POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Politecnico di Torino

Master's Degree Thesis

Design and implementation of a Sensory System for an Autonomous Mobile Robot in a Connected Industrial Environment

Supervisor

Prof.ssa Marina INDRI

Advisor at CIM 4.0

Ing. Orlando TOVAR ORDOÑEZ

Candidate

Stefano SANTORO

December 2021

Abstract

Technological evolution has brought considerable progress in the field of automation by introducing and developing systems that have become essential for operations in the industrial field. The development of autonomous systems has allowed the facilitation of human work by allowing collaboration with it but also a replacement to carry out the heaviest tasks.

The purpose of this thesis is to develop a sensory system for an Autonomous Mobile Robot (AMR) capable of moving in an industrial environment sharing space with human operators. To do this, a complex and robust system has been developed consisting of a set of sensors suitable for to the environment in which the robot should move. Exploiting the perceptions of the onboard sensors it has been possible to obtain an autonomous navigation system.

For the choice and connection of the sensors, the state of the art of the most used navigation systems has been studied taking into account a set of fundamental aspects, as well as some problems that could have limited the navigation system. It consists of the Scout Mini platform made up of four Mecanum wheels enriched by a larger structure used for maintenance purposes in an industrial environment. The latter limits the range of vision and therefore the omnidirectional movement allowed by the type of wheels used. In addition, the choice of sensors also takes into account the errors introduced by these types of wheels that are subject to vibration and slippage.

The system, developed using ROS (Robot Operating System), has been simulated and experimentally implemented by choosing an appropriate position of the sensors chosen, an architecture that allows the reduction of computational cost and using some algorithms to fuse and filter data from sensors.

Acknowledgements

I would like to thank Prof. Marina Indri for her helpfulness and her always useful advice. I would like to thank Fiorella and David for their support, feedback, and very useful suggestions.

I would like to thank all the people at CIM 4.0 for trusting my work and giving me the opportunity to do my thesis at their company. I would like to thank all my colleagues in this project for helping and supporting me throughout my thesis work.

Also, a special thanks go to my family and endless thanks to my mother. There are no words to describe the immense gratitude towards her, who made me the person I am today, and who has been and will always be a source of inspiration.

Finally, I would like to thank all my friends who in these months have encouraged me giving me a motivation to give my best.

"To the end, to new beginnings
to those who have been there, those who will be there
to the hope that there may always be
to the dreams that may come true
to life
thank you all"

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, the use of autonomous systems within an industry has increased significantly. They are represented by systems that facilitate the transport of goods, products, and tools useful to operators. In fact, through their use, people's work is facilitated by supporting them in some operations or even by replacing their work. In particular, there are different kinds of autonomous systems and they are mainly divided into Autonomous Guided Vehicle (AGV) or Autonomous Mobile Robot (AMR).
The formers have long been used to transport goods within an industry but have limitations as they can perform simple repetitive instructions and requires an invasive modification of the environment where they are moved so it is very difficult to apply changes. They are guided by cables installed in the environment in which they move or by magnetic strips and sensors located outside the vehicle itself. Instead, the latter represents an evolution of AGVs in terms of functionality and flexibility.
An autonomous mobile robot is a system capable of navigating in an unknown and unpredictable environment that through the use of sensors that are mounted on it, it can sense the surrounding environment and locate itself inside it.
The flexibility of the AMRs constitutes a fundamental requirement within an Industry 4.0 where there may be frequent changes or modification of the production line and therefore it may be necessary to quickly adapt the devices used within them.

The resulting system of this thesis should be used for maintenance purposes moving in an industry 4.0 like the one where the thesis work took place: the Competence Industry Manufacturing 4.0.
Industries 4.0 are made up of a set of emerging technologies interconnected with each other that lead to the development of the manufacturing system in all its forms. They represent an industrial evolution consisting of automated and interconnected industrial systems.

The goal of the thesis is to create a robot capable of moving autonomously in an environment acquiring information through sensors mounted on it. Therefore, suitable sensors, for this type of application, have been studied, chosen, and applied to the used robot. Sensors are of great importance because they represent the means through which the surrounding environment is perceived and the way to obtain robust and safe navigation.

Furthermore, navigation algorithms have been chosen and implemented to allow the robot to avoid obstacles and plan the best path to reach an endpoint in a map.
The robot used is an omnidirectional mobile robot with four Mecanum wheels. Wheels of this kind guarantee wide large freedom in terms of movement but they can give some errors in the implementation of the model due to the slippage. To overcome this problem some solutions and fusion from multiple sensors have been studied and implemented in the system.
On top of the robot, there is mounted a base used as a landing base for a drone and as an instrument for supporting operations inside an industry as shown in Figure 1.1. This represents a constraint and a limit as regards the robot's viewing range.
For this reason, the thesis is focused on the study of the best placement of the sensor system in the robot, to have different degrees of freedom, and to avoid a limitation in the performance of the sensors and of the robot used.
Finally, the solution has been implemented in reality using and testing different solutions of increasing complexity.

The thesis is developed using ROS (Robot Operating System) that thanks to its modularity, allows modification and interaction with the system in a simple way. ROS, being a distributed system, made it possible to connect various hardware components in a way to reduce the computational cost deriving from the data acquisition of the multiple sensors used. Many cutting edge boards and sensors have been used for the hardware components: Nvidia Jetson Nano, Nvidia Nx Xavier, LiDAR 2D, and depth cameras.

Once that the appropriate placement of the whole set of sensors in the robot has been chosen, the system has been simulated through Gazebo and RViz where the same environment of CIM 4.0 has been created.

Figure 1.1: System consisting of the robot and the maintenance system with initial configuration without sensors

1.1 Organization of the work

The thesis is structured as follows:

- The second chapter describes the state of the art of robotics focusing on the study of the wheels. The robot used in the thesis is equipped with Mecanum wheels. This chapter describes the differences between conventional and Mecanum wheels, focusing on the kinematic of these last ones.
- In the third chapter, there is a description of all the different sensors used for autonomous navigation. After that, some algorithms used for sensor fusion are explored.
- The fourth chapter explains the main differences between Path planning Algorithms distinguishing the global and the local ones.

- The fifth chapter describes the autonomous navigation of the robot, the simulation environment, tests done in that environment, it describes how the map is built, and how the robot recognize object during its movement.
- The sixth describes all the hardware used for this kind of application and there are considerations about the chosen of a certain kind of sensors, how they are implemented in the system and also some results obtained making experimental tests.
- The seventh chapter describes the performance of the system created, the limits, and how these can be improved and overcome for future developments.

Chapter 2

State of the art

The invention of robots has very ancient cultural roots. Men have always tried to create something that replicates their actions. The term robot was coined in 1920 by the Czech playwright Karel Čapek in the play *"Rossum's Universal Robots"*.
It derives from the term Slav robota which means executive labor [1]. In the beginning, the term was used to denote something similar to humans and therefore made mainly of organic material. In the following years, and in particular in 1940 with the Russian Isaac Asimov there is a different conception of the robot which is used to indicate something mainly mechanical and devoid of feelings.

2.1 Robotics interaction

Nowadays, robots are a key element in some industries. They can be of different types and they are used in various industrial, military, land, underwater, and even in environments difficult to reach for humans.
Robots operating within an industry can interact with operators by assisting them in their operations and thus supporting them. However, they can also replace them in jobs that require more effort or repetitive action.
The interaction between these two actors is called *Human-Machine Interaction (HMI)* and it can be of different types depending on the actions performed, the space in which it occurs, and the type of interaction.
In particular, there are three types of categories in which *HMI* can be classified [2]:

- **Human-Robot Coexistence:** they share the same workspace performing tasks with different aims.
- **Human-Robot Cooperation:** represents a higher level than the previous one where humans and robots perform the same tasks acting simultaneously

in terms of space and time. In this category, robots can differentiate humans from generic objects.

- **Human–Robot Collaboration**: perform complex tasks where an active interaction of humans can be physical and so with an explicit contact or contactless if physical contact is no present.

2.2 Robotic Structures

The robots are distinguished according to their structure and their use in two main types:

- **manipulator robots:** they consist of a fixed base. Their structure consists of a sequence of links that are connected through a certain number of joints as in Figure 2.1.

- **mobile robots:** they consist of a mobile base that moves freely in the space and they are often used in applications that require autonomy. A further subdivision is made in *wheeled*, that is, they consist of a rigid mobile base and a system of wheels in motion with respect to the ground, or *legged*, that is, made up of many rigid bodies linked together through joints whose ends called feet periodically come into contact with the ground to carry out the movement. The latter, very often mimic living organisms.

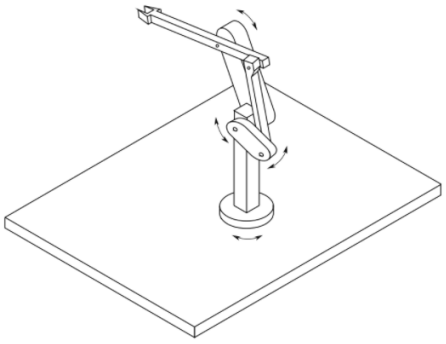

Figure 2.1: Example manipulator robot
[1]

2.2.1 Conventional Wheeled mobile Robots

There are three different types of conventional wheels for these types of robots (Figure 2.2) [1]:

- *fixed wheel:* the wheel is attached to the chassis and this allows a constant orientation of the chassis. The wheel can rotate around an axis that is in the center of it.
- *stereable wheel:* consists of two rotation axes of which the first is similar to the fixed one, while the second has a vertical direction and goes through the center of the wheel. This allows for a change of orientation with respect to the chassis.
- *caster wheel:* consisting of two rotation axes like the previous wheel but with the difference that the vertical one is displaced by a certain offset with respect to the center of the wheel. This mechanism allows the wheel to move quickly and to align with the movement of the chassis.

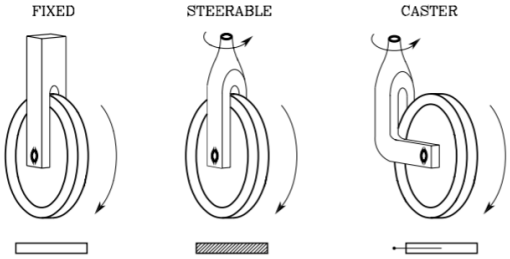

Figure 2.2: Conventional wheels [1]

Depending on the individual wheels used or by combining two or more wheels of those listed above, different kinematic structures are obtained. In particular, a *differential drive vehicle* consists mainly of two fixed wheels which are controlled separately so that the angular velocity can be easily set and one or smaller caster wheels to keep the robot balanced, acting as passive wheels.
A similar vehicle is provided using the *synchro-drive* kinematic arrangement. It consists of three aligned steerable wheels that move in a synchronized manner by two motors of which the first controls the rotation around the horizontal axis, while the second the vertical one, influencing the orientation, as in Figure 2.3.
Another type of vehicle is the *tricycle* consisting of three wheels of which the first two fixed are mounted in the rear and are driven by an engine that controls the

traction and a steerable in the front that guides the orientation.
A *car-like* instead consists of two fixed wheels at the rear and two steerable at the front as in Figure 2.4.
A robot is *omnidirectional* if it can move in any Cartesian direction.

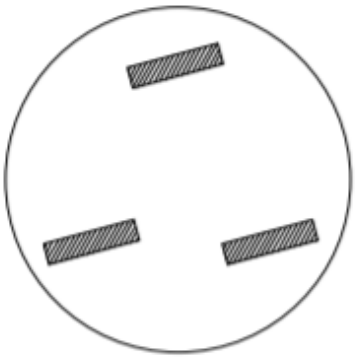

Figure 2.3: Synchro-drive vehicle [1]

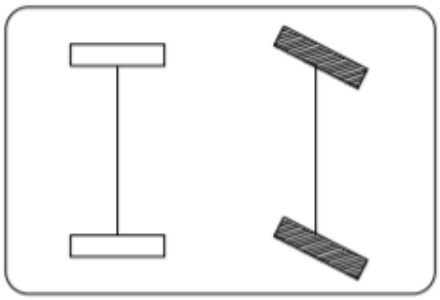

Figure 2.4: Car-like vehicle [1]

Usually, this last kind of robot is made up by four Mecanum wheels which allow movement in all directions as in Figure 2.5 without the needing of a conventional steering system.

2.2.2 Mecanum Wheeled mobile Robots

This type of wheels was invented in 1973 by the Swedish engineer Bengt Ilon and for this reason, besides being called Mecanum, they are also called Swedish wheel [3].

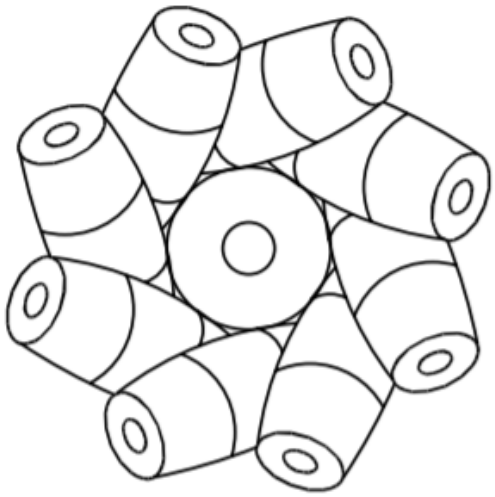

Figure 2.5: Mecanum wheel [1]

Compared to conventional wheels, they allow movement even in directions parallel to their axes and can control every degree of freedom autonomously providing movement and orientation in all directions.
Indeed, conventional wheeled vehicles such as cars are subject to *non-holonomic* constraints which prevent movement perpendicular to their driving direction. This is why, for example, to park, you need to do more maneuvers as the only movement allowed is back and forth but not sideways.
Hence, vehicles consisting of an omnidirectional wheel and therefore *holonomic constraint* have many more advantages in terms of movement than conventional ones.
In addition to the Mecanum wheels, there are two other types of wheels that move in a directional way. All of them are characterized by great flexibility that allows traction in one direction and passive motion in another. The other two types are the *universal* and *ball wheels*. The firsts are made by rollers located in the external part of the wheel, free to rotate in the direction parallel to the axis of the wheels, and they are mounted perpendicular to the axis of rotation. Instead, ball wheels use a structure that transmits power to the rollers and due to the friction, they can rotate in all directions. All three of these types together constitute the so-called *special wheels*.
Mecanum wheels are similar to the universals except for the fact that they consist of a series of rollers inclined at 45 ° with respect to the plane of the wheel. Their configuration allows having a force in the rotational direction of the wheels and a

normal one to it. Usually, four of these wheels are used whose forces due to the direction and speed of each wheel are added into a single resulting vector that allows translation and movement in any direction. The rollers are placed so that the resulting wheel is circular as depicted accurately in Figure 2.6 . The shape of the roller can be obtained by cutting a cylinder from an inclined plane of 45 ° [3].

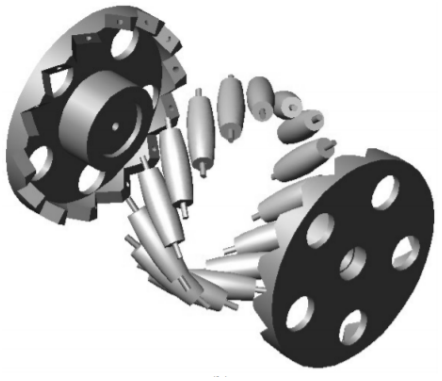

Figure 2.6: Internal wheel decomposition [3]

So the shape of wheel should have the following equation:

$$\frac{1}{2}x^2 + y^2 - R^2 = 0 \tag{2.1}$$

where R, is the outer radius of the wheel. The number of rollers, according to Figure 2.7 is obtained as:

$$n = \frac{2\pi}{\phi} \tag{2.2}$$

with:

$$\phi = 2arcsin\frac{L_\gamma}{2Rsin\gamma} \tag{2.3}$$

where L_γ is the length of the rollers:

$$L_\gamma = 2R\frac{sin\frac{\phi}{2}}{sin\gamma} \tag{2.4}$$

the size of the wheels considering γ=45° is given by :

$$L_r = 2\sqrt{2}Rsin\frac{\pi}{n} \tag{2.5}$$

$$l_w = 2Rsin\frac{\pi}{n} \tag{2.6}$$

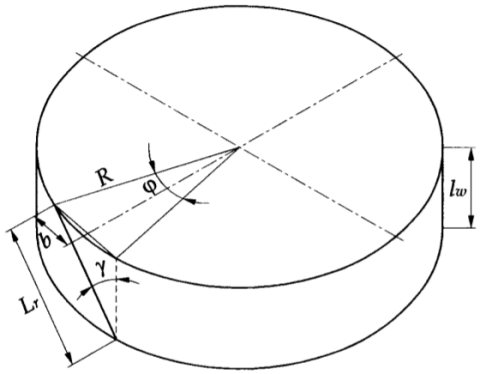

Figure 2.7: Internal wheel decomposition [3]

The result is a wheel that with little friction can move along any trajectory. It also has 3 DOFs divided into wheel rotation, roller rotation, and rotation slip on the vertical axis. The speed of the wheels can be divided into passive directions perpendicular to the axis of the roller and active that is along the axis of the roller in contact with the ground. This type of wheel allows and helps in case of difficult maneuvers in tight environments.
Considering a $x_sO_sy_s$ frame attached to the robot chassis as in Figure 2.8, the following speed equations represent the forward kinematics of the robot:

$$\begin{bmatrix} v_x \\ v_y \\ \omega_z \end{bmatrix} = \frac{R}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & -1 & 1 \\ -\frac{1}{l_1+l_2} & \frac{1}{l_1+l_2} & -\frac{1}{l_1+l_2} & \frac{1}{l_1+l_2} \end{bmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{bmatrix} \tag{2.7}$$

R is the radius of the wheels, ω the angular velocity and l_1 and l_2 the distance between the wheel axis and the center of the body. If the robot speed is set, using the inverse of the speed, and so the inverse kinematics, the angular speed of each wheel can be computed as:

$$\begin{bmatrix}\omega_1\\ \omega_2\\ \omega_3\\ \omega_4\end{bmatrix} = \frac{1}{R}\begin{bmatrix}1 & 1 & -(l_1+l_2)\\ 1 & -1 & (l_1+l_2)\\ 1 & -1 & -(l_1+l_2)\\ 1 & 1 & (l_1+l_2)\end{bmatrix}\begin{bmatrix}v_x\\ v_y\\ \omega_z\end{bmatrix} \tag{2.8}$$

The longitudinal velocity is represented by:

$$v_x(t) = (w_1 + w_2 + w_3 + w_4)\frac{R}{4} \tag{2.9}$$

The lateral one is:

$$v_y(t) = (w_1 - w_2 - w_3 + w_4)\frac{R}{4} \tag{2.10}$$

Finally, the angular velocity is:

$$\omega_z(t) = (-w_1 + w_2 - w_3 + w_4)\frac{R}{4(l_x + l_y)} \tag{2.11}$$

A limitation of this type of wheel is due to the fact that they are subject to slippage

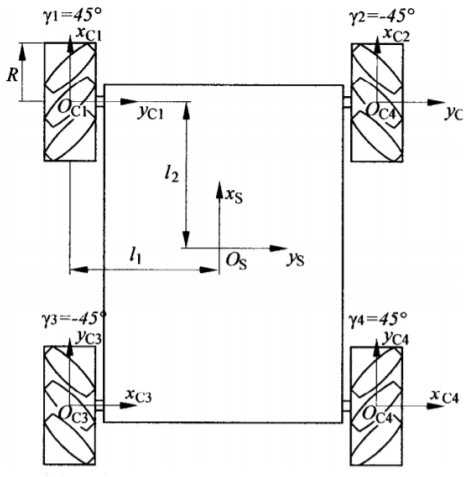

Figure 2.8: Model of four wheeled robot [3]

and therefore with the same number of rotations of the wheels lateral distances

that are different from the longitudinal ones are covered, and this difference can vary even in defence of the type of ground with which they interact. Slippage is caused by the fact that although each wheel has only one roller contact, sometimes two roller points are in contact with the ground, and due to the angular position of rollers and curved surface, a point contact or a tiny area of contact exist between ground and roller that cause slippage.
Furthermore, when the robot proceeds at high speed another problem is given from vibration. So, main problems are given from rollers and are noticed when the frontal or back wheel rotates in opposite direction [4].

Chapter 3

Sensors

The autonomous mobile robots move in the environment by acquiring data from the sensors that provide them the perception of the environment in which they move.
They represent a very important element and they can be mounted on the robot or in the environment in which they move.

Usually, for autonomous mobile robots the sensors are mounted directly on the robot itself ensuring easy adaptation even if the environment or industrial space in which they move should change. In fact, this represents a primary difference with AGVs because usually for the latter, the sensors are mounted in the environment and this constitutes a great limitation in the event that a change in the environment in which they move should be made.
So an AMR has a flexibility that allows it to navigate unpredictable environments, build a model of the environment and locate itself within it.

Each sensor usually has its own limits. The depth cameras can have difficulties in in an environment where it is difficult to see the depth, as well as the LiDARs, have limits in case they should encounter glass along the path because they cannot detect it and furthermore in outdooor application the result could also be influenced by atmospheric factors like rain, snow, and fog.
Therefore, to develop an autonomous driving system, sensors should be selected based on their use, the environment in which they should move and on the types of obstacles to be seen. Not a single sensor is usually employed but often more sensors are used together so that the limit of one is compensated by the usefulness of the other and vice versa.
In this way it is possible to develop a completely autonomous robot capable of moving freely in various spaces, correctly avoiding obstacles, and recognizing or collaborating with the humans encountered along the way.

Sensors are classified according to their use and are mainly divided into two categories [1]:

- ***Proprioceptive sensors:*** measure and evaluate the internal state of the robot giving information about position and velocities.
- ***Exteroceptive sensors:*** provide the knowledge of the surrounding environment. The goal of this typology is that specify the features that characterize the interaction of the robot with the surrounding environments so that intensify the capability of the Robot to be autonomously guided. Typical ones are represented by proximity, range, and vision sensors.

Coupling more sensors belonging to the same or different categories, it is possible to enhance their actions. In particular, by installing both typologies of proprioceptive and exteroceptive sensors, the capability of the robot can be considerably increased, guaranteeing the use of the robot not only for indoor but also for outdoor application.

3.1 Exteroceptive Sensors

They are represented by sensors that detect objects in the environment and give a measurement of the distance from the robot along a direction. They are classified in:

- ***Distance sensors:*** they give an estimation of the distance of the robot along a direction and they are also known as *range sensors*.
- ***Proximity sensors:*** they are a simplified version of the first sensor above cited that reveals the presence of an object in the proximity of the robot without physical contact.
- ***Vision sensors:*** they are characterized by the acquisition of images and geometric or qualitative information on the environment in which the robots operate through a device like a camera. This is capable of measuring the intensity of the light reflected by an object or an obstacle through the use of pixels, which are photosensitive elements that transform the energy of light into electrical energy. The most used devices used based on this effect are *CCD (Charge Coupled Device)* and *CMOS (Complementary Metal Oxide Semiconductor)* sensors. The main difference between them is given by the pixels of the CMOS sensors which, not being integrated systems, measure the throughput and therefore the pixels do not affect the neighboring ones, avoiding the blooming effect that affects the CCD sensors [1].

3.1.1 Sonar Sensors

They are the most known belonging to the class of range sensors and give an estimation of the distance from an object through the measure of the propagation of the sound. The acronym stands for Sound Navigation and Ranging and are widely used in robotics especially in underwater and mobile applications. They provide a very cheap solution to provide obstacles detection. Many times, several of these are installed to increase the ability to see obstacles in multiple directions. They are electrical devices that emit ultrasonic sound waves at higher frequencies (more than 20 kHz) than normal hearing, and through their echoes they measure the distance from an object. The distance from an object is obtained by measuring the travel time in which the acoustic wave covers the distance sensor-object-sensor also called time-to-flight and calculated as [1]:

$$d_0 = \frac{c_s t_v}{2} \tag{3.1}$$

where d_0 is the object range, t_v is the time-of-flight and c_s is sound speed.

The main component of a sonar measurement system is composed of a transducer that emits the impulse and circuitry for the excitation of its and the detection of the reflected signal.
It is common to use a single sensor used as transmitter and receiver and in this case, there is a certain latency time depending also on the mechanical inertia of the transducer. In Figure 3.1 there is an example of a unique transducer used for transmission and reception of echo. A transducer can convert acoustic energy into electric ones and vice versa. Common transducer are divided in *piezoeletrics transducer* and *electrostatic transducer*. Both of them can operate as transmitters and receivers.
However, the *piezoelectrics*, whose properties derive from the fact that they exploit the properties of crystal material, have low efficiency because, being of the resonant type, they have mechanical inertia that limits the minimum detectable range; this problem is solved using two different transducers for transmitting and receiving. Instead, the *electrostatic* ones, working as capacitors, can operate at different frequencies, have low mechanical inertia, large bandwidth, and great sensitivity. Despite this, they require a bias voltage which complicates the electronic control [1].
Although ultrasonic sensors are widely used for their low cost, high reliability, lightweight, and low power consumption, they have some limitations that prevented their use in some robotic applications. The main problems are related to angular and radial resolution, and hence on the maximum range that can be achieved. They are unable to determine the relative angle of an obstacle.
Furthermore, obstacles with large acute angles could cause the waves to bounce in

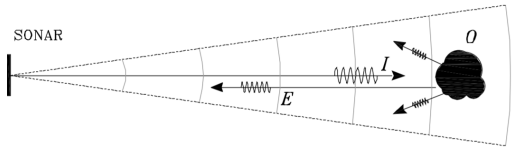

Figure 3.1: Sonar sensor working principle [1]

a different or even opposite direction to the real one, and therefore they will not see any obstacles. In addition, walls that appear very smooth with steep corners or glass surfaces give false measurements with this type of sensor. For many of the problems above, a good solution can be to use several sonar sensors and do a sort of cross-checking to distinguish false measurements from true one [5].

Other problems are related to the frequency. In fact, the radiation amplitude decreases as the frequency increases, obtaining a better angular and radial resolution, but excessively high frequencies lead to absorption phenomena on the part of the surface that generates the echo. Some of these problems can be overcome by using different transducers for emission and reception.

3.1.2 Camera

A camera is a much more complex system than devices based on photosensitive sensors and consists of many more elements such as a shutter, a lens responsible for focusing the light reflected by the object, and analog preprocessing electronics. The system using the camera can have a single camera or more than one. If several cameras are used to observe the same object, obstacle, or environment, it is possible to obtain information on the depth by estimating their distance from the vision system. In this case, we speak of *3D vision or stereo vision*, allowing to observe the same scene from different angles. However, three-dimensional vision can also be obtained with a single camera and obtaining images from different positions by estimating the depth through geometric characteristics.

An example of a stereo camera widely used in robotic applications is the Microsoft Kinect, which represents a cutting-edge and low-cost vision sensor. It was mainly invented for video games and consists of a horizontal bar mounted on a motorized pivot [6]. Furthermore, it is made up of several hardware (Figure 3.2) whose fundamental elements are represented by an RGB camera coupled with a depth sensor which constitutes an infrared projector combined with a CMOS sensor

that captures video in 3D and acquires images with 640X480 resolution, and also four- microphone arrays that provide facial and voice recognition capability as well as capture the three-dimensional motion of the entire body [7]. However, despite

Figure 3.2: Microsoft Kinect sensor [7]

its wide use in robotic applications, it has limits of accuracy and reliability. In fact, for optimal functioning, it is necessary that the object or the person to identify is exactly in front of the sensor and at a certain distance (should be 1.8 meters between user and Kinect, while for many people at least 2.5 meters of distance from the sensor), because if used in narrow or crowded environments, it cannot distinguish and map images well. Another limitation is given by light, therefore it is advisable to use it in environments with ample artificial light as sunlight worsens the quality of the information acquired.

A very good sensor based on camera is the Intel® RealSense™ depth camera D435i which combines the robust depth-sensing capabilities of the D435 with the addition of an inertial measurement unit (IMU). It is described in the sixth chapter dedicated to the hardware used for the implementation of the autonomous navigation system.

3.1.3 Laser Sensors

Laser sensors are often the preferred ones, thanks to their better characteristics: infrared rays can be used unobtrusively, they focus to give narrow beams and they do not disperse from refraction.
They can be of two types, according to the way of obtaining the measurement of the distance from the sensor [1]:

- *time-of-flight:* the distance measurement is obtained by multiplying the time it takes for the pulse of light to go from the source to the target and then to the detector at the speed of light. The limitations of this sensor are on

accuracy and are not technological but on cost: they are based on the minimum observation time, the minimum observable distance, and the temporal accuracy of the receiver.

- *triangulation laser:* the triangulation principle is based on trigonometric properties of triangles such as the cosine theorem. The method used is to find two angles and one side of the triangle and finally find the missing sides. The laser beam emitted by a photodiode is projected onto an observed surface. When this is reflected, it is focused on a CCD sensor. The position of the focused beam reflected at the receiver creates a signal proportional to the distance of the transmitter from the object. Once the position and orientation of the CCD sensor with respect to the photodiode have been obtained, through a calibration procedure it is possible to obtain the distance from the object through geometric calculations. Limits on accuracy are given by surfaces that do not favor reflection and are subject to color changes.

LiDAR sensor

The most used laser sensors in robotic applications are laser distance sensors called LiDAR. The term means ***Light Detection and Ranging*** and works as a distance measuring device by measuring distance through a series of points and using a contactless measurement process avoiding the need to apply any mechanical on the measured target. Compared to vision sensors, they have the advantage of being able to work indoors or outdoors in different atmospheric conditions, being less sensitive to interference from light.
According to the operating principle, they are distinguished in [8]:

- *1D sensor:* they are used to process only one dimension (distance) linearly and they are directed toward a certain target.
- *2D sensor:* they are used to provide an indication of distance and angle and usually the beam is moved or rotated on one level.
- *3D sensor:* they are pivoted and in this way, information is provided on all three axes.

The fundamental pieces of a LiDAR sensor are represented by the sender and the receiver, together with their high temporal resolution. The laser beam must be directed towards the object without interference in its path; only a small part of the reflected light, which depends on the properties of the objects or surfaces hit, usually reaches the LiDAR sensor receiver. The LiDAR sensor is used to measure the distance, it calculates the shortest distance from the object guaranteeing a great

advantage, because it prevents deflections, but at the same time a disadvantage in the case of glass surfaces causing interference in the measured value.

3.2 Proprioceptive sensor

Compared to *exteroceptive sensors* that directly measure the environment in which the robot moves, proprioceptive sensors measure the internal state of the robot. Various categories belong to this type, including the position and speed sensors among which the most important are given by the encoders or the attitude sensors such as Inertial measurements units (IMU) and accelerometers.

3.2.1 Encoder

An *encoder* is an electro-mechanical device used to obtain information on speed, direction, and position. It is used to convert the position or motion of a mechanical part into analog or digital signals. There are usually two types of encoders: linear and rotary. The former measure motion along a linear path, while the latter relates to rotational motion. Both of these two types use magnetic principles.
In particular, in the linear ones there is a magnetic sensor that passes on a magnetic scale detecting the magnetic changes that are proportional to the speed and to the measured displacement of the sensor. On the other hand, in magnetic rotary encoders, the sensor passes on a rotating disk of alternating regions detecting the small changes in the magnetic field due to the Hall effect or the magnetic resistive effect. As for the electronic rotary encoders, they are controlled through the rotation of a shaft connected to the encoder circuit. A further division of encoders is given by the *incremental* and *absolute* types [9].
Absolute encoders consist of a rotating disk made of circles, called tracks, each one having a sequence of transparent and opaque sectors. A beam of light is emitted in correspondence of each of them and it is intercepted by photodiodes or photoreceivers, which transform the light pulses into electrical pulses and are finally transmitted by the output electronics. Furthermore, they maintain their position even if the power is removed and the initial position does not have to be sought every time they are switched on. To encode the position, simple binary encoding is not used because it could incur problems in case of instantaneous and simultaneous variations. The Gray coding is used instead, which represents a variant of the binary code, as it has the particularity that only a single bit changes its state during the transition between two consecutive codes.
Incremental encoders are similar to absolute ones, because they always consist of an optical disc with two tracks whose transparent and opaque sectors, but this time they are arranged in quadrature. The direction of rotation is determined thanks to the 90-degree phase shift of the two traces, and the displacement is

measured by counting the pulses; a third trace is often used to define the position of absolute zero, and therefore as a reference system for the angular position. Compared to the absolute ones where the angular position is directly memorized in the optical disc, the incremental ones estimate the absolute position through suitable electronic circuits and therefore the information is available on volatile memories, making them subject to some possible disturbances. Furthermore, by using external circuitry it is also possible to obtain the speed.
Finally, the incremental ones are more used than the absolute ones because they have a lower cost and are simpler [1].

3.2.2 Inertial Measurement Unit

An *inertial measurement unit (IMU)* has been commonly used in robotic applications since ancient times. The first use dates back to 1930, where it was used only in some applications due to its size, consumption, and high costs. Later, when it became more compact, it was used extensively in many fields. It is mainly a device used to determine the movement then relative position, speed, and acceleration of a mobile system. Initially, this device consisted of an accelerometer to measure inertial acceleration and a gyroscope to estimate angular rotations. In particular, the accelerometer measures the rate of change of speed of an object in meters per second (m/s^2) or in gravity (g), and therefore the linear acceleration, and it accumulates drift and noise errors. Instead, the gyroscope has the advantage of being fast in measurements and capable of tracking fast movements, and it is used to measure angular velocity in degrees per second $(\circ/s)$ or Revolution Per Second. However, it accumulates errors when used for a long time [10]. Using both sensors gives this device six degrees of freedom corresponding to the x, y, and z axes. Both provide angle measurements and have the advantage of being error-free when near ferromagnetic materials are used. Later, the magnetometer was also introduced to estimate the magnetic direction, giving a device of this type 9 degrees of freedom. The magnetometer provides yaw angle rotation measurements and is useful in the case of dynamic orientation calculations. Usually, a suitable device is chosen based on the applications to be run. The aspects to be taken into account are the size of the device, the accuracy of the data that can be improved through the use of appropriate filters useful for calibrating the data coming from accelerometers and gyroscopes, the degree of response rate and the degree of freedom.
In navigation systems, GPS was frequently used and therefore the IMU provides an advantageous substitute for GPS where it is not possible to use it, such as in internal navigation environments, tunnels or in case of electronic interference. In addition, many navigation applications use the IMU to obtain information about orientation and encoders for position [11].

3.3 Sensor fusion

Many robotic applications use different sensors to improve the accuracy, robustness, and efficiency of the system, but on the other hand, the use of more sensors leads to the construction of more complex hardware and software to fuse the data obtained from multiple sensors.

Using a single sensor could lead to the accumulation of errors and noise over time. Instead, the use of different sensors is useful to have different results and cover a much wider range of measurements, reducing the set of uncertain interpretations of the model.

However, to have greater clarity of the results coming from different sensors, to cancel out the noise and uncertainty, and to have better accuracy, reliability, and resolution, some fusion algorithms must be applied.

The most used algorithms for sensor fusion are the Kalman Filter (KF), Extended Kalman Filter (EKF), Unscented Kalman Filter (UKF), and Particle Filter (PF) that are divided according to deal with linear and non-linear models. These algorithms are a set of mathematical equations to estimate the state of a process. The state estimation methods are used to ensure the state of a continuously changing system. State estimation phase is often used in data fusion algorithms, whose purpose is to acquire a global target state from observations [12].

3.3.1 Kalman Filter

The Kalman Filter is widely used in different fields due to its computational skills, its simple form, and its great performance when the uncertainty is not very high. It is used to estimate and improve the unknown state of a system and it is known as the easiest one to be applied to linear systems.

It processes the state of the previous time steps with the current measurement to calculate the estimate of the current state. Kalman filter state equations are a linear representation of w_k, u_{k-1} and v_k [13]:

$$x_k = Ax_{k-1} + Bu_{k-1} + w_k \tag{3.2}$$

Observation equation is a representation of x_k and v_k:

$$z_k = Hx_k + v_k \tag{3.3}$$

where x_k is the state vector, z_k is the observation, A is the status transition matrix, H is the observation matrix,w_k is the system noise vector,u_{k-1} the system control

vector and v_k is the observation noise vector. The process and measurement noise vectors that are represented by w_k and v_k are assumed to be positive definite, symmetric, and zero-mean Gaussian white noise vector, satisfying:

$$E(w) = 0, cov(w) = E(ww^T) = Q \tag{3.4}$$

$$E(v) = 0, cov(v) = E(vv^T) = R, E(wv^T) = 0 \tag{3.5}$$

the states that are obtained at $k-1$ and k are the prior state estimation $\hat{x}^-$ and the posterior one. The prior and posterior estimation errors and the covariances of prior and posterior states are :

$$e_k{}^- = x_k - \hat{x_k}{}^-, e_k = x_k - \hat{x_k} \tag{3.6}$$

$$P_k{}^- = E[e_k{}^- e_k{}^{-T}], P_k = E[e_k e_k{}^T] \tag{3.7}$$

The goal is to calculate the a posteriori state estimate $\hat{x_k}$ as a linear combination of an a priori estimate $\hat{x_k}{}^-$ and a weighted difference between an actual measurement z_k and a measurement prediction $H\hat{x_k}{}^-$ [14]:

$$\hat{x_k} = \hat{x_k}{}^- + K_k(z_k - H\hat{x_k}{}^-) \tag{3.8}$$

Hence, prediction equations are:

$$\hat{x_k}{}^- = A\hat{x_{k-1}} + Bu_{k-1} \tag{3.9}$$

$$P_k{}^- = AP_{k-1}A^T + Q \tag{3.10}$$

Instead, the updated equations are represented by:

$$K_k = \frac{P_k^- H^T}{HP_k^- H^T + R'} \tag{3.11}$$

$$P_k = (I - K_k H)P_k^- \tag{3.12}$$

where K_k is the Kalman gain matrix, $\hat{x_k}$ is the optimum state filter vector, P_k is the process covariance matrix and I is the identity matrix.
It must be underlined that the Kalman Filter can be used only for applications that are linear with Gaussian noise.

3.3.2 Extended Kalman Filter (EKF)

The Extended Kalman Filter is applied to nonlinear systems. This algorithm is based on the nonlinear first-order Taylor expansion around the state of the estimates and subsequently transforms the nonlinear system into linear. Furthermore, the result will be close to the exact result only if the observation and state equations become linear and continuous.

Moreover, the covariance matrix of system status and observation noise remains unchanged in the EKF, but if they have not been accurately estimated, the error will cause the filter to diverge.
A non-linear system can be represented by [13]:

$$x_k = f(x_{k-1}, w_{k-1}) \tag{3.13}$$

$$y_k = h(x_k, v_k) \tag{3.14}$$

where x_k represents the n-dimension state vector and y_k is the m-dimension observation vector , w_{k-1} and v_k are the process and measurement noise. The transition matrix f and the observation matrix h are non-linear function.
The *prediction equations* are given by:

$$A = \frac{df}{dx}|_x = \hat{x}_{k-1}, x_k^- = f(\hat{x}_{k-1}) \tag{3.15}$$

$$P_k^- = AP_kA^T + Q \tag{3.16}$$

where A represent the state matrix.
The *update equations* are:

$$H = \frac{dh}{dx}|_x = \hat{x}_{k-1}^- \tag{3.17}$$

$$K_k = \frac{{P_k}^- H^T}{H{P_k}^- H^T + R} \tag{3.18}$$

$$\hat{x}_k = \hat{x}_k^- + K_k(y_k - h(\hat{x}_k^-)) \tag{3.19}$$

$$P_k = (I - K_kH)P_k^- \tag{3.20}$$

where H, K, and P_k represent, respectively, the measurement equation, the Kalman gain matrix, and the process covariance matrix.

However, there are a series of problems with this filter that are given from the fact that the EKF assumes that status and observation noise are independent in white noise processing but the characteristic of noise could be different. Another problem is that there could be a change in process noise after a first-order Taylor series expansion leading to the wrong assumption for noise that is inconsistent with reality.

3.3.3 Unscented Kalman Filter (UKF)

The Unscented Kalman Filter is considered an improvement concerning the EKF in terms of performance and complexity. It is simpler because it uses the sampling strategy instead of the random ones, avoiding divergence errors.
Moreover, it approximates the weighted density distribution of non-linear function. The result will be better accuracy and convergence.

In this kind of filter, the sampling points are called *Sigma-points* and are in a small number. The most used sampling strategy is the 2n+1 symmetric-sigma. The idea of Unscented Transform is: assuming that the sampling mean is indicated with x and the covariance with P_x, we must select a set of points (Sigma point set) and apply the non-linear transformation to each Sigma sampling point. Then, it is possible to obtain the non linear transformed set of points y and P_y which represent statistics point of Sigma after the transform [13].
Filter steps are:

- initialize state error covariance matrix and state vector;
- select sigma sampling points according to the state vector and error covariance, and calculate the weighted values;
- calculate mean and covariance through the equation of states updating time through the taken sampling point;
- finish measurement update through a nonlinear observation equation by the selected sampling points;
- update Kalman Filter coefficients;

The selected sigma points are described as in the following:

$$x_0 = \hat{x}_k, x_i = \hat{x}_k + (\sqrt{(n+\Delta)P})_k, x_{i+n} = \hat{x}_k - (\sqrt{(n+\Delta)P})_k, i = 1, ..., n, \quad (3.21)$$

$$W_0^m = \frac{\Delta}{(\Delta+n)}, W_i^m = \frac{1}{2(\Delta+n)}, i = 1, ...,2n, \quad (3.22)$$

$$W_0^c = W_0^m + (1 - \alpha^2 + \beta), W_i^c = \frac{1}{2(\Delta + n)}, i = 1, ...,2n, \tag{3.23}$$

$$\Delta = \alpha^2(k + n) - n, \tag{3.24}$$

where Δ is the scaling constant, α is the spread of selected sigma point, k is the second scaling constant and is widely considered as zero, β represents status variables, W_i^m and W_i^c are respectively, the weighted sample mean and covariance. Time update is defined as:

$$\epsilon_i = f(x_i), \tag{3.25}$$

$$\hat{x}_{k+1/k} = \sum W_i^m \epsilon_i \tag{3.26}$$

$$P_{k+1/k} = \sum W_i^c (\epsilon_i - \hat{x}_{k+1/k})(\epsilon_i - \hat{x}_{k+1/k})^T, \tag{3.27}$$

in which ϵ is referred to the function of non-linear system, $\hat{x}_{k+1/k}$ is the prior state estimation. Instead, the measurement updates of UKF:

$$Z_i = h(\epsilon_i), \tag{3.28}$$

$$\hat{z}_{k+1/k} = \sum W_i^m Z_i \tag{3.29}$$

$$P_{zz} = \sum W_i^c (Z_i - \hat{Z}_{k+1/k})(Z_i - \hat{Z}_{k+1/k})^T, \tag{3.30}$$

$$P_{xz} = \sum W_i^c (\epsilon_i - \hat{x}_{k+1/k})(\epsilon_i - \hat{z}_{k+1/k})^T, \tag{3.31}$$

Filter update:

$$K_{k+1} = P_{xz} P_{zz}^{-1}, \tag{3.32}$$

$$\hat{x}_{k+1} = \hat{x}_{k+1/k} + K_k(y_{k+1} - \hat{z}_{k+1/k}) \tag{3.33}$$

$$P_{k+1/k+1} = P_{k+1/k} - K_{k+1} P_z z {K^T}_{k+1}, \tag{3.34}$$

K is the Kalman gain and $\hat{x}_{k+1}$ is posterior state estimation.
This last filter avoids the calculation of Jacobian and Hessian matrices, so, it is better and simpler but slower than EKF using $2n + 1$ points.
In conclusion, the Kalman filter is limited because more applications, in reality, are non-linear and it requires a Gaussian white noise.
The EKF has the limitations that the estimated value is close to the true one only

if system state and observation equations are continuous and linear; in addition, since the accumulating error is accumulated it could diverge if noise covariance matrices are not good enough.
Finally, the UKF is similar to the EKF in terms of calculation, but avoids the divergence phenomenon and it is more accurate and provides a faster convergence [13], [15]. Figure 3.3 sketches the differences between the two most used filters to fuse data from sensors for non-linear applications.

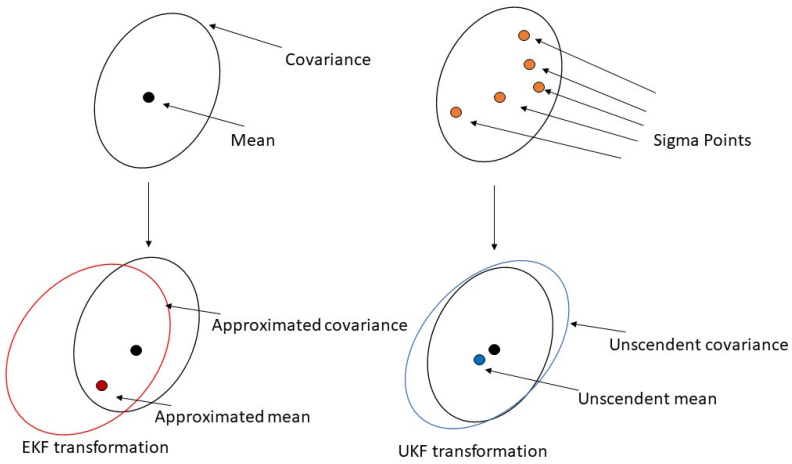

Figure 3.3: Differences between EKF and UKF

3.3.4 Particle Filters

It is a very efficient predictive tool used for systems that are non-Gaussian and to represent uncertainty in stochastic processes capable to fuse measurements from different sources.
The disadvantages of this type of filter come from the number of particles or samples, since an accurate estimate is obtained only with a large number of particles.
Particle filters are of Bayesian type and the localization approaches using them are also called Monte Carlo Localization (MCL).
The discrete state-space model is [15]:

$$P_{t+1} = g(x_t, u_t, w_t) \tag{3.35}$$

$$y_t = f(x_t) + e_t \tag{3.36}$$

where x_t is the state vector, u_t and $Y = (y_i)_{i=1}^{t}$ are input and measurement vector, respectively. The filtering density $p(x_t|Y_t)$ and the non-linear posterior prediction density $p(x_{t+1}|Y_t)$ are founded as:

$$x_{t+1}|Y_t = \int_{R^n} p(x_{t+1}|x_t)p(x_t|Y_t)dx_t \tag{3.37}$$

$$p(x_t|Y_t) = \frac{p(y_t|x_t)p(x_t|Y_{t-1})}{p(y_t|Y_{t_1})} \tag{3.38}$$

The weight and location assigned to particle represents the density with an estimation of $p(x_t|Y_t)$ with a large set of weighted $\psi_t^{(i)}$ samples $(X_i^{(i)})_{(i=1)}^{N}$ and the sum of all weight is equal to one. The location and weight of each particle are updated at each measurement and they can be used to solve the Bayesian equation performing the re-sampling to avoid divergence. To calculate the PF the following steps are performed:

- generate N samples $(X_i^{(i)})_{(i=1)}^{N}$ from $p(x_0)$
- compute and normalize weights $\psi_t^{(i)}$
- generate predictions
- increment t and recompute weights

The minimum mean square estimate is chosen to estimate t as:

$$\hat{x}_{t|t} = E(x_t) = \int_{R^n} x_t p(x_t|Y_t)dx_t \approx \sum_{i=1}^{N} \psi_t^{(i)} X_t^{(i)} \tag{3.39}$$

Moreover, the particle filter is very useful because the noise can be considered even to be non-Gaussian, it is simple and acts in a good way in presence of large noise. Drawbacks are given by the fact that it is computationally expensive compared to the previous filters with a complexity that increases with the size of the vectors.

Chapter 4

Path planning Algorithms

Navigation is one of the most important topics in mobile robotics. Over the years, several algorithms have been developed to provide autonomous and safe navigation in the environment in which a robot moves.

Furthermore, besides autonomous navigation, another problem of great importance is given by the ability of a robot to avoid or interact with the dynamic or static obstacles it encounters in its path.
For this reason, global and local navigation algorithms are distinguished.
The first refers to navigation for the robot in a known environment where it moves by selecting the path capable of avoiding obstacles.
For this type of navigation, the robot can move easily knowing its initial position on the map, the final position to be reached, and the obstacles along its path.
Instead, by local navigation, we mean the ability to move in an unknown environment through the information acquired by the sensors with which it is equipped that allow to avoid obstacles and therefore collision with them.

One of the first methods used for the global path planning, and so for the navigation in a priori known environment, computing off-line the shortest path to go from an initial point to another, is the *Dijkstra algorithm* [16].
Succesively, other algorithms were developed as the *A* algorithm* [17] and the *D* algorithm* [17]. The last one is an evolution of the *A* algorithm*.

The most used methods that provide autonomous robot navigation in an environment with obstacles, providing a local navigation system and having the advantage of constantly replanning the path every time the robot meets obstacles, can be distinguished in directional and velocity space-based approaches.
The first ones give a direction to the robot and they are divided into *Potential Field Method (PFM)* [18], *Virtual Force Field* [19], *Vector Field Histogram (VHF)*

[20] and *VHF+* [21].
The second ones give the robot the ability to manage with rotational and translational velocity controls; the most used is the *Dynamic Window Approach (DWA)* [22].

Some of the fundamental algorithms used for global and local navigation are described in the following sections.

4.1 Global path methods

The global planner calculates the shortest path to reach a target position from the starting position.
To calculate the route, it needs to know totally or partially the environment in which the robot moves. Therefore, route calculation takes place off-line only if map knowledge is available and tends to fail if the area is unknown.

The algorithms for the search and planning of the global path guarantee a good solution if applied in both internal and external environments, providing an optimal path capable of going from a starting point to an endpoint avoiding collision with obstacles in the path.

One of the most used algorithms is the Dijkstra [16]. It is used for its simplicity, calculation speed, and good performance obtained in the experiments in which it is applied.

Other algorithms developed later and widely used for the global path planner are A* and D*. Both minimize their cost function and easily and quickly re-plan the path to reach the final position.

4.1.1 Dijkstra's algorithm

This algorithm, developed by the computer scientist Edsger W. Dijkstra in 1956 and published in 1959 [23], is now present in many variations and used to find the shortest path between nodes in a graph with weighted edges.

A weighted graph is composed of vertices and edges that have a value or weight associated with them. The weight indicates the cost to move from one vertex to the other. The algorithm is divided into some steps to find the shortest path between any couple of vertices, building a set of nodes that have a minimum distance from the source.

The graph contains vertices, or nodes indicated with v or u, and weighted edges that represent the connection between nodes; an edge is denoted as *(u,v)* and *w(u,v)* indicated as the weight.

The following elements have to be initialized:

- Q, which represents a queue of all nodes; and at the end of the algorithm it will be empty,
- S, which is initially an empty set, which will contain the marked nodes that have been already visited; at the end of the algorithm it will contain all nodes of the graph,
- *dist*, which is an array of distances that goes from the source node s to the other ones in the graph. At the beginning, *dist(s)=0* and all the other ones indicated with v are initialized as *dist(v)*=∞. This is set at the starting point so that the distance from each node will be recalculated while the algorithm proceeds and ended when the shortest path is found.

After that, the algorithm proceeds as follows:

1. if Q is not empty, pop the node v that is not present in S, from Q with the smallest *dist(v)*. At the first run of the algorithm, the node s will be chosen. Subsequently the one with the smallest *dist* will be selected,
2. the node v has to be added to S to marke that it has been visited,
3. update *dist* value of adjacent nodes of the current node v in a way that for each new node u:
   - if $dist(v) + weight(u, v) < dist(u)$ update *dist(u)* to the new minimal distance value because a new ones has been found
   - otherwise *dist(u)* is not updated.

In the end, when the algorithm has visited all nodes in the graph, the smallest distance to each node has been found with *dist* containing the shortest path from the source s [16]. The pseudocode for Dijkstra's algorithm is described in **1**. Despite performing an excellent solution to find the shortest path between two vertices, Dijkstra's algorithm could be slow due to the calculation of unnecessary paths, with a sort of "blind" search. For this reason, this algorithm has been modified to speed up the calculation.

Figure 4.1 shows an example of a weighted graph marked with edge costs to which can be applied Dijkstra's algorithm [24]. In the first step, vertex A is the

Algorithm 1 Dijkstra's algorithm

```
1: function DIJKSTRA(Graph,source:)
2:     dist[source] ← 0
3:     for each vertex v in Graph do
4:         if v ≠ source: then
5:             dist[v]← infinity
6:         end if
7:         add v to Q
8:     end for
9:     while Q is not empty: do
10:        v ← vertex in Q with min dist[v]
11:        remove vfrom Q
12:        for each neighbour u of v: do
13:            alt ← dist[v] + length(v,u)
14:            if alt < dist[u]: then
15:                dist[u]← alt
16:            end if
17:        end for
18:    end while
19:    return dist[]
20: end function
```

first node examined, because it has minimal distance zero to the source, and all the others distances between any two vertices are set to infinite. So, A is moved from the queue Q to set S. Then, the distance between A and its neighbors B, C, D is updated to 2, 5, 1. Subsequently, the other nodes that have a minimum distance from the source are analyzed by repeating the procedure that has been done for vertex A. Finally, when the set Q is empty, and therefore all the nodes have been visited, the final shortest path is found.

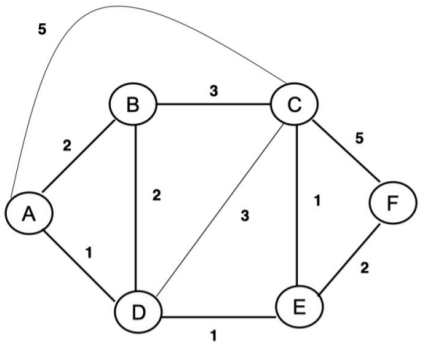

Figure 4.1: Example of weighted graph
[24]

4.1.2 A* Algorithm

This algorithm represents an evolution compared to Dijkstra's algorithm because it achieves better performance through the addition of heuristics for the path search.

The algorithm was published in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael of the Stanford Research Institute [25].

It is widely used for its simplicity and quick solution. It is formulated in terms of weighted graphs: starting from a node of a graph, it finds and follows a path to the node with the lowest known cost keeping a priority queue of alternate paths along the way. When it traverses the graph and encountered a segment of the path with a cost lower than what it is going through, it changes immediately and the process continues until the goal is reached.

So, A* works using the best-first search and finds a least-cost path from an initial node to the final ones and uses the distance-plus-cost heuristic function to choose the order in which the cheapest nodes are searched in the tree.

The distance-plus-cost heuristic function, usually denoted with *f(x)*, is:

$$f(x) = g(x) + h(x) \tag{4.1}$$

It is composed of the sum of the path-cost function which represents the cost from the starting node to the current node and denoted with g(x) plus an admissible heuristic estimate of the distance to the goal denoted with h(x). This last function must be an admissible heuristic, and for a kind of application like routing, h(x) represents the straight-line distance, i.e. , the smallest distance between two points or nodes.

Moreover, h is monotone or constant if for every edge x,y of the graph the following condition is satisfied:

$$h(x) \leq d(x, y) + h(y) \tag{4.2}$$

where d is the length of the edge. In this case, the algorithm becomes faster and powerful and there is no need to explore nodes more than once. Hence, A* is equivalent to Dijkstra's algorithm with the reduced cost:

$$d'(x, y) := d(x, y) - h(x) + h(y) \tag{4.3}$$

Moreover, the time complexity of this algorithm depends on the heuristic. In the worst case, it could be an exponential expansion of nodes in the length of the solution, in the best case it has a polynomial trend when the search space is a tree, there is a single goal state, and the heuristic function h meets:

$$|h(x) - h^*(x)| = O(log h^*(x)) \tag{4.4}$$

in which h* is the optimal heuristic that represents the exact cost to get from x to the goal. So the error of h will not grow faster than the logarithm of h* that returns the true distance [17]. Figure 4.2 shows an example of how to find a path from the start node to a goal node. In particular, the empty circles represent the nodes that have to be explored, whereas the filled circles represent the ones in the closed set. The green nodes represent the closest to the final goal.

4.1.3 D* Algorithm

This algorithm comes in three different versions and the name stands for *Dynamic A* because it behaves like the classic A* algorithm with the exception that the arc costs may change as the algorithm is executed.

The original D* algorithm, which is an informed incremental search algorithm, was

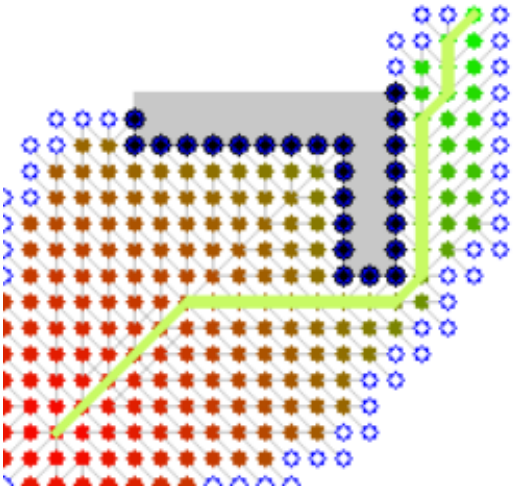

Figure 4.2: Illustration of the A* search algorithm

introduced by Anthony Stents in 1994 [26].

Although there are multiple versions of the same algorithm, they all solve path planning problems based on the same configurations, where a robot can navigate, given a final goal in unknown terrain, by finding the shortest path to reach the final coordinates starting from the initial ones.

Any new information acquired by the robot is added to the map and a new shorter route is rescheduled, if necessary, by repeating the process until it reaches the final coordinates. The re-planning of the route in the case of new obstacles takes place in a very fast and more efficient than A* algorithm.

Like the algorithms previously described, this algorithm contains a series of nodes that need to be calculated. These are denoted as *OPEN* list and can be marked as *NEW* if it has never been placed in the list, *OPEN* if it is currently in the list, *CLOSED* if it is no longer in the list, *RAISE* to indicate that the cost has increased since the last time in the list and *LOWER* in the opposite case.

The algorithm works iteratively. In particular, it takes a node from the *OPEN* list, evaluates it, propagates it, and then placed it in the list. This propagation process is called *expansion* and is different from the A* which runs the path from start to

finish. The D* starts directly from the goal node and goes backward. However, the D* requires a procedure that is computationally heavy, because it finds the path not only for the goal but also for all the nodes far from the target. Each processed node has a back pointer which refers to the next node representing the target. The algorithm ends when the next node to expand is the initial one and the path to the goal can be found by following the back pointers.

When there is an impediment along the way, all points are replaced in the OPEN list and marked as RAISE. However, before increasing the cost, the algorithm sees if it can reduce the cost of the node and if it cannot be done, the RAISE status is propagated to all the descendants who have the back pointers. After these are evaluated, they form a wave. When the RAISED node can be reduced, the back pointer is updated and the LOWER state is passed to its neighbors. Thus, some points are no longer affected by the waves and the algorithm works with points that are affected by the change in cost.

When none of the points can find a path through the neighbors, the fact of propagating their cost increases, and an alternative path can only be found outside the channel. In this way, *LOWER* waves develop which expand as unattainably marked points with new path information [17].

4.2 Local motion planning

As described above, the algorithms used for global path planning fail when the robot finds itself moving in an unknown environment that changes frequently.

So, for this purpose it is useful to use local motion planning. In this way, the robot, using some Obstacle Avoidance algorithm, acquires information from on-board sensors, and re-plans the path in real-time, avoiding the collision with the unpredictable obstacles that it encounters in its path while moving.

Obstacle Avoidance problem can be described as in [22]. In particular, it is denoted with A the robot moving in a workspace W with the configuration denoted as CS, with q_t the configuration at time t and with $A(q_t)\ \epsilon\ W$ the space occupied by the robot. Furthermore, in the robot, there is a sensor that identify a set of obstacles $0(q_t)\ \epsilon\ W$ measuring a portion of space $S(q_t)\ \epsilon\ W$.
Moreover, $u(q_t)$ is a control vector during the time δt.
So, the robot describes a trajectory $q_{t+\delta t}{=}f(u, q_t, \delta_t)$ with $\delta_t \geq 0$.
Denoting with $Q_{t,T}$ the set of configuration of trajectory from q_t with $\delta_t\ \epsilon\ [0, T]$ where T is the sampling period, the Obstacle Avoidance Problem establishes that,

using q_{target} as the final configuration, the objective is to compute a motion control u_i such that the trajectory generated avoid collision with obstacles $A(Q_{t_i}, t) \cap O(q_{t_i}) = 0$ and it makes the vehicle reach final target $F(q_{t_i}, q_{target}) < F(q_{t_i+T}, q_{target})$ as depicted in Figure 4.3, which shows a robot that avoids obstacles, acquiring information by sensors.

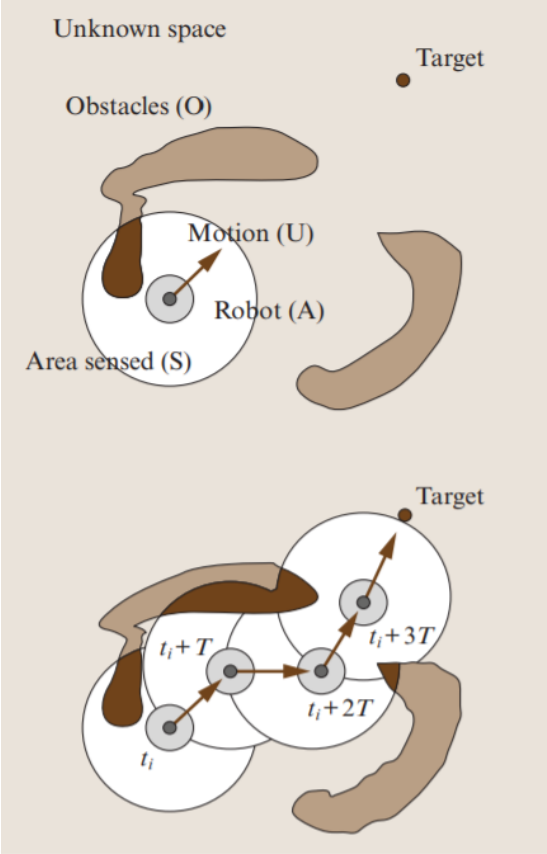

Figure 4.3: Obstacle avoidance Problem [22]

4.2.1 Potential Field Method (PFM)

This approach models the robot as a particle in space where it moves under the influence and combination of attractive and repulsive fields. This approach is widely used for its short computational time and its simplicity.
The obstacles and the final position to be reached are represented by charged surfaces that generate a force on the robot, which moves away from the obstacles

that generate a negative potential, and goes towards the final position to be reached that generates an attractive potential. The potential field is represented as an energy field and therefore, its gradient is a force [21].

Denoting with F_{att} the attractive forces generated by the final target, it can be determined as:

$$F_{att}(q_i) = K_{att} n_{q_{target}} \tag{4.5}$$

Instead, the repulsive forces F_{rep} of the obstacles, can be expressed as:

$$F_{rep}(q_i) = \begin{cases} K_{rep} \sum_j (\frac{1}{d(q_{t_i}, p_j)} - \frac{1}{d_0}) n_{pj}, \ if \ d(q_{t_i}, p_j) < d_0 \\ 0, \quad otherwise \end{cases} \tag{4.6}$$

where d_0 is the distance of obstacles p_j, q_{t_i} represents the vehicle configuration, $n_{qtarget}$ and n_{pj} are the unitary vectors pointing from q_{t_i} to the target and each obstacles p_j. This equation depends on the current robot configuration. The generalized potentials, that depend also on instantaneous robot velocity and accelerations, are found as:

$$F_{rep}(q_i) = \begin{cases} K_{rep} \sum_j (\frac{a\dot{q_{t_i}}}{(2ad(q_{t_i}, p_j) - \dot{q_{t_i}}^2)} n_{pj} \cdot n_{\dot{q_{t_i}}}, \ if \ \dot{q_{t_i}} > 0 \\ 0, \quad otherwise \end{cases} \tag{4.7}$$

where a is the maximum acceleration, $\dot{q_{t_i}}$ is the current robot velocity and $n_{\dot{q_{t_i}}}$ is the unitary vector pointing in direction of robot velocity [22].
Finally, the sum of the two forces, above found, give the resulting force to compute and control the trajectory of the robot at every time t as depicted in Figure 4.4.

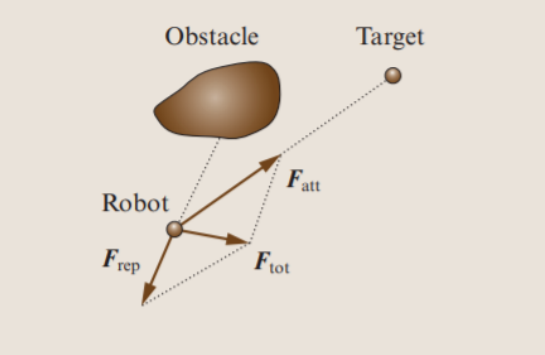

Figure 4.4: Potential Field Method [22]

4.2.2 Vector Field Histogram (VHF)

This algorithm divides the resolution of the problem into two steps where the first one calculates a range of motion directions and the second one chooses one of the computed directions.
First, the space is divided into sectors and a polar histogram H is constructed around the robot to represent the obstacles around it.
The function $h^k(q_{t_i})$ is used to map the obstacle distribution in sector k on the corresponding component of the histogram [22]:

$$h^k(q_{t_i}) = \int_{\Omega k} P(p)^n \left(1 - \frac{d(q_{ti})}{d_{max}}\right)^m dp \tag{4.8}$$

that is proportional to the probability function $P(p)$ that represents a point occupied by an obstacle, and to the distance from the obstacle. So, if the distance from the obstacle increase, the density value will be lower.
At the end, the histogram has *peaks* that represent directions where there is a high density of obstacles and *valleys* that are directions with low density.
Therefore, the robot should move in the set of candidate directions represented by adjacent sectors with a density lower than a given threshold and closest to the target direction called selected valley.
Subsequently, the right direction along which the target is must be chosen taking into account the selected area and the size of the valley. There are three possible cases: in the first one the target is in the selected valley and in this case $k_{sol} = k_{target}$; in the second one the target is not in the selected valley and the number of sectors of the valley is greater than m, and in this case the solution is $k_{sol} = k_i + \frac{m}{2}$ where k_i is the sector of the valley that is closer to k_{target}. The last case is similar to the second one, but the number of sectors of the valley is lower or equal to m. In this last case the solution is $k_{sol} = \frac{k_i + k_j}{2}$ where k_i and k_j are the extreme sectors of the selected area.
The final result is a sector k_{sol} with bisector Θ_{sol}, and the velocity v_{sol} is inversely proportional to the distance to the closest obstacle with the control denoted with $u_i = (v_{sol}, \Theta_{sol})$. Figure 4.5 shows an example of the robot motion direction (Θ_{sol}) and the obstacle occupancy distribution.

4.2.3 Dinamic Window Approach (DWA)

The DWA solves the problem of navigation in two steps, as a function of the robot's velocity space. With this algorithm, the robot can navigate at high speed avoiding collisions with obstacles.
The following subset of the control space $\mathcal{U}$ is computed considering a motion

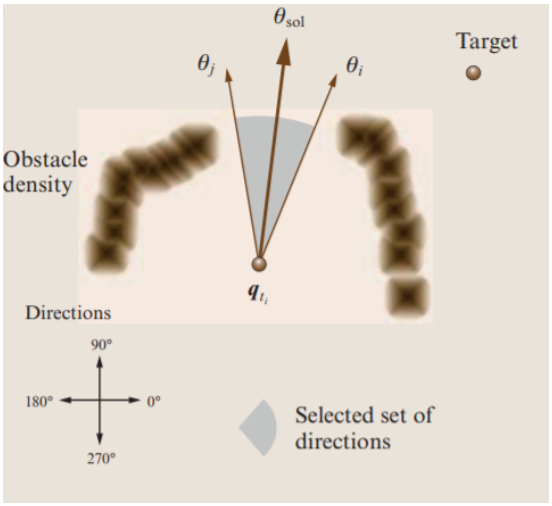

Figure 4.5: Robot and obstacle occupancy distribution in Vector Field Histogram [22]

control composed of translational and rotational velocity (v,w) [22]:

$$\mathcal{U} = \{(v, w)\ \epsilon\ \mathbb{R}^2\ \backslash v\ \epsilon\ [-v_{max}, v_{max}]\ \wedge\ w\ \epsilon\ [-w_{max}, w_{max}]\} \tag{4.9}$$

The candidate set of control is denoted with $\mathcal{U}_{\mathcal{R}}$. It contains the controls that are within the maximum velocities of the robot; it generates safe trajectories $\mathcal{U}_{\mathcal{A}}$ and it can be reached within a short period given the vehicle acceleration $\mathcal{U}_{\mathcal{D}}$. The set $\mathcal{U}_{\mathcal{A}}$ contains the admissible controls. These can be deleted before collision through the maximum deceleration (a_v, a_w):

$$\mathcal{U}_{\mathcal{A}} = \{(v, w)\ \epsilon\ \mathcal{U} | v \leq \sqrt{(2d_{obs}a_v)}\ \wedge\ w \leq \sqrt{(2\theta_{obs}a_w)}\} \tag{4.10}$$

in which d_{obs} is the distance to the obstacle and θ_{obs} is the orientation.
$\mathcal{U}_{\mathcal{D}}$ is the set that contains controls that can be reached in a short period:

$$\mathcal{U}_{\mathcal{D}} = \{(v, w)\ \epsilon\ ; \mathcal{U}\ \backslash v\ \epsilon\ [v_0 - a_v T, v_0 + a_v T]\ \wedge\ w\ \epsilon\ [w_0 - a_w T, w_0 + a_w T]\} \tag{4.11}$$

Hence, the resulting subset of control is represented by (Figure 4.6):

$$\mathcal{U}_{\mathcal{R}} = \mathcal{U} \cap \mathcal{U}_{\mathcal{A}} \cap \mathcal{U}_{\mathcal{D}} \tag{4.12}$$

The final step is the selection of the control $u_i\ \epsilon\ \mathcal{U}_{\mathcal{R}}$ maximizing an objective function:

$$G(u) = \alpha_1 \cdot Goal(u) + \alpha_2 \cdot Clearance(u) + \alpha_3 \cdot Velocity(u) \tag{4.13}$$

where there is a combination of Goal(u) that facilitates velocities towards the goal, Clearance(u) that facilitates velocities far from obstacles, and Velocity (u) that favors high speeds. The solution is given by the u_i that maximizes this function. The DWA uses information of robot dynamics, hence, it works well at high speed or with the vehicle with slow dynamic capabilities.

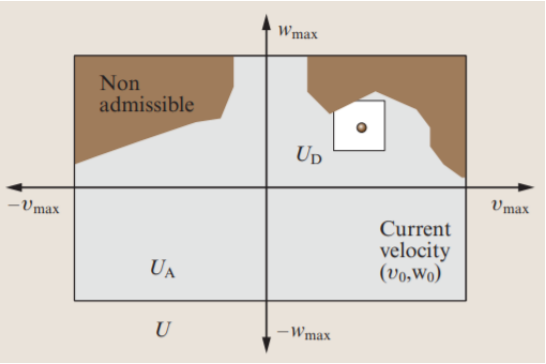

Figure 4.6: DWA: subset of control $\mathcal{U}_{\mathcal{R}}$: $\mathcal{U}$ used to contain controls within the maximum velocities, $\mathcal{U}_{\mathcal{A}}$ admissible controls, $\mathcal{U}_{\mathcal{D}}$ contains controls reachable in a short period of time [22]

Chapter 5

Autonomous Navigation

This chapter will describe how the autonomous mobile system has been developed at CIM 4.0, which is the ultimate goal of this thesis.

Firstly, it is important to remember what is autonomous navigation. The term autonomy refers to the ability to navigate in an unfamiliar, unstructured, and unpredictable environment by acquiring information regarding the affected environment through the use of sensors. Often, it is advisable to use more sensors to have a good perception of the environment and to make sure that the errors coming from them are reduced through a wider range of measurements.

This chapter will illustrate also which sensors are chosen, the reason for their choice and how they are positioned to implement this autonomous navigation system. Compared to the most used autonomous driving robots, a suitable typology of sensors has been chosen for this system to overcome the problems caused by the type of wheels and to have an omnidirectional vision and trend. In this way, the robot will be able to interact with the environment in which it moves, by extracting robust information about the objects present and will be able to plan its motion avoiding obstacles.

The problem of autonomous navigation answers some fundamental questions in mobile robotics, as later described. These can be summarized in the knowledge of the position of the robot in a certain moment, where it is going, and how it can reach the final predetermined position [27]. To do this, it is necessary to have a model of the environment that must be perceived and analyzed by sensors also called ***map-building***, find its position within this environment denoted as ***localization*** and finally plan and execute the movement adopting ***path planning*** techniques. Thus, to navigate without human intervention, the robot needs a map, location knowledge, sensors, and navigation algorithms.

To build the map of the environment, in this thesis an algorithm of ***Simultaneous Localization and Mapping (SLAM)*** has been used and adapted to the considered system. Through it, the robot builds the map of the environment by moving over unknown spaces and at the same time estimates its current location.

Once the knowledge of the map, with which the robot must interact, is acquired, autonomous navigation can occur. In particular, the robot will be able to start from an initial point, once its position and orientation in space are known, and reach the final goal using the route planning techniques adopted.

To do this, the ***ROS (Robot Operating System)*** is used and in particular the navigation stack as described below. Furthermore, ***RViz*** is used to send basic commands and see the behavior in real-time, for example during the construction of the map. First of all the system is implemented and simulated on ***Gazebo*** and then the same is experimentally implemented using the configurations described below.

5.1 Introduction to ROS

ROS (Robot Operating System) is an open-source robot meta-operating system. It is mainly used for developing Robot applications. The main difference with an operating system is given by the fact that it can be used for different combinations of hardware implementation and it runs on an existing operating system. It is composed of processes that use a virtualization layer between applications and distributed computing resources. Then, it can be considered as software that connects different software components and applications. ROS, such depicted in Figure 5.1 connects Sensor, App and control a Robot with a hardware abstraction and develops application based on existing operating system [28]. It is not a programming language, it is not only a library because it contains a lot of tools and a build system. Furthermore, it is not an integrated development environment [29].

5.1.1 Goal and philosophy of ROS

ROS is open-source and has a lot of features, as listed hereafter [30]:

- **Peer-to-peer:** systems that use ROS are made of processes that run on different hosts connected at runtime in a peer-to-peer topology. So, it has a sort of mechanism that allows processes to find each other at runtime. Individual processes communicate over defined API.

Figure 5.1: Meta operating system [28]

- **Multi-lingual:** it can support a lot of programming languages, so it can be defined as neutral because different languages can be combined and mixed as desired. It supports mainly Python, C++, Octave, and LISP.
- **Tools-based:** a design can be used where small tools perform different tasks.
- **Thin:** ROS can re-use code from different open-source projects and can update source code from external repositories, etc.
- **Free, Open-Source and Distributed processes:** it is distributed under BSD Licence and passes information using inter-process communication. It is composed of processes where each one is executed independently and exchanges data systematically.
- **Organized in packages:** if different processes have the same purpose they are managed as a package.

5.1.2 Main components

It is composed of a lot of components where the mainly used for developing applications, exchanging information, and implementing communication for the system, are given by:

- **Node:** it is a process that provides a specific task and belongs to a package and has easy reusability;

- **Master:** it is used to launch and initialize a ROS application. Without its use, it is impossible to enable communications. The command *roscore* can be used to launch a Master node ;
- **Messages:** they are data used by a node to exchange information. They are variables such as integer, floating-point or boolean and can be used as a nested structure that contains other messages like an array structure;
- **Topic:** it is used by a node to communicate and exchange asynchronous information. It is a channel, and messages are data inside it. Nodes can publish or subscribe to a topic. The *publisher* node registers its information and topic and sends the message to *subscriber* that are interested in that topic. These communications, that permit a transfer of data, is unidirectional, and are ever connected to send or receive data. For this purpose, they are useful for sensors data that have to publish messages periodically. Usually is used one publisher and n subscribers. An example is reported in Figure 5.2 [28].

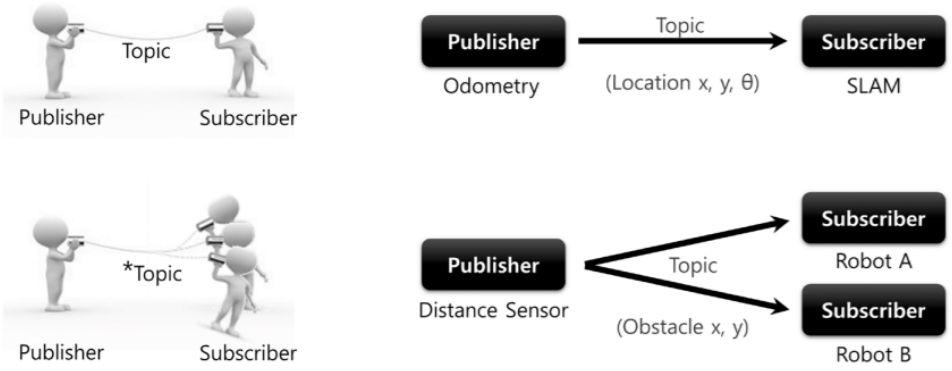

Figure 5.2: Example of communication between publisher and subscriber [28]

- **Package:** it represents a basic unit where the ROS application is developed.
- **Graph:** ROS can permit to visualize ROS graph to see the relationship between nodes, topics, and other main components.
- **Service:** it is used for bidirectional synchronous communication between nodes. Based on request and response, it is used for one-time communications. So, when the communications are finished, the connection is disconnected. It is divided into *Service Server* that receives a request and transmits the

response as an output, and the *Service Client* that requests service to the server and receives a response as an input. Request and response are in form of messages like in Figure 5.3 [28].

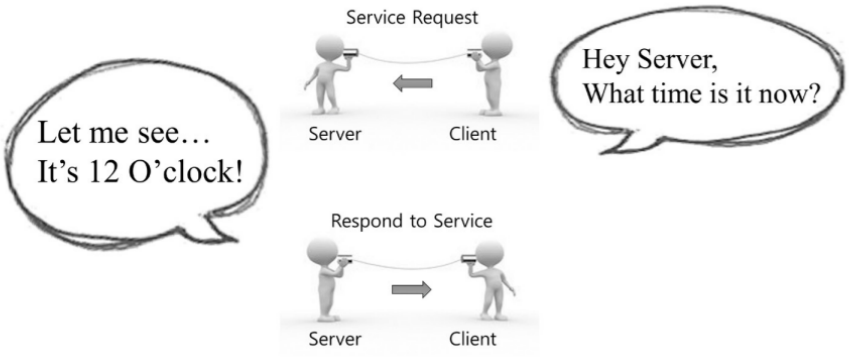

Figure 5.3: Example of server communication [28]

- **Catkin Workspace:** it represents the position where the developed code and executable nodes are located. It is the build system for Ros because it allows to build and organize packages, gives the possibility to reuse code, build nodes, create a workspace, and generate messages. Here, there is *package.xml* that gives information about the package like name, license, version, and *CMakeLists.txt* that contains information on how to build packages.
- **Launch file:** it is used for launching multiple nodes and it is written in XML.
- **Parameters:** it is useful to manipulate data and have a message communication.

Table 5.1 highlights the main differences between Server and Topic.

5.2 RViz

RViz is a 3D visualization tool for ROS that helps users to know what a robot is doing and how it moves and interact with the surrounding environment.

It provides a very useful GUI that allows the possibility to interact with the robot and to choose what parameters and information would be visualized as shown

	Topic	Service
Description	Continuous exchanging of data	Processing a request but blocks call
Features	Unidirectional	Bi-directional
Example	Sensor data and robot state	request or compute something
Communication	Asynchronous	Synchronous

Table 5.1: Main differences between Server and Topic

in Figure 5.4. The panel on the left allows selecting data that is possible to display from various topics; the central part of the screen allows to see different data in 3D. It is mainly used to see how the environment is perceived, the behavior of data from sensors like Kinect, Intel Realsense (Figure 5.5), Laser Distance Sensor data, and how a robot interact with obstacles.

It can be used both in Gazebo for simulation but also to do tests in real-life displaying ROS messages and topics giving the possibility to visually control the system. It allows the user to send a command to the robot, set its position, see how it plans its route to reach a final goal, and visualize how a map is built in real-time while the robot moves.

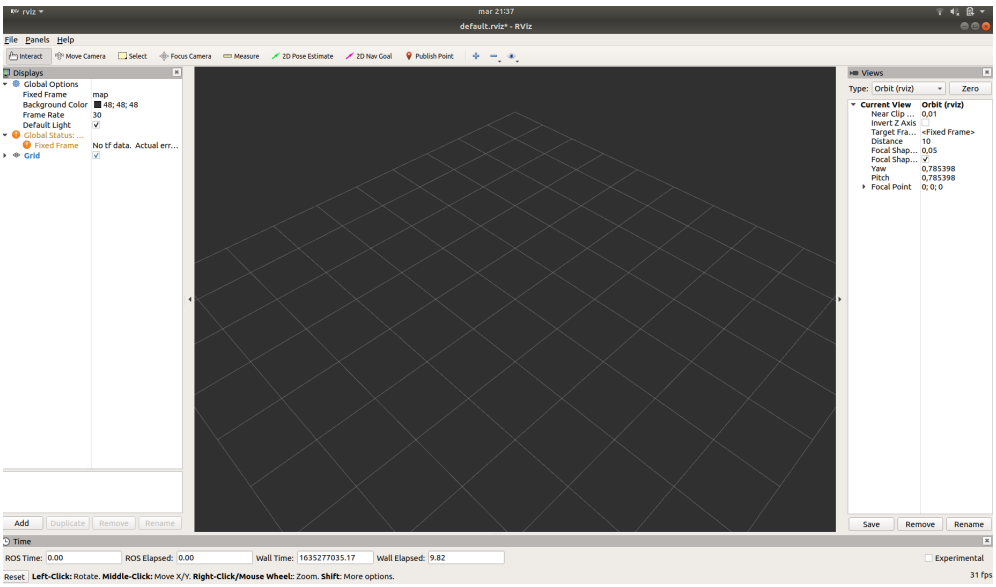

Figure 5.4: RViz Graphical User Interface

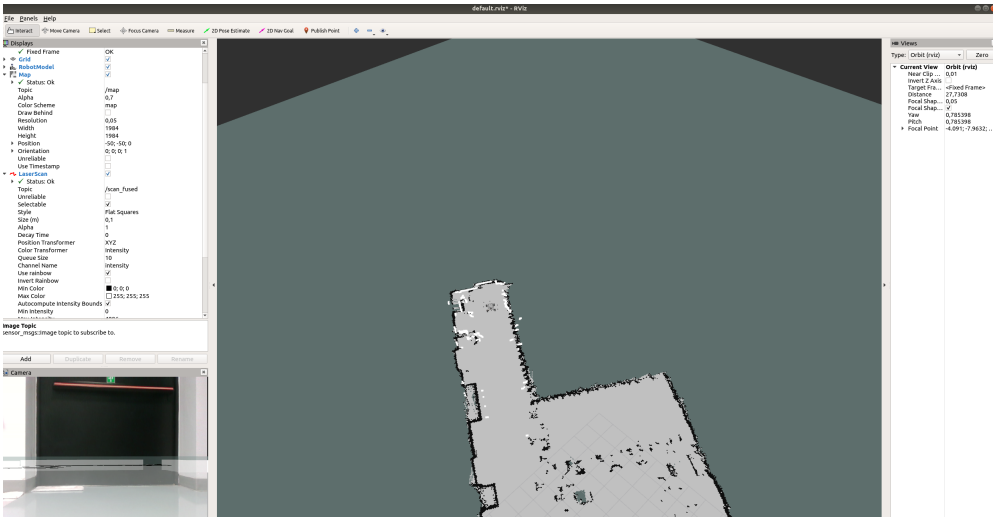

Figure 5.5: RViz visualization of image from camera. The white points are the results viewed by LiDAR

5.3 Gazebo

Gazebo is a useful ROS tool that provides a 3D robotic simulator environment. It offers realistic simulation using 3D models of robots, sensors, and the environment. It provides a Graphical User Interface (GUI) that is used to delete, add or modify 3D models, as depicted in Figure 5.6. In particular, through its use, it is possible to test algorithms and create both complex indoor and outdoor environments. The main features are [31]:

- Dynamic simulation: access multiple high performances physics engines including ODE, Bullet and Simbody;
- Advanced 3D graphics: using OGRE, this tools often used in games, provides a realistic rendering of the environment as high-quality lighting, shadows, and textures;
- Sensors and noise: it can provide data that come from sensors, like laser range finders, cameras, Kinect sensors including also noise and more;
- Plugins: develop custom plugins to simulate and test robots, sensors or to control the environment. Moreover, many robots and environments (Gazebo worlds) are provided or it is possible to build another using SDF.

ROS integrates with Gazebo using *Gazebo_ros* package that allows bidirectional communication between them.

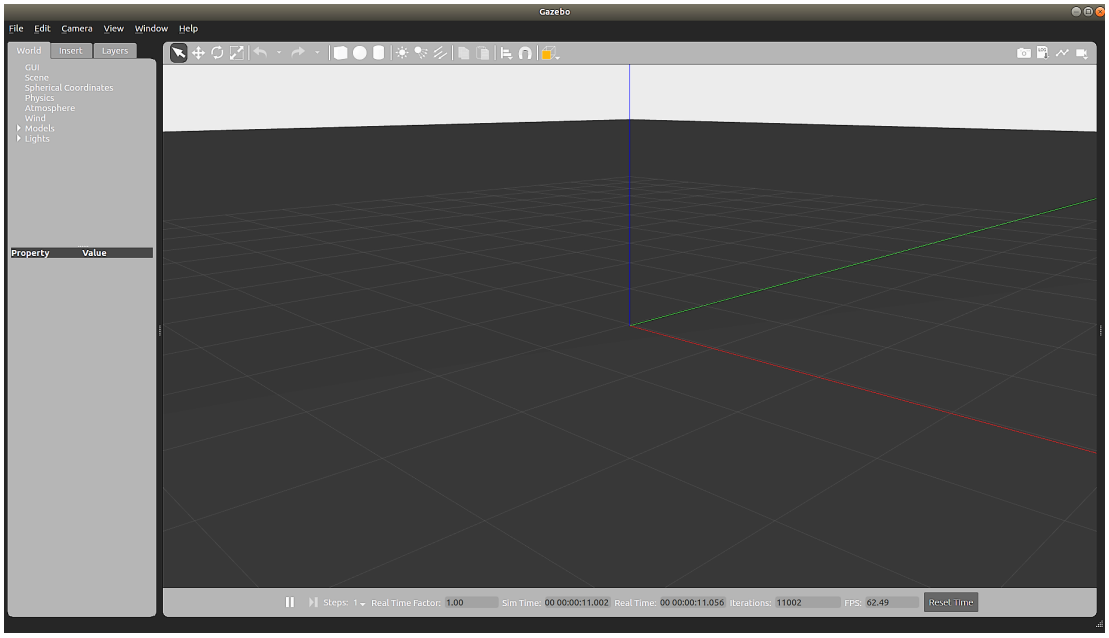

Figure 5.6: Gazebo Graphical User Interface

The robot model is simulated using the Unified Robotic Description Format (URDF), that is an XML file format used in ROS to describe all elements of a robot. This can only specify the proper kinematic and dynamic features of a robot like inertia, contact coefficients, joint dynamics giving also their visual representation and a collision model. The URDF contains simulation parameters in the gazebo tags. These contains gazebo plugins (sensors and actuators), Gazebo material properties and dynamic parameters. Hence, to overcome the problem of having and manage with only a limited set of parameters related to a single robot, a new format named Simulation Description Format (SDF) was created. It is a comprehensive description, in XML format, for everything from the world to the robot level, making it easy to add, delete or modify elements. Gazebo is able to converts a URDF to SDF automatically.

5.4 Introduction to SLAM algorithm

In robotic applications it is very important to know where the robot is moving. Simultaneous Localization And Mapping (SLAM) represents a process through which a robot can build a map of the environment and in the meanwhile, it can be used to determine its position through the sensorial information that is given by

onboard sensors. This is useful, because it permits the robot to move everywhere without the need of knowing a priori information about the surrounding environment [32].

This method extends the usual requirement of a mathematical model to build the environmental map. It needs also a representation of the robot state such as orientation and position in correlation with the knowledge of the position of extracted landmarks. These algorithms must consider a lot of parameters that concern not only the map and the representation of the environment but also the measurements that come from the sensor used.

The most important methods used for sensor application are raw range scan sensors and feature-based sensors. Furthermore, the most commonly used sensors are sonar and a laser-based sensor for landmark extraction from a scan and in this case, the method is LiDAR Slam. In particular, the second one is widely used for the high accuracy and speed able to generate a very precise measurement of the distance. Moreover, for the extraction from images, it is used Visual Slam, where the camera is used in many configurations, and the most used are the RGB-D sensors, because they provide performance similar to 3D laser sensor but at a lower cost, even if with some drawbacks related to computational constraints and visually-reflective material [33].

However, flexibility can be reached combining landmark detection with graph-based optimization [34]. Moreover, many researchers have shown how the quality of SLAM-made maps can be influenced by the dynamic environment [35].

First of all, it is important to define such algorithm as applied to a robot in motion, defining the following quantities at time k:

- x_k: robot state including position and orientation;
- u_k: control vector applied to a state x_k at time k;
- m_i: location of a landmark assumed to be time-invariant;
- z_{ik}: observation taken from the robot of the location of the i^{th} landmark.

In SLAM, a mobile robot builds a map of an environment and at the same time finds its location in that map. The trajectory of the platform and the location of all landmarks are estimated online without the need for previous knowledge of the location.

Environmental observations can be manipulated for SLAM implementation. The

first concerns the external sensors that can perceive and locate the landmarks through two consecutive observations at two successive instants of time in two consecutive positions. Instead, the second concerns recording, which is a method that creates spatial constraints by aligning the sensor data that is based on their readings. As depicted in Figure 5.7, where the true locations are never known, it can be seen that much of the error between estimated and true landmark location is common between them and because there is a unique single source. So errors in landmark location are correlated [32]. A critical factor is uncertainty associated

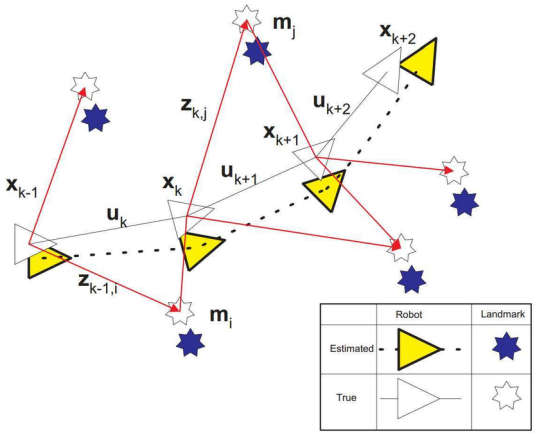

Figure 5.7: SLAM problem [32]

with predicting the robot's position. Each motion estimate introduces uncertainty by increasing the cumulative error leading to an unlimited drift in the poses estimates. This can be overcome through the loop closure or by generating spatial constraints between the estimated poses and the previous places visited. Therefore, based on non-sequential observations, loop closures create anchor points in the environment, which help to limit the incremental error in the estimation of the position.

Furthermore, loop closures can be established when after a certain time the robot returns to previously visited positions. The implementation of SLAM is usually organized in the front-end, which is responsible for the methods concerning the generation of spatial constraints and the back-end which refines the noise found in the previous results through probabilistic optimization algorithms also providing feedback for the front-end loop closure. Although this algorithm is used in a lot of applications, some problems prevent general use and they are summarized in:

1. **Accumulation of localization error:** all measures made with this algorithm tend to generate some error that is accumulated over time preventing obtaining

the actual value. So, maps obtained are distorted and correlated by successive problems. To avoid this, a solution can be that of remembering a previously visited place through, for example, as a landmark. Pose graph is needed in this case and the type of optimization is called bundle adjustment in visual SLAM [34].

2. **There is no localization and map is lost:** the discontinuous estimation of position is generated from image and point-cloud mapping, because they do not consider some characteristics related to the position of the robot. This kind of problem can be prevented by the use of other sensors from which data can be fused using an Extended Kalman Filter and particle filters. A technique to prevent the problem is to remember a previous landmark as a keyframe. To speed up the scan, it is used a feature extraction process [34].

3. **Elevated computational cost for processing and optimization:** the first arises when SLAM is operated in hardware. It is usually performed by embedded microprocessors that have a reduced processing power. Furthermore, to have an accurate localization, it is essential to execute image processing and point cloud matching at high frequency. Another problem is that optimization calculations such as loop closure are high computation processes. The challenge is how to execute such computationally expensive processing on embedded microcomputers. A solution can be to run in parallel different processes [34].

5.4.1 ROS for SLAM

In ROS there are some packages related to the SLAM algorithm:

- **hector_mapping**: useful in narrow space and emergency situations, it does not require odometry data as input [35]. It is a node for LiDAR based on low computational resources [36]. It is mainly used when an IMU (Inertial Measurement Unity) is present.

- **gmapping**: requires odometry data and the robot has to be equipped with a horizontally-mounted, fixed, laser range-finder. The slam gmapping node is used to transform each incoming scan into the odometry tf frame [37]. It utilizes Rao-Blackwellised particle filter to estimate the model [35]. However, particle filters require a lot of number of particles to obtain a good result and in this way, the computational complexity is increased [38].

- **crsm_slam**: the Critical Rays Scan Match selects only some subset of laser scan ray to reduce the noise and improve the quality of map building and does not use the odometry data [35].

- **rgbd slam**: used to perform visual odometry using the only 3D map of the environment that is built from point cloud [35]. It gives also the current position of the camera [39]. The drawback is that it is computationally heavy.

- **tiny_slam**: it is a lightweight solution and use Monte Carlo Localization for localization [35]. In addition, it uses an alternative grid map that keeps track of the stored value and it is possible to compute the probability that a cell is occupied exploiting how beams split the cell [40].

5.4.2 A review of Rao-Blackwellized Particle Filter

The algorithm chosen to acquire and build the map is **gmapping**. It has been introduced in 2007 and it is very widely used for this kind of application. It provides a laser-based SLAM capable of localizing the robot inside the map while it is creating the surrounding environment. It employes a particle filter named Rao-Blackwellized Particle Filter (RBPF), introduced by Murphy, Doucet, and colleagues [41], [42], that represents a technique for model-based estimation in which, each particle carries an individual map of the environment. Hence, it using adaptive resampling techniques reduce the computational complexity improving the position accuracy. The approach used by the Rao-Blackwellized particle filter for SLAM is that of estimating the joint posterior probability $p(x_{1:t}, m|z_{1:t,u_{1:t-1}})$ about the map m and the trajectory $x_{1:t} = x_1, ..., x_t$ of the robot given observations $z_{1:t}$ and the odometry measurements $u_{1:t-1}$ from the robots. It makes use of the following factorization:

$$p(x_{1:t}, m|z_{1:t,u_{1:t-1}}) = p(m|x_{1:t}, z_{1:t}) \cdot p(x_{1:t}|z_{1:t}, u_{1:t-1}) \tag{5.1}$$

with these equations is possible to estimate the trajectory of the robot and the map given that trajectory. Usually $p(m|x_{1:t}, z_{1:t})$ is known when mapping with known poses. A particle filter is applied to compute $p(x_{1:t}|z_{1:t}, u_{1:t-1})$, where each particle represents a potential trajectory of the robot. Usually, the sampling importance resampling (SIR) filter is used. A Rao-Blackwellized SIR filter use sensor observations and odometry readings when they are available. It updates the set of samples that represents the posterior about the map and trajectory of the vehicle in four steps [43]:

- **Sampling:** the generation of x_t^i is obtained from x_{t-1}^i

- **Importance Weighting:** a weight w_t^i is assigned to each particle according to:

$$w_t^i = \frac{p(x_{1:t}^{(i)}|z_{1:t}, u_{1:t-1})}{\pi(x_{1:t}^{(i)}|z_{1:t}, u_{1:t-1})} \tag{5.2}$$

- **Resampling:** particles are sampled with replacement proportional to their importance weight. In this way, there will be fewer amount of particles to approximate a continuous distribution. After this process, all particles have the same weight.
- **Map Estimation:** for each particle, the corresponding map estimate $p(m^{(i)}|x^{(i)}_{1:t}, z_{1:t})$ is computed based on trajectory $x^{(i)}_{1:t}$ of that sample and the history of observation $z_{1:t}$

The problem becomes inefficient when the length of the trajectory and observations grow over time. For this purpose, a new formulation is given [44]:

$$\pi(x_{1:t}|z_{1:t}, u_{1:t-1}) = \pi(x_t|x_{1:t-1}, z_{1:t}, u_{1:t-1}) \cdot \pi(x_{1:t-1}|z_{1:t-1}, u_{1:t-2}) \tag{5.3}$$

where the weights are found as:

$$w^i_t = \frac{p(x^i_{1:t}|z_{1:t}, u_{1:t-1})}{\pi(x^i_{1:t}|z_{1:t}, u_{1:t-1})} = \frac{\eta \cdot p(z_t|x^i_{1:t}, z_{1:t-1}) \cdot p(x^i_t|x^i_{t-1}, u_{t-1})}{\pi(x^i_t|x^i_{1:t-1}, z_{1_{1:t}, u_{1:t-1}})} \cdot \frac{(p^i_{1:t-1}|z_{1:t-1}, u_{1:t-2})}{\pi(x^i_{1:t-1}|z_{1:t}, u_{1:t-2})}$$
$$\propto \frac{p(z_t|m^i_{t-1}, x^i_t) \cdot p(x^i_t|x^i_{t-1}, u_{t-1})}{\pi(x_t|x^i_{1:t-1}, z_{1:t}, u_{1:t-1})} \cdot \omega^i_{t-1} \tag{5.4}$$

in which $\eta = \frac{1}{p(z_t|z_{1:t-1}, u_{1:t-1})}$ is a normalization factor. The authors describe techniques to compute accurate proposal distributions that adaptively perform re-sampling to improve the mapping. The optimal choice of proposal distribution is:

$$p(x_t|m^i_{t-1}, x^i_{t-1}, z_t, u_t) = \frac{p(z_t|m^i_{t-1}, x_t) \cdot p(x_t|x^i_{t-1}, u_t)}{\int p(z_t|m^{(i)}_{t-1}, x')p(x'|x^{(i)_{t-1}, u_t})dx'} \tag{5.5}$$

The odometry model $p(x_t|x_{t-1}, u_t)$ is often chosen as proposal distribution especially when used with a mobile robot equipped with laser range finder. This choice can be sub-optimal because the accuracy of the laser range finder gives a peaked likelihood function as in Figure 5.8. In the system, the $p(x_t|x^{(i)}_{t-1}, u_t)$ is approximated by a constant k within the interval $L^{(i)}$ defined as:

$$L^{(i)} = \left\{x|p(z_t|m^i_{t-1}, x) > \epsilon\right\} \tag{5.6}$$

Hence, the equation can be re-written as:

$$p(x_t|m^i_{t-1}, x^i_{t-1}, z_t, u_t) \simeq \frac{p(z_t|m^i_{t-1}, x_t)}{\int_{x' \in L^{(i)}} p(z_t|m^i_{t-1}, x')dx'} \tag{5.7}$$

Then, the distribution is locally approximated around the maximum of the likelihood function by a Gaussian:

$$p(x_t|m^i_{t-1}, x^i_{t-1}, z_t, u_t) \simeq \mathcal{N}(u^i_t, \Sigma^i_t) \tag{5.8}$$

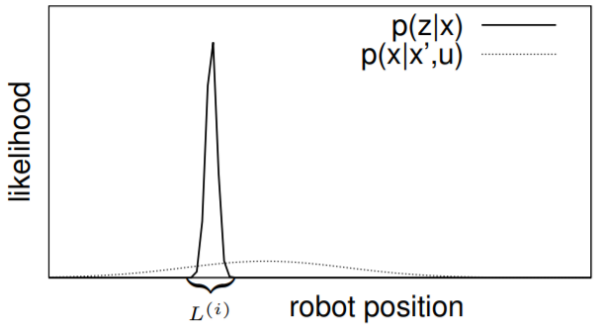

Figure 5.8: Components of the motion model [45]

For each particle i, the parameters u_t^i and Σ_t^i are determined as:

$$u_t^i = \frac{1}{\eta}\sum_{j=1}^{K} x_j p(z_t|m_{t-1}^{(i)}, x_j)\Sigma_t^i = \frac{1}{\eta}\sum_{j=1}^{K} x_j p(z_t|m_{t-1}^{(i)}, x_j)(x_j - u_t^i)(x_j - u_t^i)^T \quad (5.9)$$

in which $\eta = \Sigma_{j=1}^{k} p(z_t|m_{t-1}^i, x_j)$ is a normalizer. The importance weight $w^(i)_t$ can be approximated as:

$$w_t^{(i)} = w_{t-1}^{(i)} k\eta \quad (5.10)$$

The proposal distribution allows a robot equipped with a laser range finder to obtain good results in terms of particle accuracy as in Figure 5.9.

Finally, in [45] there is the introduction of N_{eff} that denotes the effective number of particles to perform a resampling step to estimate how well the current particle set represents the true posterior:

$$N_{eff} = \frac{1}{\Sigma_{i=1}^{N}(w^{(i)})^2} \quad (5.11)$$

The approach proposed is to resample each time N_{eff} goes below a given threshold of $N/2$, where N is the number of particles.

5.4.3 SLAM in Simulation environment

Gmapping requires a single source of scan and gives as output the 2D occupancy grid map that is the most widely output used in LiDAR SLAM techniques. So, it gives a 2D map that represents obstacles on the plane. The expression:

$$M_{grid} = m_g(x, y) \quad (5.12)$$

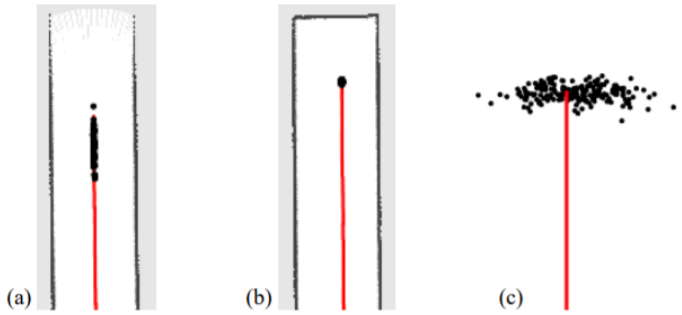

Figure 5.9: Particle distributions during mapping. In a featureless open space, the proposal distribution is the raw odometry motion model (a). In a dead-end corridor, particles uncertainty is small in all of the directions (b). In an open corridor, the particles distribute along the corridor (c) [45].

usually indicates the probability of a map cell to be occupied; it is 1 if the grid cell (x,y) is occupied or 0 if it is not occupied.

Moreover, this algorithm includes loop-closure detection, so it can recognize a location that it has already been visited. Having the loop closure, the accuracy of the map and the position in the map increase. Performing the algorithm with the viewer tool RViz, it is possible to see the building of the map in real-time in the Gazebo simulator. To move the robot in the simulation environment, it is used *teleop twist keyboard* that allows going everywhere in the map using the command from the keyboard.

The node requires to know the frame attached to the mobile base that is represented by the *base link*, the frame attached to the map that is *map* and the frame attached to the odometry system. Furthermore, the scan topic used to create the map is firstly set to only one laser, like the frontal or the back one. The problem is given by the fact that to create the map it should rotate accumulating errors. Instead, using the topic that merges the scans, it acquires a map in a single shot without rotating. Figure 5.10 shows the map of the simulation environment built in the Gazebo simulator. Instead, in Figure 5.11 there is the map acquired with the gmapping SLAM algorithm.

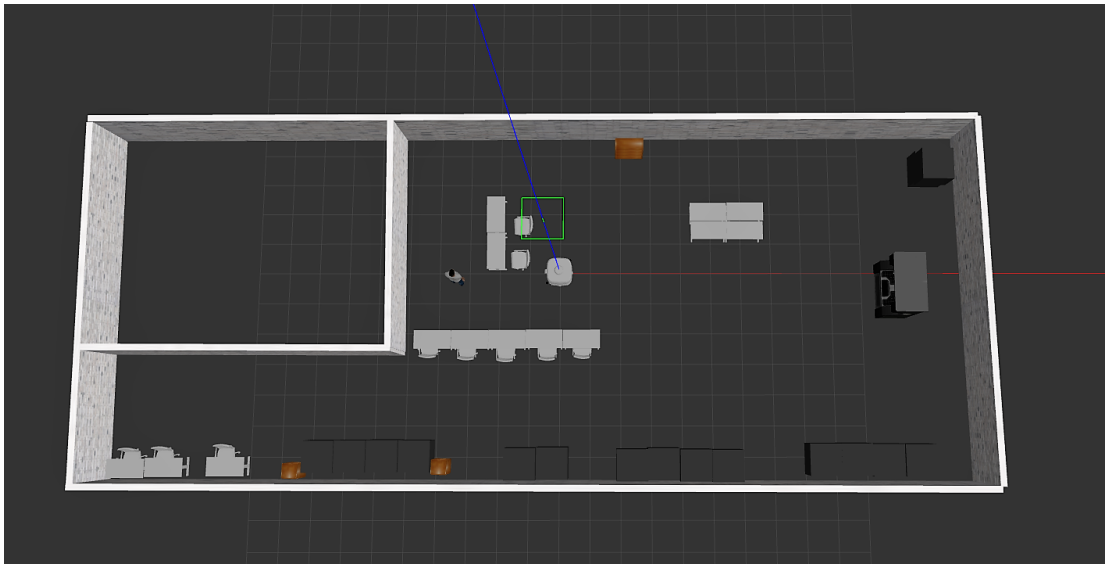

Figure 5.10: Map of simulation environment

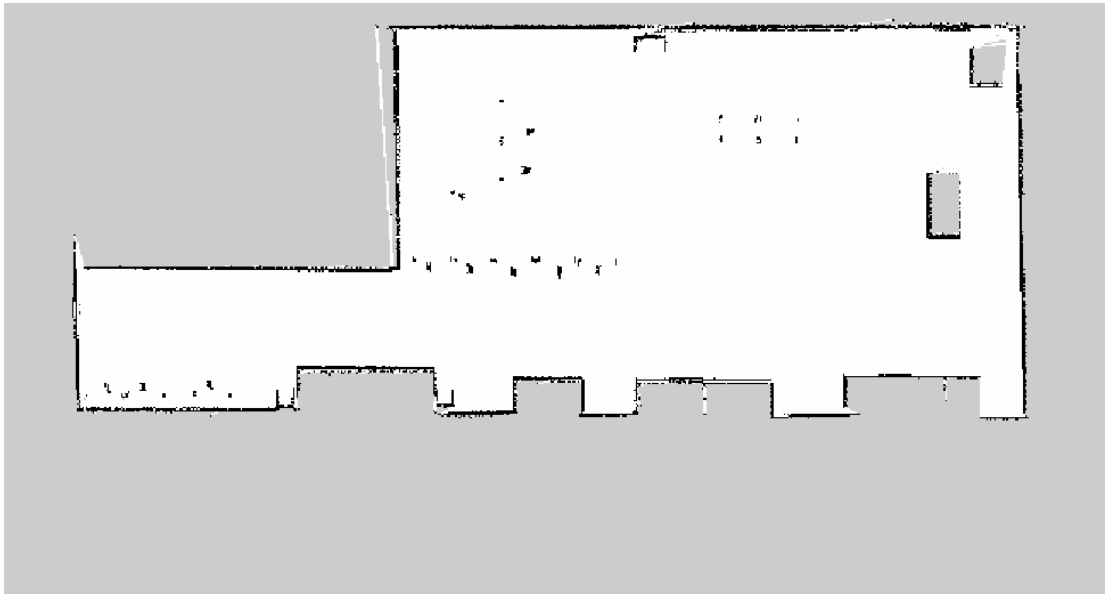

Figure 5.11: 2D map acquired with SLAM algorithm

5.4.4 Simulation environment and results

First of all, in the system Gazebo plugins of each sensor like for the camera, LiDARs, and robot model, have been used. Plugins are pieces of code that are compiled as shared library and inserted into the simulation. They have direct access to all aspects of Gazebo through the standard C++ classes. Using these plugins it is possible to manage a lot of aspects useful to simulate own model, like topics published from sensors, and control any aspect of Gazebo.

After that, to create the system, and so, to establish the connection between each frames of robot, sensor, and environment, has been created a URDF containing a set of link elements and joint elements, that allows to describe connections of the system.

In the simulation environment, the final model of the robot with all the sensors, has been created and hence with the final configuration which involves the use of four depth-cameras, one for each side, and two LiDARs placed in the diagonal of the maintenance system, as depicted in Figure 5.12. The robot movement is

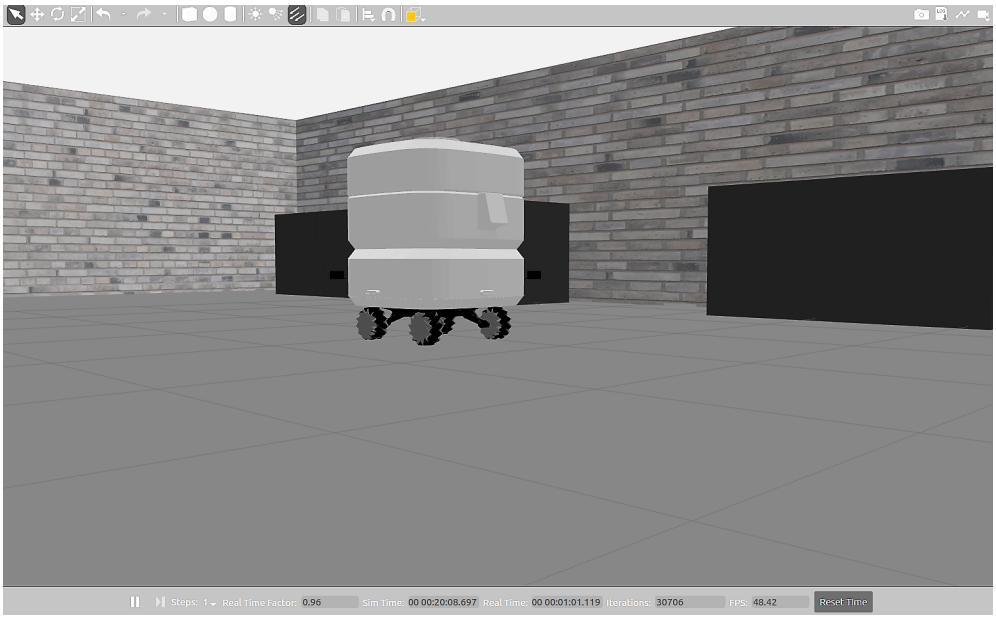

Figure 5.12: Gazebo model of the robot

simulated to go from one point to another using the navigation stack, and therefore also the ability to avoid the collision of obstacles in its path. The results obtained show the ability of this robot to move easily even in narrow environments thanks

to the translational movement allowed by the type of wheels and to perfectly follow the programmed path, compliant with both global and local path planning.

The properties that have been verified are:

- the navigation and localization: objects are arranged in a way that the robot has to traverse some difficult paths to reach the final goal. In this way, the navigation of the rover and the localization are tested. They have some errors especially in the real environment because the wheels are subject to slippage.
- the obstacle avoidance: this aspect has been tested both in simulation and in real environment by creating narrow paths or putting people in front of the moving rover.

In particular, two simulation environments have been created. The first, simpler, is made up of some basic components such as cylindrical and square objects, while the second environment has been created to simulate the CIM 4.0 environment. To simulate this, narrow paths are created with more complex objects such as work tables with a certain complex shape and particular chairs (Figure 5.13) with a shape that causes the construction of multiple branched obstacles, and people that could cause the robot to collide.

The first simple simulation environment is shown in Figure 5.14, whose map acquired through SLAM algorithm is shown in Figure 5.15. The robot plans the path to reach an endpoint previously set on the map by the user. In the case of narrow spaces, it can move without colliding with the obstacles that meet in its path. As it is possible to see from Figure 5.16, multiple obstacles represented by people have been placed randomly on the map. Although they were not previously present during the acquisition of the map, the robot can avoid them moving in tight spaces simply by using a rotational or parallel motion. Figure 5.17 shows the second simulation environment, which replicates the structure of the real CIM 4.0 environment. In particular, in this environment, narrow spaces are created with particular objects to simulate the real environment and compare the results. The figure shows Gazebo environment on the right, and the path planned by the robot on the left. This exact path is replicated and also tested in reality.

Furthermore, the robot in the simulation environment can move following the planned path without deviating much as regards the localization, and so, the position and orientation is accurate.

However, in real life, the behavior differs from that obtained in the simulation environment, due to the influence of various factors subsequently described in the hardware implementation and therefore in the experimental tests.

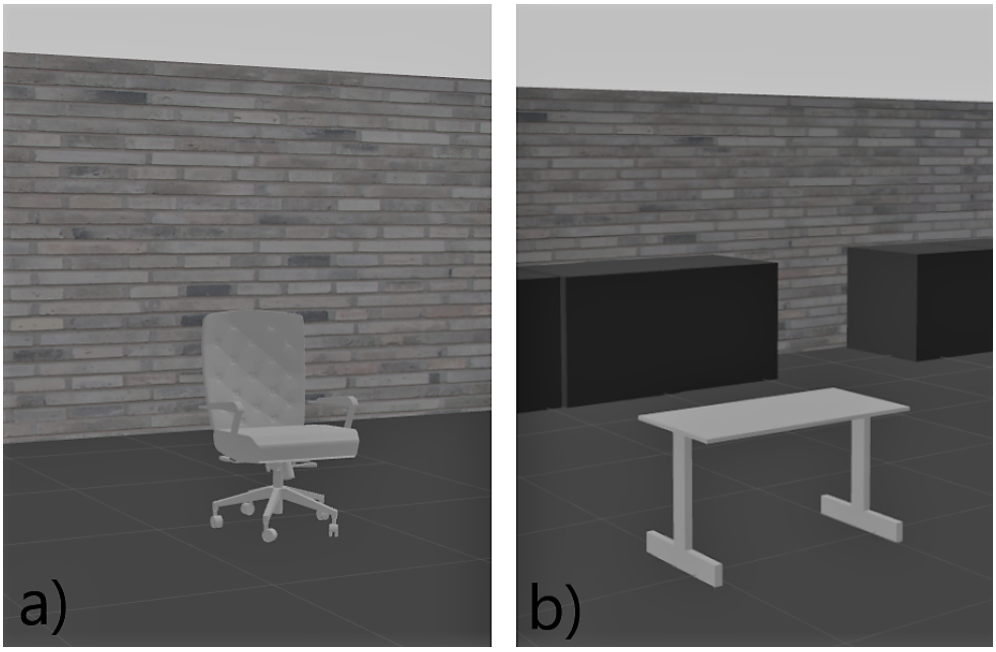

Figure 5.13: Chair (a) and Table (b) shape built in Gazebo to see the ability to the robot to see obstacles at each height.

5.5 Laser scan filtered

The laser scanners used have an omnidirectional range of vision of 360°. For this type of system, two laser scanners are used which are placed in the diagonal of the robot. A LiDAR is placed at the end of the right front side and another at the end of the left rear side.

The problem is given by the upper surface of the robot or the maintenance system which could create some wrong results. The lasers, rotating 360 degrees, also see the respective rear part which can create non-existent obstacles and therefore wrong results, as in Figure 5.18.

To overcome this problem, after positioning the LiDARs in the system and then after creating and arranging the respective reference frames, they are filtered as described in Figure 5.19. In particular, the viewing range of each LiDAR is reduced to 270° for both the rear and front, and subsequently, the results obtained are linked together as a single scan source. This turns out to be useful because for example in the construction of the map, many algorithms like the used *gmapping* require a single scan source.

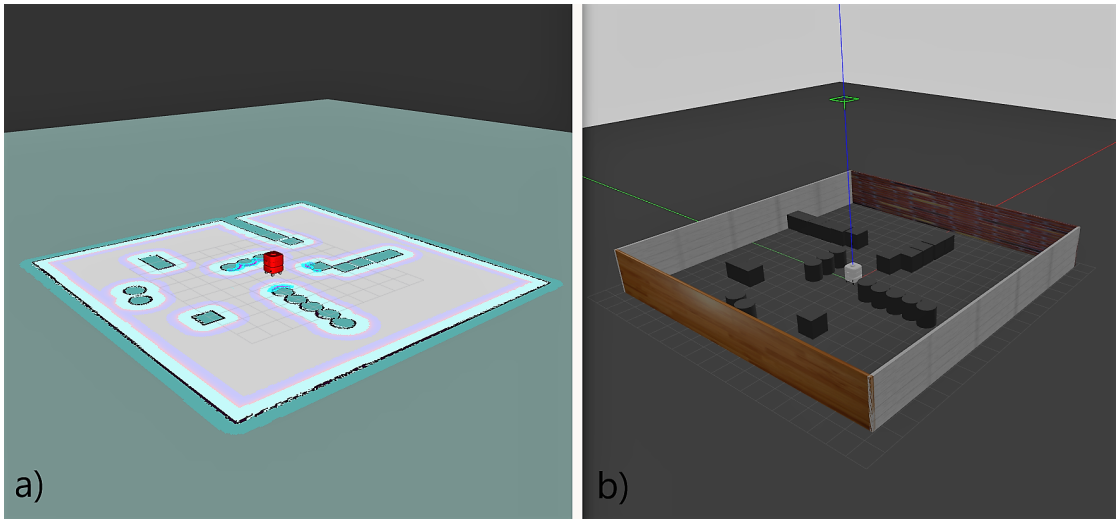

Figure 5.14: Simulation environment. On the right Gazebo simulation environment with simple basic objects (a). On the left RViz 2D visualization environment (b).

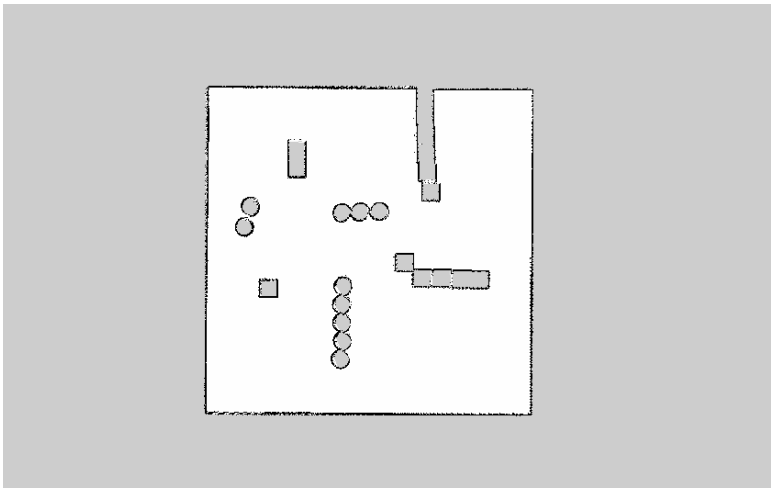

Figure 5.15: Map of the simulation environment acquired through SLAM

The laser scan message that contains raw data that are acquired from LiDARs is depicted in Figure 5.20. The laser scan data is published in a ROS topic as *sensor_msgs/LaserScan*. The most important parameters that are also used to

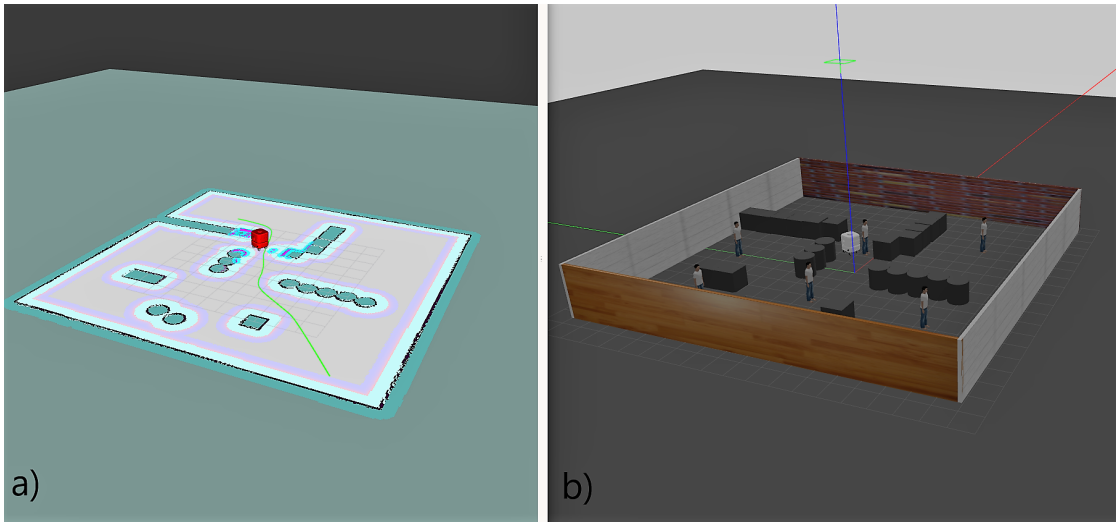

Figure 5.16: Simulation environment testing navigation and obstacle avoidance with many people randomly placed. The three-dimensional objects are displayed on the left through the Gazebo environment (a). Instead on the left there is the visualization on RViz where it is possible to see the global path represented by the green line (b).

manage with the source of scans are represented by *angle_min* and *angle_max* that indicate the angle range measured by LaserScan and *ranges* that is an array which gives the distance measured for each angle.

Hence, once set the angle parameters to -180 +180 degrees, the ranges give an array of 720 points in the 360 degrees. So, in this way, the zero point of the array corresponds to zero degrees, the point at 180 corresponds to 90 degrees, the point at 360 corresponds to 180 degrees, and so on.

To filter data and obtain only that one the goes from 0 to 270 degrees for each LiDAR, after the laser scan topics are published, the points of the array range that goes from 270 to 360 degrees are set to zero. After that, with this new array another topic is built corresponding to the LiDARs filtered from back and front side.

The next step is to take the new topics that match the filtered data from both LiDARs and merge them into a single topic. To do this, a package called $fuse\_lidars$ has been created. This, use the topics of the two filtered laser scanners respectively called $laserscan\_filtered\_front$ and $laserscan\_filtered\_back$ and the frame where the topic will be published. Subsequently, the laser scans are concatenated

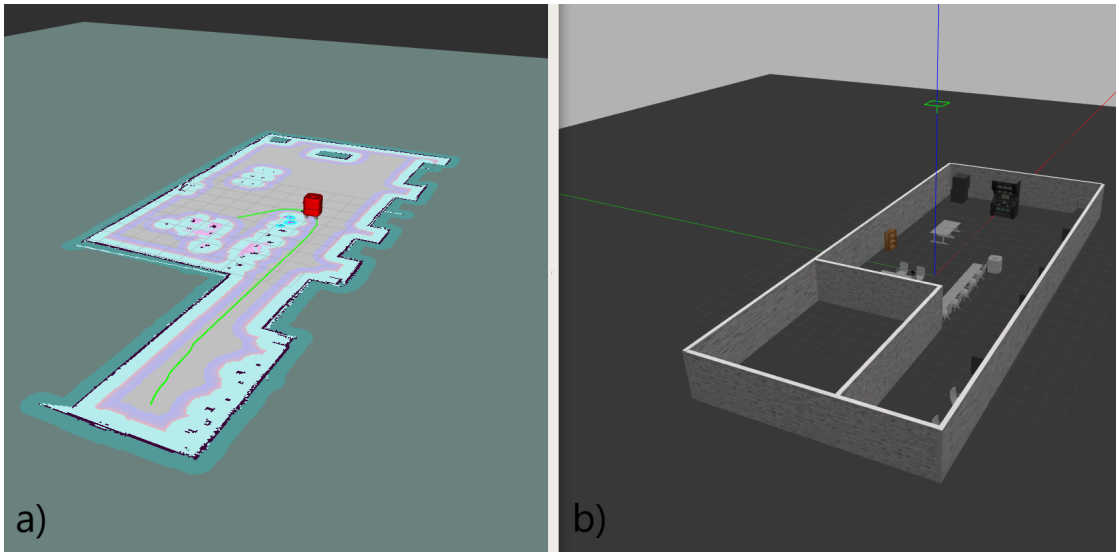

Figure 5.17: Movement of the robot in complex environment. On the right there is the simulation in Gazebo with complex obstacles (a). On the left the robot is visualized with RViz (b).

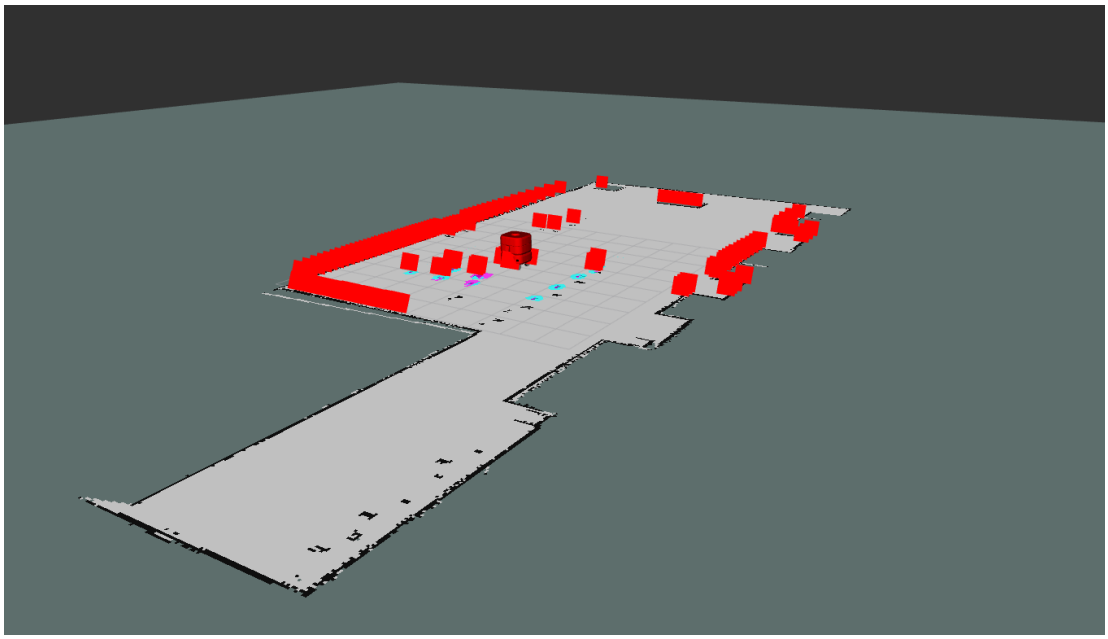

Figure 5.18: Visualization of wrong result of frontal LiDAR not filtered

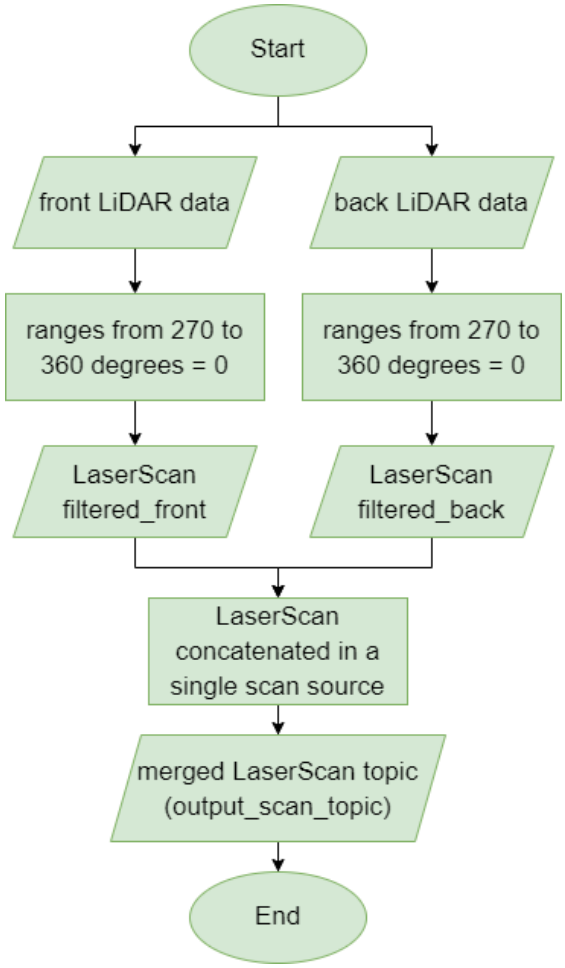

Figure 5.19: Flowchart of two LiDARs filtered and concatenated

with the PointClouds and when they are received in a synchronized manner they are concatenated using the *pcl concatenatePointCloud* function. Finally, the Point Cloud is converted into a laser scan taking into account the fact that the Point Cloud contains three-dimensional data and therefore these points are projected in the 2D plane obtaining the containment of the two lasers published in *output_scan_topic*.

The result visible in Figure 5.21 shows both the result of the filtered front LiDAR highlighted by the red color, that of the filtered rear LiDAR indicated by the white color, and finally the total scan representing the two LiDARs together represented by the multi-color.

```
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data

Header header            # timestamp in the header is the acquisition time of
                         # the first ray in the scan.
                         #
                         # in frame frame_id, angles are measured around
                         # the positive Z axis (counterclockwise, if Z is up)
                         # with zero angle being forward along the x axis

float32 angle_min        # start angle of the scan [rad]
float32 angle_max        # end angle of the scan [rad]
float32 angle_increment  # angular distance between measurements [rad]

float32 time_increment   # time between measurements [seconds] - if your scanner
                         # is moving, this will be used in interpolating position
                         # of 3d points
float32 scan_time        # time between scans [seconds]

float32 range_min        # minimum range value [m]
float32 range_max        # maximum range value [m]

float32[] ranges         # range data [m] (Note: values < range_min or > range_max should be discarded)
float32[] intensities    # intensity data [device-specific units].  If your
                         # device does not provide intensities, please leave
                         # the array empty.
```

Figure 5.20: LaserScan message [46]

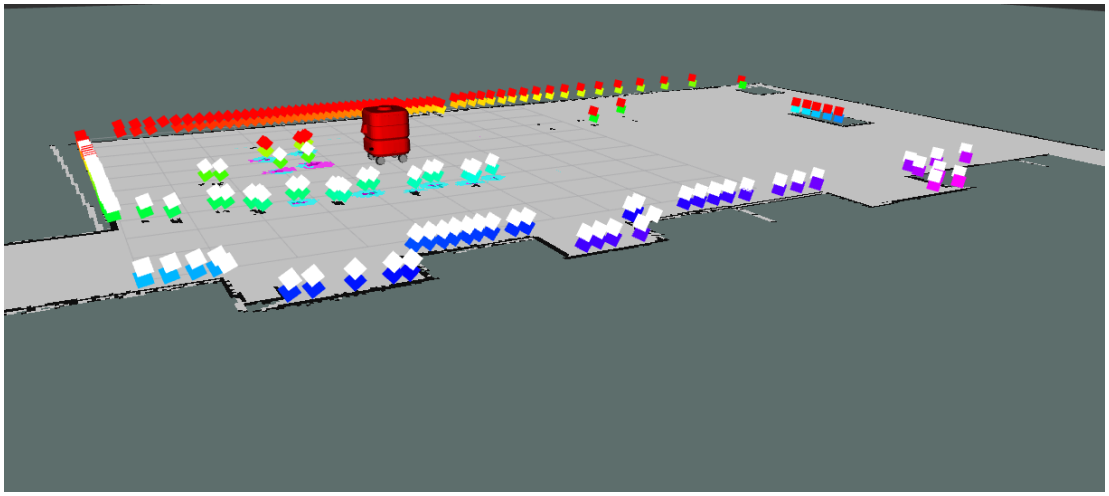

Figure 5.21: LiDARs filtered. The red color is the frontal LiDAR filtered, the white one is that ones on the back, and the multicolor one is the total scan

5.6 Navigation stack setup

The Navigation stack (Figure 5.22) [47] is used for autonomous navigation. It is a collection of ROS packages that through the information acquired from the odometry, from the sensor data and once received a final position to be reached, it sends safety velocity commands to the robot. It works well with both differential

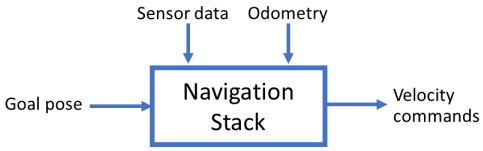

Figure 5.22: Navigation Stack

and holonomic robots and it requires the transform tree of a robot. A robot can be defined as a system composed of many components, each of which is identified by a position and an orientation in the space, that can be easily represented by a coordinate frame that is attached to the treated component. It is necessary to identify a common reference system in which the transformations between the frames and the relationships between them will be present. The main tf tree has:

- */odom* it is related to the odometry reference frame and a lot of time coincides with the point in which the robot is initialized.
- */base_link* it is related to the base of the robot.
- */base_scan* it is related to the laser scanner reference frame.

The *tf* ROS package [48] is used to represent the relationships between the frames attached to each robot, exploiting the information that is in the URDF files. It is widely used to represent the transformations between frames of the components and sensors of a moving robot.
The Navigation stack (Figure 5.23) is widely used for 2D autonomous navigation in indoor environments. The main component is represented by the ***move_base*** package that computes the velocity command to reach the final goal; it is composed

by *global planner*, *local planner*, *global costmap*, *local costmap* and *recovery behaviors* nodes, ***amcl*** that performs the localization of the robot, ***map_server*** that provides the reference map.

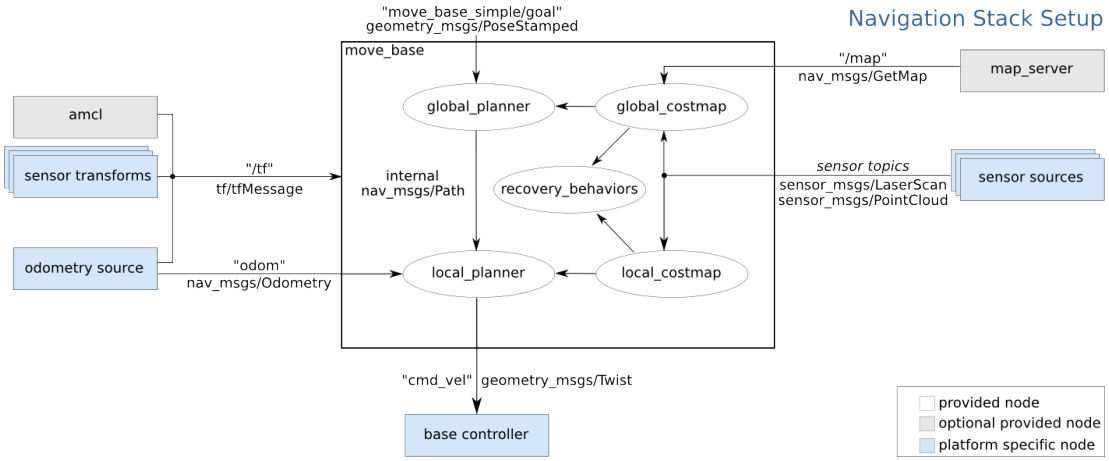

Figure 5.23: Navigation Stack setup [47]

5.6.1 Odometry source

The odometry gives the knowledge about the robot motion. It is mainly obtained through wheel encoders but in this case, it is not sufficient because of slippage and vibration given by Mecanum wheels used. Hence, to overcome the problem given from uncertainties and errors, in the system an Unscented Kalman filter is used to fuse the data from wheel encoders with that one of the Inertial Measurement Unit.

But, before to fuse together the data from both sensors, a Madgwick filter is applied. For this purpose, it is used the *imu_filter_madgwick* package that filters and fuses data taken from IMU devices and it based on code developed by Sebastian Madgwick [49]. It is used to fuse angular velocities, accelerations and magnetic readings from a generic IMU into an orientation quaternion. Then, the result is published on the *imu/data* topic [50].

Once obtained the IMU data fused together, these are fused with the wheel encoders of the scout mobile robot through the *robot_localization* package. It is a collection of two state estimation nodes called respectively *ekf_localization_node* and *ukf_localization_node*. Each of them is an implementation of a nonlinear state estimator for robots moving in 3D space. They have common characteristics and in particular, they are able to merge an arbitrary number of sensors

without having any constraints on the number of input sources, they support multiple ROS message types, they allow to exclude data that you do not want to include in the state estimation and the estimation takes place continuously as soon as the robot receives the measurement [51]. The state vector is 15-dimensional: $[x\ y\ z\ \alpha\ \beta\ \gamma\ \dot{x}\ \dot{y}\ \dot{z}\ \dot{\alpha}\ \dot{\beta}\ \dot{\gamma}\ \ddot{x}\ \ddot{y}\ \ddot{z}\ ]$. Usually, it can be used in two different ways. The first one is a continuous fusion of sensor data (like wheel encoder odometry and IMU) to give locally accurate state estimates and the other one is to fuse continuous data with global pose estimates to provide an accurate and global state estimate. As shown in Figure 5.24, for this kind of application, the data obtained from the Madgwick filter are fused with the ones of the wheel encoders through the *ukf_localization_node*.

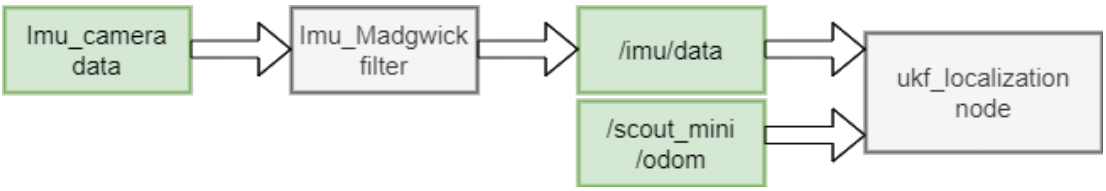

Figure 5.24: Odometry obtained from fusion of IMU and encoders

5.6.2 Map server

This node takes a map saved and offers it as a ROS service. The map file is an image *.pgm* that encodes the occupancy data with an associated *.yaml* file that describes the map meta-data as shown in Figure 5.25. It is possible to distinguish

```
image: nav.pgm
resolution: 0.050000
origin: [-50.000000, -50.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Figure 5.25: Map parameters

image that is referred to the path to the image file containing the occupancy data, *resolution* that is in meters/pixel, *origin* is the 2-D pose of their lower-left pixel in the map (x, y, yaw), *occupied_thresh* denotes that pixels with occupancy probability greater than this parameter are occupied, *free_thresh* shows as pixels

with occupancy probability less than this are free and *negate* if the convention with colors and free or occupied semantics should be reversed [52]. Instead, the image format describes the occupancy state of each cell. The 2D map is a grid where each element that is present, has an occupancy value. In Figure 5.11 it is possible to distinguish a white color that stands for free (0) cell, black for occupied (100), and unknown (-1) of grey color.

5.6.3 Costmap

The term costmap is used to indicate a map of the environment that divides the space into cells and assigns a cost to each cell. It subscribes to sensors topics over ROS updating itself. Each cell can be free, occupied, or unknown and sensors are used to mark, clear, or do both operations simply changing the cost of a cell. Other parameters that can be assigned to an occupied cell are the "lethal cost" and inflation. This last one is defined as the process of propagating the cost values from occupied cells as [53]:

$$e^{-k(d_{obst}-r_{infl})} \tag{5.13}$$

where k is a cost scaling factor, d_{obst} is the minimum distance of the robot from the obstacle, and r_{infl} is the inflation radius. This last parameter, set by users, denotes the radius to which the cost scaling function is applied. There are five main important parameters for costmaps related to inflation (Figure 5.26) [54]:

- *Lethal:* denotes an obstacle in a cell. If the center of the robot is in that cell, it will be in a collision.
- *Inscribed:* means that a cell is less than the inscribed radius of the robot away from the actual obstacle. If the robot center is in a cell at or above the inscribed cost, the robot will be in a collision.
- *Possibly circumscribed:* It has a cost similar to "inscribed" but the robot's circumscribed radius is used as cutoff distance. So, if the robot center is in a cell at or above that value, it will collide depending on the orientation. A feature of this parameter is to add a cost to an object or to an area in the map that depends on the user's preferences.
- *Free space:* the cost is assumed to be zero.
- *Unknown:* there is no information about that cell and can be interpreted as the user wants.
- All other costs are assigned a value between "Freespace" and "Possibly circumscribed" depending on the distance from the "Lethal" cell.

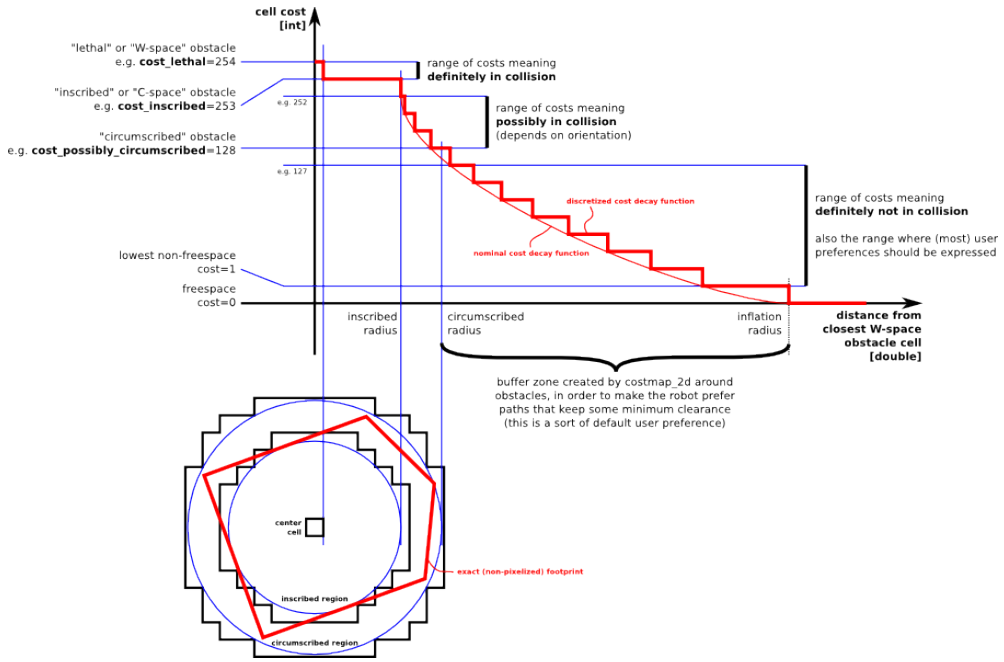

Figure 5.26: Inflation parameters [54]

The Navigation Stack uses costmaps to store information about obstacles in the world. In the definition of the local and global costmap it is possible to use a set of sensors that will pass information to the costmaps. In general, navigation systems perform path-planning on a single costmap storing information in a single grid but it can be a problem in dynamic people-filled environments. For simplicity, it is mainly used the monolithic costmap, which has data stored in the singular grid of value, using one of its for global planning and another for local ones. The problem is given when there are obstacles at a different heights. For this purpose, here, layered costmaps are used. They divide the processing costmap data into semantically-separated layers, as depicted in Figure 5.27.

Hence, each layer tracks obstacles and at the end, the master costmap is modified and updated using an approach called *layered costmaps*. The process of populating the map happens in a way that, data are not stored directly in the grid but maintain an ordered list of layers (each of which takes into account a certain functionality). Then, they are populated into the master costmap. In this way, the updated state is more delineated because, in the layered costmap approach, different types of costmap information are added to separate layers. So, many layers, as the operator desires, can be added.

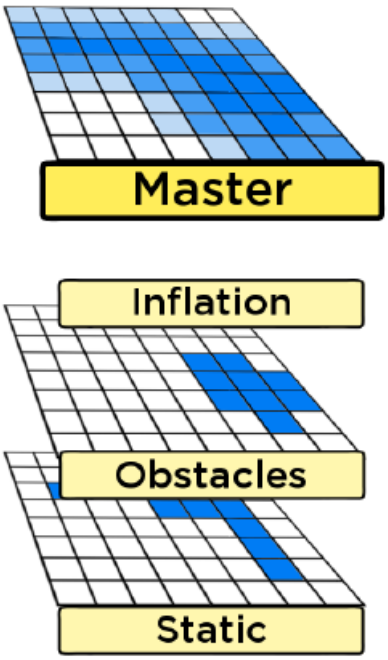

Figure 5.27: Layered costmaps [55]

The main layers used are *Static Map Layer* that is usually generated with the SLAM algorithm to know where walls and obstacles are, it is the bottom layer of the global costmap that copies its value into the master costmap directly; and *Obstacles Layer* used to collect data from sensors as lasers and RGB-D cameras placing them in a 2D grid. Others are *Voxels Layer* that is similar to *Obstacles Layer* but it is used to highlight obstacles seen at different heights, and *Inflation Layer* that inserts a buffer zone around each lethal obstacle that is the location where the robot will be in collision allowing the robot to not go too close to that zone [55].

5.6.4 Move base

This represents the main element of the Navigation Stack and it is used to give safe motion to the robot to reach a final pose.

It requires the sensor topics, odometry data, and amcl [56]. It has global and local planners to perform the navigation task through the use of two costmaps, global and local ones, that contain information about obstacles in two-dimensional space.

The global costmap represents the full environment; instead, the local costmap is a dynamic window that moves in the global one with the robot position. It is made up of five elements:

- *Global planner:* it computes the global path from a start position to a final one avoiding the collision with the obstacle met in the path. It chooses the trajectory of lower-cost relying on the available global costmap. For this system, it is used the *Navfn*, a grid-based global planner that provides a fast interpolated navigation function to create plans for the robot [57]. It uses the path search Dijkstra's algorithm, as in Figure 5.28. The use of this algorithm for a mobile robot is done by transforming the problem into a graphic search method through the information of a grid cell map. It considers the nodes reachable by the robot denoting them as free spaces. As shown in Figure 5.29, a cost is assigned to each of them and it is increased by the necessary number of nodes to pass through to reach each node [58].

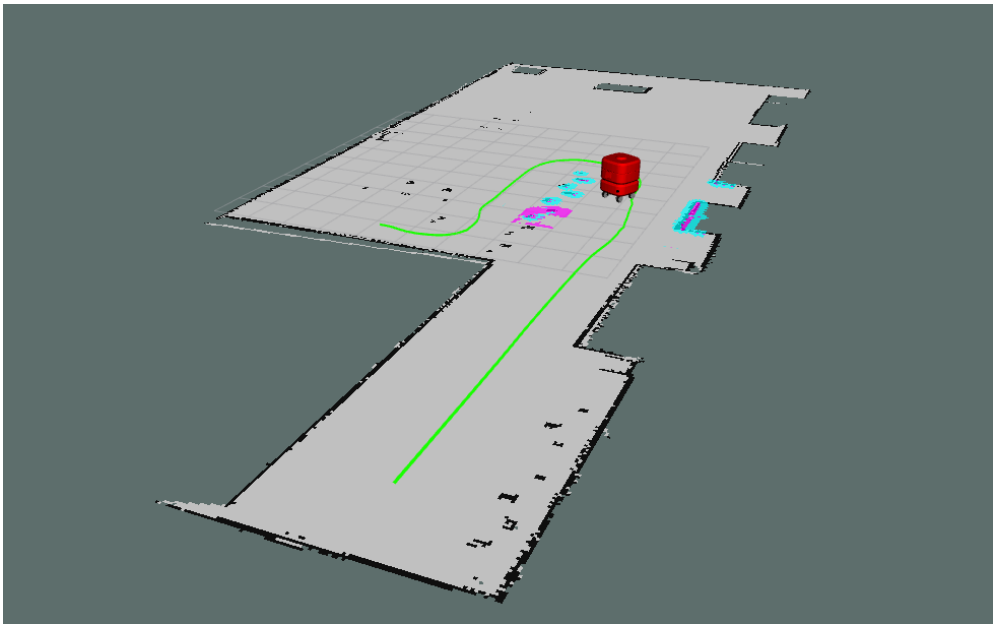

Figure 5.28: Global planner represented by green line in RViz

- *Local planner* it provides the velocity commands to the mobile robot following the global path. It is useful to avoid obstacles that are in the local costmap. Moreover, it allows the detection of smaller obstacles than the global ones due to its smaller shape and its greater definition. The *dwa_local_planner* [59] is used, that is an implementation of the Dynamic Windows Approach: after a

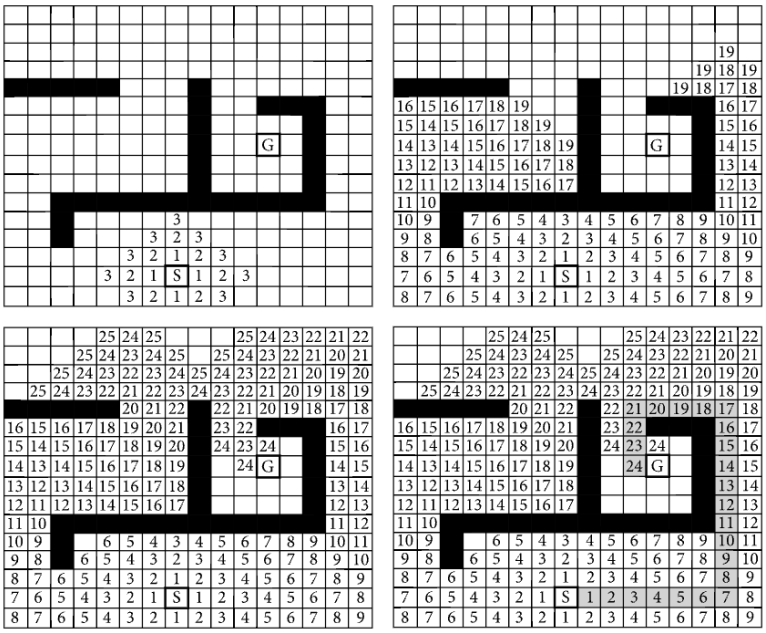

Figure 5.29: Dijkstra's algorithm to find optimal path [58]

sample of the robot's control space, for each velocity, there is a simulation of the robot to see what happens if it is applied to that model. After that, the algorithm evaluates all the trajectories that result from simulation considering goal, obstacles, and speed and discards the trajectories that are not useful. Finally, when it decides what is the best trajectory, it sends the velocity command to the robot. The useful parameters are given by the maximum velocity, acceleration, and also translation velocity, as in the example in Figure 5.30. Between the admissible velocities, a combination of translational and rotational velocities is computed to maximize an objective function. It is continuously updated while it is moving in its path. The main role is to follow the path of the global planner but it should avoid obstacles that are not considered by the global planner. Hence, it is used to create routes in small distances.

- *Global costmap:* it is represented by an occupancy grid map with some adaptable parameters like dimensions and resolution. In particular, it is the reference map of the global planner and it is dynamically updated with sensor data. Each value of the costmap has a value in the range between 0 and 254. An example is given in Figure 5.31.

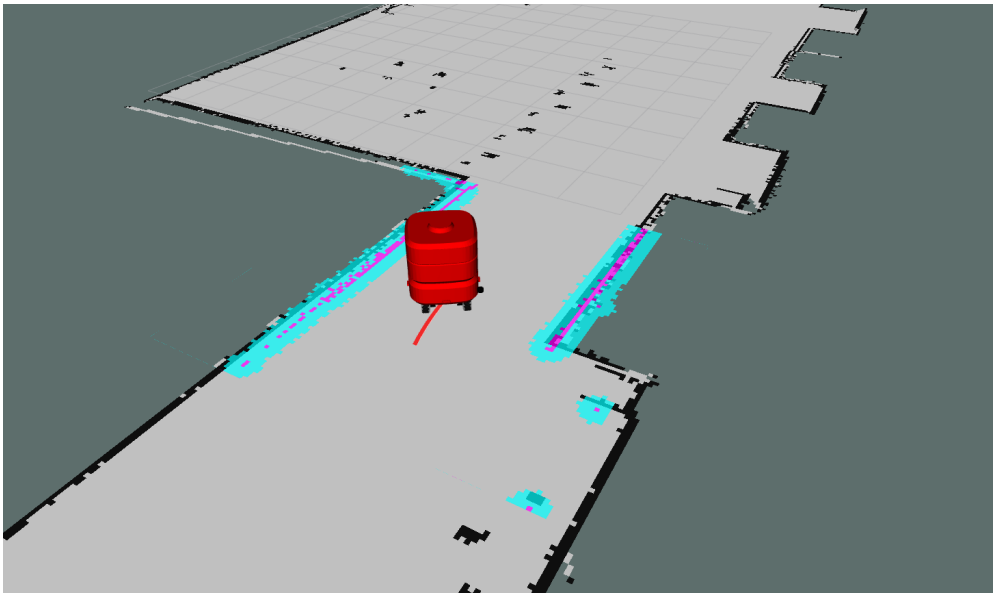

Figure 5.30: Local planner in RViz represented by red line

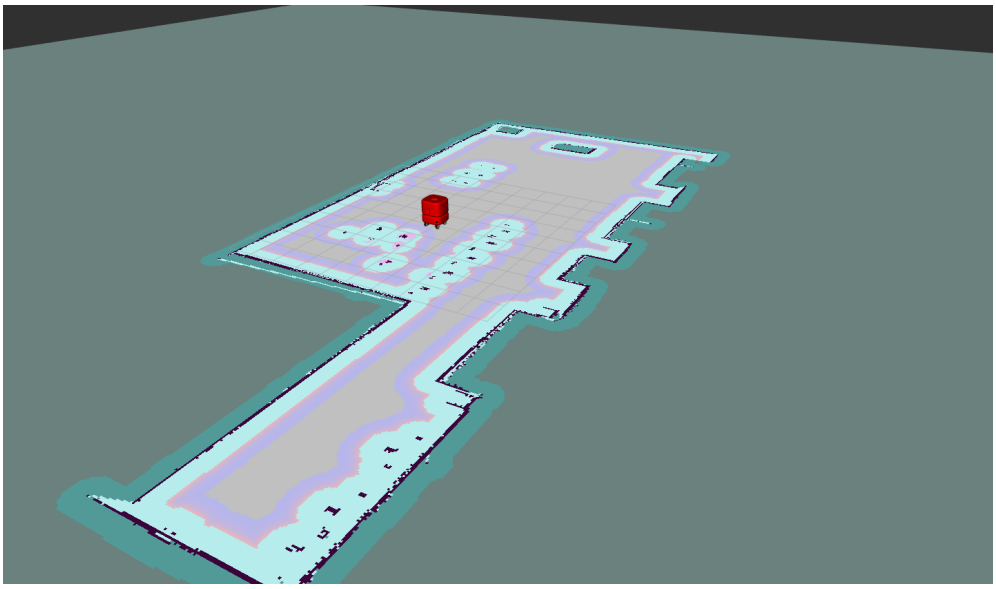

Figure 5.31: Global costmap behavior in RViz

- *Local costmap:* it takes information from the global costmap. Hence, like previously, it is updated with sensor data, the values are defined as the same

as global costmap and it is represented by a local map around the robot-like in Figure 5.32.

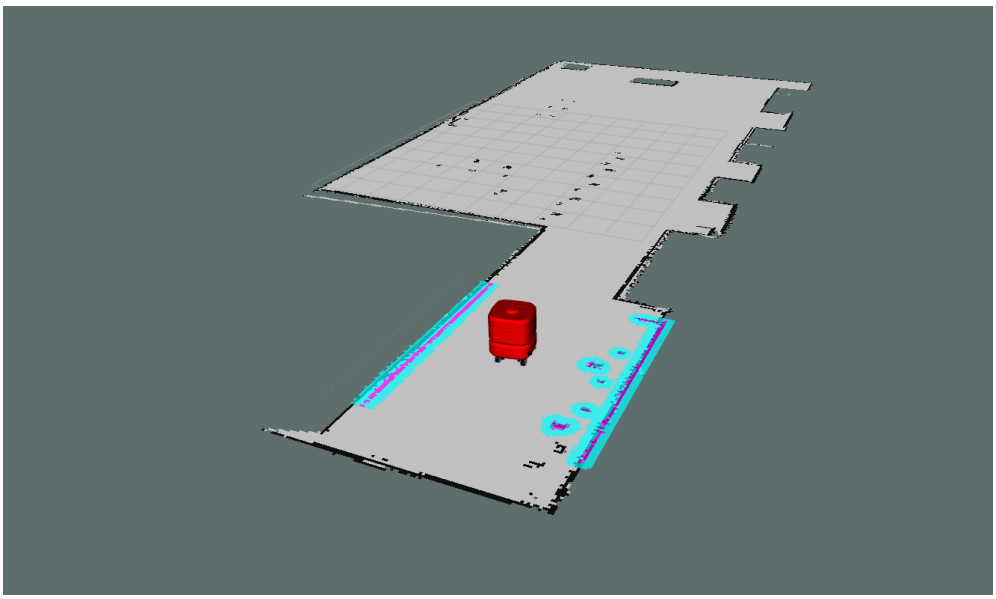

Figure 5.32: Local costmap behavior in RViz

- *Recovery behaviors:* they are not used in this system. However, if the robot does not move anymore because it cannot follow the global path, they are executed. They are made up of different actions to reset and update the local and global costmaps. In the end, if the robot is still blocked it aborts the navigation.

5.6.5 amcl node

This node provides the localization of the robot inside a map and, during the autonomous navigation, it is used with the other nodes. It takes laser scan and gives pose estimates while it moves and perceives the environment. The subscribed topics are represented by *scan* that is the output of the Laser Scan, *map* which represents the map published by *map_server* and the *initial_pose* that represents the initial pose of the robot inside the map. Instead, *amcl_pose* is the published topic. It gives an estimation of the pose of the robot inside the map [60].

This node uses the Adaptive Monte Carlo Localization method based on the Monte Carlo Localization that estimated the pose using a Particle Filter. This localization method is very easy to implement and it has many advantages with

respect to other works. It is preferable than the Kalman Filter, because it represents multi-modal distributions giving the possibility to globally localize the robot and the Markov localization, because it decreases the amount of memory and it is more accurate because the state represented in the samples is not discretized [61].

The Monte Carlo Localization (MCL), using a particle filter, can represent the distribution of the likely states in which each particle represents a possible state and where the robot could be in the map. Initially, it is assumed that the robot could be anywhere in space. Therefore, a uniform random distribution of particles is used. Subsequently, whenever the robot moves, it shifts the particles to predict the new state and if it senses something, the particles are resampled based on the Bayesian recursive estimate. The particles represent the guesses about where the robot might be on the map. So, each particle is a description of the possible future state, and these are discarded if inconsistent with the observation of the environment or other particles are generated and placed close to consistent ones. Finally, the particles should converge towards the robot's true position [62].

This node uses an improved version of the (MCL) that adaptively samples the particles based on an error estimate using the Kullback-Leibler divergence (KLD) [63]. The main difference between MCL and AMCL is that AMCL contains a dynamic set of particles, whereas MCL has a constant number of particles. Hence, AMCL generates fewer particles if it is more certain about its position and more particles in case of uncertainty. So, it covers all possible robot states. Hence, AMCL outperforms classic MCL making the filter converge faster.

5.7 Real-Time Object Detection

In the system, real-time object recognition has been implemented through the use of YOLO [64]. The name stands for "You only look once" and it is used as an object detection platform for real-time image processing. Through this system, it is possible to know objects detected and where they are placed in the space. YOLO, using a single convolutional network that can be trained to improve the accuracy, is very fast compared to other previous detection systems. It can detect a variety of objects simultaneously.

5.7.1 Working principle

The network employed uses features taken from the entire image to predict each bounding box. The input image is divided into a S X S grid, if the center of an object falls into one of these grid cell, that one is responsible for detecting that object. Each of these cells predicts bounding boxes B that describe a rectangle

that encloses one object and also scores to tell users how certain it is to have an object into that bounding box. For each bounding box, a classifier predicts a class and then the confidence score and class prediction are combined. The final number of bounding boxes is given by S X S X B .

Summarizing, Yolo, for each cell, predicts B boundary boxes, classifies one object and predicts C conditional class probabilities based on the train dataset used. Each boundary box has five elements:x, y, w, h, and confidence. The first four values denote the position and dimension of the box and they are normalized by the image width and height [64]. The final prediction has a shape defined by:

$$(S, S, B\ X\ 5\ + C) \tag{5.14}$$

Yolo should predict multiple bounding boxes for each grid cell, so there can be multiple losses for the true positive. Hence, only one of them should be chosen. This is possible considering the highest IoU (intersection over union) with the ground truth, making it easier to recognize sizes and aspect ratios of bounding boxes. The loss function, when only one of each cell prediction is predicted, can be computed by the composition of Classification loss, Localization loss, and Confidence loss.

The limitations of YOLO are given from spatial constraints on bounding boxes, because each grid cell predicts two boxes and it can have one class. This will result in the wrong detection with the nearby object that this model can predict, such as a small object in groups. However, in comparison to other detection systems, this method has good performances regard object detection in computer vision [64]. Hence, it is better than other detection systems as *Deformable parts models* because all the deformable parts are replaced with a single convolutional neural network performing future extraction, bounding box prediction, and others concurrently. Instead, it is similar to *R-CNN* with the difference that this system puts spatial constraints on the grid cell proposals helping in the mitigation of multiple detections of the same object combining individual components into a single optimized model.

Implementation of YOLO

In this system, the Darknet implementation of Yolo, available online, is used [65]. It is a ROS package developed for object detection through images from a camera. It can be used with CPU being very fast, but it becomes much faster on GPU where it is possible to install CUDA that is a parallel computing platform and application programming interface (API) model created by Nvidia.

Here, a pre-trained model of the convolutional neural network is used. It can detect some classes including data set from VOC and COCO.

This ROS package is used in parallel with another package that provides the position of the object in 3D space. Using *darknet_ros_3d* [66], it is possible to obtain 3D bounding boxes of an object contained in an object's list where it is specified its position in the space.

Using YOLO, it is possible to obtain the three minimum and maximum coordinates in each axis providing the location of the object. The distance between the camera and the detected object is calculated by finding the center of the object.

So, the coordinates of the point will be:

$$Xc = \frac{(Xmax + Xmin)}{2} \tag{5.15}$$

$$Yc = \frac{(Ymax + Ymin)}{2} \tag{5.16}$$

$$Zc = \frac{(Zmax + Zmin)}{2} \tag{5.17}$$

Then, the object distance-vector, represented by $V = (Xc, Yc, Zc)$, has been obtained. The distance of an object from camera, in meters is found as:

$$D = \sqrt{Xc^2 + Yc^2 + Zc^2} \tag{5.18}$$

In the developed configuration Yolo reads, as input, the topic from the depth camera denoted as *camera/color/image_raw* and provides as output */bounding_boxes*. The main standard parameters, in this last topic, are:

- **object_name**: it is the name of the object detected;
- **probability**: it is the probability of certainty;
- **xmin**: it represents the X coordinate in meters of the left upper corner of bounding box;
- **xmax**: it represents the X coordinate in meters of right lower corner of bounding box;
- **ymin**: it represents the Y coordinate in meters of the left upper corner of bounding box
- **ymax**: it represents the Y coordinate in meters of the right lower corner of bounding box

- **zmin**: it represents the cooordinate in meters of the nearest pixel of bounding box
- **zmax**: it represents the coordinate in meters of the furthest pixel of bounding box

The central point of the object of each coordinate, denoted as **xcen**, **ycen**, **zcen**, is added to the previous parameters together with **D**, which denotes the distance from the camera in meters.

For this kind of implementation is used the version yolov2-tiny that is much lighter in terms of computational cost. It can detect a certain class of objects selecting a certain threshold. Figure 5.33 shows the behavior of detecting a person in a simulation environment. Instead, in Figure 5.35 it is possible to see the detection of some simple objects at the CIM 4.0, their name, and the distance (Figure 5.34) at which they are located. Finally, Figure 5.36 shows how multiple complex objects are detected in CIM 4.0 environment. While the robot is moving, as it is possible to see on the top right of the image, it is capable of detecting multiple objects providing also the distance from them.

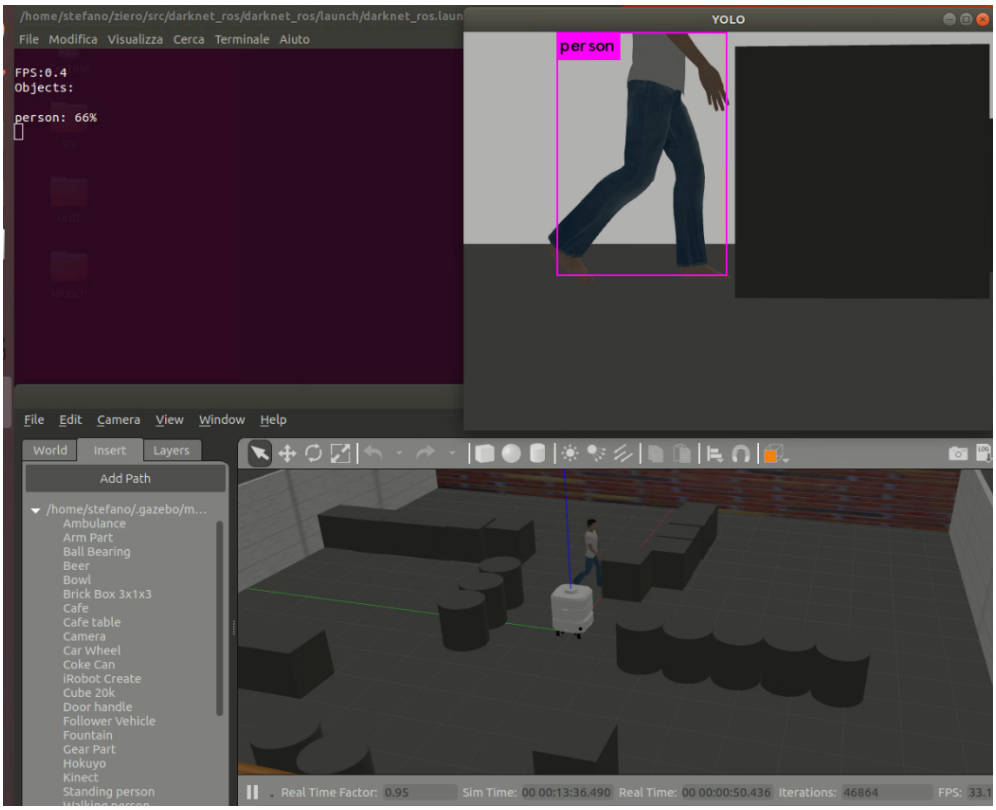

Figure 5.33: Person Detection in simulation environment

```
bounding_boxes:
  -
    Class: "bottle"
    probability: 0.739813089371
    xmin: 0.318248480558
    ymin: 0.0817204937339
    xmax: 0.856543779373
    ymax: 0.171585217118
    zmin: -0.0461931452155
    zmax: 0.152522727847
    xcen: 0.587396144867
    ycen: 0.126652851701
    zcen: 0.0531647913158
    D: 0.603242635727
  -
    Class: "mouse"
    probability: 0.760025799274
    xmin: 0.298518925905
    ymin: -0.127948746085
    xmax: 0.40471085906
    ymax: -0.0425845831633
    zmin: -0.0362912490964
    zmax: 0.0056885718368
    xcen: 0.351614892483
    ycen: -0.0852666646242
    zcen: -0.015301338397
    D: 0.362129211426
```

Figure 5.34: Distance between objects and camera

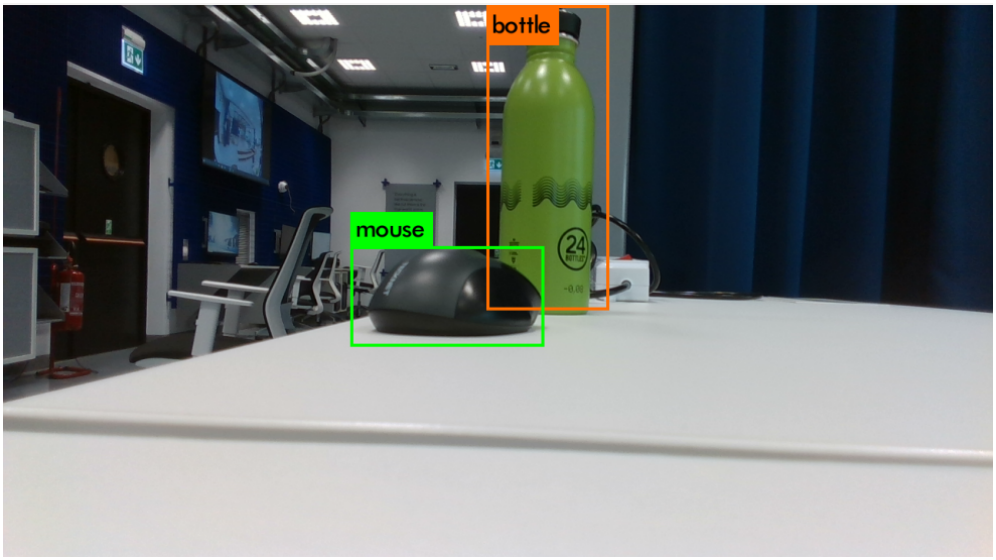

Figure 5.35: Detection of bottle and mouse objects with camera d435i

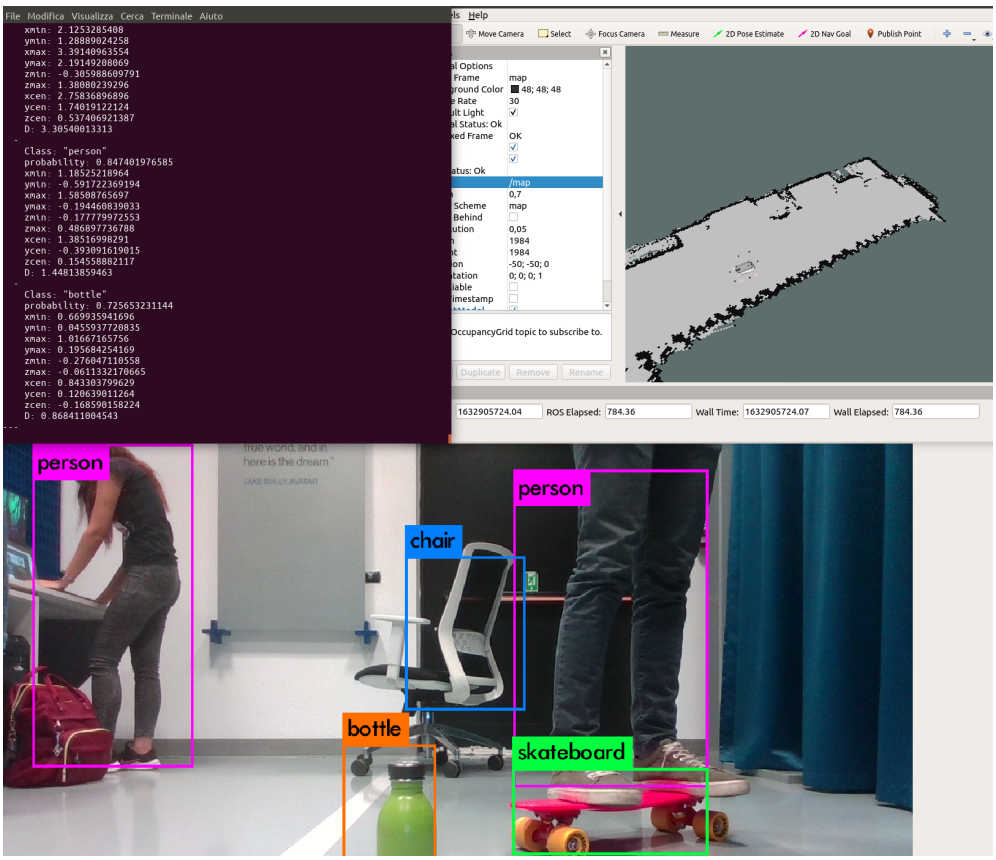

Figure 5.36: Detection of multiple objects in complex environment in real time while robot is moving

Chapter 6

Hardware implementation and experimental tests

Considering the type of application, appropriate sensors are chosen to develop the autonomous mobile robot. First of all, the robot consists of four Mecanum wheels. These type of wheels allows omnidirectional movement and therefore allow both rotation and translation. Although its remarkable maneuverability is very useful for moving in tight spaces, it leads to a great waste of energy efficiency [67]. Other than this, they have a big problem due to vibration and slippage during their movement. This obviously cannot be overlooked because it can lead to the accumulation of errors over time.

In self-driving robots, encoders are often used to have an estimate of the position and therefore of the odometry. Due to the problems described above, the use of the encoders alone is not enough to acquire correct information because the number of turns for moving laterally and longitudinally is not the same and changes with the ground condition. To overcome this problem, many solutions have been adopted in the years to obtain the odometry with these kinds of wheels. Some solutions include optical encoders located in the servomotor to obtain rotation information and ball spherical encoders to have pose information [67]. Others include the use of a video camera where different frames are captured and compared obtaining speed and direction respect a point or a reference previously calculated [68], or the use of two optical mice directed on the down part of the robot and positioned in the front and back part obtaining angular rotation through the difference between front and rear displacement [69]. In this application data from encoders are fused with an Inertial Measurement Unit (IMU) through the use of an Unscented Kalman Filter.

In addition, two LiDARs are also used for navigation. In particular, the use of two LiDARs is due to the need to cover a space of 360°. As it is possible to see from Figure 6.1, above the robot, there is a rectangular surface used for maintenance purposes that do not allow to allocate a single sensor only on one side because in this way we will have only a limited view and the rover could be limited in the calculation of the route and could end up in a collision with obstacles. There

Figure 6.1: Basic system configuration without sensors

can be two possible solutions for the position of sensors. The use of LiDARs is fundamental for many purposes. In fact, to acquire a map a laser scan is often used because, with its long-range, it can cover a long space without passing again in that place and so avoiding accumulating errors. Another reason, to use the LiDAR is that the laser scanner operates at different atmospheric conditions with a large field of view of 360°. The problems that cannot be underestimated, come from the fact that laser scanners can only scan in an horizontal mode and therefore can only see objects at the height of the laser, neglecting and failing to see objects that are at a higher or lower height, as can be seen from Figure 6.2.

To overcome this problem and therefore to have a robust system capable of seeing obstacles at any height, a good solution is to introduce depth cameras. In fact, they allow to cover both a horizontal and a vertical field of view and therefore to overcome the limits of the LiDARs. However, some disadvantages may arise from environments that are featureless such as white walls or long corridors where it is difficult to see the depth. In addition, 3D operations are computationally heavy

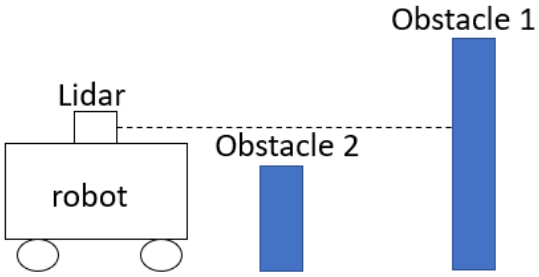

Figure 6.2: Laser scan limits

and therefore only PointCloud data are used.

Thus, there are two final configurations. Both have been simulated and tested in reality. The first configuration is the one that has been chosen as the final configuration. In both the position of the LiDARs is the same, instead what is different is the position and number of cameras. In particular, the simplest configuration regarding the position of the cameras provides an arrangement of them in the front and in the rear, as in Figure 6.3. On the other hand, the second configuration provides for an arrangement of cameras on all four sides of the system as in Figure 6.4. However, this last configuration is very complex due to the computational cost given by the streaming of the data obtained from the cameras.

6.1 Intel RealSense Depth Camera D435i

This type of sensor (Figure 6.5) is widely used in robotics due to its great performance and low cost. It is a depth camera that also includes an Inertial Measurement Unit (IMU). The components of IMU used for this kind of application are an accelerometer that measures the total force acting on the device and the gyroscope that measures the angular velocity. Together, they give the orientation in 3D space.

It is often preferred in applications like object recognition and collision avoidance, autonomous navigation and mapping solutions, but also for the gesture and poses detection.

Its large field of view (FOV) does not leave blind space and due to its low sensibility to light, it allows navigation also in space with the light off [70]. Moreover,

Figure 6.3: First configuration with two LiDARs in the diagonal and two depth cameras in the front and back

Figure 6.4: Second configuration with two LiDARs in the diagonal and four depth cameras, one on each side

navigation is allowed both outdoor and indoor without having interference with a system containing multi-camera configuration, with good accuracy within several meters operating at low power.

Figure 6.5: Intel RealSense depth camera d435i

The communication with the camera and all systems is very simple. It is also available the open-source developers kit Intel RealSense SDK which permits the integration with a lot of operating systems and programming languages like python.

As shown in Figure 6.6, the d435i is made up of different components. Moreover, the data that are acquired produce simultaneously RGB and depth images. In particular, it is composed of an RGB module with a frame resolution of 1920x1080, two infrared modules capable to acquire infrared images, and an IR projector that improves the performances of this depth camera using an active stereo method. The depth field of view is 87°x58° (FOV) with a depth resolution of 1280x720 at 90 fps. It contains the integrated Intel RealSense Vision Processor D4 which permits a detailed reconstruction of the environment processing the images acquired. The maximum range of vision is 10 meters but its accuracy depends on many factors as scene, calibration, and lighting conditions.

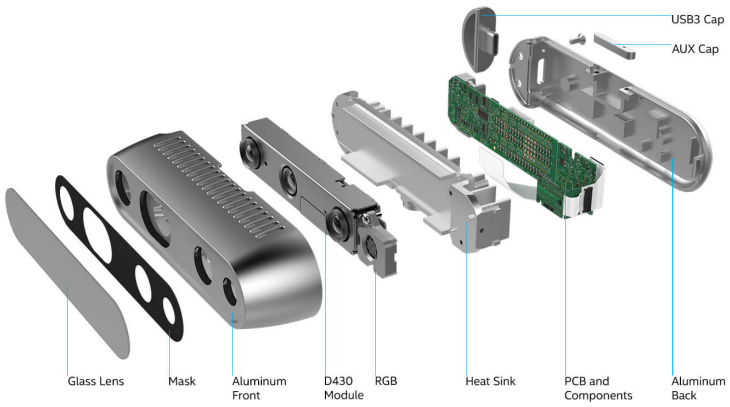

Figure 6.6: Internal structure of d435i [70]

The main features and components are shown in Table 6.1.

	Parameters	Description
Physical	Length × Depth × Height:	90mm × 25mm × 25mm
	Connectors:	USB-C 3.1
Components	Camera module:	Module D430 + RGB Camera
	Vision processor board:	Vision Processor D4
Features	Use:	Indoor/Outdoor
	Ideal range:	.3 m to 3 m
	Image sensor technology:	Global Shutter
RGB	RGB frame resolution:	1920 × 1080
	RGB sensor FOV (H × V):	69° × 42°
	RGB sensor technology:	Rolling Shutter
	RGB frame rate:	30 fps
Depth	Depth technology:	Stereoscopic
	Depth output resolution:	Up to 1280 × 720
	Depth Field of View (FOV):	87° × 58°
	Depth frame rate:	Up to 90 fps
	Depth Accuracy:	<2% at 2 m[1]

Table 6.1: Main features and components Intel RealSense d435i

6.2 RP-LIDAR A1

In the considered system, the RP-LIDAR A1 (Figure 6.7) laser scanner is used. It is widely used in robotics and in particular for autonomous navigation, localization, and mapping, because it represents a low-cost compact size sensor capable of sensing an environment rotating of 360 degrees. It is a 2D laser scanner, developed by SLAMTEC, composed of a range scanner that rotates clockwise through a motor on which there is a belt.

Moreover, it is based on the laser triangulation ranging principle and uses high-speed vision acquisition. In particular, the RPLIDAR emits an infrared laser signal and then the returning signal is detected and sampled by the vision acquisition module. It has 12 meters range scan measuring distance data more than 8000 times per second [71].

This sensor with very low power consumption has a configurable scan rate from 2-10 Hz. Other technical specifications are listed in Table 6.2.

Figure 6.7: RP-LIDAR A1

	Parameters	Description
Physical	Width x Length x Height:	96.8 x 70.3 x 55 mm
	Weight:	170g
Features	Use:	Indoor/Outdoor (without sunlight)
	Measuring Range:	0.15m - 12m
	Range Resolution:	$\leq$ 1 % of the range $\leq$ 12m
	Sampling Frequency:	8K
	Rotational Speed:	5.5Hz
	System Voltage:	5V
	System Current:	100mA
	Temperature Range	0℃-40℃
	Angular Range	360°
	Angular Resolution	$\leq$1°
	Accuracy :	1 % of the range $\leq$ 3 m
		2 % of the range 3$-$5 m
		2.5 % of the range 5$-$25 m

Table 6.2: Main features of RPLIDAR A1

6.3 Boards

6.3.1 NVIDIA Jetson Xavier NX

This board belongs to the NVIDIA Jetson that is a low-power embedded platform with high performance. In particular, the Jetson Xavier NX (Figure 6.8) contains

both a Central Process Unit (CPU) and GPU on a single chip of small size (70 mm x 45 mm). It is designed for robots applications and autonomous tasks, it is

Figure 6.8: NVIDIA® Jetson Xavier™ NX [72]

adapt for real-time execution and to perform neural networks computations due to its integrated software libraries as CUDA. Moreover, it is equipped with four 3.1 USBs, requires MicroSD to work, and supports multiple power modes.

It offers up to 21 TOPS, giving the possibility of high-performance calculation and allowing to run multiple networks in parallel, processing high-resolution data coming from multiple connected sensors [72]. More detailed features and performance are listed in Table 6.3 .

6.3.2 NVIDIA Jetson Nano

This kind of board (Figure 6.9) is widely used in robotic applications, because it has very good performance, it is low cost and it has the smallest size in the family of Jetson. Moreover, it needs only 10 W of power if set in high performances. It uses ARM Cortex CPU and a 128 core Maxwell GPU and it is often used for image processing due to its relationship between processing power and low power consumption. Therefore, it is used also for running multiple neural networks in parallel for applications like image classification, object detection, segmentation, and speech processing [73].

The only problem with this board is given by the fact that it does not have a wireless module. To overcome this problem, a dual-mode (2.4GHz / 5GHz) wireless NIC module of Intel has been applied to Jetson Nano.

In addition, the Jetson Nano has two possible types of power supply. One is

Parameters	Description
Width x Length x Height:	103 mm x 90,5 mm x 34 mm
GPU:	NVIDIA Volta architecture with 384 NVIDIA CUDA® cores and 48 Tensor cores
CPU:	6-core NVIDIA Carmel ARM®v8.2 64-bit CPU 6 MB L2 + 4 MB L3
Memory:	8 GB 128-bit LPDDR4x @ 51.2GB/s
Connectivity:	Gigabit Ethernet, M.2 Key E (WiFi/BT included), M.2 Key M (NVMe)
Display:	HDMI and display port
USB:	4x USB 3.1, USB 2.0 Micro-B
Others:	GPIO, I^2C, I^2S, SPI, UART

Table 6.3: Main features of NVIDIA® Jetson Xavier™ NX

Figure 6.9: NVIDIA® Jetson Nano™ [73]

given by the barrel-jack and is preferable to obtain better performance or to do operations that require more and constant power, the other through the type-c connector. Other technical specifications are listed in the Table 6.4.

Parameters	Description
Width x Length:	69 mm x 45 mm
GPU:	128-core Maxwell
CPU:	Quad-core ARM A57 @ 1.43 GHz
Memory:	4 GB 64-bit LPDDR4 25.6 GB/s
Connectivity:	Gigabit Ethernet, M.2 Key E,
Display:	HDMI and display port
USB:	4x USB 3.0, USB 2.0 Micro-B
Others:	GPIO, I^2C, I^2S, SPI, UART

Table 6.4: Main features of NVIDIA® Jetson Nano™

6.4 Scout mini rover

The robot used is the Scout Mini mobile base, which is the compact version of the Scout 2.0 provided by AgileX Robotics (Figure 6.10). It represents one of the leading autonomous mobile robots in the market for its speed, agility, compactness, and compatibility. Being ROS-compatible, it can be used to carry out various missions including surveillance, exploration, detention, and various educational, agricultural, and logistic services.

Usually, this type of robot can support a maximum load of 10-20 kg, but the robot used for this system is equipped with four Mecanum Wheels guaranteeing a payload of up to 50 kg. This kind of special wheels gives the possibility to translate and rotate in place in any direction. In addition, with a very small size of 625x585x222 (L × W × H (mm)) and a weight of 20 kg manages it achieves a high-speed, precise, stable, and controllable power control system of 10.8km/h.

Through its CAN interface (Figure 6.11) used as a communication interface, it is possible to install and communicate with all the elements necessary to operate with this robot. For this purpose, two aviation male plugs are supplied along with SCOUT MINI. In particular, the robot adopts a standard CAN2.0B communication with a baud rate of 500K. Via external CAN bus interface, the moving linear speed and the rotational angular speed of the chassis can be controlled; the robot will feedback on the current movement status information and its chassis status information in real-time, motor current, encoder, and temperature. Furthermore, this robot can be remotely controlled manually through an FS RC transmitter, which can be used to easily control the robot chassis through a left-hand-throttle configuration. It also contains some controls for lights, speed, and positions [74].

Figure 6.10: SCOUT mini omnidirectional [74]

Figure 6.11: Aviation Male Plug for CAN cable connection [74]

In particular, a CAN-USB adapter is used to connect the robot to the Nvidia Xavier Nx and the open-source software package (SDK) available in [75] provides a C ++ interface to communicate with the mobile platform provided by AgileX Robotics to send commands to the robot and receive the latest robot status.

6.5 Distribuited Hardware Architecture

The system is developed using Ros Melodic on Ubuntu 18.04, which is a distribution based on the Linux kernel. It is downloaded and installed on every board present in the AMR. In particular, given a large number of sensors, it was preferred to have a distributed architecture to distribute the work on several boards and therefore reduce the computational cost.

At first, multiple configurations were made and the best is the sensor connection which includes two LiDARs and two depth cameras. To do this, two LiDARs and the rover platform are connected to Nvidia Xavier Nx which, allows for good results thanks to its high performance. In fact, on that board, not only the data acquisition of the connected sensors but also the main software that allows autonomous navigation is run. However, given the limited number of USB ports, due to high computational and power cost, it was decided to stream cameras data using a Jetson Nano. Through this system, represented in Figure 6.12, it is possible to distribute the computational load. A first prototype of all the systems connected together is shown in Figure 6.13. Hence, to emulate the final system, a wooden support is created where the sensors are arranged as in the final configuration. In the beginning, this prototype was created with simple plywood, which was later replaced by heavy wooden boards to avoid vibrations due to the type of robot wheels.

Figure 6.14 shows multiple kinds of obstacles at different heights. They are represented by a chair, a stool, and a bottle. Obstacles perceived by the sensors are purple and blue colors. In particular, the white points denote what is seen by LiDAR. It is able to see only the obstacles at its height and therefore only part of the chair and the stool. Thanks to the use of the camera, it is possible to see obstacles at each level including the bottle and other parts of the obstacles listed above.

6.5.1 Network setup

ROS is a distributed system that allows the connection of multiple devices connected to the same network.

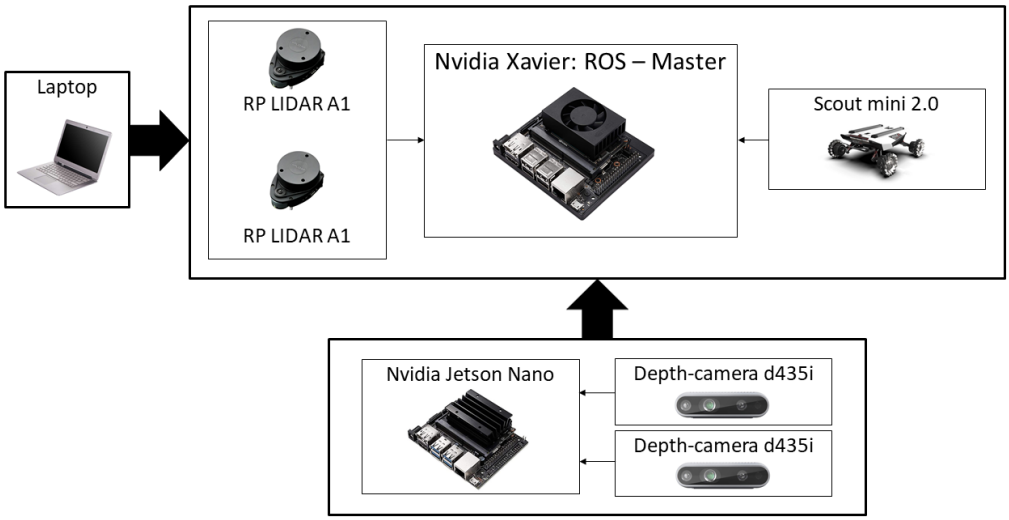

Figure 6.12: Hardware connection of the system

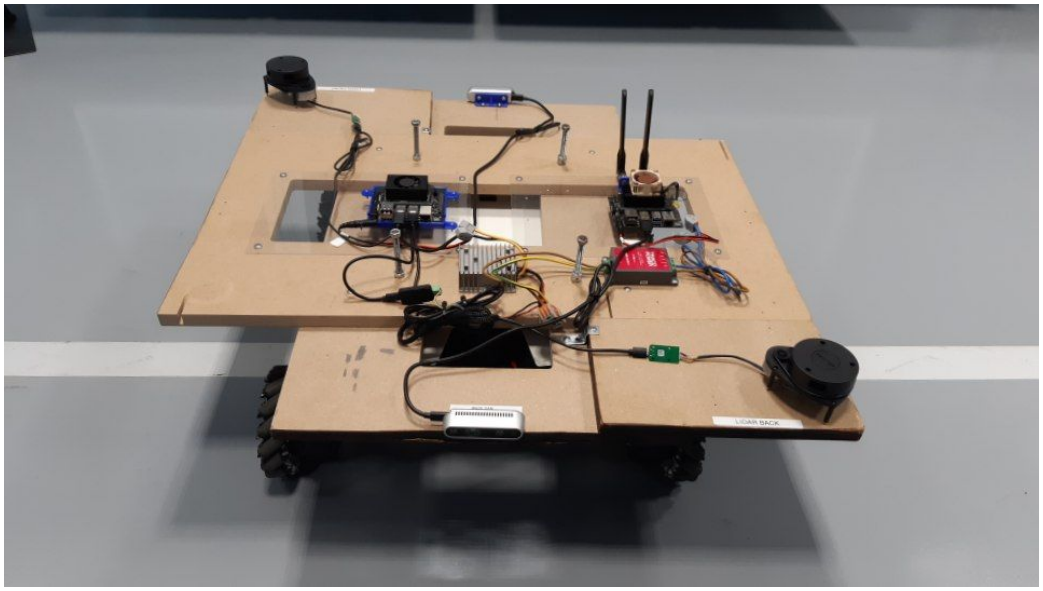

Figure 6.13: Prototype of hardware connection of the system

The hardware connected is: a remote laptop, where RViz is run, which allows the three-dimensional display of what the robot sees and to give commands to operate,

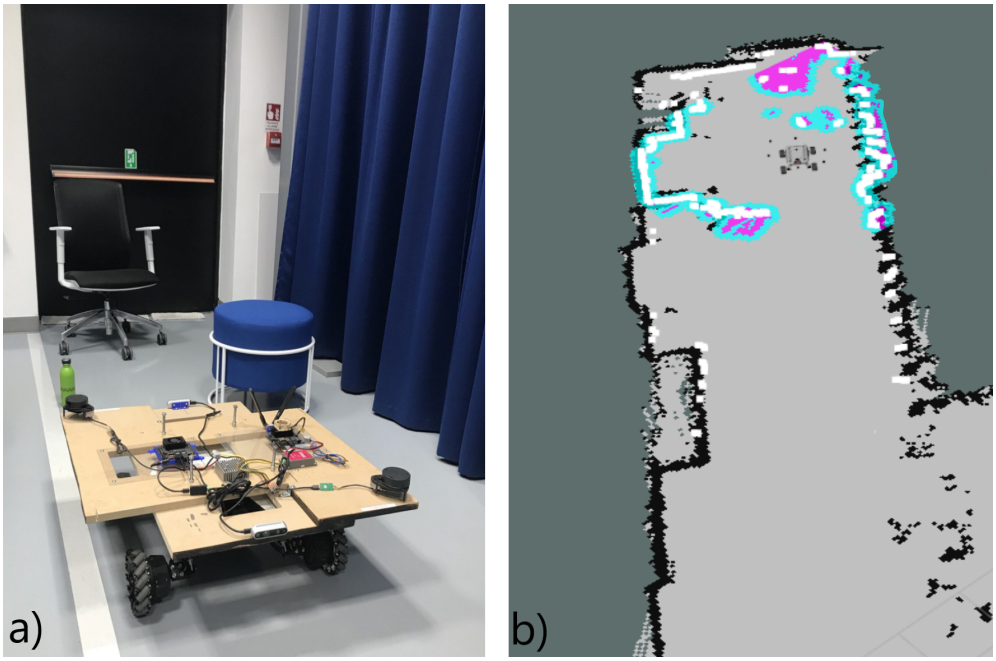

Figure 6.14: Comparison of obstacles seen in reality (a) and on Rviz (b)

the Nvidia Xavier Nx, which connects two lidars and the Scout rover and executes the main software, the Jetson Nano to which two cameras are connected and only used to acquire the data. Then, these are read by Xavier Nx, which can read and operate with its published topics.

A ROS system can consist of many nodes that run in parallel, across different multiple machines, and that can communicate with each other. This is possible because ROS is a distributed computing environment. To do this ROS needs bi-directional connectivity on all the machines to be connected, and each machine must have a name visible to all the others [76].

First of all, the machines should be connected to the same Wi-Fi network and then one has to look up each machine's IP address through the terminal of each computer used through the $\$ifconfig$ command. The next step is to edit the *.bashrc* file by adding it to the end of the document:

- export $ROS\_MASTER\_URI = http://<remote\_PC\_IP>: 11311$
- export $ROS\_HOSTNAME =< current\_PC\_IP >$

In particular, the first line denotes the address of the PC of which it is set the ROS

Master. In this case, this corresponds to that of the Nvidia Xavier Nx, and this line must be copied to all the other connected computers because this parameter identifies the IP where each device looks for the ROS MASTER node. Instead, the second line denotes the address of the PC that is working and therefore, must be connected to the ROS Master indicated above, as in Figure 6.15 . First of all, to

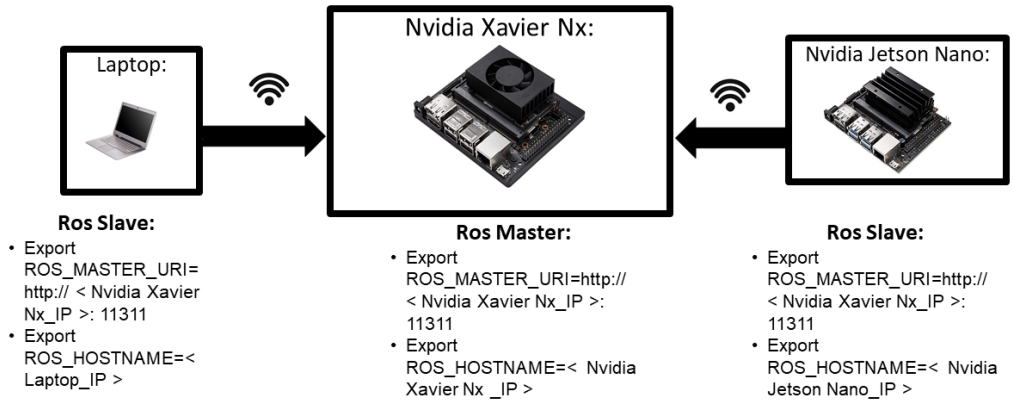

Figure 6.15: Network connection of pc, Nvidia Xavier Nx and Jetson Nano

start the autonomous navigation, a launch file is run in the terminal corresponding to the Nvidia Xavier Nx, which also includes the initialization of the ROS MASTER node. Subsequently, the launch file of the d435i depth cameras is launched in the terminal corresponding to the Jetson Nano. Finally, from a remote PC, it is possible to run RViz to see and execute the commands desired by the user to make the robot move.

Considering this architecture, another problem is given by having to differentiate more sensors of the same type. The two depth cameras used are distinguished by the serial number of each one. In fact, each Realsense d435i has a unique serial code that allows them to be distinguished from others of the same type. First of all, Intel RealSense SDK 2.0 (librealsense) for Linux and the ROS wrapper for librealsense should be installed. After, launching the $realsense2\_camera$ wrapper through the command $roslaunch\ realsense2\_camerars\_camera.launch$, the serial number will be indicated. So, after having done this procedure for both

cameras, through a launch file, each camera has been assigned a different name from the other and different serial numbers.

The two RP LiDARs are distinguished assigning to each of them a different parameter regarding the *serial_port* parameter. In particular, it is assigned */dev/ttyUSB*0 for the front LiDAR and */dev/ttyUSB*1 for the rear one. These corresponds to the respective ports to which they are connected. In addition, the name of the two is also changed to be distinct.

6.6 SLAM experimental tests

The SLAM is done by trying different settings.

In particular, two map acquisitions are tested. Using this type of algorithm it is possible to use only a single laser scanner source. The first test is done using only a front LiDAR, the second one using both the two LiDARs placed on front and on back.

Using only the front LiDAR, the topic */scan_frontal* is used. The robot is moved through the remote control to acquire the map at the speed and in the way it is desired. Using only one LiDAR, the acquisition of the map is slower because using this type of system, despite using a LiDAR capable of seeing at 360 °, the viewing range is limited to 270 ° due to the maintenance device placed above the robot. Therefore, in order to acquire the map, the robot must be rotated on itself several times, as shown in Figure 6.16.

When both LiDARs fused together are used in a single topic called */scan_fused*, the map acquisition takes place much faster than before. So, there is no need to rotate the robot several times by 360° because the whole range of vision is covered, as shown in Figure 6.17. Although the second setup allows the fastest map acquisition without the accumulation of position error that usually results from robot rotations due to Mecanum wheels, it provides a less accurate map. Figure 6.18 shows the acquisition of map in both tests. The map acquired with a single LiDAR appears sharper and therefore with more detected objects than the one obtained with two LiDARs where some objects are not well marked, especially in the central part of the map. So, in terms of accuracy, the result obtained with only one LiDAR is better than that with two ones, even if it takes longer time. This is due to the fact that using both LiDARs, even if fused together, they require the processing of a greater amount of data. So, proceeding quickly in the building of the map, some objects could be left out or not defined accurately.

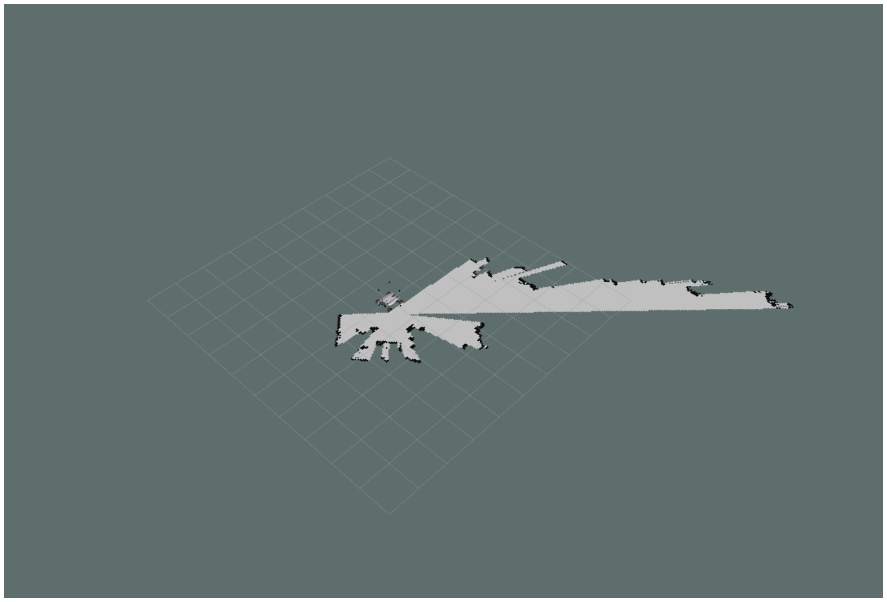

Figure 6.16: SLAM with only one frontal LiDAR

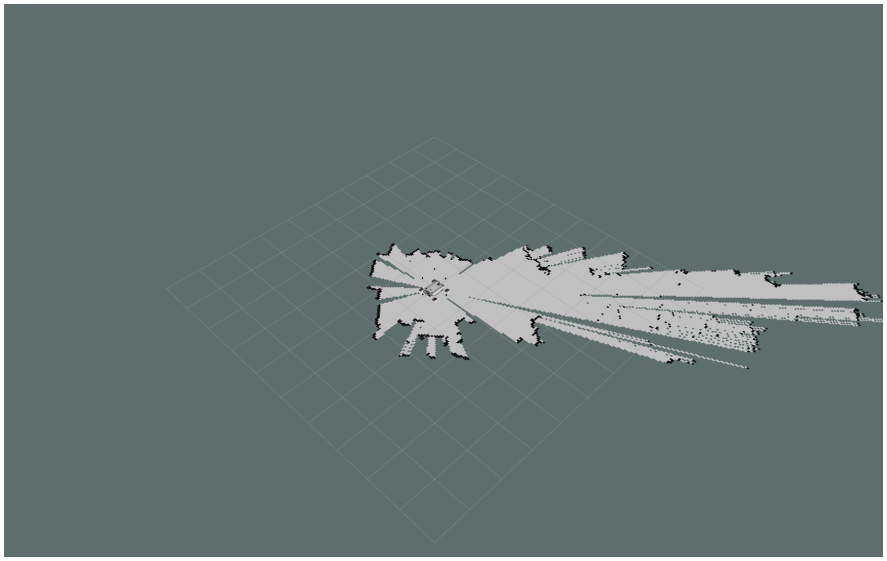

Figure 6.17: SLAM with frontal and back LiDAR

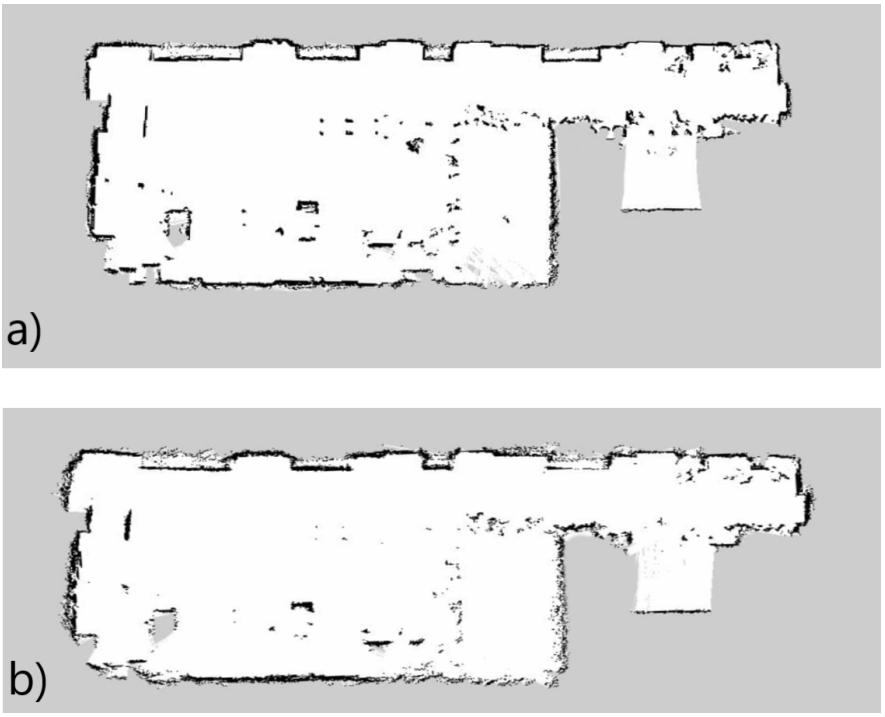

Figure 6.18: Map acquisition of the CIM 4.0 environment. In figure (a) in the upper part of the image it is possible to see the accurate acquisition with only one LiDAR. Instead in figure (b) in the lower part, it is possible to see the less accurate acquisition of the map with two LiDARs.

6.7 Experimental navigation tests

Figure 6.19 shows the final system which includes the cameras on the front and back sides and the LiDARs positioned in the diagonal of the maintenance system. Two types of tests have been done in real life where the aspect of localization is interesting. For this reason, the actual path followed has been visually compared with the planned one. In the first test, the robot navigates acquiring odometry information only through the encoders. As it is possible to see from Figure 6.20, the robot deviates from the planned behavior and therefore from the path that it should follow. It deviates considerably in the case of curves and when it moves sideways because the number of revolutions of the wheels to move straight is different from those to move sideways. Especially in the case of rotations, sometimes the orientation of the robot is incorrect. Instead, the second test is done by acquiring the odometry information from the result obtained by merging the IMU data with

Figure 6.19: Final configuration of robot equipped with the sensory system

the encoders one through the Unscented Kalman Filter. As can be seen from Figure 6.21, the navigation is much better than in the previous case. In particular, the movement of the robot and its orientation is almost the same as the planned case, and therefore the result obtained is considerably better. Moreover, the ability of the wheels to move in tight spaces has been tested, like in the simulation, with good results.

Experimental tests have been done to evaluate the behavior of the robot in the case of static and dynamic obstacles. For the first one, they are easily avoided because they have been detected during the map acquisition so, the global path planning algorithm plans the trajectory taking into account their presence, as shown in Figure 6.22.
Dynamic obstacles, i.e. those that are not present in the construction of the

Figure 6.20: Map of CIM 4.0 and wrong path made by the robot. The blue line indicates the planned path instead the purple dotted line indicates the path followed by the robot. With the use of only one sensor to obtain odometry, there is an accumulation of errors.

Figure 6.21: Map of CIM 4.0 and better path made by the robot. The blue line indicates the planned path instead the green dotted line indicates the path followed by the robot. With the use of sensor fusion to obtain odometry, a better result is obtained.

map during the SLAM and that appear suddenly, are easily avoided by the robot. When they are detected, the robot can re-plan the path so that it does not collide with them. Figure 6.23 shows how the robot can avoid the obstacle that appears

in front of the robot and how the movement takes place to avoid it. The same experiment is seen by RViz (Figure 6.24) highlighting the obstacle detection and the rescheduled path. The robot, exploiting the potential of the omniwheels and their translation and rotation movements, can avoid obstacles very easily with great precision, without having to do too many maneuvers.

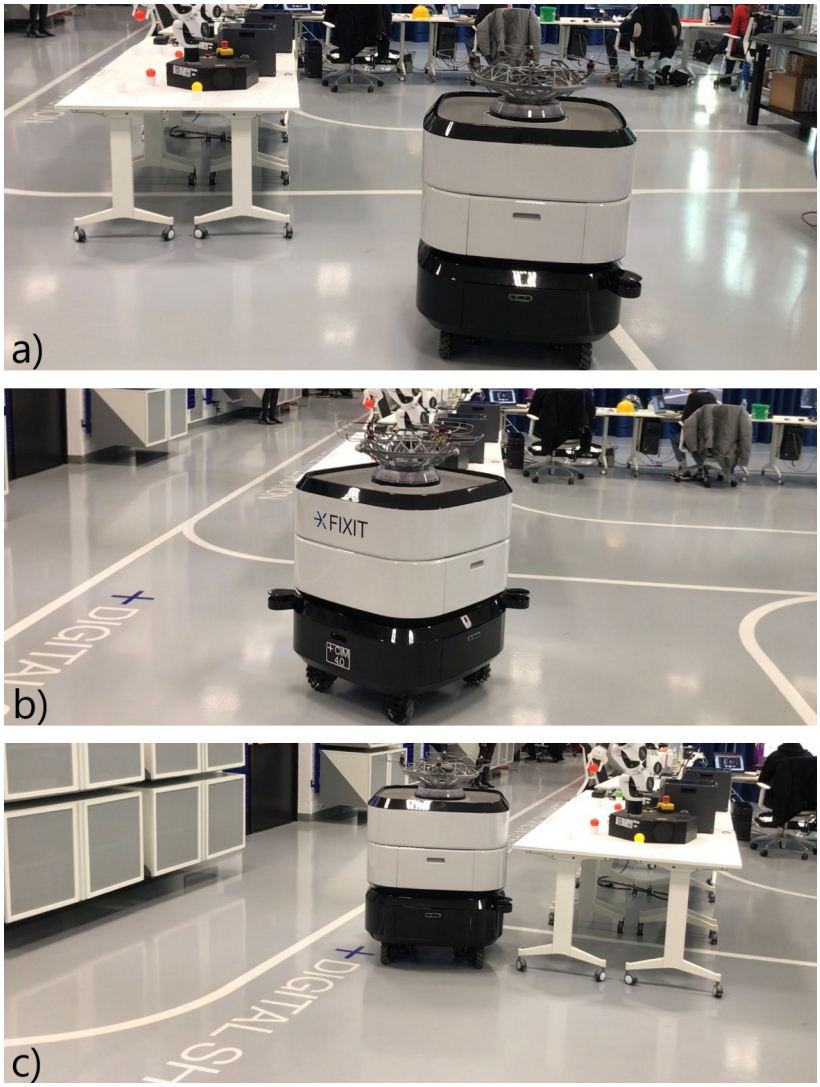

Figure 6.22: Obstacle avoidance experimental tests with static obstacles. In the first step, the robot starts to move (a), in the second one, it detects the obstacle represented by table (b), in the third one it avoids accurately the static obstacle (c).

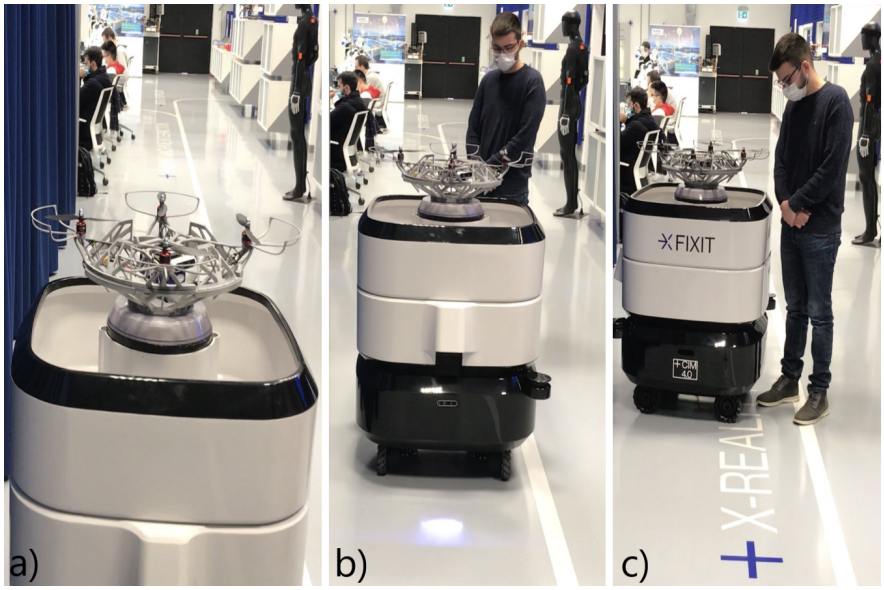

Figure 6.23: Obstacle avoidance experimental tests. In the first step, the robot starts to move (a), in the second one, it detects the obstacle (b), in the third one, it avoids the obstacle rotating around its (c).

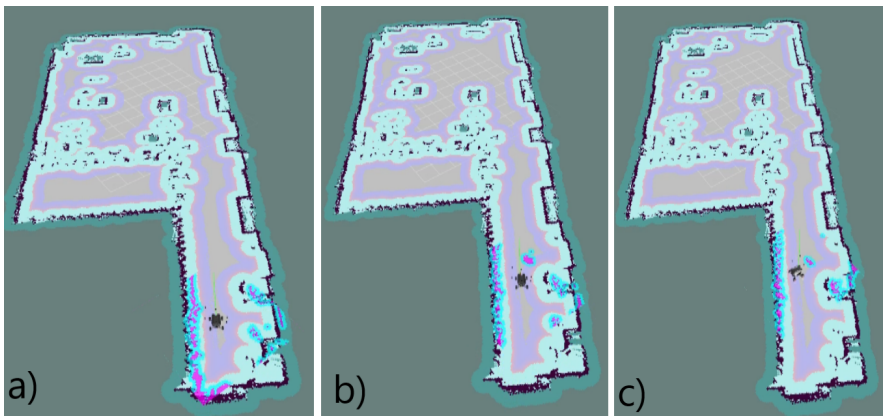

Figure 6.24: Obstacle avoidance in RViz. In the first step, the robot starts to move following the path planned (a), in the second one, it detects the obstacle (b), in the third one, it avoids the obstacles by re-planning the path (c).

Chapter 7

Conclusions

The developed system results to be very performing and robust. Sensors chosen for the application are suitable for robot movement in an indoor environment consisting of obstacles and objects of various shapes and sizes at each level.

The whole system is:

- Flexible: because it can be adapted to different situations. For example, in case of a change of the maintenance structure placed above the robot, the sensors used can be placed in different positions and the filter used for the LiDARs can be adapted to the type of shape in a very simple way. Furthermore, thanks to the use of ROS, the system can be modified quite easily to adapt it to the situation, allowing the modification of some features such as size, speed, or navigation.
- Safe: thanks to the entire sensor system used, the robot can see obstacles at any height and with an omnidirectional vision, guaranteeing the safety of objects and people in the environment in which the robot moves.
- Accurate: the robot can locate itself very accurately during navigation thanks to the sensor fusion algorithms used.

This system can be used in many cases:

- exploration and surveillance: thanks to the ability to move autonomously by acquiring information from the surrounding environment through sensors, it can be used to move in an environment that is dangerous for humans. Thus, the remote operator can see the environment in which he cannot access, for example, due to the dispersion of harmful gases, through the cameras. In addition, it can be used to check for the presence of certain objects and people in an environment.

- maintenance: the robot could be useful for carrying heavy objects and assisting the operator in maintenance operations. It can also be used to do repetitive tasks, such as repeatedly transporting something from one room to another.

7.0.1 Limits and future works

Although the developed autonomous navigation system provides good results, as desired, some aspects could be improved. A limit is given by the impossibility of seeing objects that sometimes could be very close to the robot or that are very small and therefore at floor level. For this reason, the robustness of the system could be improved by adding ultrasonic sensors to the robot wheels. In this way, thanks to this type of small and efficient sensors, it would be possible to navigate with greater safety.

Another limitation is that the robot with these types of sensors can move easily only in an indoor environment. The outdoors navigation is possible only if there is no sunlight because it could annoy the results obtained from the LiDARs. This problem can be overcome by changing the type of LiDARs used with another type suitable for outdoor navigation.

The system can be exploited to do more precise tasks. Useful actions could be implemented as future works: following a person or a moving object (like another AMR) and going to precise points following a recognized action by a person (such as after recognizing a certain gesture like a raised hand) or after recognizing a certain object (such as QR code or wearable items by operators).

Bibliography

[1] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Robotics: modelling, planning and control*. Springer Science & Business Media, 2010 (cit. on pp. 5–9, 15–18, 21).

[2] Abdelfetah Hentout, Mustapha Aouache, Abderraouf Maoudj, and Isma Akli. «Human–robot interaction in industrial collaborative robotics: a literature review of the decade 2008–2017». In: *Advanced Robotics* 33.15-16 (2019), pp. 764–799. DOI: 10.1080/01691864.2019.1636714. eprint: https://doi.org/10.1080/01691864.2019.1636714. URL: https://doi.org/10.1080/01691864.2019.1636714 (cit. on p. 5).

[3] Ioan Doroftei, Victor Grosu, and V. Spinu. «Omnidirectional Mobile Robot - Design and Implementation». In: Sept. 2007. ISBN: 978-3-902613-15-8. DOI: 10.5772/5518 (cit. on pp. 8, 10–12).

[4] Li Xie, Christian Scheifele, Weiliang Xu, and Karl A. Stol. «Heavy-duty omni-directional Mecanum-wheeled robot for autonomous navigation: System development and simulation realization». In: *2015 IEEE International Conference on Mechatronics (ICM)*. 2015, pp. 256–261. DOI: 10.1109/ICMECH.2015.7083984 (cit. on p. 13).

[5] Alessandro Manzi and Matteo Gabelletti. «Sistema di navigazione per robot mobili basato sull'anticipazione sensoriale». In: (2007) (cit. on p. 17).

[6] Lorenzo Sabattini, Alessio Levratti, Francesco Venturi, Enrica Amplo, Cesare Fantuzzi, and Cristian Secchi. «Experimental comparison of 3D vision sensors for mobile robot localization for industrial application: Stereo-camera and RGB-D sensor». In: *2012 12th International Conference on Control Automation Robotics Vision (ICARCV)*. 2012, pp. 823–828. DOI: 10.1109/ICARCV.2012.6485264 (cit. on p. 17).

[7] Zhengyou Zhang. «Microsoft Kinect Sensor and Its Effect». In: *IEEE MultiMedia* 19.2 (2012), pp. 4–10. DOI: 10.1109/MMUL.2012.24 (cit. on p. 18).

[8] Herald Weber. «LiDAR sensor functionality and variants». In: *SICK AG whitepaper* (2018). URL: https://cdn.sick.com/media/docs/3/63/963/Whitepaper_LiDAR_en_IM0079963.PDF (cit. on p. 19).

[9] Critchley Liam. «What are Encoder Sensors?» In: *AZoSensors* (2019). URL: https://www.azosensors.com/article.aspx?ArticleID=1730 (cit. on p. 20).

[10] Mary B Alatise and Gerhard P Hancke. «A review on challenges of autonomous mobile robot and sensor fusion methods». In: *IEEE Access* 8 (2020), pp. 39830–39846 (cit. on p. 21).

[11] Norhafizan Ahmad, Raja Ariffin Raja Ghazilla, Nazirah M Khairi, and Vijayabaskar Kasi. «Reviews on various inertial measurement unit (IMU) sensor applications». In: *International Journal of Signal Processing Systems* 1.2 (2013), pp. 256–262 (cit. on p. 21).

[12] Mary B. Alatise and Gerhard P. Hancke. «A Review on Challenges of Autonomous Mobile Robot and Sensor Fusion Methods». In: *IEEE Access* 8 (2020), pp. 39830–39846. DOI: 10.1109/ACCESS.2020.2975643 (cit. on p. 22).

[13] Qiang Li, Ranyang Li, Kaifan Ji, and Wei Dai. «Kalman Filter and Its Application». In: *2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS)*. 2015, pp. 74–77. DOI: 10.1109/ICINIS.2015.35 (cit. on pp. 22, 24, 25, 27).

[14] Greg Welch, Gary Bishop, et al. «An introduction to the Kalman filter». In: (1995) (cit. on p. 23).

[15] Sagar Gupta, Abhaya Pal Singh, Dipankar Deb, and Stepan Ozana. «Kalman Filter and Variants for Estimation in 2DOF Serial Flexible Link and Joint Using Fractional Order PID Controller». In: *Applied Sciences* 11.15 (2021). ISSN: 2076-3417. DOI: 10.3390/app11156693. URL: https://www.mdpi.com/2076-3417/11/15/6693 (cit. on p. 27).

[16] Thaddeus Abiy, Hannah Pang, Christopher Williams, Jimin Khim, and Eli Ross. «Dijkstra's Shortest Path Algorithm». In: *Brilliant.org* (2021). Available online, Retrieved October 14, 2021. URL: https://brilliant.org/wiki/dijkstras-short-path-finder/ (cit. on pp. 29–31).

[17] Masoud S. Nosrati, Ronak Karimi, and Hojat Allah Hasanvand. «Investigation of the * (Star) Search Algorithms: Characteristics, Methods and Approaches - TI Journals». In: *World Applied Programming* (2012) (cit. on pp. 29, 34, 36).

[18] Jean-Claude Latombe. «Potential Field Methods». In: *Robot Motion Planning*. Boston, MA: Springer US, 1991, pp. 295–355. ISBN: 978-1-4615-4022-9. DOI: 10.1007/978-1-4615-4022-9_7. URL: https://doi.org/10.1007/978-1-4615-4022-9_7 (cit. on p. 29).

[19] Saurabh Sarkar and Ernest Hall. «Virtual force field based obstacle avoidance and agent based intelligent mobile robot». In: vol. 6764. Sept. 2007. DOI: 10.1117/12.734691 (cit. on p. 29).

[20] J. Borenstein and Y. Koren. «The vector field histogram-fast obstacle avoidance for mobile robots». In: *IEEE Transactions on Robotics and Automation* 7.3 (1991), pp. 278–288. DOI: 10.1109/70.88137 (cit. on p. 30).

[21] R. Abiyev, D. Ibrahim, and B. Erin. «Navigation of mobile robots in the presence of obstacles». In: *Advances in Engineering Software* 41.10 (2010), pp. 1179–1186. ISSN: 0965-9978. DOI: https://doi.org/10.1016/j.advengsoft.2010.08.001. URL: https://www.sciencedirect.com/science/article/pii/S0965997810001018 (cit. on pp. 30, 38).

[22] Javier Minguez, Florent Lamiraux, and Jean-Paul Laumond. «Motion Planning and Obstacle Avoidance». In: *Springer Handbook of Robotics*. Ed. by Bruno Siciliano and Oussama Khatib. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 827–852. ISBN: 978-3-540-30301-5. DOI: 10.1007/978-3-540-30301-5_36. URL: https://doi.org/10.1007/978-3-540-30301-5_36 (cit. on pp. 30, 36–41).

[23] Edsger W. Dijkstra. «A note on two problems in connexion with graphs». In: *Numerische Mathematik* 1 (1959), pp. 269–271 (cit. on p. 30).

[24] Adeel Javaid. «Understanding Dijkstra Algorithm». In: *SSRN Electronic Journal* (Jan. 2013). DOI: 10.2139/ssrn.2340905 (cit. on pp. 31, 33).

[25] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. «A Formal Basis for the Heuristic Determination of Minimum Cost Paths». In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136 (cit. on p. 33).

[26] Anthony Stentz. «Optimal and efficient path planning for partially-known environments». In: *Proceedings of the 1994 IEEE International Conference on Robotics and Automation* (1994), 3310–3317 vol.4 (cit. on p. 35).

[27] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011 (cit. on p. 42).

[28] Pyo YoonSeak, Cho HanCheol, Jung RyuWoon, and Lim TaeHoon. *Ros Robot Programming*. ROBOTIS, Dec. 2017. ISBN: 9791196230715 (cit. on pp. 43–46).

[29] Jason M. O'Kane. *A Gentle Introduction to ROS*. Available at http://www.cse.sc.edu/~jokane/agitr/. Independently published, Oct. 2013. ISBN: 978-1492143239 (cit. on p. 43).

[30] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. «ROS: an open-source Robot Operating System». In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5 (cit. on p. 43).

[31] *Gazebo*. URL: http://gazebosim.org/ (cit. on p. 48).

[32] H. Durrant-Whyte and T. Bailey. «Simultaneous localization and mapping: part I». In: *IEEE Robotics Automation Magazine* 13.2 (2006), pp. 99–110. DOI: 10.1109/MRA.2006.1638022 (cit. on pp. 50, 51).

[33] T.J. Chong, X.J. Tang, C.H. Leng, M. Yogeswaran, O.E. Ng, and Y.Z. Chong. «Sensor Technologies and Simultaneous Localization and Mapping (SLAM)». In: *Procedia Computer Science* 76 (2015). 2015 IEEE International Symposium on Robotics and Intelligent Sensors (IEEE IRIS2015), pp. 174–179. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2015.12.336. URL: https://www.sciencedirect.com/science/article/pii/S1877050915038375 (cit. on p. 50).

[34] *SLAM (Simultaneous Localization and Mapping)*. URL: https://it.mathworks.com/discovery/slam.html (cit. on pp. 50, 52).

[35] Marcin Zukowski, Krzysztof Matus, Dawid Kamienski, Miroslaw Kondratiuk, Leszek Ambroziak, and Barbara Kuc. «SLAM and obstacle detection for tall autonomous robotic medical assistant». In: *Mechatronics Systems and Materials 2018*. Vol. 2029. American Institute of Physics Conference Series. Oct. 2018, p. 020085. DOI: 10.1063/1.5066547 (cit. on pp. 50, 52, 53).

[36] *Hector SLAM*. URL: http://wiki.ros.org/hector_slam (cit. on p. 52).

[37] ROS.org. *Gmapping Algorithm*. URL: http://wiki.ros.org/gmapping (cit. on p. 52).

[38] João Machado Santos, David Portugal, and Rui P. Rocha. «An evaluation of 2D SLAM techniques available in Robot Operating System». In: *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. 2013, pp. 1–6. DOI: 10.1109/SSRR.2013.6719348 (cit. on p. 52).

[39] ROS.org. *RGBD Algorithm*. URL: http://wiki.ros.org/rgbdslam (cit. on p. 53).

[40] *Tiny SLAM*. URL: http://wiki.ros.org/tiny_slam (cit. on p. 53).

[41] Arnaud Doucet, Nando de Freitas, Kevin P. Murphy, and Stuart J. Russell. «Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks». In: *CoRR* abs/1301.3853 (2013). arXiv: 1301.3853. URL: http://arxiv.org/abs/1301.3853 (cit. on p. 53).

[42] Kevin P. Murphy. «Bayesian Map Learning in Dynamic Environments». In: *In Neural Info. Proc. Systems (NIPS*. MIT Press, 2000, pp. 1015–1021 (cit. on p. 53).

[43] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. «Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters». In: *IEEE Transactions on Robotics* 23.1 (2007), pp. 34–46. DOI: 10.1109/TRO.2006.889486 (cit. on p. 53).

[44] Arnaud Doucet, N. Freitas, and N. J Gordon. *Sequential Monte-Carlo Methods in Practice*. Vol. 1. Jan. 2001. ISBN: 978-1-4419-2887-0. DOI: 10.1007/978-1-4757-3437-9 (cit. on p. 54).

[45] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. «Improving Grid-based SLAM with Rao-Blackwellized Particle Filters By Adaptive Proposals and Selective Resampling». In: Jan. 2005, pp. 2432–2437. DOI: 10.1109/ROBOT.2005.1570477 (cit. on pp. 55, 56).

[46] ROS.org. *LaserScan Message*. URL: http://docs.ros.org/en/api/sensor_msgs/html/msg/LaserScan.html (cit. on p. 65).

[47] ROS.org. *Navigation stack*. URL: http://wiki.ros.org/navigation/Tutorials/RobotSetup (cit. on pp. 66, 67).

[48] ROS.org. *tf ROS package*. URL: http://wiki.ros.org/tf (cit. on p. 66).

[49] Sebastian Madgwick. «An efficient orientation filter for inertial and inertial / magnetic sensor arrays». In: 2010 (cit. on p. 67).

[50] ROS.org. *Madgwick filter*. URL: http://wiki.ros.org/imu_filter_madgwick (cit. on p. 67).

[51] *Robot localization*. URL: http://docs.ros.org/en/melodic/api/robot_localization/html/index.html (cit. on p. 68).

[52] *Map server*. URL: http://wiki.ros.org/map_server (cit. on p. 69).

[53] Rodrigo Longhi Guimarães, André Schneider de Oliveira, João Alberto Fabro, Thiago Becker, and Vinıcius Amilgar Brenner. «ROS navigation: Concepts and tutorial». In: *Robot Operating System (ROS)*. Springer, 2016, pp. 121–160 (cit. on p. 69).

[54] ROS.org. *Inflation Parameters*. URL: http://wiki.ros.org/costmap_2d#Inflation (cit. on pp. 69, 70).

[55] David V. Lu, Dave Hershberger, and William D. Smart. «Layered costmaps for context-sensitive navigation». In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2014, pp. 709–715. DOI: 10.1109/IROS.2014.6942636 (cit. on p. 71).

[56] ROS.org. *move base*. URL: http://wiki.ros.org/move_base (cit. on p. 71).

[57] ROS.org. *Navfn*. URL: http://wiki.ros.org/navfn (cit. on p. 72).

[58] Pablo Marín, Ahmed Hussein, David Martín Gómez, and Arturo de la Escalera. «Global and Local Path Planning Study in a ROS-Based Research Platform for Autonomous Vehicles». In: *Journal of Advanced Transportation* 2018 (Feb. 2018), pp. 1–10. DOI: 10.1155/2018/6392697 (cit. on pp. 72, 73).

[59] ROS.org. *dwa local planner*. URL: http://wiki.ros.org/dwa_local_planner (cit. on p. 72).

[60] ROS.org. *amcl node*. URL: http://wiki.ros.org/amcl (cit. on p. 75).

[61] Frank Dellaert, Dieter Fox, Wolfram Burgard, and Sebastian Thrun. «Monte Carlo localization for mobile robots». In: vol. 2. Feb. 1999, 1322–1328 vol.2. ISBN: 0-7803-5180-0. DOI: 10.1109/ROBOT.1999.772544 (cit. on p. 76).

[62] Wolfram Burgard, Dieter Fox, and Sebastian Thrun. «Probabilistic robotics». In: *The MIT Press* (2005) (cit. on p. 76).

[63] Dieter Fox. «KLD-Sampling: Adaptive Particle Filters.» In: Jan. 2001, pp. 713–720 (cit. on p. 76).

[64] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. «You Only Look Once: Unified, Real-Time Object Detection». In: June 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91 (cit. on pp. 76, 77).

[65] *YOLO ROS: Real-Time Object Detection for ROS*. URL: https://github.com/leggedrobotics/darknet_ros (cit. on p. 77).

[66] *Darknet ROS 3D*. URL: https://github.com/IntelligentRoboticsLabs/gb_visual_detection_3d (cit. on p. 78).

[67] Li Xie, Christian Scheifele, Weiliang Xu, and Karl A. Stol. «Heavy-duty omni-directional Mecanum-wheeled robot for autonomous navigation: System development and simulation realization». In: *2015 IEEE International Conference on Mechatronics (ICM)* (2015), pp. 256–261 (cit. on p. 83).

[68] Ioan Doroftei, Victor Grosu, and Veaceslav Spinu. «Omnidirectional Mobile Robot - Design and Implementation». In: *Bioinspiration and Robotics*. Ed. by Maki K. Habib. Rijeka: IntechOpen, 2007. Chap. 29. DOI: 10.5772/5518. URL: https://doi.org/10.5772/5518 (cit. on p. 83).

[69] J.A Cooney, W.L Xu, and G Bright. «Visual dead-reckoning for motion control of a Mecanum-wheeled mobile robot». In: *Mechatronics* 14.6 (2004), pp. 623–637. ISSN: 0957-4158. DOI: https://doi.org/10.1016/j.mechatronics.2003.09.002. URL: https://www.sciencedirect.com/science/article/pii/S0957415803001156 (cit. on p. 83).

[70] Intel. *Intel RealSense Depth Camera D435i*. URL: https://www.intelrealsense.com/depth-camera-d435i/ (cit. on pp. 85, 87).

[71] SlAMTEC. *RPLIDAR A1*. URL: https://www.slamtec.com/en/Lidar/A1 (cit. on p. 88).

[72] NVIDIA. *JETSON XAVIER NX*. URL: https://www.nvidia.com/it-it/autonomous-machines/embedded-systems/jetson-xavier-nx/ (cit. on p. 90).

[73] NVIDIA. *Jetson Nano*. URL: https://developer.nvidia.com/embedded/jetson-nano-developer-kit (cit. on pp. 90, 91).

[74] AgileX. *Scout mini omnidirectional*. URL: https://www.agilex.ai/upload/files/38_2108.pdf (cit. on pp. 92, 93).

[75] agilexrobotics. *UGV SDK*. URL: https://github.com/agilexrobotics/ugv_sdk (cit. on p. 94).

[76] ROS.org. *Network Setup*. URL: http://wiki.ros.org/ROS/NetworkSetup (cit. on p. 96).