POLITECNICO DI TORINO

Master of Science Degree in MECHATRONIC ENGINEERING

Master Thesis

Obstacle Avoidance Algorithms for Autonomous Navigation system in Unstructured Indoor areas

Supervisor:
Prof. Marcello Chiaberge

Student:
Lorenzo Galtarossa

October 2018

Abstract

This work aims to implement different autonomous navigation algorithms for Obstacle Avoidance that allow a robot to move and perform in an unknown and unstructured indoor environment.

The first step is the investigation and study of the platform, divided into software and hardware, available at the Mechatronics Laboratory (Laboratorio Interdisciplinare di Meccatronica, LIM) at the Politecnico di Torino, on which it is implemented the navigation algorithm.

For what is concerned with the software platform, ROS has been used. The Robot Operating System is an open source framework to manage robots' operations, tasks, motions. As hardware platform the TurtleBot3 (Waffle and Burger) has been used that is ROS-compatible.

The second step is the inspection of the different algorithms that are suitable and relevant for our purpose, goal and environment. Many techniques could be used to implement the navigation that is generally divided into global motion planning and local motion control. Often autonomous mobile robots work in an environment for which prior maps are incomplete or inaccurate. They need the safe trajectory that avoids the collision.

The algorithms presented in this document are related to the local motion planning; therefore, the robot, using the sensor mounted on it, is capable to avoid the obstacles by moving toward the free area.

Three different algorithms of Obstacle Avoidance are presented in this work, that address a complete autonomous navigation in an unstructured indoor environment. The algorithms grow in complexity taking into consideration the evolution and the possible different situations in which the robot will have to move, and all are tested on the TurtleBot3 robot, where only LiDAR was used as sensor to identify obstacles.

The third algorithm, "Autonomous Navigation", can be considered the final work, the main advantage is the possibility to perform curved trajectory with an accurate choice of the selected path, combining the angular and the linear velocity (980 different motions), the LiDAR scans 180° in front of the robot to understand the correct direction. The last step is the automatic creation of the map.

This map will be analysed and compared with the one defined using the RViz software that is the official software used in ROS environment. The tool is suitable to visualize

the state of the robot and the performance of the algorithms, to debug faulty behaviours, and to record sensor data.

The improvement of this reactive Obstacle Avoidance method is to successfully drive robots in Indoor troublesome areas. As conclusion we will show experimental results on TurtleBot3 in order to validate this research and provide an argumentation about the advantages and limitations.

Contents

List of Figure

1 Introduction

1.1 Objective of the Thesis

In the past, investigation into the development of unmanned air, underwater and land vehicles has been fundamentally the domain of military related organizations. Nowadays, the technological context, availability of precise sensors, the spread of open source software and the increasing of computation power, has led the largest companies to take an interest on the concept of automation and robotization and as a result autonomous navigation has become also one of hottest topics in the research's field.

In this thesis, we study the problem of autonomous navigation through an environment that is initially unknown, with the objective of reaching the farthest point in which the robot can move avoiding the obstacles. Without prior knowledge of the map, a moving robot must recognise its surroundings through onboard sensors and make instantaneous decisions to react to obstacles as they come into view. This problem lies at the intersection of several areas of robotics, including motion planning, perception, and exploration.

Different techniques could be used to implement the navigation that is generally separated into global motion planning and local motion control. The algorithms introduced in this work are linked to the local motion planning; therefore, using the sensor mounted on it, the robot is capable of avoiding the obstacles by moving toward the free area.

This document explains three possible algorithm solution, based on Obstacle Avoidance, that address a complete autonomous navigation in an unstructured indoor environment.

The algorithms raise in complexity taking into consideration the evolution and the possible changed in which the robot will have to move, and all are tested on the TurtleBot3 robot (Waffle and Burger), where only LiDAR was used as sensor.

The implemented techniques necessitate the robot to select actions based on the construction of the environment that it has perceived. As we will observe in this thesis, standard motion planning techniques often limit performance to be conservative when deployed in unknown environments, where every unexplored region of the map may, in the worst case, pose a hazard.

To guarantee that the robot will not collide with potential obstacles, motion planners limit the robot's speed such that it could come to a stop, if need be, before the collisions.

The trajectory and the speed of the robot depend on many factors such as the type of floor, the limitations of the hardware, the size and the material of the wheels and the type of algorithm that manages the movement of the robot.

The map is built with two-dimensional Cartesian histogram grid based on the RViz software that is the official software used in ROS environment, which is updated continuously with range data sampled by onboard sensors.

In order to make this work more complete a different solution to the automatic creation of the map, has been proposed; this map will be analysed and compared with the one created by the ROS tool.

1.2 Organisation of the Thesis

The thesis is composed of six chapters, below we list the content of each of them to give the reader an overview of the work done.

Chapter 1 is introductory and outlines the motivations that stimulate researcher and get them interested in the navigation system. Afterwards, the principle objective of the thesis and a description of the its structure is given.

Chapter 2 provides an overview of the software platform ROS, Robot Operating System, explaining its characteristic and philosophy that highlight why it is used as common platform to manage robots' operations, tasks, motions.

Chapter 3 offers an outline of Robots, describing the operation of the related sensors that could be mounted. Particular emphasis is placed on the robot available at the LIM department, Turtlebot3 (Waffle and Burger), of which it is described the software and hardware platform.

Chapter 4 aims to introduce the literature survey of the various techniques used for mobile robot navigation. Navigation and obstacle avoidance are one of the fundamental problems in mobile robotics, here are described two type of control global path planning and local motion control.

Chapter 5 represents the main work of these thesis. It consists of three parts, in which in each sub-chapter is described an implemented algorithm that is gradually more

complex, to perform the obstacle avoidance, allowing the robot to move and perform a trajectory in an unknown and unstructured indoor environment.

The result and some real application of the algorithms are drawn in Chapter 6. Moreover, this chapter outline also the advantages/disadvantages and limitation of the algorithms. Finally, it proposes future approach and application as agricultural outdoor environment.

2 ROS Robot Operating System

2.1 Introduction

In the field of robotics, platforms are of increasing importance. A platform is divided into a software platform and hardware platform. A robot software platform contains tools that are used to build robot application programs such as low-level device control, SLAM (Simultaneous Localization and Mapping), navigation, manipulation, recognition of objects or humans, sensing and package management, debugging and development tools especially in the industry, within which they are nowadays mostly used. Robot hardware platforms not only study platforms such as mobile robots, drones, and humanoids, but also commercial products.

Hence, robot researchers from around the world are collaborating to discover a platform that is intuitive and open source. The most popular robot software platform is ROS, that means Robot Operating System.

ROS, the Robot Operating System, is an open source framework to manage robots' operations, tasks, motions, and other things. ROS is intended to serve as a software platform for those who build and use robots daily, but at the same time for people who are starting to use robots no long ago. This common platform allows newcomers to be increasingly inclined to read more and more and it is very easy to use.

This structure of the platform allows the use of the code and information shared by the other programmers, that implies that you do not have to write all the code in order to move the robots, for this reason, ROS has been remarkably successful.

The latter was one of the main reasons why ROS was used, furthermore, it represents the Operating System of the two TurtleBot3 (Burger, Waffle described in Chapter 2) that are the robots available at the LIM (Interdisciplinary Laboratory of Mechatronics), on which the autonomous navigation algorithms were written.

2.2 History of ROS

"In May 2007, ROS was started by borrowing the early opensource robotic software frameworks including switchyard, which is developed by Dr. Morgan Quigley by the Stanford Artificial Intelligence Laboratory in support of the Stanford AI Robot STAIR (Stanford AI Robot) project.

Dr. Morgan Quigley is one of the founders and software development manager of Open Robotics (formerly the Open Source Robotics Foundation, OSRF), which is responsible for the development and management of ROS.

Switchyard is a program created for the development of artificial intelligence robots used in the AI lab's projects at the time and it is the predecessor of ROS.

In addition, Dr. Brian Gerkey, the developer of the Player/Stage Project and 2D Stage simulator, later influence the growth of 3D simulator Gazebo, which was developed since 2000 and has had a major impact on ROS's networking program. He is the co-founder of Open Robotics.

In November 2007, U.S. robot company Willow Garage succeeded the development of ROS. Willow Garage is a well-known company in the field of personal robots and service robots." [2]

ROS is based on two licences (the BSD 3-Clause License and Apache License 2.0), which lets anyone modify, reuse and redistribute all the material available inside the platform.

This allows the development of robotic platforms able to apply ROS, some examples are the PR2 that stands for Personal Robot and TurtleBot, making ROS as the main software platform for robots.

Figure 2.1 Morgan Quigley programmed the first iteration of what grew into ROS

2.3 Meta-Operating System

ROS is an open-source, meta-operating system for the robot. It delivers the services you would imagine from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers, that has the target of simplifying the task of creating complex and robust robot behaviour across a wide variety of robotic platforms.

Contrasting conventional operating systems, it can be used for several combinations of hardware implementation. Furthermore, it is considered as a robot software platform that offers various development environments specialized for developing robot application programs.

For example, consider a simple "retrieve an object" activity, in which a robot is required to retrieve a specific object. First of all, the robot must understand the request, that means how to reach the goal. The robot must plan a sequence of actions to coordinate the object's search, which will require navigation through various rooms in a building, where the robot must be able to avoid all obstacles, optimizing the chosen path.

Once in a room, the robot must look for objects of similar size and find the required one. The robot must then return to its own steps and deliver the object to the desired position. Each of these subproblems can have an arbitrary number of issues; in the real world there are a lot of circumstance in each field that is difficult to predict and model, so no single individual can think to build a complete system from scratch.

So, ROS was built from the ground up to encourage collaborative robotics software development. In this example, a group might have specialists in indoor mapping and could contribute to a complex system for producing indoor maps; the same work could be done for an outdoor space (field or rows).

Another group may have experience in using maps to robustly navigate indoors, specialized in motion planning and Obstacle Avoidance. Another one may have discovered an approach to the vision that works with sensors able to offer capabilities such as gesture recognition, object recognition and scene recognition based on 3D depth information. ROS includes many features specifically designed to simplify this type of large-scale collaboration.

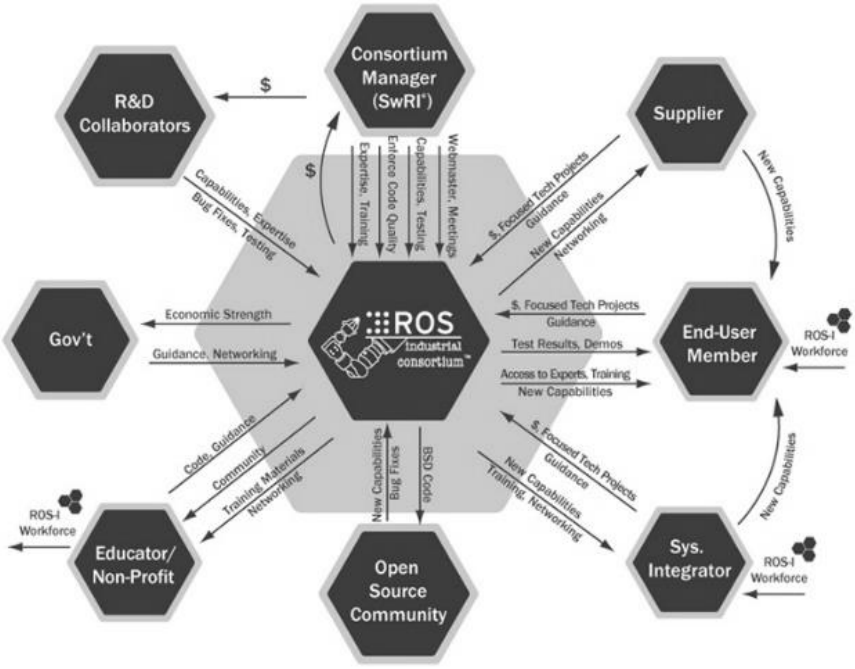


Figure 2.2 Structure of Ros

2.3.1 Characteristics of ROS

The main features of ROS can be grouped in five characteristics:

First is the reusability of the program. A user can focus on the goal related to its application that it would like to develop while downloading the corresponding package for the remaining functions. At the same time, he can share the program that he developed so that others can reuse it.

The second characteristic is that ROS is a communication-based program. Often, to provide a service, programs such as hardware drivers for sensors and actuators and features such as sensing, recognition and operating are developed in a single frame. However, to achieve the reusability of robot software, each program and feature is divided into smaller pieces based on its function. This is called componentization or modularization according to the platform.

The third is the support of development tools. ROS provides debugging tools, 2D visualization tool (such as Rqt) and 3D visualization tool (RViz) that can be used without developing the necessary tools for robot development. Tools that make it easy to visualize the state of the robot and the performance of the algorithms, to debug faulty

behaviours, and to record sensor data. A large and increasing gathering of robotics algorithms that allow you to map the environment, navigate around it, represent and interpret sensor data, plan motions, manipulate objects, and other operations is available.

For example, there are many occasions where a robot model needs to be visualized while developing a robot. The growing number of tools and their capabilities allows users not only to check the robot's model directly but also to perform a simulation using the provided 3D simulator (Gazebo).

The tool can also receive 3D distance information from cameras, as Intel RealSense or Microsoft Kinect, and easily convert them into the form of point cloud, finally display them on the visualization tool.

The fourth is the active community. Ros is a community for an open source software platform. There are over 5,000 packages that have been voluntarily developed and shared as of 2017, the Wiki pages that document many of the aspects of the framework, and a question-and-answer site where you can ask for help and share what you've learned.

The fifth is the construction of an ecosystem. Various software platforms have been developed and the most respected and used platform among them, ROS (for all the features that we already saw), is now shaping its ecosystem. It is creating an ecosystem for everyone: hardware developers from the robotic field such as a robot and sensor companies, ROS development operational team, application software developers, and users as the students, can be happy with it.

2.3.2 Philosophy of ROS

The following paragraphs describe some philosophical aspects of ROS:

Peer to peer: ROS systems consist of a small number of computer programs that are linked to one another and continuously exchange messages. These messages travel directly from one program to another. Although this makes the system more complex, the result is a system that balances better as the number of data increases.

Multilingual: ROS chose a multilingual approach. ROS software modules can be written in any language for which a client library has been written. At the time of

writing, client libraries exist for C++, Python, LISP, Java, JavaScript, MATLAB and others.

Thin: the ROS conventions encourage contributors to create standalone libraries and then wrap those libraries, so that they can send and receive messages to and from other ROS modules. This extra layer is proposed to allow the reuse of software outside of ROS for other applications, and it greatly simplifies the creation of automated tests using standard continuous integration tools.

Free and open source: the core of ROS is released under the permissive BSD license, which allows both commercial and non-commercial use. ROS foresees data exchange between modules using inter-process communication (IPC), which means that systems built using ROS can have fine-grained licensing of their various components.

As a user of ROS, I felt that the goal of ROS is to build an environment that allows robotic software development using a collaborative platform on a global level, where all the people share the code of their algorithm to help each other.

2.4 ROS Tools and Simulators

ROS has various tools that can be useful when the robot moves, or an algorithm is running, and we want to understand if it works properly or not. There are several ROS tools, including the ones that ROS users have personally released as well.

The tools we will describe, which represent the ones that have been most used during laboratory experiments/test are the following: RViz (3D visualization tool), Rqt (that is a software framework of ROS that implements the various GUI tools in the form of plugins), Rqt image-view (Image display tool), Rqt graph (tool that visualizes the correlation between nodes and messages as a graph), Rqt plot and Gazebo, a 3D simulator.

2.4.1 3D Visualization Tool (RViz)

RViz is the 3D visualization tool of ROS. The main purpose is to display ROS messages and topics in 3D, letting us to visually control data and the behaviours of our system.

There is the possibility to display also live representations of sensor values coming over ROS topics including camera data, infrared distance measurements, sonar data, and so on.

The mobile robot model can be shown and the received distance data from the Laser Distance Sensor (LDS) can be used for navigation to avoid obstacles. RViz can also display images from the camera mounted on the robot. In addition to this, it can take data from various sensors such as Kinect, LDS, RealSense and visualize them in 3D.

RViz has various functions such as interact, camera movement, selection, camera focus change, distance measurement, 2D position estimation, 2D navigation target-point, publish point.

The 3D View is in the middle of the screen (Figure 2.3), represented by a black area. It is the main screen which allows us to see various data in 3D, that can be configured in the Global Options and Grid settings on the left column of the screen.

The Displays panel on the left column is for selecting the data that we want to display from the various topics.

The ‘Fixed-Frame’ provides a static, base reference for your visualization. Any sensor data that comes into RViz will be transformed into that reference frame, so it can be properly displayed in the virtual world.

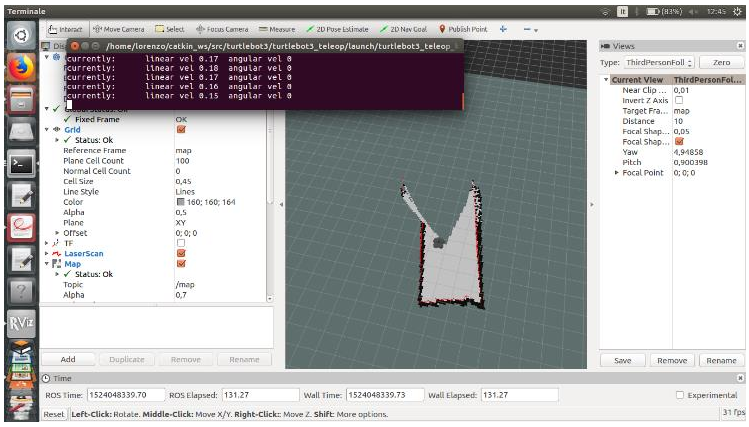


Figure 2.3 Illustration of the Waffle robot, while it is moved by 'teleop' command, within the Rviz tool

2.4.2 ROS GUI Development Tool (Rqt)

There are other tools apart from the 3D visualization tool RViz; ROS in fact supplies various GUI tools for robot development. There is a graphical tool that shows the hierarchy of each node as a diagram thereby showing the status of the current node and topic, and a plot tool that schematizes a message as a 2D graph, this kind of solution is useful to understand where the problem is when you are not able to visualize something or there is a device that does not work.

"Rqt image" view is a plugin to display the image data of a camera. Although it is not an image processing tool, it is still quite useful for simply checking an image. It is used to show what the robot sees while it is moving.

"Rqt_graph" is a tool that shows the correlation among active nodes and messages being transmitted on the ROS network as a diagram. This is very useful for understanding the current structure of the ROS network when the number of sensors, actuators, and programs is high.

Rqt plot is a tool for plotting 2D data. The plot tool receives ROS messages and displays them on 2D coordinates. As an example, let us plot the x and y coordinates of the 'turtlesim' node pose message. It is possible to see that the x, y position, direction in theta, translational speed, and the rotational speed of the turtle are plotted. As we can see, this is a useful tool for displaying the coordinates coming from 2D data.

In this example, we run the "turtlesim_node" and the "turtle_teleop_key" commands. The first command opens a blue window in which in the middle there is a turtle (that changes in shape every time), instead, by running the second command in a new

window, we will see messages and instructions that give the possibility of moving the turtle using the arrow keys on the keyboard (←, →, ↑, ↓) or, in another version, pressing letters (a, s, w, z). The turtle will move according to the arrow key as shown on the picture below.

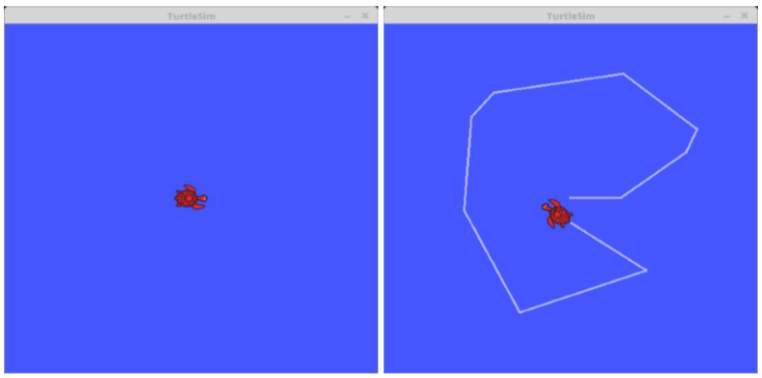


Figure 2.4 Window where the turtle is moved using the keyboard

The other Figure 2.5 shows the behavior of Rqt_graph, where the circles represent nodes (/teleop_turtle, /turtlesim) and squares (/turtle1/cmd_ vel) represent topic messages and the arrow indicates the transmission of the message.

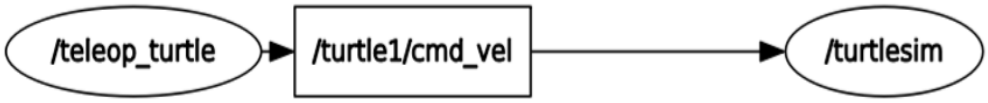


Figure 2.5 Rqt Graph representation of this example

When we executed 'turtle_teleop_key' and 'turtlesim_node', both the nodes were running respectively, and these two nodes are transmitting data with the arrow key values of the keyboard in the form of translational speed and rotational speed message.

2.4.3 Gazebo Simulator

Real robots need logistics including laboratory space, refreshing of batteries and operational quirks that often-become part of the institutional knowledge of the organization operating the robot. In a real case of work, even the best robots break periodically due to various combinations of operator errors, environmental conditions, manufacturing or design defects. These problems can be avoided by using simulated robots that move in a simulated environment.

Software robots are extraordinarily useful, in simulation we can model as much or as little reality as we desire. Sensors and actuators can be modelled as ideal devices, or they can incorporate various levels of distortion, errors and unpredicted faults. The simulated robots and environment represent the ultimate low-cost platforms.

The two-dimensional simultaneous localization and mapping (SLAM) problem was one of the greatest researched topics in the robotics community. Several 2D simulators were developed in response to the necessity for repeatable experiments as 'Stage'. Canonical laser range-finders and differential-drive robots were modelled, often using simple kinematic models. These 2D simulators are very fast computationally and they are generally quite simple to interact with.

Gazebo is a 3D simulator that provides robots, sensors, environment models for 3D simulation required for robot development, and offers realistic simulation with its physics engine. Gazebo is one of the most popular simulators for open source robotics in recent years and has been widely used in the field of robotics because of its high performance and reliability.

Gazebo uses OGRE (Open-source Graphics Rendering Engines) for the 3D Graphics, which is often used in games, not only for the robot model but also for the light, that can be realistically drawn on the screen.

A lot of sensors are already supported Laser range finder (LRF), 2D/3D camera, depth camera, a contact sensor, force-torque sensor; noise can be considered as added to the sensor data like in real environment.

Some robot models are already available in gazebo: PR2, Pioneer2 DX, iRobot Create, and TurtleBot are already supported in the form of SDF, a Gazebo model file, and users can add their own robots with an SDF file.

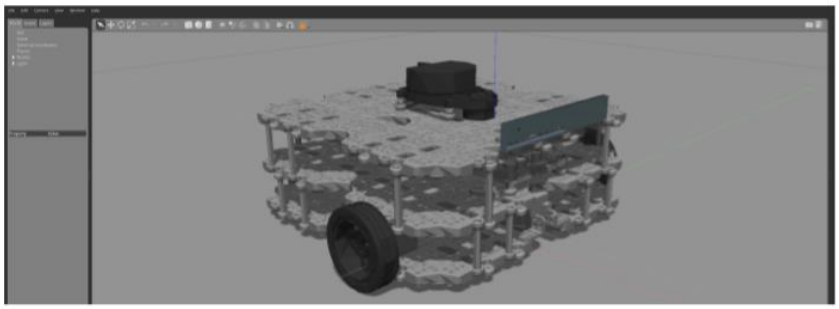


Figure 2.6 3D view of TurtleBot3 Waffle on Gazebo

Both GUI and CUI tools are supported to verify and control the simulation status. The latest version of Gazebo is 8.0, and just five years ago, it was 1.9.

Although Stage and other 2D simulators are computationally efficient and excel at simulating planar navigation in office-like environments, it is important to note that planar navigation is only one aspect of robotics. Nonplanar motion, ranging from outdoor ground vehicles to underwater and space robotics is another aspect too. Three-dimensional simulation is necessary for software development in these environments.

Robot motions can be separated into mobility and manipulation. The mobility aspects can be handled by two-or-three dimensional simulators in which the environment around the robot is static. Simulating manipulation, however, requires a significant rise in the complexity of the simulator to handle the dynamics of not just the robot, but also the dynamic models in the scene (simulators often use rigid-body dynamics, where objects are assumed to be incompressible).

ROS integrates closely with Gazebo through the Gazebo_ros package. This package provides a Gazebo plugin module that allows bidirectional communication between Gazebo and ROS. Simulated sensors and physical data can stream from Gazebo to ROS, and actuator commands can stream from ROS back to Gazebo; in this way, it is possible for Gazebo to exactly match the ROS API of a robot. When this is achieved,

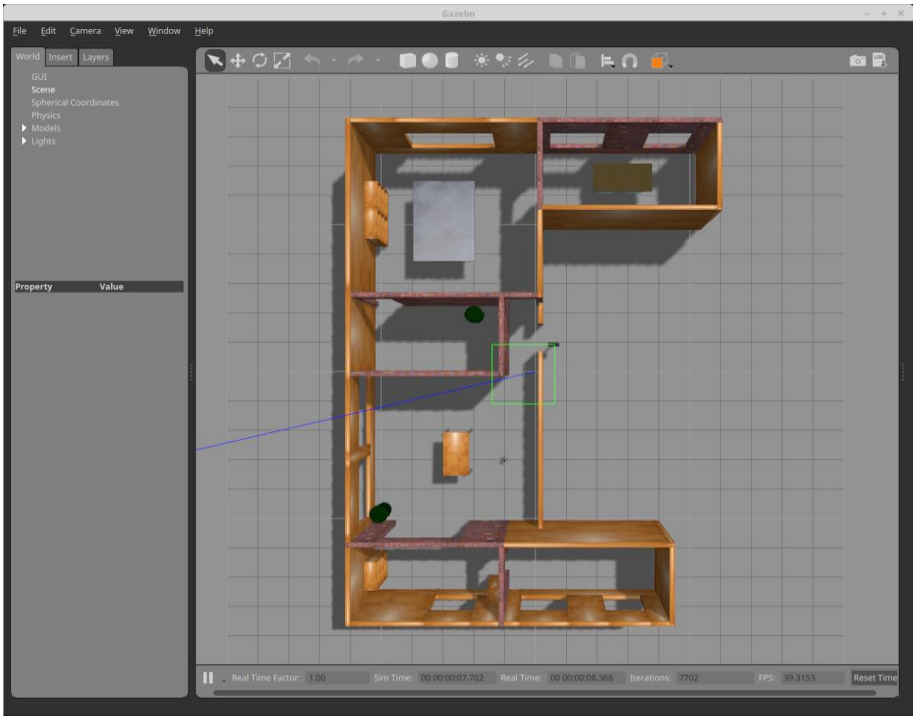


Figure 2.7 The house model of Gazebo

the robot software above the device-driver level can be represented identically both on the real robot and on the simulator.

In the above example, only the robot is loaded in the Gazebo. To perform the actual simulation, the user can specify the environment or load the environment model provided by Gazebo (as Empty-room, World, House models).

3 Robot

A robot is basically classified into hardware and software. All that is concern with a mechanism, motors, gears, circuits, sensors are considered as hardware. Micro-controller firmware that drives or controls the robot's hardware, and application software that builds the map, navigates, creates motion and perceives environment based on sensor data are classified as software. ROS can be classified as application software and depending on the specialized requests it is classified as a robot package, sensor package and motor package.

The main of robot packages are PR2 and TurtleBot; PR2 is a mobile-based humanoid robot, that has high performance and is general purpose, however, its price was not cheap enough to highlight ROS in the market, so TurtleBot was industrialized to increase the market of ROS.

Figure 3.1 PR2 (Left), TurtleBot2 (2nd from the left), TurtleBot3 (3 models on the right)

The first version is TurtleBot, follow by TurtleBot2 where KOBUKI was adopted as a mobile platform. The last one is TurtleBot3, a Dynamixel based mobile robot, developed in partnership with ROBOTIS and Open Robotics.

The following are the different types of robots that are used in almost every field: Manipulator, Mobile robot, Autonomous car, Humanoid, UAV (Unmanned Aerial Vehicle) and UUV (Unmanned Underwater Vehicle).

3.1 Sensor

Sensors play crucial roles in a robot. There are many way and sensor that can be operated to extract meaningful information from various environments and to recognize the surrounding objects using this information and transmit it to the robot. Every information that the robot can capture is used as data to perform an action, to make a plan or as input to perform some operation.

There are various types of sensors for getting such information, the most used for its effectiveness and simplicity is the distance sensor. Laser-based distance sensors such as LDS (Laser Distance Sensor), LiDAR (Light Detection and Ranging) or LRF (Laser Range Finders) and infrared based sensors such as RealSense, Kinect and Xtion are widely used as distance sensors. In addition, there are various sensors depending on information to acquire such as colour cameras used for object recognition, inertial sensors used for position estimation, microphones used for voice recognition and torque sensors used for torque control.

Depending on the kind of sensor used and, on its goal, it sends a different amount of data with a specified frequency, but each microprocessor has a limit of information that it can receive every time. 1D and 2D sensor, as Laser-based distance sensors, do not transmit heavy data, the problem is more related with the cameras which transmit a lot of data and require high processing power, so it is not easy for a microprocessor.

There are several sensor packages offered by the sensor available on ROS. Sensors are classified into 1D rangefinders (Infrared distance sensors for low-cost robots), 2D range finders (LDS is frequently used in navigation as in the algorithm presented in chapter 4), 3D Sensors (such as Intel's RealSense, Microsoft's Kinect are needed for 3D measurements), Pose Estimation (GPS + IMU), Cameras (that are commonly utilized for object and gesture recognition, face recognition and 3D SLAM), Audio/Speech Recognition and many other sensors.

3.1.1 Camera

The camera can be represented as the eyes of the robot and the images taken from the camera are useful for recognizing the environment around the robot. For example, object recognition using a camera image, facial recognition, a distance value obtained from the difference between two different images using two cameras (stereo camera), mono camera visual SLAM, colour recognition using information obtained from an image and object tracking are very useful.

3.1.2 Depth Camera

The Time of Flight (ToF) is one method with which works the depth camera, radiating Infrared Eays (IR) and measuring the distance by the time it takes to go back to the sensor. The IR transmission unit and the setting unit are a pair, and the distance measured by each pixel is read. This method represents the most expensive one due to the sophisticated hardware needed.

Microsoft's Kinect and ASUS's Xtion are based on the Structured Light technique, which applies a coherent radiation pattern. It is applied for the Depth Camera, these cameras consisting of one infrared projector and one infrared camera, which uses a coherent radiation pattern that was not present in previous ToF method. This technology is cheaper than the ToF one, therefore, they are more used on the low-cost robot.

A stereo camera, which is considered a Depth Camera, is the last method. Their idea is based on the operation mode on which work the left and right eyes of the people. The stereo camera is equipped with two image sensors for capture the image, where their distance has a specific role, it calculates the grid value using the difference between the two images, its distance is designed using binocular parallax. The stereo camera to calculate the distance applies the triangulation method, where an infrared projector emanates IR with a coherent pattern (called active stereo camera), instead, two infrared image sensors (called passive stereo camera) have the goal to interpret and create an image by the receiving infrared rays.

Intel® RealSense™ Camera R200 is one of the representative active stereo cameras. It is a long-range peripheral 3D camera, small and chapter, ideal for sensing the environment. It is widely used in robotics, drones, and other smart devices.

This represents the camera mounted on the TurtleBot3 Waffle, with Full HD colour and IR depth sensing features, the camera supports a wide variety of exciting new usage applications as the manipulation of colour and depth from multiple angles and perspectives, object recognition and 3D scanning. It is the cheapest among the Depth cameras so far (around $100).

3.1.3 Laser Distance Sensor

Laser Distance Sensors (LDS) includes a different kind of sensors such as Light Detection and Ranging (LiDAR), Laser Range Finder (LRF) and Laser Scanner. LDS is a sensor used to find the distance to an object using a laser as its source. The LDS guarantee high performance, high speed, real-time data acquisition, so it regularly adopted in a varied range of robotics field and in all the system where the measurement of a distance is required. In robotic it is one of the main sensors utilized for recognition of the distance of objects and people.

The LDS computes the difference of the wavelength when the laser source is reflected by the obstacle if it is found. The great problems, that are possible to encounter with this kind of sensor, are control issues that cannot be corrected by another sensor since, for its cheap price, only one LDS is commonly mounted on one device. A typical LDS consists of a single laser source, a motor, and a reflective mirror. The motor rotates the inner mirror while it is scanning using the laser. The range of the LDS goes from 180° to 360°.

The first image from the left of the Figure 3.2 shows how the mirror, that is tilted at a specific angle, reflects the laser on the surrounding environment. In the second and third image, while the motor rotates the mirror scanning all the environment, the sensor captures the laser that is returned and saved the return time (calculates the difference in wavelength). The LDS sensor scans objects in a horizontal plane, where closer objects are better identified, so the accuracy decreases as the distance becomes longer.

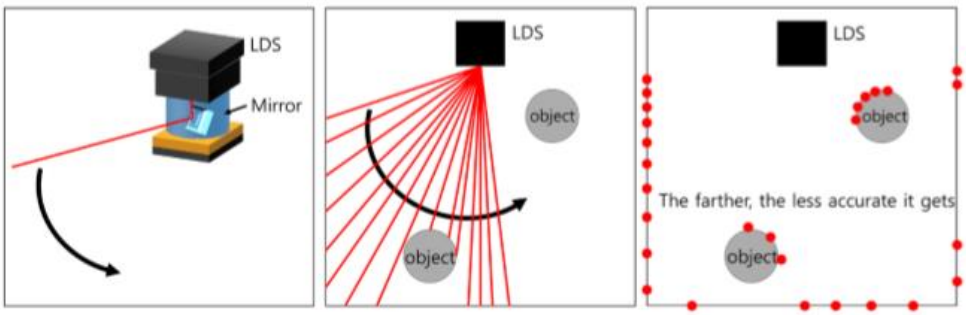


Figure 3.2 Distance measurement using LDS

However, there are some disadvantages with LDS. First, since it measures the difference in wavelength between the two waveforms (roundtrip), the objects must

properly reflect the laser source. A lot of obstacles, as plastic bottle or objects, transparent glass and mirror are inclined to reflect or scatter the laser in a different direction, changing the origin wavelength and producing an inaccurate and wrong measurement.

The second problem can be view more as a limit of this sensor, considering that it acquires 2D data, it scans only objects on the horizontal plane. The last one is related to the risk of possible damage on the eyes since the LDS uses a laser as the source, that are classified from class 1 to 4 (higher the number, higher the damage).

SLAM (Simultaneous Localization and Mapping) is one of the greatest applications of LDS. SLAM creates a map by recognizing obstacles around the robot and estimating the current position of the robot, as we will see successfully.

3.1.4 Motor Packages

The Motors page was included into ROS Wiki; it is a collection of packages of motors and servo controllers supported by the ROS. Dynamixel (DXL) is a series of high-performance networked actuators designed for robots, that has developed by ROBOTICS (a Korean manufacturer).

The Dynamixel is constituted of a reduction gear, a controller, a motor and a communication circuit. There are different versions of Dynamixel, whose feedbacks for position, speed, temperature, load, voltage and current data are enabled, thanks to a simple wire connection between devices. Dynamixel is usually applied in robotics since their offer several suitable purposes such as position, speed and torque control.

Figure 3.3 The different version of Dynamixels

3.2 Embedded System

An embedded system can be defined as a special-purpose computer embedded in a device that necessitates being controlled.

"An embedded system is an electronic system that exists within a device as a computer system that performs specific functions for control of a machine or other system that requires control. In other words, an embedded system can be defined as a specific purpose computer system that is a part of the whole device and serves as a brain for systems that need to be controlled." [22]

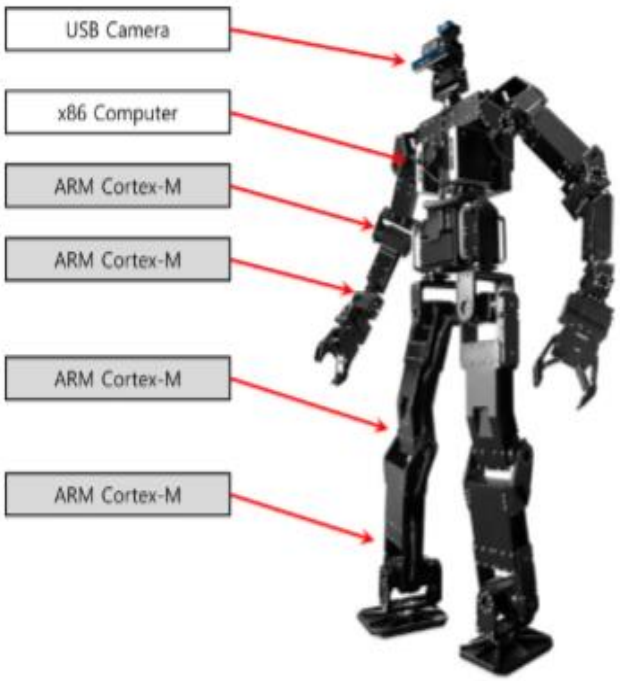


Figure 3.4 Embedded system configuration of a Humanoid robot

Many embedded devices are needed to implement the functions of robots. A microcontroller capable of real-time control is required to use an actuator or sensor of a robot, and the high-performance processor-based computer is mandatory for image processing using a camera or navigation, manipulation.

In robotics a microcontroller capable of real-time control for the sensor and the actuators is necessary, in the TurtleBot3 (Waffle, Waffle Pi and Burger) an ARM Cortex-M7 series microcontroller is used to manage the actuator and sensor, instead, to run the algorithm and to perform calculations the Raspberry Pi 3 board for Burger

and Waffle Pi, and the Intel Joule™ for Waffle are mounted on the TurtleBot3 and they are connected via USB to the other microcontroller.

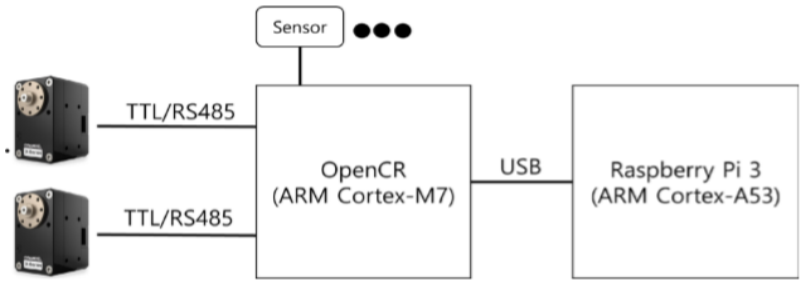


Figure 3.5 TurtleBot3 embedded system

3.2.1 OpenCR

The embedded board, that manages the operation of the TurtleBot3, is OpenCR (Open-source Control Module) which is ROS compatible. OpenCR supports STM32F7466 as MCU, it is a Hardware used to elaborate high quantity of data and to manage floating-point calculation. This microcontroller is necessary to guarantee high performance (run up to 216 MHz).

It has available different peripherals in order to supervise various kind of devices; for example, it yields the interface with Arduino, and it provides the communications for Dynamixel of the Robot (TTL and RS485) or sensors such as Camera (Raspberry Pi) and LiDAR.

OpenCR includes MPU925010 chip, which is fixed at the middle of the OpenCR board. It integrates triple-axis gyroscope, triple-axis accelerometer, and triple-axis magnetometer sensor in one chip, therefore, can be utilized for the different purpose. This kind of device is necessary to build a map of the environment or to understand what path is done by the robot.

The communication offered by the IMU (around 50Hz) is faster respect, for example, to the one of the LiDAR (5 Hz), this is due to the I2C or SPI communication. The OpenCR manages the input power source from 7V to 24V and is able to provide various levels of output 12V (1A), 5V (4A) and 3.3V (800mA).

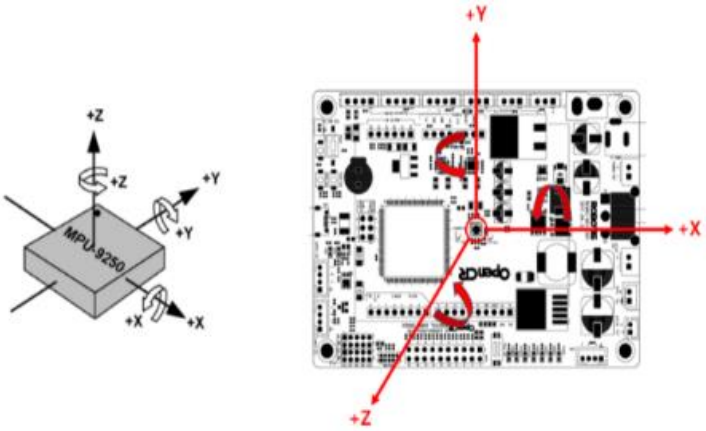


Figure 3.6 Gyroscope and Accelerometer directions

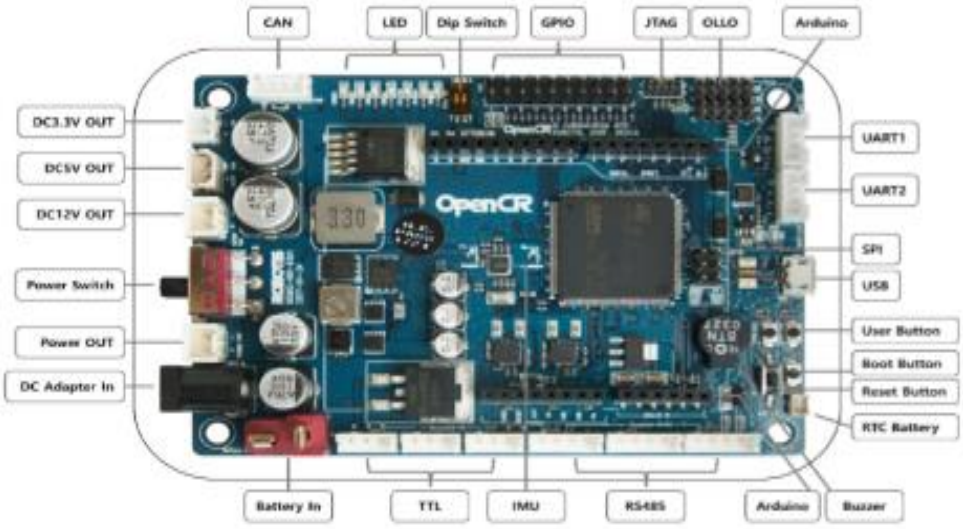


Figure 3.7 OpenCR interface configuration

3.3 TurtleBot

The ROS is one of the most utilized operating systems for the robot, more than 200 robots are built over ROS. The most famous ones are the PR2 and the TurtleBot created by Willow Garage in collaborations with Open Robotics.

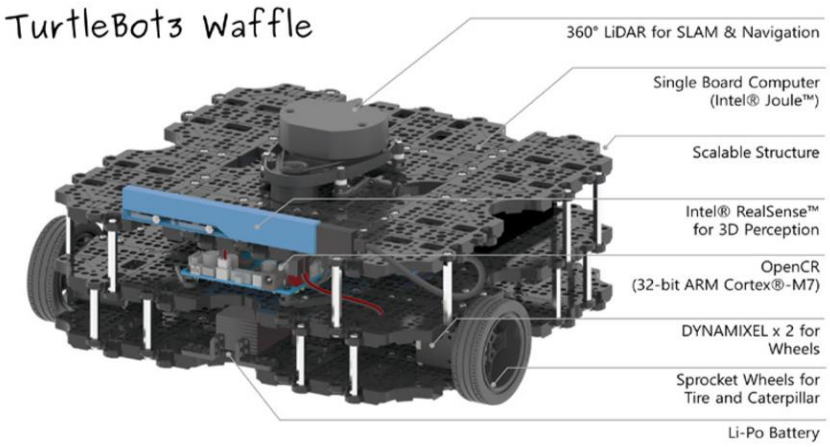
The TurtleBot logo, as the name inspires, is a turtle. TurtleBot is a standard platform of ROS and represent the one most popular robot based on it, used both by researcher and students because it is easy to learn, understand and manage even if you are not familiar with ROS.

The last version is the TurtleBot3, that includes all the functionalities of the previous versions and try to improve some characteristics like the presence of the Dynamixel as actuators.

There are three kinds of robots available with this new version (TurtleBot3): Waffle, Burger and Waffle Pi. All of them are ROS-based, designed for used in research, instruction and test. They are quite small, easy to programs and their price is relatively cheap (around $600 Burger, $1400 Waffle).

The TurtleBot3 can be modelled and modified in order to produce different configurations, there is the possibility of adding/removing sensors depending on what is the goal, which are the sensors available; or to reconstruct the mechanical parts and use optional parts such as the computer for increasing the calculations.

TurtleBot3 Burger and Waffle are the two robots that have been available at the LIM (Interdisciplinary Laboratory of Mechatronics), on which the in-door navigation algorithms, Obstacle Avoidance and mapping algorithms, have been run, as described in chapter 4.



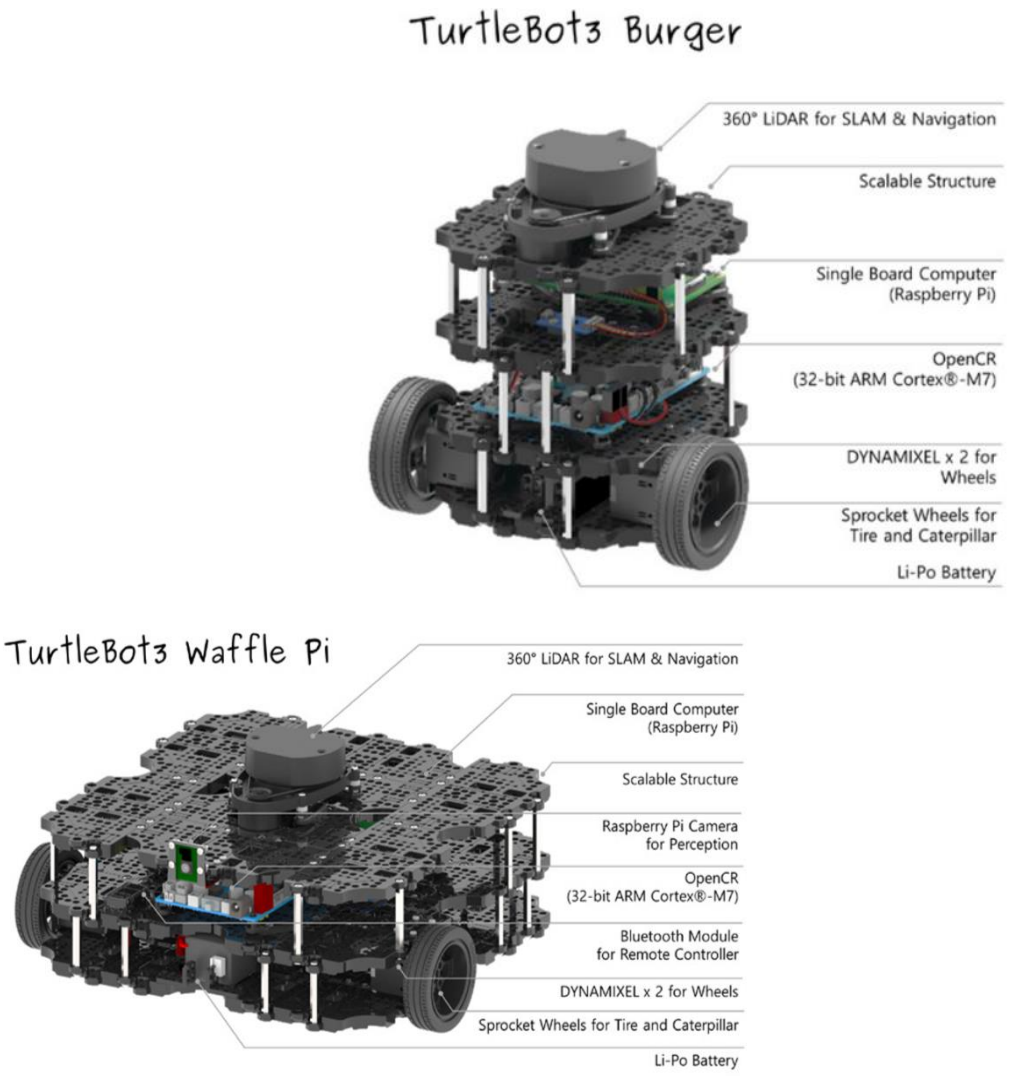


Figure 3.8 Hardware configuration of TurtleBot3

3.3.1 TurtleBot3 Hardware

As we already say there are three official TurtleBot3 models Burger, Waffle and Waffle Pi. The basic components of TurtleBot3 are actuators, an SBC for operating ROS, a sensor or more than one for SLAM and navigation, restructure mechanism, an OpenCR embedded board used as the main controller, sprocket wheels that can be used with tire and caterpillar, and three cell lithium-poly battery.

TurtleBot3 Waffle is different from Burger in terms of a platform shape, which being bigger can conveniently mount many components and sensors, use of higher torque actuators that guarantee a maximum linear speed of 0.26 m/s and an angular velocity

of 1.8 rad/s, high-performance SBC with Intel processor in terms of calculations, RealSense Depth Camera for object recognition and 3D SLAM.

Due to its characteristics, Waffle was the most used Robot during laboratory tests, also because from manual, it should have offered the possibility to use the Intel® Real Sense™ R200 camera. The Intel® Real Sense™ R200 camera should have provided depth and infrared video streams and the possibility to apply it for various applications such as gesture recognition, object recognition and scene recognition based on 3D depth information.

However, we discover that the Hub mounted on the Waffle is not able to support the Intel® RealSense™ R200 camera (probably the problem was due to the Hub that did not guarantee the 5 A of current required by the camera).

To better highlight, the malfunction of the camera, just think that the signal related to the image of the camera was transmitted at 0.2-0.5 Hz of frequency when it was connected to the TurtleBot and on it ran a navigation algorithm that involved the engines and the LIDAR. While the frequency guaranteed by specifications was 30 Hz.

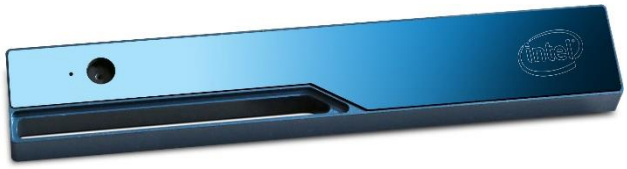


Figure 3.9 Intel Real Sense R200

Furthermore, the Intel® RealSense™ R200 camera cannot be mounted on the Burger since the camera was a USB 3.0 device and this robot could not offer this kind of port.
So, to capture the video, another camera has been mounted. The Raspberry Pi Camera Module V2 on the Burger has been used as an onboard camera during the SLAM of a real environment.

However, the Raspberry Pi Camera Module V2 couldn't provide depth and infrared video streams and was not possible to apply it for various applications: such as gesture recognition, object recognition and scene recognition based on 3D depth information (feasible using RealSense™).

TurtleBot3 Waffle Pi has the same shape as the Waffle model, but this model supports the Raspberry Pi as the Burger model, and the Raspberry Pi Camera to make it more affordable, however, is not present on the LIM.

Items	Burger	Waffle (Discontinued)	Waffle Pi
Maximum translational velocity	0.22 m/s	0.26 m/s	0.26 m/s
Maximum rotational velocity	2.84 rad/s (162.72 deg/s)	1.82 rad/s (104.27 deg/s)	1.82 rad/s (104.27 deg/s)
Maximum payload	15kg	30kg	30kg
Size (L x W x H)	138mm x 178mm x 192mm	281mm x 306mm x 141mm	281mm x 306mm x 141mm
Weight (+ SBC + Battery + Sensors)	1kg	1.8kg	1.8kg
Threshold of climbing	10 mm or lower	10 mm or lower	10 mm or lower
Expected operating time	2h 30m	2h	2h
Expected charging time	2h 30m	2h 30m	2h 30m
SBC (Single Board Computers)	Raspberry Pi 3 Model B	Intel® Joule™ 570x	Raspberry Pi 3 Model B
MCU	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)
Remote Controller	-	-	RC-100B + BT-410 Set (Bluetooth 4, BLE)
Actuator	Dynamixel XL430-W250	Dynamixel XM430-W210	Dynamixel XM430-W210
LDS(Laser Distance Sensor)	360 Laser Distance Sensor LDS-01	360 Laser Distance Sensor LDS-01	360 Laser Distance Sensor LDS-01
Camera	-	Intel® Realsense™ R200	Raspberry Pi Camera Module v2.1
IMU	Gyroscope 3 Axis Accelerometer 3 Axis Magnetometer 3 Axis	Gyroscope 3 Axis Accelerometer 3 Axis Magnetometer 3 Axis	Gyroscope 3 Axis Accelerometer 3 Axis Magnetometer 3 Axis
Power connectors	3.3V / 800mA 5V / 4A 12V / 1A	3.3V / 800mA 5V / 4A 12V / 1A	3.3V / 800mA 5V / 4A 12V / 1A
Expansion pins	GPIO 18 pins Arduino 32 pin	GPIO 18 pins Arduino 32 pin	GPIO 18 pins Arduino 32 pin
Peripheral	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4
Dynamixel ports	RS485 x 3, TTL x 3	RS485 x 3, TTL x 3	RS485 x 3, TTL x 3
Audio	Several programmable beep sequences	Several programmable beep sequences	Several programmable beep sequences
Programmable LEDs	User LED x 4	User LED x 4	User LED x 4
Status LEDs	Board status LED x 1 Arduino LED x 1 Power LED x 1	Board status LED x 1 Arduino LED x 1 Power LED x 1	Board status LED x 1 Arduino LED x 1 Power LED x 1
Buttons and Switches	Push buttons x 2, Reset button x 1, Dip switch x 2	Push buttons x 2, Reset button x 1, Dip switch x 2	Push buttons x 2, Reset button x 1, Dip switch x 2
Battery	Lithium polymer 11.1V 1800mAh / 19.98Wh 5C	Lithium polymer 11.1V 1800mAh / 19.98Wh 5C	Lithium polymer 11.1V 1800mAh / 19.98Wh 5C
PC connection	USB	USB	USB
Firmware upgrade	via USB / via JTAG	via USB / via JTAG	via USB / via JTAG
Power adapter (SMPS)	Input : 100-240V, AC 50/60Hz, 1.5A @max Output : 12V DC, 5A	Input : 100-240V, AC 50/60Hz, 1.5A @max Output : 12V DC, 5A	Input : 100-240V, AC 50/60Hz, 1.5A @max Output : 12V DC, 5A

Figure 3.10 Hardware specification of TurtleBot3

3.3.2 TurtleBot3 Software

The TurtleBot3 software is managed by a firmware (FW) of OpenCR board used like a sub-controller. The firmware of TurtleBot3 is considered the head of the robot, applying OpenCR like a sub-controller, because it is used for example to estimate the location of the robot while it is moving, calculating the encoder value produce by the Dynamixel (the driving motor of the robot), that are then accurate with the inertia

values to reduce the errors due to slip effect; or to control the velocity according to the command published by the software.

The acceleration and angular velocity are obtained from 3-axis acceleration and 3-axis gyroscope sensor mounted on OpenCR to control the direction of the robot, and it is also possible to evaluate and display using the right topic the battery state of the robot.

"TurtleBot3's ROS package includes 4 packages which are 'TurtleBot3', 'TurtleBot3_msgs', 'TurtleBot3_simulations', and 'TurtleBot3_applications'. The 'TurtleBot3' package contains TurtleBot3's robot model, SLAM and navigation package, remote control package, and bring up package. The 'TurtleBot3_msgs' package contains message files used in TurtleBot3, 'TurtleBot3_ simulations' contains packages related to simulation, and 'TurtleBot3_applications' package contains applications." [1]

The scene on which the robot moves can be an indoor or outdoor space that could be loaded in a simulative tool or in a real-environment.

The development environment of TurtleBot3 can be divided into Remote PC that performs remote control, SLAM, Navigation package, and TurtleBot PC that controls the robot components and motion, and collects sensor information coming from LiDAR, camera, and so on; which has a Wireless communication.

Figure 3.11 Setting Remote Control

4 State of the Art
4.1 Introduction

Before starting to interact directly with the TurtleBot3 in the laboratory; first, there was an analysis of the algorithms in the literature that dealt with autonomous navigation and Obstacle Avoidance in an unknown o partially unknown environment.

This chapter introduces the literature survey of the various techniques used for mobile robot navigation. Navigation and Obstacle Avoidance are one of the fundamental problems in mobile robotics, which are being already studied and analysed by the researchers in the past 40 years. The goal of navigation is to find an optimal or suboptimal path from a start point to the goal point with Obstacle Avoidance competence (wherein an indoor space could be a wall, door, chairs and so on; instead in an out-door space tree, bushes).

In order to guarantee an autonomous navigation, the robot must be able to safeguard a certain reliability in terms of position (using IMU or GPS sensor) and ensure a map sufficiently precise to generate a path without collisions and faithful to the real one.

The navigation algorithms are divided into two kinds of control: global path planning and local motion control. Global path planning considers owning a priori model or a map of the environment, on which the robot wants to move, using this information calculates the shortest path that allows the motion from a start position to the goal. Whereas local motion is more related to the real-time motion of the robot inside in unknown terrain, where monitoring the environment with the sensors, it can distinguish which and where are the obstacles and generate a motion that will avoid the collision.

A lot of global path planning methods, such as road map, cell decomposition and potential field methods have been explored. They find a complete trajectory from a starting point A to one or more goal points G, where the calculation can be computed off-line, but they produce a reliable path only if a map of the environment is already available. So, in the global navigation, the prior knowledge of the space where the robot should move, must be available.

One of the first method developed for global navigation is the Dijkstra algorithm. [28] Now a day, the A* algorithm is one of the most used of the global path planning. It is a global search algorithm which gives a complete and optimal global path in static environments. It was upgraded in D* (Stenz, A., 1994) [27] for efficient online

searching of a dynamic environment, which gives sequences of path points in the known or partially known environment.

Instead, the local navigation systems advantages are the capacity of producing a new path every time the environment changes (new obstacles found or moving obstacles identify), using the information captured by the sensors, can produce a new path in response to the environmental changes. These algorithms can be separated into directional and velocity space-based approaches.

There is a variety of directional approaches such as Potential field method (Khatib, O., 1986), Virtual Force Field (Borenstein, J. & Koren, Y., 1990) which expands to Vector Field Histogram (Borenstein, J. & Koren, Y., 1991) and Nearness Diagram algorithm (Minguez, J. & Montano, L, 2000), generate a direction for the robot. Velocity space approaches such as Curvature Velocity method (Simmons, R., 1996), and Dynamic Window method (Fox, D.; Burgard, W. & Thrun, S., 1997), achieve an exploration of the commands to manage the robot like translational and rotational velocities.

Analysing the previous considerations, it is easy to understand that a complete robot navigation system should integrate the local and global navigation systems: the global system pre-plan a global path and incrementally search the best new paths when discrepancy with the map occurs; instead, the local system uses onboard sensors to define a path when the information of the map is not yet available, and detect and avoid unpredictable obstacles.

The mobile robot can perform an optimal path from a starting area to an arrival one, utilizing the information related to its geometric points that are matched with the map. If during the designed path an obstacle obstructs its route, the local navigation algorithm is responsible to avoid the collision with the obstacle, for example allowing it to move around the perimeter until the obstacle is overcome, or pre-plan another optimal global path to reach the goal. In this way the global path planner determines a suitable path based on a map of the environment, on the other hand, the Obstacle Avoidance algorithm determines a suitable direction of motion based on the incoming sensor data (real-time events).

For instance, matching the algorithms, that will be discussed later, like the Vector Field Histogram algorithm (VFH) as local Obstacle Avoidance algorithm with the A* search algorithm as a global path planner. With the same reasoning, several well-known local Obstacle Avoidance algorithms such as Dynamic Window (DW) and Nearness Diagram (ND) can be linked with a global path planner (as A* or D*) to perform autonomous navigation in an indoor or outdoor unknown terrain.

4.2 Global Path Searching Method

A real-time Obstacle Avoidance designed considering only on local motion control (Potential Field Method or Dynamic Window Approach for example) have some shortcomings that are related to shape and the motion of the mobile robot and to the knowledge of the environment, that remains snared in local minima when it encounters a dead end. These local minima can be avoided having a global knowledge of the position of the robot respect to the goal, in this way it would be easy going out from them. This global knowledge can be recovered by applying a global path planner.

The A* and D* search algorithms represent the most widely apply either in an indoor or outdoor environment that is partially known or changing. Minimizing its cost function, these algorithms have the skilfulness of rapid re-planning when the conditions of the environment changes, to guarantee an optimal solution of the path from the start position to the goal. The optimal path from any position in the environment can be determined by the following global path information to reach the goal.

D* has been shown to be one to two orders of magnitude more efficient than the A*. This algorithm guarantees an optimal path over grid-based representations of a robot's environment and as already see can be easily combined with the real-time Obstacle Avoidance algorithm.

4.2.1 The A* Algorithm

A* is a search algorithm that is commonly considered in pathfinding and graph traversal, the method calculates an efficiently traversable path between points, that is the graph traversal are called nodes. The algorithm was described by Peter Hart, Nils Nilsson and Bertram Raphael in 1968 [16], and can be considered an extension of Dijkstra's 1959 algorithm. A* achieves better performance and accuracy using heuristics and it is widespread in the world of the robotics for the navigation.

"In 1964 Nils Nilsson invented a heuristic based approach to increase the speed of Dijkstra's algorithm. This algorithm was called A1. In 1967 Bertram Raphael made dramatic improvements upon this algorithm and he called this algorithm A2. Then in 1968, Peter E. Hart introduced an argument that proved A2 was optimal when using a consistent heuristic with only minor changes. His proof of the algorithm also included

a section that showed that the new A2 algorithm was the best algorithm possible given the conditions and finally he called the algorithm A*." [3]

The A* algorithm is designed to follow the path that generates the lowest cost, to do that it preserves a sorted priority queue of different paths that are useful when the robot must change directions. When an obstacle or something that blocks the first direction taken is encountered, the algorithm tries to find a new direction. It calculates which is the new path that minimizes the cost, so if a path with lower cost is found, at any moment, it litters higher-cost path and proceeds towards the lower-cost path.

This process lasts until the aim is reached. A* is based on the best-first search and discoveries a least-cost path from a given initial node to one final node. At the base of the reasoning of the algorithm, there is a function $f(x)$ that is the sum of the function $g(x)\ and\ h(x)$.

$$f(x) = g(x) + h(x)$$

The $f(x)$ is also called 'distance-plus-cost-heuristic' function because $h(x)$ represent the path-cost, so the weight is given by the distance of two points/nodes; instead, $h(x)$ is a heuristic estimate of the distance to the goal. Being heuristic $h(x)$ can be calculated in different ways, but the commonly used for its simplicity coincides with the straight-line distance to the goal. This heuristic function represents the shortest possible distance between two points.

The heuristic function is called monotone or consistent if it guarantees the condition

$h(x) \leq d(x, y) + h(y)$,

where d is the length of that edge. In this case, A* becomes faster and more powerful, no node needs to be processed more than once and A* is equivalent to the Dijkstra's algorithm with the reduced cost:

$$d'(x, y) \coloneqq d(x, y) - h(x) + h(y)$$

The time complexity of A* depends on the chosen heuristic function. In the worst case, it could be an exponential expansion of the nodes in the length of the solution, in the luckiest one it has a polynomial trend when the search space is a tree, is consider a single goal, and the heuristic function $h(x)$ meets the following condition:

$$|h(x) - h^*(x)| = O(\log h^*(x))$$

where h* is considered the optimal heuristic, the exact cost to get from x to the goal.

A* is rest on an optimistic estimate of the cost of the path, in fact, the true cost of a path from the node to the goal will be at least as great as the estimate, and on the admissibility criteria which certified an optimal path thanks to an equal examination of all the nodes.

However, there is also the possibility to modify the algorithm to find an approximated shortest path, in this way, it is possible to speed up the search at the expense of optimality by relaxing the admissibility criterion. Oftentimes we want to bound the relaxation criteria so that we can promise that the solution path is no worse than $(1 + \varepsilon)$ times the optimal solution path.

There are several ε admissible algorithms such as static Weighting, Dynamic Weighting and others. The path found by the search algorithms has a cost that depends on the heuristic function chosen and on the value of ε.

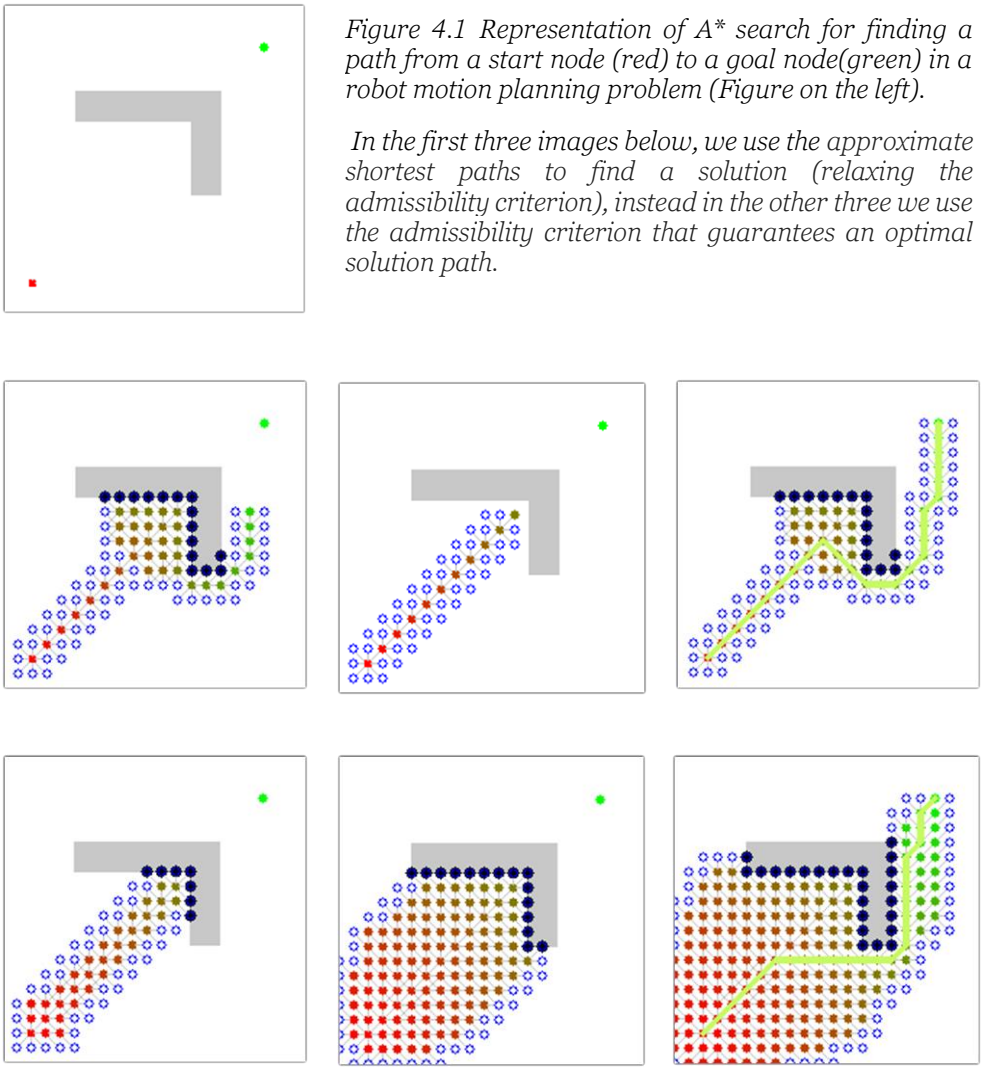


Figure 4.1 Representation of A search for finding a path from a start node (red) to a goal node(green) in a robot motion planning problem (Figure on the left).*

In the first three images below, we use the approximate shortest paths to find a solution (relaxing the admissibility criterion), instead in the other three we use the admissibility criterion that guarantees an optimal solution path.

The purpose of the example above is to show the difference between A * with the use of the admissibility criterion (second sequence of images) and the A* in which we speed up the search by relaxing the admissibility criterion (first sequence of images). On both the case the starting point and the goal is considered the same.

The empty circles represent the nodes in the open set, i.e., those that remain to be explored, and the filled ones are in the closed set. As it possible see when we use the admissibility criterion the final path is the optimal one (green path in the lower image on the right), but we pay in terms of number nodes that we must analyse, so in time. Instead using a relaxation of the criteria, we obtain a solution that is no worse than $1 + \varepsilon$ (in this case $\varepsilon = 0.5$) times the optimal solution path, gaining time because the number of nodes visited has greatly reduced, as is possible see comparing the areas of the two examples, that represent the number of nodes analysed.

4.2.2 The D* Algorithm

D * is a search algorithm for finding the minimum path on a graph, it always based like the A* on the original Dijkstra. One of the differences, from the previously presented A* algorithm, is that the procedure of the D* to look for a path starts from the goal and going backwards to the original point, choosing from time to time the less expensive arc until reaching the starting node.

The main improvement respect to the A* is that this algorithm gives the possibility to update the path every time there is a change on the environment, that means as soon as the cost of the arcs is changed. Doing so, an alternative road or an improved one, with a lower cost, could be chosen.

However, the D* requires a procedure that is computationally onerous, respect also to the A*, because it tries to find a path not only from the goal, but also for all the nodes that are about as far from the target. For this reason, is some case the A* has a higher efficiency, introducing the heuristic estimate of the distance between the start and the arrival, is able to limit the node that is analysed during the calculations.

The D* algorithm starts from the assumption that the map of the area is partially or totally know and having it, the goal is to find the safest and efficient motion for the mobile robot, given a start point, a goal point. The robot should go from the start position to the goal, being able to avoid the collision with the obstacles along the path. The paths generated at the beginning, when the map is incomplete may turn out to be

invalid or suboptimal, as soon as it receives new information from the sensor that update the original map. So, it is necessary that the robot is able to updates its map and re-plan optimal paths each time a new information coming.

For its qualities, the D* search algorithm, letting re-planning to occur incrementally and optimally in real time, is suitable for in a partially known environment with hurdle terrains.

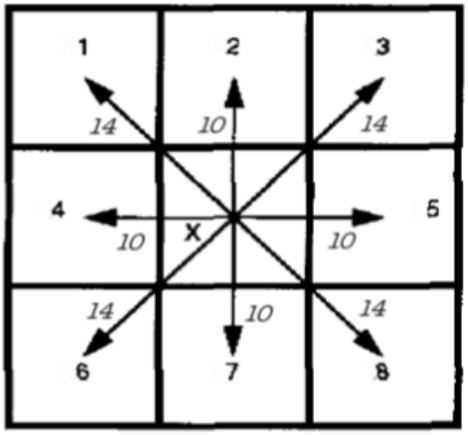


Figure 4.2 Weighted arcs that connect the neighbours' node

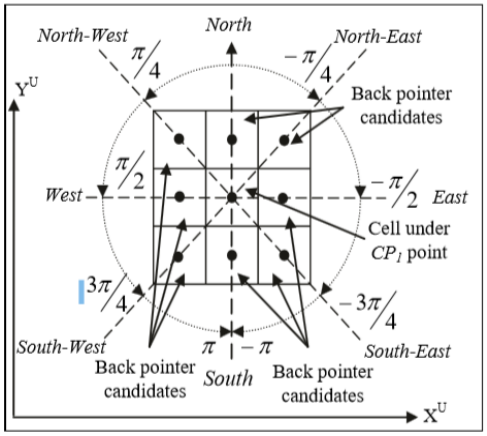


Figure 4.3 how to compute the global path information

The Figures 4.2 and 4.3 represent two-dimensional Cartesian histogram grid. The first Figure with a back-pointer, for each of the neighbours, indicates the direction to the goal (north, south, east, west, north-west, south-west, north-east and south-east).

This information is crucial for the motion of the robot because with these criteria it knows from any position the steering direction toward the goal, so finding a global path information agreement to the back-pointer. A graph consists of a set of N nodes connected to each other by weighted arcs. The map is commonly given as a grid occupation, where each node is connected to eight neighbours.

The other one shows the values of crossing a free arc, that are commonly used when the D* algorithm (or the A* algorithm) is applied. The cost is equal to 10 in the case of lateral displacements, to 14 for a diagonal movement (that correspond to the $\sqrt{2}$) and infinite when the node is occupied by an obstacle.

Figure 4.4 displays the global path information result of a simulated obstacles course with a given start and goal positions. Following this direction, the mobile robot can reach the goal, this procedure permits the robot saving time wasted in the path tracking operation.

The cost of traversing from cell Y to X, already represented in Figure 4.3, can be indicated by the arc cost function $c(X,Y)$, by the following equation:

$$c(X,Y) = \begin{cases} 10 & if\ Y\ is\ situated\ horizontally\ or\ vertically\ to\ X \\ 14 & if\ Y\ is\ situated\ on\ the\ diagonal\ to\ X \\ infinite & if\ Y or\ X\ is\ considered\ as\ an\ obstacle \end{cases}$$

So, $c(X,Y)$ indicates the cost of crossing the arc that joins X and Y. The D* algorithm is based on an open list $\mathcal{L}$ that contains all the nodes to be analysed and indicates the node in three ways NEW, OPEN and CLOSED; respectively if the node has never been in $\mathcal{L}$, the node $\in \mathcal{L}$, the node is come out from $\mathcal{L}$.

The D* algorithm uses a cost functions $h(X)$ that represent the weight of each node. This function keeps track of the sum of the path costs of each node to the goal, while the function $k(X)$, assumes the minimum between the $h(X)$ current and all the values previously taken from it since the node X was introduced into $\mathcal{L}$. Considering the i-th iteration,

$k(X) = min_i\ h_i(X)$.

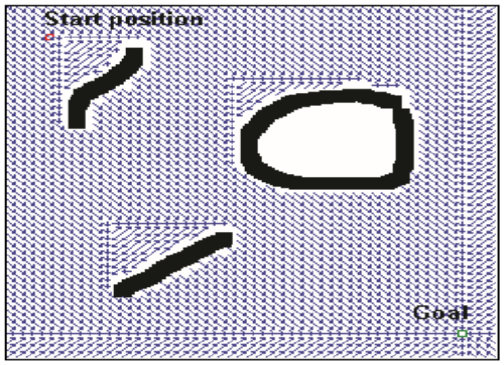


Figure 4.4 Path from a start position to the goal

With this consideration, it makes sure that the path found by the pointers corresponds to an optimal $k(x)$, examining step by step if the neighbouring nodes propose a better solution looking from the goal and advancing backwards towards the starting node.

The main advantage is the capability to quickly update the new information of the path cost in case something changes in the environment, this changing on the arc cost are managed by continuous deletions and upload of node inside the list $\mathcal{L}$. The aim is to maintain with low cost (that are the points most favourable) the nodes closer to the goal, and trying to keep the path optimal, enlarge the nodes considering until the starting point is reached. The nodes that are inside the open list $\mathcal{L}$ are analysed from the one with a lower value of $k(x)$,

$$k_{min} = min_{x \in \mathcal{L}} k(x)$$

Is the minimum possible cost present between the nodes that are in the open list $\mathcal{L}$ (that can be considered the best path). The nodes inside the function $k(X)$ are divided into two categories Raise node if $k(X) < h(X)$, and Lower node if $k(X) = h(X)$.

Analysing these two categories is easy to understand that the Lower node can be considered as the optimal solution because their function coincides with the minimum cost value. Instead, some considerations cannot be done for the Raise nodes, since the actual cost function $h(X)$, when they are investigated, do not correspond to k_{min}. This means that following the pointers starting from a LOWER node up to the goal, the resulting path is minimal.

Following the example of Figure 4.5, let imagine starting from a free-plan situation as in the sub-Figure (b), knowing the map, the starting point and the goal, and that at some point an obstacle is identified (which, in this example, occupies nodes 2, 6 and 10). These nodes, whose path was excellent, so they are LOWER, are inserted in the open list and their processing will cause the addition of all the nodes that pointed to them (nodes 1, 5, 9, 13), these are now RAISE nodes.

First of all, you search among all the neighbours, the one who has the lower cost. Then, for each neighbour the cost change is propagated if the neighbour is a successor, if the neighbour can be further improved the node itself must be inserted in $\mathcal{L}$. Finally, it is checked whether the neighbour can improve the road.

The result of these operations, implemented on the nodes in question, leads to the situation of the sub-Figure (b), in which the new path has been identified. If one of the obstacles returns free again (becomes LOWER), for example, the node number 2, the algorithm re-plans a new solution leads to sub-Figure (c).

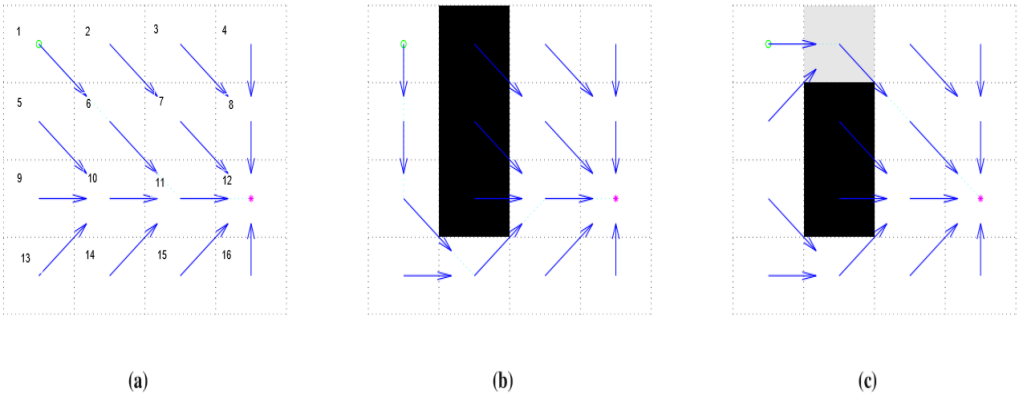


Figure 4.5 Behaviour of D Algorithm*

4.3 Local Motion Control

The techniques analysed up to now, global motion planning, are useful to calculate a collision-free trajectory for the robot, from the starting point the goal, when the around the environment in partially or totally know. However, when the robot is in a complete unknown area and does not have information about the surrounding area, these algorithms fail and do not produce any solution. For this kind of situations, the local motion planning is more suitable.

The objective, using the Obstacle Avoidance algorithm, is to move a robot towards an area that is free of collisions thanks to the information handled by the sensors during motion execution. The improvement of the Obstacle Avoidance is to find a direction for the robot by introducing the sensor information, which is steadily updated, used to control the motion real-time.

The main issue of considering the reality of the world during execution is locality, that means the localization of the robot during the execution of the algorithm in the environment (that, as we will see, is one of the problems encountered when the robot travels long path). An error in the location of the robot generates a series of problems, that depends on the first one, as a wrong map, and a trajectory that does not coincide with the real one.

Notwithstanding that limitation, Obstacle Avoidance techniques are mandatory to deal with robotics problems in the unknown and changing environment.

4.3.1 Definition of Obstacle Avoidance

"Let *A* be the robot moving in the workspace *W*, whose configuration space is CS. Let *q* be a configuration, q_t this configuration in time t, $A(q_t) \in W$ the space occupied by the robot in this configuration.

If in the vehicle there is a sensor, which in q_t measures a portion of the space $S(q_t) \subset W$ identifying a set of obstacles $O(q_t) \subset$ W. Let *u* be a constant control vector and $u(q_t)$ this control vector applied q_t during time δt. Given $u(q_t)$, the vehicle describes a trajectory

$q_t + \delta t = f(u, q_t, \delta t)$, with δt ≥ 0.

Let $Q_{t,T}$ be the set of the configuration of the trajectory followed from q_t with δt ∈ (0, T), a given time interval. T > 0 is called the sampling period. Indicating with q_{target} a target configuration. Then, in time t_i the robot A is in q_{ti}, where a sensor measurement is obtained $S(q_{ti})$, and thus an obstacle description $O(q_{ti})$." [6]

The goal is to find a trajectory for the robot, that is able to avoid the collision with the around the obstacle such that, $A(q_{ti,}T) \cap O(q_{ti}) = \emptyset$, producing a motion that brings the robot closer to the target $F(q_{ti}, q_{target,}) < F(q_{ti} + T, q_{target})$.

Scanning in real-time the environment, the robot can calculate the sequence of motions that allow the avoidance of the obstacles gathered by the sensors, while making the vehicle progress towards the target location (Figure 4.6). So, the local motion planning method tries to contrast the issue related to the locality with the advantages related to the real-time information of the mechanical devices.

We will now present some Obstacle Avoidance algorithms, which have disadvantages and advantages depending on the different factors: the type of area covered (indoor or outdoor), the performance of the robot (linear and angular velocity), and the shape of the obstacles. They can be divided into two groups, methods that find the motion in one step and the one that requires more than one.

These algorithms have been of fundamental importance for the study and writing of autonomous navigation algorithms presented in chapter 4.

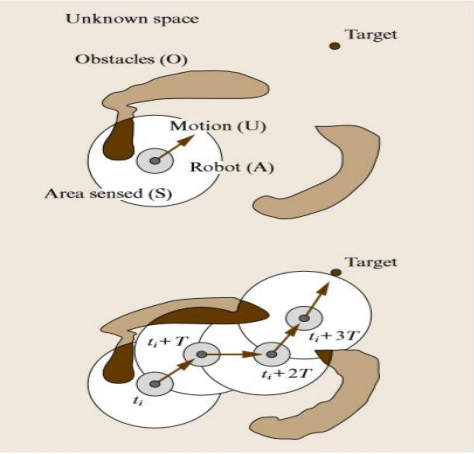


Figure 4.6 With the obstacle avoidance algorithm we can avoid collisions with the obstacles using the information gathered by the sensors while driving the robot towards the target location

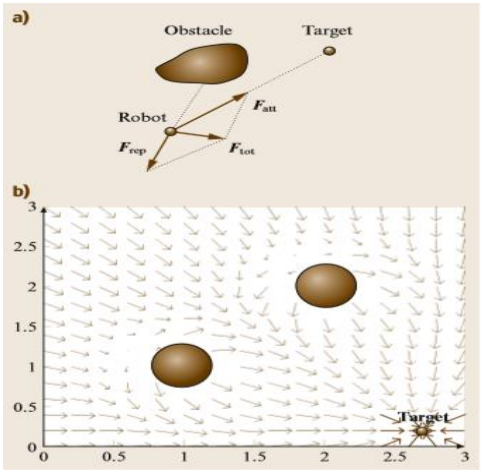


Figure 4.7 Potential field method, thanks to we compute the motion direction. The target attracts the particle F_{att} instead the obstacle exerts a repulsive force F_{rep}

4.3.2 Potential Field Method

The first method of investigating is the potential field method (PFM). It represents the robot as a particle that moves in the workspace *W*, whose configuration space is CS and is subject to forces that are produced by the surrounding environment. The forces can be of two types attractive or repulsive. Indeed, the target propagates an attractive force F_{att} for the robot, while all the obstacle captures by the sensor return a repulsive one F_{rep}.

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

The repulsive force can be calculated relating it only with the distance from the obstacles or considering also the instantaneous robot velocity and accelerations of the robot.

$$F_{rep}(q_{ti}) = \begin{cases} K_{rep} \sum_j \left(\frac{1}{d(q_{ti}, p_j)} - \frac{1}{d_0} \right) n_{pj} & if\ d(q_{ti}, p_j) < d_o \\ 0 & otherwise \end{cases}$$

where K_{att} and K_{rep} are the constants of the attractive and repulsive forces, d_0 is the influence distance of the obstacles p_j, q_{ti} is the current vehicle configuration and $n_{qtarget}$ and n_{pj} are the unitary vectors that point from q_{ti} to the target and each obstacle p_j.

$$F_{rep}(q_{ti}) = \begin{cases} K_{rep} \sum_j \left(\frac{a\dot{q}_{ti}}{[2ad(q_{ti}, p_j) - \dot{q}^2{}_{ti}]} \right) n_{pj} . n_{\dot{q}_{ti}} & if\ \dot{q}_{ti} > 0 \\ 0 & otherwise \end{cases}$$

where $\dot{q}_{ti}$(dot) is the current robot velocity, $n_{\dot{q}_{ti}}$ the unitary vector pointing in the direction of the robot velocity, and a is the maximum vehicle acceleration.

Combining these two forces, the trajectory of the robot can be computed at every time t_i:

$$F_{tot}(q_{ti}) = F_{att}(q_{ti}) + F_{rep}(q_{ti})$$

Applying a control u_i, the total force $F_{tot}(q_{ti})$ can be used to control the trajectory of the robot (Figure 4.7). This algorithm is generally used because it is simple to develop and requires short computational time.

4.3.3 Vector Field Histogram

The second Obstacle Avoidance algorithm presented, is the vector field histogram (VFH), it produces a solution dividing the problem into two steps, in the first create a set of aspirant motion directions for the robot trajectory; then in the second step, using defined rules, selects the proper one.

At the beginning, space is divided into sectors from the available sensor of the robot. For Figure 3. the histogram *H* represent the obstacles located around the robot. The function $h^k(q_{ti})$ describes the density of the obstacle that is proportional to the probability function P(p) (probability that a point is busy) and to the distance from the obstacle, the more the distance from the obstacle increase, the more the density value is lower.

The function $h^k(q_{ti})$ is:

$$h^k(q_{ti}) = \int_{\Omega_k} P(p)^n \cdot \left(1 - \frac{d(q_{ti}, p)}{d_{max}}\right)^m \cdot dp$$

The resulting histogram, produced by the Vector Field approach, has sectors with low density that represents the area free or with far obstacles, and sectors with high density (hill) that describes the area occupied by the obstacles. The set of candidate directions in which the motion is allowed to move is given by the set of the adjacent sector with a density lower than a given threshold, as much as possible closest to the direction of the target direction, this area is named the selected valley.

The procedure to choose the right direction for the robot respect to the target k_{target}, depend on where the target respect to the selected area is, and on the size of the valley (Figure 4.8). Three cases are identified, which are analyzed in sequence. First, if the goal sector is inside the selected valley, then the $k_{sol} = k_{target}$. The second, if the goal sector is not in the selected valley and the number of sectors of the valley greater than *m*, in this case, $k_{sol} = k_i + \frac{m}{2}$, where m is a fixed number and k_i is the direction of the sector that is closer to the target and has a probability lower than the defined threshold. The last one, if the goal sector is not in the selected valley and the number of sectors of the valley lower or equal to *m*; in this case, the solution is $k_{sol} = (k_i + k_j)/2$, where $k_{i,j}$ are the extremes of the area selected. The result is a component or sector k_{sol}, whose bisector is the direction solution θ_{sol}.

The velocity v_{sol} is inversely proportional to the distance to the closest obstacle. The control is $u_i = (\theta_{sol}, v_{sol})$.

Given the selected sector inside the histogram k_{sol}, calculating the direction of the motion (angle θ_{sol}) and the velocity of the robot v_{sol}, the robot can move independently avoiding the area with a high probability obstacle distribution, scanned by the sensor.

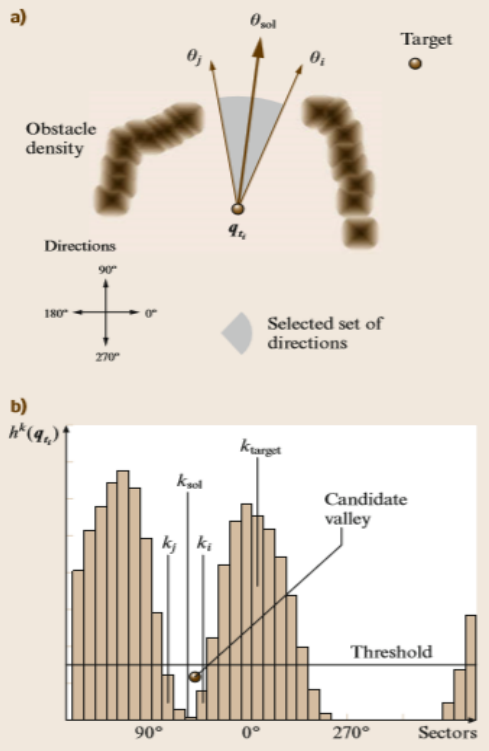


Figure 4.8 SubFigure (a): Robot motion direction θ_{sol} and obstacle occupancy distribution. SubFigure (b): The candidate valley is the set of adjacent components with values lower than the threshold. The navigation case is the third previously considered, since the sector of the target

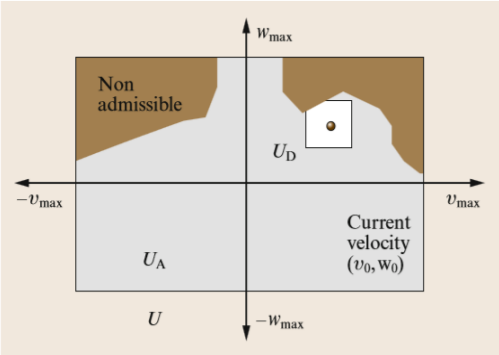


Figure 4.9 Subset of control U_R, where U contains the controls within the maximum velocities, U_A the admissible controls, and U_D the controls reachable in a short period of time

4.3.4 Dynamic Window Approach

Another Obstacle Avoidance algorithm that solves the problem in more than one step is the Dynamic Window Approach (DWA). Firstly, it defines the candidate set of control space $\mathcal{U}_R$, which is constrained by the specification of the robot, the maximum linear and angular velocities, characterized by $\mathcal{U}$,

$$\mathcal{U} = \{ (v, w) \in R^2 \setminus v \in [-v_{max}, v_{max}] \wedge w \in [-w_{max}, w_{max}] \}$$

The candidate set of controls $\mathcal{U}_{\mathrm{R}}$ contains the controls: within the maximum velocities of the vehicle $\mathcal{U}$, that generate safe trajectories $\mathcal{U}_{\mathrm{A}}$, and that can be reached within a short period of time given the vehicle accelerations $\mathcal{U}_{\mathrm{D}}$.

Instead, the array $\mathcal{U}_{\mathrm{A}}$ holds the controls to achieve an efficient and safe trajectory,

$$\mathcal{U}_A = \{ (v, w) \in \mathcal{U} \setminus v \leq \sqrt{2 d_{obs} a_v} \wedge w \leq \sqrt{2 \theta_{obs} a_w} \}$$

where d_{obs} and θ_{obs} are the distance to the obstacle and the orientation. Finally, the set $\mathcal{U}_D$ contains the set of the area that can be reached in a short period of time

$$\mathcal{U}_D = \{ (v, w) \in \mathcal{U} \setminus v \in [v_0 - a_v T, v_0 + a_v T] \wedge w \in [w_0 - a_w T, w_0 + a_w T] \}$$

The resulting subset of controls (Figure 4.9) is:

$$U_R = U \cap U_A \cap U_D$$

The last point is the selection of the proper control $u_{\mathrm{i}} \in \mathcal{U}_{\mathrm{R}}$, to do that, maximizing an objective function that depends on how much we get close to the goal, on the clearance of the path chosen and, on the velocity, reachable on that point by the robot.

$$G(u) = \alpha_1 \cdot Goal(u) + \alpha_2 \cdot Clearance(u) + \alpha_3 \cdot Velocity(u)$$

DWA solves the problem in the control space using information of the vehicle dynamics, thus The DWA method works well on the robot with slow dynamic capabilities.

The theory of the Obstacle Avoidance algorithm, which has been analysed until now, can be found in the chapter 'Moving in the environment' of the Springer Handbook of robotics. [6]

4.3.5 VFF Approach for Obstacle Avoidance

The last method treated is the VFF method. From Figure 4.10 is possible to see as these forces are applied at point CP_1, it determines the local steering direction θ_{rep} calculating the frontal and the lateral forces to avoid the collision with the obstruction, analysing the information coming from the sensors.

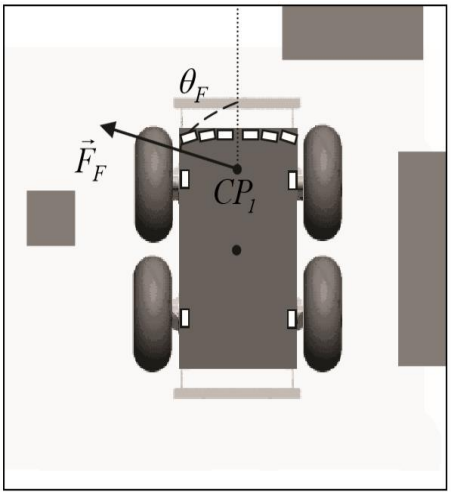


Figure 4.10 Frontal repulsive force F_F

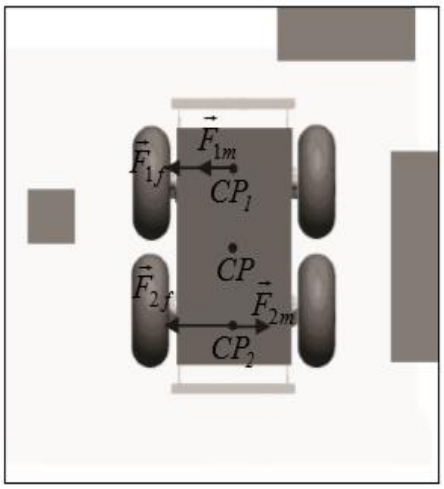


Figure 4.11 The lateral computed forces decomposition

The final repulsive force $\overrightarrow{F_F}$ is the vector sum of the individual forces generated by the obstacle, given by:

$$\vec{F}_F = \sum_i \frac{F_{cr}\, C_i}{d_i^2} \left[\frac{x_i - x_c}{d_i}\, \vec{\imath} + \frac{y_i - y_c}{d_i}\, \vec{\jmath} \right]$$

Where F_{cr} is the force constant, d_i the distance between the obstacle (x_i, y_i) and the point CP_1 (x_c, y_c), C_i is the probability that the cell *i-th* is occupied. The VFF method guarantees also a safe distance from any lateral collision, computing the lateral force $\vec{F}_s$,

$$\vec{F}_S = \frac{F_{cr}\, C_i}{d_i^2} \left[\frac{x_i - x_s}{d_i}\, \vec{\imath} + \frac{y_i - y_s}{d_i}\, \vec{\jmath} \right]$$

Where (x_s, y_s) the lateral sensor coordinates. Applying the principle of free-body diagrams, all these forces can be assembled by a single lateral force F_L and a moment M, acting on the robot on the centre point CP, and successfully divided depending on the shape of it.

On the case of the robot represented in Figure 4.11, where there are front and rear steering wheels, the forces are divided into a couple of force $F_{1,2}$ and a couple of moment $M_{1,2}$. In fact, the two force F_{1m} and F_{2m} are computed from the moment M, knowing the distance d between CP and $CP_{1,2}$ as,

$$F_{1m} = F_{2m} = \frac{M}{d} \qquad F_{1f} = F_{2f} = \frac{F_L}{2}$$

If we compare this consideration with the TurtleBot3 (Waffle or Burger) available in our laboratory, they do not have 4 wheels and those degrees of freedom, so the forces F_{2f} and F_{2m} are not taken into consideration because they have only two degrees-of-freedom.

Considering the TurtleBot3 configuration, to generate the final trajectory for the robot, the VFF approach produces a final repulsive force F_{Rep} (where the choice of the parameters α, β and γ determines the trajectory of the robot), that gives the value of the linear velocity pushing the robot away from the obstacle, and an angle θ_{Rep} that correspond to the direction (Figure 4.12).

$$\vec{F}_{rep} = \alpha\, \vec{F}_{1f} + \beta\, \vec{F}_{1m} + \gamma\, \vec{F}_F$$

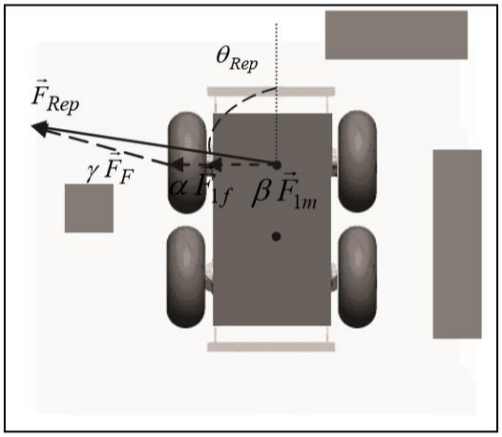


Figure 4.12 Steering Direction and Repulsive Force

4.4 General Navigation System

In this chapter has shown the global navigation planning and the Obstacle Avoidance methods integrated into real systems. On the one hand, the Obstacle Avoidance methods are local techniques to avoid the collision with the obstruction.

However, they can have the problem to fall in local minima that mean in trap situations or cyclic motions for the robot. This exposes the necessity of a different navigation system, the motion planning techniques. It computes a geometric path free of collisions, from a starting point to a defined goal with a map of the environment known; nevertheless, when the scenarios are unknown and evolve, these techniques fail, since the precomputed paths will almost surely collide with obstacles.

The solution, that is one key aspect to create a motion system is to mix together the best of the global knowledge given by motion planning and the reactivity of the Obstacle Avoidance methods.

The idea, to generate the trajectory, is to precompute a path to the target using the global motion, that is modified real-time as a function of the changes in the scenario obtained from the sensor information and to use a planner at a high frequency with a tactical role, leaving the degree of execution to the reactor.

The algorithmic tools offered in this chapter display that motion planning and Obstacle Avoidance research techniques have reached a level of maturity and complexity that allow their transfer onto real platforms.

5 Navigation
5.1 Introduction to the Navigation Algorithm

This chapter describes the main work that has been done in this thesis at the LIM, that is related to the creation of an algorithm to make TurtleBot3 perform an autonomous navigation inside a unknow indoor environment.

What does autonomous navigation mean? By definition, autonomous navigation is the ability of a vehicle to plan its route and execute its plan without human intervention. This implies two different problems: knowing how to move towards a target and having the ability to avoid any obstacles along the way.

However, both these goals are not easy to achieve, to be able to perform the right motion of the robot given by the algorithms we need other information: indeed, it is important to know where the robot is, to create a map of the given environment, to interact every time with the map and to optimize the route to get a smooth path.

The four needed features are the map, the pose of robot, sensors and a navigation algorithm.

Although the main objective of this thesis concerns the development of navigation algorithms, the other aspects mentioned above will be discuss and briefly described to provide the reader with a more complete view of the subject matter.

The first essential feature for navigation is the map. Using RViz, the navigation system is equipped with a very accurate map from the time of purchase, and the modified map can be downloaded periodically so that the robot can be driven to the destination based on the map.

Like a navigation system, a robot needs a map, so we need to create a map and provide it to the robot, otherwise the robot should be able to create a map by itself. SLAM (Simultaneous Localization and Mapping) is developed to let the robot create a map with or without the help of a human being. This is a method of creating a map while the robot explores the unknown space and detects its surroundings, estimating its current location as well as creating a map.

For what concerns this first point the tool RViz that, has been used as we will see, it is able to create a map of the environment while the robot is moving, capturing the information of the obstacles given by the sensors (in our case LiDAR) and reading the IMU data that gives back the position of the robot.

Second feature, the robot must be able to measure and estimate its pose (position + orientation), in case of a real vehicle, the GPS is used to estimate its pose. Nevertheless, the GPS cannot be used alone in an indoor area because it introduces large errors that cannot be acceptable for performing the autonomous navigations.

In order to overcome this problem, various methods such as marker recognition and indoor location estimation have been introduced, but they are still insufficient for the general use in terms of cost and accuracy. Furthermore, the application of markers assumes that the environment is structured and well known.

Currently, the most widely used indoor pose estimation method for service robots is dead reckoning, which is a relative pose estimation, it has been used for a long time and it is composed by low-cost sensors and it can obtain a certain level of accuracy in pose estimation.

The amount of movement of the robot is measured by the odometry of the wheel. However, there is an error between the calculated distance with wheel rotation and the actual travel distance. Therefore, the inertial information from the IMU sensor can be used to reduce the error by compensating position and orientation error between the computed value and the actual value.

Third, figuring out whether there are obstacles such as walls and objects requires sensors. Various types of sensors such as distance sensors and vision sensors are used. The distance sensors include laser-based distance sensors (LDS, LRF, LiDAR), ultrasonic sensors and infrared distance sensors. The vision sensors include stereo cameras, mono-cameras, omnidirectional cameras, and recently, RealSense, Kinect, Xtion, which are widely used as Depth camera, to identify obstacles.

The last essential feature for navigation is to calculate and travel through the optimal path, the navigation algorithms that are described are based on both the algorithms presented on chapter 3, the global navigation algorithms and principally on the Obstacle Avoidance algorithms.

The global path searching algorithms, as the A* and D* algorithms, always consider a starting point, a goal and a complete map that is periodically loaded; instead we analyse a different starting point and condition of the space.

We consider that the robot moves in a completely unknown indoor environment; the available robot's and sensor's hardware for the test are the TurtleBot3 and its LiDAR sensor and our goal is to completely map the space in which the robot is navigating.

5.2 Algorithms

The first goal, following some basic exercises of the 'ROS Robot Programming' book [1], was to be able to perform some movement in a simulated environment to understand the behaviour of the tools, such as ROS, that were working together at the same time. In this example (Figure 5.1), the 'World' environment in Gazebo has been loaded (as a simulation framework) with the Waffle TurtleBot3 but there are other predefined environment models that could be loaded such as: 'House' and 'Empty', which themselves could be also modified.

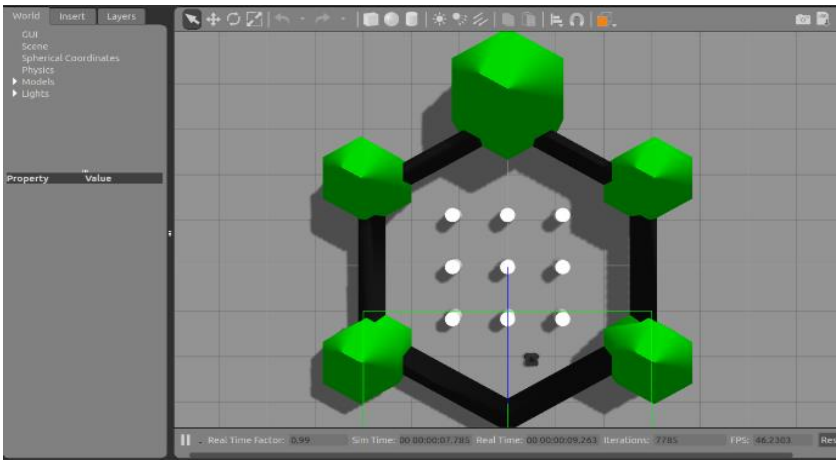


Figure 5.1 Waffle loads on the World environment of Gazebo

It is useful to understand how using RViz we have the possibility to build the map of our environment. Thanks to RViz, we could visualize the position of the Waffle operating in Gazebo given by the Odometry topic, the virtual LiDAR data (as its show on Figure 5.2) but also other information such as the camera image (not used in this example) and to virtually detect the collision.

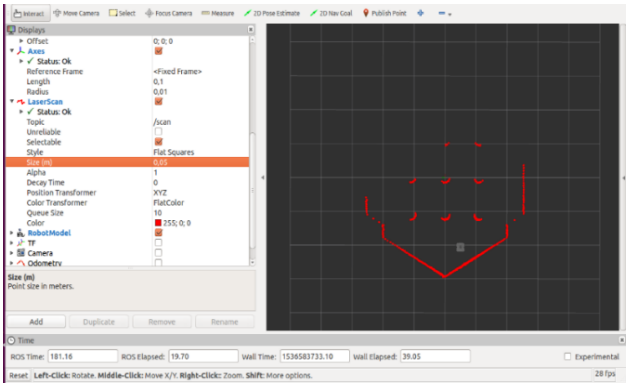


Figure 5.2 RViz views of the data coming from the sensors

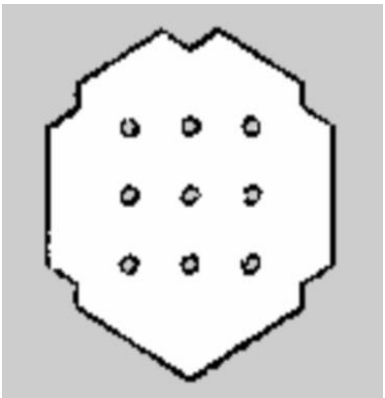


Figure 5.3 Map of the environment

The final goal of this example is to move the robot and to create a map of the environment. This is done using two command: 'TurtleBot3_teleop_key' and 'TurtleBot3_slam.launch'.

The first gives the possibility to drive the robot around, via teleoperation, using the following commands (human guidance):

key_mapping = {'w': [0, 1], increase the linear velocity of 0.01
'x': [0, -1], decrease the linear velocity of -0.01
'a': [-1, 0], increase the angular velocity of 0.1
'd': [1, 0], decrease the angular velocity of -0.1
's': [0, 0], to stop the robot}.

The second command runs SLAM (Simultaneous Localization and Mapping), that explores and creates a map of the unknown environment while continuously updating new information of the obstacles that the robot captures while it is moving and estimating the pose of the robot that is obtained from the data of the sensor.

To perform SLAM, the program needs the distance values measured from the around objects and coming from the robot sensors, together with the pose and orientation that are taken by the odometry of the robot. Encoders and inertial measurement units (IMU) are adopted for pose estimation.

Simulation is a very useful tool for testing the developed algorithms and trying to predict the behaviour of the robot without actually using it, Figure 5.3 shows the complete map of the 'World' environment that has been saved after the robot has mapped all the simulation environment of Gazebo.

However, some limitations are present due to the characteristics of the algorithm that generates the SLAM, that is the Gmapping algorithm [30].

Without going into the detail of the algorithm, since SLAM would require a long speech, there are certain constraints: "square shaped room with no obstacles, a long corridor without any distinctive objects, glasses that doesn't reflect laser or infrared light, mirrors that scatters light, wide and open environments where obstacle information cannot be acquired, such as a lake or sea". [1]

5.2.1 Follow Wall

After this brief introduction on how to use some tools, which will be investigated successively, now it is possible to focus the attention on the navigation algorithms.

The first algorithm that has been written for autonomous navigation, "Follow Wall" takes its name from its simplicity. Its purpose is to identify obstacles that can be near or far from the robot, being able to distinguish two cases of collision if there are walls or other types of obstacles that partially block its motion, and finally, choosing the best trajectory.

The sensor with which the obstacles were identified is the LIDAR available in both TurtleBot3 robots. The Waffle camera is not been used as already mentioned on previous chapter. From Figure 5.4, is possible to identify the areas in which the robot divides the space in front of it. This algorithm can evaluate five different representations of data coming from the LiDAR, to identify the obstacles around the robot itself: '*No obstacle*', '*Far obstacle*', '*Near obstacle on both the direction*', '*Left Near obstacle*' and '*Right Near obstacle*'.

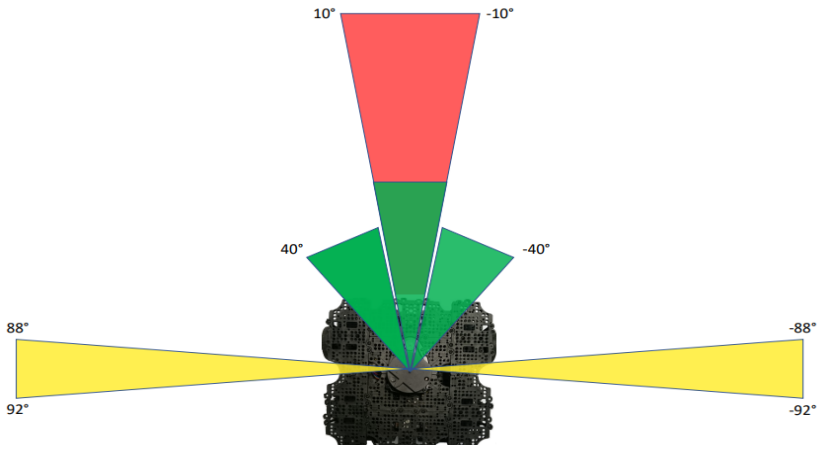


Figure 5.4 Area divided by the Follow Wall algorithm

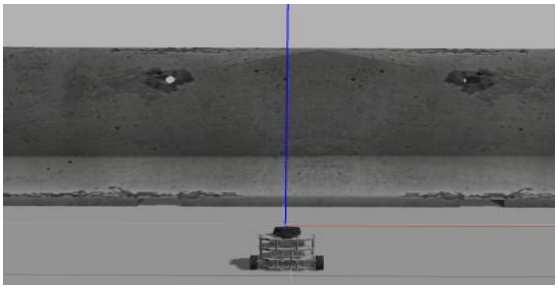


Figure 5.5 Waffle in front of a wall in the simulated environment Gazebo

'*No obstacle*' means that the robot does not find any impediment either in front of it or around it, so the red and green are free one area. The green area represents the one close to the robot, instead, the red one is used to determine far obstacle in front of it. In this case, the linear velocity of the robot is set to 0.26 m/s (that represent the maximum value reachable by the TurtleBot3) and the angular velocity is equal to 0 rad/s.

When the green area only is free, we move to the second case, that is '*Far obstacle*'. In this condition, the robot understands that there could be an obstacle in front of it, but it is still far from deciding to stop itself (that means that the LiDAR data from 1.5 to 0.8 meters identify an obstacle), so it decreases its velocity and the new value of linear velocity becomes 0.15 m/s.

All the other three cases '*Near obstacle on both the direction*', '*Left Near obstacle*' and '*Right Near obstacle*' are related because, in all of them the green area is no longer free, this means that the robot recognized an obstacle around it, the only difference is related to the position in which the obstacle is identified.

Whenever the green area is no longer free, this means that the robot recognized an obstacle around it and tries to understand if it is a wall. (Figure 5.5). This is done looking of 40° degrees in front of it at 0.8 meters of distance if for the 90% of the scanned angles it finds an obstacle (equal or higher than 36 degrees) it identifies the object as a wall.

If a wall has been detected, the robot looks if it has a free space on the right or on the left side. This reasoning is done by trying to identify an "infinite" distance, which for the capacities of the LiDAR is equivalent to almost 5 meters, looking of 5° degrees into the left and right of the robot; this check is represented by the yellow area.

If using the LiDAR, the robot recognizes an "infinite" free space, it rotates towards that direction, with a high angular velocity equal to +0.63 rad/s to the left, -0.63 rad/s to the right (if both the directions are possible, the right is chosen), while the linear one is equal to 0 m/s.

Otherwise the robot tries to avoid the obstacle by turning on the opposite direction respect to the obstacle until it finds free space in front of it. In the case where the robot identifies the obstacle on both sides, '*Near obstacle on both the direction*', the direction that moves towards the areas with lower density of hurdles will be chosen.

In this last case, the angular velocity is set to ± 0.2 rad/s depending on the kind of rotation chosen (+ anticlockwise, - clockwise).

This algorithm represents the first version of Obstacle Avoidance and it has been tested on a simulative environment, Gazebo, to control the correctness of the motion, where a shape of a wall has been loaded into an empty space. Subsequently, it is been tested on both the available TurtleBot3 (Burger and Waffle). It is a very simple algorithm, which requires brief calculation time, but which works very efficiently.

For example, when it was tested inside a square room in Gazebo, the algorithm was able to make the robot run across the entire perimeter, managing to keep the same distance from the wall. In a real environment, it was tested on the Burger, where was assembled the Raspberry Pi Camera V2 to add an onboard video feature, being able to provide real-time simultaneous localization and mapping (SLAM) of the floor using RViz. For what concerns of SLAM, the robot should have mounted a sensor capable of measuring the distance on the XY plane, such as LDS (Laser Distance Sensor), LRF (Laser Range Finder) or LiDAR.

The result is shown in Figure 5.6, that represents the map of the first floor of LIM department that the robot was able to create, using the RViz algorithm.

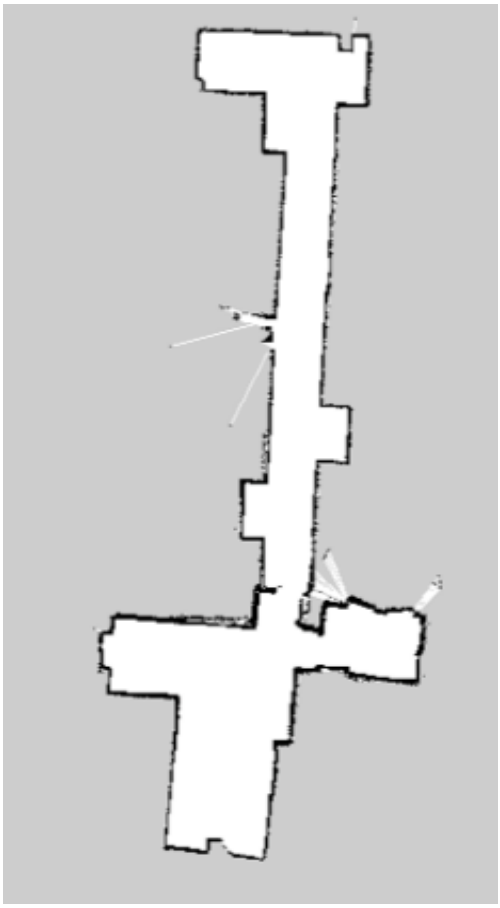


Figure 5.6 Map of the floor given by Rviz

5.2.2 Obstacle Avoidance

The second navigation algorithm, called “Obstacle Avoidance”, tries to improve the previous one. It analyses a greater amount of data coming from the LiDAR and performs more complicated robot motions.

Running the previous algorithm, the robot was only able to move straight on or to rotate around itself when it found an obstacle in front of it until it found again a free space. To improve this algorithm the idea is to use the angular and the linear velocity together, to make the robot able to perform a curved trajectory avoiding the collisions with the obstacles.

The structure of the motion given by the algorithm is based on three different spaces in which the robot could move; the robot still use the LiDAR to detect obstacles but now it scans all the 180° in front of it (instead in the previous algorithm it scanned only 80° to find on obstacle).

Then, the scanned space is divided into three different sets: from infinite to 2.5 meters (blue), from 2.5 meters to 1 meter (green) and from 1 to 0.5 meter (yellow), all represent into Figure 5.7. Each set is divided into seventeen subsets, that means seventeen possible directions, represented by the cones in which the three main areas are separated. As it is possible see for each set, the dimensions of the subsets are different (in degrees), but the total scanned space for the whole are is 180°.

When the robot moves, it looks if it has free space in front of it, starting from the farthest distance (the blue area from infinite to 2.5 meters), if there is, it proceeds in that direction. Otherwise, the robot alternatively looks at the nearer subset into the right and left sides of the same area (from the darkest to the softest colour), trying to find a free space.

In order to have a positive check inside a subsystem, the algorithm must verify that, for all the angles of that subsystem, the values received by the LiDAR are equal or greater than the minimum limit of that area (blue 2.5 m, green 1 m and yellow 0.5 m).

This analysis is repeated starting from the farthest distance (blue area), to the nearest one (yellow area), the goal is to find the farthest distance that is as close as possible to the middle of the robot direction. The algorithm each time receive new information from the LiDAR (with a frequency of 5 Hz), selects a new cone of a certain zone. The linear velocity is fixed for each zone, for the blue area is 0.26 m/s, for the green are is 0.13 m/s and for the yellow one is 0.07 m/s.

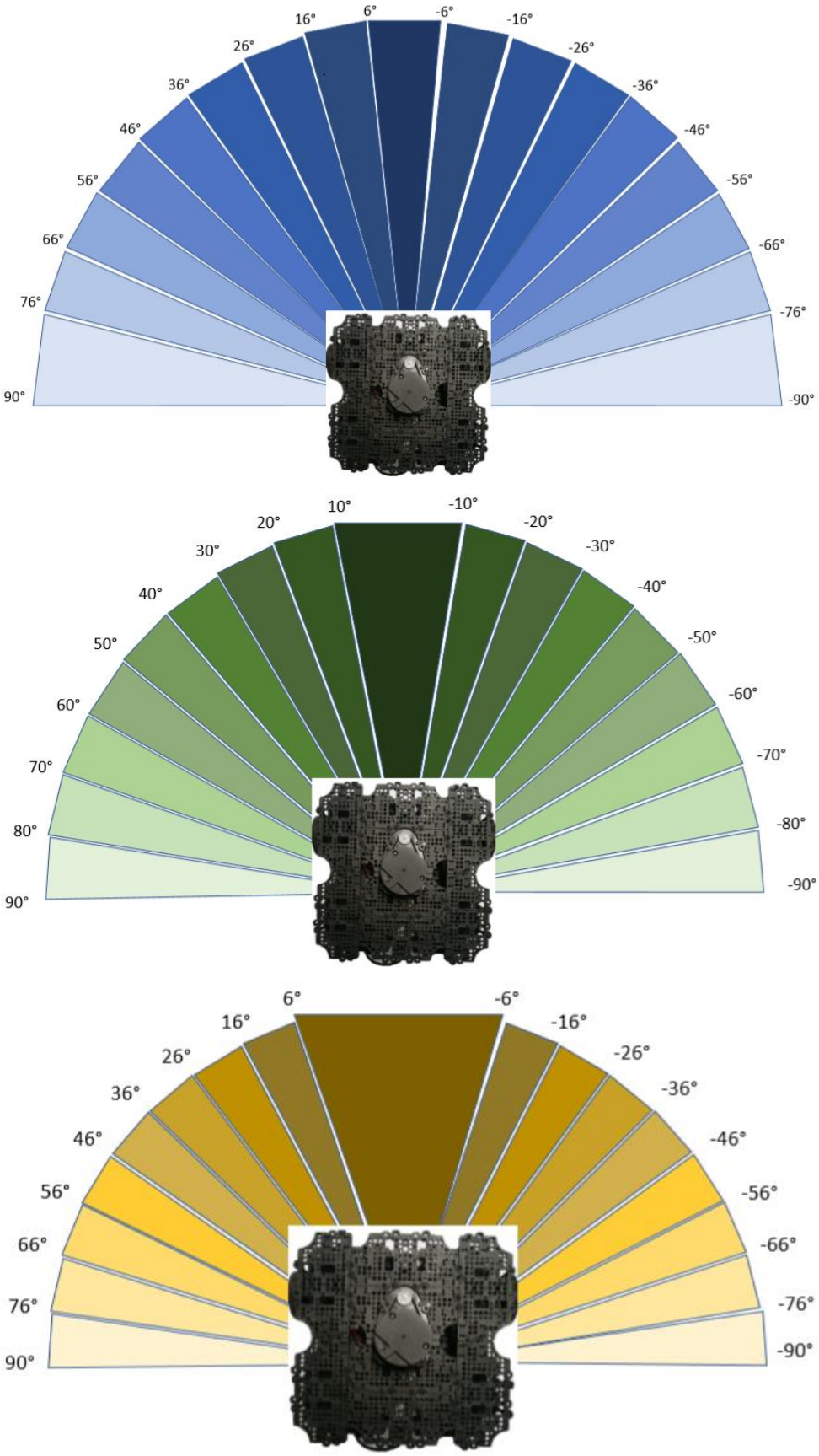


Figure 5.7 Representation of the three different area, from the farthest one (blue) to the nearest one (yellow), where are represented all the subarea in different colour, in which the darkest one are the first considered by the robot.

Instead, the angular one depends both on the main area selected and on the subset that is chosen, the more an angle with a light colour is chosen the more the angular speed will be high.

Finally, the algorithm also guarantees a safe area for the robot. This area was created because the robot could start or fall inside closed zones, where none of the previous movements is possible because the LiDAR sends information only about obstacles around 180° in front of the robot for all the areas.

Inside this safety area (Figure 5.8), two motions are possible (very similar to the previous algorithm “Follow Wall”), the robot can only proceed, very slowly straight on or turn right or left. While the orange area in front is obstacle-free the first motion is allowed, with a linear velocity equal to 0.07 m/s and an angular equal to 0 rad/s. The robot proceeds in this way until it sees objects closer to itself.

Instead, when this condition is no longer verified, it can only turn on right or left (angular velocity of ± 0.2 rad/s, linear velocity of 0 m/s) depending on where the obstacle is. This last information is given by the red area around the robot and this motion continues until the orange area is no longer free from collisions.

This algorithm greatly improves the first one, both in terms of analysis of the environment around the robot and in terms of possible movements.

The ‘Obstacle Avoidance Algorithm' generates 54 different compositions of angular and linear velocity. This means that a robot is able to move faster and to avoid obstacles more easily, performing curved trajectories. With this algorithm, the robot scans all the 180° space in front of it, instead of using the previous algorithm, considering only 80° degrees.

Being able to perform curved trajectories, the time required to map a generic area has been noticeably reduced. The time required to find a movement is low too, so we never have timing problems thanks to the simplicity of the algorithm.

However, the trajectories are not smoothed and furthermore, there are some constraints, such as the three zones or the different choices of the cones, which limit the algorithm capacities and do not make it a generalized algorithm.

These are the main reasons why a new algorithm has been written, which is inspired by the previous ideas, where some constraints are relaxed, and which leaves the robot the possibility to decide which is the best movement.

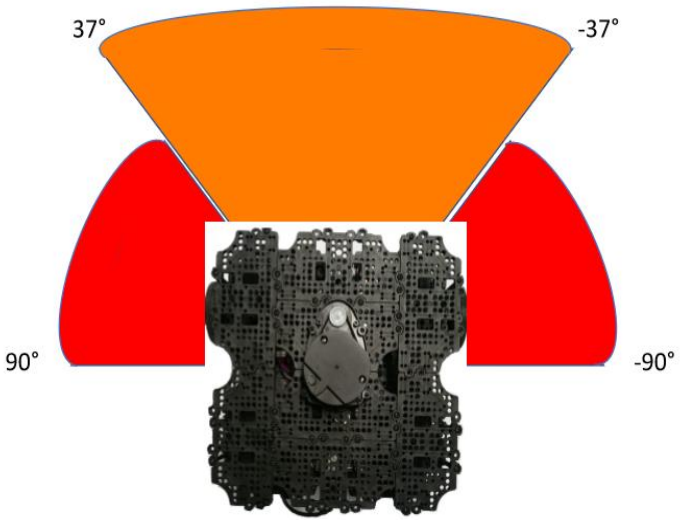


Figure 5.8 Safety area to avoid a collision

5.2.3 Autonomous Navigation with Map

This algorithm can be considered the main work of the thesis. It allows the robot to choose the best possible movement to avoid collisions, but as we will see, it is also able to create a map of the environment, without using any support tool, like RViz, which was previously used for a 2D representation of the environment.

The considerations to be made before going into the description of the algorithm are concerned the objective and starting conditions. Unlike other publications in literature whose objective is to start from a point A and arrive at a point B starting from initial conditions in which the map is partially or totally known (global navigation algorithm), the objective of the 'Autonomous Navigation' algorithm is to start from a completely unknown area (indoor) and in the shortest possible time map the whole unknown area, being able to avoid obstacle.

This algorithm is also written to be run on TurtleBot3 (Waffle or Burger), so the sensor used to detect obstacles is always the LiDAR.

The first task of the algorithm is to receive the LiDAR information, which is always sent with a frequency of 5 Hz, which contain for each of the 360° the distance of the respective identified obstacle.

To better understand the code and the various steps it is possible to follow the Flow Chart reported in Figure 5.9 or read the algorithm code in Appendix A.

The first point is the analysis of the data received from the LiDAR, this task is performed inside the *'start'* function. The function has the goal of creating a matrix, called *'maps'*, to which each corner is associated with the respective distance value in meters to which an obstacle corresponds and a vector, called *'distance_differ'*, in which all the distances of the LiDAR are sorted (eliminating the double measurements) in a decreasing order, from the farthest obstacle to the nearest one.

Of course, the LiDAR has some specifications, its range of action is limited, in case of the LiDAR mounted on the TurtleBot3, it is about 4.2 meters. When this threshold is exceeded, it means that the information returned to the sensor has too low power having travelled a long path, in this case the sensor returns a value equal to 0 meters; the function *'start'* has also the task of converting the measures of 0 meters, that corresponds to infinity, to a finite value, in this algorithm 5 meters are chosen.

The matrix and the vector created by the *'start'* function are passed to the *'direction'* function. *'Direction'* represents the most important function of the algorithm because it returns the angle and the distance at which the goal is located, which represents the direction in which the robot must move.

To reach the goal, starting from the first value inside the *'distance_differ'* vector, that means from the farthest distance capture by the LiDAR, the function tries to understand if the robot has free space in front of it at that direction.

Depending on the distance *'d'* at which the robot wants to move and, on the volume occupied by itself, the algorithm needs a certain number of consecutive angles whose value of distance from the obstacle must be greater than the distance at which the robot intends to move.

The number of angles necessary for a given distance is returned by the *'alpha'* function. The latter receives as input the distance at which the robot wants to move and the robot's dimensions and returns the number of degrees of an angle δ necessary to form a rope such that its length is greater than the width of the robot.

Concerning the TurtleBot3, the worst case of width is related to the Waffle, that has a dimension higher respect to the Burger; therefore, within the algorithm the thickness value of the robot was set at 50 cm slightly larger than the length of the Waffle diameter equal to 44 cm, to better avoid the collisions.

Once the algorithm knows the angle δ, starting from the direction in front of the robot it analyses a number of angles equal to $\delta/2$ on the right side and $\delta/2$ on the left side,

checking whether each degree has a value of distance to the obstacle (find by the LiDAR) higher than the distance d considered.

If this condition is not verified, the robot alternatively looks for the next 1° into the right and left side, until, if it does not find a direction, it scans all 180° degrees in front of the robot. In this case, when it finishes to scans all the 180° without finding a direction, it takes the second value of distance inside the *'distance_differ'* vector and the analysis restart from the calculation of the number of angles δ necessary with the new distance and continues with the control of all the 180°.

This method is repeated until either the robot finds a motion or all the distance values inside *'distance_vector'* are controlled, this second case means that the robot was placed inside a completely closed area like a hole, being blocked.

In the lucky case in which the robot is able to find a possible direction of motion (value of angle α and distance d), that correspond to the positive response "Yes" at the "Direction Found" question in the Flow Chart, the algorithm checks whether it can be also done at all the distances closer to the one found with the same angle α that has been chosen as valid.

This investigation is done by the *'control'* function, that is within the *'direction'* function, because in the case of negative response of the control, that means that the robot does not have enough space to reach the goal in all the possible path, the algorithm restarts looking for a new motion.

Finally, when this check is positive too, so when the algorithm found an angle α and a distance d that are good (in the Flow Chart it is represented by the answer “Yes” at the question “Obstacle found”), the ‘medium’ function tries to understand if there are other angles closer to the one found (α), which are free of collision considering the same distance d.

This analysis allows the robot to check if there are other free angles after the one found at the same distance d, in this way if they are found, the robot moves towards a medium angle inside the free space.

Let's analyse an example to better understand this point, consider that the algorithm finds the first possible path at a distance of d = 3 m and an angle α = 20 °. The *'medium'* function will check that at a distance of 3 m the robot can move even at 21° degrees, this procedure continues until a negative response is given.

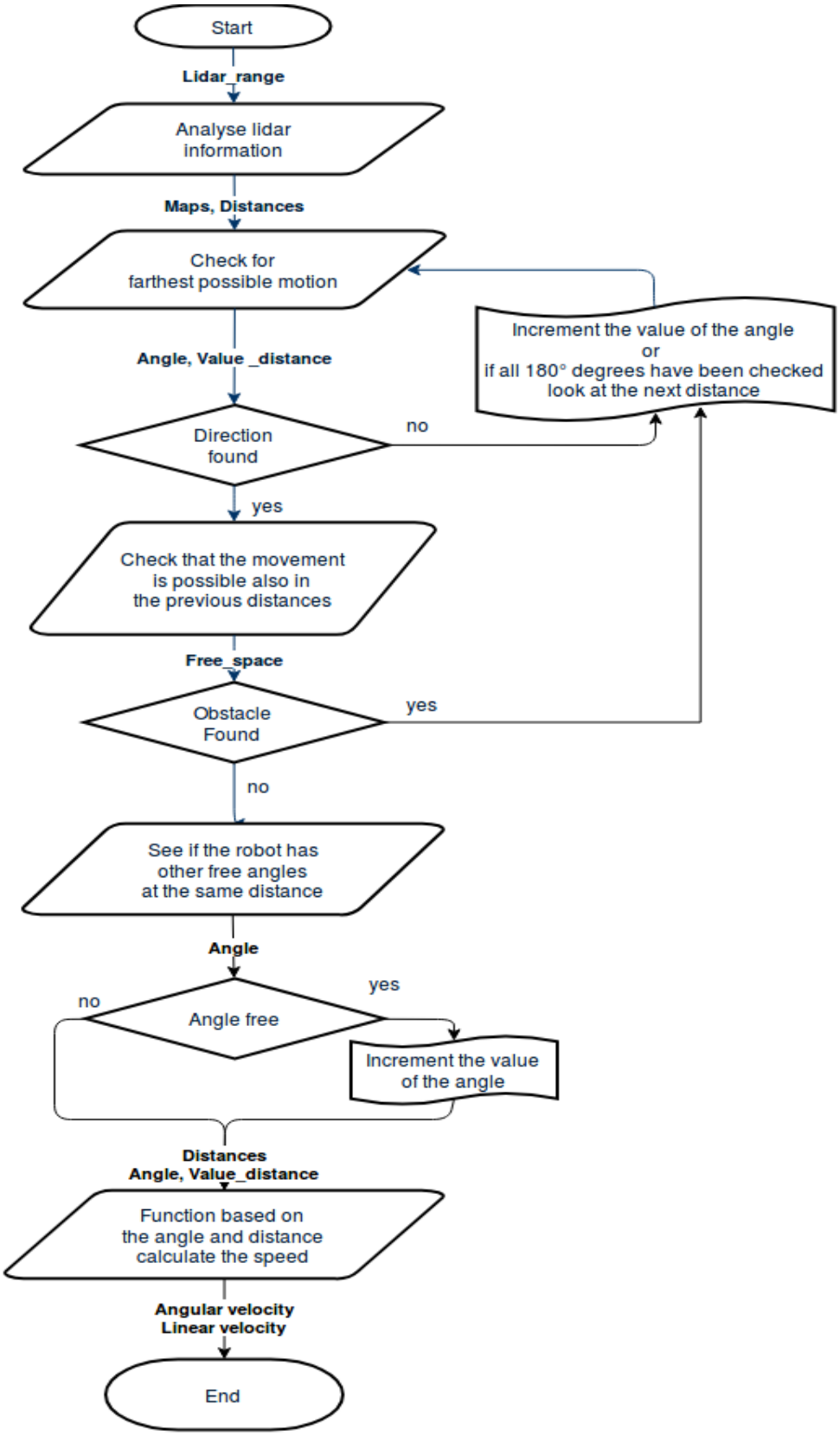


Figure 5.9 Flow Chart of the 'Autonomous Navigation' algorithm

In the hypothesis that γ = 30° is the last angle in which the robot can move at 3 meters, the result of the 'medium' function will give as a value $\beta = \frac{(\alpha + \gamma)}{2} = \frac{20+30}{2} = 25°$ degrees. The final values of distance d and angle β represent the values that are returned by the 'direction' function.

These values d and β are the ones used to give the linear and angular velocity to the robot; this is done inside the 'motion' function.

Before explaining how the linear and the angular velocities have been calculated in this algorithm, the distances travelled by the robot should be measured by computing via dead reckoning and then compensating the pose with inertial data or estimating translational speed and angular speed with an IMU sensor.

"ROS defines the pose as the combination of the robot's position (x, y, z) and orientation (i, j, k, w). The orientation is described by i, j, k, and w in quaternion form, whereas position is described by three vectors, such as x, y, and z." [31]

Considering the shape of the TurtleBot3 there are two parameters that are relevant: the distance D between the wheels and the radius r of them; the rotation speed of the left and right wheels (v_l, v_r) is given by:

$$v_l = \frac{(E_{l2} - E_{l1})}{T_{12}} \frac{\pi}{180} \text{ (rad/s)}$$

$$v_r = \frac{(E_{r2} - E_{r1})}{T_{12}} \frac{\pi}{180} \text{ (rad/s)}$$

Where T_{12} is the interval between two instants $(T_2 - T_1)$, E_{l2} *and* E_{l1} are the values given by the left encoder at the time $(T_2,\ T_1)$.

The velocities of the left and right wheel (V_l, V_r) can be calculated by knowing the radius r of the wheel:

$$V_l = v_l \cdot r \text{ (m/s)}$$

$$V_r = v_r \cdot r \text{ (m/s)}$$

Finally, from the left and right wheel velocity, it is possible to find the linear and angular velocity (v_t, w_t),

$$v_t = \frac{V_l + V_r}{2} \text{ (m/s)}$$

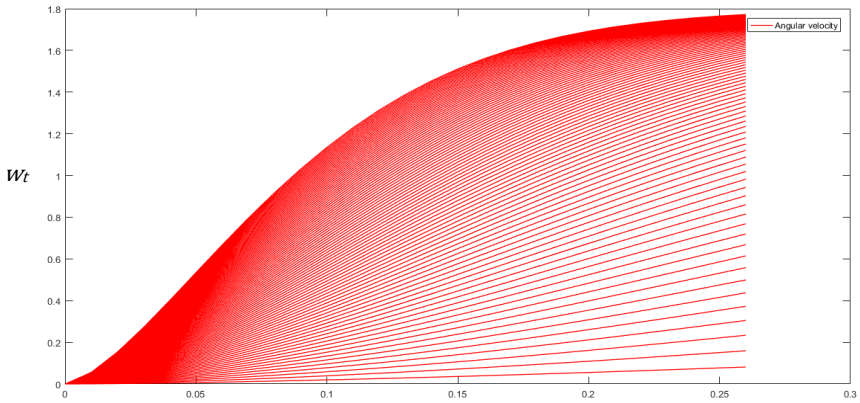
$$w_t = \frac{V_l - V_r}{D} \text{ (rad/s)}$$

v_t and w_t are the linear and angular velocity of the robot, these parameters are defined by the 'Autonomous navigation' algorithm based on the value of distance d and angle β that are returned by the 'direction' function.

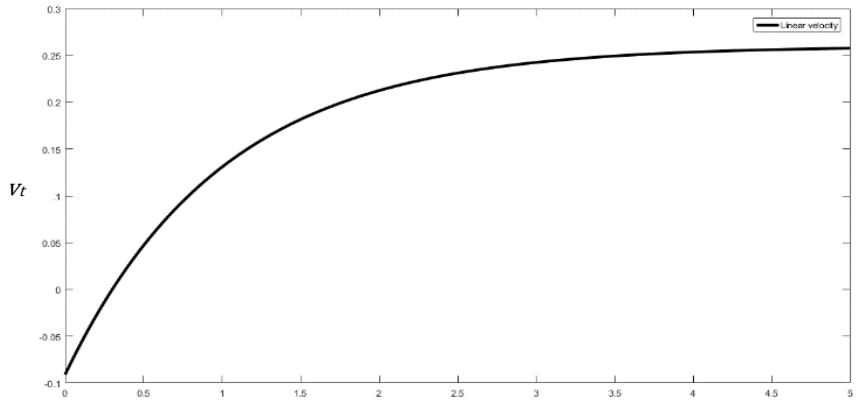
$$v_t = +\left(0.26 - 0.26 \cdot e^{-(d-0.3)}\right) \text{ (m/s)}$$

$$w_t = \pm\left(1.8 - 1.8 \cdot e^{-0.35 \cdot |\beta \cdot v_t{}^{1.5}|}\right) \text{ (rad/s)}$$

The two Figure 5.10 shows the variation of linear velocity (Sub-figure b) and of angular velocity (Sub-figure a), according to the value of distance d and of angle β.



Sub-figure a



Sub-figure b

Figure 5.10 The linear v_t (Sub-figure b) and the angular w_t (Sub-figure a) velocity applied to the robot on the y axis

For what concerns v_t, the more the distance d is high the more the linear speed increases with an exponential trend; this trend, for high values of distance, is asymptotic to the 0.26 m/s velocity, which represents the maximum linear speed at which TurtleBot3 can move.

A similar analysis can be done for the sub-figure a, for a high value of β and v_t its trend is asymptotic to 1.8 m/s, which represent the maximum angular speed of the robot. In addition, every single red line represents the trend of the angular velocity, given a specific value of the angle β, as the linear velocities change. The direction of the motion is given by the sign of the angle β, positive in the case of rotation to the left (counter clockwise), negative in case of rotation to the right (clockwise).

At this point, the description of the algorithm linked to the calculation of the variables necessary for the movement of the robot is finished, so the linear velocities and angles are calculated every time the LiDAR topic sends new information, that means every 0.2 seconds (5 Hz).

As already mentioned this algorithm is also able to generate a map of the area scanned by the robot. This part of the algorithm is managed within the '*slam*' function. The map is created by combining the LiDAR information and the robot's odometry data.

In the previous algorithms, the part related to the mapping was managed by the RViz tool, that uses the two-dimensional Occupancy Grid Map (OGM). The map obtained using RViz, already shown in Figure 5.6, colours the area in different ways: white if it is free of collision, black if it is occupied by obstacles in which the robot cannot move, and grey if it is an unknown area.

The points in the map are represented using grayscale values which range from ‘0’ to ‘255’. To compute the right value, the algorithm of RViz uses the posterior probability of the Bayes’ theorem. This theorem calculates the occupancy probability, that means the probability that a point is an obstacle or not.

The occupancy probability is expressed with a variable that the closer it is to 1, the higher the probability that it is occupied, instead, the closer is to ‘0’, the less likely the point is occupied.

The message generated by the topic of the map (generated by RViz), when the SLAM is performed, is a matrix. With '0' it indicates the free area, with '100' an occupied area and '-1' is used to point out an unknown one. Each pixel of the map can be converted to 5 cm.

The map, that the ‘Autonomous navigation' algorithm tries to create, takes inspiration on the RViz algorithm but does not use the posterior probability of the Bayes’ theorem. Whenever the algorithm is launched, once it has finished the part linked to the movement, it merges the data of the LiDAR with the information generated by the ‘Odometry' topic, to generate a matrix.

The dimension of the matrix has been defined a priori and each pixel of the map obtained from the matrix can be transformed to 1 cm. Every time (each 0.2 second), it updates the map inserting a 1 on each point where the LiDAR has identified an obstacle, and a '0' where is free.

The advantages of this map are linked to its greater precision, to the simplicity with which it was written, which makes it easy to understand and modify. This last advantage was used to represent the entire path of the robot and the shape of it (when it travels a distance greater than 2 meters).

However, the RViz algorithm has a greater complexity and it is able to assign the occupation probability of a cell with greater precision, thanks to which it avoids errors, that sometimes are present in the 'Autonomous navigation' algorithm.

Errors such as the excessive or insufficient thickness of the objects in the image compared to the real one, which is due to an excessive density of inserted points or a wrong evaluation; the presence of obstacles (random black points) outside the mapped area and errors due to the high angular velocity that is applied during some rotations.

The main advantages of the 'Autonomous Navigation' algorithm are the great improvement on the kind of motion that includes up to 980 movements due to the different combination of the angular and the linear velocity. The robot is able, to perform curved trajectories and the LiDAR scans 180° in front of the robot to understand the correct direction and all the 360° are scanned when the algorithm perform the control of the motion.

Another advantage is related to the time to map the previously considered space, the floor of the LIM department; now it is reduced to 4-5 minutes thanks to the new capabilities of the robot. The last improvement is the autonomous creation of the map that as already shown it has advantages and disadvantaged with respect to the one created by RViz.

6 Conclusion

6.1 Experimental Results

This section presents the obtained results of the proposed project evaluated on different unknown indoor environments and it finally describes the advantages and the limitation of the algorithms, proposing reasonable improvement to the Obstacle Avoidance Algorithm.

Now, the outcome of some tests performed at the LIM laboratory is shown, with the starting condition of considering an unknown indoor area, the used algorithm is 'Autonomous Navigation' on TurtleBot3 (Waffle), so the sensor used to detect and avoid the obstacles is always the LiDAR.

The first simulation scenario is a Circle. In Figure 6.1, on the left the real environment can be seen and on the right the map created by the 'Autonomous Navigation' algorithm is shown. The trajectory obtained by the robot is smooth and precise (highlighted by the blue line), the starting point is depicted by the red shape of the robot that proceeds clockwise around the waste basket.

As already said the map is updated every 0.2 seconds, with insertion of a 1 on each point where the LiDAR has identified an obstacle, and a '0' where free space is present. Even if the map of the circle is comparable to the real one, however, the left Figure highlights some troubles of this method with respect to the map created by RViz.

For example, the insufficient thickness of the basket in some sides so that it is not well defined or the presence of obstacles (random black points) outside the mapped area that can be caused by the high angular velocity that is applied during some rotations.

Figure 6.1 Circle

Figure 6.2 shows the floor of the LIM department. The area considered in this experiment is quite big, where the red position of the robot is updated every 2 meters, the blue line continues to represent the trajectory of the robot.

The map is always created using the algorithm inside “Autonomous Navigation”, in which the x and y axes represent the centimetres travel by the robot (around 5 meters on the x and 15 meters on the y).

This map can be compared with the one design by RViz (Figure 5.6), as is possible see the main difference is the thickness and precision of the wall that is well defined using the ROS tool, while in this case it presents some holes or a wrong concentration that is due to the lower complexity of the algorithm with respect to the one of RViz, that better work with higher levels of speed.

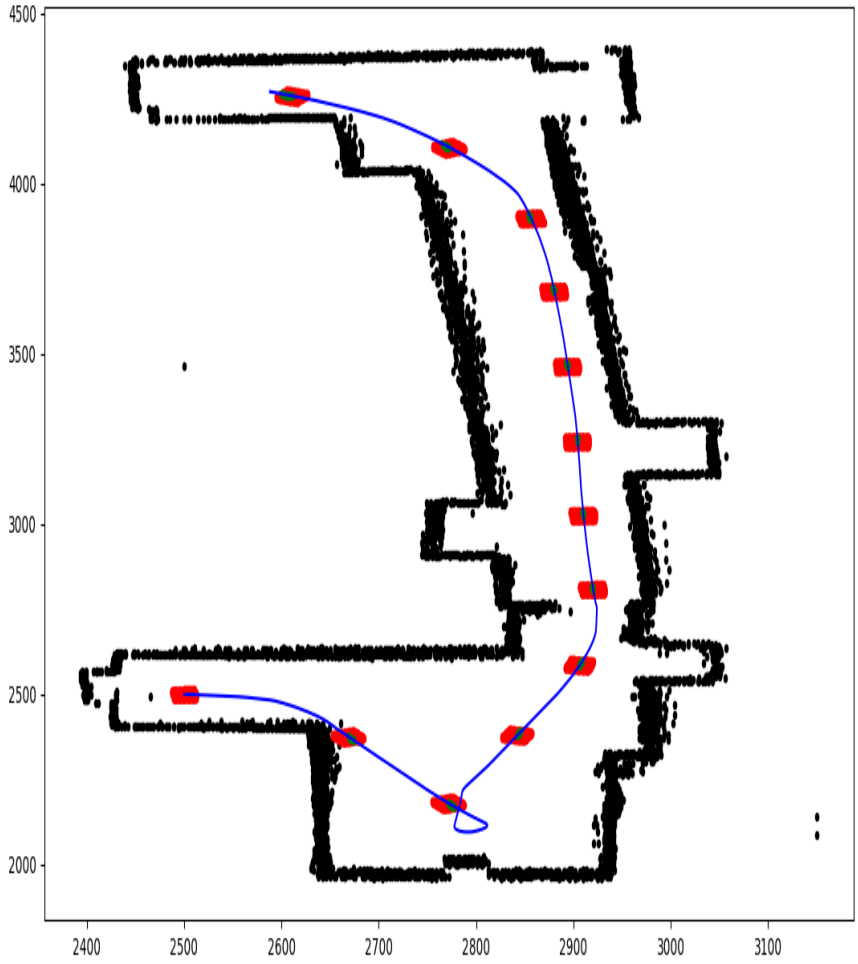


Figure 6.2 Floor of the LIM department

The third scenario in which the TurtleBot is running, tries to reproduce on an indoor environment an outdoor one, that is an agricultural field. The vineyard was replicated inside a room of the LIM department, dimension $4.5 \; x \; 7.5$ meters; where vine rows were created using polystyrene panels, dimensions $1 \; x \; 0.1$ meters and height 0.5 meters.

The goal is to create a map of the environment that could be represent a vineyard, where is possible recognize each vine row and the different agricultural land. The aim has been reached using the 'Autonomous Navigation' as Obstacle Avoidance algorithm, with few modifications with respect to the one explained before, because in this circumstance the map is created using the RViz tool.

The Waffle robot, using the LiDAR, could recognize the different vine rows, where the ones of the same parcel are separated from each other by 0.75 meters, in this way the robot is able to pass inside having considered its maximum size of 0.5 meters (Appendix A, Autonomous Navigation, line 31).

Figure 6.3 shows two images from the vineyard map. The image on the left is the one captured by RViz while the robot moves within the unknown environment and creates real-time a map. As already analysed, RViz shows the obstacles in black, the areas without collisions in white, the areas not yet explored in grey and finally the obstacles surrounding the robot captured at the instant in which the image was taken are highlighted in green.

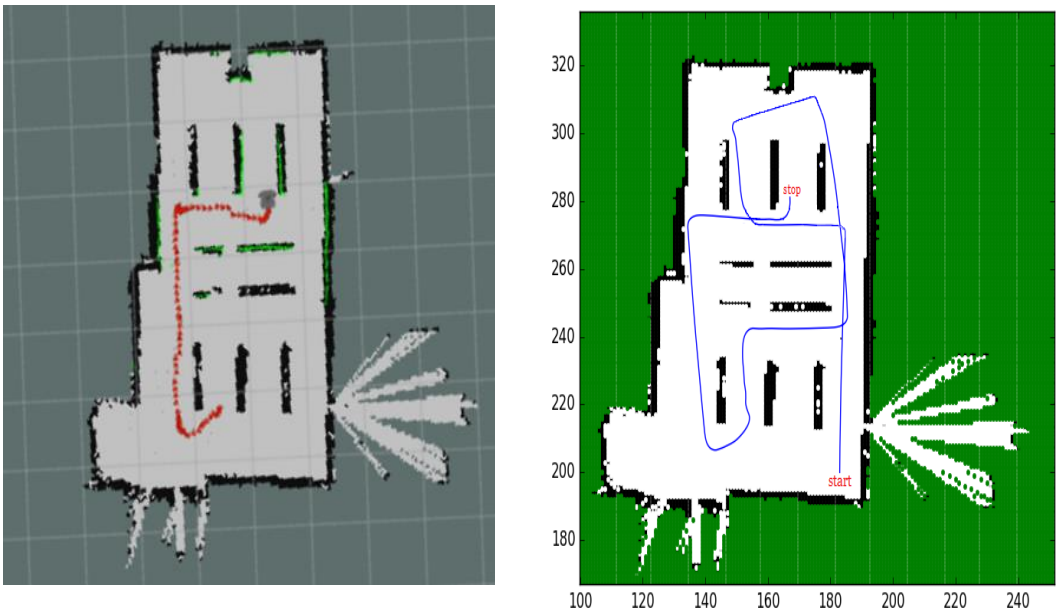


Figure 6.3 Simulation of the Vineyard

In the beam of light, at the bottom left of the image, it is not a mistake, but it is due to the presence of a slit in the obstacles that bounded the perimeter, so the rays of the Lidar, mapping this area, shone through the small hole in the wall.

The other image of Figure 6.3 is instead generated at the end of the simulation in the vineyard, using a script in python, in which the entire path executed by the robot, is represented from the starting point to the final one. It is possible to point out the presence of small errors of odometry, which are more evident when the robot performs curved trajectory and gradually spreads, decreasing the accuracy of the blue trace.

This experiment was done in collaboration with another master student, who worked on a parallel project, consisting of identifying, given a generic map of the agricultural environment in the form of a binary matrix, the parcels within it in order to generate a path plan for the robot able to cover all the environment with an optimal criterion.

Figure 6.4 defines the path inside the vine rows of each parcel, that the robot must follow, to examine each vine row, being able to minimize the distance travelled. The vineyard has been reproduced in multiple configuration, to test different scenarios of the obstacles.

Once the robot knows the trajectory to perform, that means the starting point S, the final point F and all the intermediate goals G_i to reach in which it must change the linear or the angular velocity, the motion can be begin.

This algorithm is called “Path Following”, the idea behind is very simple, knowing every time the starting position of the robot S and the sequence of goals G which allow a motion without collision; the robot performs firstly a rotation toward the next goal G_{i+1} using only the angular velocity, and subsequently, proceeds straight on toward the goal until it is get.

The procedure is repeated as far as the robot arrives at the final point F. To perform the trajectory the robot needs the distance and the orientation of each goals respect to a fixed reference system.

The last environment to analyse the performance and the behaviour of the “Autonomous Navigation” algorithm, is the replication of a maze without exit in which the robot is trapped, and it is forced to move. The maze is done using the polystyrene panels inside a real indoor environment, represented in Figure 6.4.

Figure 6.5 shows the evolution of the map created by RViz tool, and the path followed by the Waffle running the “Autonomous Navigation” algorithm. When the algorithm starts the robot does not know the environment it is going to explore, but through the

information coming in real time from the LiDAR it is able to navigate and simultaneously create the map.

Figure 6.4 Maze exploration

.

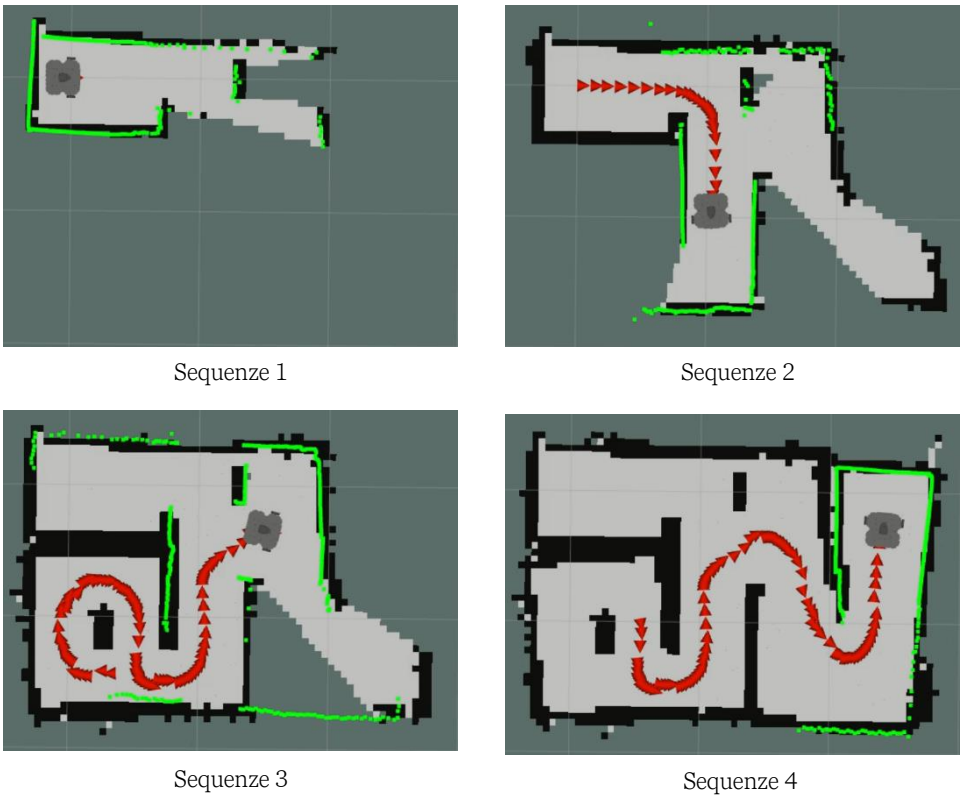


Figure 6.5 Simultaneous Localization and Mapping inside the Maze, showing the evolution of the map step by step

6.2 Future Work

In this thesis, we have shown a possible solution to the problem of both the navigation, applying Obstacle Avoidance algorithms, that allow a robot to move and perform in an unstructured and unknown indoor environment, and the realization of the map of the scanned area using RViz (the ROS tool) or an implemented method.

The final approach called "Autonomous Navigation" is based on the idea of combining different real time Obstacle Avoidance algorithms to reach a goal position (the farthest distance the robot can reach) in an unknown environment.

The mobile robot (Turtlebot3: Waffle and Burger), by analysing the information of the LiDAR, generates a free collision motion to move around the detected obstacles from its position towards the goal. No prior knowledge about the environment is assumed in this approach, which makes use only of onboard sensors to acquire information during the motion

Satisfactory results have been obtained regarding the problem of autonomous navigation of a mobile robot in unknown environments (as shown in previous chapter), but some improvements could be brought using, for example, different sensors, such as cameras, LiDAR 3D or ultrasonic sensors. Using the information from the on-board stereo camera, it would be possible to improve the navigation quality of the mobile robot, to enable 3D SLAM and navigation, or objects recognition.

A further improvement would also be necessary for the odometry of the robot, which, as analysed, get worse the more the distance of the robot increases. The use of GPS could be a solution, especially in open environments.

Finally, in a high-level planning technique that understands the world, where it is likely that robots will need to seek out very specific pieces of information. For example, in order to exit a building, the robot will be equipped with some learned knowledge in order to find an exit route. These techniques greatly improve the performances than a random search method.

In the near future, we would like to improve this approach for the implementation of an algorithm based on the idea of combining the global path planning with a real time Obstacle Avoidance algorithm in external environments such as agricultural lands.

7 Appendix A

Autonomous_Navigation.py

```
1  import time
2  import threading
3  import math
4  import numpy
5  import time as tm
6  import matplotlib.pyplot as mt
7  import rospy
8  import copy
9  from sensor_msgs.msg import LaserScan
10 from geometry_msgs.msg import Twist
11 from nav_msgs.msg import Odometry
12 from geometry_msgs.msg import PoseWithCovariance
13 from geometry_msgs.msg import PoseStamped
14 global pub, move, cnt, time, free_time, global_distance, global_mod, x, y, a,
       b, w, z, flag, computation, odometry, positionx, positiony, robotx, roboty
15 rospy.init_node('obstacle_avoidance',anonymous=True)
16 pub=rospy.Publisher('/cmd_vel',Twist,queue_size=10)
17 move=Twist()
18 flag = 0, time = 0, x = 0, y = 0, a = 0, b = 0, cnt = 0,
19 global_distance = 0, global_mod = 0, positionx = 0, positiony = 0, robotx = 0,
       roboty = 0
20 computation = []
21 odometryx = numpy.array([])
22 odometryy = numpy.array([])
23 odometryw = numpy.array([])
24 odometryz = numpy.array([])
25 map_matrix = numpy.full((5000, 5000), 0)
26 turtlebot = numpy.zeros((30, 30))
27 init = [2500, 2500]
28
29 def alpha(distance_differ, k):
30     chord = 0.5
31     d = distance_differ[k]
32     angle = 2 * 180 / math.pi * math.atan(chord/2/d)
33     return int(angle) + 1
34 def motion(mod, k, distance_differ, alpha):
35     global flag
36     distance = float(distance_differ[k])
37     linear = round(0.26-0.26*math.exp(-1*(distance-0.3)),2)
38     if linear > 0.04:
39         flag = 0
40         angular=numpy.sign(mod)*round(1.8-1.8*math.exp(-
       0.35*abs(mod*math.pow(linear,1.5) )) ,2)
41     elif flag == 0:
42         flag = 1
43         if mod != 0:
44             angular = numpy.sign(mod) * 0.4
```

```
45        else:
46            angular = 0.4
47    return linear, angular
48
49 def start2(msg, x, y):
50    angle = range(x,360) + range(0,y)
51    maps = {}
52    for i in angle:
53       if round(float(msg.ranges[i]),2) == 0:
54          value = float(5)
55       else:
56          value = round(float(msg.ranges[i]),2)
57       maps[i] = value
58    map_distances = sorted(maps.values(), reverse = True)
59    return maps
60
61 def start(msg, x, y):
62    angle = range(x,360) + range(0,y)
63    maps = {}
64    maps_copy = {}
65    for i in angle:
66       if round(float(msg.ranges[i]),2) == 0:
67          value = float(5)
68       else:
69          value = round(float(msg.ranges[i]),2)
70       maps[i] = value
71    map_distances = sorted(maps.values(), reverse = True)
72    map_angles = []
73    distance_differ = []
74    maps_copy = maps.copy()
75    for j in map_distances:
76       for i in range(len(maps_copy)):
77          if maps_copy.values()[i] == j:
78             map_angles.append(maps_copy.keys()[i])
79             maps_copy[maps.keys()[i]] = -1
80             break
81    max_map = [map_angles, map_distances]
82    for i in range(len(map_distances)):
83       j = 0
84       if i != 180:
85          if map_distances[i] != map_distances[i+1]:
86             distance_differ.append(map_distances[i])
87          j += 1
88       elif i == 180 and map_distances[i] != map_distances[i-1]:
89          distance_differ.append(map_distances[i])
90    return maps, max_map, distance_differ
91
92 def direction (maps, distance_differ, maps2):
93    global global_distance, global_mod
94    angle = range(270,360) + range(0,91)
95    mod = 0, k = 0, point = 0, beta = 0
96    clock = True
```

```
97    while (clock):
98       selection = []
99       p_start = len(angle)/2 + mod
100      point = alpha(distance_differ, k)
101      for i in range(point):
102         p = p_start - point/2 + i
103         selection.append(angle[p])
104      value = distance_differ[k]
105      flag = 0
106      for j in selection:
107         if maps[j] >= value:
108            flag = 1
109         else:
110            flag = 0
111            break
112      if flag == 0:
113         if mod < 0:
114            mod = abs(mod)
115         else:
116            mod = -mod -1
117      else:
118         controller = control(maps2, distance_differ, mod, k)
119         if controller == 1:
120            if mod == 0:
121               counter, controller = medium (mod, maps, maps2, selection, valu
   e, distance_differ, k)
122               beta = mod + counter
123               clock = False
124               break
125            else:
126               counter, controller = medium (mod, maps,maps2, selection, value
   , distance_differ, k)
127               if mod > 0:
128                  beta = mod + round(counter / 2)
129               else:
130                  beta = mod - round(counter / 2)
131               if abs(global_mod-beta) <= ((abs(global_distance-
   value)+0.04)*500):
132                  clock = False
133                  break
134               else:
135                  if mod < 0:
136                     mod = abs(mod)
137                  else:
138                     mod = -mod -1
139         else:
140            if mod < 0:
141               mod = abs(mod)
142            else:
143               mod = -mod -1
144      if mod >= 90 - point/2:
145         k += 1
```

```
146         mod = 0
147       if k >= len(distance_differ):
148         k = 'ERROR'
149         clock = False
150     return beta, k, alpha, value
151 
152 def control (maps2, distance_differ, mod, k):
153     angle = range(181,360) + range(0,181)
154     point = 0
155     z = len(distance_differ)-1
156     bandiera = True
157     free_space = 0
158     while (bandiera):
159       space_angle = []
160       p_start = len(angle)/2 + mod
161       point = alpha(distance_differ, z)
162       for i in range(point):
163         p = p_start - point/2 + i
164         space_angle.append(angle[p])
165       value = distance_differ[z]
166       for j in space_angle:
167         if maps2[j] >= value:
168           free_space = 1
169         else:
170           free_space = 0
171           bandiera = False
172           break
173       if free_space == 1:
174         z += -1
175       if z == k:
176         bandiera = False
177     return free_space
178 
179 def medium (mod, maps,maps2, selection, value, distance_differ, k):
180     counter = 0, g = 0, index = 0, controller = 0
181     flag = True, flag1 = True, flag2 = True
182     if mod != 0:
183       while(flag):
184         if mod > 0:
185           g += 2
186           index = selection[-1]+g
187         elif mod < 0:
188           g -= 2
189           index = selection[0]+g
190         prova = tm.time()
191         if maps.has_key(index):
192           if maps[index] < value:
193             flag = False
194           else:
195             controller = control(maps2, distance_differ, mod+g, k)
196             if controller == 1:
197               counter +=2
```

```
198                 else:
199                     flag = False
200             else:
201                 flag = False
202     elif mod == 0:
203         while(flag1):
204             g += 2
205             index = selection[-1]+g
206             if maps.has_key(index) :
207                 if maps[index] < value:
208                     flag1 = False
209                 else:
210                     controller = control(maps2, distance_differ, mod+g, k)
211                     if controller == 1:
212                         counter +=2
213                     else:
214                         flag1 = False
215             else:
216                 flag1 = False
217         g = 0
218         while(flag2):
219             g -= 2
220             index = selection[0]+g
221             if maps.has_key(index):
222                 if maps[index] < value  :
223                     flag2 = False
224                 else:
225                     controller = control(maps2, distance_differ, mod+g, k)
226                     if controller == 1:
227                         counter -=2
228                     else:
229                         flag2 = False
230             else:
231                 flag2 = False
232     return counter, controller
233
234 def slam(map_matrix, maps, init, odometryx, odometryy, odometryw):
235     global time, positionx, positiony, robotx, roboty, obstacle
236     index = init
237     theta = 0, space = 0, robot = 0, n = 2
238     theta = odometryw[-1], x = odometryx[-1], y = odometryy[-1]
239     for i in range(len(maps)/n):
240         i = i*n
241         maps_value = maps.values()[i]
242         if maps_value > 0 and maps_value < 5 and time % 10 == 0:
243             index = copy.copy(init)
244             index[0] += int(x*100 + maps_value*100*math.cos((theta+i)*math.pi
    /180))
245             index[1] += int(y*100 + maps_value*100*math.sin((theta+i)*math.pi
    /180))
246             if map_matrix[index[0]][index[1]] != 2 and
    map_matrix[index[0]][index[1]] != 3:
```

```
247             map_matrix[index[0]][index[1]] = 1
248             print "value %d" %(int(maps_value*100))
249         for j in range (1,int (maps_value*100-30) ):
250             index = copy.copy(init)
251             index[0] += int(x*100 + (maps_value*100-
    j)*math.cos((theta+i)*math.pi/180))
252             index[1] += int(y*100 + (maps_value*100-
    j)*math.sin((theta+i)*math.pi/180))
253             if map_matrix[index[0]][index[1]] != 2 and
    map_matrix[index[0]][index[1]] != 3:
254                 map_matrix[index[0]][index[1]] = 0
255     space = math.sqrt( (x-positionx)*(x-positionx) + (y-positiony)*(y-
    positiony) )
256     robot = math.sqrt( (x-robotx)*(x-robotx) + (y-roboty)*(y-roboty) )
257     if time == 1 or robot > 0.3:
258         robotx = odometryx[-1]
259         roboty = odometryy[-1]
260         if space > 2:
261             positionx = odometryx[-1]
262             positiony = odometryy[-1]
263         for i in range(13):
264             for j in range(13):
265                 index = copy.copy(init)
266                 index[0]+=int(x*100+i*math.cos((theta)*math.pi/180)-
    j*math.sin((theta)*math.pi/180))
267                 index[1]+=int(y*100 + i*math.sin((theta)*math.pi/180)+j*math.co
    s((theta)*math.pi/180))
268                 if j == 0 and i >= 0:
269                     if space > 2 or time == 1:
270                         map_matrix[index[0]][index[1]] = 3
271                     elif map_matrix[index[0]][index[1]] != 2 and
    map_matrix[index[0]][index[1]] != 3 :
272                         map_matrix[index[0]][index[1]] = 0
273                 else:
274                     if space > 2 or time == 1:
275                         map_matrix[index[0]][index[1]] = 2
276                     elif map_matrix[index[0]][index[1]] != 2 and
    map_matrix[index[0]][index[1]] != 3 :
277                         map_matrix[index[0]][index[1]] = 0
278                 if j > 0:
279                     index = copy.copy(init)
280                     index[0]+=int(x*100+i*math.cos((theta)*math.pi/180)+j*math.s
    in((theta)* math.pi/180))
281                     index[1]+=int(y*100+i*math.sin((theta)*math.pi/180)-
    j*math.cos((theta)* math.pi/180))
282                     if space > 2 or time == 1:
283                         map_matrix[index[0]][index[1]] = 2
284                     elif map_matrix[index[0]][index[1]] != 2 and
    map_matrix[index[0]][index[1]] != 3 :
285                         map_matrix[index[0]][index[1]] = 0
286                 if i > 0:
287                     index = copy.copy(init)
```

```
288             index[0]+=int(x*100-i*math.cos((theta)*math.pi/180)-
    j*math.sin((theta)* math.pi/180))
289             index[1]+=int(y*100-
    i*math.sin((theta)*math.pi/180)+j*math.cos((theta)* math.pi/180))
290             if space > 2 or time == 1:
291                 map_matrix[index[0]][index[1]] = 2
292             elif map_matrix[index[0]][index[1]] != 2 and
    map_matrix[index[0]][index[1]] != 3 :
293                 map_matrix[index[0]][index[1]] = 0
294             index = copy.copy(init)
295             index[0]+=int(x*100-
    i*math.cos((theta)*math.pi/180)+j*math.sin((theta)* math.pi/180))
296             index[1]+=int(y*100-i*math.sin((theta)*math.pi/180)-
    j*math.cos((theta)* math.pi/180))
297             if space > 2 or time == 1:
298                 map_matrix[index[0]][index[1]] = 2
299             elif map_matrix[index[0]][index[1]] != 2 and
    map_matrix[index[0]][index[1]] != 3 :
300                 map_matrix[index[0]][index[1]] = 0
301     return map_matrix
302
303 def LiDAR(msg):
304     global time, global_distance, global_mod, computation, odometryx,
    odometryy, odometryw, odometryz, x, y, w, z, a ,b, map_matrix, init
305     if cnt == 1:
306         time +=1
307         print '*'*5 + 'Time' + '*'*5
308         print time
309         if time == 1:
310             a = x
311             b = y
312         maps, max_map, distance_differ = start(msg, 270, 91)
313         maps2 = start2(msg, 181, 181)
314         beta, k, alpha, value = direction(maps, distance_differ, maps2)
315         linear, angular = motion(beta, k, distance_differ, alpha)
316         global_distance = value
317         global_mod = numpy.sign(beta)*int(abs(beta)-
    angular*0.2*180/math.pi)
318         move.linear.x = linear
319         move.angular.z = angular
320         print '*'*5 + " The robot moves at %.2f meters, rotating for
    %.2f degrees " %(value, beta) +  '*'*5
321         print '*'*5 + " Linear velocity = %.2f m/s, Angular velocity = %.2f rad/s "
    %(linear, angular)+ '*'*5
322         w = numpy.sign(z)*round(float(2*math.acos(w)*180/math.pi), 2)
323         print '*'*5 + " Odometry x = %.2f m, y = %.2f m, theta = %.2f degree " %(
    x-a, y-b, w)+'*'*5+"\n"
324         odometryx = numpy.concatenate((odometryx, [x-a]))
325         odometryy = numpy.concatenate((odometryy, [y-b]))
326         odometryw = numpy.concatenate((odometryw, [w] ))
327         odometryz = numpy.concatenate((odometryz, [z]))
```

```
328         map_matrix = slam(map_matrix, maps2, init, odometryx, odometryy,
        odometryw)
329     elif cnt==2:
330         move.linear.x=0.0
331         move.angular.z=0.0
332         pub.publish(move)
333         X = []
334         Y = []
335         Green_X = []
336         Green_Y = []
337         Red_X = []
338         Red_Y = []
339         file_m = open("map.txt", "w")
340         for i in range(len(map_matrix)):
341             file_m.write(map_matrix[i])
342             for j in range(len(map_matrix)):
343                 if map_matrix[i][j] == 1:
344                     X.append(i)
345                     Y.append(j)
346                 elif map_matrix[i][j] == 3:
347                     Green_X.append(i)
348                     Green_Y.append(j)
349                 elif map_matrix[i][j] == 2:
350                     Red_X.append(i)
351                     Red_Y.append(j)
352         mt.plot(X, Y, 'k.')
353         mt.plot(Red_X, Red_Y, 'r.')
354         mt.plot(Green_X, Green_Y, 'g.')
355         mt.plot(odometryx*100+2500, odometryy*100+2500 ,'b-')
356         file_m.close()
357         mt.show()
358     pub.publish(move)
359
360 def Position(msg):
361     global x,y,z,w
362     x = msg.pose.pose.position.x
363     y = msg.pose.pose.position.y
364     w = msg.pose.pose.orientation.w
365     z = msg.pose.pose.orientation.z
366
367 class Publisher(threading.Thread):
368     def __init__(self):
369         threading.Thread.__init__(self)
370     def run(self):
371         osub=rospy.Subscriber('/scan',LaserScan,LiDAR)
372         sub1 =rospy.Subscriber("/odom",Odometry, Position)
373         rospy.spin()
374
375 if __name__=='__main__':
376     p=Publisher()
377     p.start()
378     while True:
```

```
379     gostop = raw_input("Press 'g'-> to Start the navigation / 's'-
    > to Stop the robot motion:")
380     if gostop=='g':
381         cnt = 1
382     elif gostop=='s':
383         cnt = 2
```

Bibliography

[1] YoonSeok Pyo, HanCheol Cho,RyuWoon Jung, TaeHoon Lim, *ROS Robot Programming*, Dec 22, 2017, Published by ROBOTIS Co.,Ltd .

[2] Morgan Quigley, Brian Gerkey, and William D. Smart, *Programming Robots with ROS,* December 2015: First Edition, Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

[3] Masoud Nosrati * Ronak Karimi Hojat Allah Hasanvand, *Investigation of the * (Star) Search Algorithms: Characteristics, Methods and Approaches*, Vol (2), April 2012.

[4] Nebot E., Bailey T., Guivant J., *Navigation Algorithms for Autonomous Machines in Off-Road Applications*, The University of Sydney, NSW 2006

[5] A. Oualid Djekoune, Karim Achour and Redouane Toumi, *A Sensor Based Navigation Algorithm for a Mobile Robot using the DVFF Approach*, International Journal of Advanced Robotic Systems, Vol. 6, No. 2 (2009)

[6] Bruno Siciliano, Oussama Khatib, Eds., *Springer Handbook of robotics*, Springer-Verlag Berlin Heidelberg2016

[7] Chuang Ruan, Jianping Luo, Yu Wu, *Map navigation system based on optimal dijkstra algorithm*, Proceedings ojCCIS2014

[8] Javier Minguez, Associate Member, IEEE, and Luis Montano, Member, IEEE, *Nearness Diagram (ND) Navigation: Collision Avoidance in Troublesome Scenarios*, ieee transactions on robotics and automation, vol. 20, no. 1, february 2004

[9] Takahashi O, Schilling RJ (1989), *Motion Planning in a Plane Using Generalized Voronoi Diagrams*. IEEE Robotics and Automation.

[10] Bhattacharya P, Gavrilova ML (2008), *Roadmap-Based Path Planning-Using the Voronoi Diagram for a Clearance-Based Shortest Path*. IEEE Robotics and Automation.

[11] Gomez EJ, Martinez Santa F, Martinez Sarmiento FHA (2013), *Comparative Study of Geometric Path Planning Methods for a Mobile Robot: Potential Field and Voronoi Diagrams*. In IEEE International Congress of Engineering Mechatronic and Automation (CIIMA), Colombia.

[12] Abiyev R, Ibrahim D, Erin B (2010), *Navigation of Mobile Robots in the Presence of Obstacles*. Advances in Engineering Software.

[13] Soltani AR, Tawfik H, Goulermas JY, Fernando T (2002), *Path Planning in Construction Sites: Performance Evaluation of the Dijkstra, A∗, and GA Search Algorithms*. ELSEVIER Advanced Engineering Informatics.

[14] Masehian E, Amin-Naseri MR (2004), *A Voronoi Diagram–Visibility Graph–Potential Field Compound Algorithm for Robot Path Planning*. Journal of Robotic System 21.

[15] Weigl M, Siemiaatkkowska B, Sikorski KA, Borkowski A (1993), *Grid-Based Mapping for Autonomous Mobile Robot*. ELSEVIER Robotics and Autonomous Systems.

[16] Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. IEEE Transactions on Systems Science and Cybernetics SSC4. doi:10.1109/TSSC.1968.300136.

[17] Jump up Doran, J. E, Michie, D. (1966-09-20). *Experiments with the Graph Traverser program*. doi:10.1098/rspa.1966.0205. ISSN 0080-4630.

[18] https://en.wikipedia.org/wiki/Embedded_system

[19] https://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping

[20] http://www.ros.org/

[21] http://brian.gerkey.org/

[22] http://robots.ros.org/

[23] https://en.wikipedia.org/wiki/Time-of-flight_camera

[24] https://en.wikipedia.org/wiki/Structured-light_3D_scanner

[25] https://en.wikipedia.org/wiki/Range_imaging

[26] Stenz, A. *Optimal and efficient path planning for partially-known environments*, Proceedings of IEEE International Conference on Robotics and Automation, ICRA, San Diego, May 1994, CA, USA.

[27] Stentz, A. *The Focussed D* Algorithm for Real-Time Replanning*, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), Montréal, August 20-25, 1995, Québec, Canada.

[28] E. W. Dijkstra, *A note on two problems in connexion with graphs*, Numerische Mathematik, vol. 1, 1959.

[29] P.E.Hart, N.J.Nilsson, and B.Raphael, *A formal basis for the heuristic determination of minimum cost paths*, IEEE Transactions on Systems Science and Cybernetics, vol. SSC-4(2), 1968.

[30] http://wiki.ros.org/gmapping

[31] http://docs.ros.org/api/geometry_msgs/html/msg/Pose.html

Acknowledgment

I would first like to thank my thesis supervisor Prof. Marcello Chiaberge of Politecnico di Torino, always ready to help me whenever I ran into a trouble or had a question about my research.

I would also like to thank all members of the LIM department involved in this project and the university fellows, for their support that was essential but also for creating a very conducive work environment.

Then, I must express all my gratitude to my family, for being always present but never pressing, rejoicing with me of my goal achievement. Thanks to my sister that is my point of reference, that taught me the determination and the strength to achieve the goals.

My special thanks go to my friends, you supported and encouraged me during these years and you were the distraction to escape from the studies.

Finally, thanks to Martina for being always by my side. Thanks for being everything I need, because spending time together meant forgetting about all problems. For being always present, filling my life with love.