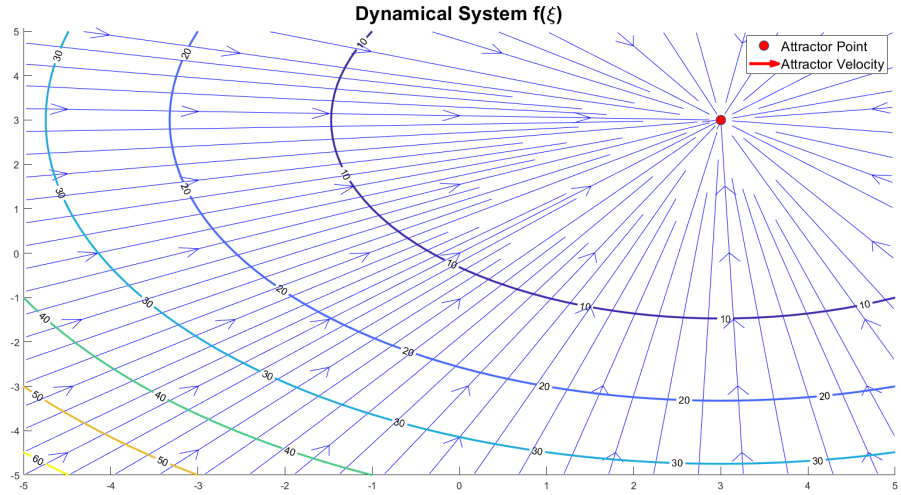
5.5 Modulated DS - Single fixed obstacle

The new modulation in Equation 5.29 differs from the one presented in the paper [5] because it has also the velocity of the attractor. In this chapter the modulated DS will be implemented with a single fixed obstacle, in order to acquire familiarity with the modulation algorithm and to understand if a moving target could produce problems in the implementation.

As first thing, we have to identify the typologies of obstacles that we can deal



(a) Potential Energy



(b) Dynamical System

Figure 5.3: DS example 2: $A = 1$ and $\xi_a = [3,3]^T$

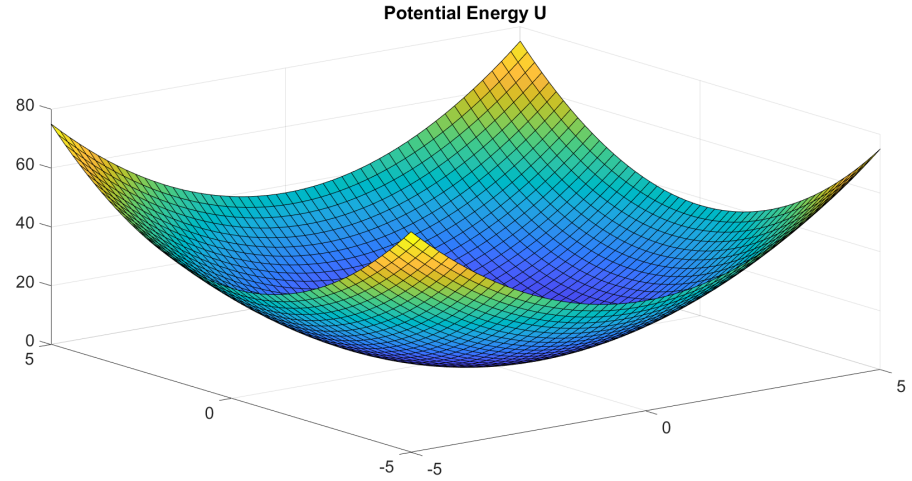
with: as said in section 5.2, the obstacle has to be convex, so the easiest way to begin is to use ellipsoid-shaped obstacles. An ellipse can be totally identified knowing the semi-minor axis r_y and the semi-major axis r_x, the center C and the tilted angle with respect to the abscissa axis θ. In this section we assume a complete knowledge of the obstacle.

Then, the modulating algorithm has been reproduced:

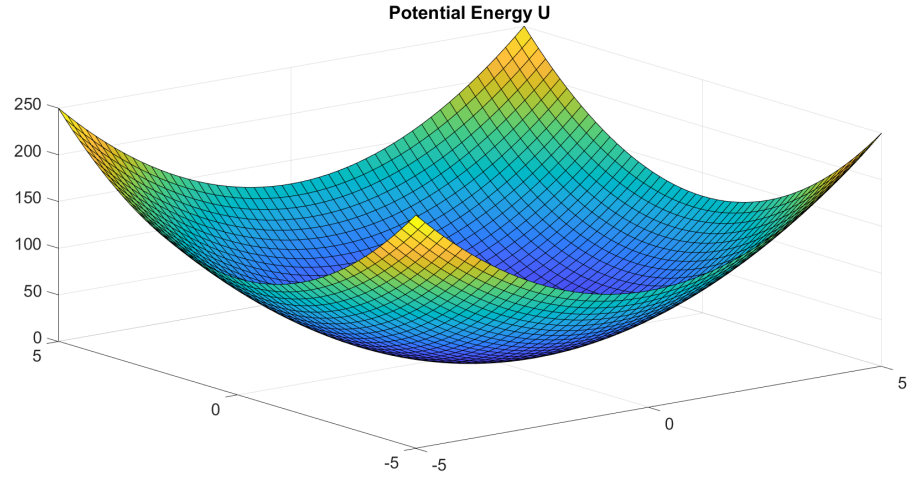
Algorithm 2 Modulation algorithm M

procedure M($\xi, r_x, r_y, C, \theta, \xi_r$)
    ▷ ξ is the state where we compute the modulation
    ▷ r_x is the semi-major axis
    ▷ r_y is the semi-minor axis
    ▷ C is the center of the ellipse
    ▷ θ is the tilted angle of the ellipse
    ▷ ξ_r is the reference point
    ▷ Execution
    ▷ ξ_r check
    if $\xi_r \notin Ellipse$ **then**
        $\xi_r \leftarrow C$ ▷ If ξ_r is outside the ellipse, it is set in the center
    end if
    $R_- \leftarrow Rot(-\theta)$ ▷ Rotation matrix with $-\theta$
    $R_+ \leftarrow Rot(-\theta)$ ▷ Rotation matrix with θ
    $\xi^0 \leftarrow R_-(\xi - C) + C$ ▷ ξ transformation to the ellipse reference frame
    $\xi_r^0 \leftarrow R_-(\xi_r - C) + C$ ▷ ξ_r transformation to the ellipse reference frame
    ▷ Intersection between ellipse and line linking ξ^0 to ξ_r^0
    if $\xi_r^0(1) = \xi(1)^0$ **then**
        $x\_edge_1 \leftarrow \xi^0(1)$
        $x\_edge_2 \leftarrow \xi^0(1)$
    else
        ▷ Find the coefficients of the line $y = mx + q$
        $m \leftarrow \frac{\xi^0(2) - \xi_r^0(2)}{\xi^0(1) - \xi_r^0(1)}$
        $q \leftarrow \xi^0(2) - m \cdot \xi^0(2)$
        ▷ Intersect line and ellipse, obtaining an equation of the form $ax^2 + bx + c = 0$
        $a \leftarrow \frac{m^2 + 1/r_x^2}{r_y^2}$
        $b \leftarrow 2(\frac{-C(1)}{r_x^2} + m\frac{q - C(2)}{r_y^2})$
        $c \leftarrow \frac{C(1)^2}{r_x^2} + \frac{q^2 - 2qC(2) + C(2)^2}{r_y^2} - 1$

▷ Find the roots
$x\_edge \leftarrow roots([a, b, c])$
$x\_edge_1 \leftarrow x\_edge(1)$
$x\_edge_2 \leftarrow x\_edge(2)$
end if
▷ Compute the axis-y coordinates
$y\_edge11 \leftarrow C(2) + r_y\sqrt{1 - (\frac{x\_edge_1 - C(1)}{r_x})^2}$
$y\_edge12 \leftarrow C(2) - r_y\sqrt{1 - (\frac{x\_edge_1 - C(1)}{r_x})^2}$
$y\_edge21 \leftarrow C(2) + r_y\sqrt{1 - (\frac{x\_edge_2 - C(1)}{r_x})^2}$
$y\_edge22 \leftarrow C(2) - r_y\sqrt{1 - (\frac{x\_edge_2 - C(1)}{r_x})^2}$
▷ Chose the right $edge\_point^0$ comparing each direction with the one that links ξ^0 to ξ_r^0
▷ Translate the ellipse in the origin
$edge\_point\_centered \leftarrow edge\_point^0 - C$
▷ Find the vector $\boldsymbol{e^0}$ orthogonal to the ellipse in $edge\_point\_centered$
▷ Compute $\boldsymbol{e}$ and $edge\_point$ in the original frame
$\boldsymbol{e} \leftarrow R_+\boldsymbol{e^0}$
$edge\_point \leftarrow R_+ edge\_point\_centered + C$
▷ Compute $\boldsymbol{r}$, direction that links ξ to ξ_r
▷ Compute the modulation matrix
▷ Compute matrix E
$E \leftarrow [\boldsymbol{r}, \boldsymbol{e}]$
▷ Compute matrix D
$d \leftarrow ||edge\_point - \xi_r)||$ ▷ Denominator of the Γ function
$h \leftarrow 1$ ▷ Exponential of the Γ function
$Gamma \leftarrow (\frac{||\xi - \xi_r||}{d})^{2h}$
▷ Compute D eigenvalues
$\lambda_r \leftarrow 1 - \frac{1}{Gamma}$
$\lambda_e \leftarrow 1 + \frac{1}{Gamma}$
$D \leftarrow diag([\lambda_r, \lambda_e])$
$M \leftarrow EDE^{-1}$
end procedure



(a) Example 3



(b) Example 4

Figure 5.4: Potential Energy varying A

The output of the Algorithm 2 is directly the modulation matrix $M(\xi)$. Considering an ellipsoid-shaped obstacle with $r_x = 0.5$,$r_y = 2$, $C = [1, 3]^T$, $\theta = \frac{\pi}{6}$ and $\xi_r = C$; and applying that matrix to a DS with $\xi_a = [0,0]^T$, $A = 1$ and the velocity of the attractor as $\dot{\xi}_a = [1,0]^T$, we obtain the result in Figure 5.6. While in Figure 5.7 $\xi_r = [1.5,2]^T$, therefore it is not in the center of the obstacle.

It is possible to notice that in both cases the streamlines avoid the obstacles. The only exceptions regards the points that are already inside the obstacle: in such cases the mass point would get trapped; the solution has been implemented by [7] in python, using a repulsive function inside the obstacle. This solves the problem,

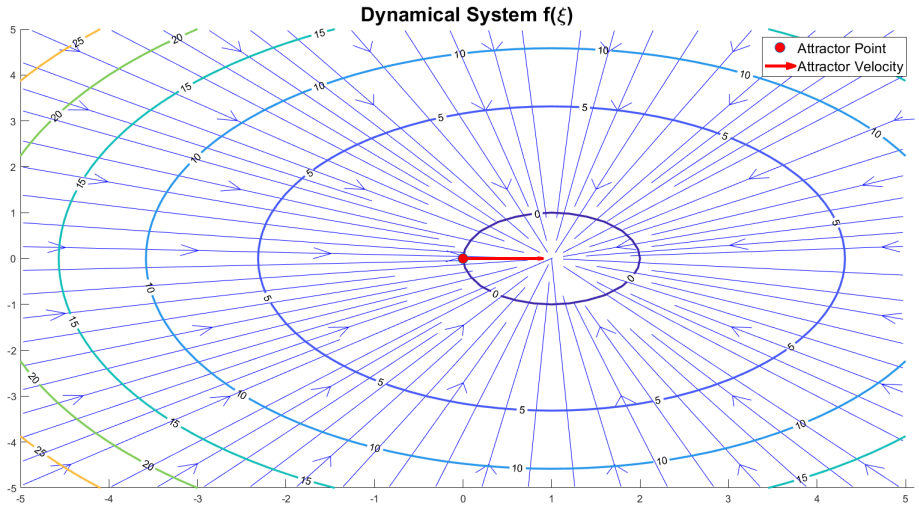


Figure 5.5: DS with moving attractor

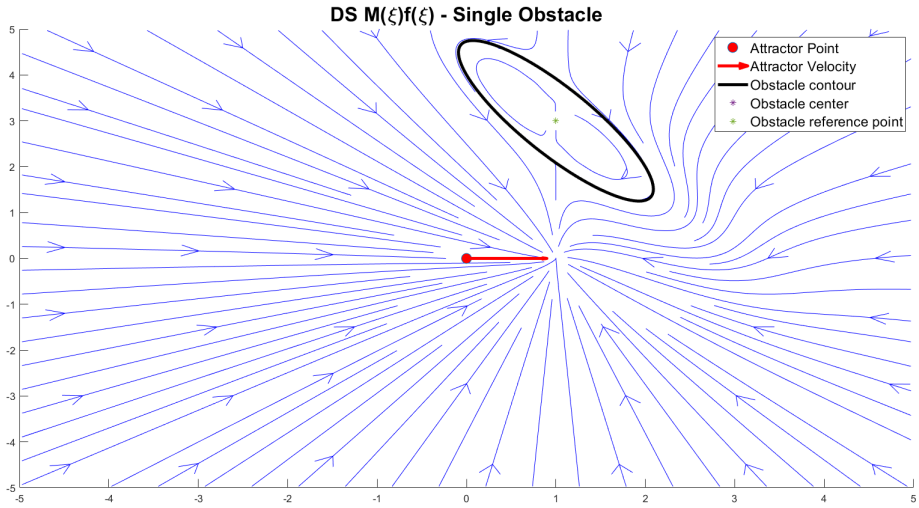


Figure 5.6: Modulated DS with moving attractor and single obstacle

but it creates a discontinuity in the final DS that could result in abrupt changes in the robot motion.

Moreover, until now we have assumed that there are not constraints on the speed that the mass point can have, this means that if the state ξ is quite far from the attractor ξ_a, the resulting velocity could be very high, since it is linearly dependent on the distance between the two points. An easy way to limit the velocity is to saturate it: this has to be done after the modulation, otherwise the final result

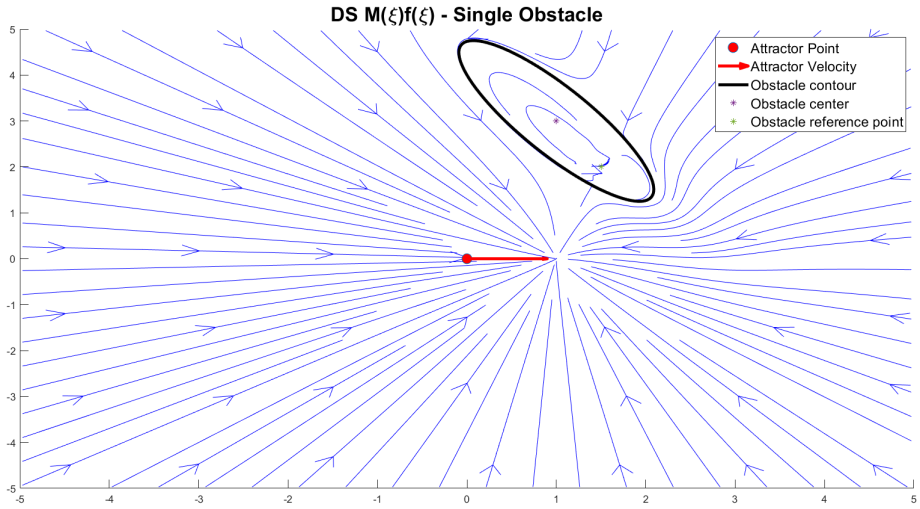


Figure 5.7: Modulated DS with moving attractor, single obstacle and $\xi_r \neq C$

could exceed the limit anyway. The drawback of this approach is evident near the obstacle, where a consistent stretching is necessary to avoid it; it may happen that limiting the point velocity brings the point to collide with the obstacle.

Since the obstacle represents a real object that has to be avoided, in order to avoid the collision different approaches are used:

- the ellipsoid-shaped obstacle is actually bigger than the real object that it represents; in such a way, even if the mass point collides with the ellipse and goes inside it, the repulsive function has still some time to correct the behavior.
- change the $\Gamma(\xi)$ function in order to sense the presence of the obstacle in advance.

5.6 $\Gamma(\xi)$ function variations

Varying the $\Gamma(\xi)$ function could be a useful method to perceive in advance the presence of an obstacle. Nevertheless, we have to remember that the this function has precise characteristics, that can not be modified, otherwise the modulation will not work. In particular, the function has the form:

$$\Gamma(\xi) = \left(\frac{||\xi - \xi^r||}{R(\xi)} \right)^{2h}$$

And has to be:

- $\Gamma(\xi) > 1$ outside the obstacle;
- $\Gamma(\xi) = 1$ on the boundary of the obstacle.

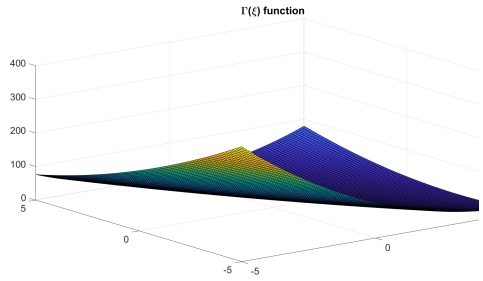
Therefore, even if we want to modify the function, we still have to guarantee these characteristics.

For instance, using the modified $\Gamma(\xi)$ function as in Equation 5.30, where c is a constant value, the over mentioned characteristics are respected, but the effect of the obstacles are perceived differently.

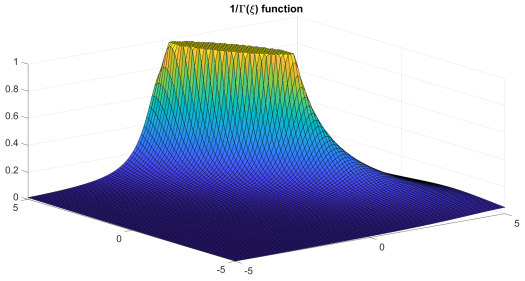
$$\Gamma(\xi) = \frac{\left(\frac{||\xi - \xi^r||}{R(\xi)}\right)^{2h} - 1}{c} + 1 \tag{5.30}$$

Considering again the obstacle with $r_x = 0.5$,$r_y = 2$, $C = [1, 3]^T$, $\theta = \frac{\pi}{6}$ and $\xi_r = C$; and an attractor with $\xi_a = [0,0]^T$, $A = 1$ and $\dot{\xi}_a = [1,0]^T$.

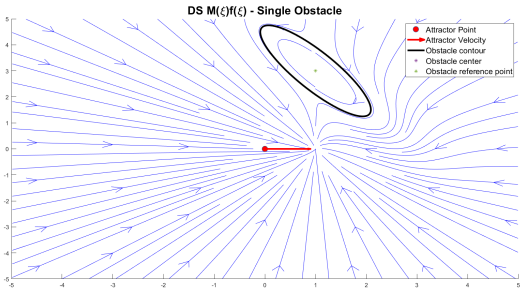
The results using $c = 1$, $c = 3$ and $c = 10$ are shown in Figure 5.8, Figure 5.9 and Figure 5.10; in this case, $\Gamma(\xi)$ inside the obstacle is set as 1, in order to avoid peaks in the function $\frac{1}{\Gamma(\xi)}$.



(a) $\Gamma(\xi)$ function

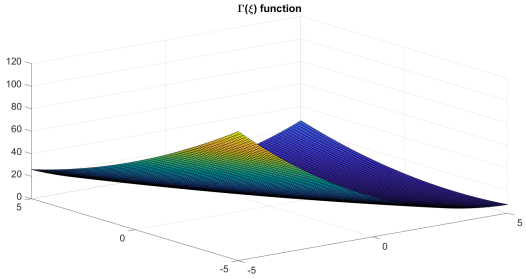


(b) $1/\Gamma(\xi)$ function

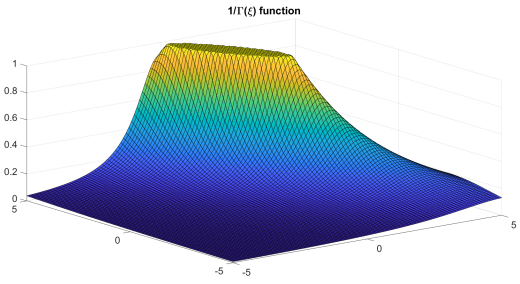


(c) DS $M(\xi)f(\xi)$

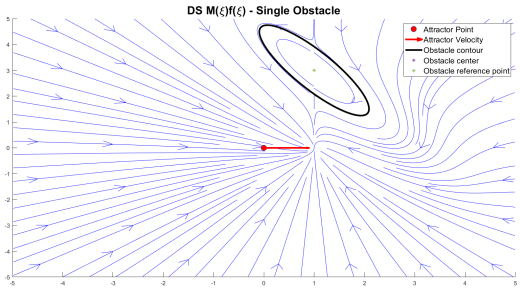
Figure 5.8: $\Gamma(\xi)$ function with c = 1



(a) $\Gamma(\xi)$ function

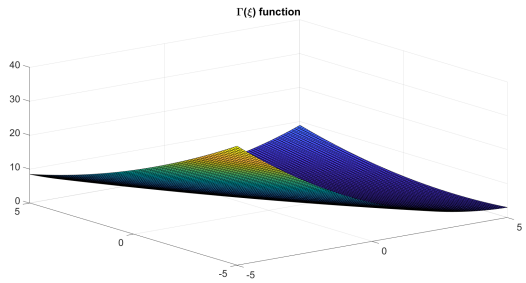


(b) $1/\Gamma(\xi)$ function

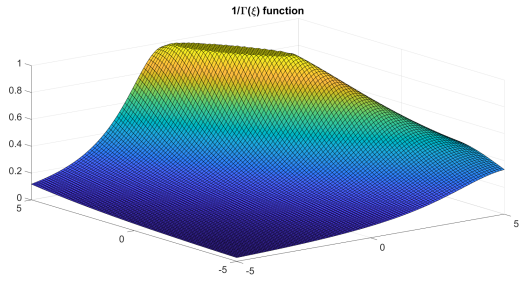


(c) DS $M(\xi)f(\xi)$

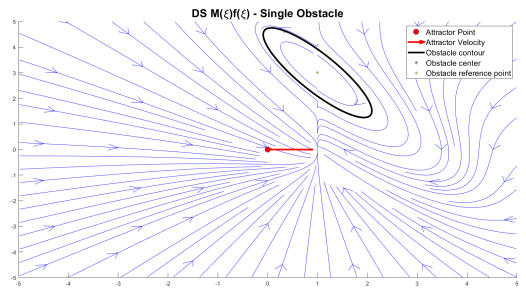
Figure 5.9: $\Gamma(\xi)$ function with c = 3



(a) $\Gamma(\xi)$ function



(b) $1/\Gamma(\xi)$ function



(c) DS $M(\xi)f(\xi)$

Figure 5.10: $\Gamma(\xi)$ function with c = 10

In the cases with $c > 1$ we can notice that the shape of the $\Gamma(\xi)$ functions are very similar, but their value decrease as c increases. This stands out especially in the functions $1/\Gamma(\xi)$, because in each of them the maximum value is 1 as expected, and then they decrease in different way.

The different shapes of $\Gamma(\xi)$, with $c > 1$, cause the streamlines to be more stretched; in particular, the effect of the presence of the obstacle is sensed quite far. Therefore, the effect is the desired one and the mass point can react in advance; nevertheless, some issues emerge: using, for instance, $c = 10$, would cause a strange behaviour in the states where the obstacle is not present and, most important, where it is not even close.

5.7 Moving obstacle

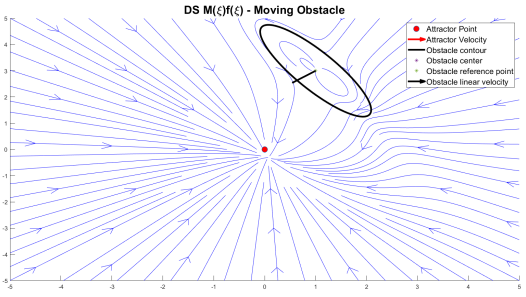
Until now the analysis focused on the modulation algorithm with a single fixed obstacle and a moving attractor. In this section we will see how a moving obstacle affects the streamlines and, hence, the DS of the mass point.

Following the procedure mentioned in Section 5.2 and remembering that the equations for the moving obstacles are Equation 5.13 and Equation 5.14, also this part of the algorithm has been implemented, in order to understand potential issues related to the moving obstacle and the moving target.

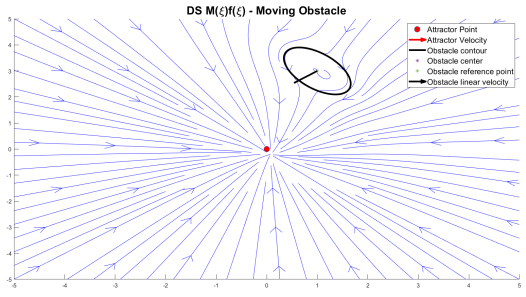
We can observe the behaviour of the streamlines with moving obstacles of different shapes, both with a fixed target ($\xi_a = [0,0]^T$, $A = 1$ and $\dot{\xi}_a = [0,0]^T$) and with a moving one ($\xi_a = [0,0]^T$, $A = 1$ and $\dot{\xi}_a = [1,0]^T$). The ellipsoid-shaped obstacles will have $r_x = 0.5$, $C = [1,3]^T$, $\theta = \frac{\pi}{6}$ and $\xi_r = C$; while we will consider $r_y = 2$, $r_y = 1$ and $r_y = 0.5$. For all the cases we will consider for the obstacle a linear velocity of the center $\dot{\xi}^{L,o} = [-0.5, -0.5]^T$ and an angular velocity $\dot{\xi}^{R,o} = -\pi/3$.

Figure 5.11 and Figure 5.12 show the results. First of all, we can observe that, using the algorithm proposed in Section 5.2 with fixed target, we obtain an expected behaviour: topographically critical points are displaced, for instance the attractor, because we are not doing a pure modulation of the DS [5]. Moreover, some streamlines are not able to avoid the obstacle anymore, especially if the obstacle is very big. This issue has been solved by Dr. Huber including a repulsive DS inside the obstacle, that force the state to go outside the obstacle; causing the global DS to be discontinuous though.

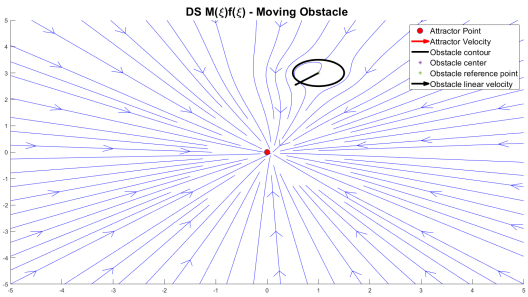
For what concern the moving attractor, the problems are basically the same. From the graphical representation (Figure 5.12) we can assert that this algorithm works in situations where the obstacles are not too big. Since we want to deal mainly with obstacles that represent people around the robot, we can expect a correct behaviour from the robot.



(a) $r_y = 2$ function

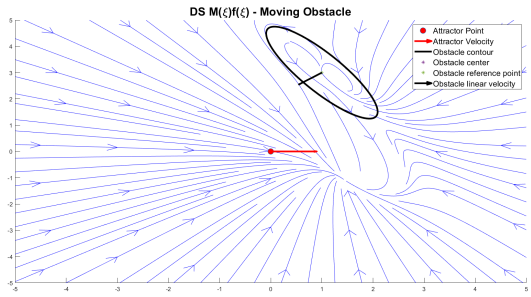


(b) $r_y = 1$ function

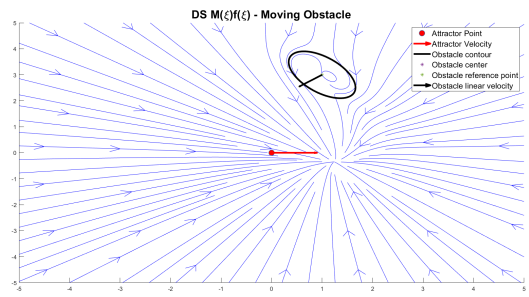


(c) $r_y = 0.5$

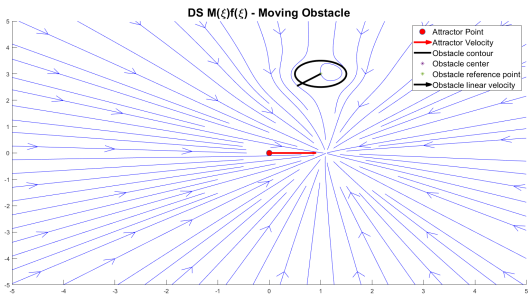
Figure 5.11: Moving obstacle and fixed target



(a) $r_y = 2$ function



(b) $r_y = 1$ function



(c) $r_y = 0.5$

Figure 5.12: Moving obstacle and moving target

5.8 Multiple obstacles

When multiple obstacles are present in the environment, the approach to follow is the one illustrated in Section 5.2 and implemented in Matlab as shown in 3. We have to compute the modulated DS velocities for each obstacle and then combining them to obtain the global DS.

We are assuming that all the obstacles have an ellipsoid shape and, therefore, the convexity assumption is guaranteed. We have to analyze the case in which two or more obstacles overlap and form a convex obstacle.

Firstly, we can test the algorithm with six different obstacles, without overlapping. The result is shown in Figure 5.13. As expected, the streamlines are able to avoid all the obstacles, suggesting us the feasibility of this algorithm.

Moving some obstacles, as shown in Figure 5.14, it is possible to create concave hulls. In this conditions, to guarantee the obstacle avoidance, the obstacles that form the hull must have the same reference point inside them. The two obstacles in the upper part of the Figure 5.14 respect that condition, while the two in the lower part have different reference points. It stands out that, in the first case the streamlines avoid both the obstacles, confirming the hypothesis of the same reference point; in the other one the algorithm is not able to produce a modulated DS capable of guarantee the obstacle avoidance. In particular, we can notice that minima are present on the border of the two obstacles; if a mass point was subject to that DS, it would get stuck and would never reach the desired attractor.

The algorithm that updates the position of the reference points inside the obstacles was not developed in MATLAB. It is a feature of the algorithm implemented in Python by Dr. Huber. It is worth to mention that this part of the algorithm represents the bottleneck of the DS modulation. Due to its complexity it slows down the entire process, making the algorithm unusable in real-time scenario; especially if there are many obstacles in the environment considered. Of course this represents a big limitation because, as we have seen, using the algorithm without updating the reference points produces local minima that can affect the performance of the whole system.

Algorithm 3 Modulation algorithm for multiple obstacles *mod*

procedure MOD($\xi, target, obs$)
    ▷ ξ is the state where we compute the modulation
    ▷ $target$ is the target we want to reach
    ▷ obs is a list with obstacles' information
    ▷ Initialization
    $\Gamma \leftarrow [\ ]$
    $n_{obs} \leftarrow length(obs)$
    $G \leftarrow [\ ]$
    $W \leftarrow [\ ]$ ▷ number of obstacles
    $vel \leftarrow [0,0]^T$ ▷ final velocity after the modulation
    ▷ Execution
    if $n_{obs} = 0$ **then**
        $vel \leftarrow \boldsymbol{f}(\xi)$ ▷ No modulation is computed
    else
        $\boldsymbol{n}^f(\xi) \leftarrow \boldsymbol{f}(\xi)$
        for $j \in n_{obs}$ **do**
            $d \leftarrow \xi - obs[j].c$
            $obs\_vel \leftarrow obs[j].lin\_vel + obs[j].ang\_vel * [d(2), d(1)]^T$
            $M\_matrix[j], \Gamma[j] \leftarrow M(\xi, obs[j])$
            $Mod\_DS[j] \leftarrow M\_matrix[j](\boldsymbol{n}^f(\xi) - obs\_vel) + obs\_vel$
            $Magn\_Mod\_DS[j] \leftarrow ||Mod\_DS[j]||$
            if $Magn\_Mod\_DS[j] = [0,0]^T$ **then**
                $\boldsymbol{n}_{Mod\_DS}[j] \leftarrow Mod\_DS[j]$
            else
                $\boldsymbol{n}_{Mod\_DS}[j] \leftarrow Mod\_DS[j]/Magn\_Mod\_DS[j]$
            end if
        end for

```
        for j ∈ n_obs do
            G ← Γ − 1
            G[j] ← [ ]                                  ▷ Remove j-th component
            num ← ∏(G)
            den ← 0
            for k ∈ n_obs do
                G ← Γ − 1
                G[k] ← [ ]                              ▷ Remove k-th component
                den ← den + ∏(G)
            end for
            W[j] ← num/dem
        end for
        ▷ Compute the magnitude of the final velocity
        Magn_DS ← ∑W[j]Magn_Mod_DS[j]
        ▷ Direction of the original DS
        if ||n^f|| ≠ 0 then
            n^f ← n^f/||n^f||
        end if
        ▷ Vector orthogonal to n^f
        e^f ← [−n^f[2], n^f[1]]^T
        ▷ Orthonormal matrix R_f
        R_f ← [n^f, e^f]
        n̂ ← R_f^T n_Mod_DS
        ▷ κ function
        for j ∈ n_obs do
            if n̂[2, j] ≥ 0 then
                κ ← acos(n̂[1, j])
            else
                κ ← −acos(n̂[1, j])
            end if
        end for
        κ̄ ← ∑W[j]κ[j]
        n̄ ← R_f[cos(||κ̄||), sin(||κ̄||)sign(κ̄)]^T
        vel ← n̄Magn_DS
    end if
end procedure
```

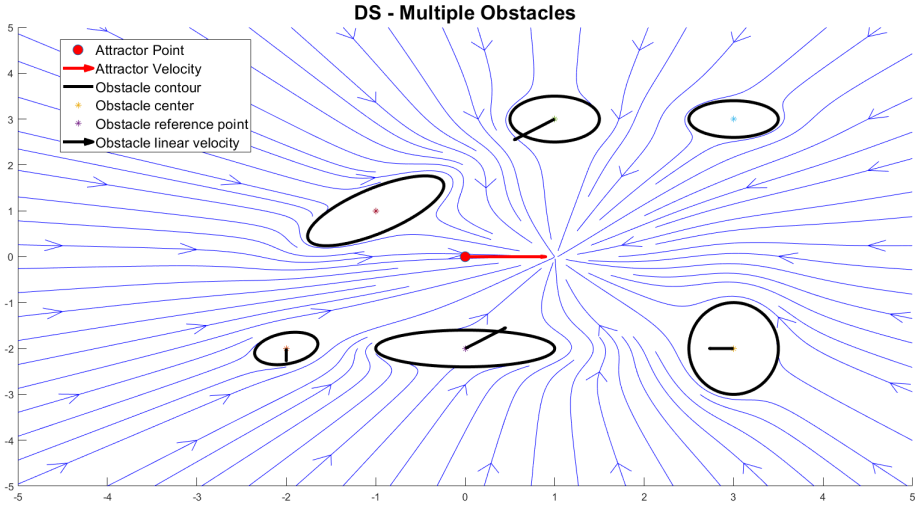


Figure 5.13: Multiple Obstacles

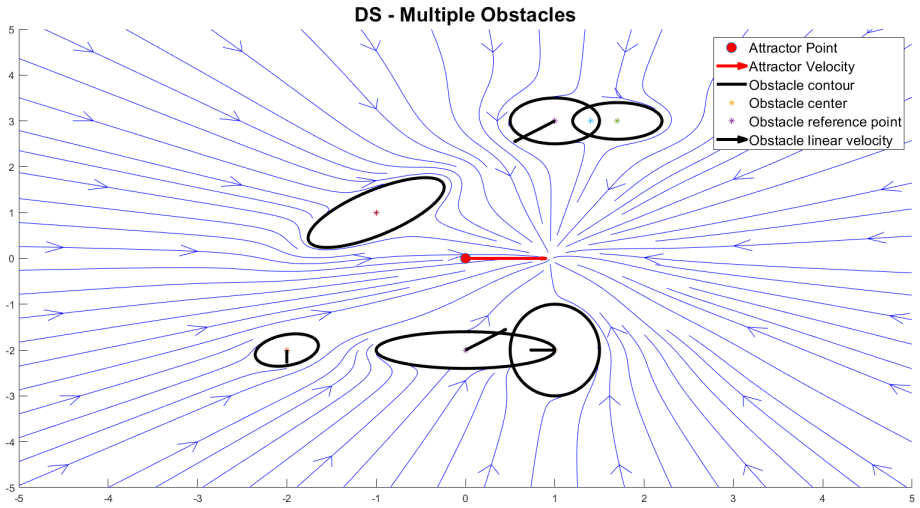


Figure 5.14: Multiple Obstacles with overlapping

5.9 Attractor inside the obstacle

An aspect of the algorithm that deserves attention concerns the presence of the attractor inside an obstacle. It is a situation that may happen in the real implementation, therefore it is worth to analyze it and understand which are the consequences.

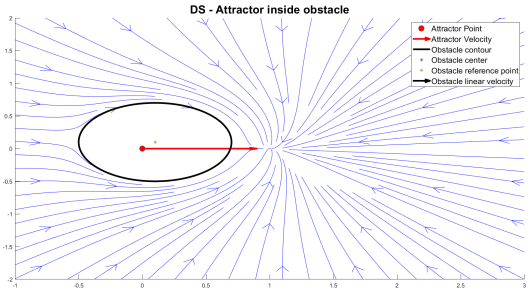
Figure 5.15 illustrates four scenarios. In all of them, the obstacles has $r_x = r_y = 0.6$ and it is fixed, so $\dot{\xi}^{L,o} = [0,0]^T$ and $\dot{\xi}^{R,o} = 0$. The attractor is $\xi_a = [0,0]^T$ and has $\dot{\xi}_a = [1,0]^T$.

(a) In Figure 5.15(a) the obstacle has $C = \xi_r = [0.1,0.1]^T$. The attractor is inside the obstacle, but due to its velocity, all the streamlines converge to a point that is outside the obstacle. This behaviour does not cause any issue, since the DS brings the state to avoid the obstacle and to reach the future position of the attractor, according to it velocity vector.

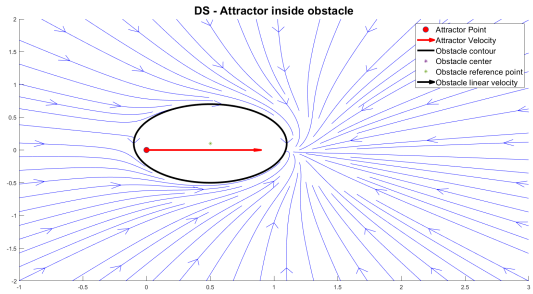
(b) In Figure 5.15(b), the obstacle has $C = \xi_r = [1.1,0.1]^T$; the attractor is outside the obstacle, but its velocity vector causes the un-modulated DS to bring the state inside the obstacle. Using the modulation we can see that the streamlines do not converge inside the obstacle, but on its border. The impenetrability of the obstacle is therefore guaranteed. Of course, in the real implementation, the robot could cross this border, but the repulsive force inside the obstacle would push it away.

(c) In Figure 5.15(c) the obstacle has $C = \xi_r = [0.5,0.1]^T$; both the attractor and its future position are inside the obstacle. Also in this case, the modulation does not allow the penetration of the obstacle and the streamlines converge to a point on the border.

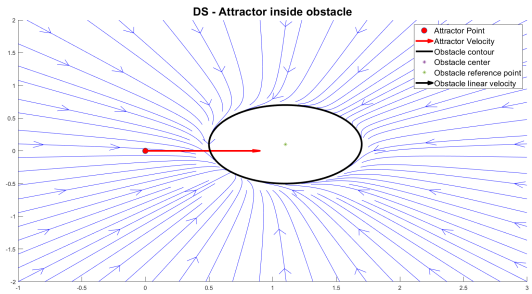
(d) Figure 5.15(d) shows a scenario that is very unlikely, but deserves an analysis. The obstacle has $C = \xi_r = [1,0]^T$; the attractor is outside the obstacle, but its future position, according to its velocity vector, is inside the obstacle and coincides with the obstacle's reference point ξ_r. Differently from the other scenarios illustrated above, the modulation is not able to guarantee the impenetrability of the obstacle and all the streamlines point toward the center of the obstacle. This is a situation that must be avoided, because the modulation algorithm is invalidated, but realistically it is almost impossible that this scenario could happen in the real-world implementation.



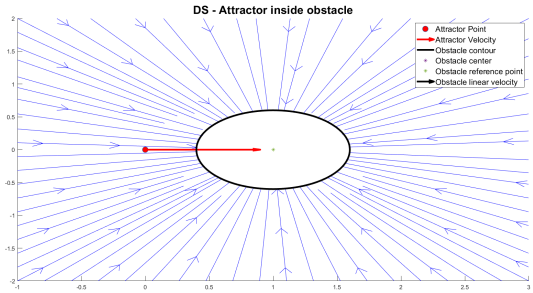
(a) Actual attractor on obstacle



(c) Actual attractor and future position on obstacle



(b) Future position on obstacle



(d) Future position on obstacle reference point

Figure 5.15: Attractor on obstacle

5.10 Robot Shape and Moving Local Attractor

Until now, the modulating algorithm has been analyzed in its different aspects and configurations. The main advantages and limitations have been illustrated.

The high level controller, though, is not limited to the computation of the desired velocity vector. Together with the Obstacle and Target Detection Algorithm, described in Chapter 7, it has to deal with the information about the target and the shape of the robot.

5.10.1 Robot Shape

A first consideration that is needed regards the state of the robot ξ where we apply the modulating algorithm: the state is a mass point and does not take into account the real dimension of the robot. We can assume that the state that we are controlling is in the center of the robot. An easy way to take into account the shape of the robot is to consider a radius, called r_{robot}, that forms a circle centered in the robot center and that embraces the robot, as shown in Figure 5.16.

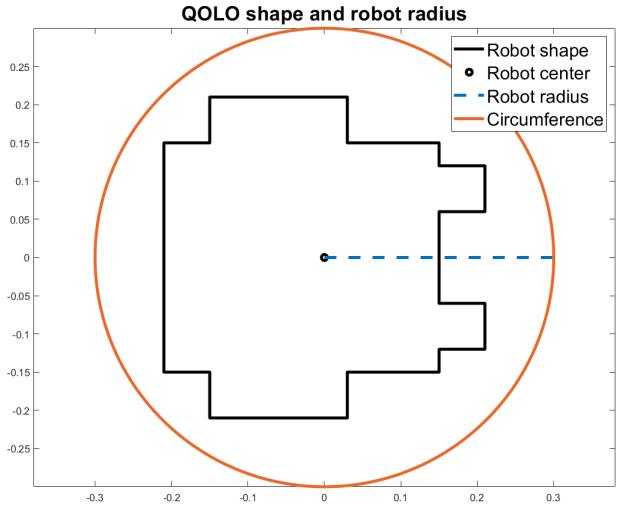


Figure 5.16: QOLO shape and robot radius

Having the information about r_{robot}, it is possible to enlarge the perceived obstacles so that the shape of the robot is taken into account. Using Equation 5.31, where r_x^O and r_y^O are the radii of an ellipsoid shaped obstacle, we obtain a new set of obstacles that can be used for the modulation of the mass point in the center of the robot.

$$\begin{aligned} new\_r_x^O &= r_x^O + r_{robot} \\ new\_r_y^O &= r_y^O + r_{robot} \end{aligned} \tag{5.31}$$

Unfortunately, using this method may cause some issues because, if the robot is not able to have the desired velocity, it may collide with an obstacle. In the Python implementation of the modulation, if the state is inside the obstacle, a repulsive force is applied to the state in order to push away it. With the new radii, computed with Equation 5.31, we can not exploit this feature of the algorithm, because a collision of the mass point with an obstacle means that the real robot has actually hit something around it.

To take into account this aspect that is related to safety of the robot user and of the people surrounding him, we can further enlarge the obstacles by a safety factor ϵ_{safe}. Equation 5.32 shows how the obstacle radii would change.

$$\begin{aligned} new\_r_x^O &= r_x^O + r_{robot} + \epsilon_{safe} \\ new\_r_y^O &= r_y^O + r_{robot} + \epsilon_{safe} \end{aligned} \tag{5.32}$$

Of course, ϵ_{safe} does not guarantee that the robot will never collide with an obstacle, but using a large enough value, could guarantee a certain level of confidence.

Other issues related to this method regards the obstacles' shape itself. Obstacles that are too big can overlap more easily and, consequently, they may create a single concave obstacle, formed by more convex obstacles. That does not allow the robot to travel the more convenient path, even if it would be possible, forcing it to avoid the whole concave obstacle.

5.10.2 Moving attractor

Having included the dimension of the robot, we have now to deal with the target to follow. As stated above, the DS modulation is applied to the center of the robot; we can not simply identify the target and using its center as the attractor ξ_a.

Here, we assume that the target center ξ_{tar} and the target speed $\dot{\xi}_{tar}$ are know. They are two column vectors; the first one identifies the position of the center of the obstacle that we identify as the target. $\dot{\xi}_{tar}$ gives us information regarding the magnitude and direction of the velocity of the target.

From the point of view of the high level controller, also the target to follow must be considered as an obstacle, in order to avoid collision with it. Therefore, also this obstacle must be enlarged using Equation 5.32.

A first method that could be used to identify the attractor ξ_a is illustrated in Algorithm 4. The attractor in this way is fixed either to the right or to the left of

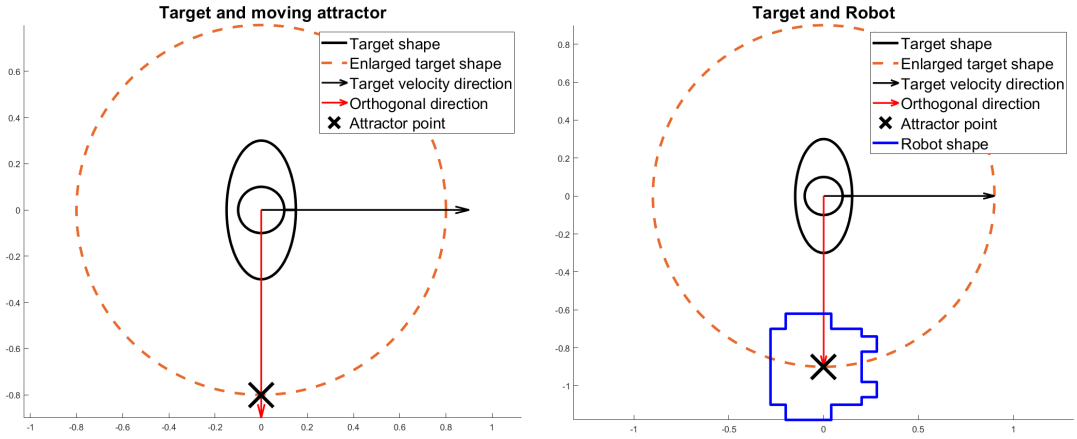
the target and moves with it. Then, the assumption regarding the speed of this attractor is that it is the same of the speed of the target: $\dot{\xi}_a = \dot{\xi}_{tar}$.

Figure 5.17 shows a target person with the moving attractor at its right and the robot near it. We can notice that using a safety distance $\epsilon_{safe} = 0.2m$ and considering $r_{robot} = 0.4m$, the robot tries to reach a point that is sufficiently distant from the target and the collisions should be avoided.

Algorithm 4 Fixed moving attractor

▷ **Initialization**
ξ_{tar} ▷ Target center
$\dot{\xi}_{tar}$ ▷ Target Speed
r_x^{tar} ▷ Semi-major axis of the enlarged obstacle
r_y^{tar} ▷ Semi-minor axis of the enlarged obstacle
θ_{tar} ▷ Tilted angle w.r.t. fixed reference frame
▷ **Attractor computation**
▷ Choose the position of the attractor w.r.t. the target (Right or Left)
▷ Compute the rotation angle θ_{rot}
if Right **then**
    $\theta_{rot} = -\pi/2$
else
    $\theta_{rot} = \pi/2$
end if
▷ Compute the rotation matrix R
$R = \begin{bmatrix} cos(\theta_{rot}) & sin(\theta_{rot}) \\ -sin(\theta_{rot}) & cos(\theta_{rot}) \end{bmatrix}$
▷ Compute the direction of target
$n_{tar} = \frac{\dot{\xi}_{tar}}{\|\dot{\xi}_{tar}\|}$
▷ Compute the orthogonal direction w.r.t the direction of target n_{tar}
$\hat{n}_{tar} = R \cdot n_{tar}$
▷ Compute the maximum between r_x^{tar} and r_y^{tar}
$r_{max} = max(r_x^{tar}, r_y^{tar})$
▷ Compute ξ_a
$\xi_a = \xi_{tar} + r_{max} \cdot \hat{n}_{tar}$

Using this point as attractor is a valid option, but the presence of the other obstacles is not taken into account. Consequently, some issues arise; for instance, if the target moves near an obstacle and the attractor is set to be on the same side of this obstacle, it may happen that the attractor goes on the it. As we saw in Section 5.9, the modulating algorithm does not allow the state to go inside the the obstacle and virtually moves the attractor on the border of the obstacle. The DS produced



(a) Target and moving attractor **(b)** Target and robot

Figure 5.17: Target and fixed moving attractor

could bring the state, and therefore the robot, to avoid the obstacle from the side opposite to the target. Until now, we have considered a complete knowledge of the environment, but in the real implementation, the robot can not lose track of the target; in a situation like the one described above, this may happen. The algorithm here developed has to take into account also this possible situation and act in advance.

A possible solution, here developed, exploits the Algorithm 4 to find the ideal position of the attractor ξ_a, that we will call ξ_{rest}, and takes into account the presence of the other obstacles to compute the position of ξ_a. The system will be modeled as an angular spring that tries to keep ξ_a at its rest position, i.e. ξ_{rest}, while the obstacles exercise a repulsive force. In this way, if the attractor goes near an obstacle, the repulsive force tends to push it away, and a virtual angular spring tends to keep it in position. We are going to use two variables, called k_{attr} and k_{attr}, to model the spring. Equation 5.33 illustrates the dynamics of θ_a, which is the angle formed by the lines that link ξ_a with ξ_{tar} and ξ_{rest} with ξ_{tar}. $\dot{\theta}_a$ is used to change the angle θ_a to a more suitable position for the attractor. Varying these two values will change the behaviour of the attractor, therefore a trial and error tuning is necessary.

$$\dot{\theta}_a = -k_{attr} \cdot \theta_a - k_{rep} \cdot \dot{\theta}_{rep} \tag{5.33}$$

Algorithm 5 illustrates the procedure.

Algorithm 5 Moving attractor

▷ **Initialization**
k_{attr} ▷ Elastic constant for the attraction
k_{rep} ▷ Elastic constant for the repulsion
▷ Use Algorithm 4 to find ξ_{rest}
if ξ_a not defined **then**
    $\xi_a \leftarrow \xi_{rest}$
else
    ▷ Compute the angle θ_a between the vectors $[\xi_a - \xi_{tar}]$ and $[\xi_{rest} - \xi_{tar}]$
    ▷ Compute the repulsion caused by each obstacle on ξ_a
    $Tot\_rep \leftarrow [0,0]^T$
    for Obstacles *neq* Target **do**
        $rep = \xi_a - C^{obs}$
        $Tot\_rep+ = \frac{rep}{\|rep\|} \cdot \frac{1}{\|rep\|}$ ▷ Closer is the obstacle, stronger is the repulsion
    end for
    ▷ Tot_rep is a vector that represents the total repulsive force that acts on ξ_a
    ▷ Compute $Tang\_\xi_a$, the tangent of the circumference of radius r_{max} in the point ξ_a
    ▷ Compute $\dot{\theta}_{rep}$, the angular velocity caused by the Tot_rep in the point ξ_a
    $\dot{\theta}_{rep} \leftarrow \langle Tot\_rep, Tang\_\xi_a \rangle / r_{max}$
    ▷ Compute $\dot{\theta}_a$, the angular velocity on the point ξ_a
    $\dot{\theta}_a \leftarrow -k_{attr} \cdot \theta_a - k_{rep} \cdot \dot{\theta}_{rep}$
    ▷ Bound $\dot{\theta}_a$ to $\max_{\dot{\theta}_a}$
    ▷ Compute the new angle θ_a, integrating $\dot{\theta}_a$
    $\theta_a \leftarrow \theta_a + dt \cdot \dot{\theta}_a$
    ▷ Compute the new position of ξ_a
    $R = \begin{bmatrix} cos(\theta_a) & sin(\theta_a) \\ -sin(\theta_a) & cos(\theta_a) \end{bmatrix}$
    $\xi_a = R \cdot \frac{[\xi_a - \xi_{tar}]}{\|[\xi_a - \xi_{tar}]\|} \cdot r_{max} + \xi_{tar}$
    ▷ If the new attractor position is ahead the rest position, the attracotr ξ_a is set as ξ_{rest}
    ▷ Compute the angle new_θ_a between the vectors $[\xi_a - \xi_{tar}]$ and $[\xi_{rest} - \xi_{tar}]$
    if $new\_\theta_a > 0$ **then**
        $\xi_a \leftarrow \xi_{rest}$
    end if
end if

Chapter 6

Low Level Control

Controlling a unicycle is not an easy task due to the non-holonomic constraints, as a matter of fact the unicycle can not move in the direction orthogonal to the wheels. In this chapter different approaches for the generation of the control inputs will be analysed in order to understand the feasibility and the possible benefits. Moreover, since the low level controller has to be analyzed in relation with the high level controller, we will see how the interaction works.

It is important to remember that we want to use the information about the robot state and the high level modulating DS to generate the most suitable inputs. Hence, we need a low level controller that is able to move the unicycle from an initial pose to a desired one.

The algorithms used are:

- Differential flatness;
- Stabilized Feedback Control;
- Non-linear MPC - Position control;
- Non-linear MPC - Velocity control;
- MPC - inputs bounding.

6.1 Differential flatness [25] [26]

By definition, the Differential Flatness is a property of non-linear dynamical systems whose state (ξ) and inputs (u) can be expressed as function of its outputs (y), called flat, and their derivatives up to a certain order r.

$$\begin{aligned} \xi &= \xi(y, \dot{y}, \ddot{y}, ..., y^r) \\ u &= u(y, \dot{y}, \ddot{y}, ..., y^r) \end{aligned} \tag{6.1}$$

Thanks to this property, starting from the temporal evolution of $y(t)$, i. e. the trajectory of the robot, it is possible to determine the states and the control inputs that induced that behaviour. Therefore, the planning problem for the input generation can be divided into:

- Path selection: the geometrical path that the robot should follow, taking into account the constraints imposed by its structure.
- Timing law definition: timing necessary to accomplish the path.

Considering ξ_i as the initial state or pose of the robot and ξ_f the final one, we want to plan a trajectory $\xi(t)$ from t_i to t_f. It is possible to describe the path as $\xi(s)$, with $\frac{d\xi(s)}{ds} \neq 0$ and timing law $s = s(t)$. A good choice for s is the curvilinear abscissa. As a consequence

$$\dot{\xi} = \frac{d\xi}{dt} = \frac{d\xi}{ds}\dot{s} = \xi' \dot{s} \tag{6.2}$$

Vector ξ' lays along the tangent to the path in the configuration space, oriented towards increasing s; while $\dot{s}$ regulates the modulus of the velocity.

The geometrical model of the unicycle is:

$$\begin{cases} x' = \tilde{v} \cdot cos(\theta) \\ y' = \tilde{v} \cdot sin(\theta) \\ \theta' = \tilde{\omega} \end{cases} \tag{6.3}$$

Considering the Cartesian path $(x(s), y(s))$, the associated state path is $\xi(s) = [x(s), y(s), \theta(s)]^T$, with

$$\theta(s) = arctan2(y'(s), x'(s)) + k\pi, \quad k = 0{,}1 \tag{6.4}$$

k can set the direction: $k = 1$ implies backward drive and $k = 0$ forward drive. Moreover, from the geometrical model we can derive the geometric inputs:

$$\tilde{v}(s) = \pm\sqrt{(x'(s))^2 + (y'(s))^2} \tag{6.5}$$

$$\tilde{\omega}(s) = \frac{y''(s)x'(s) - x''(s)y'(s)}{(x'(s))^2 + (y'(s))^2} \tag{6.6}$$

We can notice that the state and the control inputs depend on the Cartesian path and its derivatives, therefore, the unicycle model has the property of differential flatness.
To compute the path it is sufficient to interpolate between the initial and final values x_i, y_i and x_f, y_f of the flat output $x(s)$, $y(s)$, using for example Cartesian polynomials, with $s \in [0,1]$:

$$x(s) = s^3 x_f - (s-1)^3 x_i + \alpha_x s^2(s-1) + \beta_x s(s-1)^2 \tag{6.7}$$

$$y(s) = s^3 y_f - (s-1)^3 y_i + \alpha_y s^2(s-1) + \beta_y s(s-1)^2 \tag{6.8}$$

With

$$x'(0) = k_i cos(\theta_i) \qquad x'(1) = k_f cos(\theta_f) \tag{6.9}$$

$$y'(0) = k_i sin(\theta_i) \qquad y'(1) = k_f sin(\theta_f) \tag{6.10}$$

k_i and k_f are free, nonzero and constrained to have the same sign. Choosing $k_i = k_f = k > 0$ we obtain:

$$\alpha_x = kcos(\theta_f) - 3x_f \qquad \beta_x = kcos(\theta_i) + 3x_i \tag{6.11}$$

$$\alpha_y = ksin(\theta_f) - 3y_f \qquad \beta_y = ksin(\theta_i) + 3y_i \tag{6.12}$$

Having the desired path between the initial and final state, we need to define a temporal law to accomplish this path. We just have to choose a time step T during which the unicycle should complete the path. Knowing T, it is possible to generate the control inputs using the formulas in Equation 6.13.

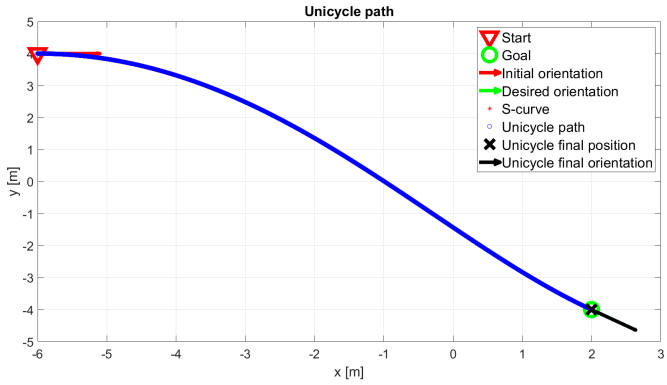
$$\begin{aligned} v &= \frac{1}{T} \cdot \tilde{v}(s) \\ \omega &= \frac{1}{T} \cdot \tilde{\omega}(s) \end{aligned} \tag{6.13}$$

6.1.1 Algorithm tests

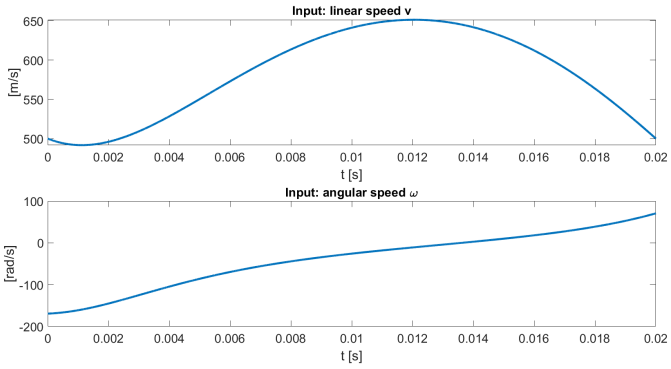
In this section the Differential flatness algorithm will be tested and examined in order to understand its feasibility and its critical issues.

Test 1

Starting from the initial state $\xi_i = [-6, +4, 0]^T$, we would like to arrive in the final state $\xi_f = [2, -4, -pi/4]^T$ in $T = 0.02s$. The value of the constant $k_i = k_f = k$ is $k = 10$. In this simulation the time T is very small, while the path to cover is more than 8 meters long. It is not a conventional scenario, but it can be useful to understand the potentiality of this algorithm. Dividing the time T into $n_{step} = 10000$, we obtain the result shown in Figure 6.1. The unicycle is able to reach the desired path, following perfectly the geometrical path (S-curve). Of course, in order to cover the path in such a small amount of time, the control inputs are huge and therefore unfeasible for the real implementation.



(a) Unicycle path

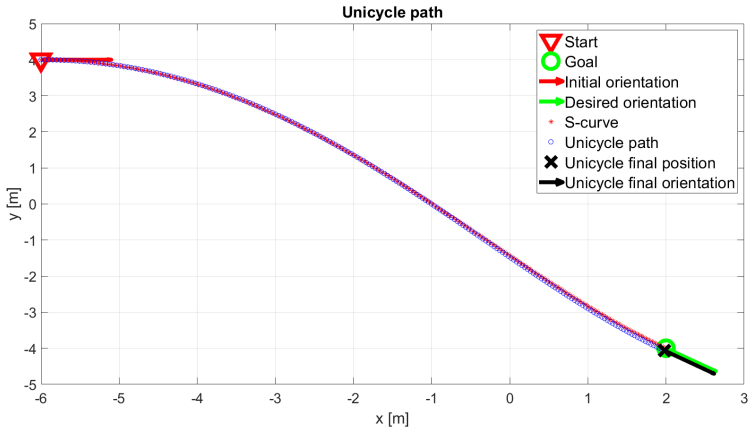


(b) Control inputs

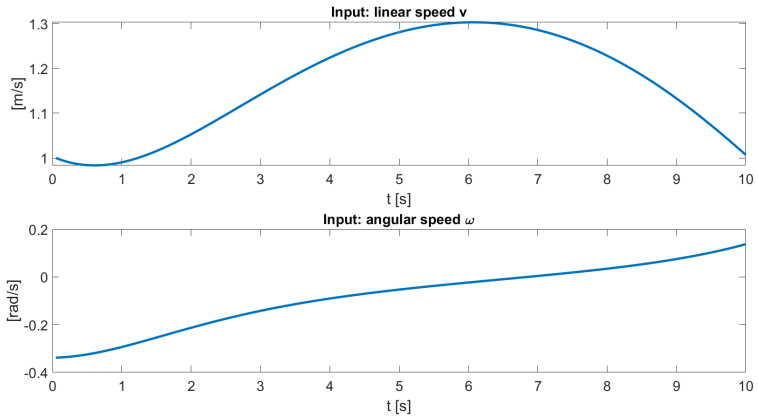
Figure 6.1: Differential Flatness - Test 1

Test 2

The scenario is the same as in Test 1, i.e. $\xi_i = [-6, +4, 0]^T$ and $\xi_f = [2, -4, -pi/4]^T$. In this case, though, the simulation time is $T = 10s$ with time step $dt = 0.05s$; therefore, $n_{step} = \frac{T}{dt} = 200$. The value of the constant $k_i = k_f = k$ is $k = 10$. Figure 6.2 shows the results of this test. The shapes of the path and of the control inputs are the same of Test 1; but in this case the control inputs can be used in the real implementation; moreover the path of the unicycle differs a bit, especially in the end, from the geometrical path.



(a) Unicycle path



(b) Control inputs

Figure 6.2: Differential Flatness - Test 2

Test 3

Test 1 and Test 2 illustrate two scenarios in which the initial and final positions are known and a path planning is employed to generate the control inputs for the system. Since we would like to use this controller as a low level controller, we need to use it together with the high level controller in order to verify its feasibility with our control structure.

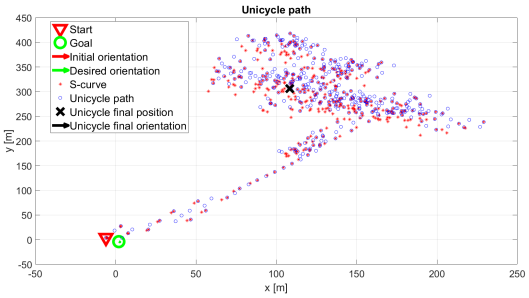
Figure 6.3 illustrates three examples of the integration of the DS algorithm with the differential flatness controller. The DS is used to generate the next desired pose and the next desired orientation. In these examples there are no obstacle, in order to see the behaviour of the unicycle in a simple environment. Hence, the DS generates a velocity vector that brings the robot to the final goal $\xi_f = [2, -4, -pi/4]^T$, starting from $\xi_i = [-6, +4, 0]^T$.

1. Figures (a) and (b) illustrate the path and control inputs obtained using $n_{step} = 1$.
2. Figures (c) and (d) illustrate the path and control inputs obtained using $n_{step} = 10$.
3. Figures (e) and (f) illustrate the path and control inputs obtained using $n_{step} = 100$.

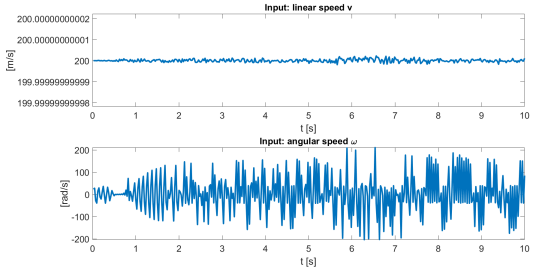
All the tests show that this kind of low level controller is not useful to produce feasible control inputs for our purpose. They are too big and do not produce a nice behaviour.

Considerations

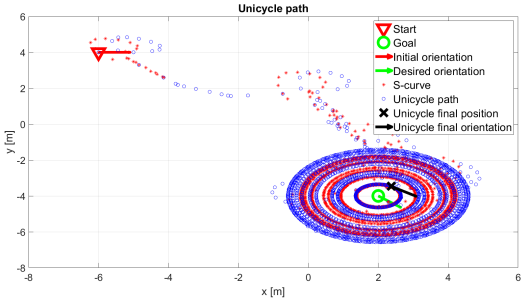
This algorithm seems to be useful if we need a path between two pose that are not very close one to the other. It is able to generate the inputs in open loop and respect the non-holonomic constraints of the unicycle. Unfortunately, it is not able to adapt the path planning to a situation like the one that we should have in the real implementation: the algorithm tends to produce circular paths between two consequent poses (generally very close one to the other), in order to make the robot reach the desired pose with the desired angle, but the outcome is totally unusable.



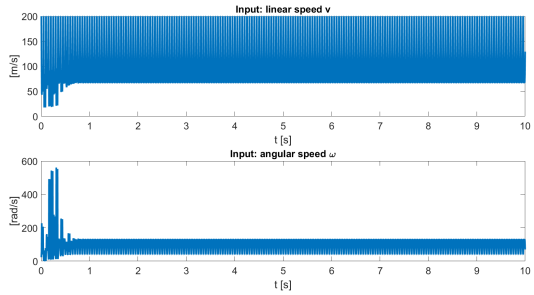
(a) Path: $n_{step} = 1$



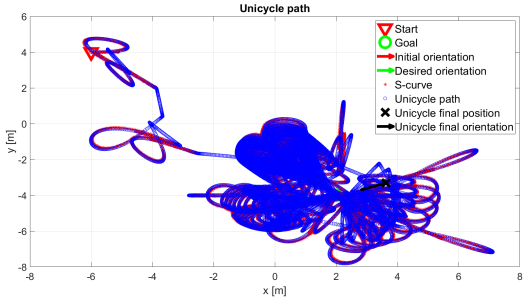
(b) Control inputs: $n_{step} = 1$



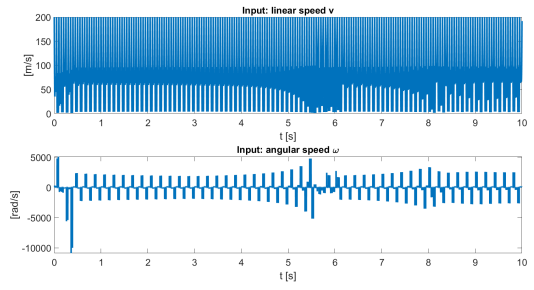
(c) Path: $n_{step} = 10$



(d) Control inputs: $n_{step} = 10$



(e) Path: $n_{step} = 100$



(f) Control inputs: $n_{step} = 100$

Figure 6.3: DS modulation and differential flatness

6.2 Stabilized Feedback Control

In control theory the feedback control is a methodology that uses the measures on the outputs of the system to produce control inputs. In this way, possible discrepancies between the desired behaviour and the real one are taken into account and can be corrected.

In this section, a Stabilized Feedback Control algorithm will be developed, using as reference the paper *'Stabilized Feedback Control of Unicycle Mobile Robots'* [27]. The algorithm proposes a controller that produces feedback control inputs to bring the robot to a final desired state (or pose), starting from a defined initial pose. It produces a kinematic solution for posture stabilization, using the polar coordinates to generate the control inputs.

First of all, it is necessary to define the transformations from the Cartesian to the polar system, remembering that the state of the robot is $\xi(t) = [x(t), y(t), \theta(t)]^T$:

$$\begin{cases} \rho = \sqrt{x^2 + y^2} \\ \gamma = atan2(y, x) - \theta + \pi \\ \delta = \gamma + \theta \end{cases} \tag{6.14}$$

In the polar system, ρ is the distance from the origin of the global frame to the origin of the robot frame, γ is the angle between the robot's frame and the environment frame, which is fixed; and δ is the rotation angle of the robot inside its frame.

Having $\xi_f = [x_f, y_f, \theta_f]^T$ as final pose, we can consider the error vector as:

$$\tilde{e}(t) = \begin{bmatrix} x(t) - x_f \\ y(t) - y_f \\ \theta(t) - \theta_f \end{bmatrix} \tag{6.15}$$

We transform the error vector $\tilde{e}(t)$ in the reference frame of the target using the rotation matrix $R(\theta_f)$:

$$R(\theta_f) = \begin{bmatrix} cos(\theta_f) & sin(\theta_f) & 0 \\ -sin(\theta_f) & cos(\theta_f) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{6.16}$$

$$e(t) = \begin{bmatrix} e_x(t) \\ e_y(t) \\ e_\theta(t) \end{bmatrix} = R(\theta_f) \cdot \tilde{e}(t) = \begin{bmatrix} cos(\theta_f) & sin(\theta_f) & 0 \\ -sin(\theta_f) & cos(\theta_f) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \tilde{e}(t) \tag{6.17}$$

Then we transform the error vector $e(t)$ in the polar-coordinate system, using Equation 6.14, obtaining ρ_e, γ_e and δ_e. At this point it is possible to define a

feedback law using a Lyapunov stability analysis. This law is defined such that it brings the system to the origin of the polar reference system; which means that the robot state is driven to the target ξ_f. The control inputs obtained with this law are:

$$\begin{aligned} v &= k_1 \cdot \rho_e \cdot cos(\gamma_e) \\ \omega &= k_2 \cdot \gamma_e + \frac{sin(\gamma_e)cos(\gamma_e)}{\gamma_e} \cdot (\gamma_e + k_3 \cdot \delta_e) \end{aligned} \tag{6.18}$$

With k_1 and $k_2 > 0$.

Particular attention as to be dedicated to the case in which $\gamma_e = 0$. The calculator may not be able to deal with this situation, due to numerical errors, therefore, if this situation arises, it would be better to use the Equation 6.19

$$\begin{aligned} v &= k_1 \cdot \rho_e \\ \omega &= k_3 \cdot \delta_e \end{aligned} \tag{6.19}$$

6.2.1 Algorithm tests

Here we present some tests useful to verify the applicability of the Stabilized Feedback Control with the modulated DS.

Test 1

Here, a test from the paper *'Stabilized Feedback Control of Unicycle Mobile Robots'* [27] is reproduced, therefore, in this case, the controller is not used together with modulated DS.

The unicycle starts from an initial state $\xi_i = [0, -5, 0]^T$ and wants to reach the final state $\xi_f = [0, 5, -\pi/2]^T$. The constants used are $k_1 = 0.1$, $k_2 = 0.3$ and $k_3 = 1.5$. The simulation, shown in Figure 6.5, produces the same output of the test done in the paper and illustrated in Figure 6.6. The whole simulation lasts $150s$, but the unicycle is able to reach the desired pose in $40 - 50s$. The control inputs are shown in Figure 6.4; they are feasible with the robot structure and produce a smooth behaviour.

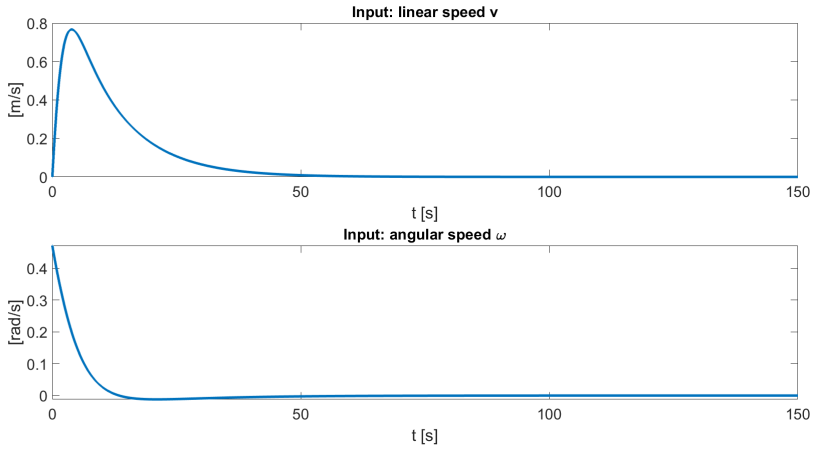
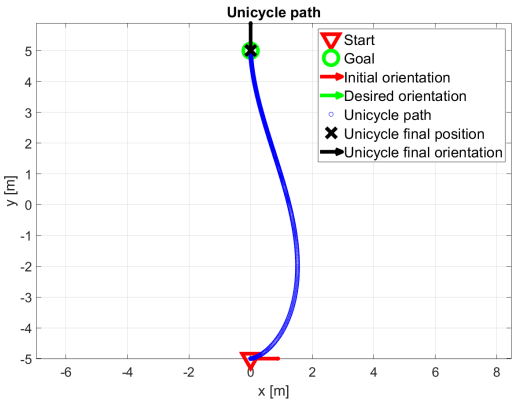
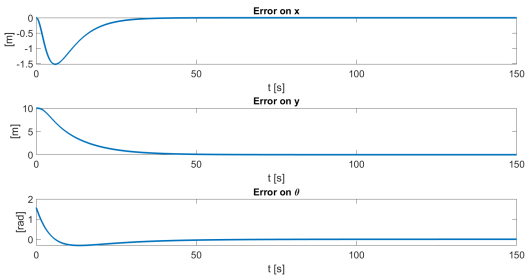


Figure 6.4: Stabilized Feedback - Test 1 - Control inputs

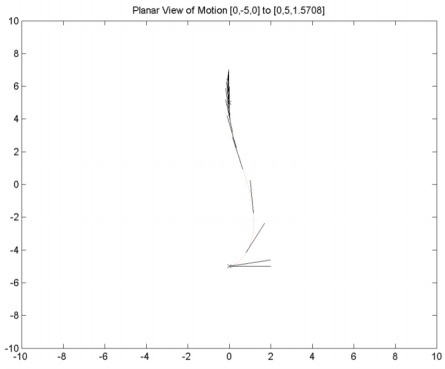


(a) Path

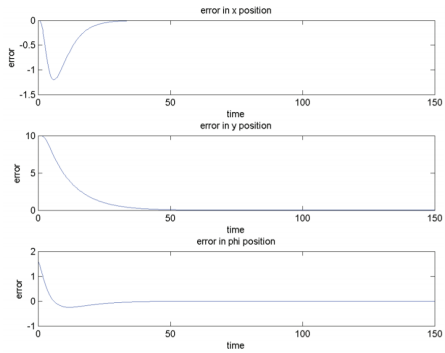


(b) Errors

Figure 6.5: Reproduced results



(a) Path



(b) Errors

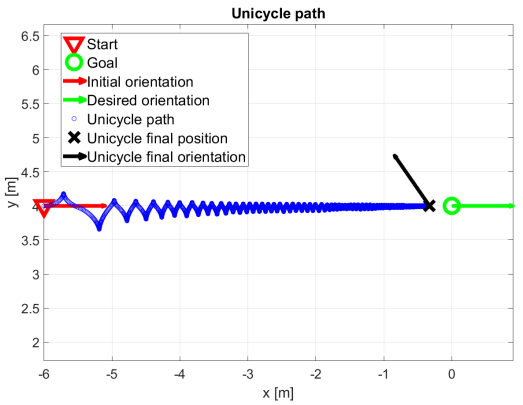
Figure 6.6: Paper results [27]

Test 2

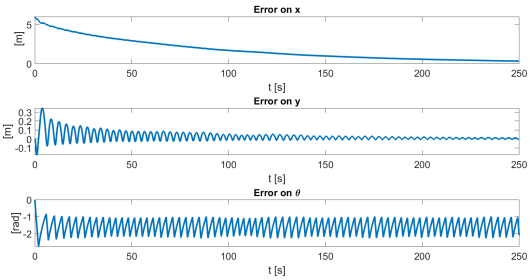
In this section a critical issue of this algorithm will be described. The algorithm will be tested in two situations very similar one to the other, showing two behaviours that are totally different. The constant k are the same used in Test 1; the starting pose is $\xi_i = [-6, +4, 0]^t$ for both the scenarios. The only difference is that in the first case, the final pose is $\xi_f^1 = [0,4,0]^T$, while in second one, it is $\xi_f^2 = [0,4.1,0]^T$. Ideally, in the first experiment, the unicycle should go straight; in the second it should steer to its left.

Figures 6.7 and 6.8 display the results of the experiments. As expected, the second scenario shows a smooth behaviour, with the unicycle steering to its left to reach the desired pose. The whole simulation lasts $250s$, but the robot is able to reach the final position in $60s$. Also the control inputs are feasible for the real system. The main issues arise in the first scenario: the initial and the final orientation are the same and, as a matter of fact, the error on θ at the time instant $t = 0$ is $\tilde{e}_\theta = 0$; but it immediately start to oscillate. A similar thing happen for $\tilde{e}_y$; moreover, the oscillatory behavior manifests itself also in the control inputs. This is caused by the fact that the angle γ_e is not bounded; in this case, it should be $\gamma_e = 0$, but it ends to be $\gamma_e = 2\pi$. The consequences are that the behaviour is totally wrong.

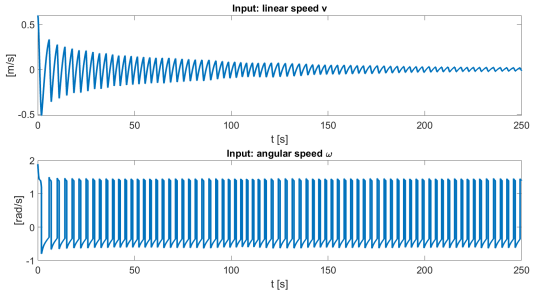
Bounding the angle γ_e and using the Equation 6.19 when $\gamma_e = 0$, we obtain the results shown in Figure 6.9. It is the expected behaviour, moreover, the control inputs are suitable for the QOLO robot and the unicycle reaches the desired pose in less than $50s$.



(a) Path

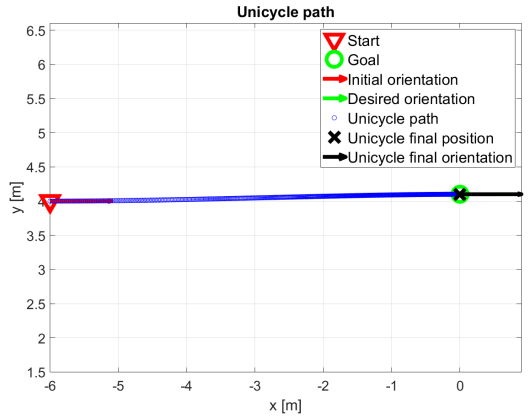


(b) Errors

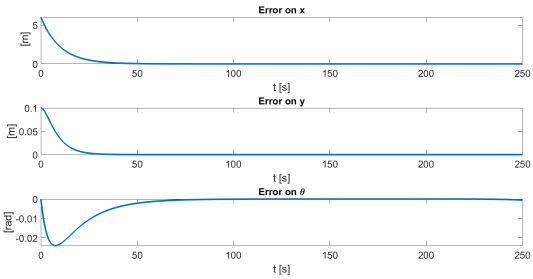


(c) Control inputs

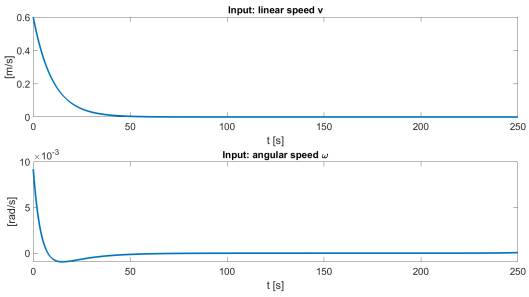
Figure 6.7: Stabilized Feedback - Test 2 - ξ_f^1



(a) Path

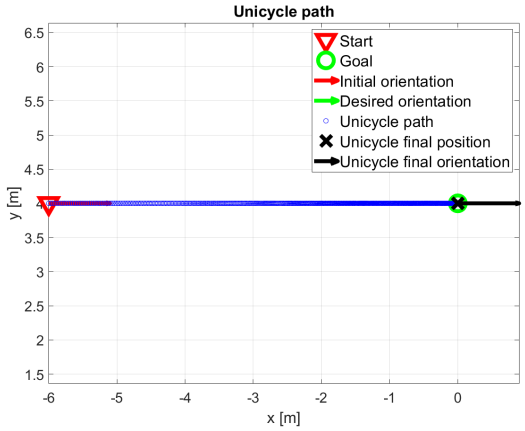


(b) Errors

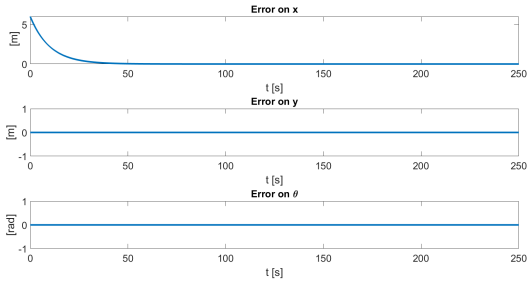


(c) Control inputs

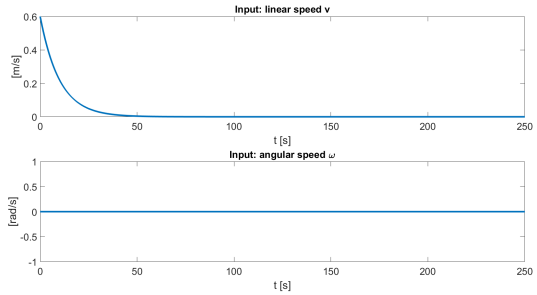
Figure 6.8: Stabilized Feedback - Test 2 - ξ_f^2



(a) Path



(b) Errors



(c) Control inputs

Figure 6.9: Stabilized Feedback - Test 2 - ξ_f^1 correct behaviour

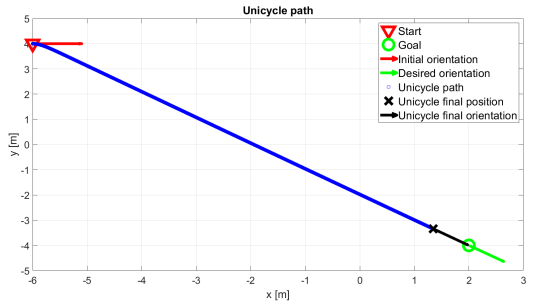
Test 3

In Test 1 and Test 2 we have verified that the algorithm works and produces suitable control inputs. Nevertheless, it is necessary to point out that the initial and the final state were distant one from the other. With this test, we try out the algorithm using it together with the high level controller, i.e. the modulated DS.

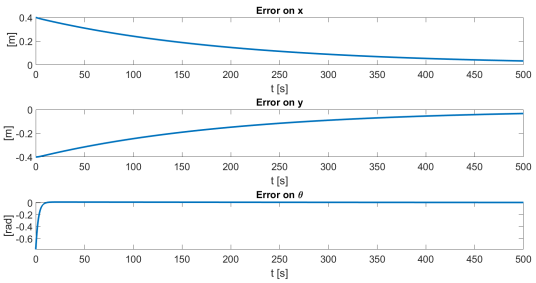
First of all, we consider an environment without obstacles, to see if the two controllers can work together.

Figure 6.10 displays the results of the simulation, using $k_1 = 0.1$, $k_2 = 0.3$ and $k_3 = 1.5$. The unicycle is able to follow the path obtained with the DS modulation, but it requires too much time, in this case $500s$. The control inputs produced are very small, therefore the movement is very limited at each time-step. Changing the values of k to $k_1 = 1.5$, $k_2 = 1.8$ and $k_3 = 2.5$, the robot behaviour is improved, as shown in Figure 6.11. After $80s$ the robot reaches the desired final position. Also the control inputs are feasible for the real implementation.

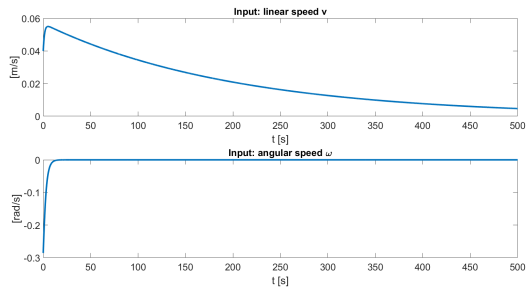
The union of these two controller may be a good solution for the final implementation, but we need to test first their behavior with the presence of multiple obstacles.



(a) Path

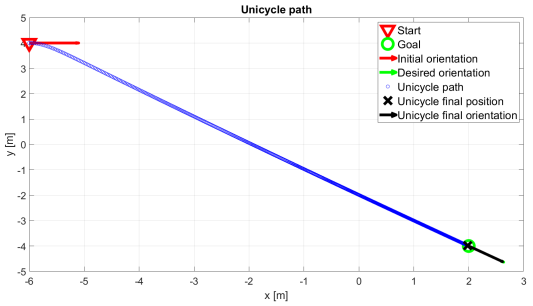


(b) Errors

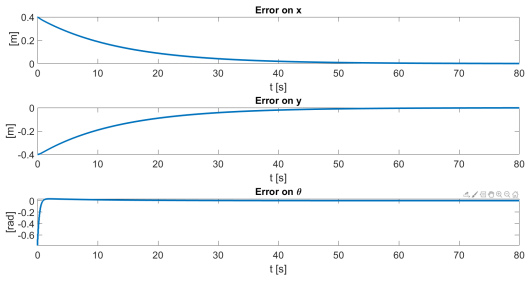


(c) Control inputs

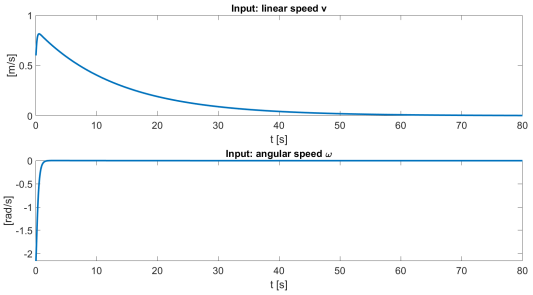
Figure 6.10: Stabilization Feedback - Test 3 - old k values



(a) Path



(b) Errors



(c) Control inputs

Figure 6.11: Stabilization Feedback - Test 3 - new k values

Test 4

In order to verify the benefits of this controller in combination with the modulating DS, a simulation in a static environment, with fixed obstacles and a fixed attractor, is performed. The robot starts from its initial pose $\xi_i = [0.0, -0.5, 0.0]^T$ and has to reach the desired final position $[x_f, y_f]^T = [10.0, 0.0]^T$.

Three ellipsoid shaped obstacles are present in the environment, in position C and with axes:

1. $C = [3.5, 0.1]^T$, $r_x = 0.3$, $r_y = 0.4$;
2. $C = [7, -1]^T$, $r_x = 0.3$, $r_y = 0.7$;
3. $C = [2, 0]^T$, $r_x = 0.7$, $r_y = 0.3$;

The constants K used are:

- $k_1 = 1.5$;
- $k_2 = 1.8$;
- $k_3 = 2.5$;

The simulation time is $T_{sim} = 25s$. It is necessary to point out that, in order to simulate a more realistic behaviour of the system, a random noise has been added both to the control inputs and to the robot state. The first random noise simulates the error caused by the motors of the robot; its maximum amplitude is the 5% of the maximum value for the control inputs:

- 0.05*1.5 = 0.075 m/s for the linear speed;
- 0.05*3.0 = 0.15 rad/s for the angular speed.

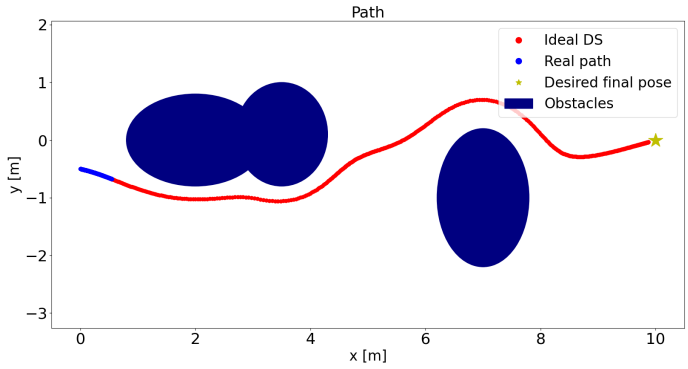
The noise on the robot state simulates the noise caused by the sensors, by the SLAM algorithm and by the delays that can occur. It is modeled using the Python function *numpy.random.normal()* [28] [29] [30] which describes a Gaussian distribution; it takes as inputs the mean and the standard deviation and outputs a value in relation with the probability distribution. The robot state is therefore perceived as:

- $x = numpy.random.normal(x,0.03)$;
- $y = numpy.random.normal(y,0.03)$;
- $\theta = numpy.random.normal(\theta,0.04)$;

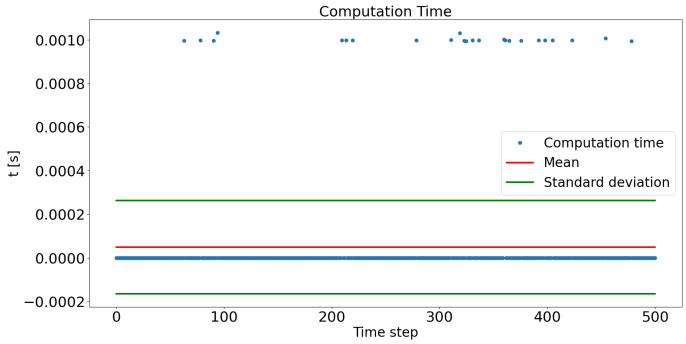
Figure 6.17 illustrates the path of the robot and the computation time. What emerges is that in 25 seconds the robot is not able to cover a sufficiently long path, making this method unusable, but further simulations should be done in order to prove it. On the contrary, the computation time is totally negligible, which means that the controller could run in real-time without any problem.

The control inputs produced by the controller and shown in Figure 6.19 present a very bad behaviour, mainly for two reasons:

- The control inputs are very irregular and their values are very low;
- The derivatives of the control inputs, which represent the linear and angular accelerations are too high to be used on the real robot.

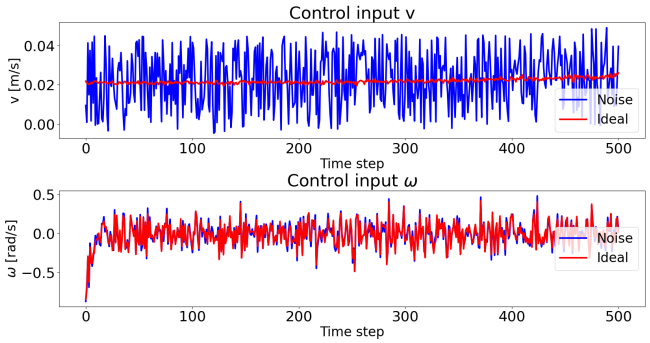


(a) Path

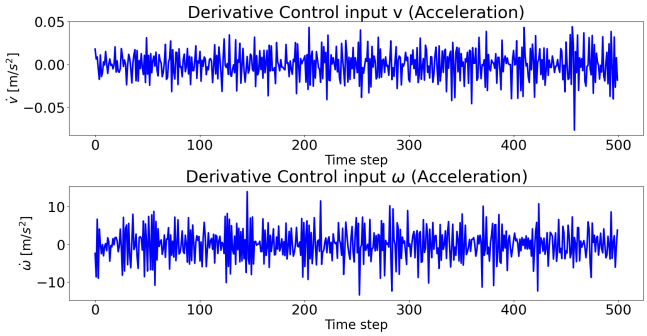


(b) Computation Time

Figure 6.12: Stabilization Feedback - Test 4 - K = [1.5 , 1.8, 2.5]



(a) Control inputs



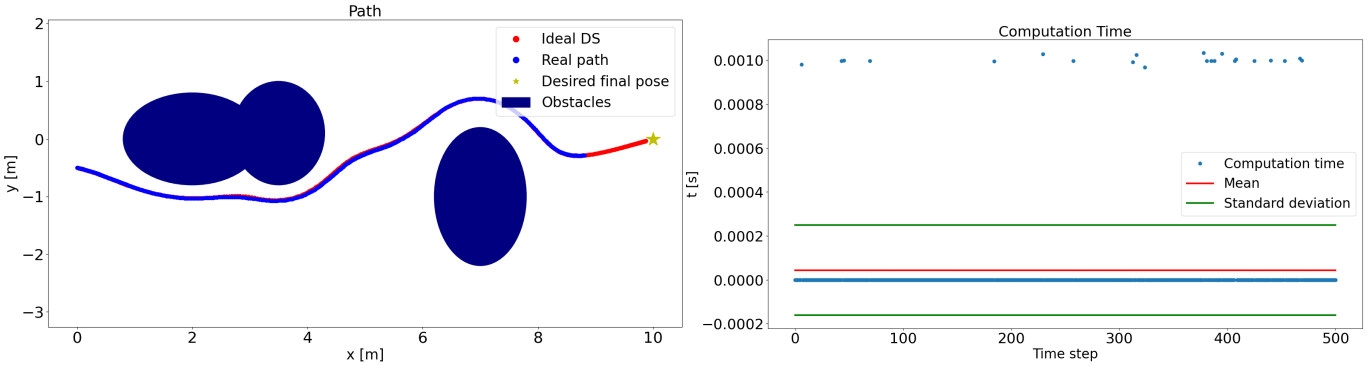
(b) Control inputs derivative

Figure 6.13: Stabilization Feedback - Test 4 - K = [1.5 , 1.8, 2.5] - Control Inputs

In order to see how the robot behaviour changes when the K parameters are varied, we set them as: The constants K used are:

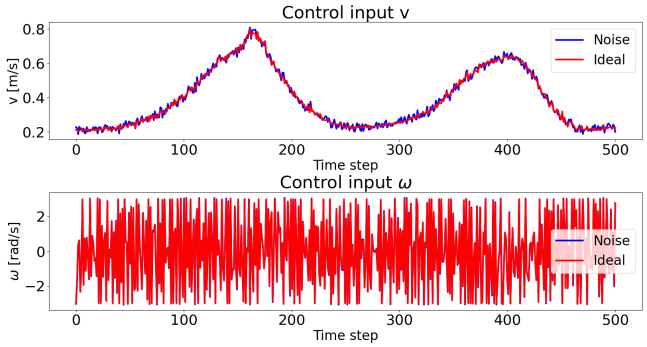
- $k_1 = 15$;
- $k_2 = 18$;
- $k_3 = 2.5$;

In this simulation, the robot is able to follow the desired path and cover a longer distance and the computation time is still negligible. The main problems are again connected to the control inputs, especially the angular velocity ω: it is very irregular and, even if the boundaries on the maximum speed are respected, the acceleration is totally incompatible with the robot implementation.

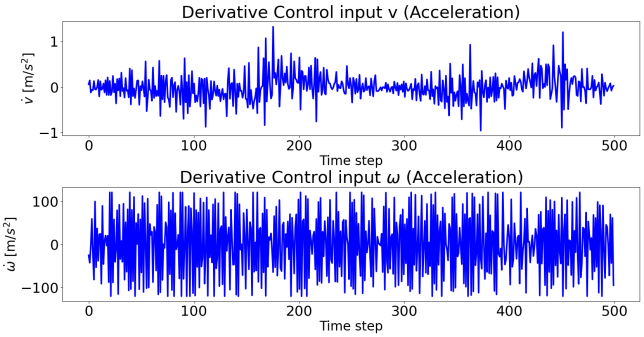


(a) Path **(b)** Computation Time

Figure 6.14: Stabilization Feedback - Test 4 - K = [15 , 18, 2.5]



(a) Control inputs



(b) Control inputs derivative

Figure 6.15: Stabilization Feedback - Test 4 - K = [15 , 18, 2.5] - Control Inputs

We have seen that using small constants K the path covered is short, but the accelerations constraints are more observed; while, using bigger constants K, the path is more consistent with the desired one, but the inputs accelerations are inconsistent for the real application.

A grid search analysis has been done in order to better understand the correlation between the constants K and the control inputs, especially their derivatives and the effect that they produce on the system. All the three parameters have been tested using the values [0.1 , 4.37, 8.64, 12.91, 17.19, 21.46, 25.73, 30]; which are eight equally spaced values between 0.1 and 30. A total of $8^3 = 512$ tests have been run, each of them having $T_{sim} = 25s$ and describing the same scenario. For every test, two cost functions have been computed:

1. Distance Cost: it is the sum of the distances between the robot and the final position at each time step. A smaller value of this cost function indicates that the robot is able to arrive near the desired position within the 25 seconds.

2. Derivative Cost: we have seen that the accelerations of the control inputs in the previous tests exceed the limits imposed by the robot. For each test we compute the accelerations as:

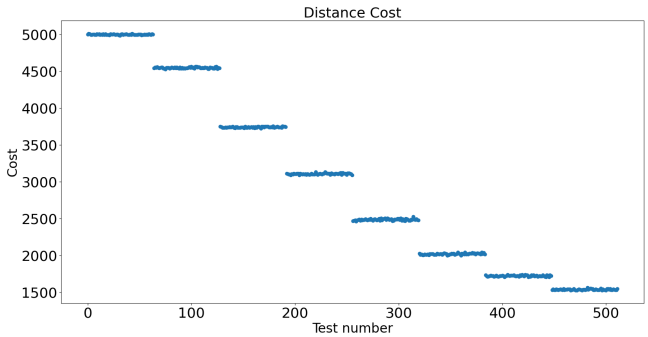
$$Acceleration = \frac{Input(t) - Input(t-1)}{dt} \tag{6.20}$$

with $dt = 0.05$ that represents the time step duration. The derivative cost is computed as:

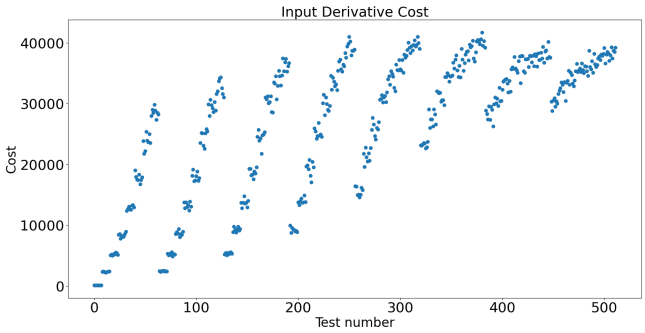
$$Derivative\ Cost = \sum \|Acceleration_v\| + \sum \|Acceleration_\omega\| \tag{6.21}$$

Figure 6.16 illustrates the result of the grid search. The aim of the grid search is to find a set of constant K for which the Distance Cost and the Derivative Cost have both small values. Having a small value on the Distance Cost means that the robot is able to reach the desired position; while, a small value on the Derivative Cost implies that the accelerations are restrained, which does not means that they observe the limits imposed by the robot, but that they are more suitable. From the Figure, it appears that the smaller is the cost of the distance from the desired position, the bigger is the cost for the accelerations.

The grid search proves that with this method, it is not possible to guarantee a good performance while observing the restriction imposed by the structure of the system.



(a) Distance Cost



(b) Derivative Cost

Figure 6.16: Stabilization Feedback - Grid Search

Considerations

The Stabilization Feedback algorithm shows good results in the production of control inputs needed:

- when it is used alone to reach a desired position when the initial and final position are quite distant;
- in combination with the DS when the movements are quite simple.

Using this algorithm together with the DS modulation, therefore to avoid the obstacles, has exhibited many issues: in simulation the kinematics of the robot does not take into account the real limitations on the accelerations, hence, the simulated robot is able to follow the desired path using the saturated control inputs, but exploiting accelerations that could not be implemented on the real robot. Consequently, this algorithm can not be used as low level controller and other solutions must be analyzed.

6.3 Model Predictive Control - Overview [31]

Among the advanced techniques for process control, the Model Predictive Control (MPC) is one of the most performing, but at the same time one of the most complex to understand and implement.

It is a digital controller and its most important characteristic concerns the fact that it is possible to satisfy a set of constraints. The MPC's main strength is the optimization: considering a finite time-horizon, the MPC optimizes N steps ahead in the future, anticipating the events and taking proper control actions. Of the N control inputs it then uses only the control inputs of the current time-step. Then, it optimizes again in the next step, repeatedly.

Model predictive controllers rely on dynamic models of the process, most often linear empirical models obtained by system identification.

The general formulation for a MPC is

$$\begin{aligned} & u^*(\xi) := argmin \sum_{i=0}^{N-1} (\xi_i^T Q \xi_i + u_i^T R u_i) + \xi_N^T Q_N \xi_N \\ & \xi_0 = \xi_m && measure \\ & \xi_{i+1} = f(\xi_i, u_i) && dynamics \\ & g(\xi_i, u_i) \leq 0 && constraints \\ & h(\xi_N) \leq 0 && constraints\ on\ final\ state \end{aligned} \tag{6.22}$$

The matrices R, Q and Q_N allow us to determine a weight for each state and control variable. They have to be tuned depending on the application. With ξ_m we identify the last available measure of the state, so the MPC starts its computation from a defined state. The function $f(\xi, u)$ is the dynamics of the system, it is generally linear; nevertheless, the MPC can deal also with non-linear systems, making this approach incredibly flexible. With $g(\xi_i, u_i)$ and $h(\xi_N)$, the constraints are defined; in particular, $g(\xi_i, u_i)$ defines all the boundaries on the dynamics during the evolution of the system for N prediction steps , while $h(\xi_N)$ describes the constraints after the N-th step, ideally to infinite. The output is $u^*(\xi)$, which are the N control inputs optimized over the horizon time.

The Non-linear MPC allows to obtain better performances with respect to a linear MPC, since it takes into account the full dynamic of the system, at the expense of the computational cost. Due to its complexity, the non-linear MPC is, in general, slower than the linear MPC.

Remembering Equation 3.3, the unicycle model is described by the differential

equation:

$$\dot{\xi} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = G(\xi) \cdot u = \begin{bmatrix} cos(\theta) & 0 \\ sin(\theta) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{6.23}$$

This system is non-linear, so it is possible to use a non-linear MPC to obtain an optimal performance. Being the MPC a digital controller, it is necessary to discretize the continuous non-linear system. This can be done using either the Euler method or the Runge-Kutta4 method. Considering a continuous function

$$\dot{y} = f(t, y) \tag{6.24}$$

with $y(t_0) = y_0$, we want to have a discretization step h.

Euler method [32] [33]
Euler method is a numerical procedure of the first order used to solve ordinary differential equation (ODE) systems, starting from an initial value. It is the simplest method among the Runge-Kutta algorithms.

It can be used to approximate the continuous functions in Equation 6.24 with y_{n+1}:

$$\begin{aligned} y_{n+1} &= y_n + hf(t_n, y_n) \\ t_{n+1} &= t_n + h \end{aligned} \tag{6.25}$$

Its simplicity makes this algorithm very fast to use, but we have to pay particular attention in the choice of the discretization step h: if this value is too large, the approximation is completely different from the original function, causing the algorithm to be unusable. On the contrary, choosing a value that is too small, may slow down the computation and, therefore, it could not be used in real-time implementation. Moreover, we have to remember that in real application, the discretization time is constrained by the hardware and by the software that we are using.

Runge-Kutta [34]
As the Euler method, the Runge-Kutta algorithms are a family of methods for the approximations of ODE systems. In this case, using the 4-th order approximation, the result is more consistent with the real shape of the continuous function. Moreover, the problems related to the discretization time of the Euler method have less effect; in this way we could use a larger value for h, without compromising the approximation. The disadvantage of this method lies in the computation time that it requires to run. Therefore, also in this case, it is necessary to consider carefully the discretization time and the hardware where the algorithm will run on.

The approximation y_{n+1} is computed:

$$\begin{aligned} y_{n+1} &= y_n + \tfrac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\ t_{n+1} &= t_n + h \end{aligned} \tag{6.26}$$

With:

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + \tfrac{h}{2}, y_n + \tfrac{1}{2}k_1 h) \\ k_3 &= f(t_n + \tfrac{h}{2}, y_n + \tfrac{1}{2}k_2 h) \\ k_4 &= f(t_n + h, y_n + k_3 h) \end{aligned} \tag{6.27}$$

Where:

- k_1 is the increment of the slope at the beginning of the time interval, like in the Euler method;
- k_2 is the increment of the slope at half of the time interval, computed using k_1;
- k_3 is the increment of the slope at half of the time interval, computed using k_2;
- k_4 is the increment of the slope at the end of the time interval.

The implementation of the MPC controllers has been done both in Matlab and Python. The optimization software used is **Casadi** [35], that allows very good results and high timing performances.

6.4 Non-linear MPC - Position control

The purpose of this low-level controller is to convert the output of the DS modulation to suitable inputs for the robot. Here, we are going to consider the complete kinematics of the systems, therefore a non-linear MPC will be used. The strategy adopted is the one illustrated in Algorithm 6

Algorithm 6 nMPC - Position control.

1: **while** System is working **do**
2: ▷ Obtain the robot state ξ
3: ▷ Obtain information about the obstacles
4: ▷ Compute the desired velocity with the DS modulation
5: ▷ Integrate the desired velocity to obtain the desired next state ξ_{des}
6: ▷ Compute the desired orientation
7: ▷ Optimize the control inputs using the nMPC
8: ▷ Use $N_{control}$ inputs to control the robot
9: **end while**

In the non-linear MPC, the variables to be optimized are the state trajectory, that we will identify with X, which has dimension $3 \times N + 1$, and the control inputs U, with dimension $2 \times N$.

$$X = \begin{bmatrix} x_0 \; x_1 \; x_2 \; ... \; x_N \\ y_0 \; y_1 \; y_2 \; ... \; y_N \\ \theta_0 \; \theta_1 \; \theta_2 \; ... \; \theta_N \end{bmatrix} \tag{6.28}$$

$$U = \begin{bmatrix} v_0 \; v_1 \; v_2 \; ... \; v_{N-1} \\ \omega_0 \; \omega_1 \; \omega_2 \; ... \; \omega_{N-1} \end{bmatrix} \tag{6.29}$$

The cost function to minimize is:

$$\sum_{i=0}^{N} (X_i - X_{ref})^T G_x (X_i - X_{ref}) + \sum_{i=0}^{N-1} U_i^T G_u U_i \tag{6.30}$$

The reference state that we want to reach is $X_{ref} = \xi_{ref} = [x_{ref}, y_{ref}, \theta_{ref}]^T$; it is obtained through the high level controller that give us the desired next pose using the modulating DS. Knowing the robot state $\xi = [x, y, \theta]^T$ and the modulating DS $[\dot{x}, \dot{u}]^T$, we compute

$$\begin{bmatrix} x_{ref} \\ y_{ref} \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + dt \cdot \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \tag{6.31}$$

$$\theta_{ref} = arctan2(\dot{y}, \dot{x}) \tag{6.32}$$

G_x and G_u are diagonal matrices composed by the gains that we have to set, depending on the importance of the variable. These gains have to be tuned.

Then we have to define the constraints on the state, on the inputs and the ones derived by the system's dynamics.

We do not have any constraints on the state X, while we have limitations on the linear speed and acceleration and on the angular speed and acceleration. For the linear and angular acceleration we have to approximate: assuming the acceleration constraint as $|\dot{U}| < acc\_constraints$, we can approximate as

$$|U_k - U_{k-1}| < h \cdot acc\_constraints \tag{6.33}$$

where k is the time instant and h is the sampling time.

It is also possible to constraint the jerk of the robot, which is linked to the derivative of the acceleration: in this case we need to approximate the second derivative of the control inputs:

$$|U_{k+1} - 2 \cdot U_k + U_{k-1}| < h^2 \cdot jerk\_constraints \tag{6.34}$$

Moreover, we have to take into account the last control inputs of the previous time-step, passing them as U_{prev}. Then, of course, we have to include the constraints given by the dynamics of the system, using the discrete function computed at each prediction step. This makes the nMPC more complex and slower with respect to a linear MPC.

We have to constraint the optimization variable X so that $X_0 = \xi_0$, which is the initial pose of the robot from the point of view of the nMPC. Basically, X_0 must be the pose of the robot at the time-step at which we use the nMPC.

To sum up, the equations that we are going to use in the nMPC algorithm are:

$$-\infty < X < \infty \tag{6.35}$$

$$-speed\_constraints < U < speed\_constraints \tag{6.36}$$

$$-h \cdot acc\_constraints < U_0 - U_{prev} < h \cdot acc\_constraints \tag{6.37}$$

$$-h \cdot acc\_constraints < U_{i+1} - U_i < h \cdot acc\_constraints \tag{6.38}$$

With $i \in [0, N-2]$.

$$-h^2 \cdot jerk\_constraints < U_1 - 2 \cdot U_0 + U_{prev} < h^2 \cdot jerk\_constraints \tag{6.39}$$

$$-h^2 \cdot jerk\_constraints < U_{i+1} - 2 \cdot U_i + u_{k-1} < h^2 \cdot jerk\_constraints \quad (6.40)$$

With $i \in [1, N-2]$.

$$X_0 = \xi_0 \quad (6.41)$$

$$X_{i+1} = f_{discrete}(X_i, U_i) \quad (6.42)$$

With $i \in [0, N-1]$.

All the constraints can be easily changed, but they are strictly related to the kinematics under consideration, other than to the structure of the robot and to the limits imposed by the sensation of comfort that the user should perceive. In Table 6.1 the values used for the contrarians are listed.

Parameter	**Value**
speed_constraints linear speed	1.5 m/s
speed_constraints angular speed	3 rad/s
acc_constraints liner acceleration	1.5 m/s^2
acc_constraints angular acceleration	3.5 rad/s^2
jerk_constraints liner jerk	0.1 m/s^3
jerk_constraints angular jerk	0.05 rad/s^3

Table 6.1: Non-linear MPC - Constraints

The outputs of the nMPC are N control inputs. Generally, only the first inputs are used and then the optimization is computed again. It would be possible to use also the other control inputs, in this case, though, the control would be in open loop.

Depending on the control frequency $f_{control}$, i.e. the frequency at which we control the motors of the robot, and on the high level control frequency f_{DS} that gives us the reference pose for the nMPC, it would be possible to use N_{inputs} control

inputs, with

$$N_{inputs} = \frac{f_{control}}{f_{DS}} \tag{6.43}$$

With the robot that we are going to consider, it would be possible to have $f_{control} = 400\ Hz$, but having $f_{DS} = 20\ Hz$ would produce $N_{inputs} = 20$; which means 20 control inputs in open-loop. Therefore, the frequencies that we will consider are listed in Table 6.2.

Parameter	**Value**
f_{DS}	$20\ Hz$
$f_{control}$	$20\ Hz$
N_{inputs}	1

Table 6.2: Non-linear MPC - Frequencies

6.4.1 Algorithm tests

Here we present some tests useful to verify the applicability of the non-linear Model Predictive Control with Position control and the modulated DS.

Test 1

A first scenario where this controller can be tested is a static environment with fixed obstacles and a fixed attractor. The robot starts from its initial pose $\xi_i = [0.0, -0.5, 0.0]^T$ and has to reach the desired final position $[x_f, y_f]^T = [10.0, 0.0]^T$ combining the DS modulation and the non-linear MPC.

Three ellipsoid shaped obstacles are present in the environment, in position C and with axes:

1. $C = [3.5, 0.1]^T$, $r_x = 0.3$, $r_y = 0.4$;
2. $C = [7, -1]^T$, $r_x = 0.3$, $r_y = 0.7$;
3. $C = [2, 0]^T$, $r_x = 0.7$, $r_y = 0.3$;

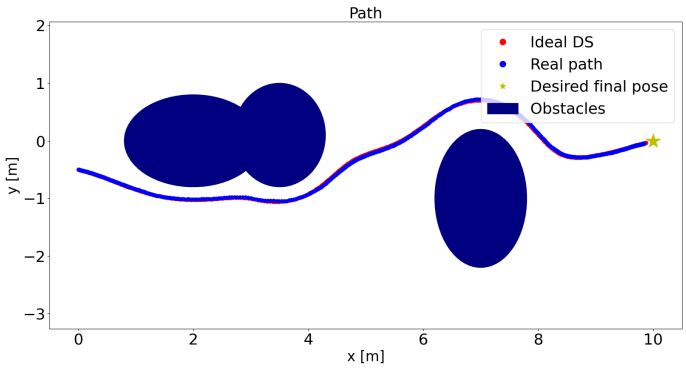
The non-linear MPC gives us the possibility to impose constraints on the maximum speeds, on the accelerations and on the jerk of the robot. In this test, only the constraints on the maximum speeds are taken into account, as shown in Table 6.1.

The gain matrices used for this simulation are:

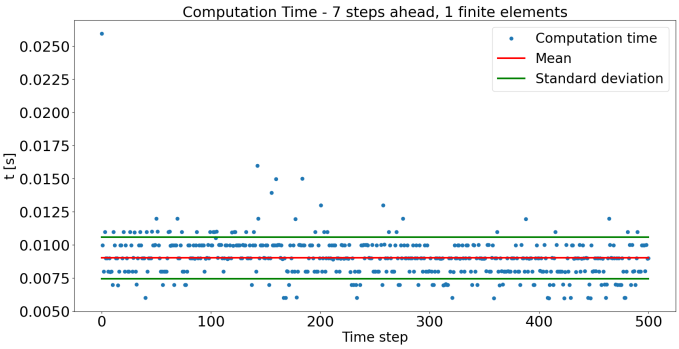
$$G_x = Q^T \cdot Q = Q \cdot Q \qquad G_u = R^T \cdot R = R \cdot R \tag{6.44}$$

$$Q = \begin{bmatrix} 10 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 0.1 \end{bmatrix} \qquad R = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{6.45}$$

The simulation time is $T_{sim} = 25s$. Figure 6.17 illustrates the path of the robot and the computation time. Both of them are promising results because the robot is able to follow the ideal path generated by the DS alone, without considering the kinematics of the robot. Moreover, the computation time is always smaller than 0.05 seconds, that we consider as the limit for the algorithm. This limit is caused by the control frequency of the robot, which is $20Hz$, therefore, $0.05s$ is the time between a control action and the next one. The control algorithm must be faster than this limit in order to output the control inputs correctly for the time step taken into account.



(a) Path



(b) Computation Time

Figure 6.17: nMPC - Position control - Test 1

As explained in Section 6.2.1, in order to simulate a more realistic behaviour of the system, a random noise has been added both to the control inputs and to the robot state. The first random noise simulates the error caused by the motors of the robot; its maximum amplitude is the 5% of the maximum value for the control inputs:

- 0.05*1.5 = 0.075 m/s for the linear speed;
- 0.05*3.0 = 0.15 rad/s for the angular speed.

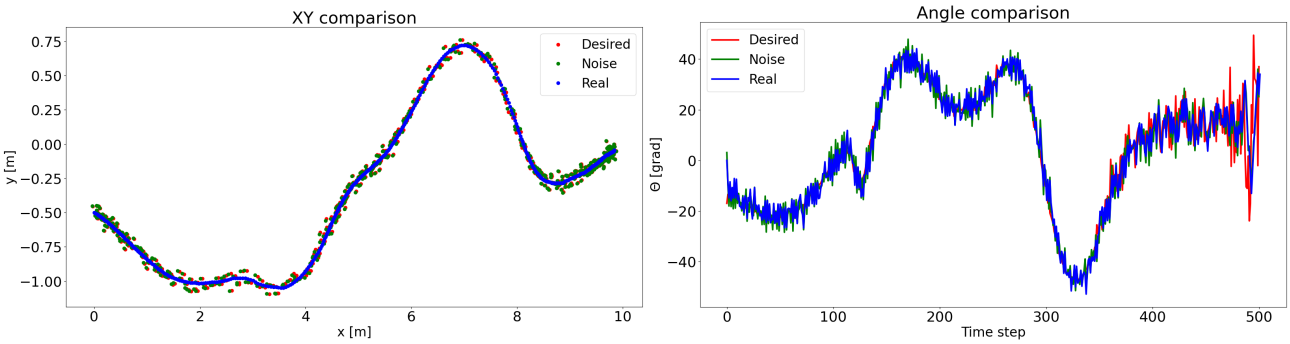
The noise on the robot state simulates the noise caused by the sensors, by the SLAM algorithm and by the delays that can occur. It is modeled using a Gaussian distribution and the robot state is therefore perceived as:

- $x = numpy.random.normal(x,0.03)$;
- $y = numpy.random.normal(y,0.03)$;
- $\theta = numpy.random.normal(\theta,0.04)$;

Figure 6.18 displays the real path of the robot, the perceived one (i.e. the noisy positions) and the desired next position starting from the perceived position. What emerges is that the DS modulation and the non-linear MPC are very robust against the noise because the control inputs obtained by the optimization, starting from the information given by the DS, produce a behaviour that is quite smooth and consistent with the desired path.

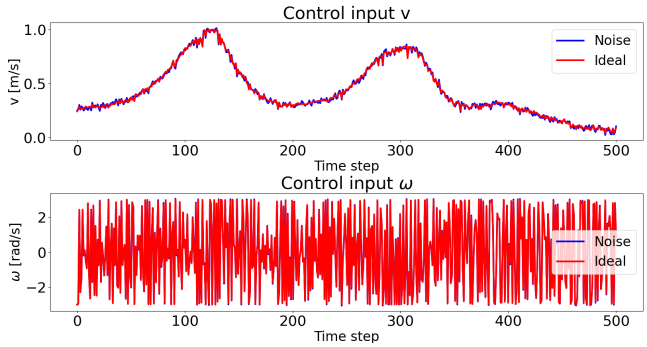
This behaviour has been obtained without considering some important constraints that affect the user's experience. Theoretically, the final outcome of the system can still be comfortable, but we need to analyze the control inputs to understand it.

In Figure 6.19 it is possible to notice immediately that there is an undesired behaviour in the control input ω. The constraints on the maximum angular speed allowed are respected, but this control input changes a lot in a very rapid manner. This would cause the robot to oscillate and to produce a very bad experience for the user.

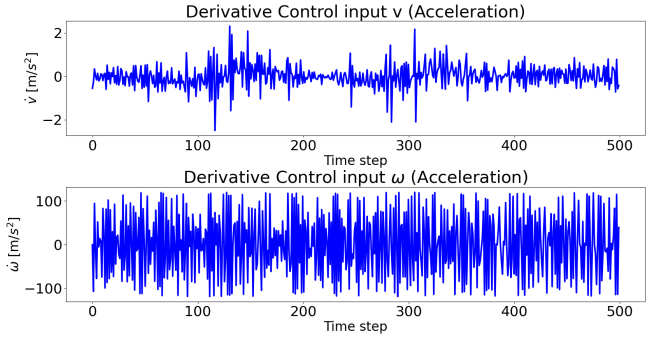


(a) Path with noise **(b)** Angle with noise

Figure 6.18: nMPC - Position control - Test 1 - Noise analysis



(a) Control inputs



(b) Control inputs derivative

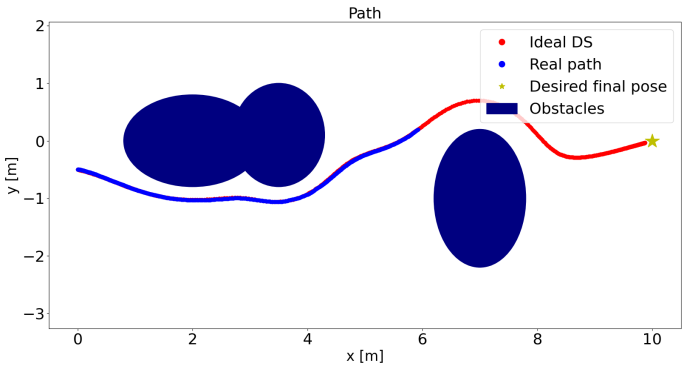
Figure 6.19: nMPC - Position control - Test 1 - Control Inputs

Test 2

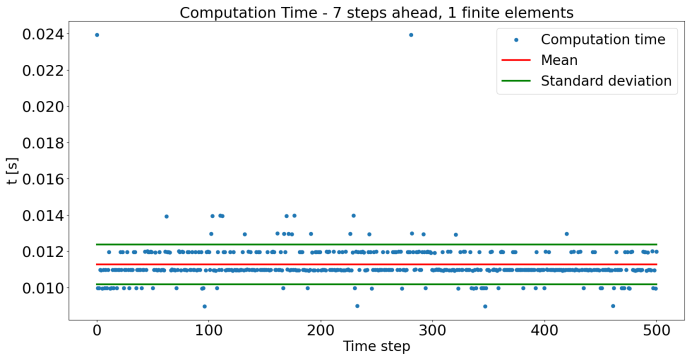
The previous simulation has shown that the non-linear MPC used in combination with the DS modulation produces good results, but the control inputs can not be used in the real implementation. In this test, the scenario and all the parameters are the same, but we will add the constraints on the accelerations. For the moment, the constraints on the jerking are still ignored.

Figure 6.20 displays the path and the computation timing. The first thing that stands out is that the robot is not able to reach the desired position. The simulation time is the same as in the previous test, but the constraints on the accelerations, slow down the whole system.

As for the control inputs, Figure 6.19 shows that the control input ω presents a behaviour that should be avoided, even if the acceleration constraints are observed. Having that kind of control inputs do not guarantee a nice user experience, for this reason the jerking constraints must be taken into account. Nevertheless, considering the results of this test, the addition of other constraints will probably worsen the behaviour of the robot, rather than improving it.

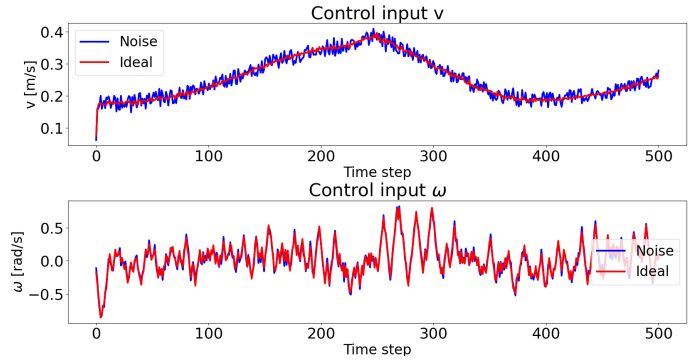


(a) Path

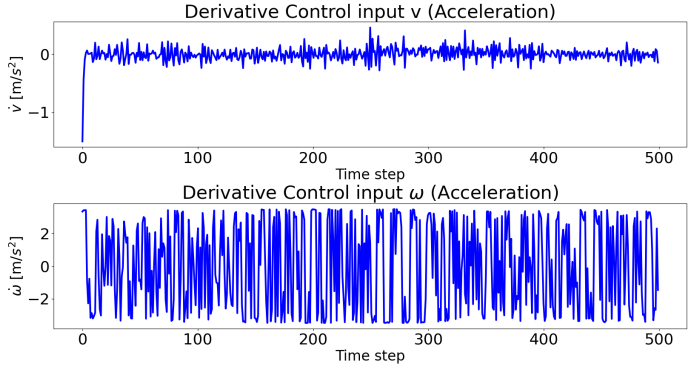


(b) Computation Time

Figure 6.20: nMPC - Position control - Test 2



(a) Control inputs



(b) Control inputs derivative

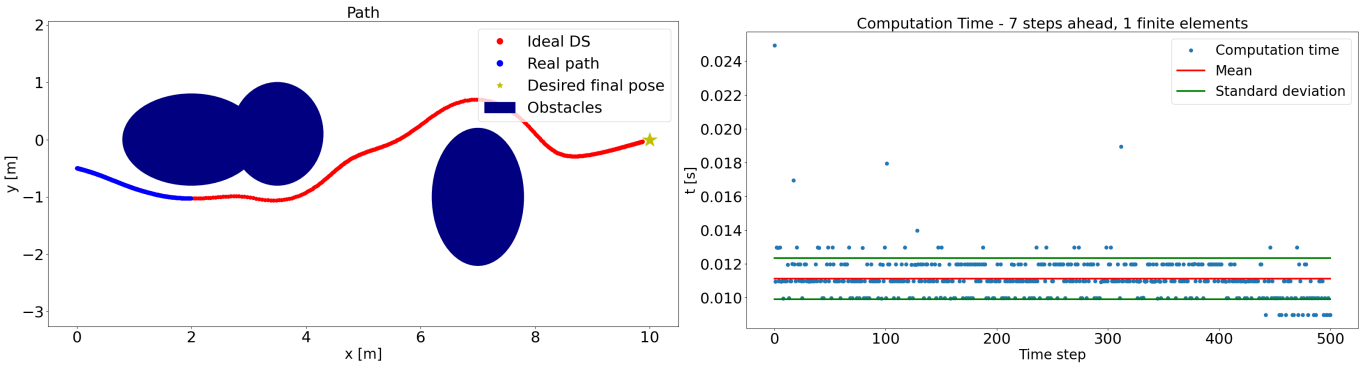
Figure 6.21: nMPC - Position control - Test 2 - Control Inputs

Test 3

Following Test 1 and Test 2, also in this simulation the scenario is the same, as the parameters used to define the nMPC. In addition, though, the jerking constraints are present, in order to see if a consistent improvement is accomplished. Figure 6.22 displays the results.

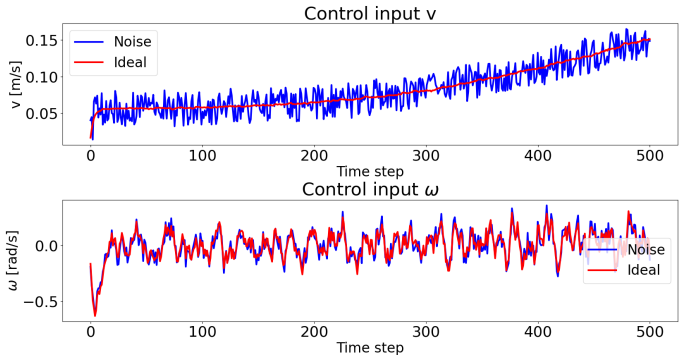
As expected, the robot's behaviour gets worse: as a matter of fact, in the same time it is able to cover more or less 2 meters, while in the first test, the path covered was more than 10 meters long. Of course, the DS modulation works properly, but the non-linear MPC is not able to produce control inputs adequate for this scenario.

If we compare the control inputs in Figure 6.23 with the ones obtained in the previous test in Figure 6.21, a slight improvement can be seen, but also in this case, using those inputs on the robot, especially ω, would cause the robot to oscillate while moving, producing an unpleasant sensation.

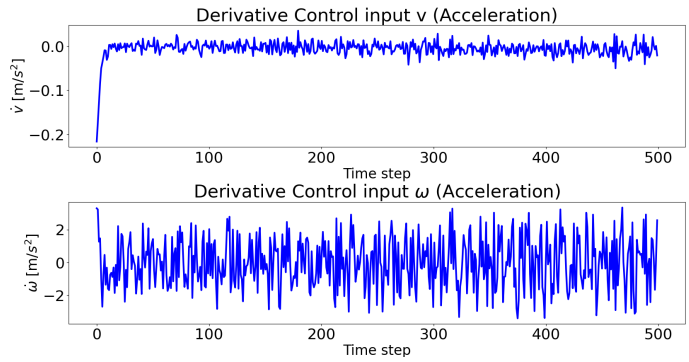


(a) Path **(b)** Computation Time

Figure 6.22: nMPC - Position control - Test 3



(a) Control inputs



(b) Control inputs derivative

Figure 6.23: nMPC - Position control - Test 3 - Control Inputs

Considerations

The low level controller illustrated here shows several strengths, such as the possibility to bound the control inputs without saturating them. It is also able to work together with the high level controller and avoid the obstacles that are on the path of the robot. The controller respects also the timing constraints of the robot's hardware.

The main disadvantages, though, are linked with the inability of the robot to accomplish the path in a reasonable time. Moreover, even with the most restricted limits, the control inputs are very rough producing a very disturbing experience.

To conclude, this controller help us understanding the method of operation of the non-linear Model Predictive Control, its limitations and its strengths. In the next section, a modified version of the non-linear MPC will be presented, that starts from this controller and tries to overcome its main issues.

6.5 Non-linear MPC - Velocity control

In this sections, as in the Section 6.4, the complete kinematics of the system is considered, therefore, we will present another non-linear MPC controller. Differently from the nMPC with position control, the controller will use directly the modulated velocity to optimize the control inputs. Algorithm 7 illustrates the control strategy.

The state trajectory X has dimension $3 \times N + 1$, and the control inputs U have dimension $2 \times N$. These are the variables used for the optimization; as in Equations 6.28 and 6.29 they are defined as

$$X = \begin{bmatrix} x_0 \; x_1 \; x_2 \; ... \; x_N \\ y_0 \; y_1 \; y_2 \; ... \; y_N \\ \theta_0 \; \theta_1 \; \theta_2 \; ... \; \theta_N \end{bmatrix} \tag{6.46}$$

$$U = \begin{bmatrix} v_0 \; v_1 \; v_2 \; ... \; v_{N-1} \\ \omega_0 \; \omega_1 \; \omega_2 \; ... \; \omega_{N-1} \end{bmatrix} \tag{6.47}$$

Algorithm 7 nMPC - Position control.

1: **while** System is working **do**
2: ▷ Obtain the robot state ξ
3: ▷ Obtain information about the obstacles
4: ▷ Compute the desired velocity with the DS modulation
5: ▷ Compute the desired orientation
6: ▷ Optimize the control inputs using the nMPC
7: ▷ Use $N_{control}$ inputs to control the robot
8: **end while**

The cost function to minimize is:

$$\begin{aligned} &\textstyle\sum_{i=0}^{N}(\theta_i - \theta_{ref})^T G_\theta (\theta_i - \theta_{ref}) \\ &+ \textstyle\sum_{i=0}^{N-1}[(v_i \cdot cos(\theta_i) - V_{ref}x)^T G_{Vx}(v_i \cdot cos(\theta_i) - V_{ref}x) \\ &+ (v_i \cdot sin(\theta_i) - V_{ref}y)^T G_{Vy}(v_i \cdot sin(\theta_i) - V_{ref}y)] \\ &+ \textstyle\sum_{i=0}^{N-1} U_i^T G_u U_i \end{aligned} \tag{6.48}$$

G_θ, G_{Vx}, G_{Vy} and G_u are gain matrices that we have to set, depending on the importance of the variable and have to be tuned.

$V_{ref}x$ and $V_{ref}y$ are the components of $V_{ref} = [\dot{x}, \dot{y}]^T$, which is the reference velocity that we want the robot to have; it is obtained through the high level controller, using the modulating DS.

θ_{ref} is the desired heading angle and it can be obtained in different ways:

1. Using the modulated velocity, we can compute the angle that the vector forms with the fixed frame.
$$\theta_{ref} = arctan2(\dot{y}, \dot{x}) \tag{6.49}$$
The problem of this approach is that, the heading angle of the robot is forced to be the same of the velocity vector, hence, the robot is not able to go backwards, but tends to rotate completely when the velocity points to the opposite site with respect to the head of the robot.

2. Using the velocity of the target.
$$\theta_{ref} = arctan2(\dot{y}_{target}, \dot{x}_{target}) \tag{6.50}$$
In this way the robot tries to have always the same heading angle of the target that wants to follow. This method is very useful when the robot is already very close to the target and does not have to perform abrupt movements. During the obstacle avoidance it could cause some problems because the heading angle of the target may be very different from the angle of the modulated velocity, causing the robot to get stuck near a fixed obstacle. Moreover, it really depends on the weight G_θ that we chose: a smaller weight implies that the real heading angle can differ quite a lot from the desired one, while a larger weight constraints the real heading angle to be very close to desired one.

3. Using a combination of the previous methods. When the robot is very close to the target, it would be better to use as reference the angle obtained from the velocity of the target itself. On the contrary, if the robot has to reach the attractor or if there are some obstacle to be avoided, the modulated velocity angle is more suitable. Nevertheless, there is a situation that would be better to be avoided: we are already using the information of the desired velocity to impose the direction of movement; using also the information of the orientation of this vector could cause the robot to spin in order to follow the desired velocity with the head of the robot in the same direction. This behaviour can be unnecessary and potentially dangerous because the robot could lose track of its position and could collide with the obstacles in the environment.

   In order to solve this issue, the sign of θ_{mod}, which is the angle obtained from the modulation, is changed if the difference between it and θ_{target} is bigger than 180ř, with θ_{target} that represents the heading angle of the target.
$$\theta_{target} = arctan2(\dot{y}_{target}, \dot{x}_{target})$$

$$\theta_{mod} = arctan2(\dot{y}, \dot{x})$$

If $\|\theta_{mod} - \theta_{target}\| > \frac{\pi}{2}$, $\theta_{mod} = -\theta_{mod}$

$$\theta_{ref} = \theta_{mod} \cdot \frac{1}{1 + e^{-s \cdot d}} + \theta_{target} \cdot \frac{1}{1 + e^{s \cdot d}} \tag{6.51}$$

The function in Equation 6.52 is called Sigmoid function and it is used to guarantee a smooth transition between θ_{mod} and θ_{target}. The parameter s is the slope of the function; we can use it to define how smooth the transition should be, as shown in Figure 6.24. The parameter d is defined as in Equation 6.53; the vector $[x_{attr}, y_{attr}]^T$ is position of the moving local attractor and n is a value that represents the distance at which the sigmoid function is 0.5. We can use n and s to define the distance at which the sigmoid starts the transition process.

$$\frac{1}{1 + e^{\pm s \cdot d}} \tag{6.52}$$

$$d = \left\| \begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} x_{attr} \\ y_{attr} \end{bmatrix} \right\| - n \tag{6.53}$$

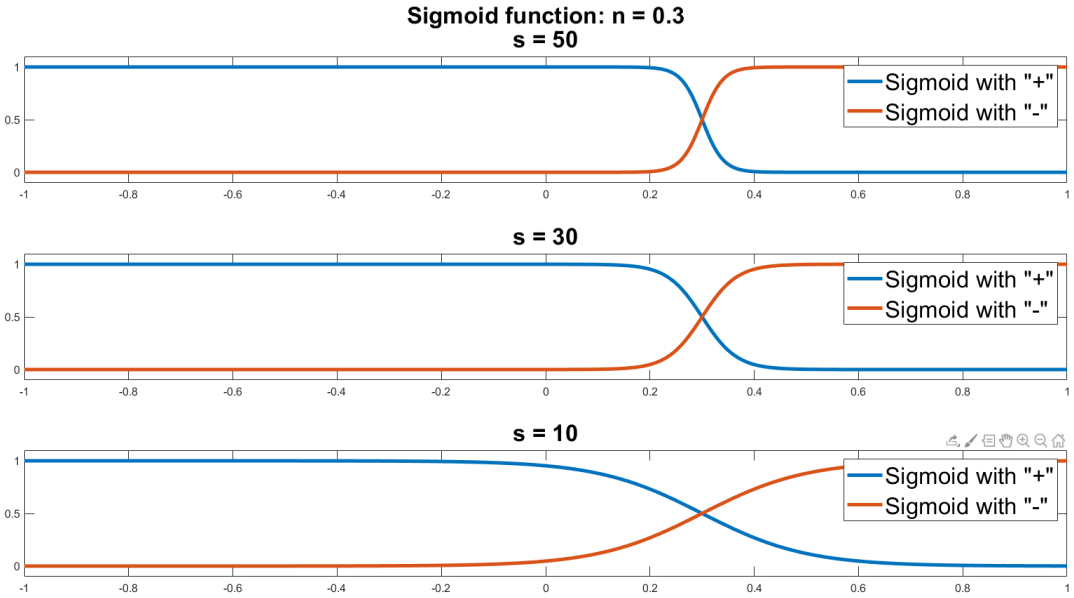


Figure 6.24: Sigmoid Function

Then, as in Section 6.4, we can define the constraints on the state, on the inputs and the ones derived by the system's dynamics. Their values are the one listed in

Table 6.1.

$$-\infty < X < \infty \tag{6.54}$$

$$-speed\_constraints < U < speed\_constraints \tag{6.55}$$

$$-h \cdot acc\_constraints < U_0 - U_{prev} < h \cdot acc\_constraints \tag{6.56}$$

$$-h \cdot acc\_constraints < U_{i+1} - U_i < h \cdot acc\_constraints \tag{6.57}$$

With $i \in [0, N-2]$.

$$-h^2 \cdot jerk\_constraints < U_1 - 2 \cdot U_0 + U_{prev} < h^2 \cdot jerk\_constraints \tag{6.58}$$

$$-h^2 \cdot jerk\_constraints < U_{i+1} - 2 \cdot U_i + u_{k-1} < h^2 \cdot jerk\_constraints \tag{6.59}$$

With $i \in [1, N-2]$.

$$X_0 = \xi_0 \tag{6.60}$$

$$X_{i+1} = f_{discrete}(X_i, U_i) \tag{6.61}$$

With $i \in [0, N-1]$.

6.5.1 Algorithm tests

Here we present some tests useful to verify the applicability of the non-linear Model Predictive Control with Velocity control and the modulated DS.

Test 1

In order to test the performance of the non-linear MPC with velocity control, a first simulation is performed in a static environment, like the simulations done in the Section 6.4. The robot starts from $\xi_i = [0.0, -0.5, 0.0]^T$ and the desired final position is $[x_f, y_f]^T = [10.0, 0.0]^T$.

Three ellipsoid shaped obstacles are present in the environment, in position C and with axes:

1. $C = [3.5, 0.1]^T$, $r_x = 0.3$, $r_y = 0.4$;
2. $C = [7, -1]^T$, $r_x = 0.3$, $r_y = 0.7$;
3. $C = [2, 0]^T$, $r_x = 0.7$, $r_y = 0.3$;

Also this non-linear MPC gives us the possibility to impose constraints on the maximum speeds, on the accelerations and on the jerking of the robot. We will consider all the constraints listed in Table 6.1.

The gain matrices used for this simulation are:

$$G_\theta = q_\theta^2 \qquad G_{Vx} = q_{Vx}^2 \qquad G_{Vy} = q_{Vy}^2 \qquad G_u = R \cdot R \tag{6.62}$$

$$q_\theta = 2 \qquad q_{Vx} = 10 \qquad q_{Vy} = 10 \qquad R = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \tag{6.63}$$

The simulation time is $T_{sim} = 25s$. Figure 6.25 illustrates the path of the robot and the computation time. All the constraints are taken into account but, differently from the tests done in 6.4, the robot is able to accomplish the path in the given simulation time. The path that the robot really covers is slightly different from the ideal one, but still, it is an acceptable behaviour that allows the robot to avoid the obstacles. Also the computation time constraints are respected: it is always smaller than 0.05 seconds, that is the threshold due to the control frequency.

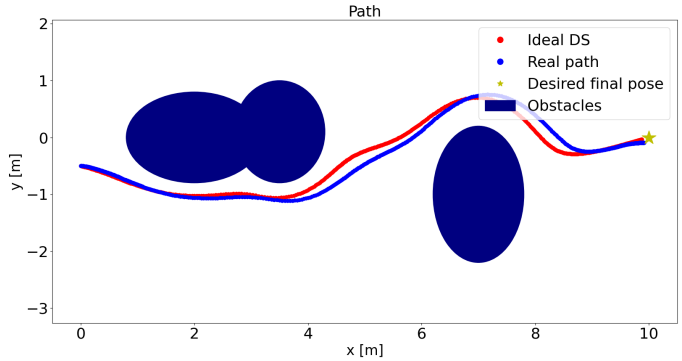
As in the simulations done in the previous section, a random noise has been added both to the control inputs and to the robot state, to simulate the error caused by the motors of the robot, by the sensors, by the SLAM algorithm and by the hardware delay. On the control inputs, a random noise is added, with maximum amplitude:

- 0.05*1.5 = 0.075 m/s for the linear speed;

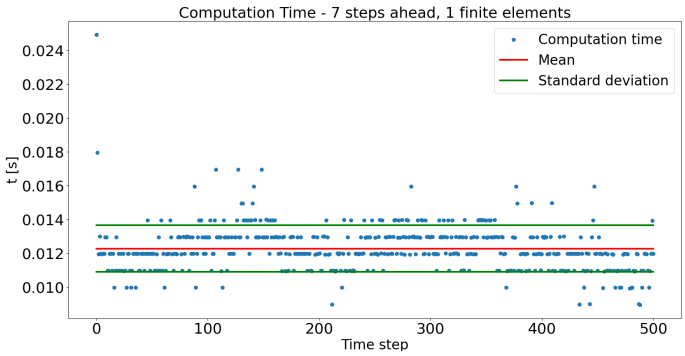
- 0.05*3.0 = 0.15 rad/s for the angular speed.

The noise on the robot state is computed as:

- $x = numpy.random.normal(x,0.03)$;
- $y = numpy.random.normal(y,0.03)$;
- $\theta = numpy.random.normal(\theta,0.04)$;



(a) Path

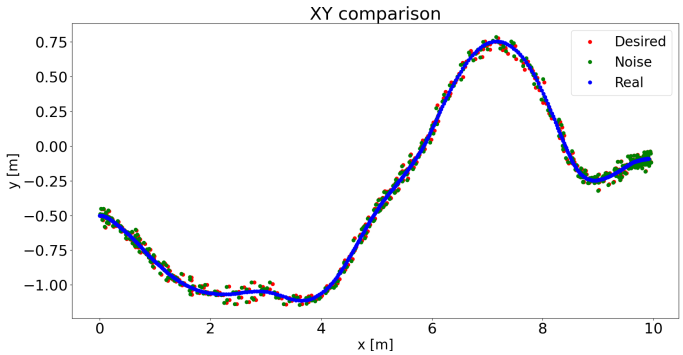


(b) Computation Time

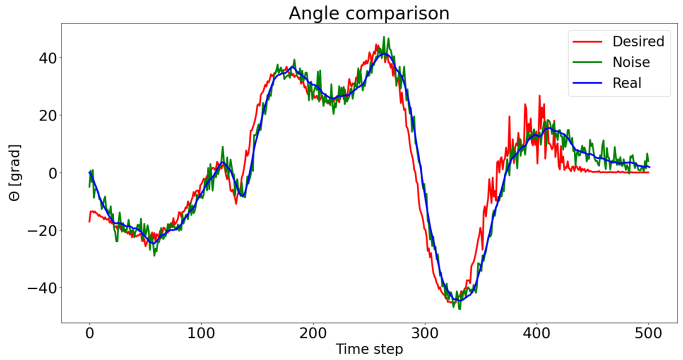
Figure 6.25: nMPC - Velocity control - Test 1

Figure 6.26 displays the real path of the robot, the perceived one (i.e. the noisy positions) and the desired next position starting from the perceived position. Also in this case, it is possible to notice the robustness against the noise obtained by the combination of the DS modulation with the non-linear MPC. The first one produces a velocity vector that is barely affected by the noise on the position; while the nMPC produces control inputs very optimized that can be slightly changed (adding the noise for instance) without compromising the behaviour of the robot. The most important thing that can be noticed concerns the control inputs them

self, that are illustrated in Figure 6.27. Particular relevance has the control input ω, because in the previous tests with other controller was the one that caused the main issues. In this case it is much smoother with respect to the previous tests. It still presents some abrupt changes, but they respect the limits imposed by the accelerations constraints and the jerking constraints.

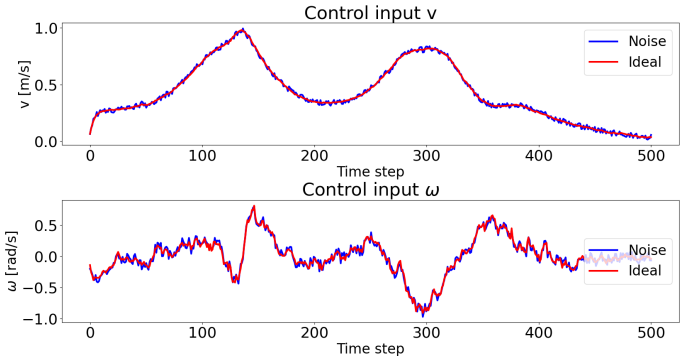


(a) Path with noise

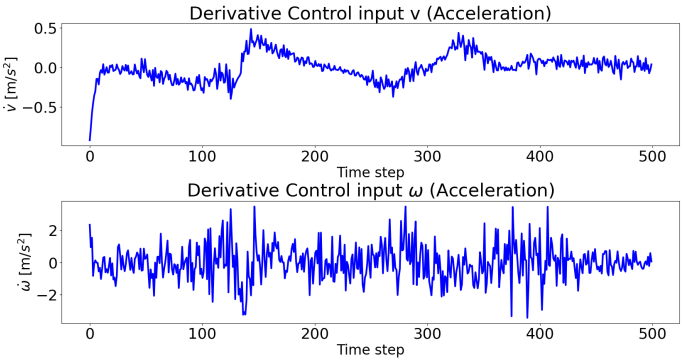


(b) Angle with noise

Figure 6.26: nMPC - Velocity control - Test 1 - Noise analysis



(a) Control inputs



(b) Control inputs derivative

Figure 6.27: nMPC - Velocity control - Test 1 - Control Inputs

Test 2

The previous test confirmed that the combination of DS modulation and the non-linear Model Predictive Controller with Velocity control is feasible in a fixed environment. This simulation tests its behaviour in an environment with fixed obstacles, but moving target.

As first thing, it is needed to remember that during the target following, the attractor is not set inside the area of the target, because this last one is also considered as an obstacle, in order to avoid collisions. The attractor in called "*moving attractor*" as explained in section 5.10, and the procedure illustrated in Algorithm 5 is used to move the attractor in a coherent way, depending on the obstacles present in the environment. In this case the value used for the constants in Equation 5.33, are set as:

- $k_{attr} = 3$
- $k_{rep} = 5$

The target moves with a constant velocity $\dot{\xi}_{tar} = [0.3, 0.0]^T m/s^2$. The prediction horizon is $N = 7$, the simulation time is set as $T_{sim} = 35s$ and the gain matrices are:

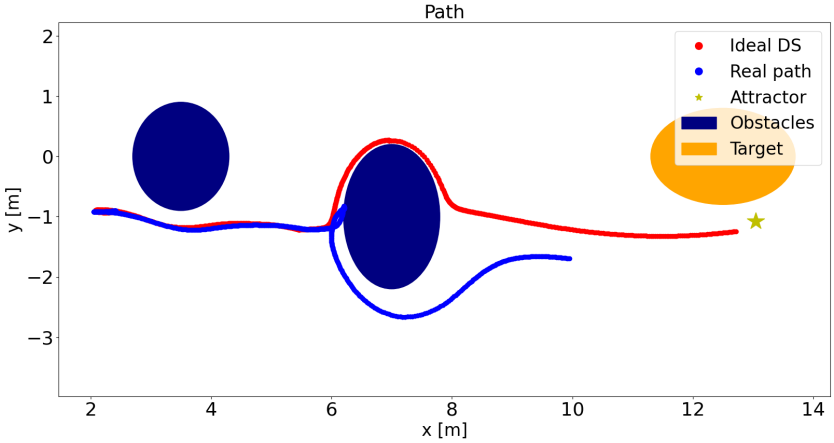
$$G_\theta = q_\theta^2 \qquad G_{Vx} = q_{Vx}^2 \qquad G_{Vy} = q_{Vy}^2 \qquad G_u = R \cdot R \tag{6.64}$$

$$q_\theta = 1 \qquad q_{Vx} = 10 \qquad q_{Vy} = 10 \qquad R = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \tag{6.65}$$

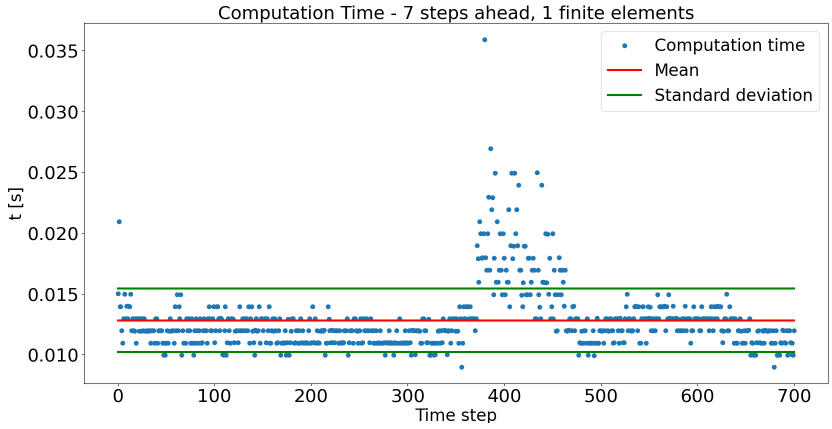
The desired heading angle can be chosen using between different methods. In this case it is given by Equation 6.50, therefore we exploit only the information on the heading angle of the target to generate the desired heading angle of the robot, while the modulated DS is used to generate the desired velocity vector.

Figure 6.28 shows the results of this simulation.

The robot is not able to follow the target in an ideal way: it goes very close to the obstacle in the center of the figure and it also collide with it. This is a situation that does not create problem if the obstacle is enlarged and a safety margin is considered. At this point the robot gets stuck because of the discontinuity of the DS on the border of the obstacle and because of the desired heading angle. Due to these two factors, the robot tends to go inside the obstacle, then the DS wants to push it away, but the heading angle does not allow the robot to steer in a proper manner. The robot moves forward and backward until the steering angle allows it to pass the obstacle from the opposite side.



(a) Path

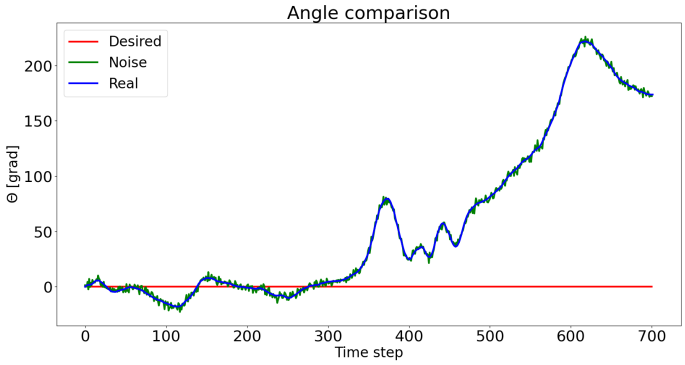


(b) Computation Time

Figure 6.28: nMPC - Velocity control - Test 2

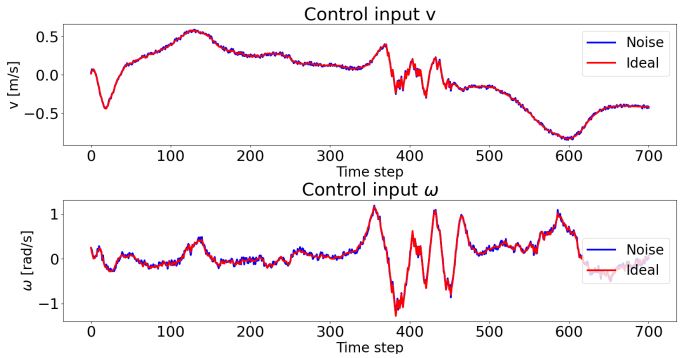
This kind of behaviour is also caused by the gain G_{theta} related to the heading angle, because bigger is this value, higher will be the effort of the nMPC to follow the desired heading angle, that in this case is always zero, as illustrated in Figure 6.29.

It is also possible to notice that, when the robot gets stuck, the computation time of the non-linear MPC tends to increase. This is probably cause by the fact that the desired velocity varies a lot (it points forward for some time steps and then backward for some others) and the optimization process of the nMPC requires more time to output the correct control inputs. These latter can be observed in Figure 6.30: with the exception of the time steps during which the robot is stuck, the control inputs show a nice behaviour, but further tests need to be done in order to overcome the issues related to the heading angle.

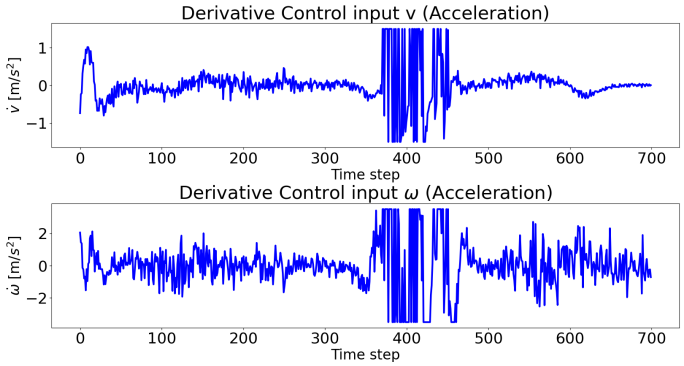


(a) Angle with noise

Figure 6.29: nMPC - Velocity control - Test 2 - Noise analysis



(a) Control inputs



(b) Control inputs derivative

Figure 6.30: nMPC - Velocity control - Test 2 - Control Inputs

Test 3

In order to verify the correct behaviour of the controller, a test with the same settings as Test 2 is done, i.e. $T_{sim} = 35s$, $N = 7$ and gain matrices:

$$G_\theta = q_\theta^2 \qquad G_{Vx} = q_{Vx}^2 \qquad G_{Vy} = q_{Vy}^2 \qquad G_u = R \cdot R \tag{6.66}$$

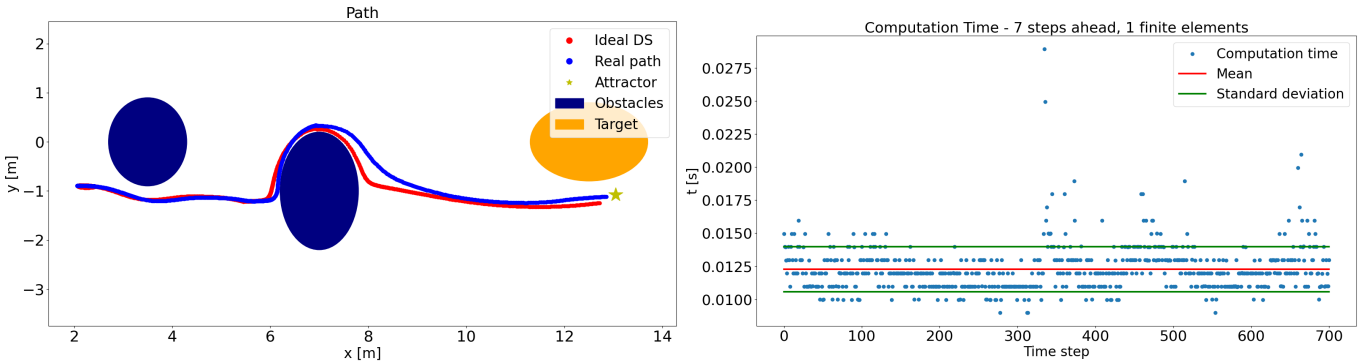
$$q_\theta = 1 \qquad q_{Vx} = 10 \qquad q_{Vy} = 10 \qquad R = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \tag{6.67}$$

In this case the heading angle is computed using Equation 8.19.

The heading angle is therefore computed using both the information on the heading angle of the target and on the desired velocity. The parameters used in the Sigmoid function are:

- s = 20;
- n = 0.5;

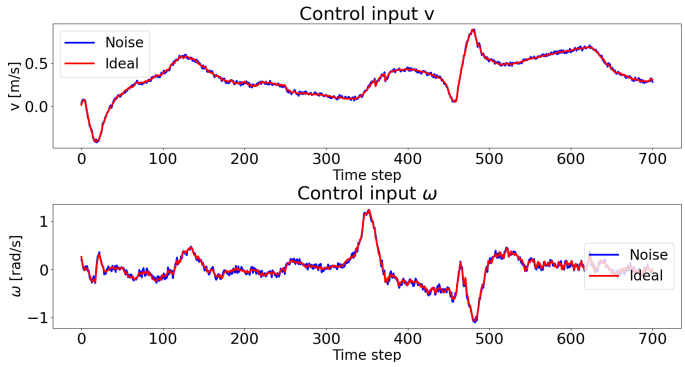
Figure 6.31 illustrates the results of this test. In this case the robot is able to follow the target while avoiding the obstacles. The computation time respects all the constraints imposed by the hardware.



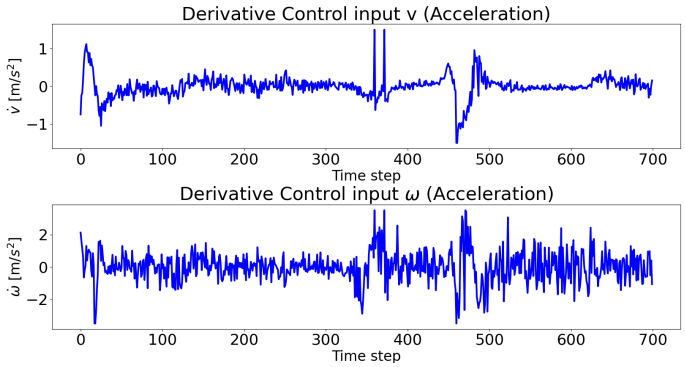
(a) Path **(b)** Computation Time

Figure 6.31: nMPC - Velocity control - Test 3

As for the control inputs, all the constraints on the maximum value allowed, the maximum speed and the maximum jerking are observed. In Figure 6.32 it is possible to observe a quite smooth behaviour that can be used on the real implementation.



(a) Control inputs



(b) Control inputs derivative

Figure 6.32: nMPC - Velocity control - Test 3 - Control Inputs

Test 4

In the previous tests it has emerged that using $N = 7$ is a good trade off because produces good control inputs and respect the time constraints. From a theoretical point of view, using a larger horizon time should produce better results because the controller optimizes the inputs having more information about the future of the robot. In order to test this assumption, a simulation with the same settings as Test 3 is done, i.e. $T_{sim} = 35s$ and gain matrices:

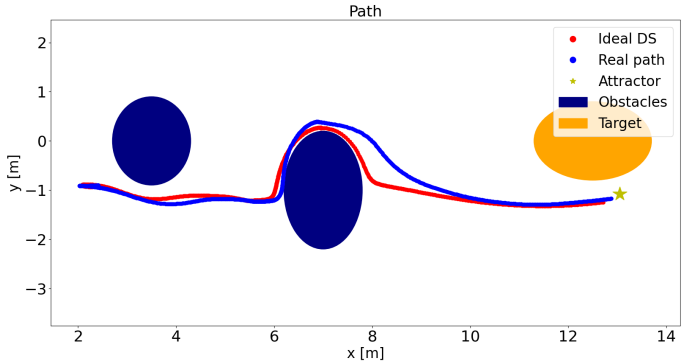
$$G_\theta = q_\theta^2 \qquad G_{Vx} = q_{Vx}^2 \qquad G_{Vy} = q_{Vy}^2 \qquad G_u = R \cdot R \tag{6.68}$$

$$q_\theta = 1 \qquad q_{Vx} = 10 \qquad q_{Vy} = 10 \qquad R = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \tag{6.69}$$

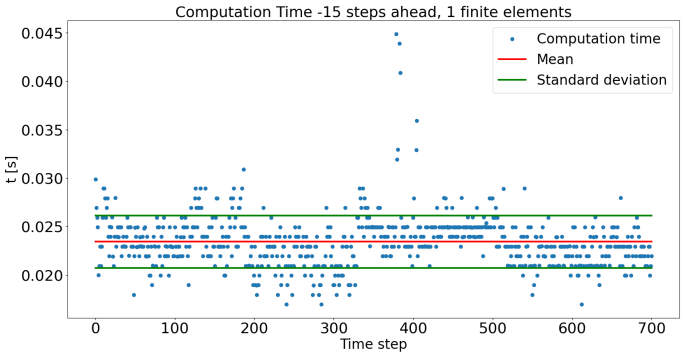
This time, though, the horizon time is doubled: $N = 15$ The Sigmoid function is used again to obtain the heading angle, with the parameters:

- s = 20;
- n = 0.5;

Figure 6.33 illustrates the results of this test. They are very similar to 6.31; the main difference, as expected, is the computation timing. The time needed for the optimization of the control inputs is excessive, therefore this horizon time can not be used on the real robot.



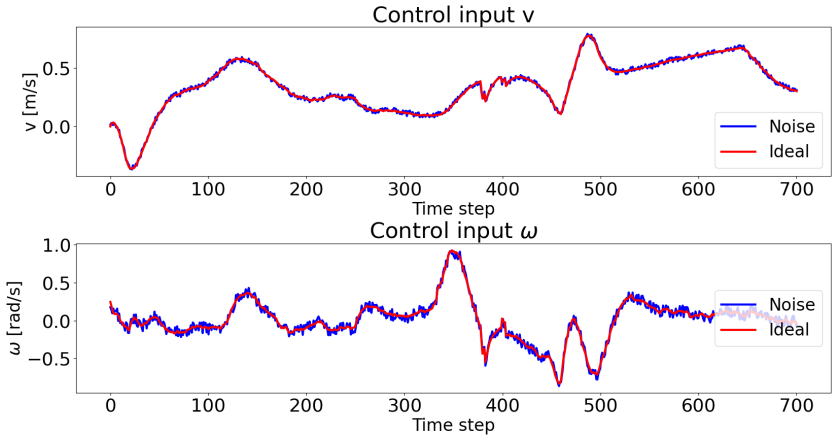
(a) Path



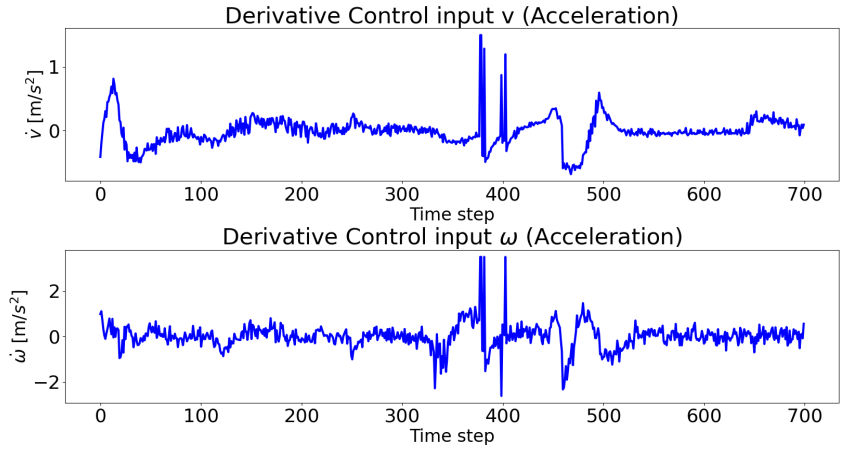
(b) Computation Time

Figure 6.33: nMPC - Velocity control - Test 4

As for the control inputs, all the constraints are observed. In Figure 6.34 it is possible to observe a better behaviour with respect to the control inputs obtained in the previous test and displayed in 6.32. This is a consequence of the larger horizon time: the nMPC is able to better optimize the control inputs and, consequently, make the inputs smoother, more robust against the noise and more suitable for the user experience.



(a) Control inputs



(b) Control inputs derivative

Figure 6.34: nMPC - Velocity control - Test 4 - Control Inputs

Considerations

The tests in the static and dynamic conditions validate the low level controller described in this section. It appears that it can be used in both the scenarios with optimal outcomes. The control inputs do not have strange behaviours and they are able to drive the robot nicely, respecting the imposed constraints. Therefore, abrupt movements should be avoided, guaranteeing a nice experience for the user. The considerations on the computation time are the same done also in the previous sections and chapters: from a theoretical point of view, this controller is fast enough and can be used on the real robot, but it entirely depends on the actual hardware. Moreover, as expected, the computation timing depends on the prediction horizon: the larger is the prediction horizon, the more time is required to compute the

optimal inputs.

It is worthy to notice that this controller has a fine response to the noise: the simulated noise on the sensors (i.e. the pose of the robot) is compensated by the ability of the controller to produce control inputs as continuous and smooth as possible.

Until now, this controller is the most promising one and can be easily used in combination with the modulating algorithm.

6.6 MPC - Inputs Bounding

Differently from the methods used in Sections 6.4 and 6.4, the Model Predictive Controller used in this section does not use the kinematics of the robot to produce the optimal inputs for the control.

This method exploits the feedback linearization of the unicycle [36]: first of all, a point P is defined that is placed at distance ϵ from the robot centre ($[x, y]^T$) in the direction of the longitudinal velocity:

$$\begin{aligned} x_P &= x + \epsilon * cos(\theta) \\ y_P &= y + \epsilon * sin(\theta) \end{aligned} \tag{6.70}$$

Differentiating and considering the unicycle dynamics, we have:

$$\begin{aligned} \dot{x_P} &= V_{Px} \\ \dot{y_P} &= V_{Py} \end{aligned} \tag{6.71}$$

We can obtain the values for V_P using the modulating DS, applied at the point P.
Using the formula in Equation 6.72, we have directly the control inputs for the system.

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} cos(\theta) & sin(\theta) \\ -\frac{1}{\epsilon}sin(\theta) & \frac{1}{\epsilon}cos(\theta) \end{bmatrix} \begin{bmatrix} V_{Px} \\ V_{Py} \end{bmatrix} \tag{6.72}$$

The inconvenient of this method is that we can not take into account the system's constraints such as maximum velocity and acceleration. A possible solution is to use the control inputs computed above, as reference inputs that we want to reach at each step. In this way we can define a simple MPC model that has state $X = [v, \omega]^T$ and input $U = [lin\_acc, ang\_acc]^T$.
The dynamic that we are considering now is:

$$\begin{bmatrix} \dot{v} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} lin\_acc \\ ang\_acc \end{bmatrix} \tag{6.73}$$

Discretizing it, we obtain:

$$X_{k+1} = A_d X_k + B_d U_k \tag{6.74}$$

$$\begin{bmatrix} v_{k+1} \\ \omega_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_k \\ \omega_k \end{bmatrix} + \begin{bmatrix} h & 0 \\ 0 & h \end{bmatrix} \begin{bmatrix} lin\_acc_k \\ ang\_acc_k \end{bmatrix} \tag{6.75}$$

With h = sampling time.

Thanks to the MPC, it is possible to constraint the state and the inputs in order to respect the systems limits and avoid abrupt movements:

$$|X_k| < state\_constraints \tag{6.76}$$

$$|U_k| < input\_constraints \tag{6.77}$$

We have to give to the MPC the initial state:

$$X_0 = X_{prev} \tag{6.78}$$

The dynamic used by the MPC is describe in Equation 6.74.

Finally, we have to define a cost function that has to be minimized:

$$\sum_{i=0}^{N-1} (X_i - X_{ref})^T Q (X_i - X_{ref}) + (X_N - X_{ref})^T Q_N (X_N - X_{ref}) + \sum_{i=0}^{N-1} U_i^T R U_i$$

Q, Q_N and R are diagonal matrices, that contain the gains to be tuned.

Remembering that in this case $X = [v, \omega]^T$ and input $U = [lin\_acc, ang\_acc]^T$; after the optimization we have the optimal accelerations and the optimal velocities. Applying to the robot the optimal velocities, it is guaranteed that the system will respect the constraints.

6.6.1 Algorithm tests

Here we present some tests useful to verify the applicability of the Model Predictive Control and the modulated DS.

Test 1

The first test to verify the characteristics of the controller is done in a static environment. The robot starts from $\xi_i = [0.0, -0.5, 0.0]^T$ and has to reach the desired final position in $[x_f, y_f]^T = [10.0, 0.0]^T$.

Three ellipsoid shaped obstacles are present in the environment, in position C and with axes:

1. $C = [3.5, 0.1]^T$, $r_x = 0.3$, $r_y = 0.4$;
2. $C = [7, -1]^T$, $r_x = 0.3$, $r_y = 0.7$;
3. $C = [2, 0]^T$, $r_x = 0.7$, $r_y = 0.3$;

The gain matrices used for this simulation are:

$$Q = Q_N = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad R = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.1 \end{bmatrix} \tag{6.79}$$

The simulation time is $T_{sim} = 25s$. This controller does not allow us to constraints the jerking of the robot, therefore, the control inputs might not be totally suitable for the real implementation, even if they respect the constraints on the speed and the acceleration. Since the computation effort is drastically reduce with respect the non-linear controller described in the previous sections, we can use a prediction horizon of $N = 20$. The distance ϵ is set as $\epsilon = \epsilon_{safe} = 0.2m$

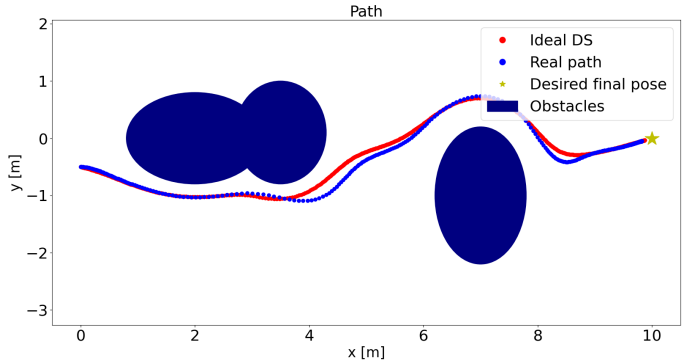
The results of the simulation are illustrated in Figure 6.35. The path covered by the robot is very similar to the ideal one and the computation time respects the limits imposed by the hardware of the robot.

Following the procedure illustrated in the previous section, a random noise has been added both to the control inputs and to the robot state, to simulate the error caused by the motors of the robot, by the sensors, by the SLAM algorithm and by the hardware delay. On the control inputs, a random noise is added, with maximum amplitude:

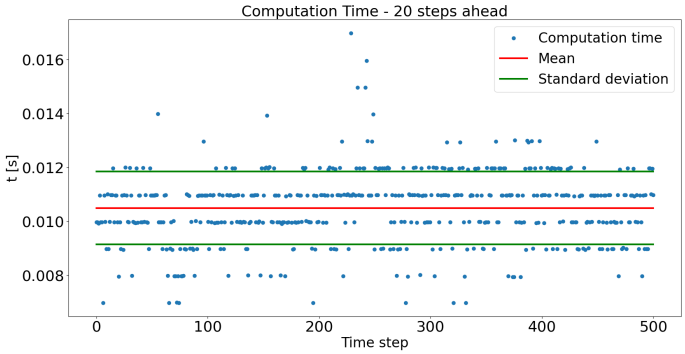
- 0.05*1.5 = 0.075 m/s for the linear speed;
- 0.05*3.0 = 0.15 rad/s for the angular speed.

The noise on the robot state is computed as:

- x = numpy.random.normal(x,0.03);



(a) Path

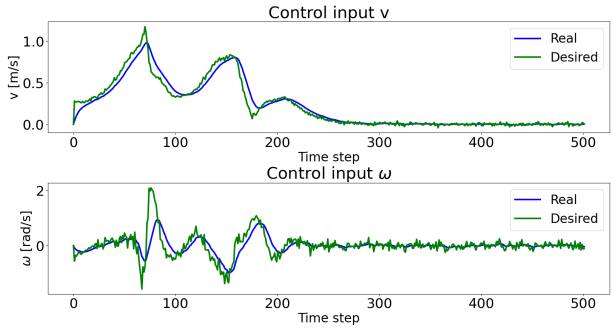


(b) Computation Time

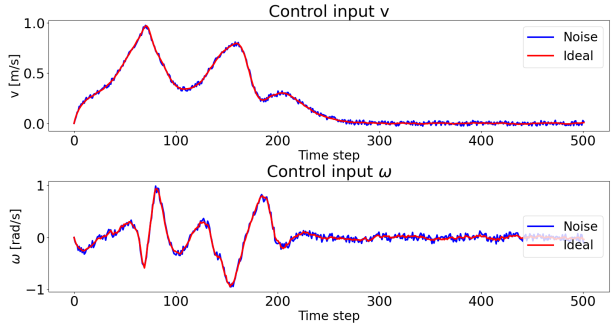
Figure 6.35: MPC - Test 1

- y = numpy.random.normal(y,0.03);
- θ = numpy.random.normal(θ,0.04).

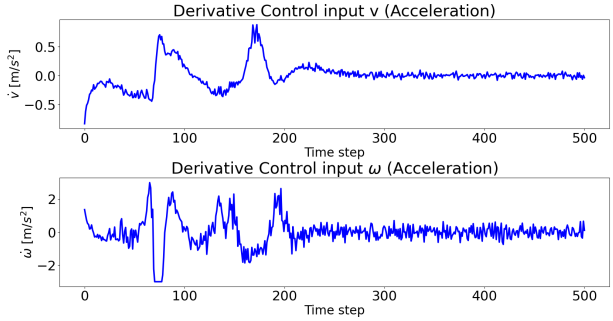
The control inputs obtained with this procedure respect all the constraints imposed, but we can not say anything about the jerking constraints since they are not included in the MPC controller. Nevertheless, they seems very smooth and suitable for the real robot. Figure 6.27 displays the control inputs of the system: Figure 6.27(a), in particular, shows the control inputs obtained using the feedback linearization that act as reference for the MPC, and compare them with the output of the MPC. Figure 6.27(b) compare the inputs obtained from the MPC and the inputs with the added noise, that are used on the system. The control inputs produced by the Model Predictive Controller are way smoother than the reference controls computed with the linearization.



(a) Control inputs comparison



(b) Control inputs



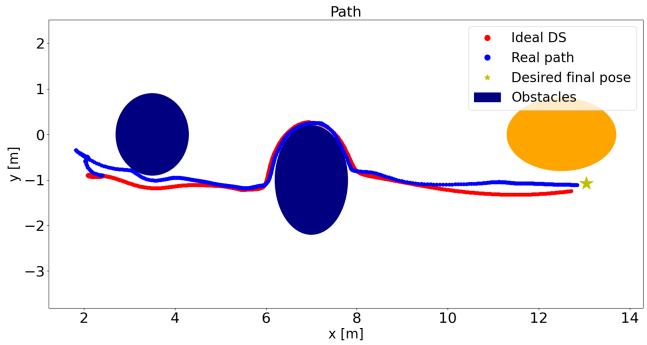
(c) Control inputs derivative

Figure 6.36: MPC - Test 1 - Control Inputs

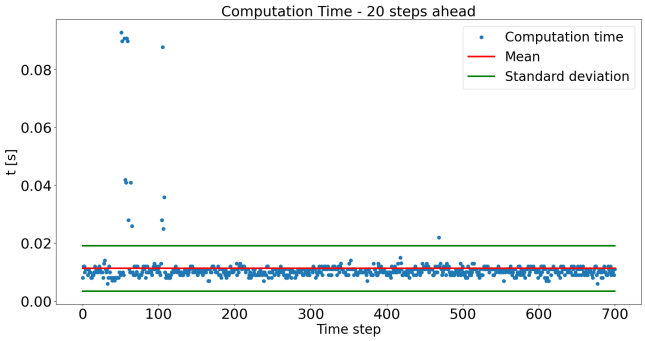
Test 2

With this test we want to verify the behaviour of the controller in a dynamical environment: as in the previous sections, the obstacles are fixed, while the target is moving. The parameters of the MPC are the same as in the previous test, $N = 20$ and $\epsilon = \epsilon_{safe}$, while the simulation time is $T_{sim} = 35s$.

Figure 6.37 illustrates the behaviour of the robot in a dynamic environment and the results are not promising as the ones in test 1. First of all, the computation



(a) Path



(b) Computation Time

Figure 6.37: MPC - Test 2

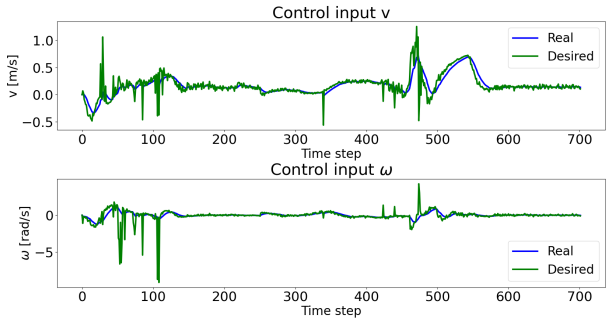
time presents some peaks that exceed the limit imposed of $0.05s$. This means that on the robot the MPC would not be able to compute in time the control inputs, forcing the system to use either the previous control inputs or control inputs equal to zero. Generally, this could be permitted if it is just a spurious event; but in this case consequent time steps present the same issue. Having the robot running in open loop could cause enormous issues to the safety of the users and of the people around them.

Moreover, at the beginning of the simulation, the robot almost hits the target because the initial position of the robot is in front of the attractor, which is moved backward due to the presence of the obstacles. This cause the robot to move backwards while steering; at this point, though, there is the main issue: with this algorithm we are not controlling the center of the robot, but a point that is positioned at ϵ meters ahead, therefore the presence of the obstacle is perceived from the point of view of this point.

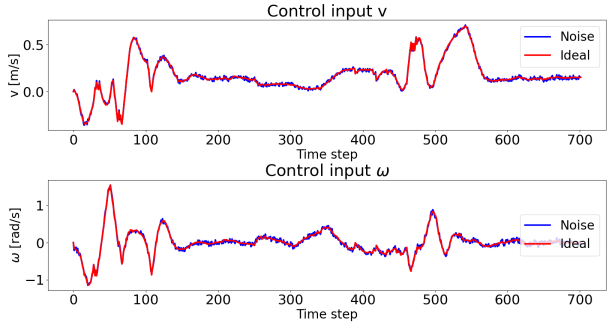
Since we are enlarging the obstacles of $\epsilon_{safe} = \epsilon$, the repulsive effect of the obstacle might be perceived when it is too late.

The control inputs shown in Figure 6.30 respect the constraints, but they present

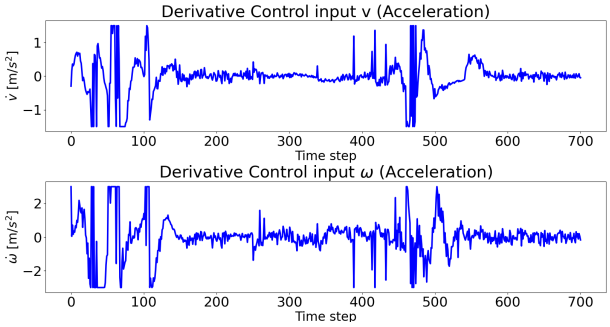
some strange behaviour at the beginning, for the reasons explained above, and between time steps 400 and 500, when the robot face the other obstacle. Also in this case, the control inputs produced by the MPC starting from the reference controls obtained by the feedback linearization, are smoother. Moreover, it is worthy to mention that the reference inputs exceed the limits imposed by the system by quite a lot.



(a) Control inputs comparison



(b) Control inputs

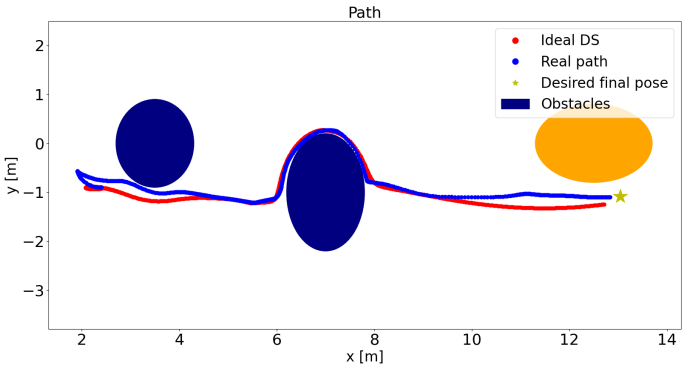


(c) Control inputs derivative

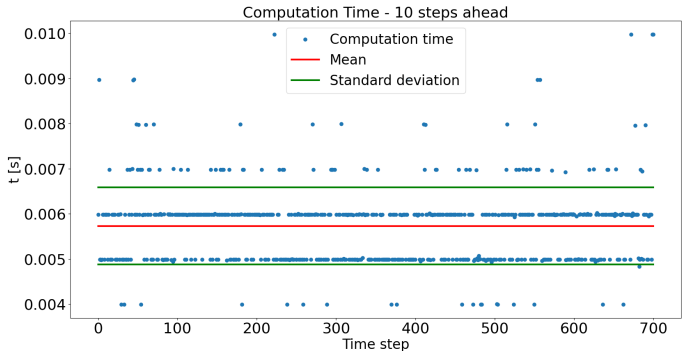
Figure 6.38: MPC - Test 2 - Control Inputs

Test 3

The same simulation as in test 2 is repeated, but this time with prediction horizon $N = 10$. The other parameters of the MPC are are $\epsilon = \epsilon_{safe}$, and the simulation time is $T_{sim} = 35s$.



(a) Path



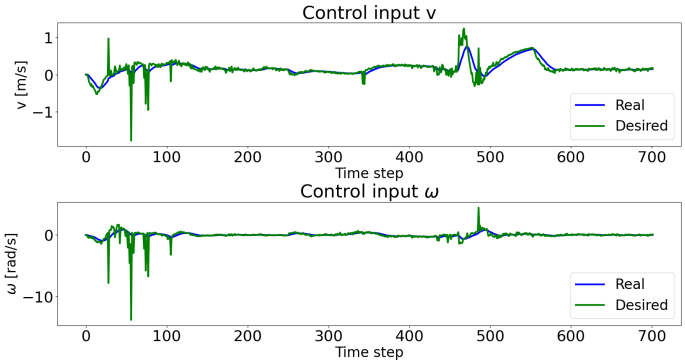
(b) Computation Time

Figure 6.39: MPC - Test 3

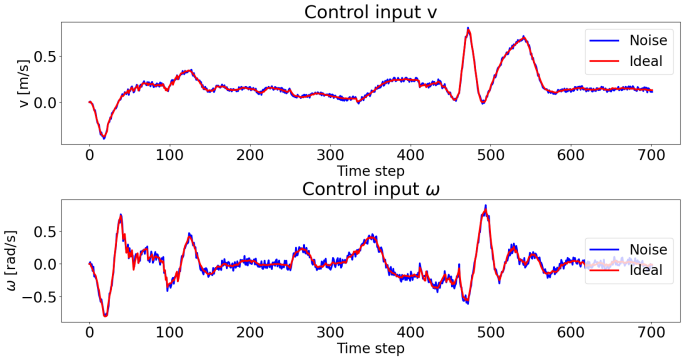
Figure 6.39 displays the outputs of the simulation. In this case the problems of the previous test are avoided, but it is not because of the reduction of the prediction horizon. Simply, the control inputs produced are not able to move the robot fast enough to go inside the area of the target and therefore the computation time observes the limits. It may still happen an analogous situation and the computation time could be higher than the limits. Nevertheless, it is true that reducing the prediction horizon lower also the computation time: a good trade off has to be found.

In this case, the control inputs, in addition to the compliance to the constraints, have a smooth trend and react well to the noise, as illustrated in Figure 6.32. As in the previous tests, the MPC is able to produce smoother control inputs with

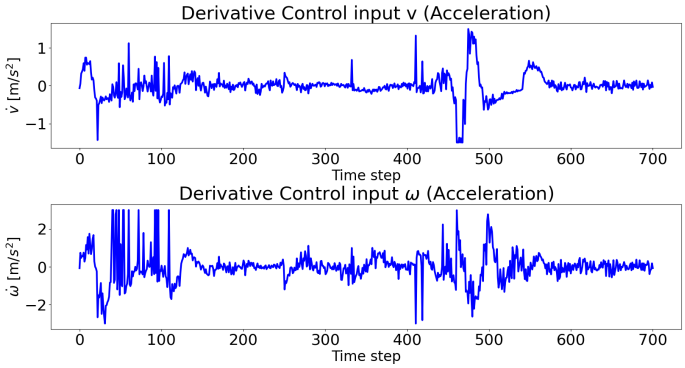
respect the reference ones, which are also not bounded and would produce an undesired behaviour if used directly.



(a) Control inputs comparison



(b) Control inputs



(c) Control inputs derivative

Figure 6.40: MPC - Test 3 - Control Inputs

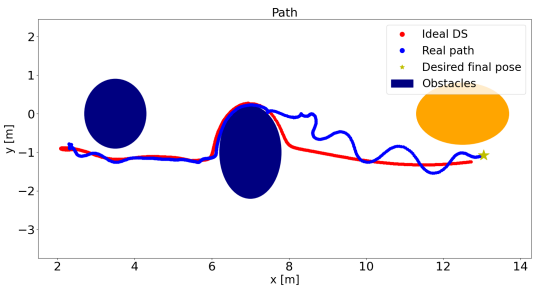
Test 4

The previous tests have explained that one of the main issues of this approach is the use of ϵ that nullify the action of ϵ_{safe} if these two values coincide. Here, two simulations will be presented, both of them with $\epsilon < \epsilon_{safe}$, $N = 10$ and $T_{sim} = 35s$.

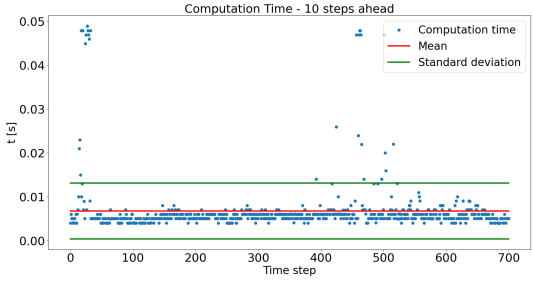
1. $\epsilon = 0.05m$
2. $\epsilon = 0.1m$

Both the simulations show very bad performances. The timing constraints are totally unobserved and the robot behaves in a very poor manner: in the first test, with $\epsilon = 0.05m$, the robot slaloms in order to reach the attractor. While, in the second case, the robot hits the target and one of the fixed obstacles.

These two tests demonstrate that changing the vale of ϵ do not change the outcome of the simulations in better, but in certain cases it worsen the behaviour obtained.

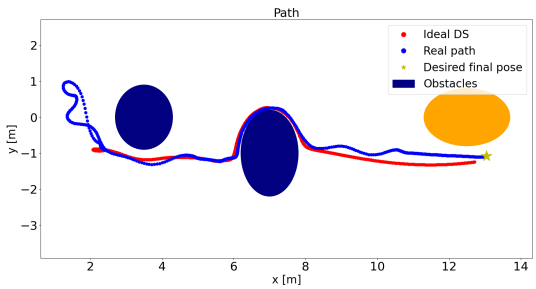


(a) Path

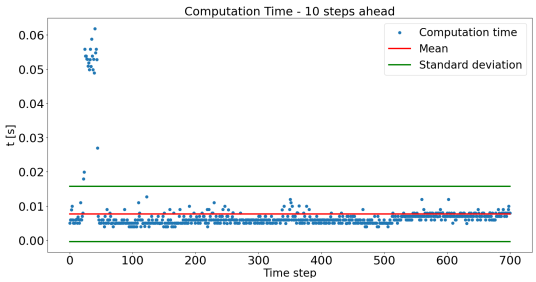


(b) Computation Time

Figure 6.41: MPC - Test 4
$\epsilon = 0.05m$

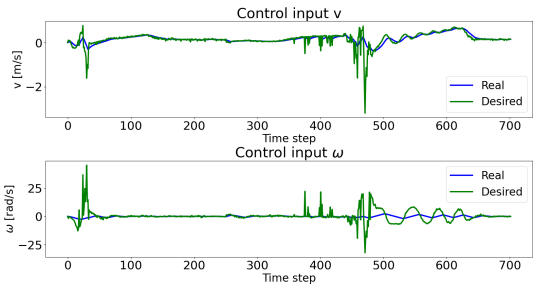


(a) Path

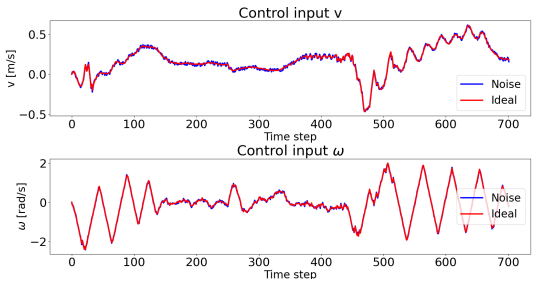


(b) Computation Time

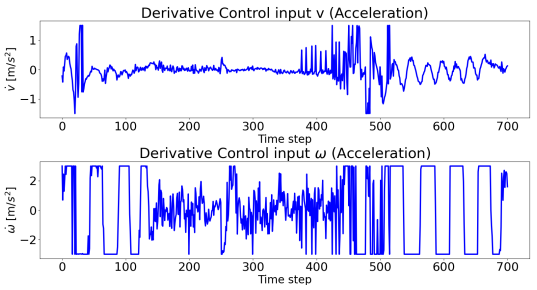
Figure 6.42: MPC - Test 4
$\epsilon = 0.1m$



(a) Control inputs comparison

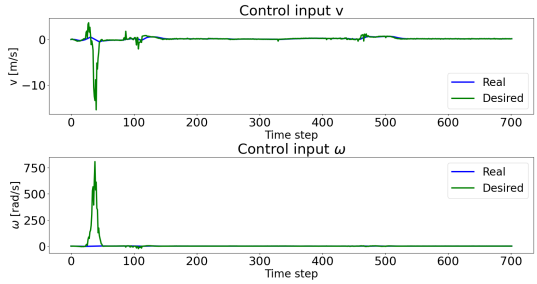


(b) Control inputs

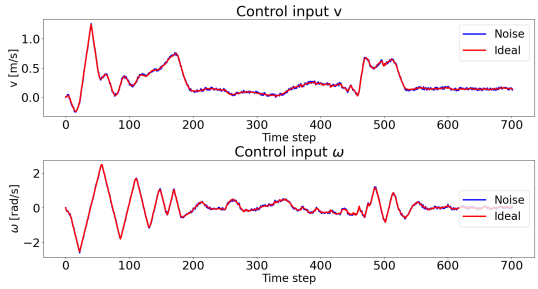


(c) Control inputs derivative

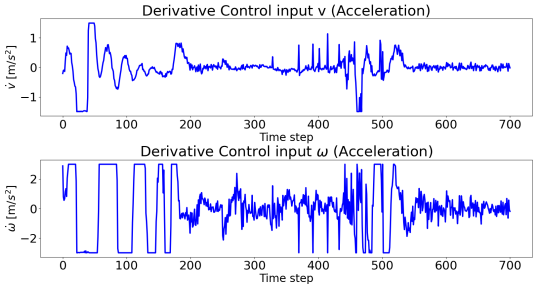
Figure 6.43: MPC - Test 4
$\epsilon = 0.05m$ - Control Inputs



(a) Control inputs comparison



(b) Control inputs



(c) Control inputs derivative

Figure 6.44: MPC - Test 4
$\epsilon = 0.1m$ - Control Inputs

Considerations

This controller presents very interesting features:

- reacts nicely to the noise;
- simple implementation;
- allows a long prediction horizon;
- smooth control inputs.

Nevertheless, the tests done in the dynamic environment prove that it obtains optimal performances only in very specific cases, while in others, if a small issue arises, the controller is not able to compensate in a smart way and tends to take a lot of time to compute the correct control inputs.

The other main issue concerns the position of the point where the modulation is computed. The enlarged obstacles take into account the robot dimension and a safety margin, considering the center of the robot the point where the modulation is applied. A possible solution to overcome this problem could have been to use $\epsilon < \epsilon_{safe}$; but the tests done in Test 4 prove that using a smaller ϵ do not improve the performance.

Chapter 7

Obstacles and Target Detection Algorithm

In the previous chapters, it was assumed a complete knowledge of the obstacles: at every time-step the obstacle position was known, as well as its linear velocity, rotational velocity and its shape. In a real world scenario, we must extract these information starting from the data obtained from the robot's LIDARs and the 3D-camera.

An existing algorithm exploits the RGB images to detect the presence of people and find out their position, orientation and velocity. Unfortunately, it often produces false positives, which means that a person is detected in a certain position, even if nothing is present in that precise spot. Moreover, the speed of the algorithm constraints deeply its utility because it can run at 10 Hz, causing the analysis of the environment to be too slow to be used in real-time.

In this chapter, some Machine Learning techniques for the environment analysis will be described. They are used mainly for the obstacle detection and modelling. The main features that we aim to obtain from these methods are:

1. Speed - the algorithm has to be fast enough to run in real-time on the robot's hardware.

2. Accuracy - the modelling of the obstacles has to be as accurate as possible, in order to identify the centroids of the obstacles, i.e. the presumed center of the obstacle, that is used for the computation of the velocity.

3. Precision - the identified shape of the obstacle has to be as similar as possible to the real one, in order to avoid problematic scenarios in which, either the modelled obstacle covers an area way bigger or smaller than the real obstacle.

Moreover, since the high level controller works with ellipsoid-shaped obstacles, also the algorithm should produce this typology of shape, in order to facilitate the algorithm integration.

The Machine Learning techniques that were analyzed for this purpose are the **Gaussian Mixture Model** and the **K-means**. In the next sections these algorithms will be explained in detail and their results, benefits and drawbacks will be analyzed.

7.1 Machine Learning - Overview [37] [38]

"Machine Learning" is a concept that has gained an incredible relevance in the last decades and it is often used to describe topics, ideas and methods very different one from the other. From a technical point of view, the Machine Learning is a branch of the **Artificial Intelligence** science and its focus are computer algorithms that exploit the experience to get better in an autonomous way.

Machine learning algorithms differ from the usual algorithm structure because they are used to solve problems without an organized approach (a programmed approach). Instead, they make use of data sets and algorithms that exploit both probabilistic and deterministic methods, to model a certain behaviour and take actions accordingly.

The fields of application are many:

- Computational statistics - a lot of machine learning algorithms use a probabilistic approach to define a model; they can be useful to predict a behaviour or an outcome.
- Data mining - analyze enormous data sets is unfeasible for a human being, machine learning algorithm are used to find features or to categorize.
- Mathematical optimization - machine learning algorithms often require mathematical tools for a fast optimization, and, can also be used for the development of new mathematical methods.
- Data-driven control - the possibility of building a behaviour, starting from empirical data allows new control methods that performs surprisingly well, but often only in very specific conditions.

Generally, the machine learning algorithms are divided into three categories:

- Supervised learning - the machine has to produce a system law starting from a set of inputs, linked with their outputs. Basically, the computer knows

that a certain output is caused by a certain input and generate the model accordingly.

- Unsupervised learning - in this case no previous knowledge is used and nor the input or the output are labeled. The machine has to find features, relationships or behaviours by its own. Interesting results could be obtained using this methods, especially because they can be unforeseen.
- Reinforcement learning - the machine learns by its interaction with a dynamical environment, using rewards and penalizations. Its goal is to maximise a cost function and find the optimal behaviour.

7.2 Gaussian Mixture Model [39]

The Gaussian Mixture Model, or GMM, is a clustering algorithm and, therefore, an unsupervised machine learning technique. A clustering algorithm is a method used to find clusters with common characteristics in a data set. In particular, we can distinguish between hard and soft clustering techniques. The first one define a clear border between the clusters, so a point of the data set can belong either to a cluster or to another one. The soft clustering, instead, associate to each point a probability that quantifies how much the point belongs to a cluster.

The GMM is soft clustering method that exploits a Gaussian Mixture, i.e. a function composed by K Gaussians, where K indicates the number of clusters. Each Gaussian is described by:

- μ - mean of the Gaussian, that defines its center;
- Σ - covariance matrix, describes the dimensions of the ellipsoid;
- π - mixing probability, is the weight of the Gaussian component and tells us if the component is small or big.

The mixing probabilities are constrained by Equation 7.1.

$$\sum_{k=1}^{K} \pi_k = 1 \tag{7.1}$$

A Gaussian identifies a probability function, called density function described by the following Equation:

$$\mathcal{N}(\mathbf{x}|\mu, \Sigma) = \frac{1}{(2\pi)^{(D/2)}|\Sigma|^{(1/2)}} exp(-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)) \tag{7.2}$$

Where μ and Σ have been defined above, $\mathbf{x}$ are the data points under analysis and D is the dimension of each point.

We are dealing with multiple Gaussians, therefore multiple density function, that tell us which is the probability that a point, identified as $\mathbf{x}_n$, belongs to the $k-th$ Gaussian. Identifying a new variable z, called "latent variable", it is possible to describe the probability that $\mathbf{x}_n$ is produced by the Gaussian k as:

$$p(z_{nk} = 1|\mathbf{x}_n) \tag{7.3}$$

Moreover, the Bayes Rule helps us defining it as:

$$p(z_k = 1|\mathbf{x}_n) = \frac{p(\mathbf{x}_n)|z_k = 1)p(z_k = 1)}{\sum_{j=1}^{K} p(\mathbf{x}_n)|z_j = 1)p(z_j = 1)} \tag{7.4}$$

All the latent variables can be identified as $\mathbf{z}$:

$$\mathbf{z} = \{z_1, ..., z_K\} \tag{7.5}$$

They are independent one from the other and they are equal to 1 only if the point belongs to the cluster k.

Knowing that, we can re-define the mixing coefficient as in Equation 7.6.

$$\pi_k = p(z_k = 1) \tag{7.6}$$

Moreover, using the independence property of $\mathbf{z}$, the probability $p(\mathbf{z})$ is:

$$p(\mathbf{z}) = p(z_1 = 1)^{z_1} p(z_2 = 1)^{z_2} ... p(z_K = 1)^{z_K} = \prod_{k=1}^{K} \pi_k^{z_k} \tag{7.7}$$

By definition, the Gaussian function tells us which is the probability that a certain point belongs to it. Therefore, we can identify the probability of observing a point $\mathbf{x}_n$, given $\mathbf{z}$ as:

$$p(\mathbf{x}_n|\mathbf{z}) = \prod_{k=1}^{K} \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)^{z_k} \tag{7.8}$$

Using the Bayes Theorem we can obtain the distribution $p(\mathbf{x}_n|\mathbf{z})$, from Equations 7.7 and 7.8

$$p(\mathbf{x}_n, \mathbf{z}) = p(\mathbf{x}_n|\mathbf{z})p(\mathbf{z}) \tag{7.9}$$

$p(\mathbf{x}_n|\mathbf{z})$ is called joint distribution; in order to obtain $p(\mathbf{x}_n)$, which is the Gaussian Mixture definition itself, we have to marginalize the distribution as shown in Equation 7.10.

$$p(\mathbf{x}_n) = \sum_{k=1}^{K} p(\mathbf{x}_n|\mathbf{z})p(\mathbf{z}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k) \tag{7.10}$$

Remembering Equation 7.4 and that:

$$p(z_k = 1) = \pi_k \tag{7.11}$$

$$p(\mathbf{x}_n|z_k = 1) = \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k) \tag{7.12}$$

We obtain:

$$p(z_k = 1|\mathbf{x}_n) = \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n|\mu_j, \Sigma_j)} = \gamma(z_{nk}) \tag{7.13}$$

Having the Mixture Model, we now need to define which are the optimal values for it. A possible way to do it, is to maximise the likelihood of the problem. The likelihood $p(\mathbf{X})$ is defined as the joint probability of the data points $\mathbf{x}_n$.

$$p(\mathbf{X}) = \prod_{n=1}^{N} p(\mathbf{x}_n) = \prod_{n=1}^{N} \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k) \tag{7.14}$$

Working with the $ln(p(\mathbf{X}))$ is easier, so we obtain:

$$ln(p(\mathbf{X})) = \sum_{n=1}^{N} ln(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)) \tag{7.15}$$

The method used to solve this problem and obtain the correct components is called **Expectation - Maximization** algorithm, an iterative method often used in the optimization problems.

Algorithm 8 illustrates the procedure.

Algorithm 8 Expectation - Maximization algorithm

▷ **Initialization**
▷ Initialize the parameter randomly or using other algorithm (e.g. K-means)
$\theta = \{\pi, \mu, \sigma\}$
▷ **Expectation**
▷ Compute $Q(\theta^*, \theta)$

$$Q(\theta^*, \theta) = \mathbb{E}[lnp(\mathbf{X}, \mathbf{Z}|\theta^*)] = \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X}, \theta) ln(p(\mathbf{X}, \mathbf{Z}|\theta^*)) \tag{7.16}$$

▷ For the GMM can be rewritten as

$$Q(\theta^*, \theta) = \sum_{\mathbf{Z}} \gamma(x_{nk}) ln(p(\mathbf{X}, \mathbf{Z}|\theta^*)) \tag{7.17}$$

▷ With $p(\mathbf{X}, \mathbf{Z}|\theta^*)$ being the joint probability of all observations and latent variables

$$p(\mathbf{X}, \mathbf{Z}|\theta^*) = \prod_{n=1}^{N} \prod_{k=1}^{K} \pi_k^{z_{nk}} \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)^{z_{nk}} \tag{7.18}$$

$$ln(p(\mathbf{X}, \mathbf{Z}|\theta^*)) = \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk}[ln(\pi_k) + ln(\mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k))] \tag{7.19}$$

▷ Substituting, $Q(\theta^*, \theta)$ becomes

$$Q(\theta^*, \theta) = \sum_{n=1}^{N} \sum_{k=1}^{K} \gamma(z_{nk})[ln(\pi_k) + ln(\mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k))] \tag{7.20}$$

▷ **Maximization**
▷ Compute θ^*

$$\theta^* = \underset{\theta}{\operatorname{argmax}}\, Q(\theta^*, \theta) \tag{7.21}$$

▷ Include in Equation 7.20 the constraints of π_k, using a Lagrange multiplier

$$Q(\theta^*, \theta) = \sum_{n=1}^{N} \sum_{k=1}^{K} \gamma(z_{nk})[ln(\pi_k) + ln(\mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k))] - \lambda(\sum_{k=1}^{K} \pi_k - 1) \tag{7.22}$$

12: ▷ Derivate $Q(\theta^*, \theta)$ with respect to π and set it equal to 0

$$\frac{\partial Q(\theta^*, \theta)}{\partial \pi_k} = \sum_{n=1}^{N} \frac{\gamma(z_{nk})}{\pi_k} - \lambda = 0 \tag{7.23}$$

$$\sum_{n=1}^{N} \gamma(z_{nk}) = \pi_k \lambda \Rightarrow \sum_{k=1}^{K} \sum_{n=1}^{N} \gamma(z_{nk}) = \sum_{k=1}^{K} \pi_k \lambda \tag{7.24}$$

13: ▷ Since $\lambda = N$, we obtain π_k^*

$$\pi_k^* = \frac{\sum_{n=1}^{N} \gamma(z_{nk})}{N} \tag{7.25}$$

14: ▷ Repeating the process of differentiation for μ and Σ:

$$\mu_k^* = \frac{\sum_{n=1}^{N} \gamma(z_{nk}) \mathbf{x}_n}{\sum_{n=1}^{N} \gamma(z_{nk})} \tag{7.26}$$

$$\Sigma_k^* = \frac{\sum_{n=1}^{N} \gamma(z_{nk})(\mathbf{x}_n - \mu_k)(\mathbf{x}_n - \mu_k)^T}{\sum_{n=1}^{N} \gamma(z_{nk})} \tag{7.27}$$

7.2.1 MATLAB implementation - Approach 1

The Gaussian Mixture Model algorithm has been implemented in MATLAB and different approaches have been used, in order to test its behaviour.

First of all, it was necessary to simulate the the behaviour of the LIDARs starting from some obstacles in the environment. This was done using the MATLAB toolbox **Mobile Robot Algorithm Design** [40]. The step needed to obtain the simulated LIDAR are:

1. Define the LIDAR position in a global frame;
2. Define the LIDAR radius, i.e. a radius beyond which nothing is perceived;
3. Define the scanning angle, which defines the density of the LIDAR points;
4. Define the obstacle in the environment;
5. Define a blank binary occupancy matrix;
6. Set the occupancy matrix with the obstacles;
7. Use the toolbox to simulate the LIDAR behaviour;

The toolbox outputs two vectors:

1. Scanning angles θ_{lidar} - identifies the angle of the measure;
2. Ranges r_{lidar} - identifies the measure, i.e. a distance that tell us how far is a point.

If there is no obstacle in correspondence of a certain scanning angle, the range is set as infinite. For convenience reasons, it is better to remove these ranges from the vector.

Remembering that the robot pose is defined by the vector $\xi = [x, y, \theta]^T$, we can rebuild the LIDAR points in the global frame using Equations 7.28 and 7.29.

$$R = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \tag{7.28}$$

$$Lidar\_points = \begin{bmatrix} x_{lidar} \\ y_{lidar} \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + R \cdot \begin{bmatrix} r_{lidar} \cdot cos(\theta_{lidar}) \\ r_{lidar} \cdot sin(\theta_{lidar}) \end{bmatrix} \tag{7.29}$$

These points represent the perceived data points of the LIDAR, therefore they describe the border of the obstacle. Since we want to take into account the

dimension of the robot r_{robot} and a safety factor ϵ_{safe}, Equation 7.30 describes a new set of data points that are closer to the center of the robot exactly of $r_{robot} + \epsilon_{safe}$.

$$\begin{bmatrix} x_{lidar} \\ y_{lidar} \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + R \cdot \begin{bmatrix} (r_{lidar} - r_{robot} - \epsilon_{safe}) \cdot cos(\theta_{lidar}) \\ (r_{lidar} - r_{robot} - \epsilon_{safe}) \cdot sin(\theta_{lidar}) \end{bmatrix} \tag{7.30}$$

We will use these new set of points to model the GMM. As first thing, we assume to know the position of the centroids of each obstacle. This is plausible because of the already existing image processing algorithm that detects the people around the robot.

Knowing the position of the centroids, it is possible to associate each data point to a centroid using a distance metrics: a data point is assigned to the nearest centroid. In this way clusters of points are defined.

Using the GMM algorithm on each of this cluster, choosing $K = 2$ components for each GMM we obtain the result shown in Figure 7.2.

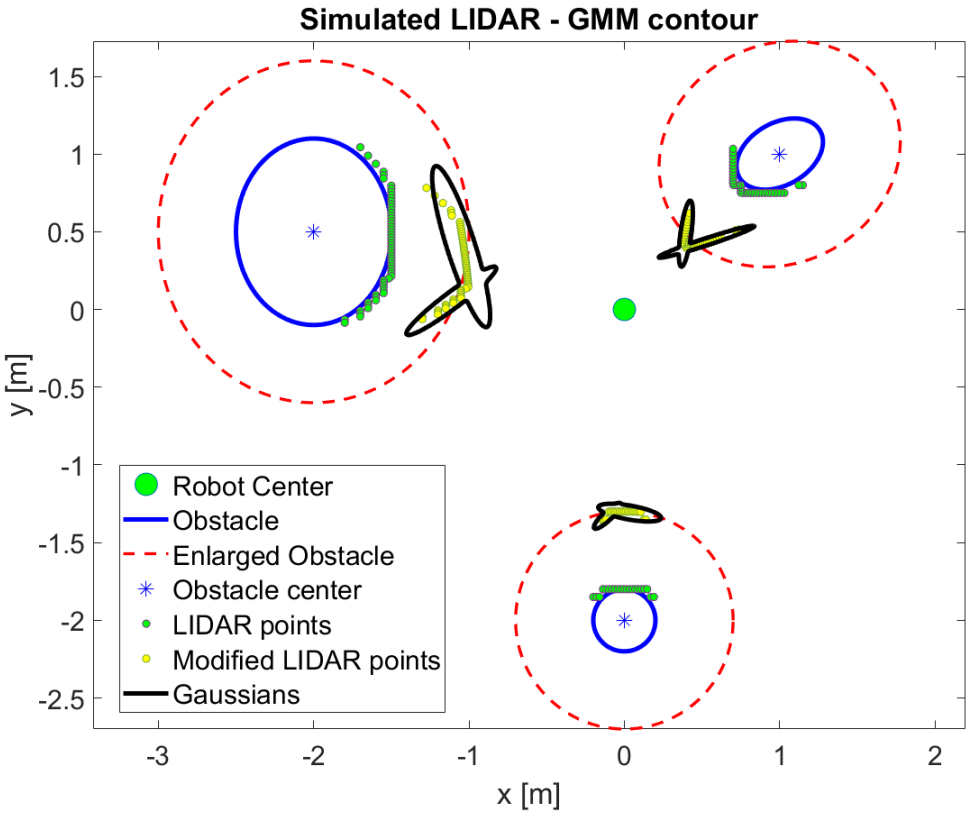


Figure 7.1: LIDAR simulation - GMM approach 1

It is important to point out that, in a real situation, the robot does not see the real obstacles (in Figure the blue ellipses), but only the lidar points (in Figure the green dots). Therefore, after the use of this algorithm, what we consider as obstacles are the black shapes in Figure.

We have chosen to use $K = 2$ so that, for each obstacle, the star-shaped condition are respected: we only need to chose as reference point for the modulation a point that is in the common area of the two Gaussians. Some problems could emerge if two obstacles are so close that the two GMMs overlaps, but also this situation could be solved using the Python implementation of the modulating algorithm.

The process illustrated above must be repeated at each time step to model the obstacles. Without the use of some memory process, that algorithm produces very noisy results, that can hardly be used in a real scenario. Moreover, the resulting shapes are thinner with respect to the real obstacle, and do not cover an area comparable with the real obstacle's one. This issue can be solved increasing the dimension of the axes of the Gaussian component, but this would mean that the information about the probability of finding an obstacle in a certain position would lose their importance.

All these aspects make the algorithm unfeasible for the real implementation, but suggest us which are the main problems to tackle and improve.

7.2.2 MATLAB implementation - Approach 2

This algorithm, as the one explained in Section 7.2.1, assumes that the centroids' positions obtained with the image processing algorithm are known. Also in this case, using the MATLAB Tool **Mobile Robot Algorithm Design** we simulate the output of a LIDAR. The Equations used are 7.28 and 7.29.

Differently from the previous approach, we use directly the lidar points, without moving them toward the robot. Firstly, a labelling algorithm is applied to the data points in order to assign each of them to a centroid. Then, using the centroid as reference point, the lidar point of an obstacle are mirrored. This process assumes that the obstacle has an ellipsoidal shape.

Being $lidar\_point_j$ and $centroid_j$ respectively the data points and the centroid of obstacle j, we obtain the mirrored points $M\_lidar\_point_j$ simply applying Equation 7.31.

$$M\_lidar\_point_j = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} (lidar\_point_j - centroid_j) + centroid_j \quad (7.31)$$

Having all the information on the number of obstacles and the data points of the LIDAR, a GMM algorithm is applied to each cluster. It is possible to choose between spherical, diagonal and full covariance matrices. Figure 7.2 illustrate the

output of the algorithm, using spherical covariance matrix. Differently from the previous algorithm, in this one the learned shape of the obstacle is very similar to the real one and it is also able to surround the real obstacle. Nevertheless, some issues must be pointed out:

1. It has not been taken into account the robot dimensions and the safety distance yet;
2. The shape of the learned obstacle depends on the shape of the real one in a non-linear way; therefore, if the real obstacle has a large shape, the contour defined by the GMM may be too big to be used in the real implementation. On the contrary, if it is too small, the contour may not guarantee the avoidance of the collision.

Despite the problems above listed, this algorithm shows good potentiality and further investigations, with real data points obtained from the robot, will be discussed in the following Sections.

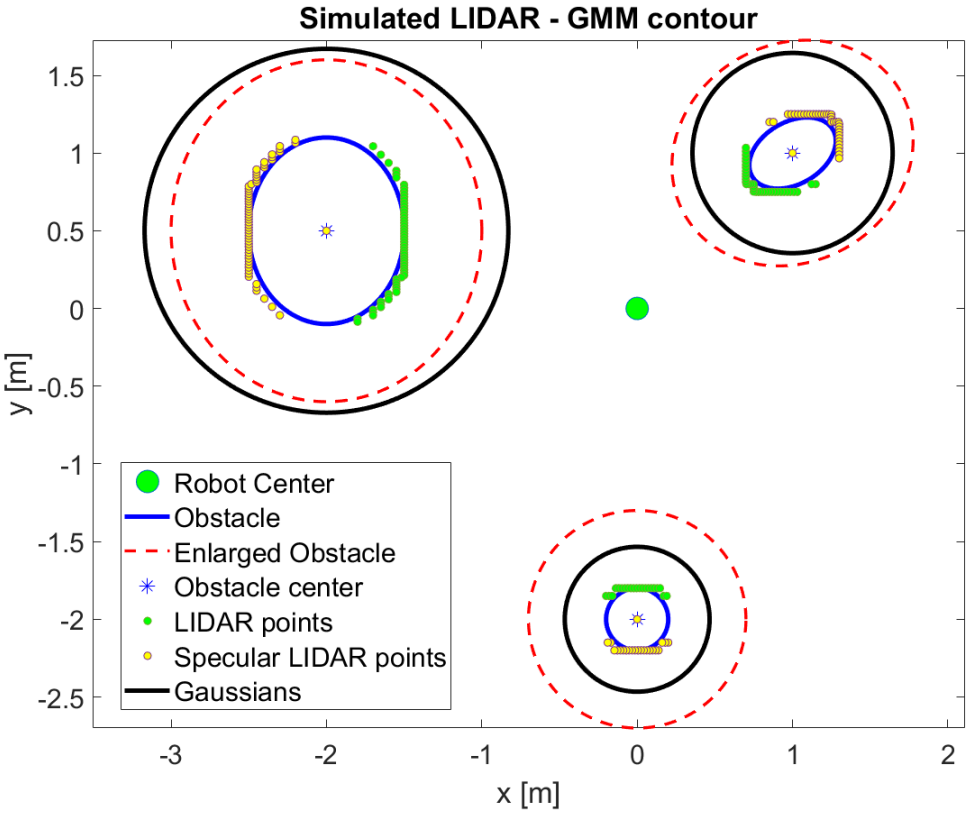


Figure 7.2: LIDAR simulation - GMM approach 2

7.2.3 Python implementation

In the previous sections the GMM algorithm has been implemented in MATLAB for didactic purposes; no optimizations were included. In this section we will consider also the timing constraints imposed by the robot hardware; therefore, a careful analysis must be done in order to choose an algorithm with the right characteristics.

Since the algorithm on the robot runs on Python, the obstacle detection algorithm, that uses GMM, has been developed also in this language, using the real data points obtained by the LIDARs. In this case, though, the GMM algorithm used was already implemented. The **scikit-learn** [41] library has been used, because it contains several Machine Learning algorithms that are optimized and can be adapted to different scenarios.

In particular, two of them have been implemented:

1. sklearn.neighbors.LocalOutlierFactor [42] is an algorithm that excludes the outliers of a data set. It has been used to filter the lidar points before applying the GMM algorithm, in order to reduce the noise caused by the presence of spurious points.

2. sklearn.mixture.GaussianMixture [43] is the algorithm that models the data points using the GMM.

Outliers exclusion

The outliers exclusion algorithm is not compulsory: it is just a pre-processing step that could improve the data points to analyse, but could also slow down the computation or, worse, remove some important information about the environment. Being Python an object oriented language, we must define the object that will deal with the outlier removal. The object is generated from the class *"sklearn.neighbors.LocalOutlierFactor()"*. The parameters that we are going to change are:

- $n\_neighbors$ - The number of neighbors to get (by default is 20);
- *contamination* - The proportion of the outliers in the data set considered;
- n_jobs - The number of processors to use for the computation;
- $algorithm$ - Typology of algorithm used to compute the nearest neighbors;
- $leaf\_size$ - A parameter peculiar for the algorithm used, it can affect the speed of the computation.

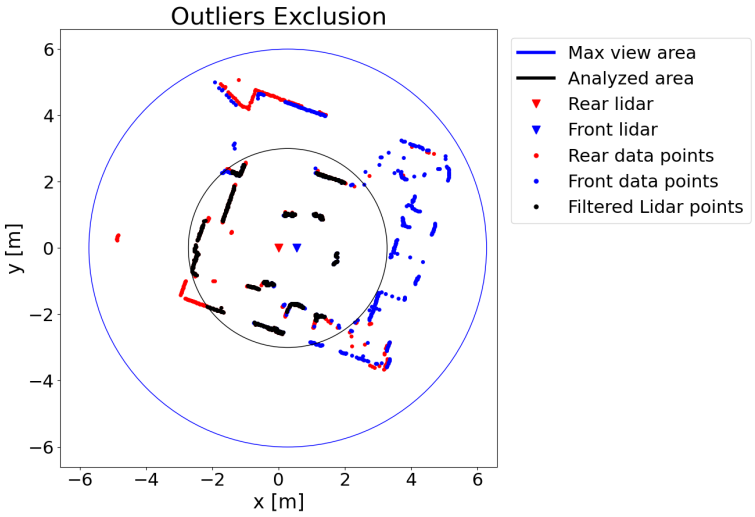
The environment perceived by the robot might be very crowded, therefore, analysing all the data points obtained by the LIDARs could increase the computation time uselessly if these points are quite far from the robot. For this reason, two variables are introduced:

- max_view_rad - The maximum distance beyond which the data points are ignored;
- $analysis\_rad$ - The radius that defines the area under analysis.

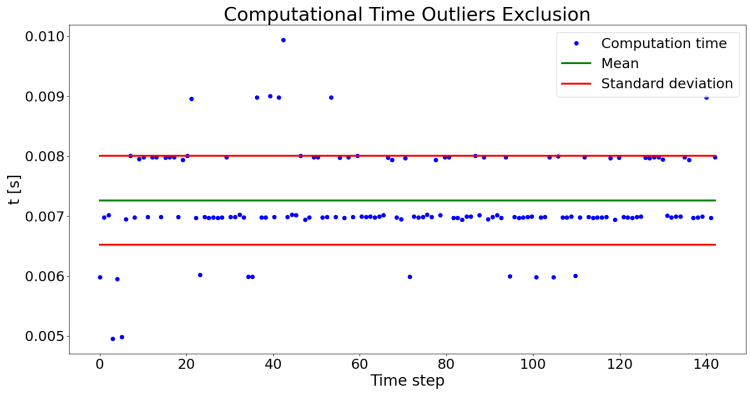
Considering only the data points withing the $analysis\_rad$, we can observe the behaviour of the outliers excluder algorithm, in different configurations:

- **Test 1** - The parameters are
  - $n\_neighbors = 20$;
  - $contamination = 0.1$;
  - $n\_jobs = 1$;
  - $algorithm =$ "$ball\_tree$";
  - $leaf\_size = 30$;
  - $max\_view\_rad = 6m$;
  - $analysis\_rad = 3m$.

  Figure 7.3 illustrates the results of the algorithm on the left and the computational time on the right. In the image that shows the results, the black dots represent the output of the algorithm. The performance seems excellent because all the small clusters of dots are kept out. Moreover, the computation time, with the exception of very few cases, is very low and compatible with the timing required on the robot.



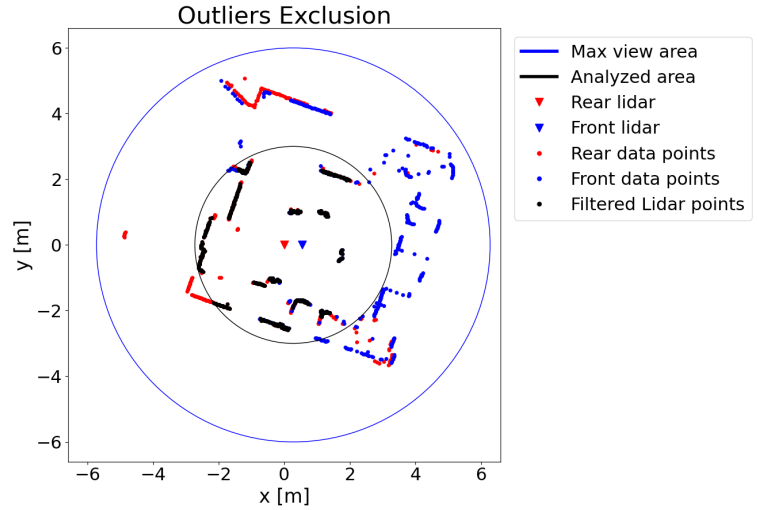
(a) Exclusion Result



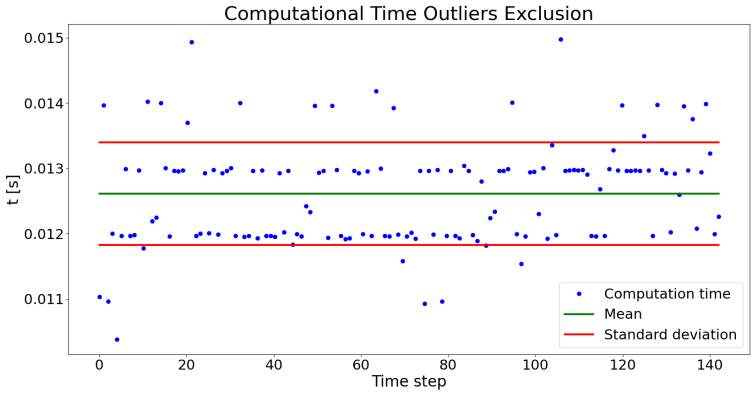
(b) Exclusion Time

Figure 7.3: Outliers Exclusion - Test 1

- **Test 2** - The same parameters as in Test 1 are used, but in this case more cores are used: $n\_jobs = 4$. The results are shown in Figure 7.4. Surprisingly, the computation timing in this scenario increases. This may be caused by the fact that a virtual machine is used to run Python or because the calculations are so fast that the computer takes more time dividing the operations on multiple cores, rather than doing everything on a single core.



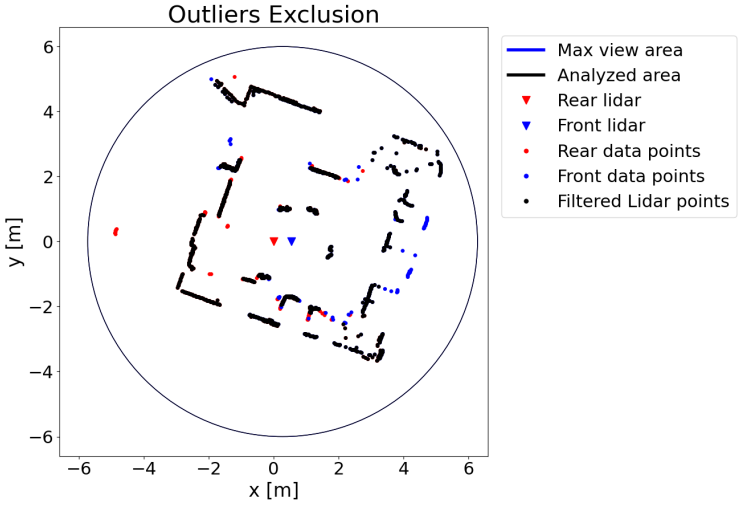
(a) Exclusion Result



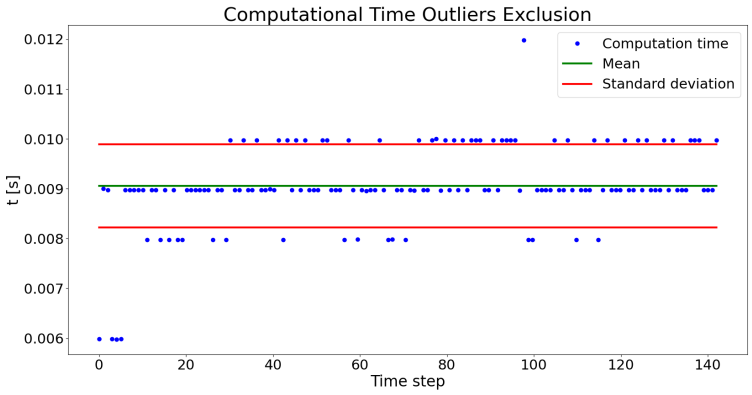
(b) Exclusion Time

Figure 7.4: Outliers Exclusion - Test 2

- **Test 3** - Using again a single core ($n\_jobs = 1$) and the same parameters used in the first two tests, we want to understand if the algorithm is able to deal also with a larger number of data points. For this reason, we set $max\_view\_rad = analysis\_rad = 6m$. The results obtained and shown in Figure 7.5 are very similar to the one obtained in Test 1. Also in this case, the computation time at the first step is larger, but in general the timing constraints are respected. Also the behaviour of the algorithm performs in a very good way.



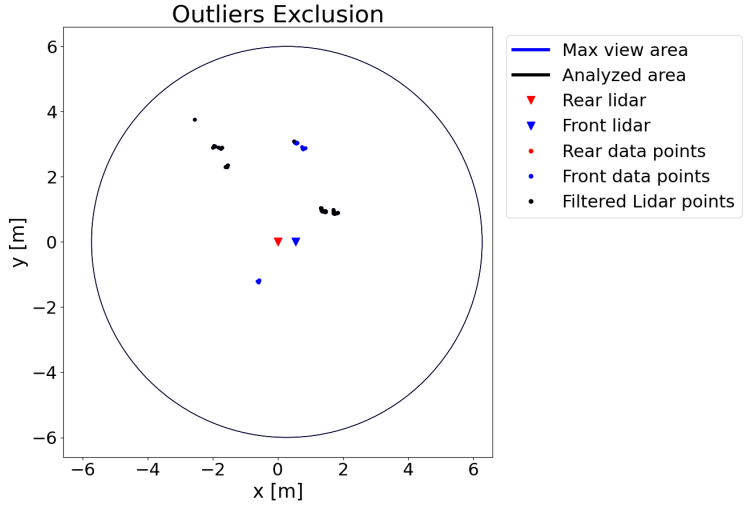
(a) Exclusion Result



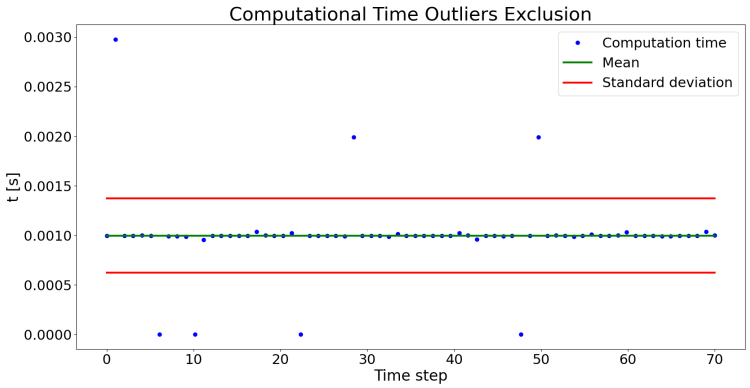
(b) Exclusion Time

Figure 7.5: Outliers Exclusion - Test 3

- **Test 4** - Figure 7.6 shows a test done on another data set. In this case the data points from the lidar are quite few and applying this algorithm can cause different problems: for instance, the points of an obstacle can be lost; if this happens, the obstacle can not be modelled or can be modelled just partially, causing a possible dangerous situation.



(a) Exclusion Result



(b) Exclusion Time

Figure 7.6: Outliers Exclusion - Test 4

GMM implementation

The goal of this Chapter is to model the obstacles around the robot using a Gaussian Mixture Model. Differently from Sections 7.2.1 and 7.2.2, the assumption regarding the knowledge of the position of the people around the robot are dropped. The reasons are simple: firstly, there could be other typologies of obstacles, not only people, therefore using the centroids of the people may not be sufficient to model the entire environment. Moreover, and this is the biggest impediment, the algorithm that produces the centroids of the people is very slow and produces often false positives. Therefore, using the algorithm described in section 7.2.2 would be impossible.

Starting from this premise, the algorithm here developed must be able to cluster the entire environment, not only a partial set of data points. As said above, the class that has been used is $sklearn.mixture.GaussianMixture()$. It is possible to generate an object that models the GMM on a given data set. Before doing so, it is necessary to understand which are the parameters that compose the class:

- n_components - the number of components K;
- covariance_type - identifies the typology of covariance matrix to use; for our purpose the typologies that will be taken into account are 'full', 'diag' and 'spherical', that correspond, respectively , to
  - full covariance matrix: all the elements are different and it corresponds to an ellipse that is tilted with respect to the global reference frame;
  - diagonal covariance matrix: it has elements only on it diagonal, therefore it corresponds to an ellipse with the minor and major axes oriented along the coordinate axes;
  - spherical covariance matrix: all the elements on the diagonal are the same; it corresponds to a circle in a 2D environment, such the one under analysis.
- warm_start - a Boolean variable that allows the GMM computation to keep track of the previous runs;
- max_iter - the maximum number of iterations allowed by the system before interrupting the GMM computation.

Test 1

Considering the data points obtained from the crowed environment and $max\_view\_rad = analysis\_rad = 6m$, we can generate the object that computes the GMM with:

- n_components = 15;
- covariance_type = 'full';
- warm_start = False;
- max_iter = 20.

As it has been explained before, a GMM is composed by a weighted sum of Gaussians. To simplify the procedure, we will consider the single components, without the weights and the sum. Therefore, we have $K = n\_components = 15$ components and each of them is considered as an obstacle. A useful feature of the class that we are using reduces automatically the number of components if these are not needed; consequently, less than 15 obstacles might be considered.

Of course, not considering the weights and the sum does not allow us to exploit the full potentiality of the GMM algorithm, because the information on the probability of the presence of an obstacle are not used correctly. Nevertheless, Figure 7.7 illustrates which is the output of the GMM algorithm on the data set under analysis.

The axis dimension of each Gaussian Component is computed starting from the covariance matrix. Assuming a generic covariance matrix as:

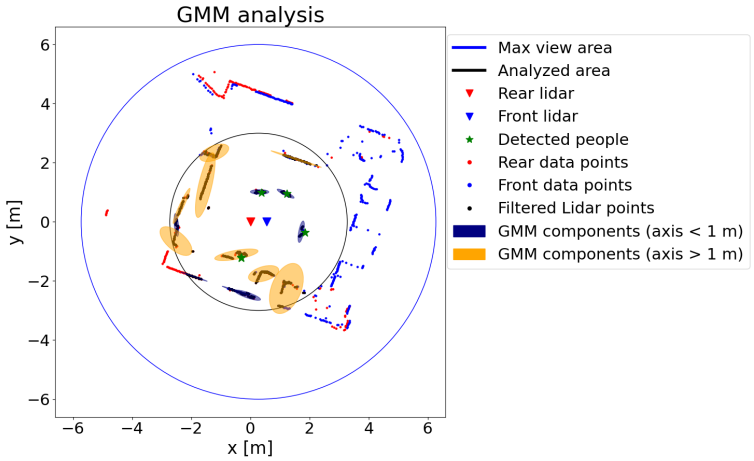
$$Cov = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \tag{7.32}$$

The axes length are:

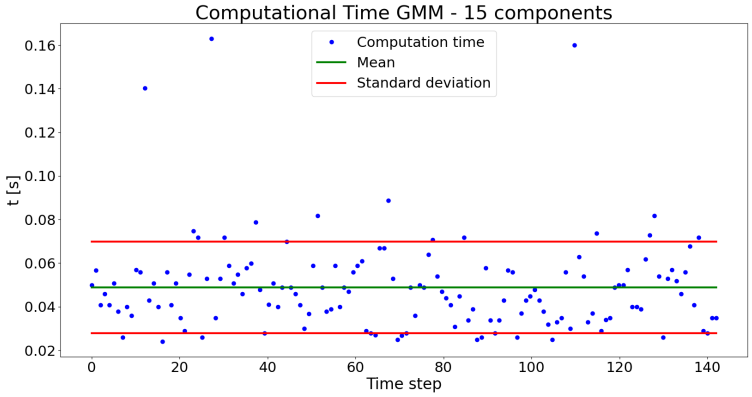
$$\begin{aligned} d_x &= \sqrt{3}a \\ d_y &= \sqrt{3}d \end{aligned} \tag{7.33}$$

While b and c gives us information about the tilted angle.

For visualization reasons, if both axes are smaller than 1 meter, the Gaussian component i represented in blue, otherwise in orange. Figure 7.7 shows that the components produced are too big because they try to model an environment too full, with a number of components that is not sufficient. Moreover, the computation time does not respect the constraints imposed by the robot.



(a) GMM Result



(b) GMM Time

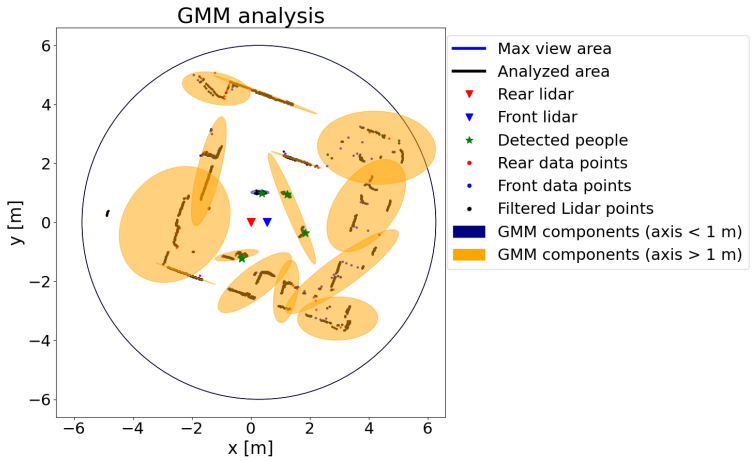
Figure 7.7: GMM - Test 1

Test 2

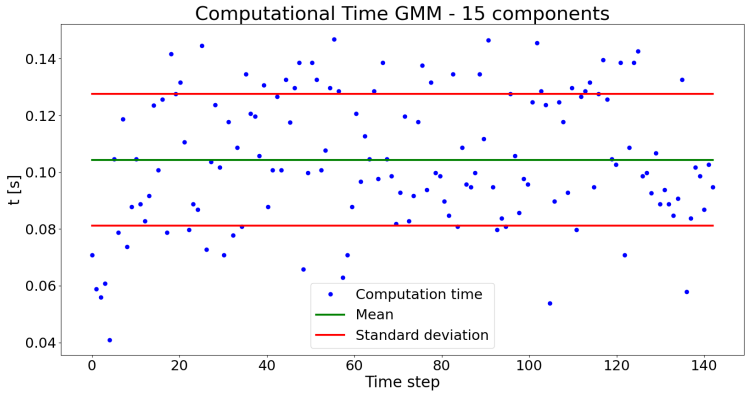
In order to reduce the computation time, we reduce the analyzed area modifying $analysis\_rad = 3m$. Figure 7.8 shows that, using the same number of components, the result is much better, both in the performance and in the computation time.

Also in this case there are some components that are quite large, but they fit in a better way the lidar points.

The computation time is quite feasible with the robot hardware, but it would be better to reduce it. Moreover, running the algorithm on the data set that changes over time, produces GMM components that vary quite a lot. This results in a noisy description of the obstacles.



(a) GMM Result



(b) GMM Time

Figure 7.8: GMM - Test 2

Test 3

With this test we try to solve the problems mentioned in Test 2, adding the memory component in the GMM: warm_start = True. In this way, the GMM algorithm takes trace of the previous calculations, and reduces the computation time. Figure 7.9 confirms the hypothesis. Unfortunately, Figure 7.10 shows that using the previous values of the GMM to compute the next one does not produce the desired behaviour. In the first step, the GMM behaves in a correct way, modelling the lidar points nicely. Nevertheless, this good behaviour is lost very quickly and the components of the GMM tend to cover the entire environment. As a matter of fact, when the simulation is over, the number of components is reduced to $K = 2$, even if the initial number was $K = 15$.

The reason is that, probably, this kind of feature (i.e. the memory of the GMM) performs properly in a semi-static environment, where the data points do not change very much. In the situation that this project wants to tackle, we have the opposite scenario: an environment very dynamic that can change very quickly.

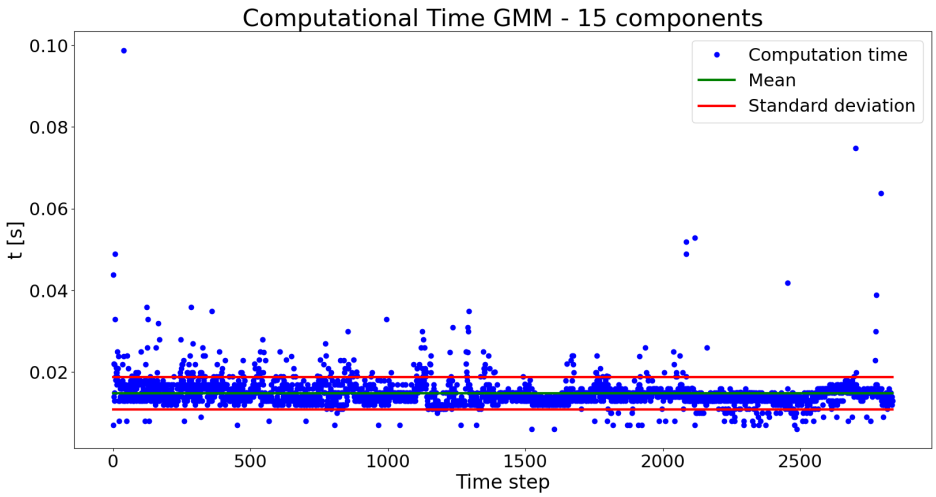
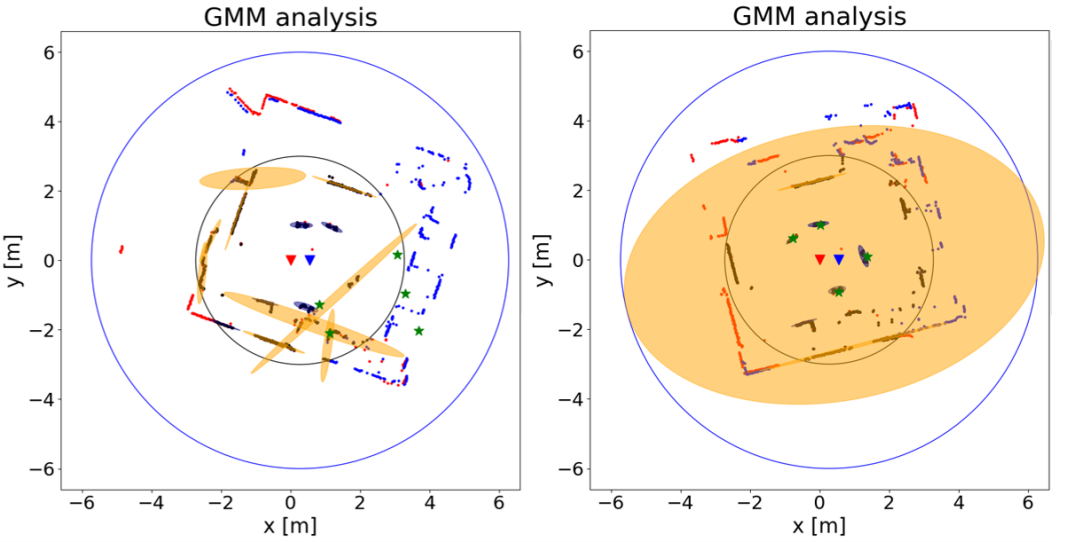
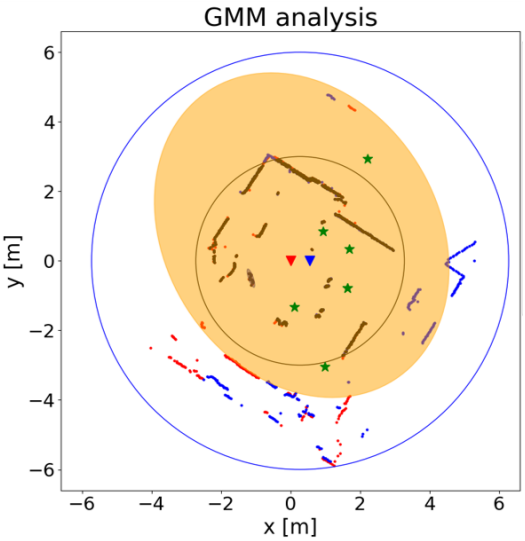


Figure 7.9: GMM - Test 3 - Time

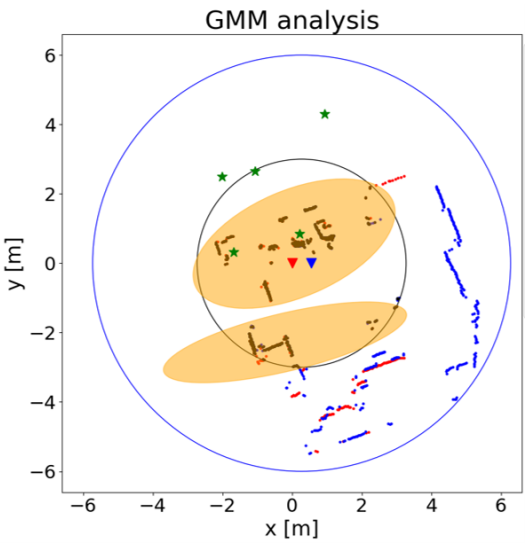


(a) GMM Result - First step

(b) GMM Result - Intermediate step 1



(c) GMM Result - Intermediate step 2



(d) GMM Result - Final step

(e)

Figure 7.10: GMM - Test 3

Test 4

In Test 3 it has emerged that using the memory feature of the GMM class, it is possible to speed up the algorithm, but at the same time, it is necessary a semi-static environment.

A possible solution to overcome the problem related to the static constraints of the environment is to reset the memory every T_{reset} seconds. In this way, for T_{reset} seconds the GMM uses the information about the previous calculations and speed up the process; then everything is resetted and the GMM algorithm virtually sees a new static environment. Of course, T_{reset} has to be short enough to guarantee the hypothesis on the static environment, but also long enough to avoid a constant reset of the memory, creating the same situation of Test 1 and Test 2.

Setting $T_{reset} = 1s$, the simulation results are obtained in Figure 7.11 and 7.12. The first one illustrates the time necessary at each step to compute the GMM model. On average, the time constraints are respected, but there are a lot of steps in which the computation time needed exceed the robot limits. In Figure 7.12 are shown four consequent time steps; after the first two, the GMM model is resetted and therefore, new Gaussian Components are considered.

It appears from the tests done until now that the main issues of this algorithm are related to the presence of walls. The Gaussian components have to model both the people around the robot and the walls, but this cause the model to be very imprecise.

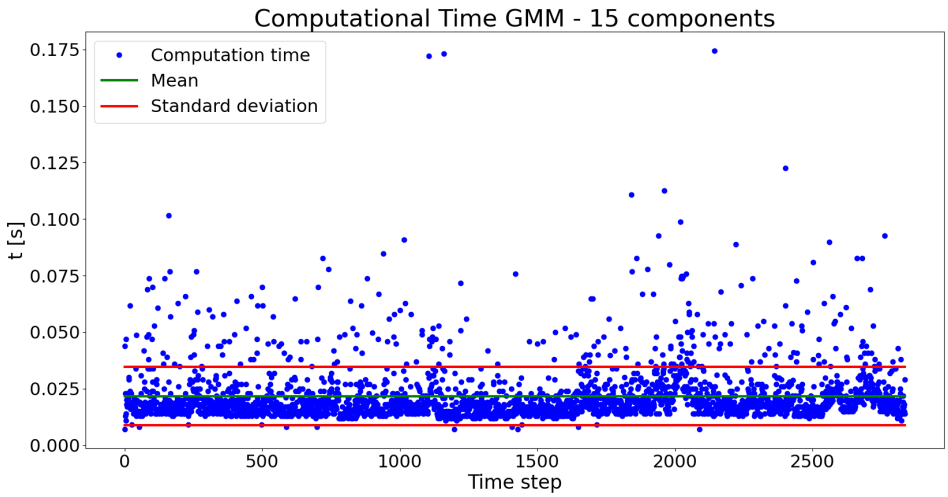


Figure 7.11: GMM - Test 4 - Time

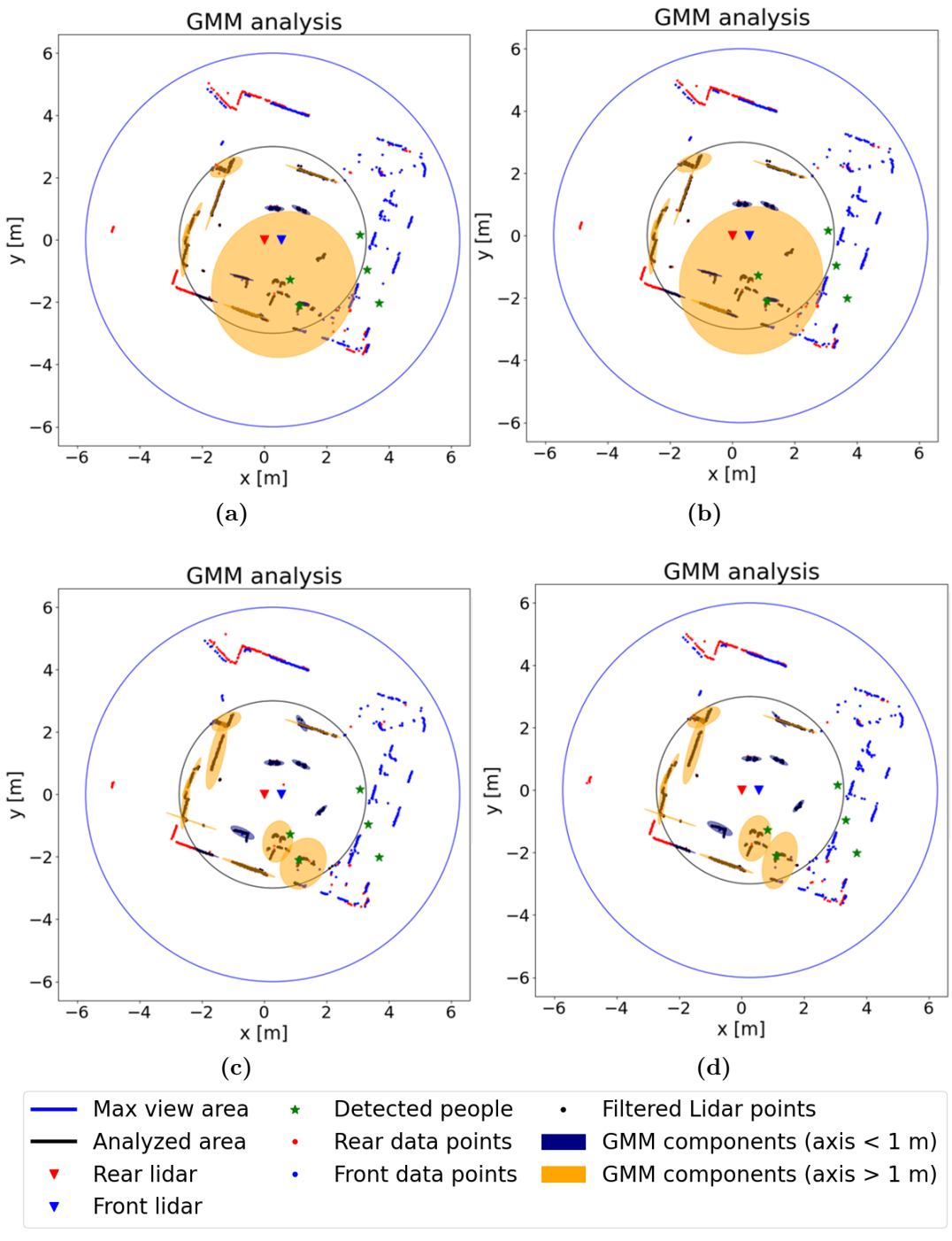


Figure 7.12: GMM - Test 4

Test 5

In order to test the algorithm without the presence of walls, a new data set, with only few people has been recorded. Setting $max\_view\_rad = analysis\_rad = 6m$ we can observe a larger area and take into account the obstacles even if they are quite far.

The results shown in Figure 7.13 are promising: the computation time is in the limits, therefore it could be used in real time. Nevertheless, there are still some problems in the output of the GMM algorithm. Figure 7.14 illustrates four consequent time step; in each of them the GMM components have different shapes; moreover, some GMM components fit perfectly the data points corresponding to a single leg of a person, while other components fit more than one person. The result is that some components are extremely precise and other cover an area way too big and that does not correspond to a real obstacle.

The algorithm could ignore the GMM components that have axes' length bigger than a given value, but this may lead to a loss on information that could compromise the safety of the user and of the people around him.

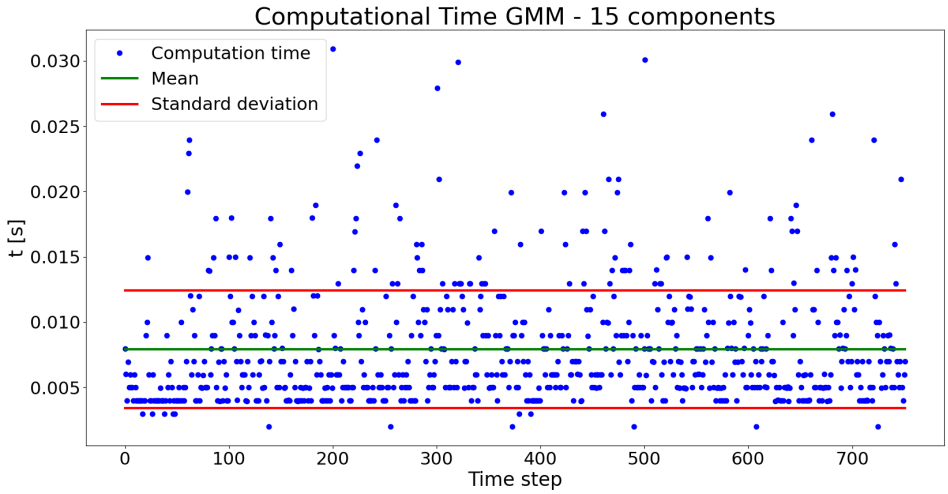


Figure 7.13: GMM - Test 5 - Time

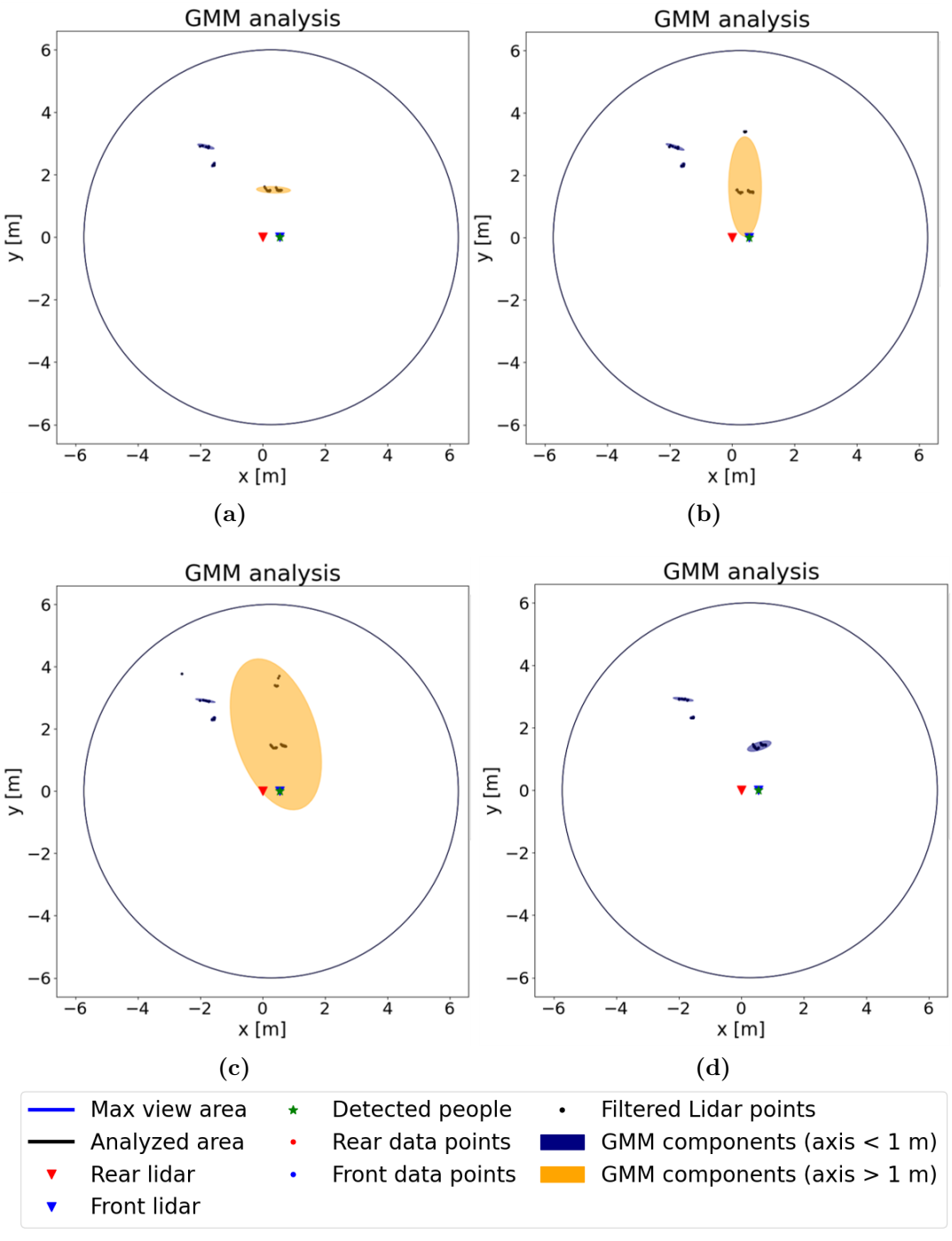


Figure 7.14: GMM - Test 5

Test 6

In this test a new approach is tried: first of all, the covariance matrix is set as 'spherical', in order to model each obstacle as a circle. The number of components is reduced to $K = 10$. Then, after the GMM algorithm, a new algorithm is implemented, that merges together all the centroids, and consequently the covariance matrices, that are closer than a given distance d_{fuse}. In this way, if the GMM algorithm models the legs of a person separately, the algorithm automatically produces a single obstacle.

The algorithm structure is described in Algorithm 9. It outputs the fused centroids and the respective covariance matrices.

Testing this new implementation with the data set containing only people, and using $d_{fuse} = 0.7m$ we obtain the results illustrated in Figures 7.15 and 7.16. Now the time constraints are fully respected: this is caused mainly by the reduction of components used and by the spherical covariance matrices; the GMM algorithm has to find a single value that describe the dispersion of the data points in the space, instead of four as in the 'full' covariance matrix.

As for the results, Figure 7.16 displays four scenarios that are not one after the other, but represent similar situations. The problem related to the presence of Gaussian components that cover an area way bigger than the real obstacle to model is not solved yet.

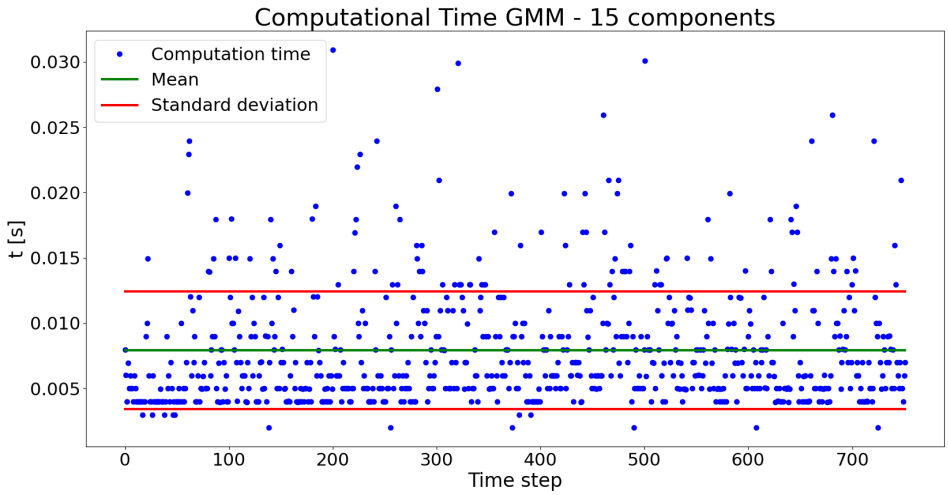


Figure 7.15: GMM - Test 6 - Time

Algorithm 9 Fuse algorithm $fuse()$

procedure FUSE($means, cov, d_{fuse}$)
    ▷ $means$ are the centroids of the GMM model
    ▷ cov are the covariance matrices of the GMM model
    ▷ d_{fuse} is the distance that the algorithm uses to fuse the points
    ▷ ret_means is a vector that contains the fused centroids
    $ret\_means \leftarrow [\ ]$
    ▷ ret_cov is a vector that contains the fused covariance matrices
    $ret\_means \leftarrow [\ ]$
    ▷ $taken$ contains information about the centroids that have been merged
    $taken \leftarrow [False, ..., False]$
    for i = 0:length(means) **do**
        if taken[i] == False **then**
            $count = 1$
            $point \leftarrow means[i]$
            $cov_i \leftarrow cov[i]$
            $taken[i] \leftarrow True$
            for j = i+1:length(means) **do**
                if $\|means[i] - means[j]\| < d_{fuse}$ **then**
                $point = point + means[j]$
                $cov_i = cov_i + cov[j]$
                $count = count + 1$
                $taken[j] \leftarrow True$
                end if
            end for
            $point = \frac{point}{count}$
            $ret\_means \leftarrow append(point)$
            $ret\_cov \leftarrow append(cov_i)$
        end if
    end for
    ▷ **Return** ret_means, ret_cov
end procedure

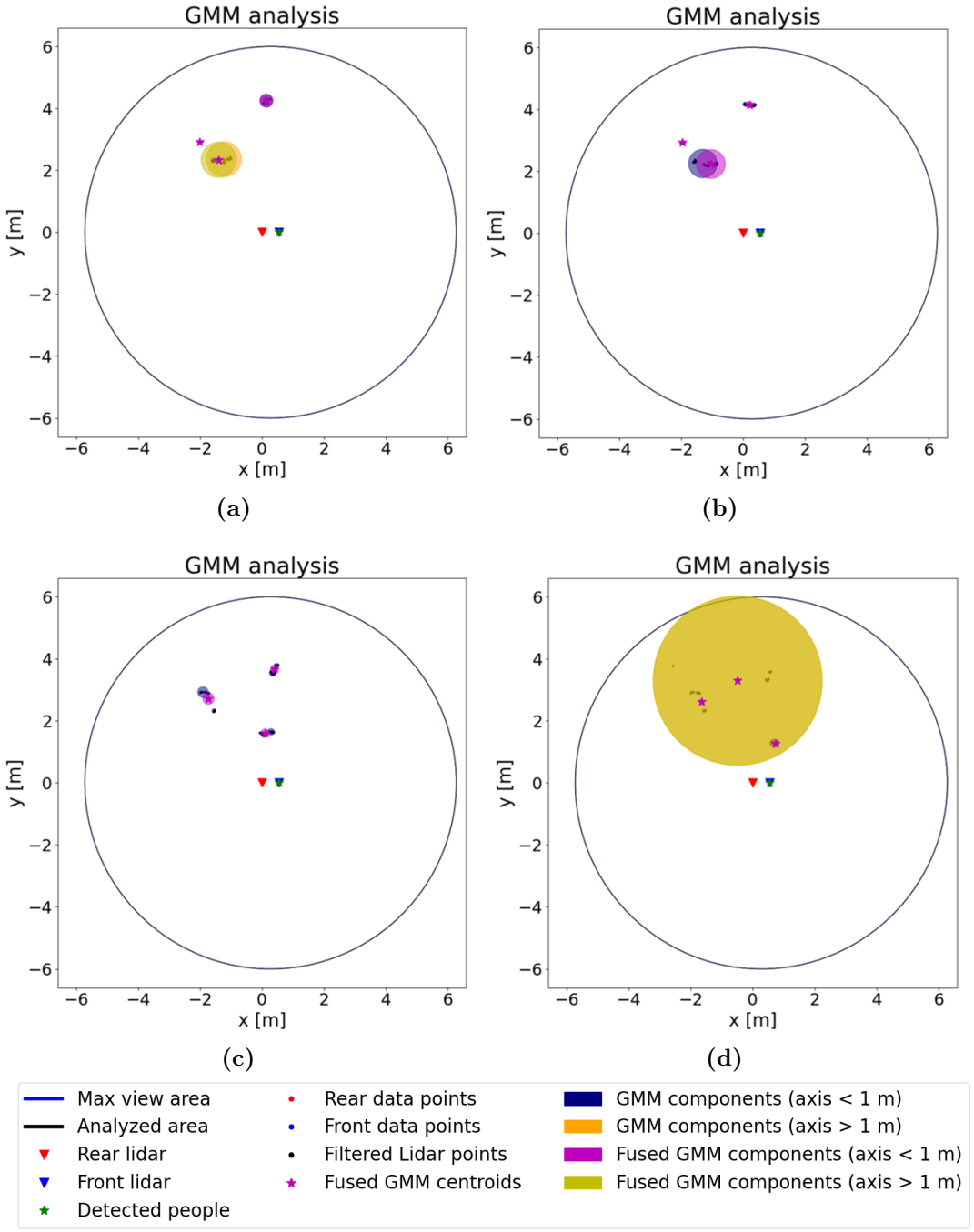


Figure 7.16: GMM - Test 6

7.3 K-means [9] [44] [45]

The K-means algorithm is one of the simplest and most known unsupervised clustering algorithms. It is an hard-clustering algorithm since it does not associate any probability to the data points of the data set under analysis. It subdivides the data points in K clusters using a distance metrics, generally reducing the sum of squares. The final output of the algorithm are a set of centroids, that identify the center of each cluster, and a set of labels for each data point.

Algorithm 10 illustrates the procedure.

Algorithm 10 K-means

▷ **Initialization**
▷ Choose a number K of clusters
▷ Randomly initialize K centroids
▷ Define a maximum number of iterations max_{iter}
▷ Define a precision threshold for the centroids $centroids_{th}$, i.e. a measure of how much the centroids change between two consecutive iterations
$n_{iter} = 0$ ▷ Initialize the iteration number
while $n_{iter} < max_{iter}$ or $precision > centroids_{th}$ **do**
    ▷ Compute the distance of every point from the centroids
    ▷ Assign each point to the nearest centroid
    ▷ Compute the mean of every cluster
    ▷ Update the centroids position
    $centroids \leftarrow means$
    ▷ Compute the $precision$
    ▷ Update n_{iter}
    $n_{iter} \leftarrow n_{iter} + 1$
end while

As in the previous sections, the K-means algorithm was not developed from scratch: the Python library "**sklearn.cluster.k_means**" [46] was used because it is able to optimize the process. The parameters considered for the implementation are:

- n_clusters - the number of clusters to compute, it corresponds to K;
- max_iter - is the maximum number of iteration before stopping the computation;
- n_init - number of runs of the algorithm, changing the initial guess for the centroids.

Differently from the GMM, the K-means algorithm does not associates circles or ellipsoid to each centroid, therefore, it is necessary to build a shape staring from the centroids and the labels associated.

Three methodologies are used and, for simplicity, we call them with the same name of the covariance matrices of the GMM:

- 'full' - the *Numpy* function *Numpy.cov()* [47] is used to compute the covariance matrix of a cluster. As for the GMM case, the matrix obtained is of the form:

$$Cov = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \tag{7.34}$$

  From that matrix we can build an ellipse with axes length:

$$\begin{aligned} d_x &= \sqrt{3}a \\ d_y &= \sqrt{3}d \end{aligned} \tag{7.35}$$

  And the parameters b and c give us information about the tilted angle of the ellipse.

- 'diagonal' - also in this case the function *Numpy.cov()* is used. The matrix obtained is:

$$Cov = \begin{bmatrix} a & 0 \\ 0 & d \end{bmatrix} \tag{7.36}$$

  Therefore, Equation 7.35 can be used to compute the length of each axis; the ellipse is not tilted.

- 'spherical' - the cluster is modeled with a circle and the radius is computed as the maximum distance between the center of the cluster (the centroid) and each of the cluster points:

$$r = max(\|centroid - cluster\_points\|) \tag{7.37}$$

The strategy of the algorithm implemented for the clustering and the modelling of the obstacle is straightforward:

1. Define the maximum number of obstacle that can be detected; it corresponds to K number of clusters for the k-means. This number affects the speed of the algorithm and the performance.

2. Define n_init and max_iter.

3. Generate the object for the k-means modelling.

4. Fit the data point of the lidar with the k-means.
5. Fuse the centroids that are closer than a given distance d_{fuse} using Algorithm 11.
6. Change the labels of the obstacles to be consistent with the previous time step using Algorithm 12.
7. Model the obstacle as ellipses or circles.

A particular focus is needed for the 6^{th} step of the algorithm. Firstly, we need to specify that the obstacles considered are always K; if the algorithm that fuses the centroids produces $N_{cen} < K$ centroids, the $K - N_{cen}$ centroids are set as $[\inf, \inf]$. Together with the fused centroids, a vector called *link_labels* is produced that containts the information that links the un-fused centroids and the respective cluster labels with the fused centroids. To be more concrete: if the k-means produces 3 centroids, all the lidar points will be labeled with a number $l \in$ [1,2,3]; if $l = 1$, the data point is associated to centroid 1, and so on. After the algorithm that fuse the centroids, the new centroids could be just 2, hence, the vector *link_labels* is needed to guarantee that all the labels of the un-fused centroids are used in a correct way; for instance, if the original centroids 2 and 3 are merged into a single centroid, the *link_labels* will be [1,2,2]. In this way $l = 1 \rightarrow l = 1$; $l = 2 \rightarrow l = 2$ and $l = 3 \rightarrow l = 2$.

The algorithm's aim is to produce a set of obstacles that is consistent at each time step. Therefore, from a time step to the next one, we would like to have obstacles labeled in the same way, if they correspond to the same obstacle in reality. The k-means and the fusion algorithm do not guarantee that feature; hence, Algorithm 12 is introduced. Assuming again to have $K = 3$ and also $N_{cen} = K = 3$; at time step i the centroids obtained with k-means, called *old_vect*, could be:

$$\begin{bmatrix} 1 & 0 \\ 3 & 3 \\ -5 & 4 \end{bmatrix}$$

While at time step $i + 1$ the centroids *new_vect* could be:

$$\begin{bmatrix} 1.1 & 0 \\ -5.2 & 3.9 \\ 2.7 & 3.1 \end{bmatrix}$$

Algorithm 12 transforms *new_vect* into:

$$\begin{bmatrix} 1.1 & 0 \\ 2.7 & 3.1 \\ -5.2 & 3.9 \end{bmatrix}$$

This function uses a distance metrics to change the position (the index) of the current centroids, in order to be in accord with the previous ones. Of course, it has also to modify the *link_labels* vector in order to take into account the changes of the indexes.

Algorithm 11 Fuse algorithm $fuse()$

procedure FUSE($centroids, d_{fuse}$)
    ▷ $centroids$ are the centroids of the K-means model
    ▷ d_{fuse} is the distance that the algorithm uses to fuse the points
    ▷ $ret\_centroids$ is a vector that contains the fused centroids
    $ret\_centroids \leftarrow [\ ]$
    ▷ $label$ is the vector that contains the new labels
    $labels \leftarrow [1, ...,1]$
    ▷ lab is a value that identifies the label number
    $lab = 1$
    ▷ $taken$ contains information about the centroids that have been merged
    $taken \leftarrow [False, ..., False]$
    for i = 0:length(means) **do**
        if taken[i] == False **then**
            $count = 1$
            $point \leftarrow means[i]$
            $cov_i \leftarrow cov[i]$
            $taken[i] \leftarrow True$
            $label[i] \leftarrow lab$
            for j = i+1:length(means) **do**
                if $\|centroids[i] - centroids[j]\| < d_{fuse}$ **then**
                $point = point + centroids[j]$
                $count = count + 1$
                $taken[j] \leftarrow True$
                $label[j] \leftarrow lab$
                end if
            end for
            $lab = lab + 1$
            $point = \frac{point}{count}$
            $ret\_means \leftarrow append(point)$
        end if
    end for
    ▷ **Return** ret_means, $labels$
end procedure

Algorithm 12 Changing labels algorithm $change_labels()$

procedure CHANGE_LABELS($new\_vect, old\_vect, labels, n$)
    ▷ new_vect are the centroids of the K-means model at current time step
    ▷ old_vect are the centroids of the K-means model at previous time step
    ▷ $label$ is the vector that contains the labels at current time step. The labels identify the correspondence between the fused centroids and the un-fused ones
    ▷ n is the number of maximum obstacles allowed
    $l_a = length(new\_vect)$ ▷ Length of the new vector
    $l_b = length(old\_vect)$ ▷ Length of the old vector
    ▷ new_vect and old_vect must have length n
    $new\_vect \leftarrow append(Inf \cdot ones(n - l_a,2))$
    $old\_vect \leftarrow append(Inf \cdot ones(n - l_b,2))$
    ▷ Compute the norm of each point in new_vect with each point of old_vect
    $norms \leftarrow \|new\_vect, old\_vect\|$
    ▷ Associate the to each element of new_vect the nearest of old_vect
    ▷ Change the position of centroids in new_vect so that:
    for i = 0:n **do**
        $new\_vect[i]$ is nearest to $old\_vect[i]$
    end for
    ▷ Change the labels in $labels$ so that they can still associate the un-fused centroids to the fused ones
    ▷ **Return** new_vect, $labels$
end procedure

In order to test the algorithm, the data set with few people was used; moreover, we set $max\_view\_rad = analysis\_rad = 6m$, $n\_clusters = K = 10$, $max\_iter = 20$, $n\_init = 3$ and the cluster is modeled using the 'spherical' method.

This means that we are always considering 10 obstacles, but some of them can be set in position $[\text{inf}, \text{inf}]$, therefore they are not considered.

Figure 7.17 shows that the computation time of the algorithm respects the limits imposed by the real system, therefore it could be used in real-time.

Figure 7.18 illustrates four consequent time instants. The algorithm is able to identify precisely the position of the obstacles and it is also able to use the same label to reference it.

Until now this is the best result obtained. Analysing the total simulation, it never happens that some modelled obstacles are bigger than the real ones or that a modelled obstacle covers a large area on the map.

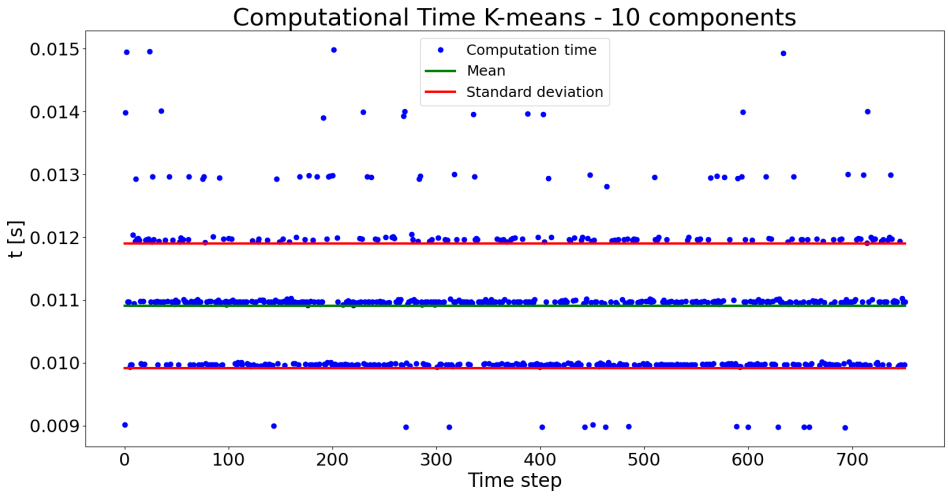


Figure 7.17: K-means - Test - Time

Having reliable information on the centroids, it is possible to compute the velocity vector of each obstacle. The procedure is straightforward: being $c_i(t)$ the $i-th$ centroid at instant t and $c_i(t-1)$ the same centroid at the precious time step, the velocity vector is computed as in Equation 7.38, with dt time step.

$$\dot{c}_i(t) = \frac{c_i(t) - c_i(t-1)}{dt} \tag{7.38}$$

The velocity computed in this way is subject to a lot of noise caused by the fact that the centroids are computed using lidar data, which present some noise them self, and because the k-means do not take trace of the previous computation.

In order to smoothen the behaviour of the velocites vectors, a temporal filter is introduced. Choosing T_{filter} as the number of time steps to filter, we can memorize the previous T_{filter} velocity vectors and compute the velocity at the current step doing a weighted sum:

$$\dot{c}_i(t) = \sum_{k=0}^{T_{filter}} w(T_{filter} - k) \cdot \dot{c}_i(t-k) \tag{7.39}$$

The weights w can be chosen in different ways, but they have to respect the constraints:

$$\sum_{k=0}^{T_{filter}} w(k) = 1 \tag{7.40}$$

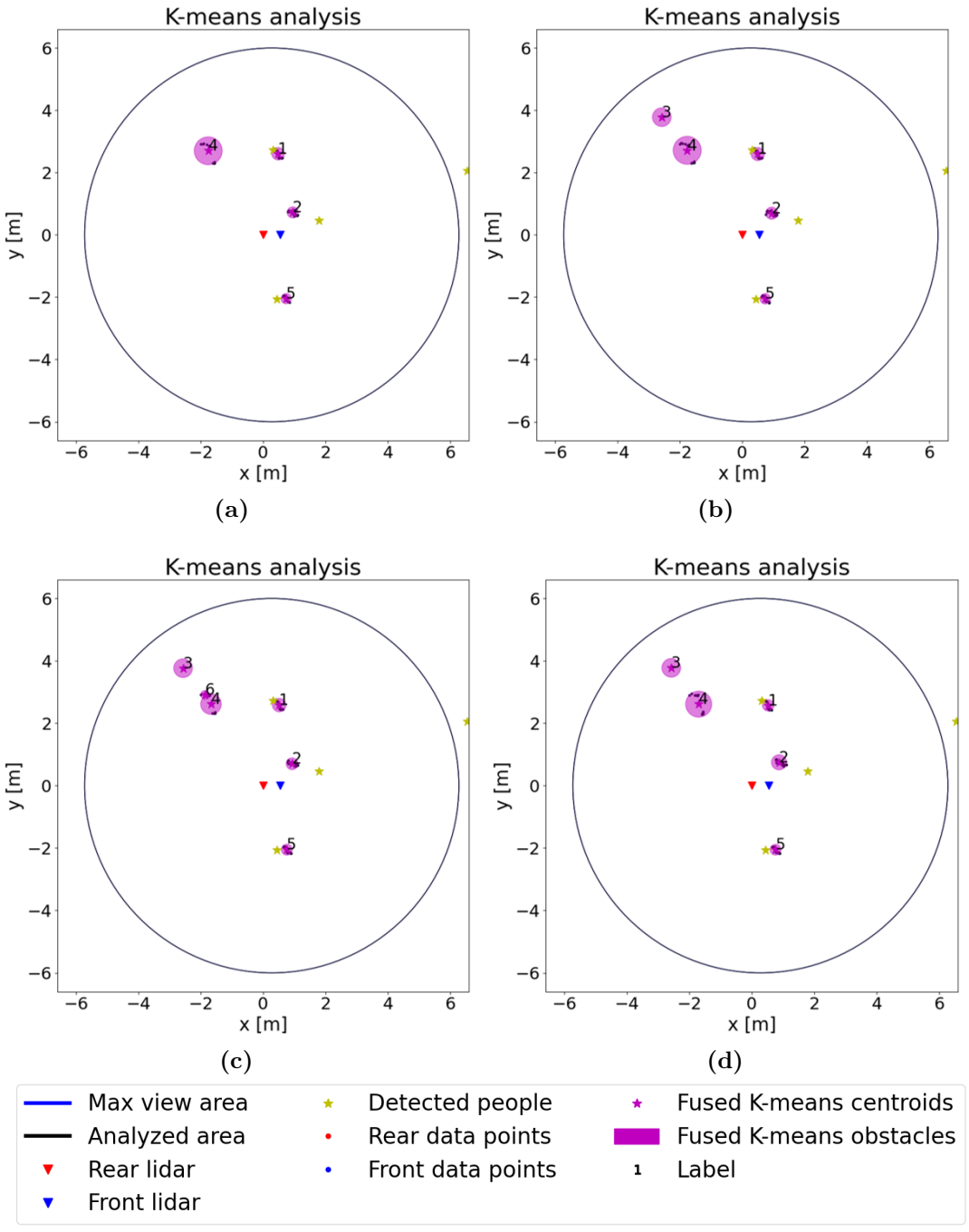


Figure 7.18: K-means - Test

In this case, a linear filter has been chosen:

$$\begin{aligned} w &= [0, \ \dots \ ,1] \\ w &= \frac{w}{\sum w} \end{aligned} \tag{7.41}$$

Repeating the simulation and applying also the velocity filter, the results shown in Figures 7.19 and 7.20 are obtained. The first one describes the computation time of the k-means algorithm, combined with the fusion algorithm that merges the centroids; with the algorithm that changes the lables in order to track in a smart way the obstacles and with the algorithm that computes and filter the velocity of each obstacle. With this results, the algorithm respects the constraints imposed by the robot's hardware, namely a computation time smaller than $0.05s$.

Figure 7.20 illustrates six consequent time steps. Again, the algorithm works properly and it is able to define the obstacle and track them during the simulation. Moreover, the velocity vectors are smooth and describe finely the direction of movement of the obstacles.

Having all these information, it is possible to use the algorithm in combination with the High Level Controller 5 and the Low Level Controller 6.

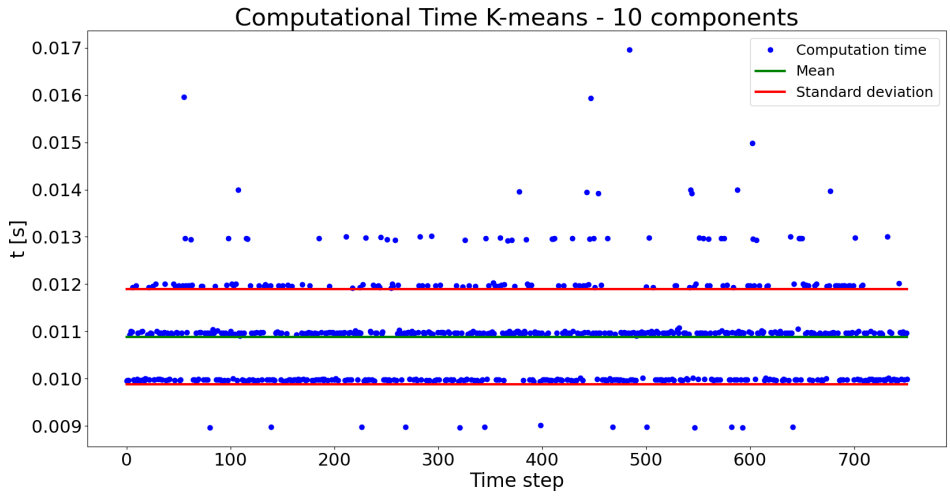


Figure 7.19: K-means - Test with velocity - Time

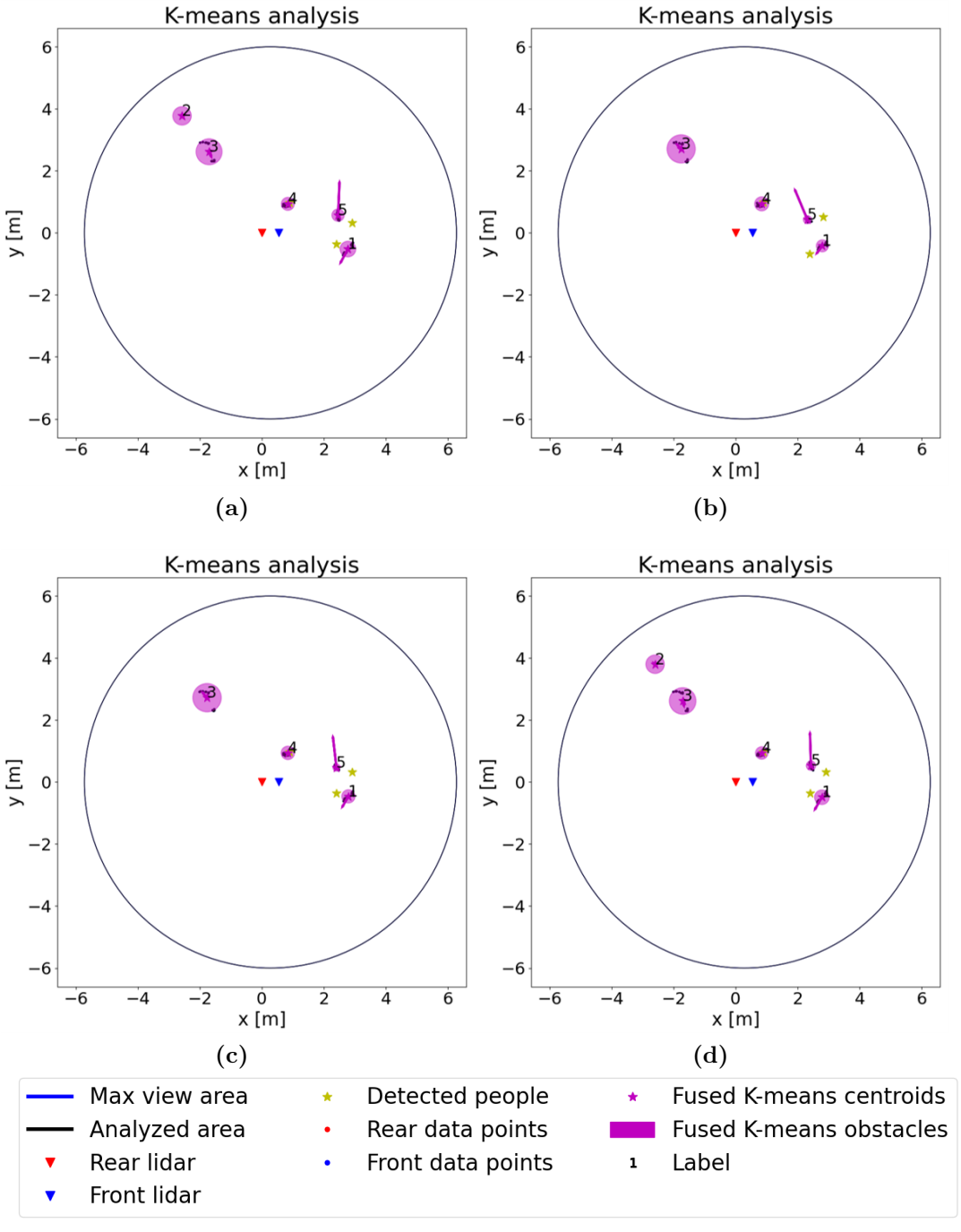


Figure 7.20: K-means - Test with velocity

Chapter 8

High Level Control, Low Level Control and Detection Algorithm integration

The previous chapters focused on the different algorithms that compose the control structure that this thesis aims to analyse. In particular, a study regarding the dynamical systems based control and the modulation algorithm has been conducted in order to understand how it works and which are the main advantages and the main issues of this method.

Then, several low level controllers have been developed, in order to take into account the non-holonomic constraints of the robot. Many of these methods can be used by them self to plan a path and produce the control inputs for the system; nevertheless, the purpose of the low level controller is to work using the information obtained from the high level controller, i.e. the modulating algorithm. Using the low level controllers together with the high level controller has often shown poor results, especially if the robot's constraints on the speed and accelerations were taken into account. Fortunately, some of the control algorithm proposed are able to produce suitable control inputs, starting from the information given by the modulating algorithm. The principal low level controller that has shown good result is the non-linear Model Predictive Control with Velocity control, but also the Model Predictive Control used together with the Feedback Linearization can be used, if some precautions are used.

Finally, an algorithm for the obstacle detection has been developed. It exploits Machine Learning techniques and the raw data from the LIDAR sensors. The first method implemented, i.e. the Gaussian Mixture Model, was full of expectations, but the results obtained with this methods are quite noisy and imprecise. Then, a K-means based algorithm has been implemented, showing that it is able to detect

in a very precise way the clusters and to track them. The possibility to monitor the evolution of the obstacles allow us also to compute their velocity, which means that we have more information to rely on.

In this Chapter, all the three algorithm will be used together. In Chapter 6 we have already analyzed the high level controller in combination with the low level one, but the knowledge of the obstacles was total: the obstacles had a defined shape and their velocities were always known. Here, we will use the information obtained with the obstacle detection algorithm and several simulations will be implemented in order to verify the benefits and the issues of the three algorithms together.

8.1 Simulated Target

The LIDAR data points used for the tests are obtained from a recording session in which the robot is not moving and three people move around it. The center of the global frame is considered to be the rear LIDAR of the robot. The control algorithm is tested on a simulated robot that has to move in the environment built from the LIDAR data points. In this section, also the target is simulated, therefore, another obstacle, that we totally know, is introduced, in addition to the obstacles detected by the Obstacle Detection algorithm.

A clarification is needed before starting the tests: the modulating algorithm implemented in Python is the one developed by Dr. Lukas Huber [7] with some modifications in order to take into account the moving target. That algorithm exploits many optimization methods and it is able to merge together the obstacles as said in Section 5.8; moreover, it is able to automatically update the reference points inside the obstacles in order to guarantee the conditions for the convergence and the impenetrability of the border. In particular, the function *update_reference_points()* deals with the problem of merging the obstacles. If the obstacles are few, the function works properly, while, if the obstacles are quite a lot (for instance, 10 obstacles), the function requires to much time to compute the border of the merged obstacles and the common reference point. This represents a huge bottleneck: this function is very important, especially in the situations that this project wants to deal with. The detected obstacles are enlarged and they might collide, without this functions, discontinuities in the DS might arise, causing wrong behaviours. Nevertheless, in many of the tests that will be conducted, this function will not be used because of the timing constraints: the control algorithm has to run in real-time and the above mentioned function slows down the computation so much that a real-time implementation is not possible.

In addition, another important problem that has emerged in the tests concerns the impossibility to update the velocities of the obstacles. This information is used by the modulation algorithm to take into account the movement of the robots and act accordingly. Without this knowledge, the performance of the modulation algorithm is therefore reduced. Consequently, the information about the obstacles' speed computed using the algorithm described in Section 7.3 are not used.

For all the tests, unless differently stated, the parameters and the features used are:

- Control frequency $f_{DS} = 20Hz$;
- Time step $dt = 0.05$;
- Number of obstacles $K = 10$;
- Additional obstacle to take into account the target for the modulation;
- Maximum number of iterations $max\_iter = 20$;
- Number of runs of the K-means $n\_init = 3$;
- Fusion distance $d_{fuse} = 0.7m$;
- Obstacle typology 'spherical';
- Filter time for the obstacles' velocities $T_{filter} = 15$;
- Low level controller typology: non-linear MPC with velocity control;
- Horizon time $N = 7$;
- Robot radius $r_{robot} = 0.3m$
- Safety margin $\epsilon_{safe} = 0.2m$

Test 1

The aim of this first test is to prove that the computation time is unusable on the robot if the function *update_reference_points()* is active. The time needed to process that function is included in the computation time of the K-means, since it is used to model the obstacles.

As said above, the target is simulated and acts as the $K+1$ obstacle. Its shape is circular with radius $r_{target} = 0.3m$, that gets enlarged using the Equation:

$$r_{target} = r_{target} + r_{robot} + \epsilon_{safe} \tag{8.1}$$

The target starts from position $\xi_{tar} = [-6., -4.]^T$ and its dynamics is described by the Equations 8.2 and 8.3.

$$\dot{\xi}_{tar} = \begin{bmatrix} 0.3 + 0.2 * cos(0.3 * t) \\ 0.3 + 0.2 * sin(0.3 * t) \end{bmatrix} \tag{8.2}$$

If $\left\|\dot{\xi}_{ref}\right\| > max_{ds}$, with $max_{ds} = 0.5m/s$

$$\dot{\xi}_{ref} = max_{ds} \cdot \frac{\dot{\xi}_{ref}}{\left\|\dot{\xi}_{ref}\right\|} \tag{8.3}$$

The attractor ξ_a is computed using the methodology described in Section 5.10 and the constants used in Equation 5.33 are

- $k_{attr} = 3$
- $k_{rep} = 6$

All the modelled obstacles have a circular shape with radius given by Equation 7.37 here reported

$$r^O = max(\|centroid - cluster\_points\|)$$

That are then enlarged using Equation:

$$r^O = r^O \cdot 1.1 + r_{robot} + \epsilon_{safe} \tag{8.4}$$

The radius computed with the K-means is multiplied by 1.1 in order to take into account the errors caused by the LIDARs' noise and avoid modelling a smaller obstacle than it really is.

The initial robot pose is $\xi = [-5., -4., \pi/2]^T$ and the gain matrices used for the non-linear Model Predictive Control with Velocity control are:

$$G_\theta = q_\theta^2 \qquad G_{Vx} = q_{Vx}^2 \qquad G_{Vy} = q_{Vy}^2 \qquad G_u = R \cdot R \tag{8.5}$$

$$q_\theta = 1 \qquad q_{Vx} = 10 \qquad q_{Vy} = 10 \qquad R = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \tag{8.6}$$

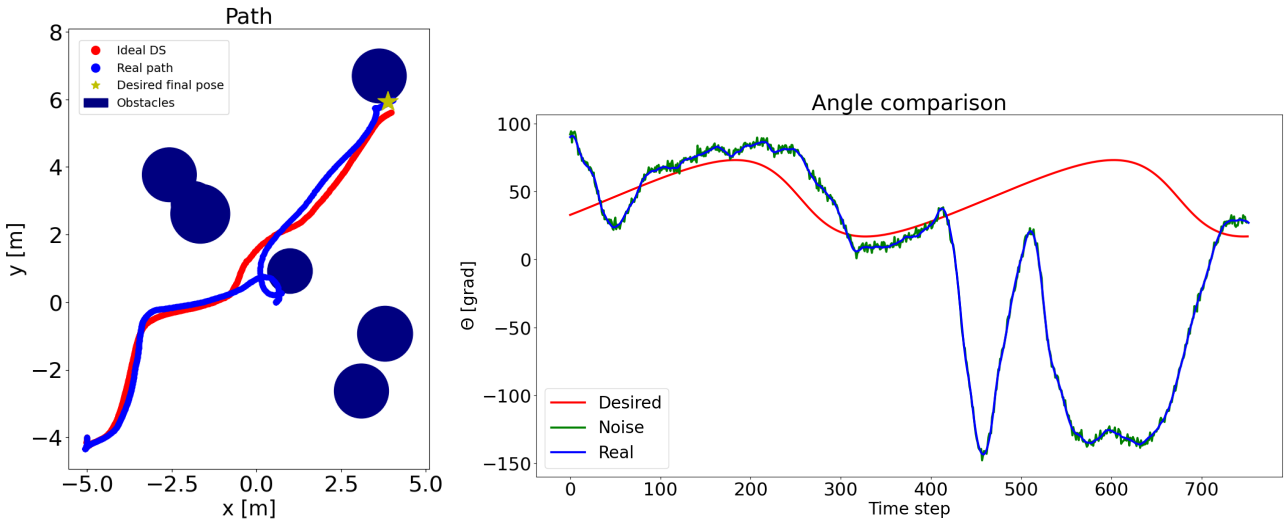
Using such matrices suggests that the most important thing for the robot to do is following the velocity vector. As a matter of fact $q_{Vx} = q_{Vy} > q_\theta$, that implies that the nMPC controller gives more importance to the velocity reference, rather then to the heading angle reference.

In Section 6.5 we have seen how to produce the heading angle reference for the robot: we can choose between the angle of the desired velocity vector, the angle of the target's velocity or a combination of the two using the Sigmoid function. Ideally, choosing a q_θ smaller than the others gains implies that the robot should try to follow the reference, but with not too effort. Consequently, this could be useful if used with the second typology of heading angle reference, i.e. the the angle of the target's velocity, because the robot aims to keep the same heading angle of the target, but if it is needed, it can differ in order to avoid an obstacle. Therefore, the heading angle reference is given by the target velocity, which is totally know at every instant.

The results of the simulation are shown in Figures 8.1, 8.2, 8.3 and 8.4. The first one illustrates the path covered by the robot and the angle compared with the desired one. The first thing that emerges is that for almost a half of the simulation, the heading angle of the robot is way different with respect to the reference. This is not necessarily an error, because this might occur during the obstacle avoidance. In this case though, the behaviour of the robot is completely wrong, as shown in Figure 8.4. The four images display different time instants at 0.75 seconds one from the other; in Figure 8.4(a) the robot is following the target and an obstacles is in front of the robot. After 0.75 seconds the robot collide with the obstacle as shown in Figure 8.4(b). Due to the repulsive force inside the area of the enlarged obstacle, the robot is pushed away (Figure 8.4(c)) , but its heading angle is the opposite of the desired one; moreover, from that position it would be difficult for the robot to see the target. In conclusion, the robot then starts moving backwards to reach the attractor.

The behaviour above illustrated can not be used in the real implementation, even if the control inputs constraints are observed, as displayed in Figure 8.3.

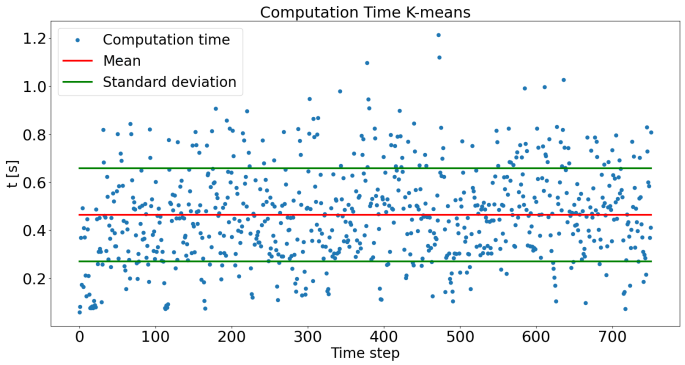
Another aspect that is worth to mention concerns the computation time of the K-means algorithm and of the nMPC optimization. Figure 8.2 illustrates both of them: the time required to optimize the control inputs and generate a feasible control action respects the limit of $0.05s$. On the contrary, the computation time for the K-means, which includes also the time necessary to run the function *update_reference_points()*, is in the order of tenths of a second: completely unusable for a real time application.



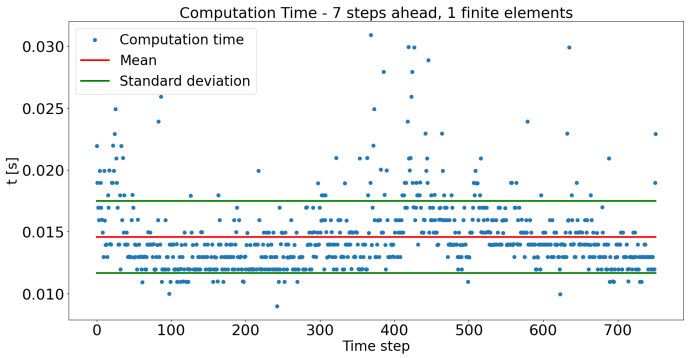
(a) Path

(b) Angle comparison

Figure 8.1: Simulated Target - Test 1

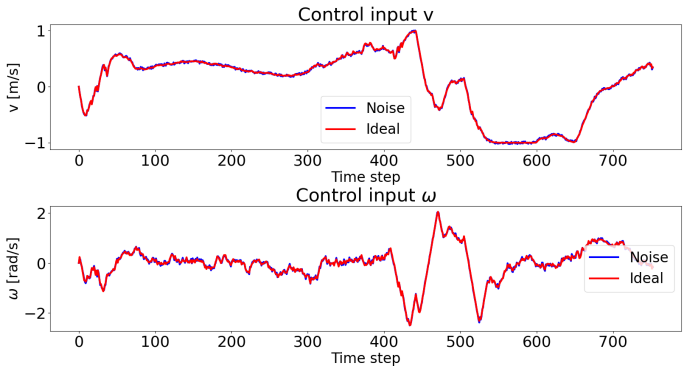


(a) Computation time K-means

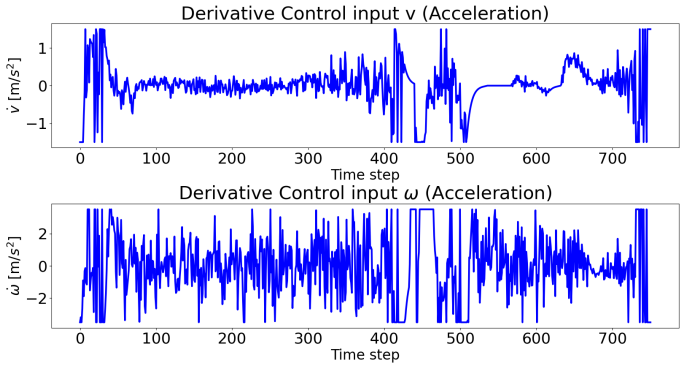


(b) Computation time nMPC

Figure 8.2: Simulated Target - Test 1 - Computation Time



(a) Control inputs



(b) Control inputs derivative

Figure 8.3: Simulated Target - Test 1 - Control Inputs

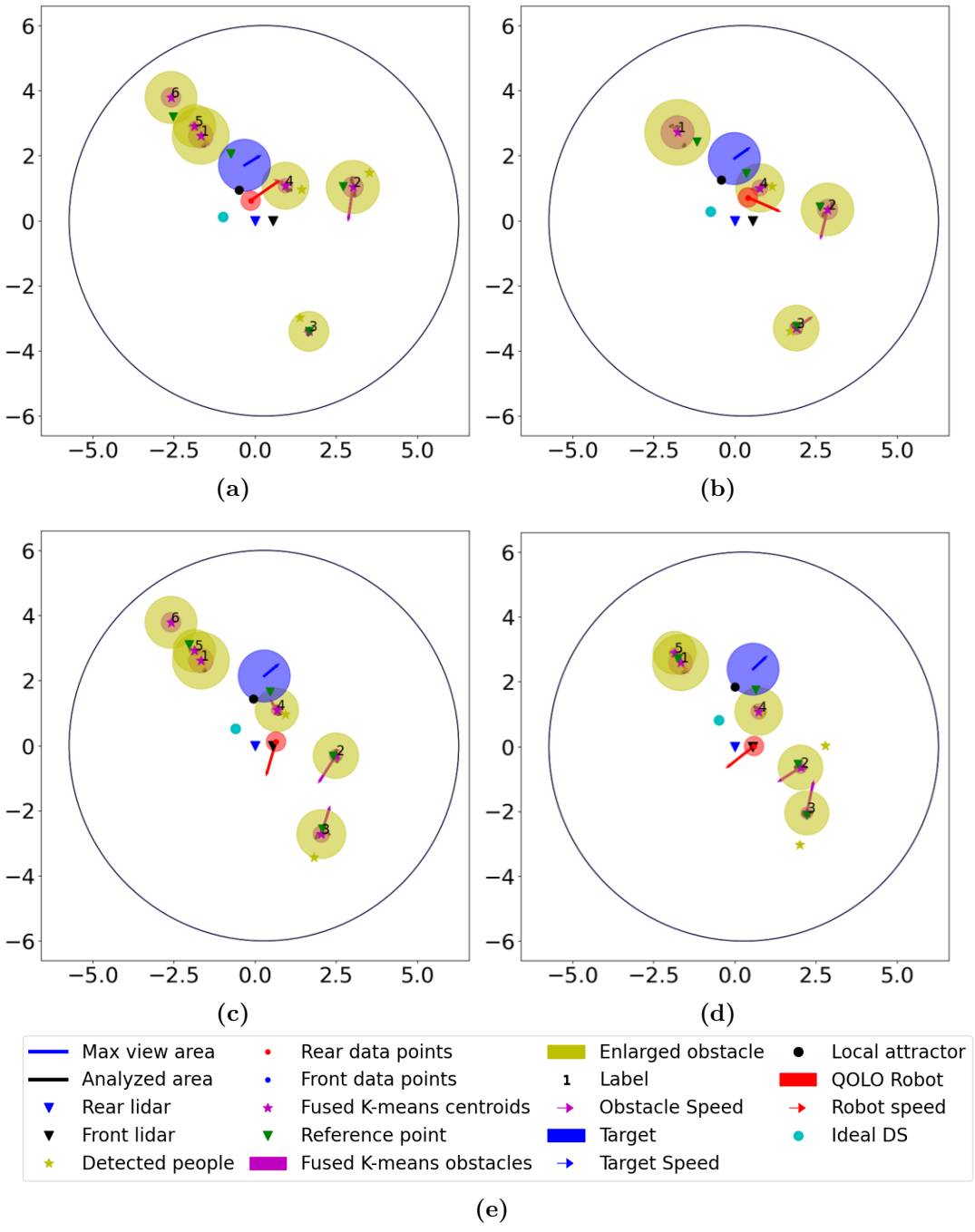


Figure 8.4: Simulated Target - Test 1 - Simulation

Test 2

In this second test, the same situation of Test 1 is presented, but in this case the function *update_reference_points()* is not used. This means that at every time instant, the reference point inside the obstacle coincide with its center.

The initial target position is again $\xi_{tar} = [-6., -4.]^T$, and its velocity vector $\dot{\xi}_{tar}$ is computed using Equations 8.2 and 8.3. The shape of the obstacle is the same as in Test 1

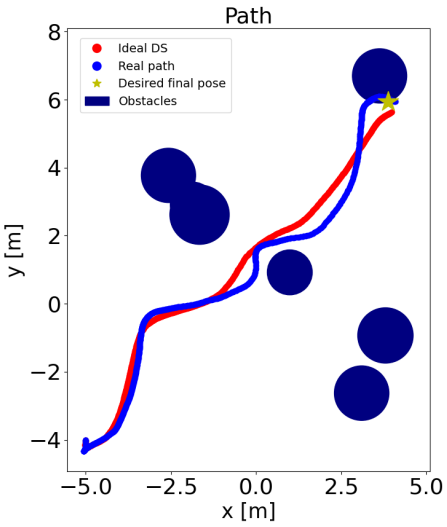
The initial robot pose is $\xi = [-5., -4., \pi/2]^T$ and the gain matrices used for the non-linear Model Predictive Control with Velocity control are:

$$G_\theta = q_\theta^2 \qquad G_{Vx} = q_{Vx}^2 \qquad G_{Vy} = q_{Vy}^2 \qquad G_u = R \cdot R \tag{8.7}$$

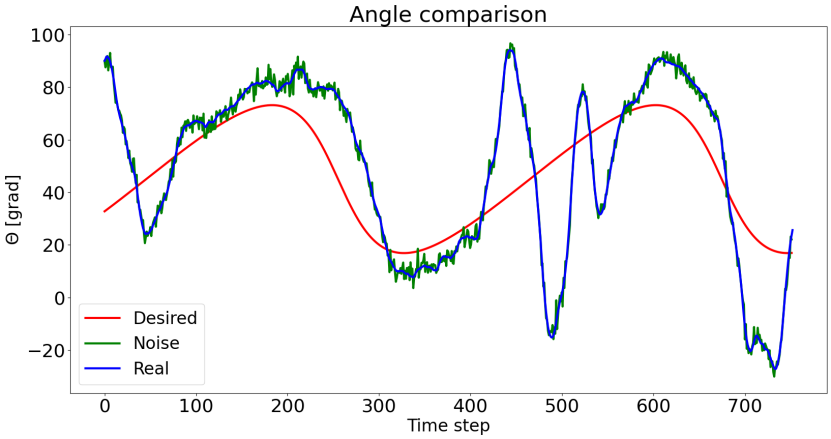
$$q_\theta = 1 \qquad q_{Vx} = 10 \qquad q_{Vy} = 10 \qquad R = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \tag{8.8}$$

In this case, Figure 8.5 displays a different behavior with respect to the one obtained from Test 1. The heading angle in this case is more similar to the desired one ad this reflects also in the simulation shown in Figure 8.8. Also in this case the images describe four instants spaced in time of 0.75 seconds. Differently from the previous case, the robot is able to avoid the obstacles and track correctly the target. Also the timing constraints are respected, as illustrated in figure 8.6: both the K-means algorithm and the non-linear MPC requires less than 0.05 seconds to run.

In conclusion, also the constraints on the control inputs are matched, as expected thank to the non-linear MPC.

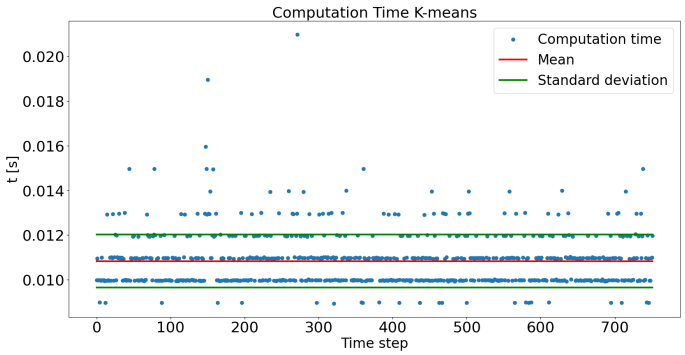


(a) Path

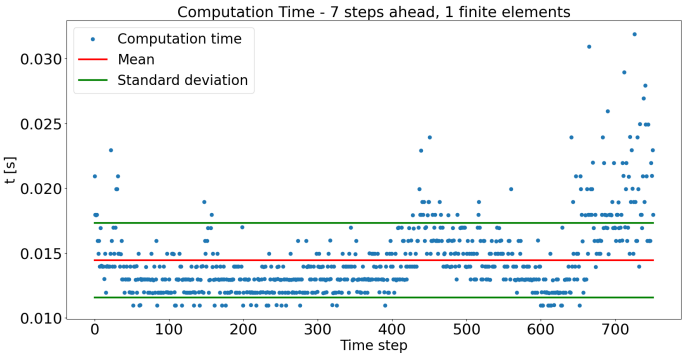


(b) Angle comparison

Figure 8.5: Simulated Target - Test 2

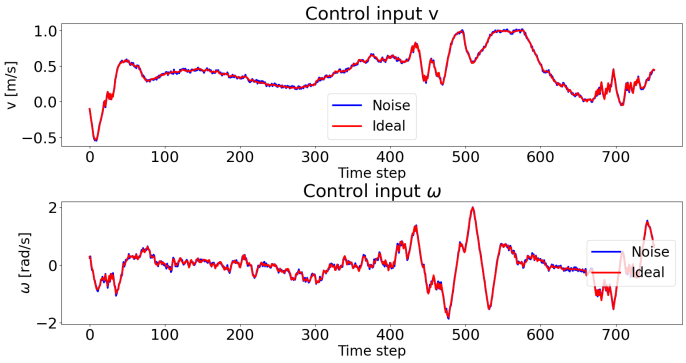


(a) Computation time K-means

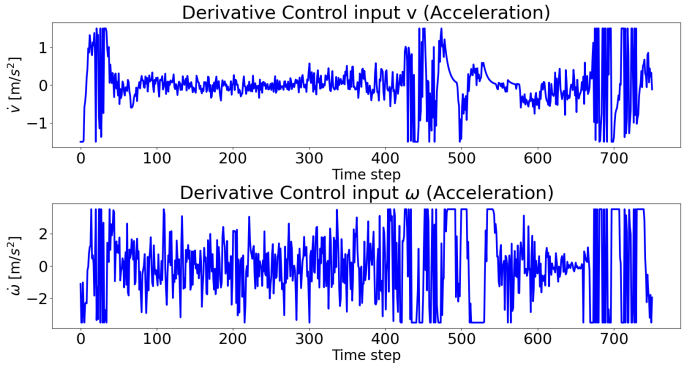


(b) Computation time nMPC

Figure 8.6: Simulated Target - Test 2 - Computation Time



(a) Control inputs



(b) Control inputs derivative

Figure 8.7: Simulated Target - Test 2 - Control Inputs

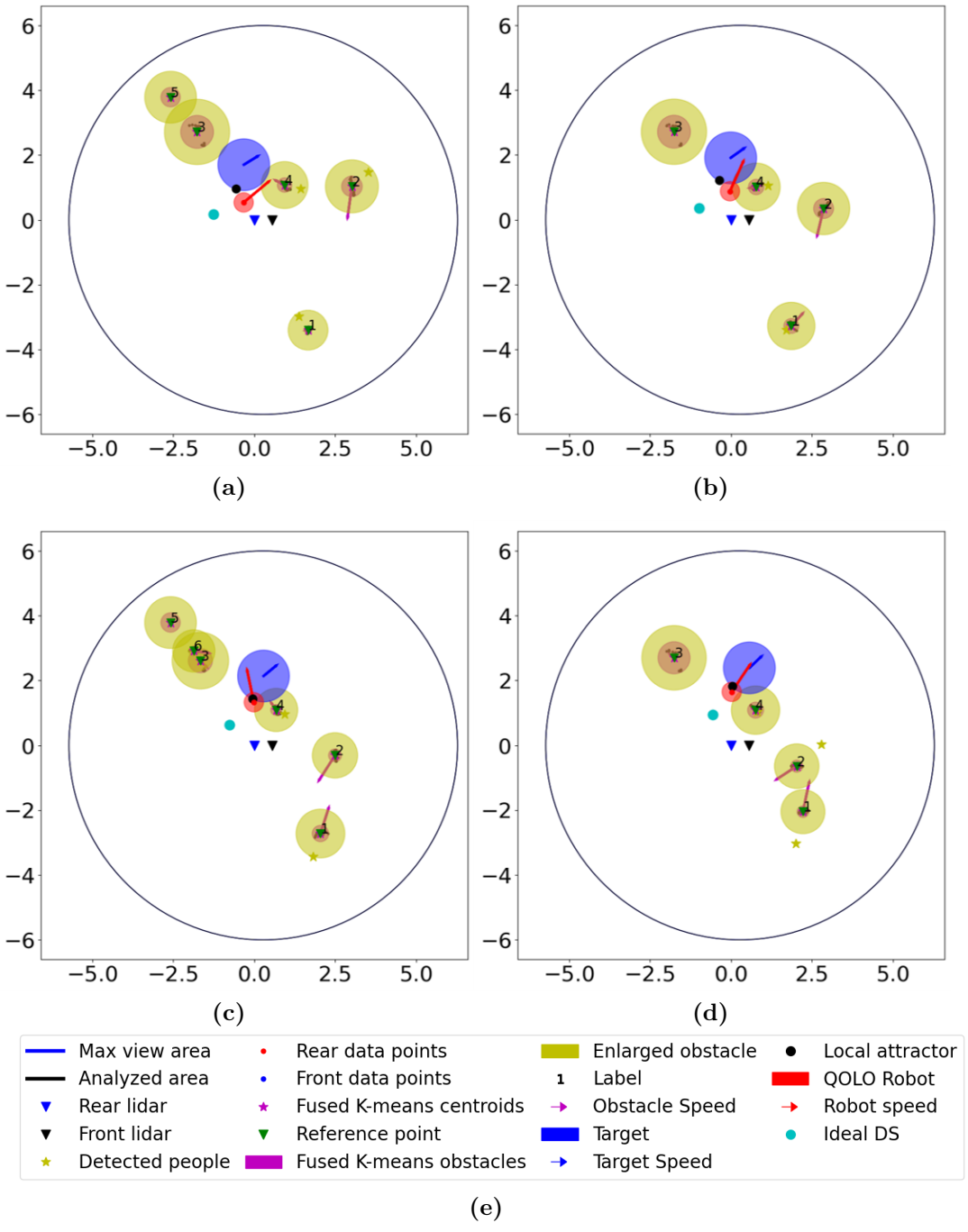


Figure 8.8: Simulated Target - Test 2 - Simulation

Test 3

From the Test 2 it appears that using the gain matrices

$$G_\theta = q_\theta^2 \qquad G_{Vx} = q_{Vx}^2 \qquad G_{Vy} = q_{Vy}^2 \qquad G_u = R \cdot R \tag{8.9}$$

$$q_\theta = 1 \qquad q_{Vx} = 10 \qquad q_{Vy} = 10 \qquad R = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \tag{8.10}$$

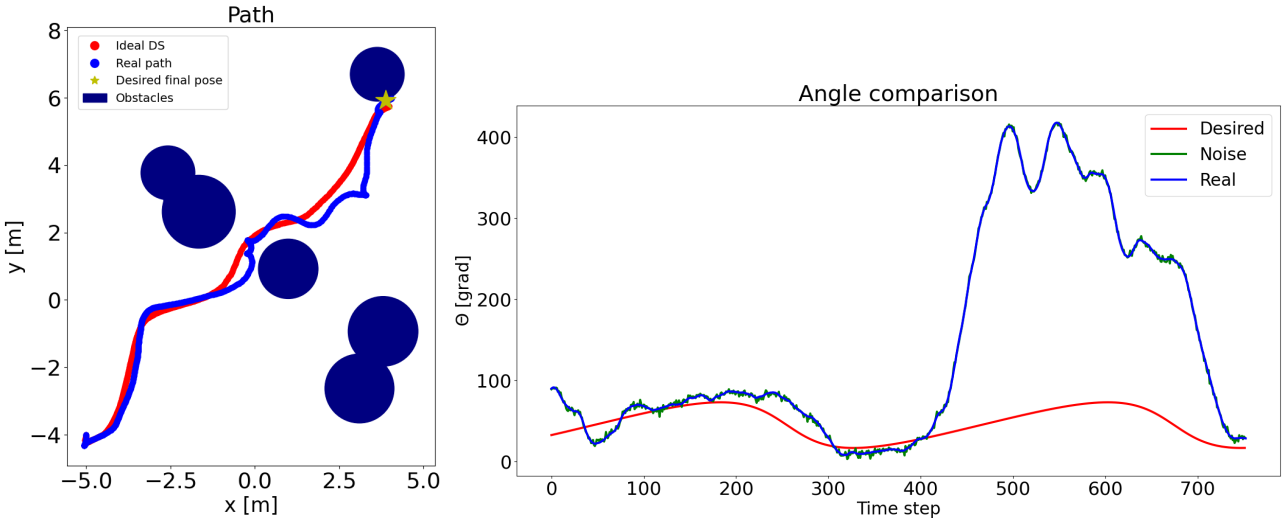
in combination with the DS modulation, without the function *update_reference_points()* that merges together the obstacles, produces very good results. Therefore, it suggests that using a gain q_θ smaller than q_{Vx} and q_{Vy} is a good idea. In order to prove this hypothesis, another test is conducted.

The scenario is the same as in the previous two tests, but in this case the radii of the obstacles are computed in a different way. The choice of this metrics is totally arbitrary and it is described by the Equations:

$$r^O = max(\|centroid - cluster\_points\|)$$

$$r^O = \sqrt{r^O} + r_{robot} + \epsilon_{safe} \tag{8.11}$$

Setting the initial target position as $\xi_{tar} = [-6., -4.]^T$ and the initial robot pose as $\xi = [-5., -4., \pi/2]^T$, and using Equations 8.2 and 8.3 for the dynamics of the target, the results are illustrated in Figures 8.9, 8.10, 8.11 and 8.12.



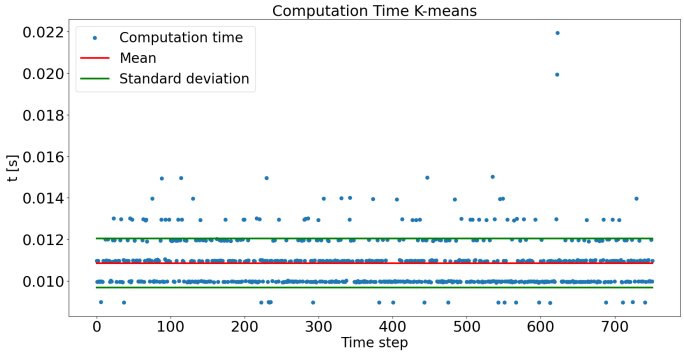
(a) Path **(b)** Angle comparison

Figure 8.9: Simulated Target - Test 3

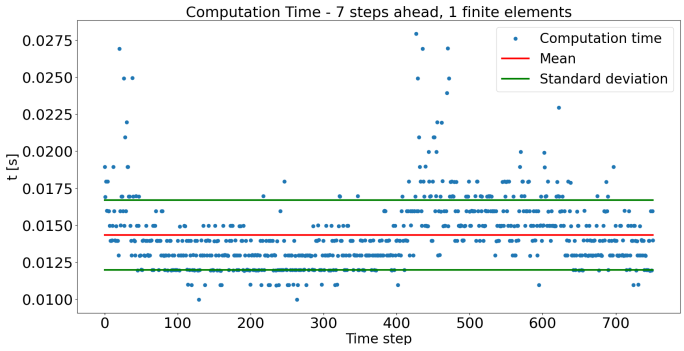
As in Test 1, the robot goes on the enlarged obstacle. This situation should be avoided for two reasons:

1. The robot inside the obstacle represents a potential dangerous situation, due to the possibility of collisions;

2. The modulated DS is not continuous on the border of the obstacle, which means that the desired velocity changes suddenly. The non-linear MPC can force the robot to respect the constraints on the velocity and of the accelerations, but the overall behavior can be compromised.



(a) Computation time K-means



(b) Computation time nMPC

Figure 8.10: Simulated Target - Test 3 - Computation Time

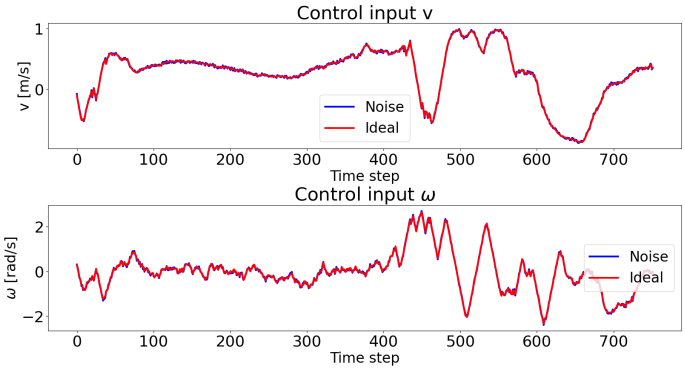
Nevertheless, the safety margin ϵ_{safe} is introduced precisely for this reason: if the robot crosses the boundary of the enlarged obstacle, there is sufficiently space to avoid a collision. The four instants in Figure 8.12 show that the repulsive force produced by the obstacles forces the robot to rotate abruptly (respecting the constraints). Having a small q_θ allows the robot to rotate despite the desired heading angle, but this produces as drawback the behaviour described in Figure:

the robot prefers to move backwards rather than rotate, because it is easier for it following the desired velocity.

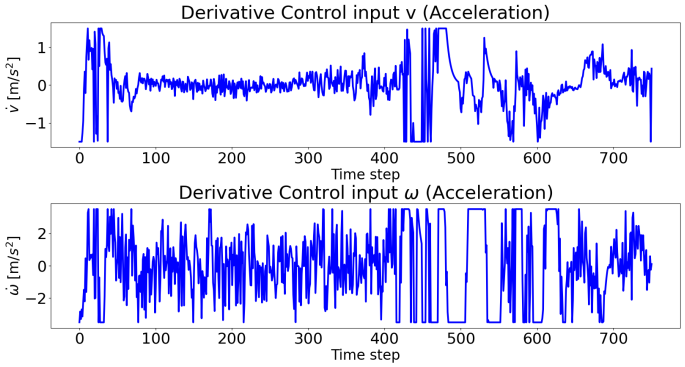
As a matter of fact, Figure 8.9 displays the differences between the real heading angle and the desired one, showing that the robot moves backwards until it reaches the attractor and then rotates.

Such a behavior has to be avoided because it is not comfortable at all for the users to move backward, especially if they are trying to follow a friend.

In spite of the results obtained, the single sub-algorithms that compose the control algorithm show again good results. The control inputs are able to respect the constraints even if the sudden change in the velocity direction caused by the discontinuity on the obstacles' border. Moreover, both the K-means algorithm and the non-linear MPC are fast enough to respect the timing contraints.



(a) Control inputs



(b) Control inputs derivative

Figure 8.11: Simulated Target - Test 3 - Control Inputs

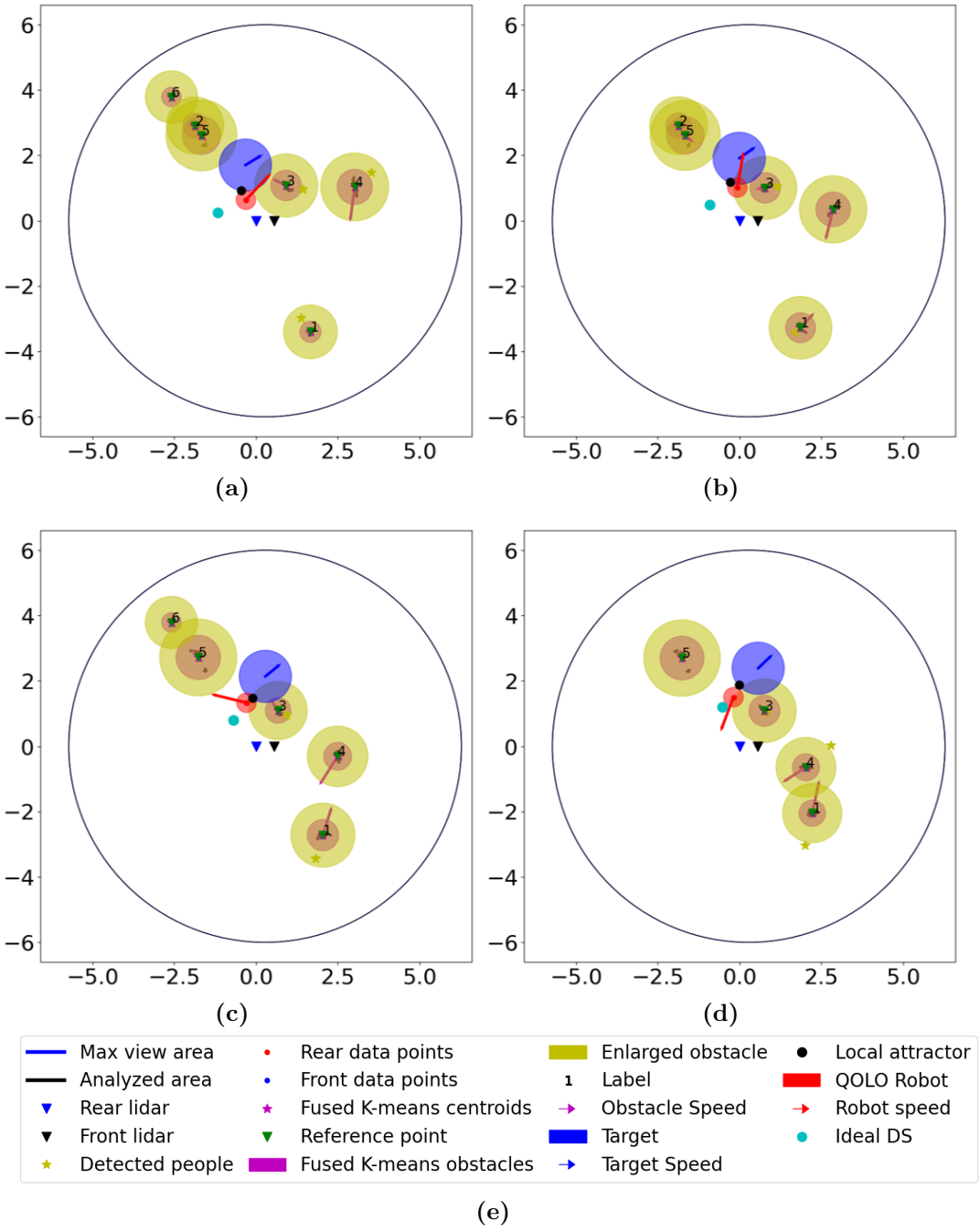


Figure 8.12: Simulated Target - Test 3 - Simulation

Test 4

From the previous tests it has emerged that using a value for q_θ quite smaller than q_{Vx} and q_{Vy} does not produce a good behaviour for the robot, because the heading angle can be very different from the desired one. Hence, this test implements different gain matrices:

$$G_\theta = q_\theta^2 \qquad G_{Vx} = q_{Vx}^2 \qquad G_{Vy} = q_{Vy}^2 \qquad G_u = R \cdot R \tag{8.12}$$

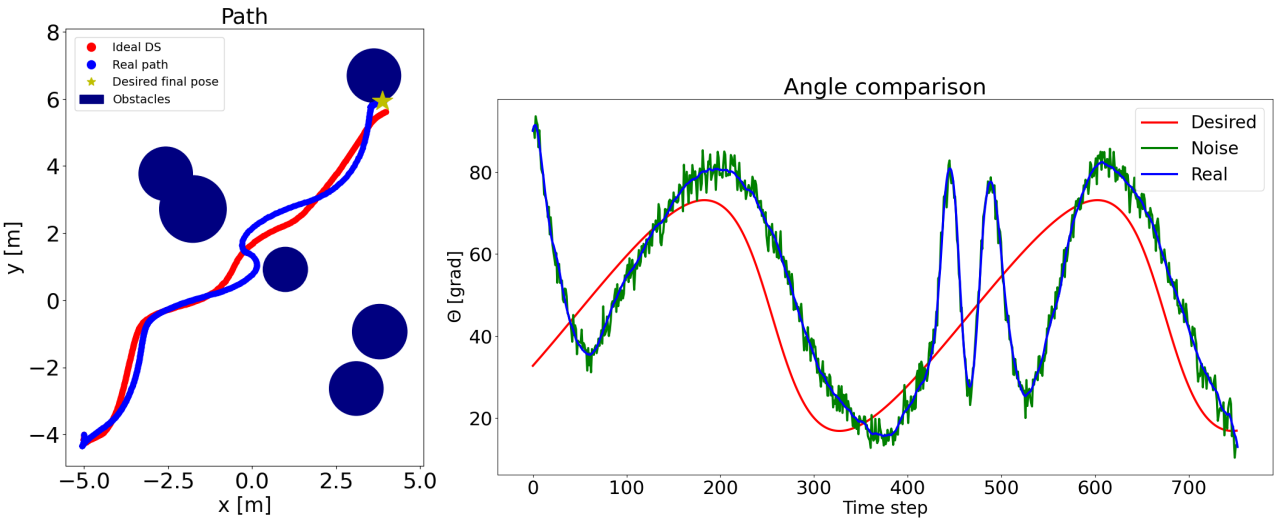
$$q_\theta = 4 \qquad q_{Vx} = 8 \qquad q_{Vy} = 8 \qquad R = \begin{bmatrix} 0.5 & 0 \\ 0 & 1.5 \end{bmatrix} \tag{8.13}$$

The shape of the obstacles is computed using the Equations:

$$r^O = max(\|centroid - cluster\_points\|)$$

$$r^O = r^O \cdot 1.1 + r_{robot} + \epsilon_{safe} \tag{8.14}$$

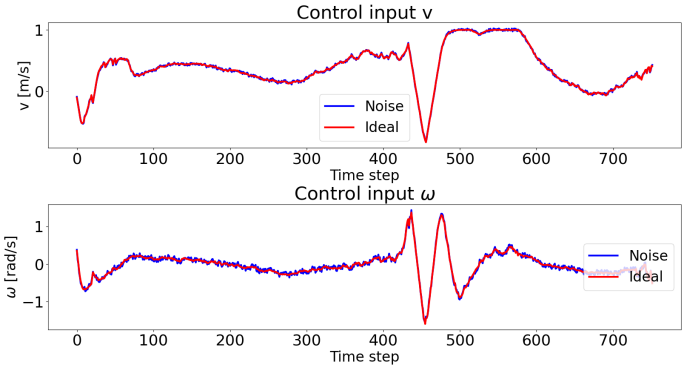
Equations 8.2 and 8.3 describe the dynamics of the target, that starts from position $\xi_{tar} = [-6., -4.]^T$. The initial pose of the robot is, as in the previous tests, $\xi = [-5., -4., \pi/2]^T$.



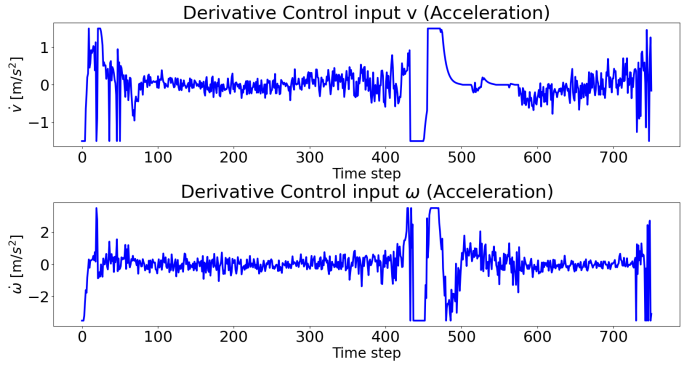
(a) Path **(b)** Angle comparison

Figure 8.13: Simulated Target - Test 4

Also in this simulation, the robot enters into one of the obstacles, as displayed in Figure 8.16. Differently from the previous examples, though, the control algorithm does not make the robot rotate suddenly. In this case, the heading angle tries to track the desired heading angle in a stronger way. Thanks to that, the robot brakes when it enters the obstacle, as displayed in Figure 8.14, where a steep deceleration can be noticed. Then it goes backward a bit and starts again to follow the target, while avoiding the obstacles. Moreover, always in Figure 8.14, the acceleration signals have a nicer shape with respect to the previous tests. This translate into a better experience for the users.



(a) Control inputs

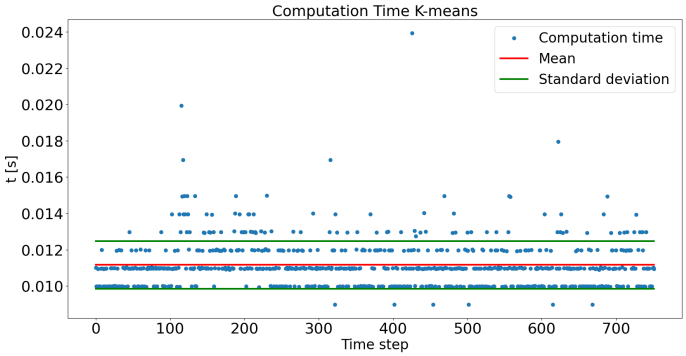


(b) Control inputs derivative

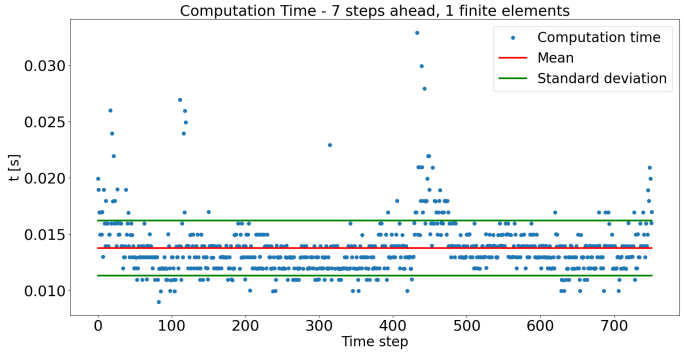
Figure 8.14: Simulated Target - Test 4 - Control Inputs

Also in this simulation, the timing constraints are respected as shown in Figure 8.16. An interesting aspect concerns the computation time of the non-linear MPC; it requires more time in correspondence of the collision with the obstacle. Of course, this is expected mainly because the non-linear MPC uses information of the past control actions to produce the control inputs; the previous control actions depend on the velocity reference, that are guaranteed to be continuous outside the

obstacles. When the robot enters in the area of the enlarged obstacle, the velocity reference is totally different from the previous one, therefore, the nMPC requires more time to optimize the control inputs.



(a) Computation time K-means



(b) Computation time nMPC

Figure 8.15: Simulated Target - Test 4 - Computation Time

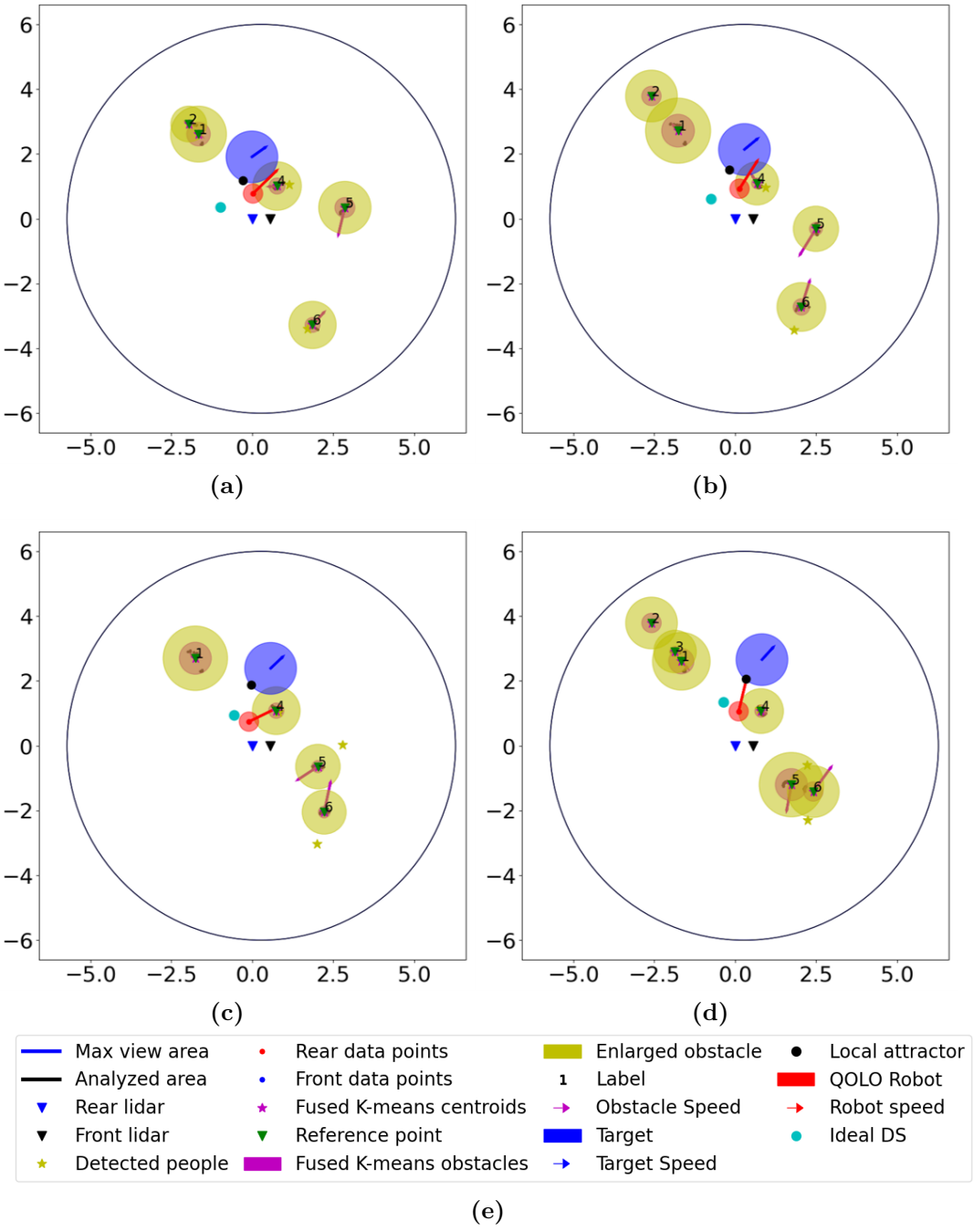


Figure 8.16: Simulated Target - Test 4 - Simulation

Test 5

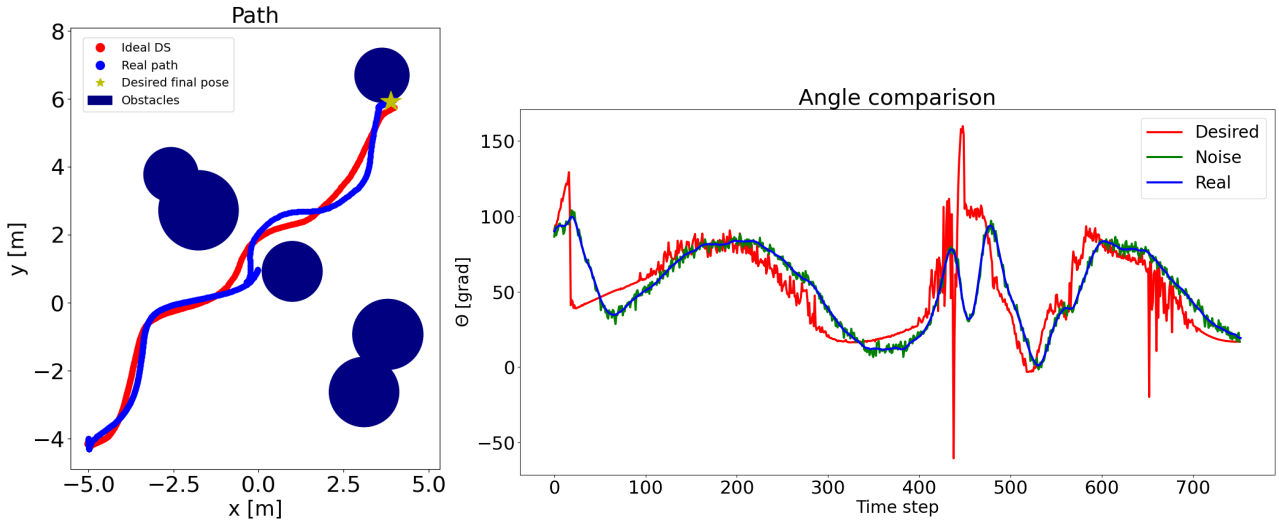
It is worth to test the same control algorithm in the same situation, using the Sigmoid function to generate the desired heading angle. In all the previous tests, the desired heading angle was given only by the direction of the target, but as we already saw, the contribution given by the modulated velocity, especially in the presence of obstacles or far from the attractor, can be very relevant. Therefore, in this test we will focus mainly on the robot beahviour, rather than on the control inputs or the computation time, since it is clear from the previous tests that all the constraints are observed.

The initial target position is again $\xi_{tar} = [-6., -4.]^T$, while the initial robot pose is $\xi = [-5., -4., \pi/2]^T$. Equations 8.2 and 8.3 describe the dynamics of the target.

The gain matrices are:

$$G_\theta = q_\theta^2 \qquad G_{Vx} = q_{Vx}^2 \qquad G_{Vy} = q_{Vy}^2 \qquad G_u = R \cdot R \tag{8.15}$$

$$q_\theta = 4 \qquad q_{Vx} = 8 \qquad q_{Vy} = 8 \qquad R = \begin{bmatrix} 0.5 & 0 \\ 0 & 1.5 \end{bmatrix} \tag{8.16}$$



(a) Path **(b)** Angle comparison

Figure 8.17: Simulated Target - Test 5

The behaviour is very similar to the previous test: the robot is not able to avoid immediately the obstacle in front of it. Firstly, the robot crosses the obstacle border and then the repulsive force pushes it away. This is mainly caused by two reasons:

1. the function that updates the reference points inside the obstacles is not used, therefore if the obstacles overlap, the modulation is not able to guarantee the obstacle avoidance.

2. the Sigmoid function generates the weights for the angle obtained from the modulated velocity direction and the angle of the target velocity. When the robot is near the attractor, the weight of the target velocity is bigger then the other, therefore the contribution given by the modulation is almost negligible, causing the robot to go straight into the obstacle until the attractor is far enough to allow the angle obtained by the modulated velocity to be the reference angle.

Of course, this behaviour presents some benefits when there are no obstacles and the robot has just to follow the target. In that case, using the Sigmoid function or directly the target's velocity as reference produces a very nice behaviour that can not be obtained using the modulated velocity for the reference angle.

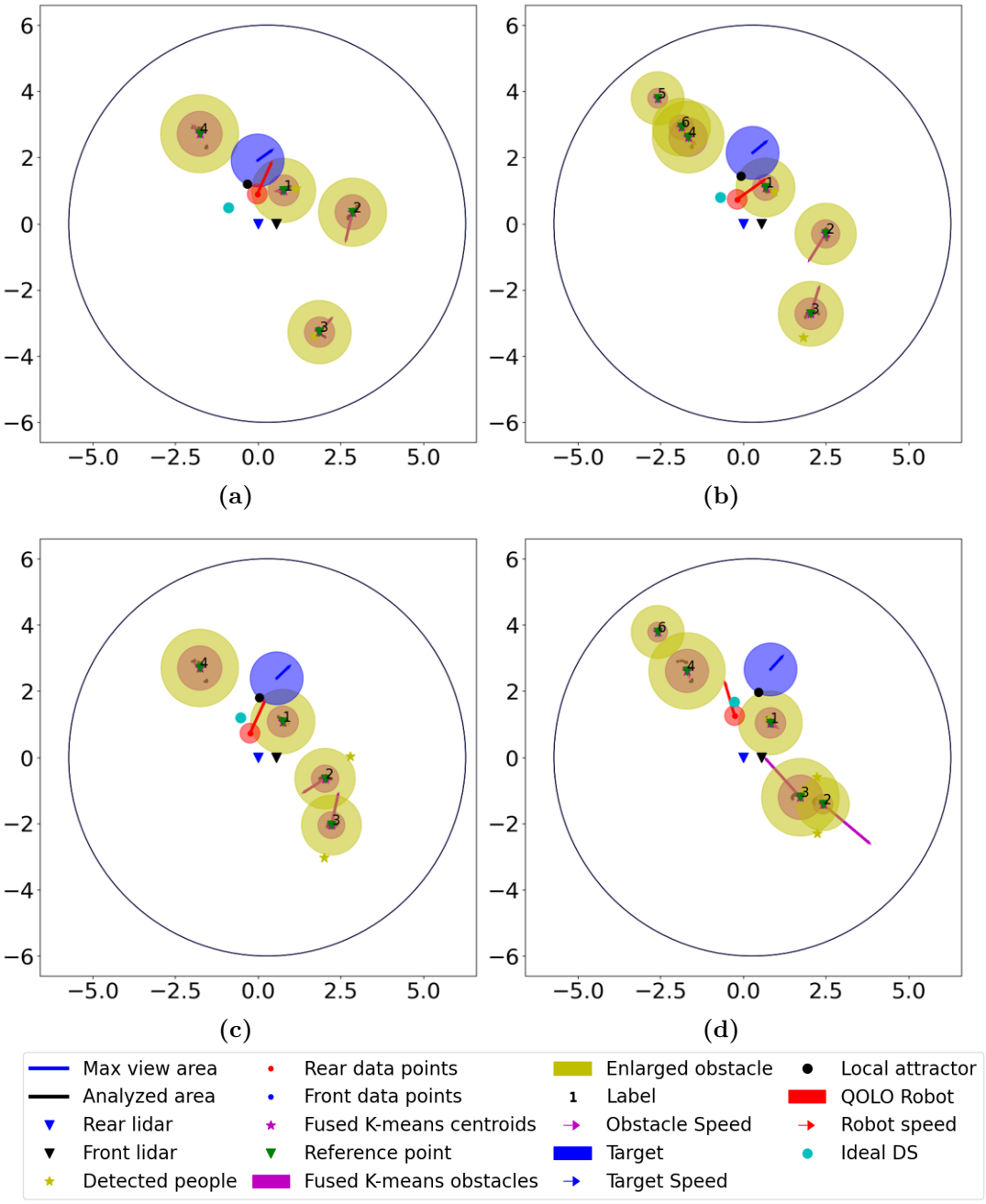


Figure 8.18: Simulated Target - Test 5 - Simulation

8.2 Detected Target

The previous tests described a scenario in which the target was simulated and all its characteristics were totally known. In this section, the target will be chosen among the detected obstacles, using the information obtained by the people detection algorithm, already implemented on the robot. The procedure its straightforward and it is illustrated in Algorithm 13. The output of this algorithm is simply the label that identify an obstacle as the target and, until the robot is able to track that particular obstacle, it is able to identify it as the target.

Knowing the obstacle that acts as target, we have also the information on its velocity; therefore, it is possible to apply the same algorithm used with the simulated target. The following tests will describe the behaviour of the QOLO robot in this scenario.

Algorithm 13 Target choice

▷ Initialize the flag $attractor_{on}$ that tells us if the target has been identified
$attractor_{on} \leftarrow 0$
▷ Initialize the variable $attractor_{lab}$ that stores the value of the target label
$attractor_{lab} \leftarrow 0$
while System is working **do**
    if $attractor_{on} == 0$ **then**
        ▷ Compute the distances between the K-means centroids and the centroids obtained with the people detection algorithm $D_{centroids-people}$
        ▷ $attractor_{lab}$ stores the label of the minimum distance
        $attractor_{lab} \leftarrow where(min(D_{centroids-people}))$
        $attractor_{on} \leftarrow 1$
    else
        ▷ Check if the target still exists
        if $Obstacle(attractor_{lab})$ does not exists **then**
            ▷ Reset the variables
            $attractor_{on} \leftarrow 0$
            $attractor_{lab} \leftarrow 0$
        end if
    end if
end while

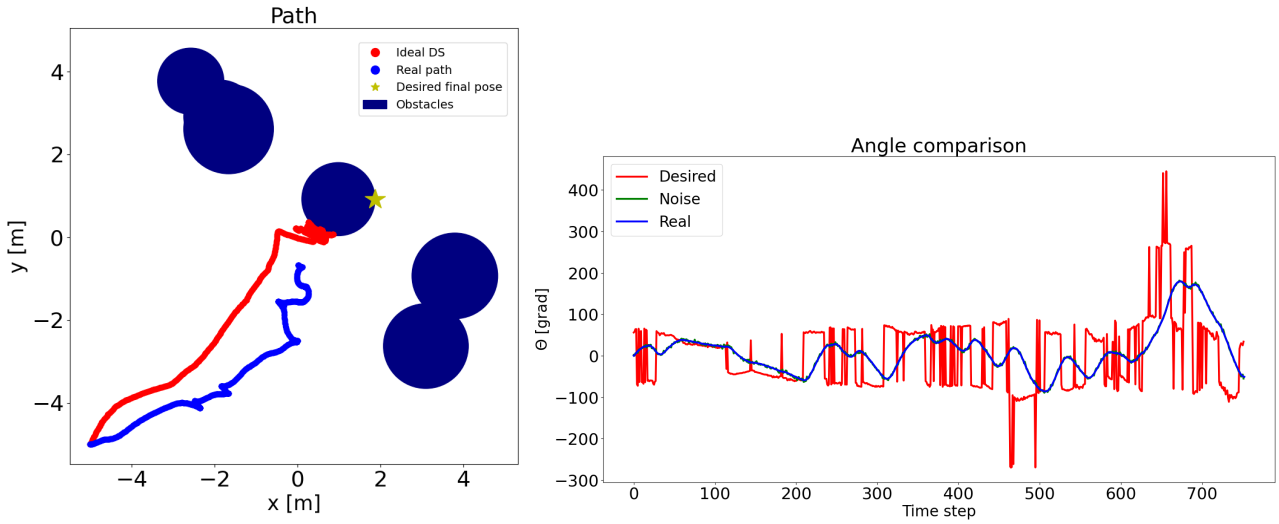
Test 1

A clarification is needed before analyzing the results of this tests: the situation here described is quite unrealistic because the data points obtained during the record session describe three people that move randomly around the robot. It would be necessary to test the robot in real-time using the real data points while the robot is moving to verify the actual functionality of the whole system. Nevertheless, this test can help us understanding some of the issues that may arise using this algorithm.

The initial pose of the robot is, as in the previous tests, $\xi = [-5., -4., \pi/2]^T$. The Sigmoid function is used to compute the heading angle and the gain matrices here used are:

$$G_\theta = q_\theta^2 \qquad G_{Vx} = q_{Vx}^2 \qquad G_{Vy} = q_{Vy}^2 \qquad G_u = R \cdot R \tag{8.17}$$

$$q_\theta = 4 \qquad q_{Vx} = 8 \qquad q_{Vy} = 8 \qquad R = \begin{bmatrix} 0.5 & 0 \\ 0 & 1.5 \end{bmatrix} \tag{8.18}$$



(a) Path **(b)** Angle comparison

Figure 8.19: Detected Target - Test 1

Figure 8.19 and Figure 8.20 display the behaviour of the robot that tries to reach the attractor. The robot is not able to go straight until it reaches the attractor or meets an obstacle: it tends to steer even if it is not necessary.

What emerges is that the algorithm that computes the heading angle, in this situation can not be used with the given gain matrices. The reason is that, far from the attractor, the heading angle is given by the direction of the modulated velocity, but it also depends on the heading direction of the target. As explained in Section 6.5, the equations that determine the reference heading angle are:

$$\theta_{target} = arctan2(\dot{y}_{target}, \dot{x}_{target})$$

$$\theta_{mod} = arctan2(\dot{y}, \dot{x})$$

If $\|\theta_{mod} - \theta_{target}\| > \frac{\pi}{2}$, $\theta_{mod} = -\theta_{mod}$

$$\theta_{ref} = \theta_{mod} \cdot \frac{1}{1 + e^{-s \cdot d}} + \theta_{target} \cdot \frac{1}{1 + e^{s \cdot d}} \tag{8.19}$$

Since in this simulation, the target changes direction very rapidly, the angle θ_{mod} changes sing quite often, causing the robot to steer uselessly.

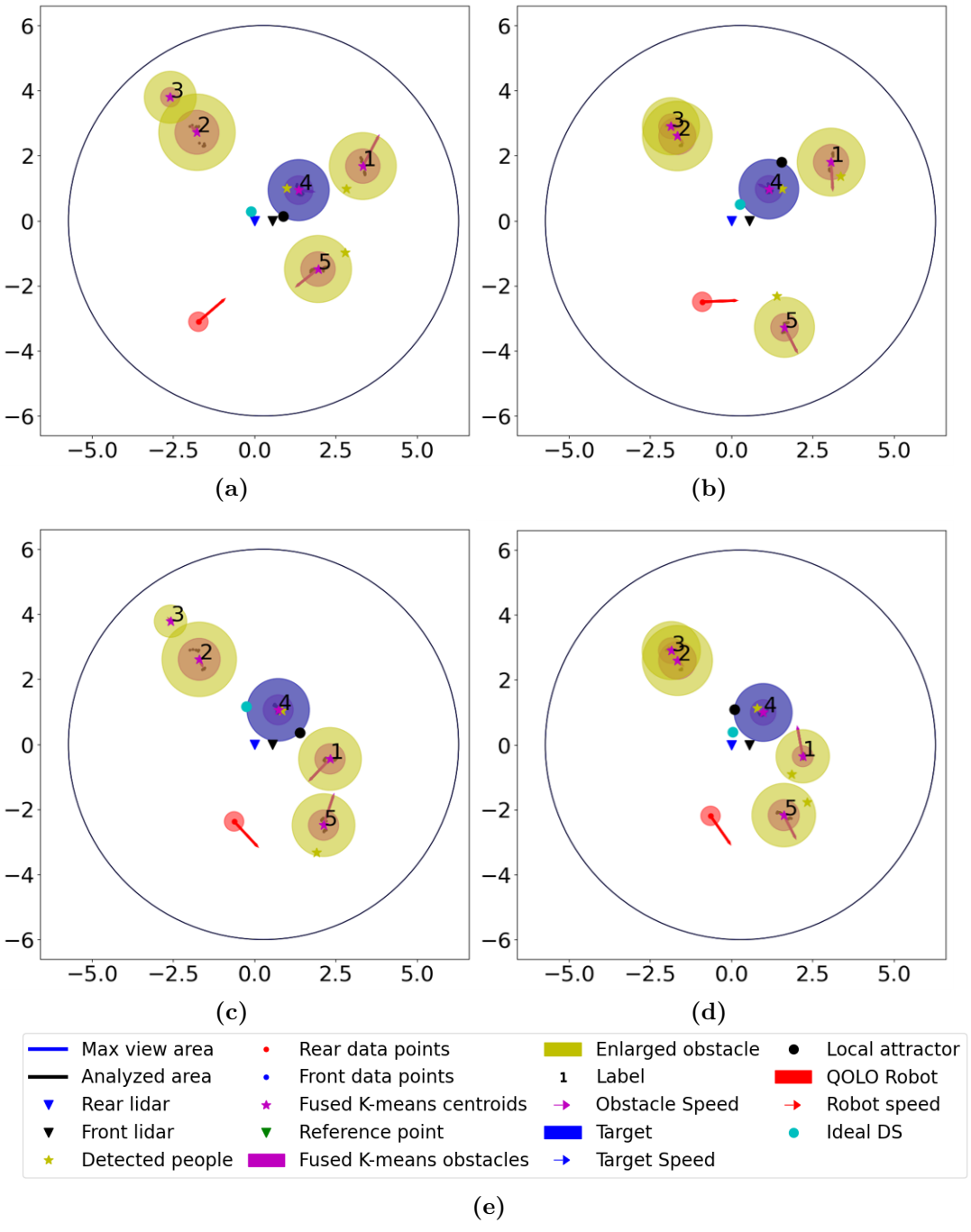


Figure 8.20: Detected Target - Test 1 - Simulation

Test 2

This second test aims to verify the assumption on the heading angle done in the previous test. For this reason, the same scenario is repeated, but in this case, the gain matrices are:

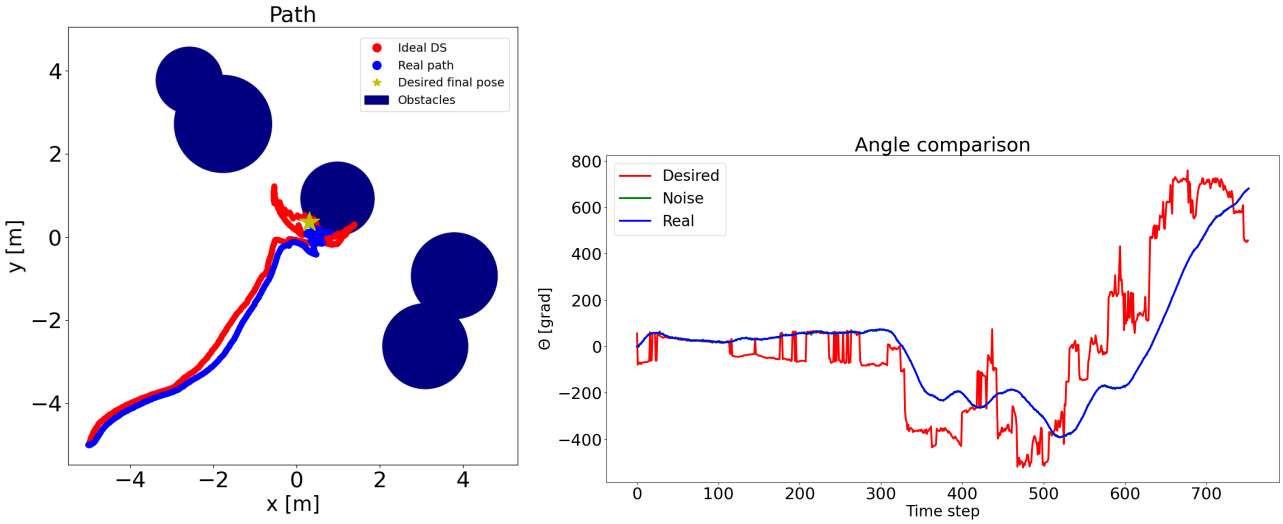
$$G_\theta = q_\theta^2 \qquad G_{Vx} = q_{Vx}^2 \qquad G_{Vy} = q_{Vy}^2 \qquad G_u = R \cdot R \tag{8.20}$$

$$q_\theta = 1 \qquad q_{Vx} = 10 \qquad q_{Vy} = 10 \qquad R = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \tag{8.21}$$

We have already seen that these matrices give more relevance to the velocity vector, rather than the desired heading angle; therefore, the expected behaviour should be more consistent and smooth in the first part, but this method may cause the robot to go backwards in other situations, as illustrated in Section 8.1 Test 3.

Figure 8.19 and Figure 8.20 display the behaviour of the robot that tries to reach the attractor. In this case, as expected, the robot reaches smoothly the target. The main problem is, of course, linked to the movement of the target itself, that rotates too much, forcing the robot to move in a very uncomfortable way.

Nevertheless, as said at the beginning of this Section, the scenario here presented is not very realistic, but it is very useful to understand the main limits of this algorithm and the scenarios that should be avoided.



(a) Path **(b)** Angle comparison

Figure 8.21: Detected Target - Test 2

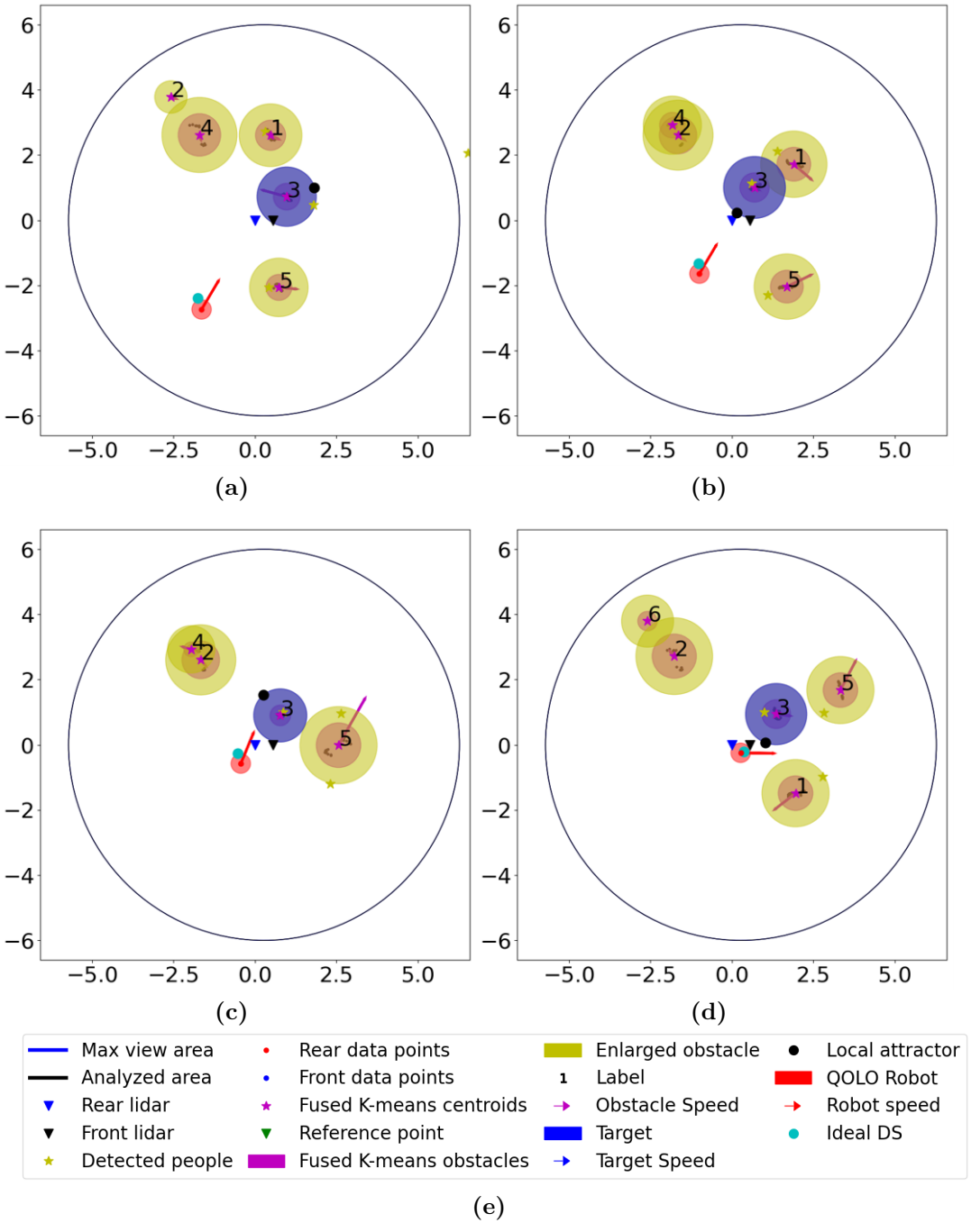


Figure 8.22: Detected Target - Test 2 - Simulation

Chapter 9

Conclusions, limits and future development of the work

The thesis project described in the previous Chapters has proved to have achieved several good results. As first thing, the addition of the attractor's velocity in the modulation algorithm, described in Chapter 5, represents a novel feature that can be used to better model a dynamic environment. Using the DS modulation alone, with a moving attractor, allow us to obtain local information on the needed velocity to reach the attractor, taking also into account its the movement. This feature acts as a predictive characteristic that anticipate the future behaviour of the target.

Another interesting aspect that emerges from this project concerns the dynamic of the local moving attractor, described in Section 5.10. It is a procedure that allows an attractor to move around a defined target depending on the presence of the obstacles in the environment. It is a useful tool that helps the modulation and therefore the robot, to avoid inconvenient situations in which the attractor goes inside an obstacle, even if there could be enough space behind the target to avoid the obstacle.

In addition, the study of the non-linear Model Predictive Control with velocity control, as illustrated in Section 6.5, represents another excellent achievement of this work. Differently from other control algorithms used for the unicycle, some of which also tried during the project, this controller does not use a position as reference, but a velocity. It also uses the desired heading angle, but this is an addition to force the robot to move facing a desired direction. The real interesting aspect is the possibility to impose a desired behavior to the unicycle, in this case the reference velocity, while including in the computation the kinematics of the robot and all the constraints that we can think about. In particular, in this project

the non-linear MPC has been exploited to guarantee that the control inputs would observe the limits imposed by the robot structure, such as the maximum speed and acceleration allowed by the motors, and that the overall experience of the users would be as comfortable as possible. The main drawback of that controller is the computation time: a longer prediction horizon could improve the system performances, avoiding unnecessary overshoots in the control inputs, but it requires a considerable computation effort.

Another good characteristic of the algorithm here developed is its versatility. The three sub-algorithms can work independently one from the other, therefore, if a better Low level controller was developed, it would be possible to remove the non-linear MPC and add a new Low level controller quite easily. Moreover, it would be possible to change completely the system: the robot taken into account for the thesis can be modeled as a unicycle, and the Low level controller deals with this kinematics. However, the algorithm developed in this work can be adapted also to other kinematics, for instance to a bicycle kinematics. The High level controller and the Obstacle detection algorithm would be the same, and it would be necessary to modify only the Low level controller in order to include the the new kinematics and the new constraints.

As for the obstacle detection algorithm, the results obtained with the K-means algorithm seems very promising, both from the point of view of the performance and from the point of view of the computation time. One of the limits of the algorithm already implemented on the robot is the computation time itself: it requires too much time to be used in a real-time scenario. The algorithm developed in this project is for sure faster and lighter, also because it does not use any information from the 3D cameras, but exploits only the raw data from the LIDAR sensors. The main limitation is therefore that it can not be used to distinguish the people from the other elements in the environment, such as walls or other static obstacles. Nevertheless, the aim of the thesis work was to identify a method to be used on people, hence, the results, obtained by the K-means algorithm in an environment where only people were present, are more than satisfying.

Considering all the sub-algorithms and how they work together, it is possible to assert that the objective of the development of a DS-based control has been achieved. No planning strategies are used to compute the control inputs for the system and only local information are needed to produce such actions. Nevertheless, it is worth to mention that the Low level controller, and in particular the non-linear MPC, acts as a filter for the control inputs because it is able to impose acceleration constraints between the previous control inputs and the next ones.

Despite the several benefits obtained during the Master Thesis work, some issues must still be solved and some inaccuracies in the control strategy have to be better

tackled.

A first aspect to point out is for sure the impossibility, for the control strategy, to take into account, in advance, the non-holonomic constraints. This affects the obstacle avoidance performance because the DS-modulation is applied on the center of the robot as if it was a mass point; from the point of view of this point, certain actions to avoid the obstacles can be taken quite near the border. Of course, changing the direction when the obstacle is quite close or when the obstacle shape change constantly (due to the obstacle detection algorithm), can be a problem for a robot that has non-holonomic constraints, other then speed and acceleration constraints. Due to the delay in the reaction of the robot, this one can cross the border of the obstacle. As explained in the previous chapters, it should not be a problem because we are considering enlarged obstacles with a safety distance that allows the robot to brake. Nevertheless, the main issue of the robot that crosses the obstacle border is the discontinuity of the DS. This is actually a huge problem because the desired velocity produced by the High Level Controller changes abruptly, forcing the Low Level Controller to produce control inputs that push away the robot without considering the presence of the attractor.

In order to solve this problem, the future developments of this project will focus on the production of a modulated DS without discontinuities. In such a way, even if the robot enters in the area defined by the obstacles, it is pushed away in a continuous way and the attractor reference is not lost. A possible solution could be to use the Gaussian Mixture Model to shape the obstacles and exploits the information on the probability distribution of the obstacles to compute a continuous modulation. In this way it could be possible to avoid the definition of the obstacles' border and therefore, to produce a smoother behaviour and a more consistent control action that relies on the robot's sensors.

Implementing such a solution would require to use a GMM algorithm for the obstacle detection, rather then the K-means here implemented. Chapter 7 describes which are the main difficulties of this strategy, but further researches could find a feasible solution for a dynamic environment, without the need of a serious computation effort.

Finally, we have seen that the choice of the reference heading angle can be a non-trivial task, that can cause minor problems when the robot is very close to the attractor and an obstacle approaches. For sure, using a continuous modulated DS would improve also this characteristic of the algorithm, especially considering that the main issues related to the angle definition have emerged when the robot goes inside an enlarged obstacle, causing the reference angle to change suddenly.

Bibliography

[1] *Cobot - Wikipedia*. URL: https://en.wikipedia.org/wiki/Cobot. (accessed: 18.10.2020) (cit. on p. 1).

[2] *CrowdBot: an European Collaboration*. URL: http://crowdbot.eu/. (accessed: 18.10.2020) (cit. on pp. 1, 7–11).

[3] *Qolo: A Standing Mobility Vehicle*. URL: http://www.ai.iit.tsukuba.ac.jp/research/046.html. (accessed: 17.10.2020) (cit. on pp. 1, 7).

[4] *MATLAB - Website*. URL: https://www.mathworks.com/products/matlab.html/. (accessed: 18.10.2020) (cit. on pp. 5, 11).

[5] L. Huber, A. Billard, and J. Slotine. «Avoidance of Convex and Concave Obstacles With Convergence Ensured Through Contraction». In: *IEEE Robotics and Automation Letters* 4.2 (2019), pp. 1462–1469 (cit. on pp. 5, 16, 18, 22, 31).

[6] *Python - Website*. URL: https://www.python.org/. (accessed: 18.10.2020) (cit. on pp. 6, 11).

[7] *Lukas Huber*. URL: http://lasa.epfl.ch/people/member.php?SCIPER=274454. (accessed: 19.10.2020) (cit. on pp. 6, 26, 146).

[8] *Gaussian mixture models*. URL: https://scikit-learn.org/stable/modules/mixture.html#:~:text=A%5C%20Gaussian%5C%20mixture%5C%20model%5C%20is,Gaussian%5C%20distributions%5C%20with%5C%20unknown%5C%20parameters.. (accessed: 19.10.2020) (cit. on p. 6).

[9] *K-means clustering*. URL: https://en.wikipedia.org/wiki/K-means_clustering. (accessed: 19.10.2020) (cit. on pp. 6, 136).

[10] *CrowdBot – Qolo*. URL: https://diegofpaez.wordpress.com/portfolio/crowdbot-qolo/. (accessed: 06.12.2020) (cit. on pp. 7–11).

[11] Diego Felipe Paez-Granados, Hideki Kadone, and Kenji Suzuki. «Unpowered Lower-Body Exoskeleton with Torso Lifting Mechanism for Supporting Sit-to-Stand Transitions». In: *IEEE International Conference on Intelligent Robots and Systems*. Madrid: IEEE Xplorer, 2018, pp. 2755–2761. ISBN: 9781538680940. DOI: 10.1109/IROS.2018.8594199 (cit. on pp. 7–11).

[12] *Crowdbot - Our bots*. URL: http://crowdbot.eu/our-bots/. (accessed: 17.10.2020) (cit. on p. 7).

[13] Yang Chen, Diego Paez-Granados, Hideki Kadone, and Kenji Suzuki. «Control Interface for Hands-free Navigation of Standing Mobility Vehicles based on Upper-Body Natural Movements». In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-2020)*. Las Vegas, USA: IEEE Xplorer, 2020 (cit. on p. 8).

[14] *Velodyne Lidar*. URL: http://m.elecfans.com/article/1224503.html. (accessed: 06.12.2020) (cit. on p. 8).

[15] *Intel RealSense Depth Camera d435*. URL: https://www.intelrealsense.com/depth-camera-d435/. (accessed: 18.10.2020) (cit. on p. 8).

[16] *RealSense SDK 2 - sample*. URL: http://unanancyowen.com/en/realsense-sdk-2-samples/. (accessed: 18.10.2020) (cit. on p. 9).

[17] *Nonholonomic system*. URL: https://en.wikipedia.org/wiki/Nonholonomic_system. (accessed: 18.10.2020) (cit. on p. 10).

[18] Maciej Michałek and Tomasz Gawron. «VFO Path following Control with Guarantees of Positionally Constrained Transients for Unicycle-Like Robots with Constrained Control Input». In: *Journal of Intelligent and Robotic Systems* 89 (Jan. 2018), pp. 191–210. DOI: 10.1007/s10846-017-0482-0 (cit. on p. 10).

[19] *Robot Operating System - Website*. URL: https://www.ros.org/. (accessed: 18.10.2020) (cit. on p. 11).

[20] *Robot Operating System - Wikipedia*. URL: https://en.wikipedia.org/wiki/Robot_Operating_System. (accessed: 18.10.2020) (cit. on p. 11).

[21] *Python - Wikipedia*. URL: https://en.wikipedia.org/wiki/Python_(programming_language). (accessed: 18.10.2020) (cit. on p. 11).

[22] *MATLAB - Wikipedia*. URL: https://en.wikipedia.org/wiki/MATLAB. (accessed: 18.10.2020) (cit. on p. 11).

[23] *Dynamical systems - Wikipedia*. URL: https://en.wikipedia.org/wiki/Dynamical_system#cite_note-3. (accessed: 20.10.2020) (cit. on p. 15).

[24] *Nature - Dynamical systems*. URL: https://www.nature.com/subjects/dynamical-systems. (accessed: 20.10.2020) (cit. on p. 15).

[25] *Flatness (systems theory) - Wikipedia*. URL: https://en.wikipedia.org/wiki/Flatness_(systems_theory). (accessed: 02.12.2020) (cit. on p. 45).

[26] Prof. Alessandro Rizzo. *Lecture notes in Robotics*. June 2019 (cit. on p. 45).

[27] Amar Khoukhi and Mohamad Shahab. «Stabilized Feedback Control of Unicycle Mobile Robots». In: *International Journal of Advanced Robotic Systems* 10 (Apr. 2013), p. 1. DOI: 10.5772/51323 (cit. on pp. 51–53).

[28] *NumPy*. URL: https://numpy.org/. (accessed: 05.12.2020) (cit. on p. 58).

[29] *NumPy - Wikipedia*. URL: https://en.wikipedia.org/wiki/NumPy. (accessed: 05.12.2020) (cit. on p. 58).

[30] *Function numpy.random.normal()*. URL: https://docs.scipy.org/doc//numpy-1.10.4/reference/generated/numpy.random.normal.html. (accessed: 05.12.2020) (cit. on p. 58).

[31] *Model predictive control - Wikipedia*. URL: https://en.wikipedia.org/wiki/Model_predictive_control. (accessed: 02.12.2020) (cit. on p. 64).

[32] *Euler method - Wikipedia*. URL: https://en.wikipedia.org/wiki/Euler_method. (accessed: 02.12.2020) (cit. on p. 65).

[33] *Metodo di Eulero - Wikipedia*. URL: https://it.wikipedia.org/wiki/Metodo_di_Eulero. (accessed: 02.12.2020) (cit. on p. 65).

[34] *Metodi di Runge-Kutta - Wikipedia*. URL: https://it.wikipedia.org/wiki/Metodi_di_Runge-Kutta. (accessed: 02.12.2020) (cit. on p. 65).

[35] Joel A E Andersson, Joris Gillis, Greg Horn, James B Rawlings, and Moritz Diehl. «CasADi – A software framework for nonlinear optimization and optimal control». In: *Mathematical Programming Computation* 11 (2019), pp. 1–36 (cit. on p. 66).

[36] Prof. Luca Bascetta. *Lecture notes in Control of Mobile Robots - Kinematics of mobile robots* (cit. on p. 94).

[37] *Machine learning - Wikipedia*. URL: https://en.wikipedia.org/wiki/Machine_learning. (accessed: 02.12.2020) (cit. on p. 107).

[38] *Apprendimento automatico - Wikipedia*. URL: https://it.wikipedia.org/wiki/Apprendimento_automatico. (accessed: 02.12.2020) (cit. on p. 107).

[39] *Gaussian Mixture Models Explained*. URL: https://towardsdatascience.com/gaussian-mixture-models-explained-6986aaf5a95. (accessed: 02.12.2020) (cit. on p. 108).

[40] *Mobile Robot Algorithm Design*. URL: https://www.mathworks.com/help/robotics/ground-vehicle-algorithms.html. (accessed: 02.12.2020) (cit. on p. 113).

[41] *Scikit-learn - Wikipedia*. URL: https://en.wikipedia.org/wiki/Scikit-learn. (accessed: 02.12.2020) (cit. on p. 117).

[42] *Scikit-learn - Local Outlier Factor algorithm*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.LocalOutlierFactor.html. (accessed: 02.12.2020) (cit. on p. 117).

[43] *Scikit-learn - Gaussian Mixture Model algorithm*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html#sklearn.mixture.GaussianMixture. (accessed: 02.12.2020) (cit. on p. 117).

[44] *K-means: A Complete Introduction*. URL: https://towardsdatascience.com/k-means-a-complete-introduction-1702af9cd8c#:~:text=Nov%5C%2019%5C%2C%5C%202019%5C%C2%5C%B79%5C%20min,classifies%5C%20them%5C%20together%5C%20into%5C%20clusters.. (accessed: 02.12.2020) (cit. on p. 136).

[45] *K-Means Clustering*. URL: hhttps://towardsdatascience.com/k-means-clustering-13430ff3461d. (accessed: 02.12.2020) (cit. on p. 136).

[46] *Scikit-learn - K-means*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.k_means.html?highlight=k%5C%20means#sklearn.cluster.k_means. (accessed: 05.12.2020) (cit. on p. 136).

[47] *Function numpy.cov()*. URL: https://numpy.org/doc/stable/reference/generated/numpy.cov.html. (accessed: 05.12.2020) (cit. on p. 137).