POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Implementation of ROS-based Multi-Agent SLAM Centralized and Decentralized Approaches

Supervisors

Prof. Marina INDRI

Advisors

Ph.D Pangcheng David CEN CHENG

Candidate

Paolo VANELLA

December 2023

Abstract

With the advent of mobile robotics, new research fields were born. In particular, one of the most interesting challenges regards the so-called Simultaneous Localization and Mapping, also known as SLAM: it concerns the task that a mobile robot accomplishes to accurately create a map of an unknown environment and localize itself in it. Since mapping large environments may take too much time if only a single agent is employed, SLAM frameworks that rely on a swarm of mobile robots begin to be developed in order to not only shorten times for mapping but also achieve higher accuracy. Thus, such methods aim at solving a multi-agent SLAM problem. There are two main ways to map the environment and track the entire squad of agents: either having a central server that does the whole computation or implementing a distributed approach that requires exchange of data between mobile robots. This work presents two different approaches that are inspired by the previous two alternatives and are developed with the help of ROS2, i.e., an open-source set of software tools for robotic applications. Moreover, the mapping is limited to be two-dimensional, hence these methods are specially designed to work with 2D LiDAR sensors. Both frameworks are tested in a simulated or real-time scenario with two copies of TurtleBot3 Burger.

The first approach is a centralized setting where each agent independently creates its own local map without exchanging additional data with others. Simply, a single-agent SLAM algorithm runs locally on each agent and only processes the laser scans of that agent. A central server merges the local maps through feature matching, thus building a global map of the explored environment. Although this naive approach still produces a good global map, the server needs some assumptions on the initial poses of the mobile robots, otherwise the relative pose between the local maps cannot be computed correctly and the feature matching may fail.

The second approach is a low-drift, fully-distributed method that incorporates two modules. The single-robot front-end module supports any LiDAR-based SLAM with inertial odometry. The collaborative aspect of the algorithm lies in the distributed back-end module: during a meeting between two mobile robots, they exchange scan data through LiDAR-Iris, i.e., a robust LiDAR descriptor that lightens the load on the communication link and formats the point cloud into a quickly processable piece of data, on which the distributed loop closure recognition operates to find possible candidates for inter-robot loop closures. Even if the front-end module does not support intra-robot loop closure, the system still provides such a functionality. Furthermore, the distributed back-end module performs outlier rejection and a graph optimization step in order to refine the global map.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As the usage of mobile robots started to spread during the last decades, one of the most difficult tasks to accomplish for such devices concerns the mapping of an unknown environment where the robot is able to move and orientate. Already existing solutions such as GPS are not very reliable and accurate to determine the exact position of a mobile robot, and building an accurate map would still be problematic for high-precision tasks, particularly for indoor environments. Thus, Simultaneous Localization and Mapping (in short, SLAM) began to grow as a brand new topic of research that is still challenging nowadays. Shortly, SLAM incorporates two tasks: localization and mapping. Localization aims at tracking the robot's trajectory as a sequence of poses, while mapping means building a map of the environment where the robot is located.

Nowadays, SLAM is applied to a plethora of different real-time scenarios in which mobile robots are involved. Industrial logistics applications rely on SLAM algorithms in order to enhance automation for inventory management, thus materials can be transported efficiently inside the factory. Some mobile robots that are designed for daily use are also equipped with a SLAM algorithm, for example, modern vacuum cleaners can map their surroundings in order to efficiently clean the entire floor. Furthermore, SLAM is essential for ground exploration missions because mobile robots can build a real-time map while wandering in unknown environments. Recently, SLAM has evolved, thus making it possible to extend exploration to underwater scenarios and for air drones. Last but not least, autonomous vehicles such as self-driving cars would not exist if localization and mapping could not be solved: even though many services already provide built maps that represent the road and other features of interest, the vehicle must find its own position and orientation within the map, because localization provides useful support for decision making and motion planning.

Another topic SLAM is strictly related to is navigation. While the map is being built and updated, a navigation system intervenes and supports the mobile robot for path planning and obstacle detection. Of course, the quality of navigation strictly depends on the quality of both localization and mapping: this is why SLAM should produce accurate results. This reason also explains why autonomous vehicles do not rely only on GPS: eventual inaccuracies in localization and/or mapping can translate into a bad decision making or misleading information for navigation.

An even more complex task concerns a swarm of mobile robots that aim at building a global map of the same environment and tracking themselves within it. In this case, it is more appropriate to speak about multi-agent SLAM or collaborative SLAM: not only it extends the properties of single-agent SLAM but additionally requires collaboration between mobile robots, hence agents must be able to communicate with each other by exchanging their measurements. In general, multi-agent SLAM algorithms follow two different approaches: they either deploy low-cost agents with low computational power and demand the whole computation to a central server, or instead exploit peer-to-peer communication between mobile robots in order to exchange relative information about local maps. Multi-agent SLAM algorithms that are based on the first approach are known as centralized, while those that follow the second approach are called decentralized.

Such extension to SLAM improves exploration and logistics, because it enhances collaboration between mobile robots, thus improving efficiency and taking less time to accomplish a specific task. Moreover, having more than a single agent improves accuracy and resilience. For example, it is faster to build a map if more mobile robots are available, and if some of them have a failure or suddenly become unavailable the global task can still go on.

This thesis work is structured as follows.

Chapter 2 describes the general properties of single-agent SLAM and includes a digression of the state-of-the-art of single-agent SLAM.

Chapter 3 illustrates the extension of SLAM to collaborative scenarios, as well as a review of the state-of-the-art of multi-agent SLAM.

Chapter 4 introduces the ROS framework and its main concepts and describes the ROS-based hardware that will be used for testing the implemented multi-agent SLAM algorithms.

Chapters 5 and 6 describe the idea behind the centralized multi-agent SLAM approach and the distributed multi-agent SLAM algorithm, respectively. Details about the implementation are also provided, as well as some results in a simulated scenario. A demonstration on the centralized setting applied on a real-world scenario is also available.

Chapter 2

Simultaneous Localization and Mapping

The SLAM problem can be defined as the following: a mobile robot has to build a map of an unknown environment and at the same time it has to keep track of its position while it moves. During the years, researchers studied the SLAM problem from a mathematical point of view and different formulations of the same problem were made. In turn, these formulations gave birth to different algorithms that process raw data and return the robot's pose and the map of the environment. As the years went by, research made progress in terms of efficiency and accuracy for both localization and mapping, and in more recent times new needs emerged; for instance, unusual domains such as underwater or underground environments and faster solutions with more than one agent become object of interest for more complex SLAM approaches. In particular, the extension of SLAM to a swarm of mobile robots is a topic that has become very popular recently.

Regardless the context the SLAM is applied to, it is accomplished thanks to both odometry and external information. Sensors are crucial because the mobile robot can capture data from its surroundings. Visual information is acquired through cameras or laser scanners such as LiDARs (Light Detection And Ranging). Another type of sensor that is usually used for SLAM is the IMU (Inertial Measurement Unit): it is able to measure a bunch of useful inertial measurements, such as angular velocity, acceleration, Earth's magnetic field and air pressure. Nowadays, IMUs are essential for mobile robots, because they acquire lots of intrinsic and external data despite being quite cheap devices, and most important IMU sensors work everywhere while other sensors may have trouble under some specific conditions (for example, cameras need light and GPS needs a connection).

For now, it is assumed that only one mobile robot is involved into SLAM, hence the task to accomplish is called single-agent SLAM (or equivalently, single-robot SLAM). Following in this chapter, the first mathematical formulation for SLAM algorithms and the main properties of SLAM are discussed. A digression about the single-agent SLAM state-of-the-art follows, giving particular focus to how the problem evolved over the years and introducing an exhaustive review of the most known algorithms that were developed. In general, more emphasis is given to laser-based approaches, since the real-world experiments are carried out by mobile robots that are equipped with LiDAR sensors.

2.1 Formulation and General Properties

In order to build a map, a SLAM algorithm takes into account all the commands $\boldsymbol{u}_1, ..., \boldsymbol{u}_t$ received and the previous observations and measurements $o_1, ..., o_{t-1}$ from the sensors and the odometry. Given the initial pose $\boldsymbol{x}_0$ of the mobile robot, the algorithm returns its current pose x_t and the map m: the output variables are usually probabilistic values that are statistically independent in general.

$$P\left[m, \boldsymbol{x}_t | \boldsymbol{x}_0, \boldsymbol{u}_{1:t}, o_{1:t-1}\right] \tag{2.1}$$

This is not a rigid definition for all the algorithms, but it gives the idea of what is required to achieve SLAM. Equation (2.1) clarifies the two tasks of SLAM: a SLAM algorithm should be able to both recover the robot's trajectory as a sequence of poses and build a map of the environment.

Under some assumptions, it is sometimes possible to separate the two tasks in such a way that localization and mapping become independent of each other. As it will be clarified later in Section 2.2, modern approaches forgo this definition in favour of more complex formulations that are usually based on the optimization theory.

The most important properties that researchers had to comprehend in detail before writing down practical solutions are convergence and sparsity.

Given the probabilistic nature of the problem, it is interesting to study under what conditions the algorithm progressively corrects its estimates until the trajectory and the map converge to reliable values. Similarly, optimization is strictly related to convergence because this property ensures that a solution can be found and the optimization theory guarantees that such solution is optimal.

To understand why sparsity is involved, it is important to anticipate that methods which rely on a formulation similar to (2.1) acquire probabilistic information that are characterized by covariance matrices. As the objective of SLAM is to estimate the robot's trajectory and the map, if these matrices are sparse the computation can be simplified [1]. Actually, some algorithms elaborate the observations in order to extract the trajectory and/or the map estimate as a sparse state. Indeed, the state may grow in dimension as the mobile robot wanders around and it is sometimes not necessary to keep the whole trajectory since the beginning of the algorithm. Sparsity is also involved in optimization problems where the optimal state to be estimated is sparse.

An important property that is strictly related to convergence concerns the suppression of the error accumulation over time. This phenomenon is known as drift and occurs when the cumulative errors in the pose and map estimates tend to grow as time progresses while the algorithm is running. Drift is usually caused by intrinsic inaccuracies of sensors or uncertainties in mathematical models.

SLAM frameworks that can solve the drift problem are described as low-drift. This means that low-drift algorithms are designed to minimize the cumulative errors, thus producing more accurate trajectories and maps. Low-drift is essential for autonomous navigation and long-running tasks. To achieve low-drift, most algorithms implement loop closure detection. IMU sensors are able to further reduce the drift, especially on the odometry.

2.2 State-of-the-Art for Single-agent SLAM

The concept of SLAM was born during the late 80's [2]. The first solutions to the SLAM problem were based on probabilistic filters: the uncertainty that affects the map and the pose estimation is modeled as a probability distribution [3]. These methods were computationally heavy, thus researchers tried to find some optimization for SLAM algorithms by studying some intrinsic properties of SLAM such as convergence and sparsity [4]. This way, during the last decade optimized algorithms that need less computational effort began to take hold. In recent years, researchers aimed at implementing new algorithms that can deal with scalability and work under unfavorable conditions concerning hardware limitations and/or difficulties related to a specific environment.

As a consequence, SLAM algorithms can be classified into filter-based and optimization-based algorithms. Filter-based algorithms rely on the theory of probabilistic filters (also known as Bayes filters) and the SLAM problem is often defined as a state estimation problem. On the other hand, optimization-based algorithms tend to directly solve a constrained state estimation problem that is usually modeled as a least-squares problem or a similar one, hence a sparse solver is often involved. Optimization-based algorithms differ from each other according to the particular form of optimization that is adopted in order to find the trajectory and build the map. This classification can be summarized in Fig. 2.1.

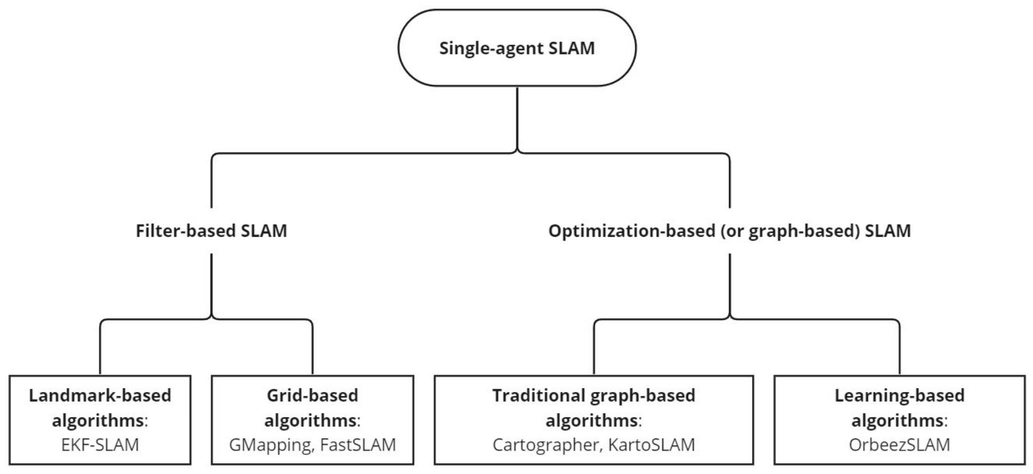

Single-agent SLAM
Filter-based SLAM
Optimization-based (or graph-based) SLAM
Landmark-based algorithms: EKF-SLAM
Grid-based algorithms: GMapping, FastSLAM
Traditional graph-based algorithms: Cartographer, KartoSLAM
Learning-based algorithms: OrbeezSLAM

Figure 2.1: A rough classification of SLAM algorithms that are designed for a single mobile agent.

In addition to this, some algorithms are specific for a category of sensors. RGB-D cameras, stereoscopic vision and LiDAR sensors are the most used technology in mobile robots to capture frames of the environment. Thus, SLAM algorithms can be further divided into visual SLAM (or V-SLAM in short) [5] and LiDAR SLAM [6].

Cameras are cheap and light devices with a long detection range, but they are highly dependent on lighting. In particular, RGB-D cameras tend to capture blurry frames if the mobile robot moves at high speed. On the other hand, laser sensors are more accurate but less resilient, more expensive, heavier and suffer from refraction phenomena. As time passed, LiDAR SLAM has taken over visual SLAM, especially for indoor applications, and nowadays LiDAR sensors are more implemented than RGB-D cameras as sensors on mobile robots. However, due to the cheap nature of cameras and incoming advanced computer vision techniques, visual SLAM is nowadays growing as a promising research field.

A worthy mention goes to deep learning: as it is a branch of research that is constantly growing in interest, some brand new approaches try to apply deep learning techniques on V-SLAM to improve its performance, since RGB-D cameras are cheap devices and frames are trivial to be processed through deep learning techniques. Neural networks represent a powerful instrument for image processing: in particular, the tasks of feature extraction, estimation (here, odometry and map building are included) and loop closure can be replaced by approaches that are respectively based on you-only-look-once algorithms, convolutional neural networks and autoencoder-like networks [7]. Other modern approaches apply deep learning to 3D point cloud data coming from LiDAR sensors: due to the sparse nature of point clouds, some approaches prefer to concatenate LiDAR data and RGB frames in order to ease the processing and achieve high accuracy [8].

The main advantages of deep learning include the capability of generalizing for different scenarios, relying on past experiences for enhanced predictions and exploiting the current observation for self-learning. Moreover, neural networks can extract some "hidden" features and correlations that the mathematical theory cannot retrieve.

Regardless of the category a SLAM algorithm belongs to, a characteristic that brings together the majority of them is the general workflow. A scheme of the workflow is reported in Fig. 2.2. Sensors capture raw data from the environment and the algorithm processes it to extract salient details of the environment: this operation is called feature extraction and allows to recognize corners, edges, colors, shadows and depths. Feature matching takes more data inputs from sensors and finds common features to better estimate the pose of the camera. After the pose is estimated correctly, a loop closure detection system intervenes and allows the robot to recognize previously visited features of the environment and enhances optimization. Loop closure is a crucial concept of SLAM, because a mobile robot is able to recognize a previously visited part of the environment to correct the estimated map.

2.2.1 A Review of Filter-based SLAM Algorithms

The first implemented SLAM algorithms relied on probabilistic filters. Two categories can be distinguished, depending on how the map is built: landmark-based approaches and grid-based approaches.

Landmark-based algorithms focus on estimating the poses of some elements of interest that are located in the environment. These approaches aim at finding a good estimate of the state of the mobile robot, which includes the trajectory and the poses of all the landmarks. Hence, SLAM becomes a state estimation problem. The idea behind the solution for landmark-based SLAM exploits the state estimation theory: as SLAM is usually set as a nonlinear problem, extended Kalman filters (EKF) can be applied to build an observation model [3].

The system that represents SLAM can be resumed by (2.2): the state $\boldsymbol{x}_k$ at time

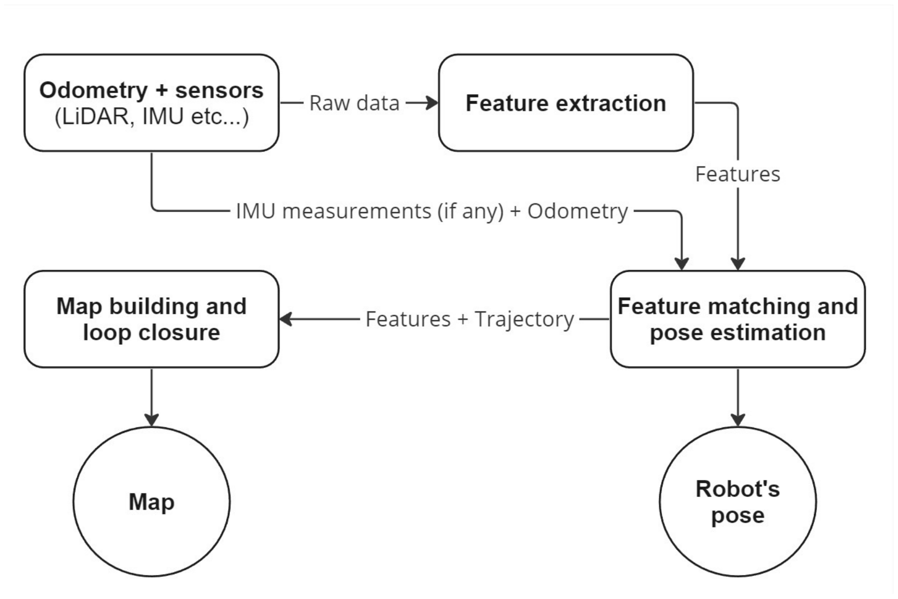

Odometry + sensors (LiDAR, IMU etc...)
Raw data
Feature extraction
Features
IMU measurements (if any) + Odometry
Map building and loop closure
Features + Trajectory
Feature matching and pose estimation
Map
Robot's pose

Figure 2.2: Workflow of SLAM algorithms.

k includes the robot's motion (typically its coordinates and yaw angle) and the position of all the landmarks. The state evolves according to the input commands $\boldsymbol{u}_k$ and the measurements o_k from sensors during each time step. The estimated map m_k is considered as the system's output and generally depends on the state and the measurements. It is also supposed that Gaussian noises $\boldsymbol{\xi}_k$, $\boldsymbol{\eta}_k$ affect the system. If the mobile robot is provided with an IMU, the motion model can be enhanced thanks to the measurements taken by the IMU sensor.

$$\begin{cases} \boldsymbol{x}_{k+1} = f(\boldsymbol{x}_k, \boldsymbol{u}_k, o_k) + \boldsymbol{\xi}_k \\ m_k = g(\boldsymbol{x}_k, o_k) + \boldsymbol{\eta}_k \end{cases} \tag{2.2}$$

EKF-SLAM [9] is one of the very first SLAM algorithms ever implemented. Considering that the vehicle motion evolves during time but the landmarks are stationary, the state could also be augmented whenever a new landmark has to be added to the model. Algorithm 1 shows the general workflow of EKF-SLAM: the prediction phase provides an initial estimate based on the odometry, then an update step exploits the information from the sensors to further refine this estimate [10]. The prediction step can be further improved if IMU sensors are available.

The main advantage of EKF-SLAM is its efficiency in terms of memory. However, it can be proven that the complexity of EKF-SLAM is $O(n^2)$ for a single step of the algorithm and $O(n^3)$ in total, where n is the number of landmarks, hence EKF-SLAM becomes less sustainable as the size of the map increases [10]. Moreover, as can be seen from the computation of the Jacobian matrices, the EKF applies linearization around the state estimate instead of the true state. This leads to two inconsistency problems: at each update step the uncertainty grows according to the variance of the heading angle, and for large uncertainties the estimated trajectory may present discontinuities and jumps [9].

Algorithm 1 EKF-SLAM algorithm.

```
1: procedure EKF-SLAM
2:     ▷ Functions $f$, $g$ refer to the system in (2.2).
3:     ▷ $\hat{\boldsymbol{x}}_{a|b}$ is the state estimate at time $a$ based on observations until time $b$.
4:     ▷ It holds that $\hat{\boldsymbol{x}}_{a|b} = \hat{\boldsymbol{x}}_a$.
5:     ▷ Similarly, $\boldsymbol{P}_{k|k-1}$ represents the covariance matrix of the state.
6:     ▷ $\boldsymbol{F}_k = \frac{\partial f}{\partial x}\Big|_{\hat{\boldsymbol{x}}_{k-1|k-1}, u_k}$, $\boldsymbol{G}_k = \frac{\partial g}{\partial x}\Big|_{\hat{\boldsymbol{x}}_{k|k-1}}$ are the Jacobian matrices.
7:     ▷ $\boldsymbol{z}_k$ represents the measurements at time $k$.
8:     ▷ $\boldsymbol{Q}_k$, $\boldsymbol{R}_k$ are the covariances of the noises $\xi_k$, $\eta_k$.
9:     ▷ Initialization
10:    $\boldsymbol{z}_0$ initialized with the first taken measurements.
11:    A first estimate $\hat{\boldsymbol{x}}_0 = \hat{\boldsymbol{x}}_{0|0}$ is initialized from $\boldsymbol{z}_0$.
12:    ▷ Execution
13:    for each step $k$ do
14:        $\hat{\boldsymbol{x}}_{k|k-1}$ is obtained from odometry, as well as $\boldsymbol{F}_k$, $\boldsymbol{G}_k$.
15:        $\boldsymbol{P}_{k|k-1} = \boldsymbol{F}_k \boldsymbol{P}_k \boldsymbol{F}_k^T + \boldsymbol{G}_k \boldsymbol{Q}_k \boldsymbol{G}_k^T$
16:        New measurements $\boldsymbol{z}_k$ are taken and eventually associated to landmarks.
17:        A new Jacobian $\boldsymbol{H}_k$ is computed from data association.
18:        $\boldsymbol{S}_k = \boldsymbol{H}_k \boldsymbol{P}_{k|k-1} \boldsymbol{H}_k^T + \boldsymbol{R}_k$                ▷ Predicted covariance
19:        $\boldsymbol{K}_k = \boldsymbol{P}_{k|k-1} \boldsymbol{H}_k^T / \boldsymbol{S}_k$                ▷ Kalman gain
20:        $\boldsymbol{\nu}_k = \boldsymbol{z}_k - g(\hat{\boldsymbol{x}}_{k|k-1})$                ▷ Measurement residual
21:        $\hat{\boldsymbol{x}}_k = \hat{\boldsymbol{x}}_{k|k-1} + \boldsymbol{K}_k \boldsymbol{\nu}_k$                ▷ Updated state estimate
22:        $\boldsymbol{P}_k = (\boldsymbol{I} - \boldsymbol{K}_k \boldsymbol{H}_k) \boldsymbol{P}_{k|k-1}$
23:        Eventually, a new landmark can be added here.
24:    end for
25:    return $(\hat{\boldsymbol{x}}_k, \boldsymbol{P}_k)$                ▷ Now, the map can be computed!
26: end procedure
```

Grid-based algorithms represent an alternative to landmark-based approaches. These methods map the unknown environment into a grid, which is initially empty and can eventually be expanded. Each cell of the grid is associated with a probability that determines whether that cell is occupied or not. A grid representation of the map is more intuitive and also contains useful information for navigation, at the cost of being more expensive in terms of memory.

The algorithms used for this purpose are mainly based on the so-called particle filters, hence these approaches are also defined as particle-based. Here, particle-based and grid-based algorithms are treated as synonyms. Particle filters are probabilistic filters that associate a possible trajectory with a weighted particle by sampling a probability distribution: the weight has the meaning of probability and a probabilistic map is associated to each particle.

Particle-based approaches perform four basic operations for each step of the algorithm: sampling, weighting, resampling and map estimation. First, particles are "sent forward" by integration of the probability distribution, thus computing the possible predicted trajectories of the robot and then the weights are updated according to the information coming from the sensors. Then, a resampling phase reduces the number of particles by deleting those with low probability. Finally, a map estimate is computed for each remaining particle. In the context of SLAM, the particle filter method for localization is also known as Monte-Carlo scan matcher [11].

An efficient way to employ particle filters is implemented in Rao-Blackwellized Particle Filters (RBPF) [12], which represents a SLAM technique based on particle filters. RBPF-SLAM aims at solving one of the biggest problems of such approaches, which is the high number of particle filters that need to be carried on during the time. Given the odometry measurements $u_1, ..., u_{t-1}$ during time, the observations $z_1, ..., z_t$ from sensors and the trajectory $x_1, ..., x_t$ with initial position x_0, RBPF-SLAM factorizes the problem by decoupling the computation for the trajectory and the map as shown in (2.3).

$$P\left[m, x_{1:t} | z_{1:t}, u_{1:t-1}, x_0\right] = P\left[m | x_{1:t}, z_{1:t}\right] \cdot P\left[x_{1:t} | z_{1:t}, u_{1:t-1}, x_0\right] \tag{2.3}$$

It is evident that the map strongly depends on the robot's trajectory which is exactly known, since the observations and the odometry measurements are known. Two different probability distributions can be distinguished, one for the map and one for the trajectory: this is the key aspect of the so-called Rao-Blackwellization [12]. The result is that the trajectory can be computed separately and efficiently by sampling, and the consequent map is analytically trivial to obtain.

An advanced particle-based approach goes under the name of GMapping [13]. It is a laser-based, more efficient version than RBPF-SLAM and introduces adaptive techniques in order to reduce the number of particles. GMapping aims at truncating particles according to some intrinsic parameters that represent a trade-off between performance and computational load. In particular, some parameters are predominant and affect the behavior of GMapping heavily: the maximum distance for laser scanning, the number of particles and the threshold of confidence for particle deletion are the most impactful parameters for the performance of the algorithm. GMapping is suitable for high-accuracy tasks, but is specific for laser-based sensors such as LiDARs.

Actually, a hybrid approach between landmark-based and particle-based algorithms exists. FastSLAM [14] was born as a faster alternative to EKF-SLAM. Given the number of landmarks n, it solves $n+1$ independent estimation problems: one uses m particle filters to estimate the robot's trajectory, while the remaining n computes the position of the landmarks through Kalman filters, one for each landmark. Actually, FastSLAM separates localization and mapping into two independent tasks under the hypothesis that n is exactly known. The values of n and m can be freely chosen: in general, having more landmarks requires less particle filters in order to achieve a good accuracy for maps. As a result, it can be proven that FastSLAM reduces the complexity of a single step of the algorithm to $O(m \cdot n)$, while a step of EKF-SLAM is quadratic with respect to n. This complexity can be further improved up to $O(m \cdot \log n)$ if the landmarks are indexed with a tree structure, as it is shown in [14].

2.2.2 Optimization-based Approaches for SLAM

The first implemented SLAM algorithms were based on probabilistic filters, but such SLAM formulation leads to some intrinsic problems. In fact, the dimension of the state grows as the robot travels and acquires landmarks, hence the state estimation problem may become unfeasible in real-time applications. Moreover, many algorithms such as EKF-SLAM present inconsistency issues due to error propagation during the time. A further problem concerns the drift, indeed such algorithms do not rely on a strong loop closure detection, thus the error tends to accumulate, especially on the odometry and for long-running tasks.

In order to overcome these two problems, researchers had to find a new paradigm that could reward the coherence between the odometry measurements, the data captured by sensors and the estimated map and trajectory, while avoiding adding too much complexity from a computational point of view.

The previous considerations inspired researchers to move towards methods based on optimization problems: in fact, such mathematical formulations are able to penalize inconsistency and a solver can retrieve the robot's trajectory, as well as the map of the unknown environment. Thus, optimization-based algorithms represent an improvement for the SLAM research field. Such approaches also go under the name of graph-based algorithms: in fact, differently from the probabilistic formulation in (2.1), the localization task is represented by a graph whose nodes are the poses and the edges are the constraints given by observations and commands, while the map can be consequently computed as the most consistent spatial configuration with respect to the measurements [15]. Algorithms that belong to this category usually model SLAM as a least-square problem or a regression problem. Different existing algorithms are able to solve constrained optimization problems: an example is the iterative Gauss-Newton method [15]. A worthy mention goes again to IMU sensors: if available, the initial pose can be determined with extreme precision, hence an inconvenient situation where the error on the initial pose accumulates through the whole graph can be easily avoided.

One of the most known SLAM approaches in the literature belongs in this category. Cartographer [16] is the algorithm developed by Google and is properly designed for real-time tasks with high performance. However, Cartographer is intended to be applied on mobile robots that are provided with LiDAR sensors. Cartographer maintains a grid structure for the map and a probability is associated to each cell, thus indicating whether a cell contains part of an obstacle or is free. Technically speaking, cells can be classified in "hits" or "misses": these two sets identify respectively occupied and unoccupied points in the point cloud that is acquired by LiDAR scans.

Since particle filters would accumulate errors over time, Cartographer copes with this issue by running a pose optimization instead. The pose $\boldsymbol{\xi} = (\xi_x, \xi_y, \xi_\theta)$ contains the translation and the motion angle of the robot: the optimization problem aims at finding the optimized pose $\boldsymbol{\xi}^*$ that is coherent with the scan points $H = \{\boldsymbol{h}_k\} \in \Re^2$, $k = 1, ..., K$, taken in the scan frame. Hence, the optimization problem is set as a scan matching problem, and Cartographer uses the so-called Ceres solver in order to find a solution.

The formulation is a nonlinear least-squares problem as it is shown in (2.5). The transformation $T_{\boldsymbol{\xi}}(\boldsymbol{h}_k)$ (2.4) shifts a scan point $\boldsymbol{h}_k$ into the submap frame according to the scan pose $\boldsymbol{\xi} = (\xi_x, \xi_y, \xi_\theta)$. The function $M_{smooth} : \Re^2 \rightarrow \Re$ is a smooth filter that maps a scan into a probability value: in the nonlinear optimization problem, it applies to the transformed scan point.

$$T_{\boldsymbol{\xi}}(\boldsymbol{h}_k) = \begin{pmatrix} \cos\xi_\theta & -\sin\xi_\theta \\ \sin\xi_\theta & \cos\xi_\theta \end{pmatrix} \boldsymbol{h}_k + \begin{pmatrix} \xi_x \\ \xi_y \end{pmatrix} \tag{2.4}$$

$$\boldsymbol{\xi}^* = \arg\min_{\boldsymbol{\xi}} \sum_{k=1}^{K} [1 - M_{smooth}(T_{\boldsymbol{\xi}}(\boldsymbol{h}_k))]^2 \tag{2.5}$$

Actually, the scan matching problem includes the most recent scans only and some small error still accumulates. This is why Cartographer is composed of two processes: one acts locally according to what it is said before, and the second one achieves global loop closure. Another optimization problem is formulated as follows: all the submap poses $\Xi^m = \{\boldsymbol{\xi}_i^m\}$ and all the scan poses $\Xi^s = \left\{\boldsymbol{\xi}_j^s\right\}$ are optimized, given some constraints defined by the covariance matrix $\boldsymbol{\Sigma}_{ij}$ and the matched pose $\boldsymbol{\xi}_{ij}$. This optimization problem is known as sparse pose adjustment (SPA) [17] and is made explicit in (2.6). Here, ρ represents a generic loss function and E is the residual of the constraints. The objective is to find the optimal sets $\Xi^{m,*} = \{\boldsymbol{\xi}_i^{m,*}\}$ and $\Xi^{s,*} = \{\boldsymbol{\xi}_i^{s,*}\}$.

$$(\Xi^{m,*}, \Xi^{s,*}) = \arg\min_{\Xi^m, \Xi^s} \frac{1}{2} \sum_{i,j} \rho(E^2(\boldsymbol{\xi}_i^m, \boldsymbol{\xi}_j^s; \boldsymbol{\Sigma}_{ij}, \boldsymbol{\xi}_{ij})) \tag{2.6}$$

In order to enhance scan matching, a branch-and-bound method is applied to a third optimization problem (2.7) that concerns pixel-accurate match according to a smooth function $M_{nearest}$ that considers the neighborhood of a pose in the feasible set. Branch-and-bound starts from a pose in the feasible set and looks for a better pose in the neighborhood of the current optimal solution. Thus, the branch-and-bound scan matching builds a research tree whose root is the set of all possible solutions and whose leaves are the single feasible solutions. Branch-and-bound is interested in finding the optimal pose: visually, the deepest leaves in the tree lead to high resolutions on the global map.

$$\boldsymbol{\xi}^* = \arg\max_{\boldsymbol{\xi}} \sum_{k=1}^{K} M_{nearest}(T_{\boldsymbol{\xi}}(\boldsymbol{h}_k)) \tag{2.7}$$

Further works on Cartographer aim at improving real-time performance and quality of generated maps. For example, authors of [18] found out that an adaptive multi-distance scheduler that affects the two main processes of Cartographer can be implemented such that it decides which process has to run according to its own internal parameters. Moreover, the local process tends to involve smaller scans that are faster to process, while the global process is invoked when the error accumulates, but it takes more time to complete. Overall, Cartographer represents a very robust approach for single-robot SLAM and is not by chance one of the most commonly implemented algorithms for real-time applications.

Other well-known algorithms that are used today in real-time applications for mobile robotics are optimization-based. KartoSLAM [19] is another LiDAR-specific SLAM algorithm whose basic principle is graph optimization. KartoSLAM retrieves the initial pose from the odometry and then solves scan-to-map matching through Cholesky matrix decomposition, which is a method for factorizing the positive-definite constraint matrix in order to solve the matching. This optimization produces a front-end graph one step at a time, but a back-end loop closure is still needed. Sparse Pose Adjustment (2.6) is involved in back-end optimization and receives the constraints among all the poses in order to update the pose graph that is built as a result of the front-end optimization.

In general, KartoSLAM is quite robust thanks to the back-end loop closure; it can work at low scan frequencies and still achieve very good accuracy. However, the Cholesky decomposition does not scale very well and mapping with KartoSLAM becomes almost unfeasible when dealing with complex and wide environments [13].

Speaking about optimization-based and laser-based methods, an approach that brings together some algorithms is LOAM [20] (abbreviation for LiDAR Odometry And Mapping). It is a low-drift, low-computational algorithm that does not rely on IMU nor on loop closure detection. LOAM is composed of two processes: one performs odometry with high frequency (usually 10 Hz) but low accuracy for LiDAR speed, while the other runs at a lower frequency (usually 1 Hz, hence once every 10 scans) for scan matching and mapping, so as to distinguish between edges and planar points within the point clouds perceived by LiDAR scans. These two processes can run in parallel, hence LOAM is well-suited to be applied in a real-time scenario. Moreover, LOAM allows to build a 3D map because it implements the iterative closest point algorithm (ICP) [21] for point cloud matching: briefly speaking, ICP solves a least-squares problem which minimizes the point-to-point, point-to-plane and point-to-line distances between two point clouds.

Given its structure, LOAM is suitable for playing the role of front-end process for more complex laser-based approaches: in fact, it does not rely on other components than LiDAR sensors but it lacks a loop closure detection system. As a consequence, researchers got inspired by combining LOAM with optimization-based loop closure techniques, thus giving birth to algorithms such as LeGO-LOAM [6]. The main difference resides in the presence of back-end optimization for both the odometry and the global map. Additionally, LeGO-LOAM improves the matching tasks by dividing the point cloud into ground and non-ground points and by detecting and filtering small clusters as outliers that may mislead the matching. This way, LeGO-LOAM represents a complete SLAM framework that can achieve low-drift and high accuracy and at the same time the computation is eased by back-end optimization.

LOAM may seem a solid front-end module for LiDAR odometry, but it still lacks integration with other measurements, for example from IMU or GPS, and also suffers from drift when the scan matcher deals with skewed point clouds. Nowadays, other front-end alternatives aim at improving the efficiency of LOAM. FAST-LIO2 [22] is an alternative odometry framework for LiDARs: it is fast and robust and also computationally efficient. FAST-LIO2 directly registers raw points to the map without extracting features, but only using the subtle features of the environment to increase accuracy and adaptability to different scan scenarios; then, the map is maintained by an incremental tree with dynamic updates and re-balancing. Just like LOAM, FAST-LIO2 does not include a loop closure module.

Besides being a complete SLAM approach, LeGO-LOAM inherits some issues from LOAM. LIO-SAM [23] is a highly accurate, real-time framework for trajectory estimation and map building that does not match scans with a global map (as for LOAM), but instead it performs scan-matching at a local scale with a sliding window approach. LIO-SAM is also especially suitable for multi-sensor fusion and includes a loop closure function based on Euclidean distance.

A back-end alternative that is suitable for LiDAR sensors is bundle adjustment (BA) [24], which is the problem of jointly solving 3D structures and finding the location of feature points and camera poses. Briefly speaking, BA is set as an optimization problem with a least-squares objective function: given an observed feature, the goal is to find its optimal pose $\boldsymbol{T}^*$, its vector $\boldsymbol{n}^*$ (depending on the feature, either the normal vector of the plane or the direction of the edge) and the corresponding feature point $\boldsymbol{q}^*$ on the 3D map, such that the distance between each plane feature point $\boldsymbol{p}_i$ (they directly depend on the pose) and the representative feature point of the map is minimized. These quantities are all optimized at once. An explicit formulation is given in (2.8).

$$(\boldsymbol{T}^*, \boldsymbol{n}^*, \boldsymbol{q}^*) = \arg\min_{\boldsymbol{T},\boldsymbol{n},\boldsymbol{q}} \frac{1}{N} \sum_{i=1}^{N} (\boldsymbol{n}^T(\boldsymbol{p}_i - \boldsymbol{q}))^2 \tag{2.8}$$

Algorithms that are based on BA are referred to as sparse bundle adjustment (SBA). By combining LOAM with BA, the BALM algorithm (abbreviation for Bundle Adjustment LiDAR Mapping) is obtained [24]. BALM runs three parallel processes: odometry and map refinement (as for LOAM), as well as feature extraction. Adding feature extraction helps to correctly recognize edge and plane features, similarly to what LeGO-LOAM does. BA is included in the map refinement process that runs every five scans taken by the LiDAR sensor and the pose optimization is applied with a sliding window approach that operates on the most recent scans.

The principle behind LOAM and its derivatives inspired other researchers to implement brand-new algorithms that maintain the same advantages. MULLS [25] is another low-drift 3D LiDAR SLAM algorithm that exploits efficient optimization techniques to achieve very good real-time performance. MULLS does not rely on IMU and contains two modules, one as front-end and the other as back-end.

MULLS front-end first performs motion compensation in order to efficiently extract the point cloud from the current frame according to the previous one. Then, the obtained point cloud is preprocessed by a dual-thresholding filter to detect ground points. Non-ground points are further classified through principal component analysis (PCA), that is a technique that is used in machine learning and data mining for classification of a dataset with a large number of features: thus, non-ground points can be divided into facade, roof, pillar, beam and vertex points. This partition is convenient for applying ICP, because the least-squares problem becomes well-posed: in fact, point-to-point distances concern vertices only, point-to-plane distances affect ground points and facade or roof points, and point-to-line distances are computed between pillar and beam points. The front-end module also contains a local map management module that maintains the feature points of the last frame.

MULLS back-end involves a processing unit that periodically saves the submaps that are generated at the front-end. Thus, a graph is computed: the robot's poses are its nodes and may be grouped to build a submap. Here, a loop closure system is implemented, aiming at finding a correspondence between different submaps. Once a loop closure is detected, a pose graph optimization (PGO) is performed on the whole graph in order to correct the transformation between the reference frames of the involved submaps and the individual poses within a submap.

A worthy mention goes to SLAM algorithms that are designed for stereoscopic, monocular or RGB-D cameras. From a computational point of view, frames can be easily manipulated through computer vision techniques that are simpler than LiDAR techniques for point cloud processing. Efficient processing, combined with optimization, gives birth to real-time visual SLAM algorithms that can compete with laser-based approaches. To cite one, ORB-SLAM [26] is a well-known algorithm that can accomplish visual SLAM. During that time, improved versions were released such as ORB-SLAM2 [27] and ORB-SLAM3 [28].

In particular, ORB-SLAM2 [27] is composed of three parallel threads: tracking, map management and loop closure. The first thread is dedicated to tracking and camera localization, accomplished through feature matching with the local map and enhanced by motion-only BA (i.e., only the trajectory and the camera poses are optimized). The second concerns local mapping and performs it through triangulation between keypoints, then an optimization step involves local BA applied to a set of visible keypoints. The third thread contains loop closure functionalities: after a loop is detected and validated, the global map is corrected by PGO and then BA is applied to the global map. However, BA can take long, hence a fourth thread starts and is dedicated to accomplish BA: an eventual incoming loop closure would force this thread to interrupt and re-launch.

An advanced visual SLAM algorithm that also relies on deep learning techniques is Orbeez-SLAM [29]. This very recent approach does not need pre-training but can be trained from scratch at the target scene in a short period of time. Orbeez-SLAM not only is able to generate dense maps in real-time and easily adapt to new scenes, but also it can work with simple monocular cameras because the input is just required to be RGB.

The key concept behind Orbeez-SLAM is the Neural Radiance Field (also known as NeRF): it is a neural network that does not require depth supervision during training and can be trained from scratch, and its goal is to reconstruct 3D scenes from 2D images taken from distinct points of view. As the name suggests, Orbeez-SLAM inherits the workflow from ORB-SLAM approaches. The map points and the optimized camera poses that are generated by the three threads are paired with the RGB images and fed to the NeRF for the initial training.

A summary of the reviewed algorithms is reported in Table 2.1. Nowadays, there is always a selection of suitable algorithms that are compatible with a specific availability of sensors and hardware resources. Furthermore, depending on the scenario where the SLAM algorithm is applied, some approaches may be more suitable than others. For instance, simple environments can be mapped with fast approaches such as KartoSLAM and Cartographer, while more sophisticated algorithms such as LeGO-LOAM and similar are more indicated if 3D mapping and/or very high precision are required.

Algorithm	**Category**	**Sensor specificity**
EKF-SLAM [9]	Filter-based (on landmarks)	None
RBPF-SLAM [12]	Filter-based (on particles)	None
GMapping [13]	Filter-based (on particles)	LiDAR only
FastSLAM [14]	Filter-based (on particles)	None
Cartographer [16]	Graph-based	LiDAR only
KartoSLAM [13]	Graph-based	LiDAR only
LeGO-LOAM [6]	Graph-based	LiDAR only
BALM [24]	Graph-based	LiDAR only
MULLS [25]	Graph-based	LiDAR only
ORB-SLAM2 [27]	Graph-based	Visual SLAM
Orbeez-SLAM [29]	Graph-based + Learning	Visual SLAM

Table 2.1: A summary of the discussed single-agent algorithms.

Chapter 3

A More Complex Topic: Multi-Agent SLAM

Single-agent SLAM still leaves some open issues. In fact, if a fault happens on the mobile robot the whole task fails, and wide environments require too much time to be fully explored by only one agent. Moreover, even if a robot can still perform loop closure, it is not strongly reliable because the agent may not visit the same features multiple times in order to improve the map and trajectory estimates. As a consequence, the need of developing collaborative solutions that are able to handle more agents arose in recent times.

SLAM can be extended to a multi-agent scenario: a swarm of mobile robots cooperates such that each robot is able to track its trajectory and collaborate with others to build a global map of the unknown environment. In this context, mobile agents are able to share their own raw and/or processed data with others according to a pre-established network infrastructure or under particular circumstances. Multi-robot SLAM brings some important advantages with respect to single-robot SLAM: it makes the SLAM task faster and resilient against attacks on mobile agents, because the application can continue to run even if a few robots become unavailable for some reason. Moreover, it enhances loop closure because the agents may perceive the same features from different points of view, thus the map estimate can be corrected more effectively.

This chapter extends the previous one by integrating more concepts about cooperative approaches for SLAM. First, the generalities of multi-agent SLAM are explained, as well as the related issues that have to be dealt with. The possible solutions to such problems are discussed in the excursiveness about the state-of-the-art and some of the most efficient multi-agent algorithms are introduced to give a better idea of how such approaches work in practice.

3.1 Characteristics of Multi-agent SLAM

For a multi-agent framework, the previous formulation that is used to describe a generic single-agent SLAM algorithm (2.1) can be generalized for n mobile robots by considering a unique map that has to be updated and n trajectories that have to be estimated from the initial poses, commands and observations of all the agents. An example of generalization for $n = 2$ mobile agents is made explicit in (3.1). The map m and the trajectories $\boldsymbol{x}_t^{(1)}$, $\boldsymbol{x}_t^{(2)}$ are given as probabilistic values that depend on the initial poses $\boldsymbol{x}_0^{(1)}$, $\boldsymbol{x}_0^{(2)}$, the command inputs $\boldsymbol{u}_{1:t}^{(1)}$, $\boldsymbol{u}_{1:t}^{(2)}$ and the observations $o_{1:t-1}^{(1)}$, $o_{1:t-1}^{(2)}$ over the time. Once again, graph-based approaches adopt a more sophisticated formulation.

$$P\left[m, \boldsymbol{x}_t^{(1)}, \boldsymbol{x}_t^{(2)} | \boldsymbol{x}_0^{(1)}, \boldsymbol{x}_0^{(2)}, \boldsymbol{u}_{1:t}^{(1)}, \boldsymbol{u}_{1:t}^{(2)}, o_{1:t-1}^{(1)}, o_{1:t-1}^{(2)}\right] \tag{3.1}$$

The generalities that were described back in Section 2.1 still hold in a multi-agent context. The concepts of convergence and sparsity are inherited with an extended meaning. Since more mobile agents explore the same environment, the expected outcome is that each estimate will converge to a global map that ideally represents the whole environment: this convergence is reached if the mobile agents correctly merge their data with information coming from others. For what concerns sparsity, besides the simplifications that it brings to computation, it is better to have sparse communication, in the sense that it is more efficient to exchange a few data at a time and to have only a few agents that communicate at a time.

However, multi-agent SLAM approaches have to face two distinct problems: scalability and communication. In fact, algorithms have to be generalized to more than one agent without computation growing too much in complexity, and at the same time, such methods have to guarantee some form of communication between the mobile robots. These two requirements make it difficult to implement a multi-agent SLAM framework that is able to achieve good real-time performance in a real-world scenario, and that's why multi-agent SLAM is still a challenging field of research nowadays and it will be in the future.

3.2 Cooperative Solutions and State-of-the-Art

As it was anticipated, scalability and communication are the biggest issues that multi-agent SLAM approaches have to overcome.

The scalability problem can be solved by either using optimized computation or having better on-board processors for the mobile agents, but for cheap devices, some hardware restrictions exist. As a consequence, multi-agent algorithms tend to be founded on concepts that can be efficiently treated from a mathematical point of view such as optimization problems.

Speaking about communication, two alternative solutions may be implemented.

- A centralized client-server approach depends on a server that (approximately) knows the location of all the agents and elaborates the data that the sensors collect during time. This solution can use very cheap agents, because the computation comes from the server and it is resilient against client losses, but the server itself represents a single-point-of-failure of the whole system.

- A distributed approach does not need a server and is based on peer-to-peer communication between the agents. In the absence of prior assumptions, the relative position between agents on the map is computed by on-board processors and robots exchange their own maps each time a rendezvous happens.

Although the client-server setting brings very convenient advantages, the single-point-of-failure problem is a too big drawback and must be avoided, hence multi-agent SLAM algorithms tend to adopt a distributed approach, even if this entails that each agent should dispose of a non-trivial processing unit. Moreover, distributed frameworks respect the sparsity property because there is no need to maintain constant communication among agents. On top of that, centralized solutions tend to perform poorly as the number of mobile agent increases, while distributed algorithms are more suitable for missions with a numerous swarm of robots. The distinction between centralized and distributed approaches extends the single-agent algorithm classification in Fig. 2.1.

Another problem that comes together with peer-to-peer communication is represented by the map merging task. While for a client-server approach each agent indirectly updates the global map individually by sending data to the server, in a distributed setting the issue is much more complicated. During a rendezvous, two mobile robots meet, compute their relative pose and match the two maps. The merging operation details may differ from algorithm to algorithm. For distributed approaches, convergence is a key aspect to take into account: more precisely, the time required for all the agents to converge their own maps to the original global one has to be evaluated. An intelligent navigation system may help in this case: if designed properly, mobile robots will tend to move towards unexplored locations of the environment.

The single-agent algorithms that were described in Section 2.2 can be extended in a multi-agent scenario by developing a centralized setup where the server knows the prior initial position of the agents or has a valid hypothesis about it. This way, map merging becomes easier. Since distributed algorithms bring more advantages, most of the multi-agent SLAM algorithms were designed as brand new approaches. Nevertheless, map merging has to be managed properly whenever two mobile agents have a rendezvous.

Cooperative SLAM algorithms are mostly graph-based: in fact, merging the maps that each mobile agent builds is not a difficult task, because the observations are combined into a unique graph and recurring nodes determine a loop closure. This especially holds for centralized frameworks, but it is also valid for distributed algorithms under the hypothesis that mobile robots will eventually encounter each other, thus having a rendezvous. Such frameworks can also afford to support relatively cheap agents thanks to optimized computation.

On the other hand, multi-robot filter-based SLAM approaches have some issues related to filter processing: it is not only computationally heavy, but also such filters are usually not compact in terms of memory. Hence, it is not guaranteed that regular on-board processor can manage such complex computation. Additionally, it is worth to clarify that not only the agents have to process their own data, but also they may have to incorporate additional information from others, thus leading to a latency due to the high quantity of data to be transferred, and such latency cannot be neglected. That's why filter-based solutions are usually avoided when implementing multi-agent SLAM frameworks in a real-world scenario.

Speaking about the general workflow, each mobile robot individually runs a single-agent local SLAM algorithm on its own, remarking the generic workflow that was shown back in Fig. 2.2. This usually constitutes the front-end of the multi-agent framework and includes the intra-robot loop closure, i.e., the loop closure detection referred to a single mobile robot. If a single-agent SLAM algorithm does not provide intra-robot loop closure, such functionality must be implemented. The back-end core of the framework comes into play during a rendezvous in a distributed setting or continuously for a central server: here, the inter-robot loop closure detection is performed and a further optimization step acts on the global estimates. This is not a rigid workflow for all algorithms, but it gives a better idea of what operations the agents carry out in a collaborative framework.

The following scheme in Fig. 3.1 reports the described workflow for multi-agent SLAM algorithms when two mobile robots are involved. Of course, it can be generalized to an arbitrary number of agents. For distributed algorithms, the exchange of data between agents during a rendezvous is included in the inter-robot loop closure module.

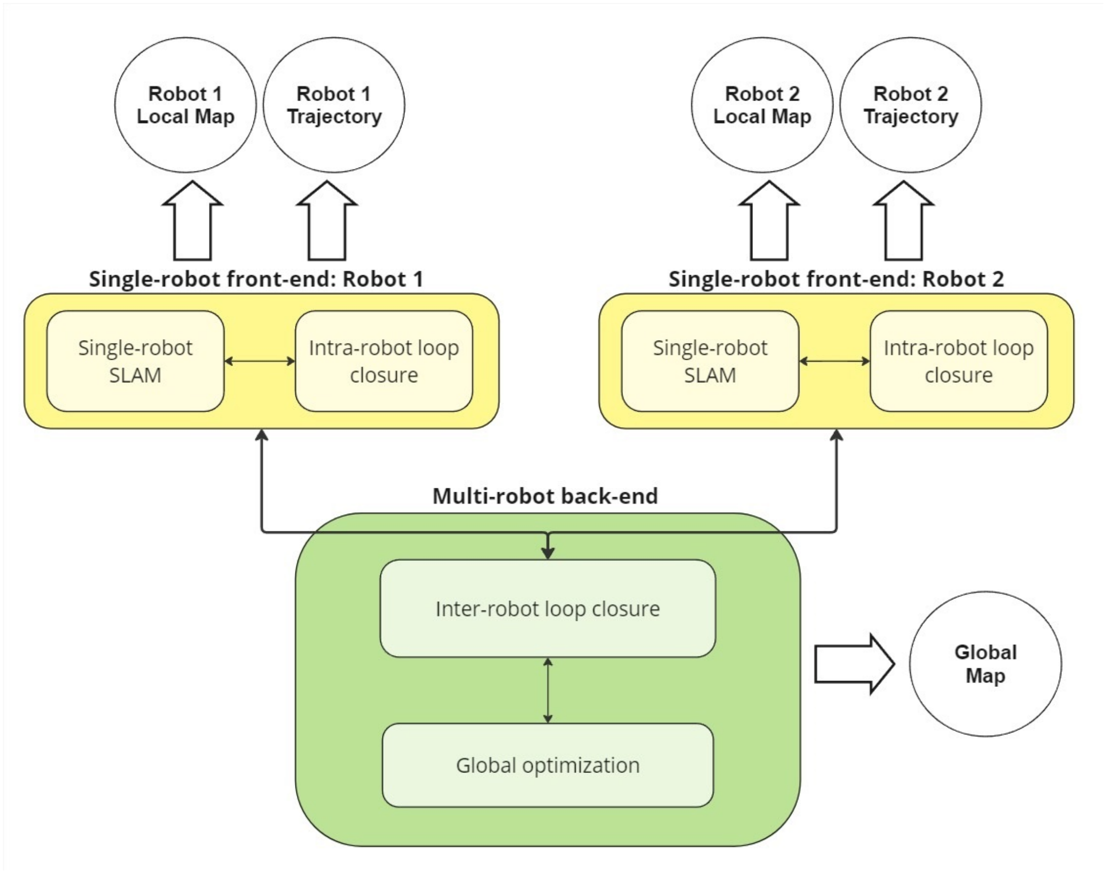


Figure 3.1: Workflow of multi-agent SLAM algorithms with two robots.

3.2.1 Cooperative Filter-based Solutions

Although it does not sound convenient to develop multi-agent SLAM frameworks that are based on particle filters, the theory on probabilistic filters has been mature in the last decade, hence the first multi-agent approaches had their foundations in particle filters. Old and filter-based algorithms can be adapted in order to work in multi-robot systems. Of course, it is pointless to take poorly performing algorithms that would not be used anyway to accomplish single-agent SLAM and build a multi-agent framework, because the resulting solution will inherit the same problems of the original approach.

A multi-agent SLAM framework that is inspired by RBPF-SLAM exploits the principle of Rao-Blackwellization and particle filters generalities in order to build a map with the help of two or more mobile robots [30]. The factorization that was introduced in (2.3) can be extended, in this case to two distinct mobile agents, by separating the computation of the two trajectories that are assumed to be independent. Once again, as (3.2) shows, a mobile robot can exactly recover its own trajectory, since the command inputs and the observations are known, and the initial pose is supposed to be known. When two robots meet, their measurements are fed to the filter and fused into a common map. The quantities m, $\boldsymbol{x}$, $\boldsymbol{u}$ and z respectively refer to the map, the trajectories, the command inputs and the observations.

$$\begin{aligned} &P[m, \boldsymbol{x}^1_{1:t}, \boldsymbol{x}^2_{1:t} | z^1_{1:t}, \boldsymbol{u}^1_{1:t-1}, \boldsymbol{x}^1_0, z^2_{1:t}, \boldsymbol{u}^2_{1:t-1}, \boldsymbol{x}^2_0] = \\ &\quad P[m | \boldsymbol{x}^1_{1:t}, z^1_{1:t}, \boldsymbol{x}^2_{1:t}, z^2_{1:t}] \cdot P[\boldsymbol{x}^1_{1:t} | z^1_{1:t}, \boldsymbol{u}^1_{1:t-1}, \boldsymbol{x}^1_0] \cdot P[\boldsymbol{x}^2_{1:t} | z^2_{1:t}, \boldsymbol{u}^2_{1:t-1}, \boldsymbol{x}^2_0] \end{aligned} \tag{3.2}$$

Actually, this model works only if each mobile robot knows the initial pose of the others. Moreover, the resulting approach would deal with a large state that the mobile robots have to sample as sparse. In the most generic situation, the initial poses of the mobile robots are unknown to others: for the case of two mobile agents, this can be modeled as an additional unknown Δ^i_s that represents the relative pose of the robot i with respect to the other at time s. The factorization gets more complicated and basically generates an additional, anticausal instance for each mobile robot which corresponds to its reversed motion, because it takes Δ^i_s as initial pose with $t = s - 1$ as initial and $t = 1$ as final time instants (the normal motion instance is defined as causal). Based on this idea, a real-time particle-based multi-robot SLAM algorithm is implemented as follows: each mobile robot keeps a queue of its own observations and during a rendezvous it augments its particle filter by acquiring the relative pose and both causal and anticausal instances of the other robot, then it splits the queue into data recorded after and before the rendezvous timestamps in order to progressively incorporate older observations into the map filter.

This extension of RBPF-SLAM for a multi-robot setup is a primitive distributed algorithm that has some structural problems. First of all, particle filters are expensive in terms of memory and mobile agents must have enough storage for both the observation queues and the filters related to the trajectory and the map. Additionally, a considerable latency is introduced during a rendezvous due the great amount of data that is exchanged between two robots. This example demonstrates how many problems filter-based collaborative algorithms have to face in practical terms.

Another extension of grid-based SLAM adopts a single-agent algorithm that was not discussed before known as VinySLAM [31]. In short, VinySLAM uses odometry and laser scans to build a map through a Monte-Carlo scan matcher and evaluates a cost function that minimizes the map discrepancy in order to better improve the map estimate; additionally, a transferable belief model (TBM) enhances the mapping robustness and accuracy by classifying the map's cells as occupied, free, conflictual or unknown. Authors of [32] extended VinySLAM into a multi-robot context, thus giving birth to Monte-Carlo Multi-Agent SLAM: it is a distributed approach that is designed for low-cost platforms with low computational capacity.

Each agent individually runs VinySLAM as the core algorithm to build its own map. As usual, cooperation comes into play whenever a robot detects another one in its proximity. During a rendezvous, the two robots must remain close and for enough time: as the data transfer concerns the robot's observation and an occupancy map with TBM cells, the process of exchanging data may take a few seconds and involve some megabytes. After the data has been exchanged, both agents compute the relative position individually through the Monte-Carlo scan matcher. All robots also run the map merging task on their own: since the two maps are basically bi-dimensional arrays, the map merging takes the form of combining two arrays according to common points and eventual ambiguities are solved by TBM.

The resulting algorithm is computationally easy and does not require high-performing on-board processors. Moreover, the TBM is able to avoid merging problems if the maps have few common cells or conflicts and/or the single-robot scan matching has somehow failed and the map is partially incomplete. The main reason why Monte-Carlo Multi-Agent SLAM seems to be quite effective is the fact that it relies on modern results of the probabilistic particles theory such as TBM.

3.2.2 Real-time Efficient Optimization-based Frameworks

Modern solutions to multi-agent SLAM prefer to base computation on optimization rather than particle filters. In particular, LiDAR sensors are mainly deployed for real-time experiments thanks to their robustness and high resolution, but frameworks tend to be more general, if possible, in terms of supported sensors.

LAMP [33] is an example of a centralized multi-agent SLAM algorithm. It deploys mobile agents that are equipped with both LiDAR sensor and RGB-D camera. Each robot first takes the point cloud data to perform localization, then the RGB-D camera looks for known objects in the area called artifacts to improve odometry. A pose graph optimization (PGO) further improves the trajectory and the map estimate of the single mobile robot. LAMP requires a base station that serves as a centralized back-end: it tracks a common global reference frame that the agents initially share. When possible, a mobile robot sends its estimates to the base station. The centralized back-end module looks for inter-robot loop closures in the global frame and performs PGO on a larger scale. LAMP also allows for a human operator to manually detect an unobserved loop closure if the base station is not aware of it.

DCL-SLAM [34] is a recent fully-distributed collaborative LiDAR-based 3D SLAM framework that relies on local communication to avoid bandwidth problems. As usual, each robot is free to operate autonomously if there are no nearby collaborators. DCL-SLAM is characterized by three components: single-robot front-end LiDAR odometry, distributed loop closure and a distributed back-end module. The distributed components come into play during a rendezvous.

The single-robot front-end module consists on estimating the robot's odometry through a LiDAR-based approach such as LOAM [20] and FAST-LIO2 [22] and also includes an intra-robot loop closure module; as an alternative, both odometry and intra-robot loop closure can be replaced by a complete LiDAR-based single-robot framework such as LIO-SAM [23].

The distributed loop closure performs place recognition according to a data descriptor. Many LiDAR-based descriptors exist, but one with great descriptive power is needed for a collaborative task. The choice fell on LiDAR-Iris [35]: it is a state-of-the-art global descriptor for LiDARs which is robust against sparse or inconsistent point clouds and rotation-invariant. LiDAR-Iris projects the point cloud to its bird's eye view and encodes it by means of simple thresholding; this is done for each incoming point cloud from nearby robots. The loop closure module further divides the bird's eye view in columns and the best one is shared with the other robot. A scan-to-map matching verifies whether the inter-robot loop closure was successful.

The distributed back-end module is dedicated to outlier rejection and PGO. Outliers are recognized as couples of spurious inter-robot loop closures whose related transformations lead to a consistency value, expressed in terms of $l2$-norm, that is lower than a likelihood threshold. The PGO aims at refining the trajectory estimates by solving a maximum likelihood problem of the robot's poses according to the measurements. The problem is solved by a two-stage distributed Gauss-Seidel method that first estimates the optimal rotations of the trajectories and then uses such estimates to find the optimized poses.

Another collaborative framework for SLAM that is worth to be mentioned is Swarm-SLAM [36]: it is a decentralized framework that fully satisfies scalability, sparsity and hardware flexibility for sensors. The Swarm-SLAM architecture is composed of three modules: one for single-robot odometry and mapping, a front-end intra-robot and sparse inter-robot loop detection, and a back-end PGO.

Both the front-end and back-end modules are involved during a rendezvous: one involved mobile robot is randomly selected for computation in order to maintain a one-way data passage. The loop closure detection acts with a two-stage approach for optimization: after one agent sends its own data descriptors, the other mobile robot (i.e., the one that is elected for computation) selects the possible candidates within the global pose graph, then both agents compute their relative 3D pose by exchanging their local pose graphs while avoiding sending vertices in common. Actually, the inter-robot loop closure detection can be completed without exchanging further data. The back-end module estimates the most likely map through a PGO that collects the odometry and both intra-robot and inter-robot loop closures, then both agents are assigned the same reference frame in order to ensure convergence.

3.2.3 The Future of Multi-agent SLAM

Nowadays, the trend of the research on SLAM is moving towards the design of collaborative special-purpose frameworks that are intended to improve performance in specific scenarios.

Even though many multi-agent SLAM algorithms are LiDAR-based or offer support to heterogeneous types of sensor, the necessity to project frameworks that can afford to rely on a poor infrastructure arose. Maplets [37] is a representative example: it is intended to work on cheap agents with RGB-D cameras and limited computational power and under communication constraints. Here, the intuitive idea is to avoid sharing raw data: instead, mobile robots exchange overlapping map elements that are used to enrich their own local maps (also called maplets) and for loop closure with PGO.

As deep learning is constantly growing as a field of research, a promising application of such techniques concerns multi-agent SLAM as well. Already existing frameworks that mix optimization and learning-based approaches include SegMap [38]. A learning-based descriptor that consists of a deep neural network with an autoencoder-like structure is in charge of the processing of the 3D point cloud. As the point cloud is fed to the descriptor, the encoder performs segmentation and the decoder is able to perform both map reconstruction and localization. thanks to transfer learning and the usage of two different loss functions. The extension of SegMap for a multi-agent context results in a centralized approach. where a central machine simulates the system via multi-threading: each thread accumulates measurements and feeds them to the descriptor that is able to be run multiple times in parallel. Even though SegMap may suffer from bandwidth problems, the idea behind it will surely inspire future works on SLAM applications based on deep learning.

The most challenging aspect of the future of SLAM is the possibility of achieving SLAM even under restrictive conditions where the task seems to be unfeasible. Some recent works aim at implementing special-purpose multi-agent SLAM frameworks that can operate in these hostile scenarios.

For example, subterranean environments present repetitive patterns that usually make loop closure detection harder and the uneven terrain may alter the odometry of mobile agents. On top of that, it is difficult to dispose of a network setup with high bandwidth for communication in such environments. LAMP [33] is able to face these issues, thanks to the support of landmark detection to both odometry and intra-robot loop closure. Although the setup requires quite expensive hardware for the mobile agents, thus losing one of the advantages of a centralized algorithm, the operations that are carried out by the agents are designed to cope with the characteristics of subterranean scenarios.

Another challenge for 3D SLAM concerns underwater environments. The biggest limitation that has to be overcome is the signal transmission because underwater robots usually communicate through acoustic signals that usually have low bandwidth and low resolutions and also lack of 3D information. Another problem concerns the lack of initial information on mobile agents because GPS is not reliable here, hence only distributed approaches should be considered. On top of that, cameras perform poorly due to low lighting, especially in turbid waters. DRACo-SLAM [39] is the first solution in this regard: it uses sonar imaging, i.e., robots transmit sound pulses and convert the returning echoes into digital images. The point clouds are built as 2D images with acoustic intensity values and then compressed in order to fit in the data link. Pose graphs are based on sequential scan matching on such point clouds and the same histogram-based descriptor is used by all the agents to detect inter-robot loop closures. DRACo-SLAM also includes an outlier rejection for erroneous loop closure according to point cloud comparison.

Chapter 4

Introduction to ROS

ROS stands for Robotic Operating System and is an open-source framework that includes software libraries and other tools that support the development of robot applications. ROS is not properly an operating system, but it still provides some functionalities that are typical for operating systems: ROS is responsible for hardware abstraction, low-level device control, message-passing between processes and package management. For this reason, ROS is usually properly defined as a meta-operating system. Initially, ROS was used for academic and research purposes, but nowadays the robotics industry tends to produce commercial robots that are implemented with ROS. Here, the abbreviation ROS refers to the generalities, while a clear distinction is made between the two existing versions (ROS1 and ROS2).

The first distribution of ROS1 [40] was released in 2007. As ROS1 was quite old, developers decided to rewrite the whole framework from the ground up: thus, ROS2 [41] was released in 2017 in order to solve the structural problems that affected ROS1. Nowadays, many existing projects have been developed through ROS1, even though the community is willing to migrate towards ROS2.

In general, ROS needs a massive usage of a CLI (Command Line Interface) in order to run tools and other programs. From a coding point of view, ROS follows an OOP (Objected Oriented Programming) approach, hence the code is usually written in C++ and Python. As a meta-operating system, ROS requires an actual operating system to run on: ROS1 is compatible only with Linux-like operating systems, while ROS2 offers additional support for others such as Windows and iOS.

Besides the choice between ROS1 and ROS2, there are some basic concepts that have to be explained in order to better comprehend ROS. In this chapter, these common concepts will be illustrated, as well as the main innovations brought by ROS2 with respect to ROS1. A description of the software tools that will be fundamental for explaining the setup for the real-time experiments follows. Finally, the chapter closes with an overview of the hardware devices that have been used to perform the experiments and collect the results. Before starting, it is crucial to specify that some ROS commands will be introduced in order to give a better workflow and eventually a practical feedback for readers: such commands are intended to work on ROS2, because it is the ROS version that will be used for the implementation of the multi-agent SLAM frameworks.

4.1 ROS Basics

Conceptually, ROS follows a "divide and conquer" approach: each task is divided into smaller programs that communicate with each other. Thus, a complex task can be represented as a computational graph that contains modules and inter-communication channels. ROS defines a proper terminology to describe the main components of a program.

A node corresponds to a sub-program of the task and represents a single process that is responsible for the behavior of a unique sub-task. Nodes are defined through ROS libraries that are compatible with C++ and Python, and they are able to send and/or receive data from others. In ROS1, the so-called master node is responsible for naming other nodes and tracking the communication link between them: it enables other nodes to locate each other such that they can communicate peer-to-peer. For this reason, it is said that ROS1 uses a master-slave communication protocol. The master node is deprecated in ROS2.

The data that nodes exchange goes in the form of a message. In ROS, a message represents a compact, aggregate and typed piece of data. A message may include some data with standard types (e.g., boolean, integer, string) or more complex ones that are defined by ROS libraries and are strongly specific for certain applications. Custom types may be defined as well, usually in textual `.msg` files.

Topics are directed inter-communication channels through which nodes send and/or receive data (i.e., messages). Given a topic, a node can either publish on this topic or subscribe to it: the node of the first type is called publisher, while the latter is known as subscriber; it is possible to have more than one publisher and/or subscriber for a single topic. A message that a publisher sends on a topic is received by all the subscribers to that topic. When subscribers receive a message, they might execute a callback function.

Services are an alternative method of communication between nodes. A service involves a node that plays the role of the server and one or more other nodes that are called clients. Similar to how client-server communication works, a client node sends a request to the server that elaborates it and sends a response back. This represents a way to implement a synchronous communication between nodes and is not intended to be used for exchanging messages continuously. ROS2 supports asynchronous services as well.

Speaking about continual communication, the last kind of communication that can be implemented in ROS is called action. Indeed, actions are intended for long-running tasks and employ a server node and one or more client nodes, similarly to services. Actions also include a goal, a result and a feedback. Actions not only provide a standard client-server communication, but they are also preemptable and can provide constant feedback if requested.

ROS uses a build system in order to compile the packages and eventually resolve dependencies between them. ROS1 uses `catkin`, while ROS2 uses `ament`: they both add API to an underlying build system which is usually CMake. A ROS workspace generally includes different folders.

- The source space (`src` subdirectory) includes the source code of the packages.
- The build space (`build` subdirectory) contains cache files and other temporary information related to the build system;
- The development space (`devel` subdirectory) represents the folder where built targets are placed before being installed, and it is particularly useful for testing. It is deprecated in ROS2.
- The install space (`install` subdirectory) contains the installed targets; in practice, it is not used at all in ROS1 because packages can be built without installing them. However, installation is mandatory in ROS2 and this is why ROS2 does not support a development space. In ROS2, the install space also inherits the utilities of the deprecated development space.
- The log space (`log` subdirectory) includes the logging information from console output during building. It is mainly present in ROS2.

Each single ROS package follows a similar hierarchical structure as well. It is good practice to use different subfolders for distinct file types. A ROS package may contain Python or C++ scripts, launch files, configuration files, C++ headers and so on. In general, all the packages contain two essential files: a build file `CMakeLists.txt` that contains instructions to be passed to CMake and a package metadata file `package.xml` that includes dependencies from other packages. Once ROS packages are built, the `install/setup.bash` file has to be sourced. Then, executables can be run by a `run` command, or through a `launch` command if a launch file is provided. A complete example of the organization of a workspace in ROS (in particular, ROS2) is reported in Fig. 4.1.

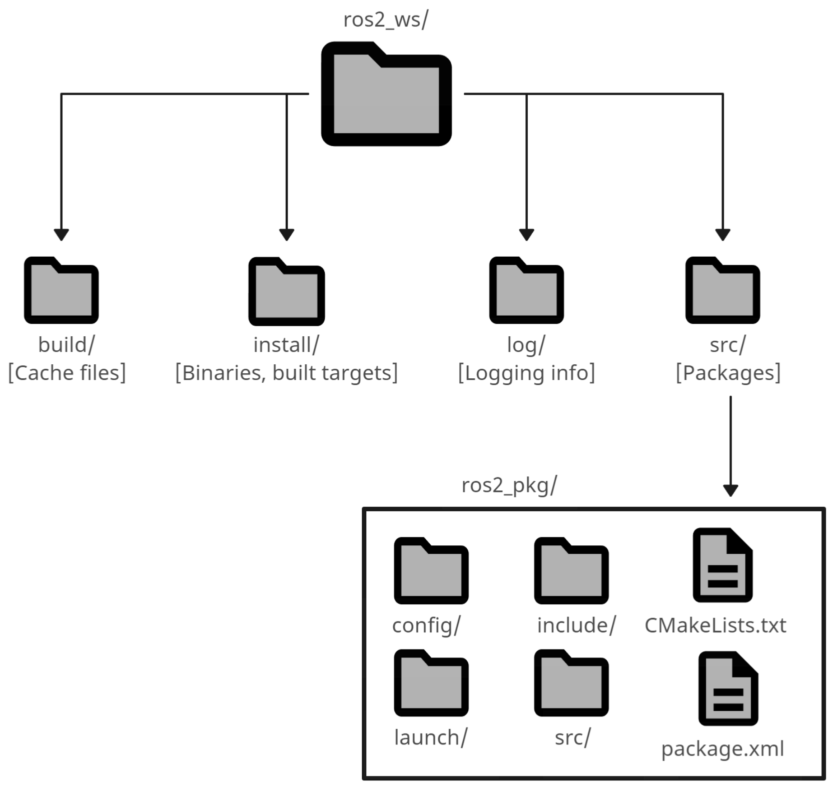


Figure 4.1: Typical organization of a ROS2 workspace.

4.1.1 ROS2

ROS2 was born to solve some problems that were left open in ROS1: these include security, real-time issues, single point of failure for the master node and so on, and such problems were crucial for the development of real-time robotic applications. As mentioned before, this led the developers to rewrite the entire framework, thus giving birth to ROS2.

The basic elements are conceptually left untouched. The biggest difference lies in the architecture: ROS1 relies on a master-slave architecture at application level and the middleware layer uses ROSTCP/ROSUDP (i.e., adaptations of TCP/UDP for ROS1) as communication protocol; on the other hand, ROS2 follows a peer-to-peer approach at application layer and adopts DDS (Data Distribution Service), thus granting higher efficiency, low latency, scalability and providing Quality of Service policies in order to better support real-time robotics applications. Moreover, since more mobile robots may communicate through the Internet, DDS provides security guarantees. The absence of a master node in ROS2 also opens to the possibility of developing fully-distributed applications. This substantial difference is highlighted in Fig. 4.2.

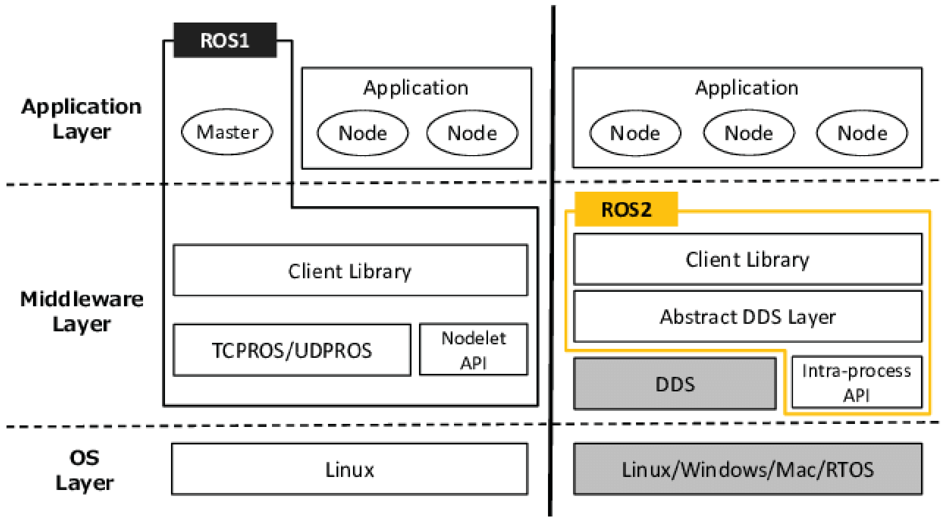


Figure 4.2: A comparison between ROS1 and ROS2 architectures [42].

ROS2 also introduces the concept of lifecycled nodes. Basically, they are nodes that can assume one of the following states: unconfigured, inactive, active and finalized; these states are regulated by a finite state machine. This feature aims at solving real-time problems: in fact, a life cycle grants that each node has been correctly instantiated before the application is executed. A node is initially unconfigured until publishers and subscribers (or other types of communication) are established, then it becomes inactive and does not send and/or process data: this state allows re-configuration without compromising the application. When the node is active, it processes data, publishes on topics, produces console output and so on. Before the node is destroyed, it translates into the finalized state: this is useful for debugging and also ensures that the node is shut down correctly. In particular, it is fundamental for real-time applications.

Besides the structural differences, the API for C++ and Python is similar between ROS1 and ROS2. Of course, there are some small differences that mostly concern libraries, definitions and solving dependencies. However, since the support for ROS2 is not as consistent as for ROS1, it may be needed to rewrite some packages from ROS1 in order to be fully compatible with ROS2.

This thesis work uses ROS2 Foxy Fitzroy as ROS distribution. ROS2 Foxy Fitzroy was released in June 2020 and is mainly supported on Ubuntu 20.04 (Focal), Mac macOS 10.14 and Windows 10. However, it reached its end of life in June 2023, hence it will not receive official support from developers anymore. Although ROS2 Foxy Fitzroy suffers from several bugs and issues and has become an end-of-life distribution, there is still a lot of documentation and support from the community.

4.2 Software Utilities

ROS includes some useful libraries, software, plugins and simulation tools that are crucial for debugging, modeling, simulation and setup for real-time tasks. Many types of file may exist in a ROS package, each with a different purpose. Here, more focus is given to ROS2, since it is the framework that will be used for the real-time experiments in Chapters 5 and 6, although the corresponding software tools exist in ROS1 as well and with small differences.

4.2.1 The tf2 Library

The tf2 ROS2 library [43] concerns the transformations between the different reference frames that are tracked by the framework over time. These transformations are kept as a tree that graphically shows the relationship between the coordinate frames. This is important for two reasons: this way, it is possible to check whether ROS is recognising all the joints of a robot (in fact, a joint usually corresponds to a reference frame on its own) and whether the transformations have been correctly set. The transformation tree can be generated by running the following command, then it is automatically saved into a file called `frames.pdf`.

```
ros2 run tf2_tools view_frames.py
```

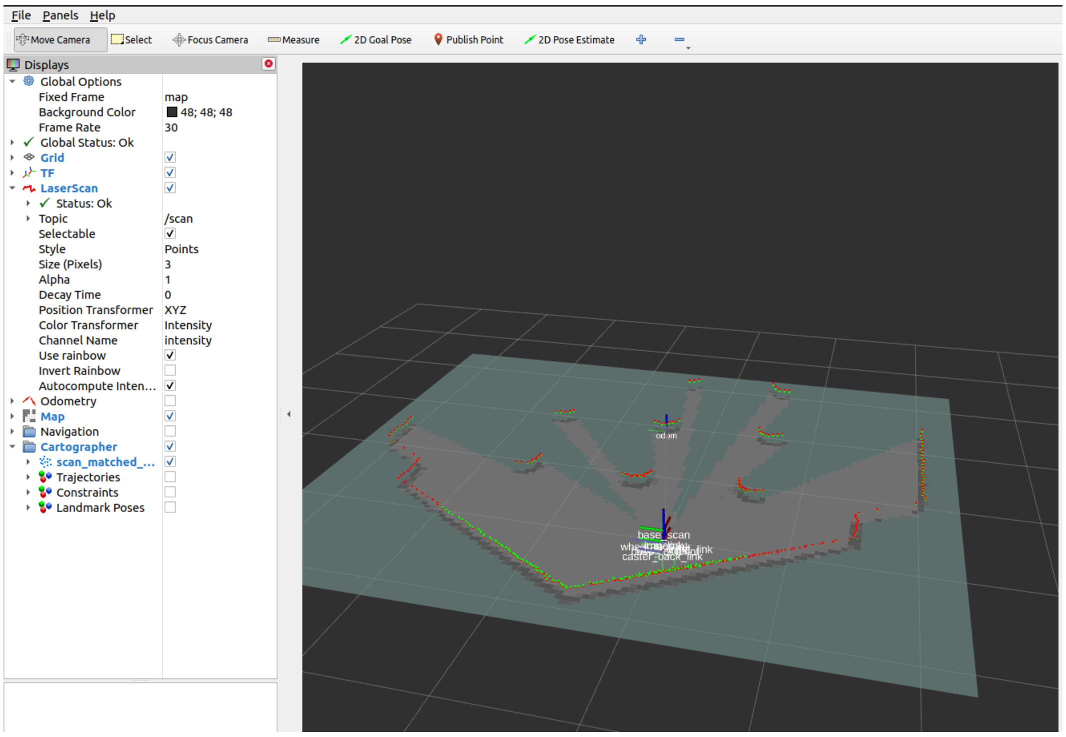
Another useful command allows to listen to the transformation between two arbitrary topics.

```
ros2 run tf2_ros tf2_echo <from_frame> <to_frame>
```

Whenever tf2 is involved, it creates two topics called `/tf` and `/tf_static`: their objective is to compute the translation and the rotation (expressed in quaternions) between two reference frames. The difference is that `/tf_static` only tracks static transformations between two reference frames whose relative position should not change during time. On the other hand, `/tf` computes only dynamic transformations: for example, the transformations between the odometry frame of a mobile robot and its base frame is dynamic and the associated translation and rotation vary while the robot moves. The previous command listens to the data that is published to the `/tf` and `/tf_static` topics.

4.2.2 RViz2

RViz2 [44] is a visualization plugin for ROS2. Its functionalities are inherited from RViz (i.e., the version for ROS1) and provide a graphical interface that displays the robot and the environment in which it is located, as well as the reference frames and other features of interest such as the trajectory. RViz2 also supports visualization for point clouds, sensing, body tracking, object detection and other ROS topics. It works in both real-world scenarios and simulations, provided that a real robot is connected or a simulated environment is given.



```
Displays
Global Options
    Fixed Frame               map
    Background Color          48; 48; 48
    Frame Rate                30
✓ Global Status: Ok
Grid                          ✓
TF                            ✓
LaserScan                     ✓
  ✓ Status: Ok
    Topic                     /scan
    Selectable                ✓
    Style                     Points
    Size (Pixels)             3
    Alpha                     1
    Decay Time                0
    Position Transformer      XYZ
    Color Transformer         Intensity
    Channel Name              intensity
    Use rainbow               ✓
    Invert Rainbow            ☐
    Autocompute Inten...      ✓
Odometry                      ☐
Map                           ✓
Navigation                    ☐
Cartographer                  ✓
    scan_matched_...          ✓
    Trajectories              ☐
    Constraints               ☐
    Landmark Poses            ☐
```

Figure 4.3: The RViz2 GUI.

The graphical environment is highly customizable through a `.rviz` file that can be passed when launching RViz2 either directly from the command line or through a launch file that executes RViz2. According to the user's preferences, visualization for topics can be enabled or disabled through the dedicated GUI while the application runs. If there is an issue related to the visualization of a topic, RViz2 also shows a warning or an error message: this represents a naive way of debugging topics while the application runs. In the context of SLAM, RViz2 is necessary because it visually gives a map estimate that the robot computes in real-time and its trajectory while it wanders in an unknown environment.

4.2.3 Gazebo

Gazebo [45] is a 3D open-source simulation software for robotics. It provides modeling of robotic systems and indoor and outdoor environments, as well as a physical simulation that offers support for different sensors and accurate models for physics. Scenarios can be highly customized: Gazebo comes up with a wide choice of terrains and obstacles such that the user is able to build an environment that is suitable according to his/her needs. These characteristics make Gazebo a reliable, realistic simulation tool for testing robotics algorithms and designing robots before moving on a real-world scenario. On top of that, Gazebo includes a user-friendly GUI that allows to manually edit the environment by adding obstacles, robot models and other elements to the scene. Alternatively, the environment can be encoded in a textual `.urdf` file. Shortly, a `.urdf` file follows a syntax that is very similar to XML files such as the `package.xml` that contains the package metadata.

Gazebo is also integrated with ROS, thus enhancing the development of ROS-based robotic systems and applications. It is good practice to test a ROS application in a simulated environment first. For what concerns SLAM, a simple indoor environment is enough to test the basic behavior of the algorithm, but more complex scenarios can be designed in order to see the response of the SLAM algorithm in certain situations. For example, an environment with a repetitive pattern could be useful in order to check the quality of loop closure.

A last note that concerns ROS is related to time. In fact, ROS applications can either use a simulated clock or the local system clock for timing. This is crucial when dealing with timers, frame transforms and synchronous calls. The `use_sim_time` parameter determines which clock should be used as reference. While testing in a simulated environment, it is required to pass `use_sim_time:=true` as an additional argument when executing a launch file. Otherwise, `use_sim_time:=false` should be set.

4.2.4 Nav2

Nav2 [46] is the ROS2 Navigation Stack. It provides a plethora of plugins with different purposes that are related to navigation. Nav2 offers software tools for loading and storing maps, localization, path planning, costmap building, obstacle perception and recovery in case of failure. For example, in order to localize a robot within a map, Nav2 implements AMCL (Adaptive Monte-Carlo Localization), that is an improvement of the Monte-Carlo scan matcher that some particle-based SLAM algorithms adopt. Citing other features, Nav2 relies on behavior trees for point-to-point path planning and integrates a smoother such that the global path tends to be continuous. Nav2 also integrates the concept of lifecycled node that was introduced with ROS2: this way, undeterministic behaviors that may affect the performance in real-time applications are avoided. The entire navigation stack can be launched with some default options by running the following command.

```
ros2 launch nav2_bringup navigation.launch.py
```

This is enough to send goal poses to the mobile robot through the RViz2 GUI to support the SLAM process. Before starting to navigate, it is crucial to initialize the robot's pose. More complex applications that strictly rely on path planning and motion control need a proper tuning.

Many of these tools are originally intended to work in static scenarios, i.e., in situations where the map is pre-built and the navigation servers control the motion of a mobile robot inside the environment. This implementation is useful for logistics and transport because it enhances the mobile robot's autonomy. Going towards additional support for SLAM algorithms, it is possible to design applications that lean on Nav2 and allow autonomous exploration for a mobile robot in an unknown environment: for example, such an application may send the mobile robot towards zones that are not explored yet, according to the local map that the robot builds during time.

4.2.5 Computational Graph Visualization

When launching a ROS executable, an effective way to understand the correct functioning of the program is to analyze which nodes and topics are present in the system, and it may be useful to understand the specific connections between nodes, i.e., to detect those topics where a process publishes or receives data. Such information is contained in the ROS computational graph. The RQT plugin (also known as ROS Qt GUI toolkit) is able to visualize the ROS computational graph at the current state. RQT also groups topics and nodes according to their namespace. Visualizing the computational graph gives an intuitive idea on the high-level workflow of a ROS executable. The following command invokes RQT and returns the ROS computational graph.

```
rqt_graph
```

4.3 TurtleBot3

TurtleBot is a series of low-cost ROS-based mobile robots that are mainly used for education and research purposes. Each TurtleBot version (the latest one is TurtleBot4) supports open-source packages for initialization, navigation and mapping and is equipped with a motion system and proper hardware that is capable of capturing and processing data. For each version, some models that differ in hardware specifics exist.

In order to carry out the real-world experiments for multi-agent SLAM, two copies of TurtleBot3 Burger are used. TurtleBot3 Burger [47] is a three-wheel mobile robot with two front wheels that are commanded by two separate motors and a rear caster wheel. A RaspBerry 4 acts as a single board computer (SBC) and should contain both the underlying OS and the ROS environment, as well as the essential packages that are designed for TurtleBot3 [48]. The SBC is connected through USB cables to the other two main hardware components of the robot: the LiDAR sensor and an electronic board. RaspBerry 4 is also provided with onboard wireless networking, hence it is possible to control the mobile robot remotely from another host (for example, through a `ssh` command). However, another step is required for ROS2 in order to have complete accessibility to the ROS functionalities that are related to TurtleBot3. In fact, DDS computes the UDP ports that are used for communication between ROS2 processes according to a ROS environment variable called `ROS_DOMAIN_ID`. Both the remote host and TurtleBot3 must be assigned the same `ROS_DOMAIN_ID`, otherwise, the devices cannot communicate through ROS.

An electronic board is also required as an interface between the SBC and the wheels of the mobile robot. OpenCR1.0 is specifically developed to support ROS embedded systems and provides open-source hardware and software. As an electronic board, it includes LEDs, switches, GPIOs, pins and supports external interrupts. OpenCR1.0 directly commands the two disjointed DYNAMIXEL motors that control the two front wheels: this way, wheels can be commanded independently and the robot is able to spin.

As it was already explained, LiDAR and IMU sensors are crucial for accomplishing SLAM. While LiDAR sensors provide visual information with high frequency, IMU sensors acquire useful measurements that can be also processed for motion

Figure 4.4: TurtleBot3 Burger [47].

correction. It is important to report the sensors' specifics because some SLAM algorithms may be not compatible with TurtleBot3 Burger and require some adaptation.

A LDS-02 LiDAR sensor (also known as LD08) is mounted atop the mobile robot. It is a 2D mechanical laser, hence it is based on a rotational mechanic that allows to reach a 360° field of view in the horizontal direction and can scan at a fast speed. As a laser sensor, it is resilient against light interference. The open-source packages for TurtleBot3 Burger also provide drivers for LDS-02. Detailed specifics are reported in Table 4.1.

Figure 4.5: LDS-02 LiDAR sensor.

The ICM-20648 IMU sensor is integrated in the OpenCR1.0 board. It includes a 3-axis gyroscope, a 3-axis accelerometer, a 3-axis magnetometer and an embedded digital motion processor that gathers and processes data from the other components

Characteristic	Value
Distance Range	160$\tilde{8}$000 mm
Scan Frequency	5.0 Hz
Angular Range	360°
Angular Resolution	1°
LiDAR Channels	8

Table 4.1: LDS-02 specifics.

in order to offload timing requirements and computational power from the SBC processor. Overall, ICM-20648 is a high-precision 9-axis IMU sensor, but it can be integrated with additional components. More details are reported in Table 4.2.

Characteristics	Value
Accelerometer Noise	0.00230 m/s^2
Gyroscope Noise	0.015 rad/s
Accelerometer Bias	0.00025 m/s^2
Gyroscope Bias	0.005 rad/s
IMU Frequency	200 Hz

Table 4.2: IMU specifics.

4.3.1 TurtleBot3 Burger Topics and Frames

The essential packages for TurtleBot3 Burger [48] are required to correctly work with a real copy of the mobile robot. Even though it is not important to go into detail for such packages, some generic concepts have to be remarked because they not only recall the basics of ROS but also are crucial to understand the implementation of multi-agent SLAM frameworks later on.

After the robot is assembled, switched on and correctly set up according to the official documentation [47], it has to be correctly recognized by the ROS environment. The `turtlebot3_bringup` package acts as a wrapper that initializes the mobile robot into the ROS framework. Three essential packages are called: `turtlebot3_node` sets up the different devices and their related ROS topics, while `turtlebot3_description` and `robot_state_publisher` provide a physical description of the TurtleBot3, as well as the various reference frames and their relative transformations. The command that initializes the TurtleBot3 Burger is reported below and must be executed on the TurtleBot3 Burger terminal.

```
ros2 launch turtlebot3_bringup robot.launch.py
```

After the command is launched correctly, it is possible to recognize that some topics have been added. The complete list of the active ROS topics can be visualized by running `ros2 topic list`.

- `/parameter_events` and`/rosout` are standard ROS2 topics: the former assigns the lifecycle of ROS parameters to the related node, while the latter is responsible of some core ROS functionalities such as publishing and subscribing.
- `/battery_state` tracks the battery state of TurtleBot3 Burger.
- `/cmd_vel` is the topic where linear and angular velocities can be published as command inputs to the mobile robot. `turtlebot3_teleop` is an essential package that creates a node that publishes to this topic, thus making it possible to remotely control the motion of TurtleBot3 Burger through a keyboard. The linear and angular velocities are capped to 0.22 m/s and 2.84 rad/s, respectively.
- `/imu` contains the information that the IMU sensor perceives: orientation, angular velocity and linear acceleration. The type `sensor_msgs::msg::Imu` describes IMU messages.
- `/joint_states` tracks the wheels' position with respect to the base and their velocity.
- `/magnetic_field` shows the measured intensity of the magnetic field.
- `/odom` contains the information about the dynamic transformation between the odometry frame and the robot's base footprint on the floor. Odometry data are collected in messages of type `nav_msgs::msg::Odometry`.
- `/robot_description` is a deprecated topic that physically describes the mobile robot according to a `.urdf` file.
- `/scan` shows data that the LiDAR sensor captures as laser scan messages. LDS-02 publishes data as scan points that are represented by messages of type `sensor_msgs::msg::LaserScan`. Some LiDAR-based algorithms may need to process data in a point cloud format (`sensor_msgs::msg::PointCloud2`), unless they internally provide a conversion. Conceptually, a laser scan is a set of points in a polar coordinate system, while a point cloud is expressed in a Cartesian coordinate system.
- `/sensor_state` shows information about the sensors' state.

- `/tf` and `/tf_static` are the topics that contain the information about the transformations between the reference frames that characterize TurtleBot3 Burger. `/tf_static` tracks the static transforms, while `/tf` refers to dynamic transformations for reference frames whose relative position should change during time.

Along with the topics, some reference frames are generated and are related to the components of TurtleBot3 Burger. The information about the joints, the frames and their relative transformation is stored in a `.urdf` file that is provided by the essential packages. The only transformation that is not defined by the `.urdf` file is the `odom` $\rightarrow$ `base_footprint` one, because the `odom` frame is defined by the odometry system during the hardware initialization. The odometry system tracks the motion of the mobile robot on the floor, and `base_footprint` represents the projection of the robot's base on the ground.

The transformation tree in Fig. 4.6 shows the relationship between the frames that characterize TurtleBot3 Burger. Besides the `odom` frame, the others refer to the physical components of the mobile robot. The `base_link` frame refers to the physical base of TurtleBot3 Burger. A reference frame exists for each wheel and for each main sensor (i.e., the IMU and the LiDAR).

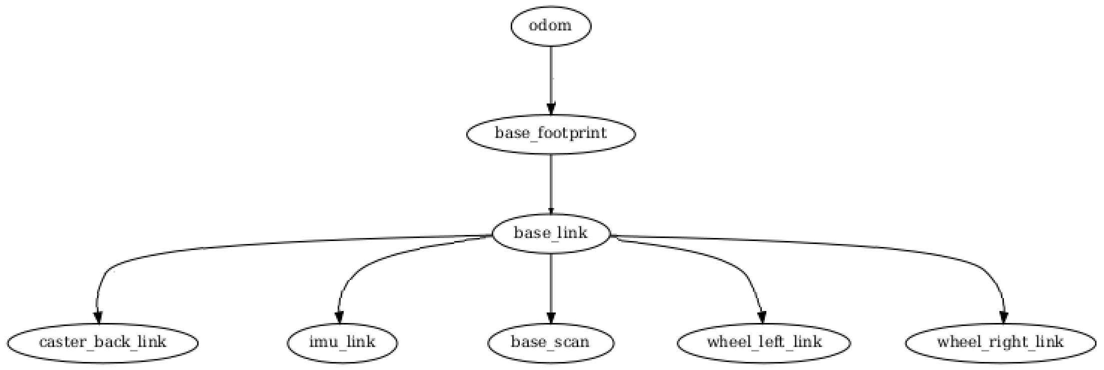


Figure 4.6: Transformation tree of TurtleBot3 Burger.

4.3.2 Namespacing TurtleBot3 Burger

As long as the SLAM task involves a single copy of TurtleBot3 Burger, basic packages do not need any change. However, multi-agent SLAM requires that each physical mobile robot is distinguishable from others. In general, ROS offers the possibility to assign a namespace to nodes, topics and reference frames. There are two strategies to implement namespacing: it can either be hard-coded by manually adding a namespace to strings in the source code, or alternatively it can be implemented in launch files through the remapping functionality. Shortly, remapping helps to rename topics that ROS nodes publish or subscribe to. Of course, remapping is strongly recommended because it requires less time and leaves the source code unaltered. However, remapping does not affect reference frames. Here is an example about adding a remapping on a launch file that invokes `turtlebot3_node`.

Listing 4.1: Adding a namespace to TurtleBot3 Burger in the launch file.

```
return LaunchDescription([
    # [Other nodes to be launched here...]
    Node(
        package='turtlebot3_node',
        executable='turtlebot3_ros',
        namespace='Alice',
        remappings=[('/imu','/Alice/imu'),
                    ('/odom','/Alice/odom'),
                    # [And other topics...]
        ],
        # [Other options if needed here...]
    )
]
```

This way, TurtleBot3 Burger and its topics are correctly registered under the namespace `Alice`. A similar concept also holds for `robot_state_publisher` in order to add a namespace to reference frames. The package `robot_state_publisher` provides a namespacing functionality for reference frames through the parameter `frame_prefix`: hence, it is sufficient to put `'frame_prefix'='Alice/'` in the launch file as parameter of the `robot_state_publisher` node.

An important observation concerns the topics related to the transformations: in fact, it is recommended to not add a namespace to `/tf` and `/tf_static`, otherwise the functionalities from the tf2 library might not work properly; instead of remapping such topics, reference frames should be given a namespace as explained before. However, it is sometimes necessary to add a namespace to the transformation topics, especially when there are two SLAM nodes that should publish transformations between homonymous frames: in this case, it can be avoided to remap frames because the two SLAM nodes would publish to different `/tf` and `/tf_static` topics. In the end, these two solutions are almost equivalent in theory, but some ROS-based SLAM algorithms would require only one of them according to their implementation.

In this work, the second method for namespace is used: hence, there are more couples of transformation topics, one for each robot.

In the following chapters, it is assumed that the two copies of TurtleBot3 Burger have their own unique namespace. During the explanation of the two multi-agent SLAM approaches, they are denoted as Alice and Bob for more ease of understanding. In the real implementation, the namespaces `a` and `b` have been used. When both robots are launched, the ROS computational graph captures nodes and topics under their respective namespaces. The graph that appears after running RQT is reported in Fig. 4.7.

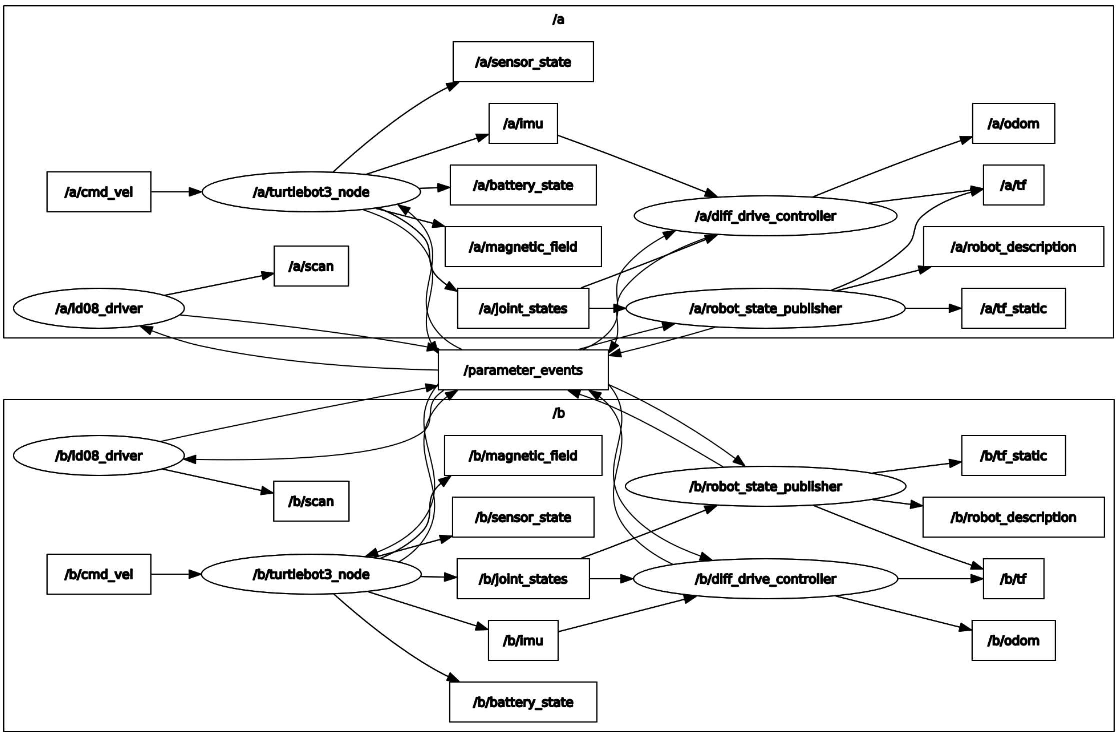


Figure 4.7: ROS computational graph after launching two copies of TurtleBot3 Burger.

Chapter 5

Centralized Setting for Multi-Agent SLAM

The first implementation of a multi-agent SLAM approach is a naive, centralized setting where the two mobile agents do not exchange any information. Instead, a single-agent SLAM algorithm is chosen as front-end of the complete approach. The selected algorithm runs locally as a ROS2 node that is related to a single mobile robot. While the two robots move around and build their own map, a back-end map merging node takes the local maps and merges them into a unique, global one. Any computer vision technique that is able to detect and match visual common features from two distinct images can do the job quite well.

Remarking some of the characteristics of centralized multi-agent SLAM algorithms, such approaches require that the initial pose of the mobile agents is (approximately) known. Client-server settings are also easier to develop, because the only component that brings complexity is the map merging node. However, it must not be forgotten that centralized multi-agent SLAM algorithms have a structural problem: indeed the central server still represents a single-point-of-failure and its stability must be guaranteed during the whole task.

The presented solution can be easily generalized for each SLAM algorithm that is based on grid maps. Following in this chapter, the choice of the single-agent SLAM algorithm to run and its implementation in ROS2 are discussed. Then, the back-end map merging node is explained and different options for feature matching are compared. A demonstration in a simulated scenario is described as well.

5.1 Evaluation for the Single-Robot Front-End

A suitable single-agent SLAM algorithm has to be selected according to the available mobile agents. Since TurtleBot3 Burger supports LiDAR sensors, there is a wide choice for what concerns the single-robot front-end module. Some algorithms that were discussed back in Section 2.2 are valid candidates to serve this purpose.

A first evaluation discards filter-based approaches in favor of graph-based solutions. In fact, filter-based algorithms may perform poorly, because they lack of an in-built loop closure functionality, and in general they get outperformed by SLAM techniques that rely on optimization. Furthermore, an essential requirement for the back-end centralized map merging node is that the single-robot SLAM algorithm must generate a grid map. This is why it is a better option to select a graph-based SLAM algorithm, especially if this requires a real-world showcase. Alternatives that rely on grid maps but not on graph optimization yet such as GMapping and FastSLAM can be contemplated. However, they certainly do not represent the most suitable choice because particles are heavier to process.

At this point, two different alternatives can be implemented as front-end algorithms of the centralized multi-agent SLAM framework. These algorithms build a grid map, and most importantly a ROS integration exists for both.

- Cartographer [16] was exhaustively discussed in Subsection 2.2.2. ROS includes some packages that allow to tune Cartographer's parameters and run it in simulation and real world. TurtleBot3 Burger standard packages also include an interface for Cartographer and a first set of parameters that respect the specifics of LDS-02.
- KartoSLAM [13] shares some common characteristics with Cartographer such as the application of SPA [17] for loop closure and back-end pose graph optimization. Currently, KartoSLAM's basic algorithm includes a Ceres solver that enhances pose graph optimization. The open-source ROS package `slam_toolbox` [49] provides such functionalities, as well as two operation modes: synchronous mapping keeps a buffer of measurements to gradually process, thus improving mapping but slowing real-time processing, while asynchronous mapping elaborates real-time measurements and does not include other valid scans until the previous elaboration has ended, hence it is more suitable for real-time tasks [50]. SLAM Toolbox is used as an evolution of KartoSLAM that improves performance during scan matching and optimization.

Nowadays, there are some works [51], [52] that compare the performance of

Cartographer and KartoSLAM during localization, mapping and autonomous exploration. Focusing on the tasks that are concerned by the SLAM, it is shown that KartoSLAM outperforms Cartographer for both localization and mapping. However, these comparisons take an out-of-the-box version of Cartographer that is not properly tuned. Hence, some parameters that influence the Ceres solver for scan matching and the SPA for pose graph optimization have been modified, such that the Ceres solver tends to reject scan matches that differ too much from those that were taken in a prior pose, and the back-end computational load is reduced for a better real-time loop closure. SLAM Toolbox is run without any essential modification and the asynchronous mode is selected.

Since the real-world experiments take place in an indoor environment, a good proving ground for the evaluation is the TurtleBot3 House that comes with TurtleBot3 Burger essential packages. The layout of the simulated environment in Gazebo is shown in Fig. 5.1 and the initial position of the simulated mobile robot is highlighted. The following command runs this simulated environment and spawns the robot.

```
ros2 launch turtlebot3_gazebo turtlebot3_house.launch.py
```

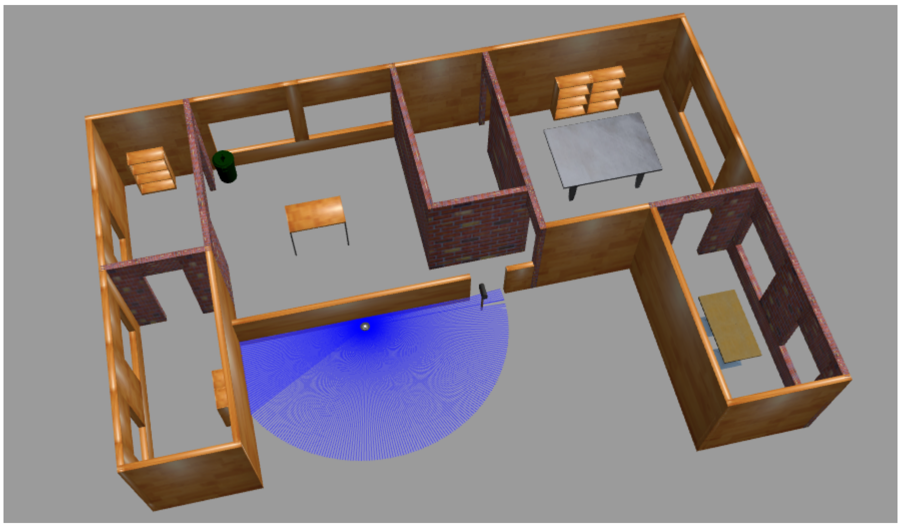


Figure 5.1: TurtleBot3 House.

Cartographer is launched first, then TurtleBot3 Burger is controlled by keyboard after launching the following command. The same procedure applies for SLAM Toolbox after restarting Gazebo.

```
ros2 run turtlebot3_teleop teleop_keyboard
```

During the two experiments, almost the same trajectory is kept. Furthermore, the imposed commands do not exceed 0.15 m/s for the linear velocity and 0.5 rad/s for the angular velocity: this is particularly useful for SLAM Toolbox because the scan matching is usually slow (and most importantly, it is not recommended to move the robot at high speed, especially while rotating). Another possibility could be running the navigation stack and interacting within the RViz2 environment by sending goals to the mobile robot. The results given by Cartographer in Fig. 5.2 and SLAM Toolbox in Fig. 5.3 are compared.

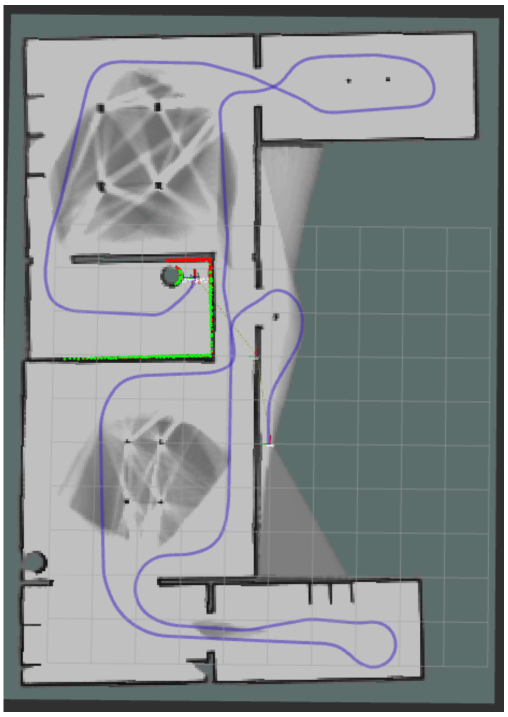


Figure 5.2: Mapping with tuned Cartographer.

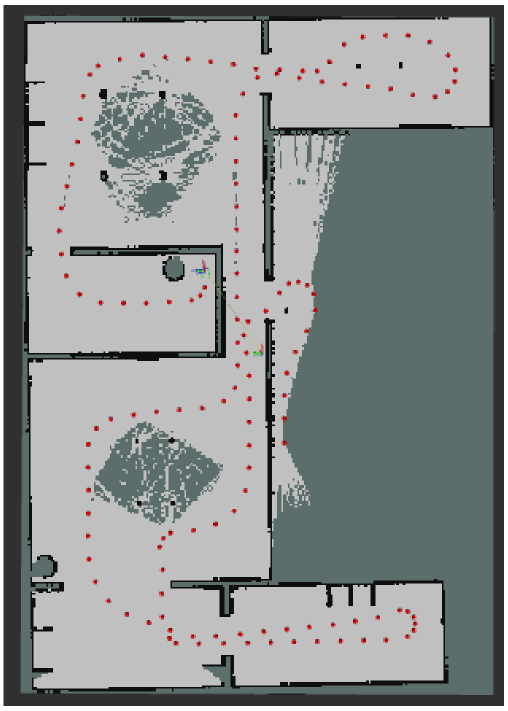


Figure 5.3: Mapping with SLAM Toolbox.

It can be seen that the quality of mapping is quite similar. The difference is that Cartographer assigns an occupation probability to cells, while SLAM Toolbox only distinguishes between free and occupied cells. This is why the map on the left contains different shades of grey. For what concerns localization, Cartographer is designed to work for real-time tasks, hence it draws a continuous trajectory on the map. Instead, SLAM Toolbox keeps a pose graph whose nodes represent the poses that the robot assumes at the time that a scan matching step succeeds. The reason is that `slam_toolbox` implements tools that save the pose graph for manipulation in post-processing, offline localization and even resuming the mapping task in a second moment. In both algorithms, the mobile robot is successfully tracked and no drift occurs in the odometry.

However, the most critical point is the scan matching. Looking at the Cartographer map, there is a small misalignment on the left: this is caused by the Ceres solver accepting a scan that is not compatible enough with the prior pose, and this happened even though the weights of the Ceres solver have been increased in order to discard such scans. This is a structural problem of Cartographer, since the scan matching problem only considers the most recent scans: if the solver is not capable of detecting these outliers, the resulting map would present a distortion. A loop closure slightly corrects this error, but the submap is still distorted due to the scan matching procedure. For what concerns SLAM Toolbox, the scan matching is a bit slow and takes a while to process new scans, but the map is built correctly.

In the end, although Cartographer and SLAM Toolbox have their pros and cons, the most relevant factor is the correctness of the generated map. As a consequence, SLAM Toolbox was chosen as front-end algorithm. Nevertheless, as it will be clarified later, the back-end module is substantially independent of the specific front-end algorithm if this can provide a grid map.

5.2 Single-Robot Front-End: SLAM Toolbox

The `slam_toolbox` package contains an evolution of KartoSLAM. As said before, this is quite different from the original formulation [13]. Unlike other SLAM algorithms such as Cartographer, KartoSLAM has undergone a series of improvements during time. Originally, the pose graph optimization was formulated as a sparse pose adjustment problem (SPA), similar to the one that was shown in (2.7). Due to the SPA complexity, another formulation of KartoSLAM replaced SPA with a sparse bundle adjustment problem (SBA) (2.8). In order to grant more flexibility and faster optimization, the latest implementation of KartoSLAM integrates a Ceres solver to find the solution of a non-linear least-squares problem on the pose graph. Since the Cholesky matrix decomposition is a slow method, the scan matcher has been improved as well. The `slam_toolbox` package refers to such a version of KartoSLAM.

Before viewing some interesting details of the implementation, there is a concept that holds for all the SLAM algorithms that are implemented with ROS. When simulated and real mobile robots are launched, a transformation tree that is similar to Fig. 4.6 is generated. The odometry system is initialized, but the robot is not correctly localized yet, because the odometry could still drift. Hence, it is good practice that SLAM algorithms publish an additional transformation `map`$\rightarrow$`odom`;

this also applies to other localization system such as the Nav2's AMCL tool. This introduces a new reference frame: `map`. It should be a fixed coordinate system and its position can be arbitrary. Furthermore, `map` can act as reference coordinates for tracking and map building.

SLAM Toolbox sets this transformation to identity, i.e., the `map` and `odom` frames coincide, and publishes it continuously. In general, this is not recommended because it does not solve the drift problem. Fortunately, SLAM Toolbox provides a back-end optimization with loop closure, hence if the odometry eventually suffers from drift the measurements are corrected. For a more reliable solution, the `map` frame should be located in a fixed position, for instance in the initial pose of the mobile robot.

The code is completely open-source. Here, the most salient components of SLAM Toolbox are reported to explain the main workflow of SLAM Toolbox.

- The scan matcher works in a restricted area of the map called correlation grid. A C++ object called `ScanMatcher` defines methods for grid initialization, matching, correlation and validation. It keeps a buffer with all the previous scans that marked cells in the grid as occupied. The scan matcher acts in the robot's neighborhood for local matching: it centers the grid in the robot's pose and tries to match the current scan with others within the grid. After finding a match, the scan matcher computes the relative pose between the scans. Furthermore, it looks for eventual loop closures by using a bigger correlation grid.

- The pose graph is represented by a `MapperGraph` object. The scan poses are its vertices and chains of consecutive scans are connected by edges. When the graph generates a loop, a loop closure is achieved.

- Originally, the scan solver was based on Cholesky decomposition matrix. In SLAM Toolbox, the `ScanSolver` object only provides some basic operations such as adding and removing nodes and constraints. The `CeresSolver` object extends these functionalities through C++ inheritance.

- A `Mapper` object wraps the scan matcher, the pose graph and the Ceres solver. When a scan is received, the mapper processes it within the correlation grid for local scan matching. If the scan is validated, the corresponding pose is added to the graph and the mapper invokes the scan matcher again to find loop closures in a larger correlation grid. The Ceres solver aims at correcting the poses to optimize the graph.

There are other functionalities that SLAM Toolbox offers such as map serialization and deserialization, optimizer customization and pose graph manipulation. A last mention goes to the compatibility with Nav2 tools: thus, the user can command the mobile robot by sending goals on the map displayed by the RViz2 GUI.

5.2.1 Ceres Solver

A C++ library defines the property of a Ceres solver [53] and the public documentation is available at [54]. In general, Ceres solver aims at solving nonlinear least-squares optimization problems. Given a vector $\boldsymbol{x} \in \Re^n$, a function $\boldsymbol{F} : \Re^n \to \Re^m$ and lower and upper bounds $\boldsymbol{L}$ and $\boldsymbol{U}$ for $\boldsymbol{x}$, a nonlinear least-squares problem is usually set as (5.1) shows.

$$\arg\min_{\boldsymbol{x}} \frac{1}{2}||\boldsymbol{F}(\boldsymbol{x})||^2 \quad s.t.\ \boldsymbol{L} \leq \boldsymbol{x} \leq \boldsymbol{U} \tag{5.1}$$

Such a problem is not feasible. Hence, the strategy is to find a local minimum by applying a small correction $\Delta\boldsymbol{x}$ to $\boldsymbol{x}$. By defining the Jacobian matrix $\boldsymbol{J}(\boldsymbol{x}) \in \Re^{m,n}$ of $\boldsymbol{F}(\boldsymbol{x})$, it is possible to approximate $\boldsymbol{F}(\boldsymbol{x}) \approx \boldsymbol{F}(\boldsymbol{x}+\Delta\boldsymbol{x})+\boldsymbol{J}(\boldsymbol{x})\Delta\boldsymbol{x}$ by linearization. Thus, the optimization problem in (5.1) becomes linear as (5.2) shows. Here, $\Delta\boldsymbol{x}$ defines a step size to look for a local minimum.

$$\arg\min_{\Delta\boldsymbol{x}} \frac{1}{2}||\boldsymbol{F}(\boldsymbol{x}) + \boldsymbol{J}(\boldsymbol{x})\Delta\boldsymbol{x}||^2 \quad s.t.\ \boldsymbol{L} \leq \boldsymbol{x} \leq \boldsymbol{U} \tag{5.2}$$

However, it is not guaranteed that the algorithm will converge. Hence, a trust strategy is involved: a trust region radius μ is defined around an initial point $\boldsymbol{x}$. This way, each step applies a $\Delta\boldsymbol{x}$ that is bounded within the trust region. Depending on the quality of the decrease of the objective function, μ can be increased (if good) or decreased (if bad) after a single step. SLAM Toolbox's Ceres solver adopts a trust strategy that relies on the Levenberg-Marquardt algorithm: it solves an unconstrained optimization problem that augments (5.2) by adding a square term that penalizes too large steps. The Levenberg-Marquardt algorithm aims at solving the linear least-squares problem in (5.3): here, μ is the trust region radius and $\boldsymbol{D}(\boldsymbol{x}) \in \Re^{n,n}$ is the diagonal matrix of $\boldsymbol{J}(\boldsymbol{x})^T\boldsymbol{J}(\boldsymbol{x})$. Ceres solver may adopt any linear solver to find the local minimum of this optimization problem.

$$\arg\min_{\Delta\boldsymbol{x}} \frac{1}{2}||\boldsymbol{F}(\boldsymbol{x}) + \boldsymbol{J}(\boldsymbol{x})\Delta\boldsymbol{x}||^2 + \frac{1}{\mu}||\boldsymbol{D}(\boldsymbol{x})\Delta\boldsymbol{x}||^2 \tag{5.3}$$

In C++, the original nonlinear least-squares problem is formulated as a generalization of (5.1) and is shown in (5.4). Here, ρ is a generic loss function, $\boldsymbol{f}_i$ is also known as a cost function and the term $\rho(||\boldsymbol{f}_i(\boldsymbol{x})||^2)$ is called residual block and includes both the cost and loss functions. If ρ is an identity function and there is only one cost function, (5.1) is obtained.

$$\arg\min_{\boldsymbol{x}} \frac{1}{2}\sum_i \rho(||\boldsymbol{f}_i(\boldsymbol{x})||^2) \quad s.t.\ \boldsymbol{L} \leq \boldsymbol{x} \leq \boldsymbol{U} \tag{5.4}$$

Adding a loss function may help to detect outliers. In fact, loss functions tend to reduce the influence of residual blocks that lead to high values on the objective function. For Ceres solver, Huber loss is usually employed for outlier detection: it is differentiable, convex and can efficiently handle small and huge errors. Given a threshold $\delta > 0$ and the value y of a residual, Huber loss combines a linear and a quadratic function, and the threshold discriminates what part of the function should be applied to the residual.

$$L_\delta(y) = \begin{cases} \frac{1}{2}y^2 & |y| < \delta \\ \delta(|y| - \frac{1}{2}\delta) & |y| \geq \delta \end{cases} \tag{5.5}$$

The optimization problem, the cost function and the loss function are respectively defined by the following C++ objects: `Problem`, `CostFunction`, `LossFunction`.

SLAM Toolbox models the optimization problem as follows.

1. After a correlation between two scans is found (i.e., an edge on the pose graph is generated), the relative position between them is computed.
2. Ceres solver creates a cost function: in SLAM Toolbox, this cost function represents the displacement of the relative pose between the scans. Then, it compacts the cost function and the loss function into a residual block that is added to the problem. The choice of the loss function is one of the parameters of SLAM Toolbox.
3. Ceres solver also generates a constraint that represents the correlation between the matched scans. The constraint is then added to the problem.

5.2.2 SLAM Toolbox Topics, Parameters and Namespace

SLAM Toolbox needs to receive data from LiDAR scans: hence, it subscribes to `/scan` (i.e., the scan topic of TurtleBot3 Burger). SLAM Toolbox does not rely on additional data from the odometry: the only requirement is a valid `odom`→`base_footprint` transformation that is used to create nodes on the map for the pose graph.

Given these prerequisites, SLAM Toolbox creates some topics and publishes data on them. The most relevant ones are reported here.

- `/map` and `/map_metadata` contain the information about the grid map. More precisely, `/map` includes the detailed grid with occupied and unoccupied cells, while `/map_metadata` stores some generalities such as the size, the resolution and the origin.

- `/slam_toolbox/graph_visualization` saves the nodes of the pose graph. These nodes can be also displayed on RViz2, thus giving a visual idea of the robot's trajectory.

The parameters that SLAM Toolbox uses are stored in a textual `.yaml` file. Solver options and parameters for local scan matching and loop closures can be properly tuned. For example, it is possible to extend the correlation grid to a wider area and/or modify the grid resolution. As a preparation for the centralized multi-agent SLAM framework, it is recommended to increase the size of the correlation grid that is used for local scan matching.

Table 5.1 shows a list of the modified parameters with respect to the default settings. The new set of parameters aims at being compatible with LDS-02, reducing the chance of spurious scan matching and increasing the area for local matching in order to build a larger grid since the first steps. It is not recommended to enlarge the local correlation grid too much, otherwise, it may lead to slowness during the scan matching.

Parameter	**Default value**	**New value**
`ceres_loss_function`	None	HuberLoss
`resolution`	0.05 m	0.04 m
`minimum_travel_distance`	0.5 m	0.4 m
`link_scan_maximum_distance`	1.5 m	1.2 m
`correlation_search_space_dimension`	0.5 m	0.7 m
`distance_variance_penalty`	0.5 m	0.2 m

Table 5.1: Modified parameters for SLAM Toolbox.

Adding a namespace to support more robots running SLAM Toolbox is quite trivial. As usual, SLAM Toolbox topics can be remapped in the launch file. Since in this case remapping applies also to the transform topics, reference frames do not need any change. It is also important to add a namespace to the topics and the header in the `.yaml` file.

5.3 Multi-Robot Back-End: Map Merging Node

Now, the single-robot front-end component is correctly set up. Loop closure detection is already included, hence there is no need to develop this functionality.

To complete the centralized multi-agent SLAM framework, a node that merges the maps of the involved mobile robots is required. Since this is a centralized approach, direct communication between the mobile robots is not needed, but each agent must be able to communicate with the central node. Another important characteristic is the initial guess on the pose of the mobile robots: the map merging node should know, exactly or approximately, the initial position of all the agents. In this scenario, a failure of the central node leads to the fault of the whole task. Typically, the central node should not receive too many data, otherwise, processing could be difficult and such complications may lead to a failure.

Based on these characteristics, a valid option that can play the role of the centralized node for map merging is offered by the `multirobot_map_merge` ROS package. However, this package was originally designed for ROS1 and nowadays only an unofficial version exists for ROS2 [55]. Furthermore, the ROS2 version is compatible with GMapping and Cartographer, but support for SLAM Toolbox is still under development, hence it requires some changes in the code. Nonetheless, `multirobot_map_merge` represents a naive way to implement a centralized multi-agent SLAM setting.

Actually, `multirobot_map_merge` meets all the traits that a central node for multi-agent SLAM should have. First of all, it is independent of the number of mobile robots. Furthermore, if the initial poses are not exactly known, it is sufficient that the agents start close to others. The only information that the central node needs to compute the global map is the grid map that can be easily encoded in a simple ROS message. On top of that, `multirobot_map_merge` can potentially work to support any single-robot front-end approach that builds a grid map.

The central node is called `MapMerge` and is a ROS node object. While active, it runs the following tasks periodically.

- It accomplishes robot discovery: simply, it looks for `/*/map`[1] topics. Then, `MapMerge` subscribes to such topics, as well as the `/*/map_updates` topics, if available. SLAM Toolbox does not provide map updates, but Cartographer defines such a topic.

- If the initial poses are unknown, the central node operates a pose estimation. It feeds a `MergingPipeline` object with the grids that the mobile robots send through their `/*/map` topic. Then, the pipeline estimates the transformation between the map frames through a feature matching technique.

- `MapMerge` commands the pipeline that executes the map merging, based on the transformations that were previously estimated by the pipeline, or according to the initial position of the mobile robots (if known).

[1]The signature `/*/map` refers to any map topic with a namespace. Example: `/a/map`.

This procedure does not require any data from odometry. Simply, the merging node receives the grid maps as `nav_msgs::msg::OccupancyGrid` messages. More precisely, a message of this type includes a header with ROS information (timestamp and reference frame), map metadata (resolution, width, height and origin) and a vector with a value for each cell. Possible values are: 0 if unoccupied, 1 if definitely occupied, and -1 if unknown.

The resulting centralized multi-agent approach follows the workflow that was described in Fig. 3.1. In fact, an inter-robot loop closure is achieved when a robot sees the feature that another robot has already visited. Furthermore, it will be explained later how the pipeline performs some optimization on the relative pose.

5.3.1 Feature Matching for Map Merging

The map merging node does not operate on the scans taken by the agents. Instead, it only requires the grid maps that the agents send through the `/*/map` topics.

The OpenCV C++ library [56] offers open-source tools for computer vision and machine learning, including image processing techniques. OpenCV also includes different feature matching options and methods to estimate the pose of a camera. These elements are crucial to perform map merging, because the feature matching aims at finding common keypoints on the maps and the camera estimation allows to optimize the map merging. Actually, this works for LiDAR sensors as well.

The `multirobot_map_merge` supports three different feature matching techniques. All of them require a confidence threshold that has to be defined properly.

- AKAZE [57] builds a nonlinear scale space in order to analyze the image at different scales while maintaining its structure. Then, AKAZE detects keypoints at different sizes and assigns an orientation and a descriptor to each. AKAZE descriptor is robust against rotation and scaling. Given two images, AKAZE exploits the information on keypoint orientation to find matches between features. Thanks to the feature orientation computation, AKAZE is suitable for image stitching, i.e., the process of combining multiple images through a partial overlap.

- SURF [58] detects features by following an approach based on the Hessian matrix on a low-dimensional feature space, hence it is fast and well suited for real-time application. For each feature, SURF calculates a vector that defines the orientation and the light intensity. A fast matching simply compares feature vectors: Euclidean distance is a valid metric for feature matching. SURF is invariant to rotation, and can become insensitive to changes in lighting as well if the feature vectors are normalized. The accuracy of the feature matching can be further improved by outlier rejection.

- ORB [59] is mainly based on corner detection: in computer vision, a corner is typically represented by an abrupt change in intensity on the image. ORB computes the orientation to each corner as well, according to the intensity pattern around the keypoint, and labels that keypoint with a descriptor that is unaffected by changes in illumination. ORB matches keypoints between distinct images by comparing descriptors according to a metric, typically Hamming distance. Outlier integration can be integrated for better results.

In this context, some characteristics are fundamental to choose a good feature matching algorithm: resilience against rotation, computational efficiency and sometimes also scale invariance. Indeed, the map frames may be rotated with respect to each other, and the feature matching should not struggle for a high number of features. A comparison between the three techniques [60] shows that ORB generally outperforms AKAZE and SURF. Hence, ORB will be used as feature matching algorithm.

For what concerns map merging, a suitable confidence threshold has to be defined properly. Its value should be neither too high nor too low. Indeed, if the threshold is too high then the feature matching may not find the initial pose, unless there are lots of features since the first scans; this matters for SLAM Toolbox, and even more in simple environments. On the contrary, if the threshold is too low then erroneous transformations are applied and the global map will be distorted.

The feature matching also acts as a module for inter-robot loop closure. At a certain point, a mobile robot may visit some features, for example, a room, that other agents may have already seen. If this happens, the feature matching between the maps finds a correspondence and corrects the transformation between the grid maps. Inter-robot loop closure ensures that the global map is computed correctly in the end, even though some critical failures happened in feature matching during the task.

5.3.2 Merging Pipeline

The `MergingPipeline` object defines some core functionalities that better explain the workflow of `multirobot_map_merge`. It is an object that stores the grid maps, the images and the transformations. When `MergingPipeline` receives a `nav_msgs::msg::OccupancyGrid`, it converts the grid map into an image.

`MergingPipeline` is commanded by the `MapMerge` node to estimate transformation and merging the grids. Hence, two important functions can be distinguished.

- The `estimateTransforms()` function receives the type of feature matching technique and its confidence threshold. First of all, it invokes feature matching and filters the matches that do not have enough confidence. Then, OpenCV offers tools that estimate the transformation. OpenCV also supports back-end optimization of the transformations: it minimizes the sum of the reprojection error squares, similarly to bundle adjustment in (2.8).

- The `composeGrids()` function merges the maps. Once the transformations are successfully computed, it warps the images and finds proper regions of interest to overlap the images, consistently with the computed transformations.

The transformation estimation makes sense only if the initial poses are not exactly known; otherwise, the information about the relative position between robots can be directly used for map merging.

In the case of unknown poses, it is strictly required that the mobile robots are initially close enough, such that the feature matching algorithm finds common characteristics and the transformations can be computed from the beginning. After that, mobile robots can move in autonomy and a rendezvous is not strictly required to close the loop.

If the mobile agents start from different points in the environment, the feature matching algorithm struggles to find common features, and also may lead to false matches. As a consequence, the relative pose is not computed, or even worse, `estimateTransforms()` returns a bad transformation, and a wrong and distorted global map is generated.

5.3.3 Support for SLAM Toolbox

Unfortunately, `multirobot_map_merge` lacks official support for SLAM Toolbox and the basic code does not work. There are several reasons why SLAM Toolbox is problematic for this map merging solution. First of all, SLAM Toolbox does not return a grid map with probability values, but instead, it marks cells only as occupied or unoccupied. As a consequence, unless the local correlation grid size is increased drastically, thus slowing scan matching too much, the initial grid maps are relatively small, and it might be difficult to compute the initial pose of the mobile robots. Hence, common features between maps are initially hard to be detected. The second issue concerns the size of such grid maps. Indeed, SLAM Toolbox dynamically increases the length and the width of the map while the mobile robot explores the environment. This means that the feature matching algorithm may deal with images of different scales, hence it will likely fail.

Nevertheless, these two problems can be solved without degrading the performance of SLAM Toolbox. It is possible to get around the first problem by starting from a region in the environment where the robots can clearly observe a common feature such as a corner or an obstacle. Thus, the feature matching algorithm computes initial transformations that can be updated at later times by the pipeline. If the initial grid maps are too small, it is recommended to move the mobile robots a bit before starting to explore separate parts of the environment. For what concerns the second issue, a piece of code is added: before passing the maps to the pipeline, each grid is resized to an arbitrary dimension. Simply, the metadata of the `nav_msgs::msg::OccupancyGrid` message is edited to fit the new height and width, and many values equal to -1 (i.e., cells with unknown status) are pushed into the vector. This rescaling of the grid maps is also called padding, due to the fact that many unknown cells are pushed into the maps.

5.3.4 An Example in a Simulated Scenario

Here is a quick demonstration of some of the most important functionalities of this centralized multi-agent SLAM approach. TurtleBot3 House (Fig. 5.1) is used again as a proving ground. The initial positions of Alice[2] and Bob[3] are not exactly known, but both agents start close to each other. The initial scenario is marked in Fig. 5.4. The confidence threshold is set to 0.6: this value was chosen by trial-and-error procedure, starting from simple scans.

Alice is sent to circumnavigate the table, while Bob is sent towards the "upper" part of the house (i.e., towards the big room with the other table). Both Alice and Bob see the garbage bin with their first scan, hence a first transformation between the grid maps is computed. After Alice and Bob move a bit, the global map in Fig. 5.5 is obtained. Fig. 5.6 highlights the approximate path that Alice and Bob took.

The global map is still distorted. The reason behind this is that the robots follow the same path. In particular, Bob sees Alice as an obstacle and vice versa. This is why there are black dots on the map on the left. Thus, the feature matching associates this phenomenon to a displacement that creates a distortion.

Then, Alice and Bob aim at exploring two distinct areas of the house. Alice refines the current room and explores the "lower" part, while Bob crosses the corridor and explores the "upper" part. When Alice and Bob have finished, their

[2]Alice is the name that is given to the first TurtleBot3 Burger, with namespace `a`.

[3]Bob is the name that is given to the second TurtleBot3 Burger, with namespace `b`.

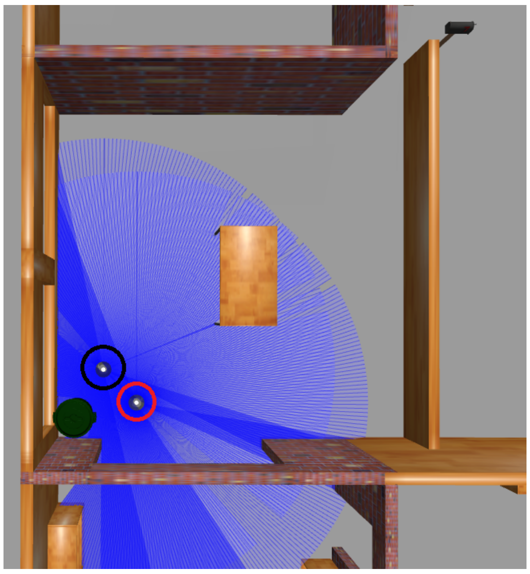


Figure 5.4: Initial positions of Alice (red) and Bob (black).

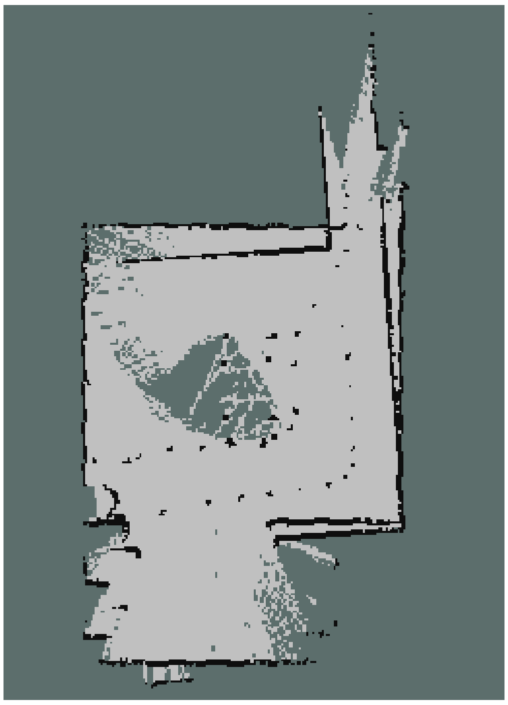


Figure 5.5: An intermediate merged global map that is computed after a while.

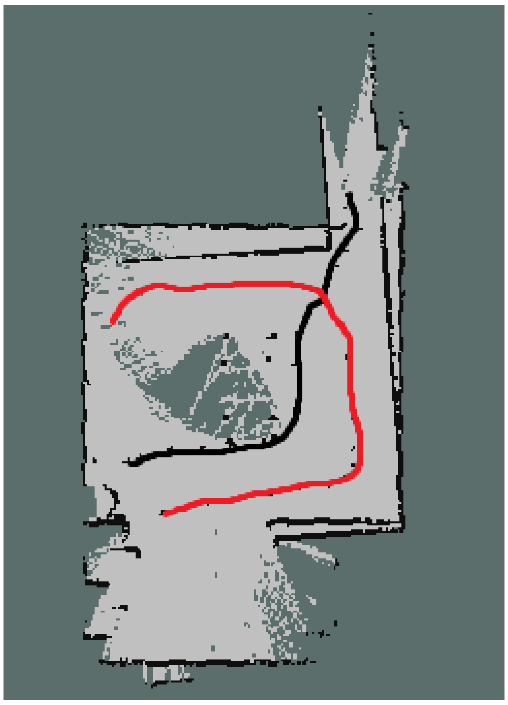


Figure 5.6: The same map, but the approximate paths of Alice (red) and Bob (black) are drawn.

local maps appear as follows, respectively in Fig. 5.7 and Fig. 5.8.

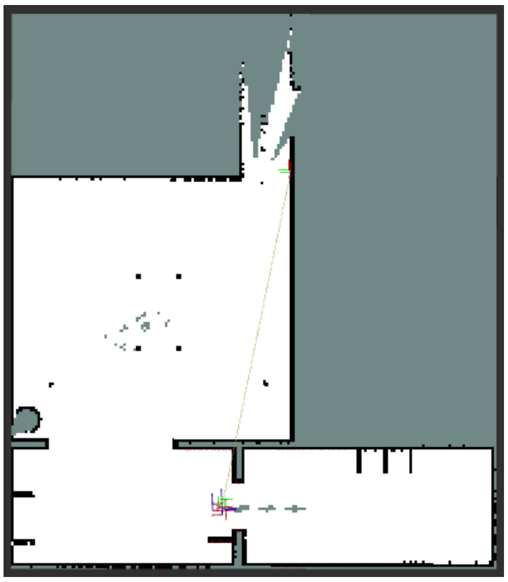


Figure 5.7: Alice's local map.

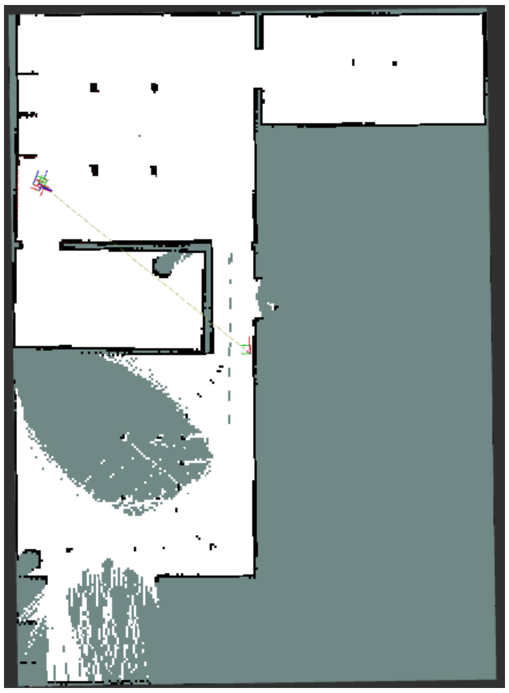


Figure 5.8: Bob's local map.

During this time, the back-end optimization from the pipeline adjusts the relative pose between maps. However, an inter-robot loop closure did not happen yet, because after Alice and Bob took separate paths none of them was sent back towards the remaining area to explore, according to the local map. Fig. 5.9 shows the status of the global map after the map merging of the two previous local maps.

As a final step, since Alice took less time than Bob to map the given part of the house, Alice was sent towards the "upper" part of the house. By the time Alice captures the same features that Bob perceived features (the shape of the room, the table and the shelves), the feature matching finds correspondences with high confidence, thus leading to a correct transformation between the maps. The resulting map is reported in Fig. 5.10.

5.3.5 Real-World Scenario for Centralized Multi-Agent SLAM

The proposed centralized multi-agent SLAM has also been tested in a real-world environment. Two physical copies of TurtleBot3 Burger have been placed in the laboratory of Department of Electronics and Telecommunications (DET) at the third floor in Politecnico di Torino. The planimetry is not available, but instead

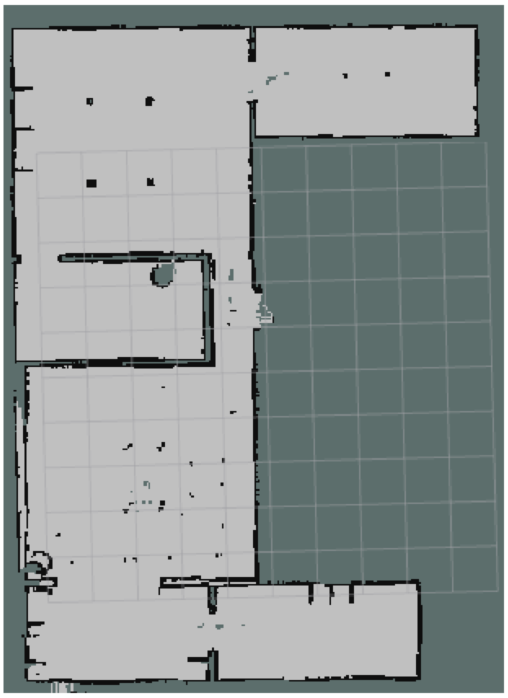


Figure 5.9: Global map without the inter-robot loop closure.

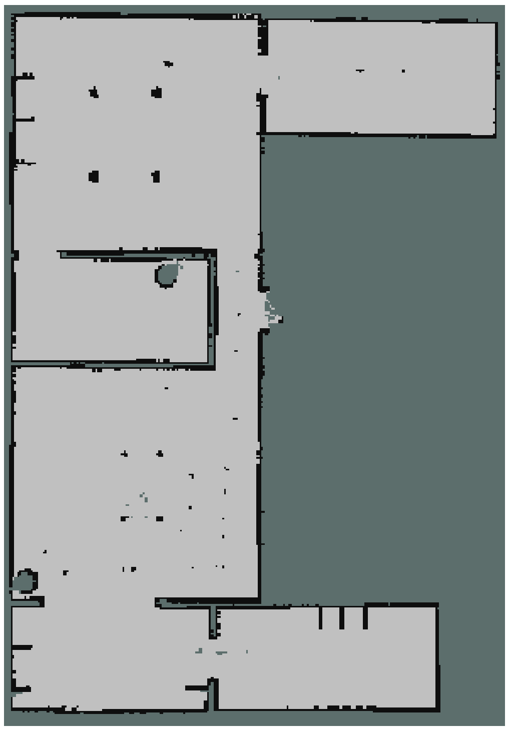


Figure 5.10: Global map after the inter-robot loop closure.

a ground truth has been recovered by running SLAM Toolbox as a single-robot SLAM algorithm in the laboratory and the corridor at the exit of the room. The ground truth is shown in Fig. 5.11: only one part of the corridor has been explored, otherwise the environment would be too big and localization may struggle. The area where the two mobile robots start for the multi-agent SLAM testing is marked by the blue circle.

The parameters of SLAM Toolbox are equal for the two mobile robots: the same modifications in Table 5.1 have been applied. For what concerns the parameters of `multirobot_map_merge`, the initial confidence threshold for feature matching is set to a low value (0.2). Each time the feature matching returns a successful transformation, the confidence threshold increases by 0.05, up to a maximum of 0.7: this way, the central node can perform an initial map merging (even if bad), and tries to improve its previous transformation estimate during time. The maximum value 0.7 is chosen empirically as a good threshold that ensures the correctness of the transformation.

Here, the evolution of the map merging procedure is described. The two mobile

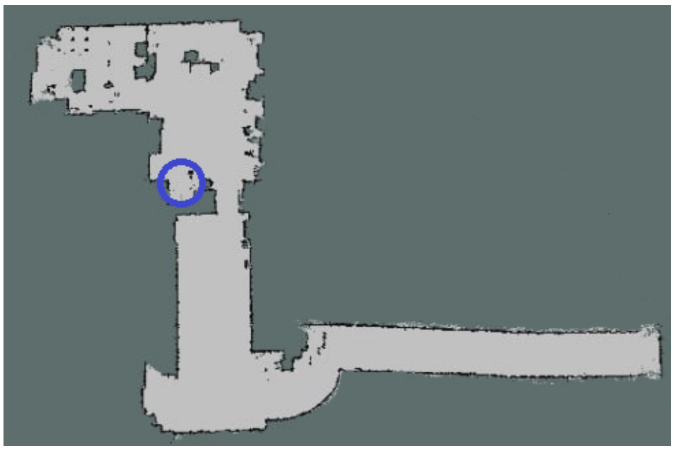


Figure 5.11: Ground truth of the laboratory of DET and the corridor. The initial position for the multi-robot experiment is marked.

robots start near to each other in the area that Fig. 5.11 shows. Alice is sent to explore the laboratory, while Bob is sent towards the exit of the laboratory to explore the corridor. After a short time after the departure, Alice and Bob enrich their local maps and the map merging node already finds a transformation, due to the low initial confidence threshold. Fig. 5.12 captures this situation: the top left local map is built by Alice, Bob's map is the lower left one, while the big map on the right is the global merged one.

This initial transformation is very good. However, due to the low threshold, the feature matching may find another transformation with enough confidence that does not represent the global map at all. For example, during the experiment, after Alice and Bob have explored part of the unknown environment, the bad merging in Fig. 5.13 is obtained. As the trajectories are difficult to be recognized due to the quality of the images, the approximate paths have been marked.

After some spurious map mergings, the threshold grows and reaches its maximum value. When the confidence threshold approaches 0.7, the feature matching estimates the transformation that is shown in Fig. 5.14. This is not a clean transformation, but the feature matching has understood the connection between the maps.

However, due to the limited localization capability of SLAM Toolbox, Bob struggles to localize itself in the corridor. Hence, Bob is stopped. This issue is related to SLAM Toolbox, but it does not affect the performance of the map merging. Alice can proceed to explore the rest of the laboratory. After Alice has finished, the central node computes the global map as it is shown in Fig. 5.15.

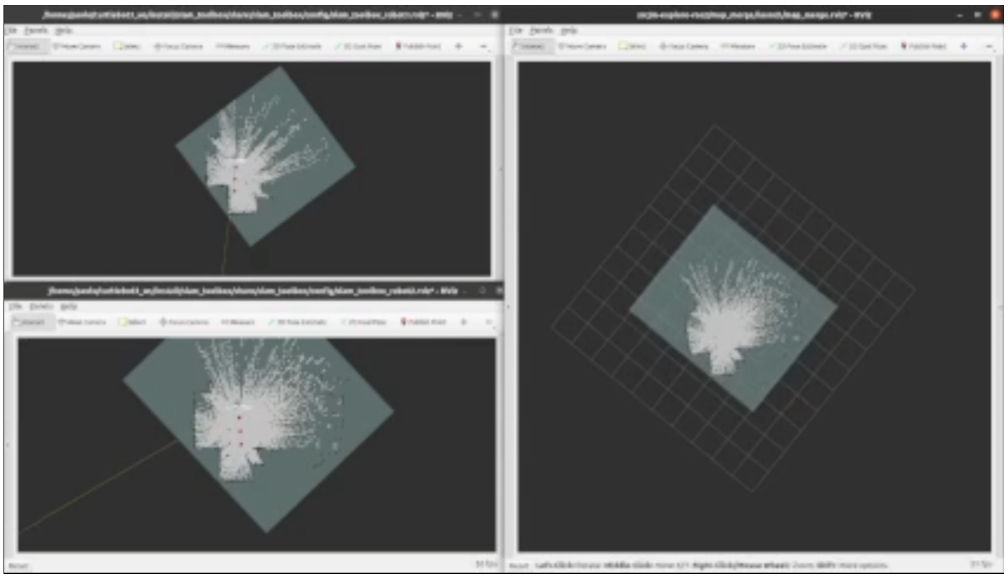


Figure 5.12: First map merging in the real-world multi-agent experiment.

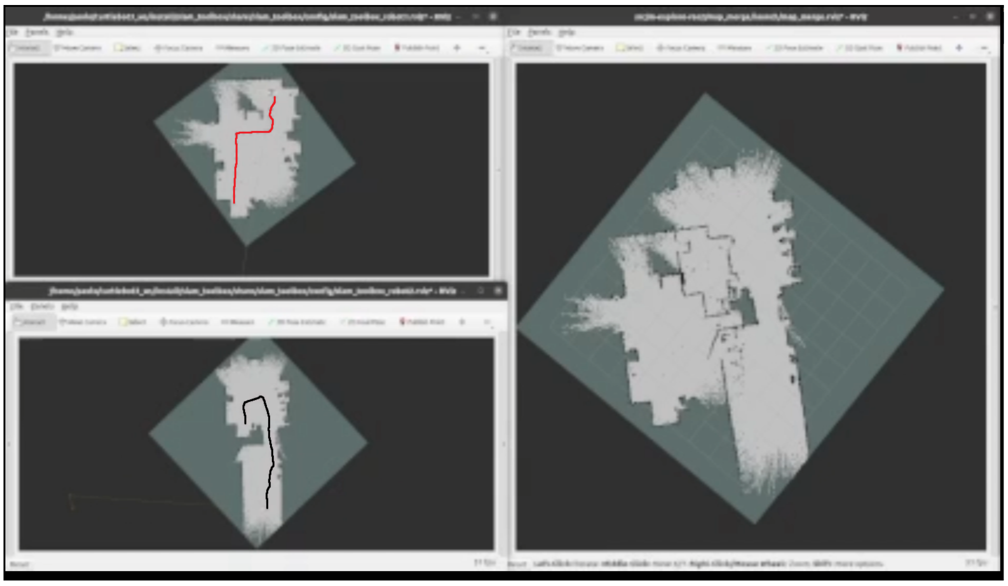


Figure 5.13: A bad map merging, caused by a low value of the confidence threshold.

In order to establish inter-robot loop closure, Alice is sent towards the exit of the laboratory. Thus, the global map in Fig. 5.16 is obtained: the quality of the merging has improved a lot, and the global map looks similar to the ground truth in Fig. 5.11.

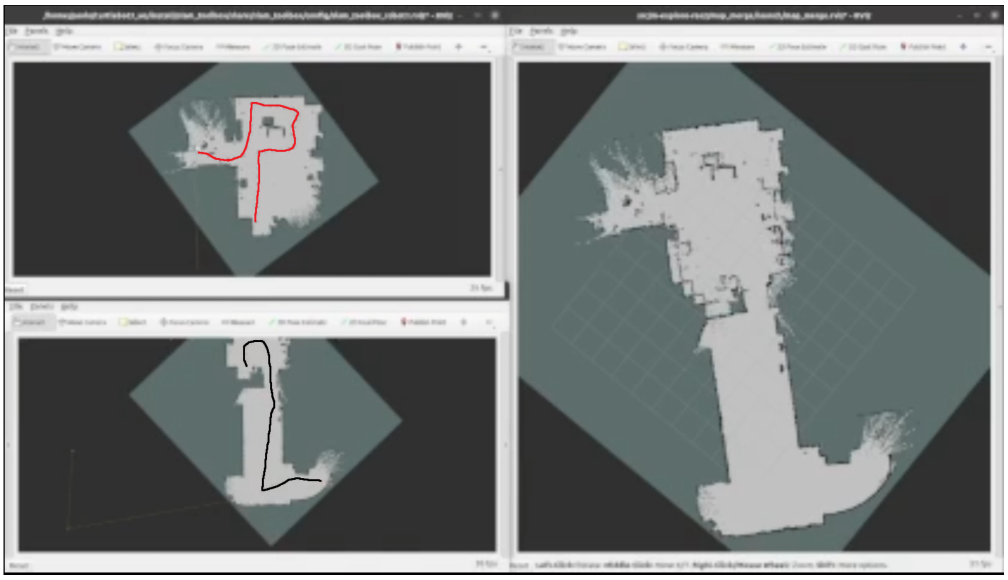


Figure 5.14: A good map merging in the real-world multi-agent experiment.

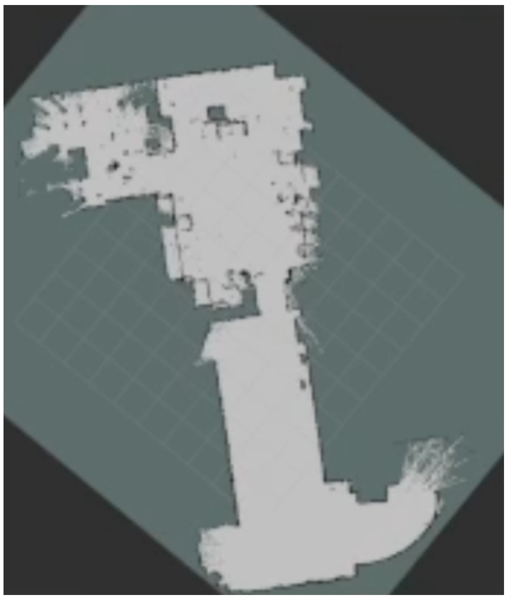


Figure 5.15: Global map without the inter-robot loop closure.

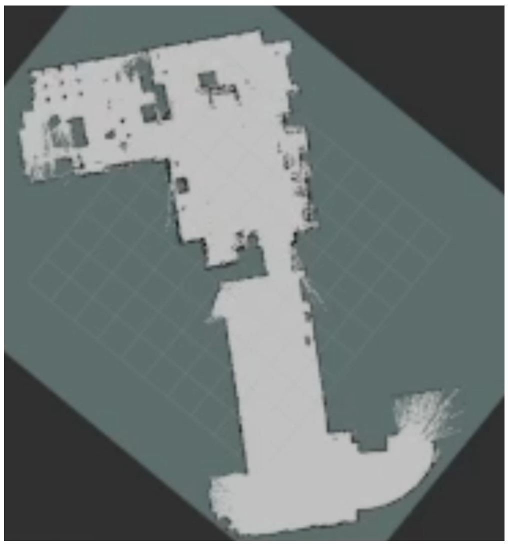


Figure 5.16: Global map after the inter-robot loop closure.

Chapter 6

Distributed Implementation of Multi-Agent SLAM

A distributed multi-agent SLAM approach exploits cooperation between mobile robots. In a centralized setting, the swarm collaborates on the same task, but the agents are not required to establish a communication channel. On the other hand, distributed approaches require that robots exchange their local information on the environment, such that the agents reach a consensus and their local maps will eventually converge into the global one.

Differently from the previous approach, having a single central node is not enough. Instead, some form of communication between robots should be implemented. In practical terms, the distributed back-end module should define proper topics that concern local loop closures, estimates and eventually data descriptors: since these topics refer to the single robot, a proper namespace has to be added. The resulting communication is defined as follows.

1. The distributed back-end module allows a mobile robot to publish data to the topics that are under its namespace.
2. Then, each mobile robot subscribes to the topics that are related to all the other agents. This way, when another mobile robot computes or corrects its estimate, other agents are warned and execute a callback function.

For example, given the two robots named Alice and Bob, Alice publishes on its local topics and subscribes to Bob's topics, and vice versa. Other details about the inter-robot loop closure detection, global optimization and other functionalities are specific to the particular multi-agent SLAM algorithm.

This chapter presents the ROS implementation of DCL-SLAM [34], a distributed multi-agent SLAM algorithm that was discussed in the state-of-the-art review in

Subsection 3.2.2. First, LIO-SAM [23] is introduced as the chosen single-robot SLAM algorithm and its main features are described. Then, the distributed back-end module that DCL-SLAM provides is analyzed into detail.

The complete distributed approach is suitable for 3D LiDAR sensors and does not officially support TurtleBot3 Burger. An attempt to adapt the code has been made in order to make DCL-SLAM compatible with TurtleBot3 Burger. Unfortunately, concrete results in a real-world scenario are not available, due to some unresolved conflicts between libraries.

6.1 Single-Robot Front-End: LIO-SAM

LIO-SAM [23] stands for LiDAR inertial odometry via smoothing and mapping. It is a low-drift, graph-based 3D SLAM algorithm that completes the standard LOAM approach [20] by adding loop closure detection and integrating IMU measurements for optimization. Since IMU data are used in the optimization process, LIO-SAM can be categorized as a tightly-coupled system: such terminology refers to algorithms that improve accuracy by fusing IMU and odometry data to optimize point cloud registration and the pose graph. Actually, LIO-SAM does not only include the robot's poses in the graph: as it will be explained later, other factors contribute to build the graph as well.

Differently from approaches such as Cartographer, LIO-SAM does not build a grid map. Instead, it performs point cloud registration and builds a map as a set of point clouds. Algorithms that provide a 3D point cloud map perform more complex operations with respect to those that elaborate 2D laser scans, due to the three-dimensional nature of the problem. This leads to the necessity of extracting edge and planar features from the environment.

Furthermore, another characteristic that distinguishes LIO-SAM from other algorithms is scan matching: rather than matching old scans on a global scale, LIO-SAM only performs local scan matching, thus significantly improving real-time performance.

6.1.1 LIO-SAM Factor Graph

LIO-SAM defines a state variable $\boldsymbol{x}$ for a mobile robot. Given the world frame $\boldsymbol{W}$ and the body frame $\boldsymbol{B}$ of the robot, the state incorporates the rotation matrix $\boldsymbol{R}$ from $\boldsymbol{B}$ to $\boldsymbol{W}$, the relative position vector $\boldsymbol{p}$, the speed vector $\boldsymbol{v}$ and the IMU bias $\boldsymbol{b}$. Equation (6.1) wraps the state definition for LIO-SAM. Scans are taken in the body frame $\boldsymbol{B}$.

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{R}^T & \boldsymbol{p}^T & \boldsymbol{v}^T & \boldsymbol{b}^T \end{bmatrix}^T \tag{6.1}$$

In LIO-SAM, the concept of pose graph is extended to not only support LiDAR odometry to define the robot's pose, but also include other measurements and results. When a node is inserted into the graph, the robot's state is associated to it at that time. Graph edges contain a state estimate as well, and edges usually represent constraints between the nodes in the graph. LIO-SAM refers to these elements (i.e., nodes and edges) as factors. Hence, the pose graph is also called factor graph in LIO-SAM. In general, the factor graph is nonlinear. A factor is added to the graph when the change in the robot's pose exceeds a given threshold.

According to the different information that a factor may contain, there are four types of factors.

- The IMU preintegration factor pre-processes IMU data to infer the motion of the robot. It integrates the IMU measurements with respect to the time over small intervals. This way, the state can be estimated better, because the motion model is already predicted. In LIO-SAM, the measurements of angular velocity and linear acceleration are affected by IMU preintegration. Such factors model constraints in the factor graph.

- The LiDAR odometry factor captures LiDAR scans and extract planar and edge feature according to the point cloud local roughness: if high, an edge is detected, otherwise the feature is planar. Not all the LiDAR scans can be added in the graph. A LiDAR odometry factor is added to the graph as a node when the change in the robot's pose exceeds a given threshold: the related scans are called keyframes. In order to estimate the state, a sliding window approach selects the most recent keyframes that are transformed to the world frame $\boldsymbol{W}$. Then, scan matching comes into play: local scans in successive times are transformed to the world frame $\boldsymbol{W}$ and matched against the set of keyframes, and a match happens if a correspondence between planar or edge features is found. The computed relative transformation between the matched scan and the set of keyframes improves the state estimate.

- The GPS factor tends to reduce drift, because GPS gives absolute measurements. When data are received, they are transformed to the robot's frame $\boldsymbol{B}$. GPS adds further constraints to the factor graph.

- The loop closure factor gets into action where a state is added to the factor graph. The closest previous state according to the Euclidean distance is returned, as well as the scans associated to that node. Local scan matching is invoked again between such scans and the recent ones after transforming all of them to the world frame $\boldsymbol{W}$: if it is successful, a new constraint is added between the new inserted state and the closest one.

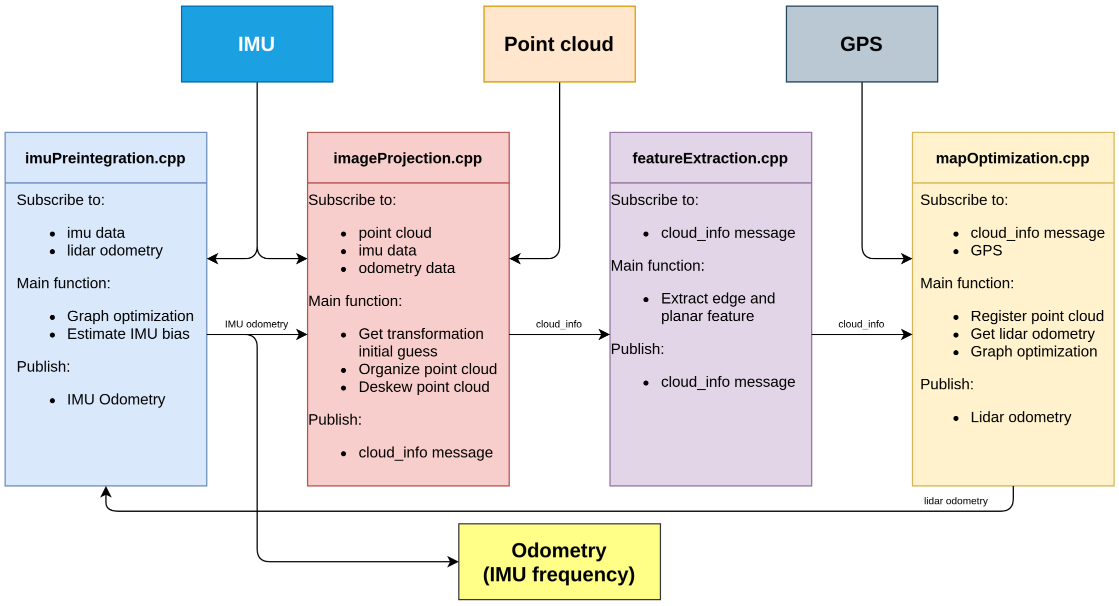


Figure 6.1: LIO-SAM architecture [61].

The state estimation problem is finally solved by optimizing the factor graph. In LIO-SAM, the iSAM2 algorithm [62] is adopted to estimate the state. The main data structures that are involved in the algorithm are the factor graph $\mathcal{F}$, the set Θ of state variables and a Bayes tree $\mathcal{T}$ that stores incremental measurements taken by the mobile robot. Shortly, a single step of iSAM2 is characterized by an update vector $\mathbf{\Delta}$ and is composed of the following operations.

1. New factors are added into $\mathcal{F}$.
2. New variables are inserted into Θ.
3. Each state variable is linearized according to the update vector $\mathbf{\Delta}$. Cliques of $\mathcal{T}$ that involved updated variables are marked: the set of these cliques is denoted as $\mathcal{M}$.
4. $\mathcal{T}$ is updated: all the cliques that are marked or affected by the new factors are recalculated, as well as their parents and so on.
5. The Bayes tree is solved: starting from the root and descending each time, a partial update $\mathbf{\Delta}_k$ is computed for each child of the current clique. The direction to follow while descending is determined by thresholding: the processing stops when the update does not exceed the threshold. This procedure returns the new update vector $\mathbf{\Delta}$ that will be used in the next iSAM2 step.
6. State variables in Θ are updated according to the current $\mathbf{\Delta}$.

6.1.2 LIO-SAM Code

LIO-SAM code is open-source [61]. The code was originally written with ROS1, but nowadays a version for ROS2 exists. Almost all the original characteristics of the algorithm are maintained in the code. Loop closure is enhanced by an adaptation of ICP [21] that is also used for loop closure detection in LeGO-LOAM [6].

LIO-SAM is composed of four main C++ modules. The system architecture and the main functions of these modules are summarized in Fig. 6.1.

- `imuPreintegration.cpp` instantiates two nodes. `IMUPreintegration` manages raw IMU data and applies preintegration as described before: this way, the motion of the robot is inferred. Furthermore, this ROS node is responsible of estimating IMU bias, creating IMU factors and adding them to the factor graph. `TransformFusion` handles the IMU data that have been integrated with the LiDAR odometry.

- `imageProjection.cpp` creates a `ImageProjection` node. It receives the point cloud and the IMU information (both raw and preintegrated). Raw IMU data is used for debugging: this is useful to check the correct setup for the IMU sensor. When a point cloud is received, `ImageProjection` applies a deskewing action according to the estimates of roll, pitch and yaw angles from raw IMU data and the initial guess on the transformation from the preintegrated IMU data combined with LiDAR odometry. The deskewed point cloud is passed to other nodes as a ROS message.

- `featureExtraction.cpp` defines a `FeatureExtraction` node that receives ROS messages from `ImageProjection` and enriches them with additional data about eventual planar or edge features that are detected. SURF [58] is used for feature extraction. Augmented ROS message are sent forward.

- `mapOptimization.cpp` contains the main operations that concern map building, trajectory, loop closure and optimization. The `MapOptimization` node receives the processed point clouds, along with their features. It stores point clouds and their poses into a k-dimensional tree, extracts their features and downsamples the visualized points before publishing point clouds on the global map. Loop closure is based on ICP that evaluates the Euclidean distances between point clouds. Optimization not only concerns the factor graph, but also the quality of feature detection is improved, according to the estimates related to the IMU. Speaking about factor graph optimization, `MapOptimization` defines LiDAR odometry, GPS and loop closure factors. The iSAM2 algorithm aims at solving the state estimation problem and correcting the robot's poses.

LIO-SAM does not use a standard `sensor_msgs::msg::PointCloud2` message to transport information between modules. Instead, it defines a more complex message `lio_sam::msg::CloudInfo` that incorporates a standard header, estimates from IMU, initial guess, the deskewed point cloud and its corner and surface features.

Processing point clouds includes operations that are not trivial. The support for such functions is offered by the Point Cloud Library (PCL) [63]. PCL is open-source and fully compatible with ROS and C++. It provides functionalities for processing 2D and 3D point clouds. These include filters, registration, visualization, feature matching, segmentation and lots of structures such as k-d trees. In particular, PCL offers an implementation for SURF feature matching and the ICP algorithm.

The definitions of factors, factor graph and the iSAM2 algorithm are included by GTSAM [64]. It is a C++ library that supports robotics and computer vision applications. GTSAM is based on factor graphs and Bayes trees for graph optimization, contrary to more classic approaches that rely on sparse matrices.

In the end, LIO-SAM should be able to build a three-dimensional point cloud map. However, there is no availability of 3D LiDAR sensors. Fortunately, a quick simulation with a 3D Velodyne LiDAR that is available at [65] has been set up, and the point cloud is shown in Fig. 6.2. Drivers for Velodyne are available at [66].

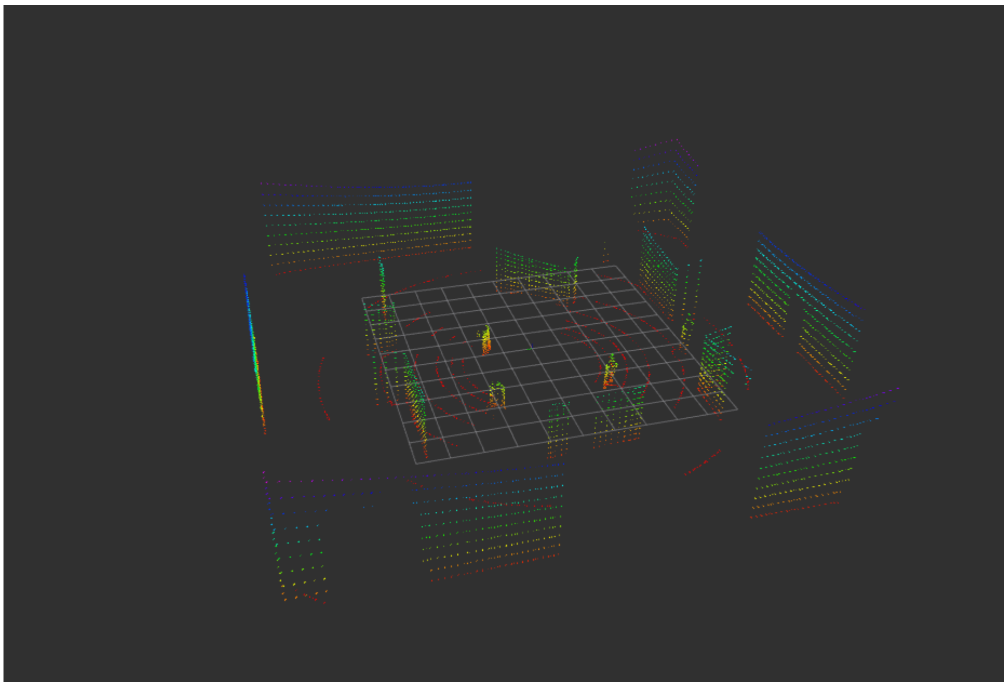


Figure 6.2: Point cloud computed by LIO-SAM with a 3D Velodyne LiDAR.

6.1.3 Adaptations for TurtleBot3 Burger

Actually, LIO-SAM code is not compatible with TurtleBot3 Burger without any modification. The reason is obvious: indeed, TurtleBot3 Burger does not publish data in a point cloud format, the configuration parameters are not suitable, GPS is not supported, and LDS-02 is a two-dimensional LiDAR. Nonetheless, it is possible to adapt the code to make it work for TurtleBot3 Burger. However, some functionalities have to be cut off.

First of all, a new ROS node called `ScanConverter` that converts laser scans into point clouds has been inserted into the original code. The `laser_geometry` ROS package [67] provides the `transformLaserScanToPointCloud()` function that executes the conversion. This function has to be invoked each time the LiDAR sensor publishes a `sensor_msgs::msg::LaserScan` message.

As a consequence, `ScanConverter` should subscribe to the `/scan` topic and invoke the conversion through a callback. Furthermore, `ScanConverter` should make available such point clouds, thus publishing on a `/cloud` topic.

Hence, new C++ file `scanConversion.cpp` is added to the LIO-SAM package. It defines the `ScanConverter` ROS node with such functionalities. The instantiation of `ScanConverter` should contain the following code that creates a subscription to the `/scan` topic and a publisher on the `/cloud` topic.

Listing 6.1: Creating a subscription and a publisher for `ScanConverter` node.

```
scan_sub =
    create_subscription<sensor_msgs::msg::LaserScan>(
    "scan", qos_lidar,
    std::bind(&ScanConverter::callback, this,
              std::placeholders::_1), scanConvOpt);
point_cloud_pub =
    create_publisher<sensor_msgs::msg::PointCloud2>("cloud", 100);
```

The definition of the callback follows: `transformLaserScanToPointCloud()` requires the scan reference frame (i.e., `base_scan` for TurtleBot3 Burger), a `sensor_msgs::msg::LaserScan` message, the `sensor_msgs::msg::PointCloud2` message (passed by reference) and the buffer that gives access to tf2 transformations.

Listing 6.2: Definition of the callback for the scan subscription.

```
void callback(sensor_msgs::msg::LaserScan::SharedPtr scan){
    sensor_msgs::msg::PointCloud2 cloud;
    projector.transformLaserScanToPointCloud(
        "base_scan", *scan, cloud, *tfBuffer
    );
    point_cloud_pub->publish(cloud);
}
```

The issue that concerns parameters can be solved by referring to IMU and LiDAR specifics that were reported in Tables 4.1 and 4.2. Topics and frames should be adapted to those that TurtleBot3 Burger defines. The transformation from the IMU frame to the LiDAR frame can be retrieved with tf2: it is static and does not change during time. The debug procedure for IMU data in the `imageProjection.cpp` file helps to check whether the parameters have been set correctly. Other parameters are left untouched.

A last mandatory modification concerns feature matching from the point cloud. As the LiDAR sensor is two-dimensional, it does not make sense to find edge or planar features from a 2D point cloud. Hence, the only processing that is required for the incoming point cloud is deskewing. Feature optimization has to be cut off as well. This heavily affects the structure of LIO-SAM, because the `FeatureExtraction` node does not play any role for 2D SLAM. Hence, `FeatureExtraction` is cut off, and the deskewed point cloud is directly passed to the `MapOptimization` node. Moreover, all the functions that are related to GPS have to be commented, because TurtleBot3 Burger does not receive any GPS data.

After applying these adaptations, LIO-SAM is able to work correctly with a TurtleBot3 Burger that is equipped with LDS-02. Fig. 6.3 shows the map that is built by registration of the deskewed point clouds. The path is correctly tracked as well. Once again, the simulation takes place in TurtleBot3 House.

6.2 Multi-Robot Back-End: DCL-SLAM

DCL-SLAM [34] represents a complete fully-distributed LiDAR 3D SLAM approach. It includes three main modules: single-robot front-end LiDAR odometry with intra-robot loop closure, distributed loop closure and back-end optimization. LIO-SAM plays the role of the first module, even though other LiDAR inertial odometry algorithms or LOAM-based approaches can be used. The back-end component of the multi-agent SLAM framework is described in the following pages and incorporates both inter-robot loop closures and global optimization. The architecture of DCL-SLAM is summarized in Fig. 6.4.

Inter-robot loop closure is based on recovering the relative pose between two mobile robots through place recognition. Such method for inter-robot loop closure aims at correcting the accumulated drift and finding correspondences between local maps. Place recognition is accomplished by sending LiDAR descriptors between robots. DCL-SLAM uses LiDAR Iris [35], but support for other descriptors is also provided. These LiDAR descriptors aim at finding good candidates for distributed

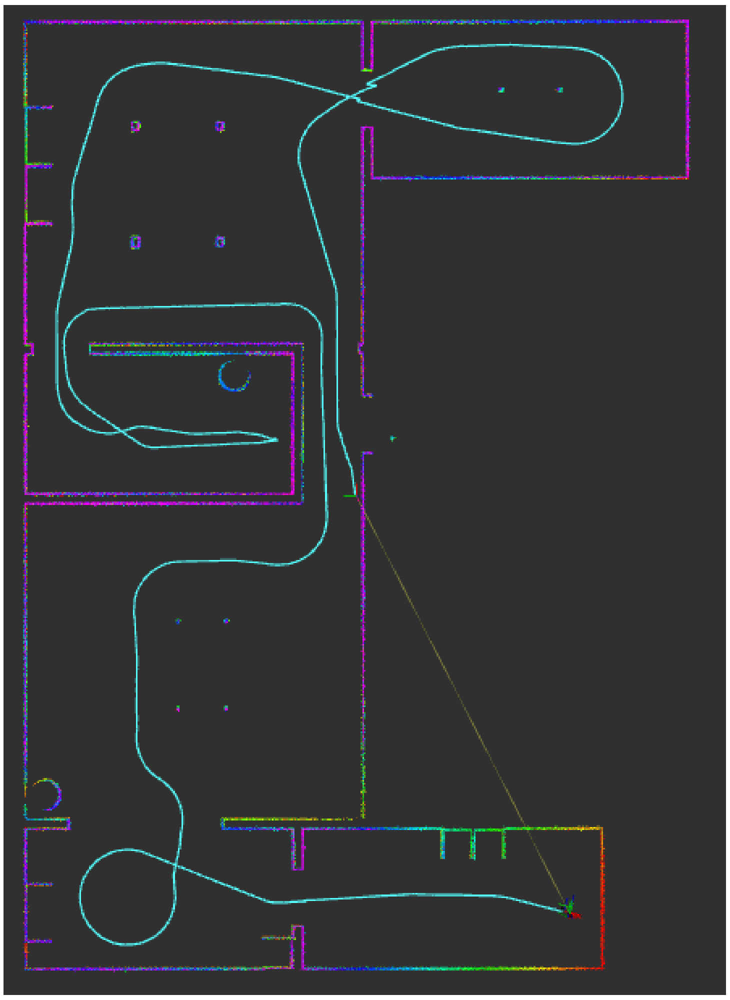


Figure 6.3: LIO-SAM with TurtleBot3 Burger.

loop closures by comparing the features that are extracted from the point clouds. After a matching is found, it goes through a validation stage that evaluates the relative pose using ICP [21].

The back-end optimization performs outlier rejection and pose graph optimization (PGO). Spurious inter-robot loop closures may alter the robot's estimated trajectory. Hence, outlier rejection needs to check the consistency between pairs of loop closures: this is done by evaluating the Euclidean distance, that should not exceed a likely threshold. Then, PGO acts and refines the global map and the robots' trajectories.

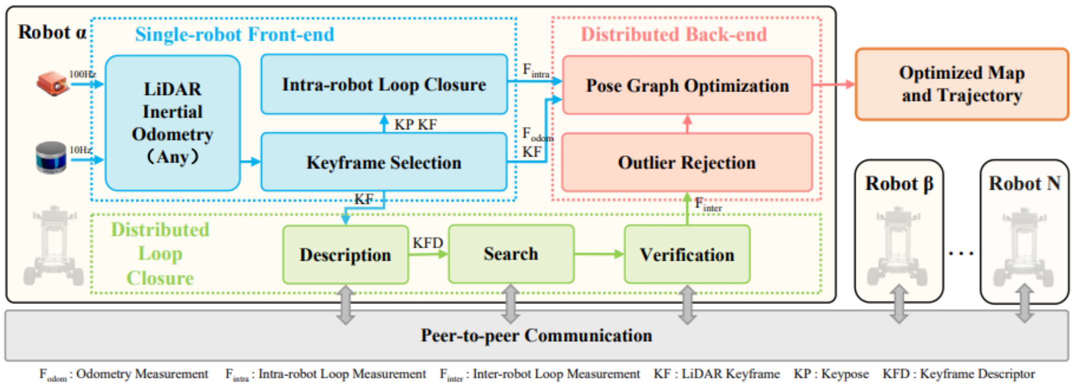


Figure 6.4: DCL-SLAM architecture [34].

6.2.1 LiDAR Descriptors for Distributed Loop Closure

In general, a data descriptor is a data structure that enriches the information about a piece of data with features or other attributes. LiDAR descriptors provide useful information of a point cloud and are usually lightweight, i.e., they contain compact data that can be easily shared between robots. This makes place recognition easier and more feasible in a multi-robot SLAM scenario.

Nowadays, the state-of-the-art of LiDAR descriptors is represented by LiDAR Iris [35]. It is a global LiDAR descriptor, i.e., it is able to extract features from big point clouds in a compact way. The first step of LiDAR Iris is the projection of the point cloud to its bird's eye view. According to the radial and angular range of the LiDAR sensor, the projection is encoded into bins that track how many points fall within them. These bins create a two-dimensional image of the corresponding point cloud.

However, such a descriptor is still not invariant with respect to translation, and the matching between two translated images may fail. As a second step, the translation estimate is recovered by means of Fourier transform: given two images $\boldsymbol{I}_1(x, y)$ and $\boldsymbol{I}_2(x, y)$ such that $\boldsymbol{I}_1(x, y) = \boldsymbol{I}_2(x - \delta_x, y - \delta_y)$ (i.e., they differ by a translation (δ_x, δ_y)), their normalized cross power spectrum is computed by applying the ratio between the Fourier transform of the two images $\hat{\boldsymbol{I}}_1(w_x, w_y)$ and $\hat{\boldsymbol{I}}_2(w_x, w_y)$.

$$C\hat{o}rr = \frac{\hat{\boldsymbol{I}}_2(w_x, w_y)}{\hat{\boldsymbol{I}}_1(w_x, w_y)} = \exp\left[-i(w_x\delta_x + w_y\delta_y)\right] \tag{6.2}$$

The inverse Fourier transform to $C\hat{o}rr$ returns the correlation $Corr$ between the two LiDAR-Iris descriptors. The last step improves the descriptive power of

LiDAR Iris by analyzing the image at different resolutions. A LoG-Gabor filter with center frequency f_0 and width σ is applied to each row of the image, thus building a feature vector that describes the image. A LoG-Gabor filter assumes the expression in Equation (6.3).

$$G(f) = \exp\left[-\frac{\log^2(f/f_0)}{2\log^2(\sigma/f_0)}\right] \tag{6.3}$$

A vector collects the LoG-Gabor frequency responses of each row of the LiDAR-Iris image. The matching between two images is evaluated in terms of Hamming distance between the corresponding feature vectors. If the distance does not exceed a given threshold, place recognition is accomplished and the two images represent a loop closure between two point clouds.

DCL-SLAM also offers support for M2DP [68] and Scan Context [69] as LiDAR descriptors for place recognition. M2DP projects the point cloud into multiple 2D planes that represent different points of view, and for each plane the spatial density distribution is extracted as a signature matrix. The singular value decomposition on the signature matrix extracts the left and right singular vectors that act as descriptors of a signature of the point cloud. On the other hand, Scan Context encodes the point cloud into a single matrix, similarly to LiDAR Iris, and performs matching between point clouds by means of first neighbor search.

LiDAR Iris not only outperforms Scan Context and M2DP, but it also covers the drawbacks of both: indeed, Scan Context lacks descriptive power due to the absence of a feature extraction step, and M2DP is also not invariant with respect to rotation.

6.2.2 Back-End Optimization

As a distributed multi-agent SLAM algorithm, DCL-SLAM provides a module for back-end optimization that includes outlier rejection and pose graph optimization (PGO) at a global scale. The distributed back-end module receives keyframes from the single-robot front-end, loop closures and odometry measurements, and returns a global map that is optimized, as well as the robots' trajectories.

For a more robust map merging, outlier rejection is involved to check if the inter-robot loop closures are consistent. DCL-SLAM combines a random sample consensus procedure (RANSAC) [70] with a method called pairwise consistent measurement set maximization (PCM) [71].

First, RANSAC aims at refining measurements and building a mathematical model that is based on the robots' trajectories. This way, outliers that do not fit such a model are recognized and filtered.

Then, PCM checks the consistency between pairs of loop closures. Given the set $\tilde{\boldsymbol{Z}}$ of loop closures, a threshold γ and a consistency metric C, PCM evaluates the consistency between loop closures as it is shown in (6.4). DCL-SLAM uses the Euclidean distance as a metric. Inconsistent loop closures are rejected as outliers.

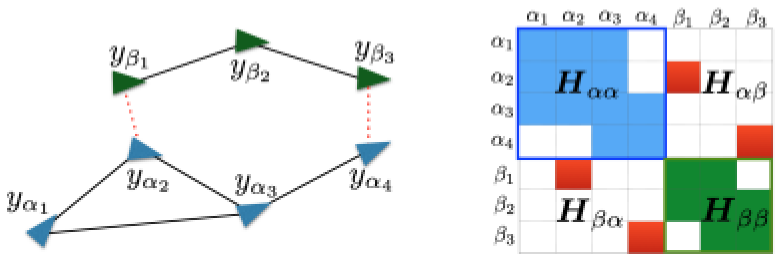
$$C(\boldsymbol{z}_i, \boldsymbol{z}_j) \leq \gamma \quad \forall \boldsymbol{z}_i, \boldsymbol{z}_j \in \tilde{\boldsymbol{Z}} \tag{6.4}$$

After outliers have been cut off, PGO refines the trajectories by solving a maximum likelihood problem. Such problem is built as follows: given a measurement $z_{\alpha_i}^{\beta_j}$ taken by the robot α at time i with respect to the robot β at time j (for example, an inter-robot loop closure), and given the set of all the trajectories $\boldsymbol{x} = [x_\alpha, x_\beta, ...]$, the solution is given by the optimal trajectories $\boldsymbol{x}^* = \left[x_\alpha^*, x_\beta^*, ...\right]$ that maximize the product of the probabilities to measure $z_{\alpha_i}^{\beta_j}$. Equation (6.5) summarizes the maximum likelihood problem for a probability density function $\psi(\cdot)$.

$$\boldsymbol{x}^* = \arg\max_{\boldsymbol{x}} \prod \psi(z_{\alpha_i}^{\beta_j} | \boldsymbol{x}) \tag{6.5}$$

The algorithm that solves (6.5) is known as two-stage distributed Gauss-Seidel (DGS) method [72]. It is an iterative algorithm that aims at optimizing the rotations of the trajectories first, and then it estimates the trajectories with all the poses. The objective is to reach a global consensus for all the mobile robots, according to the constraints that are represented by loop closures.

The algorithm starts from the initial estimates $\boldsymbol{y}^{(0)} = \left[\boldsymbol{y}_\alpha^{(0)}, \boldsymbol{y}_\beta^{(0)}, ...\right]$, where $\{\alpha, \beta...\} = \boldsymbol{\Omega}$ is the set of mobile robots. Here, $\boldsymbol{y}$ can represent either the rotations or the complete trajectories that are estimated by all the robots. DGS first improves rotations, and then it estimates the global trajectories. According to the constraints between poses (for example, loop closures), a matrix $\boldsymbol{H}$ is built. This matrix is divided into blocks that correspond to intra-robot and inter-robot loop closures. A practical example about the structure of $\boldsymbol{H}$ is reported in Fig. 6.5.



y_{β_1}
y_{β_2}
y_{β_3}
y_{α_1}
y_{α_2}
y_{α_3}
y_{α_4}
α_1 α_2 α_3 α_4 β_1 β_2 β_3
$\boldsymbol{H}_{\alpha\alpha}$
$\boldsymbol{H}_{\alpha\beta}$
$\boldsymbol{H}_{\beta\alpha}$
$\boldsymbol{H}_{\beta\beta}$

Figure 6.5: Block structure of $\boldsymbol{H}$ for DGS.

At each iteration k, one robot improves its estimate $\boldsymbol{y}_\alpha^{(k+1)}$, and the others are kept as fixed. Hence, during an iteration, only one robot at a time improves its estimate. Thus, the set $\boldsymbol{\Omega}$ can be split into two subsets: $\boldsymbol{\Omega}^+$ contains the robots that have already computed their $(k+1)$-th estimate, while $\boldsymbol{\Omega}^-$ includes those that have not updated their estimate yet. Furthermore, a vector $\boldsymbol{g}$ is defined with respect to the global reference frame, and it can represent either the rotations or the full poses in the global frame. Following this notation, Equation (6.6) shows the update step that is applied for each $\alpha \in \boldsymbol{\Omega}$.

$$\boldsymbol{y}_\alpha^{(k+1)} = \boldsymbol{H}_{\alpha\alpha}^{-1}(-\sum_{\delta\in\boldsymbol{\Omega}^+} \boldsymbol{H}_{\alpha\delta}\boldsymbol{y}_\delta^{(k+1)} - \sum_{\delta\in\boldsymbol{\Omega}^-} \boldsymbol{H}_{\alpha\delta}\boldsymbol{y}_\delta^{(k)} + g_\alpha) \tag{6.6}$$

The procedure runs until all the robots reach an agreement. It can be proven that DGS will converge from any initial estimate $\boldsymbol{y}^{(0)}$.

6.2.3 DCL-SLAM Code

A repository for DCL-SLAM exists [73]. However, this is a ROS1 package and does not compile on ROS2. The C++ source code has been modified in order to support ROS2 API, the launch files have been rewritten as Python files, and the dependencies have been exported for compatibility with ROS2 build system. At the current state, the DCL-SLAM ROS2 package can be compiled correctly on ROS2 Foxy.

DCL-SLAM defines three custom types of messages that mobile agents can exchange for inter-robot loop closure and global optimization.

- A `dcl_slam::msg::GlobalDescriptor` message wraps a header (i.e., timestamp and reference frame), an index on the factor graph and a vector of values that characterize the global descriptor that is chosen for place recognition. For example, if LiDAR Iris is used as a global descriptor, these values would correspond to the bird's eye view image that is stretched into a vector.

- A `dcl_slam::msg::LoopInfo` message contains the information about loop closures of any type. It contains a header, the names of the two mobile robots and their corresponding indices, the poses of the two agents and the relative transformation. For intra-robot loop closures, such a message refers to only one robot.

- A `dcl_slam::msg::NeighborEstimate` message provides the estimates that a mobile robot computes with respect to its neighbors. It includes a header, the index of the current robot, some flags that concern global optimization, the indices of the neighbors, and a vector with the estimated values. Such a message can contain either the relative rotation matrices between neighbors or the relative 6D poses (i.e., translation and roll, pitch and yaw angles).

Many functionalities of DCL-SLAM are related to the `DistributedMapping` ROS node. Since DCL-SLAM is a distributed multi-agent SLAM algorithm, a `DistributedMapping` node should exist for each mobile robot that is involved.

When created, it reads the namespace of the current mobile robot and configures the publishers and the subscriptions, according to the principle that was introduced at the start of this chapter. The `DistributedMapping` node also implements intra-robot loop closure: this way, DCL-SLAM can support single-agent SLAM algorithms that do not perform loop closure such as LOAM [20]. Intra-robot loop closure is based on factors and iSAM2 [62], similarly to LIO-SAM. Inter-robot loop closure follows the same steps that were described for LiDAR-Iris: after a matching between descriptors is found, it calculates the relative pose through ICP. Then, the two corresponding point clouds can be aligned to close the loop between agents.

The other purpose of `DistributedMapping` is to manage the distributed back-end module for PGO and outlier rejection. In ROS1, the `distributed_mapper` package exists: it implements both PCM for outlier rejection and the two-stage DGS algorithm for global PGO. In order to provide support for ROS2, the source code and the list of dependencies have been modified, such that `distributed_mapper` can correctly compile on ROS2 Foxy.

`DistributedMapping` contains some C++ objects that are required to implement PCM and the two-stage DGS algorithm.

First, all the local measurements and intra-robot loop closures should be stored into a `RobotLocalMap` object, and the pose estimates from neighbors should be saved as `Trajectory` objects into a local map. A `Trajectory` includes the poses and the corresponding indices to the pose graph. The `outliersFiltering()` class method takes the measurements from neighbors and applies PCM, thus removing any outlier and updating the consistent inter-robot loop closures.

For what concerns the two-stage DGS algorithm, `distributed_mapper` defines a `DistributedMapper` object (not to be confused with the `DistributedMapping` ROS node!) that integrates the global graph and an optimizer based on two-stage DGS. It stores and updates the global pose graph that contains the robots' measurements and the loop closures. `DistributedMapper` also differentiates between rotation and pose optimization, coherently with the two-stage approach of the algorithm: this is why a `dcl_slam::msg::NeighborEstimate` message can refer either to relative rotation or displacement. Indeed, global optimization runs each time a mobile robot receives such messages from another.

DCL-SLAM establishes a ROS-based peer-to-peer communication between robots that relies on topics. As it was mentioned before, each agent publishes the topics under its namespace and subscribes to the topics of other agents.

Two topics are required for inter-robot loop closure.

- `/*/d_m/global_descriptors`[1] is used for exchanging global descriptors in the form of `dcl_slam::msg::GlobalDescriptor` messages.
- `/*/d_m/loop_info` is the communication channel for intra-robot and inter-robot loop closures that are compressed into `dcl_slam::msg::LoopInfo` messages.

However, these communication channels are not sufficient to implement global optimization. Hence, other topics that allow to exchange the estimates between mobile robots have to be defined. Furthermore, some information about the state of the optimizer and the progress of the distributed algorithm is required.

- `/*/d_m/optimization_state` is a communication channel for simple messages that contain an integer that encodes the state of the optimizer for a mobile robot.
- Two channels are exploited by mobile robots to pass their estimates to neighbors. `/*/d_m/neighbor_rotation_estimates` contains the rotation estimates, while `/*/d_m/neighbor_pose_estimates` is used for pose estimates. Both topics accept messages of type `dcl_slam::msg::NeighborEstimate`.
- `/*/d_m/rotation_estimate_state` and `/*/d_m/pose_estimate_state` are other two simple topics that permit to exchange integers. They are used to synchronize the progress of the two-stage DGS algorithm. Such a message either contains 1 (equivalent to true) or 0 (false): if true, the incoming estimate has already gone through the update step of the algorithm at the current iteration. Once again, two separate topics are used, because the algorithm is two-stage and computes the global rotation and pose estimates separately.

6.2.4 Adaptations for LIO-SAM and TurtleBot3 Burger

The `DistributedMapping` C++ object also includes an intra-robot loop closure procedure. However, LIO-SAM is enhanced with loop closure as well. Hence, LIO-SAM code should be modified, such that the loop closure functionalities are replaced by those that the `DistributedMapping` node offers.

To do so, LIO-SAM has to instantiate a `DistributedMapping` object. Since this node affects loop closure, it should be created along with the LIO-SAM `MapOptimization` node. Hence, the only module that needs some adjustments is `mapOptimization.cpp`. The `DistributedMapping` node can already create and

[1]The asterisk in `/*/d_m/global_descriptors` is a generalization of the topic for any namespace. Furthermore, `d_m` is an abbreviation that stands for `distributed_mapping`. Example: `/a/distributed_mapping/global_descriptors`.

manage the factor graph by adding factors, and it also invokes iSAM2 for state estimation. In LIO-SAM, loop closure and optimization are invoked each time that a `lio_sam::msg::CloudInfo` is received from the feature extraction module. Following the same logic, `DistributedMapping` should add factors and perform optimization in the same callback.

The same reasoning can be applied to any other LiDAR inertial odometry SLAM algorithm. The idea is to call the optimization method of the `DistributedMapping` object after the SLAM algorithm has performed feature extraction on the current point cloud. This way, the `DistributedMapping` node is strictly related to that robot: it adds factors to the graph and solves the state estimation problem.

There is no need to perform any adaptation to the code to support TurtleBot3 Burger. Since the point cloud is already two-dimensional, the projection to the bird's eye view is trivial. This is why it is convenient to use LiDAR Iris as the global descriptor, besides its advantages with respect to Scan Context and M2DP.

Chapter 7

Conclusions

In this thesis work, two ROS-based solutions for multi-agent SLAM have been discussed.

The centralized approach emulates inter-robot loop closure and back-end optimization with computer vision techniques that are based on feature matching. This way, the central node only needs the local maps that are passed by the mobile agents through a simple ROS message. This also makes the central node totally independent of the specific single-robot SLAM algorithm that locally runs on each mobile robot. However, due to its centralized nature, the map merging node must not have any failure during the task.

The distributed algorithm is taken from the DCL-SLAM formulation. It has a structure that is based on three modules: single-robot loop closure, inter-robot loop closure and distributed back-end optimization. DCL-SLAM introduces peer-to-peer communication between mobile robots and implements complex functionalities that perform inter-robot loop closure and back-end optimization, according to the theory of factor graphs. DCL-SLAM can support different choices for the single-robot front-end, but a LiDAR inertial odometry SLAM algorithm is required to run on each mobile robot. Furthermore, only LiDAR sensors are currently supported.

There are some interesting aspects that make `multirobot_map_merge` a suitable choice for a multi-robot back-end module for multi-agent SLAM. It can support different single-robot front-end SLAM algorithms, as long as these publish `nav_msgs::msg::OccupancyGrid` messages. This also means that heterogeneous agents may be used: if a mobile robot is equipped with an RGB-D camera and a visual SLAM algorithm exists, then it can be added to the swarm of mobile robots even if the others are equipped with LiDAR sensors. Furthermore, the feature matching techniques are robust enough to grant convergence of the algorithm, i.e., at a certain point the local maps will be merged correctly. If the transformations are estimated correctly since the beginning of the task, the mobile robots can be sent to different areas of the environment, thus shortening the time to accomplish mapping. If n agents are employed for a task that would require a time T, the total time to complete the task can be decreased to T/n in the best-case scenario.

However, `multirobot_map_merge` brings some downsides as well. It is still a central node, and the computational load is aggravated by the OpenCV functions that require many operations. During the task, it can happen that an OpenCV function returns a C++ exception due to out-of-memory issues, and a fix to such a problem has not been found yet. If such failure happens, all the progress of the transformations is lost. Hence, if a good merging is found but the central node crashes, it cannot be recovered. The map merging node can be launched again, but without the benefits of reducing the time to complete the task, because the feature matching may need an inter-robot loop closure to recover the optimal transformation.

On the other hand, DCL-SLAM is a good solution to implement a fully-distributed multi-agent SLAM algorithm. As a distributed approach, it does not suffer from the single-point-of-failure issue that may affect a central server that is involved into multi-robot SLAM. Instead, the mobile agents enrich their data by gathering information from neighbors, such that the same global map is built by the whole team of robots. The peer-to-peer communication that is built on ROS topics does not suffer from bandwidth issues, thanks to the usage of a compact descriptor such as LiDAR Iris. Although the back-end functionalities may seem complex, the implementations of PCM and two-stage distributed Gauss-Seidel algorithm are offered by the `distributed_mapper` package. Thanks to the robustness of the distributed back-end module against outliers, DCL-SLAM is able to reconstruct a global map and estimate the global trajectories with very high accuracy. It is also shown how to add support to any LiDAR inertial odometry SLAM algorithm in order to be compatible with DCL-SLAM.

The only drawback of DCL-SLAM is that it lacks versatility: currently, DCL-SLAM only supports mobile agents that are equipped with LiDAR sensors, and it does not work properly with single-robot SLAM algorithms that do not rely on LiDAR inertial odometry. Furthermore, DCL-SLAM builds a point cloud map that does not provide any useful information for navigation, differently from a grid map.

7.1 Further Improvements

For what concerns the centralized multi-agent SLAM approach, this thesis work has presented a comparison for the single-robot front-end and some improvements for SLAM Toolbox, and it also has added additional support for SLAM Toolbox to the `multirobot_map_merge` package. Furthermore, some results have explained how such a framework operates, and tests have been provided in simulated and real-world scenarios. However, there are still some points that were left behind, and a solution to such problems has not been found yet.

The issues that are related to sporadic failures are not solved, but it is a good idea to further investigate into the low-level operations that OpenCV functions carry out, and understand the origin of such exceptions. Another improvement would concern the confidence threshold: a fixed value has been set in the simulated environment, while the real-world experiment has been based on a strategy that starts from a low threshold to get an initial transformation, and then it increases its value to correct the initial guess, until a maximum value is reached. Other strategies may be implemented as well, and their effectiveness could be evaluated on the basis of a simulated scenario. Outlier detection is still an option to improve the quality of the transformations, and it would be interesting to study how outlier rejection affects the result of a single map merging. Furthermore, the solution to the grid size issue for SLAM Toolbox can be improved by dynamically assigning the size of the biggest map to the others, such that larger environments can be mapped without problems. This improvement could not be implemented, due to OpenCV exceptions during the padding.

A last upgrade on the centralized multi-agent SLAM approach may rely on Nav2 to make the whole system completely autonomous. This requires a central navigation server that should synchronize with the map merging node: when the transformations are correctly set, the navigation server should send goals to the agents towards unexplored areas of the environment. In order to optimize time, these goals should stimulate the robots to initially explore different zones: eventually, they will retrace their steps and visit the same area of another agent, thus having an inter-robot loop closure.

The main contribution of this thesis work on the existent formulations of LIO-SAM and DCL-SLAM regards the adaptation of LIO-SAM with TurtleBot3 Burger and the compatibility of DCL-SLAM and its dependencies with ROS2 Foxy. Some results have been shown for LIO-SAM. At the current state, the code of DCL-SLAM and the `distributed_mapper` package have been successfully ported to ROS2.

Unfortunately, due to difficulties encountered in migrating the code from ROS1 to ROS2, DCL-SLAM has not been tested yet. Hence, it was not possible to provide a practical application of DCL-SLAM on two copies of TurtleBot3 Burger. Nevertheless, it was shown that LIO-SAM works with such a mobile robot. Due to the lack of testing, there are not any other known issues.

A further work on DCL-SLAM would surely include some testing on TurtleBot3 Burger, in both simulation and a real-world scenario. First of all, an implementation of DCL-SLAM that works correctly for TurtleBot3 Burger would not only amplify the educational purpose of TurtleBot3 Burger, but it would be a huge contribution for the ROS community as well. Moreover, it would be interesting to evaluate the efficiency of DCL-SLAM, for example, in terms of time required for the distributed two-stage Gauss-Seidel algorithm to converge for all the agents.

Acknowledgements

Finalmente, dopo aver concluso questo splendido percorso presso il Politecnico di Torino, posso prendermi un attimo di tempo per esprimere la mia più profonda gratitudine nei confronti di tutte le persone che mi hanno accompagnato, sostenuto, e che soprattutto hanno creduto in me fino alla fine di questi due anni.

Ringrazio la professoressa Marina Indri per avermi dato l'opportunità di mettermi in gioco e per avermi seguito dall'inizio fino all'ultimo giorno. E ringrazio anche David, che è sempre stato disponibile per chiarimenti e consigli sul prossimo passo da compiere, anche quando avevo difficoltà e non avevo un piano chiaro in mente.

Ringrazio la mia famiglia e i miei parenti vicini e lontani, perché è anche merito del loro supporto e del loro affetto se sono arrivato fino in fondo. E soprattutto ringrazio mio fratello Alessandro, non solo per avermi confortato in quei momenti in cui avevo perso il controllo, ma anche perché la sua presenza mi ha fatto sentire meno solo, meno fragile e più determinato.

Ringrazio i colleghi che ho incrociato lungo il mio percorso, per aver reso la nostra esperienza al Politecnico più lieve e più entusiasmante, e perché dopo i due anni di pandemia mi hanno fatto sentire parte di quella comunità di cui ogni studente del Poli fa parte.

Ringrazio i miei amici, compagni di uscite, di risate e di bevute, perché mi hanno insegnato a non mollare, ad affrontare le difficoltà con il sorriso, ma anche a prendersi qualche momento di pausa quando serve.

Voi tutti avete lasciato un'impronta importante nella mia vita, e farò tesoro di ciò che mi avete trasmesso.

Grazie.

Bibliography

[1] T. Bailey and H. Durrant-Whyte. «Simultaneous localization and mapping (SLAM): part II». In: *IEEE Robotics & Automation Magazine* 13.3 (2006), pp. 108–117.

[2] X. Zhou and R. Huang. «A State-of-the-Art Review on SLAM». In: *Intelligent Robotics and Applications*. Ed. by H. Liu, Z. Yin, L. Liu, L. Jiang, G. Gu, X. Wu, and W. Ren. Springer International Publishing, 2022, pp. 240–251.

[3] H. Durrant-Whyte and T. Bailey. «Simultaneous localization and mapping: part I». In: *IEEE Robotics & Automation Magazine* 13.2 (2006), pp. 99–110.

[4] L. Zhao, Z. Mao, and S. Huang. «Feature-Based SLAM: Why Simultaneous Localisation and Mapping?» In: July 2021.

[5] I. Abaspur Kazerouni, L. Fitzgerald, G. Dooly, and D. Toal. «A survey of state-of-the-art on visual SLAM». In: *Expert Systems with Applications* 205 (2022).

[6] B. Zhou, D. Xie, S. Chen, H. Mo, C. Li, and Q. Li. «Comparative Analysis of SLAM Algorithms for Mechanical LiDAR and Solid-State LiDAR». In: *IEEE Sensors Journal* 23.5 (2023), pp. 5325–5338.

[7] Y. Zhang, Y. Wu, K. Tong, H. Chen, and Y. Yuan. «Review of Visual Simultaneous Localization and Mapping Based on Deep Learning». In: *Remote Sensing* 15.11 (2023).

[8] Y. An, J. Shi, D. Gu, and Q. Liu. «Visual-LiDAR SLAM Based on Unsupervised Multi-channel Deep Neural Networks». In: *Cognitive Computation* 14 (July 2022), pp. 1–13.

[9] T. Bailey, J. Nieto, J. Guivant, M. Stevens, and E. Nebot. «Consistency of the EKF-SLAM Algorithm». In: *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2006, pp. 3562–3568.

[10] L. M. Paz, J. D. TardÓs, and J. Neira. «Divide and Conquer: EKF SLAM in $O(n)$». In: *IEEE Transactions on Robotics* 24.5 (2008), pp. 1107–1120.

[11] K. Krinkin, A. Filatov, A. Filatov, A. Huletski, and D. Kartashov. «The Scan Matchers Research and Comparison: Monte-Carlo, Olson and Hough». In: *Conference of Open Innovation Association, FRUCT*. 2017, pp. 99–105.

[12] G. Grisetti, C. Stachniss, and W. Burgard. «Improved techniques for grid mapping with Rao-Blackwellized particle filters». In: *IEEE Transactions on Robotics* 23.1 (2007), pp. 34–46.

[13] S. Liu, Y. Lei, and X. Dong. «Evaluation and Comparison of Gmapping and Karto SLAM Systems». In: *2022 12th International Conference on CYBER Technology in Automation, Control, and Intelligent Systems (CYBER)*. 2022, pp. 295–300.

[14] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. «FastSLAM: A Factored Solution to the Simultaneous Localization And Mapping Problem». In: *Proceedings of the National Conference on Artificial Intelligence*. 2002, pp. 593–598.

[15] G. Grisetti, R. Kümmerle, C. Stachniss, and W. Burgard. «A Tutorial on Graph-Based SLAM». In: *IEEE Intelligent Transportation Systems Magazine* 2.4 (2010), pp. 31–43.

[16] W. Hess, D. Kohler, H. Rapp, and D. Andor. «Real-time loop closure in 2D LIDAR SLAM». In: *Proceedings - IEEE International Conference on Robotics and Automation*. Vol. 2016-June. 2016, pp. 1271–1278.

[17] K. Konolige, G. Grisetti, R. Kümmerle, W. Burgard, B. Limketkai, and R. Vincent. «Efficient Sparse Pose Adjustment for 2D mapping». In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2010, pp. 22–29.

[18] A. Dwijotomo, M. A. A. Rahman, M. H. M. Ariff, H. Zamzuri, and W. M. H. W. Azree. «Cartographer SLAM Method for Optimization with an Adaptive Multi-Distance Scan Scheduler». In: *Applied Sciences (Switzerland)* 10.1 (2020).

[19] B. Liu, Z. Guan, B. Li, G. Wen, and Y. Zhao. «Research on SLAM Algorithm and Navigation of Mobile Robot Based on ROS». In: *2021 IEEE International Conference on Mechatronics and Automation (ICMA)*. 2021, pp. 119–124.

[20] J. Zhang and S. Singh. «LOAM: Lidar Odometry and Mapping in Real-time». In: (2014).

[21] S. Rusinkiewicz and M. Levoy. «Efficient variants of the ICP algorithm». In: *Proceedings Third International Conference on 3-D Digital Imaging and Modeling*. 2001, pp. 145–152.

[22] W. Xu, Y. Cai, D. He, J. Lin, and F. Zhang. «FAST-LIO2: Fast Direct LiDAR-Inertial Odometry». In: *IEEE Transactions on Robotics* 38.4 (2022), pp. 2053–2073.

[23] T. Shan, B. Englot, D. Meyers, W. Wang, C. Ratti, and D. Rus. «LIO-SAM: Tightly-coupled lidar inertial odometry via smoothing and mapping». In: *IEEE International Conference on Intelligent Robots and Systems.* 2020, pp. 5135–5142.

[24] Z. Liu and F. Zhang. «BALM: Bundle Adjustment for Lidar Mapping». In: *IEEE Robotics and Automation Letters* 6.2 (2021), pp. 3184–3191.

[25] Y. Pan, P. Xiao, Y. He, Z. Shao, and Z. Li. «Mulls: Versatile LiDAR SLAM via Multi-metric Linear Least Square». In: *Proceedings - IEEE International Conference on Robotics and Automation.* Vol. 2021-May. 2021, pp. 11633–11640.

[26] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardos. «ORB-SLAM: a Versatile and Accurate Monocular SLAM System». In: *CoRR* (2015).

[27] R. Mur-Artal and J. D. Tardos. «ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras». In: *IEEE Transactions on Robotics* 33.5 (2017), pp. 1255–1262.

[28] C. Campos, R. Elvira, J. J. G. Rodrìguez, J. M. M. Montiel, and J. D. Tardòs. «ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual-Inertial and Multi-Map SLAM». In: *CoRR* (2020).

[29] C.-M. Chung, Y.-C. Tseng, Y.-C. Hsu, X.-Q. Shi, Y.-H. Hua, J.-F. Yeh, W.-C. Chen, Y.-T. Chen, and W. H. Hsu. *Orbeez-SLAM: A Real-time Monocular Visual SLAM with ORB Features and NeRF-realized Mapping.*

[30] S. Saeedi, L. Paull, M. Trentini, and H. Li. «Multiple robot simultaneous localization and mapping». In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems.* 2011, pp. 853–858.

[31] A. Huletski, D. Kartashov, and K. Krinkin. «VinySLAM: An indoor SLAM method for low-cost platforms based on the Transferable Belief Model». In: 2017-September (2017), pp. 6770–6776.

[32] A. Filatov and K. Krinkin. «Multi-Agent SLAM approaches for low-cost platforms». In: 2019-April (2019), pp. 89–95.

[33] K. Ebadi et al. «LAMP: Large-Scale Autonomous Mapping and Positioning for Exploration of Perceptually-Degraded Subterranean Environments». In: *Proceedings - IEEE International Conference on Robotics and Automation.* 2020, pp. 80–86.

[34] S. Zhong, Y. Qi, Z. Chen, J. Wu, H. Chen, and M. Liu. *DCL-SLAM: A Distributed Collaborative LiDAR SLAM Framework for a Robotic Swarm.* 2023.

[35] Y. Wang, Z. Sun, C. .-. Xu, S. E. Sarma, J. Yang, and H. Kong. «LiDAR iris for loop-closure detection». In: *IEEE International Conference on Intelligent Robots and Systems*. 2020, pp. 5769–5775.

[36] P.-Y. Lajoie and G. Beltrame. *Swarm-SLAM : Sparse Decentralized Collaborative Simultaneous Localization and Mapping Framework for Multi-Robot Systems*. 2023.

[37] K. M. Brink, J. Zhang, A. R. Willis, R. E. Sherrill, and J. L. Godwin. *Maplets: An Efficient Approach for Cooperative SLAM Map Building Under Communication and Computation Constraints*. 2020.

[38] R. Dubé, A. Cramariuc, D. Dugas, H. Sommer, M. Dymczyk, J. Nieto, R. Siegwart, and C. Cadena. «SegMap: Segment-based mapping and localization using data-driven descriptors». In: *International Journal of Robotics Research* 39.2-3 (2020), pp. 339–355.

[39] J. McConnell, Y. Huang, P. Szenher, I. Collado-Gonzalez, and B. Englot. «DRACo-SLAM: Distributed Robust Acoustic Communication-efficient SLAM for Imaging Sonar Equipped Underwater Robot Teams». In: *IEEE International Conference on Intelligent Robots and Systems*. Vol. 2022-October. 2022, pp. 8457–8464.

[40] *ROS1 Official Website and Documentation*. `https://www.ros.org/`.

[41] *ROS2 Foxy Fitzroy Official Website and Documentation*. `https://docs.ros.org/en/foxy/index.html`.

[42] Y. Maruyama, S. Kato, and T. Azumi. «Exploring the performance of ROS2». In: Oct. 2016, pp. 1–10.

[43] *tf2 Documentation*. `http://wiki.ros.org/tf2`. (Visited on 11/04/2023).

[44] *RViz2 Repository*. `https://github.com/ros2/rviz`. (Visited on 11/04/2023).

[45] *Gazebo Homepage*. `https://gazebosim.org/home`. (Visited on 11/04/2023).

[46] *Nav2 Documentation*. `https://navigation.ros.org/`. (Visited on 11/04/2023).

[47] *TurtleBot3 Documentation*. `https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/`. (Visited on 01/05/2023).

[48] *TurtleBot3 Essential Packages Repository*. `https://github.com/ROBOTIS-GIT/turtlebot3`. (Visited on 01/05/2023).

[49] *SLAM Toolbox Repository*. `https://github.com/SteveMacenski/slam_toolbox`. (Visited on 04/09/2023).

[50] S. Macenski and I. Jambrecic. «SLAM Toolbox: SLAM for the dynamic world». In: *Journal of Open Source Software* 6.61 (2021), p. 2783.

[51] X. S. Le, L. Fabresse, N. Bouraqadi, and G. Lozenguez. «Evaluation of out-of-the-box ROS 2D slams for autonomous exploration of unknown indoor environments». In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10985 LNAI (2018), pp. 283–296.

[52] K. Trejos, L. Rincón, M. Bolaños, J. Fallas, and L. Marín. «2D SLAM Algorithms Characterization, Calibration, and Comparison Considering Pose Error, Map Accuracy as Well as CPU and Memory Usage». In: *Sensors* 22.18 (2022).

[53] S. Agarwal, K. Mierle, and T. C. S. Team. *Ceres Solver*. Version 2.2. Oct. 2023. URL: `https://github.com/ceres-solver/ceres-solver`.

[54] *Ceres Solver C++ Documentation*. `http://ceres-solver.org/`.

[55] *multirobot_map_merge ROS2 Repository*. `https://github.com/robo-friends/m-explore-ros2`. (Visited on 10/06/2023).

[56] *OpenCV Documentation*. `https://opencv.org/`.

[57] P. Fernández Alcantarilla. «Fast Explicit Diffusion for Accelerated Features in Nonlinear Scale Spaces». In: Sept. 2013.

[58] H. Bay, T. Tuytelaars, and L. Van Gool. «SURF: Speeded up robust features». In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 3951 LNCS (2006), pp. 404–417.

[59] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. «ORB: An efficient alternative to SIFT or SURF». In: *2011 International Conference on Computer Vision*. 2011, pp. 2564–2571.

[60] S. A. K. Tareen and Z. Saleem. «A comparative analysis of SIFT, SURF, KAZE, AKAZE, ORB, and BRISK». In: *2018 International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)*. 2018, pp. 1–10.

[61] *LIO-SAM Repository*. `https://github.com/TixiaoShan/LIO-SAM`. (Visited on 04/09/2023).

[62] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. Leonard, and F. Dellaert. «iSAM2: Incremental smoothing and mapping with fluid relinearization and incremental variable reordering». In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 3281–3288.

[63] *PCL Official Website*. `https://pointclouds.org/`.

[64] *GTSAM Official Website*. `https://gtsam.org/`.

[65] *Simulation Environment for LIO-SAM with 3D Velodyne*. `https://drive.google.com/drive/folders/15RzJOWTHs74MB8u_LxLM5RkA4gvQjYHj`. (Visited on 12/11/2023).

[66] *ROS2 Velodyne Drivers*. `https://github.com/ros-drivers/velodyne`. (Visited on 12/11/2023).

[67] *laser_geometry ROS2 Repository*. `https://github.com/ros-perception/laser_geometry`.

[68] L. He, X. Wang, and H. Zhang. «M2DP: A novel 3D point cloud descriptor and its application in loop closure detection». In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, pp. 231–237.

[69] G. Kim and A. Kim. «Scan Context: Egocentric Spatial Descriptor for Place Recognition Within 3D Point Cloud Map». In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 4802–4809.

[70] M. A. Fischler and R. C. Bolles. «Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography». In: *Commun. ACM* 24.6 (1981), pp. 381–395.

[71] J. G. Mangelson, D. Dominic, R. M. Eustice, and R. Vasudevan. «Pairwise Consistent Measurement Set Maximization for Robust Multi-Robot Map Merging». In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. May 2018, pp. 2916–2923. DOI: `10.1109/ICRA.2018.8460217`.

[72] S. Choudhary, L. Carlone, C. Nieto, J. Rogers, H. I. Christensen, and F. Dellaert. «Distributed Trajectory Estimation with Privacy and Communication Constraints: a Two-Stage Distributed Gauss-Seidel Approach». In: *IEEE International Conference on Robotics and Automation 2016*. 2016.

[73] *DCL-SLAM ROS1 Repository*. `https://github.com/PengYu-Team/DCL-SLAM/`.