POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

AMR system for autonomous indoor navigation in unknown environments

Supervisors

Prof. Marina INDRI

Ing. Orlando TOVAR ORDOÑEZ

Candidate

Alessandro REA

December 2022

Abstract

With the introduction and advancement of technologies now essential to industrial processes, technological evolution has significantly advanced the field of automation. The evolution of autonomous systems has made it possible to improve human work, facilitating collaboration with it while providing a substitute for handling the most demanding tasks. To achieve the goals of Industry 4.0, CIM4.0 has developed the FIXIT project, which aims to provide interactive support to the human operator in an industrial or logistics setting.

The objective of this thesis is to develop an Autonomous Mobile Robot (AMR) system capable of autonomous indoor navigation through an unexplored and unknown environment. In order to fulfill the predefined task, a sensoristics system suitable for the environment is deployed. Exploiting the information coming from the sensors, an Active SLAM algorithm is implemented to extend the functionalities of the classic SLAM method to plan paths toward unkown spaces while mapping the environment.

In order to carry on an analisys between different SLAM algortihms, a comparison of state of the art of Passive and Active SLAM solutions is performed. The resulting methodology is the adoption of an Active SLAM, in particular, the Google Cartographer method, which is used as the primary SLAM module to create submaps and efficiently conducting frontier detection in the geometrically aligned submaps generated by graph optimization.

The overall system is developed using ROS (Robot Operating System) and has been validated in simulation using tools as Rviz and Gazebo. The functionalities of the developed Active SLAM algorithm have been tested in different simulation scenarios to prove the robustness and the efficiency of the solution. At the end, during the experimental phase, the performances of the real robot are evaluated in the CIM4.0 laboratory. The rover shows a great ability in actively exploring the environment, passing through narrow spaces and avoiding obstacles, while locating and mapping the discovered area.

Acknowledgements

I would like to thank Professor Marina Indri for her support and advice received during my thesis journey. In addition, I would like to thank David and Fiorella for the feedback I always received quickly and accurately. Regarding CIM4.0, I especially wanted to thank Orlando for giving me the opportunity to do the thesis in this company and guiding me during this project.

Thanks should be given to all my big family who, starting from the beginning of my academic journey have always supported me. In difficult times they stood by me and never doubted me. Especially Mom, Dad and Stefano, I hope it has made you all proud of me. I wanted to warmly thank Giulia, who always believed in me, even when I did not. Without you, I would not have reached this milestone; half the credit for this milestone belongs to you.

An especially heartfelt thank you goes to my fellow CIM members. Roberto, Bessim, Christian and Wang, we have shared a lot and helped each other, if this journey has been enjoyable, it is thanks to you. Last but not least, I wanted to thank the guys of the fifth floor and especially David, the relationship we built still binds us years later.

Table of Contents

Listings

List of Tables

List of Figures

Acronyms

AGV	Autonomous Guided Vehicles
AMR	Autonomous Mobile Robot
APF	Artificial Potential Field
ASLAM	Active Simultaneous Localization and Mapping
BFS	Breadth First Searches
DOF	Degree Of Freedom
DWA	Dynamic Window Approach
EKF	Extended Kalman Filter
IMU	Inertia Measurement Unit
PF	Particle Filter

ROS	Robot Operating System
RRT	Rapidly-exploring Random Tree
SLAM	Simultaneous Localization and Mapping
UAV	Unmanned Aerial Vehicle
URDF	Unified Robot Description Format
WFD	Wavefront Frontier Detection
XML	Extensible Markup Language

Chapter 1

Introduction

The use of autonomous systems has considerably risen over the past few years inside industrial environments. They are characterized by systems that make it easier to convey supplies, goods, and tools required by human workers. Moreover, the use of autonomous systems enhance people's job by helping them with particular tasks or even taking over for them, preventing dangerous situations. There are numerous types of autonomous systems, which are primarily classified as Autonomous Guided Vehicles (AGV) or Autonomous Mobile Robots (AMR).

Even if the AGVs have been around for a while, they have restrictions due to the fact that they can only carry out simple and routine tasks and require extensive alterations to the area where they are moved, rendering it not a portable and scalable solution. They are driven by sensors and magnetic strips outside the vehicle or by wires that are put in the environment in which they move.

Instead, an AMR is a device that can move around in an unknown and dynamic environment. It performs these tasks by using sensors installed on its chassis to detect its surroundings and determine where it is. In an Industry 4.0 environment, where production lines may frequently change or be updated, the flexibility of the AMRs is a basic necessity. As a result, it may be required to rapidly reconfigure the devices utilized in these contexts.

Industry 4.0 is a combination of interconnected cutting-edge technologies that have the potential to transform manufacturing in all of its forms. They reflect the automated and connected industrial systems that make up the industrial evolution. In this context is placed the Competence Industry Manufacturing 4.0. Therefore, to embrace the requirements of Industry 4.0 and with the purpose of introducing an interactive support for a human operator in an industrial environment, CIM4.0 propose the FIXIT project.

The FIXIT project, shown in Figure 1.1, is composed by an Autonomous Mobile Robot (AMR) and an Unmanned Aerial Vehicle (UAV). The AMR platform is a mobile robot equipped with omni-directional wheels and on-board sensors, as LiDARs and cameras, in order to fully support autonomous navigation in an industrial environment. Also the UAV subsystem is capable of flying autonomously. The final objective of the FIXIT project is the collaboration and the communication of the two subsystems during their navigation tasks.

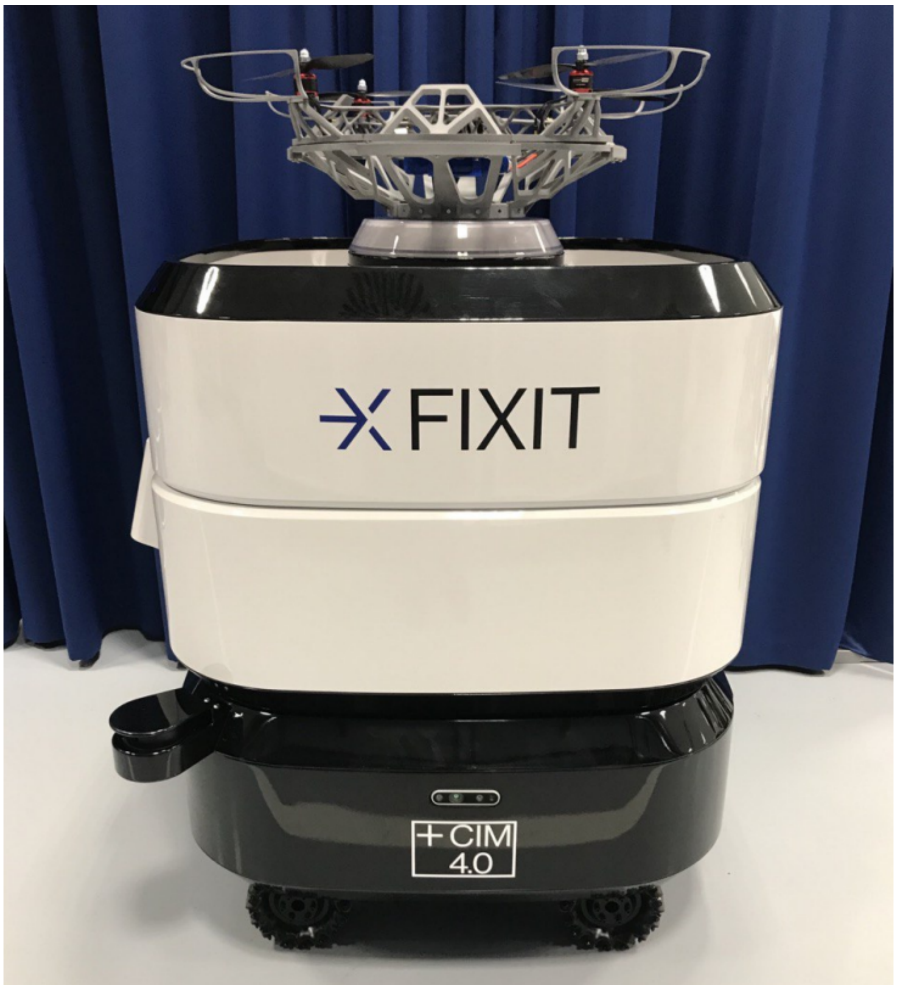


Figure 1.1: FIXIT system

The objective of this thesis is to develop and implement an Simultaneous Localization and Mapping (SLAM) solution for the autonomous exploration of an unknown environment.

The autonomous exploration of an unknown environment is a difficult task and an open field in mobile robotics research. To perform exploration, an extension of the SLAM algorithm is required, called Active SLAM, to autonomously plan paths while mapping and localize inside an environment. After an extensive study of the state of the art, I started to develop an Active SLAM solution composed by three different modules, the interaction of which guarantee to safely explore the environment. The SLAM module is in charge of mapping the environment and in the meanwhile localize the robot inside it, creating a partial map. The points in the map included between known and unknown spaces are select, transformed into navigation points, and given to the path planning module in order to be reached by the mobile robot. The Active SLAM algorithm was developed using Robot Operating System (ROS), a meta operating system widely used in robotics, and was firstly tested in a simulation environment, exploiting different software as Gazebo and RViz, and in the end it was experimentally tested on the AMR.

1.1 Structure of the Thesis

The thesis is structured as follows:

- The second chapter describes the differences between a Classical SLAM solution and an Active SLAM one, focusing on the state of the art of the latter.
- The third chapter introduces the ROS framework, analyzing the modules that compose it and explaining how was it used for developing the Active SLAM algorithm.
- In the fourth chapter, an in-depth analysis is conducted on the development of the Active SLAM solution.
- In the fifth chapter are shown the results of the algorithm in a simulation environment, exploiting tools as RViz and Gazebo.
- In the sixth chapter the hardware architecture of the rover is described in detail, starting from the sensors used and arriving to the boards adopted.

- The seventh chapter describes the experimental results of the developed Active SLAM solution obtained by the AMR.
- In the eighth chapter, the results obtained are analyzed and some suggestions for future developments are given.

Chapter 2

State of the Art

2.1 SLAM algorithms

Simultaneous Localization and Mapping (SLAM) algorithms are widely used in Autonomous Mobile Robot applications because they allow the robot to localize itself while mapping the environment [1]. SLAM solving techniques have been developed to use various sensors including wheel encoders, laser scanners and RGB cameras in order to estimate the robot's pose while building the map (2D or 3D) of the environment. A SLAM architecture is composed by two modules: Localization and Mapping. These two parts are internally dependent on each other because the map is required for localization and, at the same time, the localization is essential for mapping. The Figure 2.1 shows how a simultaneous estimate of both robot and landmark location is required. The true locations are never known of measured directly, for that reason, observation are made between true robot and landmark locations. SLAM algorithms can be considered as the core of autonomous navigation for a mobile robot.

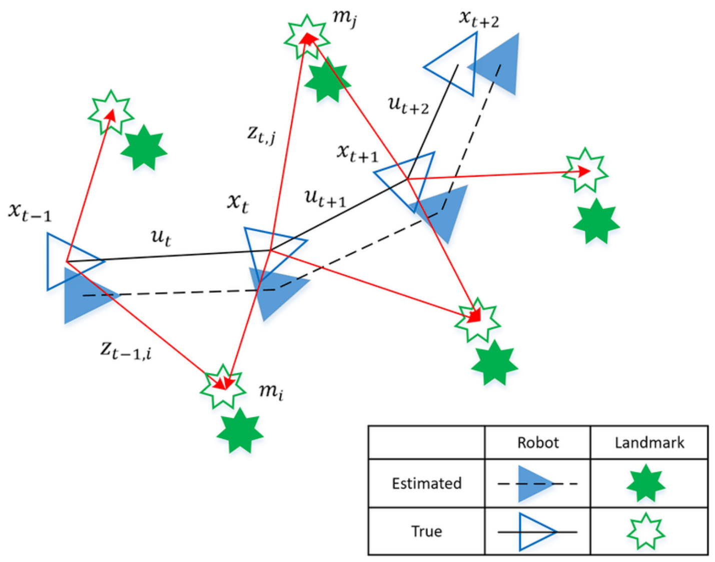


Figure 2.1: An example of how a SLAM algorithm works [2]

2.1.1 Online and Full SLAM

The literature distinguishes two main forms of the SLAM problem, and both of them can be solved using probability methods based on Bayesian estimation. In an Online method (e.g. Kalman Filter, Particle Filter), filters extract the current features of the map and thereby, estimate only the most recent pose of the robot. An Online SLAM approach can be described as in (2.1):

$$bel(x_t, m) = p(x_t, m|z_{1:t}, u_{1:t}) \propto p(z_t|x_t, m) \int_{x_{t-1}} p(x_t|x_{t-1}, u_t) bel(x_{t-1}, m) dx_{t-1} \quad (2.1)$$

where x_t represents the latest pose of the robot, m is the set of all landmarks, $z_{1:t}$ is the group of landmarks observations and $u_{1:t}$ is the control unit. The above equation describes the problem of estimating the robot state x, the map m based on a series of controls $u_{1:t}$ and sensor information $z_{1:t}$. The scheme of an Online SLAM system is depicted in Figure 2.2.
In Full method (e.g. GraphSLAM) the current state and all the previous state of the robot and the map features are estimated and, as a result, the entire path is constructed. The Full SLAM problem is defined in (2.2), as follows:

$$bel(x_{0:T}, m) = p(x_{0:T}, m|z_{0:T}, u_{0:T}) = p(m|x_{0:T}, z_{0:T}) \cdot p(x_{0:T}|z_{0:T}, u_{0:T}) \quad (2.2)$$

where $x_{0:T} = \{x_0, x_1, ..., x_T\}$ is the trajectory obtained by collecting every pose of the robot, $m_{0:T} = \{m_0, m_1, ..., m_T\}$ represents the set of all the landmarks, $z_{0:T} = \{z_0, z_1, ..., z_T\}$ are the information coming from the observations and $u_{0:T} = \{u_0, u_1, ..., u_T\}$ contains the control input commands. The scheme of a Full SLAM algorithm is shown in Figure 2.3.

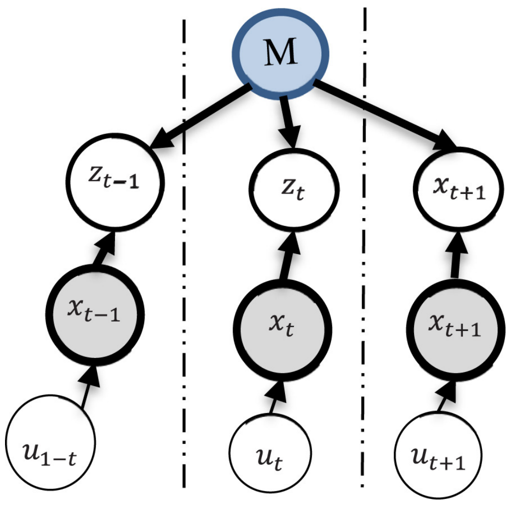


Figure 2.2: A scheme of an online SLAM [1]

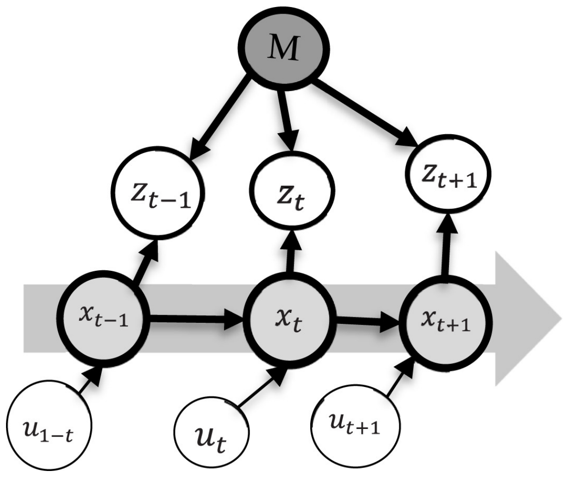


Figure 2.3: A scheme of Full SLAM algorithm [1]

2.1.2 Three main SLAM paradigms

There are three different SLAM paradigms, from which most others are derived.

- **Kalman Filter** based approaches: This family of SLAM algorithms use a single state vector to estimate the locations of the robot and a set of features in the environment, with an associated error covariance matrix representing the uncertainty in these estimates, including the correlations between the vehicle and feature state estimates [3].

- **Particle Filters**: The Particle Filters based approaches are different with respect to Kalman Filter because they are able to efficiently solve the localization problem without handling the system non-linearity and also non-Gaussian models [4].

- **Graph-Based Optimization** techniques: The Graph-Based optimization techniques (GraphSLAM is the most popular one) solve the Full SLAM problem [5].

2.1.3 Difficulties and challenges of SLAM

SLAM is by definition a complicated problem and to achieve a reliable performance different problems need to be coped. The main problems are:

- **Data Association**: The correspondence problem is the difficulty of the SLAM system to associate currently observed landmark with previously observed ones. These errors occur when a robot has a wrong perception of the same landmark as the one perceived in another position [1]. Loop closure methods can be used to overcome this problem.
- **Uncertainty**: There are two types of uncertainty in SLAM which could restrict the performance of the robot, Location and Hardware uncertainty [6]. Hardware uncertainty is caused by hardware errors and noises in the robot's components that could lead to acquire inaccurate information about the robot's pose and landmarks positions.
- **Sensor's noise**: The sum of small errors in the sensors perception gradually grow in long term navigation, which could lead the system to fail [7].
- **Time complexity**: The time complexity is a factor that should be considered, especially when the landmarks grow in number [8]. The complexity of a SLAM algorithm is inversely proportional to system performance. As the time complexity of the SLAM algorithm increases, the time required by the system to perform the actions increases.

2.2 Passive and Active SLAM

In passive SLAM algorithms, another entity is in charge of robot's control, and the SLAM algorithm become merely an observation process [9]. The great majority of algorithms fall into this category, allowing the creator of the robot to install any number of motion controllers and pursue any number of motion goals. In active SLAM, the robot actively explores its environment in the pursuit of an accurate map. Active SLAM methods tend to yield more accurate maps in less time, but they restrict robot movements. There exist hybrid techniques in which the SLAM algorithm controls only the pointing direction of the robot's sensors, but not the motion direction.

2.2.1 Passive SLAM solutions

The majority of available applications adopt this type of SLAM technique. The autonomous navigation performance is divided into two stages. At the begin, an off-line stage is performed, where the rover is totally controlled by an user when performing SLAM in the environment. During this stage, the robot is capable of building a preliminary map of the environment and localize himself in it. During the second stage, given the pre-built map of the environment, the rover can safely navigate autonomously in the room and reach a target point [10]. Currently there are a lot of different solutions and also different approaches to solve this types of problems.

One possible approach describes a rover equipped with a 3D-LiDAR scanner, an IMU sensor and a NVIDIA JETSON TX2 board [11]. The presented system is formed by three modules: mapping, localization and planning. Each of this module is implemented using ROS packages. The mapping module is implemented in ROS using the *gmapping* package, which is used to perform laser-based SLAM algorithms. The localization module can be easily implemented using the *amcl* ROS package. The *amcl* package requires as input six different data in order to estimate the pose: raw laser range data, odometer data, IMU data, frame transform data, pre-built map data, and initial pose data of the mobile robot. The planning module is implemented using the *move_base* ROS package. This package requires five data as input: raw laser range data, frame transform data, pre-built map data, estimated current pose data of the mobile robot, and goal pose data. The cooperation of these packages guarantee the autonomous navigation of the rover.

Another solution presents a mobile robot which uses a two wheel differential drive [12]. The sensors deployed in this application are an RGB-D camera and a 1D-LiDAR sensor with the addition of an IMU module. IMU is used to detect the angular velocity and acceleration of the mobile robot in three-dimensional space, and the data is used to correct the error of the odometer of the mobile robot. The ROS based SLAM algorithm uses a particle filter approach to solve the simultaneous localization and mapping. The experiments conducted were divided in two parts: in the first part a map of the environment was built using SLAM, and in the second part, given a target placed in the constructed map, the autonomous navigation was performed.

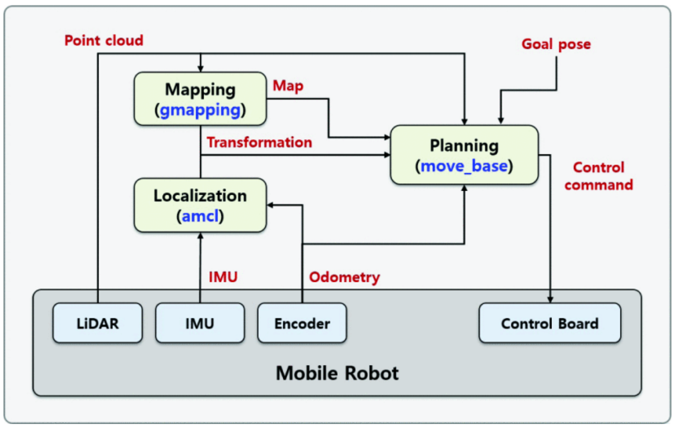


Figure 2.4: Software architecture of a Passive SLAM algorithm [11]

In [13] and [14] the solutions designed are characterized by the construction of the 2D map of the environment applying classical SLAM algorithms, but the novelty lies on the local motion planning adopted. These two papers present a reinforcement learning solution capable to proper detect dynamic obstacles and avoid them, re-planning his trajectory to successfully reach the target. A Q-learning algorithm is therefore implemented. Given the agent's present state, Q-learning is a model-free, off-policy Reinforcement Learning technique that will determine the appropriate course of action. The location of the agent in the environment will determine what will happen next. The model's goal is to determine the optimum course of action given the current state of the system.

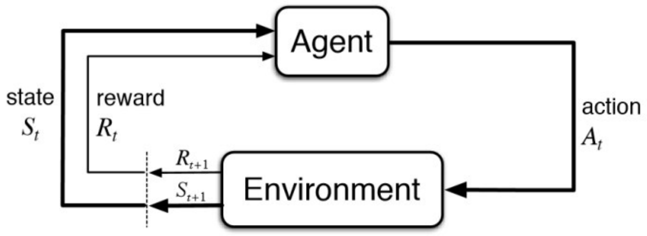


Figure 2.5: Schematic of a Q-Learning algorithm

2.2.2 Active SLAM solutions

The barriers and restrictions that the SLAM method has already overcome are taken into account by the Active SLAM approach. The Active SLAM, which resolves the autonomous search of space, can be seen of as an extension of classic SLAM approaches [9]. Active SLAM performance will be good if the robot motions are appropriate as in the robot should move in a path where the localization and map uncertainties are very small. So Active SLAM is a decision making problem. A decision needs to be made on how the robot should explore the environment before going to the planning task. Differently from passive SLAM solutions, an active SLAM approach doesn't need a pre-built map of the environment. The robot is therefore able to autonomous explore the environment and perform path planning. Active SLAM can be considered an open problem in robotics and research in this area constantly produces new approaches and solutions that try to solve major problems and improve performances of active SLAM.

A novel method for new target selection during the exploration phase is described, the SLAM algorithm is not specified and the path planning algorithm (A*) is not described in details [15]. Starting from a grid map where each cell contains a number that classifies the space (0, free beech; 1, occupied; and -1, unknown), a state matrix is created and some checks are done. Depending on the map several interesting points can be obtained. For a suitable selection of one specific point, it is necessary to create a weight function that evaluates each point separately. When creating the weight function two main factors need to be considered: the distance of the point from the current position and the number of points in the area. To obtain the multiplicity of points in the area, it is necessary to perform clustering, and in this paper K-means++ algorithm is used. After clustering, weights are assigned to each point and are inserted into a matrix. During the last step, the row with the largest weight is selected and its x and y coordinates serve as the destination point of the navigation algorithm.

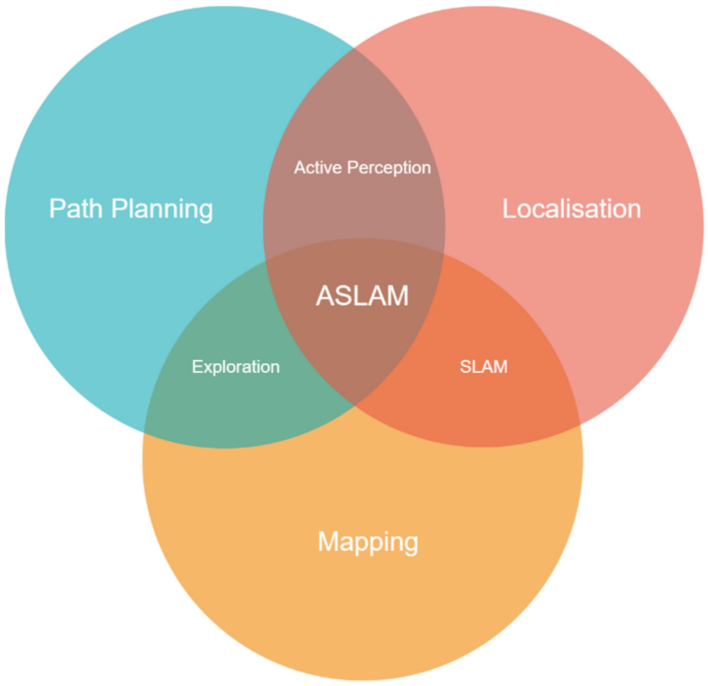


Figure 2.6: Outline of an Active SLAM algorithm [15]

In [16] a differential drive mobile robot designed for smart wheelchair applications is presented. The system is equipped with a 2D-LiDAR sensor, an RGB-D camera and uses wheel odometry to build a 3D map of the uneven environment. In this application, the construction of a 3D map of the environment is necessary for the presence of the staircases and slope, and the smart wheelchair needs to take in to account the height differences for a safe autonomous navigation. The 3-dimension map build is next collapsed in a 2D occupancy map, in order to generate a traversable map based on layer differences. Starting from the traversable map an efficient variable step size Rapidly-exploring Random Tree (RRT) planner is used as global planner. The global planner generates a list of way-points for the overall optimal path from the starting position to the goal position through the map. Then the local planner takes into account the robot kinematic and dynamic constraints, and generates a series of feasible local trajectories that can be executed from the current position, while avoiding obstacles and staying close to the global plan.

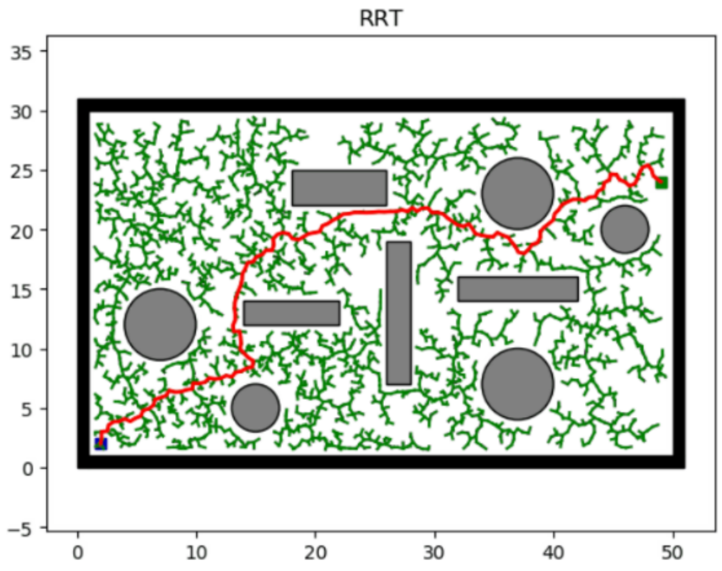


Figure 2.7: Outline of an RRT algorithm [17]

In [18], an exploration and a path planning algorithm to be attached to a classic SLAM paradigm to perform an overall Active SLAM is introduced. The technique used to perform exploration is called Frontier Detection. Given an occupancy-grid map, cells can be classified as: Unknown Region, Known Region, Open-Space, Occupied-Space. A Frontier is defined as the segment that separates known (explored) regions from unknown regions. The algorithm to perform Frontier Detection is called Fast Frontier Detector. In this paper Dijkstra's algorithm is used for global planner. Dijkstra is an algorithm for finding the shortest paths between nodes in a graph. The complexity of the algorithm in the worst case is $\Theta(|E| + |V|log|V|)$, where V and E are respectively the number of the vertexes and the edges of the graph structure.

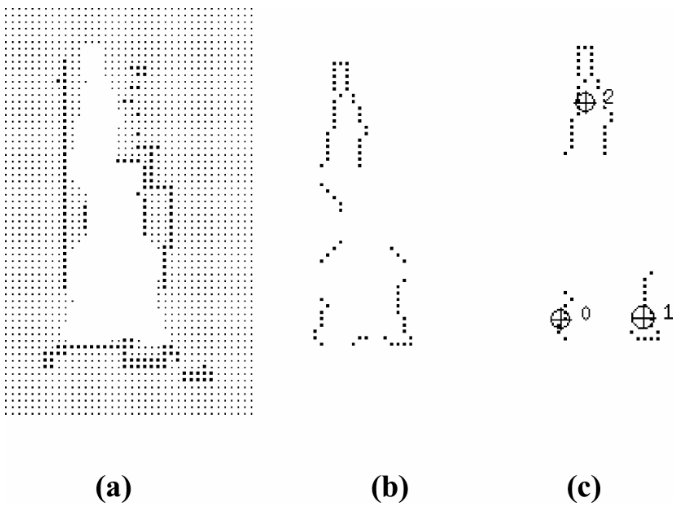


Figure 2.8: Frontier detection: (a) Evidence grid, (b) Frontier edge segments, (c) Frontier regions [18]

In [19] a system composed by a 3D LiDAR and a Kinect camera is introduced to perform Active SLAM. The system is supposed to autonomously explore unknown environments and build 3D maps simultaneously. To allow that, efficient exploration paths need to be planned online, without prior information of the environment. For this purpose an Optimized View Planning algorithm is used, which iteratively generate globally optimized view sequences while updating the map. Regarding the planned exploration sequence, valid motion plans connecting the viewpoints are generated using a sampling-based motion planning library BIT*-H. The algorithm builds on the Batch Informed Trees (BIT*) which employs ordered search and informed sampling of a heuristic (hyper)ellipsoidal subspace limiting the planning space to accelerate the convergence to optimal solutions.

In [20] an ambiguity-aware robust active SLAM (ARAS) framework that makes use of multi-hypothesis state and map estimations to achieve better robustness is introduced. The system is composed of different modules. First of all, a multi-hypothesis SLAM (MH-SLAM) is performed, ambiguous measurements and these probable estimations are taken into account explicitly for decision making and planning. After that, the ARAS framework adopts local contours for efficient multi-hypothesis exploration, incorporates an active loop closing module that revisits mapped areas to acquire information for hypotheses pruning to maintain the overall computational efficiency and demonstrates how to use the output target pose for path planning under the multi-hypothesis estimations.

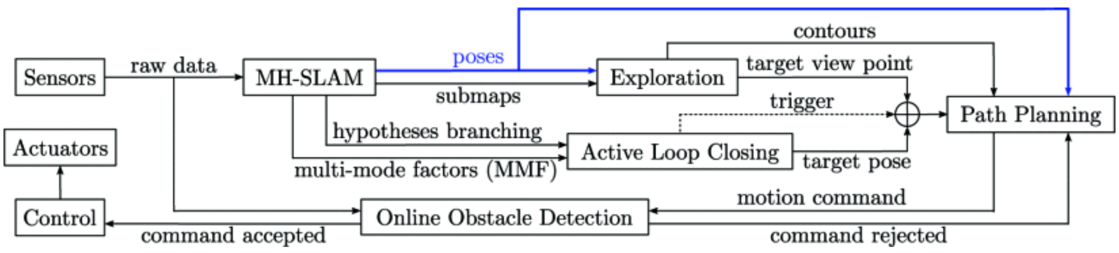


Figure 2.9: Block diagram of the ARAS framework, which consists of four main modules: MH-SLAM, exploration, active loop closing, and path planning. [20]

Chapter 3

ROS Architecture for AMR

3.1 Introduction to ROS

ROS is an open-source, meta-operating system used to interact with robots. Hardware abstraction, low-level device control, common functionality implementation, message-passing between processes, and package management are just a few of the functions it provides. An operating system should contain all of these features. It also provides tools and frameworks for locating, developing, writing, and running code on a variety of platforms, as shown in Figure 3.1. The main component of ROS is how the software operates and communicates, which enables the developer to create complicated projects without having a thorough understanding of specific hardware. A network of processes can be connected to a central hub using ROS. These operations can be carried out across many platforms and connect to the hub in a number of different ways.

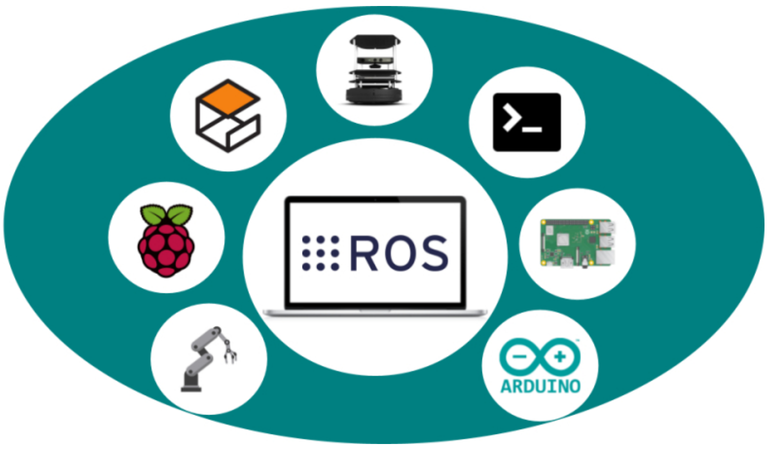


Figure 3.1: ROS interaction with different platforms

3.1.1 The basic principles of ROS

The philosophical goals on which ROS is based, are listed hereafter [21]:

- **Peer to peer**: A system created with ROS is made up of numerous processes that may run on various hosts and are connected in real-time via a peer-to-peer topology. The advantages of the multi-process and multi-host design can also be realized by frameworks based on a single server, however a central data server is troublesome if the computers are interconnected in a heterogeneous network.
- **Tools-based**: Instead of creating a monolithic development environment, a microkernel is created as a way to handle the complexity of ROS by allowing a large number of small tools to be used to build and execute the many ROS components and the runtime setting. These tools carry out a variety of functions, such as navigating the source code tree, finding and configuring parameters, and visualizing measure bandwidth using the peer-to-peer connection topology, graphical message data plotting, automatic documentation generation, and so forth.
- **Multi-language**: ROS is designed to be language-neutral, so it currently supports the utilization of four very different languages: C++, Python, Octave, and LISP, with other language ports in various states of completion.
- **Thin**: ROS utilizes code from a wide range of different open-source projects and executes modular builds inside the source code tree. The ROS build system can automatically update source code from different repositories, apply patches, and other things in order to take advantage of the ongoing community advancements.
- **Free an Open-source**: The complete source code for ROS is available to a wide community. The BSD license, which permits the creation of both commercial and non-commercial projects, is the one that controls how ROS is distributed.

3.1.2 Key features of ROS

ROS is made up of several parts, the majority of which are exploited for application development, information exchange and communication. The main components that define the architecture of ROS are listed below:

- **Node**: Nodes are computation-based processes. A robot control system often consists of numerous nodes. One node might, for instance, manage a laser range-finder, manage the wheel motors, or perform localization. A ROS client library, such as *roscpp* or *rospy*, is used while writing a ROS node.

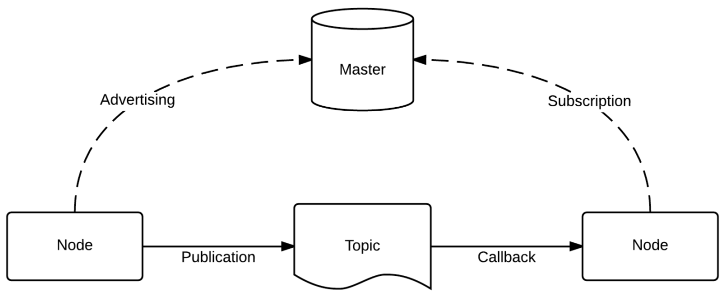
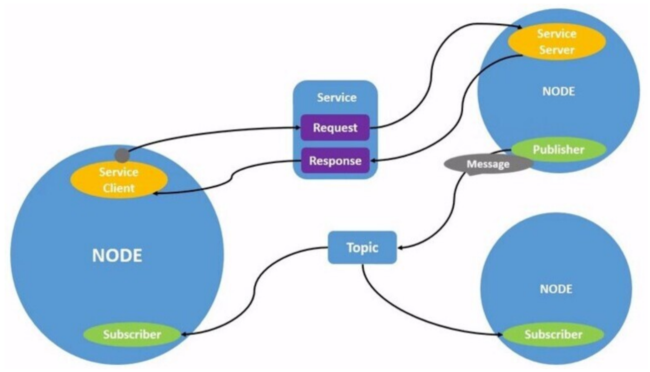
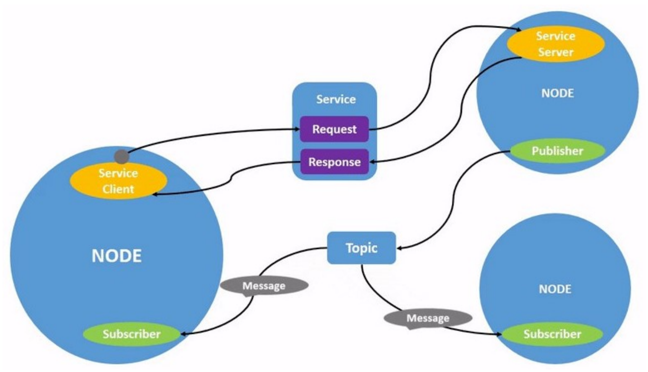


Figure 3.2: Nodes interaction through topics

- **Topic**:Transport systems with publish/subscribe semantics are used to route messages. A node publishes a message to a specific topic in order to send it out. The topic is the term given to identify the message's content. A node will subscribe to the relevant topic if it is interested in a specific type of data.
- **Master**: The other nodes in the ROS system receive naming and registration services from the ROS Master. It keeps tabs on both publishers and subscribers to topics as well as services.
- **Message**: Nodes communicate with each other by exchanging messages. A message is a simple data structure whose fields are composed by primitive types (integer, floating point, boolean, etc.) or arrays of primitive types.
- **Service**: Despite being a fairly flexible communication paradigm, the publish/subscribe approach is inappropriate for request/reply interactions, which are frequently needed in distributed systems. Services are used for request and reply, and each service has two message structures, one for request and one for reply.
- **Parameter Server**: The Parameter Server is a part of the Master and its function is to allow data to be stored in a central location.
- **Package**: In ROS, the primary unit for organizing software is the package. A package may include datasets, configuration files, nodes, a ROS-dependent library, and other items that are logically grouped together.



(a) A node publish a message on a specific topic

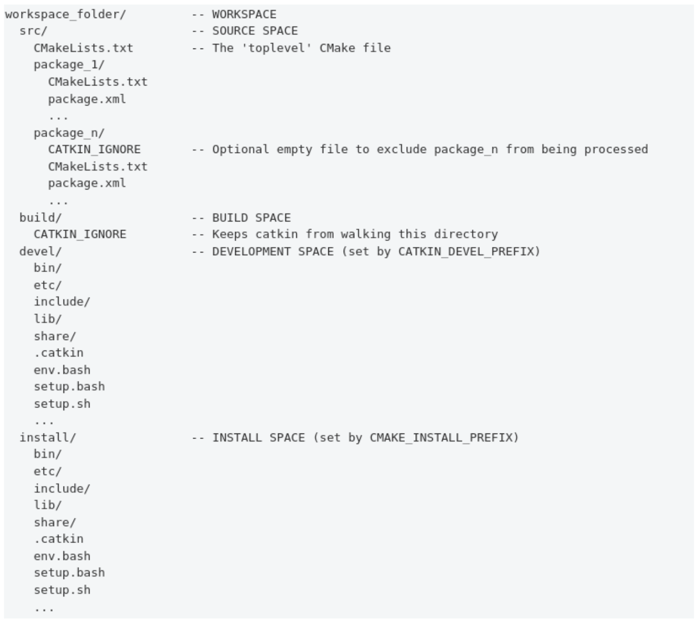


(b) All the nodes subscribed to the topic receive the message

Figure 3.3: Differences between the publish/subscribe approach and the request/reply interaction

3.1.3 ROS Workspace

The workspace, in general, can be thought of as a folder that contains packages. These packages include our source files, and the environment or workspace gives us a mechanism to compile them. It is practical to compile multiple packages at once, and it is a good approach to centralize the development of the code. In ROS, the workspaces are called *catkin* workspaces and they are the unit that contains developed projects. A typical *catkin* workspace is shown in Figure 3.4.



```
workspace_folder/         -- WORKSPACE
  src/                    -- SOURCE SPACE
    CMakeLists.txt        -- The 'toplevel' CMake file
    package_1/
      CMakeLists.txt
      package.xml
      ...
    package_n/
      CATKIN_IGNORE       -- Optional empty file to exclude package_n from being processed
      CMakeLists.txt
      package.xml
      ...
  build/                  -- BUILD SPACE
    CATKIN_IGNORE         -- Keeps catkin from walking this directory
  devel/                  -- DEVELOPMENT SPACE (set by CATKIN_DEVEL_PREFIX)
    bin/
    etc/
    include/
    lib/
    share/
    .catkin
    env.bash
    setup.bash
    setup.sh
    ...
  install/                -- INSTALL SPACE (set by CMAKE_INSTALL_PREFIX)
    bin/
    etc/
    include/
    lib/
    share/
    .catkin
    env.bash
    setup.bash
    setup.sh
    ...
```

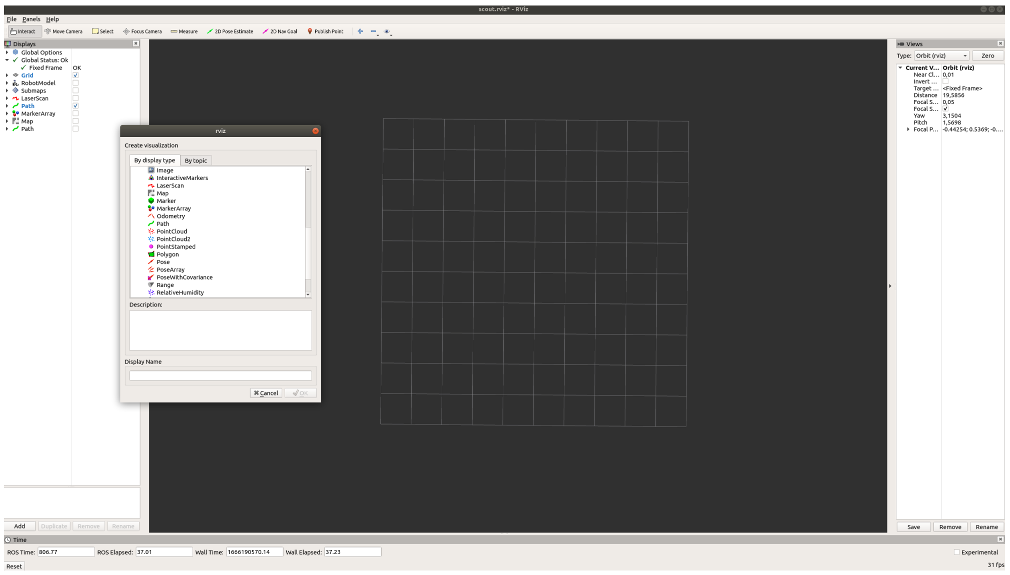
Figure 3.4: An example of a *catkin* workspace

Each folder has an important role in the structure of the workspace:

- **Source space**: Packages and projects are placed under the *src* folder, which is the source space. One of the most important files in this space is *CMakeLists.txt*. The *src* folder has this file because it is invoked by *cmake* when the packages in the workspace are configured.
- **Build space**: *Cmake* and *catkin* store the configuration data, cache information, and other intermediate files for projects and packages in the *build* folder.
- **Development space**: The compiled programs are kept in the *devel* folder.

3.1.4 RViz

One of ROS's key features is the 3D visualization tool known as RViz. RViz is capable of showing data about the robot model, map, coordinate transformation, laser scan, point cloud, path, and more. Even if RViz is a useful display tool, it does not have its own simulation function. For this purpose it can be used both with Gazebo for simulation but also to do tests in real-life, displaying ROS messages and topics giving the possibility to visually control the system. It allows the user to send a command to the robot, set its position, see how it plans its route to reach a final goal, and visualize how a map is built in real-time while the robot moves.



Type: Orbit (rviz)
Current V... Orbit (rviz)
Near Cl... 0,01
Invert ...
Target ... <Fixed Frame>
Distance 19,5856
Focal S... 0,05
Focal S... ✓
Yaw 3,1504
Pitch 1,5698
Focal P... -0.44254; 0.5369; -0....

ROS Time: 806.77 ROS Elapsed: 37.01 Wall Time: 1666190570.14 Wall Elapsed: 37.23

Figure 3.5: RViz basic interface

3.1.5 Gazebo

In both indoor and outdoor three-dimensional environments, Gazebo enables the modeling of robotic and sensor applications. It has a topic-based Publish/Subscribe inter-process communication mechanism with a Client/Server architecture. Each simulation object in Gazebo can have one or more controllers attached to it, which handle commands for managing the object and produce its state. Using Gazebo interfaces, the data generated by the controllers is published into shared memory (Ifaces). Independent of the programming language or the computer hardware platform, the Ifaces of other processes can read the data from the shared memory, enabling inter-process communication between the robot controlling software and Gazebo. Gazebo has access to high-performance rigid body physics simulation engines including Open Dynamics Engine (ODE), Bullet, Simbody, and Dynamic Animation and Robotics Toolkit (DART) during the dynamic simulation process. The 3D graphics rendering of the Gazebo environments is done by the Object-Oriented Graphics Rendering Engine (OGRE) [22].

3.2 URDF model

Robots, indoor scenes, and other objects can all be described using URDF files, which are in the XML format. According to the URDF, a robot is represented as a tree of links connected by joints. The links represent the actual parts of the robot while the joints define where the links are in space by expressing how one

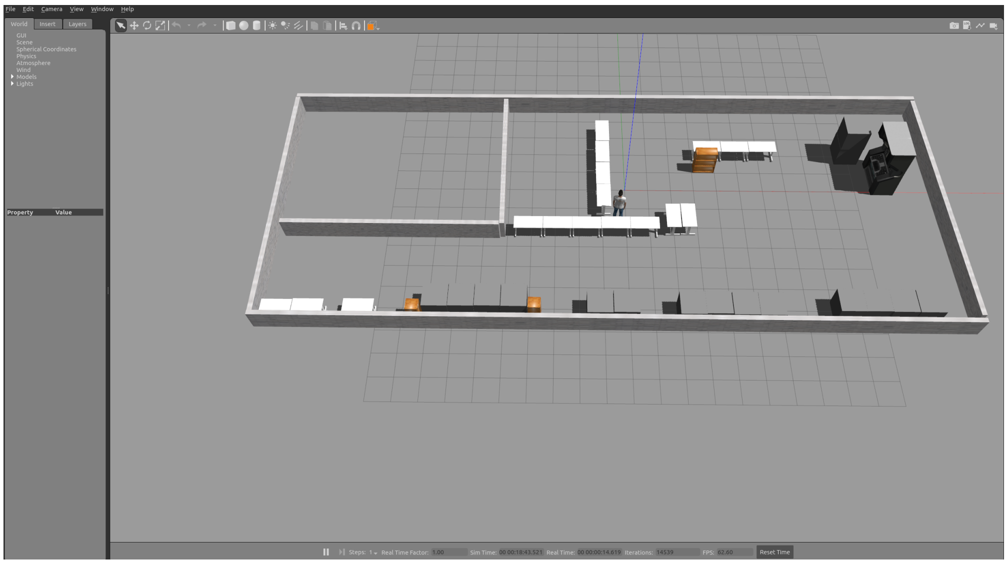


Figure 3.6: Plant of CIM4.0 laboratory in Gazebo

link moves in relation to another link, as shown in Figure 3.7.

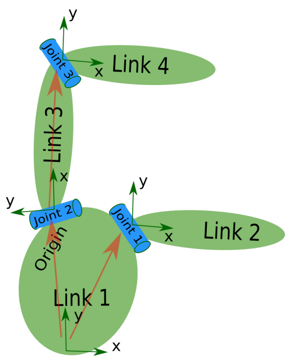
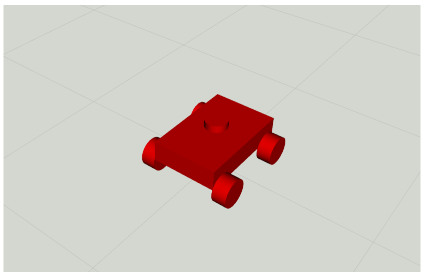


Figure 3.7: Schematic description of link-joint interaction

Due to URDF's XML foundation, everything is represented as a collection of tags that can be nested. Although there are many other tags that can be employed, it's only important to be aware of link and joint tags.

- **Link tags**: In addition to the name of the link, a link tag also allows to describe the link's visual, collision, and inertial attributes:
  - **Visual** tag is able to display in RViz and Gazebo the robot model.
  - **Collision** attribute is used for physical collision calculations.

– **Inertial** tag is also used for physical calculations but it determines how the link responds to forces.


(a) Collision element of th AMR displayed in RViz

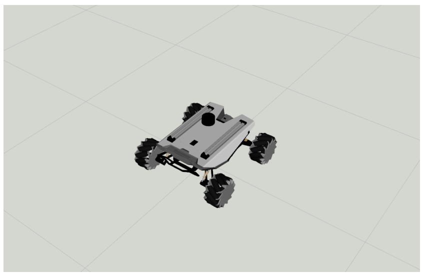

(b) Visual element of th AMR displayed in RViz

Figure 3.8: URDF model of the AMR

- **Joint tags**: Although it is commonly thought that a robot is composed of links, the joints actually determine link placements and how they move in relation to one another, therefore the joints are where all the information resides in terms of the robot's structure. Each joint needs to have these tags specified:
  - **Name** of the joint.
  - **Type** of the joint (fixed, continuous).
  - **Parent** and **Child** defines the relation between links.
  - **Origin** describes the relationship between links before any movement is applied.

Listing 3.1: Example of an XML file describing an URDF model

```
<robot name="test_robot">
  <link name="link1" />
  <link name="link2" />
  <link name="link3" />
  <link name="link4" />

  <joint name="joint1" type="continuous">
    <parent link="link1"/>
    <child link="link2"/>
  </joint>

  <joint name="joint2" type="continuous">
    <parent link="link1"/>
    <child link="link3"/>
  </joint>

  <joint name="joint3" type="continuous">
    <parent link="link3"/>
    <child link="link4"/>
  </joint>
</robot>
```

3.2.1 URDF in Gazebo

For a URDF file to work properly in Gazebo, some simulation-specific tags must be added. Even though URDFs are a helpful and standardized format in ROS, they are inadequate in many areas and have not been upgraded to meet the changing requirements of robotics. Using URDF, only the kinematic and dynamic characteristics of a single robot can be specified. The pose of the robot within the world cannot be specified using URDF. Moreover, since it does not express joint loops and lacks friction and other features, it is not a universal description format. Additionally, it is unable to describe things like lights, heightmaps, and other non-robot objects. To address this problem, a new format known as the Simulation Description Format (SDF) was developed for usage in Gazebo in order to address URDF's drawbacks. From the world level all the way down to the robot level, SDF is a comprehensive account of everything. Because the SDF format is itself documented in XML, it is straightforward to upgrade to newer versions and, additionally, it describes itself.

In order to get an URDF model properly working in Gazebo different tags needs to be added:

- An **<inertia>** tag is required within each **<link>** element.
- Add a **<gazebo>** element for every **<link>**.
- Add a **<gazebo>** element for every **<joint>**.
- Add a **<gazebo>** tag for the **<robot>** element.

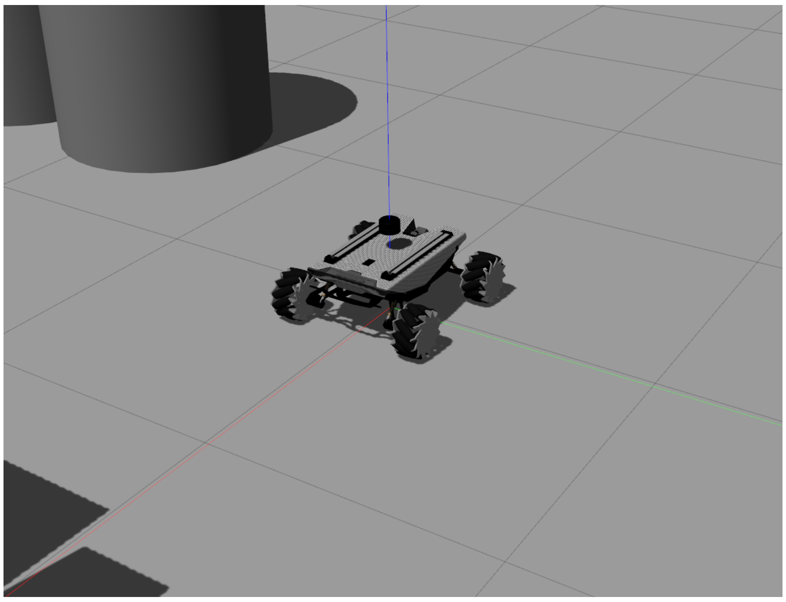


Figure 3.9: URDF model of the AMR displayed in Gazebo

3.2.2 Gazebo plugins

Gazebo plugins can integrate ROS messages and service calls for sensor output and motor input, giving to URDF models further capability. Multiple plugin types are supported by Gazebo, and they can all be connected to ROS:

- **ModelPlugins** provide access to the **physics::Model** API.
- **SensorPlugins** provide access to the **sensors::Sensor** API.
- **VisualPlugins** provide access to the **rendering::Visual** API.

Listing 3.2: Example of a ModelPlugin

```
<robot>
  ... robot description ...
  <gazebo>
    <plugin name="differential_drive_controller" filename="libdiffdrive_plugin.so">
      ... plugin parameters ...
    </plugin>
  </gazebo>
  ... robot description ...
</robot>
```

Listing 3.3: Example of a SensorPlugin

```
<robot>
  ... robot description ...
  <link name="sensor_link">
    ... link description ...
  </link>

  <gazebo reference="sensor_link">
    <sensor type="camera" name="camera1">
      ... sensor parameters ...
      <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
        ... plugin parameters ..
      </plugin>
    </sensor>
  </gazebo>

</robot>
```

3.3 Transform System

In the mathematics of robotics, coordinate transformations (or transforms) play a significant role. They are a mathematical tool for taking measurements or points that are represented from one point of view and represent them from another, more practical, point of view.

Without transformations, it's necessary to use trigonometry to complete the computations, which soon gets quite difficult with bigger problems, particularly in 3D. Assigning coordinate systems, or frames, to the proper system components is the initial step in solving coordinate transformations problems. The definition of transforms between the frames is the next step. A transform can be readily reversed to go the opposite way and provides the translations and rotations needed to change one frame into another.

It will be able possible to convert a known point in one frame to any other frame in the tree if there is a system where each frame is defined by its relationship to one (and only one) other frame.

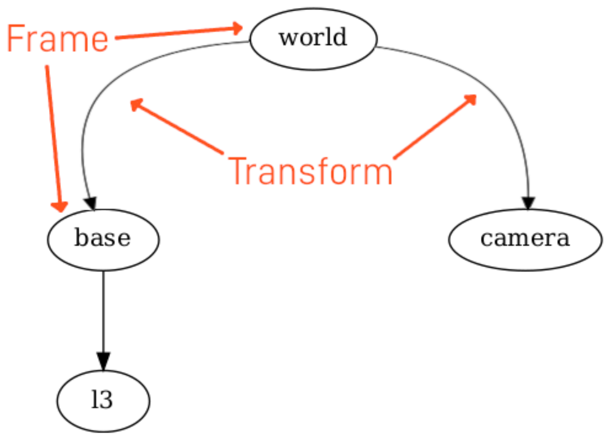


Figure 3.10: Tree structure indicating frames relationship

3.3.1 Transforms in ROS

To manage the transformations, ROS has a mechanism known as *tf2* (TransForm version 2). The *tf2* libraries can be used by any node to broadcast a transform from one frame to the next. These transforms must be organized into a tree structure where each frame is defined by one (and only one) transform from a previous frame, but may also be reliant on any number of subsequent frames. As long as they are connected in the tree, nodes can utilize the *tf2* libraries to listen for transforms and then use those transforms to convert points from any frame to any other frame. The Figure 3.11 shows frame relationship of the AMR.

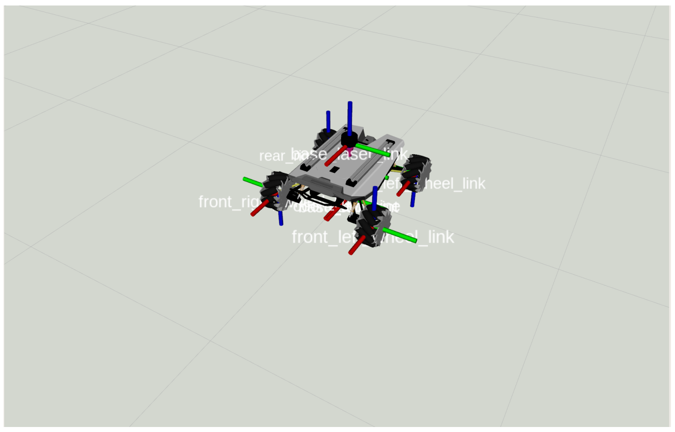


Figure 3.11: Frames composing the model of the AMR

Chapter 4

Development of an Active SLAM Algorithm

4.1 Active SLAM Algorithm

The process of actively planning robot paths while creating a map and locating within it is known as Active SLAM, also referred to as ASLAM [23]. ASLAM takes the SLAM problem one step further by attempting to make the robot move on its own during the entire mapping process. In many situations, ASLAM could make it easier to set up a navigation system because the robot could create the map without any human interaction.

Pose identification, target selection and navigation are the three iterative steps of which an ASLAM algorithm is composed, according to the literature. Pose identification identifies potential destinations, whereas optimal goal selection chooses the best one. Once selected the destination, a path planning algorithm is chosen to calculate the best route to the target position. Considering the pose identification step, a concept that is extensively used in the context of exploration is the frontier.

Frontiers are areas on the border between open space and unknown territory, points on a map between free known space and unknown territory [23]. These points are important because it is very likely that the robot can reach them since they are in the mapped space. They also provide coverage of nearby undiscovered areas. Therefore, frontiers are the best possible groups of places to reach to increase the known environment, as shown in Figure 4.1. After the frontiers points have been selected, generally a clustering operation is performed, to group together all nearest frontiers into one point, which can be chosen during the target selection phase as a target point for exploration.

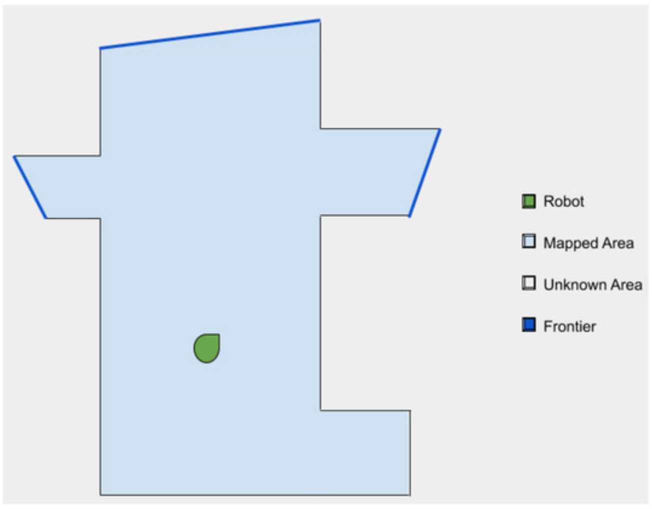


Figure 4.1: Example of a map with frontiers [23]

All the steps previously described can be practically represented by the succession of different modules, as depicted in Figure 4.2. Data coming from sensors (laser scans, point clouds and odometry information) is collected and is given to the SLAM module (Google Cartographer) which elaborate these information to map the environment and localize inside it. Starting from the partial map constructed by Google Cartographer, different Frontier points are identified between unknown spaces and known cells. Once selected the Frontier, it becomes the navigation target, and a path planning algorithm is exploited to reach the final position. All these three modules are iterated until the map is fully discovered, and the mission of autonomous exploration is completed.

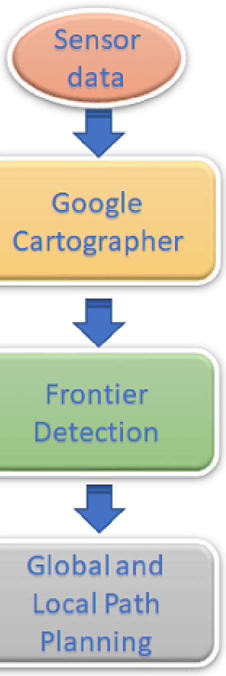


Figure 4.2: Block diagram of the developed Active SLAM solution

4.2 Google Cartographer

A real-time indoor mapping option is provided by Google's Cartographer, which creates 2D grid maps with a resolution of 5 cm. At the best predicted position, which is thought to be accurate enough for brief periods of time, laser scans are inserted into a submap. Scan matching only uses recent scans since it compares them to a recent submap, and as a result, pose estimation error in the world frame accumulates. The Google Cartographer SLAM technique does not use a particle filter to obtain good performance. A pose optimization step is regularly performed to address the error accumulation. Submaps take part in scan matching for loop closure after they have reached their completion, which means no further scans will be added to them. The loop closure is automatically applied to all completed submaps and scans. A scan matcher attempts to locate the scan in the submap if they are close enough based on current pose estimates. A loop closing constraint is introduced to the optimization problem if a sufficiently good match is discovered in a search window surrounding the current estimated pose [24]. The SLAM solution provided by Google Cartographer is composed by two distinct SLAM modules: Local and Global SLAM.

4.2.1 Local SLAM

Both Local and Global approaches seek to optimize the pose $\xi = (\xi_x, \xi_y, \xi_\theta)$ of lidar observations, also called scans. Using a non-linear optimization to align the scan with the submap, each successive scan in the local approach is matched against a small portion of the world, known as a submap M; this procedure is also known as scan matching.

The process of continuously aligning scan and submap coordinate frames, also known as frames, is defined as submap construction. The transformation T_ξ, described in (4.1), which firmly translates scan points from the scan frame into the submap frame, is what is used to represent the pose ξ of the scan frame in the submap frame [24].

$$T_\xi p = \underbrace{\begin{pmatrix} \cos\xi_\theta & -\sin\xi_\theta \\ \sin\xi_\theta & \cos\xi_\theta \end{pmatrix}}_{R_\xi} p + \underbrace{\begin{pmatrix} \xi_x \\ \xi_y \end{pmatrix}}_{t_\xi}. \tag{4.1}$$

The submaps are constructed by collecting only a few consecutive scans, they take the shape of probability grids $M : r\mathbb{Z} \times r\mathbb{Z} \to [p_{min}, p_{max}]$ which map from discrete grid points to values, which represent the probability that a cell is occupied. The closest points on the grid are used to define the equivalent *pixel* for each grid point. Every time that a scan needs to be added to the probability grid, a separate set of

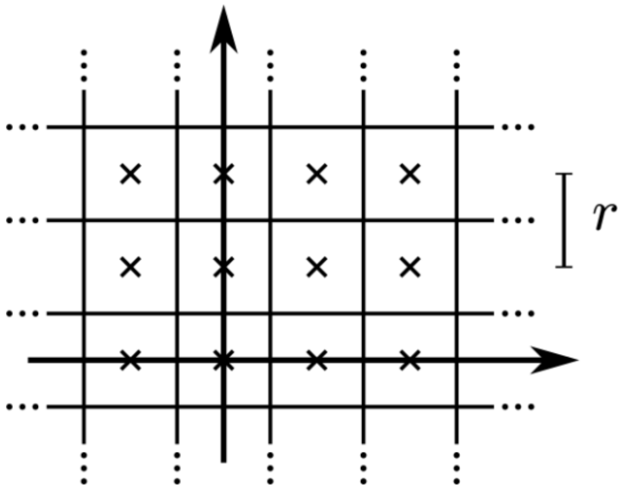


Figure 4.3: Grid points and associated pixels [24]

grid points for *misses* and a set of grid points for *hits* are computed, as shown in Figure 4.4. The nearest grid point to the hit set is added for each hit. The grid point connected to each pixel that crosses a ray between the scan origin and each scan point is added for every miss, omitting grid points that are already included in the hit set. If an unobserved grid point appears in one of these sets, it is given the probability p_{miss} or p_{hit}. Before the insertion of a scan into a submap, an

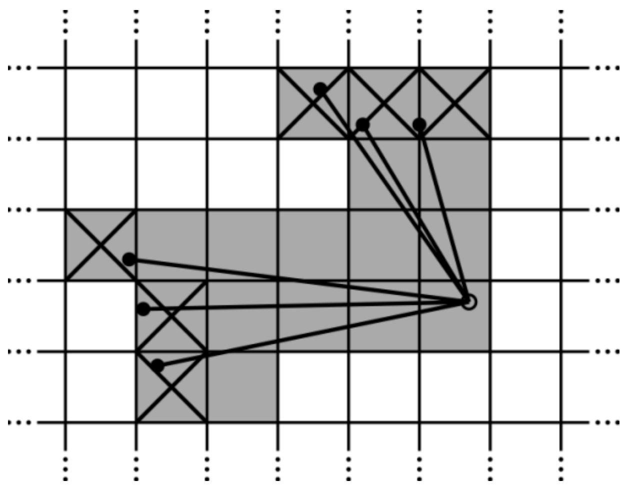


Figure 4.4: A scan and pixels associated with hits (shaded and crossed out) and misses (shaded only) [24]

optimization process is performed on the scan pose with respect to the current local submap, using the Ceres Scan Matcher. This optimization task, shown in (4.2), is a nonlinear least squares problem

$$\arg\min_{\xi} \sum_{k=1}^{K} (1 - M_{\text{smooth}}(T_\xi h_k))^2 \tag{4.2}$$

where h_k represent the information acquired during the k scan and T_ξ is the matrix used to transform h_k from the scan frame to the submap frame in accordance to the scan pose. Instead the function $M_{smooth} : \mathbb{R}^2 \to \mathbb{R}$ is a regular version of the probability values in the local submap.

4.2.2 Loop Closure Optimization

The Local SLAM module progressively increases error since scans are only compared to a submap that contains a few recent scans. In this case, the total inaccuracy is minimal because there are only a few dozen successive scans. However, when the dimension of the map grows the accumulated error increase too. For this purpose, Google Cartographer provides a Global SLAM module which is in charge of performing an additional optimization step to cope with the errors. To handle these problems, the poses of all scans and submaps are optimized, exploiting Sparse Pose Adjustment. In loop closing optimization, the relative postures where scans are placed are maintained in memory. All other pairs composed of a scan and a submap are also taken into consideration for loop closing after the submap is no longer evolving, in addition to these relative poses. In background a scan matcher is run, and if a good match is found, the relative pose is added to the optimization problem.

4.3 Frontier Detection

The kind of SLAM algorithm chosen affects how effective frontier detection is. Generally speaking, mapping in an active exploration framework can be accomplished using either a filtering-based or a graph-based SLAM method. Using a filtering-based approach, for example *gmapping*, the pose of the latest frame can be optimized without changing the previous ones. Doing that, the frontiers to be detected only belong to the latest frame. Instead, when using a graph-based SLAM solution, like *Google Cartographer*, each step of optimization changes all the frames of the constructed graph. The result is that the frontiers need to be re-detected not only in the current frame, but also in the frames of each pose (called node) of the graph. Even if frontier detection is generally faster in cooperation with filtering-based SLAM algorithms, it is more accurate with a graph-based SLAM solution. This means that, exploiting a SLAM algorithm as Google Cartographer in active exploration helps to build more accurate maps [25].

4.3.1 Reachability of Frontiers

During the discovery of frontier points, not all the information acquired can be used during the navigation, due to the fact that some of the frontier points are not accessible for the robot. For this purpose an extensive analysis on the reachability of the frontiers is performed with the aim of discarding points not physically accessible by the robot. To this end, an inflation operation on the submaps is performed, as shown if Figure 4.5, to make sure that the robot platform can reach the detected frontiers.

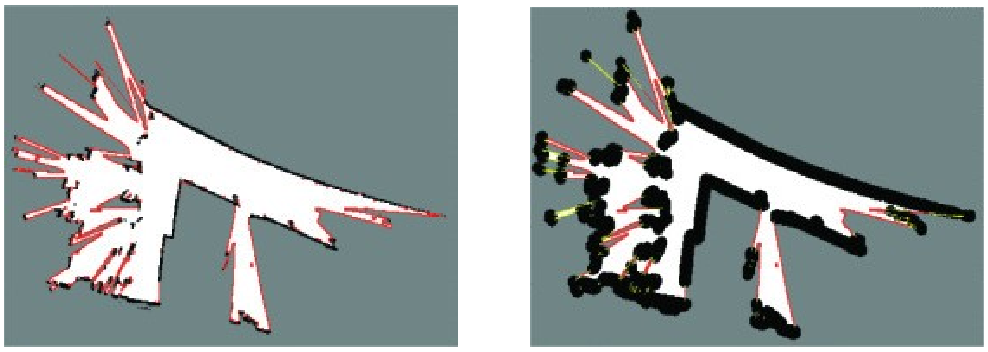

(a) Submap before inflation **(b)** Submap after inflation

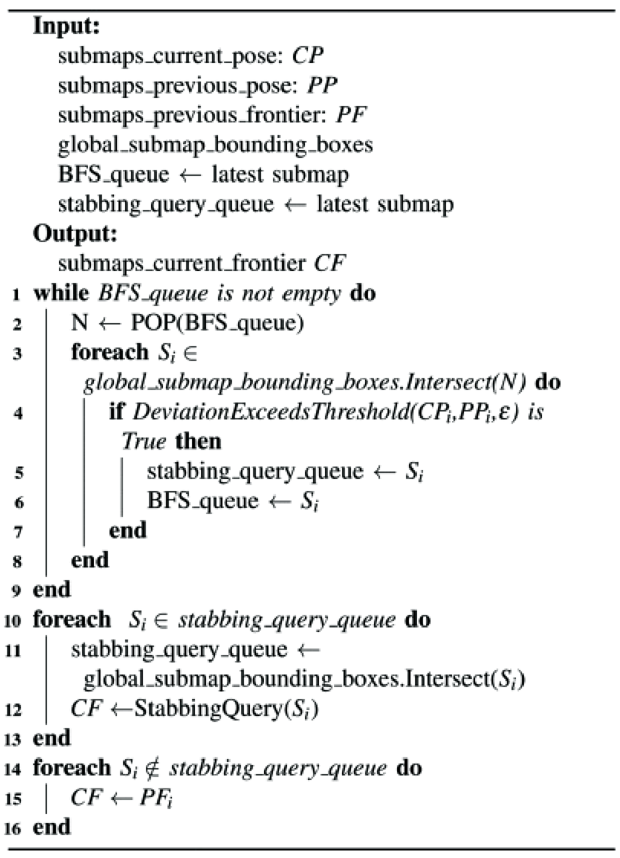
Figure 4.5: Reachability of the frontiers [25]

4.3.2 Breadth First Search

To search for frontiers in submaps, the Wavefront Frontier Detection (WFD) algorithm is used. The WFD perform two Breadth First Searches (BFS) on the submaps, one starting from the robot's location to the firs unknown space, and the second one from this unknown space to the first continuous frontiers encountered. Beginning with the most recent submap, N, the BFS searches all submaps that overlap with N. These chosen submaps are added to the BFS and stabbing-query queues if their pose changes go beyond a specified threshold, as shown in Figure 4.6. The submaps, S_i, whose pose change is higher than the threshold and those that intersect S_i, are then exposed to stabbing query. Instead, the previously discovered frontiers are utilized as replacement of submaps for which the value does not exceed the threshold.

4.3.3 Clustering frontiers into navigation points

The detected frontiers are often continuous, dense, and redundant for navigational or path-planning needs. Therefore, a clustering technique is used to sparsify the observed dense frontiers. The selected clustering algorithm is the Mean Shift. In essence, the mean-shift algorithm assigns data points to clusters iteratively by moving points toward the location with the highest density of data points, or cluster centroid [26]. An example result of this operation is shown in Figure 4.7. Ended the clusterization step, the difficulty of sorting the frontiers according to their priority can be reduced. The sorting criteria selected take into account the distance of the robot from the target point and the percentage of unknown space around the point. From now on, the mobile robot has the ability to select a few exemplary clustered sites and designate them as exploration goals.



```
Input:
    submaps_current_pose: $CP$
    submaps_previous_pose: $PP$
    submaps_previous_frontier: $PF$
    global_submap_bounding_boxes
    BFS_queue $\leftarrow$ latest submap
    stabbing_query_queue $\leftarrow$ latest submap
Output:
    submaps_current_frontier $CF$
while BFS_queue is not empty do
    N $\leftarrow$ POP(BFS_queue)
    foreach $S_i \in$
      global_submap_bounding_boxes.Intersect(N) do
        if DeviationExceedsThreshold($CP_i$,$PP_i$,$\varepsilon$) is
          True then
            stabbing_query_queue $\leftarrow S_i$
            BFS_queue $\leftarrow S_i$
        end
    end
end
foreach $S_i \in$ stabbing_query_queue do
    stabbing_query_queue $\leftarrow$
      global_submap_bounding_boxes.Intersect($S_i$)
    $CF \leftarrow$StabbingQuery($S_i$)
end
foreach $S_i \notin$ stabbing_query_queue do
    $CF \leftarrow PF_i$
end
```

Figure 4.6: Breadth First Search algorithm [25]

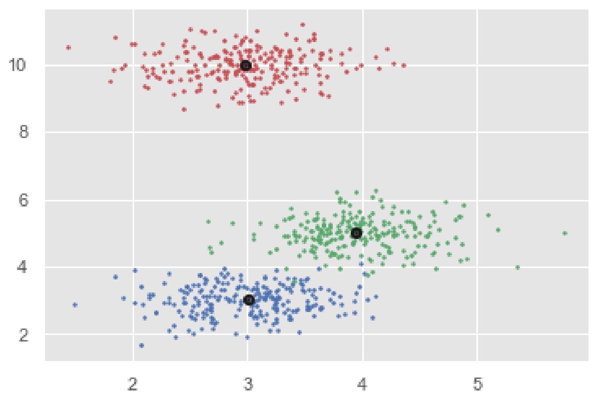


Figure 4.7: An example of a cluster operation [26]

4.4 Global Path Planning

Global path planning algorithms are used in mobile robotics to find a consistent path from the starting point to the target one, taking into consideration the possible presence of obstacles in the navigation environment.

There are different global path planning algorithms that can be exploited for indoor and outdoor navigation. Two of the most used are undeniably **A*** and **D*** algorithms. Figure 4.8 shows an example of global path planning algorithm.

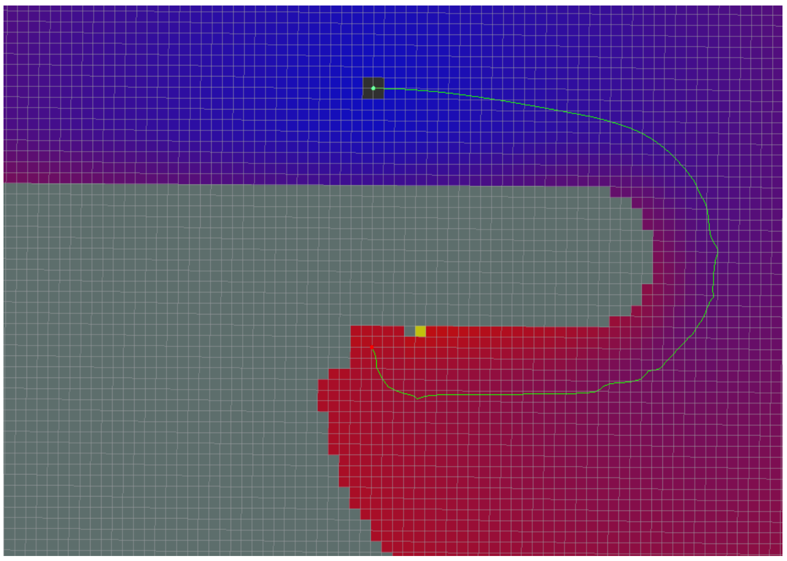


Figure 4.8: An example of a Global Path planning algorithm [27]

4.4.1 A* algorithm

A* is a widely used algorithm for path finding and graph traversal. It was introduced by Peter Hart, Nils Nilsson and Bertram Raphael in 1968 as an extension of Dijkstra algorithm [28].

A* maintains a prioritized list of possible path segments as it moves across the graph, choosing the path with the lowest known cost. Any time a path segment has a greater cost than a different path segment that has been encountered, the higher-cost path segment is abandoned and the lower-cost path segment is traversed. Until the objective is achieved, this process is continued. A* discovers the cheapest route from a given initial node to a single goal node using a best-first search.

It chooses which nodes in the tree to visit first, using a distance-plus-cost heuristic function, typically written as f(x). The distance-plus-cost heuristic function is denoted as:

$$f(x) = h(x) + g(x) \tag{4.3}$$

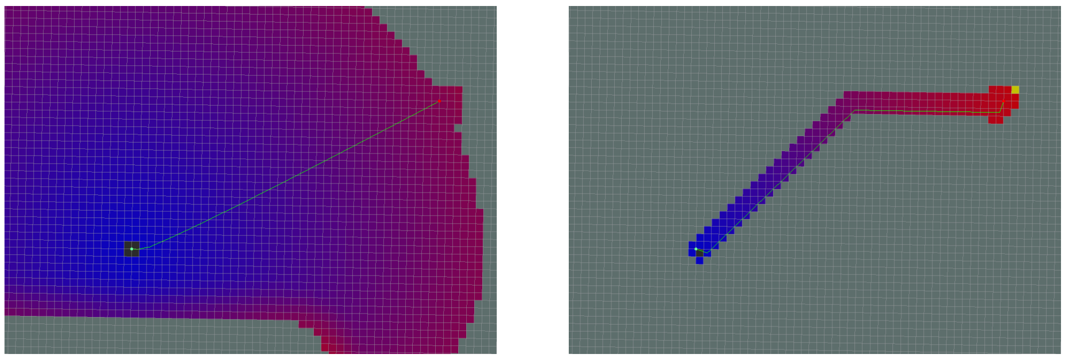
where $h(x)$ is a heuristic estimate of the distance to the target position and $g(x)$ represents the cost from the starting node to the current one.

The $h(x)$ component of the $f(x)$ function needs to be an acceptable heuristic, meaning it can't overestimate how far away the objective is. The heuristic $h(x)$ is referred to as monotone or consistent if it meets the extra requirement

$$h(x) <= d(x, y) + h(y) \tag{4.4}$$

for each edge x, y of the graph (where d specifies the length of the edge). Since no node needs to be processed more than once, A* can be implemented more effectively in this situation. In this case, A* results to be identical to running Dijkstra's algorithm at a lower cost [29]. Figure 4.9 shows two different paths using both A* and Dijkstra algorithms.

The heuristic determines the time complexity of A*. The number of nodes visited is exponential in the worst scenario, but when the search space is a tree, it is polynomial.



(a) Example of the result of Dijkstra's algorithm **(b)** Example of the result of A*'s algorithm

Figure 4.9: Global path planning algorithms [27]

4.4.2 D* algorithm

There exist three different version of the D* algorithm: D*, Focused D* and D* Lite. The same path planning issues, such as planning under the free space assumption, where a robot must navigate to specified destination coordinates in unknown territory, are resolved by all three search methods. In order to identify the shortest route from its present coordinates to the objective coordinates, it makes assumptions about the unknown section of the terrain (for example, that it is empty of obstacles) and then the robot proceeds down the path. It updates its map as it notices new information (such as previously undiscovered barriers) and, if necessary, replans the shortest route from its present coordinates to the specified objective coordinates. The operation is repeated until the desired coordinates are reached or it is determined that they cannot be attained.

Anthony Stentz first presented the D* in 1994. Since the method behaves like A* with the exception that the arc costs can change while the algorithm runs, the name D* is derived from the term "Dynamic A*". Differently from A* algorithm, which traverse the graph from the beginning to the end, D* starts by looking backwards from the goal node. The exact cost to the goal is known by each expanded node, and each node has a backpointer that points to the next node heading to the target. The procedure is finished when the start node becomes the subsequent node to expand, at which point it is easy to determine the goal's location by simply tracing the backpointers.

4.5 Local Path Planning

The information about the actual world is updated over time as the robot advances along the global path in a dynamic or unknown environment. Local Path Planning is required to respond to the impediments and changes in the environment in accordance with the data supplied by the perception system in real-time. In a local path planner, a robot is often driven by a global path created using a global path planner strategy that connects a starting point and a target point. This path is the shortest path, and the robot follows it until it detects obstacles. The robot then performs an obstacle avoidance algorithm by veering off the path while simultaneously updating some crucial data, like the updated distance between the present position and the goal point. In this type of path planning, the robot must continually be aware of the distance between the goal point and its current location in order to exactly reach the objective.

4.5.1 Dynamic Window Approach

The Dynamic Window Approach (DWA), a velocity-based local planner, determines the ideal collision-free robot velocity needed to complete a task [30]. For a moving robot, a Cartesian target (x, y) is transformed into a velocity command (v, w). The fact that this approach considers the kinematic and dynamic restrictions of a mobile robot during planning is an advantage with respect to other local path planning algorithms. The two main goals of DWA are to identify a valid velocity search space and select the ideal velocity. The search space is the set of speeds that, given the set of speeds the robot can reach in the subsequent time slice, given its dynamics, allow it to follow a safe route or stop before collision. Figure 4.10 depicts the DWA approach.

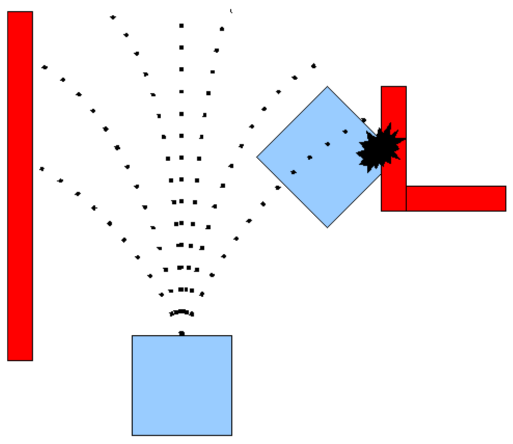


Figure 4.10: Dynamic Window Approach algorithm [31]

The strategy of the Dynamic Window Approach algorithm is the following:

1. Sample in the robot's control space $(dx, dy, d\theta)$
2. Perform a forward simulation from the robot's current state for each sampled velocity to see what would happen if it were used for a brief time period.
3. Using a metric that takes into account factors like distance to obstacles, proximity to the destination, proximity to the global path, and speed, evaluate each trajectory that emerges from the forward simulation. Throw away inaccurate trajectory (those that collide with obstacles).
4. Choose the trajectory with the best score, then send the corresponding velocity to the mobile base.
5. Rinse and repeat

4.5.2 Artificial Potential Field

The Artificial Potential Field (APF) method was introduced by Khabit and Krogh in 1995 [32]. The gradient descent search strategy, which seeks to minimize the potential function, is the basis of the APF methodology. An attractive potential field surrounds the goal point, while a repulsive potential field surrounds the obstacles that must be avoided. The attractive potential is often a bowl-shaped energy well that attracts an object toward its center if the environment is unobstructed. However, in a situation where there are impediments, attractive potential fields are supplemented with repulsive potential energy hills at the locations of the obstructions in order to repel the objects. The item is subjected to a force equal to the negative gradient of the potential. The object is pushed downward by this force until it reaches the spot where it uses the least amount of energy. The method is widely applied to path planning and real-time obstacle avoidance. The working principle of the Artificial Potential Field algorithm is shown in Figure 4.11

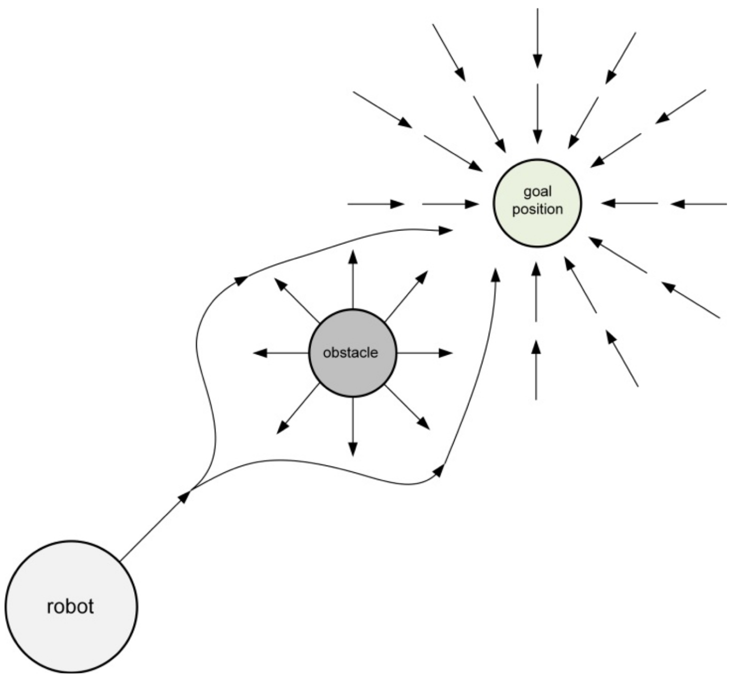


Figure 4.11: Artificial Potential Field [33]

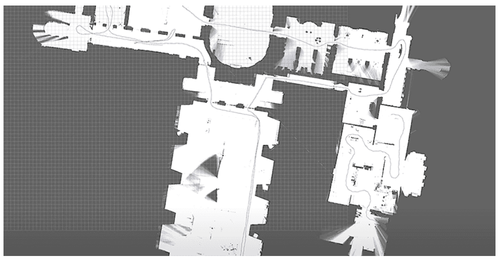
Chapter 5

Autonomous Navigation in Simulation

5.1 SLAM algorithms in ROS

Knowing where a robot is moving is essential for robotic applications. Simultaneous localization and mapping (SLAM) is a method that allows a robot to map its surroundings and simultaneously determine its location using information provided by on-board sensors. This method is advantageous because it allows the robot to move freely without having to know specific information about its surroundings in advance [34].

The SLAM challenge is addressed by a wide variety of algorithms. Based on the size of the map they produce, these algorithms can be divided into two classes: 2D and 3D. The 3D version uses more memory than the 2D version, and, in addition, the 3D version needs to estimate a full 6 DOF pose in contrast to the 2D version that simply needs to estimate a 3 DOF pose. Therefore, if the robot can be expected to move on a planar surface, the 2D SLAM form usually is enough.


(a) 2D map of an environment

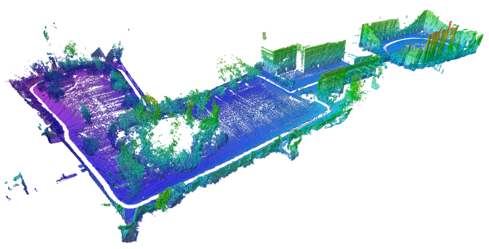

(b) 3D map of an environment

Figure 5.1: Examples of 2D and 3D maps generated during the SLAM process

5.1.1 ROS packages for SLAM

Among the various 2D and 3D SLAM algorithms, some of them have been developed in ROS and they are publicly accessible if form of *package* for developers. Each of them has peculiar characteristics, starting from the mathematical model that describes its behaviour and arriving to the performances of the algorithms, which can be more or less appropriate related to the task assigned. Some of these packages are listed hereafter:

- **gmapping**: The *gmapping* algorithm, which is based on a particle filter, is one of the solutions to the SLAM challenge [35]. When utilizing a particle filter for SLAM, a set of particles is used to approximate the robot's pose and its level of uncertainty. However, particle filters need a lot of particles to produce a reasonable output, which increases the computational complexity. The *gmapping* package requires as input odometry data and laser scans from a LiDAR sensor.

- **Google Cartographer**: Google Cartographer SLAM uses a graph-based approach rather than a particle filter as its foundation [24]. Another characteristic of this algorithm is the division of the map into a number of submaps, each of which contains a number of laser scans. All submaps are simply rasterized to create the final map. Google Cartographer is implemented in ROS using two packages:
  - *cartographer* is in charge of performing SLAM
  - *cartographer_ros* integrates the functionalities of the algorithm in ROS

- **Karto SLAM**: *slam_karto* is a graph-based SLAM algorithm. In this case, each node represents a pose of the robot along its trajectory and a set of sensor measurements. These are connected by arcs which represent the motion between successive poses. In the *slam_karto* version available for ROS, the Sparse Pose Adjustment (SPA) is responsible for both scan matching and loop-closure procedures [36].

- **Hector SLAM**: *hector_mapping* is a SLAM algorithm which integrates laser scan matching feature with the 3D navigation method by the help of the inertial system which employs the EKF. It is a SLAM algorithm which does not use the odometry data. Thus, the Hector SLAM has an advantage when used in environments which exhibit the pitch and roll characteristics. On the other hand, it might have problems when only low rate scans are available and it does not leverage when odometry estimates are fairly accurate [35].

5.2 Cartographer in ROS

Cartographer is a system that provides real-time simultaneous localization and mapping (SLAM) in 2D and 3D across multiple platforms and sensor configurations [24]. Figure 5.2 depicts the modules that compose Google Cartographer.

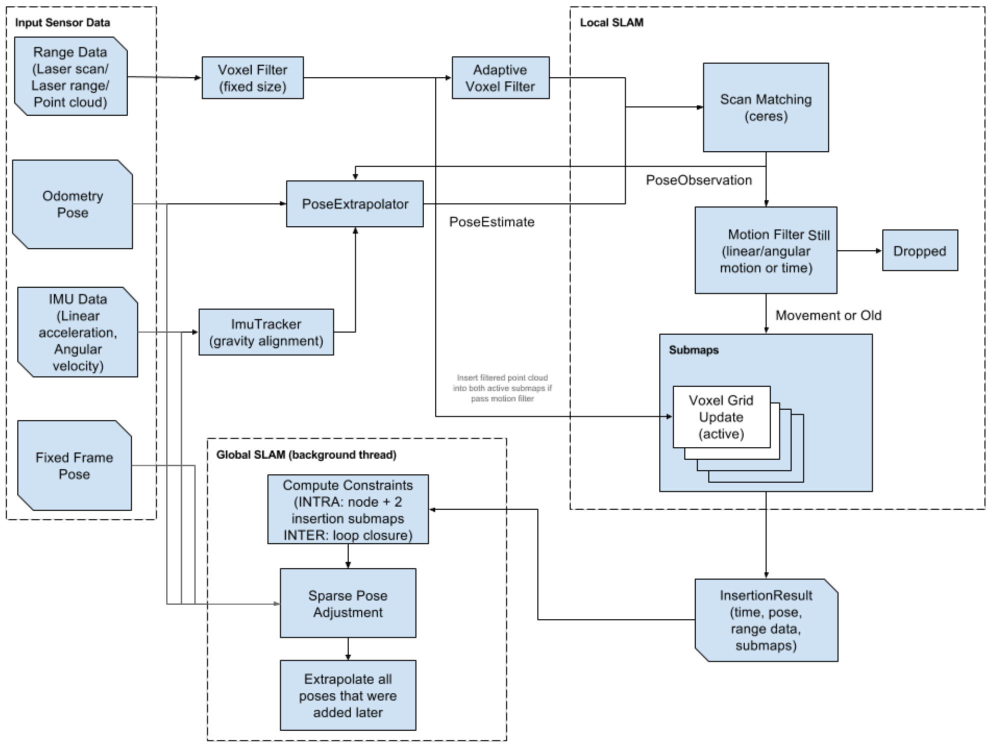

Input Sensor Data
Range Data (Laser scan/ Laser range/ Point cloud)
Odometry Pose
IMU Data (Linear acceleration, Angular velocity)
Fixed Frame Pose
Voxel Filter (fixed size)
Adaptive Voxel Filter
PoseExtrapolator
ImuTracker (gravity alignment)
Local SLAM
Scan Matching (ceres)
PoseObservation
PoseEstimate
Motion Filter Still (linear/angular motion or time)
Dropped
Movement or Old
Submaps
Voxel Grid Update (active)
Insert filtered point cloud into both active submaps if pass motion filter
Global SLAM (background thread)
Compute Constraints (INTRA: node + 2 insertion submaps INTER: loop closure)
Sparse Pose Adjustment
Extrapolate all poses that were added later
InsertionResult (time, pose, range data, submaps)

Figure 5.2: Google Cartographer overview [25]

5.2.1 Local SLAM

Cartographer can be seen as two separate, but related subsystems. Local SLAM is the first and its task is to create a series of submaps. The local SLAM algorithm can begin processing a scan after it has been put together and filtered from multiple range data. By using scan matching and an initial guess from the pose extrapolator, local SLAM adds an additional scan to the construction of its current submap. To forecast where the next scan should be included into the submap, the pose extrapolator uses sensor data from sensors besides the range finder. There exist two different scan matching strategies:

- The best location where the scan match fits the submap is found by the **CeresScanMatcher** using the previous guess as a starting point. This is accomplished by subpixel aligning the scan and interpolating the submap. Although quick, this can't correct problems that are much bigger than the resolution of the submaps. The ideal option is typically to use only the **CeresScanMatcher** if your sensor setup and timing are suitable. The ideal option is typically to use only the **CeresScanMatcher** if your sensor setup and timing are suitable.

- If there aren't any additional sensors, **RealTimeCorrelativeScanMatcher** can be enabled. It adopts a strategy analogous to how scans are matched to submaps in loop closure, but matches against the current submap. The CeresScanMatcher then uses the best match as a reference. This scan matcher is quite expensive and effectively cancels out all other sensor signals except for the range finder's, but it is reliable in situations with lots of features.

The majority of the local SLAM settings can be found in the *install_isolated* directory inside the *trajectory_builder_2d.lua* configuration file for 2D and *install_isolated* directory inside the *trajectory_builder_3d.lua* configuration file for 3D.

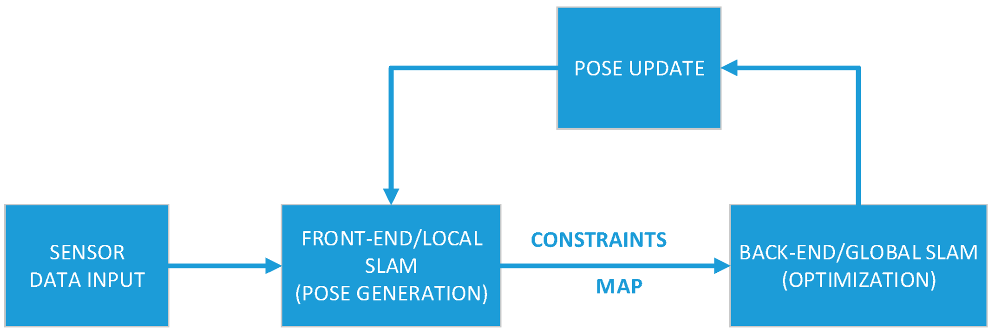

POSE UPDATE
SENSOR DATA INPUT
FRONT-END/LOCAL SLAM (POSE GENERATION)
CONSTRAINTS
MAP
BACK-END/GLOBAL SLAM (OPTIMIZATION)

Figure 5.3: Interaction between Local and Global SLAM [24]

5.2.2 Global SLAM

The global optimization task runs in the background as the local SLAM creates its series of submaps. Rearranging submaps so they produce a coherent global map is part of its functionality. This optimization, for instance, is responsible for modifying the trajectory that is now being generated to properly align submaps with regard to loop closures. The global SLAM is an example of a GraphSLAM, which essentially optimizes the pose graph by creating constraints between nodes and submaps and then optimizing the resulting constraints graph. Intuitively, constraints can be compared to little ropes connecting each node. Those ropes are completely fastened by the sparse pose adjustment. The *position graph* is the name given to the generated net. Figure 5.4 describes a map generated using Cartographer as SLAM algorithm.

- Automatically generated **non-global constraints** are formed between nodes that are closely following each other along a trajectory. These "non-global ropes" maintain the trajectory's local structure's coherence.
- **Global constraints** are continuously checked between a new submap and prior nodes that are considered to be "near enough" in space. Those "global ropes" securely connect two strands together while also introducing knots into the structure.

The majority of the global SLAM settings can be found in the *install_isolated* directory inside the *pose_graph.lua* configuration file.

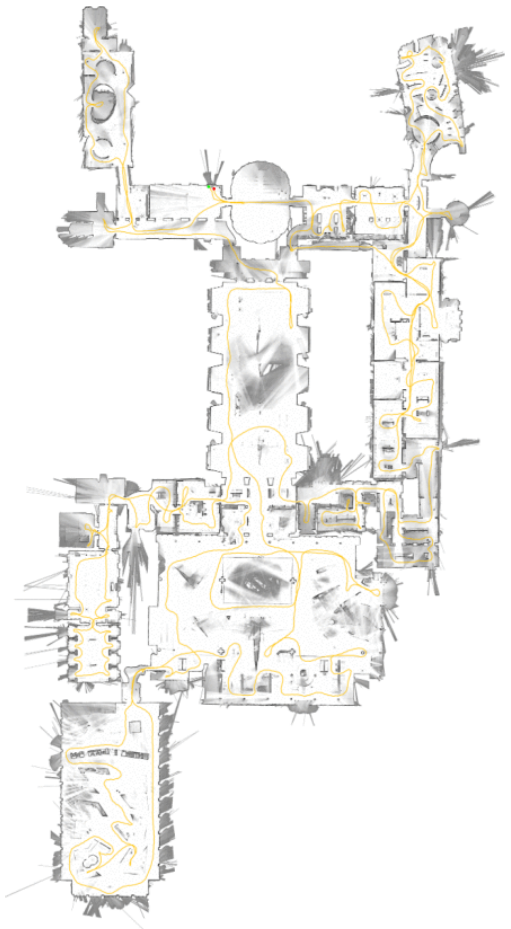


Figure 5.4: Cartographer map of the Deutsches museum [25]

5.2.3 Input Sensor Data

Sensors that use range finding offer depth data in many different directions. Some of the measurements, however, are not important for SLAM. Some of the measured distance can be interpreted as noise for SLAM if the sensor is partially covered in dust or if it is pointed at a portion of the robot. On the other hand, some of the most distant readings might also originate from undesirable sources (reflection, sensor noise), and they are equally unimportant for SLAM. Cartographer begins by applying a bandpass filter and only keeps range values between a specific min and max range to address those issues. These minimum and maximum values need to be selected in accordance with the robot's and the sensors' specifications.

5.3 Navigation Stack

Autonomous navigation is accomplished using the Navigation stack. It is a set of ROS packages that, using data from sensors and odometry sources , provides safe velocity instructions to the robot in order to reach the determined final position. ROS Navigation stack requires the transform tree (provided by *tf* package) of the robot (holonomic or differential drive). A robot can be described as a system made up of numerous components, each of which can be simply represented by a coordinate frame that is connected to the another component and is identifiable by a location and orientation in space. Therefore, finding a shared reference system where the transformations between the frames and their relationships will exist, is crucial, as shown in Figure 5.5.

The most important *tf* components used in the Navigation stack are:

- *odom* represents the odometry reference frame
- *base_link* represents the base of the robot
- *base_laser* represents the base of the sensor
- *base_footprint* represents the *base_link* projection onto the ground
- *map* represents the environment where the robot is inserted.

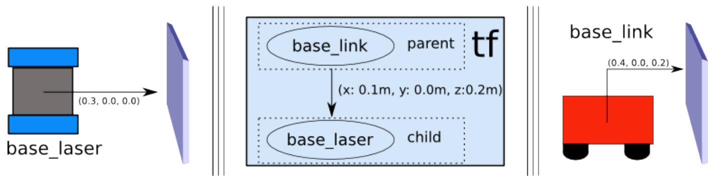


Figure 5.5: Relationship between the *base_link* and *base_laser* frames using *tf*

For 2D autonomous navigation in indoor environments, the Navigation stack is frequently utilized. The main package that compose the *navstack* is the *move_base* and is in charge of computing velocity commands to reach the final goal. It contains the *global_planner*, *local_planner*, *global_costmap* and *local_costmap*, as well as *amcl*, which carries out the robot's localization, and *map_server*, which provides the reference map. Figure 5.6 describes the overall Navigation stack system.

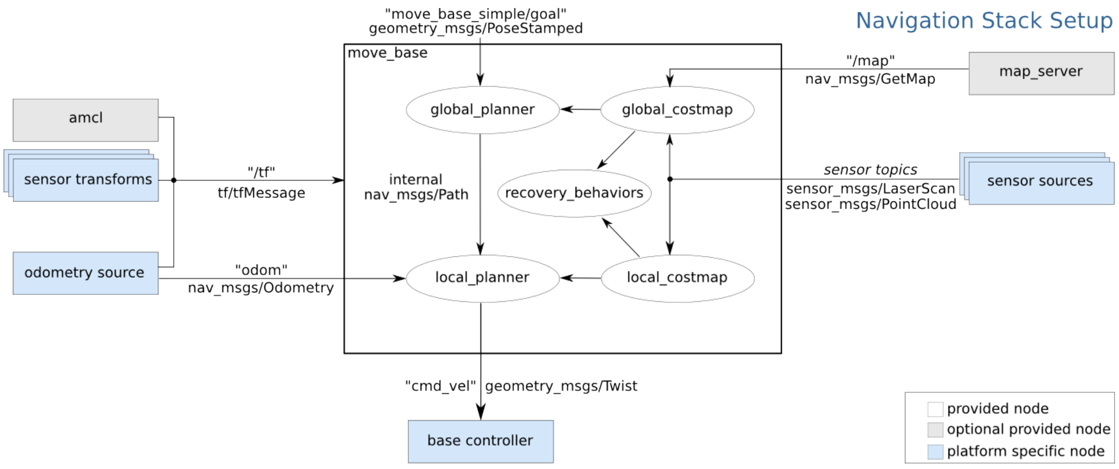


Figure 5.6: Navigation Stack setup [37]

5.3.1 Odometry information

The navigation stack employs the *tf* function to locate the robot in space and correlate sensor data to a static map. *tf*, however, has no details regarding the robot's velocity. As a result, the navigation stack necessitates that each source of odometry that publishes a transform and an *nav_msgs/Odometry* message (5.1) over ROS include velocity information.

Listing 5.1: The *nav_msgs/Odometry* message

```
# This represents an estimate of a position and velocity in free space.
# The pose in this message should be specified in the coordinate frame given by header.frame_id.
# The twist in this message should be specified in the coordinate frame given by the child_frame_id
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

The estimated pose of the robot in the odometric frame is represented by the pose in this message, which may also include an optional covariance to increase the accuracy of the pose estimation. The twist in this message corresponds to the robot's velocity in the child frame, which is often the mobile base's coordinate frame, as well as an optional covariance for the precision of that velocity estimate.

5.3.2 Sensor information

The navigation stack needs to publish sensor data correctly through ROS in order to work safely. Robots that have no data from their sensors will be driving blind and will be more inclined to crash into objects. The navigation stack may receive data from a variety of sensors, including lasers, cameras, sonar, infrared, bump sensors, and more. The data coming from the sensors must be published using either the *sensor_msgs/LaserScan* message (5.2) type or the *sensor_msgs/PointCloud* message type in order to be accepted by the navigation stack.

For robots equipped with laser scanners, ROS offers a unique message type called *LaserScan* in the *sensor_msgs* package to store data regarding a specific scan. Any laser can be used with *LaserScan* messages as long as the data returned by the scanner can be formatted to fit within the message.

Listing 5.2: The *sensor_msgs/LaserScan* message

```
#
# Laser scans angles are measured counter clockwise, with 0 facing forward
# (along the x-axis) of the device frame
#

Header header
float32 angle_min        # start angle of the scan [rad]
float32 angle_max        # end angle of the scan [rad]
float32 angle_increment  # angular distance between measurements [rad]
float32 time_increment   # time between measurements [seconds]
float32 scan_time        # time between scans [seconds]
float32 range_min        # minimum range value [m]
float32 range_max        # maximum range value [m]
float32[] ranges         # range data [m] (Note: values < range_min or > range_max should be discarded)
float32[] intensities    # intensity data [device-specific units]
```

5.3.3 Base controller

The navigation stack expects that it can command the robot to move forward by sending velocity commands over the *cmd_vel* topic using a *geometry msgs/Twist* message. This means that the (vx, vy, vtheta) ==> (cmd vel.linear.x, cmd vel.linear.y, cmd vel.angular.z) velocities must be converted into motor commands and sent to a mobile base by a node that subscribes to the *cmd_vel* topic. For this purpose platforms for base control are used. The packages and the drivers used in this area are specific to the robot, and a special purpose controller is programmed.

5.3.4 Costmap

The *costmap_2d* package offers a flexible structure that maintains an occupancy grid containing information on the robot's navigation path. The costmap stores and updates information about obstacles in the world using the *costmap_2d::Costmap2DROS* object while using sensor data and information from the static map. Through ROS, the costmap automatically subscribes to sensor topics and updates itself as needed. Each sensor can then either **mark** an obstacle (add information about it to the costmap) or **clear** an obstacle (remove it from the costmap), or both.

The underlying structure that the costmap utilizes can only represent three cost values, even if each cell can assume one of 255 different cost values. Each cell in this structure has three possible states: **free**, **occupied**, and **unknown**. Upon projection into the costmap, a unique cost value is assigned to each status. A cost of *costmap_2d::LETHAL_OBSTACLE* is given to columns with a specific number of occupied cells, a cost of *costmap_2d::NO_INFORMATION* is given to columns with a specific number of undetermined cells, and a cost of *costmap_2d::FREE_SPACE* is given to other columns.

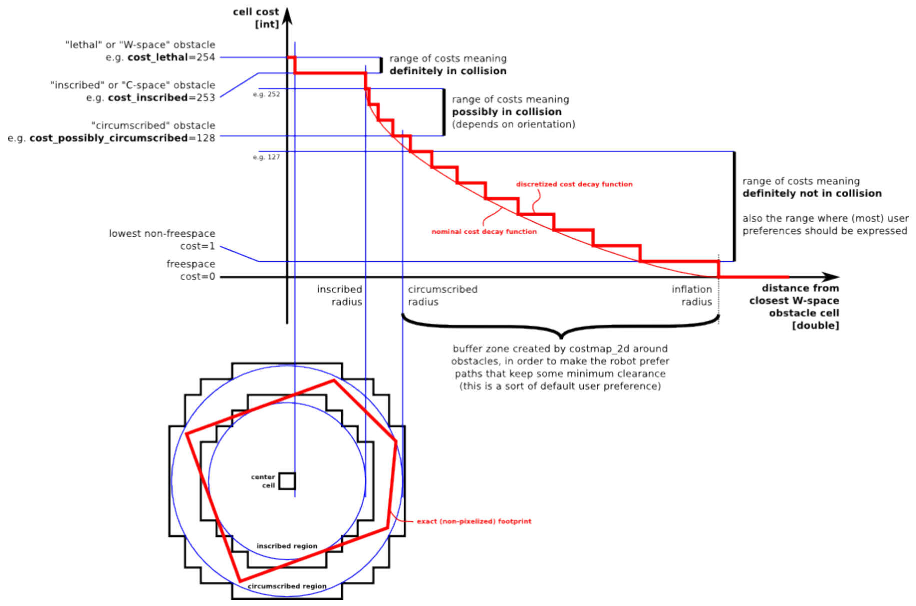

cell cost
[int]
"lethal" or "W-space" obstacle
e.g. cost_lethal=254
"inscribed" or "C-space" obstacle
e.g. cost_inscribed=253
"circumscribed" obstacle
e.g. cost_possibly_circumscribed=128
e.g. 252
e.g. 127
lowest non-freespace
cost=1
freespace
cost=0
range of costs meaning
definitely in collision
range of costs meaning
possibly in collision
(depends on orientation)
range of costs meaning
definitely not in collision
also the range where (most) user preferences should be expressed
discretized cost decay function
nominal cost decay function
inscribed radius
circumscribed radius
inflation radius
distance from closest W-space obstacle cell [double]
buffer zone created by costmap_2d around obstacles, in order to make the robot prefer paths that keep some minimum clearance (this is a sort of default user preference)
center cell
exact (non-pixelized) footprint
inscribed region
circumscribed region

Figure 5.7: Inflation parameters

Inflation is the process of spreading the cost values of occupied cells outward as a function of distance, as shown in Figure 5.7. For this purpose, five different symbols for cost values can be related to a robot.

- **Lethal** cost indicate that certainly there is an obstacle in a predefined cell. In this case, the robot would plainly be in collision if its center were in that cell.
- A cell has a **Inscribed** cost if it is closer to an actual obstacle than the robot's inscribed radius. Therefore, if the robot center is in a cell that is at or over the inscribed cost, the robot is undoubtedly colliding with some obstacle.
- Similar to inscribed, **possibly circumscribed** cost uses the robot's circumscribed radius as the lower limit. It thus depends on the orientation of the robot whether it collides with an obstacle or not, if the robot center is located in a cell at or above this value.
- Since the cost of **Freespace** is believed to be zero, the robot shouldn't be prevented from travelling there.
- A cell with **Unknown** cost indicates that there is no information available. This can be interpreted however the costmap user sees fit. In the Active SLAM application, cells with unknown cost are fundamental because,going in that direction, they allow the robot to discover new information from the environment.

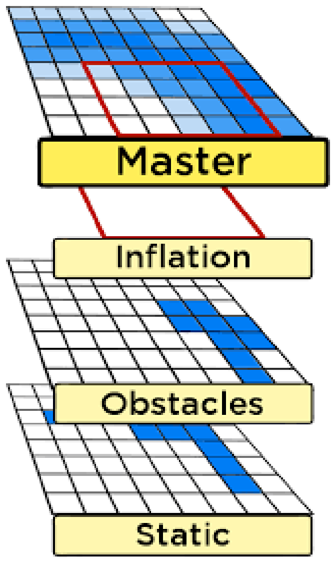


Figure 5.8: Layered costmap [38]

A costmap is used to hold the environmental data that the path planners utilize. In a classic costmap, also called monolithic costmap, all the data is kept in a single grid of values. Due to its simplicity-there is only one area to read from and write values to-the *monolithic costmap* has become the dominant technique. In order to add context and semantical information to the costmap, *layered costmaps* are used. The most important layers used include Static Map Layer, which is the bottom layer of the global costmap and directly transfers its value into the master costmap to know where walls and obstacles are. This layer is often created only using information coming from the SLAM algorithm. Sensor data was gathered by the Obstacles Layer, which organized the data in a 2D grid. Instead, the Inflation Layer is in charge of adding a buffer zone around each lethal obstacle in order to prevent the robot from colliding into it [38].

5.3.5 Move Base

The Move Base module can be considered the core element of the Navigation Stack, and it's composed of four elements: *Global costmap*, *Local costmap*, *Global planner* and *Local planner*.

During the navigation, the interaction between these modules allow the robot to safely move inside an environment:

- The **Global Costmap** is represented by an occupancy grid map. Different parameters that characterize the costmap like the resolution and the dimension can be easily changed by acting on the *global_costmap.yaml* file. The Global costmap is used by the global planner to generate a long-term navigation plan, as shown in Figure 5.9.

- The **Local Costmap** takes information from the Global costmap and the parameters that describes its function can be found inside the *local_costmap.yaml* file. The Local costmap is used by the local planner to generate a short-term navigation plan. The Figure 5.10 shows an example of Local costmap.

- The global path is calculated by the **Global Planner** from a starting location to an ending position while avoiding a collision with any obstacles that may be encountered. On the basis of the given global costmap, the Global planner selects the lower-cost itinerary. The parameters that describe the Global planner can be set inside the *global_planner.yaml* file. There are different algorithms that can be exploited for global path planning and, according to their features, selected for the navigation as shown in Figure 5.11. Among the most widely used algorithms there are A* and Dijkstra.

- The **Local Planner** gives velocity commands to the mobile platform traveling along the global path. The Local planner is used also to avoid obstacles found in the local costmap. The behavior of the Local planner can be modulated through the *local_planner.yaml* file. Among the most widely used algorithms for local path planning there is *dwa_local_planner* which is the one used as default by the Navigation Stack and is an implementation of the Dynamic Window Approach (DWA) algorithm. Figure 5.12 depict a Local planner in RViz.

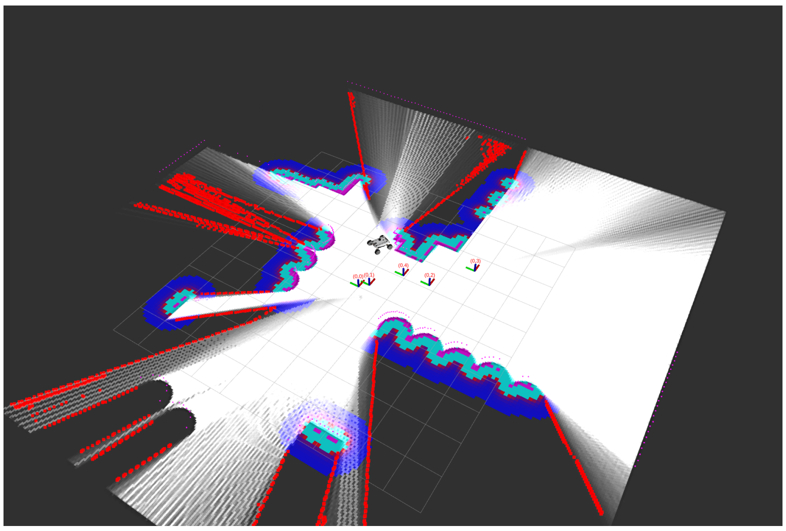


Figure 5.9: Global Costmap in RViz

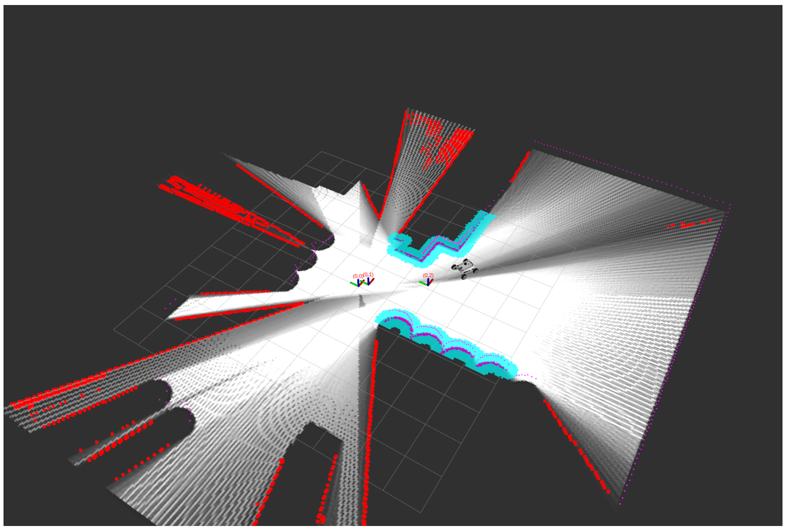


Figure 5.10: Local Costmap in RViz

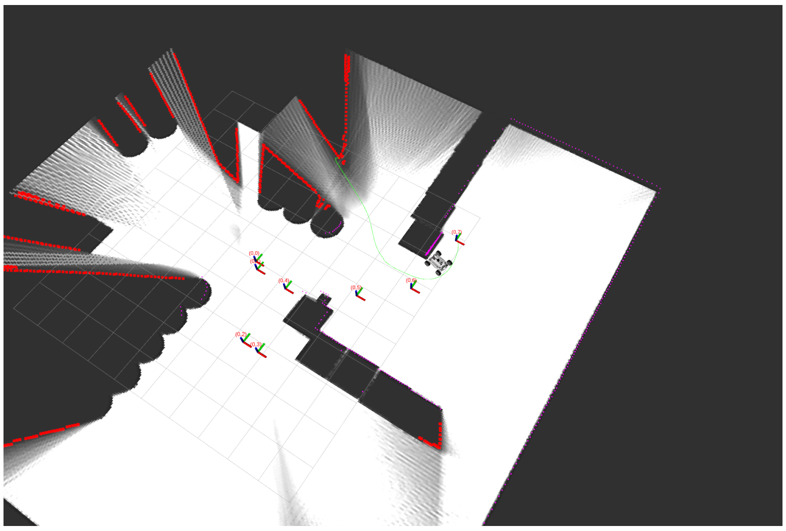


Figure 5.11: Global Planner colored in green shown in RViz

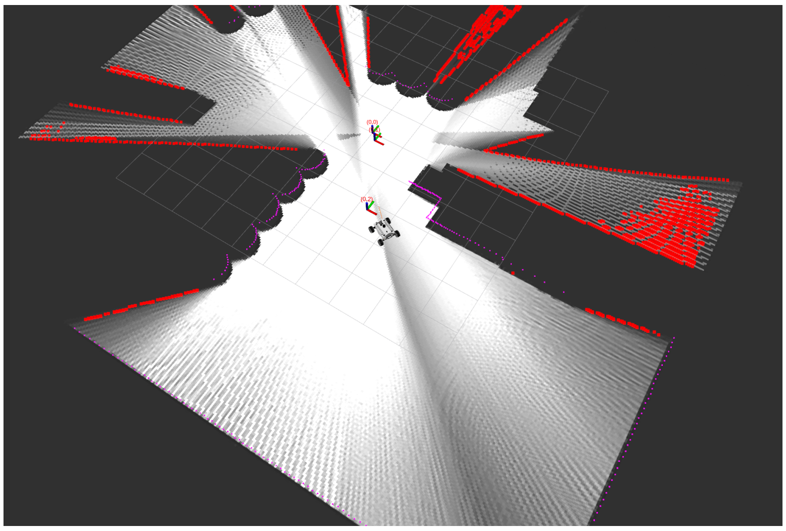
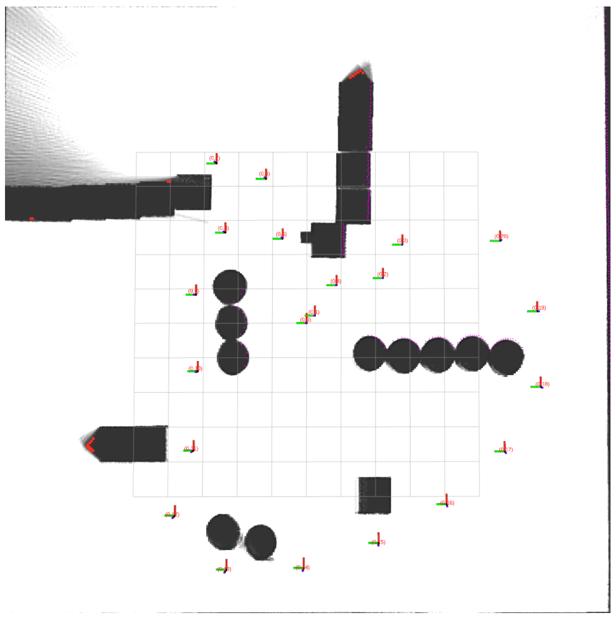


Figure 5.12: Local Planner colored in yellow shown in RViz

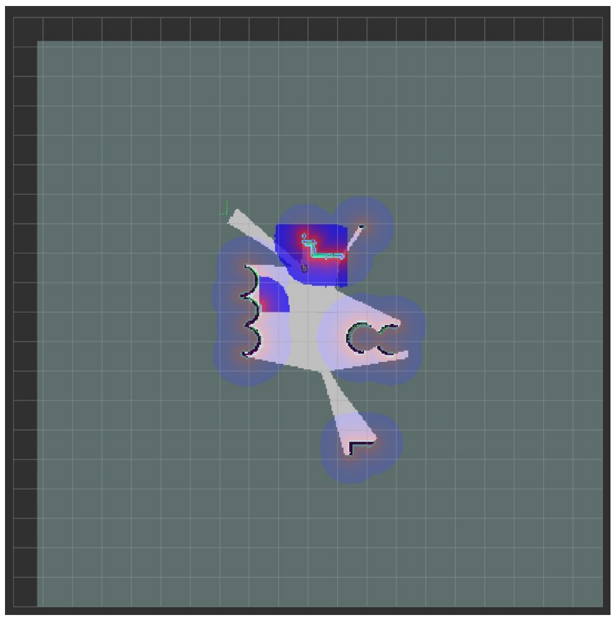
5.4 Simulation

5.4.1 Active SLAM performance comparison

The Active SLAM algorithm developed was tested extensively in a simulation environment using both RViz and Gazebo, and its performances were compared to a different Active SLAM method, proving the effectiveness and the novelty of the programmed solution.



(a) Google Cartographer used as SLAM module



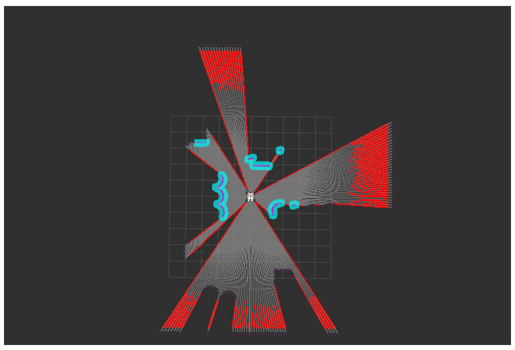
(b) Gmapping used as SLAM module

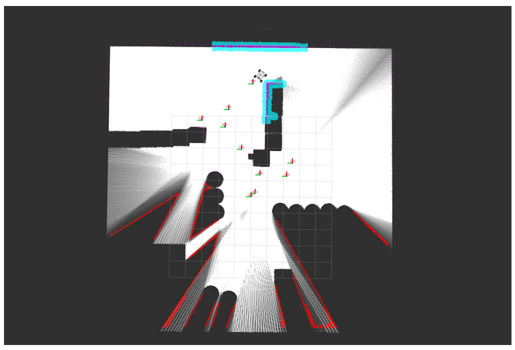
Figure 5.13: Comparison between the two Active SLAM solutions

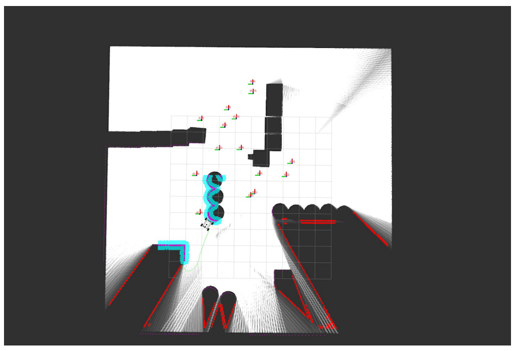
The developed method use Google Cartographer as SLAM module, and it's followed by a Frontier Detection step to detect navigation points and a Path Planning algorithm to reach unexplored areas. It is compared with another Active SLAM algorithm, which differently from the previous method employs Gmapping as the main SLAM module [39]. The selected metrics of comparison have considered the time required to discover the same environment and the quality of the map created at the end of the process. The results, as shown in Figure 5.13, demonstrate the better result obtained with the developed Active SLAM algorithm in both time required and quality of the map. The map results, depicted in the picture, were both taken after approximately three minutes of simulation, the time required by the first method to complete its mission, which was not sufficient for the success of the second method.

5.4.2 Simulation Results

The steps of the simulation are shown in Figure 5.14.


(a) The robot start sensing the environment and the frontiers (red points) are identified


(b) After selecting the frontier points the exploration starts


(c) The autonomous exploration continues, and the map is taking shape

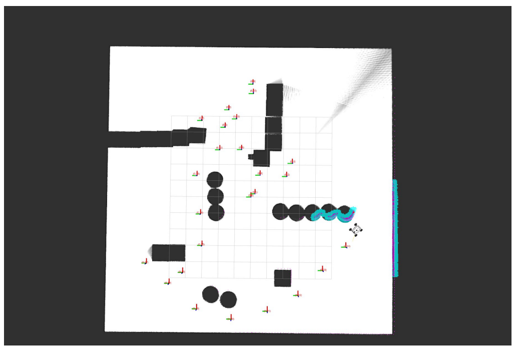

(d) The exploration is finished and the map is complete

Figure 5.14: The process of Autonomous Exploration

Chapter 6

Hardware Architecture

The development of the autonomous mobile robot begins with selecting the right sensors based on the type of application. The robot is made out of four Mecanum wheels to start. These wheels may move in any direction, allowing for both rotation and translation. Encoders are frequently employed in self-driving robots to determine location and, consequently, odometry. However, the number of rotations required to move laterally and longitudinally are different and vary depending on the kind of ground, therefore using encoders alone is insufficient to obtain accurate information. For this purpose, position data coming from encoders is merged with an Inertial Measurement Unit (IMU) using Unscented Kalman Filter (UKF). Moreover, the platform is equipped with two 2D LiDAR sensors used for mapping and navigation purposes. Due to the physical structure of the FIXIT case, as shown in figure 6.1, two LiDAR are necessary to make sure the rover has a 360°view of the environment. A partial view could be dangerous in highly dynamic spaces, especially during obstacle avoidance tasks. LiDAR usage is essential for a variety of applications. In addition, a laser scan is frequently used to collect a map since, thanks to its extended range, it can cover a large area without passing by that location again and prevent accumulating errors. Even if they are essential for navigation, 2D LiDAR sensors present some limitations caused by the fact that they only provide an horizontal view of a scene. Laser scans, even supplying a 360°field of view on the (x, y) plane, they lack in giving information about the z axis. The result is that obstacles with a lower height than the height of the sensor are not detected, as shown in Figure 6.2. To overcome this problem, two depth cameras are placed on the front and on the back of the case in order to add a vertical field of view to the system. Doing that, the rover is capable of safely navigate and avoid obstacles of different heights.

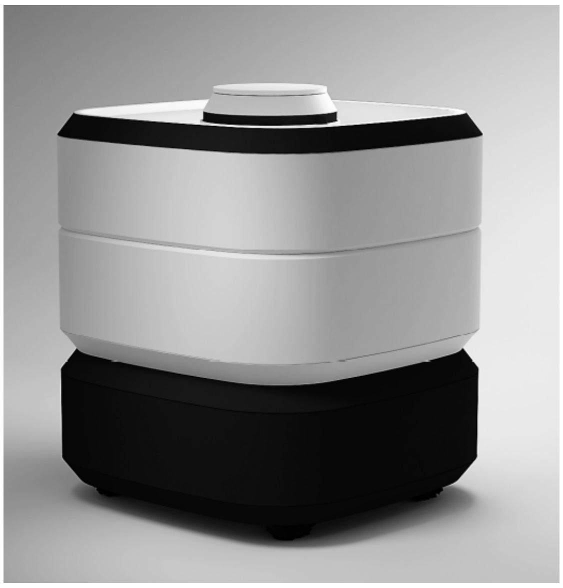


Figure 6.1: FIXIT structure

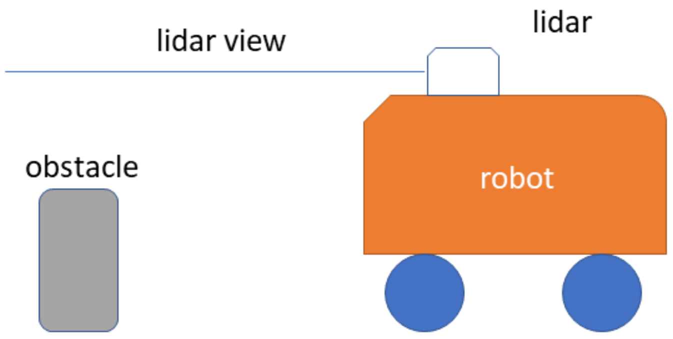


Figure 6.2: LiDAR limitations

6.1 Mecanum Wheeled mobile robot

Introducing wheeled mobile robots it is important to classify the different types of wheels with respect to their properties. A first distinction can be made talking about Conventional Wheeled mobile Robots and Mecanum Wheeled mobile Robots. Conventional types of wheels encapsulate three wheel models: fixed wheels, stereable wheels and caster wheels. These types of wheels have different characteristics based on the combination of different rotation axes. Each one of this models has a peculiar property, which can be useful to give a particular behaviour to the considered mobile robot. The Mecanum Wheels instead are composed by several rollers placed around the wheel with an angle of 45 °, as shown in Figure 6.3.

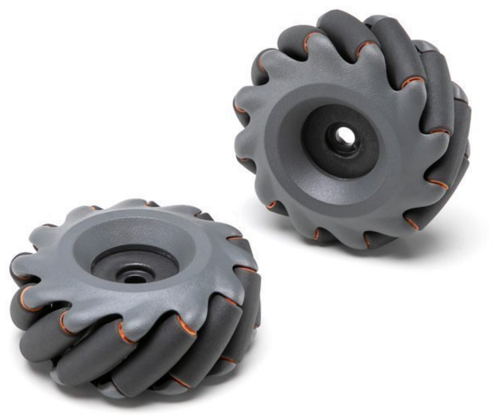


Figure 6.3: Mecanum Wheels

This particular configuration of the wheels guarantee movement even in directions parallel to their axes, and this feature is fundamental to allow the rover to move in narrow environments and to permit a better obstacle avoidance performance. For this reason a omni-wheel based robot is the optimal choice for dynamic environments, as, for example, a working environment. Figure 6.4 shows the movements possibility given by the wheels to the rover.

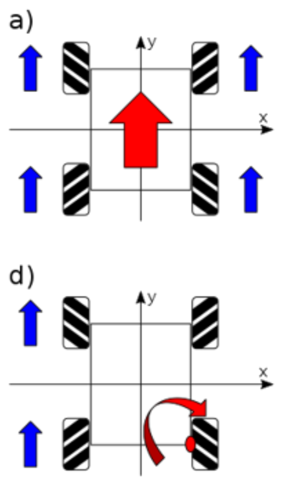
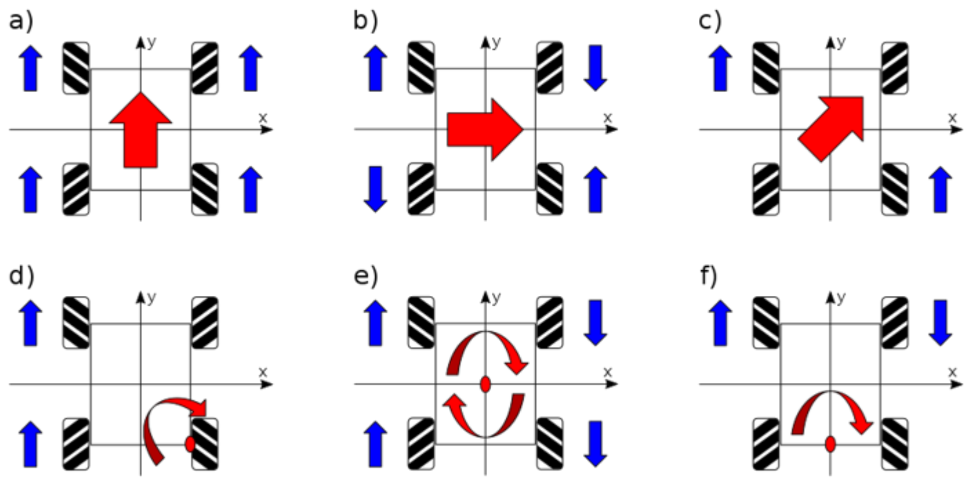



Figure 6.4: AMR movements allowed by mecanum wheels

6.2 Sensors

As described before, the mobile platform is equipped with different sensors, which cooperate to guarantee an efficient SLAM and a safe autonomous exploration. The sensors deployed, apart from the encoders and IMU which are used mainly for localization purposes, are two RP-LIDAR A1 and two Intel RealSense Depth Camera D435i, employed to active perceive the environment.

6.2.1 RP-LIDAR A1

The RP-LIDAR A1 laser scanner is utilized in the system in question (Figure 6.5). Due to its low cost and small size, it is frequently used in robotics and in particular for autonomous exploration, localization, and mapping. It is also capable of sensing a 360-degree rotating environment. It is a 2D laser scanner made by SLAMTEC, consisting of a range scanner that revolves around a motor with a belt attached in the opposite direction. Furthermore, it makes advantage of high-speed vision acquisition and is based on the laser triangulation ranging principle. In particular, the RPLIDAR generates an infrared laser signal, and the vision acquisition module subsequently detects and samples the returning signal. It scans a distance of 12 meters at a rate that can be configured, from 2Hz to 10Hz. The Table 6.1 describes in detail the characteristics of the sensor.

Figure 6.5: RP-LIDAR A1 [40]

	Parameters	**Description**
Physical	Width x Length x Height:	96.8 x 70.3 x 55 mm
	Weight:	170 g
Features		Indoor/Outdoor
	Use:	0.15m - 12m
	Measuring Range:	$\leq$ 1% of the range $\leq$12m
	Range Resolution:	8K
	Sampling Frequency:	5.5Hz
	Rotational Speed:	5V
	System Voltage:	100mA
	System Current:	0° C-40° C
	Temperature Range:	360°
	Angular Range:	$\leq$ 1°
	Angular Resolution:	1% of the range $\leq$ 3 m
	Accuracy:	2% of the range 3-5 m
		2.5% of the range 5-25 m

Table 6.1: RP-LIDAR A1 Datasheet

6.2.2 Intel RealSense Depth Camera D435i

Due of its excellent performance and low cost, this type of sensor is frequently employed in robotics. A depth camera and an Inertial Measurement Unit (IMU) are both present in this device (Figure 6.6). The accelerometer, which measures the overall force exerted on the device, and the gyroscope, which measures the angular velocity, are the IMU components utilized for this kind of application. Together, they provide the 3D space orientation. Due to its wide field of view (FOV) and low sensitivity to light, it also enables navigation in space when there is no light [41]. Additionally, navigation is possible both indoors and outdoors without interruption thanks to a system with a multi-camera configuration that operates at low power and has good precision within a few meters. The Intel RealSense d435i is composed

Figure 6.6: Intel RealSense D435i [41]

of various parts, as shown in Figure 6.7. Additionally, the gathered data generate RGB and depth images at the same time. It is made up of an RGB module with a 1920x1080 frame resolution, two infrared modules that can capture infrared images, and an IR projector that boosts the depth camera's performance using an active stereo technique. With a depth resolution of 1280x720 at 90 frames per second, the depth field of view is 87°x58°. It has an inbuilt Intel RealSense Vision Processor D4 that allows for a thorough environment reconstruction while processing the captured photos. The greatest visual range is 10 meters, although accuracy varies depending on different setting parameters as calibration, and lighting. The official characteristics listed by the vendor are shown in the Table 6.2.

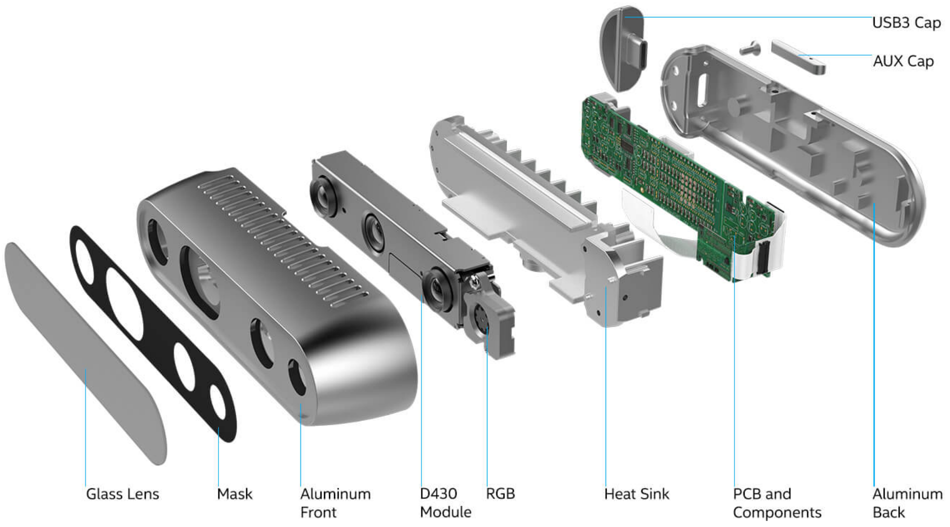


Figure 6.7: Intel RealSense D435i components [41]

	Parameters	**Description**
Physical	Length x Depth x Height: Connectors:	90mm x 25mm x 25mm USB-C 3.1
Components	Camera module: Vision processor:	Module D430 + RGB Camera Vision Processor D4
Features	Use: Ideal range:	Indoor/Outdoor 0.3 m to 3m
RGB	RGB frame resolution: RGB sensor FOV (H x V): RGB sensor technology: RGB frame rate:	1920 x 1080 69° x 42° Rolling Shutter 30 fps
Depth	Depth technology: Depth output resolution: Depth Field of View (FOV): Depth frame rate: Depth Accuracy:	Stereoscopic 1280 x 720 87° x 58° 90 fps <2% at 2m

Table 6.2: Intel Realsense Camera d435i Datasheet

6.3 On-board computers

The rover is equipped with two boards on which is stored and run the developed Active SLAM solution. The on-boards computer are a Nvidia Jetson Xavier NX and a Nvidia Jetson Nano and they are connected with the deployed sensors, in order to acquire information about the environment during the exploration phase.

6.3.1 Nvidia Jetson Xavier NX

This board is a part of the NVIDIA Jetson, a high-performance, low-power embedded platform (Figure 6.8). In particular, the Jetson Xavier NX integrates a CPU and GPU into a single, compact chip measuring about 70 mm by 45 mm. Due to its integrated software libraries, such as CUDA, it is adapted for real-time execution problems. It is developed for robot applications and autonomous tasks. Additionally, it has four 3.1 USB ports, needs a MicroSD card to function, and supports several power modes. The Table 6.3 shows in detail the features of the board.

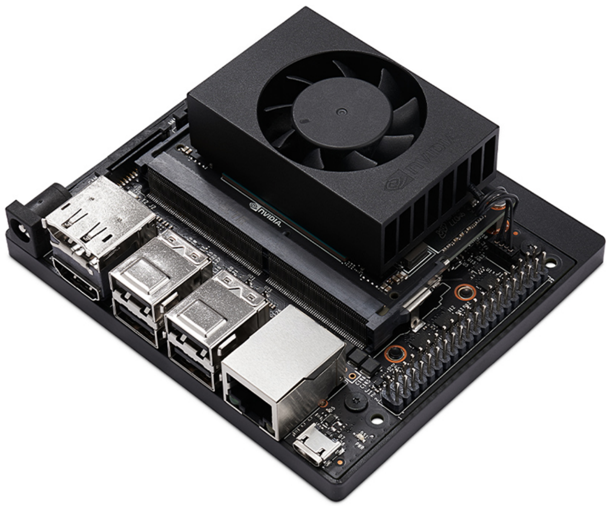


Figure 6.8: Nvidia Jetson Xavier NX [42]

6.3.2 Nvidia Jetson Nano

Due to its great performance, low cost, and smallest size in the Jetson family, this type of board is frequently employed in robotic applications (Figure 6.9). Furthermore, if set to high performance, it only consumes 10 W of power. Due to its trade-off between processing power and low power consumption, it is frequently used for image processing. It has an ARM Cortex CPU and a 128 core Maxwell GPU. The absence of a wifi module on this board constitutes its lone drawback. In order to solve this issue, Jetson Nano has been equipped with an Intel dual-mode wireless module. Table 6.4 describes in detail the functionalities of the board.

Parameters	**Description**
Width x Length x Height:	103 mm x 90,5 mm x 34 mm
GPU:	Nvidia Volta architecture with 384 Nvidia Cuda cores and 48 Tensor cores
CPU:	6-core Nvidia Carmel ARM v8.2 64-bit CPU 6 MB L2 + 4 MB L3
Memory:	8 GB 128-bit LPDDR4x
Connectivity:	Gigabit Ethernet Wi-Fi module
Display:	HDMI and display port
USB:	4x USB 3.1, USB 2.0Micro-B

Table 6.3: Jetson Xavier NX Datasheet

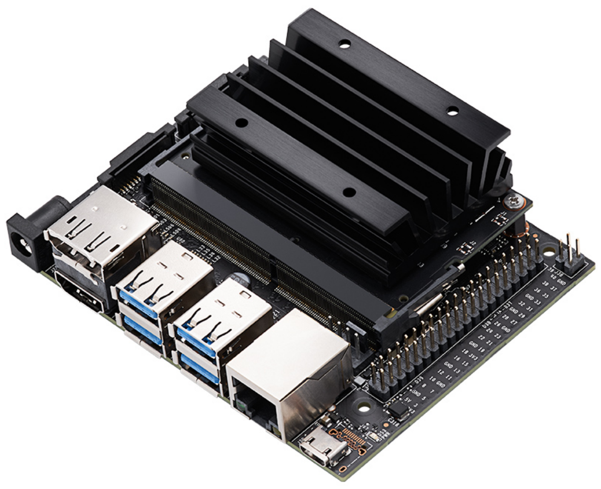


Figure 6.9: Nvidia Jetson Nano [43]

6.3.3 FIXIT-M Board

The FIXIT-M is a main board designed to have a battery based system that allows FIXIT to supply all its peripherals (Figure 6.10). In particular, the board was designed to provide a charger system to the drone when landing on the case and to give power supply to the boards needed by the AMR. The board provide a 12V and a 5V connectors to feed respectively the Jetson Xavier NX and the Jetson Nano. The final idea is to give power supply to the peripherals used by the AMR through an external battery instead the rover is supplied by its internal battery system. In this way, the overall system guarantee a much longer performance in

Parameters	Description
Width x Length:	69 mm x 45 mm
GPU:	128-core Maxwell
CPU:	Quad-core ARM A57
Memory:	4 GB 64-bit LPDDR4
Connectivity:	Gigabit Ethernet
Display:	HDMI and display port
USB:	4x USB 3.0, USB 2.0 Micro-B

Table 6.4: Jetson Nano Datasheet

terms of battery life.

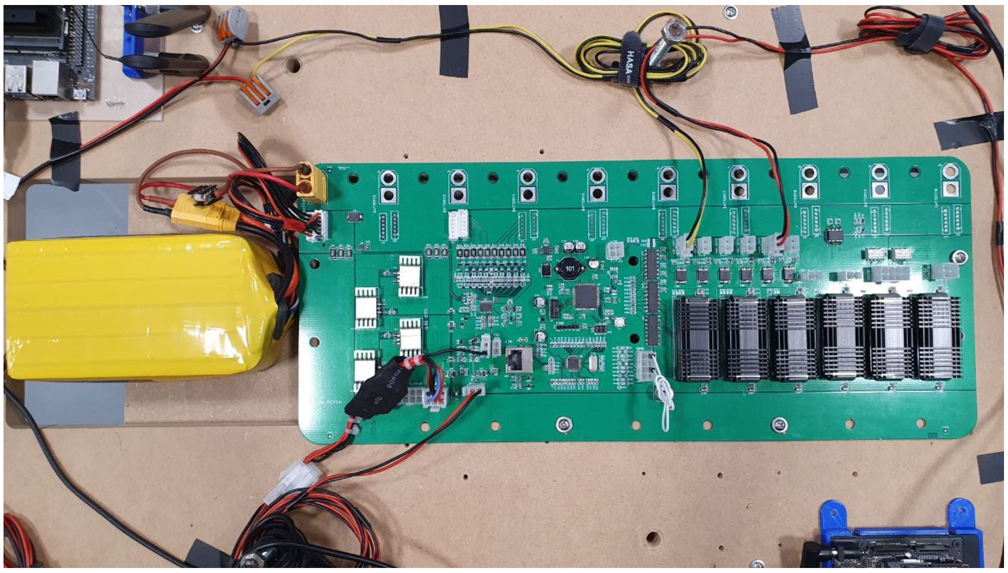


Figure 6.10: FIXIT-M board

6.4 Agilex Scout Mini

The robot in use is the Scout Mini mobile base from AgileX Robotics, shown in Figure 6.11, which is a smaller version of the Scout 2.0. Due to its speed, agility, compactness, and compatibility, it ranks among the top autonomous mobile robots on the market. It can be used to perform a variety of tasks, including surveillance, exploration, incarceration, as well as different educational and logistical services because it is ROS-compatible. This type of robot typically has a maximum load capacity of 10 kg, however the robot utilized in this system has four Mecanum

Wheels, which ensure a payload of up to 20 kg. These unique wheels enable translation and rotation in any direction. Additionally, it manages to create a high-speed, precise, stable, and adjustable power control system of 10.8km/h while having a very tiny size of 625x585x222 (L W H (mm)) and a weight of 23 kg. It is

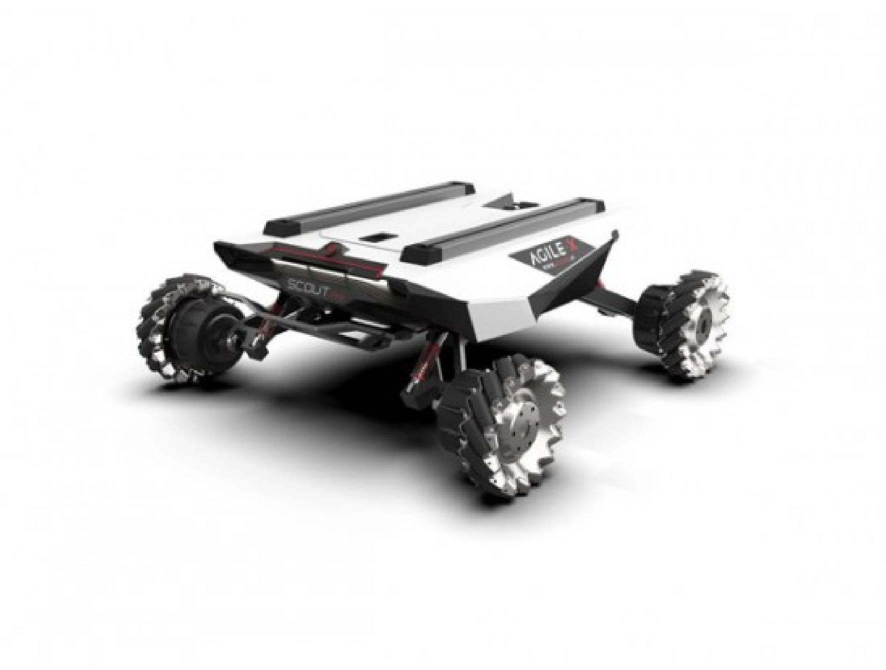


Figure 6.11: Aviation Agilex Scout Mini [44]

easy to install and interact with all the components required to operate this robot through its CAN interface, which is utilized as a communication interface. The SCOUT MINI comes with two aviation male plugs to serve this purpose, as swown in Figure 6.12. The robot, in instance, uses conventional CAN2.0B communication at a bitrate of 500K. The robot will provide real-time feedback on the motor current, encoder, and temperature as well as its current movement status information and chassis status information via an external CAN bus interface. The robot's moving linear speed and rotational angular speed can also be regulated. The open-source software package (SDK) provides a C ++ interface to communicate with the mobile platform supplied by AgileX Robotics to transmit commands to the robot and obtain the most recent robot status. Specifically, a CAN-USB adapter is utilized to link the robot to the Nvidia Xavier Nx.

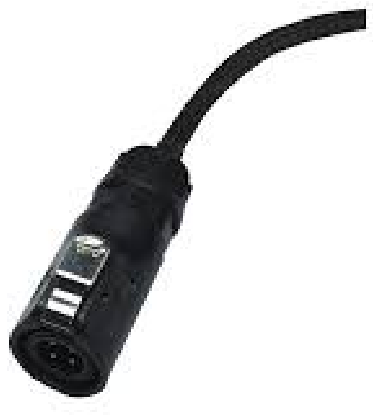


Figure 6.12: Aviation Male Plug for CAN cable connection [44]

6.5 Distributed Hardware System

On Ubuntu 18.04, a Linux kernel-based release, the system is built using Ros Melodic. On each board that is part of the AMR, it is downloaded and installed. It was particularly advantageous to have a distributed design because it spread the work over multiple boards and reduced the computational cost, given the large number of sensors. In order to build the distributed system, the Nvidia Xavier Nx board, which offers excellent results because to its high performance, is coupled to two LiDARs and the rover platform. Moreover, that board runs the primary software that enables autonomous exploration of the environment in addition to data collecting from the attached sensors. It was determined to transmit camera data using a Jetson Nano due to the limited number of USB connections, high computational cost, and power expense. It was possible to spread the computational load using the architecture shown in Figure 6.13.

6.6 Abstraction Layers of the System

The constructed system is made of different components and different abstraction levels, starting from the developed software and arriving to the actuation of the given commands. The Figure 6.14 shows all the subsystems that interacts during

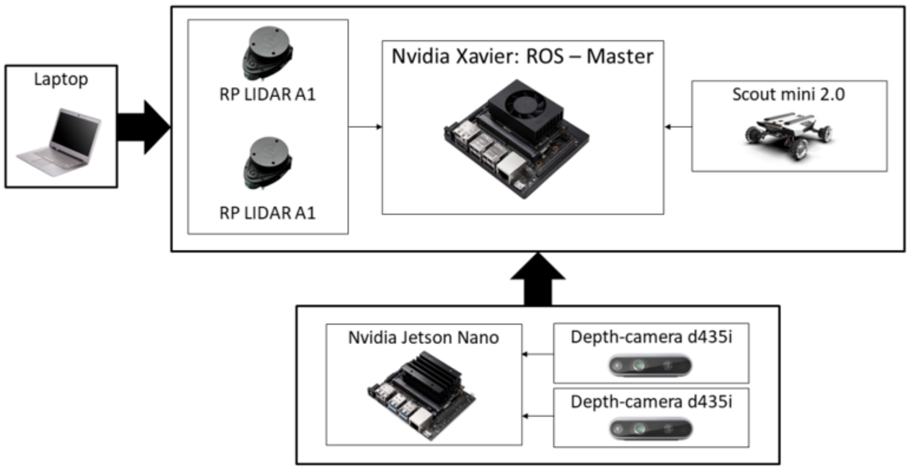


Figure 6.13: System architecture [45]

the exploration task of the rover. The higher level is represented by the tools used to monitor the navigation of the mobile robot, as RViz and Gazebo (in a simulation environment). The navigation modules include the components used for different applications, as SLAM or path planning. These modules represent the software part of the solution and the outcome velocities instructions coming from the navigation module are handled by the base controller which transforms these velocities into commands, and through the CAN bus the commands are given to the actuation part. Following this hierarchy, the software part and the hardware part cooperate to make the robot do the predefined actions.

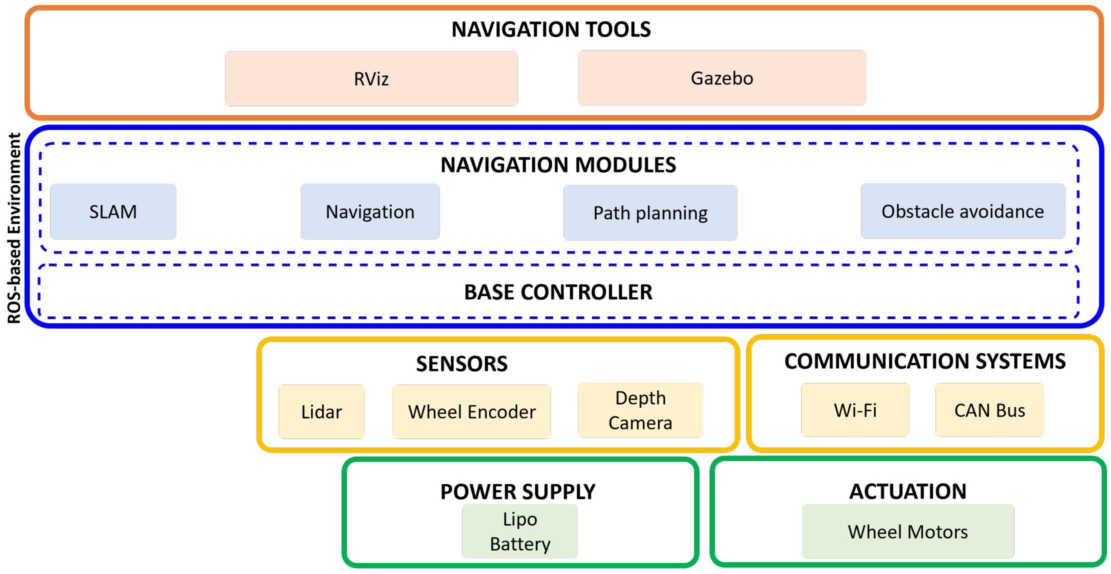


Figure 6.14: Hierarchical model of the system

Chapter 7

Active SLAM Experiments

The Active SLAM algorithm was tested in different scenarios with several configurations of both sensors and boards used. In particular three configurations have been tested, starting from a simple one and arriving to the final system. The hardware setup tested are the following:

- Experiment 1: One LiDAR and only the Jetson Xavier NX placed on the rover
- Experiment 2: All the sensors and boards located on a wood platform placed on the rover
- Experiment 3: All sensors and boards located in the FIXIT case

These three steps were necessary in order to address the problems in a modular and incremental fashion. Starting from a simpler system helped in solving software problems not encountered in the simulation phase, using a straightforward system which was easier to debug and to shut down in cases in which something went wrong. The second step was necessary in order to connect all the sensors and the boards available and test the final system performance. Two LiDARs and two cameras were used in collaboration with the two on-board computers to explore and map the environment. In this phase was also possible to easily manage the rover if errors and failures had shown up. The final system consists of the same hardware deployed in the second phase, but with the sensors placed in a slightly different height and with the FIXIT case that not easily allows to to be stopped in case of system errors. For this reason, the settings chosen in this configuration were tested to be reliable and structured in order to safely allow the exploration of the environment.

7.1 Network Setup

The on-board computers and the sensors used in this application communicates between them in the same network through the ROS principles. The system is composed of the Nvidia Xavier NX, which connects two LiDARs and is in charge of communicating with the Scout rover and runs the primary software, a remote laptop where RViz is running in order to monitor the behavior of the rover during the exploration phase, two cameras connected to the Jetson Nano that collects data and send point clouds to the Xavier NX through the use of topics. These information are then read by Xavier Nx, which process them and use these data to perform navigation. In Figure 7.1 is shown how the elements of the system are interconnected.

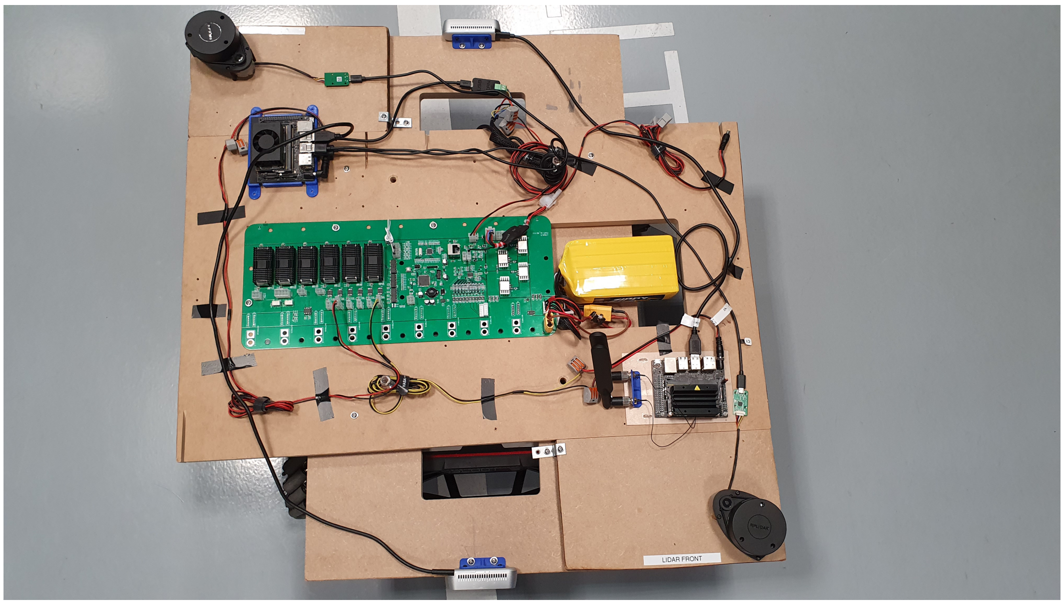


Figure 7.1: Hardware configuration of the system

A ROS-based system consists of several nodes running on different machines that communicate with each other by exploiting the distributed architecture characteristic of ROS. The communication between different boards is permitted only if the components belong to the same internet network. For this purpose, for each board or laptop that wants to receive or send to the overall system, is necessary to declare the IP address of the Master (unique in the system) and the IP address of itself. To do so, each board of the system needs to write the file */.bashrc* as follows:

- *export ROS_MASTER_URI = http : // < remote_PC_IP >: 11311*

- *export ROS_HOSTNAME =< current_PC_IP >*

More in detail, the *ROS_MASTER_URI* parameter indicates the IP address of the Master computer, which in this system is represented by the Jetson Xavier NX, while the *ROS_HOSTNAME* parameter represent the IP address of the considered board. So, once connected the hardware of the system, the sensors used need to be properly configured to get them ready for the exploration task. To start the two LiDARs is necessary to assign a serial port to both of them, and this can be done through the terminal of the Xavier (connected via SSH with the laptop) executing the following commands:

- *sudo chmod 777 /dev/ttyUSB0* for the front LiDAR
- *sudo chmod 777 /dev/ttyUSB1* for the back LiDAR

Instead, to launch the two cameras is necessary to run on the terminal of the Jetson Nano (connected via SSH with the laptop) the following launch file:

- *roslaunch realsense2_camera rs_camera.launch*

Once activated all the sensors of the platform, through the terminal of the Xavier the code for the autonomous exploration of the environment can be launched.

7.2 Active SLAM Experiment 1

The first phase in testing the Active SLAM solution for the AMR, after the simulation step, was to try a naive setup using only one LiDAR as sensor and only the Jetson Xavier NX for the computations, as shown in Figure 7.2. The reasons behind this choice lies on the fact that with a simpler configuration was easier to debug both software and hardware problems, and, in case of faults during the autonomous exploration was simpler to recover the mobile robot. Testing the effectiveness of the SLAM algorithm firstly in a less complex system helped in changing some configuration parameters, essentials in order to properly suit the SLAM solution around the considered AMR.

7.2.1 Restricted map of the environment

The experiments were taken inside the CIM4.0 laboratory, a dynamic environment with obstacles at different heights, which was particularly challenging for the AMR equipped with only one LiDAR. For this reason, at the beginning of this phase of tests, an hallway was created in the CIM4.0 environment using boxes, in order to constrain the rover to move safely. The map created, shown in Figure 7.3, depict the obtained result.

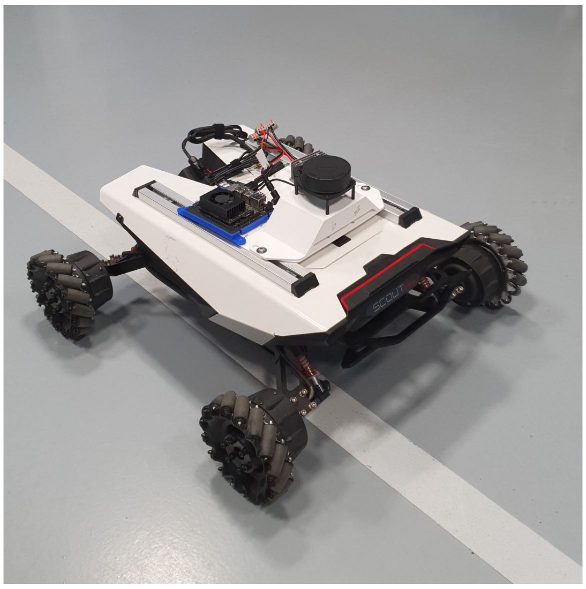


Figure 7.2: The AMR equipped with one LiDAR and the Jetson Xavier

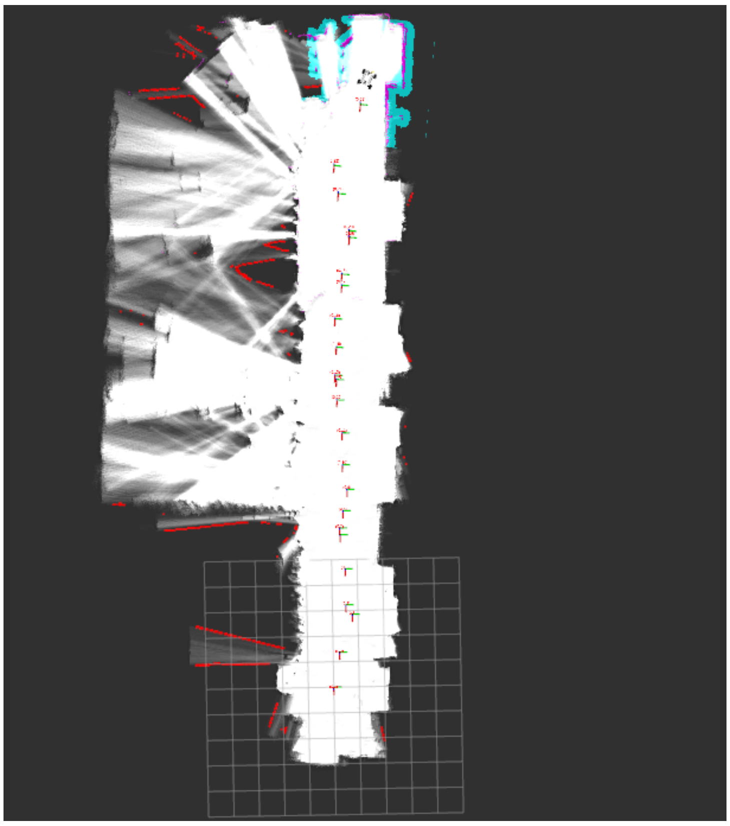


Figure 7.3: Map created exploring an hallway in the CIM4.0 laboratory

The rover, starting from the opposite part of the laboratory, crosses the constructed hallway and arrives to the final position exploring the space. The AMR finds frontier points (red points in Figure 7.3), transform them into navigation points, and navigate to them, until the environment is fully explored. The environment completely explored, in this case, is the hallway.

7.2.2 Complete map of the environment

In the second part of the tests, the rover is no longer constrained to move in an hallway, but it can explore the entire environment, even if the more challenging obstacles are covered by boxes. The result of this test, is shown in the Figure 7.4. The AMR is capable of moving and the exploring the area creating the map of the laboratory. The undiscovered frontier points are still present in the map, but they are not accessible by the mobile platform because of its dimensions.

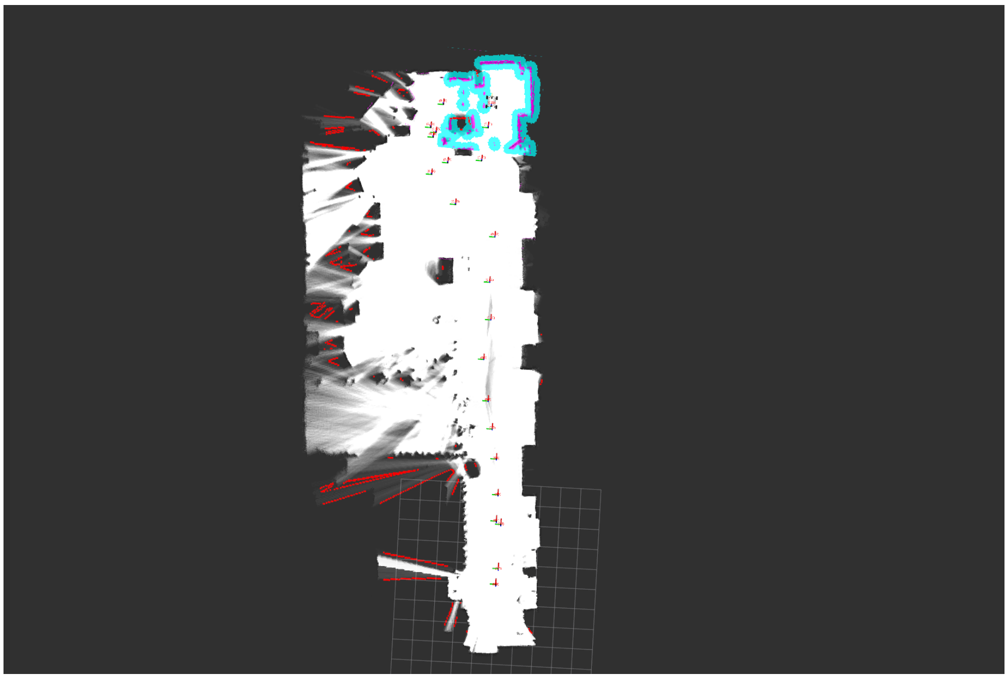


Figure 7.4: Complete map of the CIM4.0 laboratory

7.2.3 Errors during the exploration

During these tests a couple of errors have shown up related to the reachability of the frontiers. Indeed, the Active SLAM algorithm detected even the frontiers that were not reachable by the AMR. This error led the rover to consider these points as feasible navigation targets and, as a consequence, calculate paths to them. The rover, equipped with obstacle avoidance algorithms, did not collide with obstacles but instead recalculated a new path to the destination, entering in a loop condition with no escape. The Figure 7.5 shows the rover which is trying to reach a frontier point even if is not reachable. These drawbacks were solved in the intermediate configuration of the system.

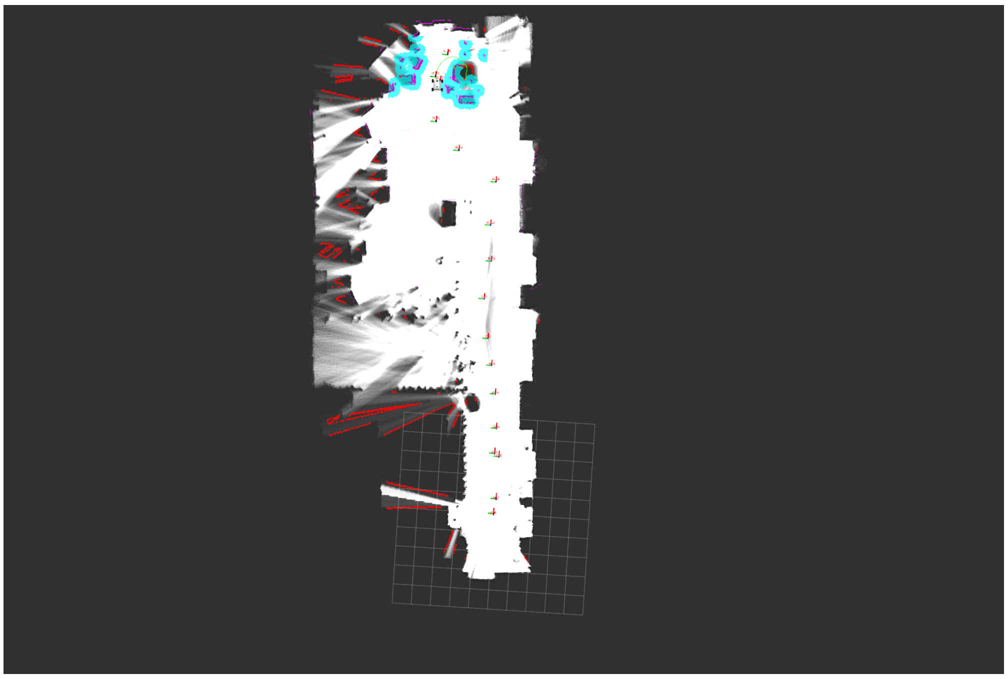


Figure 7.5: In green is shown the path calculated in order to try to reach the unapproachable frontier point

7.3 Active SLAM Experiment 2

During the second phase of tests, the Active SLAM algorithm was tested connecting all the sensors and the boards available, in order to emulate the configuration of the final system (Figure 7.6). This phase was necessary because has allowed to easily manage each component and to readily recover it if errors occurred.

7.3.1 Reachability of Frontiers

During the first phase of tests a couple of errors have shown up relating to the reachability of the frontiers. Indeed, in order to solve this problem an inflation operation is performed on the submaps. This means that the obstacles detected

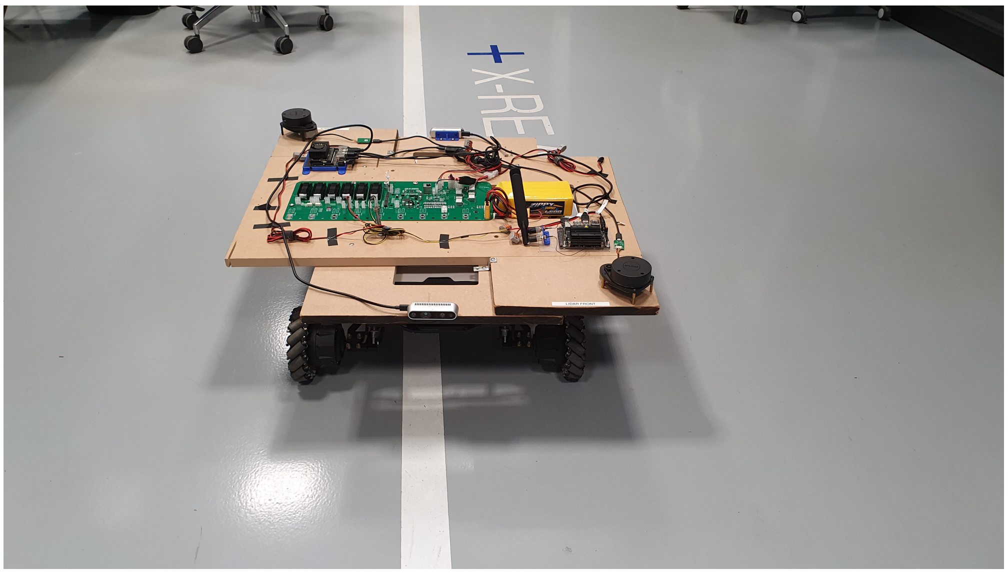


Figure 7.6: The AMR is equipped with all the available hardware placed on a wood platform

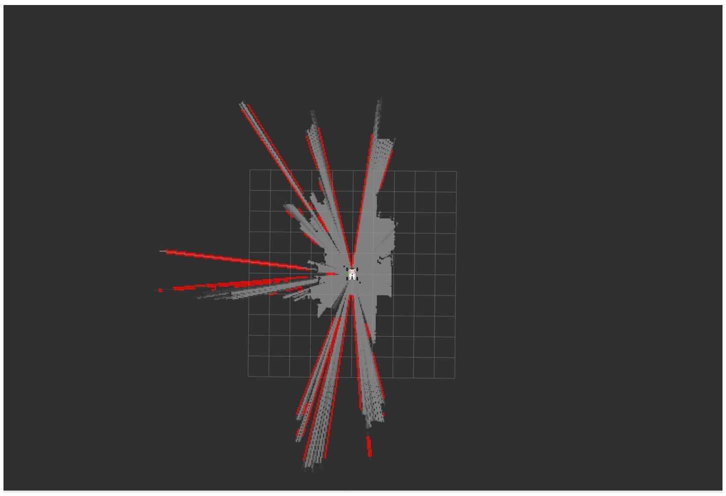
by the AMR are dilated and therefore prevent the rover to see frontier points located in positions beyond these obstacles, which are not reachable because of the dimension of the mobile robot and shapes of the environment. After this process, the AMR detects only reachable frontier points, and plan only valid paths directed to these points, as shown in Figure 7.7. In this way, the loop problems encountered in the first phase of tests are overcome.

7.3.2 Experimental results

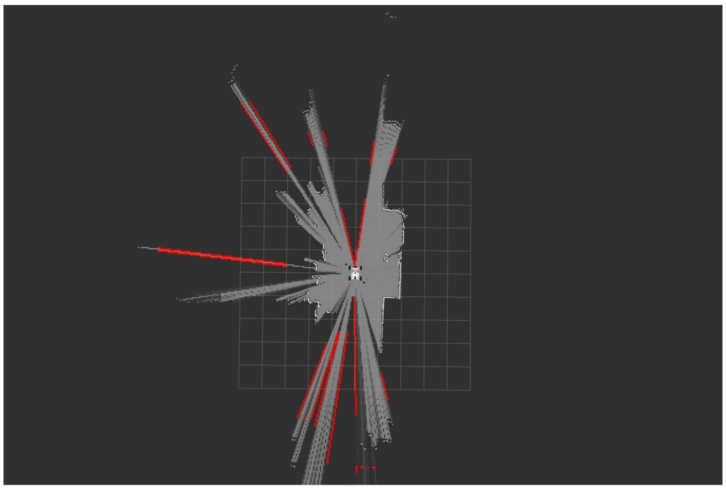
After solving the problem related to the reachability of the frontiers, the autonomous exploration is tested. The AMR starts exploring the environment identifying the frontier points and heading to them (Figure 7.8). In this way, the map is created and the locations not accessible to the rover are not discovered. The complete map is shown in Figure 7.9.

7.4 Active SLAM Experiment 3

The third phase of testing consisted of the deployment of the same hardware configuration used during the second phase of experimentation, but this time inserted inside the FIXIT case. The only difference in the system was the position of the sensors with respect to the z axis, which was slightly higher both for the LiDARs and the cameras. The process of autonomous exploration is shown in



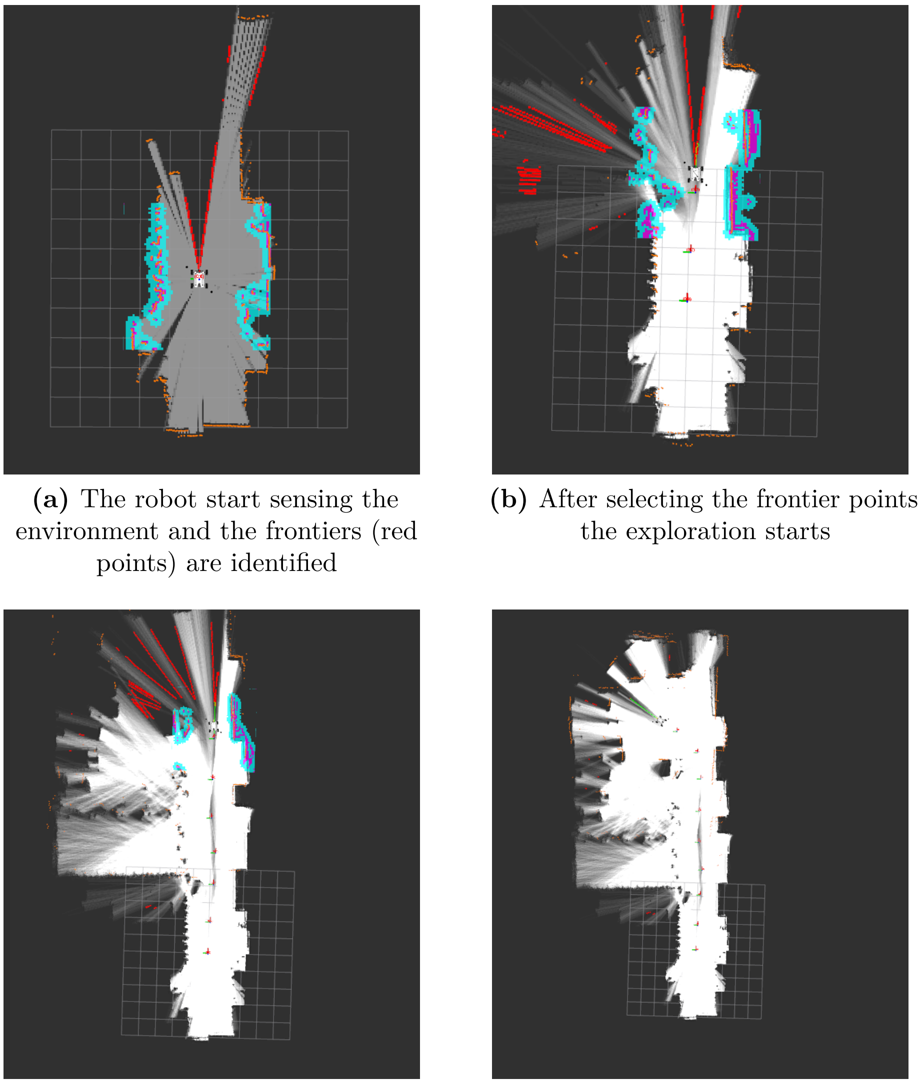
(a) The rover also perceive frontiers that are not reachable



(b) The rover perceive less frontiers, only those that it can reach

Figure 7.7: Perception of the frontiers before and after the inflation process

Figure 7.10, where the AMR moves inside the CIM4.0 laboratory and create at the same time the map of the environment. The map created during this process is shown in Figure 7.11. The result obtained was very similar compared with the one of the second phase of experimentation. The generated map of the CIM4.0 laboratory was accurate, considering the dynamic nature of the environment. The parts that were not completely uncovered were actually not accessible to the rover, given the presence of obstacles and considering the size of the structure.


(a) The robot start sensing the environment and the frontiers (red points) are identified

(b) After selecting the frontier points the exploration starts

(c) The exploration continues

(d) The exploration is finished and the map is complete

Figure 7.8: The process of autonomous exploration of the CIM4.0 laboratory

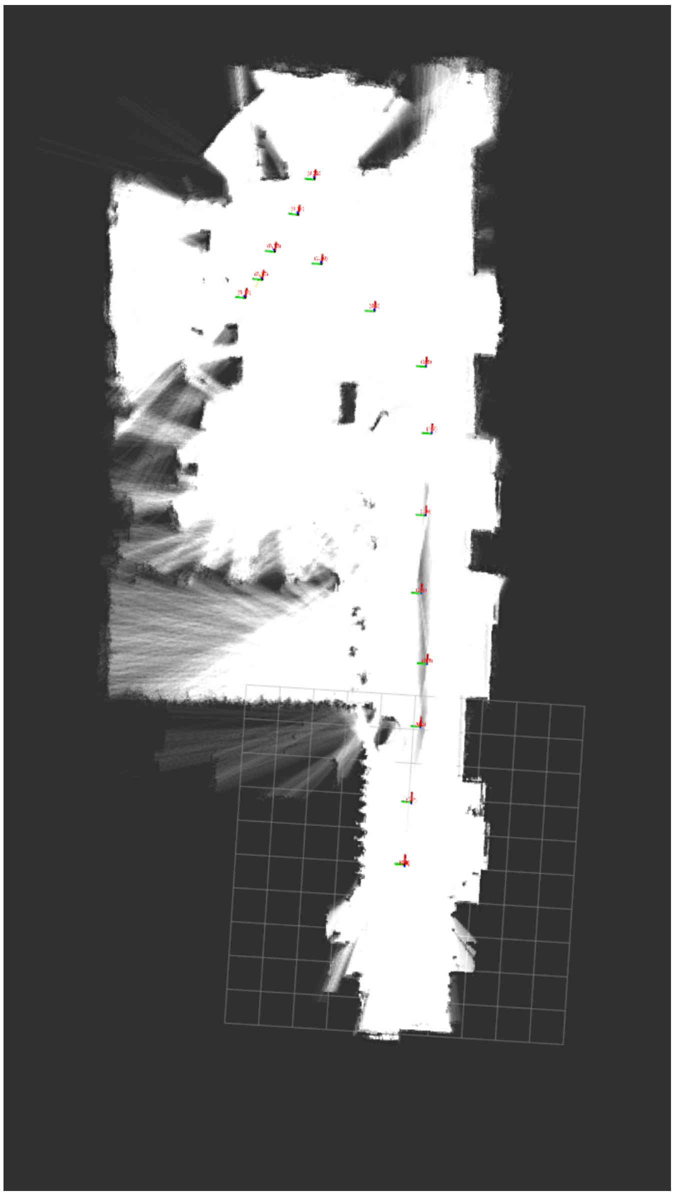
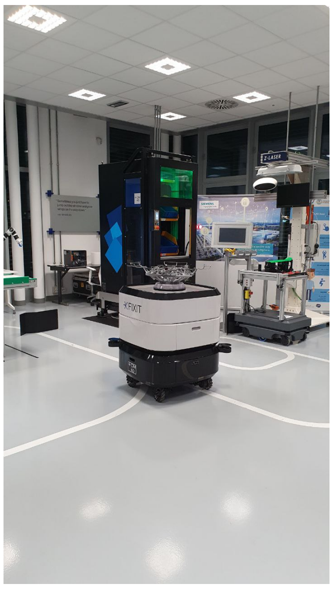


Figure 7.9: The map created at the end of the exploration process

(a) The robot start exploring the environment

(b) The exploration continues

(c) Arrived at the end of the hallway, the AMR decides to move towards the centre of the laboratory, to acquire more information about the environment



(d) Once completed the map, the rover terminate the exploration. At this point, by pointing to a spot on the map, the AMR is capable of navigate the uncovered area

Figure 7.10: Sequence of steps taken by the AMR during autonomous exploration of the environment

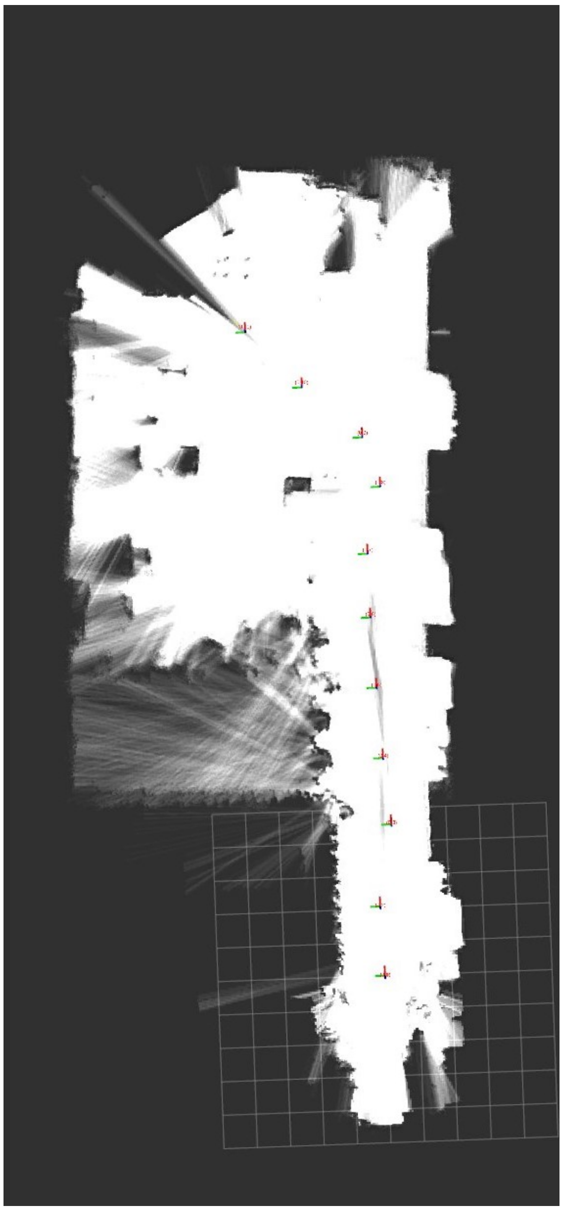


Figure 7.11: Map obtained by the AMR at the end of the autonomous exploration process

Chapter 8

Conclusions

The main objective of this thesis was to develop and test an Active SLAM algorithm to allow the AMR of the FIXIT project to autonomously explore and navigate in unknown environments. After an extensive analysis of the state of the art of SLAM algorithms in exploration tasks, an Active SLAM solution was implemented and tested firstly in a simulation environment, and then experimentally.

The designed system proves to be extremely reliable and effective. With sensors deployed on the mobile platform, the system can autonomously and safely explore a dynamic environment filled with obstacles and objects of different shapes and sizes at each level.

In particular, thanks to the Active SLAM algorithm developed, the AMR is proven to be:

- Autonomous: the mobile robot is able to autonomously explore and navigate in unknown spaces, without the human intervention.
- Safe: the rover is provided of obstacle avoidance algorithms, then, during the exploration is able to recognize obstacles, avoid them and in the end reach the target point.
- Adaptable: the system developed is flexible and scalable to different scenarios. Apart from environments in which the rover is physically constrained to move, the AMR can change workspace without changing settings.

It can be employed to explore a dangerous environment because it has the ability to move independently by gathering information from its surroundings through sensors. Moreover, the cameras allow the remote operator to view the area that he is unable to reach, for instance, because of the spread of dangerous gases.

8.1 Limits and Future works

The designed autonomous exploration system achieves the expected objectives with good results, although there are several areas that might be improved.

Even if the available sensors placed on the AMR can guarantee a safely navigation, not all the obstacles can be seen. Moreover, lower obstacles in proximity of the mobile robot may not be detected.

Another limitation of the system is given by the type of the sensor used. The deployed sensors, even if they perform well in indoor scenarios, are not suitable for outdoor spaces, due to the presence of sunlight reflection which can alter the LiDARs performances.

These limitations can be overcome by introducing 3D LiDARs adaptable for both indoor and outdoor scenarios. With this type of sensor the system becomes more stable and flexible for all type of situations. Although currently AMR and UAV systems do not communicate with each other, in the near future an implementation which permits the two systems to cooperate during the navigation objectives could be efficient for different purposes, as maintenance and collaboration tasks in an industrial environment.

Bibliography

[1] Hamid Taheri and Zhao Chun Xia. «SLAM; definition and evolution». In: *Engineering Applications of Artificial Intelligence* 97 (2021), p. 104032 (cit. on pp. 5, 7, 8).

[2] Xu Lei, Bin Feng, Guiping Wang, Weiyu Liu, and Yalin Yang. «A novel fastslam framework based on 2d lidar for autonomous mobile robot». In: *Electronics* 9.4 (2020), p. 695 (cit. on p. 6).

[3] Shoudong Huang and Gamini Dissanayake. «Convergence and consistency analysis for extended Kalman filter based SLAM». In: *IEEE Transactions on robotics* 23.5 (2007), pp. 1036–1049 (cit. on p. 7).

[4] Giorgio Grisetti, Gian Diego Tipaldi, Cyrill Stachniss, Wolfram Burgard, and Daniele Nardi. «Fast and accurate SLAM with Rao–Blackwellized particle filters». In: *Robotics and Autonomous Systems* 55.1 (2007), pp. 30–38 (cit. on p. 8).

[5] Giorgio Grisetti, Rainer Kümmerle, Cyrill Stachniss, and Wolfram Burgard. «A tutorial on graph-based SLAM». In: *IEEE Intelligent Transportation Systems Magazine* 2.4 (2010), pp. 31–43 (cit. on p. 8).

[6] María L Rodríguez-Arévalo, José Neira, and José A Castellanos. «On the importance of uncertainty representation in active SLAM». In: *IEEE Transactions on Robotics* 34.3 (2018), pp. 829–834 (cit. on p. 8).

[7] Like Cao, Jie Ling, and Xiaohui Xiao. «Study on the influence of image noise on monocular feature-based visual SLAM based on FFDNet». In: *Sensors* 20.17 (2020), p. 4922 (cit. on p. 8).

[8] Zhilin Xu, Jincheng Yu, Chao Yu, Hao Shen, Yu Wang, and Huazhong Yang. «CNN-based Feature-point Extraction for Real-time Visual SLAM on Embedded FPGA». In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2020, pp. 33–37 (cit. on p. 8).

[9] Iker Lluvia, Elena Lazkano, and Ander Ansuategi. «Active mapping and robot exploration: A survey». In: *Sensors* 21.7 (2021), p. 2445 (cit. on pp. 8, 10).

[10] Josep Aulinas, Yvan Petillot, Joaquim Salvi, and Xavier Lladó. «The SLAM problem: a survey». In: *Artificial Intelligence Research and Development* (2008), pp. 363–371 (cit. on p. 9).

[11] Samyeul Noh, Jiyoung Park, and Junhee Park. «Autonomous Mobile Robot Navigation in Indoor Environments: Mapping, Localization, and Planning». In: *2020 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE. 2020, pp. 908–913 (cit. on pp. 9, 10).

[12] Yong Li and Changxing Shi. «Localization and navigation for indoor mobile robot based on ROS». In: *2018 Chinese automation congress (CAC)*. IEEE. 2018, pp. 1135–1139 (cit. on p. 9).

[13] CCE Chewu and V Manoj Kumar. «Autonomous navigation of a mobile robot in dynamic indoor environments using SLAM and reinforcement learning». In: *IOP Conference Series: Materials Science and Engineering*. Vol. 402. 1. IOP Publishing. 2018, p. 012022 (cit. on p. 10).

[14] Hartmut Surmann, Christian Jestel, Robin Marchel, Franziska Musberg, Houssem Elhadj, and Mahbube Ardani. «Deep reinforcement learning for real autonomous mobile robot navigation in indoor environments». In: *arXiv preprint arXiv:2005.13857* (2020) (cit. on p. 10).

[15] Michal Mihálik, Branislav Malobickỳ, Peter Peniak, and Peter Vestenickỳ. «The New Method of Active SLAM for Mapping Using LiDAR». In: *Electronics* 11.7 (2022), p. 1082 (cit. on p. 12).

[16] Chaoqun Wang, Lili Meng, Sizhen She, Ian M Mitchell, Teng Li, Frederick Tung, Weiwei Wan, Max Q-H Meng, and Clarence W de Silva. «Autonomous mobile robot navigation in uneven and unstructured indoor environments». In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2017, pp. 109–116 (cit. on p. 13).

[17] ChengYi Zhang, ShuWen Dang, Yong Chen, and ChenFei Ling. «A Survey of Motion Planning Algorithms Based on Fast Searching Random Tree». In: *2021 the 7th International Conference on Communication and Information Processing (ICCIP)*. ICCIP 2021. Beijing, China, 2021, pp. 4–8. ISBN: 9781450385190 (cit. on p. 13).

[18] Xudong Sun, Fuchun Sun, Bin Wang, Jianqin Yin, Xiaolin Sheng, and Qinghua Xiao. «Robotic autonomous exploration SLAM using combination of Kinect and laser scanner». In: *2017 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*. IEEE. 2017, pp. 632–637 (cit. on p. 14).

[19] Zehui Meng, Hao Sun, Hailong Qin, Ziyue Chen, Cihang Zhou, and Marcelo H Ang. «Intelligent robotic system for autonomous exploration and active SLAM in unknown environments». In: *2017 IEEE/SICE International Symposium on System Integration (SII)*. IEEE. 2017, pp. 651–656 (cit. on p. 14).

[20] Ming Hsiao, Joshua G Mangelson, Sudharshan Suresh, Christian Debrunner, and Michael Kaess. «Aras: Ambiguity-aware robust active slam based on multi-hypothesis state and map estimations». In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2020, pp. 5037–5044 (cit. on p. 15).

[21] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. «ROS: an open-source Robot Operating System». In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5 (cit. on p. 17).

[22] Kenta Takaya, Toshinori Asai, Valeri Kroumov, and Florentin Smarandache. «Simulation environment for mobile robots testing using ROS and Gazebo». In: *2016 20th International Conference on System Theory, Control and Computing (ICSTCC)*. IEEE. 2016, pp. 96–101 (cit. on p. 21).

[23] Beipeng Mu, Matthew Giamou, Liam Paull, Ali-akbar Agha-mohammadi, John Leonard, and Jonathan How. «Information-based active SLAM via topological feature graphs». In: *2016 IEEE 55th Conference on decision and control (Cdc)*. IEEE. 2016, pp. 5583–5590 (cit. on pp. 29, 30).

[24] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. «Real-time loop closure in 2D LIDAR SLAM». In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 1271–1278. DOI: `10.1109/ICRA.2016.7487258` (cit. on pp. 31–33, 43–45).

[25] Zezhou Sun, Banghe Wu, Cheng-Zhong Xu, Sanjay E. Sarma, Jian Yang, and Hui Kong. «Frontier Detection and Reachability Analysis for Efficient 2D Graph-SLAM Based Active Exploration». In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020, pp. 2051–2058. DOI: `10.1109/IROS45743.2020.9341735` (cit. on pp. 34–36, 44, 46).

[26] *Clustering Algorithms - Mean Shift Algorithm*. `https://www.tutorialspoint.com/machine_learning_with_python/clustering_algorithms_mean_shift_algorithm.htm`. Accessed: 2010-09-30 (cit. on pp. 35, 36).

[27] *Global Path planning*. `http://wiki.ros.org/global_planner?distro=noetic`. Accessed: 2010-09-30 (cit. on pp. 37, 38).

[28] Peter E Hart, Nils J Nilsson, and Bertram Raphael. «A formal basis for the heuristic determination of minimum cost paths». In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107 (cit. on p. 37).

[29] Masoud Nosrati, Ronak Karimi, and Hojat Allah Hasanvand. «Investigation of the*(star) search algorithms: Characteristics, methods and approaches». In: *World Applied Programming* 2.4 (2012), pp. 251–256 (cit. on p. 38).

[30] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. «The dynamic window approach to collision avoidance». In: *IEEE Robotics & Automation Magazine* 4.1 (1997), pp. 23–33 (cit. on p. 39).

[31] *Local Path planning.* `http://wiki.ros.org/base_local_planner?distro=noetic` (cit. on p. 40).

[32] Kristin Glass, Richard Colbaugh, David Lim, and Homayoun Seraji. «Real-time collision avoidance for redundant manipulators». In: *IEEE transactions on robotics and automation* 11.3 (1995), pp. 448–457 (cit. on p. 40).

[33] Agnieszka Lazarowska. «A discrete artificial potential field for ship trajectory planning». In: *The Journal of Navigation* 73.1 (2020), pp. 233–251 (cit. on p. 41).

[34] Hugh Durrant-Whyte and Tim Bailey. «Simultaneous localization and mapping: part I». In: *IEEE robotics & automation magazine* 13.2 (2006), pp. 99–110 (cit. on p. 42).

[35] BAYU KANUGRAHAN LUKNANTO. «A review of 2D SLAM algorithms on ROS». In: (2020) (cit. on p. 43).

[36] Zhang Xuexi, Lu Guokun, Fu Genping, Xu Dongliang, and Liang Shiliu. «SLAM algorithm analysis of mobile robot based on lidar». In: *2019 Chinese Control Conference (CCC)*. IEEE. 2019, pp. 4739–4745 (cit. on p. 43).

[37] *Navigation Stack.* `http://wiki.ros.org/navigation/Tutorials/RobotSetup`. Accessed: 2010-09-30 (cit. on p. 48).

[38] David V. Lu, Dave Hershberger, and William D. Smart. «Layered costmaps for context-sensitive navigation». In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2014, pp. 709–715. DOI: `10.1109/IROS.2014.6942636` (cit. on pp. 51, 52).

[39] Darko Trivun, Edin Šalaka, Dinko Osmanković, Jasmin Velagić, and Nedim Osmić. «Active SLAM-based algorithm for autonomous exploration with mobile robot». In: *2015 IEEE International Conference on Industrial Technology (ICIT)*. 2015, pp. 74–79. DOI: `10.1109/ICIT.2015.7125079` (cit. on p. 56).

[40] *RPLIDAR A1.* `https://www.slamtec.com/en/Lidar/A1` (cit. on p. 61).

[41] *Intel Realsense Camera.* `https://www.intelrealsense.com/depth-camera-d435i/` (cit. on pp. 62, 63).

[42] *Jetson Xavier NX.* `https://www.nvidia.com/it-it/autonomous-machines/embedded-systems/jetson-xavier-nx/` (cit. on p. 64).

[43] *Jetson Nano*. `https://developer.nvidia.com/embedded/jetson-nano-developer-kit` (cit. on p. 65).

[44] *Scout Mini*. `https://indrorobotics.ca/wp-content/uploads/2021/04/SCOUT-MINI-User-Manual-3.0-.pdf` (cit. on pp. 67, 68).

[45] Orlando TOVAR ORDOÑEZ and Stefano SANTORO. «Design and implementation of a Sensory System for an Autonomous Mobile Robot in a Connected Industrial Environment». In: (2021) (cit. on p. 69).