POLITECNICO DI TORINO

Master's Degree in Computer Engineering
Automation and Intelligent Cyber-Physical Systems

Master's Degree Thesis

Implementation of a ROS-based Autonomous Multi-Agent Cooperative SLAM Approach

Supervisors

Prof. Marina INDRI

Ph.D. Pangcheng David CEN CHENG

Candidate

Fabrizio PISANI

December 2024

A Mamma, Papà, Dalila e Nannì.
Che mi hanno accompagnato in ogni momento della mia vita.
Che mi hanno supportato e amato in ogni cosa facessi.
E che mi hanno insegnato a
"Viaggiare in direzione ostinata e contraria".

Abstract

Nowadays, mobile robots are utilized across a multitude of domains, including everyday life and research and work environments. Consequently, research in the field of autonomous robots has expanded considerably over the past few decades. In order to perform a task, a mobile robot must be able to localize itself in the surrounding environment. However, the operating area of a mobile robot may often be unknown, inaccessible, or dangerous. In such cases, Simultaneous Localization and Mapping (SLAM) is a key functionality that allows mobile robots to construct a real-time map of an unknown environment, while simultaneously localizing themselves within it. Due to its significance, SLAM conducted by a single agent in an isolated setting has long been regarded as the state of the art in mobile robotics research. However, in complex scenarios, single-agent SLAM is susceptible to limitations. To achieve faster and more accurate results, the Multi-Agent Cooperative SLAM (C-SLAM) system was born. The motivation behind the expansion of the research domain is rooted in the increasing demand for more efficient, scalable robotic systems capable of autonomous exploration in dynamic and complex environments. C-SLAM extends traditional SLAM by allowing multiple robots to collaborate in map construction and simultaneous localization. Another important step in SLAM research is Active SLAM. It enables robots to actively plan their movements and explore unknown environments, reducing map uncertainty through strategic decision-making, without active human assistance. A further focus of this research is the integration of Active SLAM (A-SLAM) into the collaborative framework, leading to Active Collaborative SLAM. With the advent of artificial intelligence, machine learning and robot learning, the Active SLAM will be a key area of robotic research in the future.

The initial section of the thesis presents a comprehensive overview of existing SLAM techniques in both single-robot and multi-robot scenarios, highlighting the limitations of single-robot systems and the opportunities presented by multi-agent architectures. The second part of this thesis work introduces an autonomous centralized multi-agent approach, implemented using the Robot Operating System (ROS2 Humble) framework. Its performance is validated through both simulation (the Gazebo environment) and real-world experiments on two TurtleBot3 Burger platforms. In the considered system, each agent moves autonomously and creates a local real-time map independently. At the same time, a central server merges the two maps into a global, more accurate map. The single-agent SLAM algorithm used is the SLAM Toolbox. The path is planned locally on the single agent by the global planner A^*, then smoothed by the B-spline and executed by the Pure Pursuit algorithm.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Autonomous navigation has made substantial progress particularly within indoor, office-like environments, where it has become an increasingly popular technology [1]. The utilization of autonomous robots in industrial settings is also on the rise, driven by the potential for significant cost savings in areas such as manual material handling, which often involves substantial labour costs. In this context, the deployment of autonomous robotic vehicles can reduce material handling expenses by as much as 30%.

One of the fundamental challenges in this field is the ability of a robot to autonomously navigate in the surrounding environment. In this context, significant advancements have been made, in the recent decades, in the field of Simultaneous Localization and Mapping (SLAM), which has led to a notable enhancement in the autonomous navigation abilities of mobile robots [2]. SLAM enables robots to simultaneously localize themselves and create real-time maps of their surroundings without the need for prior information, thereby facilitating autonomous decision-making and control across a range of terrains and environments.

Although SLAM has undergone considerable developments over time, single-robot systems could present difficulties in terms of time efficiency, robustness and mapping accuracy, particularly when tasked with exploring vast or dynamic spaces. Multi-Agent Cooperative SLAM (C-SLAM) has emerged as a promising extension of traditional SLAM, where multiple robots cooperate to solve the mapping and localization problem.

Multi-robot systems are of significant importance in a multitude of robotics applications, including warehouse management and search and rescue missions [3]. Nevertheless, they encounter difficulties in attaining shared situational awareness, particularly in unfamiliar environments where external localization systems like GPS are unreliable. Consequently, there is an increasing necessity for multi-robot SLAM systems that can function autonomously without reliance on external support.

Serov et al. [4] identify two key of collaborative robotics. Firstly, the exploration of the environment is accelerated by employing multiple robots. Secondly, robots with varying capabilities and mobility characteristics can be utilised to carry out diverse tasks or to investigate areas that certain robots in the swarm cannot access.

A further significant area of research within the SLAM field is Active SLAM (A-SLAM). This integrates decision-making processes into the mapping and localization process, with the objective of implementing autonomous agents in practical applications, thereby eliminating the need for human control. A-SLAM has been a topic of interest for over three decades [5]. Recently, it has attracted renewed attention, particularly due to the new possibilities presented by learning-based approaches. In fact, the number of publications on A-SLAM has increased exponentially, from 53 in 2010 to over 660 in 2022.

The integration of Active SLAM into collaborative systems (AC-SLAM) offers even greater potential by allowing multiple robots to coordinate their exploration strategies. The versatility of these methods makes them applicable to a diverse range of domains, including search and rescue operations, planetary exploration, precision agriculture, autonomous navigation in crowded environments, underwater exploration, artificial intelligence, assistive robotics, and autonomous exploration [6].

1.1 Thesis Motivation

The motivation for pursuing research in Multi-Agent Cooperative SLAM is derived from the growing demand for more efficient and scalable robotic systems that are capable of autonomously operating in dynamic and complex environments. The field of multiple-robot SLAM is still in its relative early stages of development, with significant potential for further advancement and investigation [7].

1.2 Thesis Structure

The thesis is divided in the following chapters.

Chapter 2 provides an overview of the main SLAM techniques, exploring the traditional single-robot approach. It focuses on the main algorithms used in the front-end and back-end.

In Chapter 3, the concepts of Active SLAM, Collaborative SLAM and Active

Collaborative SLAM are introduced. The C-SLAM is the extension of the Single-Agent SLAM, while A-SLAM is a variation of SLAM that incorporates active decision-making into the mapping process, allowing robots to adapt their navigation strategies based on the information collected. This chapter also focuses on the Multi-Agent Cooperative SLAM problem and there is an overview of the state-of-art.

Chapter 4 focuses on the software infrastructure used, with particular attention to the ROS2 framework, which forms the basis for the implementation of the AC-SLAM system. Key tools such as RViz, Gazebo, and the Nav2, are illustrated, supporting the simulation and development of the system.

Chapter 5 outlines the main concepts related to the TurtleBot3 Burger, the robotic model used in this thesis.

In Chapter 6 there is a detailed description of the implementation of the Centralized Autonomous Multi-Agent Collaborative SLAM algorithm.

Chapter 7 presents the results of the simulation and the real world experiments conducted on two TurtleBot3 Burger.

Finally, Chapter 8 reports the conclusions of the thesis.

Chapter 2

Simultaneous Localization and Mapping

Simultaneous Localization and Mapping (SLAM) was first introduced by Smith et al. in 1986 [8]. The term SLAM was first used in 2006 by Durrant-Whyte et al. in [9] [10]. SLAM enables robots, drones, or autonomous vehicles to construct a map of an unknown environment while simultaneously tracking their own position within that environment. SLAM is a fundamental component in the navigation of uncharted areas, eliminating the necessity for pre-existing maps. Its applications include autonomous driving, indoor navigation, and the exploration of hazardous areas. The SLAM problem is the union of two interdependent problems: localization, which is the problem of determining the robot's position relative to its surroundings, and mapping, which is the task of creating a model of the environment. Localization and mapping are distinct yet interrelated problems: localization presupposes knowledge of the environment map, while mapping is based on the knowledge of the robot's pose. Consequently, these two problems must be solved concurrently in an unknown environment.

In general, SLAM systems are composed of two principal components: a front-end and a back-end. The front end is concerned with perception, encompassing data fusion and feature extraction. It processes sensor data to provide information regarding the robot's motion, loop closures, and landmarks. In contrast, the back-end utilises the output from the front-end to generate final estimates of the robot's position. This back-end process draws upon tools from probability theory, optimization, and network theory.

There are numerous ways in which SLAM algorithms can be classified:

- Classification based on sensors (front-end): SLAM can be performed with various sensors, including LiDAR, sonar, cameras, IMUs.

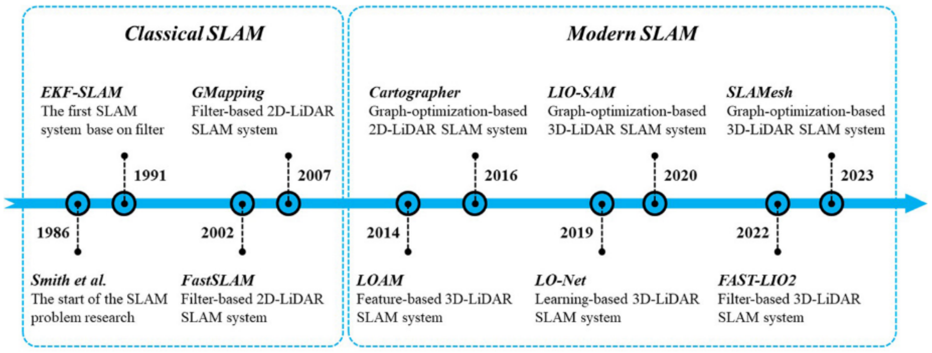
Classical SLAM
Modern SLAM
EKF-SLAM
The first SLAM system base on filter
1991
GMapping
Filter-based 2D-LiDAR SLAM system
2007
Cartographer
Graph-optimization-based 2D-LiDAR SLAM system
2016
LIO-SAM
Graph-optimization-based 3D-LiDAR SLAM system
2020
SLAMesh
Graph-optimization-based 3D-LiDAR SLAM system
2023
1986
Smith et al.
The start of the SLAM problem research
2002
FastSLAM
Filter-based 2D-LiDAR SLAM system
2014
LOAM
Feature-based 3D-LiDAR SLAM system
2019
LO-Net
Learning-based 3D-LiDAR SLAM system
2022
FAST-LIO2
Filter-based 3D-LiDAR SLAM system

Figure 2.1: Timeline of the evolution of the LiDAR SLAM algorithms, with emphasis on the most important step [10].

- Classification based on computational methods (back-end). This includes filtering-based or graph-based optimization techniques.

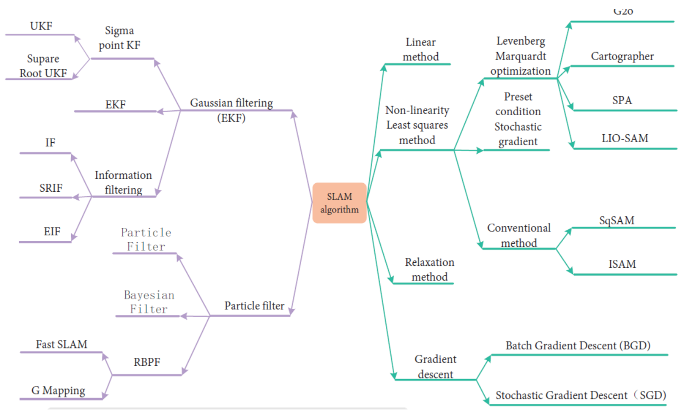

Figure 2.2: SLAM approaches taxonomy. The two main groups are traditional filtering methods (on the left) and modern optimization techniques (on the right) [11].

Figure 2.2 provides a summary of SLAM algorithms, while Figure 2.3 depicts a general SLAM framework.

Firstly, Section 2.1 contains the mathematical model of the SLAM problem. Secondly, Section 2.2 elucidates the role of the front-end in the SLAM process, with a particular emphasis on the distinctions between visual-based SLAM and LiDAR-based SLAM. Furthermore, Section 2.3 focuses on the back-end problem in a SLAM algorithm. This Section presents the current state of the art in single-agent SLAM, classifying the algorithms according to their back-end, namely filtering-based or

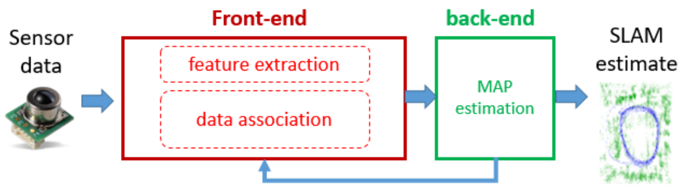

Figure 2.3: Single-robot SLAM Overview. The front-end pre-processes sensor data, while the back-end performs a MAP estimation [12].

graph-based. In conclusion, the mapping process is delineated in accordance with the various categories identified in Section 2.4.

2.1 General Formulation of the Problem

In this thesis, we consider a mobile robot navigating through an unknown environment and detecting unknown landmarks [13]. The following variables are defined at time t:

- the state vector, x_t, describes the position and orientation of the robot at time t: $\{x_1, x_2, \ldots, x_t\}$.
- the control vector, denoted by u_t, represents the input applied at time $t-1$ to move the robot from its previous state x_{t-1} to the desired state x_t at time t: $\{u_1, u_2, ...u_t\}$.
- z_t represents the measurement data obtained from exteroceptive sensors, defined as a set of landmarks observed at time t: $\{z_1, z_2, \ldots, z_t\}$
- m represents the map.

The objective of the SLAM is to estimate the value of x_t, given the control inputs $u_{1:t}$ and measurements $z_{1:t}$. A general formulation of the SLAM problem is as follows:

$$B(x_t) = p(x_t, m \mid u_{1:t}, z_{1:t}) \tag{2.1}$$

where $B(x_t)$ is the belief of the state x at time t.

This issue can be addressed in a number of ways, depending on the methodology employed for the calculation of $B(x_t)$. Accordingly, SLAM solutions can be classified into three principal categories: These include the Gaussian filter-based approach, the particle filter-based approach, and the graph optimisation-based approach. A comprehensive account of this approach, together with a detailed exposition of the associated SLAM solution, can be found in Subsection 2.3.

2.2 Front-end

The front-end is responsible for tasks such as feature extraction, odometry measurements and loop closure detection. Odometry represents a fundamental element of a SLAM system. It entails the calculation of the robot's successive movements in relation to its previous position as it traverses an environment. Commonly, the measurements are obtained through the tracking of wheel movements, the integration of data from an IMU, or the performance of geometric matching between consecutive images or laser scans. However, odometry is not a sufficient approach due to drift. Consequently, loop closure detection is crucial for monitoring a robot's trajectory, quantifying its displacement and orientation between successive time points (comparison between consecutive images or laser scans).

The ability of mobile robots to navigate effectively in both known and unknown indoor and outdoor environments depends on the reliability of their localization systems [14]. While Global Positioning System (GPS) technology provides accurate outdoor localization, it is not suitable for indoor applications. There are a number of alternative indoor localization methods, including Bluetooth, Wi-Fi and Inertial Measurement Units (IMU) sensors. However, these have limitations, including infrastructure requirements, reduced accuracy and cumulative errors.

The selection of sensors is of paramount importance in the design of the SLAM front-end. LiDAR and cameras are the most commonly used sensors, particularly in generic SLAM applications, where the environment lacks distinctive features [15].

In the context of LiDAR odometry, the utilization of scan-matching serves to establish the relative position and orientation of scans or point clouds. The deployment of LiDAR sensors in LiDAR-SLAM facilitates the acquisition of highly accurate distance measurements, conferring advantages such as precise range detection, straightforward error modelling, and dependable performance in a spectrum of environmental conditions [2].

The various LiDAR odometry methods are generally categorized in accordance with the point cloud registration techniques employed [10]:

- Point-based registration utilizes a distance-based approach to identify correspondences between the reference cloud and the target cloud.
- Distribution-based registration transforms the point cloud into a voxel grid with a continuous probability density function.
- Feature-based registration extracts geometric features from points to improve the efficiency and accuracy of registration.

Cameras play a significant role in SLAM, offering a variety of options, including monocular, binocular and RGB-D cameras, which are suitable for different environments. The principal categories of visual SLAM algorithms are visual-only, visual-inertial (comprising cameras and an inertial measurement unit, or IMU) and RGB-D. Cameras offer a number of advantages, including the provision of detailed data, affordability and a compact form factor [2]. Nevertheless, there are certain limitations, including the influence of variable lighting conditions, perspective distortion and the extraction of precise 3D data, particularly in environments characterized by inconsistent lighting and a lack of texture. Recent advances in V-SLAM have focused on the improvement of lighting models and the development of more robust feature learning methods through the application of deep learning techniques, with the objective of enhancing the system's performance and stability, particularly in scenarios characterised by changing lighting conditions. In Subsection 2.2.1 there is an overview of visual SLAM. Barròs et al. [16] offer a comprehensive review of visual-SLAM algorithms.

Additionally, there are other sensors, such as sonar and ultrasonic sensors. However, they are employed in particular environments and specialised settings, for example, underwater.

Furthermore, IMUs are employed to augment SLAM systems by measuring acceleration and angular velocity, thereby enhancing the precision of pose estimation and motion tracking [15]. These measurements are integrated with cameras or laser inputs, thereby providing resilience to issues associated with lighting changes, texture-less environments, and rapid motion. However, IMUs are susceptible to cumulative errors due to their integration over time, and therefore cannot be utilised as a standalone sensor.

According to Ahmed et al. [6], approximately 62% of the articles utilizes LiDAR, 28% employ RGB cameras, and the 19% rely on RBG-D cameras.

2.2.1 Visual SLAM

Visual-based SLAM techniques employ one or more cameras as the principal sensor, utilising 2D images as the primary source of data, as evidenced in the review of visual SLAM techniques [17]. The input data may be comprised of purely 2D images (visual-only), a combination of 2D images and IMU data (visual-inertial), or a combination of 2D images and depth data (RGB-D). In the initialization phase, global coordinates are established, and an initial map is created, which serves as the foundation for the subsequent tracking and mapping processes. The tracking phase involves the continuous estimation of the sensor's pose by matching the current image frame with the map in 2D–3D space. Mapping, in contrast, concerns the computation and extension of the 3D structure as the camera moves, with the method for calculating depth data depending on the algorithm employed.

Visual-only SLAM systems rely on processing two-dimensional images. Indeed, these algorithms are compatible with monocular or stereo cameras. The former are characterised by affordability, straightforward calibration and low power consumption. However, they are unable to estimate depth. In contrast, stereo cameras are capable of capturing depth in a single frame; however, they necessitate a larger sensor and more processing power. Visual-only SLAM can be categorized into two distinct methods: feature-based and direct.

Feature-based algorithms concentrate on the identification of keypoints across a series of images. These algorithms are more commonly utilised in the implementation of embedded systems. In environments characterised by a paucity of texture, they may encounter challenges, resulting in the generation of a sparse map.

The direct method processes raw sensor data without prior pre-processing, utilising pixel intensity values to minimise photometric error. The density of the reconstruction affects real-time performance, with dense reconstructions potentially being more computationally expensive.

The first monocular visual-only SLAM has been developed by Davison et al. and its name is MonoSLAM [18]. In the Visual-Only SLAM history, important algorithms are also PTAM [19], ORB-SLAM [20] and DTAM [21].

Conversely, SLAM systems utilising RGB-D data offer a cost-effective solution for real-time depth sensing. RGB-D sensors integrate a monocular RGB camera with a depth sensor, enabling the capture of precise depth data without the necessity for extensive pre-processing. These systems frequently employ the Iterative Closest Point (ICP) algorithm to merge depth maps and reconstruct environments. This algorithm is particularly well-suited for indoor environments due to its simplicity and ability to operate in real time. However, it requires substantial memory and power resources. An illustrative example of RGB-D SLAM is SLAM++ [22].

2.2.2 LiDAR SLAM

LiDAR, an acronym for stands for Light Detection And Ranging, it belongs to the category of Time of Flight (ToF) sensors. These sensors emit laser pulses and measure the time taken for them to return [23]. Figure 2.4 illustrates a typical LiDAR sensor.

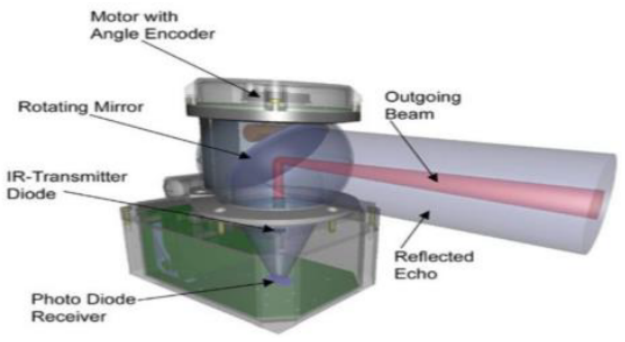

Figure 2.4: The internal structure of a LiDAR sensor. The key components are the rotating mirror, motor with angle encoder, IR-transmitter diode and photo diode receiver. The outgoing beam and reflected echo show the measure process [23].

The following equation is used to calculate the distance travelled by a light particle to and from an object:

$$D = \frac{c \cdot FT}{2} \tag{2.2}$$

where D is distance, c is speed of light, FT is flight time.

Two-dimensional (2D) LiDAR sensors employ a single axis of laser beams to capture both the X and Y coordinates. In contrast, 3D LiDAR sensors operate in a similar manner but incorporate supplementary measurements along the Z-axis, thereby facilitating the acquisition of 3D data. This third dimension is typically recorded by utilising multiple lasers set at varying angles or through longitudinal projections. Despite the enhanced accuracy and resolution offered by 3D LiDAR in comparison to 2D LiDAR, the associated cost is considerably higher. Consequently, 3D LiDAR is particularly beneficial for detailed visualisations and in-depth analysis of complex structures, such as the assessment of bend radius in technological applications.

The most commonly utilized LiDARs are mechanical devices, which offer a broad field of view (FOV) and provide coverage of up to 360 degrees [14]. This results in a horizontal panoramic scan through continuous rotation. Mechanical LiDAR can be either 2D or 3D. The 2D LiDAR sensor is employed in SLAM algorithms such as FastSLAM (2002) [24], Gmapping (2007), [25], KartoSLAM (2010) [26], HectorSLAM (2011) [27], Cartographer (2016) [28] and SLAM Toolbox (2021) [29]. In contrast, examples of 3D LiDAR applications in SLAM are LOAM (2014) [30], LeGO-LOAM (2018) [31], LIO-SAM (2020), [32], BALM (2021) [33] and F-LOAM

(2021) [34].

LiDAR is a technology that collects data on the distance, height and angle of obstacles by analysing the reflected laser light. In mechanical LiDAR, angle measurement is achieved through horizontal rotational scanning, which creates a 2D polar coordinate system.

$$\begin{cases} x = d\cos\theta \\ y = d\sin\theta, \end{cases} \tag{2.3}$$

where d is the distance to the scanned point and θ the beam angle. Mechanical LiDARs are renowned for their rapid scanning speed, resilience to light interference, and sophisticated SLAM algorithms [14]. However, despite these advantages, they have some limitations, too. These include high costs, the presence of bulky mechanical components, and sensitivity to vibration.

Conversely, a solid-state LiDAR functions without the necessity for moving mechanical components, relying on solid-state technology, instead. This renders it more cost-effective and durable than mechanical LiDARs [14]. However, it typically exhibits a reduced field of view (FOV). Recent research has led to the development of multi-channel solid-state LiDARs, which integrate data to achieve a field of view (FOV) that is comparable to that of mechanical LiDAR. There are a number of different typologies of solid-state LiDAR. Flash LiDAR (which scans the entire scene with a single flash), Phased Array LiDAR (which uses a micro-array of antennas to direct laser beams in any direction by adjusting signal timing), and MEMS LiDAR (which employs micro-electro-mechanical systems with multiple mirrors) are the three main types of solid-state LiDAR. An illustrative example of solid-state LiDAR SLAM is Livox-SLAM [35]. Table 2.1 provides a summary of the specific parameters of mechanical and solid-state LiDARs.

Traditionally, LiDAR SLAM employed mechanical LiDAR systems. However, the advent of low-cost, lightweight solid-state LiDARs has prompted the development of novel SLAM methodologies tailored to these systems. Zhou et al.[14] report a comparative analysis of SLAM algorithms for mechanical and solid-state LiDAR. The mechanical examples include LOAM [30] and LeGO-LOAM [31], while the solid-state example is Livox mapping [35]. The algorithms BALM [33] and MULLS [36] are analyzed for both mechanical and solid-state LiDAR.

Despite the advancements in LiDAR-SLAM, existing systems still encounter challenges due to the unordered, sparse, and limited information inherent in point clouds [2]. In particular, 3D SLAM presents more significant algorithmic difficulties compared to 2D SLAM, given the larger data volume, complex spatial feature

	Solid-state	**Mechanical**
Point clouds per second	$10^5 - 1.5 \cdot 10^6$	$3 \cdot 10^5 - 3.5 \cdot 10^6$
Range-finding capability	$250 - 350m$	$100 - 200m$
FOV	Horizontal $15\,\text{deg} - 120°$ / Vertical $8° - 70°$	Horizontal $360°$ / Vertical $22.5° - 105.2°$
Ranger accuracy	$2cm$	$2 - 3cm$
Price	$599 - 1200$\$	$2400 - 150000$\$
Weight	$0.7 - 1.5Kg$	$0.8 - 14Kg$

Table 2.1: Technical parameters of Solid State LiDAR and Mechanical LiDAR [14].

matching, and higher positioning accuracy requirements. However, 3D SLAM remains a crucial area of research for several reasons, including the ability to mitigate point loss in non-localised environments using 3D positional data and the capacity to perform relatively well in complex terrains.

An example of front-end agnostic modular LiDAR SLAM is SC-LiDAR-SLAM. The SLAM algorithm presented in [37] is constituted by a triad of elements: odometry, place recognition (front-end) and pose-graph optimisation. The SC-LiDAR-SLAM algorithm is structured into three main components: keyframe selection, pose-graph construction, and loop detector. The initial pose graph is constructed using a stream of input data, and the Loop Detector subsequently adds a set of constraints to the pose graph through the use of ICP (Iterative Closest Point).

2.2.3 Multi-sensor SLAM

The growing demand for SLAM has brought to light shortcomings in single-sensor SLAM systems, particularly those that rely exclusively on LiDAR sensors [2]. These limitations have their origins in a number of factors, including low vertical resolution, sparse point clouds, sensitivity to movement, degradation issues and challenges in SLAM. To overcome these challenges and enhance SLAM performance in terms of both speed and accuracy, it is crucial to adopt multi-sensor-aided SLAM approaches. The combination of data from multiple sensors, including LiDAR, cameras, inertial measurement units (IMUs), and odometry, enables the utilisation of complementary information, thereby enhancing mapping accuracy, robustness, and responsiveness across a broader range of environmental conditions. This integration facilitates the handling of dynamic environments, the compensation for the limitations of individual sensors, and the delivery of more reliable and efficient performance, collectively enhancing the overall performance of multi-sensor SLAM.

Inertial-LiDAR SLAM

The concept of Inertial Measurement Units (IMUs) was initially designed to combine multiple inertial measurements between key frames into a single relative motion constraint [2]. IMUs comprise a gyroscope and accelerometer, and provide inertial data. The formalisation of preintegration theory further enhanced this by introducing rotation noise, which facilitated the development of incremental smoothing algorithms.

The estimation of joint poses using LiDAR and IMU sensors can be categorised into two approaches: loosely coupled fusion and tightly coupled fusion. In the first approach, employed in LOAM, LiDAR and IMU estimates are treated separately. This approach is efficient, but with lower accuracy, and thus is more suited to real-time applications. In the second approach, used in LIO-SAM (explained in Section 2.3), there is a direct integration of LiDAR and IMU measurements.

Visual-inertial SLAM

The visual-inertial SLAM approach integrates visual data with inertial measurements (obtained from the IMU) to estimate both the configuration of the environment and the orientation of the sensor [17]. The integration of an IMU serves to enhance the accuracy of the system, but it also increases the complexity of the initialisation phase. Furthermore, visual-inertial SLAM algorithms are classified according to whether they are loosely or tightly coupled.

Visual-LiDAR SLAM

Vision-based methods are particularly effective in recognising scenes and textures, while LiDAR provides accurate distance measurements. The combination of these sensor types in SLAM systems allows for the utilisation of complementary data, with vision providing detail and texture, LiDAR offering spatial accuracy, and IMU facilitating scale and attitude recovery. Binocular camera-LiDAR setups use visual odometry to estimate motion, which is then refined through LiDAR frame matching. V-LOAM [38] employs IMU motion prediction and visual-inertial fusion for motion estimation, followed by LiDAR scan matching to enhance precision.

2.2.4 Loop Closure Detection

Loop closure detection is the ability of a robot to recognise when it has returned to a previously visited scene. This allows the robot to match the currently generated one with the previously generated map, effectively closing the loop [11]. This process has a great impact, as a successful loop detection can markedly reduce the accumulation of errors, thereby enabling the robot to avoid obstacles with greater accuracy and efficiency. It is therefore evident that loop detection is a crucial aspect of mapping extensive areas or constructing maps in large environments. Errors in SLAM typically originate from three primary sources: observation errors, odometry errors, and errors stemming from incorrect data associations. According to Yue et al. [10], loop closure detection typically involves two steps:

1. Position recognition, where the system identifies a point in the database that corresponds to the current observation.

2. Pose graph optimization, which adjusts the estimated pose to correct for errors when a loop is detected.

At present, two predominant methodologies exist for loop closure detection: bag-of-words models and techniques that identify potential frames based on disparity and keyframe link relationships [11]. The technique of loop closure detection is dependent on the type of sensor employed. In the case of LiDAR SLAM, the reliance is on geometric features derived from point clouds. ICP is a technique that matches point clouds from different poses. Visual SLAM is based on the analysis of visual features. This technique employs feature-based methods, such as SIFT, SURF and ORB, which are used to identify and match features across different frames, thus enabling the closure of the loop.

Lajoie et al. [39] includes loop closure detection in the front-end, while other authors include this phase in the back-end.

2.3 Back-end

The back-end component is the responsible for optimizing and refining the map and the robot's trajectory based on the data collected by the front-end. The back-end is charged with the processing of information, with the objective of ensuring global consistency and improving the accuracy of the estimated map and poses. The back-end methods can be classified into two principal categories: filter-based approaches, which include the Extended Kalman Filter and the Particle Filter, and smoothing-based approaches, which encompass graph optimisations and Bundle Adjustment [40].

Extended Kalman Filter (EKF) and particle filter-based SLAM techniques are more frequently employed than pose-graph or graph-based SLAM approaches. The former constitute 54% of the total, while the latter account for 45% [6].

2.3.1 Filters-based SLAM

Bayesian filtering approaches are frequently utilized in real-time scenarios where the objective is to estimate the robot's current pose, with previous poses effectively disregarded at each time step [39]. In this approach, the estimation of the robot's state at a specific moment is contingent upon the state estimation from the previous moment and the most recent measurements.

Two distinct categories of filters can be identified: parametric filters (including Gaussian filters), which utilize a pair of values (mean and covariance), and non-parametric filters (including particle filters), which employ a set of randomly sampled state particles [13].

By assuming the Bayesian full probability rule and the Markov assumption, the mathematical model for the localization task can be derived using a Bayesian filter method.

$$\begin{aligned} B(x_t) &= p(x_t \mid z_{1:t}, u_{1:t}) \\ &= p(x_t \mid z_{1:t-1}, z_t, u_{1:t}) \\ &= \eta \cdot p(z_t \mid x_t, z_{1:t-1}, u_{1:t}) \cdot p(x_t \mid z_{1:t-1}, u_{1:t}) \\ &= \eta \cdot p(z_t \mid x_t) \cdot p(x_t \mid z_{1:t-1}, u_{1:t}) \\ &= \eta \cdot p(z_t \mid x_t) \cdot \overline{B}(x_t), \end{aligned} \tag{2.4}$$

with $\eta = \frac{1}{p(z_t \mid z_{1:t-1}, u_{1:t})}$ a normalized constant, and

$$\begin{aligned} \overline{B(x_t)} &= \int p(x_t \mid z_{1:t-1}, u_{1:t}) \cdot p(x_{t-1} \mid z_{1:t-1}, u_{1:t}) \, dx_{t-1} \\ &= \int p(x_t \mid x_{t-1}, u_t) \cdot B(x_{t-1}) \, dx_{t-1} \end{aligned} \tag{2.5}$$

the prior of the robot pose at time t.

Thus, the Bayesian algorithm can be divided into two steps:

1. prediction step (the algorithm estimates the state x_t)

$$\overline{B(x_t)} = \int p(x_t|x_{t-1}, u_t) \cdot B(x_{t-1}) \, dx_{t-1}, \tag{2.6}$$

2. update step (the algorithm corrects the estimated error)

$$B(x_t) = \eta \cdot p(z_t|x_t) \cdot \overline{B(x_t)}. \tag{2.7}$$

Gaussian filter-based

EKF methods are based on Gaussian filters (in particular, the Kalman filter) and local linear approximation, and are designed to address the issue of nonlinearities [15]. This kind of approach assumes that the robot's estimated state can be represented by a multivariate Gaussian distribution. It employs parameters such as the mean and covariance to characterize the robot's state, and the Gaussian filter updates these parameters in order to predict the robot's future state [13].

This kind of approach yields favourable outcomes with respect to the estimation of the robot's pose and the construction of the map. However, it may encounter challenges in the presence of high noise levels, which could result in inconsistencies. Furthermore, a high degree of uncertainty leads to an inaccurate estimation of the mean and covariance in the Gaussian function, resulting in a potential deviation between the estimated pose and the actual pose. Additionally, the computational demands of EKF-SLAM increase as the number of feature points increase. Despite these potential limitations, EKF offers significant advantages, such as the ability to directly access the covariance matrix without additional computation, which is particularly useful for tasks like feature tracking or active exploration, and its reliability.

The mathematical model of the Kalman filter takes into account the following parameters:

- $A_t \in \mathbf{R}^{\mathbf{n}\times\mathbf{n}}$ describes the evolution from state $t-1$ to t.
- $B_t \in \mathbf{R}^{\mathbf{n}\times\mathbf{m}}$ represent the control evolution from state $t-1$ to t.
- $C_t \in \mathbf{R}^{\mathbf{k}\times\mathbf{n}}$ describes the mapping of the state x_t to an observation z_t
- The random variables ϵ_t, δ_t represent the process and measurement noise, respectively. These are assumed to be independent and multivariate normally distributed with covariance R_t and Q_t, respectively.
- The mean and covariance of the state vector x at time t are represented by μ_t and Σ_t, respectively.

The Kalman filter algorithm can be divided into the following three principals steps:

1. The prediction step, which corresponds to (2.6):

$$\overline{\mu_t} = A_t\mu_{t-1} + B_t u_t, \tag{2.8}$$

$$\overline{\Sigma_t} = A_t\Sigma_{t-1}A_t^T + R_t \tag{2.9}$$

2. Optimal gain computation:

$$K_t = \overline{\Sigma_t} C_t^T (C_t \overline{\Sigma_t} C_t^T + Q_t)^{-1} \tag{2.10}$$

3. The update step, which corresponds to (2.7):

$$\mu_t = \overline{\mu_t} + K_t(z_t - C_t \overline{\mu_t}) \tag{2.11}$$
$$\Sigma_t = (I - K_t C_t)\overline{\Sigma_t} \tag{2.12}$$

Nevertheless, the Kalman filter relies on the assumption of linearity in both the state and measurement equations. Consequently, the Extended Kalman filter, which accounts for nonlinearity in both equations, has been developed:

$$x_t = g(u_t, x_{t-1}) + \epsilon_t \tag{2.13}$$
$$z_t = h(x_t) + \delta_t \tag{2.14}$$

where (2.13) is the nonlinear state transition equation, whereas (2.14) is the nonlinear measurement equation.

Particle filter-based

The particle filter (PF) apporach operates by representing potential position as a set of discrete particles [15]. In a manner analogous to the Gaussian filter, the particle filter-based approach recursively estimates the robot's current state (x_t) based on its previous state (x_{t-1}), thereby adhering to the fundamental principles of Bayesian filtering [13]. The state of each particle is predicted for the subsequent time step by utilizing odometry data. The algorithm then evaluates the likelihood of each particle based on observed landmarks. Subsequently, the algorithm assesses the probability of each particle based on the observed landmarks. The system performs resampling in accordance with the assigned weight, whereby particles with a higher weight are more likely to be sampled. This process continues recursively, with particles becoming increasingly concentrated around the true position of the robot. This approach can be employed in nonlinear motion models, as it does not assume linearity. Additionally, the computational complexity of this approach is dependent on the number of particles.

The Rao-Blackwellized Particle Filters (RBPF) approach, as detailed in [41], combines particle filters with the Extended Kalman Filter [39]. This approach employs the Rao-Blackwell factorization of $B(x_t)$, which divides the localization and mapping problems into two new distinct, new problems.

$$\begin{aligned} B(x_t) &= p(x_t, m \mid u_{1:t}, z_{1:t}) \\ &= p(m \mid u_{1:t}, z_{1:t}) \cdot p(x_t, \mid u_{1:t}, z_{1:t}). \end{aligned} \tag{2.15}$$

In this algorithm, particles are used to the potential states of the robot. This hybrid method enhances the efficiency of the SLAM process and improves the overall accuracy of the state estimates. Nevertheless, the generation of accurate maps necessitates the utilization of a considerable number of particles. Accordingly, in order to identify an optimal balance between accuracy and computational complexity, the most prevalent algorithm employed in particle filters is Sampling Importance Resampling (SIR), which is comprised of four distinct stages: prediction, correction, resampling, and map estimation [42]. One of the most frequently utilized RBPF-SLAM algorithms is Gmapping [25], which employs an adaptive sampling strategy that reduces the number of required particles, and determines whether to perform resampling using a threshold [13]. Gmapping is renowned for its efficiency and effectiveness in real-time solutions. These algorithm employs a 2D mechanical LiDAR as a front-end.

2.3.2 Smoothing-based SLAM

A Smoothing-based method (or Graph optimization-based) SLAM algorithm employs a pose graph, wherein each node represents the robot's pose at a specific time and the edges between nodes represent spatial constraints based on sensor data (such as odometry and landmark observation) [15]. This algorithms operate by optimizing the entire trajectory of the robot, rather than just the current pose. They take into account all measurements and poses simultaneously in order to create a globally consistent trajectory of the robot. Thus, the goal of the algorithm is to find the configuration of poses (and map) that best satisfies all the constraints: this is framed as a nonlinear least square optimization problem.

According to [39] and [43], the prevailing approach to smoothing-based SLAM is the Maximum A Posteriori (MAP) estimation. For the purposes of this discussion, we will assume that X represents both the landmarks (that is to say, the map) and the robot's state. The measurements acquired by the moving robot can be expressed as a set, denoted by $Z = \{z_k : k = 1, \dots, m\}$. Every measurement can be expressed as a function of the state variables X, as $z_k = h_k(X_k) + \epsilon_k$, where X_k is subset of X, $h_k(\cdot)$ is the observation model (a known function), and ϵ_k is the measurement noise. The objective is to maximize the probability distribution $p(X|Z)$, that is, the belief over X given a measurement:

$$\begin{aligned} X^* &= \arg\max_X p(X|Z) \\ &= \arg\max_X p(Z|X) \cdot p(X). \end{aligned} \tag{2.16}$$

In this context, the likelihood of the measurements, denoted by Z, given a particular state X, is represented by the function $p(Z|X)$. The prior distribution over the robot's motion state, represented by the function $p(X)$, is constant if there are no a priori knowledge is available.

If the measurements are independent (i.e., the noises are uncorrelated),

$$\begin{aligned} X^* &= \arg\max_X p(X) \prod_{k=1}^{m} p(z_k|X) \\ &= \arg\max_X p(X) \prod_{k=1}^{m} p(z_k|X_k). \end{aligned} \tag{2.17}$$

The probability distribution $p(z_k|X_k)$ and $p(X)$ are factors, i.e., they are probabilistic constraints on a subset of nodes.

If we assume that ϵ_k is zero-mean, then:

$$p(z_k|X_k) \propto exp(-\frac{1}{2}||h_k(X_k) - z_k||^2_{\Omega_k}) \tag{2.18}$$

where Ω_k is the inverse of the covariance matrix.
In conclusion, the MAP estimate is a nonlinear least square problem:

$$\begin{aligned} X^* &= \arg\min_X - \log(p(X) \prod_{k=1}^{m} p(z_k|X_k)) \\ &= \arg\min_X \sum_{k=0}^{m} ||h_k(X_k) - z_k||^2_{\Omega_K} \end{aligned} \tag{2.19}$$

The solution to this optimization problem is typically achieved through the application of Gauss-Newton method, Levenberg-Marquardt method, or Ceres solver. Another mathematical model of the optimization graph based SLAM is provided by [13].

The constraints in the graph are calculated as residual errors, and the sum of these residual errors can be expressed as follows:

$$\begin{aligned} F(x_t, m) = x_0^T \omega x_0 + \sum_t [x_t - g(u_t, x_{t-1})]^T R_t^{-1} [x_t - g(u_t, x_{t-1})] + \\ + \sum_t [z_t - h(m, x_t)]^T Q_t^{-1} [z_t - h(m, x_t)], \end{aligned} \tag{2.20}$$

where:

- $x_0^T \omega x_0$ is an anchor constraint that sets the robot initial position to $(0,0,0)^T$.

- $g(u_t, x_{t-1})$ is a nonlinear state transition function that captures the robot's motion dynamics from x_{t-1} to x_t.
- $h(m, x_t)$ is a nonlinear measurement function that links the robot's pose to observed landmarks.

Figure 2.5 illustrates the fundamental principles of graph optimization.

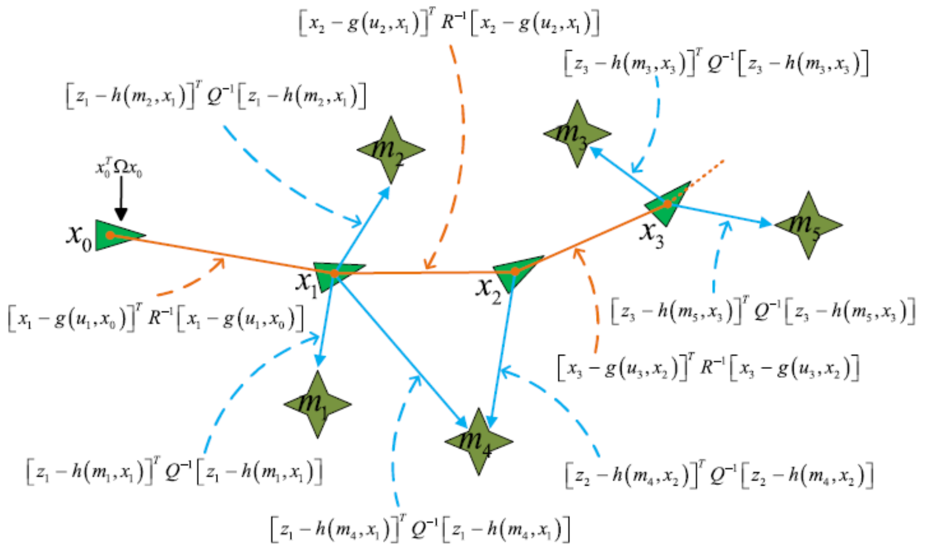
$[x_2 - g(u_2,x_1)]^T R^{-1}[x_2 - g(u_2,x_1)]$
$[z_3 - h(m_3,x_3)]^T Q^{-1}[z_3 - h(m_3,x_3)]$
$[z_1 - h(m_2,x_1)]^T Q^{-1}[z_1 - h(m_2,x_1)]$
$x_0^T \Omega x_0$
x_0
m_2
m_3
m_5
x_1
x_2
x_3
$[x_1 - g(u_1,x_0)]^T R^{-1}[x_1 - g(u_1,x_0)]$
$[z_3 - h(m_5,x_3)]^T Q^{-1}[z_3 - h(m_5,x_3)]$
$[x_3 - g(u_3,x_2)]^T R^{-1}[x_3 - g(u_3,x_2)]$
m_1
m_4
$[z_1 - h(m_1,x_1)]^T Q^{-1}[z_1 - h(m_1,x_1)]$
$[z_2 - h(m_4,x_2)]^T Q^{-1}[z_2 - h(m_4,x_2)]$
$[z_1 - h(m_4,x_1)]^T Q^{-1}[z_1 - h(m_4,x_1)]$

Figure 2.5: Graph-based optimization - The poses of the robot are represented by triangles, and the landmarks are represented by stars. These poses and landmarks serve as nodes in the graph. Solid lines in the graph, either blue (measurements) or orange (controls), denote edges that impose constraints on the SLAM system [13].

Furthermore, in this formulation, the optimal values x_t^*, m^*, can be obtained by minimizing the objective function $F(x_t, m)$ using nonlinear optimization techniques, such as the Gauss-Newton algorithm and the Levenberg-Marquardt algorithm. To reduce the residual errors, these techniques update iteratively the variables.

At the present time, the graph-based approach is the most commonly used. KartoSLAM [26] addresses the sparse linear system through non iterative Cholesky matrix decompositions [44]. The systems generates a front-end graph through the scanning and matching of LiDAR data, while simultaneously detecting loop closures. The back-end then computes the nonlinear optimization, which updates the poses. The algorithms introduces an efficient optimization technique called Sparse Pose Adjustment (SPA), which facilitates scan matching and loop closure detection. It is crucial to maintain a balance between computational efficiency and precision. In fact, KartoSLAM offers several advantages, including an accurate solution, robustness, fast convergence and complete non-linearity. Figure 2.6 depicts the KartoSLAM framework.

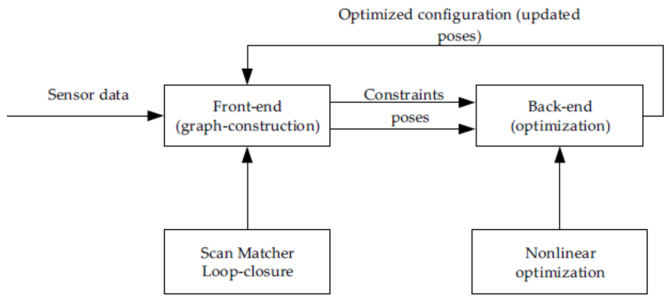
Optimized configuration (updated poses)
Sensor data
Front-end (graph-construction)
Constraints
poses
Back-end (optimization)
Scan Matcher Loop-closure
Nonlinear optimization

Figure 2.6: Karto SLAM framework [44].

Another optimization-based algorithm is LiDAR Odometry and Mapping (LOAM) [30]. The system comprises a three-dimensional inertial measurement unit (IMU) and a three-dimensional mechanical light detection and ranging (LiDAR) sensor. The principal characteristic of this system is its capacity for real-time 3D localization and mapping. Consequently, LOAM is frequently employed in contexts where the generation of precise 3D maps is of paramount importance. Additionally, there are enhanced versions of LOAM: V-LOAM [38] and F-LOAM [34]. The first approach integrates visual and LiDAR data. On the other hand, the second one uses a non-iterative two-stage distortion compensation technique, thereby offering enhanced computational efficiency and a more precise framework for real-time mapping. Moreover, LeGO-LOAM [31] builds upon the LOAM algorithm by refining the extraction of feature points and optimizing the back-end processing in order to reduce the computational time. The core components of the algorithm are segmentation, feature point extraction, LiDAR odometry and mapping. Additionally, Livox mapping [35] is an advanced version of the LOAM algorithm developed for solid-state LiDAR, as discussed in Section 2.2. It enhances accuracy and robustness by considering low-level physical characteristics of the LiDAR sensor during front-end processing.

A significant SLAM method employed in expansive indoor settings is Cartographer, developed by Google [28]. It is a real-time loop closure in 2D LiDAR SLAM, designed to operate under constrained computational resources. Cartographer effectively balances the computational cost and accuracy of the system. It is efficient, operates in real-time on limited hardware, and is suitable for large-scale indoor mapping. This is accomplished through a branch-and-bound methodology that effectively computes the transformation from the scan to the submap. The estimation of relative pose changes is a common application of scan-to-scan matching in laser-based SLAM techniques (2.2.2). Nevertheless, relying exclusively on scan-to-scan matching may result in the rapid accumulation of errors over time. In the event that an up-to-date grid map is a prerequisite, submaps can be generated and refreshed solely when necessary. This method is referred to as scan-to-submap. A submap is defined as a grid of probability values, representing the likelihood that a given area is occupied. Cartographer employs a combination of local and global SLAM techniques to optimize the LiDAR scans. Local SLAM utilises odometry and IMU data to compute the trajectory and estimate the robot's pose, while global SLAM handles loop-closure detection and optimization. A submap is only included in the loop closure process once it has been fully populated with scans. The optimization problem for loop closure is formulated as a non-linear least squares problem, whereby the system continuously optimizes the poses of both the scans and the submaps. Figure 2.7 provides a representation of the Cartographer framework.

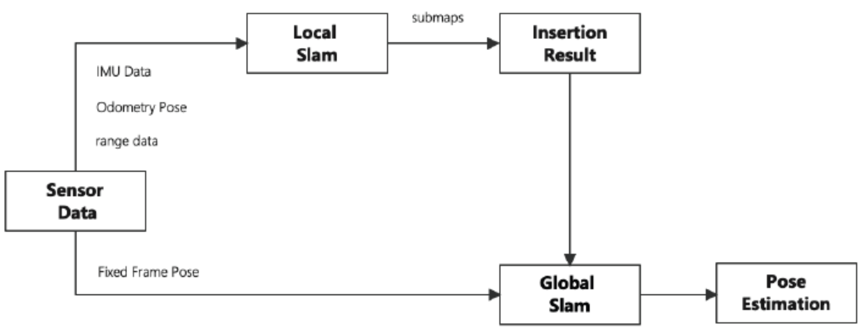

Figure 2.7: Cartographer framework [2].

Bundle adjustment is typically employed in visual SLAM to mitigate the effects of drift. Conversely, bundle adjustment with local mapping (BALM) [33] is a B.A. method tailored for LiDAR, utilizing local mapping techniques. This approach relies on point-to-line and point-to-plane correspondences, diverging from the point-to-point correspondences commonly observed in visual SLAM.

Multi-level LiDAR SLAM (MULLS) [36]) is a three-dimensional LiDAR SLAM system designed to accommodate a range of LiDAR specifications in complex environments (mulls). The system employs a single LiDAR sensor.

Modern solvers, such as g2o ([45]) or Ceres ([46]), exploit the sparsity of the pose graph, whereby only a small subset of the poses are directly connected, thus accelerating the optimisation process. In real-time operations, there is an incremental version of smoothing algorithms, namely iSAM ([47]). This approach updates the solution incrementally as new data arrive, rather than re-solving the entire problem from scratch.

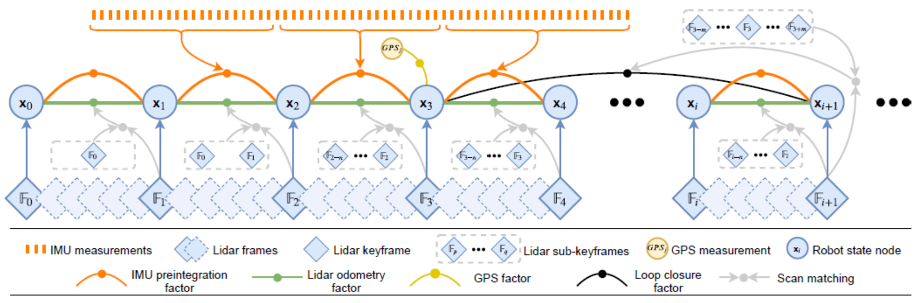

Figure 2.8: The system structure of LIO-SAM. Four types of factors are introduced to construct the factor graph: (a) IMU preintegration factor, (b) LiDAR odometry factor, (c) GPS factor, and (d) loop closure factor [32].

As previously outlined in Subsection 2.2.3, tightly-coupled LiDAR inertial odometry via smoothing and mapping (LIO-SAM) [32] represents an algorithmic approach to real-time SLAM involving the integration of diverse sensors. In contrast to loosely-coupled fusion algorithms, tightly-coupled fusion methods engage directly with the sensors in the optimization process. To perform SLAM, the algorithm integrates data from LiDAR, IMU and, where available, GPS. LIO-SAM constructs a factor graph model to integrate multiple sensor inputs into a unified optimization framework, thereby enhancing the accuracy and robustness of trajectory estimation. The algorithm is structured in four distinct phases: pre-integration of the IMU, LiDAR odometry, construction of the factor graph and optimisation. The data obtained from the IMU is used to provide estimates of motion, which serve to de-skew the LiDAR point clouds. The pre-integration process assists in estimating the robot's motion between successive LiDAR scans, a result that is further refined during the graph optimisation stage. Conversely, LIO-SAM extracts pertinent features from LiDAR point clouds and identifies the keyframe. A scan matching procedure is conducted between the LiDAR keyframes with the objective of estimating the robot's relative motion. The key innovation of LIO-SAM is the utilization of a factor graph, which models the entire system's state as nodes and sensor measurements as constraints (i.e., factors). This encompasses IMU pre-integration, LiDAR odometry, GPS and loop closure. Eventually, when the factors are added to graph, the system executes the incremental smoothing using iSAM2 [48]. Figure 2.8 illustrates the LIO-SAM process.

In conclusion, the examination of the entire trajectory demonstrates that smoothing-based SLAM algorithms reduce errors. Furthermore, they provide a global, consistent map and flexibility.

2.4 Mapping

According to Chen et al. [11], the mapping process can be categorized into five distinct functions: positioning, navigation, obstacle avoidance, reconstruction and interaction. In addition, there are three principal categories of map: sparse, dense and semantic. Sparse maps are constituted by a minimal set of points and lines. The aforementioned methods utilize minimal memory and can be generated in real time. However, they are not appropriate for complex environments, such as navigation and obstacle avoidance. Conversely, dense map models encompass all visible elements, yet demand greater computational resources and are unsuitable for real-time applications. Ultimately, semantic maps employ semantic segmentation to partition images into visual, object, and concept layers. They are utilized in SLAM systems for autonomous driving. These maps facilitate enhanced accuracy, real-time performance, safety, and robustness in vehicle localization in SLAM applications.

Chapter 3

Active Collaborative SLAM

This chapter addresses a more intricate problem than that of single-agent SLAM, namely collaborative SLAM, which is discussed in Section 3.1. Here, a state-of-the-art analysis is provided, along with the mathematical formulation. In section 3.2, the problem of active SLAM is presented. In this case, the robot actively decides its path, which introduces complications to the classic SLAM problem but also offers significant advantages. The paragraph concludes with a discussion of the combined problem, which is presented as the Active Collaborative SLAM.

3.1 Collaborative SLAM

In recent years, the field of SLAM methods for single robots has witnessed a significant advancement, culminating in the development of several highly sophisticated systems [15]. However, a single robot map may accumulate several errors over time. Furthermore, mapping an environment performed by one agent could take a considerable amount of time. Consequently, researchers developed collaborative (or cooperative) SLAM techniques for multi-robot systems. Through communication and cooperation, multiple robots work together to map the environment. Additionally, a single robot operates from a local perspective, while C-SLAM requires maintaining a global perspective across all robots [12].

The utilization of two or more robots to perform simultaneous SLAM has been demonstrated to enhance the quality of the result, improve the robustness and efficiency of the system. However, C-SLAM introduces several challenges, such as map overlays and exchanging data between agents with limited communication bandwidth [12]. Lazaro et al. [49] present a multi-robot SLAM method that has been specifically developed to address the communication and computational challenges inherent to the technology. It employs condensed measurements to facilitate the exchange of map information between the robots. These measurements can effectively compress significant areas of a map into minimal data, thereby enhancing the robustness and efficiency of the system.

In their study, Krinkin et al. [50] delineate the numerous advantages of multi-agent SLAM. The environment can be explored more rapidly, as the platforms are capable of moving in different directions, thereby increasing efficiency. In the event of a malfunction in one platform, the others can continue the mission, thus enhancing resilience. When two platforms follow intersecting paths, they can enhance the accuracy of the map and their trajectories by merging their observations, thereby reducing measurement errors and improving results.

According to Filatov et al. [51], there are multiple ways to classify multi-agent SLAM algorithms. One potential approach is to group algorithms based on their treatment of data. This could include considerations such as communication channels, data sharing (if the data are raw or processed), data distribution (centralized, decentralized or distributed), and data processing (back-end algorithm, such as EKF). Another avenue for categorisation is based on the size of the team, communication topology, and communication range, as seen in C-SLAM approaches.

In a centralized approach, the estimation problem is solved from a global perspective [39]. An agent is a mobile sensor that gathers data from its surrounding environment and subsequently transmits it to a central server (with or without preliminary processing). Furthermore, the core of the system is equipped with a comprehensive view and complete knowledge of the entire team data. In the event of the server becoming unavailable or out of range, the entire communication system will fail, thereby rendering this topology highly vulnerable to disruption [6]. Consequently, the principal issues associated with this approach are the potential for a single point of failure and the communication constraints that may arise when the number of team members is increased. Furthermore, the use of a SLAM with a large number of robots in the team could prove to be impractical. As outlined in [11], examples of Centralized multi-agent SLAM are PTAMM [52], CSFM [53], CCM-SLAM [54], CVI-SLAM [55] and CVIDS [56].

On the other hand, in a decentralized approach, each agent has access only to its own data, and there is no central server [39]. A robot constructs a local map by integrating its own data with the partial information obtained from neighbouring robots, in an iterative process. Consequently, each robot develops its own local solution, which gradually converges towards a global, consistent outcome. This approach avoids the potential for a single point of failure and reduces communication requirements, though it is more complex than the other. Indeed, the system necessitates high bandwidth and augmented network traffic, as each robot is required to communicate with the others [12]. Furthermore, each robot needs more power. In addition, Filatov et al. [51] report that in decentralized algorithms, the relative orientation of the agent may be either known or unknown (i.e., there are no assumptions regarding the initial position of the agents). In this latter case, the performance of the system is enhanced when applied to real-life problems. Approximately half of the selected articles employ this topology, as reported by Ahmed et al. [6]. Co-SLAM [57], DOOR-SLAM [3], LAMP 2.0 [58], VIR-SLAM [59], SWARM-SLAM [60], mrg-slam [4] , DCL-SLAM [61], DPGO [62] and DiSCo-SLAM [63] are examples of decentralized approaches.

DCL-SLAM represents a case study of a decentralized back-end system. The framework is modularized into three main parts, namely a single-agent front-end, distributed loop closure and distributed back-end. The latter exploits a front-end agnostic framework, whereby the front-end is not a limiting factor in the framework's functionality. The distributed back-end is responsible for gathering odometry, intra-robot loop and inter-robot loop measurements. The DCL-SLAM back-end is composed of two submodules: the outlier rejection module, which identifies and discards false positive loop closures, and the Pose Graph Optimisation module. The latter estimates the trajectories of the robots by solving a maximum likelihood problem.

$$x^* = \arg\max_x \prod \phi(x) \tag{3.1}$$

where $\phi(x) = \psi(z_{a_i b_j}|x)$ with $x = [x_\alpha, x_\beta, \ldots]$ trajectory of robots and $z_{a_i b_j}$ measurements. The optimal trajectory is reached using a two-stage DGS method to obtain consensus on the pose graph.

DOOR-SLAM (Distributed, Online, Outlier Resilient SLAM) is distinguished by its reliance on Stereo Visual Odometry (a single agent front-end), Distributed Loop Closure Detection (also a single agent front-end), Distributed outlier rejection (a distributed back-end), and Distributed Pose Graph Optimization (a distributed back-end).

DPGO (Distributed Pose Graph Optimization) proposes a distributed back-end comprising the implementation of a distributed pose graph optimization, which serves to reduce communication costs and accelerate the convergence speed of the algorithm.

The SWARM-SLAM (Sparse Decentralized Collaborative SLAM) algorithm is a relatively recent development, having been introduced in 2024. This approach is compatible with a range of sensors, necessitates minimal communication, proposes a selection of candidate inter-loop closures based on pose sparse graph theory and is fully decentralized. The front-end is composed of two types of loop closure detection: intra-robot and sparse inter-robot. The former is organised into local matching and global matching, while the latter is a novel approach that aims to approximate the complete graph with fewer edges by removing redundant edges that do not provide new information during the estimation process. This technique is particularly useful in long-term operations.

Additionally, the C-SLAM system can also be distributed [39]. This concept is distinct from the other two, as the adjectives centralised and decentralised indicate the presence or absence of a central server, respectively. The term 'distributed' implies that the computational load is shared among different robots. It should be noted that both the centralized and decentralized approaches can be distributed. A system can be considered both centralized and distributed if the central node is responsible for merging maps, while each robot performs a portion of the computation. This allows the robots to contribute to the overall process beyond their role as sensors. A review of the literature by Ahmed et al. revealed that the distributed approach was utilized in approximately 62% of the articles [6].

The primary classification is based on two distinct approaches: those that adapt single-agent SLAM algorithms for use in a multi-agent scenario and those that are designed from the outset for use in a multi-robot context. In terms of the back-end, there are three principal multi-agent laser SLAM approaches: feature-based, multi-hypotheses and graph-based. Extending feature-based SLAM and multi-hypothesis to a multi-agent scenario presents significant challenges. In contrast, graph-based SLAM is relatively straightforward. Updating a graph-based map in a multi-agent approach is not substantially different from the single-agent approach. A new observation can be added with a new node or by updating a previous one. However, the map merging process can be complex. In fact, the graph-based SLAM is the preferred method in almost 68% of multi-agent implementation [6].

A Collaborative SLAM system comprises two principal components: the front-end (Section 3.1.2) and back-end (Section 3.1.3) [64]. The front-end component encompasses the processes of data collection, landmark estimation, and loop-closure detection. In contrast, the back-end component, which includes bundle adjustment and pose graph optimization, utilises the front-end data to estimate the robot states and the map.

Saeedi et al. [7] investigate the potential challenges associated with multi-agent

SLAM. One of the key difficulties is the integration of local maps generated by each robot, as the necessary transformations for alignment are frequently unknown. This problem is closely linked to the uncertainty in the relative poses of the robots, which is influenced by model errors and sensor noise. To address this uncertainty, maps and poses require coordinated updates.

A further challenge is posed by direct observations between robots, which have the potential to enhance pose estimation when they are within each other's line of sight. Furthermore, the issue of loop closure represents a significant challenge that necessitates the collaborative efforts of multiple robots. The computational complexity of multi-robot SLAM represents a significant challenge, as algorithms must be designed to operate efficiently in real-time, particularly when robots are required to exchange information in environments with limited communication capabilities. The use of heterogeneous robots with different sensors can facilitate the generation of more comprehensive environmental maps; however, the integration of data from diverse sources represents a significant challenge. Another crucial issue pertains to the synchronization of data, which must be accurate at both the local (robot-level) and global (system-level) scales. Finally, performance measurement is essential for evaluating the reliability of the SLAM system, particularly when robots are required to perform autonomous tasks.

3.1.1 Problem Formulation

A simple formulation of the multi-agent SLAM is the generalization of what illustrated in Section 2.1, as explained in [7]. For two robots, a and b, the formulation is:

$$p(x^a_{1:t}, x^b_{1:t}, m_t | z^a_{1:t}, z^b_{1:t}, u^a_{1:t}, u^b_{1:t}, x^a_0, x^b_0) \tag{3.2}$$

On the other hand, Lajoie et al. [39] report that collaborative SLAM problem may be regarded as an extension of the single-robot SLAM maximum a posteriori (MAP) problem. We consider a scenario involving two robots, designated A and B. Let us define the state variables to be estimated for robots A and B as X_A and X_B, respectively. Conversely, the sets of measurements collected independently by robots A and B, respectively, are denoted as Z_A and Z_B. The set of inter-robot measurements, Z_{AB}, links the maps of both robots and contains relative pose estimates between one pose of robot A and one pose of robot B in their respective trajectories.

The solution of the problem (X^*_A, X^*_B) is:

$$\begin{aligned}(X_A^*, X_B^*) &= \arg\max_{X_A, X_B} p(X_A, X_B | Z_A, Z_B, Z_{AB}) \\ &= \arg\max_{X_A, X_B} p(Z_A, Z_B, Z_{AB} | X_A, X_B) \cdot p(X_A, X_B)\end{aligned} \tag{3.3}$$

In the absence of a known starting location and orientation for the robots, the initial estimation of the robots' states, represented by the probability density function p(X_A, X_B), is not determinable [39]. This results in an infinite number of potential initial alignments between the trajectories of multiple robots. In the absence of a prior distribution, the C-SLAM problem can be reduced to a Maximum Likelihood Estimation (MLE) problem.

$$(X_A^*, X_B^*) = \arg\max_{X_A, X_B} p(Z_A, Z_B, Z_{AB} | X_A, X_B) \tag{3.4}$$

3.1.2 Front-end

In C-SLAM, the front-end plays a pivotal role in the collection and processing of sensory data from multiple robots engaged in collaborative mapping of unknown environments. The C-SLAM front-end is designed to achieve the same objectives as the single-agent front-end, including the extraction of features from images and LiDAR data. The most commonly employed sensors are monocular, stereo, 2D LiDAR and IMU.

In a centralized approach, a multi-agent front-end is responsible for the fusion of data from multiple agents, ensuring the integration of information in a coherent manner to enhance mapping accuracy. In this scenario, the front-end is a single entity managed by the central server. Indeed, the single robot operates akin to a sensor, focusing solely on navigation and odometry, while the server oversees the remainder of the SLAM process.

Conversely, the key idea of the decentralized approaches is that each robot generates its own map. Consequently, the front-end of a multi-agent SLAM algorithm is similar to the single agent scenario, wherein each robot operates independently. Indeed, odometry measurement and intra-robot loop closure are conducted on a single robot.

Loop Closure Detection

As outlined in Section 2.2, loop closure detection is employed to mitigate the effect of odometry drift. In contrast to the front-end of a single-robot SLAM algorithm, in a multi-agent approach the loop closure is divided into two distinct phases: intra-robot loop closure, which identifies instances where a single robot revisits a previously observed location and facilitates the correction of odometry errors, and inter-robot loop closure, which enables the recognition and management of interactions between robots (e.g., the identification of shared landmarks or instances of overlap in their observations).

On the other hand, inter-robot loop closures involve correlating the trajectories of different robots [39]. These loop closures serve as the connecting points that merge the estimates from multiple robots, integrating individual local maps into a unified global understanding of the environment. The generation of inter-robot loop closures is a key focus of front-end development in C-SLAM systems and is crucial for maintaining consistent estimates across the entire multi-robot network. Inter-loop closures can be divided into two main categories: direct and indirect. The initial category encompasses physical encounters and direct sensing, which are employed to compute relative poses. This occurs when two robots physically meet and utilise direct sensing to ascertain their relative position, thereby offering high accuracy but being constrained by the necessity for proximity. The latter category employs retrospective map analysis to identify overlaps and estimate transformations, thereby providing enhanced accuracy and scalability but necessitating greater resources and sophisticated data handling.

3.1.3 Back-end and Map Merging

A significant challenge in C-SLAM is the integration of disparate maps, which represents a fundamental aspect of the back-end of a multi-agent SLAM algorithm. The solution of this problem depends on the algorithm [50]. According to Saeedi et al. [7], a SLAM algorithm comprises three essential components: sensors (front-end), data processing (back-end, where filtering, smoothing, and AI are employed), and map representation. The following six types of map are distinguished:

- A grid or location map represents a world section with a binary random variable indicating the probability of an object being in a given cell.
- The feature or environment is represented with a global position of the feature extracted.
- The topological model is an abstract model with path and intersections (an example is the Voronoi graph).
- The semantic model contains functional and relational information.

- The appearance model, where different pictures (nodes) are linked in an undirected graph.
- Hybrid.

In a straightforward centralized approach there is only a single copy of the map in the central server: each agent is responsible for updating this unique map. On the other hand, in decentralized approach, agents operate independently and communicate directly with one another. As a result, there is a possibility of conflicts arising. Each map represents a particular robot's point of view, as it is the result of a combination of local data and data from other agents, which may or may not include all maps. There are multiple methods for sharing information between agents. The most prevalent approach is to update the local map upon encountering another agent (i.e., when an agent meets another agent). However, cooperative SLAM can also entail assigning each agent a specific area to explore, with map updates occurring subsequently. The primary objective is to achieve consensus between agents. In graph-based models, there is a well-established theory on consensus between nodes.

Filatov et al. [51] identify three main challenges in the merging phase:

- The nature of the information to be shared, whether as a whole map or only a portion thereof.
- The resolution of discrepancies between maps.
- The calibration of a robot's confidence in a foreign environment.

Saeedi et al. [7] explain that map merging is comprised of two principal stages: firstly, determining the alignment between the maps, and secondly, integrating the information from the aligned maps into a unified map. Map merging can be approached in four scenarios: known initial poses, rendezvous, relative localization, relying on overlaps. In the initial approach, the initial poses of the robots are known. In this instance, the relative poses can be determined at any given moment, thus facilitating the merging of the maps. Nevertheless, this assumption is seldom applicable in practical scenarios, thereby limiting its utility. In the second approach, which is particularly prevalent, the robots meet at a common point, thereby enabling them to calculate their relative poses and merge the maps. However, the coordination of the rendezvous introduces an additional layer of complexity to the problem. The third approach involves a robot localizing other robots within its map without meeting them directly. This approach necessitates the presence of the other robots within the mapping robot's field of view. The final approach utilizes overlapping areas between the maps to compute the relative transformation. While the challenge lies in identifying the overlaps, this method eliminates the necessity for robots to meet or remain within each other's maps. In this scenario, each robot is autonomous.

3.2 Active SLAM

In the field of autonomous robotics, mobile robots are designed with the objective of exploring unknown and intricate environments. Traditional SLAM algorithms are passive in nature, guiding robots towards predefined waypoints. The primary goal of these algorithms is to map the environment and determine its localisation within the map, without the control of the robot's movement [5]. In contrast, Active SLAM (A-SLAM) incorporates active decision-making, whereby navigation strategies that reduce uncertainty in map and pose are proactively proposed, thereby facilitating fully autonomous navigation and mapping [64].

Active SLAM is inherently a decision-making problem [43]. A number of frameworks have been developed to assist in these decisions, including the Theory of Optimal Experimental Design (TOED), which selects actions based on predicted map uncertainty, and information-theoretic approaches that focus on information gain. Other approaches rely on control theory, such as Model Predictive Control, and computational methods like Bayesian Optimization and Gaussian beliefs propagation. The field of A-SLAM is currently the subject of considerable interest among researchers. Indeed, the number of publications on the topic has increased markedly in recent years, from 53 in 2010 to over 660 in 2022 [5].

The primary challenge within A-SLAM is to achieve an equilibrium between the objectives of exploration (enhancing environmental understanding) and exploitation (revisiting areas to achieve loop closure), as outlined in [6]. In traditional SLAM systems, the trade-off is managed by optimizing map estimation and localization. However, in autonomous navigation, this necessitates a dynamic approach, whereby the robot must continuously determine when to explore new areas and when to exploit known areas in order to achieve the optimal map accuracy.

3.2.1 A-SLAM example using a TurtleBot3 Burger

Escobar-Naranjo et al. [65] present a comprehensive review of the latest research on the design and implementation of control systems using reinforcement learning based on Deep Q-Networks (DQN). They demonstrate that this approach offers significant improvements in navigating sharp turns compared to the Pure Pursuit Control (PPC) algorithm. Furthermore, they address the topic of robot learning, particularly in dynamic environments, and the challenges of information processing in mobile robotics.

The article presents a case of study with a TurtleBot3 Burger, which employs a 2D LiDAR sensor to detect obstacles within a 360-degree range. In the Gazebo simulation, 11 LiDAR points are employed to facilitate training while ensuring effective obstacle localization. The robot's state is defined by 13 values: 11 derived from the sensor and 2 from the angle and distance to the target, calculated using odometry. These values are transmitted via ROS. Figure 3.2 illustrates the potential

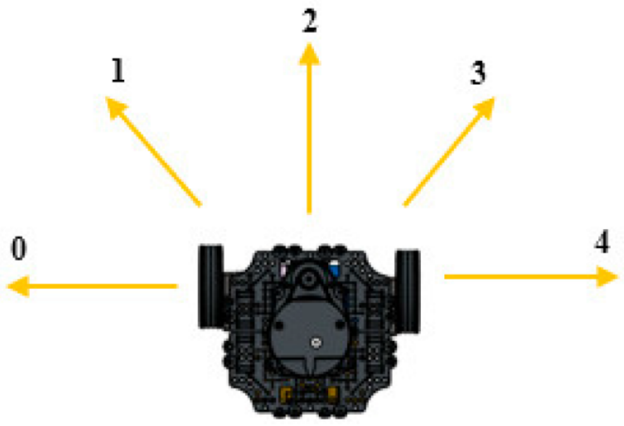

Figure 3.2: Possible actions of the TurtleBot3 Burger. Action 0 needs an angular velocity of -1.5 rad/s, Action 1: -0.75 rad/s, Action 3: 0.75 rad/s and Action 4: 1.5 rad/s. [65]

actions that the TurtleBot3 Burger may undertake during its interaction with the surrounding environment. The robot maintains a constant linear speed of 0.15 m/s, while the angular speed varies in accordance with the selected action.

The TurtleBot3 is programmed to receive a reward following the completion of each action. Upon reaching the designated goal, the robot is granted the maximum positive reward. Conversely, in the event of a collision with an obstacle, the robot is subject to the maximum negative penalty. In the remaining scenarios, the reward is as follows:

$$R = R_\theta \cdot R_d \tag{3.5}$$

where $R_\theta = \frac{5}{\theta}$ is the reward based on the angle θ between the robot and the goal, and $R_d = 2 \cdot \frac{D_a}{D_c}$ is the reward based on the current distance D_c between the robot and the goal compared to the absolute distance D_a from the starting point to the goal.

3.2.2 Problem Formulation

The A-SLAM problem can be defined as a Partially Observable Markov Decision Process (POMDP), which can be described by a seven-tuple (X, A, O, T, ρ_0, β, γ) [6]:

- X is the robot's state space,
- A is the action space,
- O is the observations space,
- $T(x, a, x')$ is the state transition function (i.e., the probability of transitioning from state x to x' given action a),
- $\rho_0(x, a, o)$ is the sensing uncertainty (i.e., the probability of observing o given the new state x' and action a,
- β is the reward associated with actions taken in specific states,
- $\gamma \in$ (0,1) is set to balance between immediate and future rewards.

In the absence of direct observation, the robot maintains a belief state, denoted by b_t, which represents a probability distribution over all possible states [5]:

$$b_t(x_t) = p(x_t|o_{1:t}, a_{1:t-1}) \tag{3.6}$$

The belief space $B(X)$ of probability density function over the sets X is:

$$B(X) = \{b : X \to \mathbf{R}| \int b(x)dx = 1, b(x) \ = 0\} \tag{3.7}$$

The optimal policy, denoted by α^*, is defined as the value that maximizes the expected reward for every state-action pair. Considering that belief is a sufficient statistic, an optimal policy α^* can be found by solving an equivalent continuous-space Markov Decision Process (MDP) over $B(X)$ [5]. This problem is defined by the 5-tuple (B, A, T, $\overline{\beta}$, γ). In this formulation, the belief-dependent reward function is:

$$\overline{\beta}(b_t, a_t) = \int_S b_t(x_t) \cdot \beta(x_t, a_t)dx_t \tag{3.8}$$

At time t, the robot has to choose the optimal policy:

$$\alpha^*(b) = \arg\max_{\alpha} \sum_{t=0}^{\infty} E[\gamma^t \overline{\beta}(b_t, \alpha(b_t))] \tag{3.9}$$

Although this approach is widely used, it is considered to be computationally expensive [6]. Consequently, in order to create an A-SLAM architecture, it is sufficient to add three sub-systems: waypoint planning, trajectory planning and a controller (as depicted in Figure 3.3). A traditional SLAM algorithm processes sensor data in the front-end and the back-end to generate a map and estimate the robot's pose. The A-SLAM algorithm extends this workflow by incorporating waypoint planning, trajectory planning and a controller.

The waypoint planning function is responsible for selecting the desired goal position in its current map, which may be a location for exploration or exploitation. The frontier-based exploration is one of the most frequently employed techniques for identifying goal positions, whereby the frontier is the boundary between the known and unknown regions of the map.

The trajectory planning function connects these waypoints over time to create a feasible path for the robot. This is achieved by computing the cost of reaching the goal and determining the utility of visiting each location. The cost function is based on metrics derived from information theory (such as Entropy or Kullback-Leibler divergence) or theory of optimal experimental design (such as A-optimality, D-optimality or E-optimality). To create a trajectory, there are two main approaches: geometric and dynamic methods.

The controller issues commands to the robot's actuators to follow the specified path, selecting and executing the action with the highest utility. In order to complete these tasks, there are three main approaches: Probabilistic Roadmaps (such as Rapidly Exploring Random Trees, D^* and A^*), the Optimal Controllers (such as Linear Quadratic Regulator or Model Predictive Control) and Reinforcement Learning. According to the study by Ahmed et al. [6], path-planning approaches such as A^* and D^* (as global planner) are employed in 19% and 25% of cases, respectively. Conversely, continuous space-planning methods, including MPC, TEB and DWA/DWB (as local planners) are used in only 11% of cases.

In the execution of an active SLAM algorithm, it is optimal for a robot to evaluate each potential action within the limits of its operational and mapping space [43]. However, the exponential increase in computational complexity with the size of the search space renders this approach impractical for real-world applications. Consequently, a smaller subset of locations is selected through of methodologies such as frontier-based exploration. In order to compute the utility of a particular action, the robot would ideally be able to predict how its pose and map might evolve by accounting for both future actions and potential future measurements. If the joint probability distribution for this were known, information-theoretic measures, such as information gain, could be employed to rank the actions. However, calculating this joint probability is infeasible. Therefore, in order to solve this problem, approximations are frequently used.

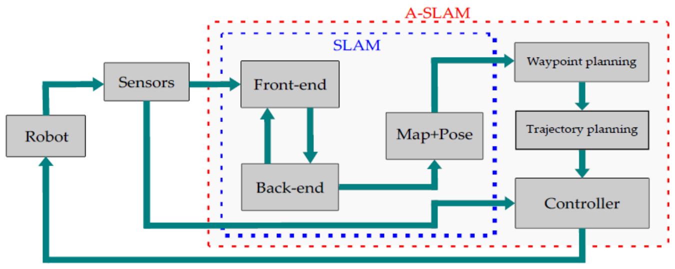

Figure 3.3: Architecture of SLAM (blue dotted lines) and A-SLAM (red dotted lines) [6].

3.2.3 Active Collaborative SLAM

Active Collaborative SLAM (AC-SLAM) builds upon the foundations of traditional SLAM and Active SLAM to create a new paradigm for collaborative multi-robot systems [6]. In AC-SLAM multiple robots work in concert, sharing information to enhance their localization precision and map the environment with greater efficiency. In this approach, the robots not only collaborate in mapping, but also actively plan their future trajectories and actions to optimize the exploration of the unknown environment. The collaborative nature of AC-SLAM introduces a new set of challenges, including the management of computational resources, the handling of communication bandwidth, and the assurance of resilience against network failures. In order to guarantee that all robots within the system are aligned with regard to the map and their respective poses, it is essential that efficient information exchange is facilitated. The key parameters that are exchanged between robots include entropy, localization information, visual features and frontier points.

Furthermore, in these methodologies, robots can alternate between processes of self-localization and the provision of assistance to other robots. In A-SLAM, achieving an equilibrium between exploration (maximising the area covered) and exploitation (revisiting known areas for loop closure) is crucial for accurate estimation of both the robot's and landmarks' poses. This, in turn, facilitates enhanced localization and mapping performance. In AC-SLAM, the existence of more than one robot, allows to the swarm to reach this equilibrium.

There are two significant challenges in this field: the development of collaborative multi-robot exploration techniques, whereby robots must coordinate their actions to efficiently and effectively explore diverse regions, and collaborative multi-robot active estimation, where robots perform actions to actively reduce uncertainty over relevant random variables [5].

Chapter 4

The Robot Operating System

4.1 ROS Concepts

ROS (Robot Operating System) is an open-source framework designed for the development of robotic software. It should be noted that, despite its designation, it is not an actual operating system in the traditional sense, but rather a middleware solution. It provides a set of tools, libraries and several abstraction layers, thereby affording scalability, flexibility and a modular construction of robotic applications. The ROS framework has been pivotal for research for a considerable period of time, since its initial development by Willow Garage and subsequent maintenance by the Open-Source Robotics Foundation (OSRF) [66]. The rapid growth of ROS is attributable to its open-source nature and extensive libraries, which are beneficial for a wide range of robots. The robust ecosystem of algorithm packages for sensors, control, and communication has significantly enhanced the productivity and functionality of the ROS framework.

The objective of ROS is to facilitate the organization of heterogeneous systems and tasks across a diverse range of robotics applications. This enables the simulation and execution of intricate control, planning and coordination tasks. ROS is structured around the concept of Node, which represents a single-purpose process executing a specific task within the robotic system. These nodes perform computations, manage data and communicate with each other through a messaging system composed of topics, services and actions. Nodes are designed to be modular and can operate independently, allowing for distributed processing across multiple machines or devices. Furthermore, the framework supports parallel and distributed processing, enabling robots to perform complex tasks in dynamic environments.

Currently, there are two versions of ROS: ROS1 and ROS2. ROS1 was first released in 2007, while ROS2 was subsequently released in 2017 as an evolution of ROS1. To avoid any confusion regarding the acronym, in this chapter, ROS will refer to the general middleware, while ROS1 and ROS2 will refer to the specific versions.

4.1.1 ROS Filesystem

ROS [67] employs a structured approach to the organization of codes and resources, with the objective of facilitating the development, reuse and sharing of software packages. These packages constitute the fundamental unit of organisation in ROS. A ROS workspace is a directory where packages are developed and compiled. The workspace contains sub-directories, including:

- `src`, containing packages
- `build` for compiled code
- `install` for installed package
- `log` for logging info.

In ROS1, the command `catkin_make` is employed for building the workspace. In ROS2 the analogous command is `colcon_build`. A package is organized in the sub-directories, such as `config`, `include`, `launch`, `src`, and contains the files `package.xml` and `CMakeList.txt`. The `package.xml` file serves to guarantee the correct construction and incorporation of packages into the broader ROS ecosystem. It contains metadata and dependencies. The `CMakeList.txt` file contains instructions for the construction of the package, including details of the compilation and linking processes, as well as information on the dependencies required. A `launch` file is employed to initiate the execution of multiple nodes concurrently. They facilitate the automation of complex system startup by specifying nodes to run, parameters, and additional configurations. The `param` and `config` file are written in `YAML` format and store parameters that can be loaded to customize node behavior without modifying the code. The source files (`src`) implement the functionality of the nodes that interact in the system. These files are written in Python or C++ and are stored within packages.

4.1.2 ROS Communication system

A representation of the ROS communication systems can be visualized as a graph, wherein the nodes represent the vertices and the edges connecting them. The ROS framework offers a variety of communication patterns, including topics, services, and actions. Topics facilitate asynchronous communication between nodes [68]. A publisher-subscriber paradigm with a strongly typed interface is employed to ensure clarity and reliability. Messages are exchanged between nodes in the form of message files with the (`.msg`) extension, which specify the types of data that can be sent, including integers, floats, and arrays. In this model, a node that wishes to publish to a topic takes on the role of the publisher, while all the other nodes that wish to receive communication from that topic take on the role of subscribers. The `publish()` method is used to send messages to the topic. Figure 4.1 illustrates the communication process over a given topic.

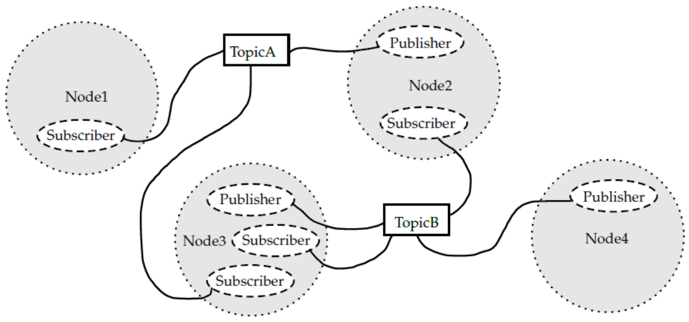

Figure 4.1: An asynchronous communication over Topics. Node 2 publishes messages to Topic A, which are received by Node 1 and Node 3, as subscriber. At the same time, both Node 3 and Node 4 publish messages to Topic B, with Node 2 and Node 3 subscribe to this Topic. This setup allows a node to send messages to itself [68].

In contrast, services represent a synchronous request-response communication pattern. They are employed for tasks that necessitate confirmation of completion or receipt of information. A node utilising a service is not required to block or wait during a call. A service file (`.srv`) is structured into two sections: the request and response. Figure 4.2 shows communication between a client and server over a Service.

Actions are designed for the execution of asynchronous, goal-oriented tasks that necessitate the provision of periodic feedback. They are employed in the context of long-running tasks and operate in a non-blocking manner. The format of an action is defined within a text file with the `.action` extension. An action is characterised by three messages. Figure 4.3 illustrates the process of synchronous communication over an action.

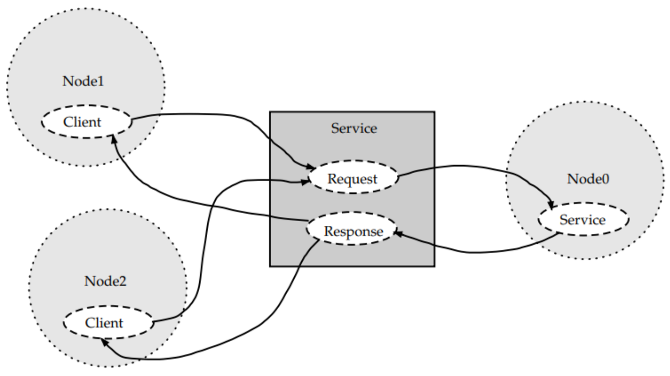

Figure 4.2: A client-server communication over Services. Node 0 functions as a server, while Node 1 and Node 2 operates as clients [68].

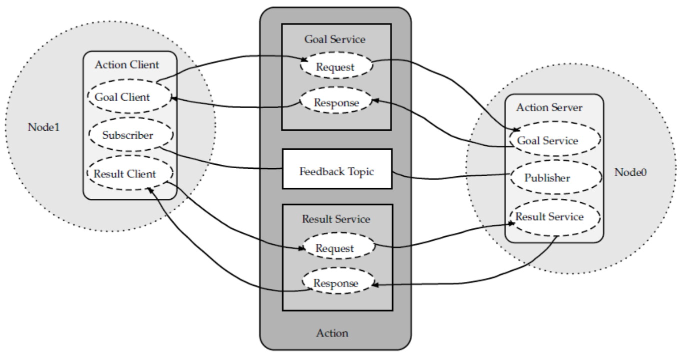

Figure 4.3: A synchronous communication over an Action. Node 0 is the server, and Node 1 is the client. An action is composed by a Goal Service, a Feedback Topic and a Result Service. The client sends goal requests to the server, which continuously publishes messages to the topic. Eventually the server ends this communication by sending the result response to the client [68].

4.2 ROS2 as the evolution of ROS1

The use of ROS1 in industrial robotics persists. However, ROS1 has a number of limitations, including the lack of real-time execution capabilities, limited support for various operating systems, a lack of built-in security measures, process synchronisation, and its status as a centralised system, which introduces the potential issue of a single point of failure. In order to address the limitations of ROS1 and to meet the needs of the industry and the wider community, Open Robotics released ROS2 in 2018, following an initial announcement in 2014 [66]. The community opted to undertake a complete redesign of the middleware from scratch. The primary distinction is the alteration of the network protocol, which shifted from TCP/UDP to Data Distribution Service (based on UDP). This modification has enhanced efficiency, security, and fault tolerance, while also offering support for real-time and embedded systems, multi-robot communication, and reliable operation in challenging networking conditions.

The design of ROS2 is based on a set of four principles and requirements, which have been developed with the objective of creating a robust and flexible framework for robotics [69].

The first principle is the Distribution. ROS2 addresses robotics challenges by decomposing them into distributed systems. Each individual task is represented by a node, which is a single thread and is isolated into its own independent component. These components operate independently and communicate with one another during operation, thereby ensuring that the system remains decentralized and secure. Examples of nodes include the management of device drivers, the handling of perception systems, and the control of different robotic functions.

The ROS2 framework provides support for Abstraction. Communication between components is guided by well-defined interface specifications, which delineate the manner in which data should be exchanged and interpreted. The objective is to achieve a balance between the level of detail displayed and the ability to replace components with alternatives without being excessively specific or tailored to a particular application.

ROS2 systems communicate in an Asynchronous manner, whereby messages are sent and received independently without requiring a response. This creates an event-driven environment where different components can operate at their own pace.

ROS2 is, in essence, an extension of the UNIX philosophy of decomposing a complex problem into a series of more straightforward sub-problems. A modular system, in this context, is one that has been designed to break itself down into smaller, more focused pieces, whether these be library APIs, message structures, command-line tools, or the overall software ecosystem.

ROS2 facilitates the development of robotics applications by providing libraries that utilize fundamental programming utilities (e.g., lock/mutex), thereby simplifying the creation of tools for concurrent computing in distributed systems [68]. This process reduces the occurrence of identical or similar components and ensures the consistency of interfaces, thereby enhancing compatibility across applications developed by practitioners in both industry and academia. The integration of code into ROS2 applications is achieved by developers through the utilization of client libraries, including `rclpy` for Python and `rclcpp` for C++. ROS2 applications are characterised by a structured architectural approach, wherein application code is encapsulated within Node modules. Technically, a Node represents a class provided by `rclcpp` and `rclpy`. It is feasible to inherit from the class `Node`, thereby gaining access to the non-private methods (which facilitate both inter-node and intra-node communications). This necessitates the implementation of a modular structure, whereby the code is divided into discrete units, enabling the concurrent execution of disparate components of the application.

In ROS2, both the libraries provide the class `Executor`, which manages the execution of callback functions across one or more threads. The executor can be configured as single-threaded or multi-threaded, the latter being the more prevalent approach for achieving effective parallel execution.

In this instance, it is recommended that callback functions assigned to specific callback groups. This approach enables the concurrent execution of different functions, thereby providing flexibility and control over parallel execution. There are two types of groups: mutually exclusive groups, in which functions within the same group cannot run in parallel, and reentrant callback groups, in which functions within the same group can run in parallel. The intermediate interface is provided by the `rcl` library. The ROS Client Library, written in C, provides common functionality to all client libraries [69]. The middleware abstraction layer, designed as `rmw`, defines the communication interfaces within ROS2. This abstraction facilitates the switching between different DDS implementations, including `cyclone_dds`, `fast_dds` and `connext_dds`. The layer enables the user to select the optimal middleware solution for their use case without substantial modifications to the application. Figure 4.4 provides an overview of the ROS2 Client Library API Stack. In Table 4.1 there is a summary of the differences between ROS1 and ROS2.

The ROS distribution employed in this thesis work is ROS2 Humble Hawksbill (2022), representing the eighth ROS2 release. It is based on Ubuntu 22.04 (Jammy). ROS2 Humble is not the last version of ROS2 (which is ROS2 Jazzy Jalisco) but it is still supported (at least until May 2027).

4.3 Software Utilities

ROS2 offers a comprehensive set of algorithms, including those for perception, SLAM, planning and other robotic tasks. It provides a multitude of tools for various aspects of development [69].

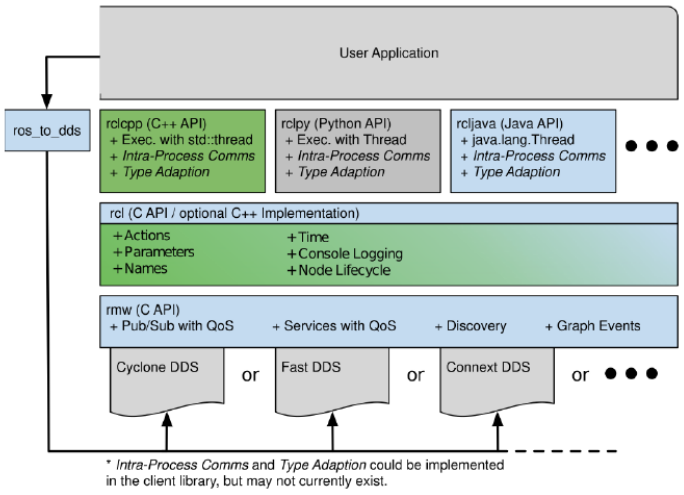

Figure 4.4: ROS2 Client Library API Stack, composed by the User Application, the API C++/Python, the C API, the middleware and the DDS [69].

Category	**ROS1**	**ROS2**
Network transport	TCP/UDP	DDS
Network architecture	Client-server `roscore`	Peer-to-peer
OS support	Linux	Linux, macOS, Windows
Node vs process	Single node per process	Multiple node per process (threads)
Threading model	Callback queues and handlers	Swappers executor
Node state management	None	Lifecycle nodes
Embedded systems	Minimal support	Supported: micro-ROS
Paramter types	Type inferred when assigned	Type declared and enforced

Table 4.1: Differences between ROS1 and ROS2 [69].

4.3.1 RViz2

RViz2 [70] is a three-dimensional visualisation tool developed for ROS2. The software allows developers to visualize sensor data, robot models and a plethora of information pertinent to a robot's environment and operations in real time. RViz2 is capable of supporting a wide range of visual elements, including point clouds, laser scans, robot states, and maps. This versatility makes it an invaluable tool for debugging and developing complex robotic systems, both in simulation and in the real world. The user is able to interactively configure displays, adjust parameters and monitor robot status, thereby facilitating the process of designing and testing algorithms such as navigation, perception and path planning. The flexibility and integration with ROS2 topics facilitate the seamless visualisation of a robot's environment and behaviour. In the context of SLAM, RViz provides developers with the ability to observe and analyze the way in which the SLAM algorithm is constructing a map and localizing the robot in real-time. This visual feedback facilitates debugging and fine-tuning of SLAM algorithms by highliting potential issues. RViz enables the visualization of the robot's trajectory, landmarks and obstacles. RViz can be employed during both simulation and real-world tests.

4.3.2 Gazebo

The Gazebo simulator is a three-dimensional environment that is employed for modelling and evaluation of the performance of sophisticated robotic systems [1]. Gazebo enables developers to assess the performance of robotic designs within virtual environments, incorporating data from a range of sensors. Users can construct models of robots, environments and dynamic interactions, such as collisions or sensor noise, to verify algorithms prior to deployment on actual hardware. Different environments can be selected, or a new one created (the definition is done in `.world` file). It is also possible to simulate multiple robots simultaneously. In the context of SLAM, Gazebo allows developers to simulate a robot moving through a virtual environment while generating realistic sensor data, including LiDAR scans, depth cameras, and IMU readings. This data can be used by the SLAM algorithm to build a map of the environment, and localize the robot within it. Gazebo's physics engine ensures that sensor readings reflect real-world conditions, including noise and dynamic interactions, making it an effective platform for validating SLAM approaches. The robots uses an XML file format designed as Universal Robot Description Format (`.urdf` file), which delineates the configuration and structural aspects of the robot.

4.3.3 Nav2

Nav2 [71] represents the second generation of the ROS Navigation Stack. It has been designed with the objective of enabling the deployment of advanced technologies from autonomous vehicles into mobile and surface robotics. Nav2 is a robust, production-grade navigation framework that offers a comprehensive range of capabilities, including perception, planning, control, localization, and moreover, to facilitate the development of highly reliable autonomous systems. The system enables autonomous path planning, navigation and obstacle avoidance in complex environments, both known and unknown. Thanks to the ROS2's modular architecture, Nav2 is adaptable to various robot platforms and configurations, regardless of robot kinematics. It includes key components, such as global and local planner, costmap, behavior trees for decision-making. The inclusion of multiple behavior trees allows robots to perform complex, varied tasks efficiently. The core of naviga-

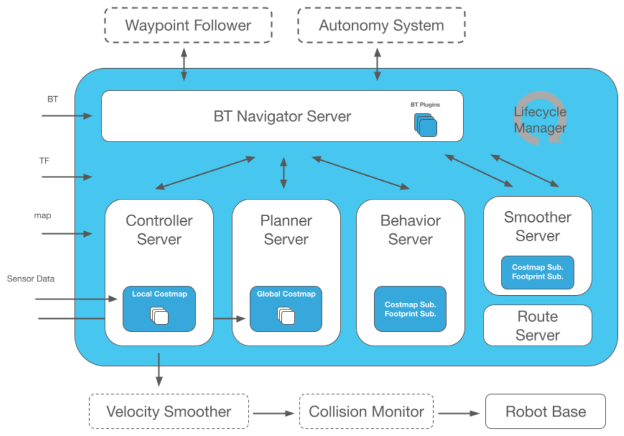

Figure 4.5: Nav2 Architecture, composed by the Local Planner, Global Planner, the Behavior Server, the smoother [71].

tion tasks is comprised of planners, controllers, smoothers, and recovery servers. These entities share an environmental representation, such as a cost map, which they utilise to process their respective tasks. A global planner computes paths or routes based on a global environmental representation and sensor data. The path can be computed in different ways, depending on the task. The smoothers improve the globally planned path by reducing sharp turns and ensuring safer distances from obstacles. The local planner (or controller) is used to adapt the globally computed path to the real world, avoiding dynamic obstacles. The robot's perception of its environment is stored in a cost map, a two-dimensional grid representing free space, obstacles, and unknown areas.

In Nav2, there are various transformations: `map` → `odom` provided by the global position system (such as SLAM), `odom` → `base_link` provided by an odometry system, and finally `base_link` → `sensor frames` of the robot.

4.3.4 Computational Graph Visualization

As explained in 4.1.2, ROS2 can be visualized as a graph, wherein the nodes represent vertices and the edges serve to connect them. Computational graph visualization is the process of graphically representing the network of nodes and their interconnections within a ROS system. The visualization is a useful tool for understanding how the nodes interact and exchange information across topics, services, and actions. The tool `rqt_graph` generates these visualizations, allowing developers to debug and optimize their systems.

4.3.5 TF2 - The Second Generation of the Transform Library

In order to perform a task, such as moving robot's arm to grasp a moving object, a robot must be able to ascertain its position relative to the surrounding world, as well as that of the object itself [72]. In the field of robotics, this relationship is represented through the use of coordinate frame transformations. As the complexity of robotic systems increases, the manual management of reference frame transformations becomes an increasingly error-prone task, particularly when data are sourced from multiple sensors or devices distributed across multiple compute nodes. This challenge has prompted the development of the `tf` library, which automates the management of reference frames, thereby enhancing efficiency and reducing the probability of error.

The library is comprised of two principal components. The role of the broadcaster is to disseminate information about transformations. It is the function of this component to disseminate information regarding transformations. When a system component is aware of the relationship between two frames, for instance, the sensor frame and the robot base frame, it transmits updates on the status of the transformation on a regular basis. Broadcasters are then tasked with disseminating this information. The listener is a component that receives messages sent by broadcasters and stores them in a chronologically ordered list. It is capable of responding to queries that require the transformation between two frames at a given time, interpolating the data in the case of partial information.

The representation of transformations between frames can be achieved through the use of a directed graph, wherein the nodes represent coordinate frames and the edges depict the transformations. To avoid ambiguity, the graph must be acyclic and organized as a tree. This design choice enables rapid lookups and dynamic management of relationships between frames, facilitating changes to the graph without data loss or the necessity to recalculate the entire structure. A timestamp is associated to each transformation, allowing the maintenance of a historical record of transformations and the retrieval of past data. Consequently, it is possible to ascertain both the current and historical connections between frames.

In ROS2, `tf` was succeeded by `tf2`. These concepts are intrinsic to `tf`, but they can be adapted to `/tf`. In addition to the `/tf` topic, the `/tf_static` topic has been introduced in ROS2. This is structurally similar to `tf`, but it only tracks transformations that do not change over time, i.e., static transformations.

Chapter 5

TurtleBot3 Burger Overview

The ROBOTIS TurtleBot3 Burger [73] is a two-wheel differential drive system designed for educational, research, and product prototyping contexts. It can be customized by modifying its mechanical components or adding optional parts, such as computers and sensors. The TurtleBot3 Burger's principal competencies encompass SLAM navigation, and manipulation, rendering it eminently suitable for domestic service robots. It is able to utilize SLAM to map environments and to navigate autonomously within them. The TurtleBot3 Burger can be controlled remotely via a laptop, joypad, or an Android smartphone, and is capable of tracking and following people as they walk.

The TurtleBot3 Burger is equipped with a 360-degree 2D LiDAR LDS-02 sensor, an OpenCR microcontroller (ARM cortex-M7) with integrated sensors (including a 3-axis gyroscope, accelerometer, and magnetometer), a Raspberry Pi 4, and servomotors to control the wheels [74]. The maximum linear velocity is $0.22m/s$, while the maximum angular speed is $2.84rad/s$. Table 5.1 provides an overview of the hardware specifications of the TurtleBot3 Burger.

According to [75], the TurtleBot3 Burger's LiDAR exhibits high accuracy in the detection of obstacles at varying distances and angles, with an average error rate of 2.389%. Upon detecting an obstacle directly in front of the robot at a distance of $10 - 15$ cm, the robot ceased its movement to avoid a potential collision. However, the study revealed that the LiDAR sensor was unable to detect obstacles within a distance of less than 10 cm, which could result in collisions. Despite this limitation, the LiDAR-based obstacle avoidance system demonstrated effectiveness in preventing collisions at greater distances.

In order to successfully launch the robot's functionality within the ROS framework, it is essential to execute the following command on the robot terminal:

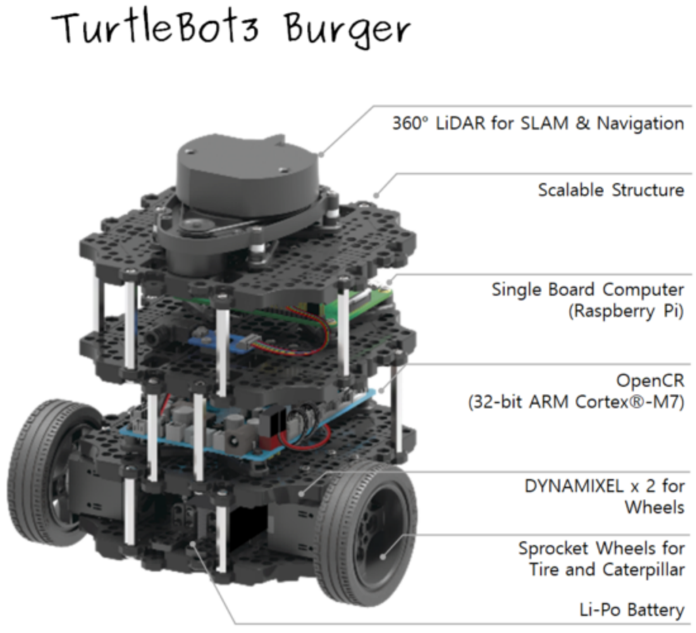
TurtleBot3 Burger
360° LiDAR for SLAM & Navigation
Scalable Structure
Single Board Computer (Raspberry Pi)
OpenCR (32-bit ARM Cortex®-M7)
DYNAMIXEL x 2 for Wheels
Sprocket Wheels for Tire and Caterpillar
Li-Po Battery

Figure 5.1: The TurtleBot3 Burger with its hardware components, such as the LiDAR, the SBC, the OpenCR, the Motors and the battery [73].

Maximum translation velocity	$0.22m/s$
Maximum rotational velocity	$2.84rad/s$
Maximum payload	$15kg$
Size ($L \times W \times H$)	$138 \times 178 \times 192mm$
Weight (SBC, Battery, Sensors)	$1kg$
Expected operating time	$2h30m$
SBC	Raspberry Pi
Board	OpenCR1.0
MCU	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)
LDS (Laser Distance Sensor)	360 LiDAR LDS-02
Camera	-
IMU	Gyroscope 3 Axis, Accelerometer 3 Axis
Motors	2 DYNAMIXEL XL430-W250executable-T

Table 5.1: Hardware specifications of TurtleBot3 Burger [73].

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

Upon successful completion of the process, certain nodes and topics will be initiated [76]. The following topics have been configured and are ready for use:

- `/battery_state` contains a comprehensive account of the battery's characteristics, including voltage levels, charge percentage, and the voltages of its individual cells.
- `/imu` presents data on linear and angular accelerations in relation to the local reference frame.
- The topic `/magnetic_field` contains information regarding the measured magnetic field.
- `/joint_state` provides information regarding the position and velocity of each wheel.
- `/scan` presents a detailed account of the distances to objects detected in the surrounding environment by each beam of the LiDAR sensor.
- `/odom` presents the position of the robot in relation to a reference point that is established at the initialization of a node.
- `/cmd_vel` is a topic where linear and angular velocity commands can be published.
- `/sensor_state` presents data regarding the number of ticks generated by each motor as the robot traverses its path, employing information obtained from the left and right encoders.

Figure 5.2 illustrates the ROS2 communication network of the TurtleBot3 Burger.

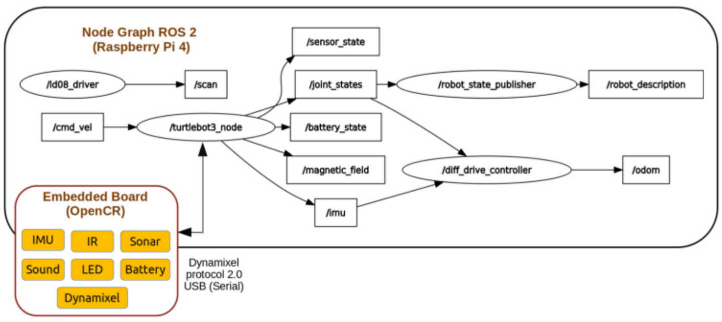

Figure 5.2: `rqt_graph` after the TurtleBot3 Burger initialization [76].

5.1 Kinematic Model

The kinematic model elucidates the interrelationships between the system's control inputs, velocities, and resultant behavior within a state-space representation [76].

The TurtleBot3 Burger is a mobile differential drive vehicle that moves in a bidimensional plane. Consequently, the pose is defined by three variables: the 2D coordinates $X(t)$ and $Y(t)$, which are relative to the external coordinate system, and the angular orientation, which is represented by the variable $\theta(t)$. In order to perform the function of navigation, the robot employs of odometry, Inertial Measurement Unit (IMU) and LiDAR. The determination of the robot's pose is dependent upon a number of key operations, including the reading of encoder values, the calculation of movement and turning angles, and the correction of errors that may be caused by skidding or mechanical issues. The IMU assists in the reduction of errors by utilizing the data obtained from the accelerometer, gyroscope, and magnetometer to refine the calculations of position and orientation [74].

The kinematic model of the a mobile differential drive vehicle is equivalent to the unicycle kinematic model:

$$\begin{pmatrix} \dot{X}(t) \\ \dot{Y}(t) \\ \dot{\theta}(t) \end{pmatrix} = \begin{pmatrix} \cos\theta \\ \sin\theta \\ 0 \end{pmatrix} v + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \omega \tag{5.1}$$

where v denotes the linear velocity and ω indicates the angular velocity. The real inputs of the mobile differential drive vehicle are the velocity of the two wheels, denoted as ω_r and ω_l.

5.1.1 Odometry

The odometry employs wheel encoders to measure angular displacement of the wheels. The distance traversed is calculated by taking into account the wheel radius and accounting for skidding. By integrating the total angular displacement of each wheel, the robot's position is determined with respect to its initial position. The rotational speed of the wheels is:

$$\omega_l = \frac{E_{lc} - E_{lp}}{T_e} \cdot \frac{\pi}{180} \tag{5.2}$$

$$\omega_r = \frac{E_{rc} - E_{rp}}{T_e} \cdot \frac{\pi}{180} \tag{5.3}$$

where E_{lc}, E_{rc} are the current encoder values, E_{lp}, E_{rp} are the previous encoder values and T_e is the elapsed time.

The linear speed of the robot is:

$$v(k) = \frac{\omega_l + \omega_r}{2} \cdot r \tag{5.4}$$

where r is the radius of the wheels. On the other hand, the angular speed of the robot is:

$$\omega(k) = \frac{(\omega_r - \omega_l)}{D} \cdot r \tag{5.5}$$

where D is the distance between the wheels.

Runge-Kutta formula allows to calculate the approximate robot's pose as:

$$X(k+1) = X(k) + \Delta s(k)\cos(\alpha(k) + \frac{\Delta\alpha(k)}{2}) \tag{5.6}$$

$$Y(k+1) = Y(k) + \Delta s(k)\sin(\alpha(k) + \frac{\Delta\alpha(k)}{2}) \tag{5.7}$$

$$\alpha(k+1) = \alpha(k) + \Delta\alpha(k) \tag{5.8}$$

where $\Delta s(k) = v(k) \cdot T_e$ is the traveled distance and $\Delta\alpha(k) = \omega(k) \cdot T_e$ is the turning angle.

5.1.2 Inertial Navigation

Given the robot's initial pose $[X(0), Y(0), \alpha(0)]$ and the initial speed $[\dot{X}(0), \dot{Y}(0), \omega(0)]$, IMU continuously measures the acceleration with accelerometer $a_x(t), a_y(t)$. It is necessary to transform measured accelerations $a_x(t), a_y(t)$ into the external coordinate systems: $a_X(t), a_Y(t)$. Integrating the acceleration is useful to determine the velocities and updated position:

$$V_X(t) = V_X(0) + \int_0^t a_X(t)dt \tag{5.9}$$

$$X(t) = X(0) + \int_0^t V_X(t)dt \tag{5.10}$$

$$V_Y(t) = V_Y(0) + \int_0^t a_Y(t)dt \tag{5.11}$$

$$Y(t) = Y(0) + \int_0^t V_Y(t)dt. \tag{5.12}$$

5.2 ROS2 Reference Frames

In order to visualize the standard TF tree of the TurtleBot3 Burger, the following command may be executed on the terminal:

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

The results of this commands are depicted in Figure 5.3. The `odom` frame is the reference frame used by a robot to determine its position, thanks to its own odometry system. The frame origin is at the point where the robot is initialized. The `base_footprint` represents the projection of the robot's base on the ground. The `base_link` is a frame attached to the base of the robot. Finally, there is a reference frame for each wheel and for each sensor.

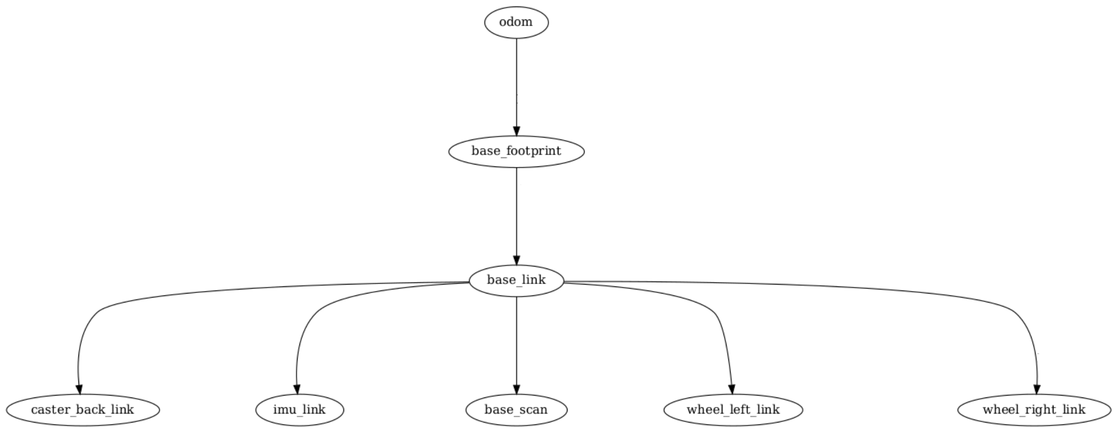

Figure 5.3: Standard TF tree

Since the `odom` frame can drift over time, SLAM algorithms usually also have a `map` frame. This is the world fixed frame and is used as a long-term global reference.

5.3 ROS2 Namespacing

In order to effectively manage a project involving multiple robots, it is essential to clearly identify and differentiate the various agents. In the context of ROS, it is possible to add a namespace to nodes, topics, and reference frames. This can be achieved by modifying the original launch files and remapping the topics. It is imperative to conduct this operation on both the server and client sides. From the robot perspective, it is of paramount importance to modify the `robot.launch.py` launch file within the `turtlebot3_bringup` package (Figure 5.4). Furthermore, in order to accurately publish the LiDAR data on the `/scan`, it is also necessary to modify the file `ld08.launch.py` (Figure 5.5). Finally, it is required to add the remapping of the reference frames in the launch file `turtlebot3_state_publisher` (Figure 5.6).

```
return LaunchDescription([
    Node (
        package='turtlebot3_node',
        executable='turtlebot3_ros',
        namespace='tb3_0',
        parameters=[tb3_param_dir],
        arguments=['-i', usb_port],
        output='screen',
        remappings=[('/battery_state','/tb3_0/
                    battery_state'),
                    ('/cmd_vel','/tb3_0/cmd_vel'),
                    ('/imu','/tb3_0/imu'),
                    ('/joint_states','/tb3_0/
                    joint_states'),
                    ('/magnetic_field','/tb3_0/
                    magnetic_field'),('/odom','/tb3_0/
                    odom'),
                    ('/robot_description','/tb3_0/
                    robot_description'),
                    ('/sensor_state','/tb3_0/
                    sensor_state'),
                    ('/tf','/tb3_0/tf'),('/tf_static','
                    /tb3_0/tf_static'),
            ]),
```

Figure 5.4: Modified `robot.launch.py` file

```python
return LaunchDescription([
    Node (
        package='ld08_driver',
        executable='ld08_driver',
        name='ld08_driver',
        output='screen',
        remappings=[('/scan','/tb3_0/scan'),
                    ('/tf','/tb3_0/tf'),
                    ('/tf_static','/tb3_0/tf_static')],
```

Figure 5.5: Modified `ld08.launch.py` file

```python
return LaunchDescription([
    Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        output='screen',
        namespace='tb3_0',
        remappings=[('/tf','/tb3_0/tf'),('/tf_static','
                    /tb3_0/tf_static')])
    ])
```

Figure 5.6: Modified `turtlebot3_state_publisher.launch.py` file

Chapter 6

Implementation of an Autonomous Collaborative SLAM approach

The successful navigation of an autonomous robotic system is dependent upon the completion of three principal activities [77]:

1. Mapping and modeling of the environment, performed by a **SLAM algorithm**.

2. Planning of a route, executed by a **global path planner**.

3. The control of the robot's movement, realized by a **path tracker** and/or a **local planner**.

This chapter presents the theoretical and technical details of the implementation of an AC-SLAM approach. The collaborative SLAM approach is centralized, in the sense that each robot is capable of functioning as a mobile sensor. From the perspective of an individual robot, the objective is to perform SLAM locally, whereby the map is produced from the robot's point of view. Consequently, the individual agent is unaware of the presence of other robots, resulting in a minimal front-end computational load, as the majority of the processing is conducted on the central server. Conversely, the back-end involves the real-time merging of the local maps on the central computer, resulting in the generation of a global map.

Nevertheless, this approach is also active. It is distinguished by the autonomy of navigation exhibited by individual robots. It is notable that during the SLAM operation, no human operator directly controls the robots; rather, the agents autonomously determine the optimal trajectory to pursue. The approach is client-server, given that it is the central computer that determines the route to be followed, while the agent's role is merely that of executing the instructions it receives.

The following chapter illustrates the selection of the SLAM algorithm as the front-end, elucidates the methodology employed for the implementation of the back-end (i.e., map merging), and offers an in-depth analysis of the robot's autonomous mobility. It provides a comprehensive account of the implementation details pertaining to the path planning and path tracker. The following paragraphs are based on the assumption that two robots are involved in the system, although the software architecture is designed in such a way as to allow for the involvement of a greater number of robots.

The initial structure of this approach was derived from a GitHub repository [78], which served as the foundation for subsequent implementation and customization.

The approach structure described in this paragraph is summarized in the graph depicted in Figure 6.1.

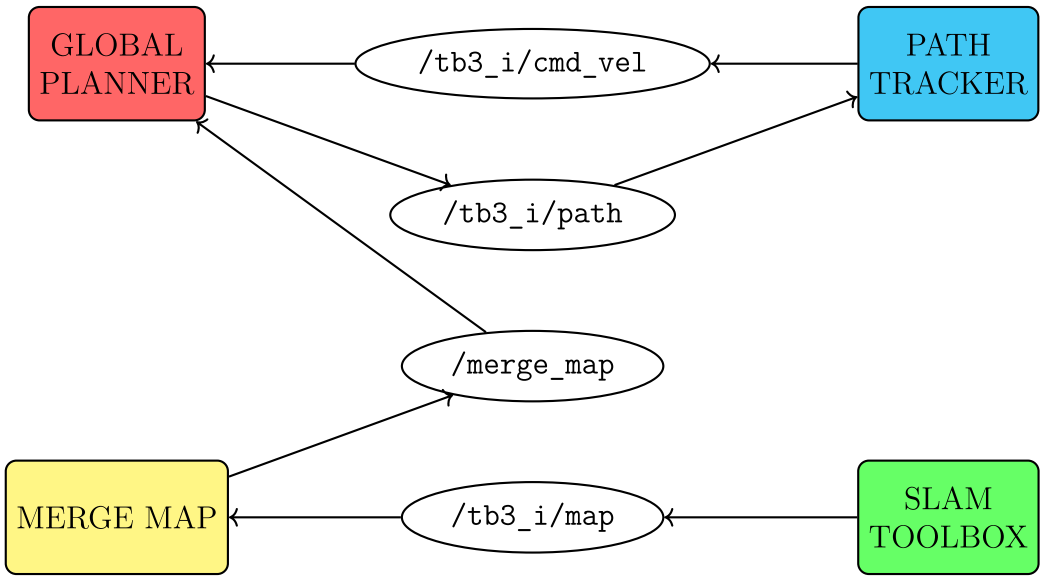

Figure 6.1: AC-SLAM approach graph. The squares represent the ROS nodes, while the ovals represent the ROS topics. An arrow leaving a node is indicative of the node's role as a publisher, whereas an arrow entering a node is indicative of the node's role as a subscriber. The prefix `/tb3_i` denotes the namespace introduced for generic robot i.

Given that this approach is ROS-based, it can be seen that the communication structure is based on nodes and topics. The main nodes, which are illustrated in detail in the remainder of this chapter, are:

- The employed front-end for the local SLAM on the single agent is the `slam_toolbox` package (Subsection 6.1.1).
- The C-SLAM back-end is managed by the package `merge_map`, which is utilized for the real-time integration of the disparate local maps (Subsection 6.1.2).
- The global path planner selected for this purpose is A^*, which generates a path that enables the robot to safely navigates around static obstacles (Section 6.2).
- In an autonomous approach, the path tracker is responsible for calculating the linear and angular velocity required for the robots to follow the path determined by the global planner. In this approach, the assumed path tracker is the Pure Pursuit algorithm (Section 6.3).

6.1 Collaborative SLAM Setup

6.1.1 Front-end: SLAM Toolbox

SLAM Toolbox is an open-source package built on the foundation of Open Karto, where a number of important updates have been made [29]. The main difference is that the existing Sparse Bundle Adjustment optimization interface has been replaced with Google Ceres, which provides faster and more flexible optimization options. It has also been selected as the default SLAM provider for ROS2, replacing GMapping. The SLAM Toolbox is integrated into the Nav2 project, enabling real-time localization in dynamic environments for autonomous navigation.

The SLAM Toolbox operates in synchronous mode, establishing a ROS Node that subscribes to laser scan and odometry topics. The SLAM Toolbox algorithm has three stages:

1. Laser and odometry acquisition
2. Data processing, which includes publish transforms, graph construction, scan matching, loop closure detection and optimization.
3. Mapping

The function `laserCallback()` gets `odom` and `scan` and pushes an object `PosedScan` into the queue. Such information is used to compute the transformation from the `odom` frame to the `base` frame. In addition, the pushed data is then popped by the `addScan` function. It calls `setTransformFromPoses` to calculate the `odom` to `map` transform, and it calls `Process`, which is the module that handles the main flow of data. It operates on the last scan, works to construct the graph, performs scan matching to improve the estimate, looks for loop closure and updates the last scan. The scan matching compares subsequent scans, looking for the best match for each current scan compared to the previous one. This process is essential to locate the robot relative to the map created.

The graph construction phase is composed of the following functions:

- `addEdges` and `addNode`, which handle the construction of the scan graph (i.e., the topology of the map).
- `LinkScan` links scans together, updating the graph with new scan information and adding constraints to the edges of the graph.
- `AddConstraint` adds nodes to the graph to ensure that poses are consistent over time. This feature is critical for improving map accuracy and pose correction.

Loop closure detection is performed by two functions `TryCloseLoop` and `Correct Pose`. The former one selects loop closure candidates and tries to close loops by detecting whether the robot returns to a previously visited position. If a loop is found, the latter applies corrections to the poses identified, while closing the loop, to maintain consistency in the global map. In addition, after the loop closure is detected and corrected, the `SetCorrectedPoseAndUpdates` function updates the robot pose estimate and map constraints. `CorrectPose` calls the Ceres nonlinear solver to optimize the pose graph. It corrects global errors in the map and reduces accumulated errors.

To sum up, a map is constructed and subsequently published, utilizing the laser scans that are associated with each pose within the pose graph. The mapping process may be either synchronous or asynchronous. In the former case, the map quality is of crucial importance, and all measurements are stored in a buffer, with processing occurring offline. In the latter case, the processing is done on a best-effort basis. The new measurements are processed only after the previous ones have been fully processed and certain update criteria are satisfied. This system is useful for real-time (online) applications as it reduces delays. The disadvantage is that some measurements may be lost.

There is another mode of operation, the pure localization mode. It is not designed to permanently record changes in the environment. Instead, it maintains a rolling buffer. An interesting consequence of this mode is its ability to function as effective LiDAR odometry when no previous mapping data is available.

Figure 6.2 summarized the SLAM Toolbox architecture.

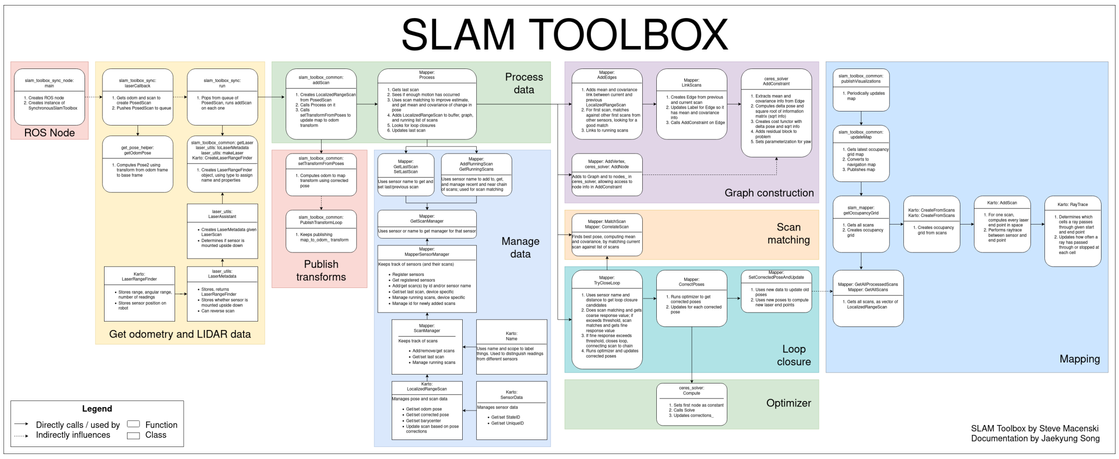

Figure 6.2: SLAM Toolbox framework [79]

In order to successfully initialize the online asynchronous mode of SLAM Toolbox, it is sufficient to execute on the central server the command:

```
ros2 launch slam_toolbox online_async_launch.py
```

However, in the multi-agent scenario it is crucial to add a namespace and to remap the various topics. The updated SLAM Toolbox launch file is reported in Figure 6.3.

Ceres Solver

Ceres Solver [46] is an open-source C++ library for modeling and solve large, complex optimization problems. It is particularly suited to solving robust-bounded nonlinear least squares problems:

$$\begin{aligned} \arg\min_{x} \quad & \frac{1}{2}||F(x)||^2 \\ \text{s.t.} \quad & L \leq x \leq U \end{aligned} \tag{6.1}$$

where $x \in \mathbf{R^n}$ is a n-dimensional vector of variables, $F(x) = [f_1(x), \ldots, f_m(x)]^T$ is a m-dimensional function of x, L and U are vector lower and upper bounds of x, respectively.

This general formulation is an intractable problem. Consequently, the objective is not to identify a global minimum, but rather a local one. The conventional methodology for addressing non-linear optimization problems is to apply a sequence of approximations to the initial problem, thereby reducing its complexity. Let

```
start_async_slam_toolbox_node = Node(
    package = 'slam_toolbox',
    executable='async_slam_toolbox_node',
    name='slam_toolbox',
    namespace='tb3_0',
    remappings=[('/map','/tb3_0/map'),
                ('/map_metadata','/tb3_0/map_metadata'),
                ('/slam_toolbox/feedback','/tb3_0/slam_toolbox/feedback'),
                ('/slam_toolbox/graph_visualization','/tb3_0/slam_toolbox/graph_visualization'),
                ('/slam_toolbox/update','/tb3_0/slam_toolbox/update'),
                ('/slam_toolbox/scan_visualization','/tb3_0/slam_toolbox/scan_visualization'),
                ('/tf','/tb3_0/tf'),
                ('/tf_static','/tb3_0/tf_static'),],
    output='screen')
```

Figure 6.3: Modified SLAM Toolbox launch file with the insert of namespace

$F(x + \Delta x) \approx F(x) + J(x)\Delta x$ be a linearization of the original cost function, where $J(x)$ is the Jacobian of $F(x)$ and Δx is a correction to the vector x. This transformation allows to express the problem in the form of a linear least squares problem:

$$\arg\min_{\Delta x} \quad \frac{1}{2}||J(x)\Delta x + F(x)||^2 \tag{6.2}$$

The naive approach of solving a sequence of these problems and updating x may prove to be an ineffective method for guaranteeing convergence. In order to achieve convergence, it is necessary to introduce a trust region. This approach constitutes an optimization technique whereby the objective function is approximated by a simplified model function within a restricted area of the search space. If the model is shown to effectively reduce the true objective function within this region, the trust region is expanded. Conversely, if the model is unable to improve the objective function, the trust region is reduced, and the optimization problem is solved again with the updated region size. A basic trust region algorithm is defined by the following equation:

$$\begin{aligned} \arg\min_{\Delta x} \quad & \frac{1}{2}||J(x)\Delta x + F(x)||^2 \\ \text{s.t.} \quad & ||D(x)\Delta x||^2 \leq \mu \\ & L \leq x + \Delta x \leq U \end{aligned} \tag{6.3}$$

where μ is the trust region radius and $D(x) = J(x)^T J(x)$ is a non-negative diagonal matrix. In this context, the Levenberg-Marquardt algorithm is employed by Ceres to solve the optimization problem. The problem can be formulated as an unconstrained optimization problem of the form:

$$\arg\min_{\Delta x} \quad \frac{1}{2}||J(x)\Delta x + F(x)||^2 + \frac{1}{\mu}||D(x)\Delta x||^2 \tag{6.4}$$

In the context of the SLAM Toolbox, the Ceres Solver is employed for the purpose of correcting the poses and updating them.

6.1.2 Back-end: The `MergeMap` Node

The objective of the back-end is to merge occupancy maps from various robots. Each map is received on a specific topic, and when all the maps are available, they are merged into a single map that is then published. The merging process takes into account the coordinates of the maps, the resolution, and the overlap. The cells of a map are copied in the merged global map only if they have not already been set by the other maps. The architecture of the back-end is simple and it is composed only by the function `merge_map(maps)` and the class `MergeMapNode(Node)`

The function `merge_maps(maps)` combines multiple maps, which are occupancy grid objects, into a global map (`merged_map`), which is also an occupancy grid object. Firstly, the minimum and maximum extents (boundaries) of all maps are calculated in order to determine the overall size of the merged map, thereby ensuring that merged map is capable of fully encompassing all input maps. Secondly, the function sets the map's resolution to the smallest resolution among the input maps, thus retaining as much detail as possible. Thirdly, the merged map's data is initialized with unknown values (-1), indicating unobserved areas.

For each cell in the input maps, the function `merge_maps(maps)` calculates the corresponding position in the merged map's coordinate system. This involves translating the coordinates from the original map's frame to the merged map's frame. In the event that the corresponding cell in the merged map is unobserved, the cell data from the input map is copied over; otherwise, existing data in the merged map are given precedence. The result is a cohesive occupancy grid that includes data from all input maps, while resolving overlapping cells based on the order of processing.

The ROS2 node `MergeMapNode(Node)` is responsible for the merging of maps. It publishes `OccupancyGrid` objects on the topic `/merge_map`, and also subscribes to the various map topics.

This class presents a single method for each map, designated as `map_callback`. Once a map is received from one of the robots, it is stored in `self.maps[index]`. In the event that all maps have been received, the `merge_maps` function is invoked to merge the maps and the resulting map is published.

6.2 Path Planning

Path planning is a fundamental topic in robotics, pivotal to autonomous navigation [77]. Navigation is the process of determining and following a route for an autonomous robot to move safely from one location to another. Autonomous robots must be able to plan optimal, collision-free routes from start to destination, handling uncertainties in sensor data and interactions with obstacles.

Path planning is a category of problem that is classified as non-deterministic polynomial-time (NP-hard) [80]. The inherent degrees of freedom in the system contribute to the complexity of this category of problem. Path planning can be classified into local or global categories:

- Global path planning is concerned with determining an optimal path using comprehensive environmental information, rendering it particularly well-suited to static and fully known environments. In this scenario, the algorithm generates a complete path from the initial position to the final position before the robot begins to follow the planned trajectory. Global path planning does not consider dynamical obstacles.
- Local path planning is typically employed in situations where the environment is either unknown or dynamic. This type of planning occurs while the robot is in motion, utilizing data from local sensors to create new paths in response to changes in the environment, such as dynamical obstacles.

Consequently, a global planner determines the trajectory from a starting point to a destination, whereas a local planner is responsible for modifying the selected path to accommodate dynamic obstacles that may arise. Although an application may function without a local planner, it is only advisable to employ one in static environments, which are free of dynamic obstacles. In the context of SLAM, such as the specific application in this thesis, the environment can be assumed to be static, as dynamic obstacles do not need to be considered for the map creation process.

6.2.1 Global Planner: The A^* Algorithm

The global path planning approach is founded upon two fundamental elements: firstly, the robot's representation of the environment, which is referred to as C-space, and secondly, the selected algorithm [77]. C-space representations include Voronoi diagrams, occupancy grids, cones, quad-trees and vertex graphs. These maps indicate the locations of free and obstructed areas within the workspace. A range of algorithms is employed to explore and manipulate environmental maps, including graph search methods, genetic algorithms, potential field methods and roadmap strategies.

A variety of path planning and pathfinding algorithms exist [80]. The specific conditions under which they can be applied are dependent on a number of factors, including the kinematics of the robot, the dynamics of the surrounding environment, the availability of computational resources and the data provided by the robot's sensors.

The A^* algorithm represents a widely used method for path planning, particularly in the context of graph traversal. The A^* algorithm operates in a manner similar to that of Dijkstra's algorithm, which is employed to find the shortest path between a source vertex and all other vertices in a graph. The A^* algorithm is particularly effective in identifying near-optimal solutions based on the available dataset or node information. It is commonly used in static environments, although it has also been successfully applied in dynamic settings. A^* offers a compromise between speed and accuracy. Users can reduce the algorithm's time complexity by increasing the amount of memory allocated or alternatively, allocate more memory to conserve speed, thus ensuring that the shortest path is still identified.

The A^* algorithm is composed of three principal elements: an open list, a closed list, and a heuristic function [81]. The open list comprises nodes that have yet to be evaluated, whereas the closed list contains nodes that have already been visited. The heuristic function calculates the distance from a given node to the goal. As a best-first search (BFS) algorithm, A^* assesses each cell within the configuration space based on its associated fitness function:

$$f(n) = g(n) + h(n) \tag{6.5}$$

where $g(n)$ represents the cost from the start node to node n, whereas $h(n)$ denotes the heuristic estimate of the cost from node n to the goal. The algorithm assesses each adjacent node in relation to the current one, and identifies the one with the lowest $f(n)$ value as the point of departure for further exploration.

A^* is considered to be a computationally simpler and less resource-intensive algorithm than others, rendering it well-suited for embedded systems and resource-constrained environments [80]. A variety of heuristic functions may be utilized to facilitate the search process, including those based on the Euclidean distance ($\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$), Manhattan distance ($|x_1 - x_2| + |y_1 - y_2|$), or diagonal/octile distance ($\max |x_1 - x_2| + |y_1 - y_2|$).

Expansions distance

The traditional A^* algorithm may generate paths that are in close proximity to obstacles, which presents a significant risk of collision [81]. Expansion distance is a key factor in ensuring safety, as it involves maintaining additional space around obstacles. In the context of path planning, robots employ grid-based maps, wherein the expansion distance is applied by extending the grid around obstacles in an outward direction. In practical applications, the expansion distance is frequently set to the radius of the robot's model. This distance serves to guarantee both the dependability of the path and the minimization of travel space expended unnecessarily. Figure 6.4 illustrates the concept of expansion distance.

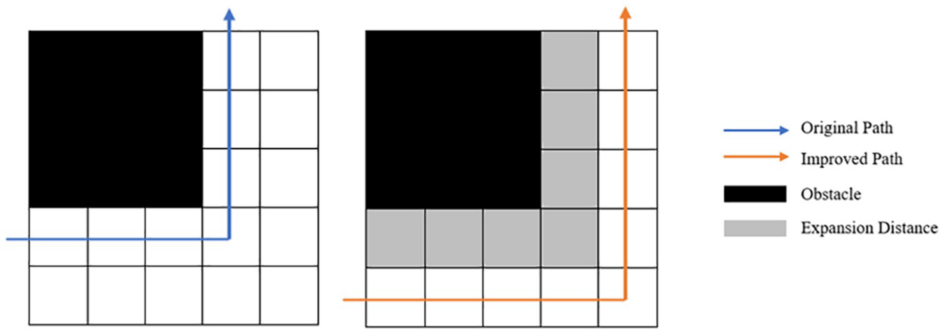

Figure 6.4: Example of expansion distance. The original path, shown in blue, is depicted on the left. The improved path, shown in orange, incorporates the concept of expansion distance, which is represented by the gray squares [81].

Path Smoothing: B-spline

The traditional A^* algorithm generates a path comprising a series of nodes interconnected by polyline segments. This type of structure presents three principal disadvantages [81]. Firstly, the algorithm priorities the minimization of path length over the reduction of sharp turns or irregular segments, frequently resulting in a compromise between smoothness and the shortest distance. Secondly, the resulting path is discontinuous. Furthermore, right-angle turns necessitate a sudden deceleration of the robot, which compromises both speed and path robustness. In order to address these issues, it is necessary to apply a smoothing process to the path produced by the conventional A^* algorithm. Figure 6.5 illustrates the discrepancy between a planned path that has been smoothed and one that has not undergone this process.

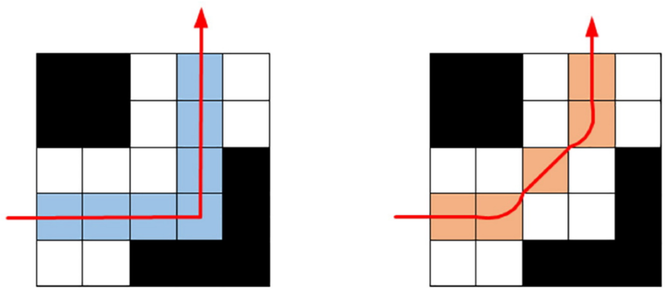

Figure 6.5: The illustration on the left depicts a path generated without the smoothing phase, whereas the path on the right was generated by a generic smoothed A^* algorithm [81].

A B-spline, also known as a basis spline, is a piecewise polynomial function. According to [82], B-splines are employed in practical applications for three main reasons: local control, flexibility, and continuity. Firstly, each segment of the curve is influenced by a limited number of control points, thereby enabling a particular part of the curve to be modified without affecting the rest. Secondly, B-spline curves facilitate the modeling of more complex and adaptive shapes. Furthermore, B-splines ensure continuity between segments. Each segment of a B-spline curve is a polynomial of degree p, which depends on $p + 1$ control points that influence the overall shape of the curve. The area over which a given segment is active is determined by the arrangement of the nodes, which are defined as $\{t_0, \ldots, t_n\}$. These nodes divide the area of the curve into sub-intervals and determine the structure of the curve. The general formula for a B-spline of degree p is:

$$C(t) = \sum_{i=0}^{n} P_i N_{i,p}(t) \tag{6.6}$$

where P_i are the control points, and $N_{i,p}$ are the basic functions. The latter can be defined recursively:

$$N_{i,0}(t) = \begin{cases} 1 & \text{if } t_i \leq t < t_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{6.7}$$

$$N_{i,p} = \frac{t - t_i}{t_{i+p} - t_i} N_{i,p-1}(t) + \frac{t_{i+p+1} - t}{t_{i+p+1} - t_{i+1}} N_{i+1,p-1}(t) \tag{6.8}$$

6.2.2 The `Exploration` Node

Path planning is performed by the `Exploration` ROS2 Node, which is the global path planner and it constitutes the core of autonomous navigation. It is designed to be performed by one or multiple robots navigating an unknown environment, as well as exploring and mapping new areas. The function takes as input a set of parameters describing the map, the robot's position, and the robot's identifier, in order to support multi-robot behavior. The algorithm is organized in various steps.

The initial step entails the invocation of the `costmap()` function, which serves to extend the boundaries of the occupancy map in order to provide a safety buffer zone around potential hazards. The `costmap()` function is employed to identify all the cells that contain walls. Subsequently, the obstacles are expanded by the specified expansion size in eight directions around each wall. The costmap is transformed into a binary map, where cells with values greater than a predefined threshold (representing obstacles or walls) are marked as inaccessible. Conversely, cells with values below the threshold are marked as accessible. The cell of the costmap corresponding to the robot's current position is set to zero, indicating that the robot is situated in a free and accessible area.

The subsequent stage of the process is the calculation of frontiers, which is performed by the function `frontier_cells()`. Frontiers represent the boundaries between areas that have been explored and those that remain unexplored or unknown. In an autonomous SLAM context, robots search for these areas with the objective of expanding the explored map, while avoiding the intrusion into areas that have not yet been explored and therefore classified as free or occupied. The identified boundaries are then organized into groups through the utilization of a depth-first search (DFS) algorithm, and finally sorted in accordance with their respective sizes.

The final phase of the process involves the formulation of a route to the nearest frontier group, taking into account the robot's current location, the map resolution, and the origin coordinates. This stage employs the A^* algorithm as a global path planner to identify the optimal subsequent exploration point. The planned path is then smoothed by the B-spline algorithm.

The `Exploration` node subscribes to the `/merge_map` topic in order to receive the global map, and the node also subscribes to the `/odom` topics. The route is designed in consideration of these two factors. Furthermore, it subscribes to velocity (`/cmd_vel`) topics, one for each robot. The computed path is delivered to the `PathTracker` ROS2 Node via the `/path` topics.

6.3 Path Tracker: The Pure Pursuit Algorithm

Pure Pursuit was introduced by Campbell in 2007 [83]. In the context of vehicle motion, the term *Pure Pursuit* is used to describe a process, in which the vehicle follows or pursues a point on a given path that is located a distance ahead of the vehicle's current position [84]. Pure Pursuit employs basic geometry principles to determine the required curvature for steering a robot to a specified point on a given path [85]. The algorithm is designed to calculate the linear and angular velocity required for a robot to follow a predefined path. In the majority of implementations, a constant velocity is used as the default setting, making this algorithm suitable for robots that do not require particularly high safety constraints. Pure Pursuit and its variants do not account for the dynamic effects of the vehicle. Since the algorithm is purely geometric, it does not incorporate vehicle dynamics into the path tracking process.

It considers a path P as an ordered sequence of points $P = \{p_0, p_1, ..., p_n\}$, where each point $p_i = (x_i, y_i)$ lies on the path. The algorithm defines a function f that calculates the linear and angular velocities required to follow a reference path P_t at time t:

$$(v_t, \omega_t) = f(P_t). \tag{6.9}$$

The algorithm first identifies the closest point p_r on P_t to the current position of the robot. Then, using a predefined lookahead distance L, it selects the lookahead point p_l as the first point p_i that is at least at L distance from p_r:

$$d(p_i) = \sqrt{(x_r - x_i)^2 + (y_r - y_i)^2} \tag{6.10}$$

$$p_l = p_i, \tag{6.11}$$

$$d(p_i) \leq L \tag{6.12}$$

Once the lookahead point p_l has been found, the curvature of the circle connecting the robot to this point can be calculated using basic geometry. In the robot's local coordinate frame P'_t, the curvature κ can be calculated as:

$$\kappa = \frac{2y'_l}{L^2} \tag{6.13}$$

where κ is the curvature required to guide the robot to the lookahead point. The key parameters of the Pure Pursuit algorithm are the robot's maximum linear speed and the lookahead distance used to select the lookahead point. In its standard formulation, the lookahead distance is tuned to achieve a balance between minimization of oscillations around the path (at shorter distances) and faster convergence to the path (at longer distances). Typically, the linear velocity is constrained when the steering angle is excessively large, a phenomenon that can be attributed to the necessity of avoiding overly acute steering.

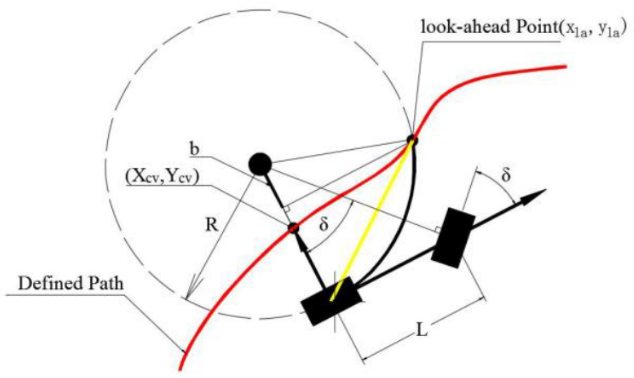

Figure 6.6: Geometric explanation of pure-pursuit. The red line represents the route that the vehicle is required to follow. The vehicle (a quadricycle) is represented by two black rectangles (i.e. the wheels). Its length is represented by L, and its distance between the rear wheels is b. (X_{CV}, Y_{CV}) is the current position of the vehicle, while (x_{la}, y_{la}) is lookahead point. R is the radius of the circle, and δ is the steering angle [84].

6.3.1 The `PathTracker` Node

The `PathTracker` node has the responsibility of managing the Pure Pursuit algorithm. In fact, it is a subscriber to the `/path` topic, from which it receives the path calculated by the `Exploration` node. Furthermore, the `PathTracker` node is a subscriber to the `/odom` topic and is a publisher to the `/cmd_vel` topic, sending to the `Exploration` node linear and angular velocity information.

Chapter 7

Experiments and Results

The following chapter presents the experimental results obtained through the approach outlined in Chapter 6. The experiments are structured in three principal phases. The first phase includes SLAM operations conducted on a guided TurtleBot3 Burger (Section 7.1), with the objective of establishing a ground truth. The second phase encompasses experiments on the AC-SLAM approach, executed in simulation (Section 7.2) and in the real world (Section 7.3). Although the AC-SLAM approach is applicable to a variable number of robots, all experiments are conducted with two robots.

7.1 Experimental Setup and Ground Truth

In the preliminary phase of the investigation, the SLAM Toolbox was selected as the front-end, and experiments were conducted with the objective of evaluating the robot's autonomous navigation abilities and its capacity to perform SLAM. Once the algorithm's correct functionality was established on a single robot, the experiments were conducted on two robots, both in simulation and on the real robot. The objective was to evaluate the effectiveness of the Active Collaborative SLAM approach.

All the simulation experiments were conducted using Gazebo, specifically within the TurtleBot3 House environment, which is depicted in Figure 7.1. This environment was selected due to its similarity to an office setting, which provides various obstacles and spaces that closely replicate real-world indoor environments. By conducting tests in this simulated environment, it was possible to assess the performance of SLAM in a controlled yet realistic setting before proceeding to physical experiments with the actual robot.

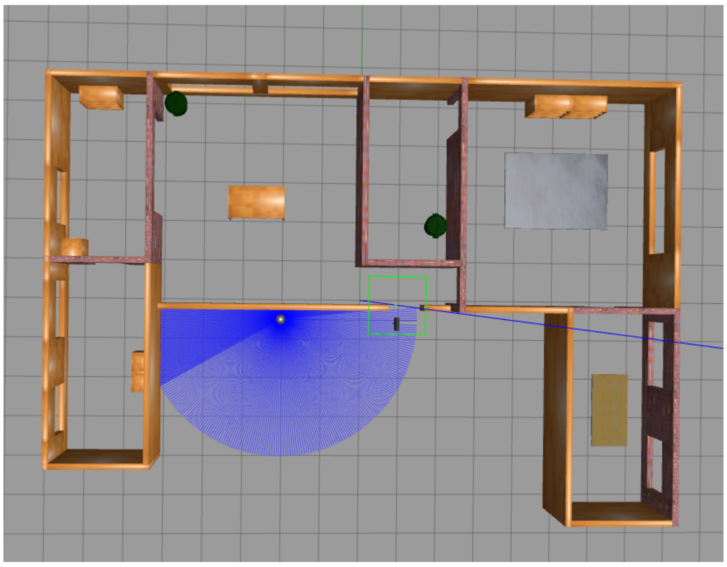

Figure 7.1: TurtleBot3 House environment in a Single-Agent scenario

All of the real-world experiments were conducted in the Robotics Laboratory and in the corridor of the Department of Electronics and Telecommunications (DET) on the third floor of Politecnico di Torino. The indoor environment is analogous to an office space, and thus exhibits structural similarities to the intended operational environment of the TurtleBot3 House. The controlled nature of the laboratory provided an ideal setting for preliminary testing with a single robot. Given the relatively limited size of the laboratory, the corridor was also explored to assess the potential benefits of collaborative mapping between two robots. This multi-agent scenario aimed to evaluate whether such collaboration could lead to a more comprehensive and efficient map of the extended environment, compared to mapping solely within the confined space of the laboratory.

Figure 7.2 presents the SLAM Toolbox ground truth of the House environment, while Figure 7.3 depicts the real-world ground truth.

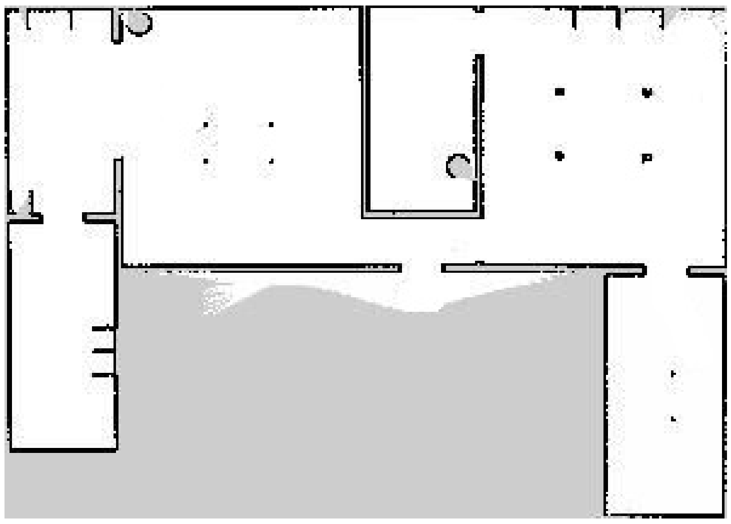

Figure 7.2: Ground truth of the TurtleBot3 House environment.

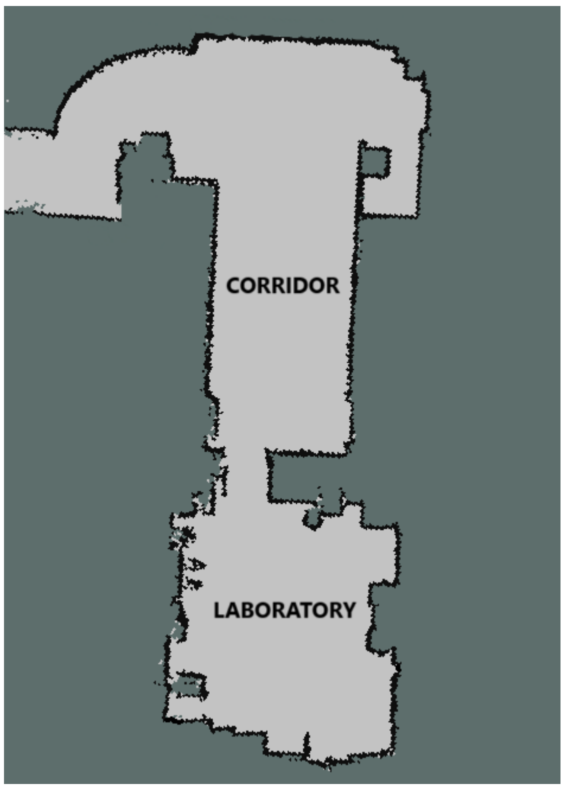

Figure 7.3: Ground truth for the real-world tests.

7.2 Simulations Experimental Results

The objective of the Active Single-robot algorithm test conducted in Gazebo was to fine-tune key parameters, namely the lookahead value (a hyper-parameter of the Pure Pursuit algorithm), the expansion size (a hyper-parameter of A^*) and the robot's maximum achievable speed. Table 7.1 presents the optimal parameter settings obtained during this tuning process. The optimal values were selected through a trial-and-error approach. Based on these optimized values, experiments involving two robots were subsequently conducted out to further assess the algorithm's performance in a multi-robot context.

Hyper-parameter	**Value**
Speed	$0.15 m/s$
Lookahead distance	$0.2 m$
Expansion size	4

Table 7.1: Optimal Values used in the Simulation tests

Figure 7.4 illustrates the initial settings of the House environments with two robots.

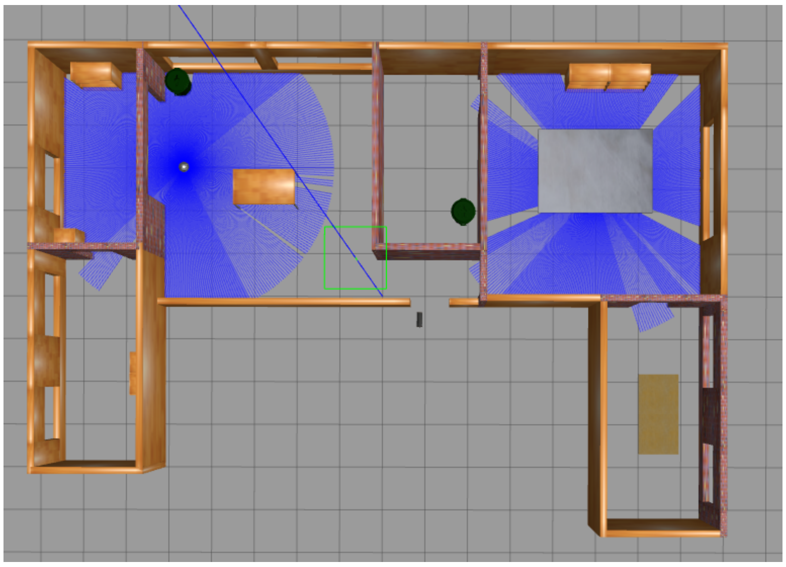

Figure 7.4: TurtleBot3 House environment in a Multi-Agent scenario

The algorithm demonstrated satisfactory performance in the simulation, enabling the autonomous navigation of a single robot within the environment without collisions with static obstacles. Additionally, it successfully executed local SLAM.

Furthermore, the map merging process also worked correctly. Two principal challenges were identified with this approach: the presence of low obstacles and the necessity to account for dynamic obstacles. An illustrative example of a low obstacle is provided in Figure 7.5. The depicted table has a base that is significantly lower than the robot's bidimensional LiDAR sensor, which precludes the robot from detecting and avoiding the obstacle. The table base is situated at a distance from the robot that is sufficient for the sensor to fail to register it. The second issue concerns dynamic obstacles, specifically the other mobile robot in this case. The algorithm does not include a local planner, as SLAM algorithms are generally designed to account for static obstacles only, not dynamic ones.

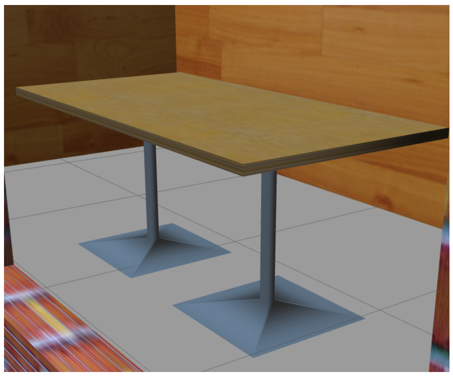
Figure 7.5: Example of a table with a low base

The two local maps are reported in Figure 7.6.

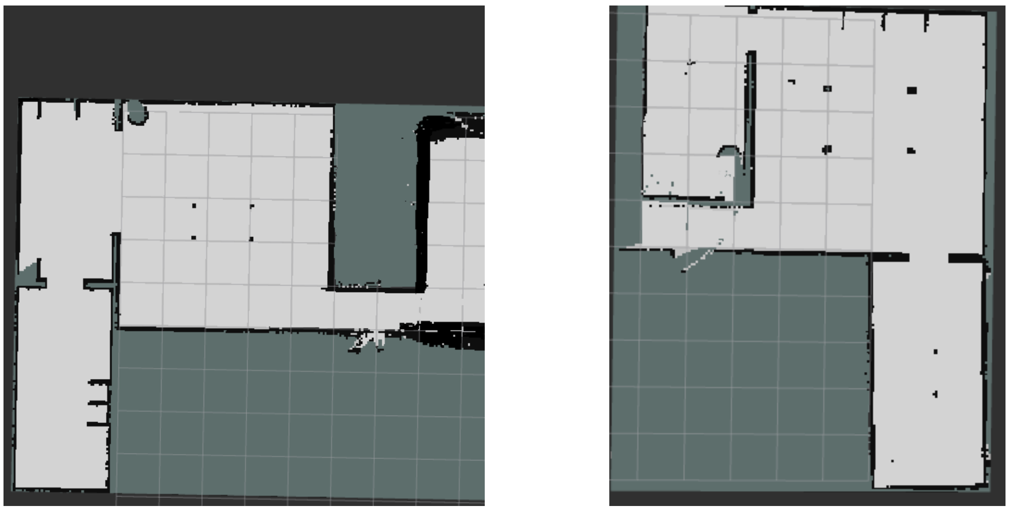
Figure 7.6: Local Maps of the House environment. On the left, the map produced by Robot 1, on the right the map generated by Robot 2.

With the exception of a minor irregularity in the map produced by Robot 1, which was caused by the extended runtime of the algorithm, resulting in the map being overwritten, both maps are of an adequate standard and accurately reflect the environment.

Figure 7.7 depicts the global map, which corroborates the comprehensive success of the experiment. This outcome substantiates the effectiveness of the mapping process, with the robots effectively capturing the pivotal characteristics of the environment. Furthermore, the global maps exhibit no substantial irregularities, thereby underscoring the advantages of a multi-agent approach over a single-agent one.

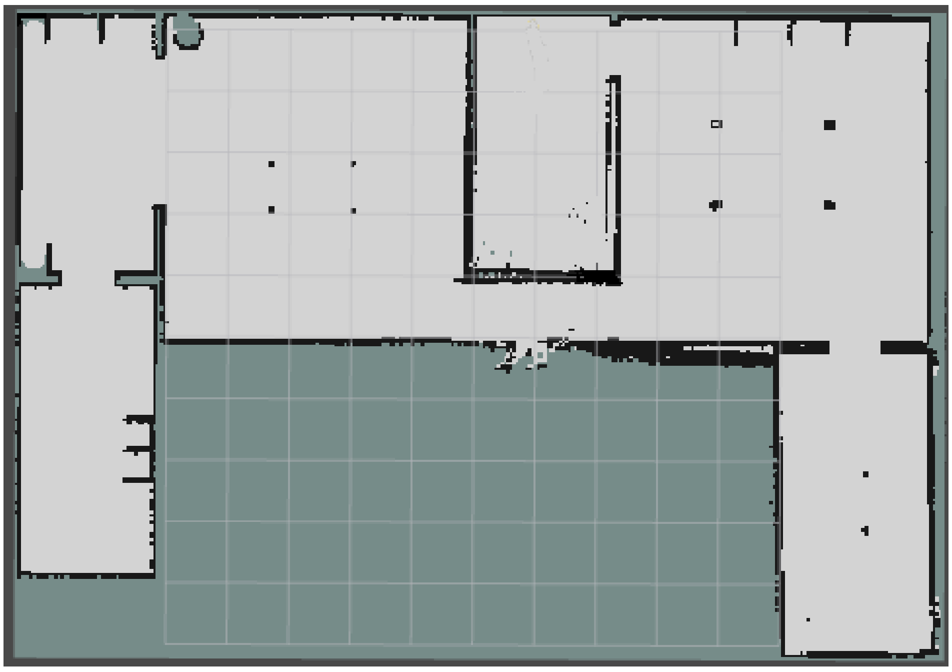

Figure 7.7: Global map of the House environment.

7.3 Real-World Experimental Results

7.3.1 Single-Agent Scenario

The transition from simulation to real-world testing was not immediate. Indeed, some factors contributed to the relatively smooth testing experience in Gazebo. These included the precisely known starting positions of the robots, the idealized shape of obstacles, and the consistently controlled environment (such as the House model). As a result, testing in Gazebo was largely free of significant issues, aside from those already discussed in the previous section.

The initial stage of the process entailed the assessment of the implemented methodology on a solitary robot in order to ascertain its operational efficacy and calibrate the hyper-parameters. The final values, established through a process of trial-and-error, are presented in Table 7.2.

Hyper-parameter	**Value**
Speed	$0.08m/s$
Lookahead distance	$0.15m$
Expansion size	4

Table 7.2: Optimal Values used in the Real-world tests

Initially, the robot's velocity was decreased. This alteration was imperative due to the incompatibility of the elevated speed utilized in Gazebo with the actual operational conditions, as the robot frequently collided with stationary obstacles.

An attempt was made to reduce the expansion size to 3. However, as anticipated, the real robot was more prone to approach obstacles in a closer manner, frequently resulting in collisions. Consequently, the expansion size was increased to 5, thereby ensuring a safe margin from obstacles. Nevertheless, due to the confined and obstacle-dense environment in which the algorithm was tested, the robot exhibited a tendency to either remain in the same area or consistently explore only certain parts, thereby avoiding navigation toward the corridor.

With regard to the lookahead distance, a reduction to 0.15 was implemented in order to facilitate the robot's ability to traverse paths that are more aligned with the characteristics of the test environment. Once the tuning phase was complete, experiments were conducted to validate the approach. The results are presented in Figure 7.8. As can be seen, the outcome closely resembles the ground truth (Figure 7.3). Therefore, the approach that utilizes only SLAM and autonomous path planning has been demonstrated to be effective.

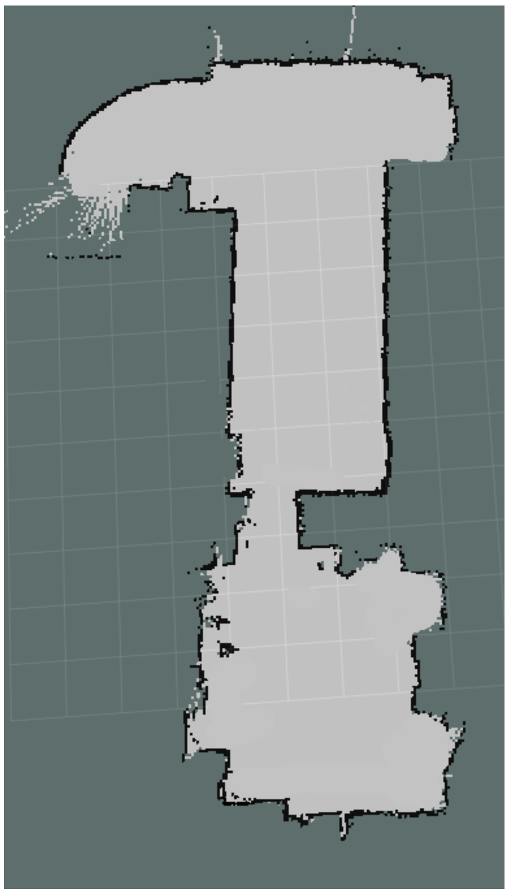

Figure 7.8: Map obtained in an Autonomous Single-Robot SLAM scenario.

The experiments confirmed both the strengths and limitations of the autonomous SLAM approach. While the ability to automate this process offers significant advantages, in terms of timing and optimality, the robot does not always make choices that would be considered ideal by a human operator. Furthermore, the time required to complete SLAM is considerably longer than in a manually controlled scenario, given the same robot speed.

7.3.2 Multi-Agent Scenario

The most significant challenge was encountered in the multi-agent scenario. While the autonomous path planning process was relatively straightforward, the initial merging of the two maps presented a significant challenge. In the simulation, the quality of map merging was consistently high, due to the ideal nature of the environment and the high positional accuracy of both robots and the origin of the reference frame. However, in the real world, the primary issue is the misalignment of the two robots' origins, which can lead to two potential issues. For instance, if the initial positions of the robots are set at a distance of approximately five to ten centimeters from each other, the maps will exhibit a degree of overlap (an illustrative example of this behaviour can be observed in Figure 7.9). Conversely, if the initial positions are farther apart, the result will be an image with two completely disconnected maps. An effective solution was to position the two robots in close proximity (approximately 4 cm apart), in a side-by-side configuration with identical orientation, and initiate the exploration process from the corridor. This configuration yielded results that can be considered satisfactory.

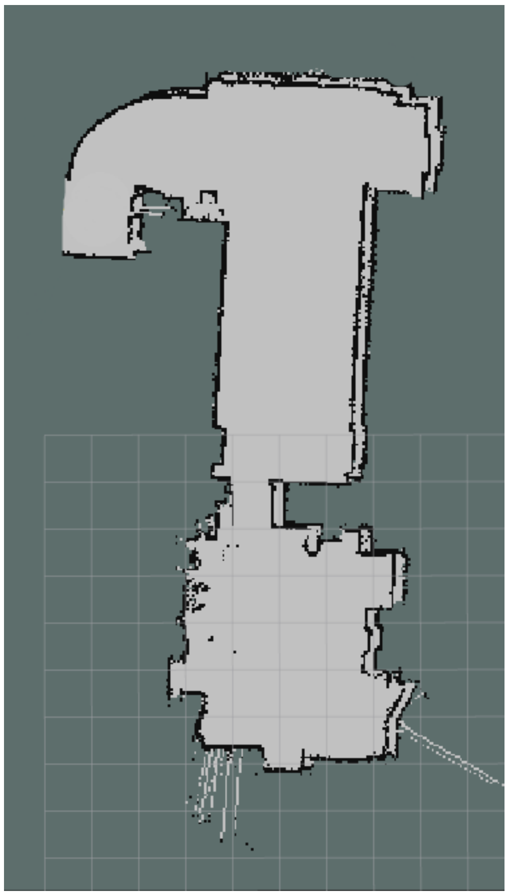

Figure 7.9: Global Map with overlaps

In order to demonstrate the capabilities of AC-SLAM in terms of both map quality and time efficiency, it would be optimal to assign one robot to explore the corridor while the other explores the laboratory. This approach would require approximately half the time of a single active SLAM approach. However, the near-identical initial conditions of the two robots and the absence of a human or artificial supervisor result in the non-deterministic nature of path selection. Consequently, the two robots occasionally traverse the same area, exploring the environment in formation. While this enhances the precision of the map quality, it does not result in any time improvement. In this scenario, the global map is of a high quality, but the two local maps are almost identical.

Despite the aforementioned issues, the experiments yielded optimal outcomes, with the two robots successfully navigating to the designated areas. Figure 7.10 depicts the local maps generated by the robots, while Figure 7.11 illustrates the finalized global map. In this test, Robot 1 explored the corridor of the DET, while Robot 2 concentrated on navigating in the laboratory.

It is noteworthy that the optimal time for a comprehensive exploration of the environment was approximately two minutes, which underscores the remarkable potential of AC-SLAM.

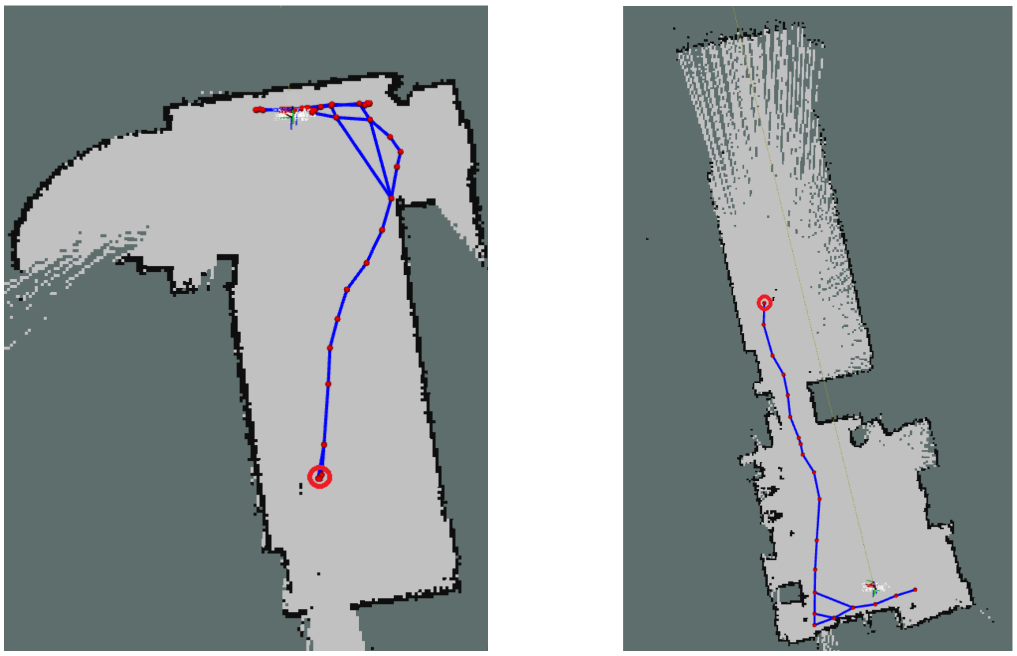
Figure 7.10: Local Maps of the AC-SLAM approach test. The corridor map, explored by Robot 1, is displayed on the left, while the laboratory map, explored by Robot 2, is displayed on the right. The blue line represents the path traversed by the robot, and the red circle indicates the robot's starting point.

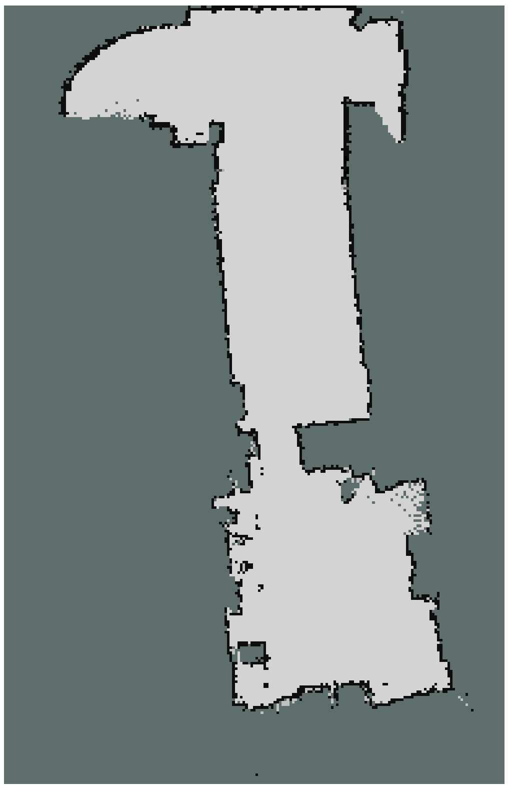

Figure 7.11: Global Map of the AC-SLAM test.

Chapter 8

Conclusions and Future Works

The objective of this thesis was to develop a ROS-based prototype solution to the problem of SLAM in the context of multi-agent cooperation. To this end, an autonomous and centralized approach was presented. The methodology was tested and validated using the ROS2 framework and the TurtleBot3 Burger platform. This was done both in simulation via the Gazebo environment and in real-world indoor environments.

The application of a multi-agent approach to the problem of SLAM represents a significant advancement over traditional single-agent SLAM. Indeed, a collaborative mapping system could provide more rapid and precise results than a single robot, thereby overcoming many of the limitations associated with the use of a single agent. In order to implement a multi-agent structure, a centralized approach has been developed. Each robot produces its local map with SLAM Toolbox, as a single-agent approach. The global map is generated by the central server back-end, which merges all the local maps into a single and comprehensive map.

The multi-agent approach presented in this thesis is distinguished by the autonomy of the robots. This is a fundamental aspect, as it enables robots to operate in uncharted environments without the need of human intervention. This autonomous approach has been developed in a centralized setting, with the central server computing a path for each robot. The A^* algorithm has been employed as global planner, while the Pure Pursuit has been used as path tracker.

Potential improvements for future research include the integration in the active part of machine learning and robot learning techniques, which may facilitate the dynamic adaptation of exploration parameters and enhance the robots' capacity for autonomous decision-making in real-time, particularly in more complex scenarios, such as an outdoor environment. Furthermore, the effectiveness of multi-agent exploration may be augmented through the incorporation of a supervisory element (either human or artificial), tasked with the oversight of exploration regions across multiple robots, with each robot designated to a single area. Indeed, the A^* algorithm is insufficient for assigning a specific area to a robot, as it relies solely on free, occupied, or unknown spaces. To enhance the optimization of path planning, a frontier-based approach could be an effective solution, as each robot is inclined to the boundaries of the known map. In fact, the lack of supervision limits the trade-off between exploration and exploitation. For example, in the results presented in Chapter 7, not all of the corridor was explored, as the robots preferred to return rather than explore an unknown area.

Further enhancement could be the inclusion of a local planner into the software architecture. In the context of the SLAM application within an office-like environment, the requirement for such a role may be limited, given that the environment in question is essentially static. Conversely, the involvement of a local planner could prove to be of paramount importance in SLAM scenarios that are more complex and realistic.

Moreover, despite the encouraging merging results observed in the simulation were promising, the transition to actual settings revealed some technical constraints and significant challenges. In the real world, minor discrepancies in the positions and orientations of the agents resulted in map misalignment, with fusion results frequently exhibiting overlaps or, in extreme cases, duplicated sections. The initial positioning of the robots can have a significant impact on the overall mission outcome. By positioning them in close proximity and with the same orientation at the outset of the mission, the probability of accumulated errors is reduced. While this technique has been demonstrated to be effective, it may be less practical in larger or more dynamic environments where robots must start from different locations or where the movements of the agents are not synchronized. Incorporating an inter-robot loop closure into the merging algorithm could prove essential in addressing these challenges.

Finally, the implementation of a decentralized architecture may prove beneficial in enhancing the performance of the system, particularly in light of the potential issues associated with a centralized approach. These include the risk of a single point of failure and the computational overhead. In this scenario, the management of mapping and data fusion would permit each agent to contribute independently to the final result, thereby reducing the system's dependency on initial configuration and the necessity for synchronization. Furthermore, this approach would enhance the scalability of the system, facilitating the addition of new robots without significant alterations to the communication network or the overall architecture.

Bibliography

[1] Sumegh Thale, Mihir Prabhu, Pranjali Thakur, and Pratik Kadam. «ROS based SLAM implementation for Autonomous navigation using Turtlebot». In: *ITM Web of Conferences* 32 (Jan. 2020), p. 01011. DOI: 10.1051/itmconf/20203201011 (cit. on pp. 1, 45).

[2] Leyao Huang. «Review on LiDAR-based SLAM Techniques». In: Nov. 2021, pp. 163–168. DOI: 10.1109/CONF-SPML54095.2021.00040 (cit. on pp. 1, 7, 8, 11–13, 22).

[3] Pierre-Yves Lajoie, Benjamin Ramtoula, Yun Chang, Luca Carlone, and Giovanni Beltrame. «DOOR-SLAM: Distributed, Online, and Outlier Resilient SLAM for Robotic Teams». In: *IEEE Robotics and Automation Letters* 5.2 (2020), pp. 1656–1663. DOI: 10.1109/LRA.2020.2967681 (cit. on pp. 1, 27).

[4] Andreas Serov, Joachim Clemens, and Kerstin Schill. «Multi-Robot Graph SLAM Using LIDAR». In: Feb. 2024, pp. 339–346. DOI: 10.1109/ICARA60736.2024.10553070 (cit. on pp. 2, 27).

[5] Julio A. Placed, Jared Strader, Henry Carrillo, Nikolay Atanasov, Vadim Indelman, Luca Carlone, and José A. Castellanos. *A Survey on Active Simultaneous Localization and Mapping: State of the Art and New Frontiers*. 2023. arXiv: 2207.00254 [cs.RO]. URL: https://arxiv.org/abs/2207.00254 (cit. on pp. 2, 33, 35, 37).

[6] Muhammad Farhan Ahmed, Khayyam Masood, Vincent Fremont, and Isabelle Fantoni. «Active SLAM: A Review on Last Decade». In: *Sensors* 23.19 (2023). ISSN: 1424-8220. DOI: 10.3390/s23198097. URL: https://www.mdpi.com/1424-8220/23/19/8097 (cit. on pp. 2, 8, 14, 26–28, 33, 35–37).

[7] Sajad Saeedi, michael trentini michael, Mae Seto, and Howard Li. «Multiple-Robot Simultaneous Localization and Mapping: A Review». In: *Journal of Field Robotics* 33 (Jan. 2016), pp. 3–46 (cit. on pp. 2, 28, 29, 31, 32).

[8] Randall Smith and Peter Cheeseman. «On the Representation and Estimation of Spatial Uncertainty». In: *The International Journal of Robotics Research* 5 (Feb. 1987). DOI: 10.1177/027836498600500404 (cit. on p. 4).

[9] H. Durrant-Whyte and T. Bailey. «Simultaneous localization and mapping: part I». In: *IEEE Robotics & Automation Magazine* 13.2 (2006), pp. 99–110. DOI: `10.1109/MRA.2006.1638022` (cit. on p. 4).

[10] Xiangdi Yue, Yihuan Zhang, and Miaolei He. *LiDAR-based SLAM for robotic mapping: state of the art and new frontiers*. 2023. arXiv: `2311.00276 [cs.RO]`. URL: `https://arxiv.org/abs/2311.00276` (cit. on pp. 4, 5, 7, 14).

[11] Weifeng Chen, Xiyang Wang, Shanping Gao, Guangtao Shang, Chengjun Zhou, Zhenxiong Li, Chonghui Xu, and Kai Hu. «Overview of Multi-Robot Collaborative SLAM from the Perspective of Data Fusion». In: *Machines* 11.6 (2023). ISSN: 2075-1702. DOI: `10.3390/machines11060653`. URL: `https://www.mdpi.com/2075-1702/11/6/653` (cit. on pp. 5, 14, 24, 26).

[12] Khalil Aloui, Amir Guizani, Moncef Hammadi, Mohamed Haddar, and Thierry Soriano. «Systematic literature review of collaborative SLAM applied to autonomous mobile robots». In: *2022 IEEE Information Technologies and Smart Industrial Systems (ITSIS)*. 2022, pp. 1–5. DOI: `10.1109/ITSIS56166.2022.10118378` (cit. on pp. 6, 25, 27).

[13] Qin Zou, Qin Sun, Long Chen, Bu Nie, and Qingquan Li. «A Comparative Analysis of LiDAR SLAM-Based Indoor Navigation for Autonomous Vehicles». In: *IEEE Transactions on Intelligent Transportation Systems* 23.7 (2022), pp. 6907–6921. DOI: `10.1109/TITS.2021.3063477` (cit. on pp. 6, 15–20).

[14] Baoding Zhou, Doudou Xie, Shoubin Chen, Haoquan Mo, Chunyu Li, and Qingquan Li. «Comparative Analysis of SLAM Algorithms for Mechanical LiDAR and Solid-State LiDAR». In: *IEEE Sensors Journal* 23.5 (2023), pp. 5325–5338. DOI: `10.1109/JSEN.2023.3238077` (cit. on pp. 7, 10–12).

[15] Xianzhe Zhao, Shiliang Shao, Ting Wang, Chuxi Fang, Jin Zhang, and Hai Zhao. «A Review of Multi-Robot Collaborative Simultaneous Localization and Mapping». In: *2023 IEEE International Conference on Unmanned Systems (ICUS)*. 2023, pp. 900–905. DOI: `10.1109/ICUS58632.2023.10318435` (cit. on pp. 7, 8, 16–18, 25).

[16] Andréa Macario Barros, Maugan Michel, Yoann Moline, Gwenolé Corre, and Frédérick Carrel. «A Comprehensive Survey of Visual SLAM Algorithms». In: *Robotics* 11.1 (2022). ISSN: 2218-6581. DOI: `10.3390/robotics11010024`. URL: `https://www.mdpi.com/2218-6581/11/1/24` (cit. on p. 8).

[17] Andréa Macario Barros, Maugan Michel, Yoann Moline, Gwenolé Corre, and Frédérick Carrel. «A Comprehensive Survey of Visual SLAM Algorithms». In: *Robotics* 11.1 (2022). ISSN: 2218-6581. DOI: `10.3390/robotics11010024`. URL: `https://www.mdpi.com/2218-6581/11/1/24` (cit. on pp. 8, 13).

[18] Andrew J Davison, Ian D Reid, Nicholas D Molton, and Olivier Stasse. «MonoSLAM: Real-time single camera SLAM». In: *IEEE transactions on pattern analysis and machine intelligence* 29.6 (2007), pp. 1052–1067 (cit. on p. 9).

[19] Georg Klein and David Murray. «Parallel tracking and mapping for small AR workspaces». In: *2007 6th IEEE and ACM international symposium on mixed and augmented reality*. IEEE. 2007, pp. 225–234 (cit. on p. 9).

[20] Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. «ORB-SLAM: a versatile and accurate monocular SLAM system». In: *IEEE transactions on robotics* 31.5 (2015), pp. 1147–1163 (cit. on p. 9).

[21] Richard A Newcombe, Steven J Lovegrove, and Andrew J Davison. «DTAM: Dense tracking and mapping in real-time». In: *2011 international conference on computer vision*. IEEE. 2011, pp. 2320–2327 (cit. on p. 9).

[22] Renato F Salas-Moreno, Richard A Newcombe, Hauke Strasdat, Paul HJ Kelly, and Andrew J Davison. «Slam++: Simultaneous localisation and mapping at the level of objects». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2013, pp. 1352–1359 (cit. on p. 9).

[23] Misha Urooj Khan, Syed Azhar Ali Zaidi, Arslan Ishtiaq, Syeda Ume Rubab Bukhari, Sana Samer, and Ayesha Farman. «A Comparative Survey of LiDAR-SLAM and LiDAR based Sensor Technologies». In: *2021 Mohammad Ali Jinnah University International Conference on Computing (MAJICC)*. 2021, pp. 1–8. DOI: `10.1109/MAJICC53071.2021.9526266` (cit. on pp. 9, 10).

[24] Michael Montemerlo. «FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem». In: *Proc. of AAAI02* (2002) (cit. on p. 10).

[25] B.L.E.A. Balasuriya, B.A.H. Chathuranga, B.H.M.D. Jayasundara, N.R.A.C. Napagoda, S.P. Kumarawadu, D.P. Chandima, and A.G.B.P. Jayasekara. «Outdoor robot navigation using Gmapping based SLAM algorithm». In: *2016 Moratuwa Engineering Research Conference (MERCon)*. 2016, pp. 403–408. DOI: `10.1109/MERCon.2016.7480175` (cit. on pp. 10, 18).

[26] Kurt Konolige, Giorgio Grisetti, Rainer Kümmerle, Wolfram Burgard, Benson Limketkai, and Regis Vincent. «Efficient Sparse Pose Adjustment for 2D mapping». In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2010, pp. 22–29. DOI: `10.1109/IROS.2010.5649043` (cit. on pp. 10, 20).

[27] Stefan Kohlbrecher, Oskar Von Stryk, Johannes Meyer, and Uwe Klingauf. «A flexible and scalable SLAM system with full 3D motion estimation». In: *2011 IEEE international symposium on safety, security, and rescue robotics*. IEEE. 2011, pp. 155–160 (cit. on p. 10).

[28] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. «Real-time loop closure in 2D LIDAR SLAM». In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 1271–1278. DOI: `10.1109/ICRA.2016.7487258` (cit. on pp. 10, 21).

[29] Steve Macenski and Ivona Jambrecic. «SLAM Toolbox: SLAM for the dynamic world». In: *Journal of Open Source Software* 6.61 (2021), p. 2783 (cit. on pp. 10, 58).

[30] Ji Zhang, Sanjiv Singh, et al. «LOAM: Lidar odometry and mapping in real-time.» In: *Robotics: Science and systems*. Vol. 2. 9. Berkeley, CA. 2014, pp. 1–9 (cit. on pp. 10, 11, 21).

[31] Tixiao Shan and Brendan Englot. «Lego-loam: Lightweight and ground-optimized lidar odometry and mapping on variable terrain». In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pp. 4758–4765 (cit. on pp. 10, 11, 21).

[32] Tixiao Shan, Brendan J. Englot, Drew Meyers, Wei Wang, Carlo Ratti, and Daniela Rus. «LIO-SAM: Tightly-coupled Lidar Inertial Odometry via Smoothing and Mapping». In: *CoRR* abs/2007.00258 (2020). arXiv: `2007.00258`. URL: `https://arxiv.org/abs/2007.00258` (cit. on pp. 10, 23).

[33] Zheng Liu and Fu Zhang. «Balm: Bundle adjustment for lidar mapping». In: *IEEE Robotics and Automation Letters* 6.2 (2021), pp. 3184–3191 (cit. on pp. 10, 11, 22).

[34] Han Wang, Chen Wang, Chun-Lin Chen, and Lihua Xie. «F-LOAM: Fast LiDAR Odometry And Mapping». In: *CoRR* abs/2107.00822 (2021). arXiv: `2107.00822`. URL: `https://arxiv.org/abs/2107.00822` (cit. on pp. 11, 21).

[35] Jiarong Lin and Fu Zhang. «Loam livox: A fast, robust, high-precision LiDAR odometry and mapping package for LiDARs of small FoV». In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 3126–3131. DOI: `10.1109/ICRA40945.2020.9197440` (cit. on pp. 11, 21).

[36] Yue Pan, Pengchuan Xiao, Yujie He, Zhenlei Shao, and Zesong Li. «MULLS: Versatile LiDAR SLAM via multi-metric linear least square». In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2021, pp. 11633–11640 (cit. on pp. 11, 22).

[37] Giseop Kim, Seungsang Yun, Jeongyun Kim, and Ayoung Kim. «SC-LiDAR-SLAM: A Front-end Agnostic Versatile LiDAR SLAM System». In: *2022 International Conference on Electronics, Information, and Communication (ICEIC)*. 2022, pp. 1–6. DOI: `10.1109/ICEIC54506.2022.9748644` (cit. on p. 12).

[38] Ji Zhang and Sanjiv Singh. «Visual-lidar Odometry and Mapping: Low-drift, Robust, and Fast». In: vol. 2015. May 2015. DOI: 10.1109/ICRA.2015.7139486 (cit. on pp. 13, 21).

[39] Pierre-Yves Lajoie, Benjamin Ramtoula, Fang Wu, and Giovanni Beltrame. «Towards Collaborative Simultaneous Localization and Mapping: a Survey of the Current Research Landscape». In: *Field Robotics* 2.1 (Mar. 2022), pp. 971–1000. ISSN: 2771-3989. DOI: 10.55417/fr.2022032. URL: http://dx.doi.org/10.55417/fr.2022032 (cit. on pp. 14, 15, 17, 18, 26, 28–31).

[40] Bill Triggs, Philip F. McLauchlan, Richard I. Hartley, and Andrew W. Fitzgibbon. «Bundle Adjustment — A Modern Synthesis». In: *Vision Algorithms: Theory and Practice*. Ed. by Bill Triggs, Andrew Zisserman, and Richard Szeliski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 298–372. ISBN: 978-3-540-44480-0 (cit. on p. 14).

[41] Arnaud Doucet, Nando de Freitas, Kevin P. Murphy, and Stuart Russell. «Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks». In: *CoRR* abs/1301.3853 (2013). arXiv: 1301.3853. URL: http://arxiv.org/abs/1301.3853 (cit. on p. 17).

[42] Zhang Xuexi, Lu Guokun, Fu Genping, Xu Dongliang, and Liang Shiliu. «SLAM Algorithm Analysis of Mobile Robot Based on Lidar». In: *2019 Chinese Control Conference (CCC)*. 2019, pp. 4739–4745. DOI: 10.23919/ChiCC.2019.8866200 (cit. on p. 18).

[43] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian D. Reid, and John J. Leonard. «Simultaneous Localization And Mapping: Present, Future, and the Robust-Perception Age». In: *CoRR* abs/1606.05830 (2016). arXiv: 1606.05830. URL: http://arxiv.org/abs/1606.05830 (cit. on pp. 18, 33, 36).

[44] Jianwei Zhao, Shengyi Liu, and Jinyu Li. «Research and Implementation of Autonomous Navigation for Mobile Robots Based on SLAM Algorithm under ROS». In: *Sensors* 22.11 (2022). ISSN: 1424-8220. DOI: 10.3390/s22114172. URL: https://www.mdpi.com/1424-8220/22/11/4172 (cit. on pp. 20, 21).

[45] Giorgio Grisetti, H Strasdat, K Konolige, and W Burgard. «g2o: A general framework for graph optimization». In: *IEEE International Conference on Robotics and Automation*. Vol. 2. 2011, p. 1 (cit. on p. 22).

[46] Sameer Agarwal, Keir Mierle, and The Ceres Solver Team. *Ceres Solver*. Version 2.2. Oct. 2023. URL: https://github.com/ceres-solver/ceres-solver (cit. on pp. 22, 60).

[47] Michael Kaess, Ananth Ranganathan, and Frank Dellaert. «iSAM: Incremental Smoothing and Mapping». In: *IEEE Transactions on Robotics* 24.6 (2008), pp. 1365–1378. DOI: 10.1109/TRO.2008.2006706 (cit. on p. 23).

[48] Michael Kaess, Hordur Johannsson, Richard Roberts, Viorela Ila, John Leonard, and Frank Dellaert. «iSAM2: Incremental Smoothing and Mapping Using the Bayes Tree». In: *International Journal of Robotic Research - IJRR* 31 (May 2012), pp. 216–235. DOI: `10.1177/0278364911430419` (cit. on p. 23).

[49] M. T. Lázaro, L. M. Paz, P. Piniés, J. A. Castellanos, and G. Grisetti. «Multi-robot SLAM using condensed measurements». In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2013, pp. 1069–1076. DOI: `10.1109/IROS.2013.6696483` (cit. on p. 25).

[50] Kirill Krinkin, Anton Filatov, and Artyom Filatov. «Modern multi-agent slam approaches survey». In: *Proceedings of the XXth Conference of Open Innovations Association FRUCT*. Vol. 776. 2017, pp. 617–623 (cit. on pp. 26, 31).

[51] Anton Filatov and Kirill Krinkin. «Multi-agent SLAM approaches for low-cost platforms». In: *2019 24th Conference of Open Innovations Association (FRUCT)*. IEEE. 2019, pp. 89–95 (cit. on pp. 26, 27, 32).

[52] Robert Castle, Georg Klein, and David Murray. «Video-rate localization in multiple maps for wearable augmented reality». In: Jan. 2008, pp. 15–22. DOI: `10.1109/ISWC.2008.4911577` (cit. on p. 26).

[53] Christian Forster, Simon Lynen, Laurent Kneip, and Davide Scaramuzza. «Collaborative monocular SLAM with multiple Micro Aerial Vehicles». In: Nov. 2013, pp. 3962–3970. DOI: `10.1109/IROS.2013.6696923` (cit. on p. 26).

[54] Patrik Schmuck and Margarita Chli. «CCM-SLAM: Robust and efficient centralized collaborative monocular simultaneous localization and mapping for robotic teams». In: *Journal of Field Robotics* 36 (Dec. 2018). DOI: `10.1002/rob.21854` (cit. on p. 26).

[55] Marco Karrer, Patrik Schmuck, and Margarita Chli. «CVI-SLAM—Collaborative Visual-Inertial SLAM». In: *IEEE Robotics and Automation Letters* 3.4 (2018), pp. 2762–2769. DOI: `10.1109/LRA.2018.2837226` (cit. on p. 26).

[56] Tianjun Zhang, Lin Zhang, Yang Chen, and Yicong Zhou. «CVIDS: A Collaborative Localization and Dense Mapping Framework for Multi-Agent Based Visual-Inertial SLAM». In: *IEEE Transactions on Image Processing* 31 (2022), pp. 6562–6576. URL: `https://api.semanticscholar.org/CorpusID:252897017` (cit. on p. 26).

[57] Zou Danping and Ping Tan. «CoSLAM: collaborative visual SLAM in dynamic environments». In: *IEEE transactions on pattern analysis and machine intelligence* (Apr. 2012). DOI: `10.1109/TPAMI.2012.104` (cit. on p. 27).

[58] Yun Chang et al. *LAMP 2.0: A Robust Multi-Robot SLAM System for Operation in Challenging Large-Scale Underground Environments.* 2022. arXiv: 2205.13135 [cs.RO]. URL: https://arxiv.org/abs/2205.13135 (cit. on p. 27).

[59] Yanjun Cao and Giovanni Beltrame. «VIR-SLAM: Visual, Inertial, and Ranging SLAM for single and multi-robot systems». In: *CoRR* abs/2006.00420 (2020). arXiv: 2006.00420. URL: https://arxiv.org/abs/2006.00420 (cit. on p. 27).

[60] Pierre-Yves Lajoie and Giovanni Beltrame. «Swarm-SLAM: Sparse Decentralized Collaborative Simultaneous Localization and Mapping Framework for Multi-Robot Systems». In: *IEEE Robotics and Automation Letters* 9.1 (Jan. 2024), pp. 475–482. ISSN: 2377-3774. DOI: 10.1109/lra.2023.3333742. URL: http://dx.doi.org/10.1109/LRA.2023.3333742 (cit. on p. 27).

[61] Shipeng Zhong, Yuhua Qi, Zhiqiang Chen, Jin Wu, Hongbo Chen, and Ming Liu. «Dcl-slam: A distributed collaborative lidar slam framework for a robotic swarm». In: *IEEE Sensors Journal* (2023) (cit. on p. 27).

[62] Cunhao Li, Peng Yi, Guanghui Guo, and Yiguang Hong. *Distributed Pose-graph Optimization with Multi-level Partitioning for Collaborative SLAM.* 2024. arXiv: 2401.01657 [cs.RO]. URL: https://arxiv.org/abs/2401.01657 (cit. on p. 27).

[63] Yewei Huang, Tixiao Shan, Fanfei Chen, and Brendan Englot. «DiSCo-SLAM: Distributed Scan Context-Enabled Multi-Robot LiDAR SLAM with Two-Stage Global-Local Graph Optimization». In: *IEEE Robotics and Automation Letters* PP (Dec. 2021), pp. 1–1. DOI: 10.1109/LRA.2021.3138156 (cit. on p. 27).

[64] Muhammad Farhan Ahmed, Matteo Maragliano, Vincent Fremont Carmine, Tommaso Recchiuto, and Antonio Sgorbissa. *Efficient Frontier Management for Collaborative Active SLAM.* 2024. arXiv: 2310.01967 [cs.RO]. URL: https://arxiv.org/abs/2310.01967 (cit. on pp. 28, 33).

[65] Juan Escobar-Naranjo, Gustavo Caiza, Carlos A Garcia, Paulina Ayala, and Marcelo V Garcia. «Applications of Artificial Intelligence Techniques for trajectories optimization in robotics mobile platforms». In: *Procedia Computer Science* 217 (2023), pp. 543–551 (cit. on pp. 33, 34).

[66] Andrea Bonci, Francesco Gaudeni, Maria Cristina Giannini, and Sauro Longhi. «Robot Operating System 2 (ROS2)-Based Frameworks for Increasing Robot Autonomy: A Survey». In: *Applied Sciences* 13.23 (2023). ISSN: 2076-3417. DOI: 10.3390/app132312796. URL: https://www.mdpi.com/2076-3417/13/23/12796 (cit. on pp. 38, 41).

[67] *ROS*. URL: `http://ros.org` (cit. on p. 39).

[68] Matti Kortelainen. «A short guide to ROS 2 Humble Hawksbill». In: (2023) (cit. on pp. 40–42).

[69] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. «Robot operating system 2: Design, architecture, and uses in the wild». In: *Science robotics* 7.66 (2022), eabm6074 (cit. on pp. 42–44).

[70] *Rviz2*. URL: `https://github.com/ros2/rviz` (cit. on p. 44).

[71] *Nav2*. URL: `https://docs.nav2.org` (cit. on pp. 45, 46).

[72] Tully Foote. «tf: The transform library». In: *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*. Open-Source Software workshop. Apr. 2013, pp. 1–6. DOI: `10.1109/TePRA.2013.6556373` (cit. on p. 47).

[73] *Turtlebot3 Burger*. URL: `https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/` (cit. on pp. 48, 49).

[74] Oleg S. Amosov and Svetlana G. Amosova. «Wheeled-Robot Orientation and Navigation Algorithm Using Visual-Inertial System». In: *2023 16th International Conference Management of large-scale system development (MLSD)*. 2023, pp. 1–4. DOI: `10.1109/MLSD58227.2023.10303768` (cit. on pp. 48, 51).

[75] Hendawan Soebhakti, Rahel Yulianti, Faiz Risi, and Yeni Pratiwi. «Obstacle Avoidance System Using LiDAR on Robot Turtlebot3 Burger». In: *Proceedings of the 5th International Conference on Applied Engineering, ICAE 2022, 5 October 2022, Batam, Indonesia*. 2023 (cit. on p. 48).

[76] José Francisco Molina Santiago, José-Armando Fragoso-Mandujano, Samuel Gómez-Peñate, Victor David Castillo González, and Francisco-Ronay López-Estrada. «Trajectory Tracking and Obstacle Avoidance with Turtlebot 3 Burger and ROS 2». In: *2023 XXV Robotics Mexican Congress (COMRob)*. 2023, pp. 93–98. DOI: `10.1109/COMRob60035.2023.10349744` (cit. on pp. 50, 51).

[77] N Sariff and Norlida Buniyamin. «An overview of autonomous mobile robot path planning algorithms». In: *2006 4th student conference on research and development*. IEEE. 2006, pp. 183–188 (cit. on pp. 56, 63, 64).

[78] Abdulkadir Ture. *multiRobotExploration-RobotArmy*. URL: `https://github.com/abdulkadrtr/multiRobotExploration-RobotArmy.git` (cit. on p. 57).

[79] *SLAM Toolbox*. URL: `https://github.com/SteveMacenski/slam_toolbox` (cit. on p. 60).

[80] Karthik Karur, Nitin Sharma, Chinmay Dharmatti, and Joshua E. Siegel. «A Survey of Path Planning Algorithms for Mobile Robots». In: *Vehicles* 3.3 (2021), pp. 448–468. ISSN: 2624-8921. DOI: 10.3390/vehicles3030027. URL: https://www.mdpi.com/2624-8921/3/3/27 (cit. on pp. 63–65).

[81] Huanwei Wang, Shangjie Lou, Jing Jing, Yisen Wang, Wei Liu, and Tieming Liu. «The EBS-A* algorithm: An improved A* algorithm for path planning». In: *PLOS ONE* 17.2 (Feb. 2022), pp. 1–27. DOI: 10.1371/journal.pone.0263841. URL: https://doi.org/10.1371/journal.pone.0263841 (cit. on pp. 64–66).

[82] GE Farin and G Farin. *Curves and surfaces for CAGD: a practical guide, Morgan Kaufmann*. 2002 (cit. on p. 66).

[83] Stefan Forrest Campbell. «Steering control of an autonomous ground vehicle with application to the DARPA urban challenge». PhD thesis. Massachusetts Institute of Technology, 2007 (cit. on p. 68).

[84] Moveh Samuel, Mohamed Hussein, and Maziah Binti Mohamad. «A review of some pure-pursuit based path tracking techniques for control of autonomous vehicle». In: *International Journal of Computer Applications* 135.1 (2016), pp. 35–38 (cit. on pp. 68, 69).

[85] Steve Macenski, Shrijit Singh, Francisco Martín, and Jonatan Ginés. «Regulated pure pursuit for robot path tracking». In: *Autonomous Robots* 47.6 (2023), pp. 685–694 (cit. on p. 68).

Acknowledgements

Questa tesi conclude il mio percorso di studi in Ingegneria. Ricordo ancora il momento in cui ho deciso di intraprendere questo cammino. Nel 2016, durante una visita d'istruzione all'Istituto Italiano di Tecnologia di Genova mi sono appassionato all'automazione e alla robotica. Questa esperienza ha inciso sulla scelta del mio percorso universitario. Oggi, dopo 2272 giorni da studente di Ingegneria, sono consapevole di aver fatto la scelta giusta: in questi sei anni ho amato studiare, imparare e mettermi alla prova.

Questa facoltà, oltre a formarmi come futuro ingegnere, è stata una grandissima palestra di vita, che mi ha insegnato a confrontarmi con sfide diverse e di difficoltà crescente. Mi sono sempre sentito spronato a migliorare, anche quando pensavo di non avere le capacità per raggiungere l'obiettivo. Un ringraziamento speciale va al Politecnico di Torino, un'università *maieutica* che mi ha messo alla prova ogni giorno, facendomi imparare tanto e preparandomi caratterialmente al mio futuro professionale. Ringrazio la professoressa Marina Indri, perché mi ha affidato con fiducia questo progetto di tesi che ho amato sin dal primo giorno. Ringrazio David, che mi ha seguito costantemente nel lavoro di tesi, supportandomi e guidandomi verso il raggiungimento dei miei obiettivi.

Vorrei dedicare questa tesi alla mia famiglia, che mi ha sostenuto nei momenti difficili e mi ha elogiato in quelli felici. Mi ha sempre permesso di essere me stesso, accompagnandomi in tutto il mio percorso di vita senza farmi mancare niente.

Infine, un pensiero speciale va a tutti i miei amici, vecchi e nuovi, che mi hanno sostenuto, incoraggiato, ascoltato e motivato in tutti questi anni di università.

Fabrizio