POLITECNICO DI TORINO

DEPARTMENT OF ELECTRONICS AND TELECOMMUNICATIONS

Master's Degree in Mechatronic Engineering

Automatic Map Update for Autonomous Navigation in GNSS-Denied Environments

Supervisors:

Prof: Marcello Chiaberge

Ing: Antonio Marangi

Candidate:

Leonardo Crotti

ACADEMIC YEAR 2024/2025

Abstract

This thesis addresses the problem of automatic map updating in dynamic environments, with a particular focus on indoor airport environments, where even small variations in the structural layout can compromise the autonomous navigation capabilities of mobile robots. After an initial analysis of the ROS 2 framework, the slam_toolbox package, and the main Simultaneous Localization and Mapping (SLAM) algorithms, a methodology is proposed for identifying and classifying differences among maps acquired at different time instants. The experimental validation was carried out in simulated environments using Gazebo and RViz, by generating custom maps and developing Python algorithms based on the OpenCV and NumPy libraries for identifying discrepancies. Furthermore, a system integrated into the ROS 2 navigation stack in a decentralized environment was implemented, responsible for map management, pose estimation, and data storage through specific nodes. The results confirm the feasibility of automatic map updating, paving the way for the development of more robust and adaptable autonomous robotic systems suitable for real-world scenarios.

Contents

Introduction

The proposed work lies in the field of autonomous mobile robotics and addresses the problem of automatic map update in dynamic environments, with particular focus on scenarios without GNSS support. In such contexts, the accuracy of localization and the consistency of the spatial representation rely entirely on *Simultaneous Localization and Mapping* (SLAM) techniques, which allow the robot to construct and update the space it navigates.

The necessity of continuously maintaining the maps updated and reliable stems from the fact that real scenarios are subject to constant change: moving objects, temporary obstacles or structural modifications can make the stored representation obsolete, thereby compromising the ability of the robot to navigate and plan safe trajectories.
To satisfy this requirement, in the last years research has focused on *lifelong mapping*, intended to allow the continuous adaptation of the maps to the environment's changes. However, these approaches often result complex from a computational point of view and are hard to implement on real platforms.

The presented work proposes therefore the development and validation of a system able to automate the update phase of the map, simulating the behavior of an autonomous robot operating in GNSS-denied environments. The objective is to realize a pipeline that, starting from data provided by *slam_ toolbox* and *Nav2*, is able to automatically detect the differences of maps constructed in different times.
The system has been developed using *ROS2*, *Gazebo*, and *RViz* as main tools for nodes management, simulation and debugging.
Its pipeline integrates modules of automatic saving of the maps, file monitoring, geometric alignment and difference detection through image processing. This structure allows a completely automated management of the update process, paving the way to future implementation on real robots.

The thesis articulates in six chapters.
Chapter 1 introduces the general context of the work, along with the motivations and the objectives of the project.
Chapter 2 provides an overview on the ROS2 framework and the simulation tools Gazebo and RViz.
Chapter 3 describes the main types of SLAM techniques and the SLAM implementation in ROS2.
Chapter 4 presents the algorithm implemented to detect the differences between

the maps.
Chapter 5 illustrates the results achieved in simulation and online testing of the algorithm.
Chapter 6 summarizes the conclusions and proposes possible paths for future ideas.
An appendix has also been included to give a deeper insight in the used algorithms.

Alba Robot

Alba Robot s.r.l is a small and medium-sized enterprise (SME) based in Turin founded by the CEO Andrea Bertaia Segato in 2019 as innovative-startup and incubated by the Politecnico di Torino start-up incubator I3P. The concept originated from a concrete case of reduced mobility and subsequently developed into a broader vision: enabling greater autonomy and inclusion through advanced technological solutions.
The company proposes a micromobility B2B service that embeds autonomous electric vehicles, Italian design and advanced technologies in the fields of AI, IoT, automotive and robotics. This vision has brought Alba Robot to cooperate and test with relevant partners all over the world, such as International Airlines Group, as well as presenting its vehicles in fairs and global events such as Gitex Dubai, SMAU Milano, Airport PRM Leadership Conference in Paris, and Dubai Airshow.

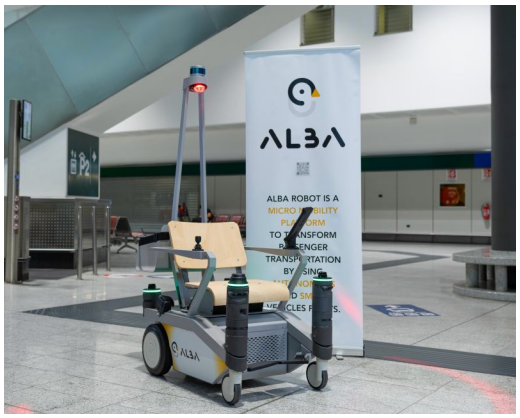


Figure 1: Alba Robot's logo and the SEDIA micromobility platform.

The core project is **SEDIA** (SEat Designed for Intelligent Autonomy), an autonomous mobility platform realized for public facilities such airports, museums and hospitals, where fleets of autonomous mobile robots, able to carry also objects, improve the accessibility and moving experience of the user.
This solution is based on a dual architecture:

- the **hardware layer**, composed by sensors, perception systems, focused on data collection and movement control;

- the **software layer**, which manages localization algorithms, path planning and detection and obstacle avoidance.

This integration allows the system to adapt in real time to environment modifications and guarantee a fluid and safe navigation. The internal activity of research and development aims to progressively strengthen the platform's performance, with the objective of transforming SEDIA in a technological reference point for autonomous mobility.

Chapter 1

Background and Motivation

1.1 General context and thesis' goal

Airports are well known for being highly dynamic environments. This trait is not only due to the constant passengers flow, but also to the facility itself, which is often subject to structural changes. From a passenger's perspective, such changes are often unnoticed, as they typically occur on a much longer timescale than the average stay at the airport. However, it becomes particularly interesting to observe what happens when the time window is extended of just a few days. For instance, Figure 1.1 considers two maps of the same facility constructed in different days of the same week.

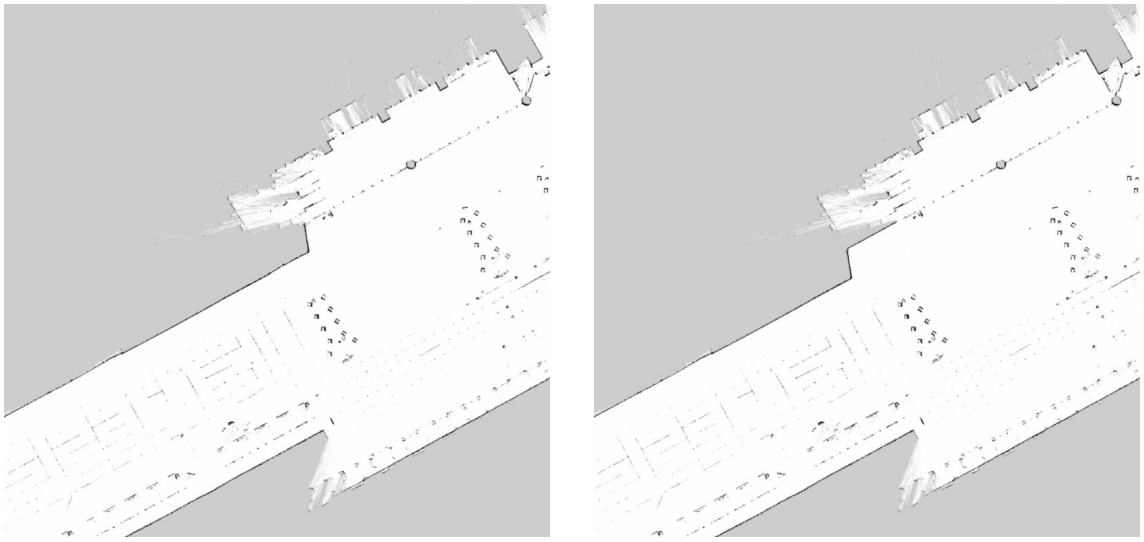


Figure 1.1: Facility map evolution.

As can be seen in the middle of the second image, a section of the upper wall of the facility was modified over the course of several days.
Reliable navigation requires robots to know their precise location within the environment and maintain an accurate representation of its structure. This is typi-

cally achieved through SLAM (Simultaneous Localization and Mapping) techniques, where a map of the environment is incrementally built while the robot simultaneously estimates its pose.
Yet, robust autonomous navigation relies on an up-to-date global map. If the environment changes, as illustrated in Figure 1.1, the existing map can become outdated, potentially leading to navigation errors.
Remapping from scratch and uploading the new map to every robot is a time-consuming process, especially in large areas where multiple robots operate simultaneously.
This thesis addresses this challenge by proposing the following steps:

- as the robot roams it constructs local maps of the environment via SLAM;
- the local maps are sent to a central map server;
- the server processes the images by comparing the newly obtained maps with the global map currently used for navigation by the platforms;
- if new obstacles are detected, they are reported;
- under certain conditions the maps can be automatically updated on the basis of the collected data.

Thus, the responsibility for applying updates to the map lies with the server rather than with individual robots. This approach simplifies synchronization across robots, facilitating the exchange of the most up-to-date map.

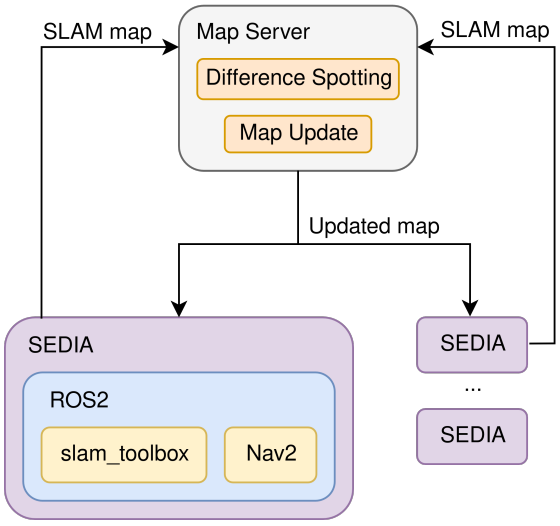


Figure 1.2: Proposed architecture scheme.

1.2 SLAM-based Automatic Map Update: State of the Art

In recent years, the scientific community has increasingly focused on the problem of automatic map updating in GNSS-denied scenarios, where SLAM techniques represent the primary tool for ensuring localization and autonomous navigation.
One of the main challenges lies in the dynamic nature of real environments: temporary obstacles, movable furniture, or the simple presence of people continuously alter the perceived structure of the scene, potentially affecting map's reliability. To address this issue, several approaches treat the map as a dynamic entity capable of evolving over time through probabilistic models that distinguish between transient and static information. This allows the system to preserve global consistency while preventing obsolete data from compromising localization accuracy.

At the same time, scalability has came up as an additional critical aspect. In In large-scale environments, adopting monolithic maps results difficult to manage, both in terms of memory usage and elaboration time. To address this problem, the most recent solutions have introduced multi-layer or hierarchical representations, in which high resolution local maps are integrated within simplified global models.
In this way it is possible to mitigate the computational requirements needed and, at the same time, the system is able to dynamically adapt the level of detail depending on the specific task, guaranteeing an optimal trade-off between precision and efficiency.

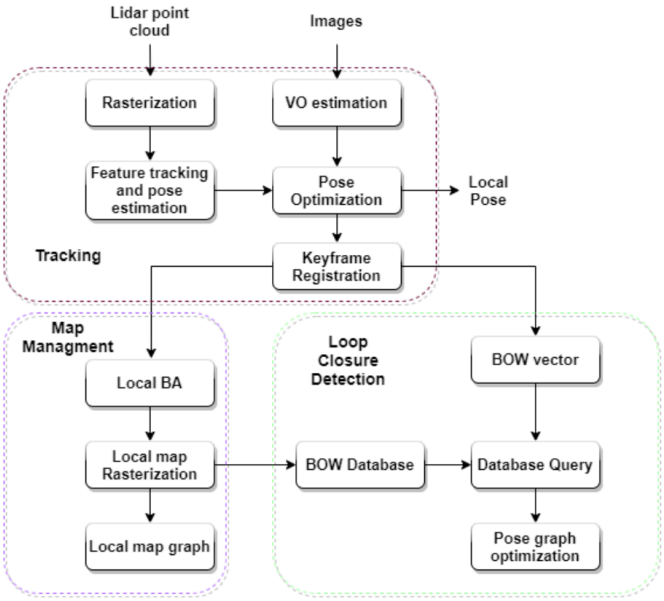


Figure 1.3: Architecture of the local-map management framework (Ali et al.).

In the work proposed by Ali et al., the authors present an architecture in which the global map is not maintained as a single entity, but rather as a collection of

rasterized local maps, each represented as an image derived from LiDAR readings. The system includes a dedicated map management component that keeps track of previously generated local maps, assigns temporal tags, and employs culling mechanisms to limit memory consumption. For loop closure and relocalization, a Bag-of-Words (BoW)-based approach operates directly on the rasterized images, achieving high performance on both indoor datasets and the KITTI benchmark, with accuracy and recall exceeding 90%. One of the main strengths of this solution lies in its trade-off between accuracy and computational cost, which remains sustainable even in long-term operation.

A pioneering contribution to long-term mapping in dynamic environments was introduced by Pomerleau et al. This work relies on 3D laser scans and proposes a probabilistic methodology to determine whether each mapped point should be considered static or dynamic, based on multiple observations distributed over time. The system preserves a *most-likely* model of the static geometry, ignoring dynamic or transient objects without explicitly modeling them. A key feature is the storage of movement patterns of observed mobile objects.
Experiments conducted on data collected in urban spaces over several months demonstrated that a consistent and reliable map can be maintained for localization, even in highly dynamic environments.

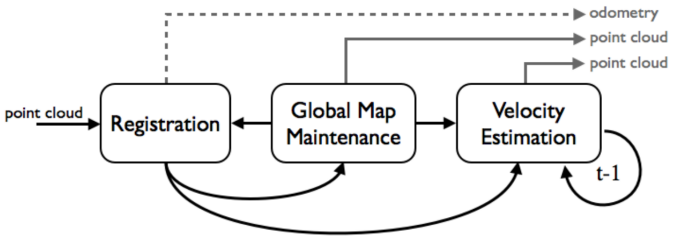


Figure 1.4: Overview of the long-term 3D map maintenance system (Pomerleau et al.).

A further line of research, as the one conducted by Cai et al., address the problem of map fusion for maps obtained in different session or by multiple agents. The proposed system, named LAMM, introduces a procedure to automatically combining heterogeneous LiDAR submaps, facing two major challenges: firstly, the adoption of a *bidirectional temporal filter*, which allows to remove dynamic objects and noise; secondly, the employment of *outlier rejection algorithm* able to prevent errors during loop closures, particularly in presence of repetitive environments or similar spaces. This methodology allows to integrate maps acquired in different times and conditions with precision, maintaining stability, scalability and robustness even with large datasets.

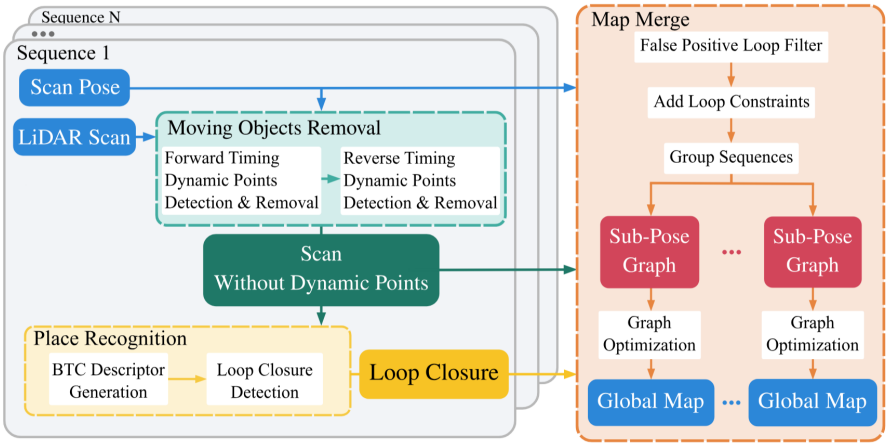


Figure 1.5: LAMM framework for multi-session LiDAR map merging (Cai et al.).

Further advances push towards *semantic* and *persistent-object* mapping, as exemplified by the work of Adkins et al. In their approach, geometric and visual features are combined with object-level semantic information obtained through deep learning, leading to compact and robust maps that remain stable despite changes in illumination or the presence of temporary objects. The system continuously updates its internal representation after each deployment, preserving both local and global consistency.
These studies demonstrate that reliable map updating requires not only geometric information, but also metadata describing the appearance, temporal persistence, and semantic nature of the observed elements.

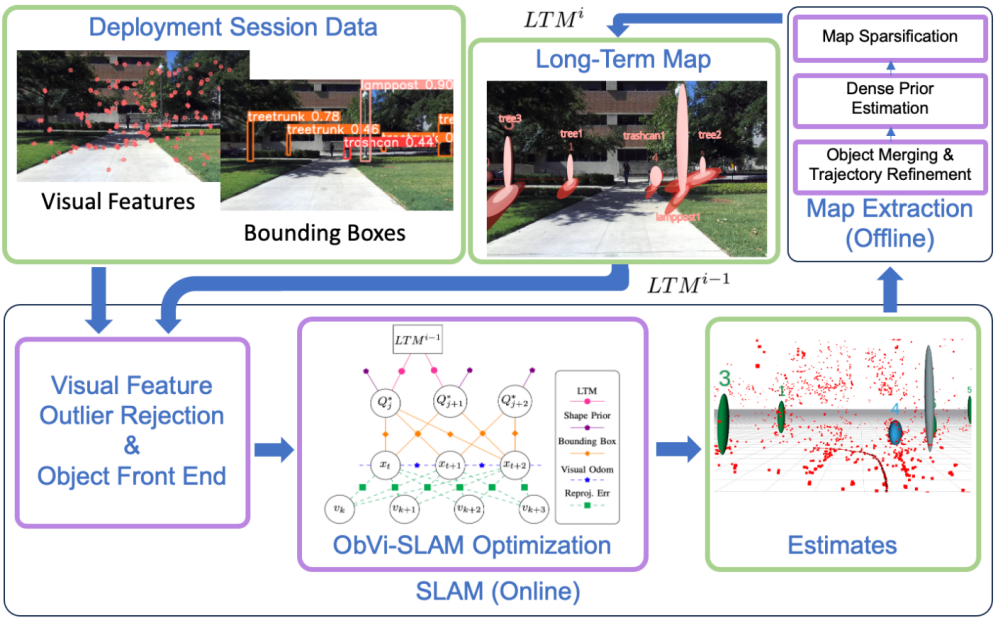


Figure 1.6: Semantic and persistent-object mapping pipeline (Adkins et al.).

Overall, the state of the art has evolved from simple collections of submaps to

complex pipelines that combine three complementary strategies:

- long-term maintenance and dynamic filtering to distinguish static from transient map elements;
- multi-session fusion techniques for the integration of data collected at different times or by different agents
- incorporation of semantic and temporal information to guide selective updates.

Research on 3D map maintenance has shown that repeated observations are sufficient to estimate the stability of a given portion of the environment, enabling the automatic removal of moving or transitory elements and preserving a reliable geometric basis for long-term localization. At the same time, modern multi-session merging methods allow for the combination of heterogeneous submaps, the computation of complex inter-session transformations, and the application of global optimization procedures that reduce drift and misalignment—an essential capability in GNSS-denied environments.

Recent studies emphasize the integration of temporal and semantic metadata within maps, enabling systems to recognize object categories, estimate their persistence, and eventually plan revisits for targeted updates, thereby reducing the computational cost of continuous mapping.

Finally, complementary research directions explore the integration of alternative sensing modalities, such as *radio-SLAM* based on *Signals of Opportunity* (SoOp), and end-to-end pipelines that coordinate autonomous exploration, change detection, and data fusion. The collective outcome of these efforts is a vision of automatic map updating not as a static mathematical operation, but as a modular and adaptive process in which heterogeneous sensors, temporal belief models, multi-session integration, and semantic reasoning cooperate to maintain a coherent and up-to-date representation of the environment over time.

Chapter 2

ROS 2

ROS 2 has been the building block of this thesis project. It is, as reported in the official documentation, *a set of software libraries and tools for building robot applications* and being open source it has become the landmark for many developers since its launch back in 2007. In this chapter, the logic behind ROS 2 will be discussed along with an analysis over the differences between ROS 1 and ROS 2.

2.1 Basic ROS concepts

ROS 2 is a middleware based on a strongly-typed, anonymous publish/subscribe mechanism that allows for message passing between different processes. At the center of the ROS 2 system lies the ROS graph, a network of interconnected *nodes*, which can exchange data through dedicated pathways called *topics*, *services* and *actions*. A key feature of ROS2 is its *language-agnostic* design, which enables developers to code both in Python and C++ through the dedicated client libraries *rclpy* and *rclcpp*.

Figure 2.1: ROS2 logo and Humble Hawksbill poster: distribution employed in this thesis.

2.1.1 Nodes

ROS 2 nodes represent the fundamental computational units of the system. Each node corresponds to a process with a specific role, such as data acquisition from sensors, information processing, or actuator control. As previously mentioned, nodes communicate through messages exchanged over dedicated channels, defined as topics, services, and actions. This communication model allows the system to be distributed, scalable, and modular, thereby facilitating the development and maintenance of complex robotic applications that are easily extendable. Moreover, testing and debugging are simplified, since ROS 2 provides a set of dedicated tools to monitor and inspect different parts of the ecosystem.

2.1.2 Messages

Messages are the means through which nodes exchange data with each other without expecting a response. They are defined in `.msg` files, located in the `msg/` directory of a ROS package.
Each file consists of two sections: fields and constants. Each field is composed of a type and a name, separated by a space; for example:

```
string my_string
int32 my_integer
```

Field types can be both built-in and Messages description defined on their own, such as `geometry_msgs/Twist.msg`. This typed structure ensures consistency in the exchanged data and enables interoperability between nodes implemented in different programming languages.

2.1.3 Topics

A ROS 2 topic is a pathway through which nodes can continuously exchange data in an asynchronous manner. The data flow is based on a **publisher-subscriber** structure where the publisher nodes broadcast data over a named bus, the topic, which can have multiple subscribers. Different nodes can exchange data through the same topic by using *namespaces*, which help organize and distinguish topics in complex systems. Topics carry ROS messages containing the transmitted information, encoded in a specific message type that the subscriber node must be able to interpret and process.

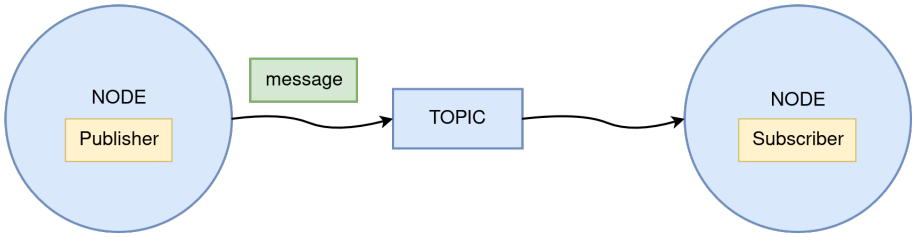


Figure 2.2: ROS nodes, messages and topics.

2.1.4 Services

ROS 2 services provide another way for data exchange between nodes, differing from topics by adopting a **client-server** structure which yields to a *request-respond* inner structure. Here multiple clients can send requests to a single server per service. Once the requested action is completed, the server sends a completion message to the client and the service call terminates.
Services are particularly useful when handling raw topic messages directly in a node is cumbersome or inefficient. For example, in this thesis the service `SaveMap` of the *slam_ toolbox* package has been used to store the `.yaml` and `.pgm` files corresponding to the maps generated via SLAM. This service internally retrieves the necessary information from the message of type `/nav_msgs/OccupancyGrid` published by the *slam_ toolbox* node on the topic `/map`, a task that otherwise would be complex to manage manually.
It is important to point out that service calls in ROS 2 are synchronous: the client node remains blocked until the server responds. It is then required that the server provides the service in an adequate time frame to avoid delays or even deadlocks in the system's operation.

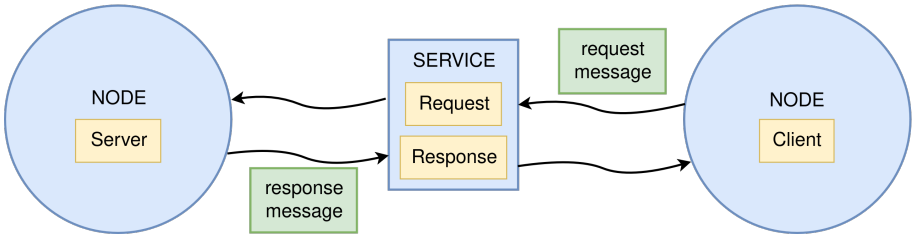


Figure 2.3: ROS service.

2.1.5 Actions

The third and final mechanism through which nodes can exchange data is via actions. These are somewhat similar to services, but additionally they can provide a feedback message about the task that is being executed, allowing the client to receive periodic updates on the progress of the request.
Unlike services, actions do not block the operation of the client node, yielding to an

asynchronous client/server model. This makes them particularly suitable for long-running tasks, such as robot navigation, where continuous progress monitoring is crucial.
Furthermore, actions can be preempted or aborted during execution, diversely from services, offering greater flexibility, which must complete once called. This feature is key in highly dynamical circumstances, where task priorities may vary during run time.

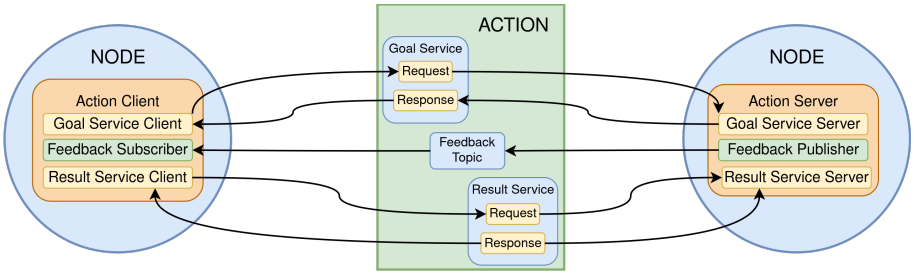


Figure 2.4: ROS action.

In the following table a comparison between topics, services and actions is reported.

Characteristic	**Topics**	**Services**	**Actions**
Communication paradigm	Publisher-Subscriber	Request-Response	Goal-Feedback-Result
Communication	continuous, asynchronous	synchronous, asynchronous, one-shot	asynchronous, long-lasting
When to use it	Data streams (e.g., sensors)	Instantaneous operations (e.g., parameter request)	Long or complex operations (e.g., robot navigation)
Message structure	`.msg` (standard or user defined)	`.srv` (request + response)	`.action` (goal + feedback + result)
Data storage	Only last message if no subscriber	Response only to the requesting client	State maintained until the action is completed
Example	Publishing odometry	Pose request of a robotic arm	Moving the platform with feedback

Table 2.1: Comparison between Topics, Services and Actions in ROS 2

2.1.6 ROS packages

In ROS 2, a package the fundamental unit for software arrangement. Each package contains the source file of messages, services, actions, configurations and metadata necessary to develop nodes and components of the robotic system. Packages are the way through which the modularity, reusability and distribution of the ROS 2

software is managed.
Each package usually contains:

- Nodes directories (`src/`), where nodes are implemented either in Python or C++;
- Messages, services and actions definitions (`msg/`,`srv/`, `action/`);
- Configuration and parameters files (`config/`);
- Metadata such as `package.xml` and `CMakeLists.txt`, which allow ROS 2 to build, install and embed the package on the workspace.

Packages can be published and shared through repositories, allowing developers to easily interact with ready built functionalities.

2.1.7 Managed Nodes

A crucial introduction of ROS 2 with respect to ROS are managed nodes, or lifecycle nodes. The four primary states are the ones depicted in Figure 2.6 and these are:

- **Unconfigured**: initial state of the node after its creation. Resources are not allocated yet and the parameters have not been configured. It also the state where the node is returned if errors occur;
- **Inactive**: the node is configured, but not actively performing computations. Interfaces (publisher, subscriber, timer, action, etc.) are instantiated, but do not produce or consume data;
- **Active**: operational state of the node, in which it carries out its primary functions: publishing, subscribing, processing data, and acting as an active component of the system;
- **Finalized**: terminal state in which resources are released and the node can no longer be reactivated.

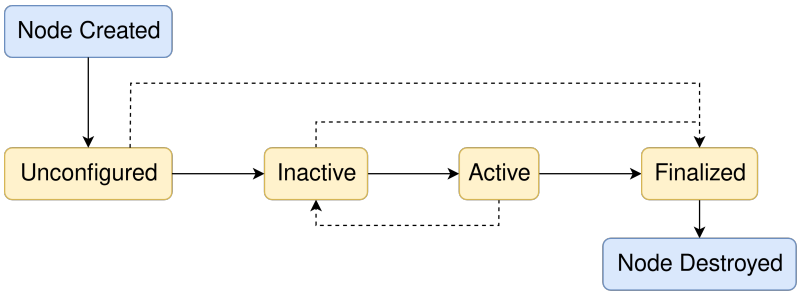


Figure 2.5: ROS managed nodes, simplified scheme.

Besides the primary states, there are also six transition states and seven possible transitions between states, which are respectively Configuring, CleaningUp, ShuttingDown, Activating, Deactivating, ErrorProcessing and create, configure, cleanup, activate, deactivate, shutdown and destroy.
A more detailed scheme is reported to better show in the following figure.

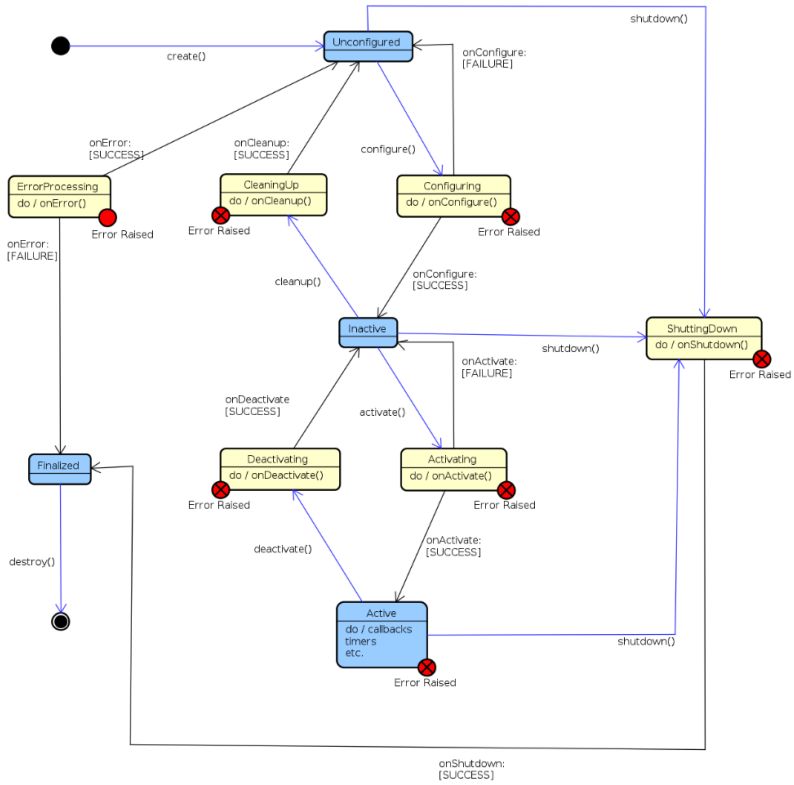


Figure 2.6: ROS managed nodes, complete scheme.

2.1.8 Recording and replaying topics: rosbag

Rosbags are the standard way for recording and replaying ROS messages enabling offline testing, simulation, debugging and data sharing with developers who do not have direct access to the hardware.
Once a rosbag is recorded, its content is saved in a directory containing a file called `metadata.yaml` with useful information on the saved topics and one or more `.db3`

2.2 Simulation and visualization tools: Gazebo and RViz

The two key tools for simulation and visualization within the ROS2 framework are Gazebo and RViz. These are completely optimized to be a part of the ROS2 ecosystem as they can operate as standalone nodes.
Gazebo is a 3D dynamic simulation environment for robotics, suitable both for in-

door and outdoor scenarios. It provides accurate physics, a wide library of sensors and customizable plugins that enable testing of to test algorithms in realistic conditions. At the time of writing, *Gazebo Ionic* is the latest actively maintained version, whereas this thesis relied on *Gazebo Classic* (EOL: January 2025), as it was the version that better fit the company's workflow.
On the other hand, Rviz is a visualization tool that enables the developer to monitor and debug the main topics, such as sensors output, cost maps used for navigation, reference frames and the transformations between them. It supports multiple visualization modalities (e.g., occupancy grids, point clouds, and robot states) and allows interactive features, such as defining navigation goals directly from the graphical interface. As a result, RViz plays a crucial role in both development and debugging phases.
These two components are at the basis of the ROS 2 development cycle as they help reducing the risks and costs associated with testing on physical hardware.

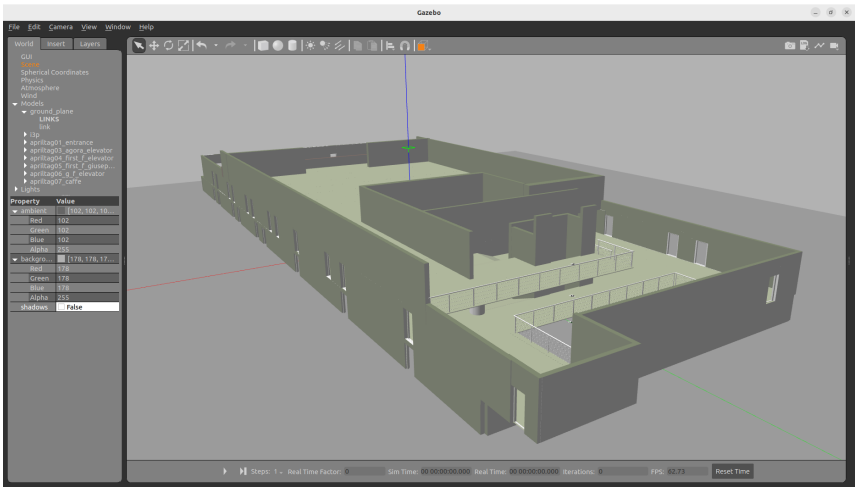


Figure 2.7: Gazebo front-end with I3P 3D model.

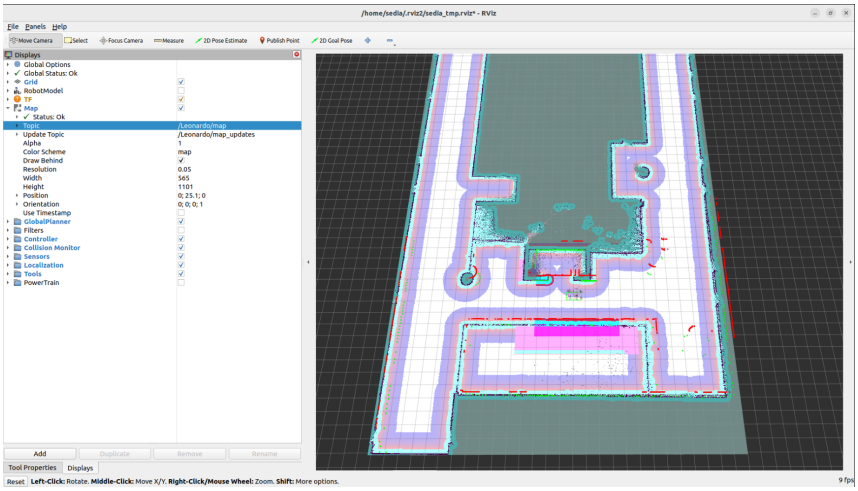


Figure 2.8: RViz2 front-end.

2.3 Navigation Stack in ROS2: Nav2

Navigation2, in short Nav2, has been developed as the successor of the ROS Navigation Stack, the open-source project proposed in 2010, which featured a flexible and robust navigation solution that has been optimized to work with different types of robots. Nav2's architecture is based on **Behaviour Tree**, which coordinate the main phases of the navigation: global path planning, local control and recovery actions in case of failures. This choice replaces the traditional finite state machines, offering greater flexibility, reusability and configurability during run time.
Each key functionality is implemented as a standalone asynchronous server exposing standard interface and supporting plugin loading. This allows the user to select among different planners, controllers or recovery strategies depending on the specific necessity.

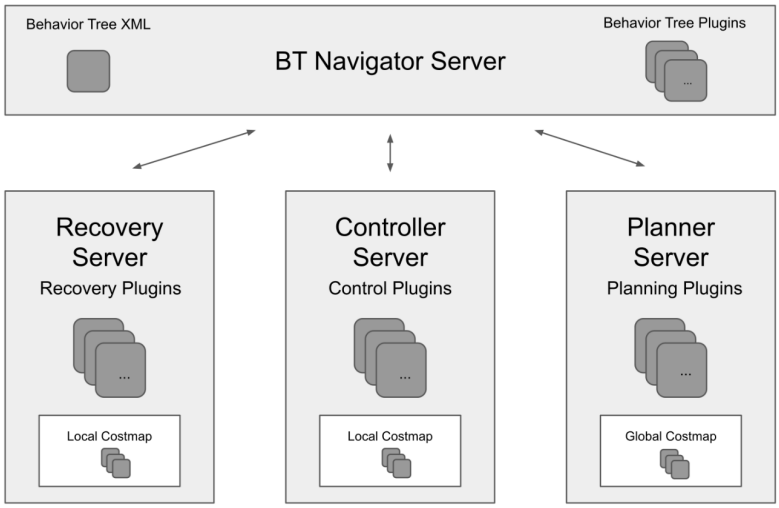


Figure 2.9: Nav2 architecture design.

From a perceptive point of view, Nav2 uses *layered costmaps* , which combine information from multiple sensors and can include *Spatio-Temporal Voxel Layer (STVL)* to represent dynamic and three-dimensional obstacles in a scalable way.
For path planning and control, the framework offers algorithms such as $A*$ for global planning and the *Timed Elastic Band (TEB)* controller for generating optimized local trajectories in environments with moving obstacles.
Localization is supported by consolidated tools like *AMCL* and *slam_ toolbox*, integrated with sensor fusion frameworks (e.g., *Robot Localization*) to combine odometry, IMU and vision.
Nav2 fully exploits ROS 2 features, such as *managed nodes* for deterministic lifecycle management of the processes and the DDS-based real-time communication, making it suitable for industrial and safety-critical applications. Thanks to its modular architecture, this system supports different types of robots (differential, holonomic, Ackermann, and even legged) and can be configured for a wide range of scenarios.

The experiments carried out in demonstrate Nav2's ability to operate in complex and densely populated environments over long periods of time without human intervention, ensuring robust navigation, automatic recovery and absence of collisions.

2.4 Quality of Service in ROS 2

A fundamental aspect of ROS 2 is the management of *Quality of Service (QoS)* in the communication between nodes. While in ROS 1 message exchange relied on the TCPROS/UDPROS protocols with limited configurability, ROS 2 ins built on top of the *Data Distribution Service (DDS)* middleware, a standard developed by the Object Management Group (OMG) for real-time distributed systems.
The DDS introduces the *Data-Centric Publish-Subscribe (DCPS)* model, where processes share data in a global data space and each communication is governed by a series of QoS policies. ROS 2 abstracs much of the complexity associated with DDS, but still allows the user to configure crucial parameters, which allows to balance reliability, performance and resources consumption.
The main QoS policies include:

- **Reliability**: choose between *best-effort* (minimum latency, but possible message loss) and *reliable* (guaranteed delivery, at the cost of retransmissions and increased latency);
- **History & Depth**: defines whether to store only the last published message or a configurable window of past messages, useful for late-joining subscribers.
- **Durability**: specifies whether messages are preserved for future subscribers (*transient_local*) or discarded immediately (*volatile*).
- **Deadline**: sets the maximum time within which data must be updated, supporting real-time constraints.

These options provide a level of flexibility unavailable in ROS 1 and are particularly important in scenarios with strict requirements, e.g., mobile robots, autonomous vehicles, embedded systems. Obviously, more stringent QoS configurations introduce overhead: experimental results show higher latencies with large messages, but also improved robustness and predictability.
The fact that ROS 2 inherits such properties directly from the DDS represents one of the main innovations compared to ROS 1. DDS has been developed for mission-critical domains such as aerospace, defense, and transportation. Its integration into ROS 2 combines the modularity of the ROS framework with communication tools typical of distributed real-time systems.
The studies by Maruyama et al. provide an experimental evaluation of ROS 2 performance in terms of end-to-end latencies, throughput, resources consumption, and overhead introduced by the reliability mechanism. For instance, using *reliable*

QoS increases the robustness in case of loss of packages, but also introduces overhead and increases message delivery time. In contrast, *best effort* mode prioritizes performances sacrificing reliability. Similarly, durability with *transient local* allows late-joining nodes to receive past messages, at the expense of higher memory usage. In general, the ability to dynamically configure these options makes ROS 2 well suited for heterogeneous scenarios, that range from soft real-time applications such as service robotics, to safety-critical circumstances, where reliability and determinism have priority with respect to latency.

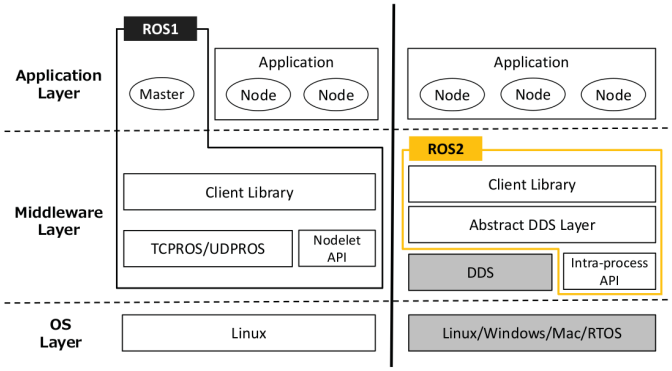


Figure 2.10: ROS1 and ROS2 architectures.

Figure 2.11 illustrates an example of phow the ROS 2 ecosystem could operate in an assisted driving application. The camera node *camera* senses the environment and published its data on the *images* topic, which is subscribed to by two other nodes: *pedestrian detection* and *car detection*.

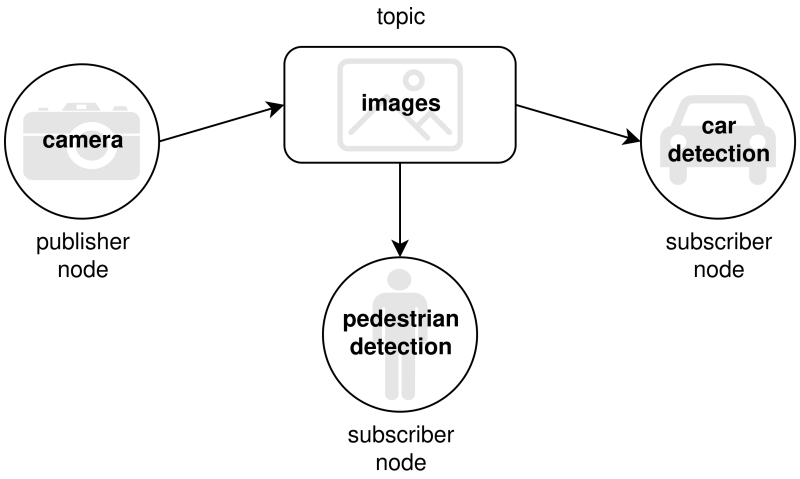


Figure 2.11: ROS nodes, messages and topics.

In such context, QoS policies play a major role in ensuring that data is exchanged according to the application requirements. For example, the topic *images* could use a *best effort* policy, prioritizing speed over reliability, since the loss of some

frames would not compromise the overall system's functionality. Conversely, the *pedestrian detection* and *car detection* nodes may benefit from a *reliable* policy, which guarantees the delivery of each message, reducing the risk of missing critical detections. Likewise, adopting the *transient local* policy would allow new subscribers to receive the most recent messages without waiting for the next camera acquisition cycle.

2.5 ROS 1 vs ROS 2

Table 2.2 summarizes the main differences between ROS 1 and ROS 2 in a concise format.

Feature	**ROS 1**	**ROS 2**
Architecture	Centralized, requires a *ROS Master* for name resolution and coordination	Decentralized, based on DDS (Data Distribution Service) middleware, no central master node
Communication	Custom protocols (TCPROS, UDPROS), limited flexibility	DDS with configurable *Quality of Service* (QoS) policies (reliability, durability, deadlines, etc.)
Real-time support	Very limited, not suitable for hard real-time systems	Designed with real-time capability in mind, better suited for industrial/embedded applications
Platform support	Mainly Linux, partial support for Windows and macOS	Multi-platform: Linux, Windows, macOS, and real-time OS (RTOS)
Security	No built-in support, requires external solutions	Built-in security extensions (authentication, encryption, access control) through DDS-Security
Node management	Basic node lifecycle management	Lifecycle/managed nodes with explicit states (inactive, active, shutdown)
Community status	Mature, widely used in research and prototyping	Actively developed, standard for new projects, growing industrial adoption

Table 2.2: Comparison between ROS 1 and ROS 2.

Chapter 3

SLAM

SLAM, which stands for *Simultaneous Localization And Mapping*, is the technology that enables a robot to build a map an unknown environment while simultaneously position itself in the map just created. To accomplish this task, sensors such as IMUs, cameras and LiDAR are commonly employed. Depending on the sensors used, SLAM can be classified as either LiDAR-based or visual SLAM.
In this problem hides a *chicken-and-egg* dilemma, since a map is required for localization and localization is necessary for mapping. For this reason, the SLAM problem remains one of the most critical problems to solve in order to implement fully autonomous mobile robots.
In this chapter, the mathematical tools behind in the SLAM problem will be introduced, together with an overview of its paradigms.

3.1 Mathematical basis

For mobile robots operating on a flat ground, the location at time t can be represented by a three-dimensional vector x_t, where the first two components represent the coordinates in a 2D plane and the third denotes the heading angle. The sequence of positions over time, that is to say the *path*, is then given by

$$X_T = \{x_0, x_1, x_2, \dots, x_T\} \tag{3.1}$$

where T is the final time instant and the initial position x_0 is assumed to be known. Let's then consider the odometry data obtained from the wheel encoders or the motors control inputs:

$$U_T = \{u_0, u_1, u_2, \dots, u_T\} \tag{3.2}$$

If the measurements u_i were all noiseless, the true path of the robot would be easy to recover, but these measurements are subject to significant noise and drift, which introduce uncertainty and cumulative error over time.
Let finally m be a known position on the map, which may correspond to a landmark, an object or a specific surface in the environment that the robot can sense and thus

we can introduce the set of measurements taken at specific time instants $t \in T$

$$Z_T = \{z_1, z_2, z_3, ..., z_T\} \tag{3.3}$$

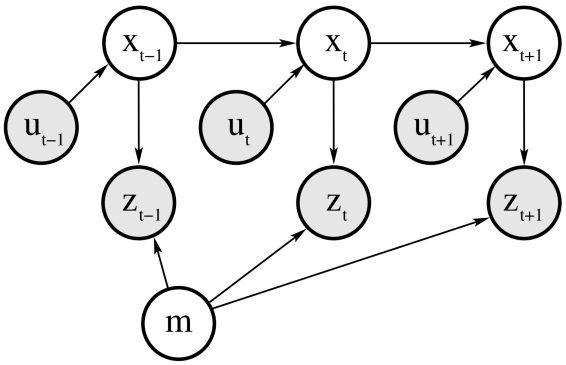


Figure 3.1: Graphical representation of the SLAM problem. The arcs are causal relationships and the shaded nodes are what is directly observable by the robot.

The graphical representation of Figure 3.1 helps better understand the dependencies between the variables in the SLAM problem. The problem is then casted into recovering the sequence of robot poses X_T and a model of the world m, using odometry and sensor measurements.

3.1.1 Types of SLAM

The presence of uncertainty coming from the measurements makes a probabilistic approach necessary. In this context, two main formulations of the SLAM problem can be distinguished: the **full SLAM** and the **online SLAM**. The first consists in estimating the posterior over the entire robot trajectory together with the map.

$$p(X_T, m \mid Z_T, U_T) \tag{3.4}$$

The algorithms for this type of SLAM process data in batches, that is to say they process all data simultaneously.
The second is defined as follows:

$$p(x_t, m \mid Z_T, U_T) \tag{3.5}$$

Online SLAM aims to estimate only the current robot pose instead of the entire path. Here the adopted algorithms are generally incremental and process one element at a time; such methods are referred to as *filters*.
Both formulations require two mathematical models to be solved: one that relates the odometry measurements u_t to the robot poses x_{t-1} and x_t, and one that relates the measurements z_t to the environment m and the robot pose x_t. These models are represented by the arcs in Figure 3.1.

It is common to model them as probability distributions: $p(x_t \mid x_{t-1}, u_t)$ denotes the probability distribution of the location x_t given a previous pose x_{t-1} and odometry measurements u_t. Similarly, $p(z_t \mid x_t, m)$ is the probability fo observing z_t at pose x_t in a known environment m. Clearly, neither the robot pose nor the environment are directly observable; however, as will be shown later, Bayes' rule plays a key role in transforming these models in probability distributions over those latent variables, conditioned on the measured data.
In addition to the described distinction, the literature often refers to further classifications of the SLAM problem, such as:

- **Volumetric versus feature-based**: the first one represents the environment as a continuous volume (for instance, 3D occupancy grids), whereas the second is based on feature extraction (points, lines, landmarks) to reduce the map's complexity;
- **Topological versus metric**: in topological SLAM the environment is described ad a network of nodes and connections, while in the metric case the positions are represented as precise geometric coordinates;
- **Known versus unknown correspondence**: in the first case it is assumed to know the correspondence between observations and landmarks of them map; in the second the correspondence needs to be estimated making the problem harder.
- **Static versus dynamic**: in the static case the environment is considered time-invariant, whereas in the dynamic case the system needs to be able to deal with moving objects and obstacles;
- **Small versus large uncertainty**: it depends on the measurements noise level, if it is high, algorithms need to be more robust and complex;
- **Active versus passive**: in active SLAM the robot actively plans its motion in order to maximize the map's accuracy and uncertainty reduction, while in the passive problem it just elaborates the observation collected over a fixed trajectory;
- **Single-robot versus multi-robot**: in the first case the map construction is managed by only one agent, while in the second more robots cooperate to sharing data and partial maps increasing efficiency, but also complicating the process due to the need of data fusion.

3.1.2 Paradigms of SLAM algorithms

In the previous sections, many SLAM problems variants were listed, this yields to plenty of algorithms proposed for their solution. In this section the main three paradigms, i.e., *EKF*, *graph-based optimization* and *particle methods*, for the solution of the SLAM problem will be presented.

Extended Kalman Filters

This approach, developed in the late 1980's, was the first successful attempt of finding the solution to the SLAM problem using the tools of Gaussian recursive estimation.
The Kalman Filer itself is a recursive Bayes filter designed for linear systems under Gaussian noise assumption. However, the linearity assumption is not realistic for mobile robots operating in the real world. The *Extended Kalman Filter* (EKF) overcomes this limitation by linearizing the nonlinear motion and observation models through a first-order Taylor expansion. The EKF provides a solution to the online SLAM problem and its algorithm is reported below.

Algorithm 1 Extended_Kalman_filter($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$):

1: $\bar{\mu}_t = g(u_t, \mu_{t-1})$
2: $\bar{\Sigma}_t = G_t \Sigma_{t-1} G_t^\top + R_t$
3: $K_t = \bar{\Sigma}_t H_t^\top (H_t \bar{\Sigma}_t H_t^\top + Q_t)^{-1}$
4: $\mu_t = \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t))$
5: $\Sigma_t = (I - K_t H_t)\bar{\Sigma}_t$
6: **return** μ_t, Σ_t

Each iteration of the filter starts from the previous belief, represented by the mean estimate μ_{t-1} and the covariance matrix Σ_{t-1}, the new control input u_t, and the new observation z_t.
Lines 1–2 perform the *prediction step* of the algorithm as they take into account the robot's motion, predicting the belief given the control commands and not taking in consideration the observation. Lines 3–5 correspond to the *correction step*, where the Kalman gain K_t balances the uncertainty between prediction and measurement, considering also the observation.

State representation. The state vector μ_t contains the robot pose and the position of the N landmarks:

$$\mu_t = (x, y, \theta, m_{1,x}, m_{1,y}, ..., m_{n,x}, m_{n,y})^\top \tag{3.6}$$

which, more compactly, becomes:

$$\mu = (x, y, \theta) + 2N_{landmarks} \tag{3.7}$$

For example, in a 2D world with 100 landmarks, the state vector will have a dimension of 203.
Each landmark is assumed to correspond to a specific and fixed index.

The belief is represented as the pair (μ, Σ), with

$$\mu = \begin{bmatrix} x_R \\ m_1 \\ \vdots \\ m_N \end{bmatrix}, \qquad \Sigma = \begin{bmatrix} \Sigma_{x_R x_R} & \Sigma_{x_R m_1} & \cdots & \Sigma_{x_R m_N} \\ \Sigma_{m_1 x_R} & \Sigma_{m_1 m_1} & \cdots & \Sigma_{m_1 m_N} \\ \vdots & \vdots & \ddots & \vdots \\ \Sigma_{m_N x_R} & \Sigma_{m_N m_1} & \cdots & \Sigma_{m_N m_N} \end{bmatrix}. \tag{3.8}$$

More compactly

$$\mu = \begin{bmatrix} x \\ m \end{bmatrix}, \qquad \Sigma = \begin{bmatrix} \Sigma_{xx} & \Sigma_{xm} \\ \Sigma_{mx} & \Sigma_{mm} \end{bmatrix}. \tag{3.9}$$

Here, $\Sigma_{x_R x_R}$ represents the uncertainty on the robot pose, $\Sigma_{m_i m_i}$ represents the uncertainty of landmark m_i, while the off-diagonal blocks model the *correlations* between robot and landmarks and among the landmarks themselves. These correlations are fundamental in SLAM: when the robot observes a landmark, not only the uncertainty of that landmark decreases, but also the uncertainty of the robot and of other landmarks correlated with it.
To better understand how the algorithm works, the Figure 3.2 offers a visualization of what is updated at each step.

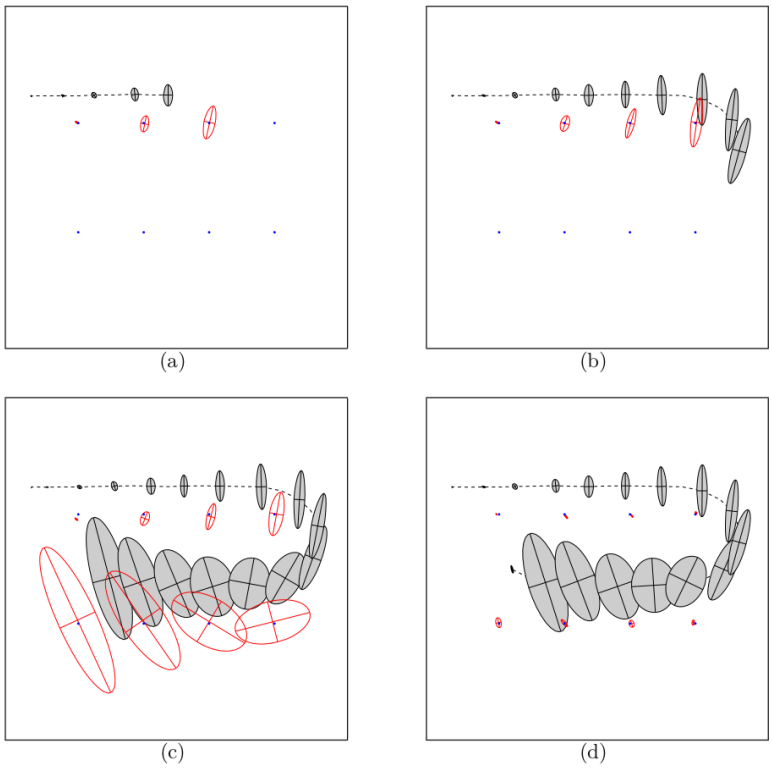


Figure 3.2: The pictures represent the path of the robot (dashed), the landmarks (blue dots) an the uncertainties on the robot's position (gray) and on the landmarks' position (red).

A concrete example on how the EKF-SLAM problem can be implemented assumes:

- a robot moving on a 2D plane;
- velocity-based motion model;
- landmarks treated as point features;
- range-bearing sensor (measuring distance and angle to landmarks);
- known data association (each observation corresponds to a known landmark);
- known and fixed number of landmarks.

Firstly, the system needs to be initialized. The assumption made is that the starting point is set where the platform begins its mission.

$$\mu_0 = (0\ 0\ 0\ \dots\ 0)^\top, \qquad \Sigma_0 = \begin{pmatrix} 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \infty & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \infty \end{pmatrix}. \tag{3.10}$$

It can be noticed that the uncertainty on the starting position is equal to 0, whereas the landmark have an infinite uncertainty due to the fact that the exploration of the environment has not started yet.
At this point, it is necessary to define the functions appearing in the EKF algorithm. As already mentioned, the prediction step builds on top of the motion model. If a differential drive robot is considered, the update depends on the rotation and translation velocity

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \underbrace{\begin{pmatrix} x \\ y \\ \theta \end{pmatrix} + \begin{pmatrix} -\frac{v_t}{\omega_t}\sin\theta + \frac{v_t}{\omega_t}\sin(\theta+\omega_t\Delta t) \\ \frac{v_t}{\omega_t}\cos\theta - \frac{v_t}{\omega_t}\cos(\theta+\omega_t\Delta t) \\ \omega_t\Delta t \end{pmatrix}}_{g_{x,y,\theta}(u_t,(x,y,\theta)^T)} \tag{3.11}$$

The apostrophe indicates the updated state vector.
In this way the update is on three dimensions, but the state vector is 2N+3 dimensions, so a function that maps the update to the needed dimensions is required. This adjustment is performed by matrix F_x^T defined in such way that the overall function g only affects the motion and not the landmarks.

$$\begin{pmatrix} x' \\ y' \\ \theta' \\ \vdots \end{pmatrix} = \underbrace{\begin{pmatrix} x \\ y \\ \theta \\ \vdots \end{pmatrix} + \underbrace{\begin{pmatrix} 1 & 0 & 0 & \underbrace{0\dots0}_{2Ncols} \\ 0 & 1 & 0 & 0\dots0 \\ 0 & 0 & 1 & 0\dots0 \end{pmatrix}^\top}_{F_x^T} \begin{pmatrix} \frac{v_t}{\omega_t}\sin\theta + \frac{v_t}{\omega_t}\sin(\theta+\omega_t\Delta t) \\ \frac{v_t}{\omega_t}\cos\theta - \frac{v_t}{\omega_t}\cos(\theta+\omega_t\Delta t) \\ \omega_t\Delta t \end{pmatrix}}_{g(u_t,x_t)} \tag{3.12}$$

Next, the covariance matrix needs to be updated through the previous covariance matrix, the uncertainty on the motion, represented by R_t and the function G, which is the Jacobian of the update function g.

We can then update the covariance matrix

$$\begin{aligned}\bar{\Sigma}_t &= G_t \Sigma_{t-1} G_t^\top + R_t \\ &= \begin{pmatrix} G_t^x & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} \Sigma_{xx} & \Sigma_{xm} \\ \Sigma_{mx} & \Sigma_{mm} \end{pmatrix} \begin{pmatrix} (G_t^x)^\top & 0 \\ 0 & I \end{pmatrix} + R_t \\ &= \begin{pmatrix} G_t^x \Sigma_{xx} (G_t^x)^\top & G_t^x \Sigma_{xm} \\ (\Sigma_{xm})^\top (G_t^x)^\top & \Sigma_{mm} \end{pmatrix} + R_t\end{aligned}$$

The two off-diagonal elements impact of the uncertainty of the robot's pose on the uncertainty of the landmarks.

Now that the prediction step is completed it is time for the correction step to be performed.

To compute the Kalman gain

$$K_t = \bar{\Sigma}_t H_t^\top (H_t \bar{\Sigma}_t H_t^\top + Q_t)^{-1}, \tag{3.13}$$

the only unknown quantity is H, since Q is a user defined function representing the uncertainty on the observation, so it is based on the sensor's properties. H is the Jacobian of the observation function h, which computes the predict observation. We thus need to find h.

The range-bearing observation is modeled as the following vector

$$z_t^i = (r_t^i, \phi_t^i)^\top \tag{3.14}$$

In the case of a landmark that has not been observed yet, the initialization can be performed as follows

$$\begin{pmatrix} \bar{\mu}_{j,x} \\ \bar{\mu}_{j,y} \end{pmatrix} = \begin{pmatrix} \bar{\mu}_{t,x} \\ \bar{\mu}_{t,y} \end{pmatrix} + \begin{pmatrix} r_t^i \cos(\phi_t^i + \bar{\mu}_{t,\theta}) \\ r_t^i \sin(\phi_t^i + \bar{\mu}_{t,\theta}) \end{pmatrix} \tag{3.15}$$

The location of the landmark is then inferred using the estimate location of the robot plus the relative measurement. Once both estimate of the landmark position and the robot's pose are known, it is possible to compute the Euclidean distance

$$\delta = \begin{pmatrix} \delta_x \\ \delta_y \end{pmatrix} = \begin{pmatrix} \bar{\mu}_{j,x} - \bar{\mu}_{t,x} \\ \bar{\mu}_{j,y} - \bar{\mu}_{t,y} \end{pmatrix}, \tag{3.16}$$

whose quadratic form is referred to as

$$q = \delta^\top \delta \tag{3.17}$$

The expected orientation is then computed as

$$\hat{z}_t^i = \begin{pmatrix} \sqrt{q} \\ \operatorname{atan2}(\delta_y, \delta_x) - \bar{\mu}_{t,\theta} \end{pmatrix} = h(\bar{\mu}_t), \tag{3.18}$$

and its Jacobian as

$$H_t^i = \frac{\partial h(\bar{\mu}_t)}{\partial \bar{\mu}_t} \tag{3.19}$$

As a matter of fact, this matrix is obtained by derivation over only for the five non-zero variables $(x, y, \theta, m_{j,x}, m_{j,y})$, so its dimension is reduced.
The reduced Jacobian results

$$\text{low } H_t^i = \frac{1}{q}\begin{pmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 & \sqrt{q}\delta_x & \sqrt{q}\delta_y \\ \delta_y & -\delta_x & -q & -\delta_y & \delta_x \end{pmatrix} \tag{3.20}$$

As discussed earlier, the dimensions can be adjusted through a matrix $F_{x,j}$

$$H_t^i = \text{low } H_t^i F_{x,j}, \qquad F_{x,j} = \begin{pmatrix} 1 & 0 & 0 & 0\cdots0 & 0 & 0 & 0\cdots0 \\ 0 & 1 & 0 & 0\cdots0 & 0 & 0 & 0\cdots0 \\ 0 & 0 & 1 & 0\cdots0 & 0 & 0 & 0\cdots0 \\ 0 & 0 & 0 & 0\cdots0 & 1 & 0 & 0\cdots0 \\ 0 & 0 & 0 & \underbrace{0\cdots0}_{2j-2} & 0 & 1 & \underbrace{0\cdots0}_{2N-2j} \end{pmatrix} \tag{3.21}$$

At this stage, all the elements have been defined and the algorithm can be implemented.

Figure 3.3 shows how the covariance matrix gets populated and how the uncertainty around landmarks reduces as the platform explores the environment.

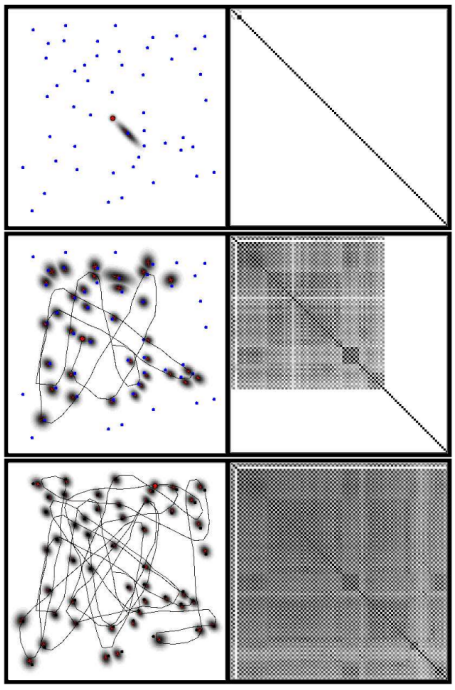


Figure 3.3: Covariance matrix evolution.

In Figure 4.7 one can see how as new landmarks are detected, they are initialized with maximum uncertainty, which decreases monotonically over. Note that the

uncertainty does never converge to zero: asymptotically, the covariance associated with a landmark's position estimate will approach to the initial covariance of the vehicle's location estimate

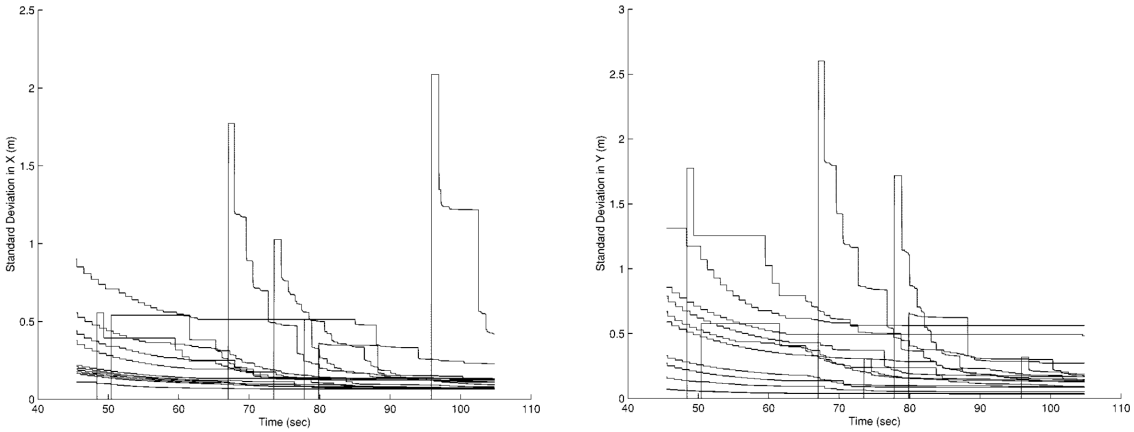


Figure 3.4: Uncertainty of the landmarks position over time.

Graph-Based Optimization Techniques

This solution builds on top of the notion of treating subsequent robot poses as nodes of a graph, whose arcs represent spatial constraints based on the odometry data or sensor observation between the poses. Clearly, these should be considered soft constraints being the odometry measurements affected by noise.
If, after a period of roaming, the robot encounters a previously visited location. In the SLAM problem this is referred to as **loop closure**. In this case new arcs, or constraints, will be created between not subsequent poses.
A graph-based SLAM approach aims to find the node configuration that minimizes the error introduced by the constraints. The SLAM problem reduces to a minimization problem that can be solved through *Least Squares* optimization.
The following pictures, depict the evolution of both the map and the pose-graph.
At first, the robot explores the environment generating a map using the scans coming from the sensors and the odometry measurements. This method carries many criticalities in terms of errors, as can be clearly seen in the upper left side of Figure 3.5 where the two environments look pretty similar. They are indeed the same room, yet they appear as separated.

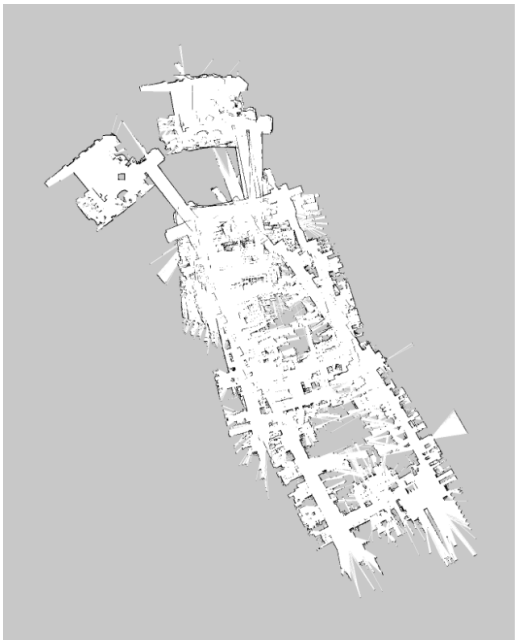


Figure 3.5: Map obtained by odometry.

If the pose-graph is superimposed on the map, as shown in Figure 3.6, straight connections between corresponding poses become apparent.

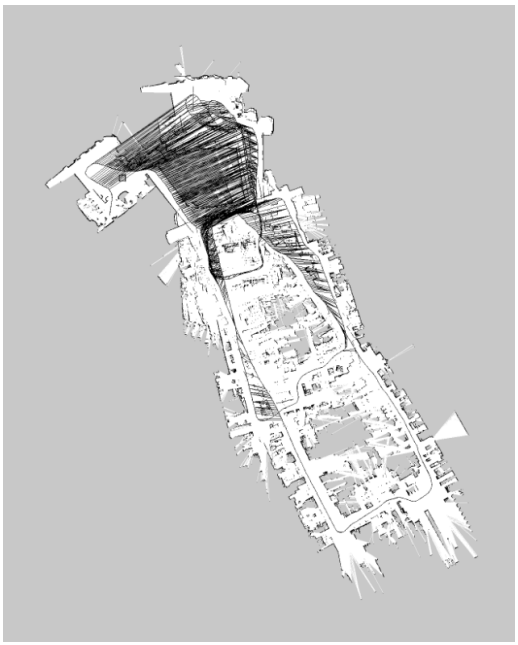


Figure 3.6: Visualization of the pose graph over the map.

By minimizing the errors on the graph, the poses can be rearranged, as illustrated in Figure 3.7, in order to obtain a more accurate map, see Figure 3.8

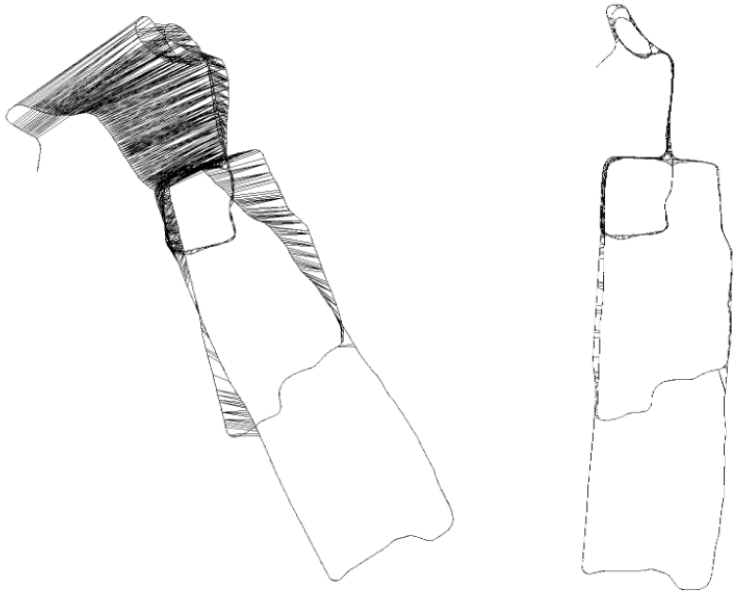


Figure 3.7: Visualization of the pose graph before after pose rearrangement.

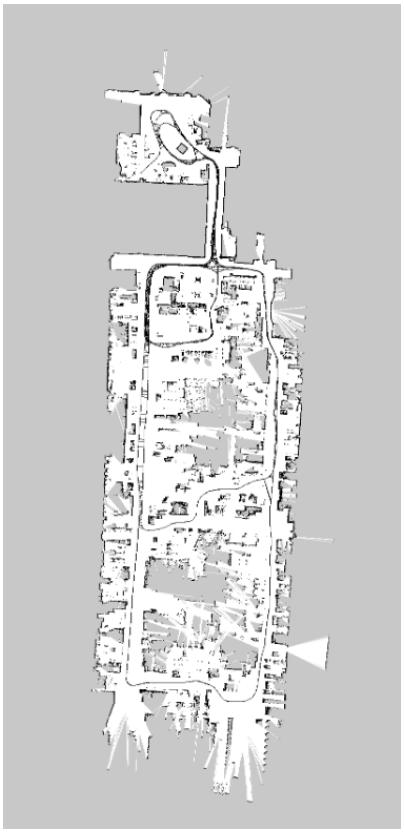


Figure 3.8: Map reconstruction via pose-graph optimization.

Taking the observation and turning them into constraints is what is referred as *front-end*, whereas the graph optimization is called here the *back-end*. These two

parts are in constant communication between each other.
The SLAM problem approached with a graph-based method can be then visualized as follows.

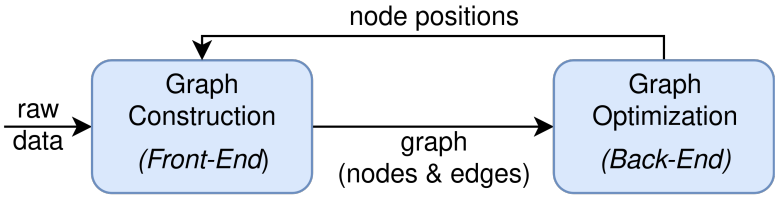


Figure 3.9: Graph-based SLAM scheme.

As previously mentioned, there are two cases in which an edge is created:

- between consecutive poses, connected by constraints derived from odometry measurements: x_i and x_{i+1};
- between non-consecutive poses that observe the same portion of the environment: x_i and x_j.

In the second case, x_i and x_j can be connected with an arc that represents the relative transformation of x_j as observed from x_i. This transformation is expressed using homogeneous coordinates (see Appendix), which allow rigid body transformations to be represented by a single matrix.
Each observation is affected by some level of noise, which is collected in the so-called *Edge Information Matrix* Ω_{ij}. Its entries encode the confidence in the observation: the larger the value the more more weight the corresponding observation has in the optimization process. Typically, sensor measurements are considered more reliable than odometry data, and this is reflected in the respective information matrices.
In addition, the geometry of the environment may cause the information to be anisotropic. For example, in long corridors the uncertainty is higher along the main axis of the corridor and smaller in the orthogonal direction.
The goal is then to minimize the following sum:

$$x^* = \arg\min_x \sum_{i,j} e_{ij}^\top \Omega_{ij} e_{ij}, \tag{3.22}$$

which appears to be suitable for a LS minimization.
The state vector contains all the successive poses, each one representing a node in the graph.

$$x^\top = (x_1^\top \; x_2^\top \; \dots \; x_n^\top) \tag{3.23}$$

The error function for a single constraint is given by

$$e_{ij}(x_i, x_j) = \text{t2v}(Z_{ij}^{-1}(X_i^{-1} X_j)), \tag{3.24}$$

whereas for the whole state vector one has

$$e_{ij}(x) = \text{t2v}(Z_{ij}^{-1}(X_i^{-1}X_j)), \tag{3.25}$$

where $X_i^{-1}X_j$ represents the estimated relative transformation of pose x_j with respect to x_i, Z_{ij} denotes the observed transformation between i and j, and $\text{t2v}(\cdot)$ (*transformation to vector*) converts the transformation matrix into a vector of minimal error $(\Delta x, \Delta y, \Delta\theta)$, which can be used in the optimization process (e.g., Gauss-Newton).
Notice that if the observation perfectly matches the graph configuration, i.e., $Z_{ij} = (X_i^{-1}X_j)$, the resulting error becomes zero. In this case the argument of $\text{t2v}(\cdot)$ is the identity matrix, which yields a zero error vector.
The next step involves linearizing the error function around an initial guess x through a Taylor expansion.

$$e_{ij}(x + \Delta x) \simeq e_{ij}(x) + J_{ij}\Delta x \tag{3.26}$$

with J_{ij} the Jacobian matrix of the error function

$$J_{ij} = \frac{\partial e_{ij}(x)}{\partial x} \tag{3.27}$$

The shape of the Jacobian matrix plays a crucial role in the efficiency of solving the state estimate problem. In particular, each error term $e_{ij}(x)$ depends only on the states x_i and x_j and not on the entire state vector.

$$e_{ij}(x) = e_{ij}(x_i, x_j) \tag{3.28}$$

As a direct consequence, most entries of the Jacobian are zero. Non-zero elements appear only in the rows corresponding to x_i and x_j, leading to a sparse block structure.
The obtained Jacobian has then the following structure:

$$J_{ij} = \begin{pmatrix} 0 & \cdots & 0 & \underbrace{\dfrac{\partial e_{ij}(x_i)}{\partial x_i}}_{A_{ij}} & 0 & \cdots & 0 & \underbrace{\dfrac{\partial e_{ij}(x_j)}{\partial x_j}}_{B_{ij}} & 0 & \cdots & 0 \end{pmatrix} \tag{3.29}$$

The two blocks A_{ij} and B_{ij}, each composed by three elements, are the only non-zero entries. This property induces sparsity in the Jacobian, which enables the an efficient resolution of real world SLAM problems. Indeed, the Jacobian appears in the computation of the vector b and the matrix H, which are required for solving the linear system:

$$b^\top = \sum_{ij} b_{ij}^\top = \sum_{ij} e_{ij}^\top \Omega_{ij} J_{ij} \tag{3.30}$$

$$H = \sum_{ij} H_{ij} = \sum_{ij} J_{ij}^\top \Omega_{ij} J_{ij} \tag{3.31}$$

The sparsity of the Jacobian leads to sparsity also of H, as shown in Figure 3.10.

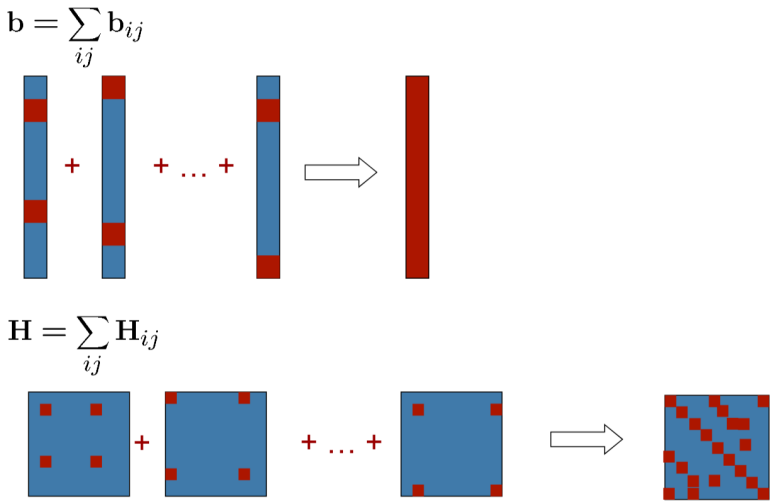


Figure 3.10: Illustration of sparsity: the non-zero elements are colored in red, while the zeros in blue.

While all elements in b are eventually be filled, since every node is involved in at least one constraint, the structure of H reflects the graph connectivity. Each edge contributes with four non-zero blocks: two diagonal terms, which reinforce the information on the involved poses, and two symmetric off-diagonal terms, which encode the relative constraint between the connected nodes (e.g., from odometry or loop closures). As a result, H can be then be seen as the adjacency matrix of the pose-graph. Typically, the larger the environment the sparser the matrix.

To summarize, an edge ij contributes only to the i^{th} j^{th} block of b_{ij} and to the blocks ii, jj, ij, and ji of H_{ij}. This structure promotes sparsity in the system, since each edge affects only a small portion of the full matrix. As a result, the linear system can be efficiently constructed by summing up the contributions of all edges, and solved using sparse optimization methods such as Sparse Cholesky decomposition or conjugate gradients.
The overall optimization loop looks like this:

Algorithm 2 optimize(x)

1: **while** !converged **do**
2: $(\mathbf{H}, \mathbf{b}) \leftarrow$ buildLinearSystem(x)
3: $\Delta x \leftarrow$ solveSparse($\mathbf{H}\Delta x = -\mathbf{b}$)
4: $x \leftarrow x + \Delta x$
5: **end while**
6: **return** x

A key consideration to address is the initialization of the first node.
Let's consider a trivial example with two nodes, x_1 and x_2, in a one-dimensional

world, and an observation z_{12} between the two claiming that they are one meter away apart. The state vector is initialized with both nodes in the origin:

$$x_o = (x_{1,0}, x_{2,0})^\top = \begin{pmatrix} 0 & 0 \end{pmatrix} \tag{3.32}$$

The measurement and its associated information matrix are:

$$z_{12} = 1 \tag{3.33}$$

$$\Omega = 2 \tag{3.34}$$

The error is then given by the difference between the observation and the predicted relative distance:

$$e_{12} = z_{12} - (x_2 - x_1) = 1 = 1 - (0 - 0) = 1 \tag{3.35}$$

The Jacobian is computed as the derivative of the error function over x_1 and x_2

$$J_{12} = \begin{pmatrix} 1 & -1 \end{pmatrix} \tag{3.36}$$

From this, b and H can be computed:

$$b_{12}^\top = e_{12}^\top \Omega_{12} J_{12} = \begin{pmatrix} 2 & -2 \end{pmatrix} \tag{3.37}$$

$$H_{12}^\top = J_{12}^\top \Omega_{12} J_{12} = \begin{pmatrix} 2 & -2 \\ -2 & 2 \end{pmatrix} \tag{3.38}$$

At this point, the update Δx is obtained by solving the linear system:

$$\Delta x = -H_{12}^{-1} b_{12} \tag{3.39}$$

However, H is singular. This problem arises because the constraint is relative between both nodes, meaning that no absolute position of the pair is defined. To solve this issue, a reference node is fixed by adding a prior constraint on its position. For example, anchoring x_1 at the origin yields:

$$H = \begin{pmatrix} 2 & -2 \\ -2 & 2 \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \tag{3.40}$$

this way, $\Delta x_1 = 0$ and the system can be solved:

$$\Delta x = -H^{-1} b_{12} \tag{3.41}$$

$$\Delta x = \begin{pmatrix} 0 & 1 \end{pmatrix}^\top \tag{3.42}$$

Particle Methods

With highly efficient microprocessors, in recent years the third SLAM paradigm, that is particle filters, have become more popular. Here the posterior is seen as a set of particles, each one representing a concrete guess of the true value of the state. Under mild conditions, this method converges to the true posterior as the number of guesses, or particles, tends to infinity. Having a substantial number of particles is not an issue in the SLAM problem, but this is generally increases exponentially the complexity of the problem.
To address this issue a trick was introduced in the literature is **FastSLAM**. To better understand the algorithm, let's consider the simplified point-landmark example. At any time instant, the algorithm maintains K particles of the form:

$$X_t^{[k]}, \quad \mu_{t,1}^{[k]}, \ldots, \mu_{t,N}^{[k]}, \quad \Sigma_{t,1}^{[k]}, \ldots, \Sigma_{t,N}^{[k]} \tag{3.43}$$

where [k] identifies the sample. Each particle then contains:

- a sample path $X_t^{[k]}$;
- a set of N 2-dimesional Gaussians with means $\mu_{t,n}^{[k]}$ and variances $\Sigma_{t,n}^{[k]}$, one for each landmark in the environment.

where n indexes the landmark. In total one has K particles that have K path samples and KN Gaussians, each one modeling every landmark for each particle.
FastSLAM initialization involves setting each particle to the robot's known starting position and clearing the map. Each particle is then updated as described below:

- As new odometry readings are received, for each particle new location variables are generated stochastically. Those location particles are generated from a distribution determined by the motion model:

  $$x_t^{[k]} \sim p(x_t \mid x_{t-1}^{[k]}, u_t) \tag{3.44}$$

  where $x_{t-1}^{[k]}$ is the previous location. This probabilistic sampling step can be easily performed for any robot with computable kinematics

- A soon as new measurement z_t is received, two things happen: firstly, the probability of the new measurement z_t is computed by FastSLAM. Let n be the index of the sensed landmark, then the desired probability is:

  $$w_t^{[k]} := \mathcal{N}(z_t; \mid x_t^{[k]}, \mu_{t,n}^{[k]}, \Sigma_{t,n}^{[k]}) \tag{3.45}$$

  where the factor $w_t^{[k]}$ is defined as the *importance weight*, as it expresses how coherent is a particle with the new sensor measurement. Even in this case a normal distribution $\mathcal{N}$ is adopted, but this time it is evaluated on the observation z_t. Subsequently, the weights associated to all the particles are normalized

such that they sum up to one.
At this point, FastSLAM proceeds with a phase called *resampling*: some particles are extracted by the current set, with the possibility of picking the same particle multiple times, with a probability depending on the normalized importance weight. In other words, the particles which better justify the observation have a better chance of being preserved in the new set.
Finally, for each remaining particle, the mean $\mu_{t,n}$ and covariance $\Sigma_{t,n}$ are updated depending on the measurement z_t. This update follows the standard rules of the EKF.

FastSLAM, although it copes with a complex problem such SLAM, is incredibly easy to implement. The main operations, such as sampling from the motion model, evaluation the importance weights, especially with Gaussian noise, and the update of the particle filter, are all pretty straightforward.
The true strength of FastSLAM is that it can approximate well the full SLAM posterior, thanks to three fundamental techniques: Rao-Blackwellization, condi-tional independence, and resampling. Rao-Blackwellization is a technique used in statistics that allows to improve the sampling efficiency: instead of generating particles from a joint distribution between trajectory and map, only the the robot's trajectory is sampled and to each particle an analytic representation of the conditioned map is attached, often in Gaussian form. This approach reduces the filter's variance and improves the estimates' quality.
Another important intuition in this context is that once the new robot's trajectory is known, the estimates of the landmarks become conditionally independent. This means that it is not necessary to represent the map as a unique big Gaussian distribution with correlations between all landmarks: it is instead possible to divide it in many smaller gaussian distributions, one for each landmark, sensibly facilitating the computation. This property is shown in Figure 3.11

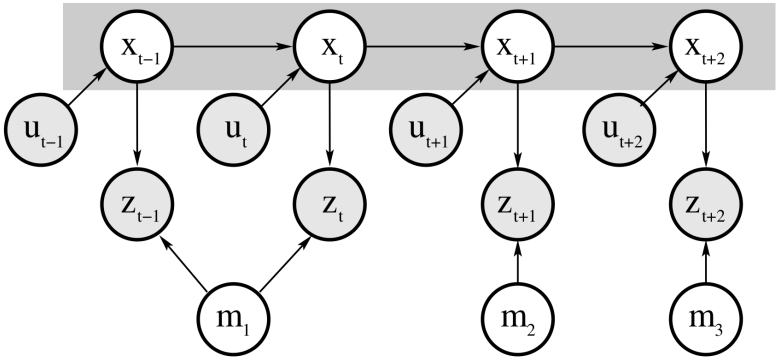


Figure 3.11: Knowing the trajectory allows decoupling landmark between each other.

Another advantage of FastSLAM is the possibility of making an hypothesis on the single particle, instead of making it on the whole filter. This allows the algorithm particularly robust in complex and uncertain evironments. Moreover, FastSLAM

is extremely efficient computationally, since it can be build through advanced tree methods to represent the maps. This way the updates can be computed logarithmically with respect to the map dimension and linearly with respect to the number of particles. This makes it suitable also for large scale scenarios.
There is an important version of FastSLAM that is based on occupancy grids, where Gaussians are replaced by grids that represent the probability of occupancy of the space. Here, each particle keeps its map and it is evaluated based on the consistency between the measurements and the map itself. When a loop closure occurs, the resampling picks the particles with the most consistent maps, allowing to obtain better representation of the environment.
Finally, FastSLAM inspired other variants such as DP-SLAM and methods based on ancestry trees, which further improve the efficiency in managing grid-based maps. Thanks to its flexibility and easy imlementation, FastSLAM has become one of the most valued algorithms adopted in mobile robotics.

Paradigm comparison

The three main paradigms enclose most of the research in the SLAM field. The EKF-based approach is limited in scalability due to its high computational cost. Some extensions help mitigating this limitation by building local submaps, which essentially resemble graph-based methods.

Graph-based approaches tackle the full SLAM problem and, inherently, are not online algorithms. They model the environment as a sparse graph of soft constraints, corresponding to a motion or measurement event. Thanks to efficient nonlinear optimization algorithms for sparse graphs, graph-based SLAM has become the preferred choice for large-scale offline mapping.
Moreover, data association integrates naturally into this framework, and several search techniques exist to identify the most suitable correspondences. Online variants also exist, which progressively reduce the amount of robot poses maintained in the graph.

Particle filter methods, such as FastSLAM, avoid some of the criticalities arising from inter-feature correlations that affect the EKF. By sampling robot poses, individual landmarks become independent and uncorrelated.
FastSLAM thus represents the posterior distribution through a set of sampled poses and independent Gaussian estimates for each landmark. This approach is computationally more efficient, requiring linear-logarithmic time instead of the quadratic time of the EKF, and easily handles uncertain data association.
However, the number of particles required can become very large, especially in complex environments with multiple loops. Extensions using occupancy grids instead of Gaussian landmarks have demonstrated state of the art performance in large-scale mapping.

3.2 ROS 2 framework and *slam_ toolbox*

In the ROS 2 ecosystem, the SLAM problem is handled by the *slam_ toolbox* package ideated by American robotics engineer Steve Macenski. This tool offers a wide range of functionalities for 2D mapping, providing a wide range of functionalities both for real-time and offline situations. It is built upon a graph-based SLAM approach, and inherits its algorithm from the *open_ karto* package, integrating advanced optimization techniques through Google's *Ceres* solver. The latter is used as dynamic plugin and supports both Sparse Bundle Adjustment and data structures such as KD-Tree for efficient scan matching and loop-closure. Furthermore, the package is implemented as lifecycle node, allowing modular and controlled management of its life cycle within the ROS 2 architecture. Thanks to its reliability and flexibility, slam_toolbx is now the default SLAM package in the Nav2 framework, becoming de facto the standard for localization and mapping in 2D environments.

3.2.1 Operation modes

The *salm_ toolbox* package offers three main operation modes, each designed to address different requirements in terms of localization and mapping.

- **Synchronous SLAM**: this mode offers a buffered processing of the measurements, with processing of the data. It is particularly suitable for creating offline maps, where the priority is the quality and accuracy of the map rather than the update speed. This approach allows for greater accuracy in the construction of the pose-graph, making ideal for static environments and planned mapping sessions.

- **Asynchronous SLAM**: here the processing occurs in real-time, updating the map only when the update criteria are met and the previous measurements have been completed. This approach guarantees that the system never falls behind the data flow, even in the presence of complex loop closures. It is suitable in dynamic scenarios where real time localization is crucial, with a trade-off in map completeness.

Both these modes support *multi-session SLAM*, i.e., the ability to resume a previous session to keep refining and expanding the map. Through pose-graph serialization and raw data, it is possible to manually manipulate nodes, rotate maps and facilitate complex loop closures, even in large-scale environments.

- **Pure localization**: This mode does not update the existing map, it focuses solely on estimating the robot's pose in an already available map. However, slam_toolbox introduces an innovative mechanism called *elastic pose-graph deformation*, which uses a dynamic buffer of recent measurements to temporarily adapt to variations in the environment. New scans are integrated into the pose-graph with new constraints, improving localization in the presence of moved

objects or new characteristics. As time goes by, the measurements will eventually "expire" and get removed, resetting the graph to its original state for that region. This approach ensures robust localization in dynamic environments.

An interesting side effect is that in absence of an existing map, the localization mode can be used as odometry based on LiDAR, exploiting the local buffer to estimate the robot's future motion in potentially infinite spaces.

3.2.2 Package configuration in ROS2

The slam_toolbox architecture perfectly integrates in the ROS2 ecosystem, exploiting several fundamental inputs for localization and map construction. The main parameters are:

- **2D laser scans (LiDAR)**: used for matching between scans and pose transformations estimate;
- **Odometry**: provides an estimate of the robot's motion between measurements;
- **TF**, **IMU** and, optionally, a **map server**: used to improve localization and consistency between reference frames.

The typical workflow consists in a comparison of the new laser scan with the previous to determine the variation of the position. To this extent, a scan matching algorithm, boosted by a KD-Tree structure, that allows an efficient correspondence between measurements, is adopted. Each new measurement is represented as a node in a graph, where the arcs represent the transformations between poses, building, as previously discussed, the so called *pose-graph*.
When a loop closure occurs new constraints are added to reduce the cumulative error and subsequently the graph is globally optimized through a Ceres solver, which improves the overall consistency of the map.
From the optimized structure an *occupancy grid* is generated, whose map can be saved through an appropriate service call.
During execution, the SLAM node is subscribed to the following topics:

- `/scan`: LiDAR scans;
- `/odom`: odometry estimate (if available);
- `/tf`: transformations between frames such as `base_link`, `odom` and `map`.

The constructed map is then published on the topic `/map`. In Figure 3.12 the rqt_graph of slam_toolbox is reported.

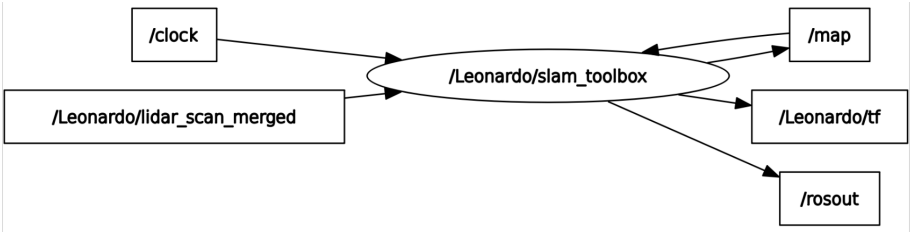


Figure 3.12: In this `rqt_graph` it is reported the slam node and the key topics. Note the presence of namespacing. The reason behind the necessity of a namespace will be discussed in the next chapter.

Moreover, there are some important parameters that need to be configured, which are the following:

- `use_sim_time`: to use the simulated time of Gazebo;
- `odom_frame`: odometry reference frame;
- `map_frame`: frame with respect of which the map gets constructed;
- `base_frame`: robot's frame, it is either `base_link` or `base_footprint`;
- `mode`: operating mode, mapping or localization.

In this thesis, slam_toolbox has been used in its asynchronous mode.

Chapter 4

Methodology and development

In this chapter, the methodology proposed as a solution to the automatic map updating problem in dynamic environments is presented. The goal is to describe the entire workflow—from test map generation to the system's integration within ROS 2, highlighting both the design choices and the corresponding software implementations.
This pipeline can be divided into three main components:

- **Map generation of test scenarios** in simulated environment.
- **Difference detection** between maps through computer vision algorithms.
- **ROS 2 integration of the system** through the realization of dedicated nodes.

4.1 Maps construction

As first step, a set of bi-dimensional maps was created via *slam_ toolbox* to test and validate the algorithm of difference detection. For this extent, the 3D Gazebo model of the I3P first floor was used.
Each map was created by manually guiding the robot through the aisles of the I3P building using the `Teleop_twist` node and activating *slam_ toolbox*. The first map was considered as the clean one (Figure 4.1), while the others have been progressively corrupted with a variety of elements added in the Gazebo model. In particular, the following kinds of variations have been considered:

- **new walls**: to simulate permanent modifications to the structure of the building, such as the closing of passages;
- **new static obstacles**: additional architectural elements, located in strategic positions, with the scope of verifying whether the system can detect stable and not transitory modifications;

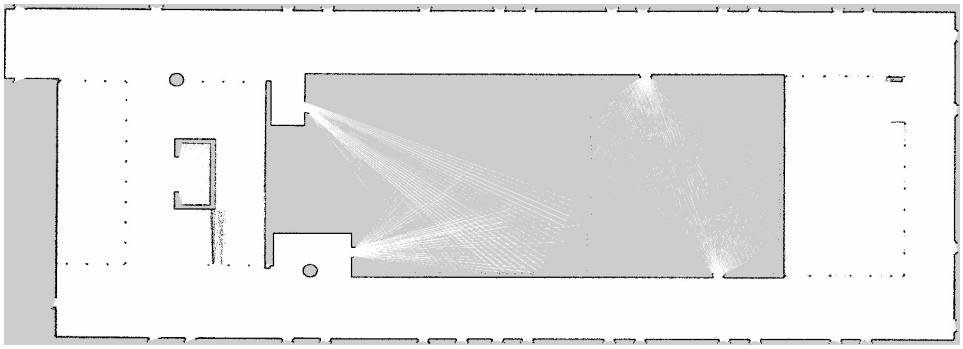


Figure 4.1: Clean map.

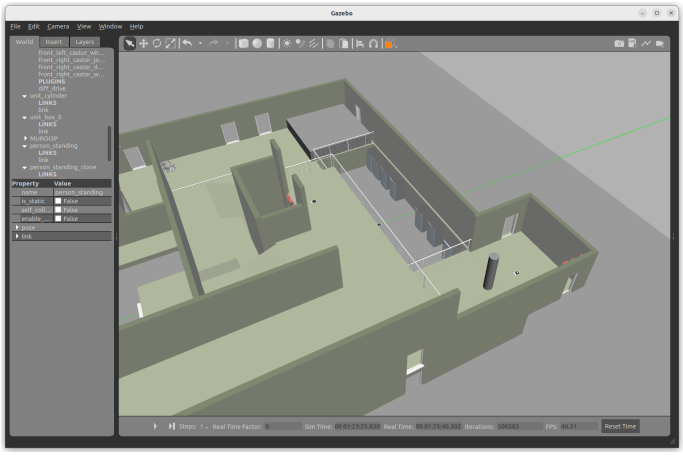


Figure 4.2: Permanent new wall and new column added to the world.

As it can be seen in Figure 4.5 a completely new wall in the top left side and two new columns in the right corridor were added.

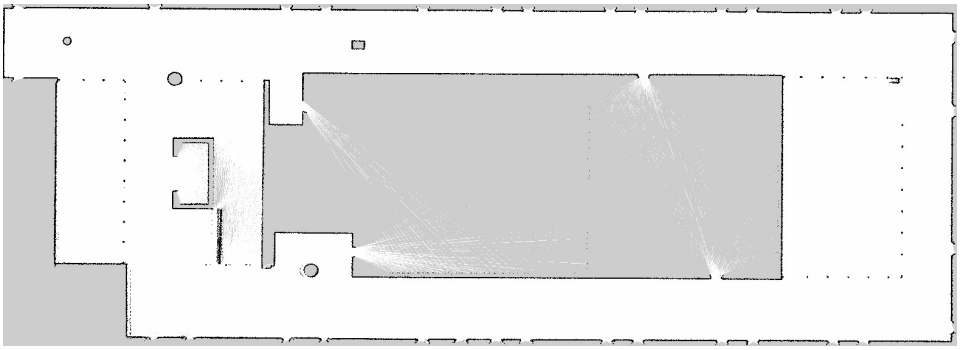


Figure 4.3: Map with structural differences referred to Figure 4.2.

- **temporary obstacles**: persons, chairs and other mobile objects, introduced to tune the algorithm to filter variations not to be added.

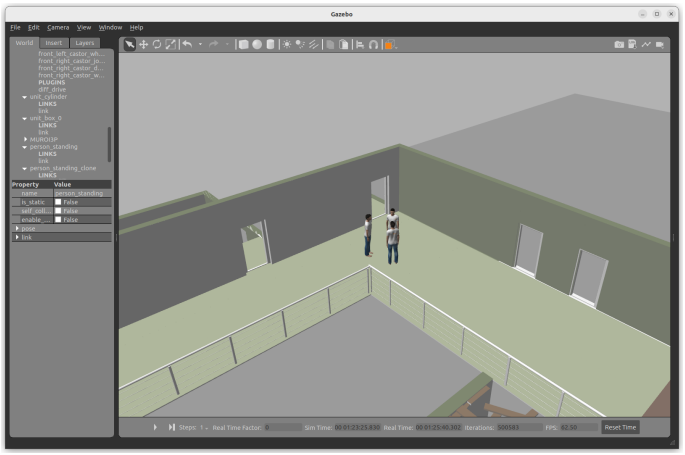


Figure 4.4: Standing people added to the world.

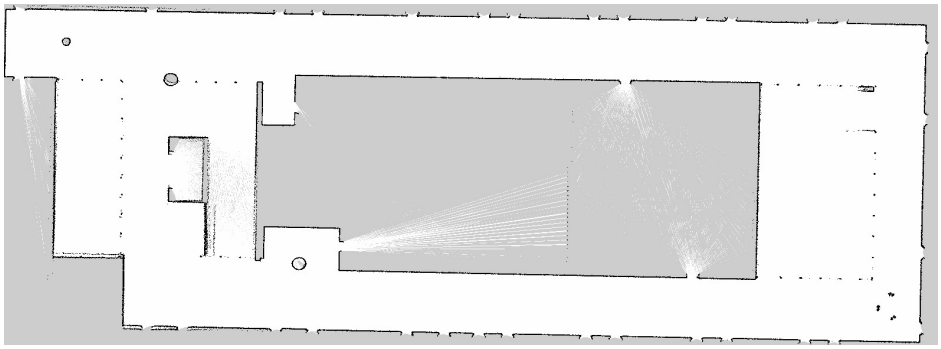


Figure 4.5: Map with temporary differences referred to Figure 4.4.

The maps where acquired with the aid of RViz, which allowed to verify the correct map generation procedure and visualize in real time the difference between each map version. The possibility of visualizing sensor data and occupancy grids in RViz has been crucial to monitor the behavior of the system and detect potential failures in the map construction phase.

To save the generated maps, the slam_toolbox package provides the service `SaveMap`, which can be called directly from the command line (CLI).
When invoked, this service provides two output files:

- a **.pgm file**, which contains the map in image format;
- a **.yaml file**, which stores the associated metadata necessary for correctly interpreting the image by RViz or Nav2.

An example of the metadata file is reported below:

```
image: mappa_I3P_pulita.pgm
mode: trinary
resolution: 0.05
origin: [-12.9, -65.8, 0]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.25
```

The `resolution` parameter specifies the size of each pixel in meters: in this case, 0.05 m/pixel. This means that every pixel in the occupancy grid corresponds to a real-world square of side 5 cm, thus defining the metric granularity of the map. A lower resolution value results in a more detailed but also more memory-intensive map.
The `mode` field indicates the representation strategy adopted for the occupancy grid. With the value `trinary`, each cell can assume one of three possible states: *occupied*, corresponding to obstacles, *free*, that is the navigable space, and *unknown*, representing unexplored or uncertain areas.
The classification is determined by two *normalized grayscale thresholds*:

- `free_thresh` $= 0.25 \rightarrow$ pixels with a value below 0.25 are considered free space;
- `occupied_thresh` $= 0.65 \rightarrow$ pixels with a value above 0.65 are considered occupied;
- values between 0.25 and 0.65 are classified as unknown.

The field `negate` specifies the adopted color convention. When set to 0 (false), the standard ROS convention applies: white for free space, black for occupied, and gray for unknown.
Together, the spatial distribution of these pixels generates the so-called *Occupancy Grid*, a 2D discrete representation of the environment widely used in mobile robotics. This format provides a convenient abstraction that allows planners and navigation systems to distinguish between traversable and non-traversable regions in a straightforward way.
Finally, the `origin` field defines the pose of the map within the global `map` reference frame. The three values represent, respectively, the X and Y coordinates of the bottom-left corner of the image in meters, and the orientation (yaw angle) of the map with respect to the world frame. This ensures that the occupancy grid is correctly placed in RViz and aligns consistently with the coordinate system used by the navigation stack.
This metadata, combined with the `.pgm` file, allows ROS 2 to interpret the image not as a simple picture but as a structured world model, suitable for path planning, localization, and autonomous navigation.

4.2 Difference Detection Python Code

Once the corrupted maps were obtained, they were compared against the reference (clean) map in order to highlight all structural differences. This task was implemented in Python by exploiting the libraries *OpenCV* and *NumPy*. OpenCV (Open Source Computer Vision Library) is an open-source library that offers efficient tools for image and video processing, broadly adopted in robotics and artificial vision applications. The images are loaded as NumPy arrays, which makes them easier to manipulate as matrices of pixel values.

From the perspective of computer vision, an image can be regarded as a two-dimensional array of pixels, each pixel characterized by one or more channels encoding the color standard.
In common RGB images, instead, each pixel is defined by three color channels (Red, Green, Blue) plus an optional α channel encoding transparency. However, it is important to note that OpenCV internally uses the BGR convention instead of RGB. The images representing the maps are in `.pgm` (Portable Gray Map) format, in which each element of the array is represented by an unsigned integer (unit8) ranging from 0 to 255, indicating the gray level of the pixel.

4.2.1 Binary Difference Computation

Conceptually, difference detection consists in converting the two compared images into binary (black and white) representations and then perform a *pixel-wise subtraction*. If the subtraction returns 0, the corresponding pixels match, and no difference is detected. Conversely, if the subtraction results in 255 (white), a discrepancy is identified.

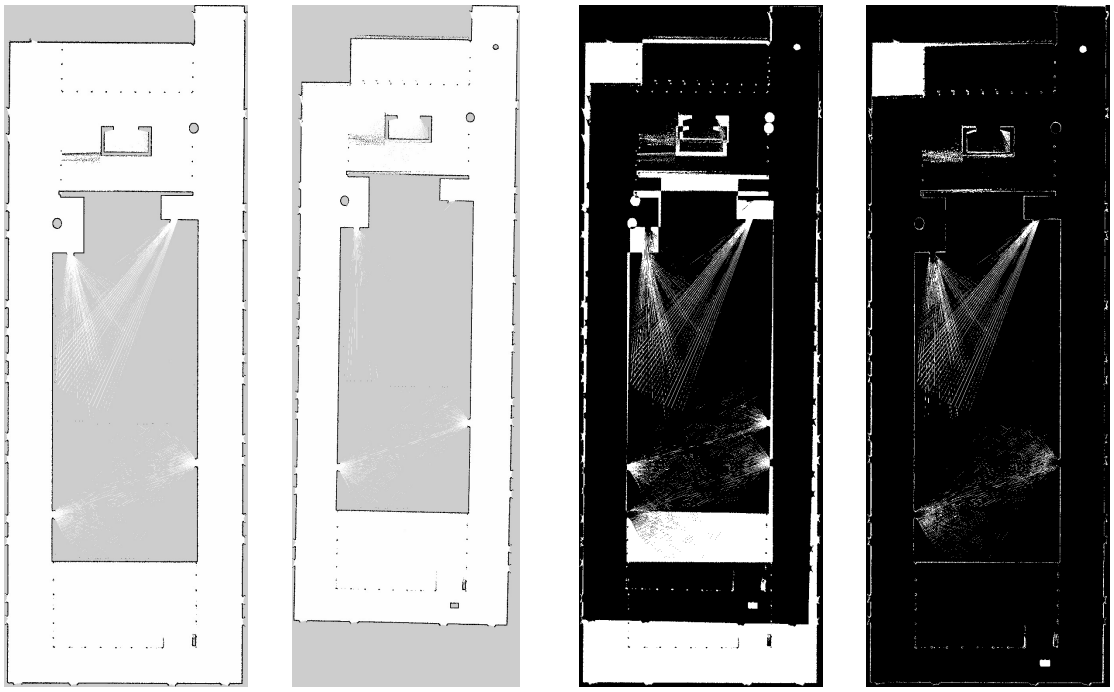
In the case of occupancy grids, the binarization process requires particular attention: both obstacles and unknown regions are typically represented as dark values, black and gray respectively. Therefore, an appropriate thresholding strategy is applied to correctly differentiate navigable (white) against non-navigable (black/gray) areas.

4.2.2 Image Registration and Alignment

For this procedure to work correctly, the two maps must represent the same physical area with precise pixel-level alignment. This requirement is non-trivial, since occupancy grid maps produced by the *SaveMap* service are subject to noise—originating both from LiDAR measurements (e.g., beams passing through narrow spaces such as doorways and windows) and from discretization artifacts.

To address the issue of misalignment, the literature offers several image registration techniques based on feature extraction and matching. Among the most relevant there are SIFT, SURF, and ORB, which work by detecting distinctive keypoints and matching them across images. The main differences lie in the methods to detect and describe these features.

In this work, the chosen method was **ORB** (Oriented FAST and Rotated BRIEF). ORB is fully open-source, computationally efficient, and provides robust performance even when maps are rotated or partially distorted. Once features are extracted and matched, a filtering stage based on **RANSAC** (Random Sample Consensus) is applied to remove outliers, and a **homography matrix** is computed. This rigid transformation aligns the corrupted map with the reference one, ensuring pixel-level consistency.



(a) Clean map. (b) Corrupted map. (c) No registration. (d) Registered.

Figure 4.6: Evolution of the difference image through the filtering pipeline.

4.2.3 Noise Filtering

After alignment, the raw difference image is subject to noise. LiDAR measurements can introduce spurious points in unexplored areas, while slight contour mismatches of real obstacles may also be falsely flagged as differences.

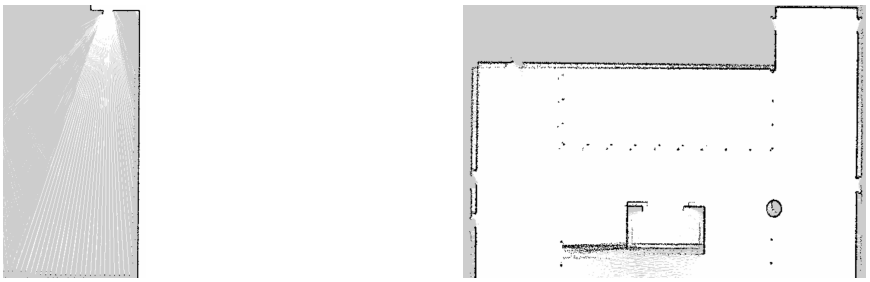
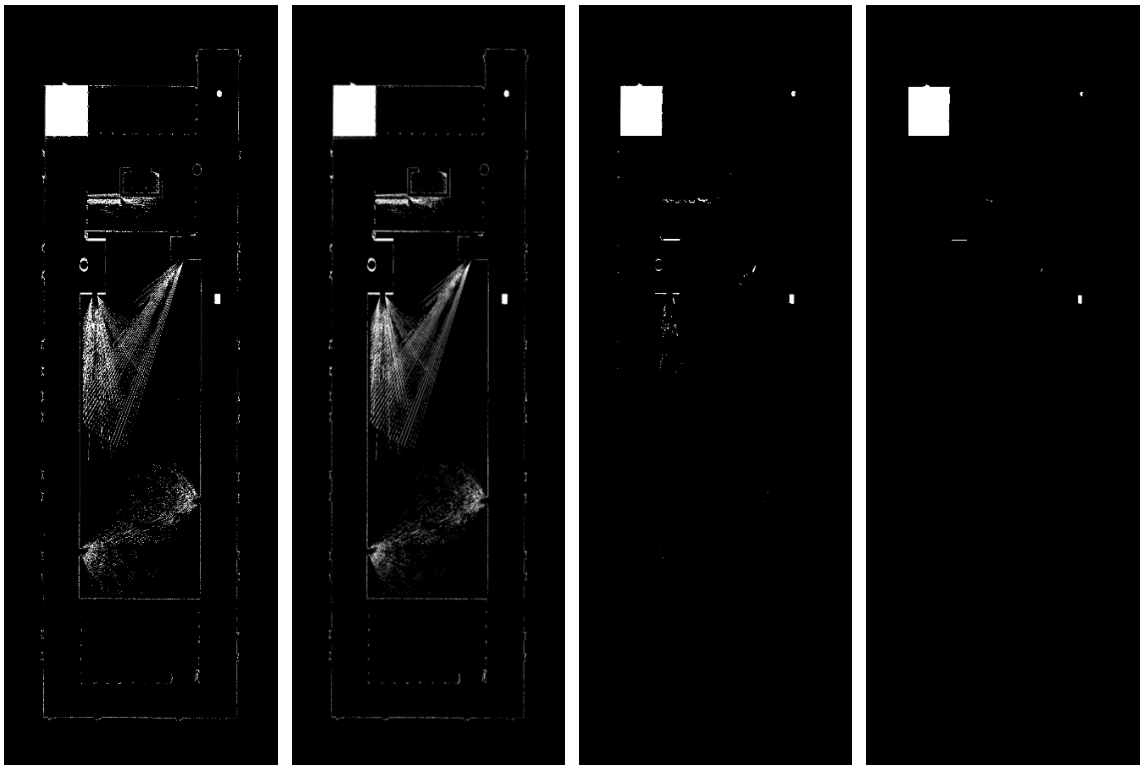


Figure 4.7: Example of noise sources.

Since these should not be considered candidate structural changes, a *filtering stage* was introduced.
The first step applies a **Gaussian blur** with a kernel of 5×5 pixels, which smooths the image by averaging each pixel with its neighbors. The blurred image is then thresholded to obtain a binary representation, a process that already removes most of the noise associated with the LiDAR rays. **Dilation** was also implemented to eliminate further small speckles of noise and reinforce relevant structures.



(a) Raw difference (b) Gaussian blur (c) Thresholding (d) Filtered result

Figure 4.8: Evolution of the difference image through the filtering pipeline.

4.2.4 Clustering of Detected Differences

After filtering, the candidate differences are analyzed using the **DBSCAN** (Density-Based Spatial Clustering of Applications with Noise) algorithm, implemented in the *Scikit-Learn* Python library. DBSCAN is particularly well suited for this task because it groups together spatially dense regions while discarding isolated noisy pixels.

The output of DBSCAN provides clusters of pixels corresponding to candidate changes in the map. For each cluster, a *centroid* is computed, which defines the center of a bounding box, whose dimensions are derived from the cluster's spatial extent, providing both:

- a visual feedback on the detected difference,

- and the metric coordinates of the discrepancy, usable by higher-level systems.

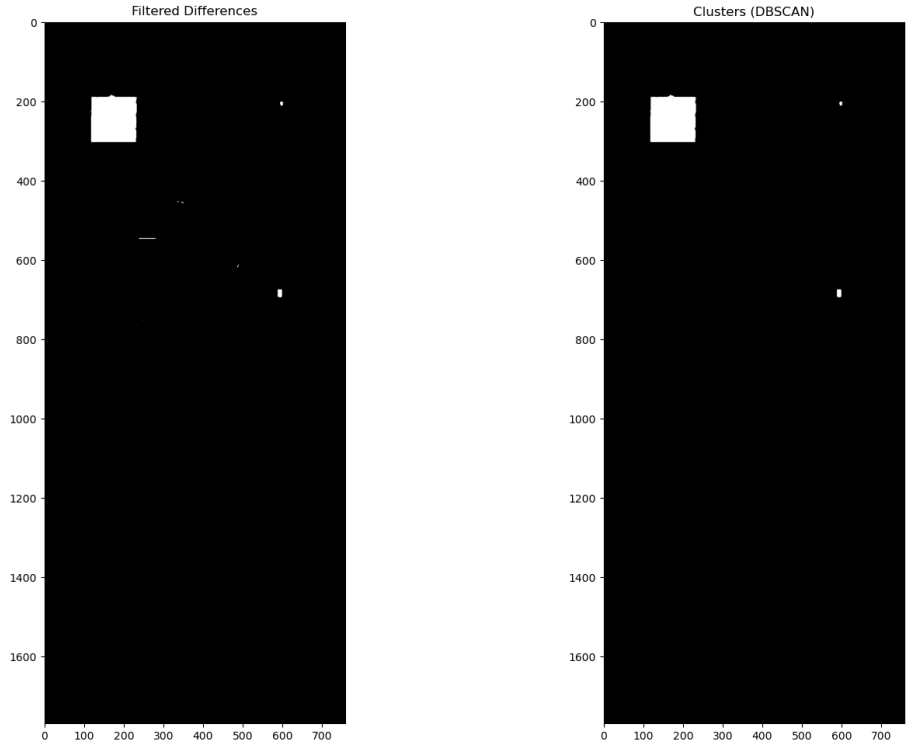


Figure 4.9: Effects of clustering with DBSCAN.

The final result of this process is illustrated in Figure 4.10, where bounding boxes highlight the structural differences between the clean and corrupted maps.

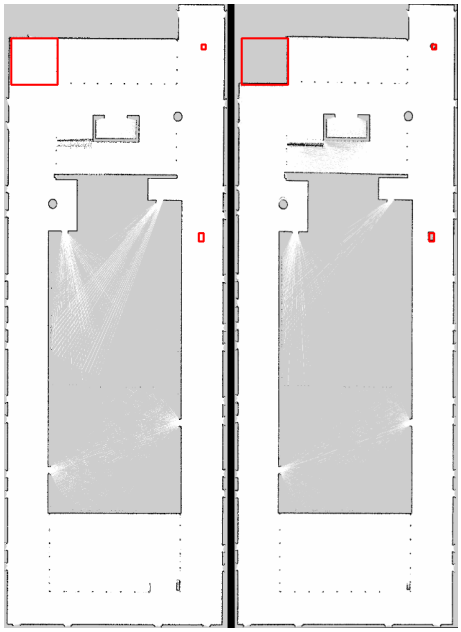


Figure 4.10: All the differences between clean and corrupted map are detected.

4.3 Local Differences

Once the results obtained on full-map comparisons were satisfactory, the focus shifted towards the identification of *local differences*. In this case, the comparison is not performed between two complete maps, but between the global map used by the navigation stack and a smaller, locally generated map obtained from *slam_ toolbox*. This adds an additional layer of complexity: image registration techniques alone are not sufficient to guarantee a correct alignment. In large-scale environments such as airports, multiple areas may share similar geometric features, and a feature-based approach could mistakenly align different portions of the environment.

To overcome this limitation, the system is integrated with ROS 2, exploiting the robot poses to place the partial map over the corresponding part of the global map, so that the difference finder algorithm is applied to the correct area.

For the system's integration with ROS, four topics from the simulation were fundamental:

- `/map`: the local map generated by *slam_ toolbox* as the robot roams the environment;
- `/Leonardo/pose`: the robot's pose expressed in the *slam_ toolbox* map frame;
- `/Leonardo/map`: the global map used by the Navigation2 stack for path planning and localization;
- `/Leonardo/amcl_pose`: the robot's pose in the Nav2 map frame, estimated through the AMCL (Adaptive Monte Carlo Localization) algorithm.

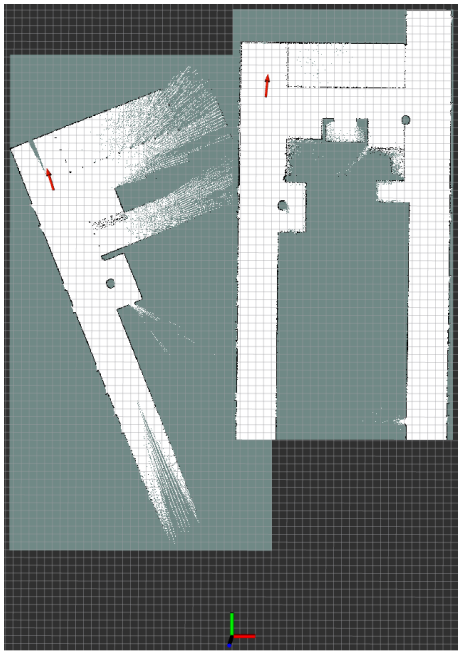


Figure 4.11: Nav2 map (left) and *slam_ toolbox* map (right) as visualized in RViz.

It is important to note that all the topics above are defined with respect to the default RViz reference frame, denoted as `/map`. However, this symbolic reference should not be confused with the `/map` topic published by *slam_ toolbox*.

As illustrated in Figure 4.11, the global Nav2 map and the local *slam_ toolbox* map do not initially overlap in space, nor do their pose estimates share the same orientation.

4.3.1 Pose-based Alignment of Maps

To correctly overlap the two maps, the relative rotation and translation with respect to the `/map` reference frame need to be computed.

Angular alignment. The angular offset θ is computed as the difference between the robot orientations in the two maps, which are expressed in the `/map` frame. Since orientations in ROS 2 are represented as quaternions, they are first converted into Euler angles $(\text{yaw}, \text{pitch}, \text{roll})$. The heading offset is then given by:

$$\theta = \text{yaw}_{Nav2} - \text{yaw}_{slam}. \tag{4.1}$$

Translational alignment. Let $\mathbf{w}$ be the displacement vector between the robot poses expressed in the two frames:

$$\mathbf{w} = \begin{bmatrix} x_{Nav2} - x_{slam} \\ y_{Nav2} - y_{slam} \end{bmatrix}. \tag{4.2}$$

ROS and OpenCV adopt different coordinate conventions:

- in ROS, the map origin is placed at the **bottom-left corner**;
- in OpenCV, the origin is at the **top-left corner**.

Hence, a vertical offset proportional to the map height (in pixels) must be added when converting coordinates. Denoting by h the map height (in pixels) and by r the map resolution, the conversion from ROS to OpenCV coordinates is expressed as:

$$(x_{CV},\, y_{CV}) = \frac{1}{r}(\, x_{ROS},\, h + y_{ROS}\,). \tag{4.3}$$

Once both origins $O_{CV_{Nav2}}$ and $O_{CV_{slam}}$ are expressed in the OpenCV frame, the point P, obtained by translating $O_{CV_{slam}}$ of $\mathbf{w}$, represents the position of the *slam_ toolbox* origin if the SLAM and Nav2 poses are aligned. Finally the translation vector ξ is then defined as:

$$\boldsymbol{\xi} = O_{CV_{Nav2}} - P. \tag{4.4}$$

In other words, $\mathbf{w}$ describes the relative displacement of the robot between the two maps, while $\boldsymbol{\xi}$ represents the transformation required to align their spatial origins.

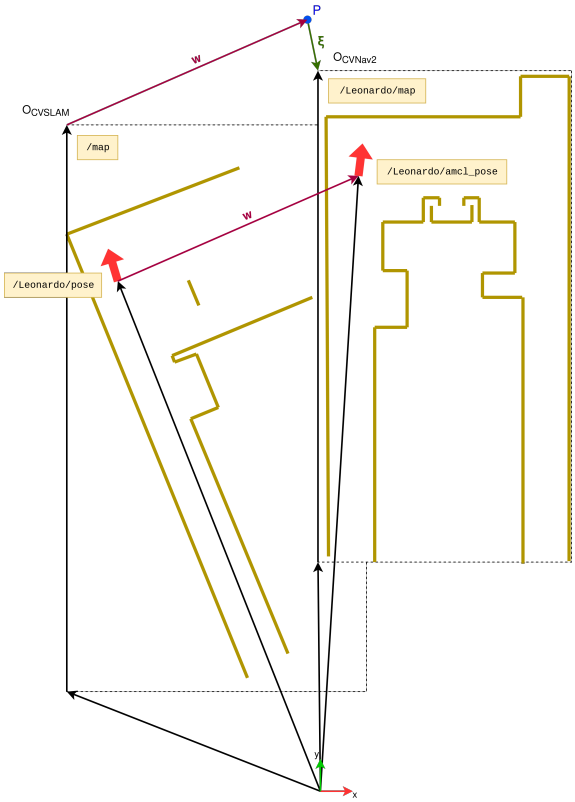


Figure 4.12: Vector representation for the alignment of local and global maps.

4.4 Map Server Pipeline

Through this procedure, the local map generated by *slam_ toolbox* can be correctly overlaid on the global Nav2 map using the robot's pose information. This integration bridges the gap between computer vision-based image registration and ROS2 pose estimation, enabling the difference finder algorithm to work reliably even in large, repetitive environments such as airports, where pure feature-based matching could otherwise fail.

In this section, the centralized architecture for map management and processing (**Map Server**) is presented. The basic idea is to move part of the processing outside of ROS, simulating a cloud-based system locally on a computer. This approach allows greater control over the data flow, ensuring more organized storage of maps and associated parameters.

The process follows a well-defined sequence of operations. First, the entire *navigation stack* is launched via the *sedia_ bringup* launch file, which loads the world in Gazebo and the RViz graphical interface. Once the environment has been initialized, a *goal pose* is set in RViz, thus activating autonomous navigation of the robot. During motion, another launch file is executed to enable the *slam_ toolbox* configuration in asynchronous mode and to start the mapping phase. At the same time, the `map_saver` node, responsible for handling map data, is activated. After

a predefined time interval, another trigger launches the *ShutdownSaver* node, in charge of saving the maps and poses related to both Nav2 and *slam_ toolbox*.

4.4.1 ShutdownSaver Node

The *ShutdownSaver* node represents the core element for orderly closure and data saving at the end of mapping. Its logic is based on subscribing to two key topics: `/Leonardo/amcl_pose` and `/Leonardo/pose`. Both provide `PoseWithCovarianceStamped` messages, whose contents are stored via subscription callbacks.

A control loop constantly checks the availability of the two poses; only when both are acquired is the `shutdown_sequence` function activated. This function is divided into three main blocks:

1. the first performs a call to the Nav2 `SaveMap` service, used to save both the SLAM-generated map and the Nav2 map, with the possibility of specifying destination directories. The output files include both the image in `.pgm` format and the description in `.yaml`, organized in separate folders;

2. the second introduces a 5-second pause to ensure correct file writing, subsequently checking their size and readability with OpenCV. This step is necessary since map creation by *slam_ toolbox* can take variable times;

3. the third block saves the poses in a text file, converting the coordinates from quaternions to heading angles to make the data more immediately interpretable.

Once these operations are completed, the node is terminated, ensuring the consistency of the saved data.

4.4.2 Image processing

In parallel with the launch of the *ShutdownSaver* node, a chain of scripts for map processing and comparison is activated: `watchdog_folder.py`, `watchdog_transform.py`, and `watchdog_rotation.py`. The interaction among these modules realizes a continuous flow of monitoring, transformation, and analysis of the acquired maps.

watchdog_folder.py

The first script monitors the `slam_shutdown_data` folder, which contains the files created by the saving node. The `SlamShutdownHandler` class, derived from `FileSystemEventHandler` is defined to automatically react to the creation of new files. An observer from the *watchdog* library is initialized, staying active indefinitely until manual interruption.

At startup, the handler registers already existing files in a set called `already_seen` to distinguish them from newly created ones. In addition, a working directory called `watchdog_data` is created (or cleaned up if already existing), ensuring a clean environment at every execution.

Each time a new file is detected, the `on_created()` method is invoked. If the file is new, the system waits until its size is acceptable, thus avoiding the copying of empty or incomplete files, and then processes it with the `process_new_file()` function. This function copies the file into the working directory with its metadata and increments a counter. Once the counter reaches the desired number of flies, the `watchdog_transform.py` script is launched as a subprocess, while the original files are moved to an archive directory called `slam_shutdown_data_old`, yielding the possibility of further offline examination.

watchdog_transform.py

The next script processes the barely saved maps, retrieving the parameters required for geometric alignment. After verifying that all files are available, it extracts:

- the `.pgm` images, used to convert dimensions from pixels to meters through the *resolution* parameter;
- the `.txt` files containing the robot poses in the two maps.

The retrieved information includes the origin of the SLAM map and the Nav2 map in the RViz reference frame, together with the robot poses relative to both. These data are translated into the OpenCV reference system and used to compute alignment points. The heading angles are then used to determine the angular offset between the two poses, as previously shown for correct overlapping.

watchdog_rotation.py

The core of the differential analysis is the `watchdog_rotation.py` script, which manages the comparison between maps. After loading the images and alignment parameters, the SLAM map is superimposed on the Nav2 map, expanding the canvas to avoid cropping. A *Region of Interest* (ROI) is then defined, either as a bounding box around the pose or along the robot's trajectory, thus focusing only on relevant areas and excluding non-significant peripheral zones.

To refine alignment, ORB is applied, provided that the images contain a sufficient number of features.
Once registration is completed, the pipeline proceeds with difference detection between the two maps just as it was for the global case. The identified areas are enclosed in bounding boxes, with their coordinates computed in the Nav2 map reference frame. All results are finally stored in a `.yaml` file enriched with a timestamp, along with annotated images and highlighted regions of interest.

This sequence enables not only the detection of discrepancies between the updated SLAM map and the one used for navigation, but also the systematic storage of detected differences, thus laying the foundations for a subsequent incremental map update phase.

Figure 4.13 reports the pipeline proposed to simulate a cloud system.

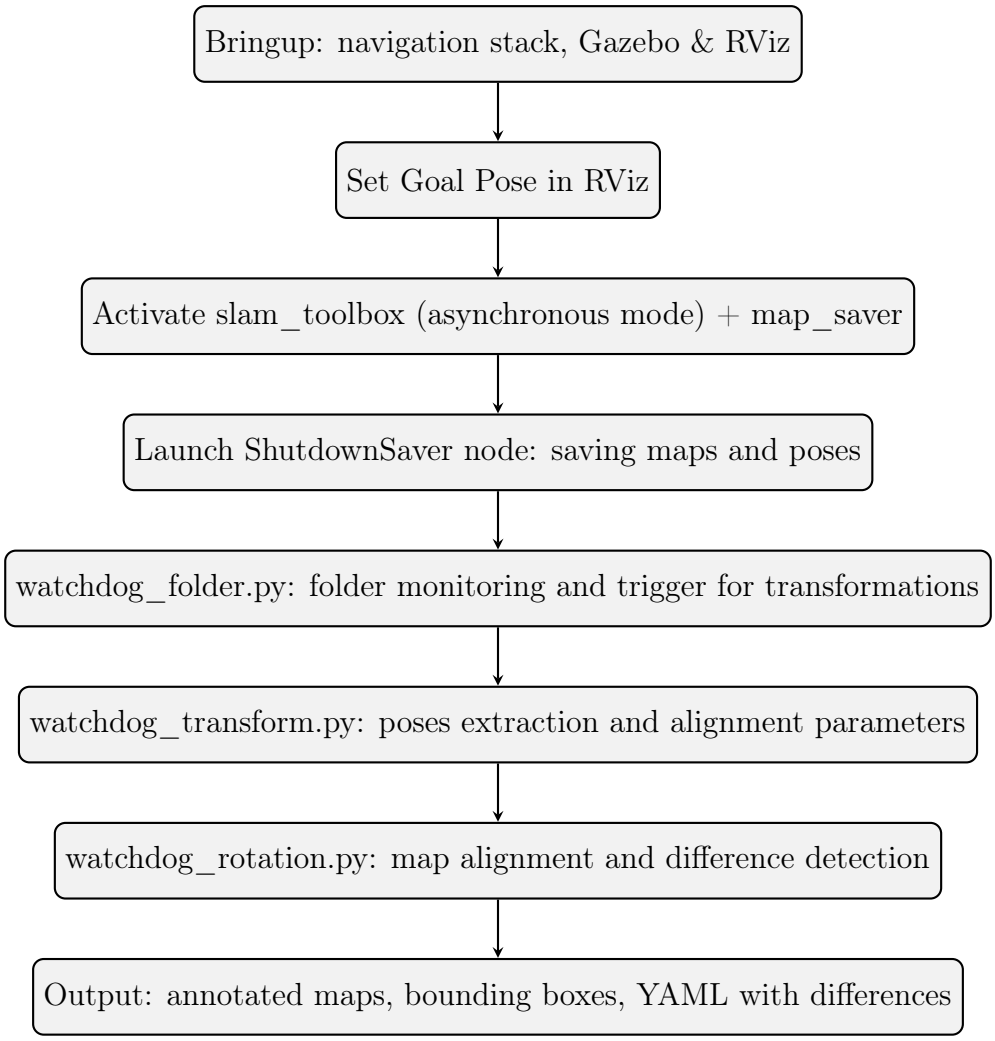


Figure 4.13: Centralized pipeline for map management and difference detection.

Chapter 5

Experiments and Results

The purpose of this chapter is to evaluate the performance of the proposed system for automatic map updating in dynamic environments. After presenting the methodology and implementation details in the previous chapters, this section focuses on the experimental validation, which was carried out in simulation through ROS 2, *slam_ toolbox*, Gazebo, and RViz.

The experiments were designed with two main objectives:

- to verify the capability of the system to correctly identify structural differences between maps, both at global and local scales;
- to assess the robustness of the approach along the entire robot trajectory.

Different simulation scenarios were created to reproduce realistic conditions that may occur in large indoor spaces. Permanent modifications (e.g., walls, columns, fixed obstacles) were introduced to test the sensitivity of the difference detection algorithm.

The main tools exploited were:

- **ROS 2 Humble** for communication and data management;
- **slam_toolbox** for real-time mapping;
- **Nav2** for navigation and costmap handling;
- **Gazebo** + **RViz** for simulation and visualization.

To improve repeatability and reduce computational load, recorded *rosbags* were used to reproduce navigation sessions under controlled conditions. Three representative cases were then analyzed to validate the overall pipeline, together with an evaluation of different ROI selection strategies.

5.1 Region of Interest (ROI) Selection Strategies

An important feature of the proposed pipeline is the definition of the Region of Interest (ROI) for map comparison, in order to reduce outliers detection. Different strategies were analyzed, each with specific advantages and limitations:

- **Fixed bounding box around the robot:** Simple approach that considers a square or rectangular area around the robot's pose. Despite its simplicity, it often considers unexplored or irrelevant regions, leading to spurious detections.
- **Trajectory-based ROI:** The ROI follows the robot's path, focusing only on regions actually traversed. This method reduces false positives behind the robot but may still include unobserved zones ahead of it.
- **Yaw-based ROI:** The ROI is oriented according to the robot's heading, considering primarily the area in front of the platform. This solution proved particularly effective in our experiments, since it excludes zones that are likely to remain unobserved.
- **Adaptive ROI:** A more advanced strategy in which the ROI's size and shape vary dynamically depending on feature density or local map alignment confidence. Even though potentially more accurate, it rises computational cost.

In the context of this work, the yaw-based ROI was adopted as a good compromise between robustness and efficiency, significantly reducing false positives in partially explored environments.

5.2 Case Studies

Two scenarios were designed to evaluate the proposed system.

5.2.1 Scenario 1: Permanent Obstacles

In the first scenario, a column and a wall were added in the Gazebo world.

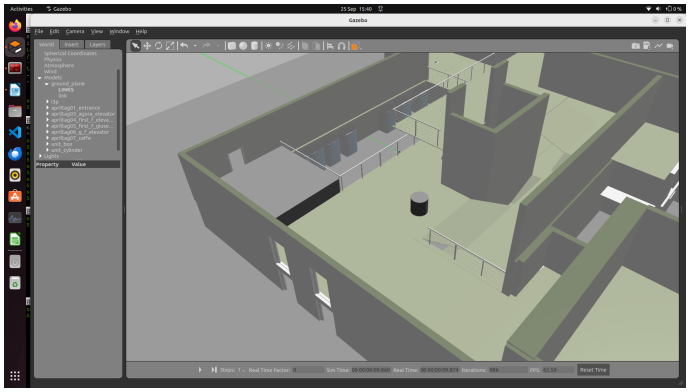


Figure 5.1: Permanent obstacles added in the Gazebo world.

As shown in Figure 5.2, these obstacles initially appear only as LiDAR readings in the costmap before being integrated into the map representation.

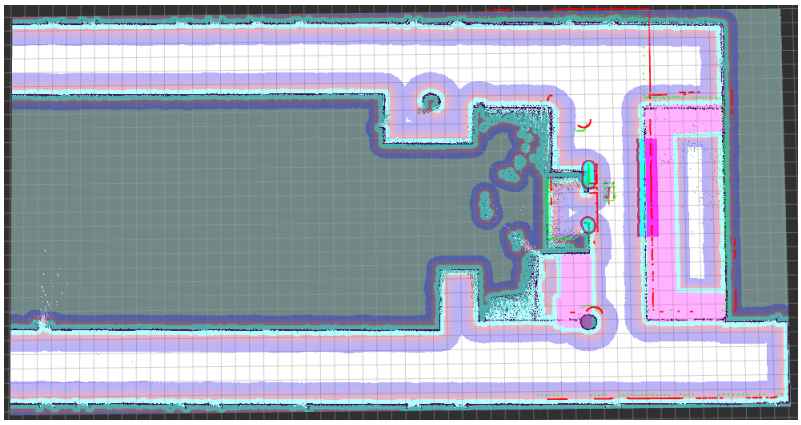


Figure 5.2: Costmap visualization in RViz.

Figure 5.3 shows the maps saved through the service call within the *ShutdownSaver* node.

(a) Nav2 map. (b) SLAM map.

Figure 5.3: Maps from Nav2 and SLAM. The red dot represents the robot's position.

Figures 5.4 and 5.5 show the comparison results when the ROI is defined as a square centered on the robot's pose. This approach can be problematic, as it may

include unexplored regions in front of the robot. When these areas are compared with the navigation map, false differences are likely to appear, as visible on the right side of Figure 5.5.

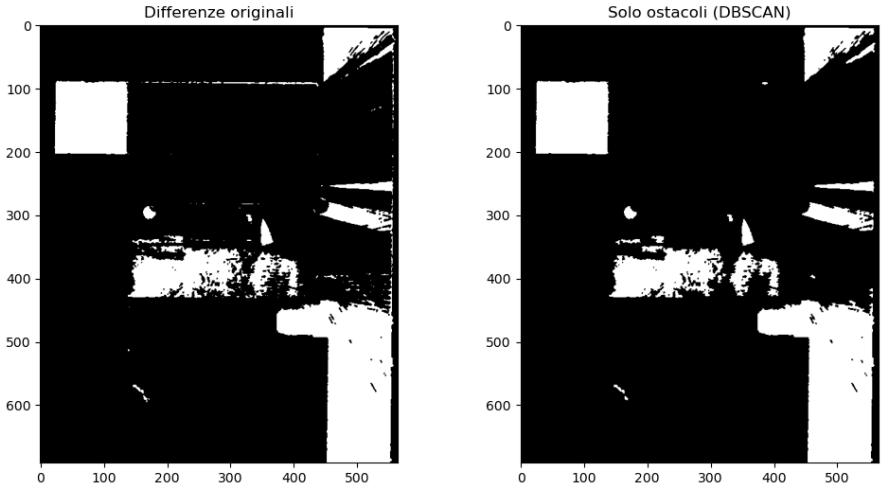


Figure 5.4: Before and after clusterization (case 1).

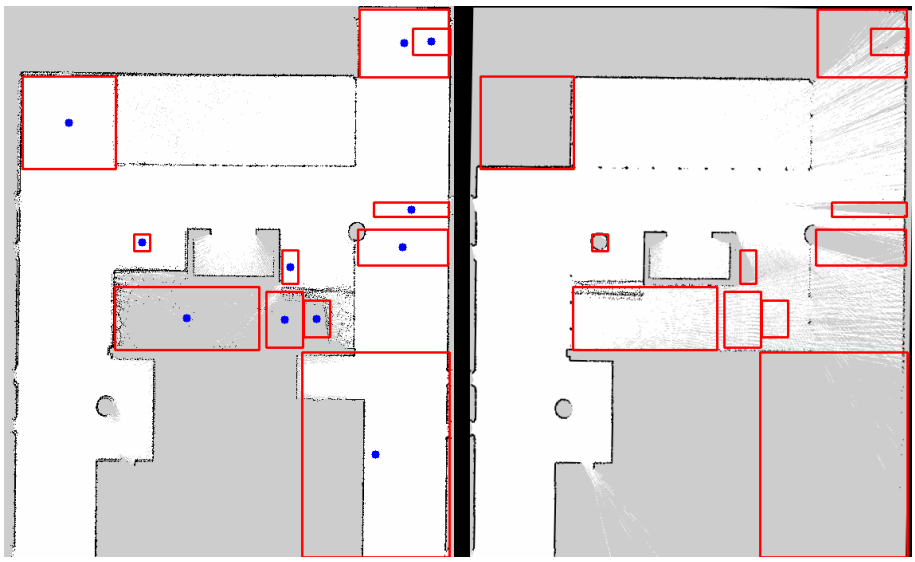


Figure 5.5: Detected differences and centroids (case 1).

Figure 5.6 highlights which differences should be considered valid and which are outliers. In this case, the robot's heading points to the left of the page, and it becomes evident that restricting the ROI in the opposite direction filters out false detections while retaining meaningful structural changes.

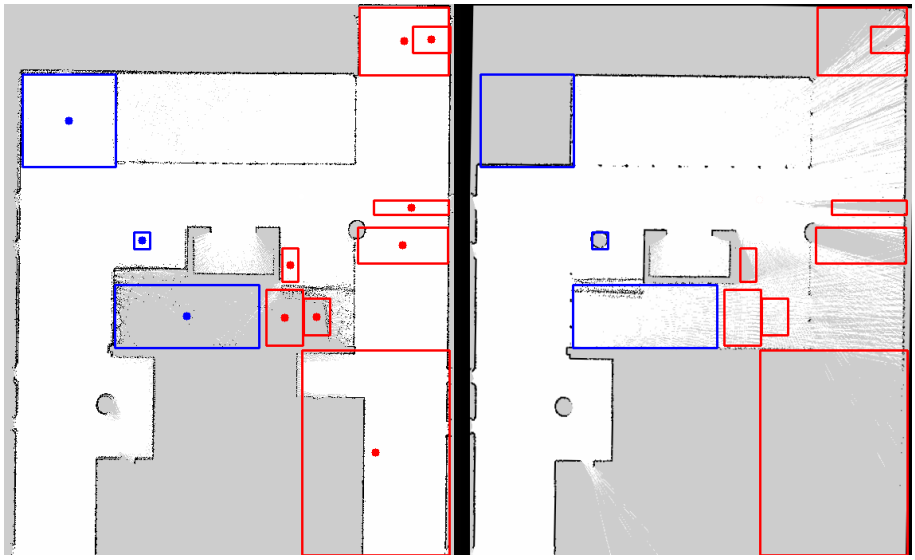


Figure 5.6: Outlier filtering: red boxes indicate false detections, blue boxes highlight true differences.

The result after applying the yaw-based ROI reduction is shown in Figure 5.7.

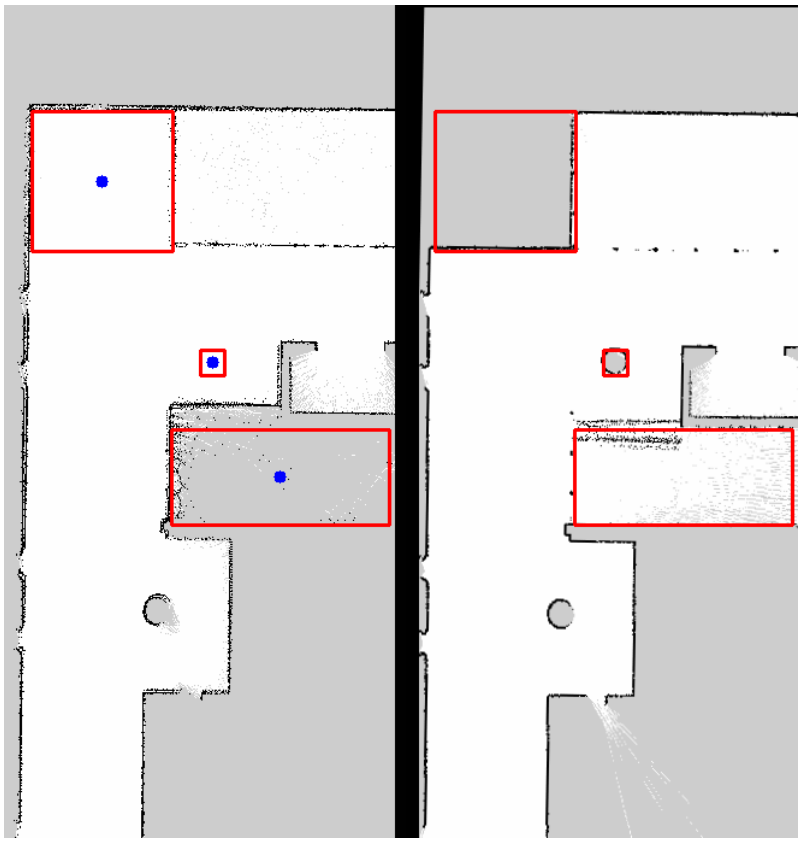


Figure 5.7: Reduced ROI excluding unexplored regions.

5.2.2 Scenario 2: Additional Obstacles

In the second scenario, a different set of obstacles was introduced into the simulated environment, as shown in Figure 5.8.

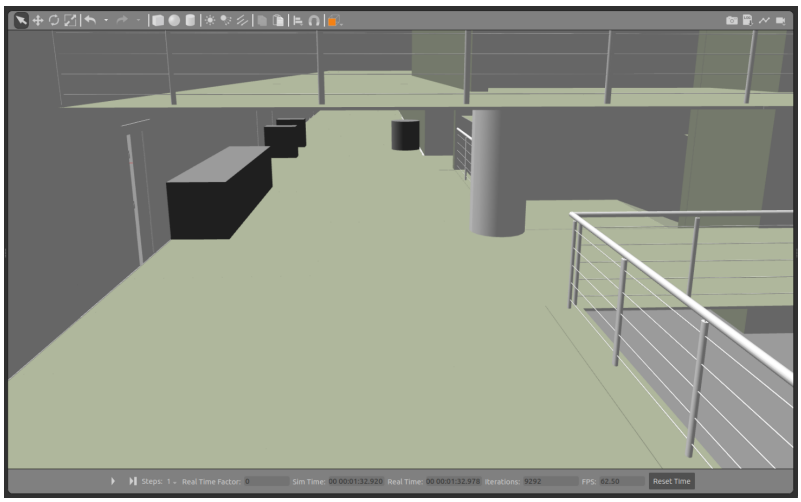


Figure 5.8: Added obstacles (in dark gray).

The corresponding Nav2 and SLAM maps are presented in Figure 5.9.

(a) Nav2 map. (b) SLAM map.

Figure 5.9: Comparison between Nav2 and SLAM maps (case 2). The red dot represents the robot's position.

The clustering and difference detection phases are illustrated in Figures 5.10 and 5.11, while the effect of the ROI reduction is visible in Figure 5.12.

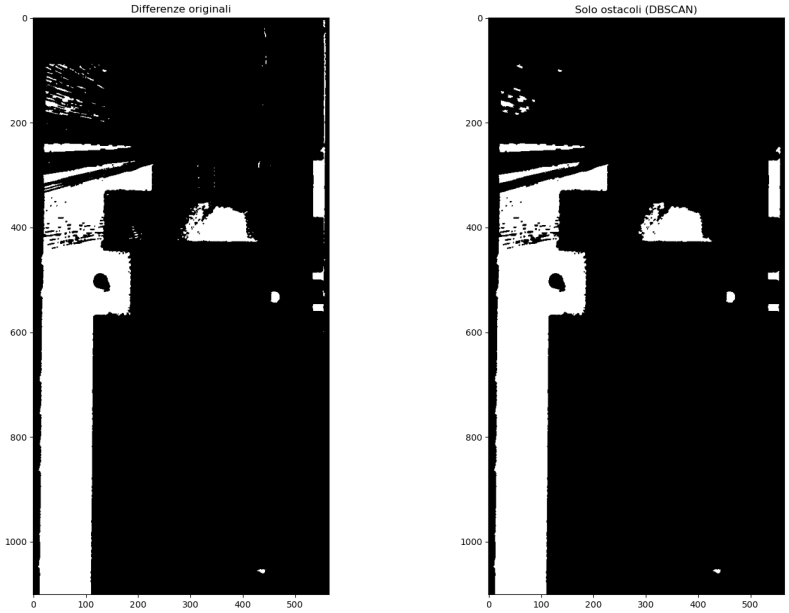


Figure 5.10: Before and after clusterization (case 2).

Figure 5.11: Detected differences and centroids (case 2).

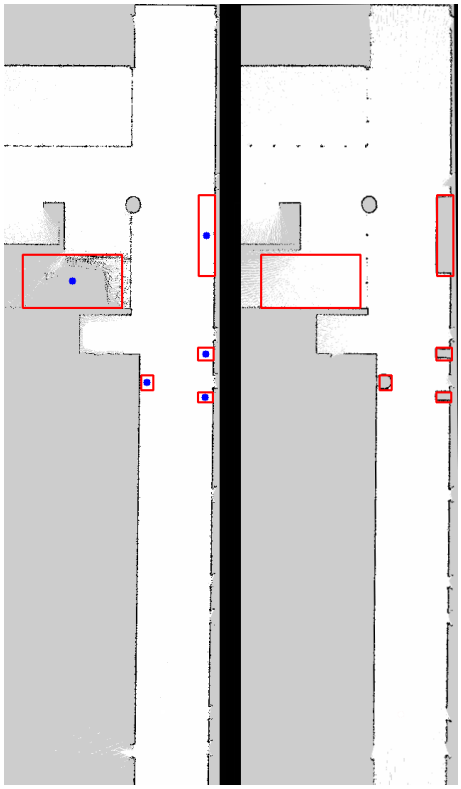


Figure 5.12: Reduced ROI excluding unexplored areas (case 2).

The experiments confirm that the proposed approach effectively detects structural differences in dynamic environments.
Nevertheless, there are still some limitation that need to be taken care of. Future work should focus on integrating semantic filtering for temporary obstacle removal and adaptive SLAM strategies for online map correction, further improving robustness in real-world applications. The next chapter will discuss the system's limitations, along with possible extensions toward a fully autonomous map update framework.

Chapter 6

Discussion and Conclusions

The work presented in this thesis demonstrates the feasibility of an automatic map update pipeline designed for autonomous navigation in GNSS-denied environments. The developed system integrates the ROS 2 ecosystem in order to save, monitor, align and difference detection between maps, allowing an automated map management.

The obtained results are promising: the proposed architecture is able to detect structural variations with high precision, establishing a foundation for deployment in real-world scenarios. A completely image-based approach could represent a practical and computationally effective alternative to the more complex probabilistic and semantic models presented in the literature. Even though it does not consider time persistence nor the dynamic behavior of the objects, the method results easier to implement and to integrate in the ROS 2 navigation stack and it is potentially scalable to a multi-robot scenario.

6.1 Limitations and Future work

Despite promising results, during the experimentation phase some limitations arose:

- ROI selection is critical: naive choices can significantly bias detection results.
- The system is sensitive to yaw rate variations, which may cause misalignment during map comparison.
- Image registration becomes less effective in short mapping sessions, where limited features hinder reliable alignment.
- The current implementation does not explicitly discriminate between temporary and permanent changes.
- Experiments were conducted in relatively small indoor environments (e.g., I3P corridors). While representative of typical structured spaces such as hospitals or airports, larger areas, such as museums or airport halls, should be tested to

evaluate scalability. Nevertheless, this limitation can be mitigated by adopting intelligent ROI selection strategies.

Numerous other directions of research and development exist which are worth to be explored to make the solution even more robust, scalable and applicable to real scenarios. Firstly, the presented approach focuses on a bi-dimensional representation of the environment. A natural evolution would be extending the method to 3D map management, obtained with LiDAR or depth cameras.
Three-dimensional maps would allow to capture vertical and complex details, such as shelving, ramps, and suspended objects, sensibly improving the autonomous navigation capability in multi-level environments, like parking lots, warehouses or big airport hubs. Such improvement carries new challenges, such as the computational cost and the necessity of more sophisticated registration algorithms, such as Iterative Closest Point (ICP) or Normal Distributions Transform (NDT).

Secondly, at the time of writing, the system detects only geometrical differences. However, distinguishing the nature of the differences represents a crucial aspect: for instance, a permanent new wall deserves different attention compared to a plasterboard wall for site operations.
A possible improvement is thus to introduce a semantic level, via computer vision or machine learning, able to classify relevant changes and decide whether to include them or not in the map.
In this context, the integration of neural networks for object detection or the use of semantic segmentation algorithms would enable the distinction between static and dynamic obstacles, significantly reducing the risk of non-meaningful updates.

It is also worth noting that validating the system in simulation made it possible to evaluate the proposed architecture under controlled conditions. Even if Gazebo and RViz allowed to reproduce realistic instances, the proposed framework's robustness needs to be tested in real scenarios, facing further issues such real sensor noise, light conditions, unpredictable presence of persons and moving obstacles.
This phase is essential to verify the system's performance in real operating environments like airports, hospitals, and museums.

Another improving aspect concerns the optimization of computational cost: the real time elaboration of big maps requires strategies to manage memory and pruning of outdated or irrelevant information.

Finally, an interesting extension could be the development of predictive models able to predict the probability of modification in particular areas, exploiting previous maps history. This would enable the system to evolve from a reactive paradigm to a proactive one.

6.2 Final Considerations

This thesis proposes a concrete proof-of-concept for an automatic map update system compatible with ROS 2.

The approach demonstrated that even relatively simple image processing techniques, if properly integrated in a robotics architecture, can contribute to life-long mapping in dynamic GNSS-denied environments.

The proposed work paves the way for further implementation, contributing to the evolution of autonomous robots able to constantly adapt the navigated environment.

Appendix

Least Squares

The *Least Squares* method is a fundamental mathematical technique used to estimate a model's parameters minimizing the error between the observation and the prevision. It is particularly useful for an overdetermined system, in which the number of equations exceeds the number of unknowns.

Given a linear system expressed as

$$A\mathbf{x} \approx \mathbf{b},$$

where $A \in \mathbb{R}^{m \times n}$ is the observation matrix, $\mathbf{x} \in \mathbb{R}^n$ is the vector of unknown parameters, $\mathbf{b} \in \mathbb{R}^m$ represents the vector of the measurements, the LS solution is obtained by minimizing the following cost function:

$$J(\mathbf{x}) = \|A\mathbf{x} - \mathbf{b}\|^2.$$

The minimization yields to the so-called *normal equations*:

$$A^T A\mathbf{x} = A^T \mathbf{b},$$

whose solution provides the parameters' estimate:

$$\hat{\mathbf{x}} = (A^T A)^{-1} A^T \mathbf{b},$$

assuming $A^T A$ invertible.

ORB

The algorithm Oriented FAST and Rotated BRIEF (ORB) is an effective and robust technique to detect and describe key points in digital images. It has been introduced by Rublee et al. in 2011 as open-source and computationally lighter alternative to more complex methods such as SIFT and SURF, maintaining good properties in terms of rotation and scale invariance.

ORB is based in two fundamental components:

- **FAST** (Features from Accelerated Segment Test) for keypoints detection. It is extremely quick, yet non rotation invariant.

- **BRIEF** (Binary Robust Independent Elementary Features) for keypoints description. it is a binary descriptor that compares the intensity of couples of pixels within a patch.

To obtain rotation invariance, ORB introduces a keypoints orientation mechanism. The orientation θ of a keypoint is computed as:

$$\theta = \arctan\left(\frac{m_{01}}{m_{10}}\right)$$

where m_{pq} are the central moments of the patch intensity:

$$m_{pq} = \sum_{x,y} x^p y^q I(x, y)$$

Once the orientation is computed, the BRIEF descriptor is also rotated, obtaining the so-called *steered BRIEF*, which is rotation invariant.

Furthermore, ORB selects also the binary tests of BRIEF in an optimized manner, selecting the ones with more variance by selecting those with higher variance and lower mutual correlation, thereby improving the discriminability of the descriptor.

For what concerns image registration, ORB is adopted for:

1. detecting the keypoints in two or more images;
2. computing the binary descriptors associated with each keypoint;
3. perform *matching* between descriptors using the Hamming distance;
4. estimating the geometric transformation (homography) through RANSAC.
5. applying the transformation to align the images.

ORB is particularly suitable for real-time scenarios thanks to its linear computational complexity with respect to the number of keypoints. Moreover, the binary nature of the descriptors allows an extremely fast matching.

ORB offers several advantages: it is open-source, it is faster than SIFT and SURF, and it is robust to rotations. However, it presents also some limitations in terms of robustness for heavy scale variations and perspective deformations.

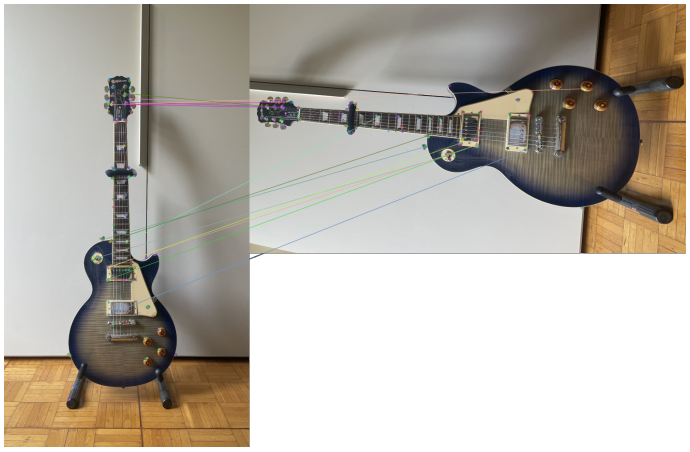


Figure 6.1: Example of feature extraction and matching via ORB.

DBSCAN

The real strength of *density-based* algorithms lie on their ability to identify complex and non-convex shaped clusters, hardly detectable by methods based on the hypothesis of regular geometrical shape (e.g., k-means). In this approach the clusters are interpreted as *high-density regions*, whereas isolated points are considered *outliers*.

The *Density-Based Spatial Clustering of Applications with Noise* (DBSCAN) algorithm introduces three fundamental concepts:

- **Core points**: points that have a minimum number of neighbors within a specified radius;
- **Border points**: points that do not satisfy the density criteria, but are reachable by a *core point*;
- **Noise points**: points that do not belong to a cluster since are not reachable by any *core point*.

To operate, DBSCAN requires two input parameters:

- $minPts$: minimum number of neighbors necessary for a point to be considered a *core point*;
- ε: search radius that defines the neighborhood between two points.

Namely, given a point p, its *neighborhood* ε defined as:

$$N_\varepsilon(p) = \{q \in D \mid dist(p, q) \leq \varepsilon\}$$

where D is the dataset and $dist(\cdot, \cdot)$ is an arbitrary distance function (the Euclidean distance, for instance).
A point p is a *core point* if:

$$|N_\varepsilon(p)| \geq minPts$$

The algorithm proceeds to iteratively expanding the clusters starting from the *core points*, including the points reachable by density (*density-reachability*) and connecting the points that belong to the same dense region (*density-connectedness*). The points that do not satisfy these conditions are labeled as *noise*.

With this density-based definition, DBSCAN is able to:

- detect clusters of any shape;
- manage the presence of noise in the data;
- it does not require the number of clusters a priori.

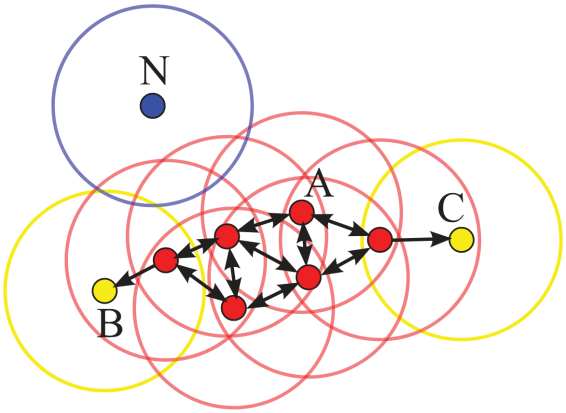


Figure 6.2: Standing people not to be added to the map

Bibliography

[1] Abdulrahman Saleh Al-Batati, Anis Koubaa, Mohamed Abdelkader, *ROS 2 in a Nutshell: A Survey*, Preprints.org, 2024.

[2] Chown, Eric, and Byron Boots. "Learning cognitive maps: Finding useful structure in an uncertain world." Robotics and Cognitive Approaches to Spatial Mapping. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. 215-236.

[3] Schubert, Erich, et al. "DBSCAN revisited, revisited: why and how you should (still) use DBSCAN." ACM Transactions on Database Systems (TODS) 42.3 (2017): 1-21.

[4] Deng, Dingsheng. "DBSCAN clustering algorithm based on density." 2020 7th international forum on electrical engineering and automation (IFEEA). IEEE, 2020.

[5] Macenski, Steve, and Ivona Jambrecic. "SLAM Toolbox: SLAM for the dynamic world." Journal of Open Source Software 6.61 (2021): 2783.

[6] Open Robotics. (2022). ROS 2 Humble Hawksbill Documentation. Retrieved August 21, 2025, from https://docs.ros.org/en/humble/

[7] Stachniss, Cyrill, John J. Leonard, and Sebastian Thrun. "Simultaneous localization and mapping." Springer handbook of robotics. Cham: Springer International Publishing, 2016. 1153-1176.

[8] Macenski, Steve, et al. "The marathon 2: A navigation system." 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2020.

[9] Sebastian Thrun, Dieter Fox, Wolfram Burgard, *Probabilistic Robotics*, The MIT Press, 2005

[10] Federica Schena, *Development of an automated benchmark for the analysis of Nav2 controllers*, Politecnico di Torino, 2024.

[11] Ali, Waqas, et al. "A life-long SLAM approach using adaptable local maps based on rasterized LIDAR images." IEEE Sensors Journal 21.19 (2021): 21740-21749.

[12] Pomerleau, François, et al. "Long-term 3D map maintenance in dynamic environments." 2014 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2014.

[13] Wei, Hairuo, et al. "Large-Scale Multi-Session Point-Cloud Map Merging." IEEE Robotics and Automation Letters (2024).

[14] Adkins, Amanda, Taijing Chen, and Joydeep Biswas. "ObVi-SLAM: long-term object-visual SLAM." IEEE Robotics and automation letters 9.3 (2024): 2909-2916.

[15] Maruyama, Yuya, Shinpei Kato, and Takuya Azumi. "Exploring the performance of ROS2." Proceedings of the 13th international conference on embedded software. 2016.

[16] Macenski, Steve, et al. "From the desks of ROS maintainers: A survey of modern & capable mobile robotics algorithms in the robot operating system 2." Robotics and Autonomous Systems 168 (2023): 104493.

[17] Dissanayake, MWM Gamini, et al. "A solution to the simultaneous localization and map building (SLAM) problem." IEEE Transactions on robotics and automation 17.3 (2001): 229-241.

List of Figures

List of Tables

Acronyms

AMCL Adaptive Monte Carlo Localization

BoW Bag of Words

CLI Command Line Interface

DBSCAN Density-Based Spatial Clustering of Applications with Noise

DDS Data Distribution Service

DCPS Data-Centric Publish-Subscribe

EKF Extended Kalman Filter

EOL End Of Life

GNSS Global Navigation Satellite System

IMU Inertial Measuring Unit

LiDAR Light Detection and Ranging

OMG Object Management Group

ORB Orientated FAST and Robust BRIEF

PGM Portable Gray Map

QoS Quality of Service

RANSAC Random Sample Consensus

ROI Region Of Interest

ROS Robot Operating System

SoOp Signals of Opportunity

SEDIA SEat Designed for Intelligent Autonomy

SIFT Scale-Invariant Feature Transform

SLAM Simultaneous Localization and Mapping

SME Small and Medium-sized Enterprise

STVL Spatio-Temporal Voxel Layer

SURF Speeded-Up Robust Feature

TEB Timed Elastic Band

TCPROS Transmission Control Protocol over ROS

UDPROS User Datagram Protocol over ROS

YAML Yet Another Markup Language