# POLITECNICO DI TORINO

Master's Degree in Computer engineering


Politecnico di Torino

Master's Degree Thesis

# Experimental setup for collision avoidance algorithms for mobile robots

**Supervisors**

Prof. Stefano MAURO

Eng. Matteo MELCHIORRE

**Candidate**

Micaela Mara POSSETTO

December 2022

**Abstract**

Collision avoidance is a topic of utmost relevance in mobile robot navigation, where robots should be able to reach a goal and to avoid obstacles on their way autonomously, to operate consistently in a real-life environment. For this purpose, several algorithms have been developed over the last years.

This work aims at providing a real-time collision-free path for mobile robots, implementing in real world a pre-existent collision avoidance algorithm, which has been tested so far in a simulation setting only. Specifically, the chosen technique improves the classical artificial potential fields by considering local attractors in addition to repulsors, in order to drive the robot towards a goal, while avoiding obstacles along preferred directions.

The first phase consisted in reproducing the conditions of the simulated environment in a laboratory setup made of a single obstacle and a mobile robot controlled by the algorithm, that has been implemented in ROS (Robot Operating System). In this first step, the pose of the obstacle was known and fixed and the robot pose was estimated by odometry sensors in terms of position and orientation. At the end of this process, it was possible to identify potential improvements with the purpose of perfectioning position measurements and obtaining an online obstacle detection. This was achieved by using a color camera and ArUco markers, opportunely placed in order to track the pose of the relevant elements: robot, obstacle and final pose.

With the results achieved in the first phase, the effort focused in studying a significant application for mobile robots, where obstacles can either be objects or humans. Thus, the final step of this work consisted of different tests to prove the effectiveness of the proposed technique. Each test is characterized by the same initial and final pose of the robot, but different obstacle poses, so that the robot approaches to the object from different directions. Results show that the robot is able to pass on the desired side with respect to the obstacle, i.e. the side where the local attractor is placed. Future works will involve the study of some aspects that are still to be developed to obtain a reliable result and bring this technique in real life, such as dynamic obstacle and multiple obstacles.

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Industry 4.0

The importance of automation in the industrial environment has remarkably increased in recent years. The main goal was to reduce human intervention in processes, in order to reduce costs, increase production and improve quality standards. This was made possible involving robots.

Robots have been introduced in industrial sector after the 3$^{rd}$ industrial revolution. They are defined by the American Robot institute as *re-programmable multi-functional manipulator designed to move materials, parts, tools, or specialized devices through variable programmed motions for the performance of a variety of tasks* [1]. Their capacity to emulate humans' ability is the reason why robotics widespread in industrial automation, especially in the manufacturing sector. However, mostly on account of their rigid structure, robots can be a source of hazard for operators. Because of that, the industrial application has been limited to dedicated areas, surrounded by physical fences, not allowing direct contact with workers.

Over the last decade, with the 4$^{th}$ industrial revolution, commonly known as Industry 4.0, one more step has been made. The concept of Industry 4.0 puts the focus on both automation and computer science.

Concerning automation, it aims at employing autonomous, flexible and cooperative robots, to build smart factories where physical fences, dividing robots and operators, are eliminated, resulting in a shared workspace [2]. As a result, it arises the idea of collaborative robotics which expect that robots and humans cooperate to accomplish one task together and simultaneously. This is what can be called human-robot collaboration and it is mainly

related to manipulators, applied in industrial sector. Concurrently to this concept, innovation was brought in mobile robotics too. Mobile robots can be considered as service robots *that perform useful tasks for humans or equipment* [3]. In this field although, cooperation is not required most of the time. Robots work closely to humans, doing repetitive and monotonous tasks on their behalf and reducing their physical effort. Sometimes they are applied together with cooperative robots to benefit of the capability of co-bots to cooperate with humans, on one side, and, on the other side, of the possibility to move. Most of the times instead, mobile robots are applied alone. An example in a non-industrial field, can be domestic where, for example, vacuum cleaners are quite popular applications.

Furthermore, computer science can be considered as well the earth of this fourth revolution. Actually, the common thread led by Industry 4.0, lies in the capacity of robots to complete tasks in an intelligent way, thanks to the new technologies applied, reported in Figure 1.1. Therefore, the main difference from what we were used to see before is represented by the intensive employment of informatics. This is partly due to the necessity to accomplish the requested independence, meaning that a great amount of information must be handled. This has been made possible thanks to the central role played by IT in Industry 4.0, which enables to bring both automation and digitization at a much higher level. For example, in both collaborative and service robotics, robots move side by side to humans. Therefore, it is important to mitigate safety hazards considering real-time responses that are obtained exploiting the capabilities of IoT (Internet of Things).


**Figure 1.1:** Technologies related to industry 4.0 [4]

## 1.2 AGVs and mobile robots

### 1.2.1 Automated Guided Vehicles

Despite manipulators represent the majority of robots applied in industry, another type of robot is gaining popularity in the last years due to increasing transportation demand. They are called Automated Guided Vehicles (AGVs) and are mainly implemented in logistics, manufacturing and medicine, to move parts and tools from point A to B [5]. The main task demanded to AGVs is path planning, consisting in searching for a sequence of segments to reach the final pose. To do so, the vehicle shall be able to locate itself, and this is made possible through different types of sensors, eventually combined to obtain a more consistent feedback. For example, it is quite common to find in factories AGVs locating through the detection of guidelines on the floor, as it is shown in 1.2. In this case, they are not able to know their exact position, they just stay on the track. More advanced AGVs instead, are able to know their absolute position, combining the previous method with a knowledge of the distance covered that can be obtained from encoders on the wheels or lasers mounted on the vehicle. Generally, these types of vehicles are designed to be used in a structured environment where everything is known and a basic collision-free path has to be computed. However, AGVs can be considered as precursors of mobile robots, where autonomy is requested, and the goal is to move to an unstructured setting. This means from a static to a dynamical planning.


**Figure 1.2:** AGVs following tracks on the floor [6]

### 1.2.2 Mobile robots

Mobile robots have been introduced with collaborative robotics, but they are not mainly intended for cooperating with humans, as co-bots instead are designed to be. As previously stated, they are related to the concept of service robotics and are becoming consistently important, in view of autonomous navigation, which is required in different sectors, from the industrial environment to the human's assistance in daily life.

A robot can be considered autonomous when it does not require the supervision of humans. Using mobile robots, an efficient transportation of goods can be perceived, meeting the Industry 4.0's paradigm. However, it is necessary to implement a way of making decisions dynamically and gathering information, in order to adapt to circumstances. Actually, in real world, while a robot is following the computed path it may face obstacles that can be unforeseen objects, a person or eventually other mobile robots acting at the same time. Therefore, the new task is to provide a real-time collision-free path, exploiting information of the surrounding environment that can derive, for example, from sensors, or cameras.

Mobile robots differ from fixed ones, because of their locomotion system, consisting in a mobile base which allows the robot to move freely. Two types can be distinguished: wheeled and legged. The former consists of a rigid body and a system of wheels, the latter is made up of multiple rigid bodies interconnected by joints.

Mobile robotics is emerging in different sectors, such as domestic, agriculture, military and, of course, manufacturing. It is mainly represented by wheeled robots, thanks to the simpler use of wheels than legs or treads, and less balance issues. Even if is a field that still has to be studied a lot, wheeled mobile robots are no more just prototypes. Today, we can find them in ordinary houses: cleaning mobile robots are already on the market, produced by several companies, such as Dyson and iRobot [7]. In the industrial environment, they are starting to substitute AGVs which were not able to face unforeseen obstacles: the company Mobile Industrial Robotics already manufactures different models of autonomous robots, like MiR100 or MiR250 [8]. In other sectors they have emerged just in the last years, such as in agriculture. There, they are exploited, for example, for the analysis of the culture, as the model Terra Sentia from EarthSense does, or, combined with robotic arms to replace humans in the harvesting process [9]. Some models of the previous examples are shown in figures 1.3.

**(a)** *Domestic: iRobot j7 [7]*

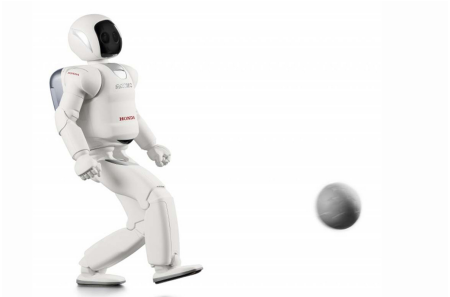
**(b)** *Industrial: MiR100 [8]*

**(c)** *Agriculture: TerraSentia [9]*

**Figure 1.3:** Wheeled mobile robots in different sectors

Humanoid robots, entertainment pets, and so forth instead, belong to the category of legged robots. They are inspired to living organism provided of legs. This feature enables them to move on irregular terrains on one hand, but on the other side, increase their complexity. Boston dynamics produces Spot which is an agile mobile robot, designed for factories that navigates irregular terrains, performing inspection tasks and data capture autonomously [10]. Other examples of legged robots are humanoid, designed to duplicate the complexity of humans [11]. In 1.4 some examples are illustrated.


**(a)** *Boston dynamics Spot [10]*



**(b)** *Honda ASIMO [11]*

**Figure 1.4:** Legged mobile robots in different sectors

Focusing on the most prevalent mobile robots, different types of wheels can be distinguished: fixed, steerable and castor, eventually combined. In 1.5 all the basic types are illustrated [1].

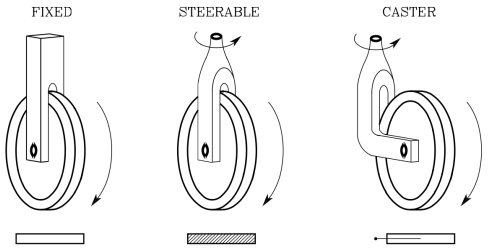
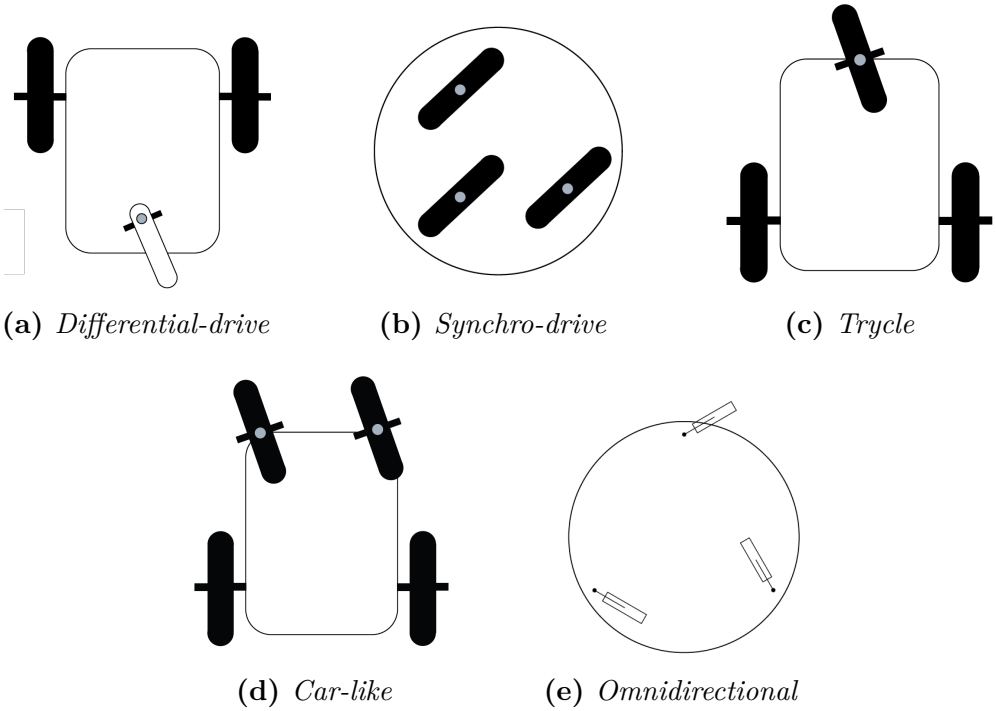



**Figure 1.5:** Types of wheel in wheeled mobile robots [1]

The vehicles deriving from the combination of these three types of wheels can all be classified in: differential-drive (1.6a), synchro-drive (1.6b), tricycle (1.6c), car-like (1.6d) and omnidirectional vehicles (1.6e).



**(a)** *Differential-drive* **(b)** *Synchro-drive* **(c)** *Trycle*

**(d)** *Car-like* **(e)** *Omnidirectional*

**Figure 1.6:** Types of vehicles [1]

Despite all the positive aspects, mobile robots are subjected to constraints,

called nonholonomic. These restrictions are related to the admissible motions and derive from the features of the implied wheels that prevent the possibility of attaining any position or orientation. This means that the number of degrees of freedoms is lower, and consequently, these constraints must always be considered when planning a movement.

## 1.3 Mobile robot navigation

### 1.3.1 Global navigation techniques

As previously stated, AGVs can be considered as precursor of mobile robots, since the latter represent an evolution of the former, in the way they navigate. Traditional AGVs are limited to predetermined path in structured environments, in a sort of separation not necessarily physical, from humans and eventual obstacles. The considered background is static: variations on time are not accounted and there is a prior knowledge that enables an offline planning. In this approach, the trajectory from the starting to the final goal is computed before the task has started, and it will not be modified during the execution, reason why it is also regarded as a global method.

Path planning in these conditions is based on classical approaches, such as cell decomposition and roadmaps [12]. Cell decomposition consists in dividing the region in cells recursively, classifying them in cells without and containing obstacles. Then, it computes the optimal path relying on the free ones and exploiting the graph theory. Roadmaps techniques generate a map summarizing the free space and compute the path following a decided criterion that can be, for example, Voronoi diagrams.

Global methods are usually treated as optimization problems, resulting in infinite solutions found. To select the optimal path, it should be decided a criterion that can be, for example, minimum time, or lowest energy consumption. This choice only depends on the particular case analyzed. For instance, for AGVs in factories, choosing the less energy consuming option, could be a great idea, in order to both minimize recharging and improve transportation. However, even if an optimal solution is computed, cell decomposition and roadmaps techniques have to do with a great amount of data that make this approach computationally heavy. This is the main disadvantage of AGVs, together with the inability to react to unpredictable obstacles. Indeed, in real life, the main problem is represented by humans, whose future moving directions are difficult to predict, but necessary to obtain a relevant result in

navigation [13].

### 1.3.2 Local navigation techniques

Mobile robots instead, are more compatible with real world, where requiring prior knowledge would restrict the domain of the possible applications. They must be able to navigate autonomously with enough intelligence to react and make decisions based on the perception received from the surrounding environment [14]. There are different ways of obtaining the required information: they can derive from sensors or cameras. Sensors are directly applied on the mobile robot and can be encoders, infrared, gyroscopes, and so on. Cameras can be 2D or 3D and integrated on the robot or not. Once data is received, it is important to analyze it properly and extract information about the environment which is subjected to changes over the time. In order to accomplish the desired behavior, mobile robots must be able to adapt on the go. Online strategies, also accounted as local methods, consist of observing the proximity of the robot and acting reactively. The main task which focuses on this aspect is called collision avoidance.

Generally, a mobile robot can have a prior knowledge of the environment on which it computes a first path. However it should always be aware of the presence of unpredictable dynamical obstacles that must be detected and avoided. This type of approach tries to combine offline and online methods' advantages. As mentioned in the previous section, mobile robots are already used in real world. They exploit these types of techniques to fulfill collision avoidance. They can be used singularly or in fleets. In the first case, they can have a partial knowledge of the environment, such as domestic robots that perform a preliminary mapping of the areas that need to be clean, in order to not go blind. In this case, robots are prepared to avoid humans or objects performing an online approach, with an additional awareness that enhances performances. A great example of fleets of mobile robots are Amazon fully automated warehouses where a hybrid solution between online and offline strategies is perceived. There, the autonomous mobile robots move pallets around autonomously. Though, having control on all the vehicles, trajectories are previously programmed to prevent possible collisions. Nevertheless variation are always considered and eventual operators or other occasional robots are detected and avoided [15].

Online approaches require a significant effort in terms of programming to react to uncertainties in an intelligent way. In fact, a first approach to pursue

this aim, could be to slow down the robot when an obstacle is detected by the applied sensors, eventually stopping. Even if this approach does not require difficult implementations, applying this strategy would result in a performance leakage. The process would result stuck, even permanently on a deadlock, and those strategies would be worthless. Consequently, this would reflect on task time, having performances comparable, or even worse, to sequential processes. Aiming at obtaining a better result, other approaches have been investigated. Some worth mentioning examples are probabilistic roadmaps and APF-based (Artificial Potential Fields) methods. The first method tries to simplify the corresponding offline approach, to reduce the computation cost. In APF-based methods, target is treated as an attractor and obstacles as repulsors, to move the robot towards the goal, following the negative gradient. Probabilistic roadmaps, as well as other similar methods, need to build a map to navigate and this makes them not so efficient in an environment that is potentially always changing. APF instead, represents one of the best solutions, because it does not require onerous operations. Reason why, this work will focus on this approach.

## 1.4 Aim of the thesis

As stated in the previous sections, mobile robots emerged with Industry 4.0, as an evolution of AGVs, and the most challenging topic related to them is: real-time collision avoidance.

This thesis focuses on a customized existent collision avoidance technique, based on APF method [16]. This technique can be applied at any type of obstacle, meaning that both random objects or humans are considered. For this reason, it meets the main requirement of Industry 4.0's paradigm: the safe coexistence of humans and robots in the same area.

More specifically, this technique implements a way to influence robot's trajectory on preferred regions to avoid an obstacle. This is related to the fact that, in real-world there are some path which are preferable to others. Especially when humans are considered in the working environment, this feature would be of great benefit. Some studies proved that humans would feel more comfortable if they knew where the robot should pass. Conversely, applying basic APF methods, a robot preventing an obstacle would run on arbitrary directions, depending on minimal deflections of motion. Even if these researches related to humans perceptions are mainly related to

anthropomorphic co-bots, it can be supposed that this assumption is valid for mobile robots too. In conclusion, the aim of this novel technique is to make robot's behaviour predictable.

In [17] and [16] this approach was tested on a simulation setting, respectively considering an anthropomorphic co-bot and a mobile robot. This thesis will focus on the implementation in real-world of the trial with a mobile robot. There, a basic setup where obstacle position was priory known was considered. After implementing the basic behaviour, the next step of this dissertation will be to move in direction of a more relevant result where obstacle position is detected on the go.

## 1.5 Thesis outline

Intending to give a brief overview on how this thesis is organized, chapters can be summarized in this way:

- Chapter 1 is a general introduction on mobile robots.
- Chapter 2 is devoted to a detailed explanation of the approach on which this dissertation is based.
- Chapter 3 presents the experimental setup. An introduction to the tools used is done, along with an explanation of how they were exploited. In addition to that, eventual suggestions of different strategies that could have been applied is given.
- Chapter 4 is the heart of this work. It reports the development of different tests and analyzes the obtained results.
- Chapter 5 aims at drawing the conclusions of this dissertation, summarizing the advantages and disadvantages remarked. Moreover, possible applications and future work are also presented.

# Chapter 2

# Path planning using potential fields with local attractors

## 2.1 State of the art: collision avoidance based on APF

Artificial potential fields based method belongs to the classical approaches used in mobile robot path planning. This technique is based on the concept of potential field in physics, which regards the movement of an object influenced by two types of forces: attractive and repulsive.

According to [18], it stands out of the other classical approaches, thanks to some positive aspects. First among all, its simplicity. Indeed, it requires little information, low computational effort and it is easy to implement. Actually, unlike others classical approaches, such as cell decomposition and roadmaps, it does not require an explicit representation of the environment [14]. This feature makes this approach faster and valid in dynamical environments for real-time navigation too. Furthermore, deciding motion instant by instant, not doing a division of the map, it performs smoother trajectories compared to the others.

This technique was firstly introduced by O. Khatib [19] and consists of considering the mobile robot as a particle influenced by a potential field in which it is submerged. This potential field is created by obstacles and

target position, imaginary acting as charged surfaces that respectively depict repulsive and attractive potentials. In this way, the repulsive force accounted to the obstacle, results in pushing the robot away from it. Simultaneously, the attractive force assigned to the goal, pulls the robot towards it.

Thanks to the potential of this approach, after this first formulation, significant attention has been payed on improving different aspects of the APF technique. One aspect that has been accounted is the control law. In [19] the robot moved in its working environment thanks to the virtual forces resulting from the total potential field. Thus, the command law used to be the force to impress to the robot, computed as the negative gradient. Later, new techniques were experienced. These works focused on exploiting the gradient in a different way. For example, in [20] and [21] the command is considered in terms of velocity vector. In the first example, the command vector is directly obtained in function of the distance from the obstacle. Whereas in the second, the gradient is chosen according to the direction of the negative gradient, pointing at the target. For this dissertation, this last technique is chosen, according to the one presented in [16]. This approach is named gradient tracking since it performs a perfect tracking of the gradient lines. The commands are given in terms of linear velocity $v$ and angular velocity $\omega$. The first is chosen proportional to the angular error $\varphi$ between the desired and the current direction. In 2.1 the relation for $w$ is given.

$$\omega = K\varphi \tag{2.1}$$

where K is the proportional gain.

Assuming that speed at starting and final point are 0, the magnitude of the linear velocity is computed according to the relation 2.2.

$$v = min(a_0 t, v_0, (2a_0 d_r(t))^{\frac{1}{2}}) \tag{2.2}$$

where $a_0$ is the maximum acceleration, $v_0$ the maximum velocity and $d_r$ the position error. This last according to $d_r(t) = ||x_d - x_r(t)||$, where $x_r$ is the position at time t and $x_d$ is the target position, hereafter also intended as the the global attractor.

Summarizing, in the classic version, the artificial potential field that affects the motion of the robot is given by the sum of two fields, as reported in 2.3:

$$U_{art}(x) = U_{x_d}(x) + U_o(x) \tag{2.3}$$

where $U_{x_d}(x)$ represents the global attractive potential field in $x$ and $U_o(t)$ is the repulsive potential field in $x$. In the following sections it is analyzed how the attractive and repulsive potential fields are modelled.

### 2.1.1 Design of the attractor

As previously stated, the desired position can be seen as an attractive potential field. The goal is to attract the mobile robot towards target position. Aiming at doing that, in [19], the attractive field is modelled as a quadratic function. Thus, the field results:

$$U_{x_d}(x) = \frac{1}{2}\sigma||x - x_d||^2 \tag{2.4}$$

where $\sigma$ is a positive fixed parameter, used to specify the intensity of the field, $x_d$ is the target position and $x$ is the only variable of $U_{x_d}(x)$, representing the generic position of the robot.

Figure 2.1 shows an example of an attractive potential field, modelled as described above, considering $\sigma = 0.5$ and the target destination positioned in $x_d = [2,0,0]$.

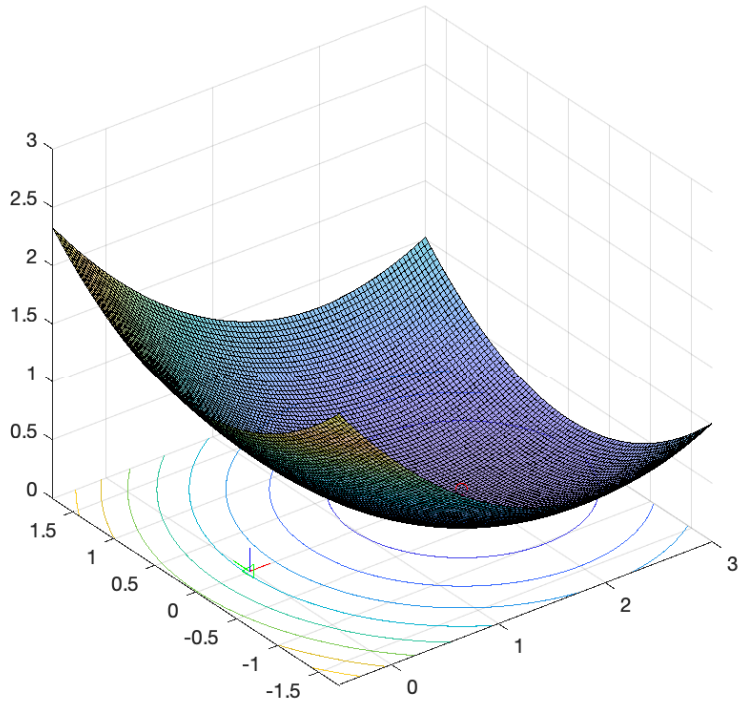



**Figure 2.1:** Example of attractive target potential field

Since the robot is programmed to follow the negative gradient, its motion derives from the computation of it:

$$\frac{\partial}{\partial x} U_{x_d}(x) = x - x_d \tag{2.5}$$

From 2.5, it can be understood that the quadratic potential attracts the robot in the direction of $x_d$ [22].

### 2.1.2 Design of the repulsor

As mentioned before, an obstacle can be seen as a repulsive potential field. However, compared to the attractive one modelled above, it is subjected to some additional requirements.

Firstly, it should be defined a region enveloping the obstacle, outside of which the gradient results 0, to not disturb robot's trajectory. Concurrently in this region, as the robot approaches the obstacle, the value of the potential field should increase. In this way, the robot would be repelled [19].

Moreover, differently from the target point, it is necessary to consider the geometry of the obstacle. Since objects can be difficult to model, a simplification can be done, considering simple shapes such as spheres or cylinders. In the case presented, the obstacle is supposed enveloped into a disc of radius $R_o$, centred in $x_o$.

In order to meet all the requirements, the repulsive field can be modelled as an exponential function [19] [22]:

$$U_o(x) = \beta_o e^{-\frac{\gamma_o}{2}||x-x_o||^2} \tag{2.6}$$

where $x_o$ is the obstacle position and $\beta_o$ and $\gamma_o$ are positive parameters, used to specify the strength of the repulsor. In particular, the final two respectively represent the peak value and the exponential decay. Similarly to the attractive field, $x$ is the only variable of $U_{x_o}(x)$, accounting the position of the robot.

The gradient of 2.6 repelling the robot, results:

$$\frac{\partial}{\partial x} U_{x_o}(x) = -\beta_o \gamma_o (x - x_o) e^{-\frac{\gamma_o}{2}||x-x_o||^2} \tag{2.7}$$

Figure 2.2 shows an example of a repulsive potential field considering $\gamma_o = 80.3475$, $\beta_o = 1$ and the obstacle centred in $x_d = [1,0,0]$.

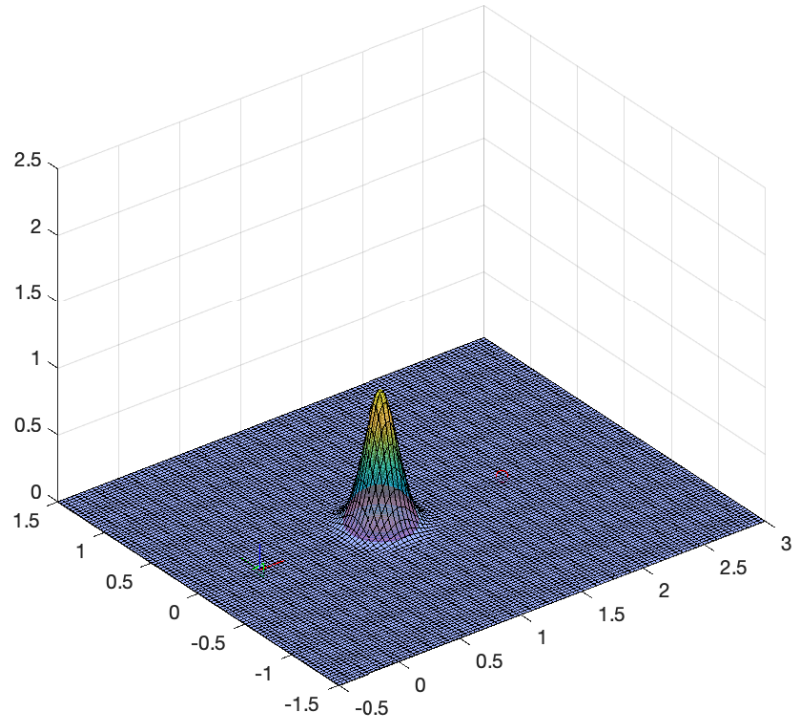



**Figure 2.2:** Example of obstacle potential field

### 2.1.3 Resulting APF

Resuming equation 2.3 and combining the equations outlined above in 2.4 and 2.6, the resulting artificial potential field is:

$$U_{art}(x) = \frac{1}{2}\sigma||x - x_d||^2 + \beta_o e^{-\frac{\gamma_o}{2}||x-x_o||^2} \tag{2.8}$$

Figure 2.3 shows the resulting artificial potential field 2.8, derived from the sum of the attractor and the repulsor depicted in 2.1 and 2.2.

Figure 2.4 shows the gradient lines related to the APF in 2.3. In addition, the obstacle center and the target point, as well as two circles around the obstacle. The inner circle represents the obstacle contour, while the outer circle identifies the limit where the potential field related to the obstacle goes to zero. Notice that, if the gradient tracking method is chosen as the control law, i.e. if the robot moves following the direction of the gradient, the gradient lines in Figure 2.4 can be used to figure out the robot path.

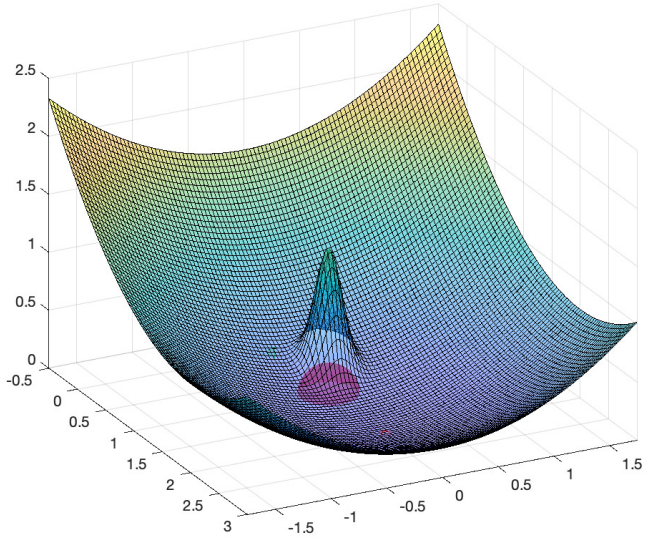



**Figure 2.3:** Example of the resulting APF

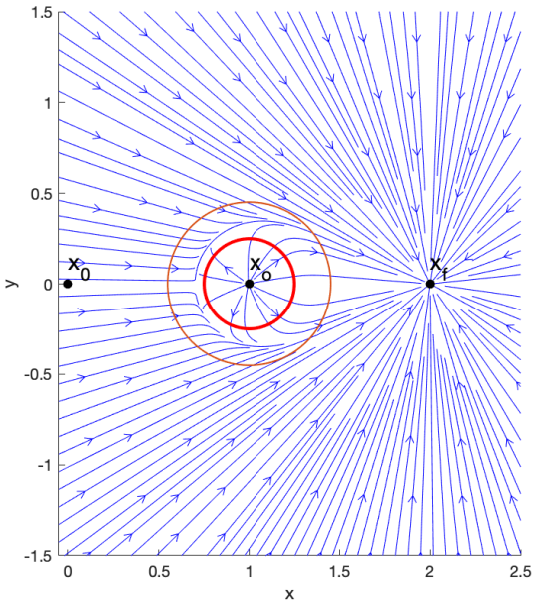



**Figure 2.4:** Gradient lines of the APF generated

### 2.1.4 Analysis of the APF method

One first issue related to the APF method, regards the control law and the possibility for the mobile robot to be stuck in a saddle point. This occurs especially when the control law is chosen according to the gradient tracking error and starting point, obstacle centre and target point are aligned, as in the example in Figure 2.3. However, in real world, the robot always strays a bit from the correct direction to take. Consequently, this condition does not occur and this is just a limit case.

A second aspect, is related to some basic capabilities that, according to [23], are required to mobile robots. First of all, reliability, safety and easiness are requested, but already satisfied by the chosen APF method. This was introduced as an offline approach, but converted in an online method, to perform in a real-time context that aims at fulfilling these goals. Moreover, the other requirements are related to the way the robot should behave preventing collisions. Obstacles' motion are unpredictable, especially if humans are considered in the environment, since they change speed and direction arbitrarily [13]. From humans point of view, robots are unpredictable too. Actually, mobile robot's movements, subjected to APF, depend on local conditions and directions of velocity, instant by instant. So, in similar conditions, minimal deflections of mobile robot's motion reflects in running different paths. This means that, obstacle will be overcame on right or left side arbitrarily. Aiming at improving this aspect, in the following section a customized APF method is described.

## 2.2 APF with local attractors

Following the suggestions pointed out in Section 2.1.4, a novel collision avoidance technique was presented in [16] [17]. It is named Multiple Attractors Potential (MAP) and its main goal is to affect mobile robot's motion, increasing security perception of human workers. For instance, the human can know a priori that, no matter of the robot approaching direction, it will avoid him on a certain side. This approach was designed for humans, but could be generalized to any type of obstacle. For example, an obstacle can be an object which has a preferential direction for collision avoidance. Consequently, applying this approach would be a great idea. So, the robot has the same behaviour with respect of the obstacle, imposing preferred collision avoidance directions.

The aim of this approach is perceived in MAP introducing local attractors, conditioning the robot to pass on preferred areas. Since the resultant movements of the robot depend on the local conditions, adding a local attractor would bend the path towards the attractor side.

Consequently, 2.3 must be updated adding a local attractor:

$$U_{art}(x) = U_{x_d}(x) + U_o(x) + U_a(x) \tag{2.9}$$

where $U_a(x)$ represents the local attractive potential field in x. The following section reports details about the design of the local attractor.

### 2.2.1 Design of the local attractor

The local attractor is arbitrarily positioned near the obstacle, on a side depending on where the mobile robot shall pass. To obtain the desired behaviour, the action performed by the local attractor should be similar to the one carried out by the repulsor, but opposite. Analogously, the attractive action should be intense, but limited to a region, outside of which it does not influence. Therefore, this local attractive field can be modelled as a negative exponential function:

$$U_a(x) = -\alpha_a e^{-\frac{\gamma_a}{2}||x-x_a||^2} \tag{2.10}$$

where $x_a$ is the center of the local attractive source and $\alpha_a$ and $\gamma_a$ are positive parameters, used to specify the strength of the attractor. In particular, the final two respectively represent the intensity and the exponential decay. Similarly to the repulsive and the global attractive field, $x$ is the only variable of $U_{x_a}(x)$, accounting the position of the robot.

The gradient of 2.10, attracting the robot in its direction, can be computed as follows:

$$\frac{\partial}{\partial x} U_{x_a}(x) = -\alpha_a \gamma_a (x - x_a) e^{-\frac{\gamma_a}{2}||x-x_a||^2} \tag{2.11}$$

Figure 2.5 shows an example of a local attractive potential field obtained considering $\alpha_a = 0.2770$ and $\gamma_a = 17.5406$. The local attractive source is placed 215° from the center of the obstacle in a counterclockwise direction, so in $x_a = [0.3856, -0.4302, 0]$.

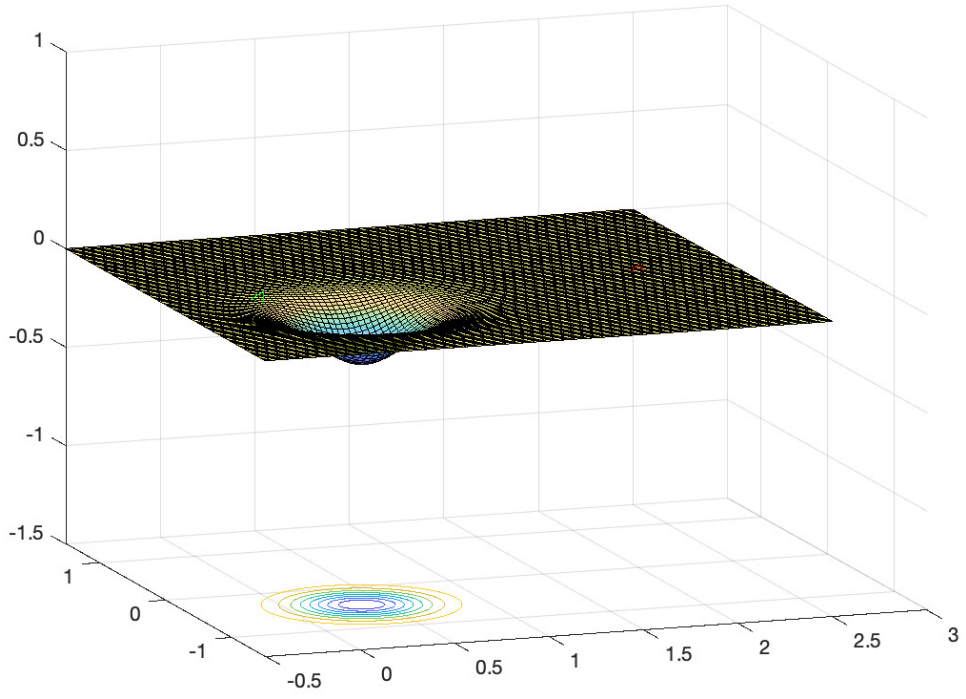



**Figure 2.5:** Example of local attractive potential field

### 2.2.2 Resulting APF

From the result achieved in the previous section, the resulting artificial potential field 2.9 is equal to:

$$U_{art}(x) = \frac{1}{2}\sigma||x - x_d||^2 + \beta_o e^{-\frac{\gamma_o}{2}||x-x_o||^2} - \alpha_a e^{-\frac{\gamma_a}{2}||x-x_a||^2} \tag{2.12}$$

Figure 2.6 shows the resulting artificial potential field 2.12, derived from the combination of the artificial potential field depicted in 2.3 and the local attractor in 2.5, introduced in this section.

Figure 2.7 shows the gradient lines related to the APF in 2.6, indicating the starting point, the obstacle and the local attractor's center and the target point. The red circles are related to the obstacle. The green ones to the attractive region. In both, the outer one, represents the active region outside which its influence is null.

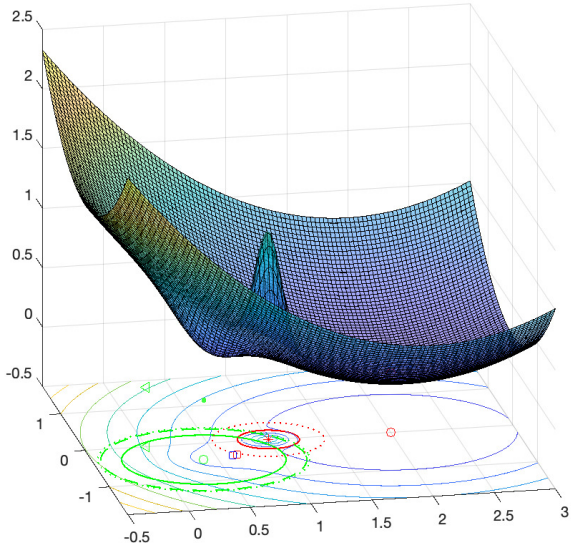



**Figure 2.6:** Gradient lines of the APF generated

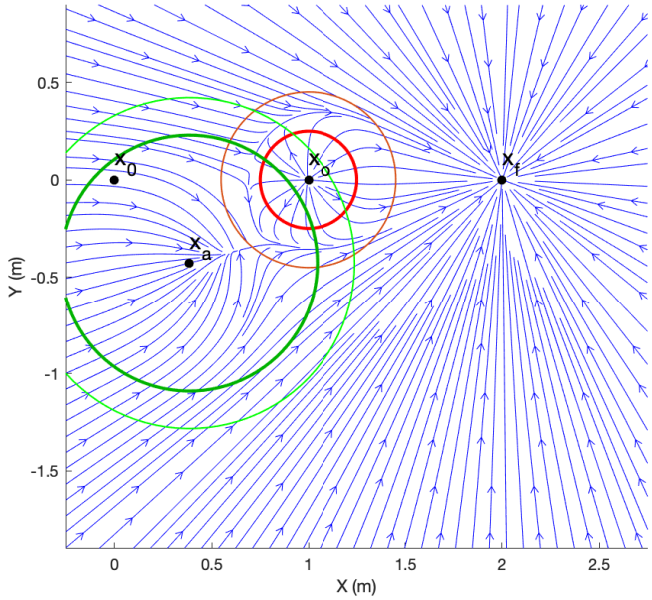



**Figure 2.7:** Gradient lines of the APF generated

### 2.2.3 Analysis of the APF method with local attractors

In this approach, the position and the size of the local attractor are subjected to some constraints. First of all, in order to obtain a simplification of the local minimum problem that will be pointed later, the local attractor must be positioned sufficiently far from the active region of the obstacle. In [16] this constraint is underlined with the relation 2.13:

$$||x_a - x_o|| > R_o^* + \widetilde{\epsilon} \tag{2.13}$$

where $R_o^*$ is the radius of the active region of the obstacle and $\widetilde{\epsilon}$ is a positive quantity that depends on the line segment $\overline{x_a x_d}$. If this segment does not intersect the active region of the obstacle, $\widetilde{\epsilon} = 0$. Otherwise $\widetilde{\epsilon} = \epsilon$, where $\epsilon = x_a - \widetilde{x}$ and $\widetilde{x}$ would be the the point were the saddle would occur [16].

Secondly, the attractor must not incorporate the target in its attractive region. Otherwise, the global minimum would be perturbed. In [16] this constraint is expressed with the relation 2.14:

$$||x_a - x_d|| \geqslant R_a^* \tag{2.14}$$

where $R_a^*$ is the radius of the active region of the attractive source.

Furthermore, it is necessary to pay attention to another possible shortcoming. In APF methods the robot is attracted to the target, exploiting the global minimum of the attractive potential field. Inserting a local attractor, a potential local minimum is added to system too. This would reflect in the possibility to be stuck in an equilibrium point. Therefore, $\alpha_a$ and $\gamma_a$ must be appropriately tuned, in order to not generate a local minimum.

As pointed out before, 2.13 simplifies the problem of the local minimum. Thanks to this constraint, the eventual local minimum would appear in a region where the influence of the obstacle is null. Thus, only the two attractors must be accounted in the analysis of the global minimum. In [16] the influence of $\alpha_a$ is studied with respect to the local minimum problem. From this analysis, derives that no local stationary point occurs if $\alpha_a < \widetilde{\alpha}_a$. Therefore, $\widetilde{\alpha}_a$ represents the upper bound of $\alpha_a$. Consequently, choosing $\alpha_a$ just below $\widetilde{\alpha}_a$, the maximum attraction is obtained.

In 2.16 is reported the equation computing the value of $\widetilde{\alpha}_a$, obtained

substituting the values of 2.15.

$$\begin{aligned} \widetilde{x}'(x'_a, \gamma_a) &= \frac{2}{3} x'_a [cos(\frac{\theta + 4\pi}{3}) + 1] \\ \theta(x'_a, \gamma_a) &= cos^{-1}(\frac{27}{2\gamma_a x'^2_a} - 1) \end{aligned} \tag{2.15}$$

$$\widetilde{\alpha}_a(\sigma, x'_a, \gamma_a) = \frac{-\sigma \widetilde{x}'}{\gamma_a (\widetilde{x}' - x'_a) e^{-\frac{\gamma_a}{2}(\widetilde{x}' - x'_a)^2)}} \tag{2.16}$$

where $x'_a$, $\sigma$, $\gamma_a$ are fixed parameters. $\sigma$ and $\gamma_a$ are respectively the same of 2.4 and 2.10. $x'_a$ instead, represents the position of thee local attractor in the frame centred in $x_d$ and whose $x'$ axis is aligned with $x_a$. In addition to this, $\widetilde{x}'$ represents the point where the inflection arises, expressed in the same frame of $x'_a$, i.e. where the saddle point should be.

Undergoing the limit imposed by 2.16, only a soft deflection is generated. Thus, no problems related to local minimum are faced, as indicated by the equipotential contours in Figure 2.6.

# Chapter 3

# Software and Hardware tools

This chapter intends to give a detailed description of the software and hardware tools exploited during the thesis. In the next chapter that focuses on implementation, tools will be only cited and for any further description this chapter will be the reference.

Firstly a general overview of the chosen programming environment is given. Together with this description, the reasons that led to this choice are explained, analyzing pros and cons. Then, all the tools used are presented, giving a general idea of where they were exploited. This aspect is deeply explored in the next chapter.

## 3.1 Robot Operating System

ROS (Robot Operating System) is a project started by a team of researchers at Stanford University in 2007. It was took over from the company Willow Garage and it has developed a lot over the last years, releasing different distributions of it.

ROS is defined as *an open source software development kit for robotics applications. ROS offers a standard software platform to developers across industries that will carry them from research and prototyping all the way through to deployment and production* [24]. Actually, the peculiarity of ROS is that is an open-source platform and this feature reflects in many aspects. First of all, it means that it is accessible either to experienced

user or newcomers. Anyone interested can explore this world, learning with official tutorials and developing its own project. This, without the necessity to purchase anything. Secondly, it provides many libraries and tools for programming a robot. Everything under the same environment, not like the software property of manufacturing companies. Lastly, a large amount of code is shared by the research community, promoting reusability. This last assumption means that time spent developing basic operation can be saved.

For all this reasons, many robots have already integrated ROS. An example in the research field are Turtlebots [25], mobile robots equipped of sensors based on ROS. However, it is not only used in this sector, but it is emerging in industry too. Many commercially available industrial robot are partially implemented using ROS, others can be integrated using it. Thus, ROS represents the standard of robot programming today.

In order to integrate developed functions, ROS offers packages. Packages represent the way ROS is organized. They enclose in a folder all what is needed to run tasks available in it: code, executables, libraries and configuration files. They are responsible of small tasks, according to a modular structure endorsed by ROS. Actually, ROS implements the "divide and conquer" paradigm: small parts of code are tested and combined to pursue more advanced aims.

Concerning the way it works, ROS is organized in nodes and topics. This structure can be easily represented by means of a graph where nodes are vertices and edges, topics.

Nodes are pieces of software performing a specific task. In order to start a process, a ROS master node must run. This node keeps running until the end of the process and plays a supervisory role, monitoring the information exchanging between the others. This communication can occur through topics or services. However, for the aim of the thesis, only topics are used.

Topics can execute two kind of actions: publishing or subscribing. Publishers nodes, after creating a channel of communication registering to the ROS master, start publishing messages on it. Subscribers nodes interested in listening to one topic, register to it through the ROS master, and then receive messages. A third type of node exists and it integrates the two actions.

The way topics exchange messages is asynchronous. Concerning the messages swapped, they are characterized by specific types. Consequently, nodes' subscription/publication should be consistent in terms of type.

The fundamental concepts related to how a ROS process works are all shown in Figure 3.1.

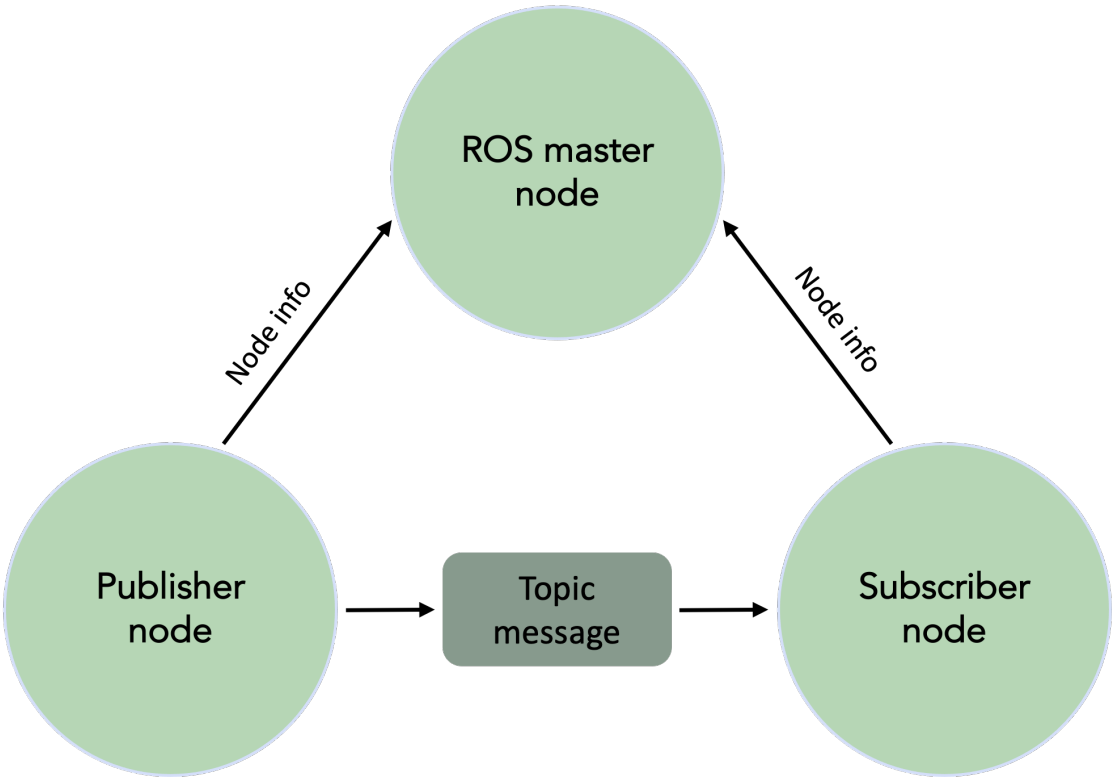



**Figure 3.1:** Communication in ROS

Regarding the way a developer interacts with ROS, it is almost through command line on UNIX systems. Commonly ROS is installed on Linux Ubuntu, paying attention to the fact that each distribution is supported on a specific release of the OS. The choice of the distribution, meaning the working setup, mainly relies on the target application which requires specific versions too.

Basically, a process in ROS starts with the launch of the ROS master, obtained executing the command *roscore*, with prior appropriate network configuration in terms of IP address and port number. Then, nodes are launched with the command *rosrun* or *roslaunch*. The former is used to run a single node in the package. The latter executes multiple nodes. The code executed can be expressed in one of the programming languages supported by ROS. Python and C++ are the most used. In the next section Python will be deeply presented in the context of ROS, along with the considerations that led to this choice.

### 3.1.1 Programming ROS

In 1.1 the essential role of computer science in autonomous applications has been explained. Therefore, a critical analysis of pros and cons of different programming languages used in this application is worth-mentioning.

This section starts analyzing the way the application of MAP was done in [16]. On the basis of this remarks, section 3.1.1 explains the choice of using Python as programming language for this work.

#### Remarks on MAP

In [16] an application of the MAP technique is outlined. In this test, the gradient procedure was computed in Matlab. Afterwards, this setup was tested on the simulation setting Gazebo, available in ROS. There an accessible emulation of the mobile robot Turtlebot was exploited.

To pursue the aim of this thesis, the usage of ROS (Robot Operating Systems) is imperative. This because ROS offers powerful tools for both simulations and real robots. For instance, MAP in [16] is simulated using a Turtlebot and this model will be used also in the real implementation, goal of this dissertation. Thus, using the same model will give the possibility to exploit results obtained in simulation. Anyway, ROS supports different languages to perform the desired task. Python, Matlab and C++ are examples. Deciding which language to use, ROS can be considered as a compulsory background. Therefore, the analysis should not only be focus on the advantages of each programming language, but also on the way they interact with ROS.

Actually, in [16] the heart of the algorithm was run on Matlab. ROS was only exploited for the simulation tool, using the ROS communication bridge available in Matlab ROS toolbox. Matlab is a powerful tool arising for two main features: interactive command line and aesthetically pleasing and simple graphical representation of data. Regarding the possibility to have interactive sessions, it is something that can be useful to test small parts of code. And this is somehow linked to the second assumption. As a matter of fact, implementing an algorithm, it is useful to enhance performances analyzing data through plots in a trial and error procedure. As an example, in [16] a tuning of parameters was performed to obtain a good setup for the simulation. Thus, these features make Matlab a good choice for the experimental phase.

However, Matlab is not free and open, while ROS it is. Even if it has the advantage of running on all OS, Matlab needs licenses. In addition to the limited availability, this reflects in a lack of open source code too.

In conclusion, Matlab is a good choice for the experimental phase where the focus is studying a new approach. There, a tuning of parameters and analysis of plot is essential to obtain good results, as it was done in [16]. However, aiming at real-world implementations, subject of this dissertation, a good idea can be to move to a free and object-oriented programming language, such as C++ or Python. Actually, in this next step the setup can be based on the results obtained in simulation. Moreover this assumption is supported by another aspect. In [16] is stated that the simulation run with a control frequency of 30 Hz. Using a different programming language this could be easily increased to 60 Hz and this aspect is something to consider to obtain a good control on a real scenario.

### Programming languages comparison

ROS supports different types of programming languages. The most used are C++, Python and Matlab. Others are just partially implemented, such as Octave and Lisp. In this section the main ones are compared, in order to explain why Python was chosen to pursue the aim of the thesis.

First of all, Python and C++ are based on the open source paradigm. This means that there is a larger sharing of code. Thus, more advanced applications can be achieved in a faster way, reusing code. This ideology fits well with the one promoted by ROS. On the contrary, Matlab is not free and open, but despite the others, it offers a more friendly interface that can be something preferable for some users. However, as stated in 3.1.1, using Matlab could be a good idea for the experimental phase when data analysis and plotting are needed. In the implementation step C++ and Python are better choices.

One thing to know about ROS is that it can be defined as an agnostic language. It does not matter if one node is written in C++ and one in Python, they rely on a lower layer. This means that communication is always allowed, not depending on a specific language. Therefore, if some tasks would benefit of a Python implementation and others of C++, there will not be any problem.

One first thing that makes the difference between Python and C++ is related to the tools needed. Actually, even if most libraries are available on

both, some can be fully developed only on one of them. However, in the case considered here all the implementations required were available on both.

Both Python and C++ are object-oriented programming languages. The first represents an extension of C, while the second started from scratch. What really makes the difference among the two depends on the purpose of the project. C++ is a compiled and intermediate-level programming language. This means that it performs faster. Python instead is an high-level programming language and it is considered easier than C++, by means of writing code, since it takes more things for granted. However, it is an interpreted language and that stands for lower performances. If the goal is something that has to be applied in industry, times matters and C++ could be a better choice. Indeed, for research purposes Python is perfect. Actually, it is faster to build a prototype, allowing testing many new solutions. In a few words, more flexible. This is the reason that led to choosing Python in this thesis.

## 3.2 TurtleBot

TurtleBot is a ROS standard platform robot [25]. Its name derives from the strict relation with ROS, whose logo is a turtle. Indeed, it was created in 2010 by the same company that took over ROS at the beginning, Willow Garage. Their goal was to create a small, affordable, programmable, ROS-based mobile robot kit, that would increase the market of ROS. Today Turtlebots are commonly used in research and educational field. Moreover, thanks to the low prices, they are often used just as a hobby.

Thanks to their strict relation with ROS, this software offers many packages with tools useful when dealing with TurtleBots. First and foremost, these robots are modelled and can be exploited in simulation environment provided by the software, such as Gazebo.

Until today four versions of TurtleBot have been released. The first two, TurtleBot1 and TurtleBot2, were developed on the basis of research robots. The first was based on iRobot Roomba, the second on iClebo Kobuki. Then, TurtleBot3 took a big step forward, collaborating with ROBOTIS and Open Source Robotics Foundation. Finally, TurtleBot4 has just been released, promising better computing power and better sensors, compared to the previous. These last two versions are the only ones still on the market.

For this thesis, a model of the family of TurtleBot3 is used. The reason is

mainly related to the fact that the aim is to test the approach on a simple platform, and in the future apply it to a industrial application or with any other mobile robots on the market. In 3.2.1 a general overview of this version is given, as well as a description of the way it communicates with ROS.

### 3.2.1 TurtleBot3

The real innovation in TurtleBot was brought by TurtleBot3, developed in 2017. The aim of this version was to correct the defects of its predecessors and meet the demands of users.

In Turtlebot3 the main change is accounted to the adoption of ROBOTIS modular smart actuator Dynamixel. In addition to this, this robot is provided of a Single Board Computer (SBC), Rasberry Pi, and an embedded controller developed for ROS, OpenCR. Regarding sensors, it is provided of a 360° Laser Sensor, LiDAR. Furthermore, Turtlebot3 gives the possibility to customize the robot in various ways, depending on what is needed for the implementation wanted.

This robot can be applied for different purposes. Actually, its core-technology are:

- Navigation: it regards the ability to perform localization together with path planning. Localization consists in knowing its own position and orientation, while path planning concerns the schedule of a path to reach a destination.
- SLAM (Simultaneous Localization and Mapping): it is related to the ability of the mobile robot to locate in a unknown area. This is made possible through a mapping of an unknown area, thanks to the sensors it is provided of.
- Manipulation: it regards the possibility to manipulate objects, integrating the mobile robot with a manipulator.

However, being customizable, programmable and open-source, it offers the possibility to pursue different aims. It can be combined with technologies not naively-implemented on it, such as machine learning or computer vision.

TurtleBot3 represents a family of robots. Three models are available in it: Burger, Waffle and Waffle Pi. For this thesis work, a burger type is used, thanks to its similarity with commercial mobile robots. This feature fits well with the aim of this work: testing an algorithm to develop on mobile robots

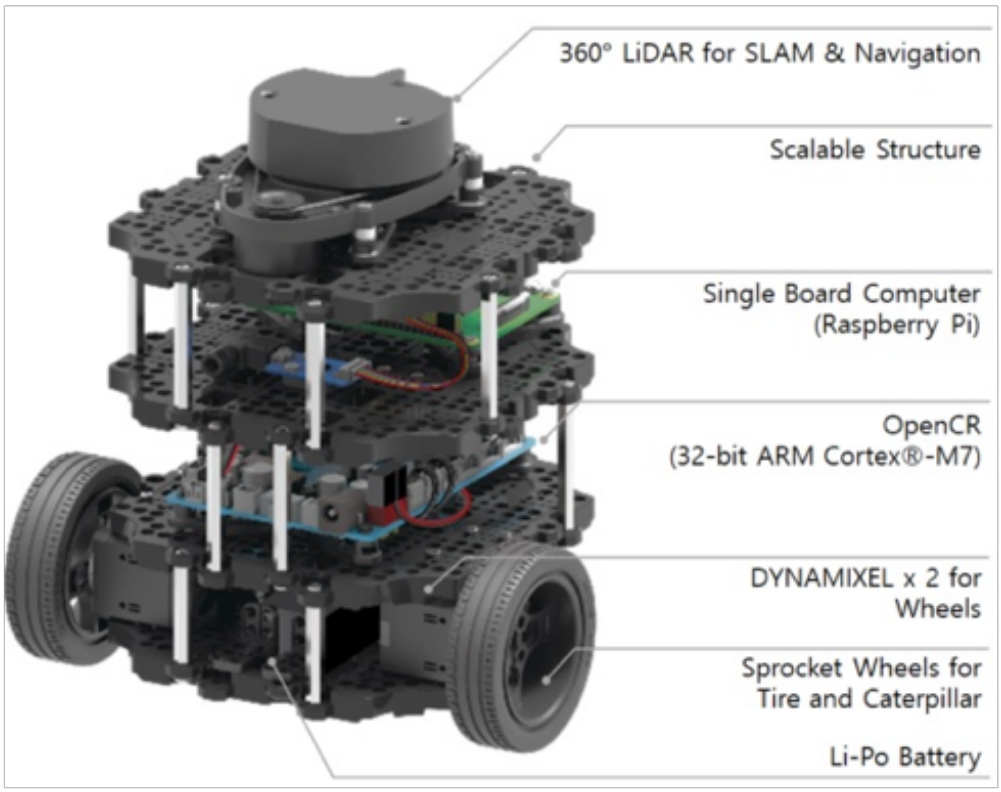



**Figure 3.2:** TurtleBot3 Burger [25]

in the future. Actually, this is the main reason that makes this model popular The other two models instead are often applied together with manipulators, since they are already set up for this.

TurtleBot3 burger is classified as a differential-drive wheeled mobile robot, provided of two wheels. Figure 3.2 shows this model, indicating the main components.

One additional specification that can be of interest, is that it can reach a maximum translational speed of 0.22 m/s and a maximum rotational velocity of 2.84 rad/s.

In order to use this mobile robot, a channel of communication should be created. In this work, TurtleBot burger communicates with ROS using

some API available in Python, which interface with ROS topics. This communication is shown in Figure 3.1. In 3.2.2 it will be deepened in the context of Python.

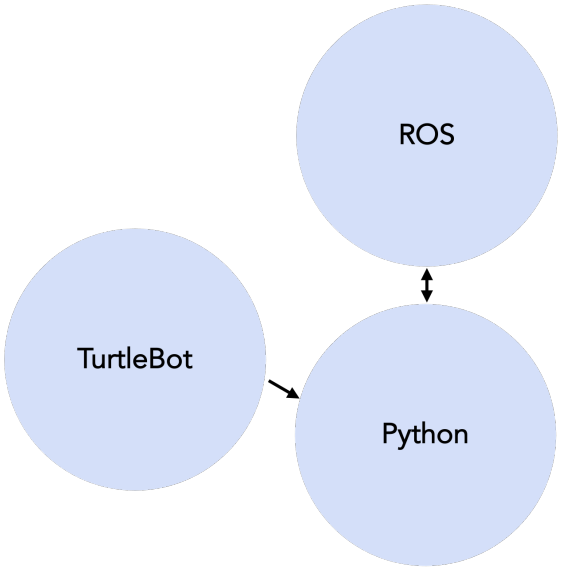



**Figure 3.3:** Communication TurtleBot3 - ROS

### 3.2.2 Communication between TurtleBot3 and ROS

TurtleBot communicates with ROS in the standard way, via topics. This communication is enabled by an interface in Python, *rospy*.

TurtleBot offers different types of topics that can be divided between subscribing and publishing ones [26]. Concerning the subscribing topics, the user sends the messages to the robot that receives and process. On the contrary, publishing topics give information received from sensors, such as the motor status or the position of the robot.

Concerning the implementation of the communication in Python, in *rospy* is created using two classes: *rospy.Publisher* and *rospy.Subscribers*. The one relative to the publisher, once initialized, publishes the command using the method *pub*. While the subscriber, once initialized, autonomously delivers the result recalling a callback.

In Figure 3.4 a graph reporting some of the possible messages exchanged when TurtleBot is applied.

Among the subscribing topics, "/cmd_vel" deserves attention. This topic

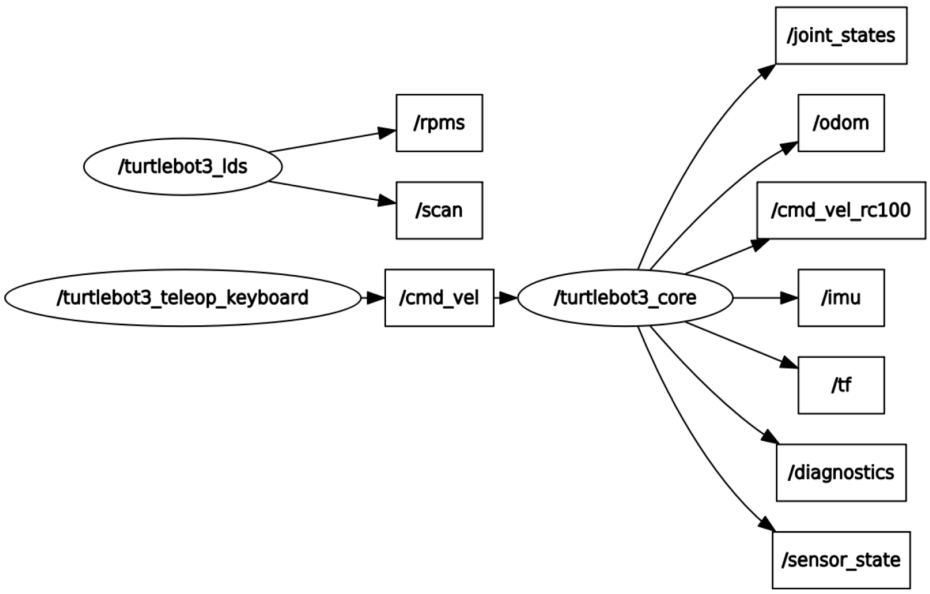



**Figure 3.4:** TurtleBot3 Topics [25]

allows the control of the robot. In this way, the user can control the translational and rotational speed of the robot in terms of m/s. This command belongs to the "geometry_msgs/Twist" type, defined by two vectors. One for linear and one for angular speed. However, due to nonholonomic constraints, some commands are forbidden. Consequently, only linear x and angular z are used.

"/Odom" topic instead belongs to the publishing ones. It is used to obtain odometry information that, in TurtleBot3, rely on the gyroscope and the encoder. These values are based on the recording of the driving information and are required to perceive navigation purposes, since they return the pose of the TurtleBot. This estimation is given by means of a message of type "geometry_msgs/Pose" that is divided in position and orientation, through a vector and a quaternion.

The two messages briefly described above are the one used in the designed application. In the first phase both are utilized, while in the second "/Odom" is substituted by a different approach. This relies on an OpenCV tool illustrated in 3.4.2.

## 3.3 Realsense d435

In the second part of the thesis, a camera is added to the system, mainly to consider a dynamical obstacle in the implementation. The reasons that led to this decision will be explained in the next chapter.

The chosen camera is a Intel® RealSense D435. It is a stereo camera that belongs to the D400 family. It can stream both RGB color data and depth information, thanks to the depth sensor. It is a low-cost, lightweight and powerful compact model that enables the development of applications that can deal with their surroundings. For all these reasons, it is widely used in robotic field for robotic navigation and object detection.

Figure 3.5 shows the main modules.

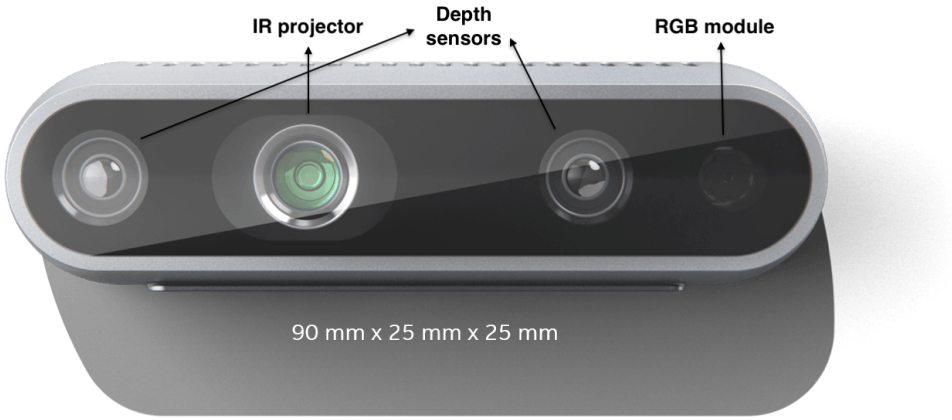



**Figure 3.5:** TurtleBot3 Topics [27]

For this dissertation, depth information is not required. It is used as a normal 2D camera provided of a RGB module. Thus, among all the specifications, the FOV (Field of View) of the RGB is of interest for this application.

FOV relies on camera lens, focal length and sensor size. It represents the maximum observable area that can be captured by the camera. The RGB sensor of RealSense D435 offers a FOV of 69° × 42° (h x v). This information will be used later when the laboratory setup will be described.

In order to understand better how FOV works, in Figure 3.6 is reported an illustration where all the parameters affecting this measure are reported. In this case $\alpha = 69°$ and $\beta = 42°$.

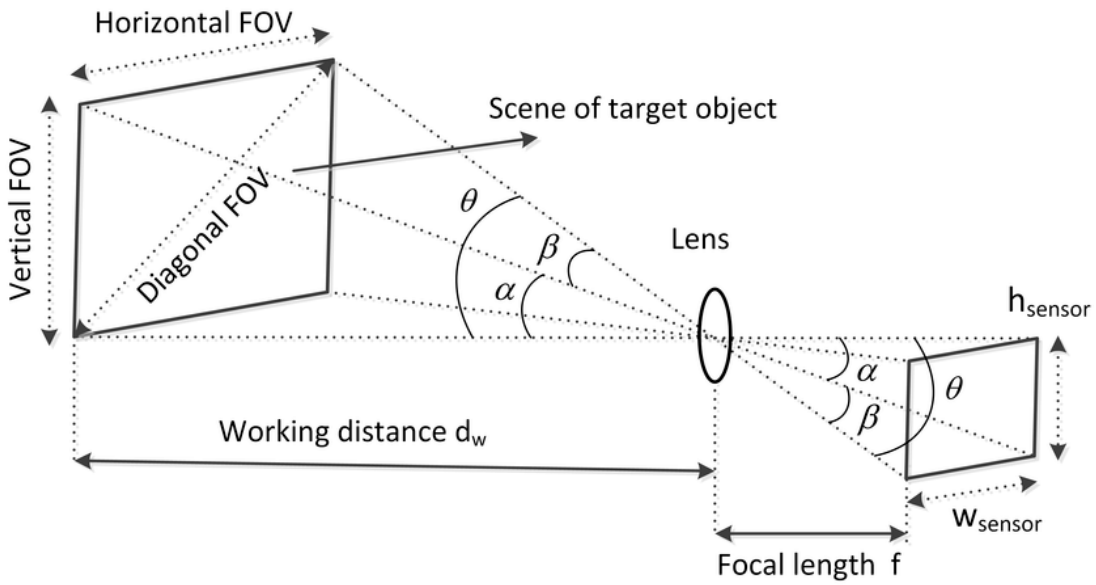



**Figure 3.6:** Illustration of camera lens's FOV [28]

For what concerns the communication, it is supported by a Python library that provides useful API. The library of interest is pyrealsense2. In this work, it is applied following the approach in [29], where the streaming is performed through a pipeline, properly initialized. In the code in 3.1 is reported the initialization which is done according to a loaded json file that reports the preset configuration of the camera. In this way, the configuration can be customized depending on what is needed.

**Listing 3.1:** Initialization of the pipeline for realsense

```
1
2  def init_realsense(self, name):
3      jsonObj = json.load(open(name))
4      pipeline = rs.pipeline()
5      config = rs.config()
6      config.enable_stream(rs.stream.color, int(jsonObj['viewer'][
       'stream-width']), int(jsonObj['viewer']['stream-height']),
       rs.format.bgr8, int(jsonObj['viewer']['stream-fps']))
7
8      cfg = pipeline.start(config)
9
10     return pipeline
11 }
```

Regarding the way it is used, a basic streaming is done according to the code in 3.2. Here, the recalled pipeline is the one initialized above. Then a visualization of the frame received from the pipeline is done. In addition to this, transformations or computations can be added to this code, before the image is shown. For instance, these can be done in order to get information about the surroundings or to process the image. OpenCV's libraries are usually exploited for this aim.

**Listing 3.2:** Basic code for streaming with realsense

```
[...]
while(True):
    frames = pipeline.wait_for_frames()
    color_frame = frames.get_color_frame()
    color_image = np.asanyarray(color_frame.get_data())
    [...]
    # here transformations of the frame can be made
    cv2.imshow("Image", color_image)
[...]
pipeline.stop()
```

## 3.4 ArUCo markers

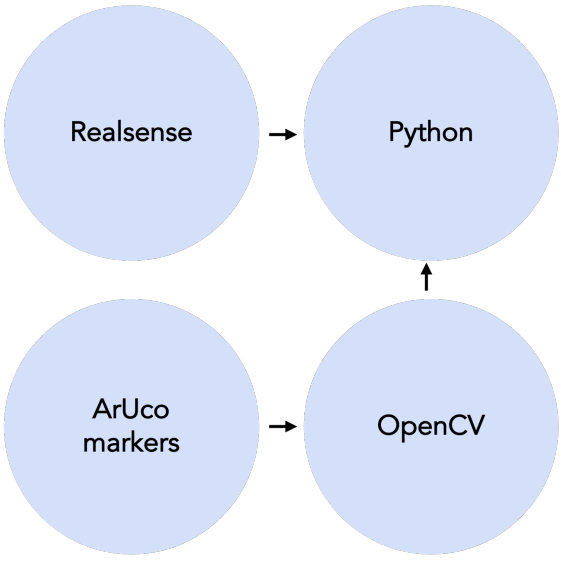



**Figure 3.7:** Connection between camera stream and markers

ArUco markers are computer vision tools available in the library OpenCV. They are used in the second phase of this work to substitute the odometry feedback. In order to exploit them, a camera is needed. Thus, a Python code should manage the stream of frames coming from the camera to elaborate them. This connection can be visualized better in Figure 3.7.

In this section OpenCv is presented, as well as ArUco markers.

### 3.4.1 OpenCV

OpenCV (Open Source Computer Vision Library) is an open-source library that provides useful tools for both computer vision and machine learning [30]. OpenCv is referred as a cross-platform since it interfaces with different programming languages and supports different OS. Python and Linux are one of them. It provides more than 2500 algorithms, which are highly optimized, since it focuses on real-time applications. It contains a wide collection of image processing methods that perform a manipulation of images, doing several types of transformations. Those include classical and state-of-the-art computer vision and machine learning algorithms, as well as more innovative. Examples of applications that can be developed with OpenCV are object identification, faces detection, and so on. Innovation is mainly intended in the way these solutions are provided. An example is deeply described in the 3.4.2.

In this work is exploited for two reasons, both related to the second phase of the implementation. In this phase, vision is added to the system and OpenCv represents the heart of the solution. The main support is related to the use of ArUco Markers, deeply presented in 3.4.2 and described in the context of the implementation in 4.3. In addition to this purpose, it also plays a supportive role for what concerns the streaming of the camera.

### 3.4.2 ArUco markers

#### ArUco library

ArUco is an open-source library of OpenCv, popular in computer vision. It is used both for detecting and pose-estimating squared planar markers. An AruCo marker is defined by OpenCV as a *synthetic square marker composed by a wide black border and an inner binary matrix which determines its identifier (id)* [31]. An example is shown in Figure 3.8.

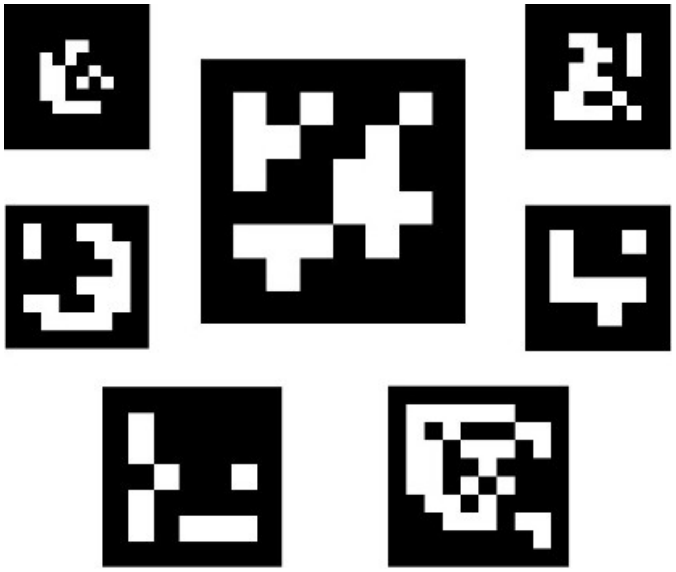


**Figure 3.8:** Examples of ArUco markers [31]

The inner region is the one that identifies the marker. It permits to determine unequivocally the pose of it in the environment in which is inserted. This is also done thanks to the black border that facilitates the detection.

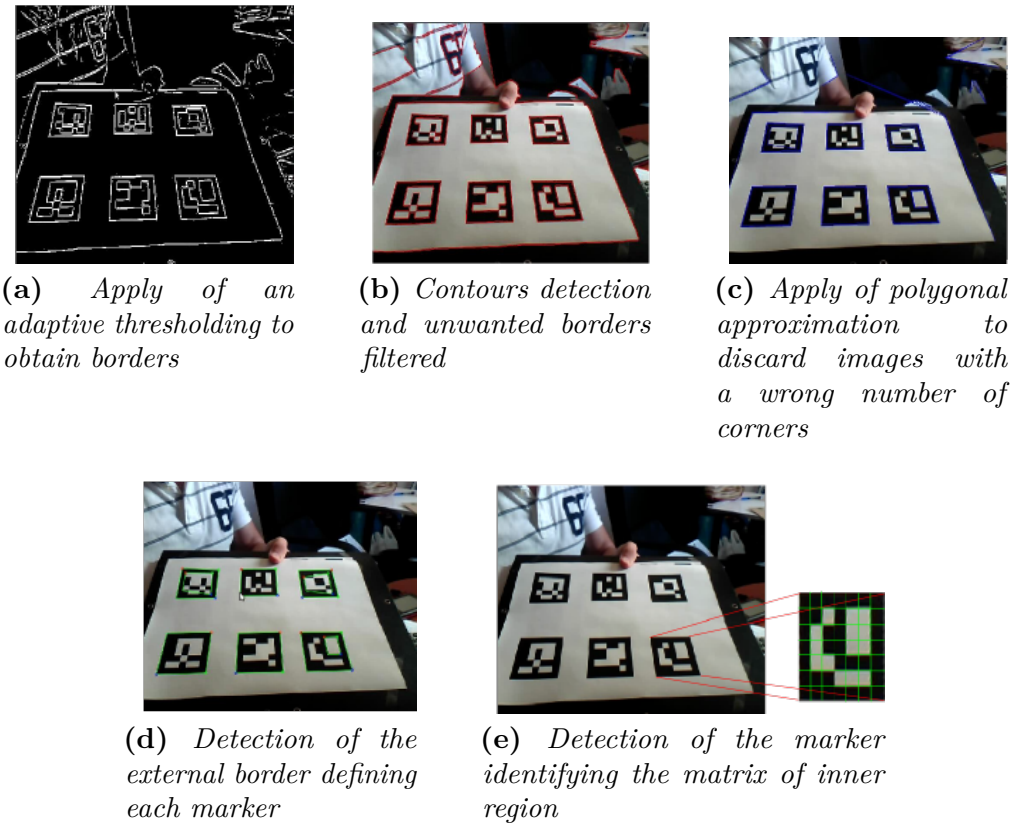
There are different types of markers, belonging to specific dictionaries. These are characterized by the dimension of the dictionary and the marker size. The former defines the number of markers included in it, the latter the size of the internal matrix. This last is also accounted as the number of bits of the markers and markers can have more or fewer bits. The more are the bits, the smaller is the chance of confusion. However, more bits means that more resolution is required too. Consequently, it should be find the correct trade off, depending on the application to develop.

### ArUco markers detection

A general frame containing ArUco markers can be elaborated in order to detect them. The returned result of this elaboration includes two information: the id of the marker and the position of the corners in the image.

The marker detection process of ArUco is done in two steps. In Figure 3.9 are shown the main elaboration done to get the result. Firstly, the image (or the frame if it is a video) is analyzed looking for possible markers. This elaboration of the imagine is done applying an adaptive thresholding to

obtain the borders of it. In this step, along with real markers, undesired contours are detected too. Thus, different filters are applied to discard unwanted borders.



**(a)** *Apply of an adaptive thresholding to obtain borders*

**(b)** *Contours detection and unwanted borders filtered*

**(c)** *Apply of polygonal approximation to discard images with a wrong number of corners*

**(d)** *Detection of the external border defining each marker*

**(e)** *Detection of the marker identifying the matrix of inner region*

**Figure 3.9:** Process of ArUco detection [21]

The second step consists in the markers identification. This is done starting by extracting the marker bits of each marker (the result of the first step 3.9e). To do so, a perspective transformation is firstly applied to obtain a frontal view. Then, this image is thresholded to underline the difference between white and black bits. Since the chosen dictionary is given, the image is divided in different cells according to the marker size and the border size. Finally, the bits are analyzed to determine if the marker belongs to the specific dictionary or not.

Concerning the implementation in Python, all this steps are done calling the method *detectMarkers()*. This requires in input the image to analyze and the dictionary used, and returns corners and respective ids detected.

However, in the case considered, detection is not enough. ArUco markers are exploited to obtain a estimation of the pose of an object. Therefore, it is necessary to perform camera pose estimation. This can be done only if a correct camera calibration is done. After the explanation of this next phase, a code gathering all this steps will be reported.

### Camera Calibration

In order to estimate the pose of a marker, calibration parameters of the camera must be known. These are required to perform a geometric transformation from the projected image plane to the 3D environment.

In particular, these parameters are the camera matrix and distortion coefficients. The first is related to intrinsics parameters, while the second to extrinsics. Intrinsics are specific of the camera applied. They concern the focal length of the camera lens ($f_x$, $f_y$) and the optical center of the sensor ($c_x$, $c_y$). The camera matrix is of the form:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

**Figure 3.10:** Camera matrix

Whereas, the distortion coefficients are related to 3D rotations and translations that transform the camera reference system to an arbitrary one. These coefficients model the lens distortion produced by the camera and use them to compensate it. Distortion coefficients are expressed through a vector of five elements.

To obtain these parameters, OpenCv offers the method *calibrateCamera()* that requires several images of a board, captured with the camera used. In addition to this, information about the size of the board and of board's squares is needed too. With this, it performs a calibration detecting board corners. In order to obtain a good result, at least 25 images should be provided. However, calibration is not part of the entire application. It should be done just once and then it is sufficient to save the parameters needed.

In Figure 3.11 an example of calibration is shown. Some of the images captured with the Realsense for a process of calibration are represented. In the figure these images are shown elaborated, after the call of the method *calibrateCamera()*. In order to have a correct calibration, these images must

not be redundant. These must show the board in different positions/orientations and must cover all the area of the camera to have information about all the field of view. In this way, correct parameters can be computed.

**Figure 3.11:** Example of calibration

The calibration parameters referred to the images are reported below:

$$K = \begin{bmatrix} 612.7908848 & 0 & 437.45801011 \\ 0 & 613.83484813 & 237.90178247 \\ 0 & 0 & 1 \end{bmatrix}$$

$$dist\_coef = \begin{bmatrix} 0.1439838 & -0.3959496 & -0.0003509 & 0.0010755 & 0.2730598 \end{bmatrix}$$

**Figure 3.12:** Calibration parameters of the example 3.11

### ArUco markers pose estimation

Once camera calibration parameters are available, a correct pose estimation can be performed. In this process, the relative pose of the camera with respect to the center of the marker is estimated.

In Python it is performed calling the function *estimatePoseSingleMarkers*. This requires in input the corners detected in the previous step, together with the size of the marker to be estimated and the calibration parameters. It returns two vectors: rvecs and tvecs. Each element of these correspond to the a specific marker. Rvecs are rotation vectors and define the orientation of each marker. Tvecs are translation vectors. Together, these vectors represent the camera pose with respect to a marker, i.e. the 3D transformation from the marker coordinate system to the camera coordinate system. Therefore, a change of reference frame can be done with these vectors, exploiting the homogeneous transformations.

At this point, all the elements required for the pose estimation have been explained. Therefore, a code gathering all the steps is reported in 3.3. In this application a realsense camera is supposed to be used.

**Listing 3.3:** Detection of ArUco markers

```
1
2 def realsensedetect(pipeline, cameraMatrix, distCoeffs,
    aruco_dictionary, parameters, aruco_dim):
3
4     while (True):
5         frames = pipeline.wait_for_frames()
6         color_frame = frames.get_color_frame()
7         color_image = np.asanyarray(color_frame.get_data())
8         gray_frame = cvtColor(color_image, cv2.COLOR_BGR2GRAY)
9
10        # Function 'cv2.aruco.detectMarkers() called'
11        corners, ids, rejectedImgPoints = detectMarkers(
    gray_frame, aruco_dictionary, parameters=parameters)
12
13        # Draw detected markers on the image:
14        color_image = drawDetectedMarkers(image=color_image,
    corners=corners, ids=ids, borderColor=(0, 255, 0))
15
16        # Draw rejected markers:
17        color_image = drawDetectedMarkers(image=color_image,
    corners=rejectedImgPoints, borderColor=(0, 0, 255))
18
```

```
19          # rvecs and tvecs are the rotation and translation
    vectors of each marker
20          rvecs, tvecs, _ = estimatePoseSingleMarkers(corners,
    self.aruco_dim, cameraMatrix, distCoeffs)
21
22          for rvec, tvec in zip(rvecs, tvecs):
23              drawAxis(color_image, cameraMatrix, distCoeffs, rvec
    , tvec, self.aruco_dim)
24
25          drawAxis(color_image, self.cameraMatrix, self.distCoeffs
    , self.rvecW, self.tvecW, self.aruco_dim)
26          imshow("Image", color_image)
27
28          return color_image
```

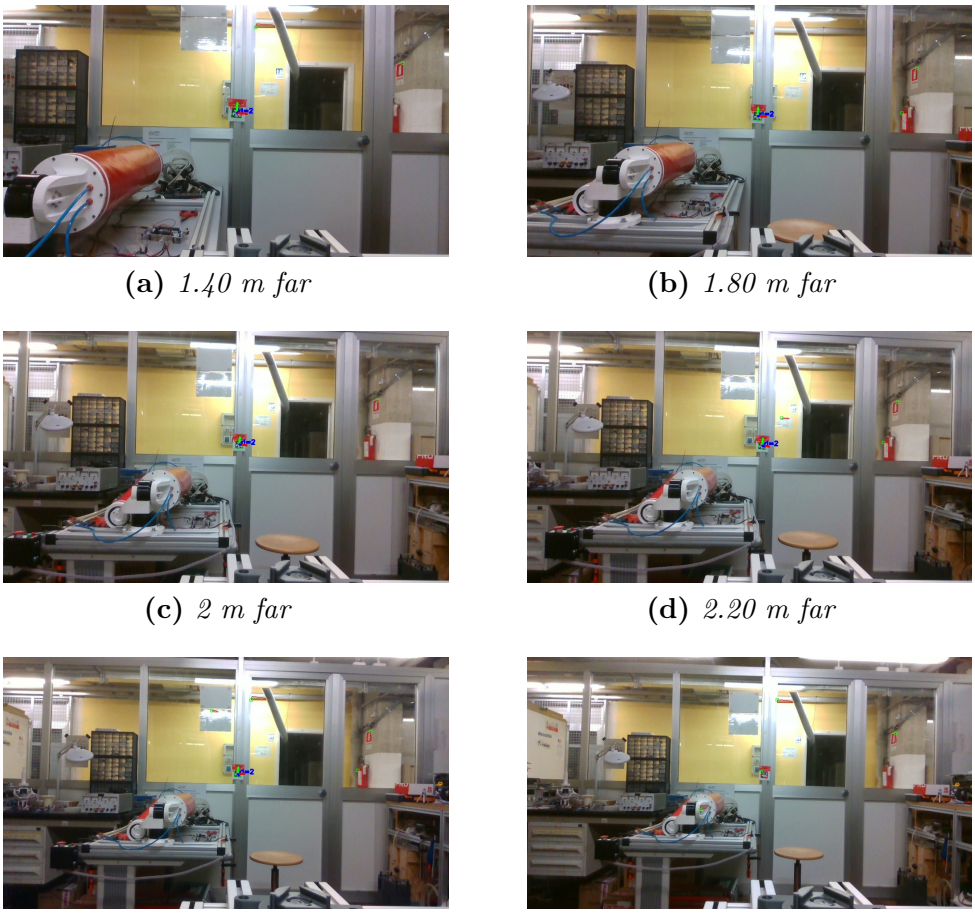
At this point, a general overview of the tools used during the implementation has been done. However, in order to exploit ArUco, some tests have been made. Just to be sure that they could have led to a good result.

### Test 1 on ArUco: Detection at different distances

One first test consisted in verifying if an ArUco would have been detected at a certain distance. To do so, an ArUco was attached to the wall and kept fixed. Concurrently, the camera was incrementally moved away from it. The size of the ArUco and the distance to consider was supposed on the basis of the target application. Details regarding how they were chosen, will be better explained in 4.1.

In Figure 3.13 are shown captures of the results relative to the size of the ArUco used in the implementation. As can be seen in the figure, the test was done on a environment sufficiently complex, in order to test the algorithm on the worst case.

From the pictures can be evaluated that an ArUco of the dimension of 0.067 m performs in a good way until it is kept at a distance of maximum 2.40 m. In 3.13f where the ArUco is 2.60 m far, can be observed that borders are detected, but resolution is not sufficient to identify the marker. Observing the video in real-time can be seen that sometimes it is identified. Whereas, incrementing the distance, the ArUco is no more detected. This means that 2.60 m approximately represents the upper limit to observe with this ArUco. However performances can not be considered consistent already at 2.60 m where detection results irregular. If this distance, or even more, is required, the size of the ArUco must be incremented.



**(a)** *1.40 m far* **(b)** *1.80 m far*

**(c)** *2 m far* **(d)** *2.20 m far*

**(e)** *2.40 m far* **(f)** *2.60 m far*

**Figure 3.13:** Test on the ArUco detection at different distances

In conclusion, this test permitted to apply ArUco of the size of 0.067 m that turned out to be consistent in the detection of a marker at a distance of about 2.40 m.

### Test 2 on ArUco: Analysis of the error in pose estimation

The second test consisted in obtaining a measure of the error in the pose estimation, both in terms of orientation and position. This was realized considering a setup consistent with the one tested previously (ArUco of size 0.067 m and camera 2 m far away).

This test was carried out comparing the error of two ways of obtaining an

estimation of the pose. However, one first issue was related to the measure to consider correct in the comparison. Since there was no way to obtain a precise measurement, the following analysis regards only the deviation from the mean value, to give an idea of data oscillation. Moreover, measurements have been referred to the world frame, which is built from the pose of one marker, that is considered to be fixed. Once the frame was available, the marker defining the origin was kept fixed and its measure was transformed in the relative designed frame through homogenenous transformations. In this way, the two measurements compared in each method were the ones relative to the origin that, in an ideal world, should have been the same.

The first method used to build the frame was the classical one presented in 3.4.2. Here, one ArUco is inserted in the environment, a picture is taken and the estimation is made using the calibration parameters along with the size of the ArUco. In this test, the frame to exploit is simply the one relative to the ArUco detected and estimated.

The second method followed an approach that would have increased performances. This was done because of the ambiguity problem stated in [21]. The problem is related to a strange behaviour of the z-axis that presents a change flip between two poses. This is due to the fact that there is an ambiguity in the projection. In order to overcome this problem, the measure was done relatively to a frame designed with three ArUco. These three were printed in such a way to build a right-handed triad. In this way, the flipping was no more a problem, because the frame must had been consistent.

Results were obtained taking detection data over a reasonable period of time and modelling a normal distribution. Data was given in terms of tvec and rvec, i.e. 3x1 array, and the distribution was computed on the norm derived from the sum of the three elements of each array. In Figure 3.14 and 3.15 results are shown. Here, the first method is named "1Aruco", while the second "3Aruco".

In Figure 3.14 the position error expressed in meters is shown. It can be observed that there is not a great difference between the two methods. Whereas, in Figure 3.15, reporting the orientation error in radiants, a more significant difference is visible.

However, this test evaluated that there is not a huge difference between the two approaches. Since for the target implementation, applying three ArUco would have been difficult, the classical method was exploited.

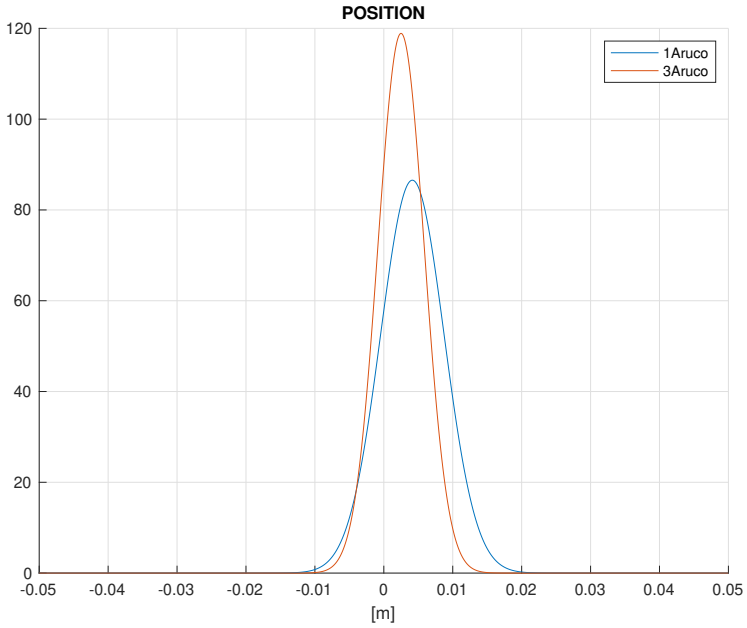



**Figure 3.14:** Position error

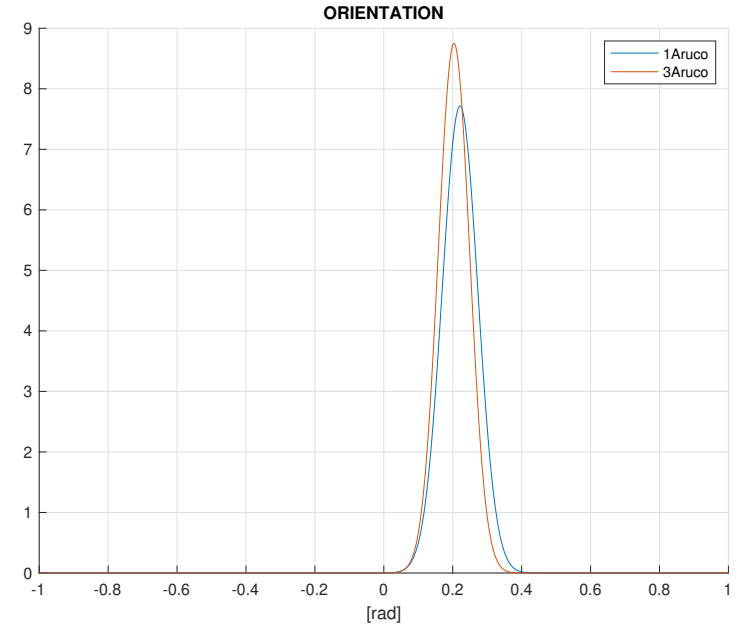



**Figure 3.15:** Orientation error

# Chapter 4

# Experimental tests

## 4.1 Laboratory setup

This first section intends to give an overview of how the laboratory setup was structured on the base of tools presented in the previous chapter. The connection between these tools is shown in Figure 4.1.

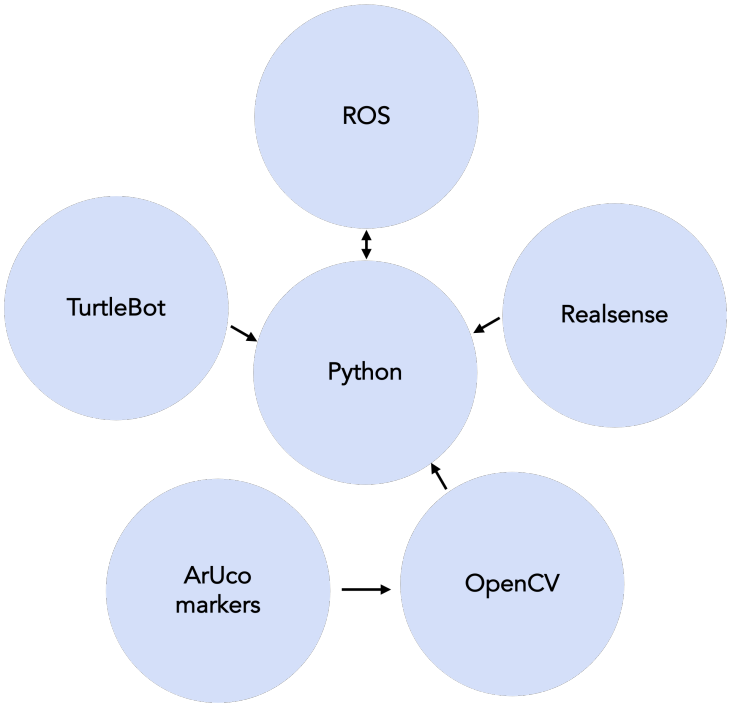



**Figure 4.1:** Tools exploited in the implementation

First of all, as stated in 3.1.1, the usage of ROS in this thesis was compulsory. Since it was used a TurtleBot3 burger, a compatible release of ROS was necessary, and consequently of Linux Ubuntu. In this case, the choice was ROS Noetic together with Ubuntu 20.04. This was not installed on a virtual machine, to enhance performances right from the beginning, avoiding potential delays. Whereas, the used computer was partitioned and Ubuntu was installed on it. Then, the required version of ROS along with the packages needed were installed. Concerning the connection between ROS and TurtleBot, they needed to be connected to the same network to exchange topics.

For this aim, it was used a mobile router. In the second phase, when a camera was integrated in the original setup, the Realsense was connected via a USB 3.0 cable to the computer. From this starting point, Python was chosen with respect to others programming languages, thanks to its versatility. Remarks that brought to this decision are deeply explained in 3.1.1.

In this work, Python played the central role of the interface between all the others components. The written code required to correctly manage topics's exchange between ROS and TurtleBot, deriving from the MAP computation. In addition to that, on the second phase, ArUco were integrated. Consequently, a parallel elaboration of images was needed. This included a managing of the communication with the camera. To do so, a concurrent scheduling scheme was applied, in order to have two streams working at the same time. In 4.2.1 and 4.3.1 the general structure of the code will be analyzed, underlying the differences between the first and the second phase of the implementation. These did not only regard the camera and the elaboration of frames, but the dynamical recognition of the obstacle and the target too. This means that a real-time sculpting of the potential field was programmed.

Concerning the setup, in a first phase, the intention was to reply a test simulated in Gazebo. This was done according to the values of the example of Chapter 2 and the instructions given in [16]. The result of this test is reported in Figure 4.2. In this test, the target was positioned 2 m far on the x-axis of the TurtleBot, which represented the origin. While the obstacle at about 1 m. Therefore, following the results, a area of at least 2 m x 1 m was required. Testing was carried-out marking on the ground these coordinates associated to the setup. The object used as a obstacle was a box whose dimensions could have been enveloped in the radius of a disc. Thus, this

setup followed the simplification made on [16].

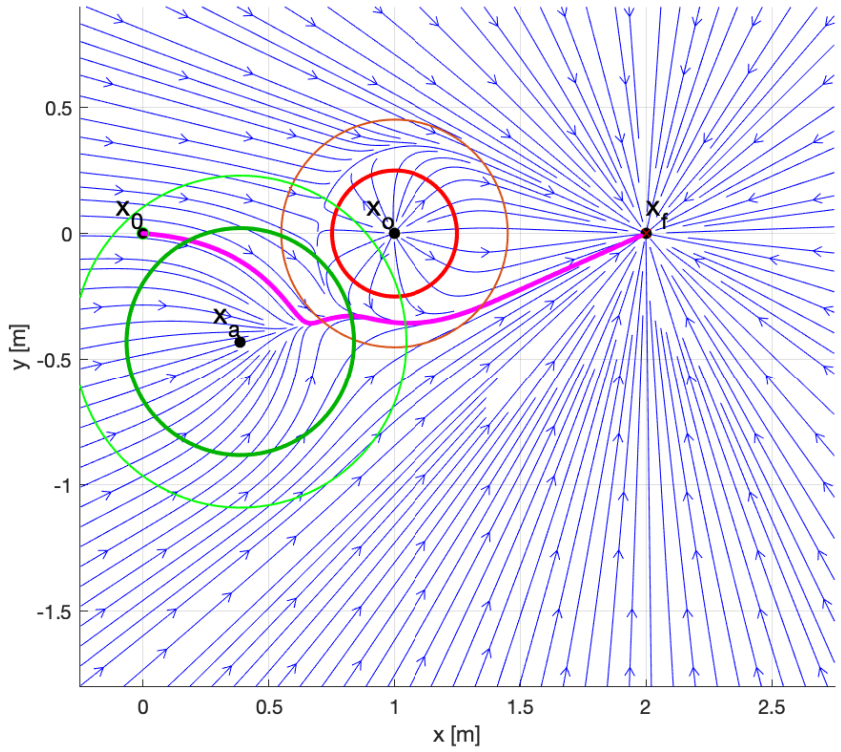



**Figure 4.2:** Result of a test simulated in Gazebo

In the second phase of the implementation, a different way of obtaining the positions of the obstacle and the TurtleBot was desired with respect to [16]. This because of bad performances registered with the odometry feedback that will be deepened in 4.2. Machine learning could have been a choice. However, it was a choice computationally heavy and the aim was to obtain a result as much "light" as could have been done. Thus, ArUco were added to the system, along with a camera. The idea was to test the MAP approach in a setup not so supportive to obtain a consistent result. Different upgrades can be done on this setup, starting from the inclusion of deep learning or working in a more friendly environment. However, this aspect will be extended in the last chapter.

Therefore, the second phase of this implementation involved the integration of a camera. From the requirement on the area needed, the choice of the

distance between floor and camera derived. The camera was positioned at about the centre of an area sufficiently large, looking at the floor. A picture of the experimental setup is reported in Figure 4.3.

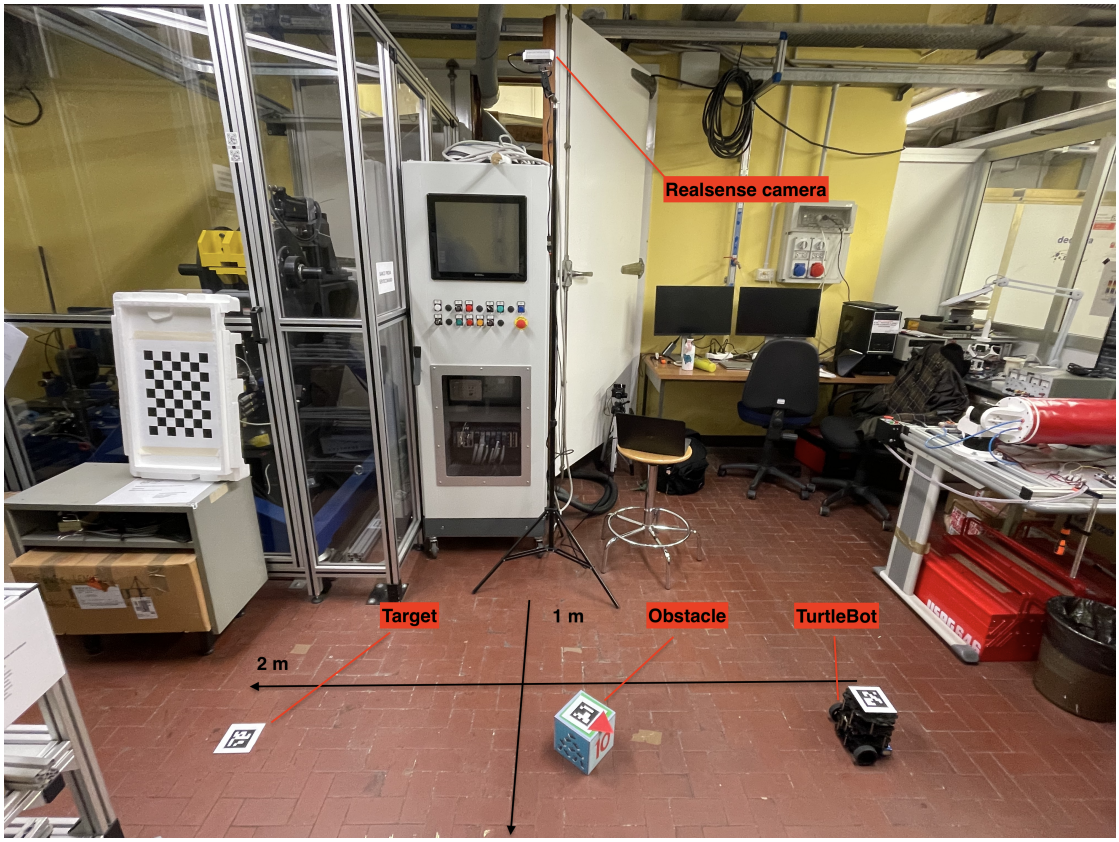



**Figure 4.3:** Laboratory setup

Following the specification of the applied camera, the required distance from the floor was computed. According to 3.3, Realsense d435 offers a FOV of 69°x 42°(hxv). Supposing to place the TurtleBot in the way shown in Figure 4.3, 69° was approximately relative to the x-axis of the TurtleBot, while 42° to the y-axis. Since 2 m were required on the x-axis, it resulted that the camera should have been placed at, at least, 1.5 m far. In order to consider a robust setup, a bigger area was considered. In this way, eventually unexpected behaviour of the TurtleBot would have been detected. Thus, the camera was placed 2 m far from the floor. This, consistently with the results of the two tests in 3.4.2.

The way ArUco were applied is already shown in Figure 4.3. One on the

target, one on the obstacle and the last one on the TurtleBot. Regarding the choice of the ArUco, in terms of dictionary and size, several tests were taken. ArUcos coming from the dictionary DICT_6x6_100 of the size of 0.067 m resulted to perform in a good way in the setup considered. In addition to the tests shown in 3.4.2, they were tested in motion too. This because one of them was applied on the TurtleBot and the pose estimation should have been done while moving.

In Figure 4.4 the setup is shown from the camera point of view. In this picture the setup is relative to the second phase. In the first phase, the setup was the same, just not considering markers and tracing on the floor the obstacle and the target position. It can also be recognized the obstacle. It was applied a sort of arrow on it to indicate the preferred region on which the TurtleBot is expected to pass.

**Figure 4.4:** Laboratory setup from camera point of view

## 4.2 Odometry feedback

The starting point of the implementation in a real scenario was a simulated test. In particular, the setup followed the one shown as an example in Chapter 2. The aim was to reproduce a setup that was supposed to work

well. Consequently, obstacle and target positions in this first phase were assumed known and fixed. Thus, this can be considered a static approach.

Firstly, in 4.2.1 an overview of the structure of the code in this phase will be given. Then, results and limitations of this first solution will be analyzed, explaining the reasons that led to the second phase.

### 4.2.1 Structure of the code

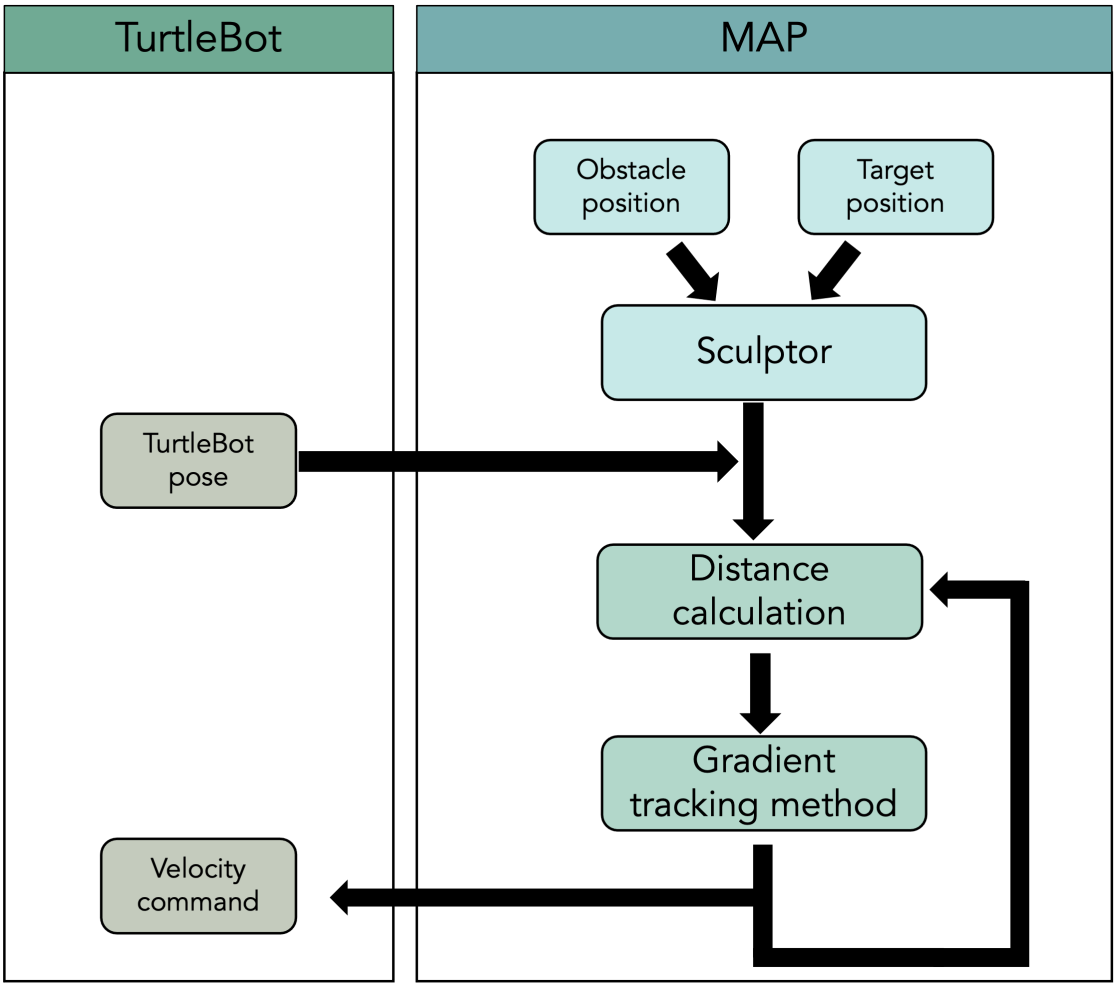



**Figure 4.5:** Structure of MAP implementation in Odometry feedback implementation

In Figure 4.5 the skeleton of the code written to implement MAP in a real scenario is shown. The structure of the code is divided in two parts.

One related to the TurtleBot, the other to the MAP. The former regards the communication with the robot, while the latter represents the heart of the whole algorithm. So, in order to understand it, it should start looking at this last one.

The first step, named sculptor, is all devoted to the computation of the potential field. This receives in input obstacle and target position and returns back the attractive field position, as well as $\sigma$, $\beta_o$, $\gamma_o$, $\alpha_a$, $\gamma_a$ which characterize the potential field. As previously stated, starting, obstacle, and target position are supposed fixed and known. Concerning the attractor position, it is defined by a fixed angle that place it with respect to the obstacle, at a fixed distance that in this tests in considered equal to $3r_o$, where $r_o$ is chosen as 0.25 m to surround the cube. Thus, this step results purely "static". Unless the angle or the positions are changed manually, these parameters will always be the same. This because little oscillations of obstacle and target position, which would be considered consistent to the reality, are not accounted.

The second part is the one on which was dedicated more attention in this phase. It is nothing but the application of the MAP approach with the gradient tracking control law. It consists in a while loop, based on the distance between target and actual position of the robot. It loops until this distance is lower than a certain threshold. Inside the loop, the pose of the robot is updated, thanks to a topic received from the TurtleBot. Then, the negative gradient is calculated, using the updated pose of the robot.

The body of the loop ends calculating the value of the velocity to transmit to the TurtleBot to move towards the target. This value is computed in terms of linear and angular velocity and it is based on the control law described in 2.1. Thus, angular velocity is calculated according to the formula 2.1 and is limited to the maximum velocity reachable, which is 2.84 rad/s. In addition to that, attention is paid to the sign of this speed, checking the vector product between actual and desired velocity. Then, three terms are computed for the magnitude of the linear velocity, as in the relation 2.2. However, concerning this last, it is limited to half of the maximum value, 0.1 m/s. This is to prevent possible slippage. The direction of it instead is chosen according to the negative gradient calculated previously, following the gradient tracking method. Once the robot reaches a position whose distance from the target is under the threshold of 0.1 m, the while loop ends and the robot is stopped, sending a null velocity as command.

On the basis of the control law used, Figure 4.6 shows the gradient tracking

approach. Referring to the labels in the figure and the formulas in Chapter 2, assume that $\varphi = \angle v_d v_r$ and $v_d = - \bigtriangledown U_{art}$, i.e. the negative gradient.

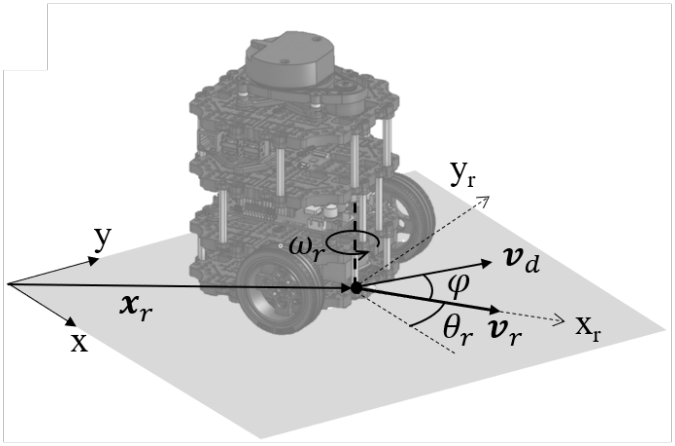



**Figure 4.6:** Gradient tracking [16]

Regarding the communication with TurtleBot, as anticipated, the two topics exploited are: odom, to receive odometry feedback, and cmd_vel, to send velocity commands. In a few words, odom is a publishing topic transmitting orientation and position of the robot, while cmd_vel is a subscribing topic receiving the command velocity to apply to the robot.

The graph related to this connection is shown in 4.7. In the figure, the circular communication can be visualized. Odometry comes from the TurtleBot and goes to the node relative to the implementation of MAP. Whereas, from this last comes the command velocity that is transmitted to the robot.

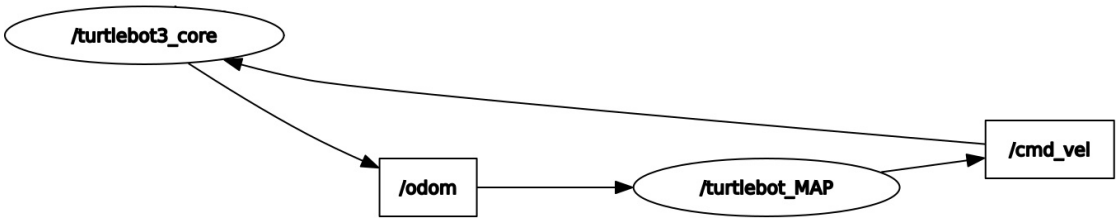



**Figure 4.7:** RQT graph in Odometry feedback implementation

Even if not reported in the scheme, at the beginning a general initialization is done. The relevant part in this is related to the connection with TurtleBot and significant steps are shown in the extrapolation of code 4.1. Here, after

the initialization of the node 'turtlebot_MAP', the publisher pub and the subscriber sub are created. Regarding the first, in the initialization is passed the type of message that want to be published. Instead, in the subscriber, together with the information relative to the type of message to subscribe, a callback is passed. This refers to a method that is called when a message of type odometry is received. This method, along with the main implementation of MAP, is reported in Appendix A.

**Listing 4.1:** ROS topic inizialization

```
1    [...]
2
3    rospy.init_node('turtlebot_MAP', anonymous=False)
4
5    self.pub=rospy.Publisher('/cmd_vel', Twist, queue_size=10)
6    self.sub=rospy.Subscriber('/odom', Odometry, self.callback)
7
8    [...]
```

### 4.2.2 Results

The results shown in this part presented a similar setup to the the ones proposed in [16].

This test was made considering the same positions of the example in Chapter 2 and setting up the laboratory in the way described in 4.3. Summarizing, it was considered the following setup:

- starting position in $x_0 = [0,0,0]$
- a *global attractor* placed in the target position in $x_d = [2, 0, 0]$ with $\sigma = 0.5$
- a *repulsor* placed in the obstacle position in $x_o = [1, 0, 0]$, in the middle between the starting and the destination position, with $\beta_o = 1$ and $\gamma_o = 80.3475$; the obstacle was enveloped in a radius $R_c = 0.135$ m but, as suggested in [16], the radius of the active region of the obstacle was extended for a wider range, including TurtleBot, from which derived $R_o = 0.25$ m
- a *local attractor* placed considering as preferred region of the obstacle the one below it; thus, it was positioned in $x_a = [0.3856, -0.4302, 0]$, at

215° counterclockwise from the obstacle position, with $\alpha_a = 0.2770$ and $\gamma_a = 17.5406$

The test was run using a control frequency of 60 Hz, a threshold of 0.1 and a gain ($K$) equal to 1 m.

The result relative to this test is shown in 4.8. Following the same representation of Chapter 2, the obstacle's region is represented in red, while the attractor's in green. In both, the inner circle represents the effective radius of the obstacle, while the outer, the active region outside of which the influence of the attractor/repulsor is null.

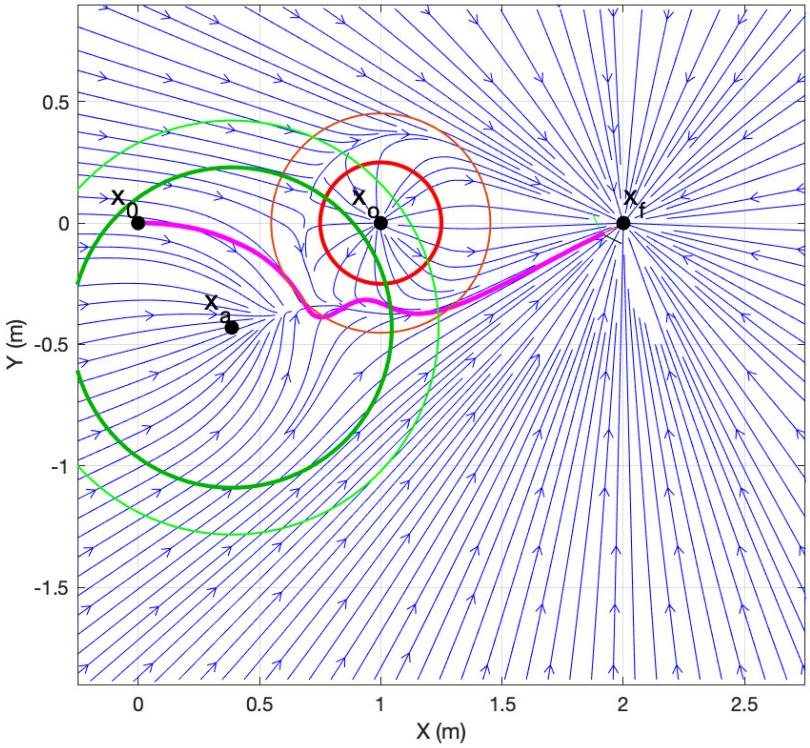



**Figure 4.8:** Result of odometry feedback with preferred region below the obstacle

Secondly, a symmetrically opposite setup was tested. The aim of this second test was to consider as preferred region the one above the obstacle, demonstrating that the algorithm worked properly and concording to the MAP. That is, that changing the definition of the attractor position, the

mobile robot would have followed an opposite trajectory. Thus, the attractor was placed 145° counterclockwise from the center of the obstacle. The only value that changed from the previous setup was the local attractor position, which resulted: $x_a = [0.3856, -0.4302, 0]$.

The result relative to this second test is shown in 4.9. The representation agrees with the previous one.

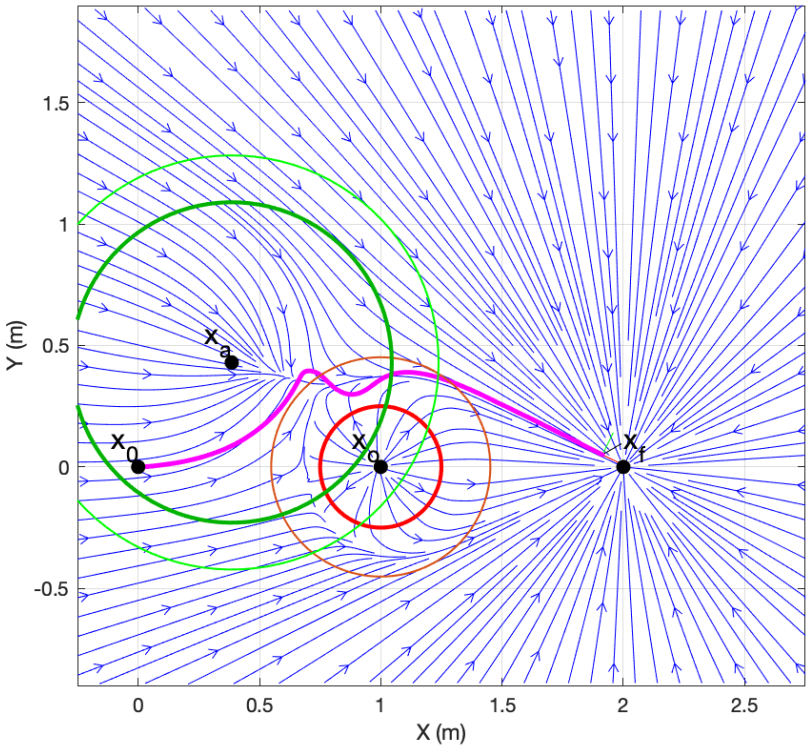



**Figure 4.9:** Result of odometry feedback with preferred region above the obstacle

### 4.2.3 Remarks on the results

Even if from Figure 4.8 and 4.9 a perfect behaviour can be evaluated, in the reality this was not true. This was mainly due to the irregular floor on which the robot moved. If the robot got stuck or slipped on the floor, odometry would have kept going on and no defect were detected. In order to prove

this behaviour that is not visible in the plot of the odometry, it was made a comparison with a measure that reflected reality. This was done using ArUco for pose estimation. The comparison relative to Figure 4.8 is shown in Figure 4.10. Here, in blue is represented the ArUco estimation, while in magenta the odometry.

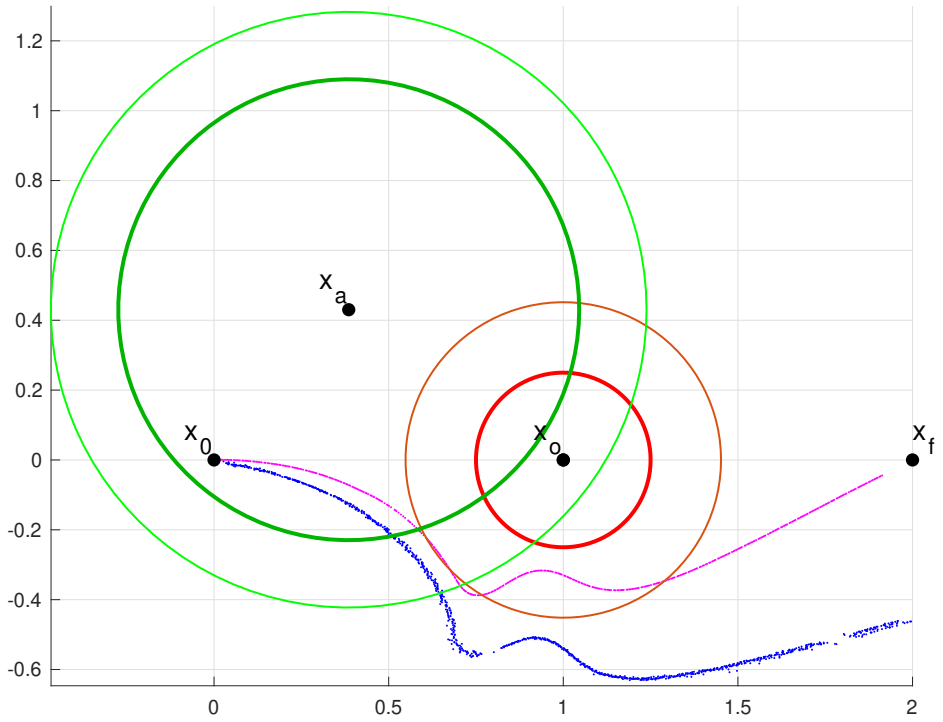



**Figure 4.10:** Comparison between odometry and ArUco pose estimation in the test shown in 4.8

However, more striking cases occurred, such as the one reported in Figure 4.11 where the robot got stuck and the behaviour completely deviated from the one registered by odometry.

One idea could have been to move to a more "friendly" floor. However the aim was to obtain a consistent result, with as less limitations as could have been done. From this assumption, the decision was to drop odometry and move to a different type of feedback. The idea was to integrate a camera in order to have a bird's eye view that would have given a knowledge about

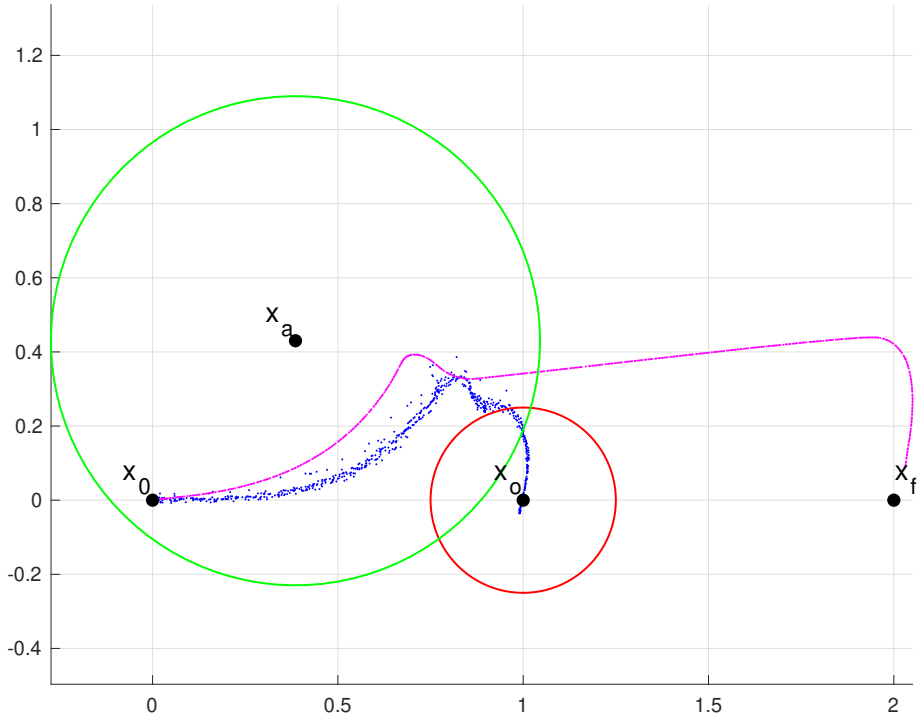



**Figure 4.11:** Comparison between odometry and ArUco pose estimation when the robot got stuck

the whole environment in which the robot moved. Implementing a camera, machine learning could have been a choice, but required great computational effort. Thus, ArUco seemed to be the best trade off to both enter dynamical components in this implementation and obtaining a different feedback of the pose of robot.

## 4.3 ArUco feedback

For the reasons explained in 4.1 and 4.2.3, ArUco markers were employed as the new feedback. In this way, together with obtaining a consistent feedback, it was possible to consider a more dynamical setup. In this phase, target and obstacle were still assumed stationary, but their position was estimated at the beginning of every test. Thus, different tests estimated slight displacements

in the positions. So, the improvement regarded both the removal of the assumption of the prior knowledge, as well as a new way of getting the pose of the robot that would have been more realistic. In order to do this, one ArUco was applied on the obstacle, one on the target point and the last one on the TurleBot. This last was used to substitute the odometry feedback.

Firstly, an idea of the structure of the code will be given in this section, underlying the differences and the improvements from the previous scheme. Then, some testing results will be analyzed and compared too.

### 4.3.1 Structure of the code

The starting point of this improved implementation was the structure described in the previous section. The upgraded version of the second phase is shown Figure 4.12.

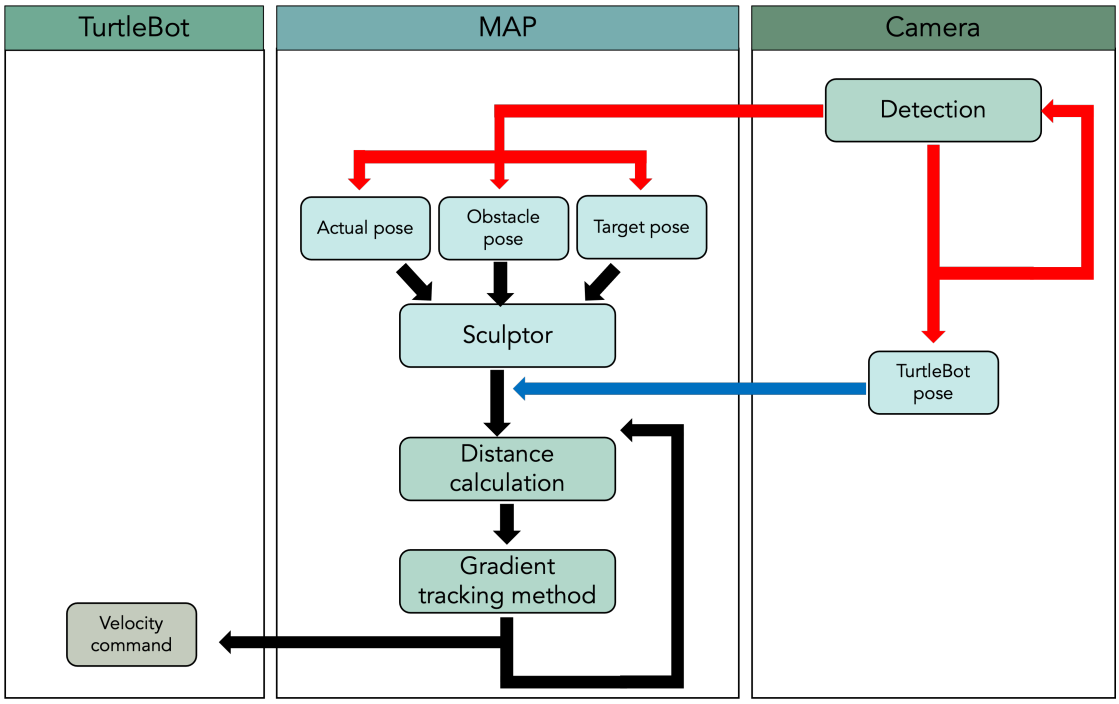



**Figure 4.12:** Structure of MAP implementation in Test 2

Concording to the previous structure, the heart of the algorithm is represented by MAP. The effort in this second phase mainly focused on the improvement of the first part of the code in it. Namely, on the one before the while loop that was previously defined "static". So, in order to have a

more dynamical layout, the first thing to do is the detection of the three pose of interest: obstacle, target and TurtleBot. This is done following the pose estimation described in 3.4.2. After detecting these three, the sculpting already employed in the previous phase is done. This step always leads to different results, since the positions always vary a bit. This assumption is something that makes this approach more consistent to what happens in a real scenario.

However, in sculptor there is another difference from the previous implementation. There, the position of the attractor used to be defined by a fixed angle, that was changed manually, This, used to define the preferred region with respect to the obstacle position. Aiming again at a more dynamical approach, ArUco should be positioned in such a way that it defines the region where the attractor should be placed. In order to do so, ArUco is fixed to an object, assuming that the marker's y-axis defines the direction on which the attractor should be placed. In this way, before the attractor's parameters are computed, the attractor is positioned with respect to the obstacle's frame in the direction pointed by y-axis, at a fixed distance (in the examples it is equal to $3r_o$). After this, the attractive source position is transformed to the reference frame and sculptor follows the same structure of the first implementation.

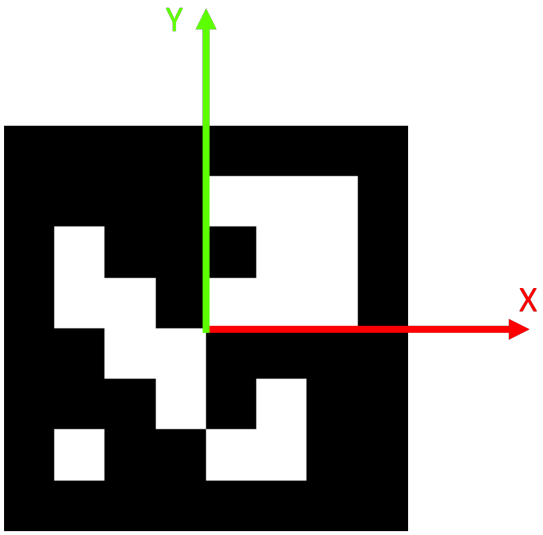



**Figure 4.13:** ArUco's frame

In Figure 4.13 is reported an example of ArUco with the frame drawn.

Because of the way the approach was implemented, it should be payed attention to how it is positioned on the object. Its orientation defines the attractor and consequently where the robot will pass.

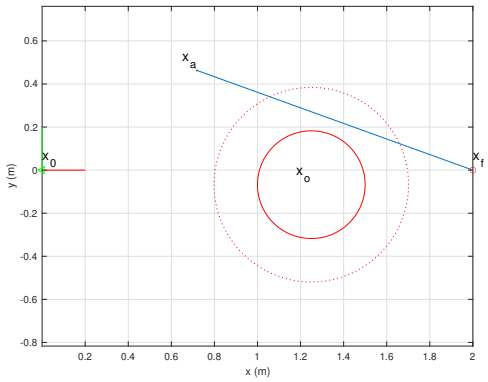
The second part follows a structure that seems to be similar to the original. Actually, it is, except for the the substitution of the odometry with the pose of the ArUco. Therefore, the graph representing the communication between TurtleBot and MAP, results as the one depicted in 4.14.


**Figure 4.14:** RQT graph in Test 2

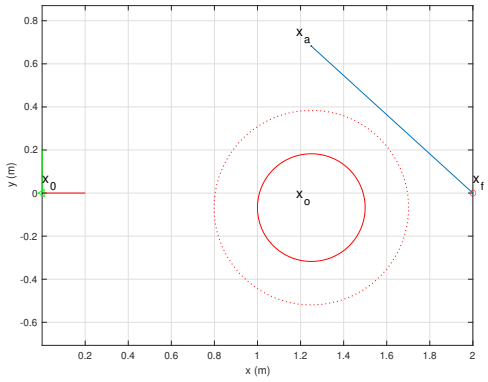
However, even if the skeleton is the same, this part presents a deep programming difference. Once sculptor is finished, main starts a new thread, relative to the camera, in order to have a continuous flow of data about the pose of the robot. The method associated to it is the one reported in 3.3. Thus, this second thread, named "camera" in the figure, implements pose estimation and works independently from the main one. Following this approach, together with a more consistent measure of the pose, which keeps updating, it also makes a better detection of the ArUco, because no delays relative to the interruption of the stream are accounted. However, concurrent programming requires attention, especially if data have to be shared. In this case, the updated pose of TurtleBot is written on a file that, when needed, is read from the main. In order to guarantee consistent data on the file relative to the pose estimation, a lock is employed. This lock is shared between the two threads and ensures an atomic access to the file. The camera thread consists in a loop that continues until the main thread is stopped. At every iteration, after performing a new estimation, it waits for the ownership of the lock. When the writing ends, it releases it. The main thread analogously asks for the ownership every time it iterates, needing an update of the pose and then releases it. In this way, the two thread work simultaneously as shown in the Figure 3.3 where in black is represented the flow of the main thread, in red the one relative to the camera thread, and in blue the atomic operation performed by the lock.

The structure of the code for this second phase has been described. However, there is one more aspect that was implemented in this upgrade. It regards a case already outlined in [16] and anticipated in Chapter 2. This is

relative to a possible scenario that can be faced and should be distinguished.




**(a)** *Scenario 1:* $\overline{x_a x_d} \cup U_o \neq 0$




**(b)** *Scenario 2:* $\overline{x_a x_d} \cup U_o = 0$

**Figure 4.15:** Two scenarios in MAP

Two scenarios can be faced when a local attractor is modelled through the orientation of the obstacle and are displayed for a better comprehension in an example in 4.15. In the first one (4.15a) that is the one on which the first implementation was designed, the line segment $\overline{x_a x_d}$ intersects the active region of the obstacle. Thus, the potential saddle point would overlap with the obstacle if the restriction 2.13 is not respected. And this would reflect in a loss of control on the local attraction, that leads to pushing the robot towards the obstacle. In the second case instead, this limit is relaxed and the attention focuses on the restriction 2.14.

The two restrictions to whom the first case is subjected, were already implemented in the first phase. Whereas, the second case represents a simplification and, in this case, the local attractor can be designed with a higher radius eventually enveloping the obstacle. It just has to pay attention to not incorporate the final point.

In conclusion, this upgraded version of the previous implementation accounts new elements, both from a programming point of view and from the approach to the implementation of MAP. The upgraded implementation of MAP is reported in Appendix B.

### 4.3.2 Results

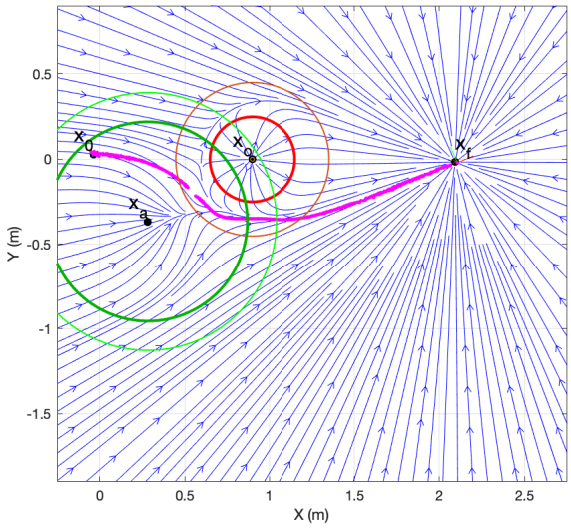
Concerning this upgraded implementation, the setup was no more static. The three elements were positioned in a such a way that reproduced the setup of the first test, but necessarily they were not the same. TurtleBot, obstacle and target were not positioned randomically from the beginning, in order to be sure that the setup, at least theoretically, would have worked. Actually, this is a good point in view of a purely dynamic approach, because it accounts non predictable positions.

All the tests that will be shown in this section were run using a control frequency of 60 Hz and a gain (K) equal to 1. Concerning the threshold, two different values were applied during the tests. Generally, it was decreased compared with the one used in the previous phase (0.1 m). In the first phase with lower values of threshold, the robot kept turning around, not stopping, because odometry was more subjected to error. Instead, in this phase it was set at 0.02 m in all tests, except for the last one where velocity was increased and it was fixed to 0.05 m. Probably this augmentation increased the error in the ArUco estimation, especially when velocity was taken to the maximum. In each example the values relative to the potential field will be reported, in order to understand how the behaviour changes. The labels of these values will replicate the ones used in Chapter 2.

In addition to that, all the tests were done using a fixed frame as reference. Along with the plot of the trajectory with respect to the gradient, some shoots of what was captured from the camera are shown. In this way, the reference frame of all the detected ArUco will be represented, as well as the one relative to the "World". In order to emphasize the direction on which the robot is expected to pass, it was attached a red arrow respectively to the y-axis identifying the preferred region.

**Scenario 1:** $\overline{x_a x_d} \cup U_o \neq 0$

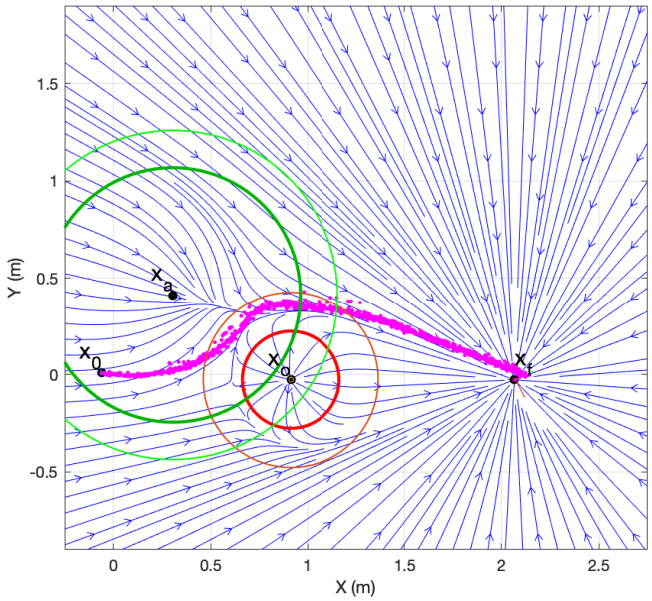
The first results are referred to a setup remarking the scenario 1 (4.15a) that roughly replicate the examples in 4.2.



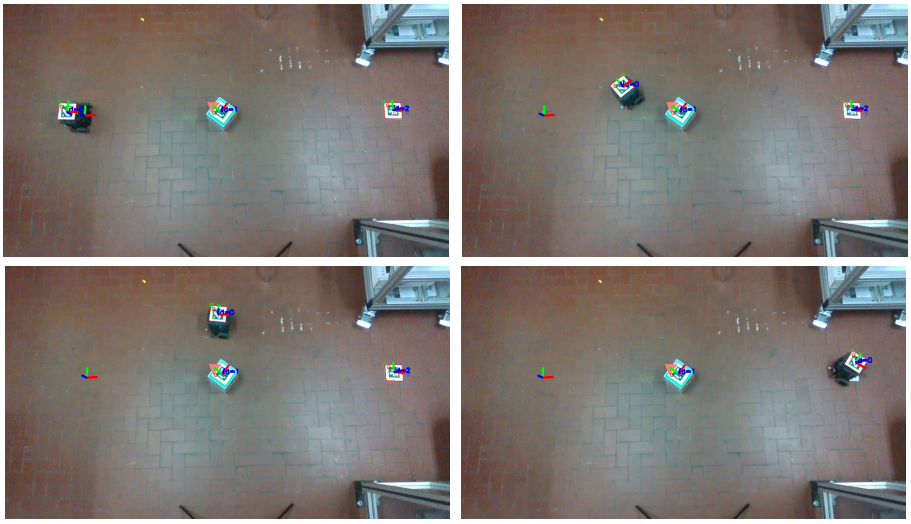

**(a)** Trajectory of the TurtleBot on the APF

**(b)** ArUco detection from camera point of view

**Figure 4.16:** Result of ArUco feedback with preferred region below the obstacle and $\alpha_a = 0.9\widetilde{\alpha_a}$. APF parameters: $\sigma = 0.5$, $\beta_o = 1$, $\gamma_o = 80.3475$, $\alpha_a = 0.2559$ and $\gamma_a = 22.1587$

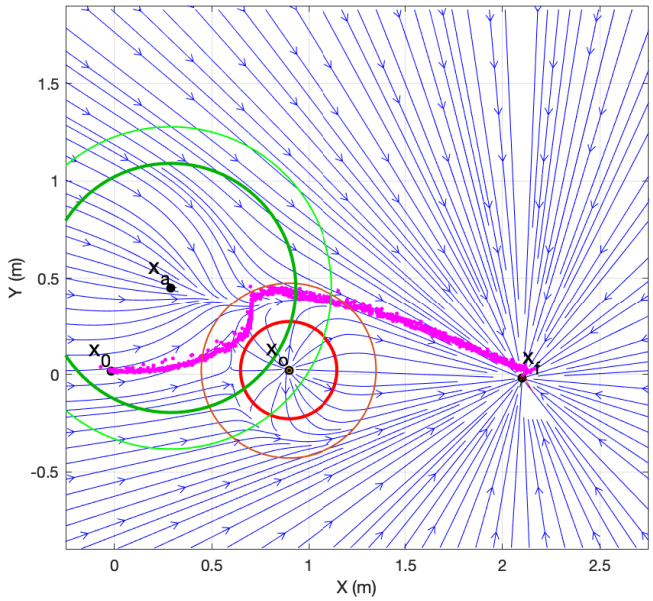



**(a)** Trajectory of the TurtleBot on the APF

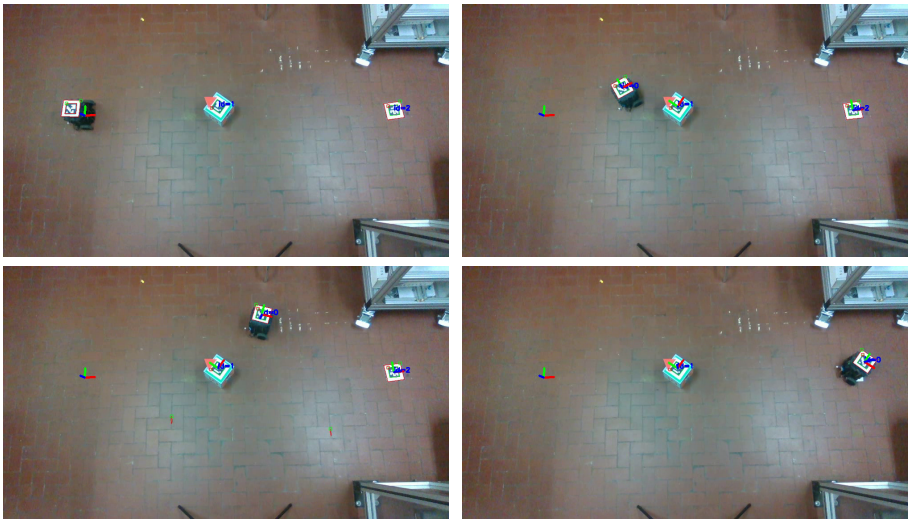


**(b)** ArUco detection from camera point of view

**Figure 4.17:** Result of ArUco feedback with preferred region above the obstacle and $\alpha_a = 0.9\widetilde{\alpha_a}$. APF parameters: $\sigma = 0.5$, $\beta_o = 1$, $\gamma_o = 80.3475$, $\alpha_a = 0.2760$ and $\gamma_a = 17.7249$




**(a)** Trajectory of the TurtleBot on the APF

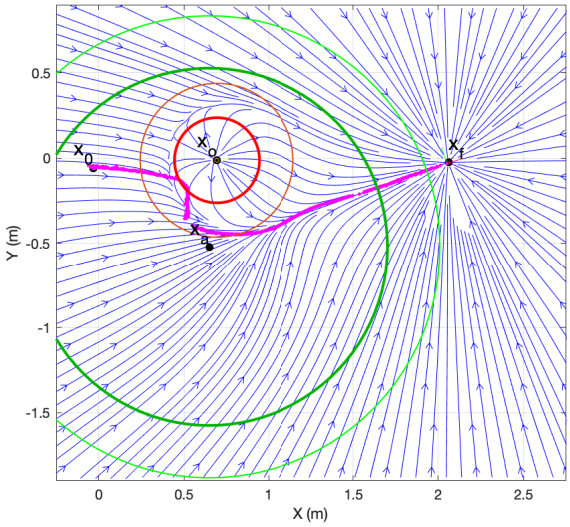


**(b)** ArUco detection from camera point of view

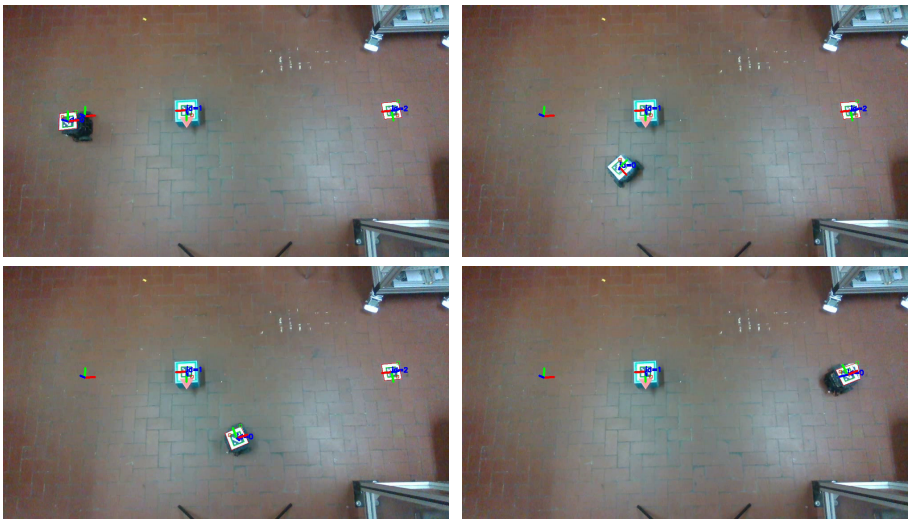
**Figure 4.18:** Result of ArUco feedback with preferred region above the obstacle and $\alpha_a = 0.8\widetilde{\alpha_a}$. APF parameters: $\sigma = 0.5$, $\beta_o = 1$, $\gamma_o = 80.3475$, $\alpha_a = 0.2498$ and $\gamma_a = 18.4824$

**Scenario 2:** $\overline{x_a x_d} \cup U_o = 0$

This result is referred to a setup remarking scenario 2.




**(a)** Trajectory of the TurtleBot on the APF



**(b)** ArUco detection from camera point of view

**Figure 4.19:** Result of ArUco feedback with preferred region below the obstacle and $\alpha_a = 0.8\widetilde{\alpha_a}$. APF parameters: $\sigma = 0.5$, $\beta_o = 1$, $\gamma_o = 80.3475$, $\alpha_a = 0.27069$ and $\gamma_a = 6.9058$

### Linear velocity increasing in scenario 1

These results are referred to a setup similar to scenario 1 4.15a, but the velocity was incremented until the limit of TurtleBot. Notice that, in order to stop the robot, the threshold was increased to 0.05 m.

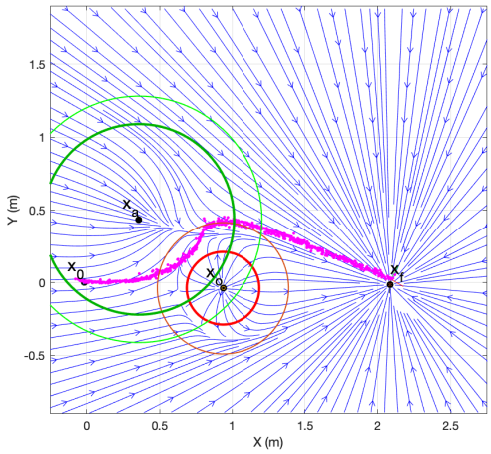



**Figure 4.20:** Result of ArUco feedback with preferred region above the obstacle, $\alpha_a = 0.9\widetilde{\alpha_a}$ and linear velocity $v = \frac{2}{3}v_{max}$

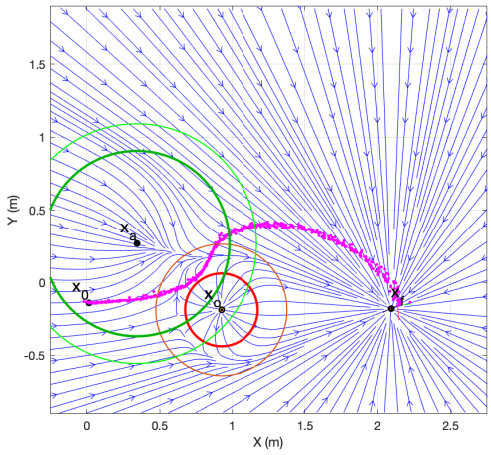



**Figure 4.21:** Result of ArUco feedback with preferred region above the obstacle, $\alpha_a = 0.9\widetilde{\alpha_a}$ and linear velocity $v = v_{max}$

### 4.3.3 Remarks on the results

In order to stress the implementation designed, different tests were made. The main goal was to test the behaviour of the implementation in different situations to ensure that the robot would always be directed in the correct preferred region. In this implementation, this is defined accordingly to the direction of the y-axis of the marker on the obstacle. Thus, this aspect mainly relies on the accuracy of the estimation of the orientation of the obstacle at the beginning of the test.

Comparing Figure 4.16 and Figure 4.17 can be seen how the behaviour changed in a correct way, rotating the object of about 90°. In the first case, it passed above, while in the second, below.

One effect that was tested, was the change of intensity in the local attractor. Keeping all the parameters equal, just decreasing $\alpha_a$, as in 4.17a and 4.18a, the robot performed well. However, since $\alpha_a$ represents the intensity of the local attractor, it can be seen how in 4.17 the robot was more attracted out of the obstacle's active region, than in 4.18.

The test was made also on scenario 2, added with this implementation. In this scenario the radius is subjected to less restring boundaries and consequently it can encapsulate the obstacle. In Figure 4.19 can be seen that the test succeeded.

Another test that was made is depicted in Figure 4.20 and 4.21. Here are not reported the frames relative to the camera, since they were similar to the ones in 4.17. Linear velocity used to be bounded to $v = \frac{1}{2}v_{max}$ in order to avoid slipping and the test were done gradually incrementing this value until the maximum $v_{max}$. In both tests the implementation succeeded, but the threshold used in the main while was increased to 0.05 m, otherwise the robot would not have stopped. However, it can be observed that the behaviour of the first test, where $v = \frac{1}{3}v_{max}$, was quite similar to one obtained in 4.17. Whereas, in the second example, it can be noticed that the trajectory consistently deviated from the gradient lines. This is mainly related to slippage.

In general, the results obtained showed a good behaviour of this implementation approach. It can be said that gradient tracking is largely perceived out of the obstacle region. Whereas, in this region sometimes it deflects from the gradient lines. This is probably due to delays in transmitting the command law, in an area where the gradient repels. Another aspect that can noticed is that positions are not so precise, respect to the first implementation. It can

be said that there is a discontinuous behaviour of the detected position. This is due to the inaccuracy of the ArUco pose estimation, but it is not visible on the motion of the TurtleBot that runs in a continuous way. This thanks to the fact that both the camera and the communication with TurtleBot run at a frequency of 60 Hz. In this way, continuous update are made and small errors do not invalidate the results. Even in cases like 4.16, where the pose is not estimated for a piece of path, the behaviour seems to be consistent. Obviously, this must not happen for a large period, otherwise the application would fail. So, this little oscillations on the position does not affect the success of the application and can be overlooked. Therefore, what can be said is that the results reported behave in a way that is consistent to what it was expected.

## 4.4 Discussion

The very first setup did not represent an implementation that could have been applied in a real scenario. This because it required a static situation, where there was a complete knowledge of the background. However, this proved that the approach would have worked. In addition to that, it has given the suggestion to not rely on odometry for what concerns mobile robots. Thus, it represented the starting point for a more relevant result. The implementation which followed, took a big step in the direction of dynamism. However, as things stand, this implementation is not ready for the real world too. Nevertheless, some developments can be made to obtain a reliable result and bring this technique to real life.

One first thing that can be made is considering a different environment. The floor on which the mobile robot moves affects the success of the implementation. This because ArUco would be detected more easily on a floor in contrast, or at least smooth. In this way, tiles would not be confused with markers' boarder, eliminating some source of noise. In addition to that, odometry would benefit of a smooth path too. However, the main assumption was to obtain a context as much general as could have been done. This was the reason that led to the choice to not move in a different environment.

Anyway, the way MAP is implemented represents the basis for an upgraded development. First of all, from the tests outlined in 4.3 can be assumed that ArUco are reliable tools, but need to be improved. One first simple

enhancement can come from the application of a filter on the position, in order to remove the noisy effect shown. Another idea that requires a little bit more effort, is related to the use of odometry. Odometry in 4.2 resulted unreliable, since it was tested on a context of mobile robots with an unfriendly floor. However, if combined with ArUco it can brings to a more consistent result than the other two. It would perform a sort of sensor fusion. Another approach can be to use more sophisticated ArUco or more sophisticated way of obtaining the estimations. Some are outlined in [21].

Another aspect is related to the fact that employing the camera, the working area is limited to its field of view. In order to overcome this limitation, an idea can be to integrate more cameras. Once synchronized, they would expand the workspace. In this way, it will be more easy to test MAP also with more than one obstacle. Considering a small area, as the one used in this phase, makes difficult to consider this situation.

These are some of the ideas that can implemented to upgrade the proposed method. However, in the next chapter, others idea of different tools to implement on a possible future development will be given.

# Chapter 5

# Conclusion and future works

This work focused on the implementation in a real scenario of a novel collision avoidance technique based on the APF method. This approach aimed at obtaining a safe and predictable robot behaviour, where safety is intended as the ability to avoid the obstacle and predictability as the possibility to know a priori the side where the robot would pass.

In this work the attention focused on implementing this approach on a mobile robot. The aim was firstly perceived in a static way, then employing ArUco to get in the way of a more dynamical setup. The obtained results through the experimental tests, showed a behaviour consistent to expectations in both phases. However, in order to apply it on a real scenario, upgrades still have to be made.

During the dissertation some areas of enhancement have been outlined. The main ones regarded the use of ArUco markers. Even if their performances run well and can be improved in different ways, as suggested in Chapter 4, their use alone represents a great limitation. However, they can be exploited together with other tools to improve the application. Future works may start from this last assumption. For example, an idea can be to used LiDar, already provided by TurtleBot. SLAM is already implemented on it and can be somehow combined with ArUco, in a sort of sensor fusion.

Another possible improvement can be made on the camera. A Realsense was already integrated in this work, but it was used as a normal 2D camera. Future works may use the point cloud that Realsense provides to model a 3D environment.

Moreover, one way can be to move directly to a different approach, integrating a new vision-based solution. This could be perceived exploiting deep learning. However, despite the of the potentiality of this tool, as already stated during the thesis, it is a computationally expansive solution that requires a huge dataset. Thus, in this case, improvement should be made on the equipment too.

Anyway, there are different ways of improving this work in order to exploit this interesting approach in a proper way. Consistent results will be achieved with one of these enhancements when the implementation will behave in a good way with dynamical or multiple obstacles that in a real scenario are taken for granted. This will definitely meet the Industry 4.0's paradigm.

Once results will be achieved, it will be possible to integrate this technology on mobile robots. Actually, the results of this work have been drawn testing the implementation on a simple robotic platform, such as TurtleBot3 burger. Whereas, this model was exploited because of its similarity to mobile robots on the market. Future applications will be based on a implementation designed on it and may involve industrial robot. In this new scenario they must be ready to face any type of obstacle during their navigation.

# Appendix A

# Odometry feedback - main methods

```
1
2  def callback(self, msg):
3      # callback function called when a new message of type
       Odometry is received by the subscriver
4      rospy.loginfo(rospy.get_caller_id() + 'The odometric
       position is:\n %s ', msg.pose.pose)
5
6      self.p  = msg.pose
7
8      self.v_turtle_feedback[self.iteration] = msg.twist.twist.
       linear.x
9      self.w_turtle_feedback[self.iteration] = msg.twist.twist.
       angular.z
10     matom = quaternion_matrix([self.p.pose.orientation.x, self.p
       .pose.orientation.y, self.p.pose.orientation.z, self.p.pose.
       orientation.w])
11     self.mat = matom[:3, :3]
12     self.position_matrix[self.iteration][0] = self.p.pose.
       position.x
13     self.position_matrix[self.iteration][1] = self.p.pose.
       position.y
14
15 def main(self):
16     Vlinear_max = 0.2    # [m/s]
17     Vangular_max = 2.84 # [rad/s]
```

```
18
19      final_error = norm(np.subtract(np.array([[self.p.position.x,
        self.p.position.y, self.p.position.z]]), np.array([[self.p_f
    .position.x, self.p_f.position.y, self.p_f.position.z]])))
20      i = 0
21
22      t_totale = np.zeros((1, 5000))
23      velMsg = Twist()
24
25      while final_error >= 0.1:
26
27          tic = time.time()
28          if i == 0:
29              t_totale[0][0] = 0
30              self.additional_time_vector[0] = 0
31          else:
32              t_totale[0][i] = t_totale[0][i-1] + additional_time
33              self.additional_time_vector[i] = additional_time
34
35          rospy.wait_for_message('/odom', Odometry, timeout = 5)
36
37          roll = math.atan2(self.mat[1][0],self.mat[0][0])
38          self.roll_vector[i]= roll
39          theta_turtle = roll
40          actual_position = self.getActualPosition()
41          final_error = self.normToGoal()
42
43          V_apf = -(self.globalAttractorGradient(sigma,
    actual_position, np.array([[self.p_f.position.x], [self.p_f.
    position.y], [self.p_f.position.z]])) + self.obstacleGradient
    (beta_o, gamma_o, actual_position, disc_centre) + self.
    localAttractorGradient(alpha_a, gamma_a, actual_position,
    attractive_source))
44
45          if (np.linalg.norm(V_apf) == 0):
46              V_apf_direction = np.array([[0], [0], [0]])
47          else:
48              V_apf_direction = V_apf / np.linalg.norm(V_apf)
49
50          # SLIDING MODE CONTROL algorithm
51          a0_sliding = 0.15
52          v0_sliding = Vlinear_max/2
53          d_sliding = final_error
54          v_sliding1 = a0_sliding * t_totale[0][i]
55          v_sliding2 = v0_sliding
```

```
56          v_sliding3 = sqrt(2*a0_sliding*d_sliding)
57          v_sliding = np.array([v_sliding1, v_sliding2, v_sliding3
    ])
58          V_apf_intensity = np.min(v_sliding)
59          V_apf_vector = V_apf_intensity * V_apf_direction
60          v_apf_turtle = V_apf_vector[0:2]
61
62          # Motion planning turtlebot
63          v_turtle = np.linalg.norm(v_apf_turtle)
64          v_turtle_vect = np.array([[cos(theta_turtle)], [sin(
    theta_turtle)]])
65          e_turtle = acos(self.dotProduct(v_apf_turtle,
    v_turtle_vect)/(np.linalg.norm(v_apf_turtle)*np.linalg.norm(
    v_turtle_vect)))
66          w_turtle = e_turtle*1
67
68          if w_turtle > Vangular_max: # primo ciclo non entra, ma
    ok
69              w_turtle = Vangular_max
70
71          angle_coeff =  v_turtle_vect[0]*v_apf_turtle[1]-
    v_turtle_vect[1]*v_apf_turtle[0]
72          if angle_coeff < 0:
73              w_turtle = -w_turtle
74
75          # command vector
76          v_turtle_com  = np.array([[v_turtle], [w_turtle]])
77
78          # send to turtlebot
79          velMsg.linear.x = v_turtle
80          velMsg.angular.z = w_turtle
81          self.pub.publish(velMsg)
82
83          self.rate.sleep()
84          additional_time = time.time() - tic
85          i += 1
86
87      # stop turtlebot
88      velMsg.linear.x = 0
89      velMsg.angular.z = 0
90      self.pub.publish(velMsg)
```

# Appendix B

# ArUco feedback - main methods

```
def sculptor(self, intersection_index = True):

    sigma = 0.5
    disc_centre = self.p_o.position
    disc_centre_OF_oriented, mat = self.generate_message(self.
rvecOC, np.array([[0],[0],[0]]))
    disc_centre_OF = disc_centre_OF_oriented.position
    disc_radius = 0.27/2 + 0.105 + 0.01
    lambda_o=0.3
    r_o = disc_radius
    gamma_o = (-1/r_o**2)*lambertw(-(lambda_o)**2/exp(1),-1).
real
    beta_o = 1
    s_epsilon = 10**(-2)
    ro_star = sqrt((-1/gamma_o*lambertw(-s_epsilon**2/(gamma_o*
beta_o**2),-1)).real)
    angle_as = radians(90) #phi
    direction_as = np.array([[cos(angle_as)], [sin(angle_as)],
[0]])
    distance_as_magn = 3*r_o #das
    distance_as = distance_as_magn * direction_as
    attractive_source_CF = np.array([[disc_centre_OF.x +
distance_as[0][0]], [disc_centre_OF.y + distance_as[1][0]], [
disc_centre_OF.z + distance_as[2][0]]])
```

```
19      attractive_source = transformTvecA(self.tvecW, self.tvecOC,
    self.rvecW, self.rvecOC, attractive_source_CF)
20      attractive_source[0][2] = 0
21      mu_epsilon = 10**(-2)
22      mu_a = 0.1
23
24      if(intersection_index is True):
25          epsilon_lim = norm(attractive_source-disc_centre)-
    ro_star
26          teta_epsilon= acos(-27/(2*lambertw(-mu_epsilon**2/exp(1)
    ,-1)).real-1)
27          gammaa_lim = 1/9*(-1/epsilon_lim**2*lambertw(-mu_epsilon
    **2/exp(1),-1))*(1-2*cos((teta_epsilon+4*np.pi).real/3))**2
28          r_alim = sqrt((-1/gammaa_lim*lambertw(-mu_a**2/exp(1)
    ,-1)).real)
29          r_a = 0.99*r_alim
30          gamma_a = (-1/r_a**2*lambertw(-mu_a**2/exp(1),-1)).real
31          ra_star = sqrt((-1/gamma_a*lambertw(-mu_epsilon**2/exp
    (1),-1)).real)
32      else:
33          r_a = distance_as_magn+ro_star+0.05
34          gamma_a = (-1/r_a**2*lambertw(-mu_a**2/exp(1),-1)).real
35          ra_star = sqrt((-1/gamma_a*lambertw(-mu_epsilon**2/exp
    (1),-1)).real)
36
37      x_a_primo_limite = sqrt((27/(4*gamma_a)).real)
38      x_a_primo_limite1 = ra_star
39      x_a_primo = norm((attractive_source-self.p_f.position
40      teta_cubica = acos((27/(2*gamma_a*x_a_primo**2)-1).real)
41      x3_primo = 2/3*x_a_primo*(cos((teta_cubica+4*np.pi).real/3)
    +1)
42      alfa_a_lim = (-sigma*x3_primo)/(gamma_a*(x3_primo-x_a_primo)
    *exp((-gamma_a/2*(x3_primo-x_a_primo)**2).real)).real
43      alfa_a = alfa_a_lim*0.9
44      teta_epsilon= acos((-27/(2*lambertw(-mu_epsilon**2/exp(1)
    ,-1))-1).real)
45      epsilon = 1/3*sqrt((-1/gamma_a*lambertw(-mu_epsilon**2/exp
    (1),-1)).real)*(1-2*cos((teta_epsilon+4*np.pi).real/3))
46      ra_star_icinco = sqrt((-1/gamma_a*lambertw(-s_epsilon**2/(
    alfa_a**2*gamma_a),-1)).real)
47
48  return sigma, self.p_o, beta_o, gamma_o, attractive_source,
    alfa_a, gamma_a
49
50  def main(self):
```

```
51
52      Vlinear_max = 0.2    # [m/s]
53      Vangular_max = 2.84 # [rad/s]
54
55      final_error = np.linalg.norm(np.subtract(np.array([[self.p.
    position]]), np.array([[self.p_f.position]])))
56      i = 0
57
58      velMsg = Twist()
59      t1 = threading.Thread(target = self.realsensedetect, args =
    [self.cameraMatrix, self.distCoeffs, self.aruco_dictionary,
    self.parameters, self.aruco_dim])
60      t1.start()
61      time.sleep(1)
62
63
64      while final_error >= 0.02 and self.stop.is_set() is False:
65          tic = time.time()
66
67          if i == 0:
68              self.t_totale[0][0] = 0
69              self.additional_time_vector[0] = 0
70
71          else:
72              self.t_totale[0][i] = self.t_totale[0][i-1] +
    additional_time
73              self.additional_time_vector[i] = additional_time
74
75          self.updateAruco()
76          roll = math.atan2(self.mat[1][0], self.mat[0][0])
77          self.roll_vector[i]= roll
78          theta_turtle = roll
79
80          if(i == 0 ):
81              theta_turtle = 0
82          actual_position = self.getActualPosition()
83          final_error = self.normToGoal()
84
85          V_apf = -(self.globalAttractorGradient(self.sigma,
    actual_position, self.p_f.position + self.obstacleGradient(
    self.beta_o, self.gamma_o, actual_position, self.disc_centre)
     + self.localAttractorGradient(self.alpha_a, self.gamma_a,
    actual_position, self.attractive_source))
86
87          if (np.linalg.norm(V_apf) == 0):
```

```
88              V_apf_direction = np.array([[0], [0], [0]])
89          else:
90              V_apf_direction = V_apf / np.linalg.norm(V_apf)
91
92          # SLIDING MODE CONTROL algorithm
93          a0_sliding = 0.15 # maximum acceleration
94          v0_sliding = Vlinear_max/2
95          d_sliding = final_error
96
97          v_sliding1 = a0_sliding * self.t_totale[0][i] # va bene
    tempo???
98          v_sliding2 = v0_sliding
99          v_sliding3 = sqrt(2*a0_sliding*d_sliding)
100         v_sliding = np.array([v_sliding1, v_sliding2, v_sliding3
    ])
101         V_apf_intensity = np.min(v_sliding)
102
103         V_apf_vector = V_apf_intensity * V_apf_direction # array
     3x1
104
105         v_apf_turtle = V_apf_vector[0:2]
106
107         # Motion planning turtlebot
108         v_turtle = np.linalg.norm(v_apf_turtle)
109
110         v_turtle_vect = np.array([[cos(theta_turtle)], [sin(
    theta_turtle)]])
111         e_turtle = acos(self.dotProduct(v_apf_turtle,
    v_turtle_vect)/(np.linalg.norm(v_apf_turtle)*np.linalg.norm(
    v_turtle_vect)))
112         w_turtle = 1*e_turtle # omega definita come errore*
    guadagno
113
114         if w_turtle > Vangular_max:
115             w_turtle = Vangular_max
116
117         angle_coeff =  v_turtle_vect[0]*v_apf_turtle[1]-
    v_turtle_vect[1]*v_apf_turtle[0]
118         if angle_coeff < 0:
119                 w_turtle = -w_turtle
120
121         # command vector
122         v_turtle_com  = np.array([[v_turtle], [w_turtle]])
123
124         velMsg.linear.x = v_turtle
```

```
125             velMsg.angular.z = w_turtle
126
127             self.pub.publish(velMsg)
128
129             # plot
130             self.v_turtle_set[self.iteration] = v_turtle
131             self.w_turtle_set[self.iteration] = w_turtle
132
133             self.rate.sleep()
134             additional_time = time.time() - tic
135             i += 1
136
137         # stop turtlebot
138         velMsg.linear.x = 0
139         velMsg.angular.z = 0
140         self.pub.publish(velMsg)
141         self.stop.set()
142         t1.join()
143         self.pipeline.stop()
```

# Bibliography


[1] B. Siciliano, L. Sciavicco, L. Villani, and Oriolo G. *Robotics: modelling, planning and control*. Springer Science and Business Media, 2010 (cit. on pp. 1, 6).

[2] K. Bahrin, M. Aiman, Othman, M. Fauzi, N. Azli, N. Hayati, Talib, and M. Farihin. «Industry 4.0: a review on industrial automation and robotic». In: *Jurnal Teknologi* 78 (June 2016). URL: https://journals.utm.my/jurnalteknologi/article/view/9285 (cit. on p. 1).

[3] International Organization for Standardization. *Provisional definition of Service Robots English*. Sept. 2018. URL: https://ifr.org/service-robots/ (cit. on p. 2).

[4] *Industry 4.0 Technologies*. URL: https://industry40marketresearch.com/blog/industry_4-0_technologies/ (cit. on p. 2).

[5] De Ryck M., Versteyhe M., and Debrouwere F. «Automated Guided Vehicle Systems, State-Of-The-Art Control Algorithms and Techniques». In: *Journal Of Manufacturing Systems* 54 (2019), pp. 152–173 (cit. on p. 3).

[6] *AGV following guidance on the floor*. URL: https://www.crossco.com/resources/articles/the-difference-between-agvs-and-mobile-robots/ (cit. on p. 3).

[7] iRobot. *Roomba*. URL: https://www.irobot.it/roomba (cit. on pp. 4, 5).

[8] Mobile Industrial Robots. URL: https://www.mobile-industrial-robots.com/it/ (cit. on pp. 4, 5).

[9] EarthSense. *Terra Sienta*. URL: https://www.earthsense.co (cit. on pp. 4, 5).

[10] Boston Dynamics. *Spot*. URL: https://www.bostondynamics.com/products/spot (cit. on p. 5).

[11] Honda. *ASIMO*. URL: https://asimo.honda.com (cit. on p. 5).

[12] B.K. Patle, Ganesh Babu L, Anish Pandey, D.R.K. Parhi, and A. Jagadeesh. «A review: On path planning strategies for navigation of mobile robot». In: *Defence Technology* 15 (2019) (cit. on p. 7).

[13] Lingqi Zeng and Gary M. Bone. «Mobile Robot Collision Avoidance in Human Environments». In: *International Journal of Advanced Robotic Systems* (2012) (cit. on pp. 8, 17).

[14] Rubio F, Valero F, and Llopis-Albert C. «A review of mobile robots: Concepts, methods, theoretical framework, and applications». In: *International Journal of Advanced Robotic Systems* 16 (2019) (cit. on pp. 8, 11).

[15] Amazon. *Meet Amazon's First Fully Autonomous Mobile Robot | Amazon News*. URL: https://www.youtube.com/watch?v=AmmEbYkYfHY (cit. on p. 8).

[16] M. Melchiorre, L. Scimmi, L. Salamina, S. Mauro, and S. Pastorelli. «Robot collision avoidance based on artificial potential field with local attractors». In: *Department of Mechanical and Aerospace Engineering, Politecnico di Torino* () (cit. on pp. 9, 10, 12, 17, 21, 26, 27, 47, 48, 53, 54, 61).

[17] M. Melchiorre, S. Mauro, and S. Pastorelli. *Real-Time Trajectory Planning for Human-Friendly Collaborative Robotics*. 2021 (cit. on pp. 10, 17).

[18] Han-ye Zhang, Wei-ming Lin, and Ai-xia Chen. «Path Planning for the Mobile Robot: a review». In: *Symmetry* 10 (2018) (cit. on p. 11).

[19] O. Khatib. *Real-time obstacle avoidance for manipulators and mobile robots*. 1985. URL: https://ifr.org/service-robots/ (cit. on pp. 11–14).

[20] S. Mauro, S. Pastorelli, and L. S. Scimmi. «Collision avoidance algorithm for collaborative robotics». In: *International Journal of Automation Technology* 11.3 (2017), pp. 481–489 (cit. on p. 12).

[21] Rafael Munoz Salinas. «ArUco: an efficient library for detection of planar markers and camera pose estimation». In: () (cit. on pp. 12, 38, 44, 71).

[22] Beard, Randal W., McLain, and Timothy W. *Motion Planning using Potential Fields*. 2003 (cit. on p. 14).

[23] O. Khatib, K. Yokoi, K. Chang, and A. Casal. «Robots in Human Environments: Basic Autonomous Capabilities». In: *The International Journal of Robotics Research* (1999) (cit. on p. 17).

[24] URL: `https://www.ros.org/blog/why-ros/` (cit. on p. 23).

[25] URL: `https://www.turtlebot.com` (cit. on pp. 24, 28, 30, 32).

[26] Leon Jung Darby Lim Yoonseok Pyo and Hancheol Cho. *ROS Robot Programming (English)*. ROBOTIS, 2017 (cit. on p. 31).

[27] URL: `https://www.intelrealsense.com/depth-camera-d435/` (cit. on p. 33).

[28] URL: `https://www.researchgate.net/figure/Illustration-of-camera-lenss-field-of-view-FOV_fig4_335011596` (cit. on p. 34).

[29] URL: `https://intelrealsense.github.io/librealsense/python_docs/_generated/pyrealsense2.pipeline.html` (cit. on p. 34).

[30] URL: `https://opencv.org/about/` (cit. on p. 36).

[31] URL: `https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html` (cit. on pp. 36, 37).