POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Simulation and Testing of Navigation for an Autonomous Mobile Robot

Tesi di Laurea Magistrale in
Automation and Control Engineering - Ingegneria dell'Automazione

Author: **Giovanni Porcellato**

Student ID: 10745779
Advisor: Prof. Matteo Matteucci
Co-advisors: Simone Mentasti
Academic Year: 2021-22

Abstract

Keywords: here, the keywords, of your thesis

Abstract in lingua italiana

Qui va l'Abstract in lingua italiana della tesi seguito dalla lista di parole chiave.

Parole chiave: qui, vanno, le parole chiave, della tesi

Contents

1 | Introduction

Now far from being science fiction, robots are little by little entering our world. The tendency to replace humans with machines has existed for centuries, but it is only during the 20th century that technology has enabled us to achieve astonishing results.
Hence, the robots we know today come in different forms: from robots with human form, to mechanical arms, to robotic heads. In each case, there was extensive research and development before arriving at the results we can enjoy today. Given the initial instability of robotic systems and the danger and cost, the creation of these devices went hand in hand with the development of special simulators.
It is indeed very inconvenient to repeat tests on hardware, for the reasons mentioned above. It is precisely in this circumstance that the need arises to simulate in a virtual environment the movements and actions that the robot would then have to repeat in the real world. It is therefore necessary to set up a testing procedure that has precise specifications and is reliable and repeatable over time.
In particular, the simulated behaviour of the robot must be consistent with the measurements and physical specifications of the robot in order to have a reliable simulation.
From an automation engineering point of view, the main objective is to eliminate human intervention even on the testing procedure, in order to have a semi-autonomous testing software, which returns data that can be easily analysed both for internal development and for supply to third parties.
The operator must in any case monitor the progress of the test in order to obtain useful information on anomalous behaviour.

This work proposes a tool for testing the navigation of a humanoid robot in a simulated environment, offering a faithful and secure solution and obviating the problematic testing on hardware.

In addition, lateral solutions are presented to problems that arose in the testing phase, such as the denoising of the image received from the robot‘s video terminal.

1.1. Oversonic Robotics

This project has been realized as an application for Oversonic Robotics.
Oversonic Robotics is an Italian robotics start-up founded in 2020 in Besana in Brianza (MB) that wants to provide companies with intelligent systems that can assist humans in the most psychologically and physically demanding and exhausting jobs, thus enabling people to devote themselves to tasks that more effectively enhance intelligence. Their first project, Robee (Figure 1.1), is an autonomous humanoid robot, 1.75 m tall with a weight of around 75 Kg. He operationally replicates the mechanical structure of the human body, with 36 movable joints and a complete set of sensors. This enables him to see and navigate the surrounding space. The interaction is managed via a voice interface, capable of carrying out a normal conversation. He has arms and hands that allow him to cover all kinds of tasks. These include the simplest gestures such as pointing or counting, to a solid grip for handling objects. For greater awareness of his surroundings and for better communication with operators, Robee is equipped with a variety of cameras that use facial recognition and object detection.

Figure 1.1: "Robee", the humanoid robot developed by Oversonic Robotics Repubblica [19]

1.2. Contribution

The main contributions of this work are as follows:

- Development of a simulated environment to test the robot's new navigation features.
- Development of a module to test the navigation performance of the robot.
- Implementation of a point-cloud filter in order to improve obstacle handling and consequently navigation performance in critical brightness scenarios

1.3. Document Structure

The content is divided into two large sections: the first refers to the **Theoretical and Technological Background**, where the state of the art and the technologies used are introduced, in order to fully understand the system; the second one contains the **Contribution** of the thesis and reports the architecture of the system, its implementation and results. 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 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 first introduces robots providing a brief historical overview. It is then described Oversonic robot, Robee, describing its components, both software and hardware.
- 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 this thesis. It introduces first Gazebo simulator, afterwards it goes step by step through the building of the sim. robot and environment. It then encompasses the measurement module implemented. Results and issues are presented.
- Chapter 6 introduces point-cloud. This section proposes to solve an issue encountered while testing on the physical robot.

Theoretical background

From now on some basic knowledge needed to comprehend the reasoning of this work's study object is described and explained. This section consists of:

- **Chapter 2:** ROS
- **Chapter 3:** Robot
- **Chapter 4:** Navigation Stack

2 | ROS

2.1. Introduction

In the field of robotics, platforms are of increasing importance. A platform is divided into a software platform and a hardware platform. There are a variety of features that make up robot software platforms, including low-level device control, SLAM (Simultaneous Localization and Mapping), navigation, manipulation, recognition of objects or people, sensing and package management, debugging and development tools, which are mostly used in the industrial sector where robot software platforms are currently primarily employed. 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. The design of the ROS platform allows the use of code and information shared by other programmers, which means you do not have to write all code in order to move the robots.

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." [14]
Robots are computer-controlled electromechanical devices. First dedicated robot programming languages arose in the 1970's. There were robot-centric data types and some robot function libraries. They did not allow neither hardware abstractio, nor multi-robot interaction nor integrated simulation. There did not exist code reuse or standardization. The efforts to build robot programming system continued through 80's, 90's and especially in the 2000's were there was a high push to standardize robot components, their interfaces and basic functions.
Hence, ROS as it is known today was initially developed in 2007 at the Standford Artificial Intelligence Laboratory. Since 2013 it is managed by OSRF and nowadays it is used by many robots, universities and companies, becoming the de facto standard for robot programming.

Figure 2.2: Willow Garage PR2 robot

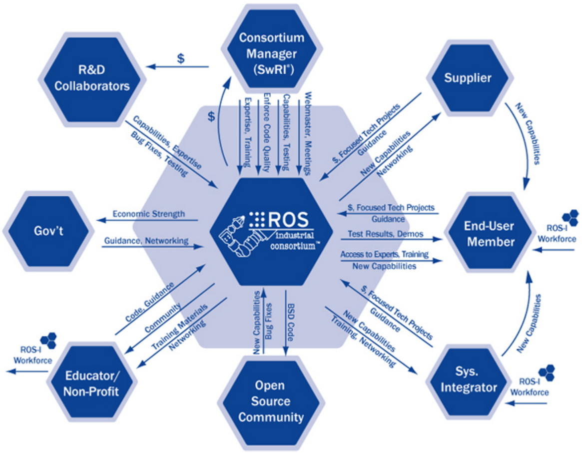


Figure 2.1: ROS Industrial Consortium

2.3. Meta-Operating System

ROS is an open source, meta operating framework for robots, hence it is nothing else than a middleware.
A middleware can be defined as a piece of software that gives an extra level of abstraction to the developer through a layer between the operating system and the applications. It basically sits in the middle of software components and facilitates their interaction.
Its purpose is to provide an abstraction model for functions and at the same time provides the low-level implementation. Every middleware must provide:

- Portability: common programming model regardless the programming language and system architecture
- Complexity management: low-level aspects are managed by libraries and drivers inside the middleware itslef
- Reliability: middleware allows robot developer to discard low level details
- abstraction from sensors/actuators hardware;
- communication protocol for data transport

As a result, they play an essential role in the development of complex applications that rely on a number of hardware and software tools.
While they are still under active development, they are not yet capable of providing a complete set of functions for general purpose robots.

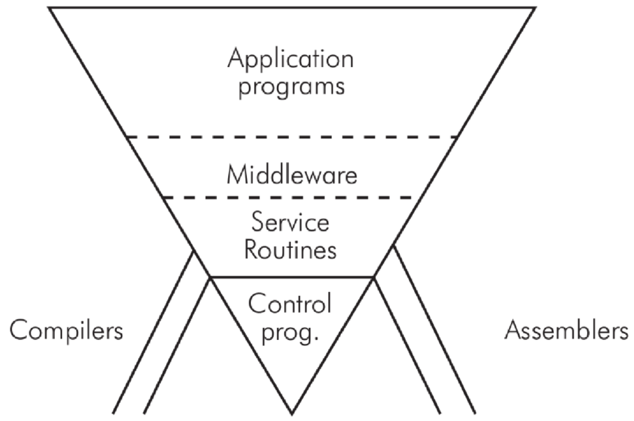


Figure 2.3: Meta Operating System

Several robotic middlewares have been proposed in the past years (OROCOS, ORCA, YARP, BRICS etc.) and eventually came ROS.

2.3.1. Phylosophy 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.

- *Distributed*: Programs can be run on multiple computers and comunicate over the network.

- *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.

2.4. ROS Architecture

ROS is based on a graph architecture where processing takes place in nodes, which communicate with each other by exchanging messages asynchronously through the use of topics to which they can subscribe to and/or on which they can publish, and synchronously with the calling of services, similar to RPCs. Structurally, ROS is developed on 3 conceptual levels, File-system Level, Computational Level and Community Level, of which we are going to examine the constituent elements and their role in the architecture.

2.4.1. File-system Level

The Filesystem Level includes all resources used in ROS, in particular

- Packages
- Metapackages
- Manifest
- Message types
- Service types

Packages

Packages are the main structure for organising ROS Wiki [31]. The processes, libraries, configuration files, datasets, and all the files used by the system at runtime are stored in these files. They are the structure that can be found within a ROS-based system. At the filesystem level, the package is represented by a directory. The structure within it includes some subfolders to manage the elements In order to facilitate its development, in particular:

- *include/packagename*: C++ include headers (make sure to export in the CMakeLists.txt)

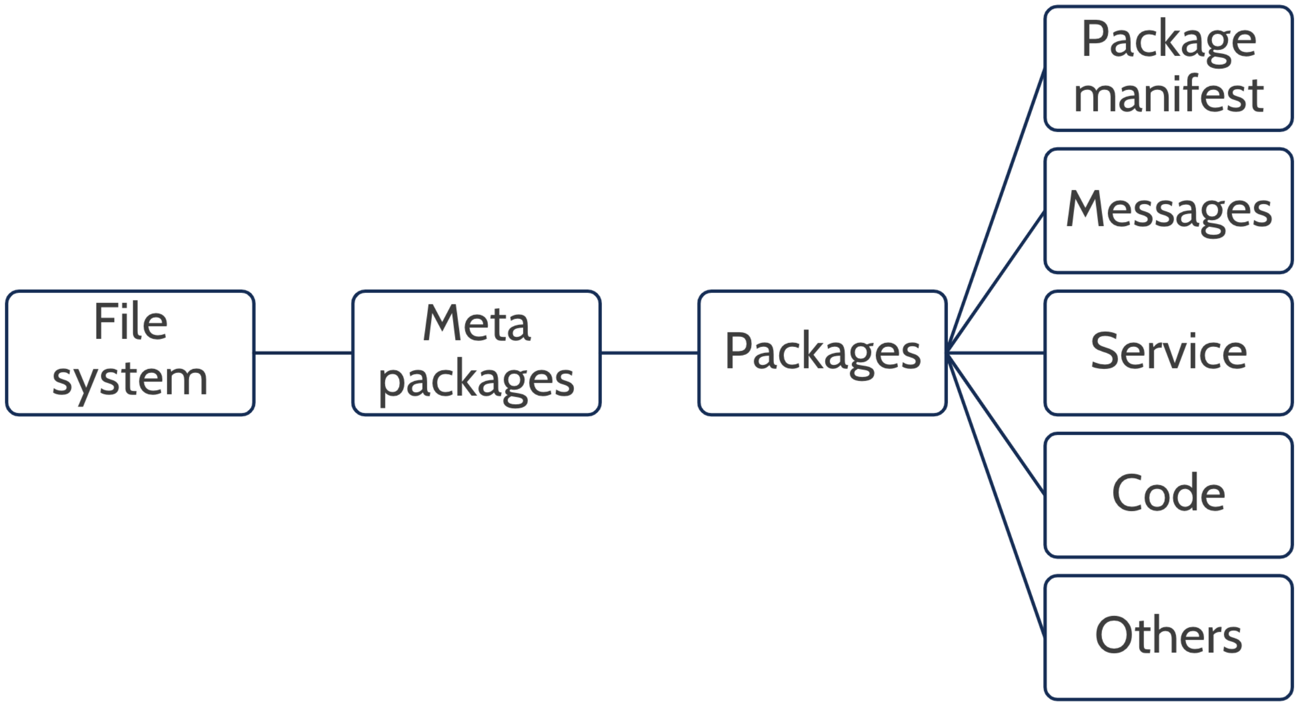


Figure 2.4: File System level representation

- *msg*:Folder containing Message (msg) types
- *src/packagename*: Source files, especially Python source that are exported to other packages.
- *srv/*: Folder containing Service (srv) types
- *scripts*: executable scripts
- *CMakeLists.txt*: CMake build file
- *package.xml*:XML file containing package structure
- *CHANGELOG.rst*: Many packages will define a changelog which can be automatically injected into binary packaging and into the wiki page for the package

Metapackages Metapackages are specialised structures whose only task is to represent a group of packages that have common characteristics with each other. The metapackages that are created in the context of older versions of ROS and later updated may also result from the conversion of older stacks that perform similar functions in the context of older versions of ROS Wiki [28].

Manifest

A package manifest consists of an XML file named package.xml that must be included in the root folder of any catkin-compliant package. It contains information about the package, including its name, version number, authors, maintainers, and dependencies on other catkin packages. There is a strong similarity between this concept and the manifest.xml file used in the legacy rosbuild build system. System package dependencies are declared in package.xml Wiki [27].
There are a minimal set of tags that need to be nested within the <package> tag to make the package manifest complete.

- *<name>*: The name of the package
- *<version>*: The version number of the package;
- *<description>*: A description of the package contents;
- *<maintainer>*: The name of the person(s) that is/are maintaining the package;
- *<license*: The software license under which the code is released.

Message types
Message types define the structure of messages sent by ROS Wiki [29]. Each file, with extension .msg, represents a different type of message. Within the file each line represents a message field. Each line, in turn, contains two columns: the first one for the data type of the field (Int32/int (C++/Phyton), bool, string, time, etc.), the second for the name. It is possible to assign values to the fields within these le, in this case we speak of constants. Example of msg file (C++):

Service types
Service type are files that define the structure of request/response for ROS services Wiki [34]. These are directly built upon the msg format to enable communication between nodes. They are stored in dedicated .srv files in the srv/ subdirectory of a package. Example of srv file (C++):

```
bool add(beginner_tutorials::AddTwoInts::Request  &req,
         beginner_tutorials::AddTwoInts::Response &res)
  {
    res.sum = req.a + req.b;
    ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b)
    ROS_INFO("sending back response: [%ld]", (long int)res.sum);
   return true;
  }
```

2.4.2. ROS Computational Graph Level

The Computation Graph is the peer-to-peer network of ROS processes that are processing data together. The basic Computation Graph concepts of ROS are nodes, Master, Parameter Server, messages, services, topics, and bags, all of which provide data to the Graph in different ways Wiki [26].

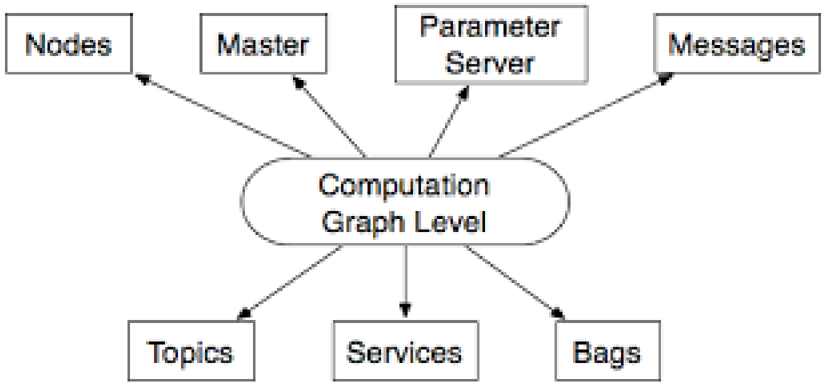


Figure 2.5: Computation Graph

Nodes

A node is a process that performs computation. Nodes are combined together into a graph and communicate with one another using streaming topics, RPC services, and the Parameter Server Wiki [30]. Following the concept of modularity of the system, each node will be related to only one specific functionality.

ROS in fact discourages the creation of 'omnipotent' nodes that perform many functions in order to make the system more maintainable, reusable and clear.

The use of nodes in ROS provides several benefits to the overall system. There is additional fault tolerance as crashes are isolated to individual nodes. Code complexity is reduced in comparison to monolithic systems.

Topics

Topics are buses identified by a proper and unique name that allow messages to be exchanged between nodes. They implement a mechanism of publishing and subscribing: nodes can be publishers and/or subscribers if they are set up to send or receive messages. The division between data producer and data user is clear and separated by anonymity policies between the nodes. Each topic may have a maximum number of messages to be kept in the queue in case they accumulate, those in excess are not added to the queue and lost Wiki [35].

Services

Services are a two-way communication tool between nodes. It is a mechanism that extends that of messages with the possibility not only to send commands to a specific node but also to remain in listening and receiving a structured response from it. Each service is first described in an .srv file where the parameters and type of service are indicated in addition to the name of the service. of the service, as well as the parameters and the type of return data (see Service type). Within the server node, the service is represented by a function which takes as input two pointers to objects of the server class: one one will include the function parameters (Request), the other will collect the return value. will collect the return value Wiki [33]

Messages

The nodes in the graph communicate by exchanging messages. These may be simple and of a primitive type (integer oat, string, char, etc.) or arrays or even more complex, with structures similar to those seen in C.

Bags

Bags represent the system by which ROS saves logs and keeps track of all messages exchanged within a topic. The rosbag tool, once associated with a topic, saves each message exchanged within a related file with the extension .bag. It is very useful for storing data from the sensors as it allows the developer to create a sort of "black box" of the robot. black box" of the robot. ROS also provides a playback tool that allows the visualise and playback the collected data via a graphical interface.

Master

The master in ROS first takes care of registering new nodes within the network, then managing the connection between the nodes in the graph, routing messages and allowing access by one node to the services of another. It is the heart of the software and can only be active one master at a time. It can be started via the roscore command or launched automatically at the start of a node through the correct implementation of the file.

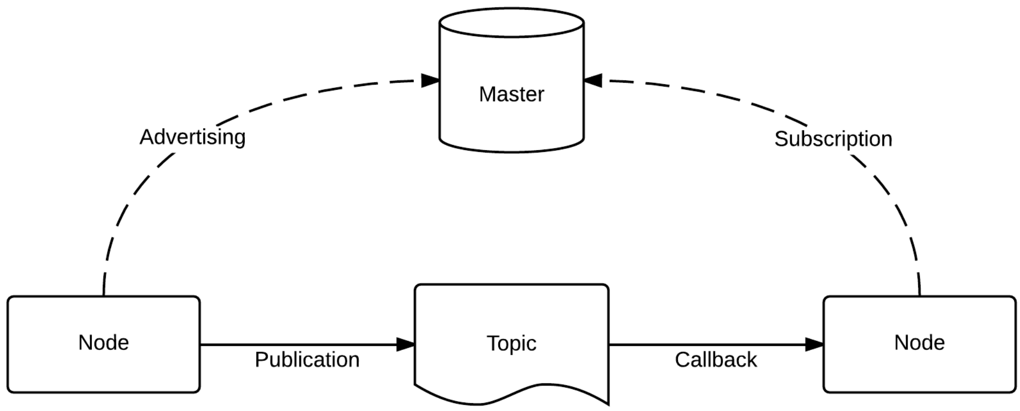


Figure 2.6: Visualization of Master-Node-Topic relationship

Parameter server

The parameter server is basically a component of the master, allowing certain configurations accessible via the network API to be shared publicly with all nodes. Although not an extremely high-performing it is nevertheless useful for the testing phase of the software. Parameters are named using the normal ROS naming convention. This means that ROS parameters have a hierarchy that matches the namespaces used for topics and nodes, Wiki [32].

Coordinate Frames and Transforms

Typically, a robot has many 3D reference systems that change over time. The ROS tf package keeps track of all these coordinate systems in a tree structure. This concept is also necessary to understand later on how URDF files handle the various parent and child links. The tf package therefore, keeps track of all existing relationships between the co-ordinate frames of points and calculates the transforms between them. The transform tree can also be viewed by developers for debugging purposes using the command view frames.

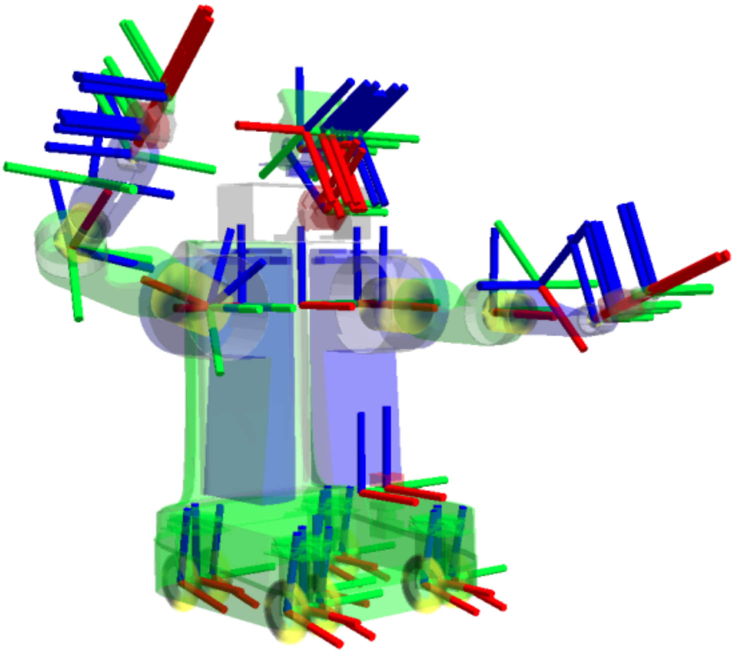


Figure 2.7: Transform Frames of a robot

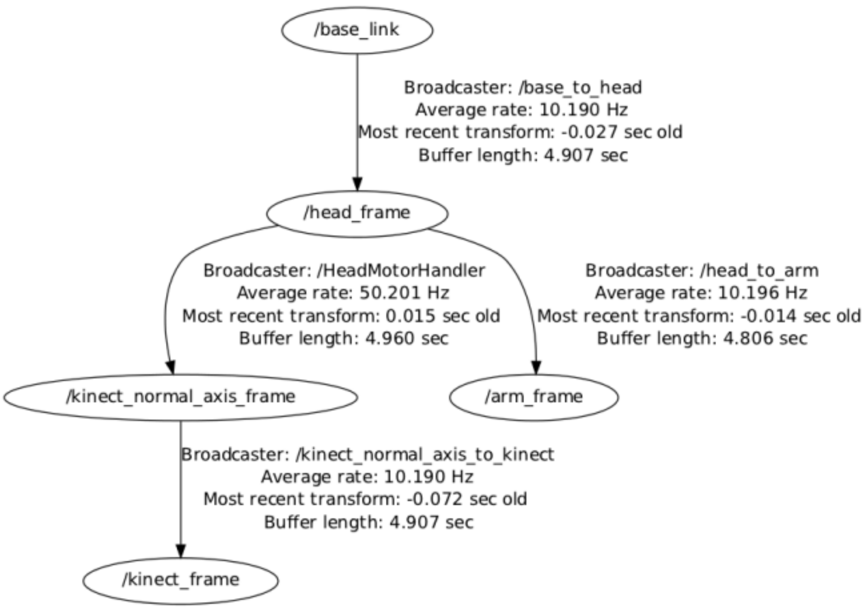


Figure 2.8: Transform Frames Tree of a robot

2.5. ROS Tools and Simulators

ROS provides built-in tools that come to hand when developing robotic applications or when we want to investigate deeper in some issues. The tools described below, although others exist, are the one that were used the most during Oversonic project: RViz (3D visualization tool), Rqt (framework that enables GUI tools: Rqt graph, that displays correlation between nodes and messages in graph form, and Rqt plot) and finally, special emphasis will be placed on Gazebo, a 3D simulator that has long been used in the course of this project.

2.5.1. RViz: 3D Visualization Tool

RViz is a 3D visualization built-in tool of ROS. Its main purpose is to visualize ROS messages and topics in three dimensions, helping the user out to display data and to understand how our system behaves. When opening a new RViz window from scratch, what appears is a black 3D scene. The user can indeed build the environment by customising global options (e.g. the fixed frame that provides a static reference view) and grid settings. It is possible in fact to select which features we want to visualize, ticking them directly from the left panel (picture 2.9). RViz can visualize topic from camera sensor, showing the images on a dedicated box. This feature applies to any kind of sensor that communicates via ROS to our robot, e.g. Lidar, tracking camera, RGB camera etc.

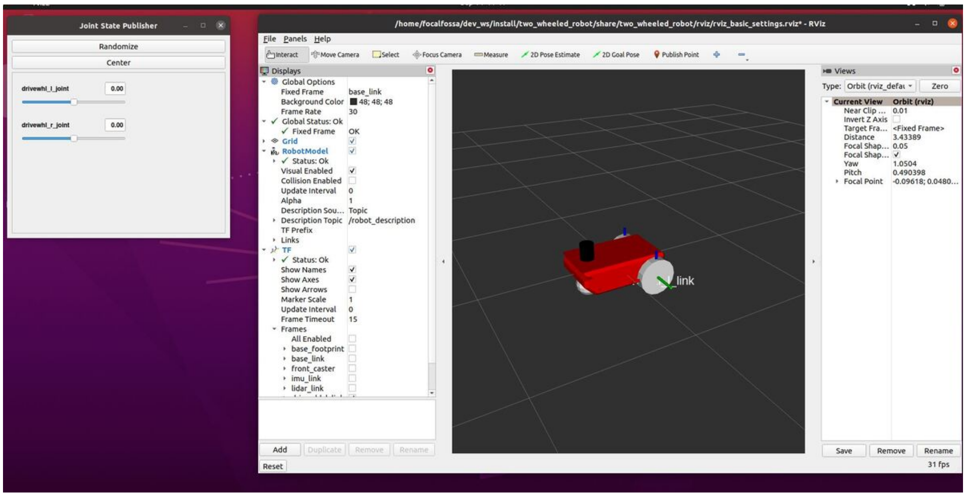


Figure 2.9: RViz GUI

2.5.2. Rqt: ROS GUI Development Tool

Rqt is a software framework of ROS that implements various GUI tools in form of plugins. In particular, particularly useful is rqt graph. The main goal of this tool is to let visualize ROS nodes, topics and messages in order to facilitate debugging and understanding of the system. In fact, when using ROS it is useful to display the on-going graph to better understand how the various nodes are communicating and how messages are being exchanged. Rqt graph thus results useful in:

- Having a global overview of the system
- Debugging code in case there exist issues in nodes communication (e.g. two nodes are not connected in reality or too many nodes publish on the same topic)

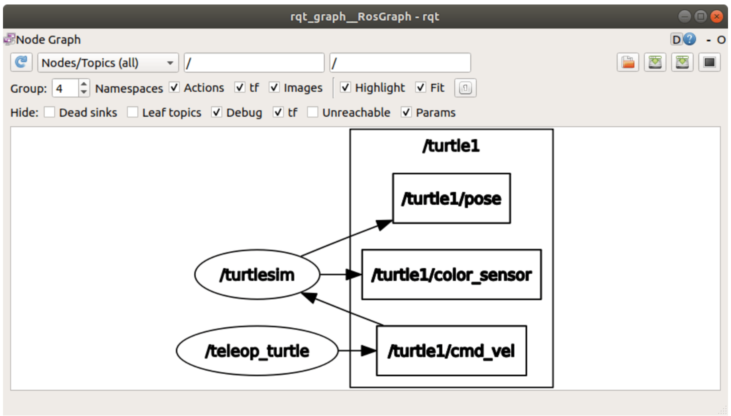


Figure 2.10: Graph from turtlesim

In figure 2.10 two nodes have been initialized and from the graph we can see the path of the messages.

2.5.3. Gazebo Simulator

Dealing with real robots means using physics labs, charging batteries, calibrating sensors and many other small tasks involving hardware. In real-life cases, even the best robots break down periodically due to human error, wear and tear or structural defects. This is where simulators come in: in simulation, we can faithfully simulate the actions that the robot will perform, without the disadvantages listed above. It is also possible to model actuators and sensors either as ideal devices or by adding varying degrees of distortion, error or failure. Thus, the solution of simulating robots in a virtual environment is efficient and cost effective. The problem of SLAM (simultaneous localisation and mapping) has always been one of the most important research topics in the community. In response to this need, 'Stages', for example, with high computational capabilities and different kinematic configurations, built-in sensors have been developed over the years. In the context of this paper, it was decided to use Gazebo, a 3D simulator that provides various models of robots, sensors, environments, offering faithful and reliable simulations thanks to its physics engine. Gazebo is in fact one of the best-known simulators used in open source robotics.

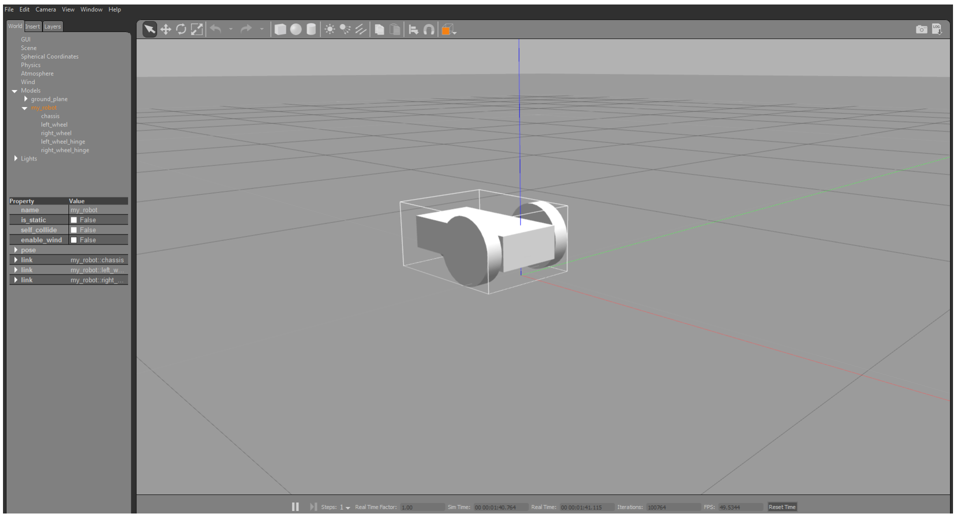


Figure 2.11: Gazebo GUI

To use the simulator, it is either possible to download a pre-fabricated model, available among the Gazebo robots (e.g. TurtleBot, PR2, Pioneer2 and other known robots) or to build your own robot model (we'll talk later on about SDF and URDF). As far as robots are concerned, a wide variety of sensors can be applied to your model: stereo camera, RGB camera, tracking, contact sensors. It is also possible to add the noise model to the sensors. ROS integrates very closely with Gazebo, thanks to the gazebo ros package. This provides a plugin module for the simulator that enables bi-unique communication between Gazebo and ROS. Simulated sensors and physical simulation data are thus transmitted bi-directionally between the two platforms.

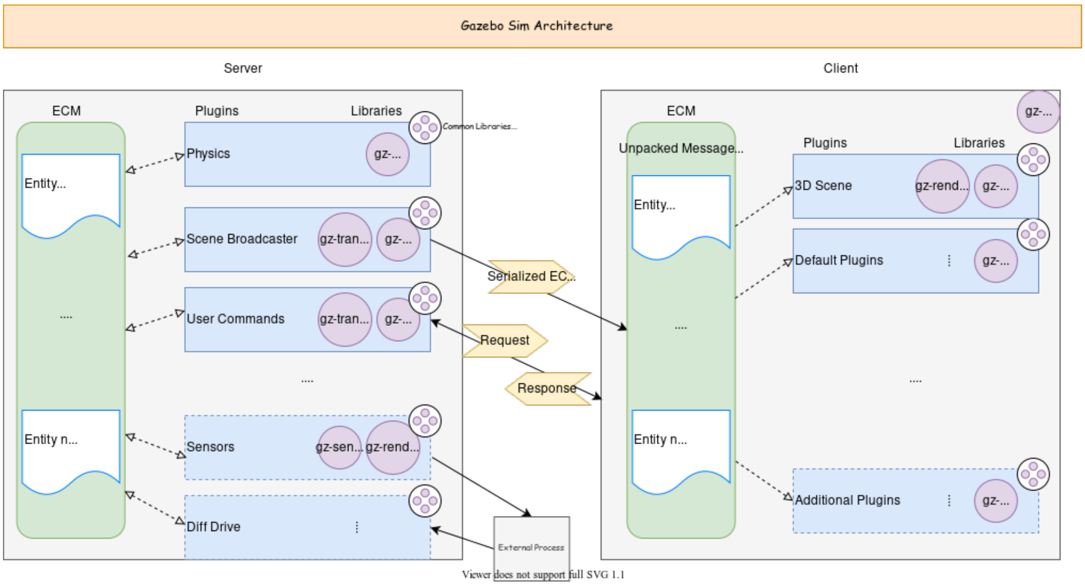


Figure 2.12: Gazebo Sim Architecture

Robot Modelling Formats

As mentioned above, one of the possible ways of describing a robot model is through the URDF. The unified robot description format is a package containing a file in XML format. A URDF file is written in such a way that each link of the robot is a child of some other parent link, with joints connecting each link. In turn, the joints are defined by an offset from the reference frame of the parent link and their axis of rotation. This creates a complete kinematics model. Below is reported a simple sample of the creation of a base link:

```
<link name="base_link">
  <visual>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
    <material name="blue"/>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </collision>
</link>
```

When we are developing complex robot models, it can happen that the notation within the XML file becomes verbose and prone to confusion. It is precisely for this reason that the XACRO (Macros XML) model was created, which is nothing more than an XML that allows the design phase of the model to be divided into sub-parts, which are then imported one by one into the main file. This results in cleaner code reading and facilitates debugging.

Below is an example of importing arguments from an external file:

```
<xacro:property name= robotname value= R001 />
<link name= ${robotname}s_leg />
```

In order to use the model created within the Gazebo environment (which, as we have already mentioned, works with SDFormat files), it is necessary to convert first from XACRO to URDF and finally from URDF to SDF (Simulation Description Format). In fact, files with this format include a description of the world in which the robot will be placed, a series of features related to the simulated physical world (static and dynamic objects, lighting, terrain and even physics), and plug-in additions. Below is an example of SDF file defining a model from scratch:

```
<?xml version='1.0'?>
<sdf version='1.9'>
  <model name='my_model'>
     ...
  </model>
</sdf>
```

3 | Robot

3.1. Introduction

«In the twenty-first century the robot will take the place which slave labor occupied in ancient civilization. There is no reason at all why most of this should not come to pass in less than a century, freeing mankind to pursue its higher aspirations.»
Nikola Tesla (1856 - 1943)

«Robots of the world!
The power of man has fallen!
A new world has arisen:
the Rule of the Robots! March!»
Rossum's Universal Robot (1920)

Man has always spent his life working. Dangerous and degrading work has been the cause of death for many people for centuries. In this sense, there has always been a tendency to try to relieve man of the heaviest jobs by looking for machines or automatic systems to replace him. In a sense, with the advent of the industrial revolution, we witnessed the first real process of robotizing in history. On the other hand, with the evolution of discoveries in the medical field, the desire and curiosity arose in man to try to clone himself, artificially constructing his own like. It is here that these two needs and tendencies come together in what we now call humanoid robots. Indeed, humanoid robots are designed and built to replace humans in the most physical and repetitive tasks, in order to ensure greater well-being.

3.2. History of Robotics

In recent years, the general public has become increasingly interested in robots and robotics research. New developments, e.g. robotic competitions, which "push beyond the boundaries of current technological systems" (such as Defense Advanced Research Projects Agency (DARPA) in the United States), especially in the area of robotics, have promised and delivered fully integrated systems, Lima et al. [11].
But the idea of creating intelligent, useful machines for humans has existed since the beginning of mankind. In fact, ever since civilisation, one of the most unattainable desires and ambitions for mankind has been to create artifacts of his own image.
From a historical perspective, the first example that can be interpreted as such dates back to 3500 B.C., with the legend of the giant Talus, the slave forged by Hephaestus. Continuing in time and reaching the Babylonians in 1400 B.C., we can observe the creation of the first automatic machine, the clepsydra water clock. Continuing through the centuries, creations became more and more technologically advanced and jumping back to the 1500s, we encounter Leonardo da Vinci and his many inventions.

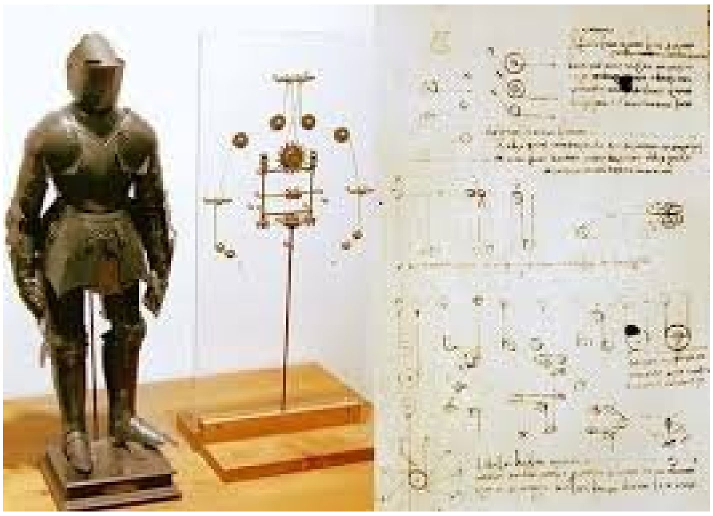


Figure 3.1: Leonardo da Vinci's mechanical knight: sketches on the right, rebuilt and showing its inner workings on the left.

The concept of the robot then gradually entered people's minds thanks to this long process, but it was only in the 20th century that it took on a real physical connotation.
The term 'robot' was introduced in 1920 by the play 'Rossum's Universal Robot', by Karel Čapek: it derives etymologically from the Slavic root word 'robota' meaning subordinate labor.

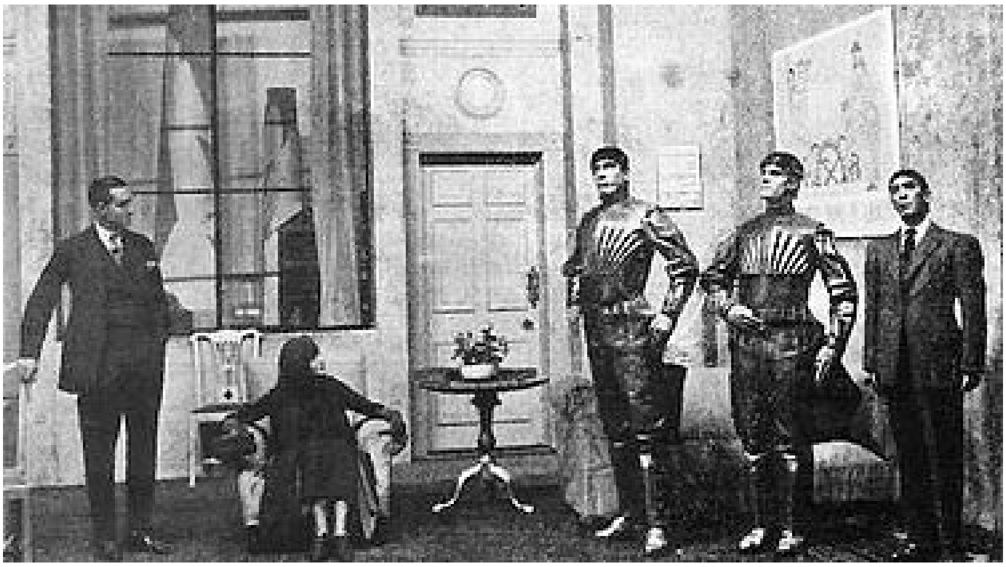


Figure 3.2: A scene from Rossum's Universal Robot play, showing three robots

Later, during the middle of the century, the first research into the connection between human and machine intelligence was undertaken, marking the beginning of Artificial Intelligence (AI). Between 1950 and 1980, Isaac Asimov wrote the so called "Three Laws of Robotics" in his book 'Runaround'. They are encoded in the "positronic brains" and are defined as follows, Asimov [2]:

- A robot may not injure a human being or, through inaction, allow a human being to come to harm.
- A robot must obey the orders given to it by human beings, except where such orders would conflict with the First Law.
- A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

Around those years, the first robots were created, they stemmed from the confluence of advances in two fields: numerically controlled machines for precision manufacturing and remote control to handle highly radioactive materials. In fact, these two fields already featured modern applications of technologies such as mechanics, control, computational science and electronics. The first robots were therefore master-slave arms, designed to reproduce the mechanics of the human arm but with rudimentary control and little perception.
During the second half of the century, the development of integrated circuits, digital computers and miniaturised components allowed terminal-controlled robots to be designed and developed.

In fact, in the 1980s, robotics was defined as the science that studies the connection between action and perception.In fact, action involves a locomotion apparatus that moves in the environment and a manipulation apparatus that performs actions, modifying its surroundings, thanks to special actuators and end-effectors.
Perception is then extracted from the sensors that provide information about the state of the robot (e.g. position and speed) and its surroundings (e.g. range and vision). In the 1990s, research was further accelerated by the need to rely on robots to replace human presence in critical environments.
As we enter the new millennium, robots have undergone profound transformations both in their scope of use and in their shapes and sizes.

Humanoid Robots

As reported in the article "Humanoid Robots:Historical Perspective, Overview and Scope", Siciliano and Khatib [22]:

"*The long saga of humanoid robots in science fiction has influenced generations of researchers, as well as the general public, and serves as evidence that people are drawn to the idea of humanoid robots. Humans generally like to observe and interact with one another. In their social behavior, people are highly attuned to human characteristics, such as the sound of human voices and the appearance of human faces and body motion.*
Infants show preferences for these types of stimuli at a young age, and adults appear to use specialized mental resources when interpreting these stimuli. By mimicking human characteristics, humanoid robots can engage these same preferences and mental resources. Throughout history, the human body and mind have inspired artists, engineers, and scientists, using media as diverse as cave paintings, sculpture, mechanical toys, photographs, and computer animation.
Humanoid robots serve as a powerful new medium that enables the creation of artifacts that operate within the real world and exhibit both human form and behavior.
The field of humanoid robotics focuses on the creation of robots that are directly inspired by human capabilities and/or selectively imitate aspects of human form and behavior. Humanoids come in a variety of shapes and sizes, from complete human-size legged robots to isolated robotic heads with human-like sensing and expression."
Thus, humanoid robots were developed to be employed as multi purpose mechanical workers, and were designed to work alongside humans in daily tasks, being a support, living in the same environment and using the same tools. It must also be considered that when the robot moves around in the work environment, there can be multiple risks for the worker;

in this respect, a subfield of robotics, called cognitive robotics, has taken hold. Indeed, robots can take advantage of the traditional communication methods used among humans to become more aware of their surroundings. An even more ambitious aim is to interpret human gestures through vision (eye gaze, body language). On the other hand, this could put a human in a difficult relationship with the robot, modelled by the phenomenon called 'uncanny valley', a concept introduced in the 1970s by Masahiro Mori, a professor at the Tokyo Institute of Technology. Masahiro in fact argues that:
"*I have noticed that, in climbing toward the goal of making robots appear human, our affinity for them increases until we come to a valley, which I call the uncanny valley.*"

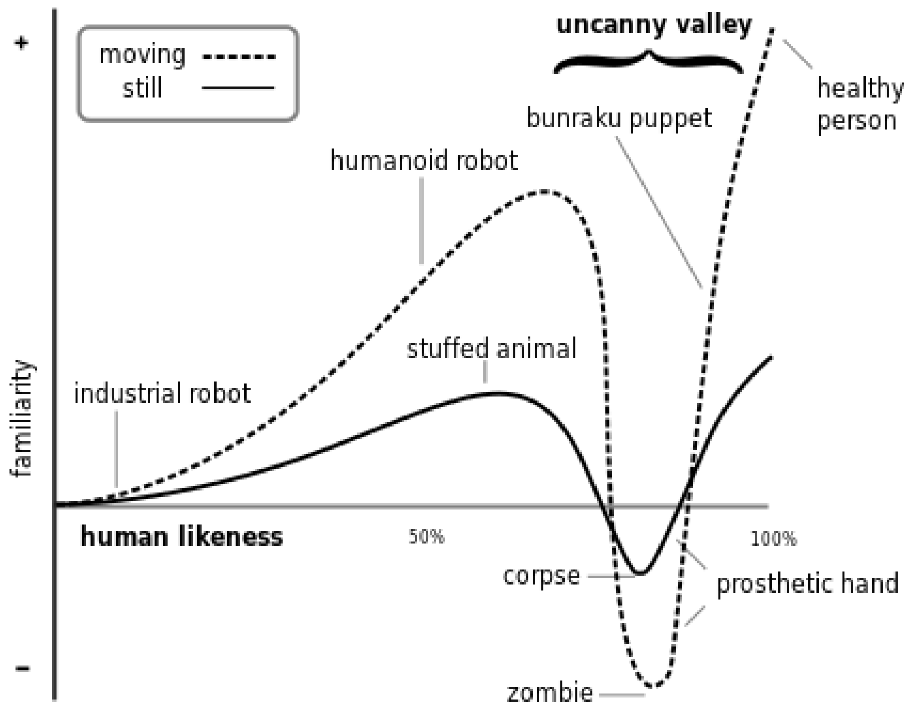


Figure 3.3: Mori Uncanny Valley

Mori better explains this concept with the example of the prosthetic hand:
"*One might say that the prosthetic hand has achieved a degree of resemblance to the human form, perhaps on a par with false teeth. However, when we realize the hand, which at first site looked real, is in fact artificial, we experience an eerie sensation. For example, we could be startled during a handshake by its limp boneless grip together with its texture and coldness. When this happens, we lose our sense of affinity, and the hand becomes uncanny.*"
On the other hand, many scientists and researchers in the robotics community see humanoid robots as a possibility to better investigate human nature itself. A part from the

roles mentioned above, a humanoid robot could work as an avatar for telepresence, test ergonomics and serve for any other roles that a human can do. Even though in the past decades, humanoids have only been applied in research field, times seem to be mature to put these robots on field and let them cooperate with humans.

3.3. Robee: Oversonic Robotics configuration

In order to make physical sense of the results obtained within this project, it is important to define what technologies were used and what materials made up Robee's hardware.

3.3.1. Hardware components and software architecture

It is important to bear in mind that the Oversonic project has an architecture split between the robot (also referred to as the edge) and the cloud, and these two components coexist in a hybrid. Describing the system from the cloud, the hardware component consists of a scalable node pool based on the 2.35Ghz AMD EPYC 7452 processor that can achieve a boosted maximum frequency of 3.35GHz with 32 GB RAM memory, running a Kubernetes instance on top of Linux Ubuntu 18.04 (Bionic Beaver). As far as the robot is concerned, all the computational power is provided by 2 Intel NUCs 8 including an Intel Core i5-8259U Processor (6M Cache, up to 3.80 GHz), 8 GB RAM and Integrated Graphics Intel Iris Plus 655. The operating system which is mounted on is Linux Ubuntu 20.04 (Focal Fossa), and all the modules are running containers that on turn are managed by KubeEdge, a containers orchestration system built on Kubernetes.

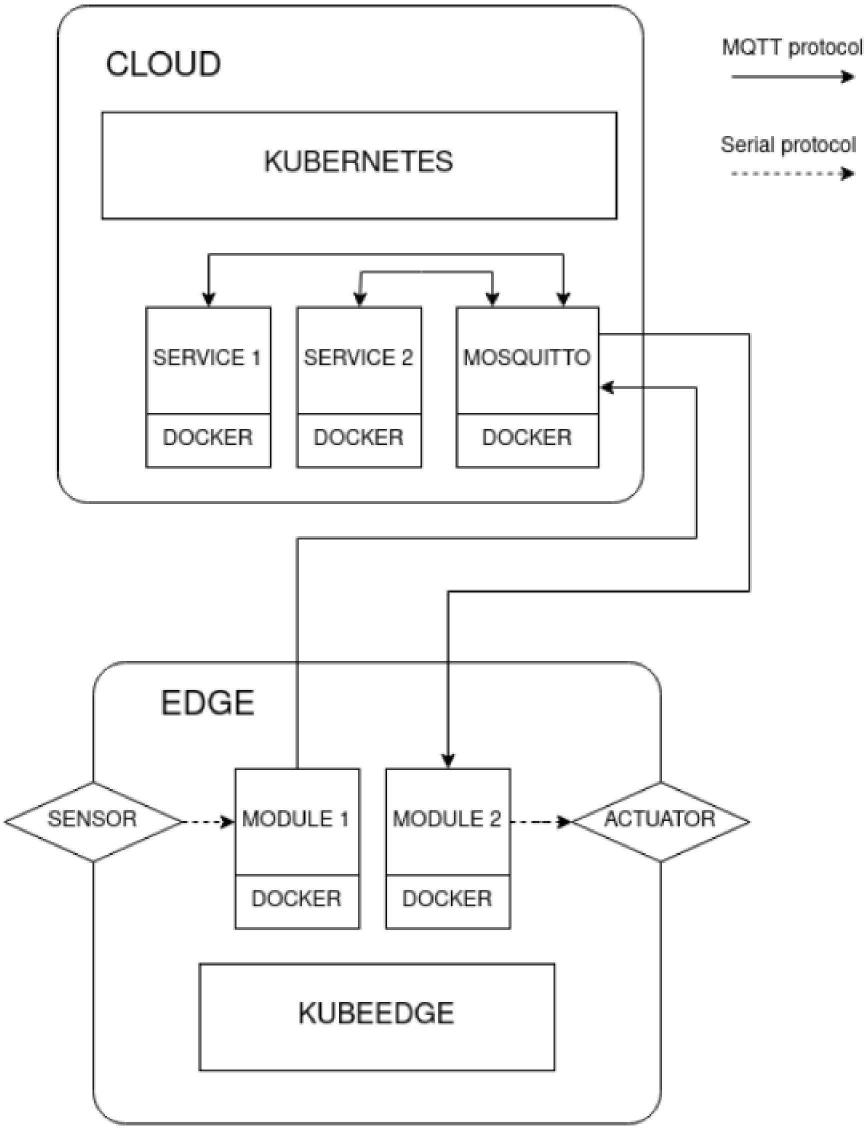


Figure 3.4: Oversonic Architecture

Internet of Things

In the case of the Robee project, the architecture is therefore composed of various software modules that are containerised and must be able to communicate with each other. The MQTT protocol is an optimal choice for this case.
From the official MQTT.org site: "*MQTT is an OASIS standard messaging protocol for the Internet of Things (IoT). It is designed as an extremely lightweight publish/subscribe messaging transport that is ideal for connecting remote devices with a small code footprint and minimal network bandwidth. MQTT today is used in a wide variety of industries,*

such as automotive, manufacturing, telecommunications, oil and gas", MQTT.org [15]. MQTT therefore operates at the application layer of the OSI model, relying on TCP at the transport layer. The MQTT protocol establishes two kinds of entities in the network: a message broker and a number of clients. The broker is nothing more than a server that receives all messages from all clients and then routes these messages to the relevant destination clients. A client is anything that can interact with the broker to exchange messages. The messages are routed to clients basing on topics: every message is published over a specific topic, and only the clients subscribed to it will receive the message.
A client, therefore, can be an IoT sensor or an application in a data centre that processes IoT data. Each MQTT message has a command and a payload. The command defines the type of message:

- CONNECT: initial message sent from client to broker, to instantiate a new connection
- DISCONNECT: final message sent from client to broker to end the connection
- PUBLISH: command to publish a message over a specified topic, it is sent from client to broker and then routed from broker to every client that appears to be subscribed to that topic
- SUBSCRIBE: message sent from client to broker in order to request a subscription to a specified topic

All MQTT libraries provide simple ways to handle such messages directly and can automatically populate certain required fields, such as 'message' and 'client Id'.

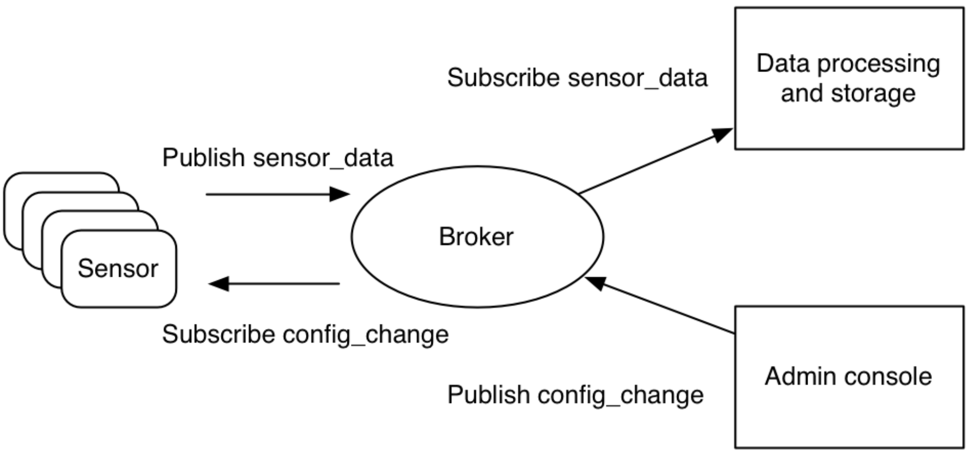


Figure 3.5: The MQTT publish and subscribe model for IoT sensors

3.3.2. Robots Configurations

The robots covered by the work in this thesis are mainly three

- R007 is a small autonomous mobile robot used by Oversonic as a prototype in the testing phase and features skid steering kinematics. In fact, it has two belts with two torque motors. The system is based on an Intel NUC and peripherals: two or four lidar sensors, a tracking camera and a depth camera mounted on the top base. The use of this AMR (autonomous mobile robot) is mainly conceived in conjunction with its larger 'brother' Robee or in industrial logistics environments.

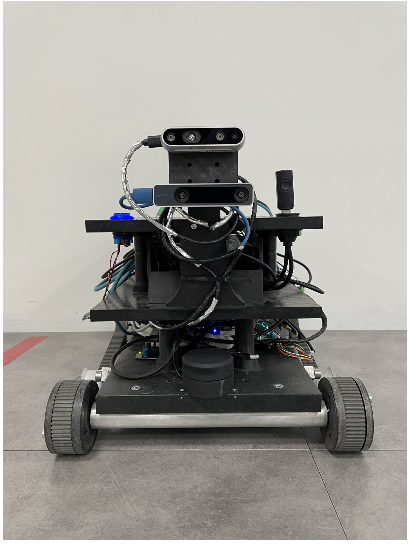


Figure 3.6: R007

- R012 is the humanoid robot developed in Oversonic Robotics, now in its fourth evolution from the initial prototype and featuring a differential drive base. The system is divided into two parts, a lower body and an upper body, each featuring an NUC terminal and several sensors, but in this analysis we will focus exclusively on the lower part. The lower body in fact contains within it the two torque motors, which move two wheels actively. For the robot's stability, two passive caster wheels have been added to the front and rear of the base. Two Lidar sensors are then mounted on the base and going up about halfway up the torso are a tracking and depth camera.

Figure 3.7: Robee R012

- N002 is a robot used for the testing phase of Robee's lower body. It has a similar purpose of use to R007. It has a differential drive base with two torque motor actuators, like R012 but with a physical arrangement of peripheral sensors but only one NUC.

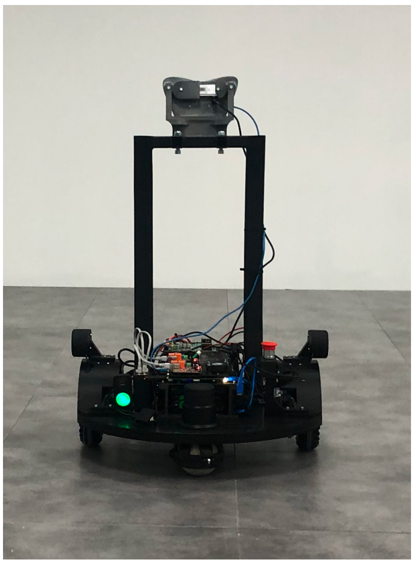


Figure 3.8: N002

The main features of the skid steering and differential drive kinematics will be listed below. **Skid Steering**

Skid Steering is a particular kinematics configuration featuring two tracks. It is composed of two tracks on its basic configuration, left and right, and the control variables are indeed left and right speed. Another configuration entails a 4 wheels set up featuring a low wheelbase configurations so that they can be deemed as two tracks. When rotating the central point does not move as the track is sliding on the ground. It is clear that this drive needs proper calibration and slippage modeling in order to be reliable. Some assumptions are needed for this kinematics model: the first, mass is placed in center of the fictitious medium, the second, all the wheels on the same side have the same speed. While in motion, this kind of drive presents multiple ICR and all of them share the same ω_z.

$$\begin{bmatrix} v_x \\ v_y \\ \omega_z \end{bmatrix} = J_\omega \begin{bmatrix} \omega_l r \\ \omega_r r \end{bmatrix} \tag{3.1}$$

The wheels are turning and sliding simultaneously, resulting in two fictitious instantaneous centers of rotation: ICR_{left} and ICR_{right}. Under proper assumptions, skid-steering can be simplified to a differential drive kinematics.

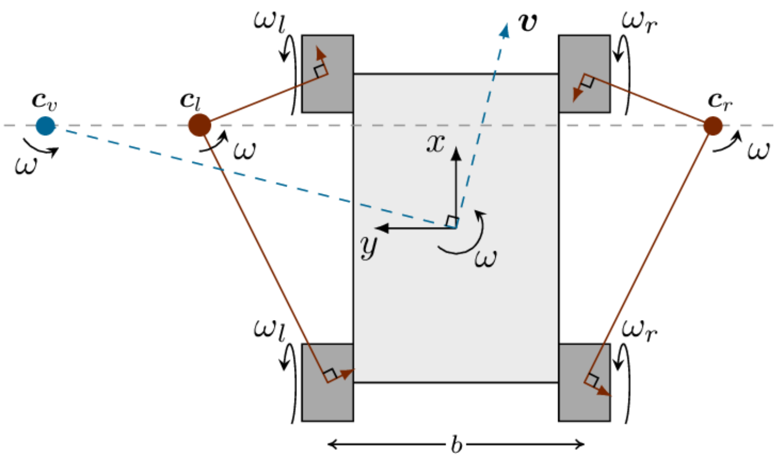


Figure 3.9: Skid Steering Kinematics

Under proper assumptions, skid-steering can be simplified to a differential drive kinematics.

Differential Drive

Differential drive configuration present the following construction:

- two wheels working on the same axis

- two independent motors, one for each wheel
- one or two passive caster wheels

Control input in this case are the linear and the angular velocity of the robot, v and ω. Wheels move around an Istantaneous Centre of Rotation on a circular path with istantaneous radius R and angular velocity ω.

$$\begin{bmatrix} x' \\ y' \\ \theta \end{bmatrix} = \begin{bmatrix} \cos\omega\delta t & -\sin\omega\delta t & 0 \\ \sin\omega\delta t & \cos\omega\delta t & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x - ICR_x \\ y - ICR_y \\ \theta' \end{bmatrix} + \begin{bmatrix} ICR_x \\ ICR_y \\ \omega\delta t \end{bmatrix} \tag{3.2}$$

It is therefore possible to reconstruct robot pose from direct kinematics:

$$x(t) = \frac{1}{2}(\int_0^t (V_R(t') + V_L(t')) \cos\theta(t')\, dt') \tag{3.3}$$

$$y(t) = \frac{1}{2}(\int_0^t (V_R(t') + V_L(t')) \sin\theta(t')\, dt') \tag{3.4}$$

$$\theta = \frac{1}{b}(\int_0^t (V_R(t') - V_L(t'))\, dt') \tag{3.5}$$

where V_R and V_L are defined as follows:

$$V_R = \omega(R + \frac{b}{2}) \tag{3.6}$$

$$V_L = \omega(R - \frac{b}{2}) \tag{3.7}$$

and as a consequence the following is derived:

$$V = \frac{V_R + V_L}{2} \tag{3.8}$$

$$\omega = \frac{V_R - V_L}{2} \tag{3.9}$$

It becomes clear that we can compute robot odometry by integrating the so-called control variables and knowing the parameters of the wheels, namely the direct kinematics. On the contrary, we can derive control variables from a desired pose or velocity.

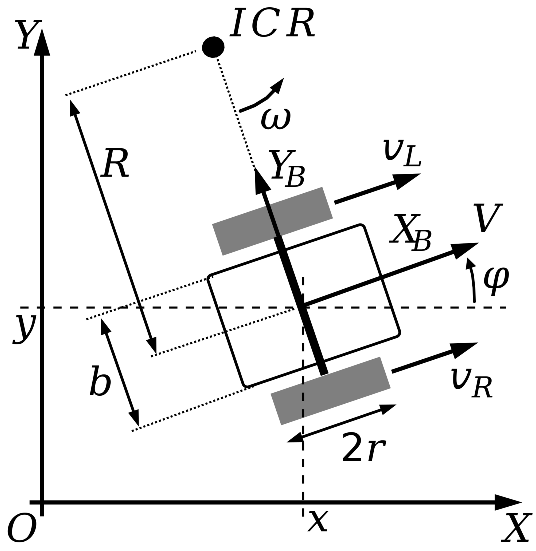


Figure 3.10: Differential Drive Kinematics

It is interesting to mention three borderline cases for this kind of drive:

- $V_L = V_R$: forward linear motion is straight
- $V_L =$ - V_R: rotation in place
- $V_L = 0$ or $V_R = 0$: respectively, rotation about left and right wheel

3.3.3. Sensors

In order for a robot to perceive the world around it and to complete tasks autonomously, sensors are required. We distinguish between proprioceptive (internal state of the robot) and exteroceptive (state of the external environment) sensors. In this section, we focus on the exteroceptive sensors that have been used in Robee, providing a brief overview of their functioning.

D400 Intel Depth Camera

Depth cameras are a type of sensor widely used in robotic applications. They normally consist of two parts: a traditional digital camera, which captures RGB data, and a projector, which captures depth data.The depth system can work in several ways, for example by projecting a grid of light structured in a non-visible spectrum into a scene and then analysing the distortion created in this pattern to determine the distance and/or shape of any object placed in front, Jonasson et al. [9]. In our application, the main reason for using the d455 camera in Robee's lower body is ????distance measurement?????. Traditional digital cameras shoot out an image as a grid of pixels in two dimensions. Each pixel is then associated with three values ranging from 0 to 255, which define the red, green and blue components, so black, for example, is (0,0,0) and a pure bright red would be (255,0,0). . This type of representation is called an RGB image. Thus, each image is composed of three channels each storing the values pertaining to each colour component.

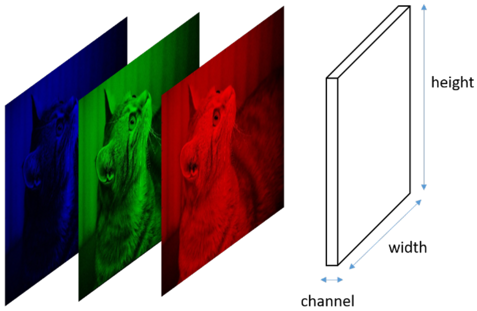


Figure 3.11: Channel decomposition

In the case of a depth camera, on the other hand, the pixels have different numerical values associated with them, where the number represents the distance of the corresponding pixel from the camera, thus the depth. Thus, by unifying this representation we will have a colour map where cooler colours represent closer obstacles, and warmer colours represent more distant obstacles, in depth.

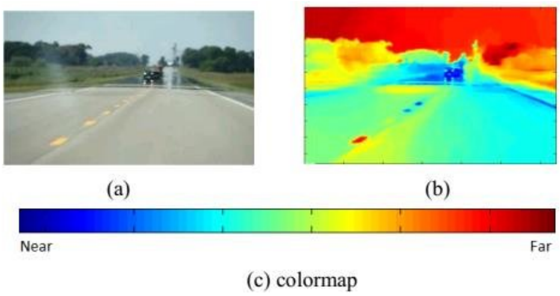


Figure 3.12: Depth-map representation

Figure 3.13: D455 Intel Tracking Camera

Therefore, there are two types of three-dimensional image formats, the first being RGB-D and the second being pointcloud. The first has already been introduced above, and we recall that for each pixel, identified by the coordinates (x,y), four properties (R,G,B,depth) are associated. The substantial difference between the point cloud and RGB-D data is that in the pointcloud, the coordinates (x,y) represent the real world value instead of integer values. When viewing the two types of data, in fact, the former is presented in a sparse structure, while the latter is based on grid-aligned images. A practical application of point cloud will be provided in chapter 6

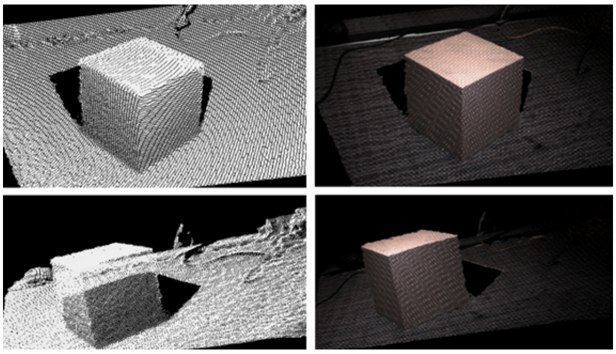


Figure 3.14: Point Cloud sample

Thus, the point cloud can be constructed from RGB-D format images. In fact, by knowing an RGB-D dataset and the camera's intrensic values through a process called camera calibration. Since pointclouds are sets of disordered vectors, it is common for researchers to change the structure of the pointcloud data into 3D voxel grids. The voxel grid geometry is in fact a grid of values in three dimensions, organised in layers of rows and columns. The reason for this conversion also comes from the fact that one often then has to deal with deep learning models that expect highly regular input data formats.

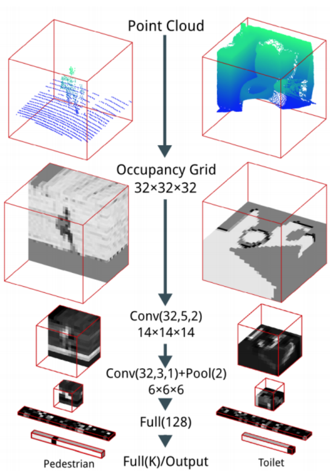


Figure 3.15: Voxel Grid example: A 3D Convolutional Neural Network for Real-Time Object Recognition

T265 Intel Tracking Camera

The T265 tracking camera is designed to integrate odometry data from the robot. It is in fact an independent and robust support for visual-inertia odometry and re-localisation. A key strength of visual-inertial odometry is that the various sensors available complement each other. The images from the visual sensors are supplemented by data from an onboard inertial measurement unit (IMU), which includes a gyroscope and accelerometer. The aggregated data from these sensors is fed into simultaneous localization and mapping (SLAM) algorithms. The tracking is done by comparing the information collected by the two fish-eye cameras, which collect images at 30 fps, Intel [8].

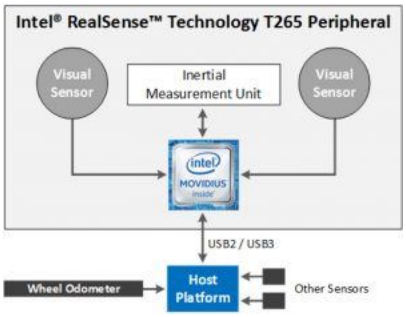


Figure 3.16: Block diagram of Intel T265

Figure 3.17: T265 Intel Tracking Camera

YDLidar

LiDAR (Light Detection And Ranging) identifies technology that measures the distance to an object by illuminating it with laser light, while at the same time being able to return high-resolution three-dimensional information about the surrounding environment. A LiDAR typically uses several components: lasers, photodetectors and readout integrated circuits (ROICs) with time-of-flight (TOF) capability to measure distance by illuminating a target and analysing the reflected light.

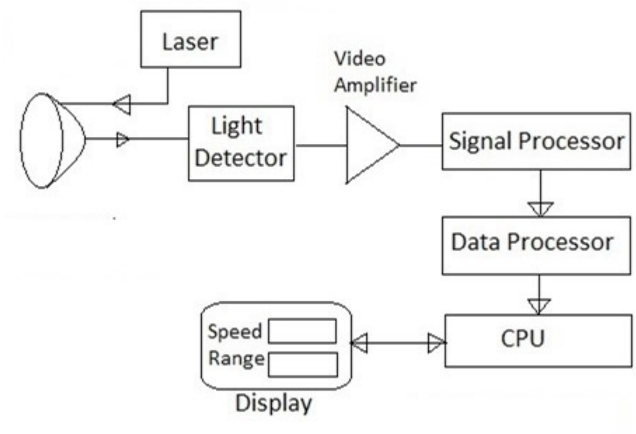


Figure 3.18: Lidar scheme

Figure 3.19: YDLidar

YDLIDAR G2 is a 360-degree two-dimensional rangefinder developed by YDLIDAR. Based on the principle of triangulation, it is equipped withrelatedoptics, electricity, and algorithm design to achieve high-frequency and high-precision distancemeasurement. The mechanical structure rotates 360 degrees to continuously output the angle information as well as the point cloud data of the scanning environment while ranging.

Visual Fiducial System

During the use and development of Robee's navigation, so-called Apriltags were used, a particular visual fiducial system chosen for its robustness and integration with the simulated Gazebo environment. Visual fiducials are nothing more than artificial landmarks, designed to be easily recognisable within the working environment and distinguishable from one another. The methodology is similar to that of a common QR code but with significant applications and objectives. In fact, unlike a QR code, where the user has to frame the tag with the camera and capture the high-resolution snapshot, these types of tags are designed to work with a small amount of information (even as little as 12 bits) but with performance and ease of use that is clearly superior to QR codes. In fact, these types of tags are designed to be automatically detectable and localisable even in low resolution conditions, even providing the relative position and orientation of the tag with respect to the camera. In terms of size, the Apriltags used range from 50 to 100 pixels, including the payload, Olson [17].

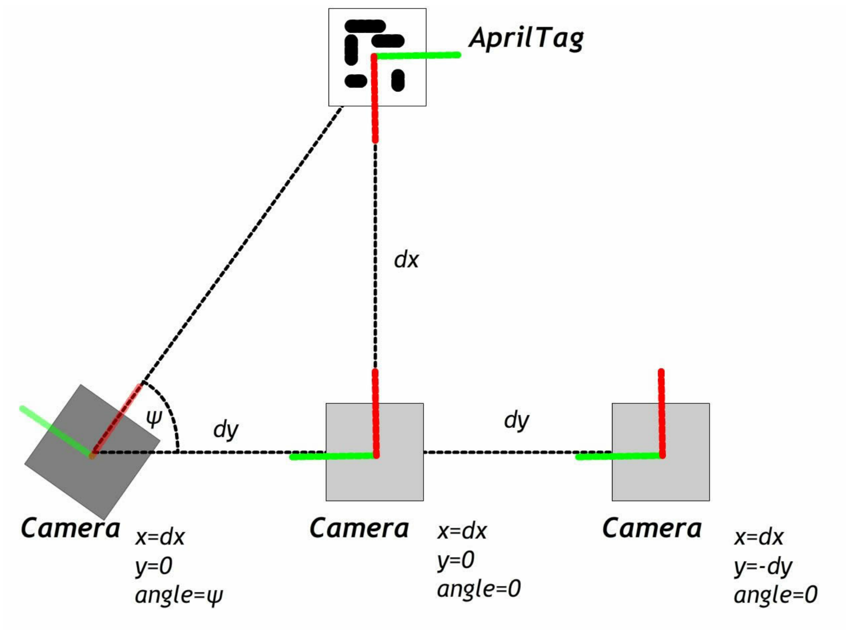


Figure 3.20: AprilTag distance and orientation measurement

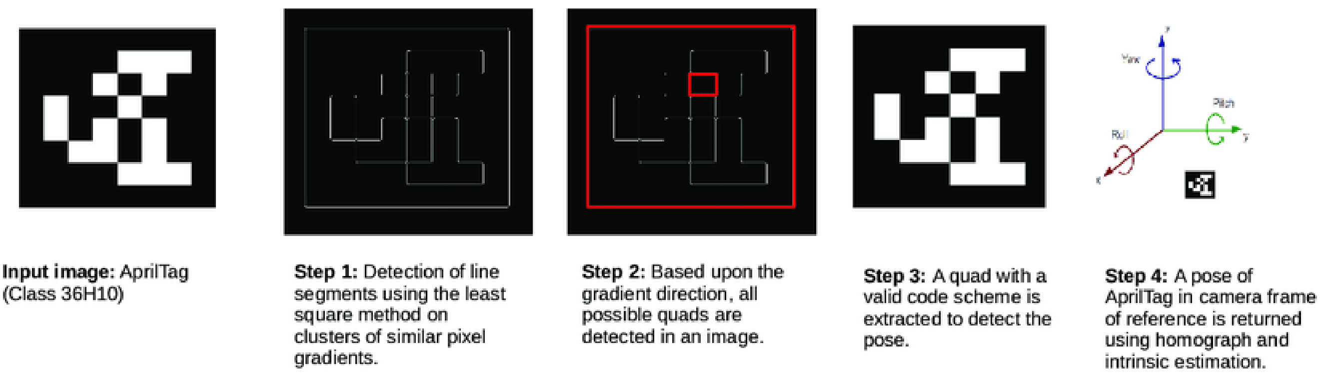


Figure 3.21: How AprilTag detection is performed

Visual fiducial systems have been used in robotics to improve human/machine interaction, enabling the development of commands such as 'follow me'. In the context of this work, tags were used for SLAM (Simultaneous Localisation and Mapping), as an independent support to the sensors mentioned in the previous section. In fact, in both the simulated and real environment, tags were placed in strategic positions in order to reposition the robot to the correct pose by comparing the information coming from the sensors and that coming from the tags. AprilTag provides a package for perfect ROS integration.

Figure 3.22: ROS TF

The apriltag ROS package then takes as input the topic of the rectified image and returns a list of recognised tags and their positions in 3 dimensions. However, in order to work, one must specify in the configuration files (settings and tags) which tag families to search for. A practical application will be provided in chapter 5

Indoor Positioning System

For the purpose of this thesis, another technology that has been used is the Indoor Positioning System. This is in fact a new approach to the problem of localisation when remote GNSS satellites, which are commonly blocked indoors, are not available. There is now a wide offer for this type of solution, even with different communication protocols at its base: from Wi-Fi signals to Bluetooth to ultrasound. In Robee, the choice was made to use an IPS system provided by Marvelmind, which is based on ultrasonic and time-of-flight (ToF) measurements with trilateration, yor [1], and which also provides for communication via ROS topics, thus providing integration with the robotics platform used. The system proposed by Marvelmind consists of a series of static beacons (four were used in our case), placed on the walls of the production area of Oversonic Robotics.

Figure 3.23: MarvelMind beacons kit

Each beacon sends and receives a stream of hypersonic signals continuously. There is also a modem, which is connected to the PC on which the supplied software is run, and a beacon called a 'hedgehog' which is placed on the vehicle to be located, in this case Robee. The hedgehog then receives the signals from the four beacons and sends them to the modem, which proceeds to triangulate. The communication frequency is customisable and directly affects localisation accuracy, which in the basic configuration is claimed to be +/- 2 cm.

4 | Navigation Stack

4.1. Introduction

In this chapter, we will address SLAM (simultaneous localization and mapping) and navigation techniques used in this thesis. We will focus exclusively on the literature overview of techniques used in practice, without comparing the various existing approaches. We will therefore overview all the components of the see-plan-act architecture. The sense-plan-act architecture in fact explains the entire process that starts from the map building, to the global and local planning up to sensor fusion. It is indeed composed of:

- Map
- Sensors
- Current Position
- Goal Position
- Trajectory Planning
- Trajectory Following and Obstacle Avoidance

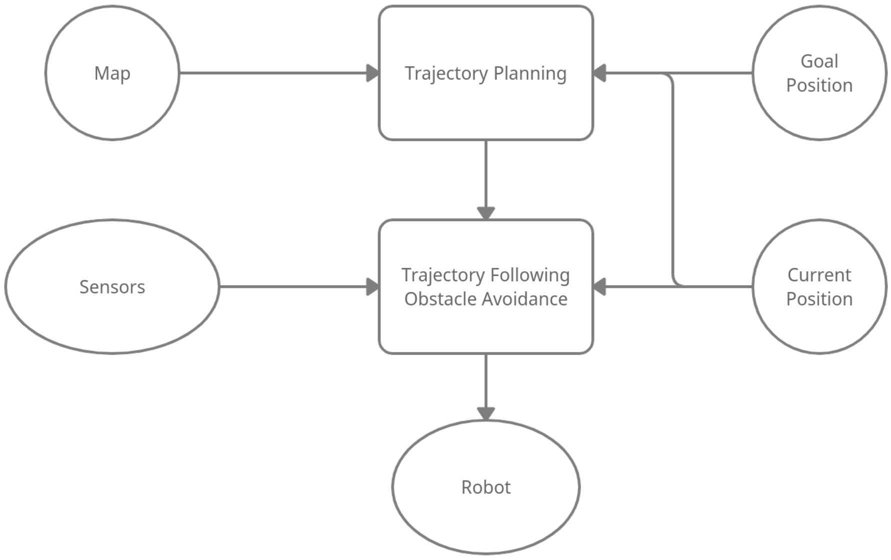


Figure 4.1: Block scheme representation of sense plan act architecture

4.2. Simultaneous Localization and Mapping

In this first subsection we will focus on map side, understanding how maps are build, interpreted and how the agents localize inside the map. There exists two main types of representing a map:

- Landmark-based: a particular type of representation, mainly used for localization, that is based on detecting landmarks (?). This technique results in a sparse representation of the space, leaving much to the unknown;
- Grid maps: the map results in a discredited version of the environment, where each cell contain information about occupation/non occupation/unkwown. It results in a very dense representation where almost the totality of the cells are caught.

In the scope of this work we will focus on occupancy grid map.

4.2.1. Occupancy Grid Map

As anticipated, occupancy grid map is a peculiar map representation that attempts to discretize the continous environment into a two dimensional grid map. The grid map is again divided into array cells of size from 5 to 50 cm and each of them hold a probability value that stands for the likelihood to be free or occupied. Thus, occupancy grid maps try to solve the problem of reconstructing consistent maps from noisy and uncertain measurement data, under the hypothesis of knowing the robot pose. The reasoning behind most occupancy grid mapping algorithm is to calculate the posterior over maps, given the data, in a probabilistic way.

$$p(m|z_{1:t}, x_{1:t}) \tag{4.1}$$

where m is the map,$z_{1:t}$ is the set of measurements up to time t and $x_{1:t}$ the set of all the poses taken by the robot, namely its path. Being m_i the i-th grid cell and having it a binary occupancy value that states if a cell is free or occupied (1 for the cell being occupied, 0 free), we can define:

$$m = \sum_i m_i \tag{4.2}$$

.

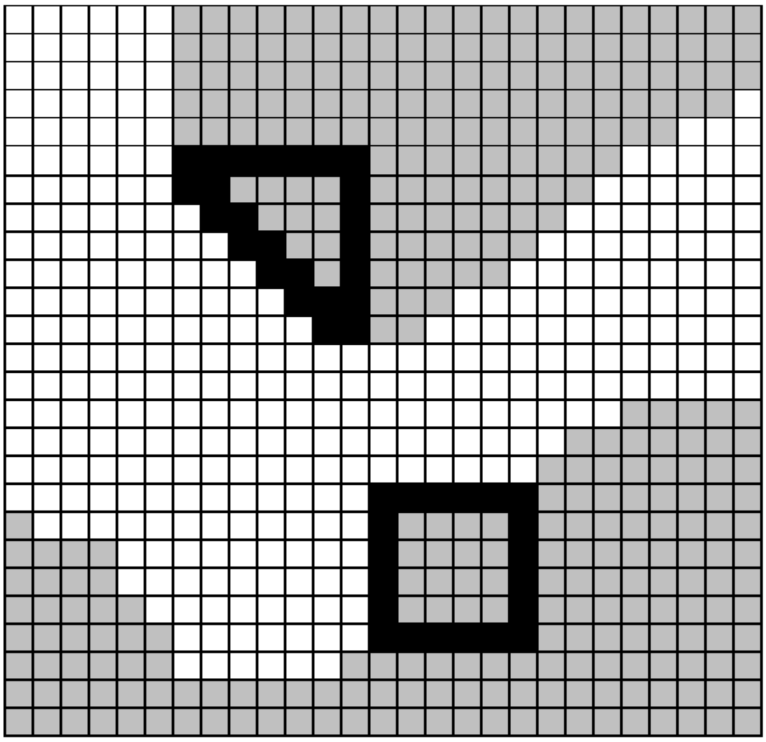


Figure 4.2: Sample of an Occupancy Grid

Due to the curse of dimensionality, the probabilistic approach reduces to estimate the single cell occupancy rather than the entire map:

$$p(m_i|z_{1:t}, x_{1:t}) \tag{4.3}$$

To calculate the single cell occupancy we resort to Bayes rule:

$$p(m_i|z_{1:t}, x_{1:t}) = \frac{p(z_t|m_i, z_{1:t-1}, x_{1:t})p(m_i|z_{1:t-1}, x_{1:t})}{p(z_t|z_{1:t-1}, x_{1:t})} \tag{4.4}$$

Additionally recurring to Markov assumption, stating that the current state depends on only a finite fixed number of previous states, measurement z_t depends only on x_t and m_i. It is common use at this point to adopt log-odds representation of occupancy, as to avoid being with probabilities close to 0 and 1:

$$l_{t,i} = \log \frac{p(m_i|z_{1:t}, x_{1:t})}{1 - p(m_i|z_{1:t}, x_{1:t})} \tag{4.5}$$

The process loops and assumes the form of the algorithm 4.1. It is important to note that only the cells which fall under the sensor cone of measurement are updated through the inverse sensor model, Thrun et al. [25].

Algorithm 4.1 Occupancy Grid Algorithm

Algorithm occupancy grid mapping($l_{t-1,i}, x_t, z_t$)
for all cells m_i **do**
  if m_i in perceptual field of z_t **then**
    $l_{t,i} = l_{t-1,i} + inverse_sensor_model(m_i, x_t, z_t - l_0)$
  else
    $l_{t,i} = l_{t-1,i}$
  end if
end for
return $l_{t,i}$

where the inverse sensor model is defined as follows:

$$inverse_sensor_model(m_i, x_t, z_t) = p(m_i|z_t, x_t) \tag{4.6}$$

The motivation for the "inverse" denomination is because it reasons from effects to causes: it provides an information about the world where that same information was derived from a measurement caused by the world it self:

$$p(m_i|z_{1:t}, x_{1:t}) = \eta \int m : m(i) = m_i p(z|x, m)p(m)dm \tag{4.7}$$

A function approximator has to be used, since this algorithm cannot be computed due to the large map space.

4.2.2. SLAM algorithm

At this point we turn to the problem of SLAM, Simultaneous Localisation and Mapping. This stems from the robot's need to map a new environment, of which nothing is known, and at the same time to localise the robot itself within the map being created. The problem is particularly difficult as one does not have access to the robot's poses and uncertainty is kept on all the components. Moreover, it proposes to correct both odometry and uncertainty of estimated position and landmark. Two approaches to SLAM can be defined, from a probabilistic point of view:

- Full SLAM: simultaneous estimate of path and map
- Online SLAM: simultaneous estimate of the most recent pose and map

Full SLAM Full SLAM addresses the problem of estimating the joint probability of the entire trajectory and landmark.

$$p(x_{1:t}, m|z_{1:t}, u_{1:t}) \tag{4.8}$$

For this purpose we propose a significant example: FastSLAM. Fast Simultaneous Localization and Mapping uses a sampled particle filter distribution model, solving the full SLAM problem. If we consider the full trajectory X_t rather than a single pose x_t, the following holds:

$$p(X_t, m|z_t) = P(X_t|z_t)P(m|X_t, z_t) \tag{4.9}$$

where $P(X_t|z_t)$ is the estimate of the trajectory and $P(m|X_t, z_t)$ is the estimate of the map given the trajectory. Thus, in FastSLAM the trajectory X_t is represented by particles $X_t(i)$ while the map is represented by a factorization called Rao-Blackwellized filter. The approach so is to treat each pose particle as it it was the entire trajectory, processing all of the feature measurements independently.

$$P(m|X_t^{(i)}, z_t) = \prod_j^M P(m_j|X_t^{(i)}, z_t) \tag{4.10}$$

Indeed, once the trajectory is known, all of the features become uncorrelated.

$$p(x_{1:t}, l_{1:m}|z_{1:t}, u_{0:t-1}) = p(x_{1:t}|z_{1:t}, u_{0:t-1})p(l_{1:m}|x_{1:t}, z_{1:t}) \tag{4.11}$$

where we have SLAM posterior, robot path posterior and landmark positions respectively. Previous equation can be simplified by factorization as follows:

$$p(x_{1:t}, l_{1:m}|z_{1:t}, u_{0:t-1}) = p(x_{1:t}|z_{1:t}, u_{0:t-1}) \prod_{i=1}^{M} p(l_i|x_{1:t}, z_{1:t}) \tag{4.12}$$

In this way the dimension of state space is reduced making particle filtering possible:

$$O(N \times \log(M)) \tag{4.13}$$

with N being the number particles and M the number of map features

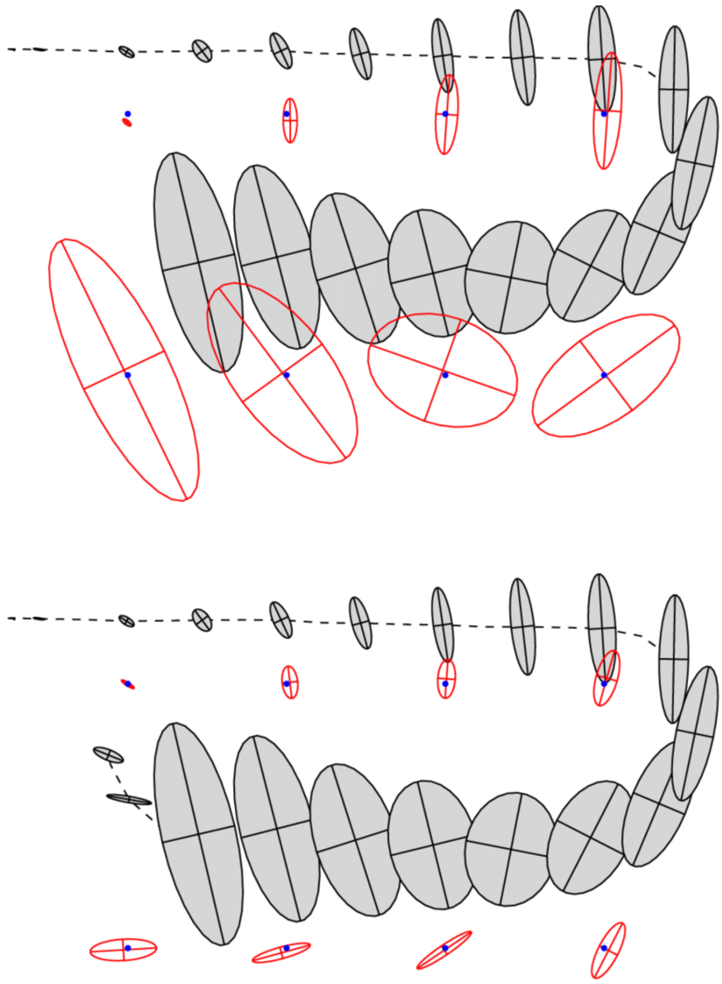


Figure 4.3: SLAM problem: initial uncertainty on pose and consequent decrease thanks to previously seen landmarks, Thrun et al. [24]

Summarizing, FastSLAM adopts a Rao-Blackwellized particle filtering based on landmarks, Montemerlo et al. [13], where each particle is a trajectory, each landmark is represented by a 2x2 EKF and therefore each particle has to maintain M EKFs.

Online SLAM Online SLAM entails estimating the posterior over the last pose along with the map: $p(x_t, m_i|z_{1:t}, u_{1:t})$ where x_t is the pose at time t, m is the map, $z_{1:t}$ $u_{1:t}$ the measurements available up to time t. The fact that we refer to this technique as online SLAM directly derives from the fact that we're considering data at time t, estimating last pose only.

$$p(x_t, m|z_{1:t}, u_{1:t}) = \int\int ... \int p(x_{1:t}, m|z_{1:t}, u_{1:t})dx_1 dx_2 ... dx_{t-1} \tag{4.14}$$

As one can notice, the online SLAM problem is the result of integrating out one at a time past poses from the full SLAM problem. A significant example of a proposed solution to Online SLAM is proposed: EKF SLAM. Extended Kalman Filter Simultaneous Localization and Mapping uses a linearized Gaussian probability distribution.

Algorithm 4.2 Extended Kalman Filter

Extended Kalman filter$(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t)$
$\mu' = \mathrm{g}(u_t,\ \mu_{t-1})$
$\Sigma_t' = G_t \Sigma_{t-1} G_t^T + R_t$
$K_t = \Sigma_t' H_t^T (H_t \Sigma_t' H_t^T + Q_t)^-1$
$\mu_t = \mu_t' + K_t(z_t - h\mu_t)$
$\Sigma_t = (I - K_t H_t)\Sigma_t'$
return μ_t, Σ_t

EKF-SLAM promises good performances but it has two main drawbacks: it employs linearized models of non-linear motion and observation models, inheriting many caveats; it is computationally demanding. One possible solution to this problem is the above reported FastSLAM (Rao-Blackwellisation filter).

4.2.3. SLAM Toolbox

During the experience at Oversonic Robotics, the SLAM toolbox was chosen for the simultaneous localisation and mapping problem, though many predecessors exist. The ROS packages responsible of SLAM can be divided into Bayes-based filter implementations, like GMapping and HectorSLAM, and graph-based implementations, as Cartographer and Karto SLAM. The SLAM Toolbox package is an open source software developed by Steve Maceski which use graph based approach and occupancy grid map. It has been widely used on the various ROS distros and has become the default SLAM algorithm for ROS2. It arose from the need to build accurate maps of large environments, where previous SLAM tools had shown shortcomings.
SLAM Toolbox provides three operating modes, Macenski and Jambrecic [12]:

- Synchronous Mapping: provides the ability to map and localize in an environment while keeping a bunch of measurements to be added to SLAM. This results useful when the quality of the map is important.
- Asynchronous Mapping: on the contrary, this mode manages new measurements only when previous measurement has been completed. This makes this modality useful when real time localization is crucial.
- Pure Localization: cannot detects changes in the space. It tries to match a local bunch of measurements with the data originally gathered.

——————-COSA AGGIUNGERE ANCORA??————————

4.3. Global Planning

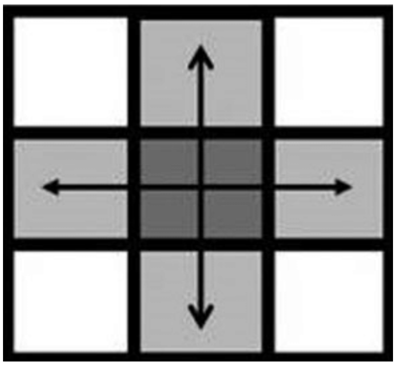
Mobile robots are meant to move from their current positions to some goal inside the map. Once that the SLAM task has been performed and a map has been obtained, we can address the This is known as trajetory planning and it is managed by the so-called global planner. Robot motion planning goals are:

- collision-free trajectories
- most efficient or most optimal (depending on the chosen optimality criterion) trajectory

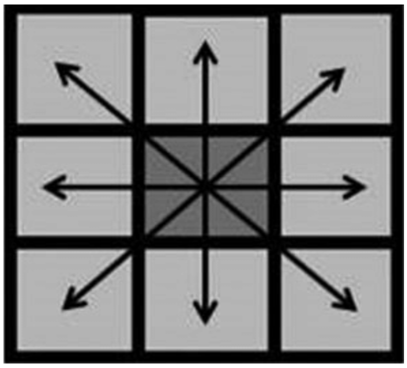
The problem that global planner addresses regards finding a collision free path between an initial pose and the goal, taking into account the existing constraints. It is important to distinguish between some concepts used in this scope:

- Path: a geometric locus of way points
- Trajectory: a path for which a temporal law is specified
- Manouver: a series of actions that a vehicle should execute

In the scope of this thesis we are going to analyze a path planning algorithms, the so-called A*. This algorithms is part of the graph based planning family. The underlying idea is to construct a discretized representation of the map, building a graph out of it (4 or 8 neighbors connectivity are possible) and eventually searching for the shortest path in the graph, namely the optimal solution. It is relevant to report that the resolution of the grid directly influences the accuracy of the plan: a more dense resolution will describe in a more complete way the map it is investigating, thus resulting in a deeper analysis of the possible paths.



(a) a



(b) b

Figure 4.4: Comparison of 4 connectivity (a) and 8 connectivity (b)

This connectivity scheme reproduces on a grid the kinematics motion a robot is supposed to do in reality, so that performing the path search on the grid is representative of how the robot would move in reality. The usual approach to search graphs for the optimal path

4.3.1. A* algorithm

The A* algorithm was developed on the basis of the Dijkstra algorithm m improving its performance and is therefore one of the most widely used in path finding and graph traversal today. The components of this algorithm are the two points (start and end point), the grid and the nodes. In this approach what is important is the cost of moving from one edge to the others. The predecessor of A*, Dijkstra algorithm, in fact focuses on the idea of cost: each part of the path has an intrinsic cost and the algorithm visits all of the existing edges trying to lower the overall cost. The algorithm manages a queue list where it keeps all the nodes that are still to analyzed, where the nodes with the smallest distance to the starting point is the first node in the queue, Herzog [7]. Every time we move to some node, we encounter other nodes that previously were unaccessible, since we are dealing with k connectivity framework, and if these nodes are still to be traversed they are added to queue. This process ends as soon as the priority queue is emptied, namely when there are no nodes left to be investigated. Every time the algorithm shifts to some new node it records the path that led to that node and the specific costs, so the shortest path to each node can be computed by going backwards in the path. A* is based on the best first search speeds up this process by splitting the cost into a function:

$$f(x) = g(x) + h(x) \tag{4.15}$$

where g(x) is cost of the shortest path from the starting point to the current node and h(x), the so-called heuristic function, is an estimate of the cost of the shortest path from the current state to the goal. The kind of heuristic is a matter of choice, still it needs to comply with the following three properties:

- Completeness: the algorithm is guaranteed to terminate when dealing with finite graphs having non negative edge weights.
- Admissibility: the heuristic never overestimates the cost of reaching the goal.

$$h(x) \leq h^*(x) \tag{4.16}$$

  where h*(x) is defined as the optimal cost to reach a goal from the current node.

- Consistency: the estimate of the algorithm is always less than or equal to the estimated distance from any neighbouring node to the goal, plus the cost of reaching that node.

Time complexity strongly depends on the heuristic and in its worst case (the case in which the search space is unbounded), the number of nodes exploded is exponential in the depth of the solution:

$$d : O(b^d) \tag{4.17}$$

where b is defined as the branching factor. In its best case (the search space is a tree and there exists only one goal) it would develop in a polynomial fashion, provided that the following condition on the heuristic holds:

$$|h(x) - h^*(x)| = O(\log h^*(x)) \tag{4.18}$$

where h* is the optimal heuristic. For this reason, a bounded relaxation is applied: it is possible to speed up the process by considering also approximate shortest paths. This process is bounded by a factor ϵ so that optimality suffers a decrease that is not greater than $(1 + \epsilon)$ times the optimal solution, computed without the hypothesis relaxation. Several possible algorithms exists for ϵ, below is reported as an example the Dynamic Weighting, Pohl [18]:
cost function is defined as

$$f(n) = g(n) + (1 + \epsilon w(n))h(n) \tag{4.19}$$

where w(n) is

$$w(n) = \begin{cases} 1 - \frac{d(n)}{N}, & \text{if } d(n) \leq N \\ 0, & \text{otherwise} \end{cases} \tag{4.20}$$

with d(n) being the depth of the search and N the anticipated lenght of solution. Below is reported the pseudocode of the A^* algorithm:

Algorithm 4.3 A algorithm

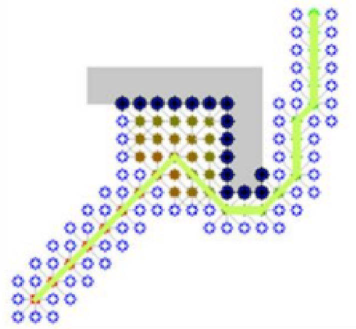
```
  Input: A graph G(V,E) with source node start and goal node end
  Output Least cost path from start to end
  open list = start
  closed list =
  g(start) = 0
  h(start) = heuristic_function(start, end) f(start) = g(start) + h(start)
while open list is not empty do
    m = Node on top of open list, with least f
    if m == end then
      return
    end if
    remove m from open_list add m to closed_list
for each n in child(m) do
      if n in closed list then
        continue
      end if
      cost = g(m) + distance(m,n)
      if n in open list and cost < g(n) then
        remove n from open list as new path is better
      end if
      if n in closed list and cost < g(n) then
        remove n from closed list
      end if
      if n not in open list and n not in closed list then
        add n to open list
        g(n) = cost
        h(n) = heuristic function(n, end)
        f(n) = g(n) + h(n)
      end if
end for
end while
return failure
```

It is interesting to note that the algorithm returns either the optimal plan, hence it is an exact algorithm.

Figure 4.5: A* initial problem: red point is the starting node, green point is the goal

(a) a

(b) b

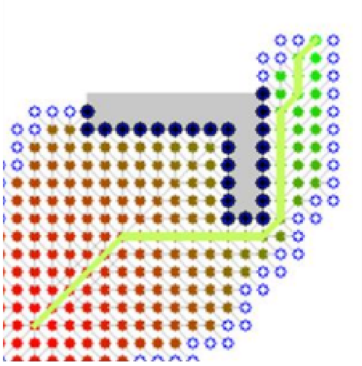


(c) c

Figure 4.6: Shortest path relaxing the admissibility criteria

(a) a

(b) b



(c) c

Figure 4.7: Optimal path obtained using the admissibility criteria.

4.4. Local Planning

A strong assumption on which global planning technique was based is the fact that the space surrounding the robot from its starting point to its goal is almost totally known. Nonetheless, robots moving in every environment must be able to deal with unforeseen changes and adapt to them. For this purpose, a local planner is paired with the global planner: while the latter is responsible of trajectory planning and works at a low frequency rate, the first one deals with trajectory following and obstacle avoidance at a much higher frequency rate. So, local planning is deemed to solve two tasks:

- Ensure path following
- Perform obstacle avoidance for objects that are not tracked in the map

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 qt 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
$q_t + \delta_t = f(u, q_t, \delta t), with \delta t \geq 0$. Let Q_t, T be the set of the configuration of the trajectory followed from q_t with $\delta t \in (0, T)$ a given time interval. T > 0 is called the sampling period. Indicating with $q_t arget$ a target configuration. Then, in time t_i the robot A is in $q_t i$, where a sensor measurement is obtained $S(q_{ti})$, and thus an obstacle description $O(q_t i)$." Siciliano and Khatib [21] The overall goal of the obstacle avoidance algorithm is to find a trajectory from that brings the robot closer to the goal in a non colliding way:
$A(q_{ti}, T) \cap O(q_t i) = \emptyset$
$f(q_{ti}, q_{target}) \leq F(q_{ti} + T, q_{target})$

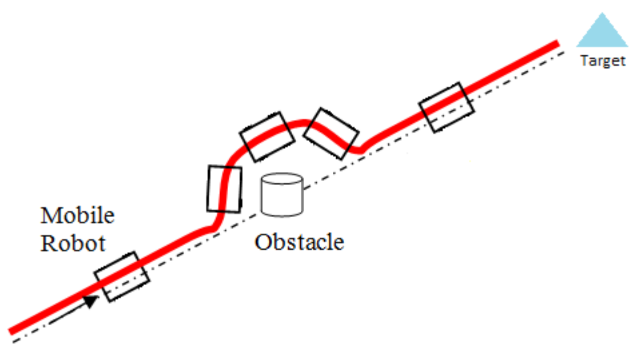


Figure 4.8: Sample of the most simple obstacle avoidance technique

In the following subsections an overview of the most famous local planning method will be provided, in particular:

- Vector Field Histogram
- Curvature Velocity
- Dynamic Window Approach

4.4.1. Vector Field Histograms

Vector Field Histogram was presented in 1991 by Borenstein and Koren [3] and ensured fast obstacle detection and collision avoidance, not requiring the vehicle to stop. It is composed of two steps, where the first one all the possible motions are evaluate and in the second one the best one is traversed.

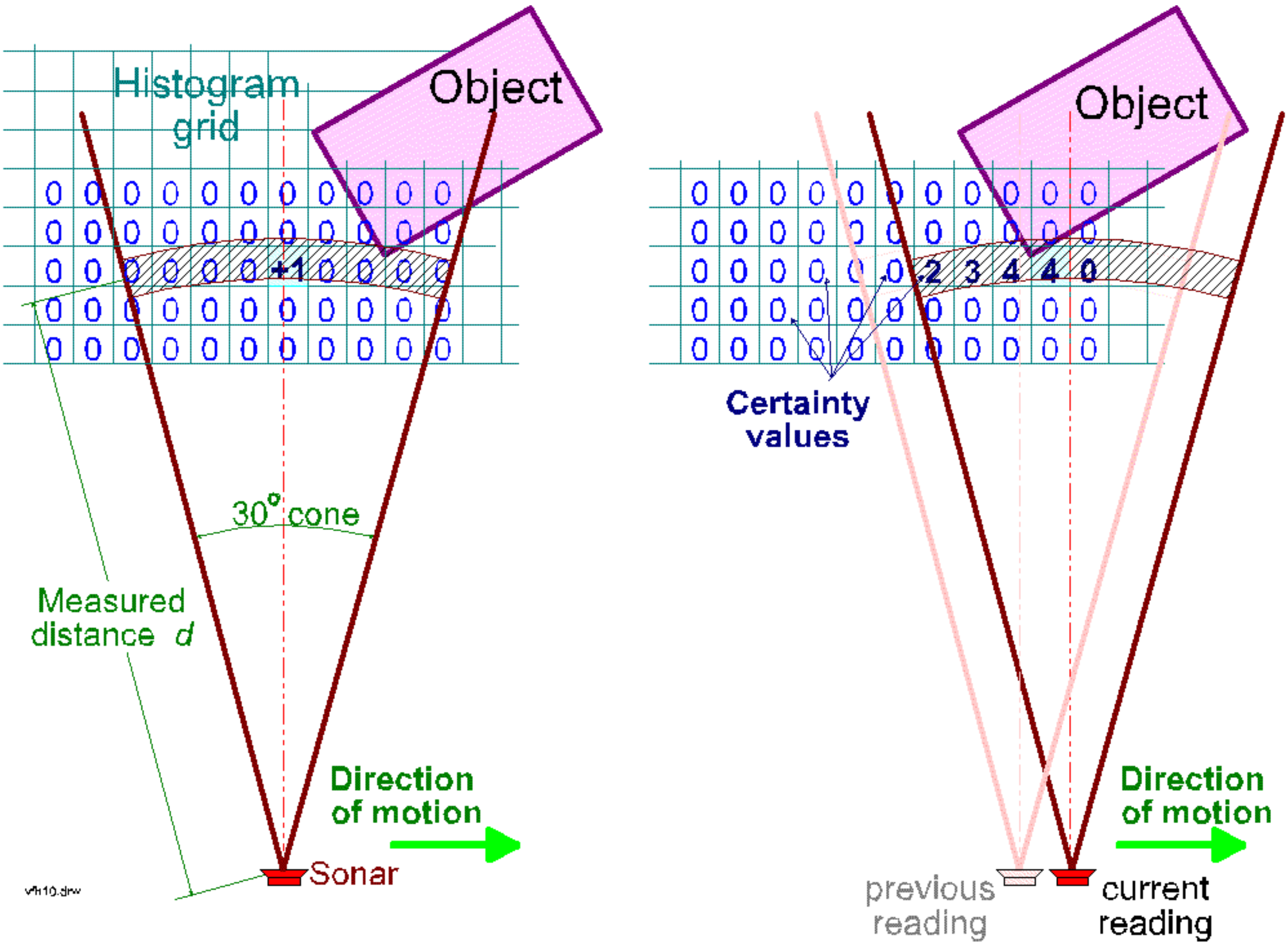


Figure 4.9: Vector Field Histogram

The Vector Field Histogram is based on the concept of virtual force field, a concept that resort somehow to the idea of imaginary forces acting on a robot, Khatib [10]. VFF is composed of:

- A two-dimensional Cartesian histogram grid for obstacle representation, where the grid is composed of cells defined by some coordinate (i,j) and $c_{i,j}$ holds the probability of the occupancy. A probability distribution is created by updating only one cell in the histogram grid for each range reading. More specifically, the function $h^k(q_{ti})$ describes the density of the obstacle, on turn proportional to the probability of point occupancy P(p) and to distance from the obstacle, that is to say that the more the distance increases, the lower is the density. The function $h^k(q_{ti})$ is defined as:

$$h^k(q_{ti}) = \int_{\Omega_k} P(p)^n \left(1 - \frac{d(q_{ti}, p)}{d_{max}}\right)^m dp \tag{4.21}$$

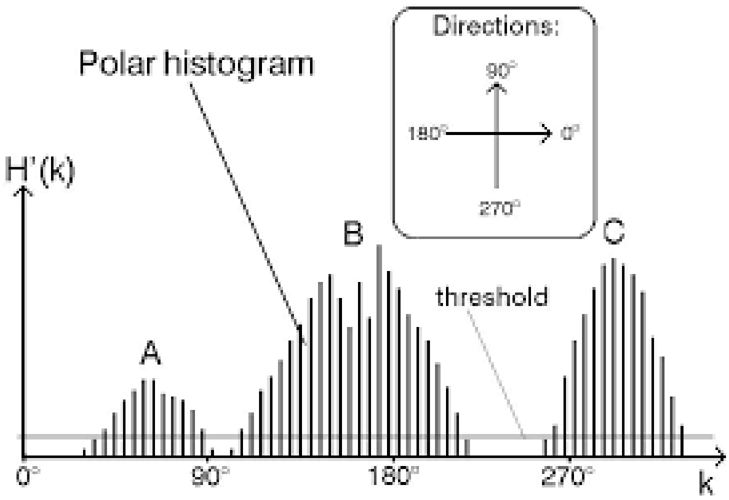


Figure 4.10: Vector Field Histogram

- Application of potential field idea to the histogram grid
- Previous two component are combined in real time enables sensor data to perform obstacle avoidance

All the directions ranged from the sensors are evaluated but only those that fall under the defined threshold are further investigated. A cost function is then established as follows:

$$G = \alpha target_direction + \beta wheel_orientation + \gamma previous_direction \tag{4.22}$$

where α represents the direction of the goal, β the smoothing of the action and γ the previous direction of motion. Every direction, falling under the threshold, is evaluated through the newly defined cost function, that becomes the selection method.

4.4.2. Curvature Velocity Methods

Curvature Velocity Methods were developed by Reid Simmons in 1996. This approach to obstacle avoidance treats the problem as a constrained optimization in the velocity space of the robot, rather than in Cartesian space. The robot is deemed to travel along arcs of circles rather than straight lines, still it cannot turn instantaneously. This method works by adding constraints to the velocity space, defined as the set of controllable velocities and choosing the point in that space that complies with all the constraints and maximizes an objective function, that balances speed, safety and goal-directedness, Simmons [23].

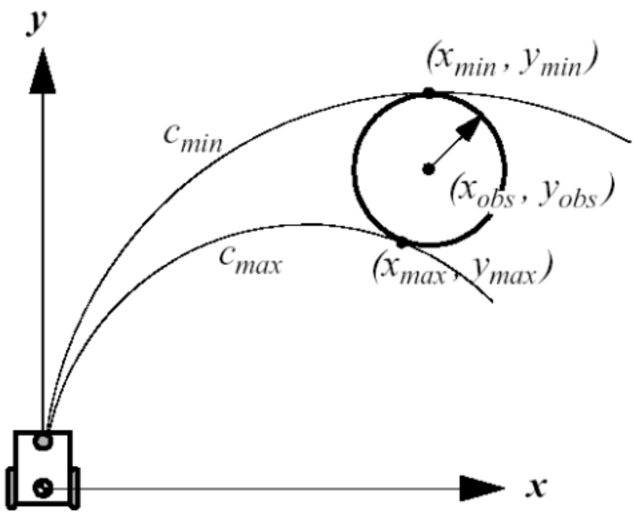


Figure 4.11: Curvature Velocity Methods

Only the interval of curvatures between c_{min} and c_{max} are considered, where the set of considered trajectories is obtained by using the curvatures tangent to obstacles to divide the velocity space into regions of constant distance.

$c_{min} = 2(x_{obs} - y_{obs})/\sqrt{x_{obs}^2 + y_{obs}^2 + r_{obs}^2}$ $c_{min} = 2(x_{obs} - y_{obs})/\sqrt{x_{obs}^2 + y_{obs}^2 + r_{obs}^2}$

4.4.3. Dynamic Window Approach

The dynamic window approach is an obstacle avoidance technique developed by Dieter Fox, Wolfram Burgard and Sebastian Thrun in 1997. In the DWA approach, the search for commands to control the robot takes place directly in velocity space. Compared to the previously seen method, the robot's dynamics are also integrated, thus further constraining the velocity search space to those that respect the dynamics constraints and are safe with respect to the obstacle. The process can be divided into search space and optimization, Fox et al. [6]. The search space of the possible velocities is reduced in three points:

- Circulare Trajectories: only circular trajectories are considered, determined by pair of (v,w) translational and rotational velocities
- Admissible Velocities: restriction to consider only safe velocities
- Dynamic Window: further restriction that applies to the admissible velocities, selecting only those that can be reached within a short time interval, respecting the constraints on acceleration.

The dynamic windows search space reduces to $V_r = V_s \cap V_a \cap V_d$

Optimization step proposes to maximize the objective function

$$G(v,w) = \sigma(\alpha heading(v,w) + \beta dist(v,w) + \gamma vel(v,w)) \tag{4.23}$$

where *heading* is a measure of progress towards goal location, *dist* is the distance towards the closest obstacle and *vel* is the the foward velocity of the robot.

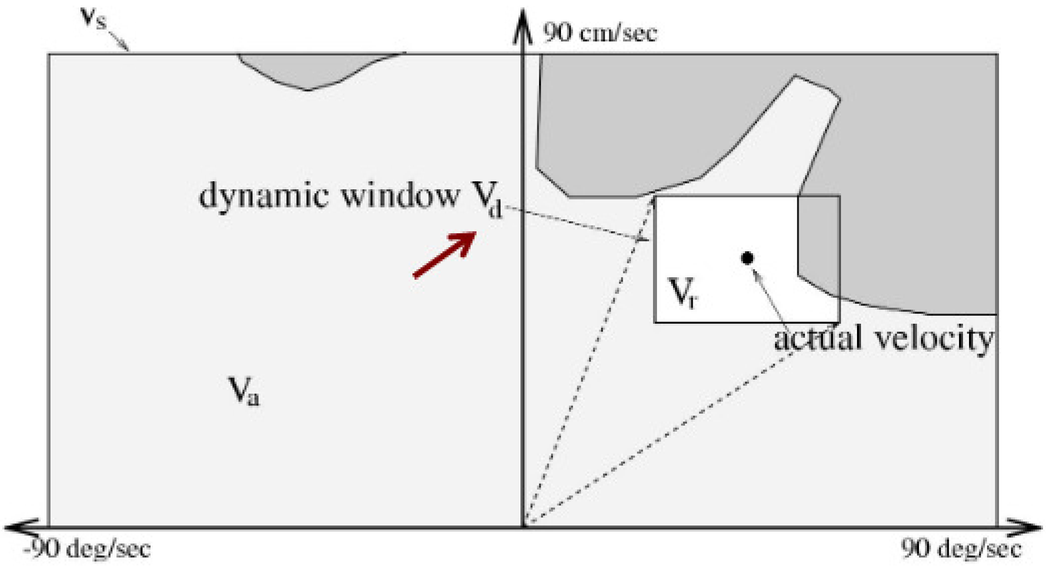


Figure 4.12: Dynamic Window

Algorithm 4.4 DWA algorithm

BEGIN DWA(robotpose, robotGoal, robotModel)
desiredV = calculateV(robotPose,robotGoal)
laserscan = readScanner()
$allowable_v$ = generateWindow(robotV, robotModel)
$allowable_w$ = generateWindow(robotW, robotModel)
for each v in $allowable_v$ **do**
  for for each w in $allowable_w$ **do**
    dist = $find_dist$(v,w,laserscan,robotModel)
    breakDist = calculateBreakingDistance(v)
    if (dist > breakDist) **then**
      heading = hDiff(robotPose,goalPose, v,w)
      clearance = (dist-breakDist)/(dmax - breakDist)
      cost = costFunction(heading,clearance, $abs(desired\_v - v)$)
      if (cost > optimal) **then**
        $best_v = v$
        $best_w = w$
        optimal = cost
      end if
    end if
  end for
end for
set robot trajectory to $best_v, best_w$

Contribution

This section contains the contribution of this thesis and attempts to summarise the developments and results obtained This section consists of:

- **Chapter 5: Simulator and Testing Platform**
- **Chapter 6: Pointcloud Filter**

5 | Simulator and Testing Platform

5.1. Introduction

Part of the work carried out at Oversonic Robotics was the development of a simulator and testing platform. The simulator, described in section 5.2, is primarily intended to be a safe and reliable test environment where the new navigation algorithms can be tested. It is not intended to have a 1:1 simulation with the real robot, which is outside the scope of this thesis. As already mentioned, building robots means dealing with both the software and hardware side, the two components are inseparable and one does not exist without the other. Dealing with the hardware part of the robot is a source of danger for people and the environment as software bugs can cause damage. In addition, testing a robot's behaviour in real life, the results of which are unknown, can cause mechanical damage and consequently much time would be lost in repair. Another not inconsiderable motivation lies in the wear and tear of the robot and its components: the batteries, for example, have a predefined life cycle and must be recharged in any case. All these reasons support the development of a simulated environment to test new developments before implementing them on the real robot. The testing module, described in section 5.3, addresses the need to have an indication of navigation performance. This need comes both from within the company, where it is important to evaluate how developments are improving it, and from outside, where the various stakeholders and customers can evaluate the project and understand how they can exploit the technology developed by Oversonic. Not only this, as reported by Dhillon [5], among the many types of tests, two of them are significant:

- reliability: obtain knowledge of failure occurrence patterns
- performance: building reliable indicators on performance status

Tests are moreover meant to have the following features:

- Accuracy: whether the test can actually measure what it sets out to measure
- Resolution: how precisely it measures the proposed feature
- Repeatability: how repeatable it is, so that test performed over time are comparable

5.2. Simulator

5.3. Testing Module

5.3.1. Algo/Code

5.3.2. Path definition

The path created involves passing through a 1 m gap.

Waypoint navigation (used to impose a path on the robot. A new waypoint is sent as a goal to the robot when it arrives at a distance of less than one metre from the last received waypoint)

The following values of interest are given for each measurement configuration:

- total average speed average speed over the entire path [m/s]
- fw average speed average speed while travelling (excluding on-site rotations) [m/s]
- rotational average speed rotational average speed [rad/s]
- nav distance total distance calculated by odometry [m]
- navigated time total time spent [s]
- moving forward time forward time [s]

Figure 5.1: Map visualization

Waypoint Coordinate	x [m]	y [m]
WP1	2.1	6.3
WP2	4.2	5.7
WP3	4.2	3.1
WP4	3.9	-2.2
WP5	1.9	-2.5
WP6	-0.2	-2.4
WP7	0.0	1
WP8	1.2	3.4
GOAL	0.5	6.6

Table 5.1: Waypoints and Goal coordinates

Settings	Robot: R012	
τ [-]	5	
max. acc [m/s^2]	0.9	at speed = 0.6 [m/s]
	1.0	at speed = 0.7 [m/s]
	1.05	at speed = 0.8 [m/s]
slip [-]	0.9	if speed <[0.8 m/s]
	0.8	if speed >= [0.8 m/s]

Table 5.2: Robot Configuration

	AVG SPEED [m/s]	ROT AVG SPEED [rad/s]	NAV DISTANCE [m]	MOVING FW AVG SPEED [m/s]	NAV TIME [s]	MOVING FW TIME [s]
AVG						
MIN						
MAX						
VAR						

Table 5.3: Requested Speed = [m/s], n° of measurements = , configuration

	Avg Speed [m/s]	**Rot avg speed [rad/s]**	**Nav Distance [m]**	**Fw avg speed [m/s]**	**Nav time [s]**	**Fw mov time [s]**
Requested Speed = 0.6 [m/s]						
AVG	0,48	0,54	0,73	25,00	0,63	52,73
MIN	0,39	0,52	0,49	24,69	0,62	49,89
MAX	0,50	0,55	0,82	25,57	0,63	64,69
VAR	0,0015	0,0001	0,0121	0,0883	0,0000	28,0762
Requested Speed = 0.7 [m/s]						
AVG	0,57	0,64	0,82	25,29	0,73	44,01
MIN	0,56	0,63	0,70	25,11	0,72	43,64
MAX	0,58	0,65	0,90	25,42	0,74	44,59
VAR	0,000062	0,000067	0,004414	0,017557	0,000067	0,091824
Requested Speed = 0.8 [m/s]						
AVG	0,64	0,73	0,84	25,57	0,83	39,93
MIN	0,60	0,72	0,73	25,35	0,82	38,59
MAX	0,66	0,75	0,94	26,00	0,84	41,99
VAR	0,0004	0,000081	0,004781	0,063281	0,000048	1,482262

Table 5.4

Features: The automatic relocation algorithm using fiducial tags of known position on the map is also tested. There are no fixed or moving obstacles other than those already mapped in order to take a snapshot of the current status and robustness of navigation under 'stable' conditions

$$NP = W * X = \begin{bmatrix} w_1 & w_2 & w_3 & w_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \tag{5.1}$$

$$W = \begin{bmatrix} 15 & 30 & 30 & 25 \end{bmatrix} \tag{5.2}$$

$$x_1 = \frac{NavigatedDistance}{PathDistance} \tag{5.3}$$

$$x_2 = \frac{ForwardAverageSpeed}{RequestedSpeed} \tag{5.4}$$

$$x_3 = \frac{TargetNavigationTime}{NavigationTime} \tag{5.5}$$

$$x_4 = \frac{TargetRotationTime}{RotationTime} \tag{5.6}$$

$$NP = \frac{w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + w_4 * x_4}{w_1 + w_2 + w_3 + w_4} = \frac{15 * x_1 + 30 * x_2 + 25 * x_3 + 30 * x_4}{100} \tag{5.7}$$

$$a_{21} = -\frac{k_s}{J_1} \quad a_{21} = -\frac{k_s}{J_1} a_{21} = -\frac{k_s}{J_1} \tag{5.8}$$

PROBLEMS ENCOUNTERED AND FUTURE DEVELOPMENTS

Depth camera: occasionally the camera sees ghost shadows (already known problem, but limited under the lighting conditions in the test environment).

Automatic relocation: only works correctly when the robot is static. It usually happens that the robot arrives at the tag rotating and as soon as it is localised it tends to consistently correct the position but not the angle: it is planned to insert position control logic (to be corrected only if inaccurate) and to stop the robot if localisation is necessary (only when the robot is stationary can the position be re-initialised)

Robot fall: caused by wheel climbing up the side of the ramp. It is assumed that motor current peaks will be checked so that dangerous situations can be recognised in advance.

OBSERVATIONS

It can be seen that by going from 0.7 to 0.8 m/s, the overshoot in terms of rotation after turns increases. In fact, the slip data has been modified, with a clear improvement in navigation reliability (more stable trajectories away from objects)

5.3.3. Current Measure

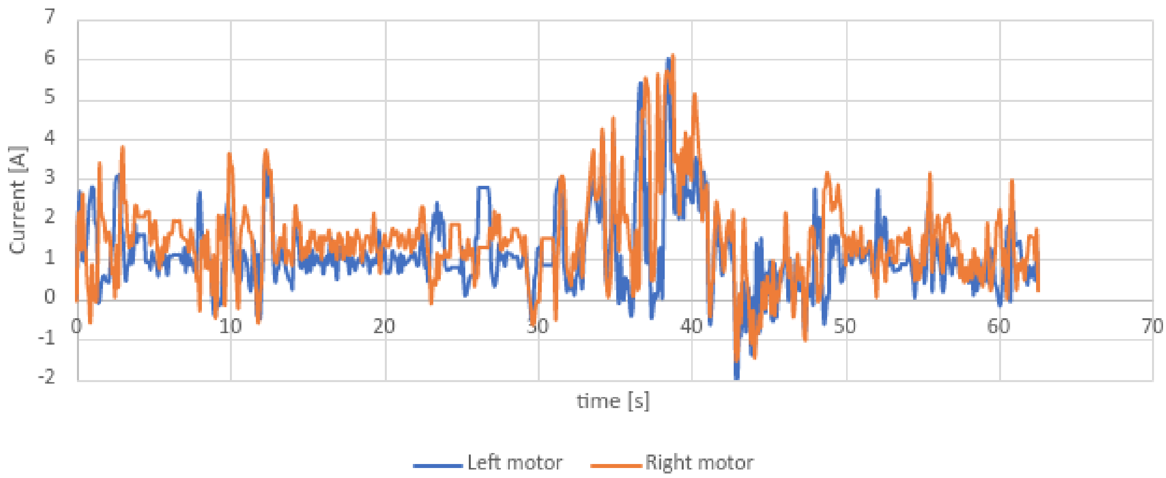


Figure 5.2: Left and Right Motor Current plot

Hypotetical Consumption	[Wh]	[Ah]
1 hours	124,0	2,6
8 hours	991,7	20,7

Table 5.5: Energy Consumption

Total Time [s]	62,6
Avg. Absorption [Ah/s]	0,001
Avg. Consumption [Wh/s]	0,034

Table 5.6: Average Absorption and Consumption

6 | Pointcloud Filter

6.1. Introduction

As introduced in chapter 3, point cloud is a data structure used to represent a collection of multi-dimensional points and is commonly used to represent three-dimensional data. In a 3D point cloud, the points usually represent the X, Y, and Z geometric coordinates of an underlying sampled surface. When color information is present, the point cloud becomes 4D. Point clouds can be acquired from hardware sensors such as stereo cameras, 3D scanners, or time-of-flight cameras, or generated from a computer program synthetically, Rusu and Cousins [20]. In the scope of this thesis, Realsense camera D455 was used in order to perform point cloud generation that was in turn used to generate the occupancy grid of the space around the robot. In figure 6.1 are reported the features of D455 camera.

Sensor Technology	Global Shutter
Ideal Range	from 0.6 m to 6 m
Depth	Technology: Stereoscopic
	Field of View: 87° × 58°
	Output Resolution: up to 1280 × 720
	Accuracy: < 0.02 m at 4 m
	Frame rate: up to 90 fps
RGB	Technology: Global Shutter
	Field of View: 90° × 65°
	Output Resolution: up to 1280 × 800
	Sensor Resolution: 1 MP
	Frame rate: up to 30 fps

Table 6.1: Highlighted camera's specifications.

The sensor transmits the pointcloud data that are interpreted by ROS in a dedicated format. The PointCloud2 object is an implementation of the `sensormsgs/PointCloud2` message type in ROS. The object contains meta-information about the message and the point cloud data. To access the actual data, use readXYZ to get the point coordinates and readRGB to get the color information, if available. For the pointcloud use in ROS and visualization in RViz environment, two ROS nodes were set up:

- read: reads a point cloud from the camera sensor, publishing it as `sensor_msgs/PointCloud2` message. This message contains a collection of N-dimensional points. The data may be in one dimension, so unordered, or in two dimensions, like images data are stored.
- write: subscribes to the topic `sensor_msgs/PointCloud2`

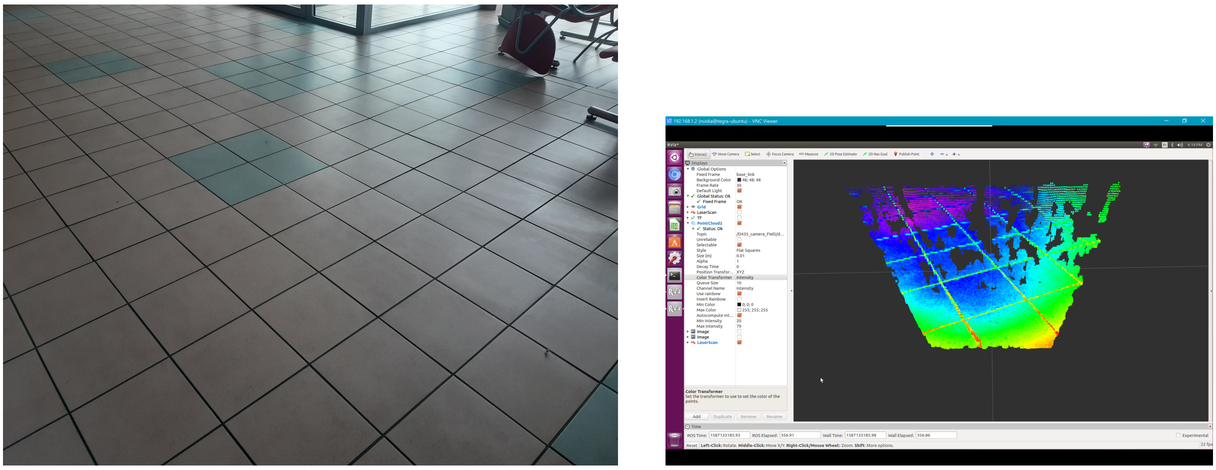
In order to manage pointcloud data, PCL library was adopted. The Point Cloud Library (PCL) is a standalone, large scale, open project for 2D/3D image and point cloud processing, that provides advanced methods for pointcloud data management and API integration. PCL makes us if a slightly different data class, pcl::PointCloud. This is the core point cloud class, though having a similar structure to that used in ROS, this enabling a straightforward conversion from PointCloud2 to PCL and viceversa. The motivation for this data class to exist stays in the fact that it enables nodes to work with individual data point as objects rather than with their raw data.

6.2. Problem Explanation

Part of the work at Oversonic Robotics involved testing the state of navigation, as seen in the previous chapter, in order to draw up useful reports both for internal benchmarking and to provide to customers. During these tests, it was realised that, in certain particular indoor and/or outdoor environment conditions, the robot perceived phantom obstacles, which did not exist in reality. This is critical as the objective for the robot's navigation is to perform the programmed path optimally and in the shortest possible time. It is therefore clear that if the local planner encounters these phantom obstacles, a great deal of time is lost and the robot travels a greater distance, also resulting in greater energy consumption. It was therefore decided to devote time to identifying and solving this problem. The conditions that encountered particular problems are:

- reflective surfaces
- tiled floor with highly repetitive patterns

In figure 6.5, we can appreciate how the reflective zones of figure a are interpeted as dark areas in figure b, where it is shown RViz GUI.



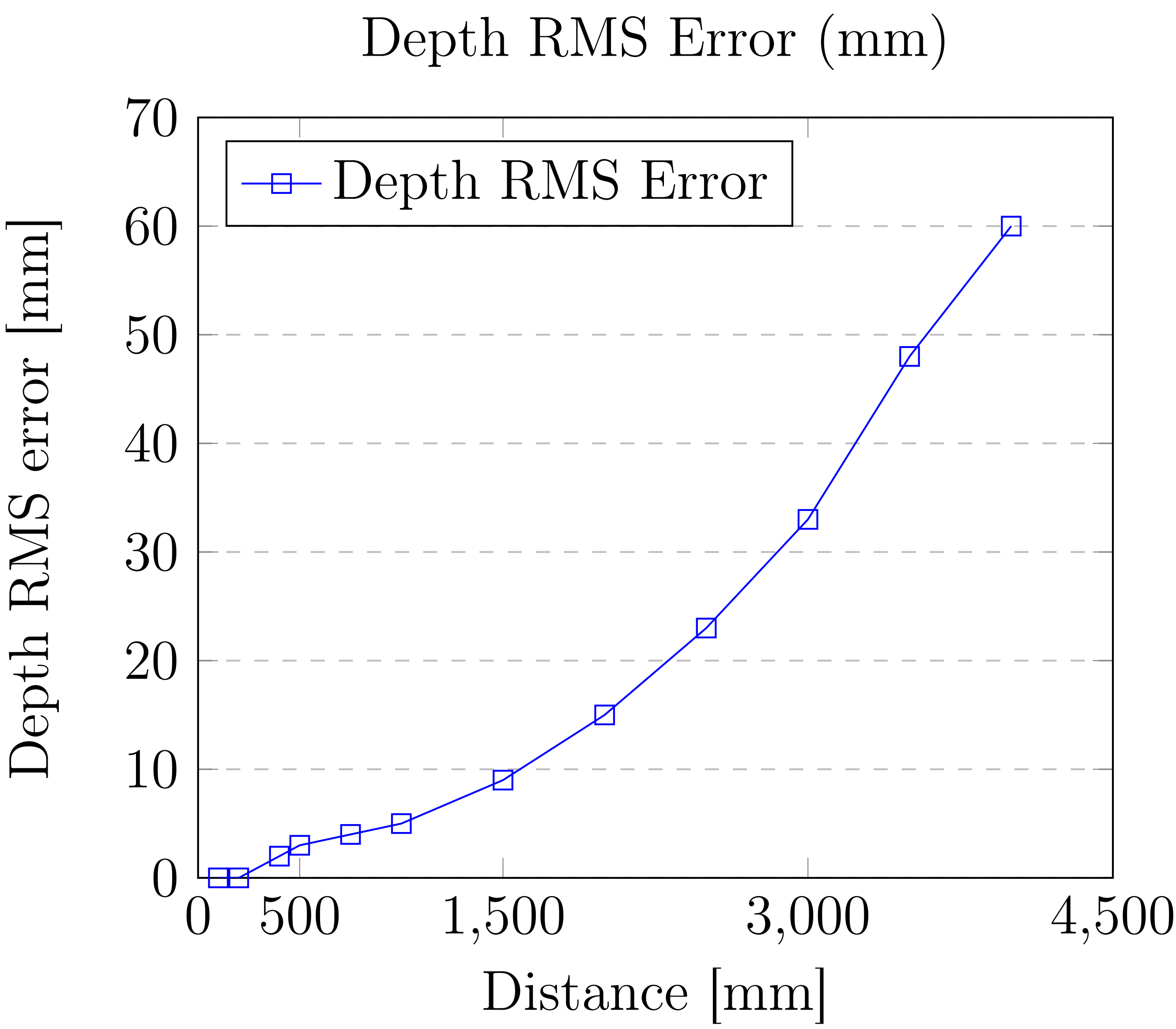
(a) a (b) b

Figure 6.1: Visualization of the reflection issue experimented at Oversonic's office tiled floor

After careful research, it was discovered that many users of the intel realsense chambers were experiencing this problem but no solution was provided by Intel. With the Intel RealSense D400 series of stereo depth cameras, depth is derived primarily from solving the correspondence problem between the simultaneously captured left and right video images, determining the disparity for each pixel (i.e. shift between object points in left vs right images), and calculating the depth map from disparity and triangulation. The Depth RMS error that defines the depth noise for a localized plane fit to depth values can be defined as:

$$DepthRMSerror(mm) = \frac{Distance(mm)^2 \times Subpixel}{focallength(pixels) \times Baseline(mm)} \tag{6.1}$$

Below is reported the findings for D455 RMS error:



The curve is obtained usind D455 with HFOV=90 deg, Xres=1280, baseline=50 mm and subpixel=0.08. Initially, it was decided to try applying a polarising film to the outer lens of the camera. In figure a polarizer is used to reduce the glare of sunlight reflecting off a window by only passing P-polarized light. Polarizers indeed can be used to selectively attenuate different components of light in order to enhance human vision and photography. The motivation for the use of optical filters comes from Fresnel equations: by properly adopting the filter angulation at Brewster angle, all reflected S-light is polarized, so that all reflections are cancelled. Fresnel divided light into S- and P- polarization states, where S has the electric field normal to the plane of incidence and P has the electric field co planar with it. Glare can lead to local saturation of portions of the image and false depth can result from objects partially reflected from the surface. Glare-induced saturation is usually associated with bright light sources such as sunlight or light bulbs. Reflection-related false depth typically occurs with textured objects reflected on shiny surfaces with little texture

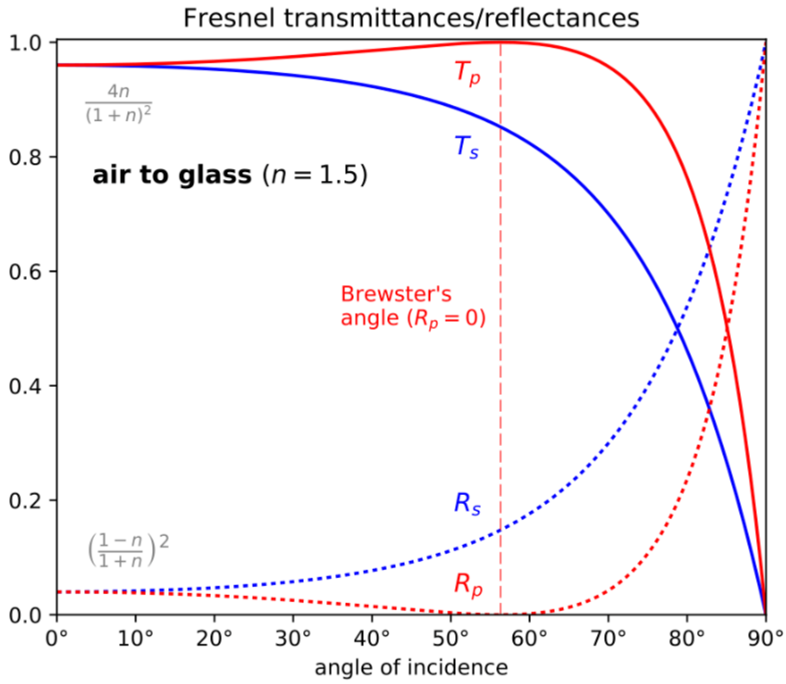


Figure 6.2: Fresnel transmittances/reflectances

Polarizers work well in condition of natural sunlight, as in figure 6.3, but in our working environment the optical filter showd no improvement on detecting ghost obstacles.

Figure 6.3: On the right there is the image taken from D400 RealSense camera, on the left there is the image taken after applying the physical filter on the lens.

In fact, even after applying the optical filter, the robot kept visualizing glare in form of black spaces, misleading obstacle detection.

In order to test the difference in performance with the application of the polarising film, a test identical to that already presented in Chapter 5 was used. In the two tables below, the navigation metrics before and after film application can be appreciated.

———INSERIRE TABELLE————

6.3. Proposed Solution

Since the attempted solution by application of an optical filter led to only marginal improvements, it was decided to resort to a post-processing solution. Intel RealSense SDK provides some buil-in methods to perform filtering, in particular:

- decimation: reduce resolution of depth frame
- disparity: transformation between depth and disparity domains
- spatial: edge-preserving smoothing
- temporal: filter depth data by looking into previous frames

Promising results were achieved by applying these filters to saved file of pointcloud data, but this approach did not allow to perform real time filtering, so it was left for an online post-processing solution. For this purpose, as already mentioned, the PCL library was used. This library provides methods for filtering pointcloud data. The approach was to treat ghost reflections as a mass of data too dense to be analysed and a statistical outlier removal filter was tried. In fact, the idea behind it is to remove noisy measurements from the pointcloud, as these scattered outliers corrupt the results. The solution therefore proposes statistical analysis around each point neighborhood: for every point compute the distance from the point itself to all of its neighbors. A similar approach was proposed by Ning et al. [16]: The sparse pointcloud P is defined by N points such that $P = p_1, p_2, p_3, ..., p_N$ whereas the k nearest neighbor points of a point p_i are defined as $KNN(p_i)$, a set of k points such that $Q = q_1, q_2, q_3, ..., q_k$. The algorithm removes points that are considered outliers based on geometric information.

The average geometric distance of a point p_i from its neighbors is defined as:

$$d_i = 1/k \sum_{j=1}^{k} dist(p_i, q_j) \tag{6.2}$$

where dist(x,y) is a function that returns euclidean distance between two points. By assuming that the resulted distribution is Gaussian with a mean and a standard deviation, we can derive μ_d the overall mean distance and σ_d, the standard deviation of distances. Hence, by tuning α, the so-called standard deviation multiplier, we can define a threshold $T = \mu_d + \alpha\sigma_d$. Every point in the cloud, whose its kNN distance falls out of the interval defined by T, are directly discarded. Tuning of α parameter was performed by trial and error. Below is reported the resulting algorithm, algorithm 6.2:

Algorithm 6.1 Statistical Outlier Removal

Set k
Set α
for P_i in input_data **do**
  Locate kNN to point P_i
  Compute d_i
end for
Compute μ_d
Compute σ_d
Compute $T = \mu_d + \alpha\sigma_d$
if d_i > T **then**
  Trim point P_i from the pointcloud
end if

———-LOCAL DENSITY ALGORITHM??——————

In figure 6.4 the overall process flow is described.

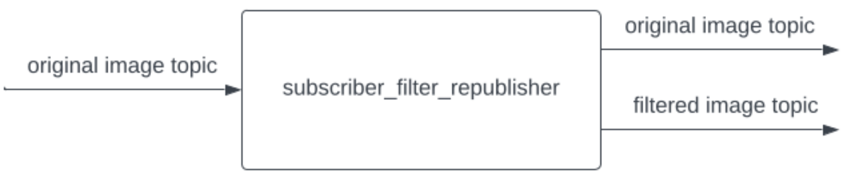


Figure 6.4: Overall process block scheme

As initially stated, the goal of this filter is to perform online processing of point cloud. In our specific case, we want to take input data, perform the data transformations and then output both the filtered and original data, so that the user can choose which to keep. Input data are transmitted from and to the ROStopic `/d400_lb/depth/color/points` through the subrscribe and publish paradigm. In order to work on the data with the PCL methods, it is necessary to transform the message structure from `PointCloud2` to

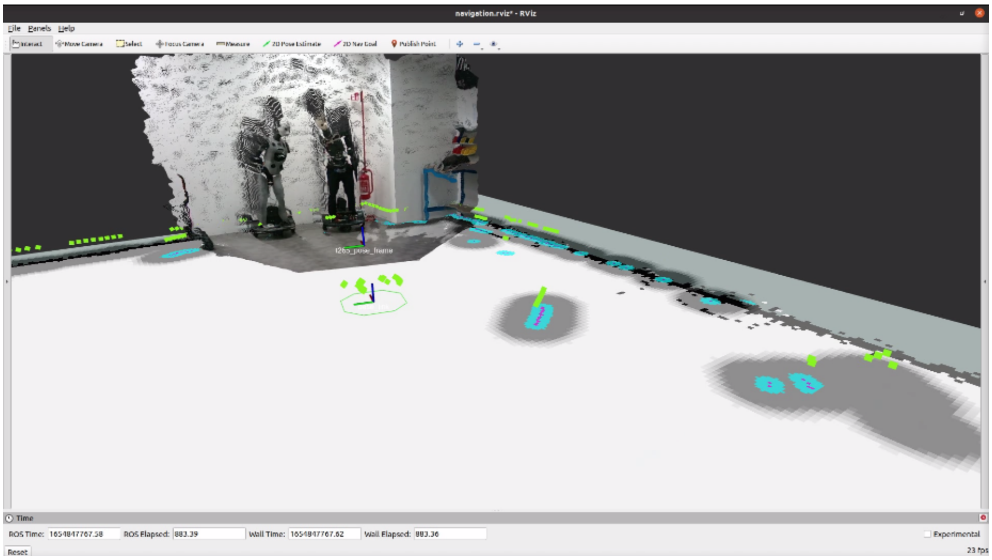
PCL format and transform back to `PointCloud2` after the filtering has occured. A crucial point for this approach is computing time, given the fact that we want the filter to process the data at real time. It was necessary to provide an indication of the time required for the filter to elaborate the data, in order to understand its practical feasibility. Initially, pre processing was made on the data and the filter was dealing with the entire amount of point cloud information at the same time. This led to an extremely high computing time (0.5 s) that is unacceptable for obstacle detection, in fact this brought to a further drop of navigation performance. It was thought that the problem was caused by a too strong filtering action, so we tried to decrease further the threshold. This was a misconcept, since after a careful analysis it was discovered that the delay was introduced by the conversion of data format from PointCLoud2 to PCL format. The only solution left at this point was to try and pre-process the data using a decimation filter, in order to feed the code with the least required amount of data. Decimation filter is has been designed by Intel RealSense to effectively reduce the depth scene complexity. The filter run on kernel sizes [2x2] to [8x8] pixels. For patches sized 2 and 3 the median depth value is selected. For larger kernels, 4-8 pixels, the mean depth is used due to performance considerations, Dev [4]. In order to understand which configuration worked better inside the computational time constraint, several tests were conducted. In table 6.2 average results are reported, key indicators are:

- Converting Time represents the 2-way cost of converting data: from PointCloud2 to PCL and from PCL back to PointCloud2
- Processing Time represents the computational cost of performing the statistical outlier removal on the input cloud

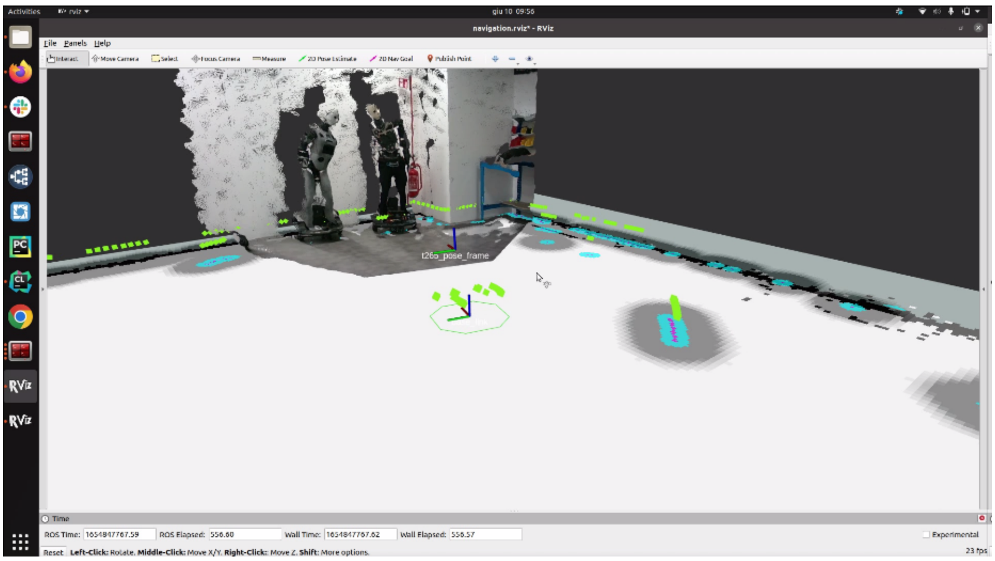
Configuration	**Converting Time [s]**	**Processing Time [s]**	**Total Time [s]**
Pre-filter: none Filter: K=50	0.42	0.13	0.55
Pre-filter: none Filter: K=9	0.25	0.09	0.34
Pre-filter: decimation Filter: K= 50	0.07	0.03	0.10
Pre-filter: decimation Filter: K= 9	0.04	0.01	0.05

Table 6.2: Comparison of average computational times implied by different filter - prefilter configurations

This data was collected and analysed by running the code on a camera detached from the robot. Once the configurations to be tested had been chosen, the physical robot was moved on. The idea was to define an initial benchmark, obtained from chapter 5, and to collect useful data to compare the same metrics with the chosen configurations. The aim was to analytically assess whether and to what extent the various configurations could improve the management of phantom reflections and the performance of navigation in general.



(a) a



(b) b

Figure 6.5: Figure a is pre filtering, figure b is post filtering.

.

Overall, the designed process flow resulted as reported in algorithm: 6.2:

Algorithm 6.2 Statistical Outlier Removal

```
Set magnitude of decimation filter
Set k
Set α
Subscribe pointcloudtopic
Input_data: pointcloud topic
for P_i in input_data do
    Convert from PointCloud2 to PCL::cloud
    Locate kNN to point P_i
    Compute d_i
end for
Compute μ_d
Compute σ_d
Compute T = μ_d + ασ_d
if d_i > T then
    Trim point P_i from cloud
end if
Output_data = cloud
for P_i in cloud do
    Convert from PCL to PointCloud2
end for
publish pointcloud topic and pointcloud_filtered topic
```

The complete code written in C++ can be found in appendix A.

6.4. Experimental Results

————–TABELLE CON DATI A SUPPORTO—————–

7 | Conclusions and future developments

A final chapter containing the main conclusions of your research/study and possible future developments of your work have to be inserted in this chapter.

Bibliography

[1] On hedgehogs and marvelous minds: A new technology for point data collection?

[2] I. Asimov. *Runaround*. 1942.

[3] J. Borenstein and Y. Koren. The vector field histogram - fast obstacle avoidance for mobile robots. *Robotics and Automation, IEEE Transactions on*, 7:278 – 288, 07 1991. doi: 10.1109/70.88137.

[4] I. R. Dev. Decimation filter. URL `https://dev.intelrealsense.com/docs/post-processing-filters`.

[5] B. S. Dhillon. *Robot Testing and Information Related to Robots*, pages 210–225. Springer New York, New York, NY, 1991. ISBN 978-1-4612-3148-6. doi: 10.1007/978-1-4612-3148-6_12. URL `https://doi.org/10.1007/978-1-4612-3148-6_12`.

[6] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics Automation Magazine*, 4(1):23–33, 1997. doi: 10.1109/100.580977.

[7] M. Herzog. Dikstra's algorithm. URL `https://www-m9.ma.tum.de/graph-algorithms/spp-dijkstra`.

[8] Intel. Robust visual-inertial tracking from a camera that knows where it's going. URL `https://www.intelrealsense.com/visual-inertial-tracking-case-study/`.

[9] E. T. Jonasson, L. Ramos Pinto, and A. Vale. Comparison of three key remote sensing technologies for mobile robot localization in nuclear facilities. *Fusion Engineering and Design*, 172:112691, 2021. ISSN 0920-3796. doi: https://doi.org/10.1016/j.fusengdes.2021.112691. URL `https://www.sciencedirect.com/science/article/pii/S0920379621004671`.

[10] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, volume 2, pages 500–505, 1985. doi: 10.1109/ROBOT.1985.1087247.

[11] P. Lima, D. Nardi, G. Kraetzschmar, J. Berghofer, M. Matteucci, and G. Buchanan. Rockin innovation through robot competitions [competitions]. *Robotics Automation Magazine, IEEE*, 21:8–12, 06 2014. doi: 10.1109/MRA.2014.2314015.

[12] S. Macenski and I. Jambrecic. Slam toolbox: Slam for the dynamic world. *Journal of Open Source Software*, 6(61):2783, 2021. doi: 10.21105/joss.02783. URL `https://doi.org/10.21105/joss.02783`.

[13] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. Fastslam: A factored solution to the simultaneous localization and mapping problem. 11 2002.

[14] B. G. Morgan Quigley and W. D. Smart. *Programming Robots with ROS*. O'Reilly Media, 1 edition, 2015 2015.

[15] MQTT.org. Mqtt.org. URL `https://mqtt.org/`.

[16] X. Ning, F. Li, G. Tian, and Y. Wang. An efficient outlier removal method for scattered point cloud data. *PLoS ONE*, 13, 2018.

[17] E. Olson. AprilTag: A robust and flexible visual fiducial system. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3400–3407. IEEE, May 2011.

[18] I. Pohl. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, page 12–17, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[19] Repubblica, 2022. URL `https://www.repstatic.it/content/contenthub/img/2022/05/24/114226177-5cef6076-990d-4f67-b7ae-03d1cc734ee3.jpg?webp`.

[20] R. B. Rusu and S. Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[21] B. Siciliano and O. Khatib, editors. *Springer Handbook of Robotics*. Springer, Berlin, Heidelberg, 2008. ISBN 978-3-540-23957-4. URL `http://dx.doi.org/10.1007/978-3-540-30301-5`.

[22] B. Siciliano and O. Khatib. *Humanoid Robots: Historical Perspective, Overview and Scope*, pages 1–6. Springer Netherlands, Dordrecht, 2020. ISBN 978-94-007-7194-9. doi: 10.1007/978-94-007-7194-9_64-1. URL `https://doi.org/10.1007/978-94-007-7194-9_64-1`.

[23] R. Simmons. The curvature-velocity method for local obstacle avoidance. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 4, pages 3375–3382 vol.4, 1996. doi: 10.1109/ROBOT.1996.511023.

[24] S. Thrun, M. Montemerlo, D. Koller, B. Wegbreit, J. Nieto, and E. Nebot. Fastslam: An efficient solution to the simultaneous localization and mapping problem with unknown data. *Journal of Machine Learning Research*, 4, 05 2004.

[25] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. Intelligent Robotics and Autonomous Agents series. MIT Press, 2005. ISBN 9780262201629. URL `https://books.google.ch/books?id=k_yOQgAACAAJ`.

[26] R. Wiki. Concepts, . URL `http://wiki.ros.org/ROS/Concepts`.

[27] R. Wiki. Manifest, . URL `http://wiki.ros.org/Manifest`.

[28] R. Wiki. Metapackages, . URL `http://wiki.ros.org/Metapackages`.

[29] R. Wiki. msg, . URL `http://wiki.ros.org/msg`.

[30] R. Wiki. Nodes, . URL `http://wiki.ros.org/Nodes`.

[31] R. Wiki. Packages, . URL `http://wiki.ros.org/Packages`.

[32] R. Wiki. Parameter server, . URL `http://wiki.ros.org/Parameter20Server`.

[33] R. Wiki. Services, . URL `http://wiki.ros.org/Services`.

[34] R. Wiki. srv, . URL `http://wiki.ros.org/srv`.

[35] R. Wiki. Topics, . URL `http://wiki.ros.org/Topics`.

A | Appendix A

Listing A.1: Postprocesser code

```
// First, include ros library
#include <ros/ros.h>

// Include then pcl library required
    #include <pcl_conversions/pcl_conversions.h>
    #include <pcl/point_cloud.h>
    #include <pcl/point_types.h>
    #include <pcl/filters/voxel_grid.h>
    #include <pcl/filters/statistical_outlier_removal.h>
    #include <pcl/filters/statistical_outlier_removal.h>
    #include <pcl/io/pcd_io.h>

// Include PointCloud2 message
#include <sensor_msgs/PointCloud2.h>

// Topics (corrected, needs to be checked)
static const std::string IMAGE_TOPIC = "/d400_lb/depth/color/points";
static const std::string PUBLISH_TOPIC = "/pointcloudfiltered";

using namespace std::chrono;

// ROS Publisher
ros::Publisher pub;
int i;

void cloud_cb(const sensor_msgs::PointCloud2ConstPtr& cloud_msg)
{
```

```
    auto start = high_resolution_clock::now();
    // Container for original & filtered data
    pcl::PCLPointCloud2* cloud = new pcl::PCLPointCloud2;
    pcl::PCLPointCloud2ConstPtr cloudPtr(cloud);
    pcl::PCLPointCloud2 cloud_filtered;

    // Convert to PCL data type
    pcl_conversions::toPCL(*cloud_msg, *cloud);
    //filtering with StatisticalOutlierRemoval
    pcl::StatisticalOutlierRemoval<pcl::PCLPointCloud2> sor;
    sor.setInputCloud(cloudPtr);
    sor.setMeanK(9);
    sor.setStddevMulThresh(1.0);
    sor.filter(cloud_filtered);

    // Convert to ROS data type
    sensor_msgs::PointCloud2 output;
    pcl_conversions::moveFromPCL(cloud_filtered, output);

    // Publish the data
    pub.publish (output);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);
    std::cout << duration.count() << std::endl;
}

int main (int argc, char** argv)
{
    // Initialize the ROS Node ""
    ros::init (argc, argv, "subscriber_filter_republisher");
    ros::NodeHandle nh;
    //std::cout << "Type 0 for Voxel grid filtering, 1 for Statistical Outli
    //std::cin >> i;
    //std::cout << "You have chosen: " << i;
    // Print "Hello" message with node name to the terminal and ROS log file
    ROS_INFO_STREAM("Hello from ROS Node: " << ros::this_node::getName());
```

```
    // Create a ROS Subscriber to IMAGE_TOPIC with a queue_size of 1 and
    ros::Subscriber sub = nh.subscribe(IMAGE_TOPIC, 1, cloud_cb);

    // Create a ROS publisher to PUBLISH_TOPIC with a queue_size of 1
    pub = nh.advertise<sensor_msgs::PointCloud2>(PUBLISH_TOPIC, 1);

    // Spin
    ros::spin();

    // Success
    return 0;
}
```

B | Appendix B

```
\caption{Testin module}
#!/usr/bin/env python
from math import sqrt
from pathlib import Path

import rospy
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from openpyxl import Workbook
from openpyxl import load_workbook

from cros.topics import ROSTopics

rospy.init_node("measures")
rospy.loginfo("Started measuring ...")
RATE = 10  # Hz
rate = rospy.Rate(RATE)
# filename = "../measurement.xlsx"  # save in isc_slam
rounding = 2
filename = Path.home() / 'MEASURES/measurement.xlsx'


def cb_status(msg):
    global goal_status
    status_array = msg.status_list
    try:
        goal_status = status_array[len(status_array) - 1].status
    except:  # ??????????????????
        # rospy.loginfo("status not received yet")
```

```
        goal_status = 0


def cb_cmd_vel(msg):
    global cmd_vel, cmd_vel_received
    cmd_vel = msg
    cmd_vel_received = True


def cb_odom(msg):
    global odom, top_speed
    odom = msg
    top_speed = max(top_speed, msg.twist.twist.linear.x)


movement = 0  # [0: not started, 1: navigating, 2: rotating,\\
3: backwards, 4:stuck]

cmd_vel = Twist()
odom = Odometry()
goal_status = 0
nav_time = 0
nav_dist = 0
nav_speed_integral = 0
nav_avg_speed = 0
top_speed = 0

cmd_vel_received = False
fw_time = 0
fw_speed_integral = 0
fw_avg_speed = 0

rot_time = 0
rot_speed_integral = 0
rot_avg_speed = 0

back_time = 0
```

```
stuck_time = 0

stopped = True
rospy.Subscriber(ROSTopics.GOAL_STATUS.name, ROSTopics.GOAL_STATUS.data_c
rospy.Subscriber(ROSTopics.CMD_VEL_MUX_OUT.name, ROSTopics.CMD_VEL_MUX_OU
rospy.Subscriber(ROSTopics.T265_ODOM.name, ROSTopics.T265_ODOM.data_class


while not rospy.is_shutdown():
    last_time = rospy.Time.now()
    last_pos = [odom.pose.pose.position.x, odom.pose.pose.position.y]
    while not goal_status == 0:  # if navigation started at least once
        dt = (rospy.Time.now() - last_time).to_sec()
        ds = sqrt((odom.pose.pose.position.x - last_pos[0]) ** 2 + (odom.
        last_time = rospy.Time.now()
        last_pos = [odom.pose.pose.position.x, odom.pose.pose.position.y]
        # print("cycle - ", last_time.to_sec())

        if goal_status == 1:  # if in navigation
            if stopped:
                stopped = False

            # GENERAL IF MOVE BASE IS NAVIGATING
            nav_time += dt
            nav_dist += ds
            nav_speed_integral += odom.twist.twist.linear.x * dt
            nav_avg_speed = nav_speed_integral / nav_time

            # IF ROBOT IS REQUESTING FORWARD MOVEMENT
            if cmd_vel_received and cmd_vel.linear.x > 0.05:  # if receiv
                if movement != 1:
                    rospy.loginfo("Navigating...")
                    movement = 1
                fw_time += dt
                fw_speed_integral += odom.twist.twist.linear.x * dt
                fw_avg_speed = fw_speed_integral / fw_time
```

```
            # IF ROBOT IS ROTATING ON PLACE
            if cmd_vel_received and abs(cmd_vel.linear.x) < 0.05 and abs(
                    cmd_vel.angular.z) >= 0.1:  # if received cmd_vel request
                if movement != 2:
                    rospy.loginfo("Rotating on place...")
                    movement = 2
                rot_time += dt
                rot_speed_integral += abs(odom.twist.twist.angular.z) * dt
                rot_avg_speed = rot_speed_integral / rot_time

            # IF ROBOT IS MOVING BACKWARDS
            if cmd_vel_received and odom.twist.twist.linear.x < - 0.05:
# measuring time while going backwards
                if movement != 3:
                    rospy.loginfo("Going backwards...")
                    movement = 3
                back_time += dt

            # STUCK ON PLACE
            # if cmd_vel_received and abs(odom.twist.twist.linear.x) < 0.05 a
            if cmd_vel_received and abs(odom.twist.twist.linear.x) < 0.05 and
                    and cmd_vel.angular.z < 0.1:
                if movement != 4:
                    rospy.loginfo("Stuck...")
                    movement = 4
                stuck_time += dt

            cmd_vel_received = False  # resetting to False to check if cmd_v

        elif goal_status == 3 and not stopped:  # stopped to print results ju
            rospy.loginfo("Robot arrived to goal")
            # PRINT SPEEDS
            print("nav_avg_speed: ", round(nav_avg_speed, 2),
                  " | mov_fw_avg_speed:", round(fw_avg_speed, 2),
                  " | rot_avg_speed:", round(rot_avg_speed, 2),
                  " | nav_dist:", round(nav_dist, 2),
                  " | top_speed:", round(top_speed, 2))
```

```
        # PRINT TIMES
        print("nav_time:       ", round(nav_time, 2),
              " | moving_fw_time:   ", round(fw_time, 2),
              " | rotating_time:", round(rot_time, 2),
              " | back_time:", round(back_time, 2),
              " | stuck_time:", round(stuck_time, 2))

        # List = [rospy.get_param("/wheel_driver_speed"), round(nav_a
        #           round(fw_avg_speed, rounding), round(rot_avg_speed,
        #           round(top_speed, rounding), round(nav_time, roundin
        #           round(rot_time, rounding), round(back_time, roundin
        List = [nav_avg_speed, fw_avg_speed, rot_avg_speed, nav_dist,
                top_speed, nav_time, fw_time, rot_time, back_time, st
        List = [round(num, rounding) for num in List]

        try:
            wb = load_workbook(filename)
            ws = wb.worksheets[0]  # select first worksheet
        except FileNotFoundError:
            headers_row = ['Speed req.', 'tot_avg_speed', 'fw_avg_spe
                           'rot_avg_speed', 'nav_dist', 'top_speed',
                           'nav_time', 'moving_fw_time', 'rotating_ti
            wb = Workbook()
            ws = wb.active
            ws.append(headers_row)
        ws.append(List)
        wb.save(filename)

        stopped = True
    rate.sleep()
```

List of Figures

List of Tables

List of Symbols

Variable	Description	SI unit
ω	angular speed	rad/s
V	linear speed	m/s

Acknowledgements

Here you might want to acknowledge someone.