ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA

SCHOOL OF ENGINEERING

DEPARTMENT ELECTRICAL, ELECTRONIC AND INFORMATION ENGINEERING

"GUGLIELMO MARCONI" (DEI)

MASTER'S DEGREE IN AUTOMATION ENGINEERING

GRADUATE THESIS

in

Autonomous and Mobile Robotics

DEVELOPMENT AND TESTING OF DOCKING FUNCTIONS IN INDUSTRIAL SETTINGS FOR AN AUTONOMOUS MOBILE ROBOT BASED ON ROS2

CANDIDATE:

Filippo Guarda

SUPERVISOR:
Prof. **Gianluca Palli**

ASSISTANT SUPERVISORS:
PhD. **Emilio Tirelli**
Ing. **Gabriele Fabbri**
Ing. **Federico Orazio**

Academic Year 2022/2023
Session II

Bologna, October 14th, 2023

During this eight-month internship at GD s.p.a. I had the possibility to be inserted in an engaging and multifaceted working environment. Special thanks go to all the members of the team I had the opportunity, and will gladly continue, to work with: Emilio Tirelli, Gabriele Fabbri and Federico Orazio. Thank you for the continuous teachings and uninterrupted encouragement in everything that concerned work, workplace, and beyond. I would like to thank Professor Gianluca Palli as well, for the above and beyond availability and support granted throughout the realization of this dissertation, especially in the most critical moments.

I would also like to thank my girlfriend, family, and friends: Anna Giorietto, Anna Donati, Caterina Guarda, Emilio Guarda, Francesco Guarda, Annarita Angelini, Gaia Paruto, Adileo Barone, Enrico Caracciolo. Thanks to all of you for sticking with me in a very challenging period of my life.

Filippo Guarda

ABSTRACT

This dissertation is the result of a six-months internship at G.D S.p.A. for the preparation of the thesis project.

The final goal is to develop algorithms on the ROS2 framework that could be used to control an Autonomous Mobile Robot during the operations of detection and approach of a docking station with high precision, needed to operate a recharge of the AMR itself or some operation on the host machines.

The automation of these operations ensures a substantial increase in safety and productivity within a warehouse or host machine lines since it permits to the AMR to work without requiring an operator for longer time or even to substitute the operator itself.

The presented method uses both lidars and an onboard camera. The trajectory from the starting position to the approximate area of the docking station is computed using data obtained from the three lidars around the AMR body.

The final approach is implemented by detecting an ARUCO code positioned on the dock assembly through a camera.

A sequence of intermediate positions is defined according to the pose estimations, and then reached with a mix of standard navigation and a proportional position control in the very last part of the movement trajectory.

The precision of the docking position turned out to have less than one centimeter error around the desired target, the orientation error is a fraction of a degree. The docking times vary based on how far the AMR is from the docking station, but the last phase of the procedure is always completed in around seventeen seconds.

The solution is implementable and will be evaluated on the real platform in the coming months.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF ACRONYMS

AGV	Automatic Guided Vehicle
AMCL	Adaptive Monte Carlo Localization
AMR	Autonomous Mobile Robot
ARUCO	Augmented Reality University of Cordoba
DDS	Data Distributed Services
EKF	Extended Kalman Filter
IMU	Inertial Measurement Unit
LIDAR	Laser Imaging Detection and Ranging
NAV2	NAVigation 2
ROS2	Robot Operative System 2
RPY	Roll Pitch and Yaw
SLAM	Simultaneous Localization and Mapping
YAML	YAML Ain’t Markup Language
XML	eXtensible Markup Language

1 INTRODUCTION

An Autonomous Mobile Robot, also abbreviated into AMR, is defined, as the name suggests, as a mobile robot that can navigate the environment, known or unknown, without the need for external control. AMRs are the direct evolution of Automated Guided Vehicles, AGV for short. While AGVs need a predetermined path to follow in order to function properly, AMRs do not need physical tracks nor software-defined travel paths. Henceforth, AGVs are simpler to build, but are less flexible, require greater supporting infrastructure and have a higher upfront cost.

To navigate the environment an Autonomous Mobile Robot uses an array of sensors such as lidars, IMUs and 3D cameras. Then, a series of algorithms (that can vary from simple state machines to behavior trees, up to machine learning and deep neural networks), utilize the data provided by the sensors to localize the robot, compute travel paths, and react to unexpected obstacles or faults.

This thesis revolves around a particular AMR, the industrial Mobile Robot, which differs from household appliances (like Roombas) both in sheer size and for the compliance with stricter safety standards dictated by the European regulation of heavy equipment.

This AMR will be mostly used in two distinct roles. The first role is to move pallets through a warehouse in a forklift configuration. The use of Autonomous Vehicles for moving loads in a controlled environment has been expanded over the years in such a way that now an Autonomous Mobile Robot can also manage semi-controlled environments, such as a warehouse where workers are present and not constrained to move in set areas. This configuration may use a neural network to detect pallets through an image feed and depth perception to position the fork under the pallet.

The second role is to execute ancillary tasks on a different machine using specialized tooling. While executing these operations, the robotic base will have stricter positional requirements dictated by the close interaction of previously mentioned tools with sensitive machinery. In this configuration, a 3D camera will also be equipped on the robot body.

Both configurations are dictated by a trend in the industry where robots are used to substitute human operators in dangerous tasks such as transport of heavy loads, or tasks where errors in routine maintenance could present a liability to the production plant.

The main objective of this project is to develop the localization and navigation functions of the previously mentioned AMR with the scope of docking to a station where high precision positioning is needed, like a recharging station, which was implemented using ROS2. ROS2, or "Robot Operating System" is a framework which allows for the use of an ample set of software libraries and tools that are open-source and extendable by users and it is commonly referred as the state of the art for AMR development.

The first part of the project is focused on fusing the signals from three lidars positioned around the robot body in order to create a single virtual sensor that can be utilized for simultaneous location and mapping (SLAM for short); and on configuration of the libraries necessary to robustly navigate a simulated (at the time) industrial environment.

The aim of this part is to use the safety laser scanner, compulsory present to guarantee the safety of the operators present in the area, and not to mount another laser scanner just to navigate.

After perfecting the navigation code, the second part of the project is set on developing a method for defining the positional relation between the robot and the docking station. The resulting function is based on the use of Aruco markers attached to the docking station. Aruco markers are univocal symbols easily detected by a computer vision algorithm, which, knowing the parameters of the optics and the size of the marker, can compute the relative position of the Aruco code with respect to the optical center of the camera.

The result of this project is a function that navigates the AMR to a set point where an Aruco code is expected to be in the cameras field of view. The function then generates a quaternion coordinate between the camera and the marker, which is extended on one side from camera to robot base, and on the other from Aruco marker to the power connection. The robot moves one meter in front of the dock through standard navigation, then completes the approach by a PD positional control.

Testing is implemented using the Gazebo, simulation software shipped in conjunction with ROS2. An approximative industrial environment was modeled inside Gazebo, then both the 3D models of a pallet (from an open-source collection) and a docking station with Aruco code attached were imported into the scene. Finally, the robot physical model complete with inertias was created through a ROS2 configuration file.

The project has been developed as a result of an internship in GD s.p.a. in Bologna, a subsidiary and founder company of the COESIA group. GD has the need to constantly move inventory to support the construction of its packaging machines and is expected to start using Autonomous Robots to support its more advanced production lines in the future.

2 STATE OF THE ART

In this chapter, we will define the differences between Automated Guided Vehicles and Autonomous Mobile Robots. After having sketched a generalized picture of the two platforms, we will explore the state of the art for applications of AGVs in the logistic field, with an emphasis on methods used by market leaders. Finally, we will briefly touch on what smart applications are and their potential in an industrial setting.

2.1 AGV vs AMR

The difference between an Automated Guided Vehicle and an Autonomous Mobile Robot lies in, as the name suggests, the ability to move freely through space.

AGVs often are used to move objects along a factory line or inside a controlled warehouse. This can be done either by towing a cart, in which case a way of connecting the moving base with the cart is needed, or by loading the object on top of the machine, in a bed that can be either fixed or equipped with a conveyor belt.

On the other hand, Autonomous Mobile Robots do not need specific paths to be set, but instead can compute their own path after being given a reachable goal and an abstraction of the surrounding environment. An AMR will be able to move around obstacles that were not present at the time of the generation of its simulated environment, affording it additional flexibility and fault tolerance, while an AGV will have to stop and will not be able to compute alternative routes in the case where its set path becomes obstructed.

2.2 AGV guidance systems

While we will discuss AMR navigation in detail in the next chapter, we think it is important to also illustrate the different guidance systems used by AGVs.

Wired Track: a wire is run a couple of centimeters under the factory floor and used to transmit a radio signal. The AGV detects the radio signal strength through a sensor mounted on its bottom, and then uses it to regulate steering, effectively following along just as it would with a steel track. [1]

Guide Tape: the tape can be of two kinds, magnetic or colored. The AGV is then fitted with either a magnetic or optical sensor and follows the line by counter-steering as the magnetic strength decreases or if the optical sensor loses line of sight with the colored tape. Tape has the advantage over conductive wire of being easy to move and reapply but is more affected by dirt and wear. [1] [2]

Laser Target Navigation: in this case the path to follow is software-defined and the AGV locates itself in each layout by detecting reflective tape by a combination of rotating laser emitters and sensors. The sensor often only detects the angle of the markers, but sometimes it can also identify their distance. This allows the controller software to triangulate the position of the AGV by comparing it with the marker layout set in the software, and to steer it in order to keep it on the set path. [3]

Laser emitters can be either pulsed or modulated. For maximum precision, pulsed emitters must interpolate the readings of reflection intensity to precisely compute the position of a marker's center.

Natural feature (Natural Targeting) navigation: navigation that does not need the retrofitting of the workspace. It is based on the use of range finding sensors, such as LIDARS, and Inertial Measurements Units, used to locate the AGV by comparing detected environmental features with the map defined in the software layout through a Monte-Carlo/ Markov algorithm. Once the absolute position of the machine is obtained, the AGV is steered along software set paths just as in Laser Target Navigation. [3]

Vision Guidance: vision guided AGVs operate by using cameras to record features along the route, and then by replaying the route by using recorded features for navigation. This approach is made possible using an Evidence Grid based on probabilistic volumetric sensing. The sensing equipment used by vision guided AGVs consists in 3D stereo cameras, which, together with image information, also provide a volumetric and depth layer. [4]

2.3 Use of AGVs in logistics

AGVs have seen extensive use both in the logistics and industrial field in the last four decades, they have been defined as "battery-powered driverless vehicles, centrally computer-controlled and independently addressable, that are used for moving jobs between workstations on a factory floor" [5]. Many more definitions exist, but all revolve around an AGV being a vehicle that moves materials in 2D space without a human operator. During years of development, AGVs had time to diversify into a swathe of different models, each specialized for a different type of load. In this section we will illustrate four different AGVs: one specialized in moving pallets, one which is designed to move containers, one for roller carts and the last one specialized in towing.

2.3.1 Underride AGV

An underride AGV positions itself under a cart or material wagon and lifts it slightly, detaching the container from the floor. Alternatively, it locks the container through an attachment point and tows it to the destination. Naturally the second option is possible only in the case of a wheeled cart. This configuration presents some advantages both in dimensions and in maneuverability as the occupied volume depends almost exclusively on the dimensions of the container rather than the controlled platform. [6]

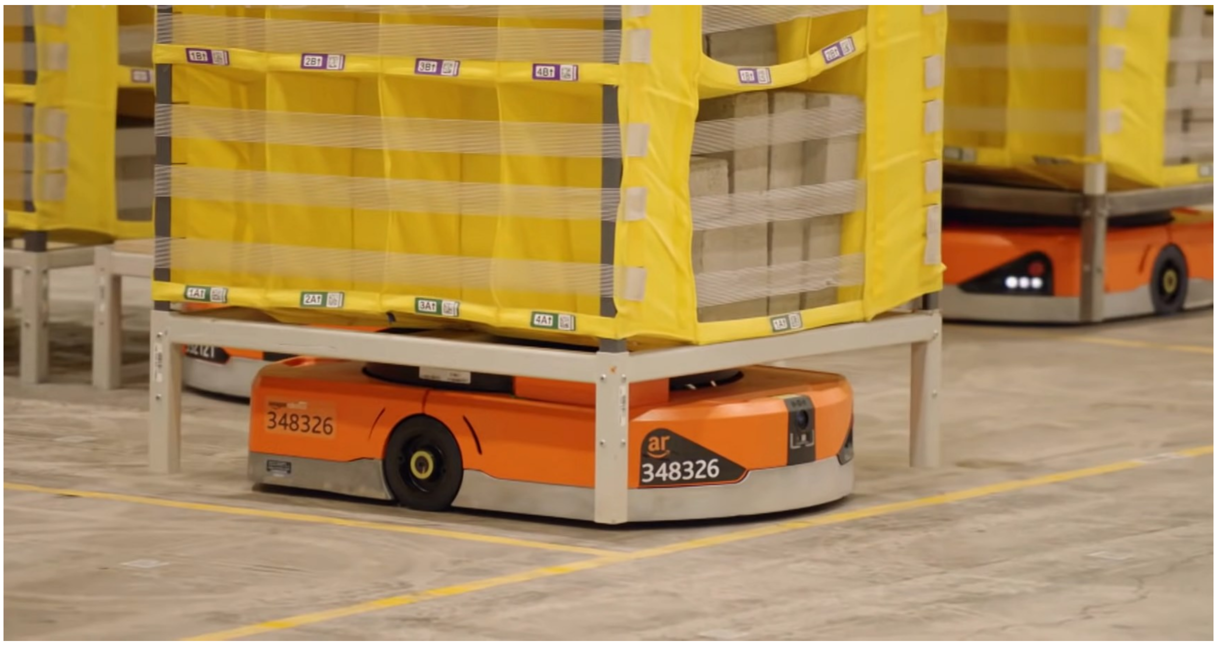


Figure 1: An underride AGV used in Amazon warehouses.

2.3.2 Piggyback AGV

Piggyback AGVs can load and unload without the need for maneuvering under the target load. Instead, they need the load to be at a predetermined height from which it will be transferred to the AGVs bed, which can be implemented either with rollers, for larger platforms that can carry pallets, or with a conveyor belt more suited for smaller containers. This type of loading procedure is faster and uses less space but needs specialized loading platforms able to interact with the AGV by synchronizing the moving surfaces. Piggyback AGVs for this reason are best suited for filling in the gaps between successive handling stations, for example from a pallet loader to a wrapping station. [1]

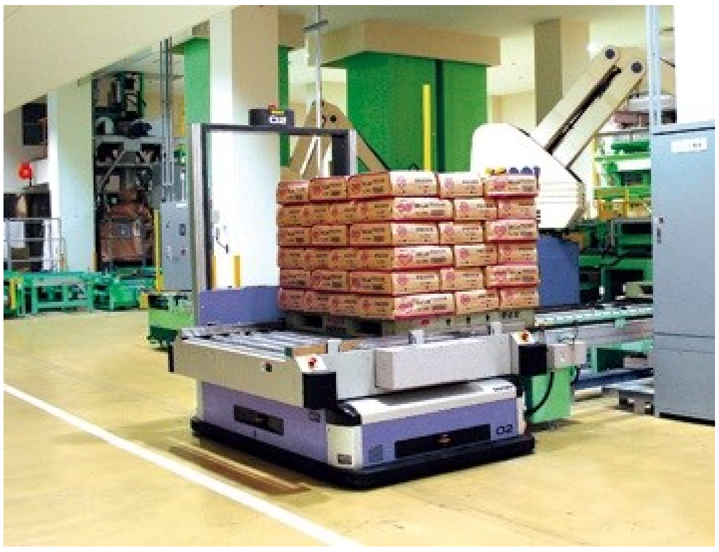


Figure 2: Side-loading AGV with rollers

2.3.3 Forklift AGV

Forklift AGVs are separated into two distinct categories: specially designed and automated serial equipment. Specially designed forklift AGVs are created from the ground up with the only expected use of autonomous movement and control applied only through a software manager. On the other hand, automated serial equipment forklifts are human-controlled machines that can be retrofitted or upgraded from the factory to be either both human and software controlled, or otherwise to lose the driver's equipment in favor specialized hardware. [1] [7]

While the first category of forklift AGV is more sophisticated, with a smaller footprint and the possibility of autonomous charging operations, the latter has more support from an already present production line which can supply eventual replacement parts, thus making maintenance more streamlined.

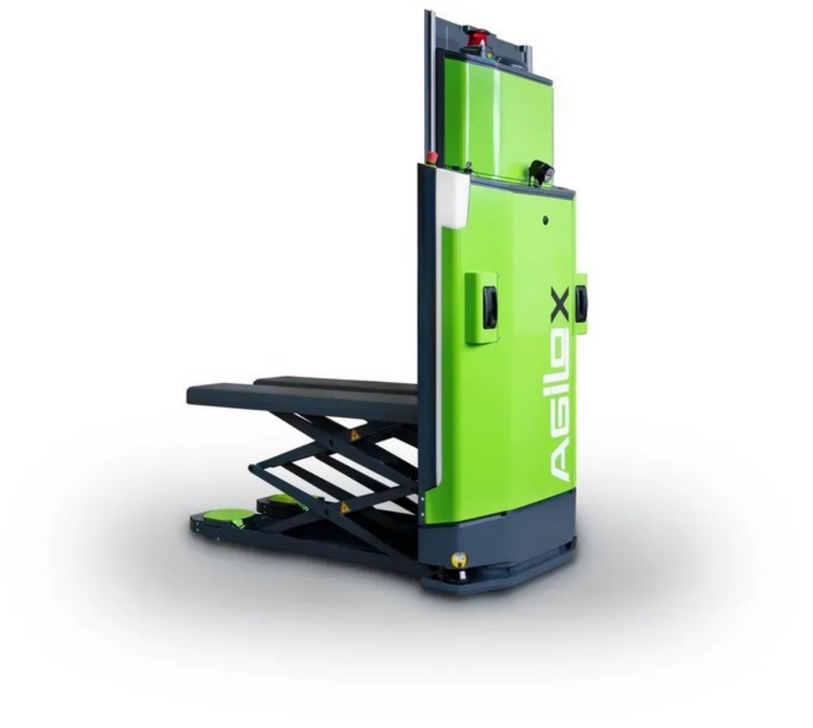


Figure 3: Specially designed Agilox Forklift AGV [7]

2.3.4 Towing AGV

Just like forklift AGVs, towing AGV can be either specially made for autonomous movement or adapted from a model initially designed for a human driver. Towing AGVs operated as the name suggests by towing a series of wheeled containers behind them. This configuration paired with wired tracks was one of the first models of AGV created. [1]

2.4 Smart applications

With the evolution from AGVs to AMRs and the introduction of machinery that is more capable, independent and has more computing power and sensors onboard, a whole swathe of new functionalities can be implemented.

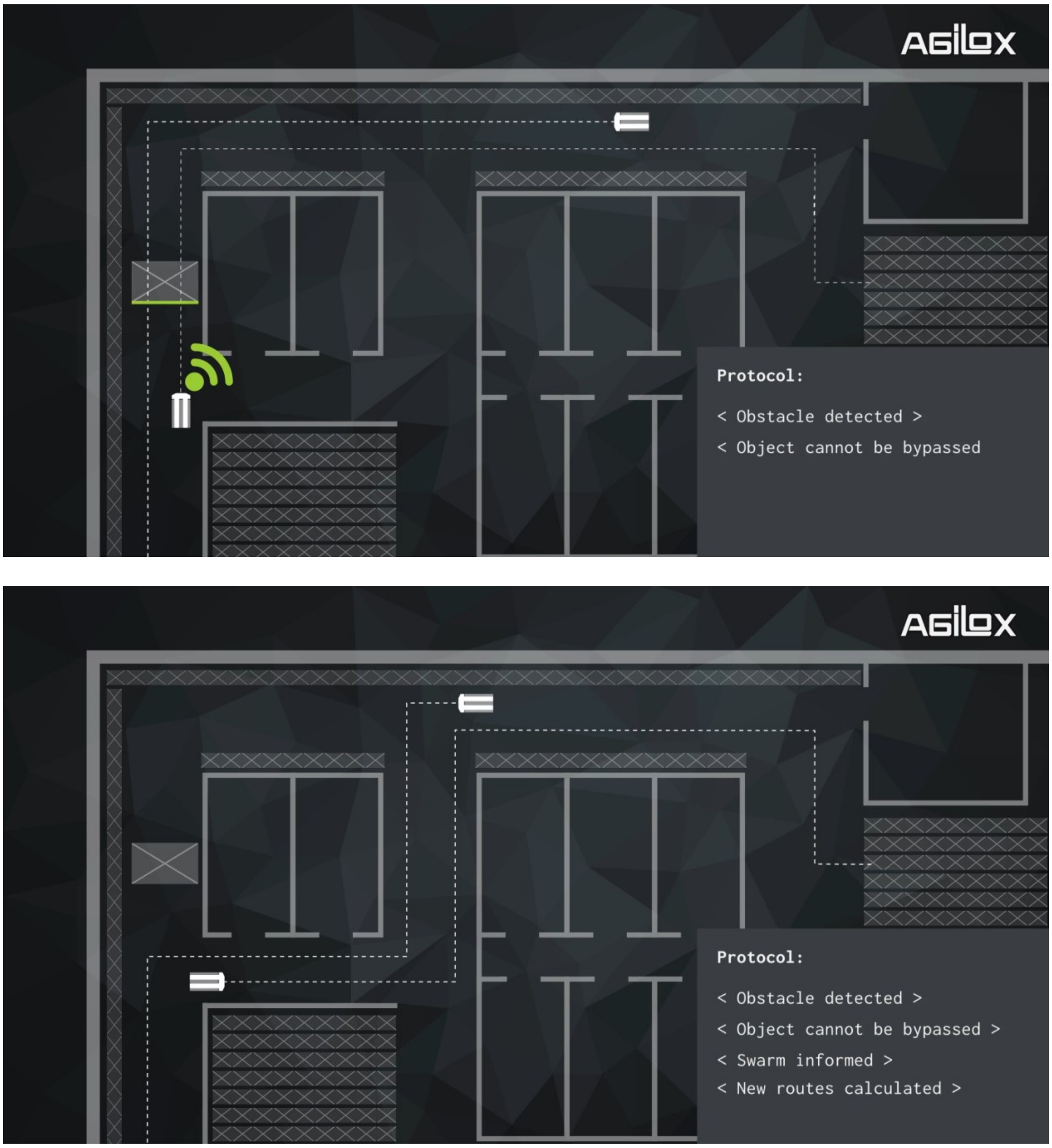
Among these functionalities are those that are called “smart applications” of logistics and navigation software, which are based on the cooperative work of multiple robots. Agilox systems is one of the companies which come into prominence in the field of AMR manufacturers thanks to these functionalities, Agilox was also the company that inspired part of this work. In this subsection we will briefly discuss some of them.

2.4.1 Task allocation

Task allocation consists in taking the weight off the user in assigning tasks to a specific robot in the fleet. Instead, given an updatable list of tasks that need to be taken care of, task allocation software will consider various parameters and machine states (such as AMR position, battery charge status, equipped sensors and hardware) to determine the right robot for the job. While this seems trivial for a small factory floor, in a logistics setting where thousands of jobs must be completed everyday task allocation can massively increase productivity if the software is correctly optimized, not to mention the man hours that can be redirected to less repetitive mansions. [8] [9]

2.4.2 Routing and navigation

In a setting which is shared with human operators, it can happen that new obstacles are introduced on the warehouse floor, disrupting the predetermined routes required by an AGV and such preventing its functioning. In the case of an AMR, not only becomes possible to circumvent obstacles that don’t completely obstruct the passageways, but it is also possible to share data about obstacles positions to other AMRs part of the fleet. This allows for recalculating a new route that not only the AMR which has detected the object will use, but also other members of the fleet might use without needing to incur firsthand in the obstacle. [8]



Protocol:

< Obstacle detected >

< Object cannot be bypassed

Protocol:

< Obstacle detected >

< Object cannot be bypassed >

< Swarm informed >

< New routes calculated >

Figure 4: Dynamic routing example, courtesy of Agilox [8]

2.4.3 Efficient deadlock avoidance

Having a fleet of AMRs navigating the typical logistics warehouse with long, narrow corridors and tight turning angles might generate deadlocks in the path generation. For example, two robots might be too wide to travel along an isle in opposed directions. In this case knowing all AMRs positions, assigned tasks, and a complete map of the environment can help generating new paths which avoid deadlock. [8]

While in an AGV setting, with fixed paths, this can result in a trade-off between most efficient routes and routes that are deadlock-safe; an AMR can compute and recompute paths at running time. By doing this it can chose the efficient routes in normal conditions while switching to safe routes while a generated path from another AMR conflicts with its own.

3 PROJECT SPECIFICATIONS

3.1 Development environment: ROS2

The Robot Operating System 2 (ROS2) [10] is a powerful and versatile framework that serves as the backbone of modern robotics development. A successor to ROS 1 [11], ROS 2 offers several key advantages for those in the robotics field. It provides a comprehensive ecosystem of tools, libraries, and middleware that simplifies the development and deployment of robotic systems. One of its most significant features is its enhanced flexibility, as ROS 2 supports various operating systems and real-time capabilities, making it adaptable to a wide range of robotic platforms and applications. ROS 2 is backed by a collaborative and open-source community, which enables robotics experts, researchers, and developers to share and take advantage of each other's work, accelerating innovation in the field. With its focus on robustness, security, and scalability, ROS 2 can play an important role in streamlining the development of robots in the fields of manufacturing, healthcare, logistics, and more. Finally, and most importantly for this thesis, ROS2 can be easily integrated with other libraries such as OpenCV [12], which are quite common in robotic libraries.

3.1.1 Nodes

A node is a fundamental ROS2 element that serves a single, modular purpose in a robotic system. [13] So, there will be one node for controlling wheel motors, one node for controlling a laser rangefinder, and so on. Each node can send and receive data to other nodes via topics, services, actions, or parameters, that are other ROS2 tools. ROS2 breaks complex systems down into many modular nodes, indeed a full robotic system is comprised of many nodes working in concert. In ROS2 a single executable can contain one or more nodes.

3.1.2 Message Topics, Services, and Actions

At the base of ROS2, there is the communication layer, based on the Data Distribution Service (DDS) standard. DDS is a standardized communication protocol that allows for data exchange in distributed systems. [14] This is one of the greatest jumps forward with respect to ROS1, where all the communication was managed by a central node: the ROS core. In ROS2, the DDS defines a method of communication between different languages API (such as *rclpy* and *rclcpp*). [15] This shift from a core-based model to an API model allowed robust, reliable, and real-time communication.

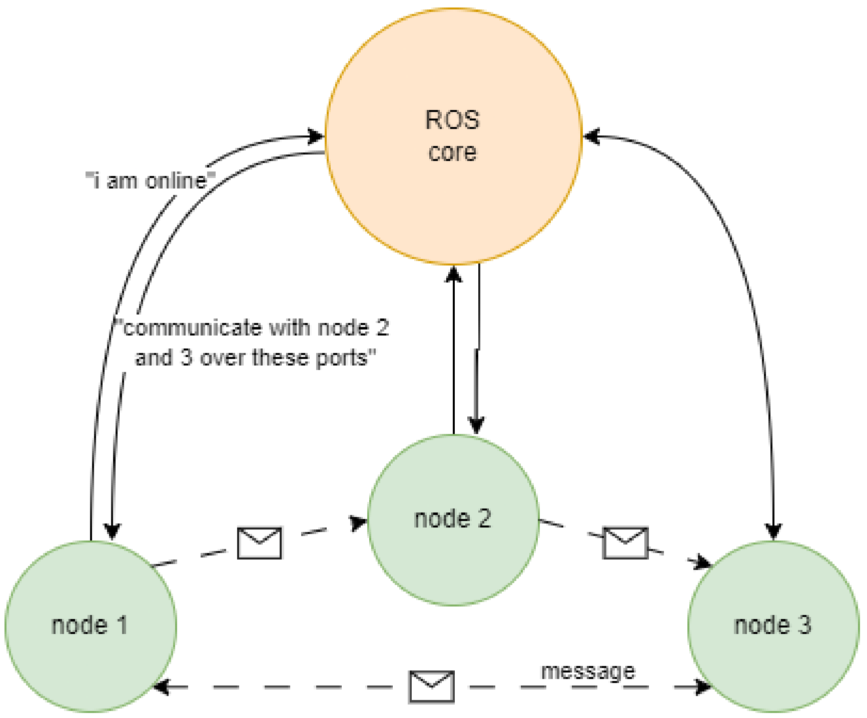


Figure 6: ROS1 communication between nodes

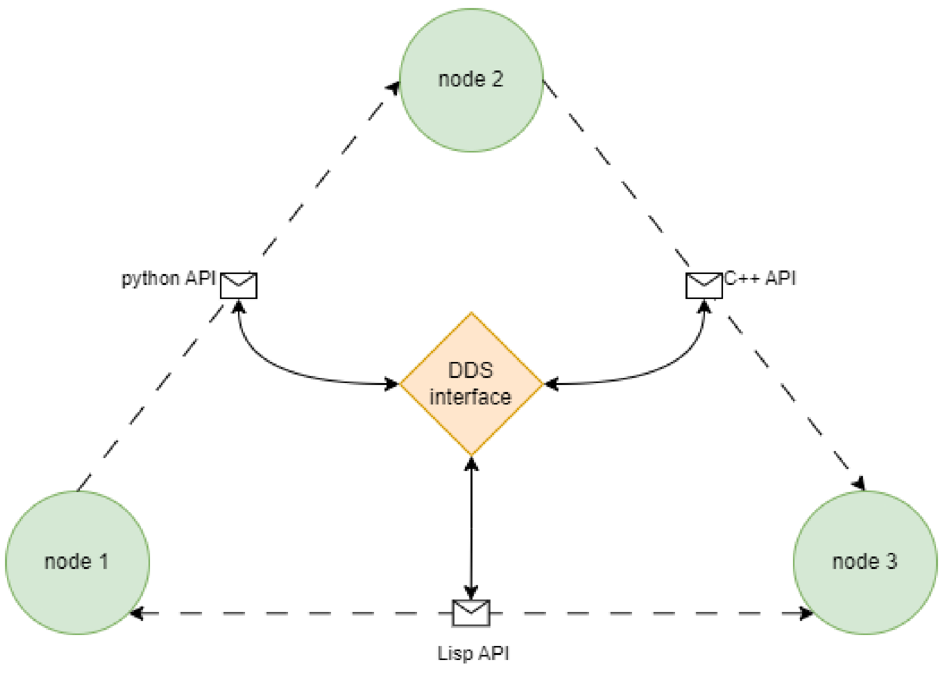


Figure 5: ROS2 DDS protocol

Topics are then established trough the DDS protocol, they act as unidirectional pipes trough which data is moved between nodes. [16] While nodes can either publish or subscribe to a topic, they can publish on or be subscribed to multiple topics at a time. This means that topics can be used not only to communicate one-to-one but also one-t- many or many-to-one.

Nodes and Topics together define a ROS graph, which is a network of interconnected computing modules processing data at the same time. [17]

This communication can happen trough three different kind of interfaces: messages, services, or actions.

Messages are the most basic concept used for communication between different parts of a robotic system, they contain data and represent typed information that the nodes exchange trough the ROS graph. [16] Messages are defined trough a plain-text format called "Message definition language" (.msg) these message definitions specify the type and structure of the data contained within the message. Each message type is composed of one or more fields containing both data type and name. A node will publish a message of defined type on a specific topic, which will be accessed and read by the subscriber node; this is the "default" kind of data transmission in ROS. Messages come pre-defined with some specific libraries or can be user-defined trough the message definition language for custom uses.

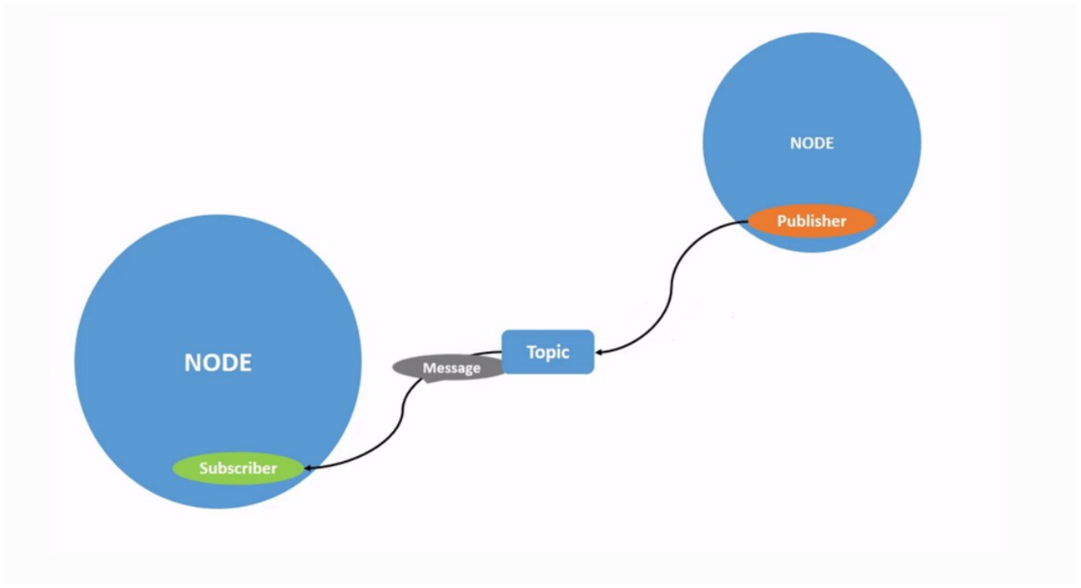


Figure 7: ROS2 message structure

Services present to nodes a mechanism to request and receive a specific, singular remote operation from another node in a request-response kind of communication. Just like messages, services must be defined trough a "service definition language" (.srv) which specifies both the structure and type of request and response messages. The request message defines the data that the client node sends to the server node, while the response message defines the data that the server node sends to the client. Services in ROS2 are typically synchronous and blocking, so the client blocks and waits a response from the server before proceeding with its task. [18]

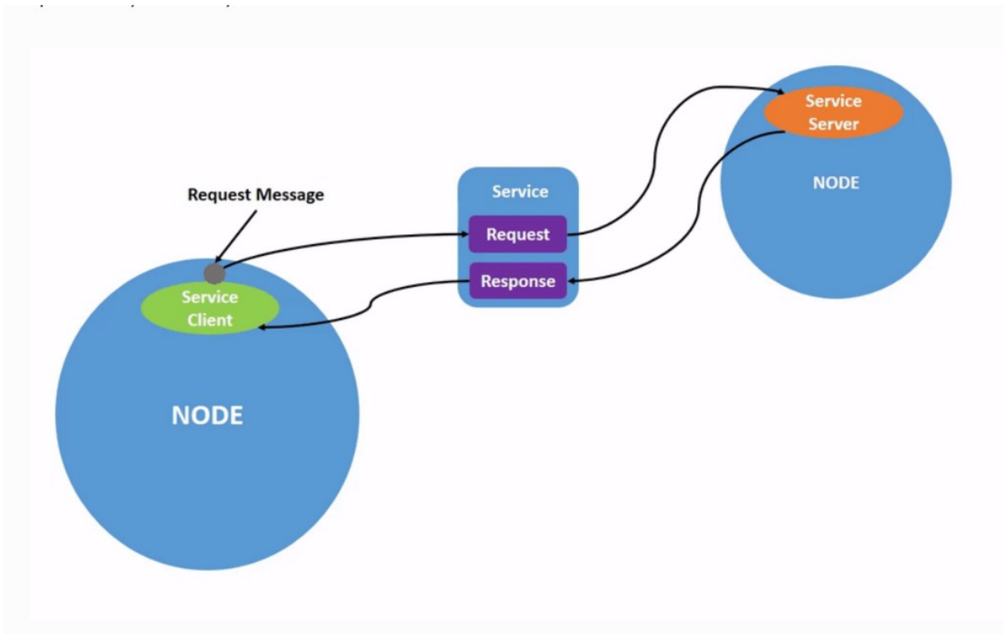


Figure 8: ROS2 service structure

Actions provide a more advanced form of communication between nodes compared to services and messages, allowing for asynchronous interactions between nodes. Actions are very useful for tasks that require feedback or progress and the ability to cancel or preempt an ongoing operation. As messages and services actions are defines trough an "action definition language" (.action) which specifies the type and structure of the content of an action message. An action message is split between a request and a response, like services, with an additional field reserved for feedback. [19]

Actions are of two types: simple actions and Goal-Status actions. Simple actions have a single goal, and the feedback is continuously updated as the action progresses, while Goal-Status actions allow the client to send multiple goals at once and receive feedback and results for each goal.

A node can be both an action server and an action client. An action server specifies which action are supported by it and what callbacks execute when a goal is called from a client. An action client sends a goal messages to the action server and receives both feedback and results. Action goals contain information about the task to be executed as specified in the .action file. During execution of the action, the action client can continue its task without waiting for the server response, this allows for asynchronous execution. During execution action goals can be both preempted and cancelled by the action client.

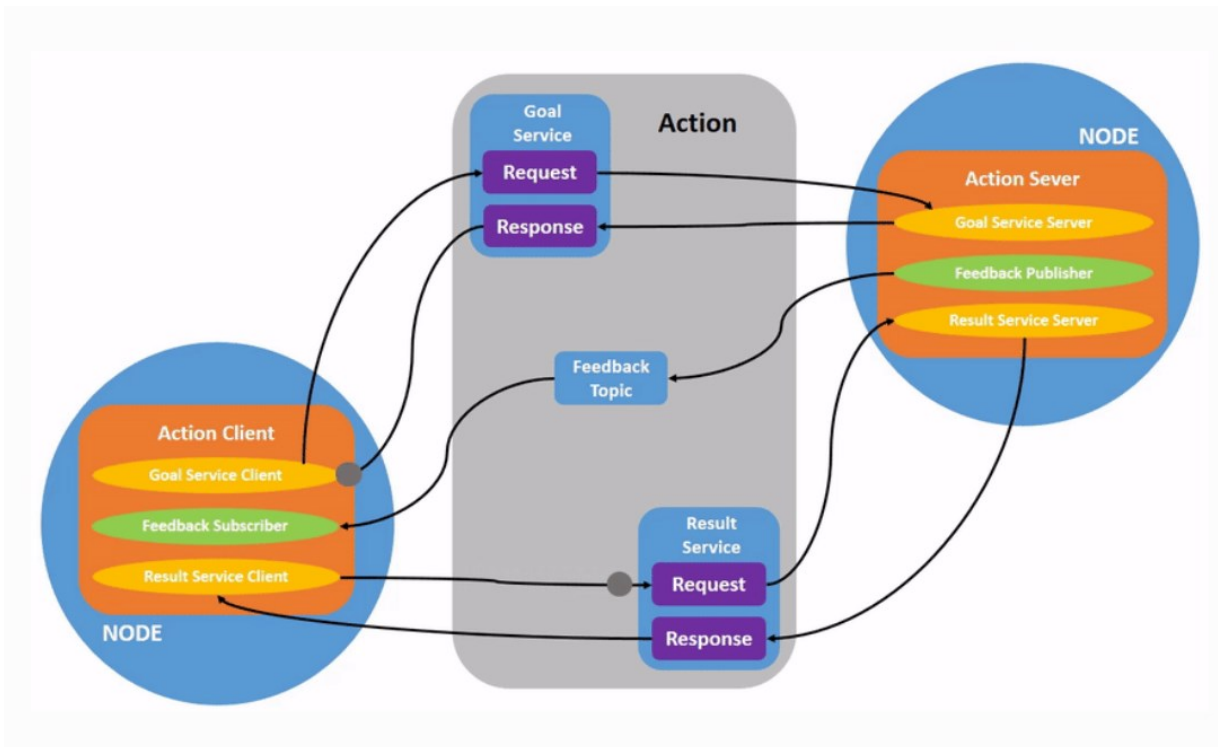


Figure 9: ROS2 action structure

3.1.3 Config and launch files

In ROS2 two types of files are available that make the process of launching nodes easier: configuration and launch files.

Launch files (.py/.xml) are files that specify the launch sequence of a collection of nodes, using specified settings and times. This simplifies the process of launching nodes and components by providing a standardized process of running nodes to execute a specific task. Launch files also make for a more streamlined robot startup procedure. [20]

Configuration files (.yaml) are files written in markup language that specify parameters and settings of the node that are going to be launched [21], [22] this is a way to manage code without having to recompile the source code every time some parameters need to be modified. YAML, (which stands for “YAML ain’t a markup language”) is a file extension similar to. Json or .xml but much more human readable. The configuration file containing the desired parameters is defined in the launch file, which is responsible to substitute the default values with the desired ones at the node launch.

3.2 Simulation environment

Since the development and testing of the docking functions of an AMR was the focus of this thesis, it became critical to define a safe and cost-effective way to evaluate the robotic behavior. Fortunately, with the ROS2 package come both the Gazebo and Rviz library, which are used respectively to simulate a physical machine and to visualize the ongoing processes and data.

3.2.1 Gazebo

Gazebo is a powerful and versatile open-source simulation software platform that has become a key element in the fields of robotics, autonomous systems, and artificial intelligence research. Gazebo provides a rich, 3D simulation environment for modeling, testing, and validating robotic systems in complex scenarios. [22]

Among the several benefits of using a simulation environment to test the developed code instead of a real machine, some of the most important are [22]:

Safety and Risk Mitigation: Testing docking functions in a simulated environment allowed us to identify and correct potential issues, such as collisions or navigation errors, without putting physical assets or personnel at risk.

Accessibility and Availability: Like in our case, access to a physical environment for testing may be limited or restricted, whereas a Gazebo simulation can be set up and conducted at any time, providing flexibility and availability for testing at various stages of development.

Rapid Prototyping: In a simulated environment, we could rapidly prototype and iterate on docking algorithms and behaviors without waiting for physical hardware modifications (adding a camera was a minute operation, instead of months) or adjustments, accelerating the development cycle.

Variability and Reproducibility: Gazebo simulations can replicate a wide range of scenarios, this way we could test complex behavior in a large environment and use a smaller one for benchmarking the navigation precision and docking speed.

Data Collection and Analysis: Gazebo provides tools for collecting detailed data and metrics during simulations. Which was of paramount importance to gather metrics on how well or badly the AMR was behaving while using one function or another.

One last benefit of Gazebo is the possibility to define environments and models from an editor internal to the platform, while also being able to import models as meshes from the internet, during testing this was extremely useful as we could quickly generate a warehouse-like testing ground and populate it with models of people and pallet to check the goodness of the obstacle avoidance functions. Later, as previously mentioned, a smaller map with a docking station was created to further test function metrics with more precision and repeatability.

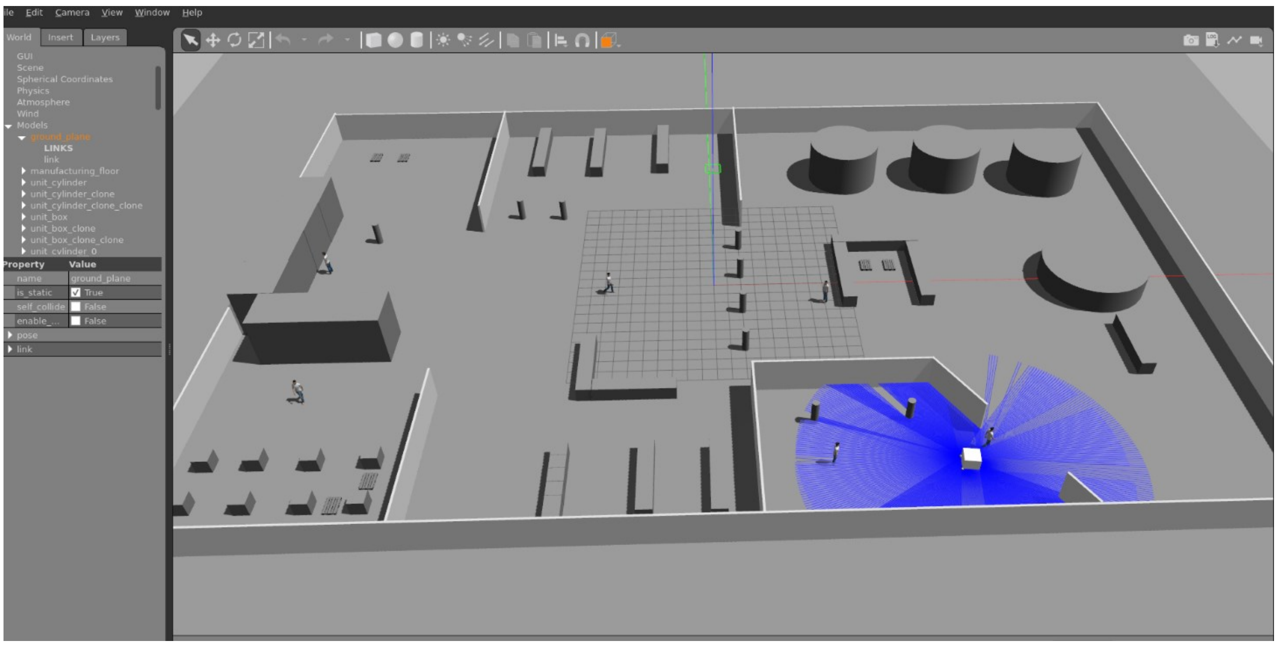


Figure 10: a view of the large warehouse environment with pallets and mannequin models

3.2.2 AMR model

Beside static models like the docking station and pallets, the Gazebo simulation can be also populated with moving objects which are connected to the ROS stack. This allows for generating a robot composed of several rigid or moving links able to simulate the real machine. The robot is described in the Unified Robot Description Format (.urdf) used to define the physical structure, kinematics, and various properties of a robot or robotic system. URDF is used as a standardized and concise way to represent the geometric, kinematic, and dynamic aspects of a robot. Here's a breakdown of its key components and functions:

Robot Structure: URDF describes the physical structure of the robot, including its links and joints. Links represent rigid bodies or components of the robot, while joints define how these links are connected and can move with respect to each other. [23]

Geometry: The specification of the geometric properties of robot components, such as meshes, collision geometries, and visual representations. This geometric information is crucial for visualization and collision detection.

Inertial Properties: URDF includes parameters for defining the inertial properties of links, such as mass, center of mass, and inertia matrices. These properties are essential for simulating the dynamics and motion of the AMR. [24] [25]

Sensors: the URDF file can be extended to include sensor descriptions, such as cameras, LiDAR, and IMU (Inertial Measurement Unit) sensors, along with “real” parameters and imperfections such as field of view, optical center, range, and noise. This information is essential for simulating sensor data and sensor-based perception in robot simulations. [26]

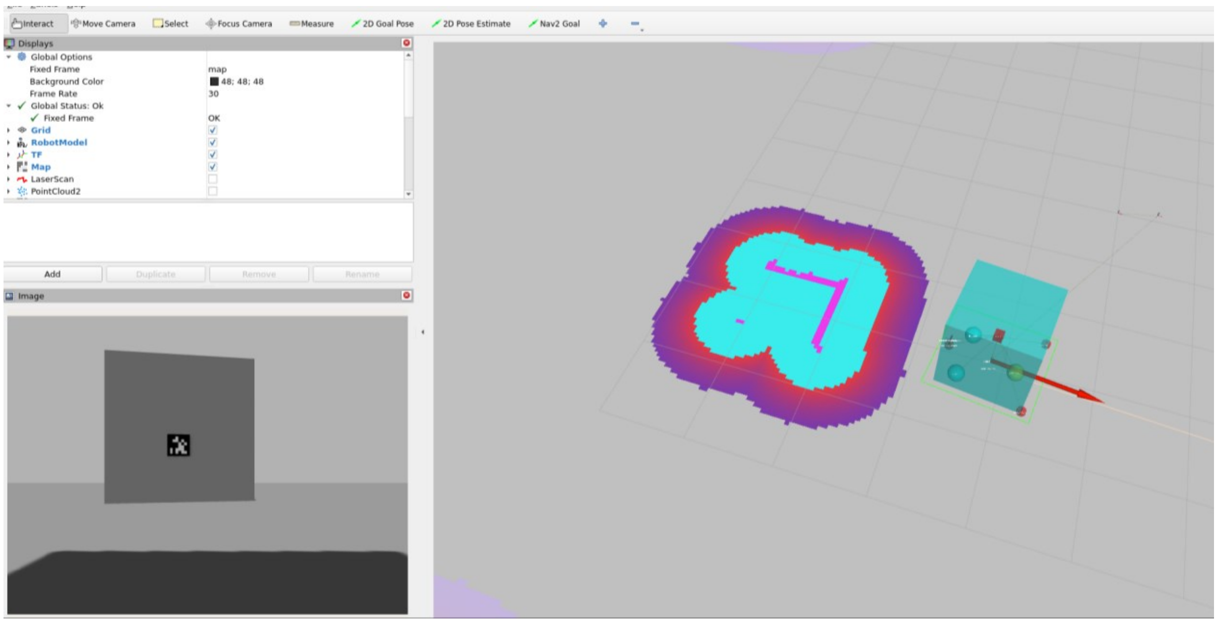
3.2.3 Rviz

RViz, short for "ROS Visualization," is a visualization software tool used by ROS2. [27] It provides a rich and interactive 3D visualization environment, allowing users to better understand the state, behavior, and perception of the robotic system that is being monitored.

RViz is a versatile and highly configurable platform that serves multiple crucial functions. First and foremost, it enables real-time visualization of robot sensor data, such as point clouds, laser scans, and camera images, providing a means of assessing the robot's perception of its surroundings. Moreover, RViz allows users to visualize the robot's internal state, including joint configurations, trajectories, and odometry information, increasing the understanding of its kinematics and dynamics; thus, making debugging of the robotic platform easier. [28]

Its interactive tools enable users to set goals, plan paths, and even teleoperate robots within the visual interface, making it an essential tool for robot testing. RViz's extensibility helps developers to tailor the visualization environment to specific robotic platforms and applications, enhancing its adaptability across various chassis and use cases, from industrial automation to autonomous vehicles and robotics research.

During the duration of this thesis, RVIZ was essential to interact with the robotic base, check for malfunctions of the docking process and visually check the progress of given tasks at times where the command line output was not enough verbose to understand what was happening.



Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	✓
TF	✓
Map	✓
LaserScan	☐
PointCloud2	☐

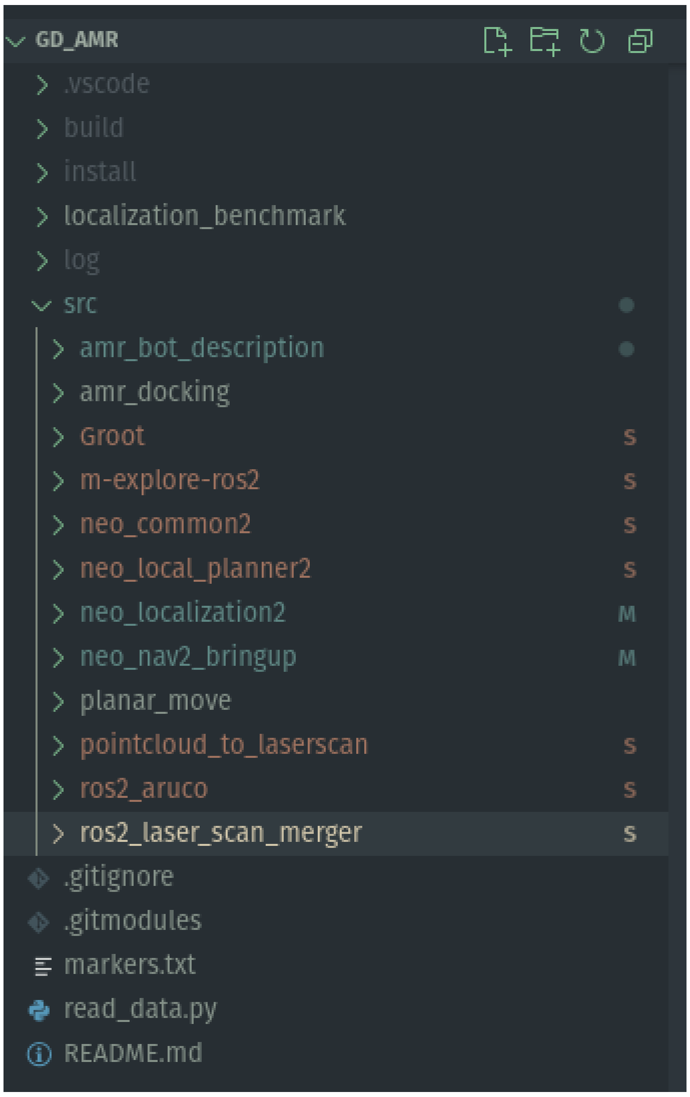
Figure 11: RVIZ visualization, we can see the camera input, robot model, inflation map and frames.

4 METHODS DESCRIPTION

In this chapter we discuss the methods used to complete all the passages that were necessary to finalize a complete docking function. While most steps of the project had been considered at the start of development, some others surfaced only because of the work progression.

4.1 Set up

The first passage of the project was to install the Ubuntu 22.04 operating system along with the ROS 2 framework. After the installation, a directory was created to be used as the workspace, with this initial structure:



```
GD_AMR
  > .vscode
  > build
  > install
  > localization_benchmark
  > log
  v src                              ●
    > amr_bot_description            ●
    > amr_docking
    > Groot                          S
    > m-explore-ros2                 S
    > neo_common2                    S
    > neo_local_planner2             S
    > neo_localization2              M
    > neo_nav2_bringup               M
    > planar_move
    > pointcloud_to_laserscan        S
    > ros2_aruco                     S
    > ros2_laser_scan_merger         S
  .gitignore
  .gitmodules
  markers.txt
  read_data.py
  README.md
```

Figure 12: The workspace directory, in the src folder are listed all submodules used.

In the workspace folder [29] are contained the main three subdirectories:

- In the *install* directory all the files that are necessary for the various libraries to function are contained, this includes binaries and automatically generated header files required by some of the code written in C++.
- In the *build* directory all the files generated by the build process are contained, this includes build reports and error lists.
- The *src* directory contains all the source code.

Src is effectively the working folder of the project, since all the code that is written is contained in this folder, which then gets separated into different independent directories. Each directory is completely self-sufficient and contains its own build files. This allows for copying one of the directories where a specific function is programmed into a different workspace, running the build program *$ colcon build* in the second folder and having ROS manage the dependencies during the build process.

4.2 Sensor Fusion

After building the NAV 2 library and creating both the simulations of the AMR and the Gazebo environment, we encountered the first problem: the localization library could not accept more than one scanner input at a time, this approach collided with the desired configuration of the AMR sensor suite, as three different LIDARs were expected around the robot body. Aside from that, the presence of a 3D camera allowed for different approaches to near-target navigation and to further increase the localization capabilities. The option to utilize this data stream was later considered and evaluated.

4.2.1 Laser scan merging

During the configuration of the NAV 2 library, setting more than one source of scanner data in the section of the *.config* file relative to localization resulted in a series of errors, as it is visible from this *.config* file snippet, it is possible to set up only one source for that sensor type.

```
amcl:
 ros__parameters:
  scan_topic: scan
  pointcloud_topic: cloud
```

It was later understood that NAV 2 has not been set up to read only one scanner topic at the time, this is due to the scanners publishing not a three dimensional array of coordinates, but instead a two dimensional array of distances, point from which these distances are computed is not included in the message structure, this means that setting more than a scanner source with different origins would result in errors during the localization process. Since working without NAV 2 meant having to rewrite both the navigation and localization functions, along with the full suite of controllers, drivers, and simulation packages, it became clear that it was necessary to find some expedient to circumnavigate this issue while maintaining functionality.

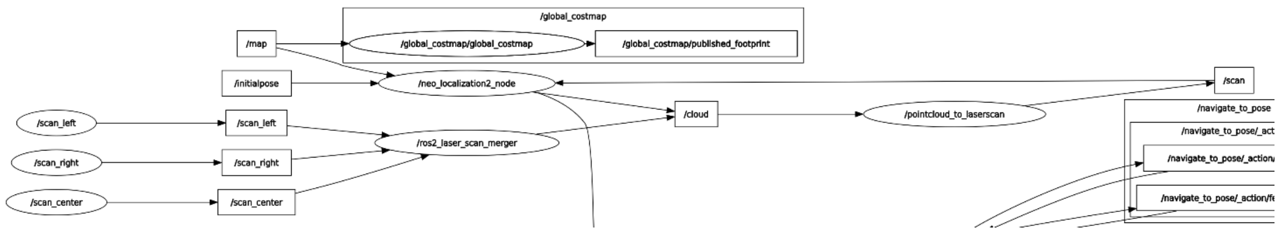


Figure 13: The topic graph of the lidar merger and pointcloud to laserscan pipeline, it is possible to see the three lidars merged into one.

We started researching different options to fuse together the scan topics, one of these options was a library written for ROS 1 [30] [31]. While this solution had been proven to work, it was not compatible with the ROS 2 framework. Fortunately, after additional exploration a remarkably similar candidate, compatible with ROS 2, was found. This library, called "lidar merger" [32], works by generating a virtual lidar, with an origin that can be defined at will. The lidar merger function listens to the desired scan topics, all of which have a position and orientation preset in the lidar merger configuration file.

```
ros2_laser_scan_merger:
 ros__parameters:
  pointCloudTopic: "/cloud"
  pointCloutFrameId: "lidar_frame"
```

```
use_sim_time: True
scanTopic1: "/scan_right"
laser1XOff: 0.5
laser1YOff: -0.5
laser1ZOff: 0.0
laser1Alpha: -180.0
laser1AngleMin: 1.0
laser1AngleMax: 265.0
laser1R: 255
laser1G: 0
laser1B: 0
show1: True
```

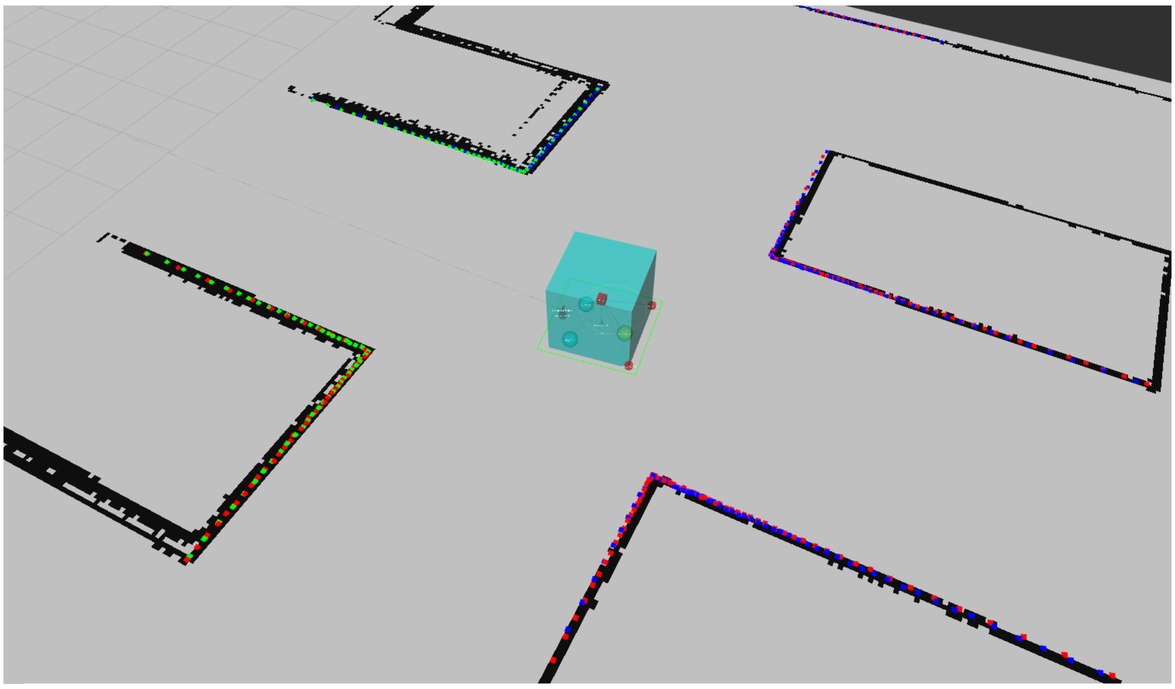


Figure 14: The detected points from the three separate laser scanners as seen in the pointcloud, with color separation in blue, red and green.

In this configuration file it is possible to see how the position and field of view of each LIDAR (the back left one in this case) are defined in the adapter node. Initially it was not possible to correctly run the node as it outputted and exotic error message. Fortunately, after a lot of debugging and contacting the developer behind the laser scan merger repository, it was possible to find out that the error depended on the type of some of the variables relative to the point cloud color array encoding, after changing the type from *int* to *uint8_t*, the code started working correctly [33].

```
this->get_parameter_or<uint8_t>("laser1R",laser1R_, 0);
    this->get_parameter_or<uint8_t>("laser1G",laser1G_, 0);
    this->get_parameter_or<uint8_t>("laser1B",laser1B_, 0);

std::string topic1_, topic2_, topic3_, cloudTopic_, cloudFrameId_;
  bool show1_, show2_, show3_;
  float laser1XOff_, laser1YOff_, laser1ZOff_, laser1Alpha_, laser1AngleMin_, laser1AngleMax_;
  uint8_t laser1R_, laser1G_, laser1B_;
```

The software reads the received distance arrays, computes a trigonometric function to compute the distance of the collision point detected by scanner A with respect to the desired virtual LIDAR position and populates a three-dimensional array (cloud topic) with the computed coordinates.

```
void update_point_cloud(){
  // RCLCPP_INFO(this->get_logger(), "Hello basic");
  refresh_params();
  //pcl::PointCloud<pcl::PointXYZRGB> cloud_;
  pcl::PointCloud<pcl::PointXYZ> cloud_;
  std::vector<std::array<float,2>> scan_data;
  int count = 0;
  float min_theta = 0;
  float max_theta = 0;
  if(show1_){
    for (float i = laser1_->angle_min; i <= laser1_->angle_max; i += laser1_->angle_increment){


      pcl::PointXYZ pt;
      float temp_x = laser1_->ranges[count] * std::cos(i) + laser1XOff_;
      float temp_y = laser1_->ranges[count] * std::sin(i) + laser1YOff_;
      pt.x = temp_x * std::cos(laser1Alpha_ * M_PI / 180) - temp_y * std::sin(laser1Alpha_ * M_PI /
180);
      pt.y = temp_x * std::sin(laser1Alpha_ * M_PI / 180) + temp_y * std::cos(laser1Alpha_ * M_PI /
180);
      pt.z = laser1ZOff_;
```

```
        if (i < (laser1AngleMin_ * M_PI / 180)){
          //do nothing
        }else if(i > (laser1AngleMax_ * M_PI / 180)){
          //do nothing
        }else{
          cloud_.points.push_back(pt);
          float r_ = GET_R(pt.x, pt.y);
          float theta_ = GET_THETA(pt.x, pt.y);
          std::array<float,2> res_;
          res_[1] = r_;
          res_[0] = theta_;
          scan_data.push_back(res_);
          if(theta_ < min_theta){
            min_theta = theta_;
          }
          if(theta_ > max_theta){
            max_theta = theta_;
          }
        }
        count++;
      }
   }
```

The following functions are used to compute the distance from center of the scanner to the detected points (GET_R), find the angle of the detected point with respect to the mounting angle of the LIDAR (GET_THETA + interpolate):

```
float GET_R(float x, float y){
   return sqrt(x*x + y*y);
}
float GET_THETA(float x, float y){
   float temp_res;
   if((x!=0)){
      temp_res = atan(y/x);
```

```
  }else{
    if(y>=0){
      temp_res = M_PI / 2;
    }else{
      temp_res = -M_PI / 2;
    }
  }
  if(temp_res > 0){
    if(y < 0 ){
      temp_res -= M_PI;
    }
  }else if(temp_res <0){
    if(x < 0){
      temp_res += M_PI;
    }
  }
  // RCLCPP_INFO(this->get_logger(), "x: '%f', y: '%f', a: '%f'", x, y, temp_res);
  return temp_res;
}
float interpolate(float angle_1, float angle_2, float magnitude_1, float magnitude_2, float
current_angle){
  return (magnitude_1 + current_angle * ((magnitude_2 - magnitude_1)/(angle_2 - angle_1)));
}
```

In the case where more than one collision point is detected for the same virtual scanner angle, only the closest point is considered. This can happen when the robot is close to an uneven surface like a corner, in this case while the scanner A sees at the virtual angle 90 degrees a collision at distance 50cm, scanner B sees at the same angle a collision at distance 100cm; this problem is due to parallax errors.

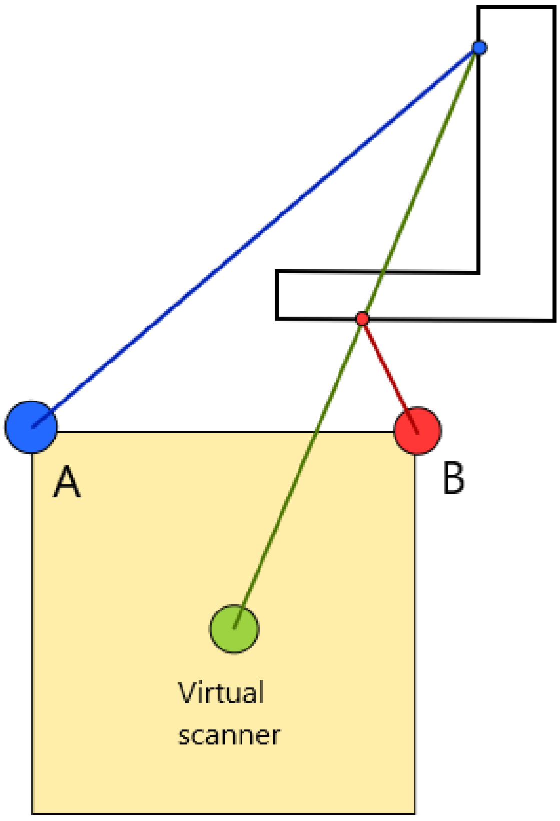


Figure 15: Schema illustrating a possible instance of the parallax problem

The output of the previous snippets of code is sent as cloud type message topic to the node *pointcloud_to_laserscan*. Using the *cloud* topic published by the *lidar_merger* as a "collector file" the *pointcloud_to_laserscanner* node then computes back a two-dimensional array which contains the distances of the obstacles detected by the three lidar scanners from the virtual LIDAR at all selected angle fractions and subsequently publishes a message on the topic connected with the generated virtual scanner. As we can see in the next piece of code, all points that are visible in the pointcloud but would not be in the distance and height range of the virtual scanner specifications are rejected:

```
// Iterate through pointcloud
```

```
for (sensor_msgs::PointCloud2ConstIterator<float> iter_x(*cloud_msg, "x"),
  iter_y(*cloud_msg, "y"), iter_z(*cloud_msg, "z");
  iter_x != iter_x.end(); ++iter_x, ++iter_y, ++iter_z)
{
  if (std::isnan(*iter_x) || std::isnan(*iter_y) || std::isnan(*iter_z)) {
    RCLCPP_DEBUG(
      this->get_logger(),
      "rejected for nan in point(%f, %f, %f)\n",
      *iter_x, *iter_y, *iter_z);
    continue;
  }

  if (*iter_z > max_height_ || *iter_z < min_height_) {
    RCLCPP_DEBUG(
      this->get_logger(),
      "rejected for height %f not in range (%f, %f)\n",
      *iter_z, min_height_, max_height_);
    continue;
  }

  double range = hypot(*iter_x, *iter_y);
  if (range < range_min_) {
    RCLCPP_DEBUG(
      this->get_logger(),
      "rejected for range %f below minimum value %f. Point: (%f, %f, %f)",
      range, range_min_, *iter_x, *iter_y, *iter_z);
    continue;
  }
  if (range > range_max_) {
    RCLCPP_DEBUG(
      this->get_logger(),
      "rejected for range %f above maximum value %f. Point: (%f, %f, %f)",
      range, range_max_, *iter_x, *iter_y, *iter_z);
    continue;
  }
  double angle = atan2(*iter_y, *iter_x);
  if (angle < scan_msg->angle_min || angle > scan_msg->angle_max) {
    RCLCPP_DEBUG(
      this->get_logger(),
      "rejected for angle %f not in range (%f, %f)\n",
      angle, scan_msg->angle_min, scan_msg->angle_max);
    continue;
  }

  // overwrite range at laserscan ray if new range is smaller
  int index = (angle - scan_msg->angle_min) / scan_msg->angle_increment;
  if (range < scan_msg->ranges[index]) {
    scan_msg->ranges[index] = range;
  }
}
```

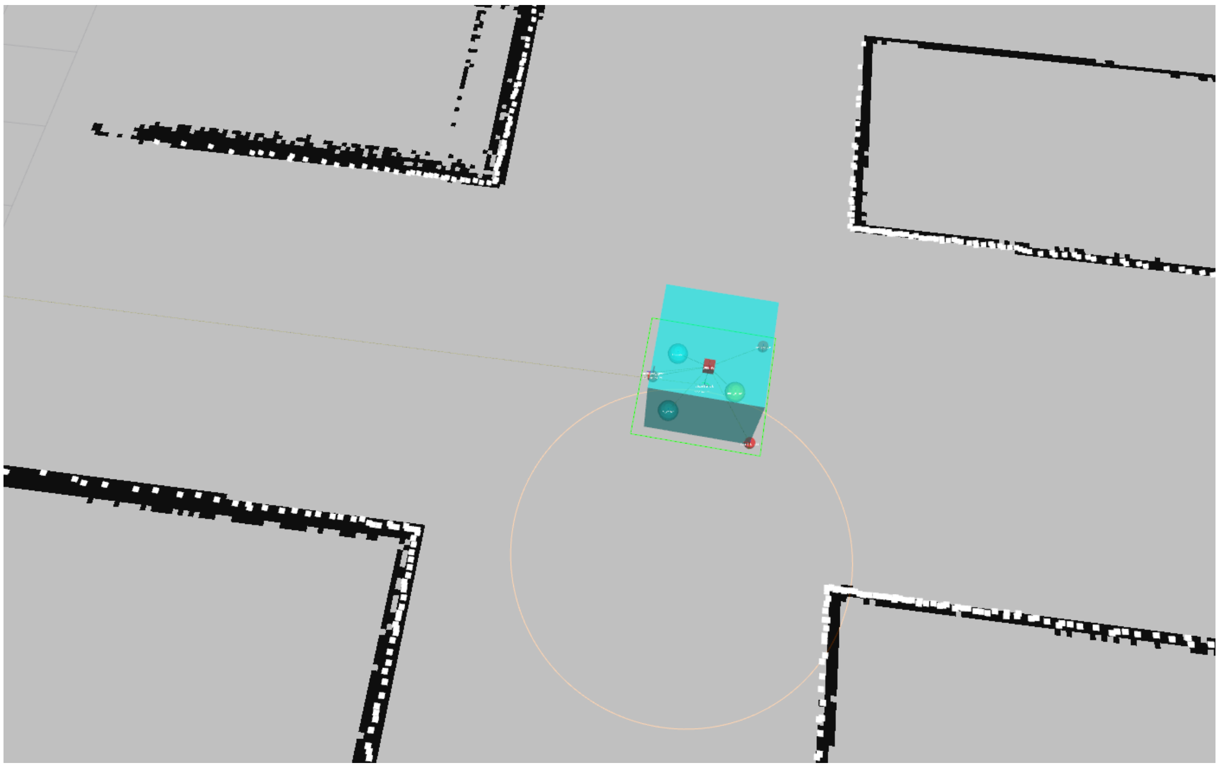


Figure 16: Unified scanner topic, the detected points with parallax correction are visible in white.

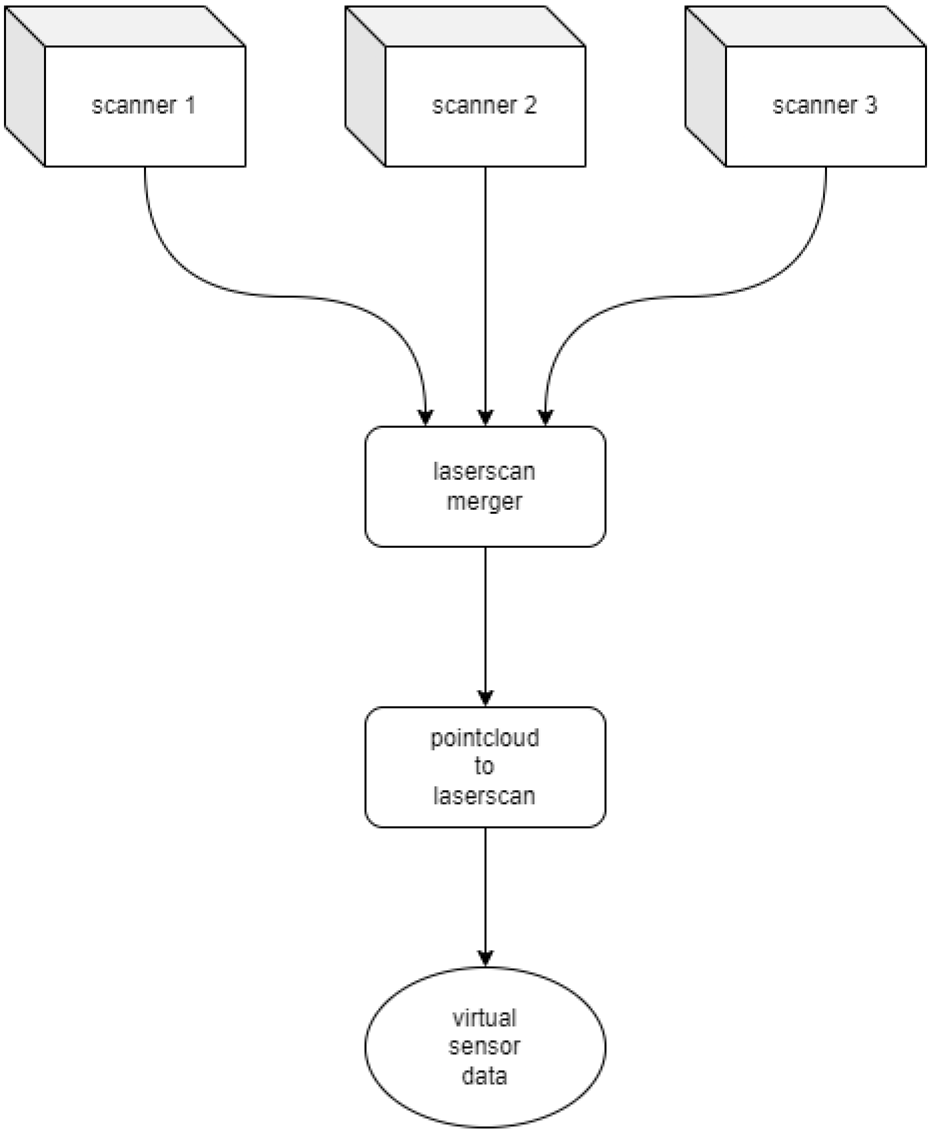
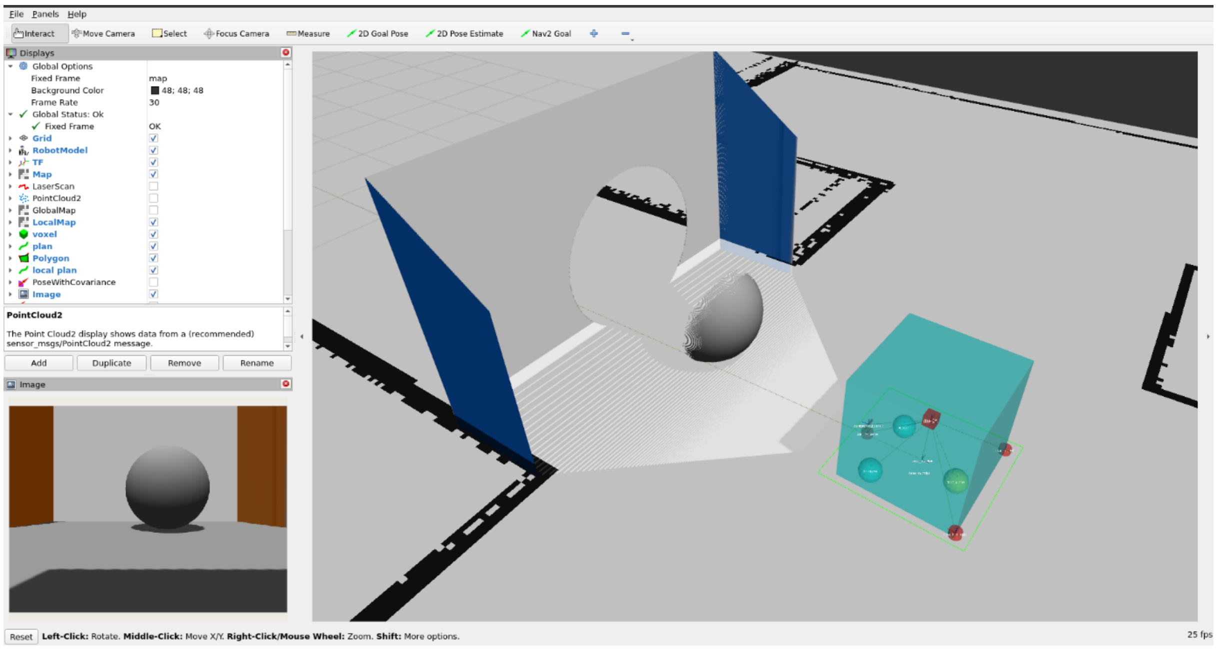


Figure 17: Final lidar merger pipeline

4.2.2 3D camera cloud

Figure 18: Framos 430e depth camera [38]

During development, the possibility of extending the obstacle detection through an IR emitter stereo depth sensor was considered. The 3D camera works by projecting a lattice of points in the infra-red band, from there, the depth of placement of each point is detected by determining the position in the frame of both IR cameras positioned at the sides of the sensor. This depth of field data is then sent to ROS 2 both as a point cloud in a tree dimensional array and trough an image augmented with depth defined as a six-dimensional tensor [34] (both point in space and color of pixel expressed in the RGB channels); the color value is obtained through a third sensor positioned in the center, near the emitter.



```
Global Options
    Fixed Frame          map
    Background Color     48; 48; 48
    Frame Rate           30
Global Status: Ok
    Fixed Frame          OK
Grid                 ✓
RobotModel           ✓
TF                   ✓
Map                  ✓
LaserScan
PointCloud2
GlobalMap
LocalMap             ✓
voxel                ✓
plan                 ✓
Polygon              ✓
local plan           ✓
PoseWithCovariance
Image                ✓
```

PointCloud2

The Point Cloud2 display shows data from a (recommended) sensor_msgs/PointCloud2 message.

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:** Zoom. **Shift:** More options.

Figure 19: depth camera simulated inside azebo

The cloud topic message can be used by NAV 2 to extend the obstacle detection from a single plane to the three-dimensional space by projecting a voxel layer in the simulation. Although this might help avoid collisions with hanging obstacles, the computation overhead and the added complexity were considered not worth implementing. Moreover, in order to compute the robot navigation path, the voxel layer gets compressed in the same plane as the one generated by the LIDAR data contained in the scan topic, making it redundant for all type of obstacles other than the hanging ones.

4.3 Navigation

In this section we discuss the process of configuring a working localization and navigation stack. The starting points was to install and test Nav2 (Navigation 2). [35] Despite being well documented and tested by open-source developers, the process has been affected by a couple of sensible setbacks. The first one was dependent on some changes to the newest ROS 2 version main, while the second one stemmed from the configuration of the AMR. Both have been addressed in their subsection.

4.3.1 Installation

The installation process of NAV 2 is easily explained in the library documentation, [35] it works in a slightly different way from standard ROS 2 packages as it can be installed directly on the operating system without the need for copying the navigation stack folder in the project workspace and building from there, this is done using the commands:

```
$ sudo apt install ros-humble-navigation
$ sudo apt install ros-humble-nav2-bringup
```

After installing the package, it will be possible to run any of the available scripts directly from a ROS 2 directory.

4.3.2 Configuration of NAV 2

As with other ROS2 libraries, NAV2 is configured through a YAML file, here we discuss the most important parameters that define the behavior of the localization stack. [36]

```
amcl:
 ros__parameters:
   use_sim_time: True
   alpha1: 0.2
   alpha2: 0.2
   alpha3: 0.2
   alpha4: 0.2
   alpha5: 0.2
   base_frame_id: "base_footprint"
   beam_skip_distance: 0.5
   beam_skip_error_threshold: 0.9
   beam_skip_threshold: 0.3
   do_beamskip: false
   global_frame_id: "map"
   lambda_short: 0.1
   laser_likelihood_max_dist: 2.0
   laser_max_range: 20.0
   laser_min_range: 0.75
   laser_model_type: "likelihood_field"
   max_beams: 60
   max_particles: 2000
   min_particles: 500
   odom_frame_id: "odom"
   pf_err: 0.05
   pf_z: 0.99
   recovery_alpha_fast: 0.0
   recovery_alpha_slow: 0.0
   resample_interval: 1
   robot_model_type: "nav2_amcl::OmniMotionModel"
   save_pose_rate: 0.5
   sigma_hit: 0.2
   tf_broadcast: true
   transform_tolerance: 1.0
   update_min_a: 0.2
```

```
update_min_d: 0.25
z_hit: 0.5
z_max: 0.05
z_rand: 0.5
z_short: 0.05
scan_topic: scan
pointcloud_topic: cloud
set_initial_pose: True    # default: {x: 0.0, y: 0.0, z: 0.0, yaw: 0.0}
```

To localize, NAV2 uses adaptive Monte-Carlo localization, here we define the most important parameters that were initially set:

Alpha values: expected noise levels in odometry position and rotation estimates.

Laser values: define the type of sensor used by the robot, in this case the virtual sensor at the center of the AMR.

Max and mean particles: the starting and final number of particles on which is built the probability distribution function of the robot position. Each particle is a prediction of the robots next position, the number of particles decreases as much as the probability of one position over the other possible ones increases.

Robot model type: defines the type of robot chassis/controller, the available configurations are differential, omnidirectional (our case) and steering.

Z values: weights of the model which sum must equal 1.

Scan and cloud topic: the topic over which the data used for checking the particles precision is collected.

4.3.3 Local and Global map error

NAV 2 localization and navigation both are dependent on a local and global map, both created from detected obstacles. In the first case the local map is generated as 10 meters by 10 meters grid containing all the points coming from the laser scanner in real time. Instead, the global map is generated with both points coming from a map that can be manually set trough a bitmap (or generated through a SLAM algorithm such as cartographer) and the points that are detected trough the local map and "remembered" in the same position even when the robot leaves that area.

NAV 2 localization works with a Monte-Carlo algorithm, by matching the local map with the global at randomized points and selecting the ones that better approximate the actual position at successive steps. Instead, the navigation stack uses the global map to generate movement paths and the local map to avoid collisions with unexpected obstacles.

The problems with both maps arose after the configuration of the library, during testing inside the Gazebo simulation. We noticed that, even if the robot correctly simulated lidar rays, the laser merger was working and creating a functional virtual lidar, and Rviz projected the detected obstacle points, it was impossible to visualize both local and global map.

After intensive search and a lot of changes in the configuration file, it was found that the problem did not depend on the NAV 2 library but instead on the DDS middleware: in the ROS Humble version we were using a problem exist where the default DDS vendor (Fast DDS) is incompatible with the NAV 2 message update frequency. [37] [38] Substituting Cyclone DDS to Fast solved the problem, this was done using the commands:

```
$ sudo apt install ros-humble-rmw-cyclonedds-cpp
$ ~/.bashrc export RMW_IMPLEMENTATION=rmw_cyclonedds_cpp
```

After changing the DDS vendor, local and global map started showing, and testing could proceed.

4.3.4 Neobotics controller

The second problem encountered was relative to the robot controller. At the start of the project a robot specification was defined for omnidirectional wheels, but the motion of the AMR had to follow a differential "pure pursuit" path planning (while maintaining omnidirectional capabilities) that was not possible to implement with the available NAV 2 controllers. This research for a controller that made available differential control on omnidirectional wheels led to the Neobotics code stack.

Neobotics is a private company specialized in omnidirectional robot for logistics and has adapted the NAV libraries for that particular use case. [39]

```
controller_server:
 ros__parameters:
   # controller server parameters (see Controller Server for more info)
```

```
controller_plugins: ["FollowPath"]
controller_frequency: 100.0
failure_tolerance: 0.3
odom_topic: "odom"
controller_plugin_types: ["neo_local_planner::NeoLocalPlanner"]
____________________
      …
____________________
FollowPath:
  plugin: "neo_local_planner::NeoLocalPlanner"
  acc_lim_x : 0.50
  acc_lim_y : 0.50
  acc_lim_theta : 0.8
  max_vel_x : 1.2
  min_vel_x : -0.5
  max_vel_y : 0.5
  min_vel_y : -0.5
  max_rot_vel : 1.0
  min_rot_vel : -1.0
  max_trans_vel : 1.2
  min_trans_vel : -0.5
  # lower limits for localization precision to accept goal reached
  yaw_goal_tolerance : 0.005
  xy_goal_tolerance : 0.01
  # not strictly a time but a set of distances based on robot speed
  goal_tune_time : 2.0
  lookahead_time : 0.4
  lookahead_dist : 1.0
  start_yaw_error : 0.5
  # gains for the pure pursuit algorithm
  pos_x_gain : 10.0
  pos_y_gain : 10.0
  static_yaw_gain : 15.0
  cost_x_gain : 0.1
  cost_y_gain : 0.1
```

```
    cost_y_lookahead_dist : 0.0
    cost_y_lookahead_time : 0.3
    cost_yaw_gain : 2.0
    low_pass_gain : 0.1
    max_cost : 0.95
    # defines lower limit for curve radius at speed
    max_curve_vel : 0.3
    max_goal_dist : 0.2
    # only used for steering robots
    max_backup_dist : 0.0
    min_stop_dist : 0.6
    # our controller acts just as a differential drive, but has the possibility to switch to holonomic near
target
    differential_drive : false
```

4.3.5 Neobotics localization

While configuring the Neobotics controller, we found that Neobotics also released a localization package also based on AMCL but with update optimization using the Gauss-Newton iterations. [41] We found we could obtain much better localization precision using this module, so it was added to the stack along with the controller submodule. The following are the relative configuration parameters used:

```
neo_localization2_node:
  ros__parameters:
    base_frame: "base_footprint"
    odom_frame: "odom"
    # exponential low pass gain for localization update (0 to 1)
    #   (higher gain means odometry is less used / relied on)
    update_gain: 0.5
    # time based confidence gain when in 2D / 1D mode
    confidence_gain: 0.01
    # how many particles (samples) to spread (per update)
```

```
sample_rate: 10
# localization update rate [ms]
loc_update_time: 100
# map tile update rate [1/s]
map_update_rate: 0.5
# map tile size in pixels
map_size: 1000
# how often to downscale (half) the original map
map_downscale: 0
# how many 3x3 gaussian smoothing iterations are applied to the map
num_smooth: 5
# minimum score for valid localization (otherwise 0D mode)
#    higher values make it go into 0D mode earlier
min_score: 0.2
# odometry error in x and y [m/m] [1]
#    how fast to increase particle spread when in 1D / 0D mode
odometry_std_xy: 0.01
# odometry error in yaw angle [rad/rad] [1]
#  how fast to increase particle spread when in 0D mode
odometry_std_yaw: 0.01
# minimum particle spread in x and y [m]
min_sample_std_xy: 0.025
# minimum particle spread in yaw angle [rad]
min_sample_std_yaw: 0.025
# initial/maximum particle spread in x and y [m]
max_sample_std_xy: 0.5
# initial/maximum particle spread in yaw angle [rad]
max_sample_std_yaw: 0.5
# threshold for 1D / 2D decision making (minimum average second order gradient)
# if worst gradient direction is below this value we go into 1D mode
# if both gradient directions are below we may go into 0D mode, depending on disable_threshold
# higher values will make it go into 1D / 0D mode earlier
constrain_threshold: 0.1
# threshold for 1D / 2D decision making (with or without orientation)
#   higher values will make it go into 1D mode earlier
```

```
constrain_threshold_yaw: 0.2
# minimum number of points per update
min_points: 20
# solver update gain, lower gain = more stability / slower convergence
solver_gain: 0.1
# solver update damping, higher damping = more stability / slower convergence
solver_damping: 1000.0
# number of gauss-newton iterations per sample per scan
solver_iterations: 20
# maximum wait for getting transforms [s]
transform_timeout: 0.2
# if to broadcast map frame
broadcast_tf: true
# Scan topic
scan_topic: scan
# Initial Pose topic
initialpose: initialpose

# Map Tile topic
map_tile: map_tile

# Map Pose topic
map_pose: map_pose

# particle_cloud topic
particle_cloud: cloud

# amcl_pose topic
amcl_pose: amcl_pose
```

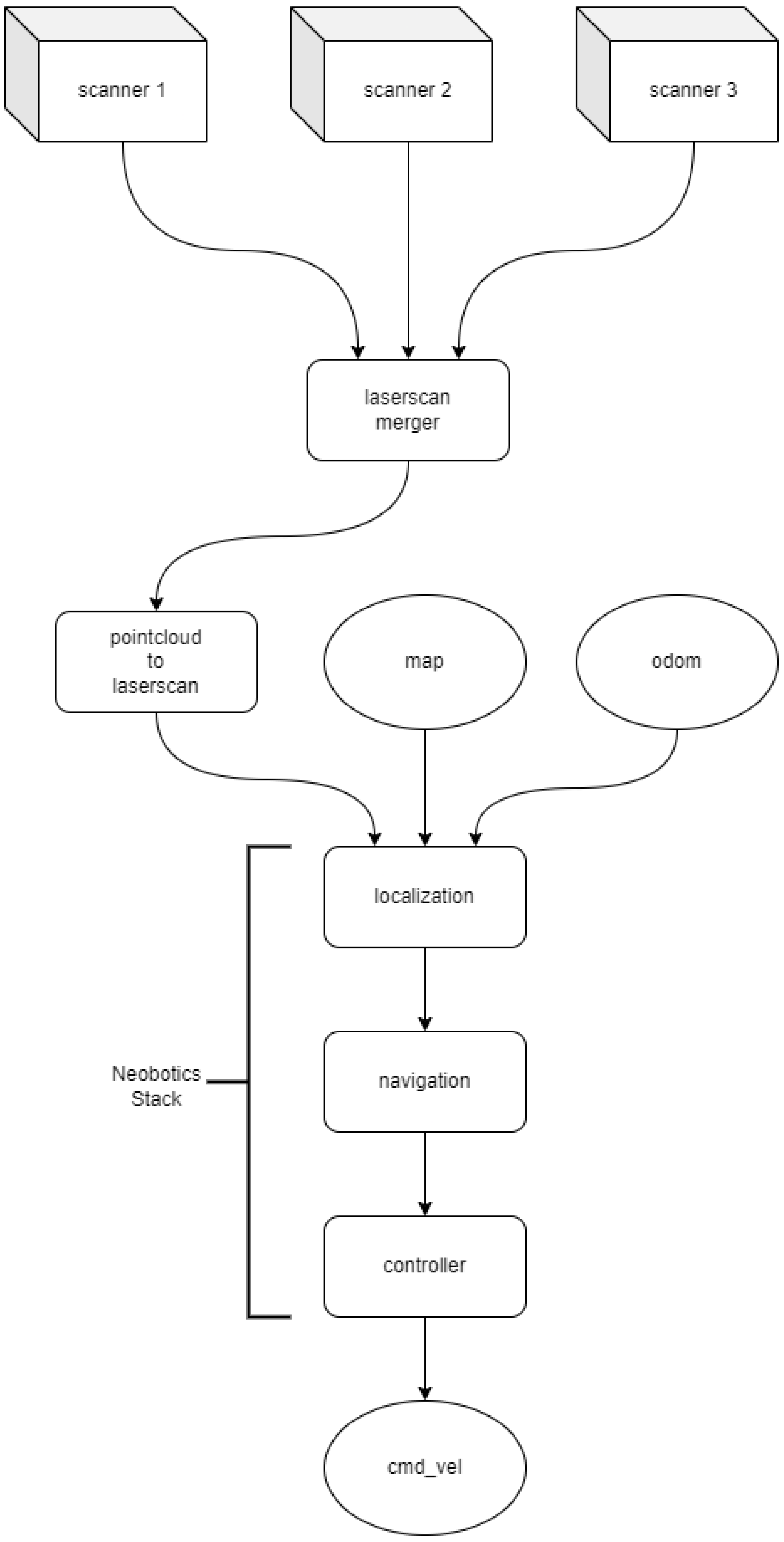


Figure 20: Complete navigation stack

4.4 Near-target docking methods

The problem behind docking a rechargeable vehicle is not new and has seen many applications in different fields. There are several ways that have been explored in order to autonomously connect two bodies for power transmission, in this chapter we will only take into consideration ones where a moving platform approaches a stationary target without the need for any kind of adjustment done by the latter or any uncontrolled collision used as alignment factor (such as mechanical guides).

The most utilized techniques can be grouped into four separate approaches: geometry detection, reflective markers, emitter detection and optical detection.

Geometry detection: in this case the automated vehicle (or robot) comes equipped with lidars through which it can see a particular geometric feature on the docking station's body. From this feature it can define a precise orientation and distance and then it computes the correct placement of dock connections. An example of this technique can be seen for the Mir Robot charging station. Despite being a simple approach, it lacks precision and flexibility, as the geometric feature must be univocal and could be "hallucinated" by the localization software in other places inside the working environment. [23]

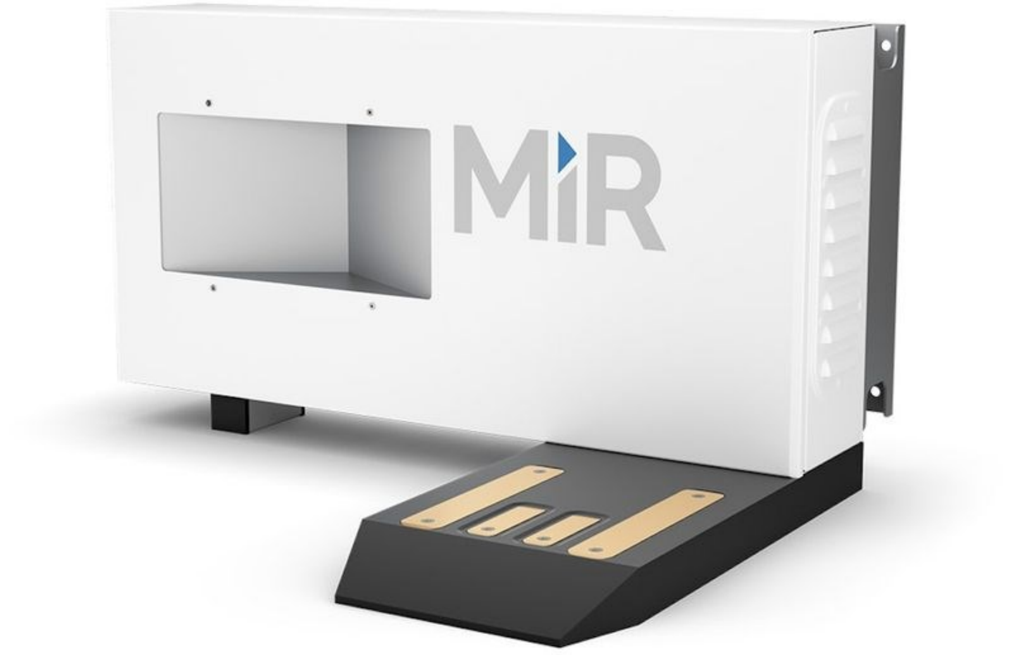


Figure 21: Mir charging station, notice the triangular indentation.

Reflective markers: while still using lidars, this approach involves using retro-reflective tape strips used as landmarks by the navigation software equipped by the AGV. The onboard sensors measure both the angle of the reflective surface with respect to the lidar and the intensity of returned light. Since these markers are placed on rigid locations like walls or columns and reflect at a wide angle of incidence, they can be trusted for a more precise long and short-range positioning, especially when present in substantial numbers. Still, the position of the strips must be inputted in the software manually or mapped and, especially in the case of a big environment like a warehouse, this can be a tedious and time-consuming process, also prone to user error. [24] [25]

Emitter detection: this approach is based on detection of low-power infrared emitters placed on the charging station. When in need to dock, the robot starts rotating to seek the emitter's light; after detecting both emitted beams, the robot can then center itself in a direction perpendicular to the charging station by controlling its position until the two emitted beams have the same intensity. Once the robot is perpendicular to the dock, it proceeds forward until a preset intensity corresponding to the goal is reached. While being cost effective and quite precise, this approach entails adding additional emitters to the docking station, and specialized sensors on the robot body. Even not considering the added cost and lack of flexibility given by not being able to reuse already present sensors, this approach is not feasible also because it is regulated by a patent filed by the iRobot company. [26]

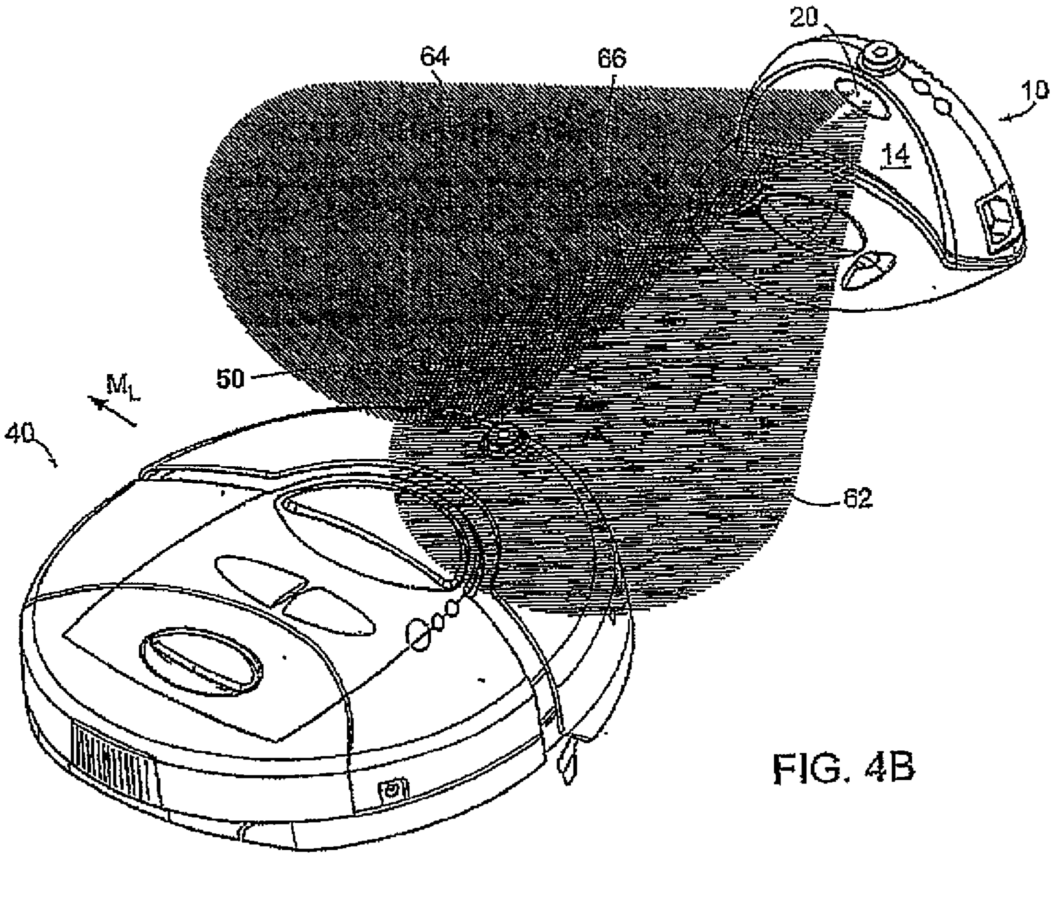


Figure 22: iRobot ir emitter apprach

Optical detection: Optical detection methods use cameras to detect some predetermined and easily recognizable features through computer vision to define a particular space orientation. For this to work the cameras must be properly calibrated to rely accurately on real world coordinates.

The camera will be able to define a coordinate of the target by a previously set of features, for this to be feasible, it is necessary to use clearly visible and dimensionally sound markers, such as Aruco markers.

4.5 Aruco approach

Aruco [46] [47] is a binary square fiducial marker that can be used for camera pose estimation. It is composed of a wide black border and an inner binary matrix that determines its identifier. The black border facilitates its fast detection in the image, and the binary codification allows its identification and the application of error detection and correction techniques. The Aruco module in OpenCV includes the detection of Aruco markers and tools to employ them for pose estimation and camera calibration. In our Particular Case Aruco has been used for pose estimation purposes, through a ROS 2 library that applies OpenCV to generate a six-dimensional position and orientation frame. [48] This frame is then converted by a custom algorithm into a quaternion to be easier to work with as a NAV 2 parameter, and a secondary frame is created with a predefined offset dependent on the Aruco recognition code.

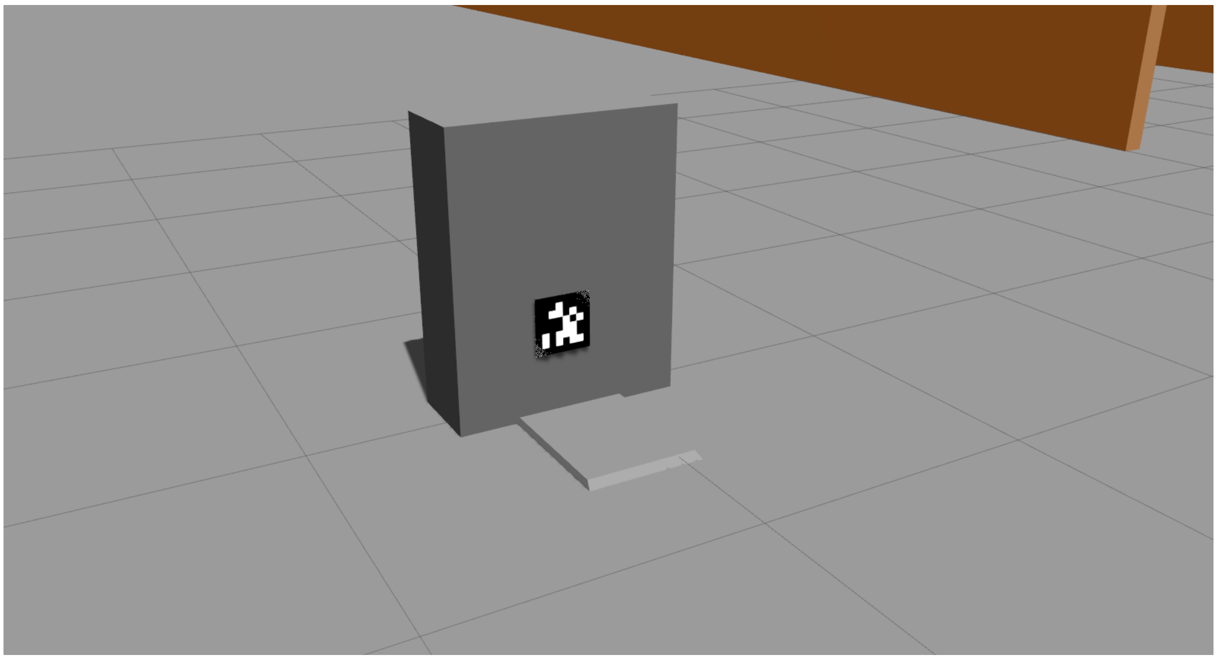
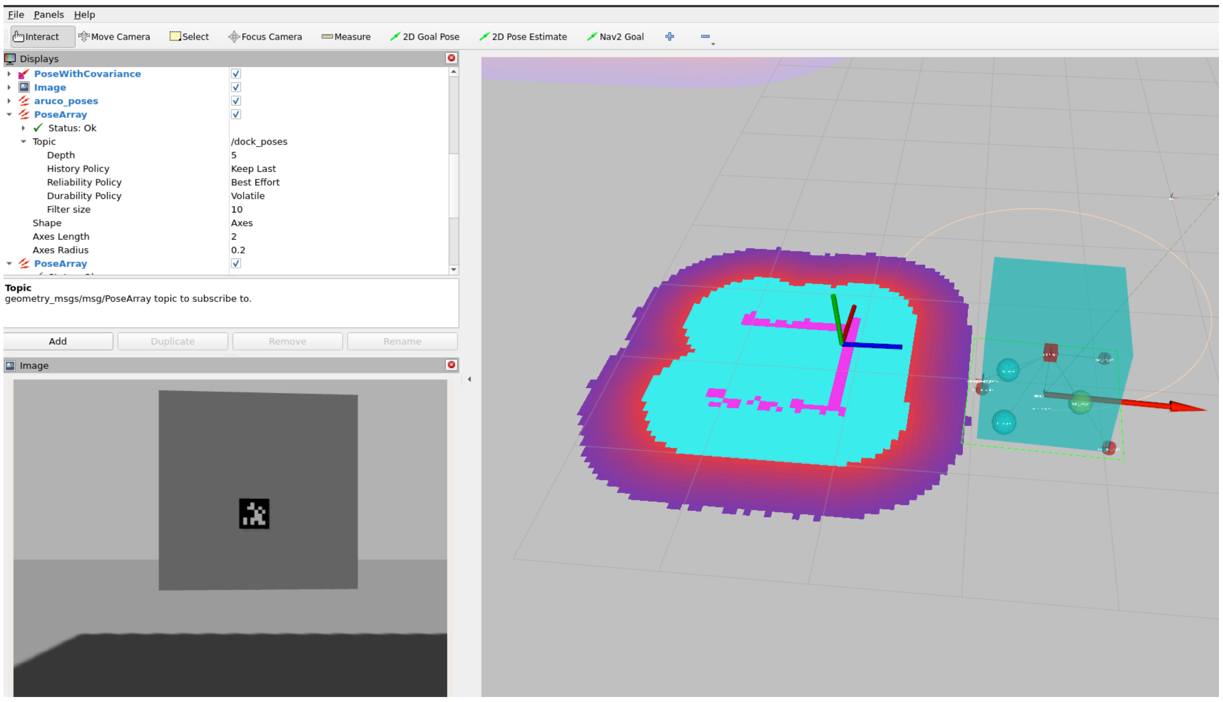


Figure 23: The 3D sketch of a docking station equipped with an Aruco fiducial marker, later a cube was used.

4.5.1 Ros2_Aruco

In order to define a position and orientation pair from a univocal Aruco code we used a plugin based on the OpenCV library capable of detecting the fiducial markers from a ROS-enabled camera. The Ros2_Aruco [49] library generates a position vector message of the type:

```
std_msgs/Header header


int64[] marker_ids

geometry_msgs/Pose[] poses
```



```
PoseWithCovariance        ✓
Image                     ✓
aruco_poses               ✓
PoseArray                 ✓
  Status: Ok
  Topic                   /dock_poses
    Depth                 5
    History Policy        Keep Last
    Reliability Policy    Best Effort
    Durability Policy     Volatile
    Filter size           10
  Shape                   Axes
  Axes Length             2
  Axes Radius             0.2
PoseArray                 ✓
```

Topic
geometry_msgs/msg/PoseArray topic to subscribe to.

Figure 24: Aruco node detecting the fiducial marker attached on the dock placeholder.

To detect markers from the AMR camera, the ros2_aruco node subscribes to the image_topic published by the simulated AMR. Usually, it would be necessary to calibrate the camera in the real case to correctly identify the camera parameters, but being it defined a priori based on preferred settings, we can skip this step as it would not improve the positional precision of detected fiducial markers. It is fair to note that the ros2_aruco node offers the possibility to calibrate cameras directly from its library using a simple checkerboard pattern attached to a rigid surface.

The following piece of code is the one responsible for retrieving image data from the camera and publishing the detected markers positions on a specified topic, as we can see openCV is used to detect corners of the marker and then a set of transformations is applied to recover the deformation of the detected square:

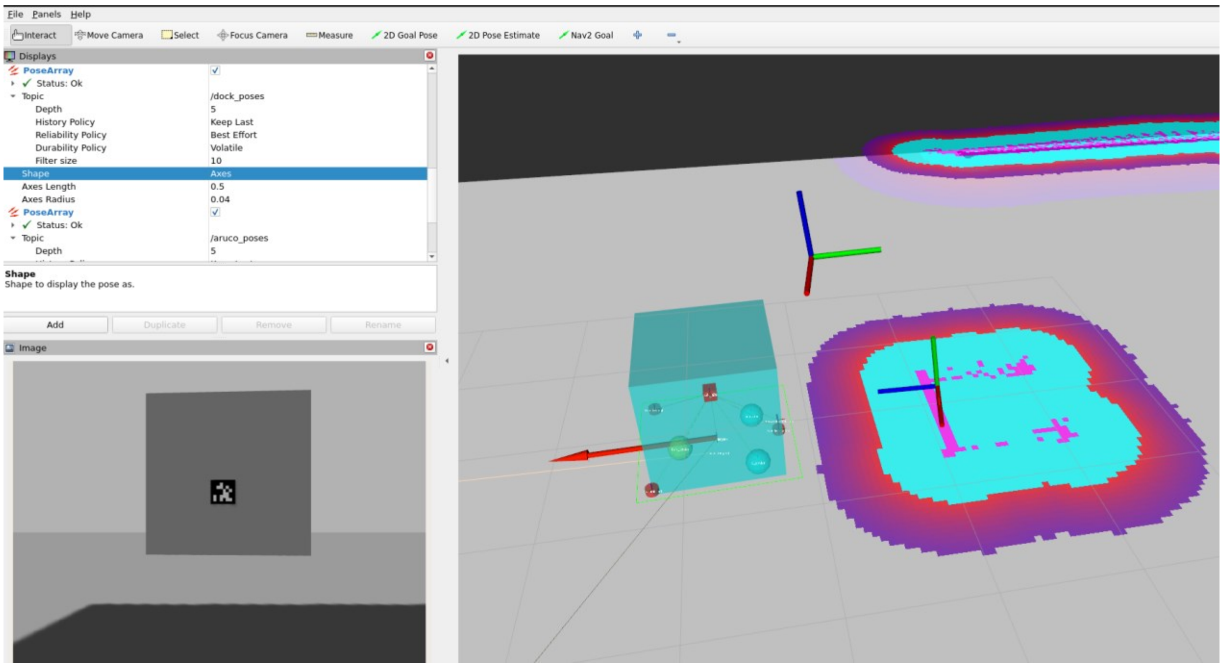
```
def image_callback(self, img_msg):


    if self.info_msg is None:
        self.get_logger().warn("No camera info has been received!")
        return


    cv_image = self.bridge.imgmsg_to_cv2(img_msg,
                                 desired_encoding='mono8')
    markers = ArucoMarkers()
    pose_array = PoseArray()
    if self.camera_frame is None:
        markers.header.frame_id = self.info_msg.header.frame_id
        pose_array.header.frame_id = self.info_msg.header.frame_id
    else:
        markers.header.frame_id = self.camera_frame
        pose_array.header.frame_id = self.camera_frame



    markers.header.stamp = img_msg.header.stamp
    pose_array.header.stamp = img_msg.header.stamp


    corners, marker_ids, rejected = cv2.aruco.detectMarkers(cv_image,
                                          self.aruco_dictionary,
                                          parameters=self.aruco_parameters)
    if marker_ids is not None:


        if cv2.__version__ > '4.0.0':
            rvecs, tvecs, _ = cv2.aruco.estimatePoseSingleMarkers(corners,
                                              self.marker_size, self.intrinsic_mat,
                                              self.distortion)
        else:
            rvecs, tvecs = cv2.aruco.estimatePoseSingleMarkers(corners,
                                              self.marker_size, self.intrinsic_mat,
                                              self.distortion)
        for i, marker_id in enumerate(marker_ids):
            pose = Pose()
            pose.position.x = tvecs[i][0][0]
            pose.position.y = tvecs[i][0][1]
            pose.position.z = tvecs[i][0][2]

            rot_matrix = np.eye(4)
            rot_matrix[0:3, 0:3] = cv2.Rodrigues(np.array(rvecs[i][0]))[0]
            quat = transformations.quaternion_from_matrix(rot_matrix)

            pose.orientation.x = quat[0]
            pose.orientation.y = quat[1]
            pose.orientation.z = quat[2]
            pose.orientation.w = quat[3]

            pose_array.poses.append(pose)
            markers.poses.append(pose)
            markers.marker_ids.append(marker_id[0])

        self.poses_pub.publish(pose_array)
        self.markers_pub.publish(markers)
```

4.5.2 Docking_node

After obtaining the position of the Aruco code, the docking node translates the orientation part of the vector from a quaternion into roll, pitch and yaw angles. After obtaining the RPY coordinates it then computes a predefined transformation between the Aruco coordinates and the actual docking position. Having obtained the transformation, the orientation is translated back into a quaternion and published as docking coordinates. Transferring the rotation coordinates from a quaternion to Euler angles and then back is done to manage changes to the code faster and with less errors.

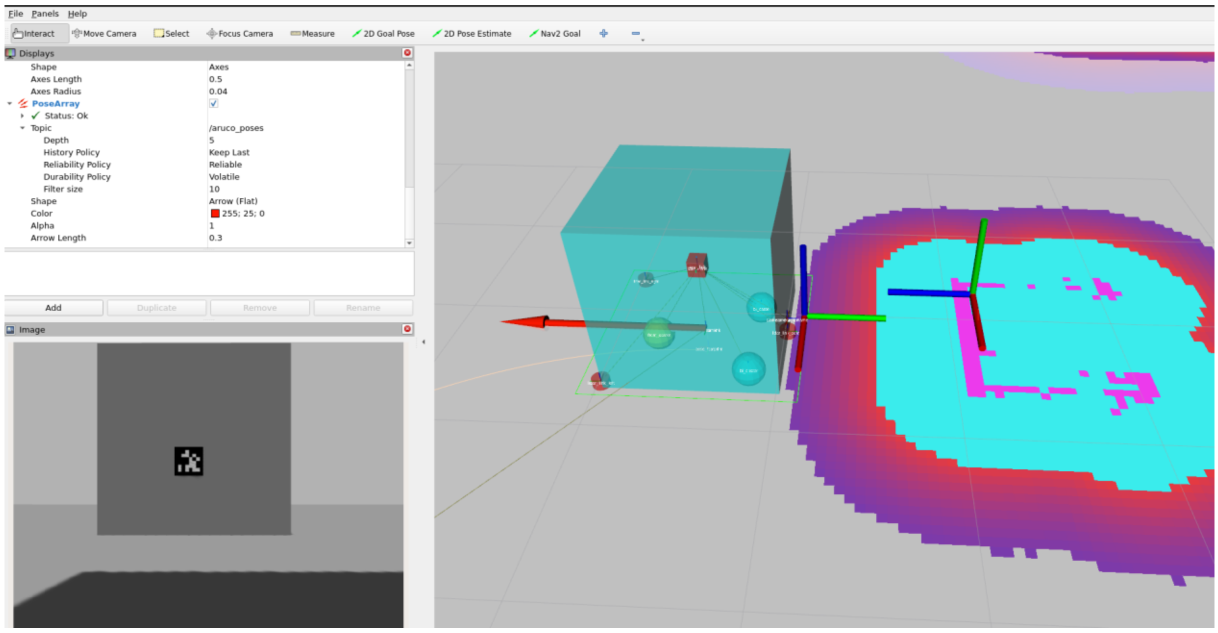
```
def marker_callback(self, marker_msg):

  self.marker_msg = marker_msg
  self.marker_poses = self.marker_msg.poses
  self.marker_ids = self.marker_msg.marker_ids

  docks = ArucoMarkers()
  docks_array  = PoseArray()

  docks.header.frame_id = self.camera_frame
  docks_array.header.frame_id = self.camera_frame
  if self.marker_ids is not None:
    for i, marker_id in enumerate(self.marker_ids):
      pose = Pose()
      mark = self.markers_transforms
      pose.position.x = self.marker_poses[i].position.x + mark[7*marker_id]
      pose.position.y = self.marker_poses[i].position.y + mark[7*marker_id + 1]
      pose.position.z = self.marker_poses[i].position.z + mark[7*marker_id + 2]
      quaternion1 = np.array((0,0,0,0), dtype=np.float64)
      quaternion1[0] = self.marker_poses[i].orientation.x
      quaternion1[1] = self.marker_poses[i].orientation.y
      quaternion1[2] = self.marker_poses[i].orientation.z
      quaternion1[3] = self.marker_poses[i].orientation.w
      quaternion0 = mark[7*marker_id+3 : 7*marker_id+7]
```

```
        final_orientation = transformations.quaternion_multiply(quaternion1, quaternion0)
        pose.orientation.x = final_orientation[0]
        pose.orientation.y = final_orientation[1]
        pose.orientation.z = final_orientation[2]
        pose.orientation.w = final_orientation[3]

        docks_array.poses.append(pose)
        docks.poses.append(pose)
        docks.marker_ids.append(marker_id)

    # self.get_logger().info("aruco node %d" %(docks.marker_ids[0]))
    self.dock_poses_pub.publish(docks_array)
    self.docks_pub.publish(docks)
```


File Panels Help
Interact Move Camera Select Focus Camera Measure 2D Goal Pose 2D Pose Estimate Nav2 Goal
Displays
Add Duplicate Remove Rename
Image

```
PoseArray              ✓
  Status: Ok
  Topic                /dock_poses
    Depth              5
    History Policy     Keep Last
    Reliability Policy Best Effort
    Durability Policy  Volatile
    Filter size        10
  Shape                Axes
  Axes Length          0.5
  Axes Radius          0.04
PoseArray              ✓
  Status: Ok
  Topic                /aruco_poses
    Depth              5

Shape
Shape to display the pose as.
```

Figure 26: Generated dock position, at this stage, we were considering wireless charging from a plate placed on thop of the robot


File Panels Help
Interact Move Camera Select Focus Camera Measure 2D Goal Pose 2D Pose Estimate Nav2 Goal
Displays
Add Duplicate Remove Rename
Image

```
  Shape                Axes
  Axes Length          0.5
  Axes Radius          0.04
PoseArray              ✓
  Status: Ok
  Topic                /aruco_poses
    Depth              5
    History Policy     Keep Last
    Reliability Policy Reliable
    Durability Policy  Volatile
    Filter size        10
  Shape                Arrow (Flat)
  Color                255; 25; 0
  Alpha                1
  Arrow Length         0.3
```

Figure 25: Second configuration with a docking area placed on the ground, to change placement it was only necessary to change the transform vector

4.6 Planar controller

In this section the action responsible for steering the robot omnidirectionally to a set position is shown. This action is based on a modified pure pursuit controller [50] and uses a proportional derivative control to determine the velocities sent to the robot wheels. The action does not consider possible obstacles, this is done on purpose, since the standard navigation stack considers the docking station itself as an obstacle, generates an inflation map from the detected points and prevents the robot from reaching the contact point if the latter is in the restricted area. In fact, with both the Neobotics and Nav2 stack, the robot stops moving and goes into error mode as soon as part of the collision area enters the inflation map.

The following is the piece of code implementing the planar controller:

```
class PlanarController():
  def __init__(self, linear_k, angular_k, linear_max, angular_max):
    self.linear_k = linear_k
    self.angular_k = angular_k
    self.linear_max = linear_max
    self.angular_max = angular_max

  def clip(self, val, min_, max_):
    return min_ if val < min_ else max_ if val > max_ else val

  def compute_error(self, curr_robot_pose,  curr_goal_pose):


    robot_quat_exp = [curr_robot_pose.pose.orientation.w, curr_robot_pose.pose.orientation.x,
            curr_robot_pose.pose.orientation.y, curr_robot_pose.pose.orientation.z]
    robot_euler = quat2euler(robot_quat_exp)

    goal_quat_exp = [curr_goal_pose.orientation.w, curr_goal_pose.orientation.x,
            curr_goal_pose.orientation.y,  curr_goal_pose.orientation.z]
    goal_euler = quat2euler(goal_quat_exp)

    robot_roll, robot_pitch, robot_yaw = robot_euler[0], robot_euler[1], robot_euler[2]
    goal_roll, goal_pitch, goal_yaw = goal_euler[0], goal_euler[1], goal_euler[2]
```

```
        err_local = [- curr_goal_pose.position.x ,
                  -  curr_goal_pose.position.,
                  robot_yaw - goal_yaw]




        dist_x_error = err_local[0]
        dist_y_error = err_local[1]
        rot_error =  err_local[2]
        return dist_x_error, dist_y_error, rot_error

    def    compute_velocities(self,    curr_robot_pose,        curr_goal_pose,    rot_to_goal_satisfied,
dist_x_to_goal_satisfied, dist_y_to_goal_satisfied):
        robot_quat_exp = [curr_robot_pose.pose.orientation.w, curr_robot_pose.pose.orientation.x,
                    curr_robot_pose.pose.orientation.y, curr_robot_pose.pose.orientation.z]
        robot_euler = quat2euler(robot_quat_exp)
        goal_quat_exp = [curr_goal_pose.orientation.w, curr_goal_pose.orientation.x,
                    curr_goal_pose.orientation.y,  curr_goal_pose.orientation.z]
        goal_euler = quat2euler(goal_quat_exp)

        robot_roll, robot_pitch, robot_yaw = robot_euler[0], robot_euler[1], robot_euler[2]
        goal_roll, goal_pitch, goal_yaw = goal_euler[0], goal_euler[1], goal_euler[2]

        err_local = [- curr_goal_pose.position.z,
                  - curr_goal_pose.position.x,
                  goal_yaw - robot_yaw]

        k1 = self.linear_k
        k2 = self.angular_k
        max_v = self.linear_max
        max_w = self.angular_max



        if not (dist_x_to_goal_satisfied) and rot_to_goal_satisfied:
```

```
        v_x_in = k1 * err_local[0]
    else:
        v_x_in = 0.0

    if not (dist_y_to_goal_satisfied) and rot_to_goal_satisfied:
        v_y_in = k1 * err_local[1]
    else:
        v_y_in = 0.0

    if not rot_to_goal_satisfied:
        w_in = k2 * err_local[2]
    else:
        w_in = 0.0

    v_x_in = self.clip(v_x_in, -max_v, max_v)
    v_y_in = self.clip(v_y_in, -max_v, max_v)
    w_in = self.clip(w_in, -max_w, max_w)

    return v_x_in, v_y_in, w_in
```

The planar controller is contained in a class for easier implementation and use trough an action server, the class contains two main functions which depend on parameters which can be defined trough a configuration file just like other ROS2 libraries:

The compute error function outputs the difference between the z axis orientation of the robot and the orientation of the marker, it also generates two x and y errors based on the position of the Aruco marker relative to the camera. This is done by obtaining a transformation from the map to the base frame of the AMR (base_footprint in our case) and then comparing the obtained coordinates and quaternion with the Pose message received from the /dock_poses topic, this function is used to check if the AMR has reached the target.

The compute_velocities function takes computes again the errors, in this way it can work without the need to call the error function, then uses the computed errors multiplied by a gain parameter (defined in the configuration file) to define the velocities to send to the cmd_vel node.

The planar controller is called by the planar move action server, which is started by calling an action from the command line with the name of the desired topic containing the dock poses. The following code represents the most important part of the planar move action server:

```
def execute_callback(self, goal_handle: ServerGoalHandle):

    # self.subscription.destroy()

    topic_name = goal_handle.request.topic_name

    # Subscribe to the specified topic
    self.subscription = self.create_subscription(
        PoseArray,
        topic_name,
        self.message_callback,
        1
    )

    # goal_pose = goal_handle.request.goal_pose
    goal_pose = self.goal_pose

    # goal_time = goal_handle.request.time
    self.get_logger().info("Received a goal from client")
    # self.get_logger().info(str(goal_pose.orientation))

    dist_to_goal_satisfied = False
    rot_to_goal_satisfied = False
    dist_x_to_goal_satisfied = False
    dist_y_to_goal_satisfied = False
    rate = self.create_rate(10)

    feedback_msg = PlanarMove.Feedback()
    result = PlanarMove.Result()
    while not (dist_to_goal_satisfied and rot_to_goal_satisfied) and rclpy.ok():


        # goal_pose = goal_handle.request.goal_pose
        # goal_pose = self.goal_pose


        self.get_logger().info("Processing goal")


        curr_robot_pose = self.helpers.get_curr_robot_pose(
            now=self.get_clock().now(),
            logger=self.get_logger(), base_frame=self.base_frame)
        self.get_logger().info(str(curr_robot_pose))


        curr_dist_to_goal = self.helpers.pose_euclidean_dist(
            curr_robot_pose.pose, goal_pose)


        if dist_x_to_goal_satisfied and dist_y_to_goal_satisfied:
            dist_to_goal_satisfied = True


        dist_x_error, dist_y_error, rot_error = self.controller.compute_error(
            curr_robot_pose, goal_pose)


        self.get_logger().info(str(rot_error))


        if not goal_handle.is_active:
            self.get_logger().info('Goal aborted')
            return PlanarMove.Result()


        if goal_handle.is_cancel_requested:
            goal_handle.canceled()
            self.get_logger().info('Goal canceled')
            return PlanarMove.Result()


        if abs(rot_error) < self.rotation_error_tolerance:
            self.get_logger().info(
                "Corrected the heading")
            rot_to_goal_satisfied = True

        if rot_to_goal_satisfied and abs(dist_y_error) < self.dist_error_tolerance:
            self.get_logger().info(
                "We are at y goal now, adjusting to correct heading")
            dist_y_to_goal_satisfied = True

        if rot_to_goal_satisfied and abs(dist_x_error) < self.dist_error_tolerance:
            self.get_logger().info(
                "We are at x goal now, adjusting to correct heading")
            dist_x_to_goal_satisfied = True

        feedback_msg.distance = curr_dist_to_goal
        goal_handle.publish_feedback(feedback_msg)

        if (dist_to_goal_satisfied and rot_to_goal_satisfied):
            goal_handle.succeed()
            result.target_reached = True
            self.get_logger().info("Navigation was a success")
            # destroy subscription for next exec callback

        v_x_in, v_y_in, w_in = self.controller.compute_velocities(
            curr_robot_pose,         goal_pose,         rot_to_goal_satisfied,         dist_x_to_goal_satisfied,
dist_y_to_goal_satisfied)

        # Publish required velocity commands
        computed_velocity = Twist()
        computed_velocity.linear.x = v_x_in
        computed_velocity.linear.y = v_y_in
        computed_velocity.angular.z = w_in
        self.pub.publish(computed_velocity)
        rate.sleep()

    return result
```

```
def message_callback(self, pose_msg):


    self.pose_msg = pose_msg
    # we later will be able to choose between multiple detections by iterating the poses array
    self.goal_pose = self.pose_msg.poses[0]


    # self.get_logger().info(str(self.goal_pose))
```

The execute callback function is called every time an action is requested by the action server. When activated, the callback function generates a subscription to the PoseArray topic specified in the action request. The message callback function is responsible for unpacking the first detected pose contained in the poses array and making it available for the execute callback function. When this happens, the execute callback enters a loop which end only when the goal is reached or when the action is aborted.

Inside the loop, the target reached condition is periodically checked. While the condition is not satisfied, the execute callback keeps updating speeds sent to the cmd_vel with the values obtained from calling the planar controller compute velocities function.

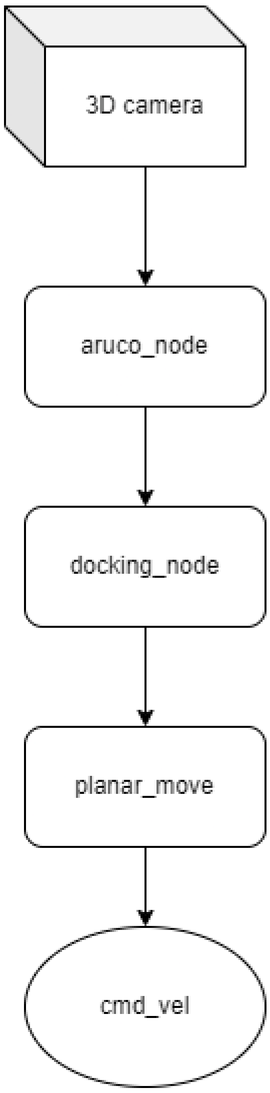


Figure 27: Docking stack with Aruco detection.

4.7 Complete approach and docking stack.

As we have seen in the previous sections, during this thesis we worked on two different separate movement functions.

The first one is based on automatic Monte Carlo localization: it localizes itself in the environment by comparing the data it receives with the data it should see at a variable number of plausible candidate positions. The AMR uses the resulting coordinate to plan a path from the current position to the desired one, the computed path takes into account obstacles using an inflation map and is capable of updating the path if additional obstacles are detected.

The second movement function instead of considering the position of the robot relative to the map, computes the difference between the current robot position and the desired target, using OpenCV to detect a marker that has a known distance from the contact point between AMR and docking station. This is mandatory for a two fold reason: it permits us to detach from the standard navigation stack, which as previously said would prevent us from reaching points close to potential obstacles, and also gives us a fairly precise method of near-object localization that would be difficult to implement precisely with lidars.

Both the movement functions send updates to the cmd_vel node, which is responsible of relaying the actuator speeds (in our case wheel rotation both around z and x axis) to the AMR drivers.

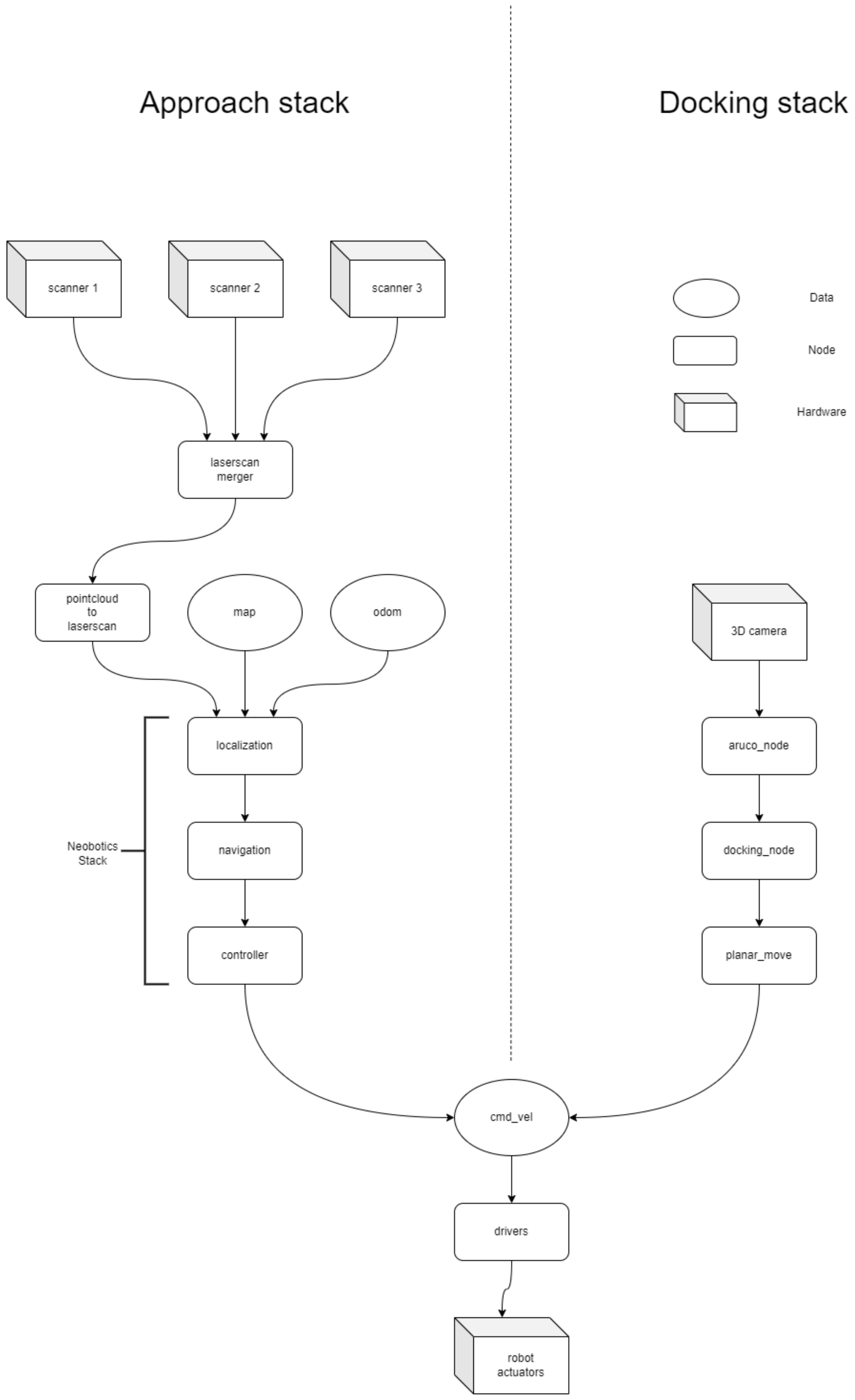


Figure 28: The complete approach and docking stack

4.8 Approach and dock action

In this section, we will describe in detail the complete function that allows for navigation from a random point in space up to the completion of docking has been created as a set of successive actions. Firstly, the robot navigates to a goal that has been set through the action message, then it checks for the Aruco marker in the field of view of the camera. If the fiducial marker is detected, the function generates a separate goal in the global map to which the robot moves, again using standard navigation. Finally, the robot checks again for the Aruco marker and moves forward using a feedback position control comparing the position of the center of the robot with the goal in the 2D map.

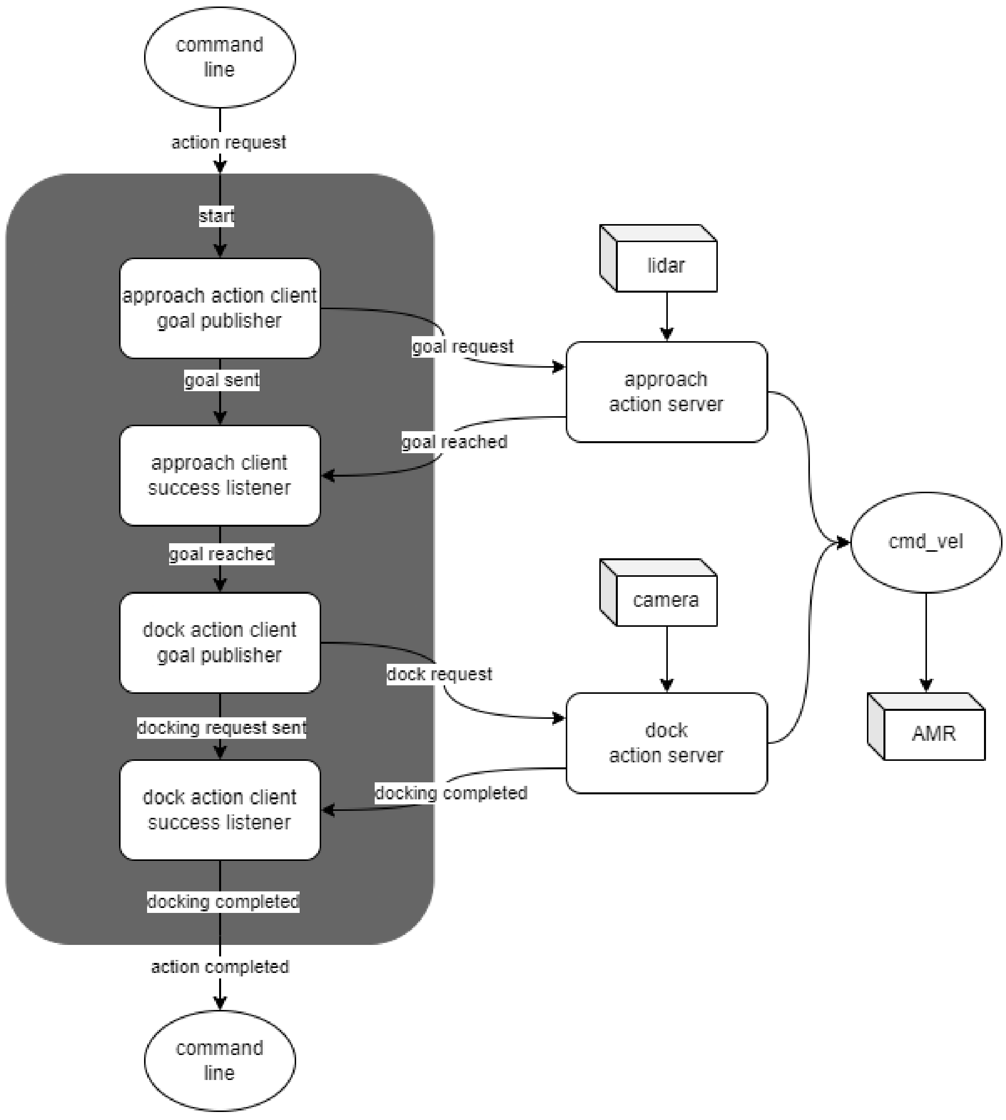


Figure 29: Flow chart of the approach and docking procedure

4.8.1 Initial navigation

Navigation is started sending the action message:

```
$ ros2 action send_goal -f /amr_actions amr_actions/action/Approach "goal_pose: {header: {frame_id: map}, pose: {position: {x: 0.0, y: 0.0, z: 0.0}, orientation:{x: 0.0, y: 0.0, z: 0, w: 1.0000000}}}"
```

To piece together approach and docking, a state machine was used, which states are defined as follows:

```
class ApproachState(Enum):
  READ_POSE = 0
  GO_TO_POSE = 1
  WAIT_REACH_POSE = 2
  GO_TO_DOCK = 3
  TARGET_REACHED = 4
```

After receiving the action message, the AMR navigation stack guides the robot to the specified location by following the generated path through the Neobotics controller. This location is expected to be in the right spot and orientation where an Aruco code is visible in the camera field of view.

```
if self.state == 1:    #GO TO 1st TARGET POSE
        self.navigator.goToPose(self.goal_pose)
        self.state = 2

if self.state == 2:    #WAIT TO REACH TARGET POSE
        self.yaw_error = np.abs(self.currentAngle - self.TargetAngle)
        self.x_error = np.abs(self.currentPose[0] - self.target_offset[0])
        self.y_error = np.abs(self.currentPose[1] - self.target_offset[1])
        if self.navigator.isNavComplete() or (self.x_error < self.diff_x and self.y_error < self.diff_y and
self.yaw_error < self.diff_yaw):
            self.navigator.cancelNav()
            self.yaw_error = np.abs(self.currentAngle - self.TargetAngle)
            self.x_error = np.abs(self.currentPose[0] - self.target_offset[0])
            self.y_error = np.abs(self.currentPose[1] - self.target_offset[1])
            self.get_logger().info("Orientation and 1st offset position reached! \nErrors: {:.3f} {:.3f}
{:.3f}".format(self.x_error, self.y_error, self.yaw_error))
            self.get_logger().info("Actual X: {:.3f}\nActual Y: {:.3f}\nActual Angle:
{:.3f}".format(self.currentPose[0], self.currentPose[1], self.currentAngle))
            time.sleep(5)   #Sleep to let stabilize pose estimation
            self.get_logger().info("Waiting for new Pose Estimation...")
            self.target_received = False
            self.state = 3
```

4.8.2 Position feedback controller

The function then switches the control of the speed and orientation of the robot from the NAV 2 stack to a position feedback controller which increased or decreased forward speeds and angles depending on the positional error between the offset goal and the center of the robot until reaching a small enough limit. The action is called from the action client and waits for a response from the docking action.

```
if self.state == 3:     DOCKING
        self.dockingClient.send_goal("/dock_poses")
        self.state = 4
if self.state == 4:     TARGET REACHED
       if self.dockingClient.reached = True:
        self.dockingClient.send_goal("/dock_poses")
       goal_manager.finished = True
```

When the docking action is completed, the approach and docking action server returns a completed task.

5 RESULTS

In this chapter we will analyze the results obtained during the project. While not strictly connected with the final desired docking action, the intermediate steps were important to ensure the functionality of the AMR.

5.1 Localization precision

This section presents the results of the precision of localization for the autonomous robot in various scenarios. The experiments were conducted to evaluate the performance of the localization system under different conditions and to define its accuracy and robustness.

5.1.1 Experimental Setup

The experiments were conducted in the same simulated environment as the laser merged tests. In this case, while still being equipped with three different lasers, we considered the robot to only have a single scanner placed at the center of the chassis, using the previously defined merging methods.

On top of that an IMU (Inertial Measurement Unit), and a camera were simulated. The localization algorithm used for this study was based on an Extended Kalman Filter (EKF) [51] that fused data from IMU and Odom sensor to better estimate the robot's odometry (position and orientation) in real-time and sent it to two different localization stacks: the one provided from nav2, and the one developed by Neobotics. In addition, the nav2 stack was tested for its response to the use of only the /cloud, /scan or both topics together. This was done to if there was a difference using data piped trough the pointcloud_to_laserscan node. To measure the localization precision, the computed covariance contained in the amcl_pose topic was used.

The position covariance estimates the precision of the localization during the automatic Monte Carlo localization, and as such is a good tool to acquire the localization precision in a simulation. The required value of x, y and z orientation covariances are recorded together with the contents of the amcl_pose messages using the ros2 **bag** [52] function, which generates a database with .db3 extension. Working with a database was not a good option for us, so we opted to convert all data to a text .csv file which made it easier to work with both using excel or a python script. This conversion was applied using another Ros package: **ros2bag_convert** [51] which automatically generates a .csv file from the database ingested.

Finally, from the .csv file, a python script was used to generate a box graph of the recorded covariances.

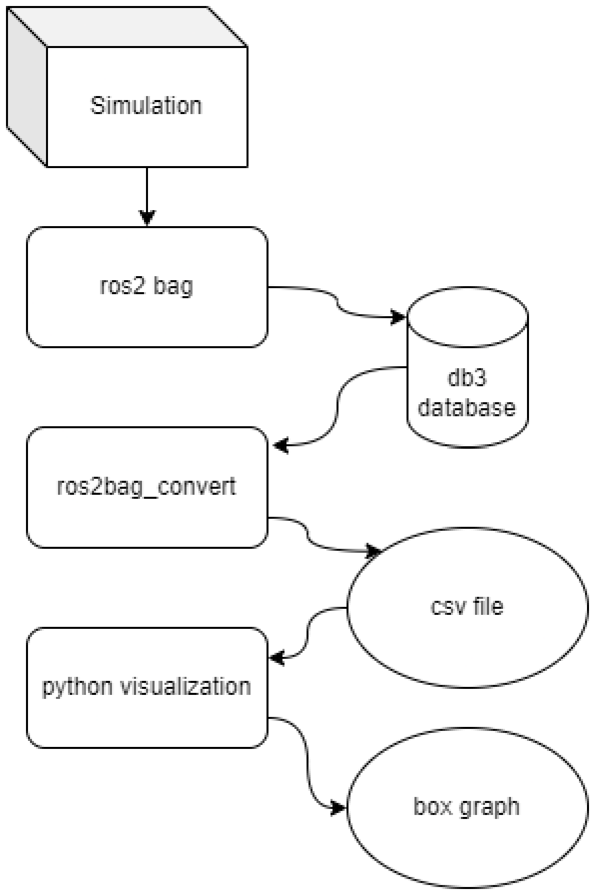


Figure 30: Covariance measurements pipeline

5.1.2 Static Localization

In the first set of experiments, the robot was placed at the center of the benchmark map, in a position from which only a moderate amount of features from the map was visible, the position covariances measured were recorded in a ros2 bag of the amcl_poses and then, as before, passed through the visualizer.py file which parses recorded data and plots the box graphs relative to the variance and covariance of x and y axes, plus the z orientation.

5.1.3 Static Localization Precision Results

In these experiments, the robot's estimated pose closely matched the ground truth values, with average errors in the order of fraction of a centimeter for position and degrees for orientation. These results demonstrate the high precision of static localization.

5.1.4 Dynamic Localization

Dynamic localization tests were conducted to assess the system's performance while the robot was in motion. The robot followed predefined trajectories along different routes that simulated the operation in three different environments: a corridor, a wall, and a feature rich mazelike building. While in motion, the registered covariance was measured and stored in a bag file. The following are the results for the localization precision in motion using:

A) The nav2 stack using only the data coming from the simulated laser scan topic (parallax effect removed)

B) The nav2 stack using only the data coming from the pointcloud where all three scan points are contained

C) The nav2 stack using data coming from both the /cloud and /scan topics

D) The Neobotics stack

All four instances were measured in the three different scenarios and graphed in a box plot.

We start with the measurements in the aisle:

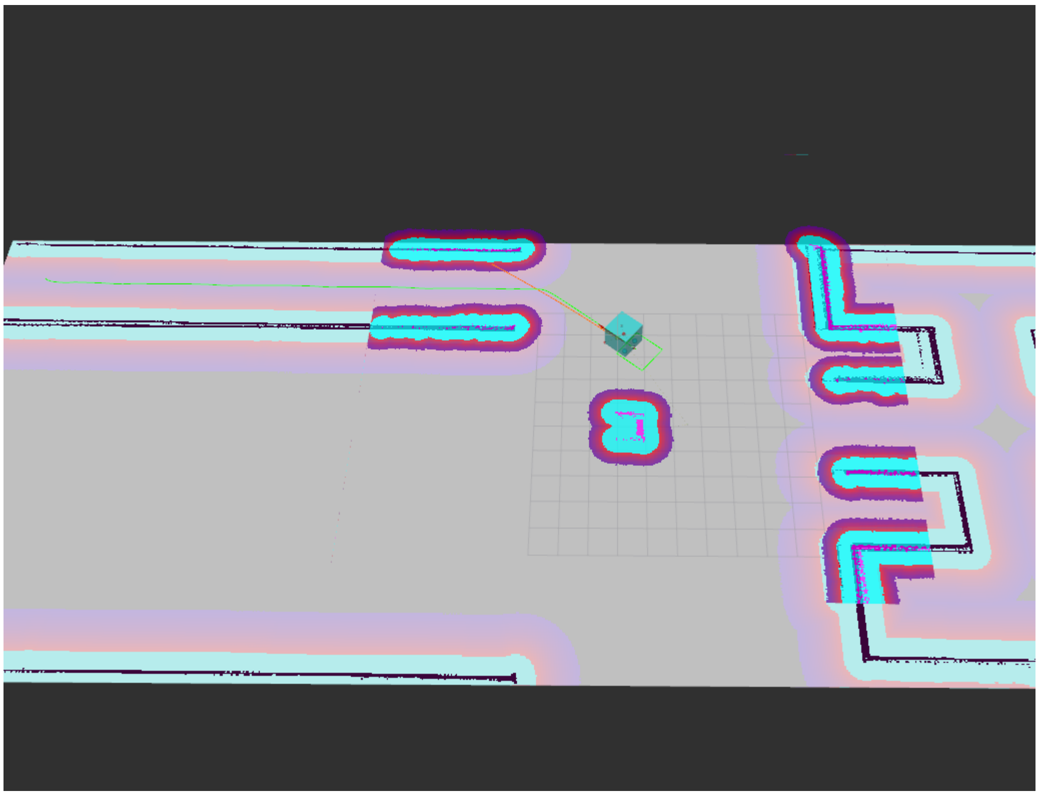


Figure 32: Aisle path on RVIZ

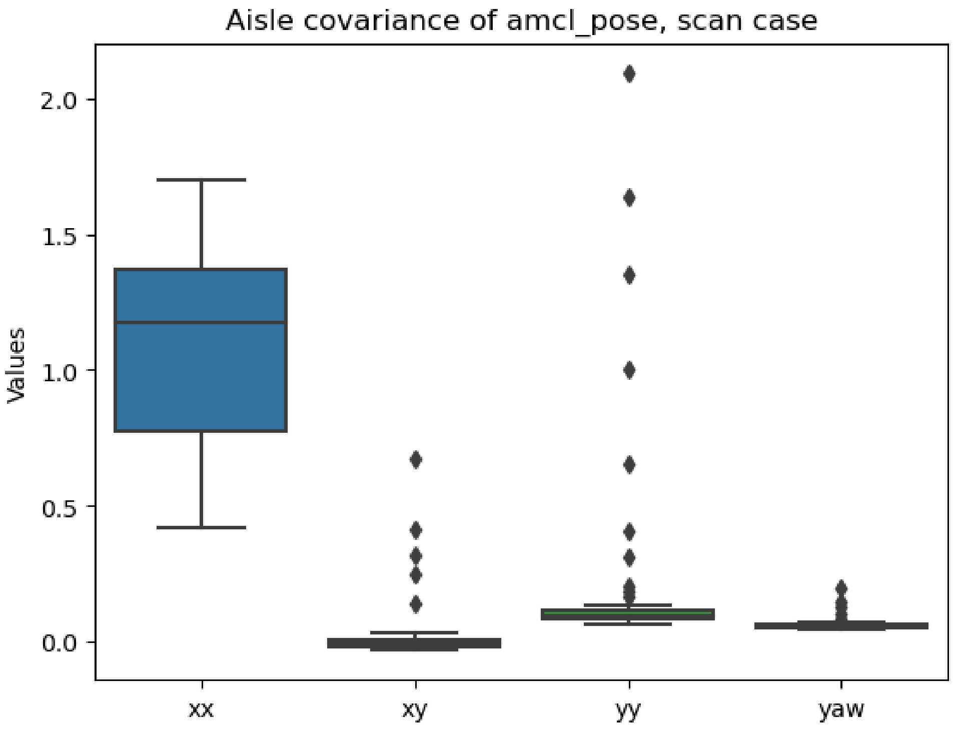
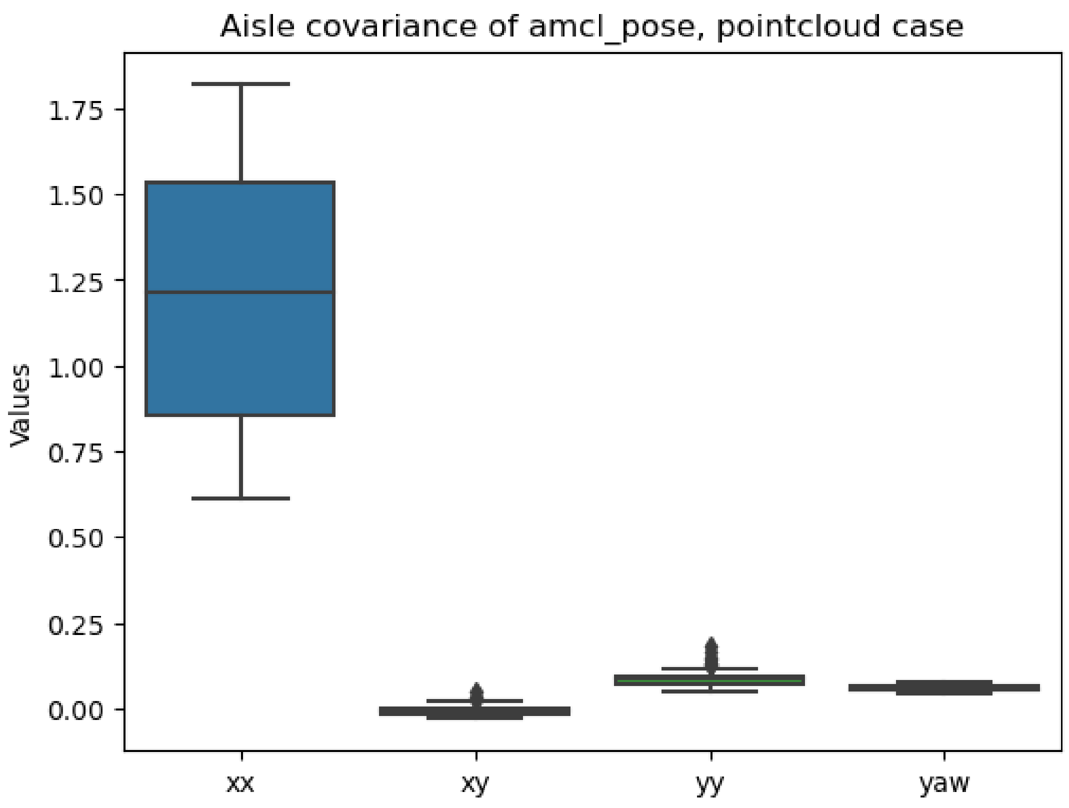


Figure 31: box graph of nav2 aisle covariance using only the LIDAR data.



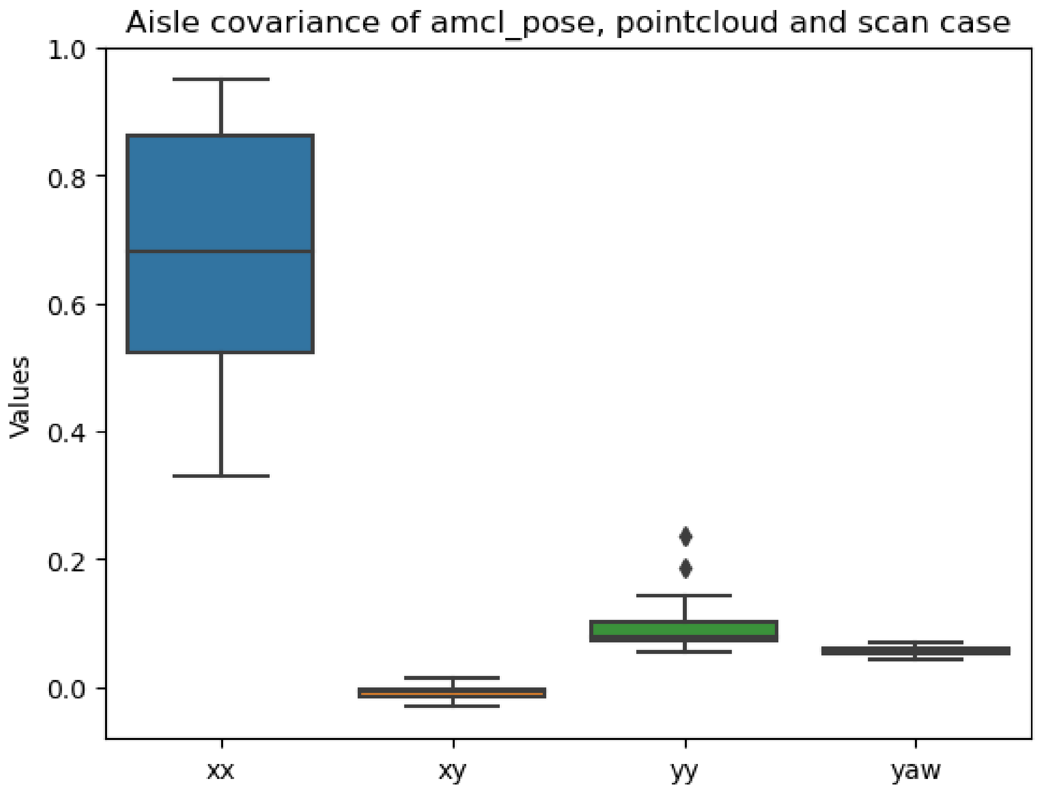


Figure 33:box graph of nav2 aisle covariance using both scan and pointcloud data.

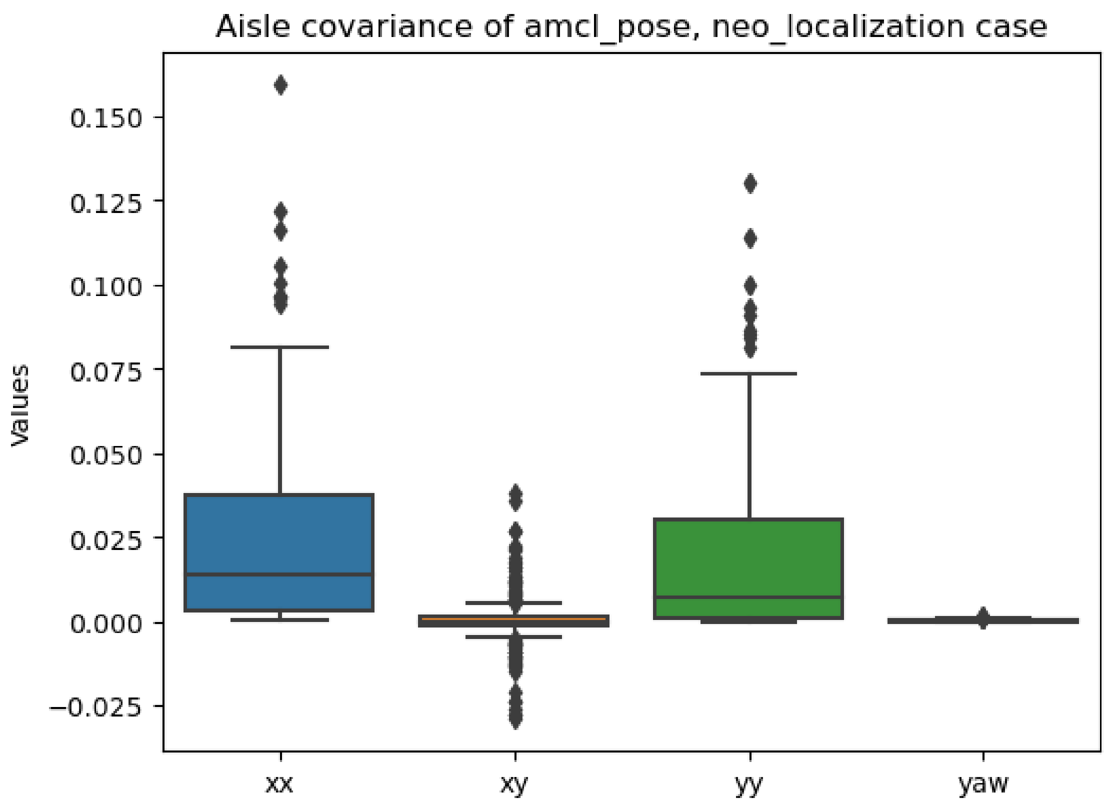


Figure 36: Box graphs of covariance values during navigation trough the aisle with neobotics

Then we proceed with the measurements taken while moving along a wall:

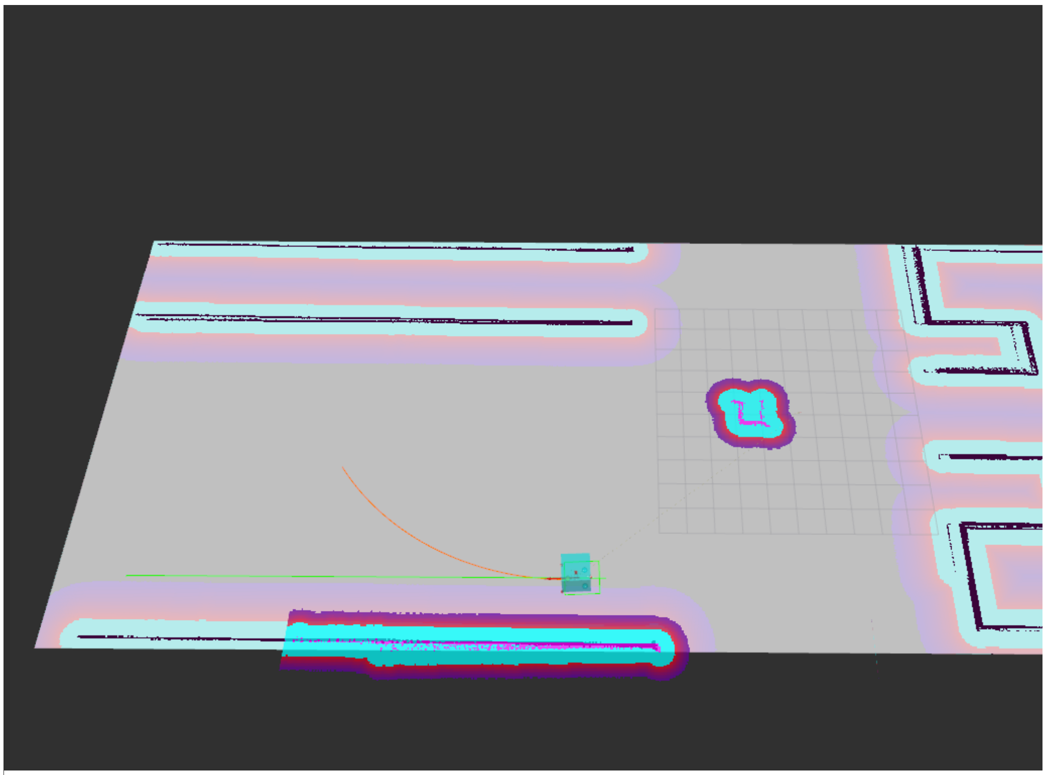


Figure 35: RVIZ visualization of wall path

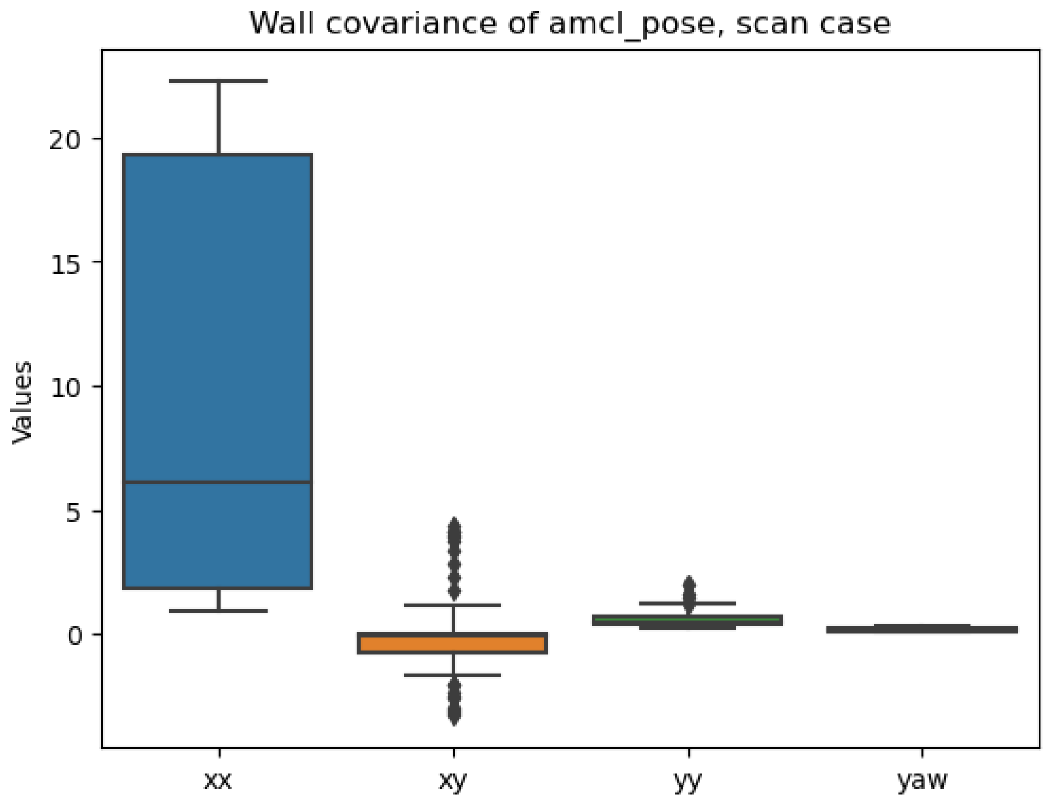


Figure 38: box graph of nav2 wall covariance using only the LIDAR data

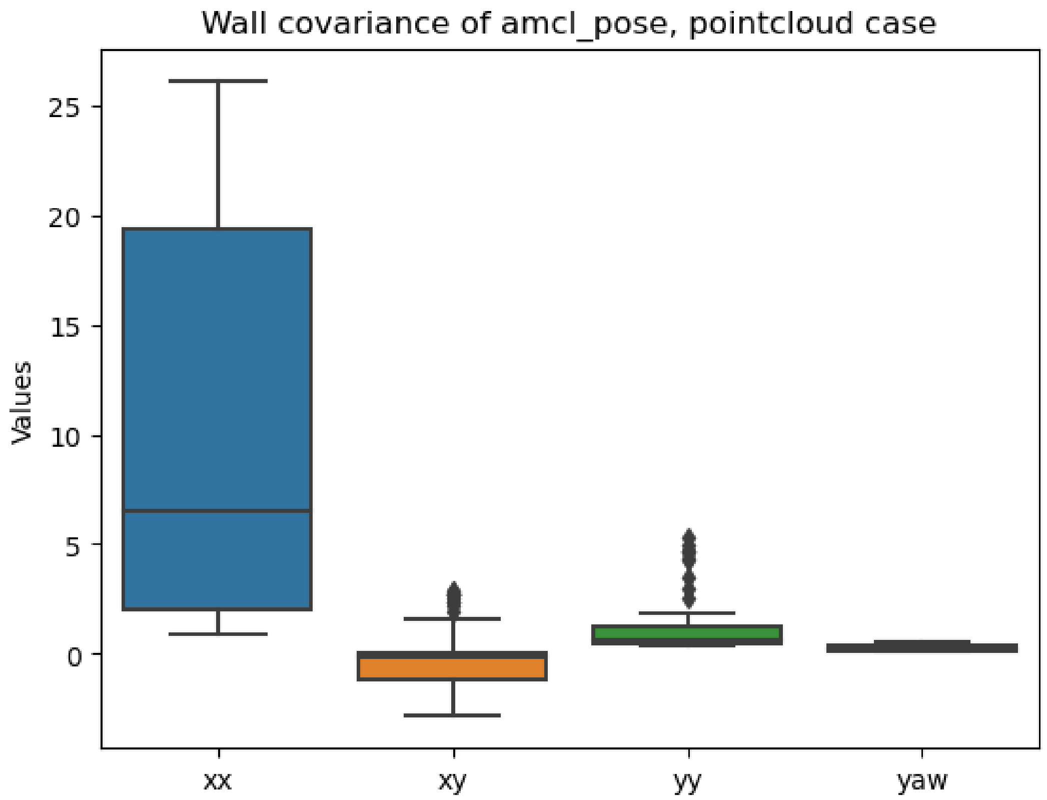


Figure 37box graph of nav2 wall covariance using only the pointcloud data.

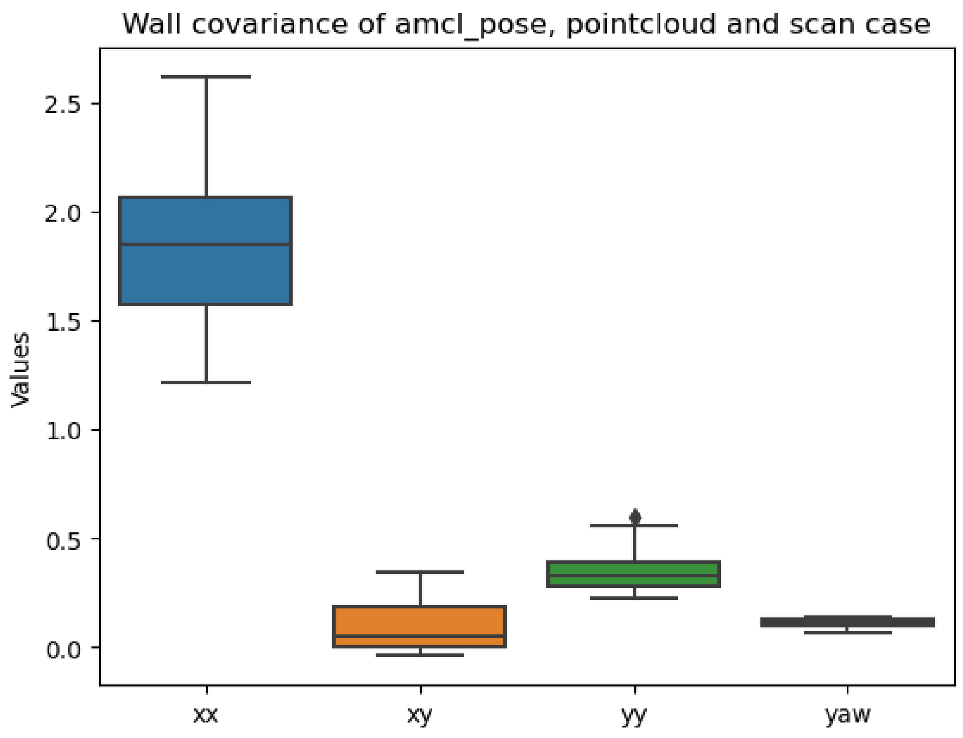


Figure 40:box graph of nav2 aisle covariance using both scan and pointcloud data.

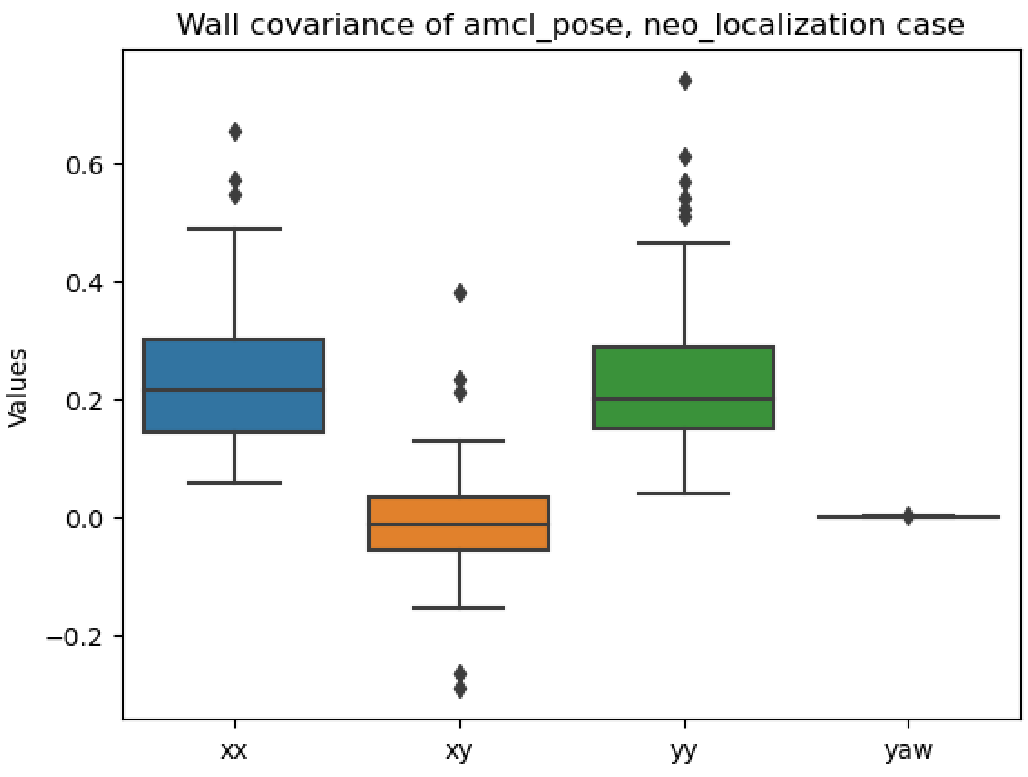


Figure 39: localization covariance along the wall with nav2 or neobotics

Finally, we consider the measurements taken inside the "building", a placeholder for a feature rich environment:

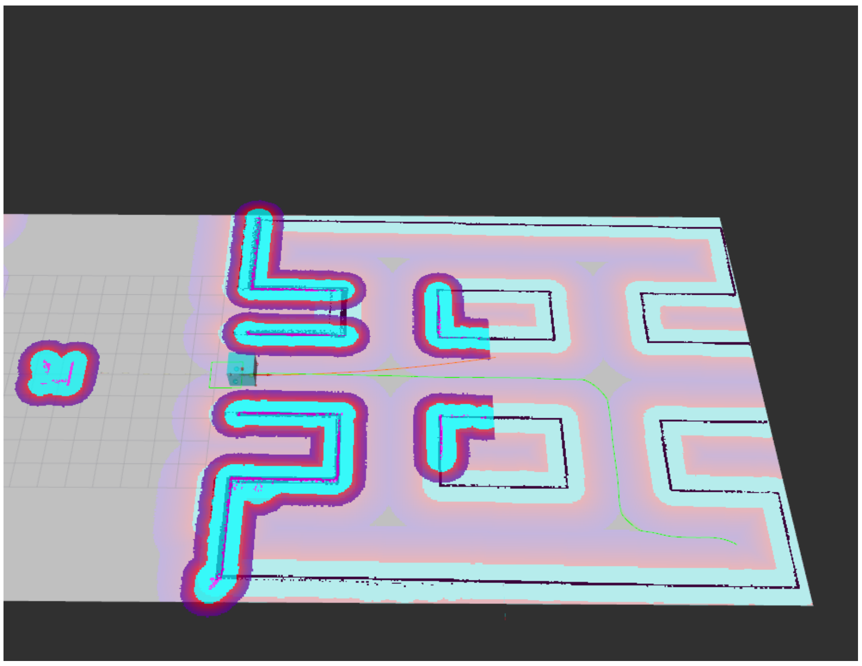


Figure 41:RVIZ visualization of the path inside the building

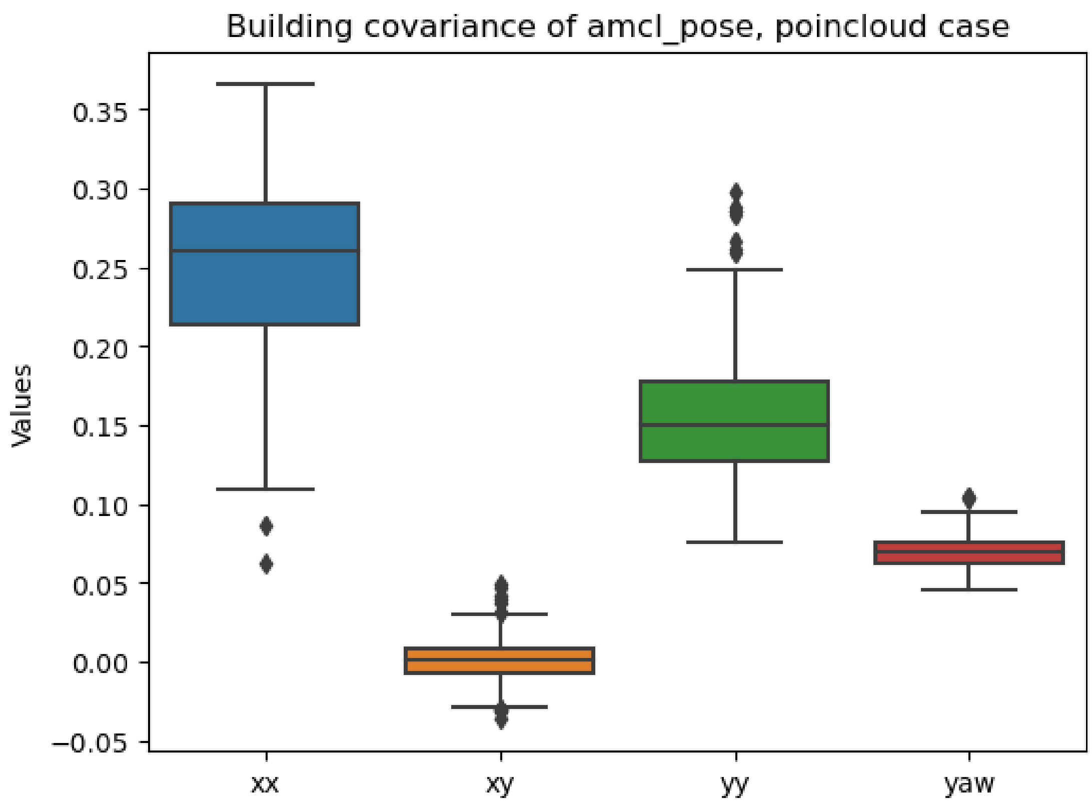


Figure 42box graph of nav2 building covariance using only the pointcloud data.

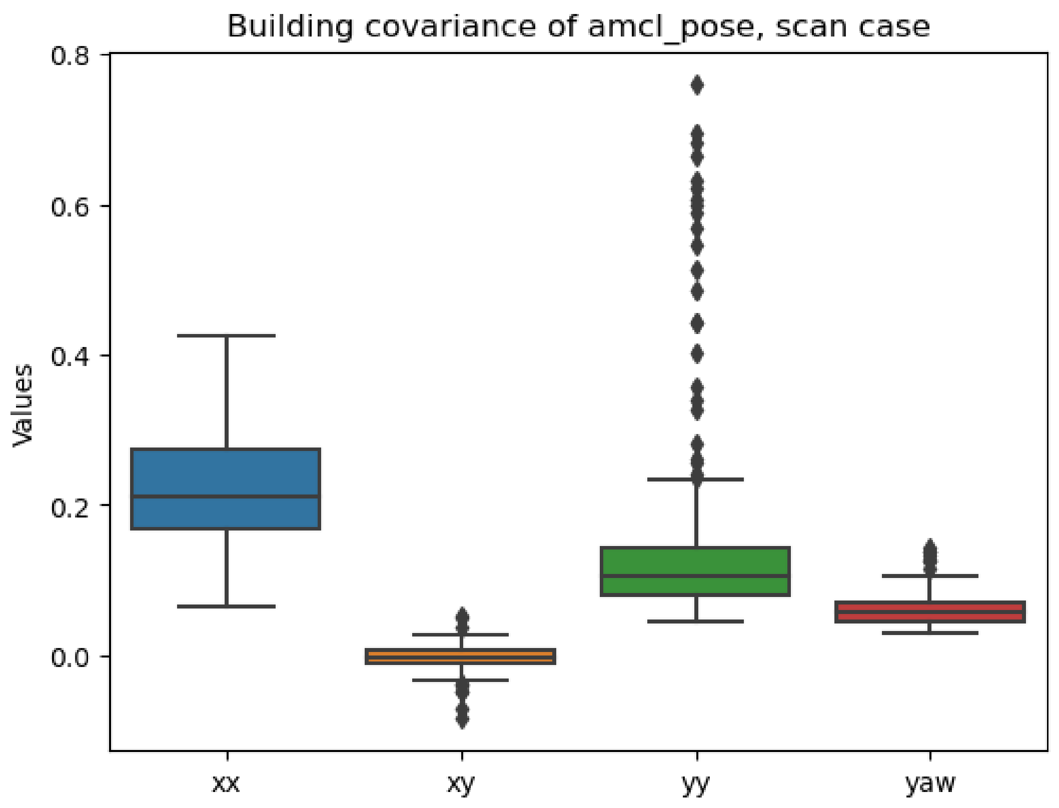


Figure 43: box graph of nav2 building covariance using only the LIDAR data.

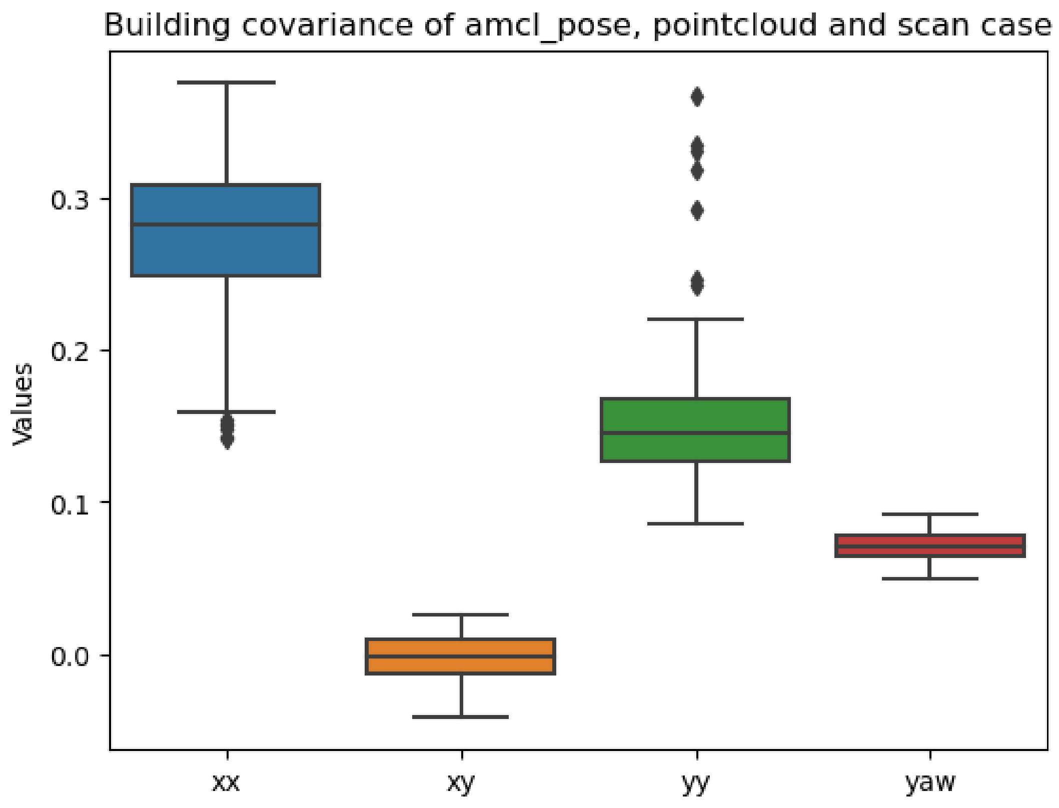


Figure 44:box graph of nav2 aisle covariance using both scan and pointcloud data.

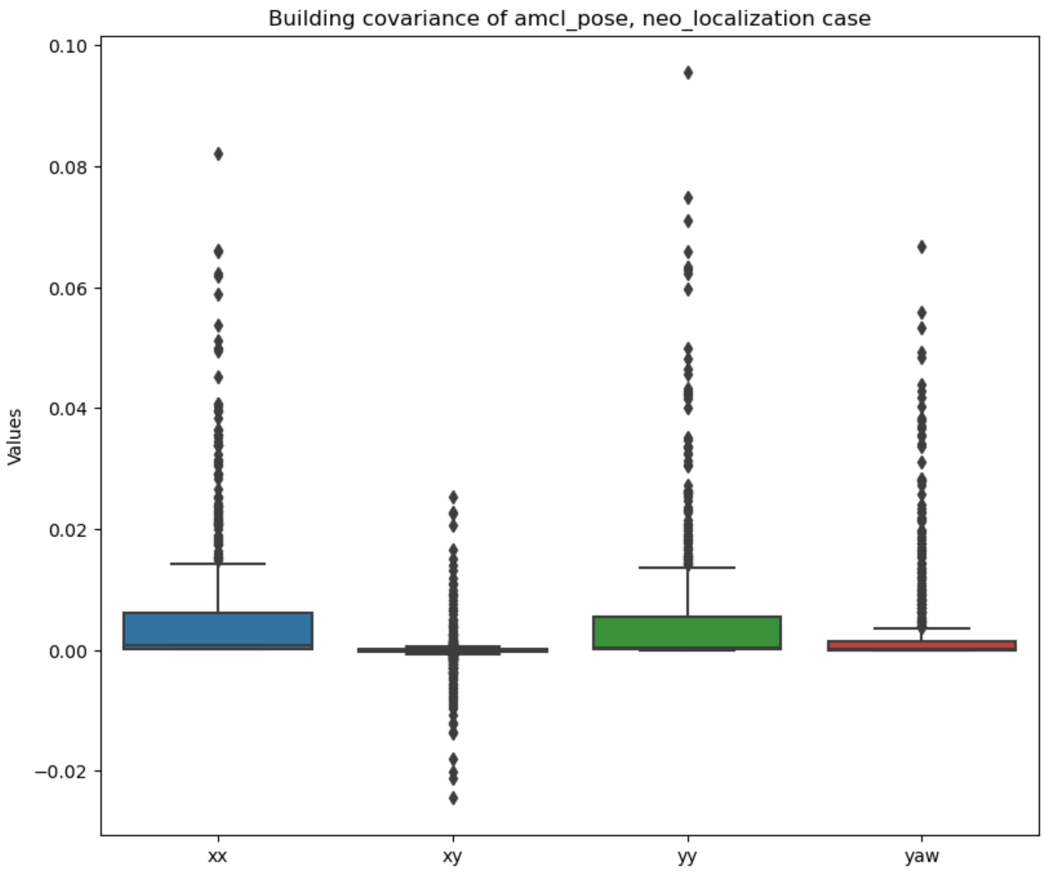


Figure 45: Box graphs of covariances of neobotics localization inside the building

5.1.5 Robustness Assessment

To further evaluate the performance of the localization system, experiments were conducted in challenging scenarios, including mesh obstacles and obstructed views. The system demonstrated robustness by consistently providing accurate localization estimates in these scenarios, albeit with slightly increased uncertainty.

5.1.6 Discussion

These results indicate that the localization system maintains a high level of precision even when the robot is in motion, but more importantly that the use of the virtualized scanner instead of the point cloud (or even both of them together) is not comparable in precision with respect to the Neobotics stack, which has shown to be more reliable and precise in all of the scenarios presented. We noticed the greatest uncertainty in positioning was recorded while navigating near the wall in the direction parallel to it, this is due to the absence of features from which the localization stack can calculate the positioning probability using AMCL.

The least precision difference between the Neobotics localization stack and the nav2 was recorded while navigating along the wall using both scan and pointcloud source of data for the nav 2 localization. In that case, the Neobotics localization was three times more precise, dropping from 1.5 meters squared on the x axis (worst offender among the four parameters) to 0.5 m^2, than the nav2 library. In the worst case, which was recorded by also navigating along the wall but using only data from the virtual lidar source for the nav2 library, dropping from 23 m^2 to 0.5 m^2 on the x axis, the difference in precision with respect to the Neobotics library was almost fifty times higher. On average, Neobotics was ten times more precise in localizing the robot than the Nav2 library using both scan and pointcloud source, and 30 times more precise than the Nav2 library using only the virtual lidar data.

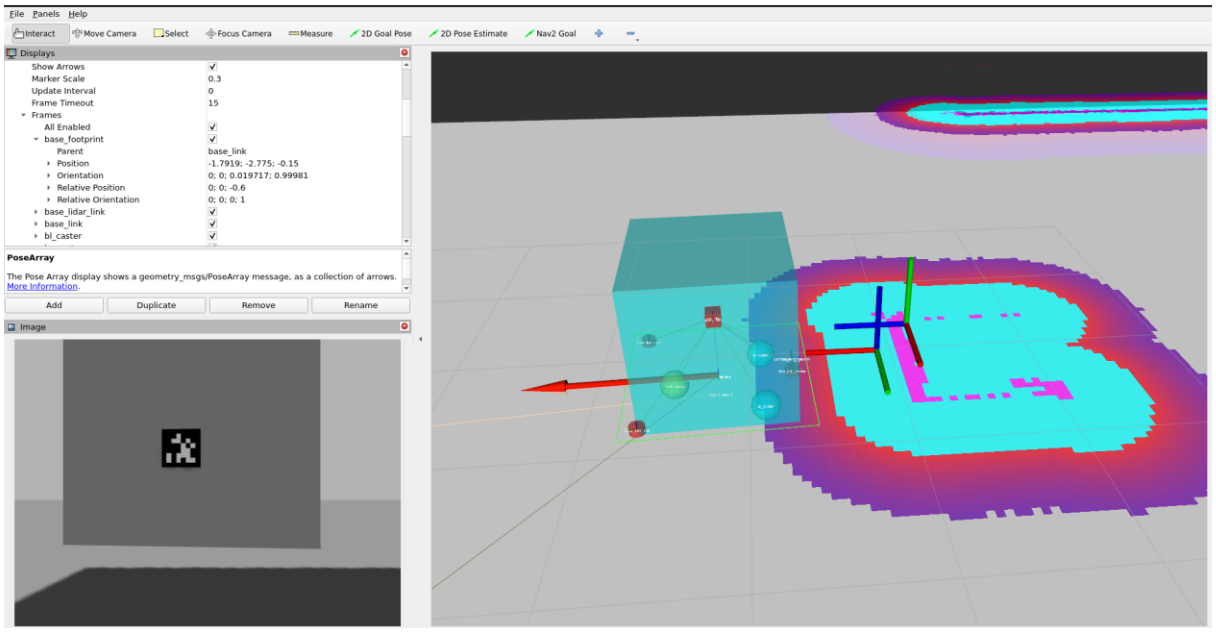
There was not much difference in precision between the only scan topic case and only the pointcloud case, while precision increased while using both. We theorize that the simple fact of having more data, even if redundant, helped the Nav2 localization algorithm.

5.2 Aruco and dock positioning precision

This section presents the results of using the Aruco codes as positioning tool, with frame coordinates recorded in the simulation. The dock placement precision was also evaluated.

5.2.1 Experimental Setup

Much in the same way to what was done for measuring the precision of the localization stack (approach stack) a benchmark environment was implemented with the addition of a dock in a fixed position with respect to the AMR, such that the Aruco marker would be visible in the camera field of view. Firstly, the AMR was kept still to measure the precision of positioning of the markers in a stationary condition, then the AMR was moved in a straight line away from the marker at a slight angle with the dock, to keep the marker still in view but measure the reaction to a different angle of visualization.



```
Show Arrows                 ✓
Marker Scale                0.3
Update Interval             0
Frame Timeout               15
Frames
  All Enabled               ✓
  base_footprint            ✓
    Parent                  base_link
    Position                -1.7919; -2.775; -0.15
    Orientation             0; 0; 0.019717; 0.99981
    Relative Position       0; 0; -0.6
    Relative Orientation    0; 0; 0; 1
  base_lidar_link           ✓
  base_link                 ✓
  bl_caster                 ✓
```

PoseArray

The Pose Array display shows a geometry_msgs/PoseArray message, as a collection of arrows. More Information.

Figure 46: Experimental setup for marker detection precision

Other than that, while measuring the precision of the maker detection, the generated dock frame position, the one to which the robot was meant to navigate to, was measured. This was done to check how imprecisions in the marker detection would detect target frame positioning.

5.2.2 Data capture and visualization

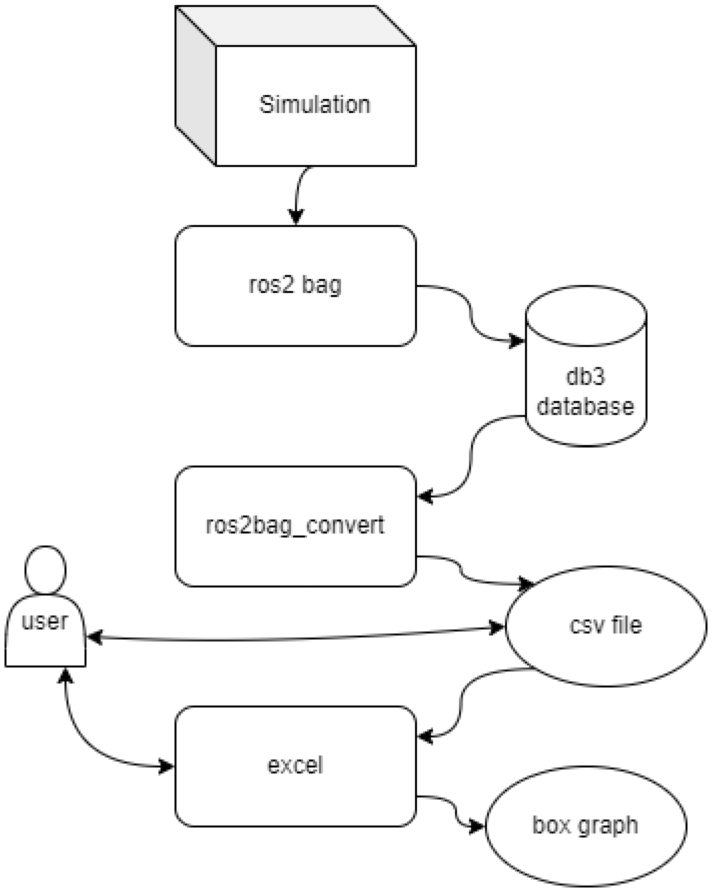


Figure 47: marker and dock variation measurement pipeline

To capture necessary data for measurements the ROS2 bag function was used, and the ingested data was transferred from a .db3 database to a .csv file for ease of use. In this case though, being the data easy enough to process by hand and not in large number, instead of using a python script the box graphs were generated by manually adapting the .csv file to use with excel and from there using the insert graph functionality after converting the quaternion to a z orientation.

5.2.3 Aruco placement Accuracy

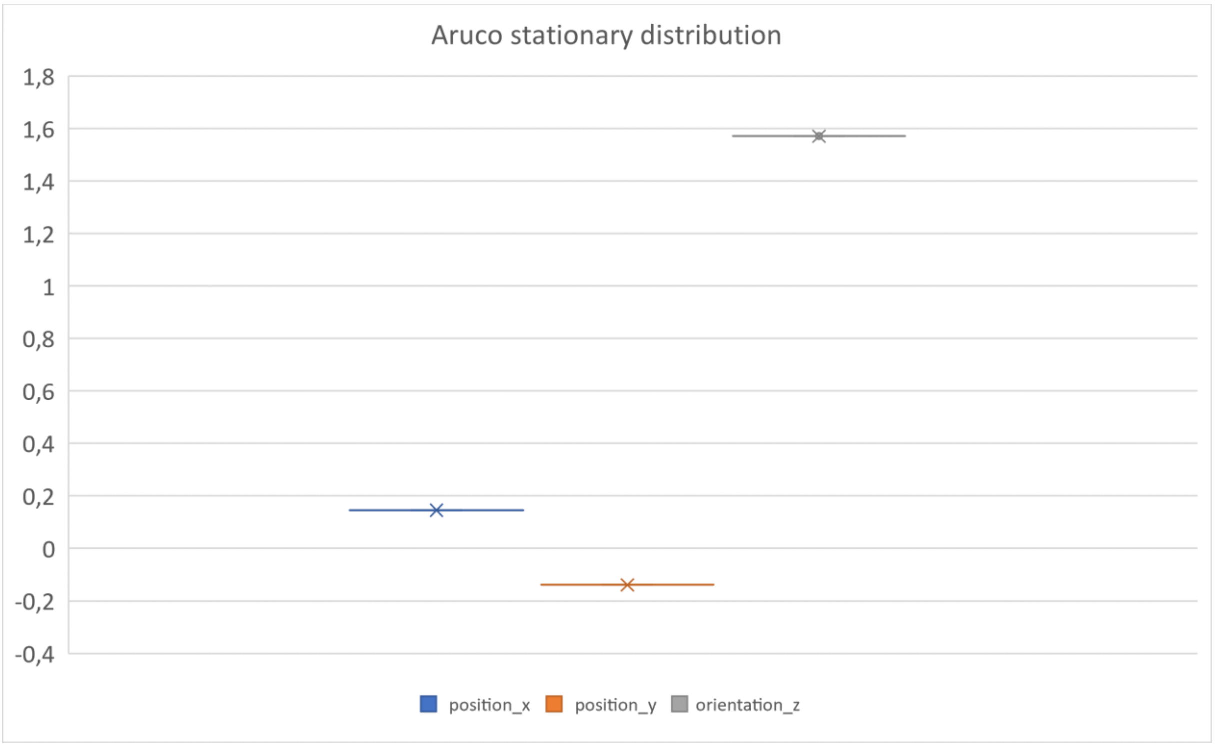


Figure 49: Box graph of Aruco positioning precision with a stationary robot.

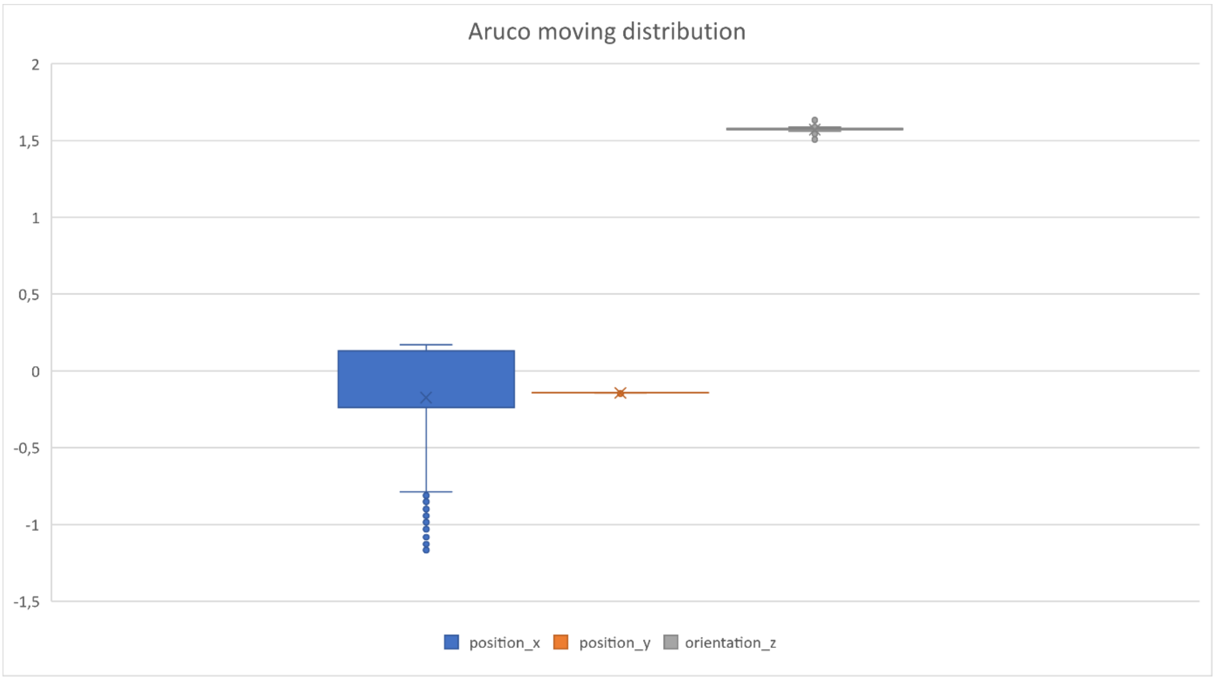


Figure 48: Box graph of Aruco positioning precision with a moving robot.

5.2.4 Aruco Accuracy Results

The accuracy of placement of the marker is extremely high in the static case, naturally this is in part due to the ideal condition of the simulated benchmark, but it shows nonetheless the robustness of the method. While moving we can see the lateral precision drop, especially as the robot moves at an angle and gets further away from the marker, with less pixels available to compute the correct orientation and placement. In this case, the greatest dispersion of marker localization coordinates was of ± 0.3 cm on the x axis.

5.2.5 Dock placement Accuracy

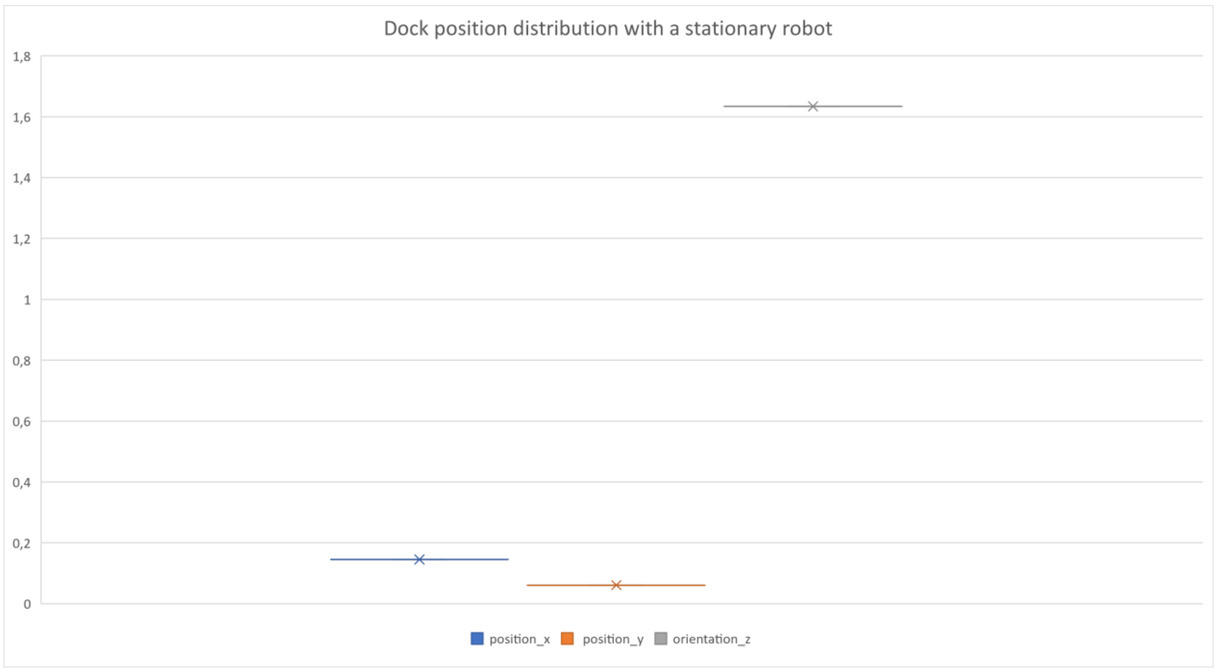


Figure 50: Box graphs of dock positioning precision with a stationary robot.

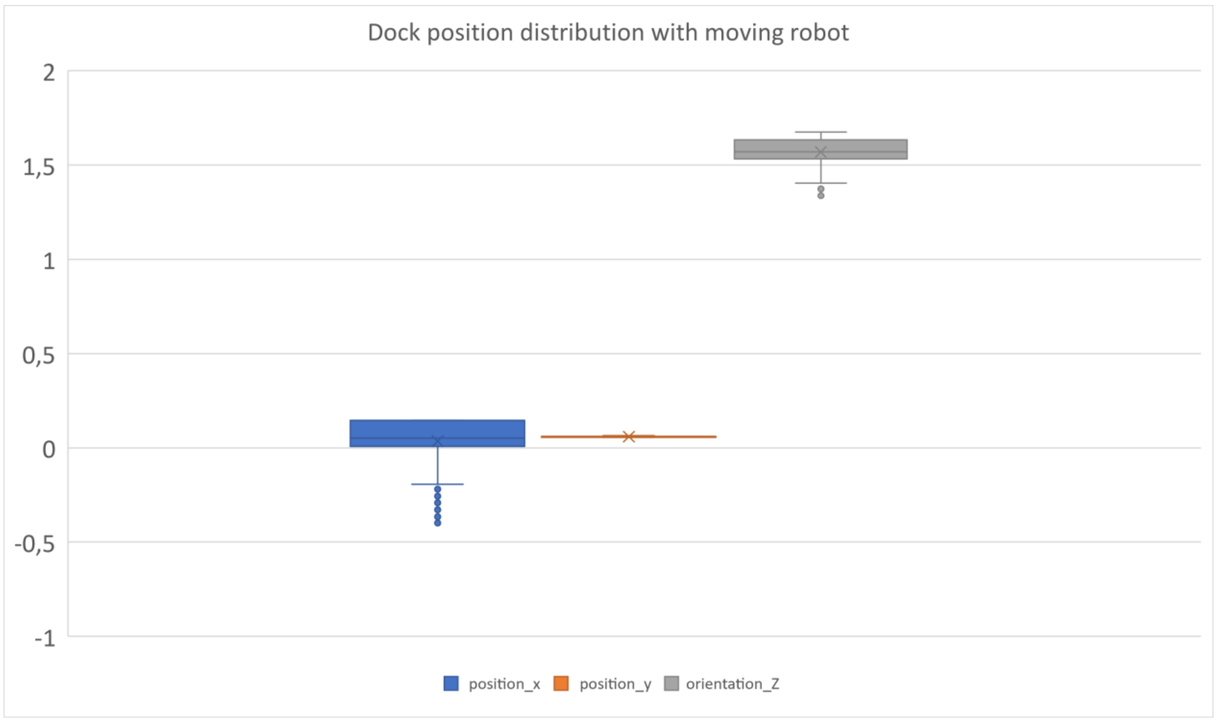


Figure 51: Box graphs of dock positioning precision with a moving robot.

5.2.6 Dock Accuracy Results

As much as with the marker, we see optimal accuracy in the static case, and variable accuracy with the robot in motion, this is worsened by the fact the placement of the dock with a fixed position with respect to the marker acts as a lever of sort, making positioning errors of the markers worse as the desired displacement from connection point and marker is increased. This means that the marker has to be positioned as close as possible to the connection point to reduce drifts.

5.3 Docking action precision and time to complete

This section presents the results of the final docking function of the robot, which is a critical component of its autonomous navigation and interaction capabilities. The experiments aimed to evaluate the precision of the docking process in terms of alignment accuracy and the time required to successfully complete the docking operation. Also in this case, the worst recorded spread of connection point coordinates was ± 0.4cm along the x axis and ± 0.03° of orientation around the z axis.

5.3.1 Docking Station

The docking station as usual was equipped with a fiducial marker for visual recognition and alignment. The robot's task was to autonomously navigate to the proximity align itself with the docking station. A lip was added on the bottom of the dock to simulate a connection point, and check whether it could interfere with the approach of the robot.

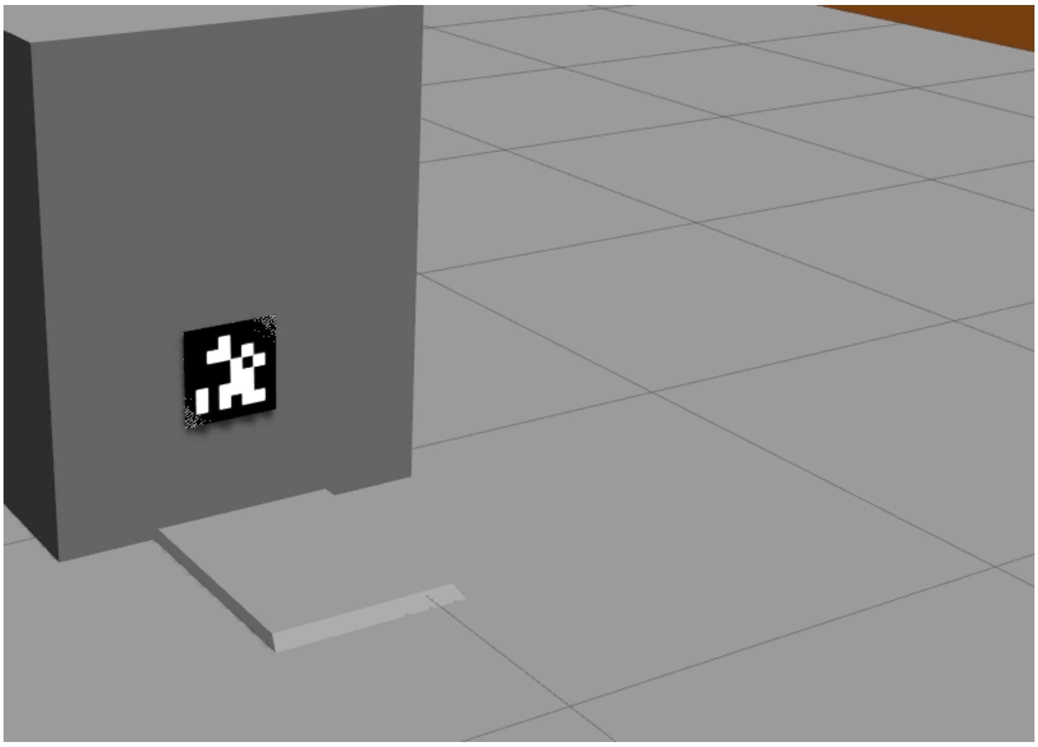


Figure 52: Improved docking station model with added bottom lip

5.3.2 Alignment Accuracy

The primary measure of precision was the alignment accuracy achieved during the docking process. To evaluate this, the robot was tasked with approaching the docking station from different starting positions and orientations. The alignment accuracy was quantified by measuring the final position and orientation error relative to the desired docking position.

In general, the total accuracy was mostly dependent on the docking part of the action, with results depending on the distance the dock target was set with respect to the Aruco marker, we managed to get a precision of **± 1cm** with x and y axis position, and **± 0.3 degrees** of orientation around the z axis.

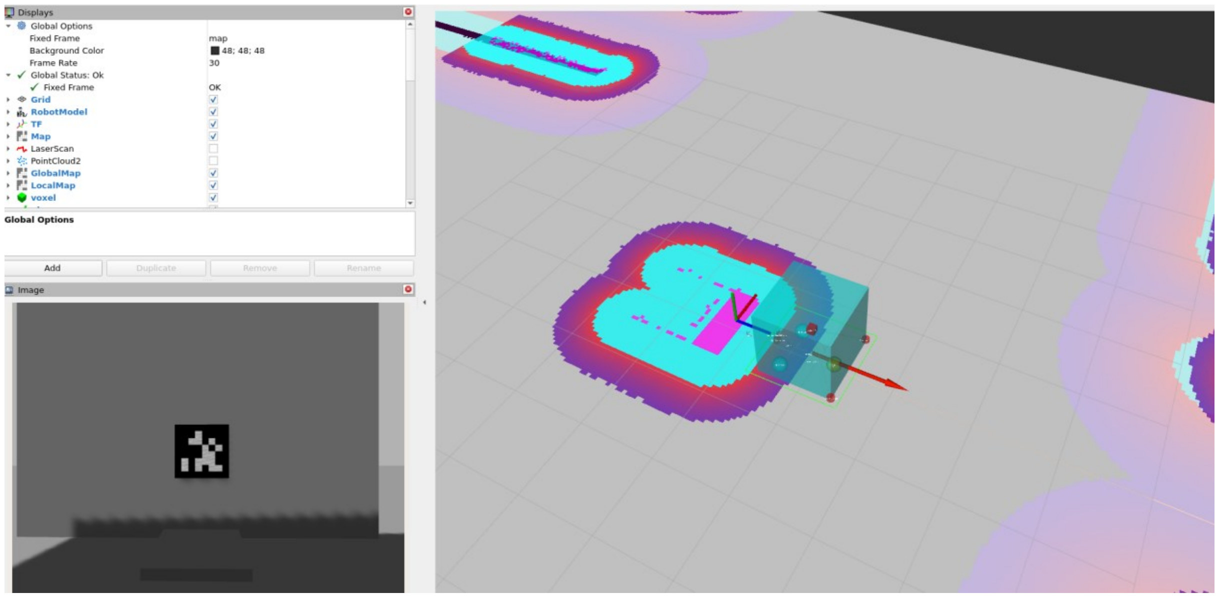

```
Displays
▾ Global Options
    Fixed Frame          map
    Background Color     ■ 48; 48; 48
    Frame Rate           30
▾ ✓ Global Status: Ok
    ✓ Fixed Frame        OK
▸ Grid                   ✓
▸ RobotModel             ✓
▸ TF                     ✓
▸ Map                    ✓
▸ LaserScan              ☐
▸ PointCloud2            ☐
▸ GlobalMap              ✓
▸ LocalMap               ✓
▸ voxel                  ✓
Global Options
```

Figure 53: RVIZ visualization during approach and docking

5.3.3 Approach and Docking Time

The time required for the robot to complete the docking operation was another critical metric of performance. The approach time was measured from the start of movement up until reaching the final position with the camera pointed toward the marker. The docking time was measured from the moment the robot started moving towards the docking station until it successfully exited the action.

Since the docking action always happened at the same distance from the docking station, the recorded time were mostly the same at around 17 seconds. The only case where the action slowed down slightly was due to the shadow from the robot interfering with the marker precision and generating some fluttering in the detection placement.

The approach times naturally varied based on distance and variation of the path, times were measured from a set of starting positions, naturally in the optimal conditions of a simulation times from the same position were almost identical, here are the results:

Table 1: Approach times from different starting positions

Position	Arrow number	Path distance (m)	Time (s)
Front right	1	1	15
Front near	2	3	20
Far right	3	9	34
Building center	4	12.5	40
Building far left	5	25	55

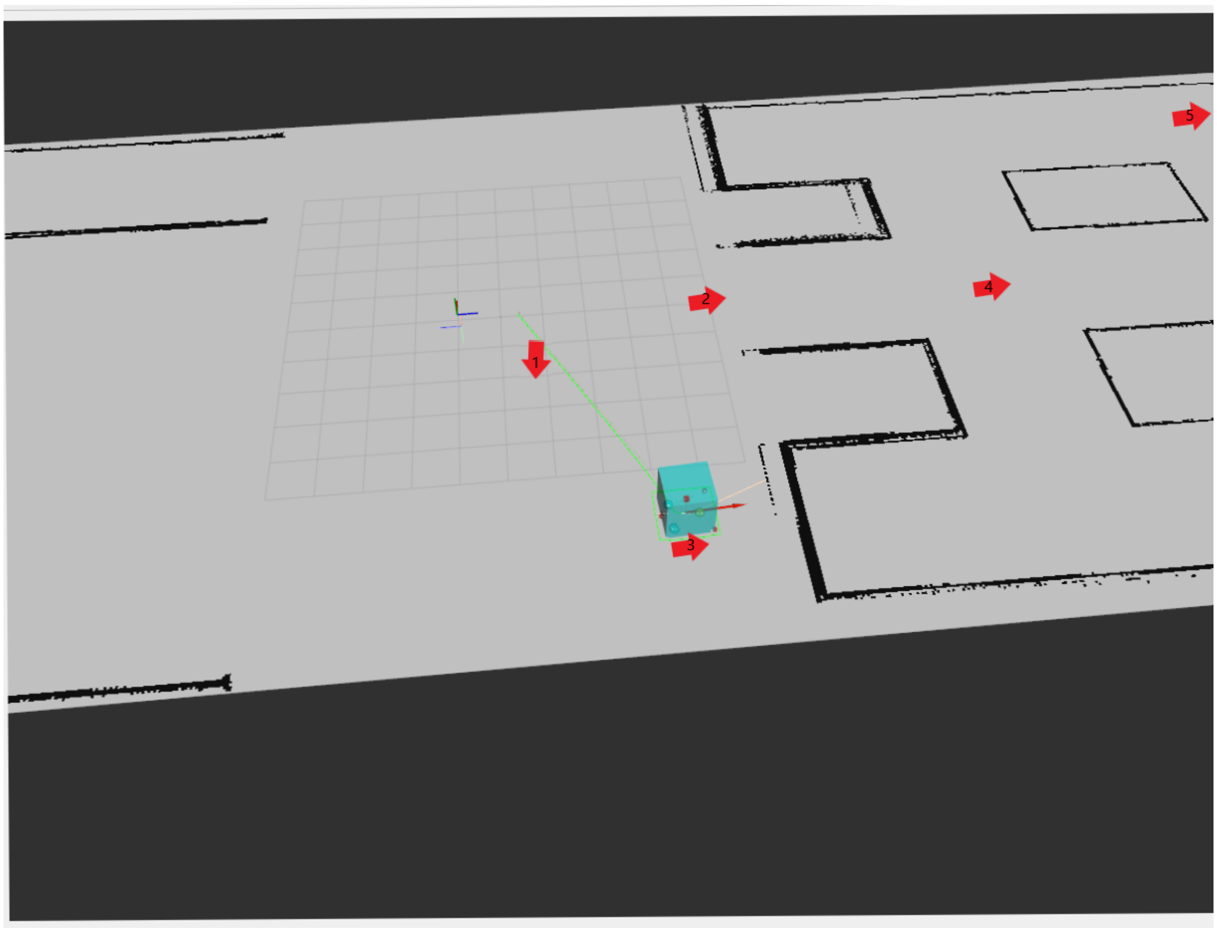


Figure 54: starting points of approach time measurements.

5.3.4 Discussion

The results of the final docking function experiments demonstrate the robot's precision and efficiency in aligning itself with the docking station. The achieved alignment accuracy, with minimal final position and orientation errors, will ensure a secure and reliable connection with the docking station.

Furthermore, the efficient docking time of approximately 17 seconds confirms the practicality and effectiveness of the robot's docking algorithm in real-world applications, such as autonomous recharging.

These results underscore the successful development and implementation of the robot's final docking function, enhancing its autonomy and usability in various industrial and logistics scenarios.

6 CONCLUSIONS

When the thesis started, the final goal of the project was defining and testing technique to safely navigate and dock an AMR in a warehouse environment using lidars and a camera in the shortest possible time.

The objective has been achieved by combining a mix of long-range navigation based on adaptive Monte Carlo localization (AMCL) and short-range navigation based on Aruco marker positioning. In the proposed algorithm, firstly the robot base receives an action goal from the user on the approximate location of a docking station, the action is started and calls for the Neobotics neo-nav2 library to navigate to the set location.

Once the position is reached, the action switches to the custom docking action we developed. The docking action takes the target dock frame generated by the amr_docking node we discussed in chapter 4, section 5 which is responsible for localizing the docking station contact point based on the detected Aruco marker.
The docking action steers the AMR to the precise location of the contact point (which can be configured as a transformation with respect to the fiducial marker) using a planar controller with proportional action.

The developed function can complete the docking procedure with times that are variable based on the distance of the robot from the desired position, but in general the final procedure which brings the AMR from an approximate position in view of the marker to the connection point lasts around 17 seconds, with very small variations based on lighting conditions. The precision of the procedure is of 1 cm for the x and y axis of movement and of 0.3 degrees of rotation around the z axis positioned at the center of the robot.

It is necessary to underline that, currently, the developed Auotonomous Mobile Forklift is still in testing phase, and as such the docking procedure cannot be tested yet in a real scenario. However, considering that in simulations the method is very reliable, accurate and fast, it is possible to assume that with the proper changes to the configuration files, and a correct calibration of the camera sensor, it could be possible to test it effectively soon on the physical robot.

Future additions to the function would probably be to develop another service to put between the approach and docking phase which checks the dock poses topic in order to assess if any marker is visible to the camera, and eventually to revert to navigation if not.

Another upgrade to the docking action could be to change the planar controller from a proportional control to a proportional-integral-derivative one, to improve the speed and precision of the robot by sending smoother velocity curves to the cmd_vel node.

In conclusion, it can be stated that the original objective of the thesis has been achieved and the proposed docking function, along with the supporting nodes for marker detection and sensor fusion, ensures a safe and precise movement up to the docking connection with minimal resources required in term of docking station remodeling. Moreover, the introduction of autonomous docking for an AMR fleet ensures longer uptimes, less man hours needed and an improvement to efficiency in a logistics and industrial environment.

BIBLIOGRAPHY

[1] U. G., Automated Guided Vehicle systems, Springer, 2015.

[2] IMA, "Imamagnet," IMA, [Online]. Available: https://imamagnets.com/en/blog/agv-magnetic-tape-and-its-advantages/.

[3] R. Rayner, "bluebotics," Bluebotics, [Online]. Available: https://bluebotics.com/agv-navigation-methods-virtual-path-following/.

[4] Bastian Solutions, "Bastian Solutions," [Online]. Available: https://www.bastiansolutions.com/solutions/technology/automated-guided-vehicles/vision-guided-vehicles/.

[5] T. Ganesharajah, N. Hall and C. Sriskandarajah, "Design and operational issues in AGV-served manufacturing systems," *Annals of Operations Research,* vol. 109, no. 154, p. 76, 1998.

[6] C. Feledy and M. S. Luttenberger, *A State of the Art Map of the AGVS Technology,* Lund: Lund University, 2017.

[7] Agilox, "One," [Online]. Available: https://www.agilox.net/en/product/agilox-one/.

[8] Agilox, "Agilox Sofware," [Online]. Available: https://www.agilox.net/en/software/.

[9] J. David and T. Rognvaldsson, "Multi-Robot Routing Problem with Min–Max Objective," *MDPI,* 2021.

[10] ROS2, "ROS2 home," [Online]. Available: https://docs.ros.org/en/humble/index.html.

[11] ROS, "ROS-Robot Operating System," [Online]. Available: https://www.ros.org/.

[12] OpenCV, "OpenCV," [Online]. Available: https://opencv.org/.

[13] ROS2, "Understanding Nodes," [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html.

[14] CycloneDDS, "CycloneDDS," [Online]. Available: https://cyclonedds.io/.

[15] The Robotics Back-End, "ROS1 vs ROS2, Practical Overview For ROS Developers," [Online]. Available: https://roboticsbackend.com/ros1-vs-ros2-practical-overview/.

[16] ROS2, "Topics," [Online]. Available: https://docs.ros.org/en/humble/Concepts/Basic/About-Topics.html?highlight=topics.

[17] ROS2, "Basic Concepts," [Online]. Available: https://docs.ros.org/en/humble/Concepts/Basic.html?highlight=graph.

[18] ROS2, "Services," [Online]. Available: https://docs.ros.org/en/humble/Concepts/Basic/About-Services.html.

[19] ROS2, "Actions," [Online]. Available: https://docs.ros.org/en/humble/Concepts/Basic/About-Actions.html.

[20] ROS2, "Launch," [Online]. Available: https://docs.ros.org/en/humble/Concepts/Basic/About-Launch.html.

[21] ROS 2, "Parameters," [Online]. Available: https://docs.ros.org/en/humble/Concepts/Basic/About-Parameters.html.

[22] Open Robotics, "Gazebo Features," [Online]. Available: https://gazebosim.org/features.

[23] ROS2, "Building a visual robot from scratch," [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Intermediate/URDF/Building-a-Visual-Robot-Model-with-URDF-from-Scratch.html.

[24] ROS2, "Building a movable robot model," [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Intermediate/URDF/Building-a-Movable-Robot-

Model-with-URDF.html.

[25] ROS2, "Adding Physical and collision properties," [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Intermediate/URDF/Adding-Physical-and-Collision-Properties-to-a-URDF-Model.html.

[26] NAV 2, "Simulating Sensors using gazebo," [Online]. Available: https://navigation.ros.org/setup_guides/sensors/setup_sensors.html?highlight=sensors#simulating-sensors-using-gazebo.

[27] Rviz, "Rviz package summary," [Online]. Available: http://wiki.ros.org/rviz.

[28] Nav 2, "Environmental Representation, Costmaps and Layers," [Online]. Available: https://navigation.ros.org/concepts/index.html.

[29] Ros 2, "Creating a Workspace," [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Creating-A-Workspace/Creating-A-Workspace.html.

[30] A. Gupta, "Merging data from multiple Lidars in ROS," [Online]. Available: https://medium.com/@amritgupta1999/merging-data-from-multiple-lidar-s-in-ros-e890fb60cbbf.

[31] C. Igor, "pc_convert_concat_lidar_scan," [Online]. Available: https://github.com/crescent-igor/pc_convert_concat_LIDAR_scan/blob/master/concat_convert_pc_laserscan.cpp.

[32] M. Jonathan, "ros2_laser_scan_merger," [Online]. Available: https://github.com/mich1342/ros2_laser_scan_merger.

[33] F. Guarda and M. Jonathan, "error: call of overloaded ‘PointXYZRGB(int&, int&, int&)’ is ambiguous," [Online]. Available: https://github.com/mich1342/ros2_laser_scan_merger/issues/1.

[34] ROS, "sensor_msgs/PointCloud2 Message," [Online]. Available: https://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/PointCloud2.html.

[35] S. Macenski, F. Martin, R. White and J. G. Clavero, "The marathon 2: a navigation system," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS*, 2020.

[36] Nav 2, "Getting started," [Online]. Available: https://navigation.ros.org/getting_started/index.html.

[37] Nav 2, "configuration guide," [Online]. Available: https://navigation.ros.org/configuration/index.html.

[38] A. Aposhian, "Fast-DDS Service Reliability sometimes hangs lifecycle manager," [Online]. Available: https://github.com/ros-planning/navigation2/issues/3033.

[39] Y. Goumaz, "Fix random Nav2 failure with Fast DDS," [Online]. Available: https://github.com/cyberbotics/webots_ros2/pull/672.

[40] Neobotix, "neobotix navigation," [Online]. Available: https://github.com/neobotix/navigation2.

[41] neobotics, "neo_localization," https://neobotix-docs.de/ros/packages/neo_localization.html.

[42] S. V. e. al, "Automatic precision docking for autonomous mobile robot in hospital logistics - case-study: battery charging," in *Conf. Ser.: Mater. Sci. Eng.*, 2021.

[43] A. Federman, "Using laser reflectivity for navigation markers," jan 2022. [Online]. Available: https://discourse.ros.org/t/using-laser-reflectivity-for-navigation-markers/23854.

[44] R. Cassinis, F. Tamparini, P. Bartolini and R. Fedrigotti, "Docking and charging system for autonomous mobile robots," *Research Gate,* pp. 1-6, 2005.

[45] P. R. Mass, A. D. Cohen, J. Lynch and C. Vu, "Method of docking an autonomous robot". Europe Patent EP2273336A2, 21 01 2004.

[46] F. &. M.-S. R. &. M.-C. R. Romero-Ramirez, "Speeded Up Detection of Squared Fiducial Markers," *Image and Vision Computing,* p. 76, 2018.

[47] S. &. M.-S. R. &. M.-C. F. &. M.-C. R. Garrido-Jurado, "Generation of fiducial marker

dictionaries using Mixed Integer Linear Programming.," *Pattern Recognition,* p. 71, 2015.

[48] OpenCV, "Detection of Aruco Markers," no. https://docs.opencv.org/4.5.2/d5/dae/tutorial_aruco_detection.html.

[49] JMU-robotics, "Ros2_Aruco," no. https://github.com/JMU-ROBOTICS-VIVA/ros2_aruco.

[50] R. C. Coulter, "Implementation of the Pure Pursuit Path 'hcking Algorithm," *Carnegie Mellon ,* p. 15, 1991.

[51] T. Moore and D. Stouch, "A Generalized Extended Kalman Filter Implementation for the Robot Operating System," pp. 335-348, 2016.

[52] ROS 2, "Ros2 bag," [Online]. Available: https://github.com/ros2/rosbag2.

[53] fishros, "ros2bag_convert," [Online]. Available: https://github.com/fishros/ros2bag_convert.

[54] Framos, "INDUSTRIAL 3D DEPTH CAMERA," [Online]. Available: https://www.framos.com/en/products-solutions/3d-depth-sensing/d400e-depth-camera.