RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

ÉCOLE NATIONALE POLYTECHNIQUE

المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

Département d'Électronique

End of Studies Project Report

In partial fulfillment of the requirements for the State Engineer Degree in Electronics

---

Multi Agent Mobile Robot System for Active Collaborative SLAM Application

---

Aymen Meddas & Mohamed Khaled Ferroukhi

Presented and defended publicly on (15/06/2026)

Jury Members:

President:	Pr. Adel Belouchrani	ENP
Examiner:	Mr. Mohamed Oussaid Taghi	ENP
Supervisor:	Dr. Salah Bouhoun	ENP
Supervisor:	Pr. Mourad Adnane	LDCCP/NHSAST

ENP 2026

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

ÉCOLE NATIONALE POLYTECHNIQUE

المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

Département d'Électronique

End of Studies Project Report

In partial fulfillment of the requirements for the State Engineer Degree in Electronics

---

Multi Agent Mobile Robot System for Active Collaborative SLAM Application

---

Aymen Meddas & Mohamed Khaled Ferroukhi

Presented and defended publicly on (15/06/2026)

Jury Members:

President:	Pr. Adel Belouchrani	ENP
Examiner:	Mr. Mohamed Oussaid Taghi	ENP
Supervisor:	Dr. Salah Bouhoun	ENP
Supervisor:	Pr. Mourad Adnane	LDCCP/NHSAST

ENP 2026

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

ÉCOLE NATIONALE POLYTECHNIQUE





Département d'Électronique

Mémoire de Projet de Fin d'Études

Pour l'obtention du diplôme d'Ingénieur d'État en Électronique

---

Système Multi-Agents de Robots Mobiles pour une Application de SLAM Actif Collaboratif

---

Aymen Meddas & Mohamed Khaled Ferroukhi

Présenté et soutenu publiquement le (15/06/2026)

Composition du jury :

Président :	Pr. Adel Belouchrani	ENP
Examinateur :	Mr. Mohamed Oussaid Taghi	ENP
Encadrant :	Dr. Salah Bouhoun	ENP
Encadrant :	Pr. Mourad Adnane	LDCCP/NHSAST

ENP 2026

Département d'Électronique / ENP, 10 Rue des Frères OUDEK, El Harrach 16200 Alger Algérie.



In The Name of Allah, The Most Gracious, The Most Merciful

الملخص

يتناول هذا المشروع تصميمًا ومحاكاةً وتنفيذًا عمليًا لإطار عمل للتموضع ورسم الخرائط المتزامن النشط والتعاوني (AC-SLAM) للأنظمة متعددة الروبوتات، والمبني على ROS2، والذي جرى التحقق من صحته في بيئة NVIDIA Isaac Sim وعلى منصات روبوتية حقيقية. يجمع الإطار بين SLAM النشط، الذي يوجّه الاستكشاف الذاتي باستخدام مخطط اتصال مسبق، وSLAM التعاوني، حيث تستكشف عدة روبوتات البيئة في الوقت نفسه وتدمج خرائط الإشغال الخاصة بها في خريطة موحّدة. يوفّر SLAM القائم على مخطط الوضعيات (Pose-Graph SLAM) من خلال SLAM Toolbox عن طريق بناء الخرائط في الوقت الفعلي، بينما يتولى Nav2 مهام الملاحة وتجنّب العوائق. كما تُستخدم خوارزمية BFS لتقسيم مخطط الاتصال المسبق وتوزيع مناطق الاستكشاف على الروبوتات المختلفة. وقد جرى نشر النظام على روبوت تفاضلي الدفع مطوّر خصيصًا وروبوت Festo Robotino، مع تنسيقهما من خلال منصة للتحكم بالمهام.

الكلمات المفتاحية — رسم الخرائط المتزامن القائم على مخطط الوضعيات، خرائط اشغال الشبكات، الأنظمة متعددة الروبوتات، SLAM النشط، SLAM التعاوني، الاستكشاف الذاتي، ROS2، Nav2، NVIDIA Isaac Sim، دمج الخرائط، روبوت تفاضلي الدفع، روبوت متعدد الاتجاهات.

Résumé

Cette mémoire présente la conception, la simulation et l'implémentation d'un cadre de Localisation et Cartographie Simultanées Actives et Collaboratives (AC-SLAM) pour les systèmes multi-robots, développé sous ROS2 et validé dans NVIDIA Isaac Sim et matériel physique. Le cadre combine le SLAM Actif, qui guide l'exploration autonome à l'aide d'un graphe de connectivité préalable, avec le SLAM Collaboratif, où plusieurs robots explorent simultanément et fusionnent leurs cartes en une carte globale unifiée. Le SLAM par graphe de poses, via SLAM Toolbox, assure la construction de carte en temps réel, et Nav2 gère la navigation et l'évitement d'obstacles. Un partitionnement BFS du graphe de connectivité préalable attribue à chaque robot une région d'exploration. Le système est déployé sur un robot à entraînement différentiel construit sur mesure et un Festo Robotino, coordonnés via une plateforme de contrôle.

Mots-clés — SLAM par graphe de poses, Cartographie par grille d'occupation, Systèmes multi-robots, SLAM Actif, SLAM Collaboratif, Exploration autonome, ROS2, Nav2, NVIDIA Isaac Sim, Fusion de cartes, Robot à entraînement différentiel, Robot omnidirectionnel.

Abstract

This thesis presents the design, simulation, and real-world implementation of an Active Collaborative Simultaneous Localization and Mapping (AC-SLAM) framework for multi-robot systems, built on ROS2 and validated in NVIDIA Isaac Sim and on physical hardware. The framework combines Active SLAM, which guides autonomous exploration using a prior connectivity graph, with Collaborative SLAM, where multiple robots explore simultaneously and merge their occupancy maps into a unified global map. Pose-graph SLAM, via SLAM Toolbox, provides real-time map construction, while Nav2 handles navigation and obstacle avoidance. A BFS partitioning of the prior connectivity graph assigns exploration regions to each robot. The system is deployed on a custom-built differential-drive robot and a Festo Robotino, coordinated through a control platform.

Keywords — Pose-Graph SLAM, Occupancy Grid Mapping, Multi-Robot Systems, Active SLAM, Collaborative SLAM, Autonomous Exploration, ROS2, Nav2, NVIDIA Isaac Sim, Map Merging, Differential-Drive Robot, Omnidirectional Robot.

Dedication

To Mom and Dad.
Who have accompanied me every step of my life.
Who have supported me in everything I did.
And who taught me to
"Travel in a stubborn and contrary direction."

Aymen

Dedication

Mic check, one two, one two, all good? Okay let's go. You'll notice that my partner kept his dedication short and sweet. Then there's a huge one here, that's literally shows the contrast between our two personalities. I dedicate this to anyone who sat through all my terrible puns.

To my **mother**, for all your prayers and your unconditional love, even though I caused you so much trouble and made you angry more times than I can count. You never lost faith that one day I'd become a better person, and for that I am truly grateful and deeply sorry. To my **father**, who despite debating me on literally everything, never stopped supporting and standing by me. To my two little **sisters**, who have been with me from the very beginning I hope you surpass every achievement I've made. To my aunts, uncles, and cousins.

I once regreted not reaching my dream goal at first. I tried to cope by telling myself it was simply "El Mektoub." But now I see it literally as the best decision, written by Allah all along. **ENP** became my second home, and it welcomed me from the very first day. To the squad I walked through the Prepa days with, **Mehdi**, **Walid**, and **Sofiane**. I wouldn't be standing here without you though and we'll meet again soon, inch'Allah.

To the crazy ELN students; To both the older generations and the younger generations. And of course, to my classmates **Sci-Fi**, **Mesklil**, **EL BINOME**, **Saadeddine**, **Mimo**, and **Agnous**, three years but countless memories.

To my second family, the **Vision & Innovation Club**. To **Imed** who guided me in my first year. To every member of the **2024 IT department**, and to those who put their faith in me. To the **2024 Committee Board**: **Amira**, **Mazouz**, **Abdelmadjid**. To the "**Droj**", I wish you all the very best in whatever comes next. And to every VICer who ever read this.

To V Study, later to by named by a mysterious figure "Vniverse". I wasn't the most active member, but the community always welcomed me with open arms; **XR-79V**;

To 🦊🦇. I was always the troublesome one, bragging that I'm the main character, the all-knower. And yet, despite all of that, you pushed yourselves to your limits all for one purpose. You won, and you earned every bit of it. I couldn't be prouder, but at the same time, I truly apologize for every unnecessary conflict I created.

To my brother in everything but blood, **Raouf**. All these years, you always picked me as your sibling and treated me as your own. Your guidance and your help with literally everything meant more than I can put into words. I am forever in your debt.

To the people who cared the most, **Ibtihal**, **Basset**, **Ascem**, and **Houssem Eddine**. It's strange how people you meet almost by chance end up becoming the ones you can't imagine your life without. You cared in ways I didn't always know I needed, and showed up in moments I won't forget. I am grateful for every one of you.

And finally to my partner **Aymen**. This project started as YOUR ambition, a dream to carry forward a field you truly believe in. Yet you still chose someone like me, who was far from the most competent, to share that journey with you. Thank you for your patience, for everything you taught me, and for never making me feel like I didn't belong here. I hope to see your name on the biggest stages one day, inch'Allah.

HiCore

Acknowledgments

First and foremost, we would like to express our deepest and most sincere gratitude to our supervisors, **Dr. Salah Bouhoun** and **Pr. Mourad Adnane**, for their invaluable guidance, and the trust they placed in us throughout this journey. Their expertise, constructive feedback, and unwavering support were the compass that kept us on the right path from the earliest uncertainties to the final milestones. We are truly grateful for everything they offered, both technically and humanly.

We would also like to extend our heartfelt thanks to the members of the jury, **Mr. Mohamed Oussaid Taghi** and **Pr. Adel Belouchrani**, for the honor they have given us by evaluating this work. Their time, attention, and thoughtful assessment are deeply appreciated.

Our sincere appreciation goes to **Mr. Younes Laouar** and all the lab engineers at NHSAST, whose generosity in providing the necessary materials and workspace made a significant part of this work possible. Their support went beyond logistics, it was a reflection of genuine encouragement.

We are also grateful to the head of the Mechanical Department at ENP for providing the hardware components that contributed to the realization of our robots, and to ALCADWorks for the precision and professionalism with which they machined all the mechanical parts.

We would like to take this opportunity to thank all the professors of the Electronics Department at ENP who, over the past three years, shared their knowledge, their passion, and their experience with us. What we have learned in their classrooms laid the very foundation of this project.

Finally, to all our fellow ELN Department students, those who shared the long study sessions, the stressful deadlines, and the small victories along the way, thank you.

Contents

List of Tables

List of Figures

List of Abbreviations

General Introduction **17**

2 State of the Art **20**

2.1 Introduction 20

2.2 General SLAM 20

2.2.1 Probabilistic Approaches 21

2.2.2 Algorithmic SLAM 21

2.2.3 SLAM Software Frameworks 23

2.3 Multi-Agent Robot Systems 25

2.3.1 Collaborative SLAM 26

2.4 Active SLAM 29

2.4.1 Active Collaborative SLAM 30

2.5 Real-World Applications 31

2.6 Conclusion 32

3 Active Collaborative SLAM **33**

3.1 Introduction 33

3.2 Pose Graph SLAM 34

3.2.1 Mathematical Formulation of SLAM 34

3.2.2 Graph-Based SLAM Formulation . . . 37
3.3 SLAM using Connectivity Graphs as Priors . . . 40
3.4 Active Collaborative SLAM System . . . 42
3.4.1 BFS Graph Partitioning . . . 42
3.4.2 Distributed Exploration . . . 43
3.4.3 Map Merging . . . 44
3.5 Conclusion . . . 45

4 Simulation **46**
4.1 Introduction . . . 46
4.2 The Robot Operating System . . . 47
4.2.1 ROS Filesystem . . . 48
4.2.2 ROS Communication System . . . 48
4.2.3 ROS2 Tooling . . . 49
4.3 Simulation Environment . . . 50
4.3.1 NVIDIA Isaac Sim . . . 50
4.3.2 ROS2 Integration with Isaac Sim . . . 52
4.4 SLAM Toolbox . . . 55
4.4.1 SLAM Toolbox Components . . . 55
4.5 Navigation Stack . . . 57
4.5.1 Nav2 servers . . . 57
4.5.2 Costmaps . . . 58
4.5.3 Navigation Workflow . . . 58
4.6 Active Collaborative SLAM Simulation . . . 59
4.6.1 Prior Environment Graph Construction . . . 59
4.6.2 Multi-Robot ROS2 Architecture . . . 62
4.7 Conclusion . . . 65

5 Deployment on Real Robots **66**
5.1 Introduction . . . 66
5.2 First Robot: Custom Made . . . 67
5.2.1 Hardware . . . 67

5.2.2 Low-Level Layer . . . 73
5.2.3 High-Level Layer . . . 78
5.3 Second Robot: Festo Robotino . . . 86
5.3.1 Robot Description . . . 86
5.3.2 Holonomic Drive . . . 89
5.3.3 Added Extensions . . . 92
5.4 SLAM and Navigation Deployment . . . 97
5.4.1 SLAM Configuration . . . 97
5.4.2 Navigation Parameter Tuning . . . 98
5.4.3 Navigation Challenges . . . 100
5.5 Active Collaborative SLAM Deployment . . . 100
5.6 AC-SLAM Mission Control Platform . . . 101
5.6.1 Architecture Overview & ROS Bridge Integration . . . 101
5.6.2 User Interface . . . 101
5.7 Conclusion . . . 104

6 Results & Discussions **105**
6.1 Introduction . . . 105
6.2 Experimental Results and Evaluation . . . 105
6.2.1 Evaluation Metrics . . . 105
6.2.2 SLAM Performance Evaluation . . . 107
6.2.3 Navigation Performance Evaluation . . . 112
6.2.4 Comparative Analysis . . . 114
6.3 Conclusion . . . 116

Conclusion **117**

Bibliography **119**

List of Tables

2.1 Open-source SLAM software. Adapted from [7]. 23
2.2 *Awesome SLAM datasets* snippet [30] 24

4.1 Differences between ROS1 and ROS2. 47
4.2 Comparison of robotics simulation platforms. 50

5.1 Motor Specifications. 68
5.2 RPLIDAR A2M12 Specifications. 69
5.3 MPU-6050 Specifications. 70
5.4 Hardware components summary. 71
5.5 PID controller gains for the left and right motors. 75
5.6 STM32F429 Pin Mapping. 78
5.7 Motor and Gearbox Specifications. 88
5.8 Confirmed field offsets in the 144-byte REC frame. 95
5.9 RPLIDAR A2M8 Specifications. 96

6.1 SLAM evaluation metrics in simulation 110
6.2 SLAM evaluation metrics on real hardware 112
6.3 Single-robot versus multi-robot exploration performance 115

List of Figures

2.1 Key milestones of general SLAM. 21
2.2 Single-Agent System vs. Multi-Agent System Interaction. 25
2.3 Challenges of autonomous mobile robots. 26
2.4 Communication approaches in C-SLAM 27
2.5 Architecture of SLAM and A-SLAM. Adapted from [2]. 29
2.6 Mecalux's AMR Series [67] 31
2.7 Autonomous robots in a warehouse environment. 32
2.8 MedBot by Richtech Robotics. 32

3.1 SLAM Problem Illustration. Adapted from [1] 34
3.2 Pose-graph representation of a SLAM process 37
3.3 Typical graph-based SLAM system. 38
3.4 Pose graph representation with a loop-closure constraint. 39
3.5 Pose graph before and after optimization. 39
3.6 CPP-based exploration on a topo-metric graph. 40
3.7 Eulerianisation by odd-degree matching. 41
3.8 BFS partitioning of G_0 into three sub-graphs $\{G_1, G_2, G_3\}$. 43

4.1 ROS2 Humble Hawkbill Logo. 47
4.2 ROS2 workspace filesystem structure. 48
4.3 ROS 2 node interfaces: topics, services, and actions. Adapted from [74] 49
4.4 NVIDIA Isaac Sim's Interface. 50
4.5 Robot models used in Isaac Sim 51
4.6 Simple differential control action graph. 52

4.7 Action Graph for LiDAR scan publishing via the `/scan` topic. 53
4.8 Action Graph for odometry and transform tree publishing via `/odom` and `/tf`. . 53
4.9 Action Graph for velocity command subscription and differential drive controller. 54
4.10 Action Graph for velocity command subscription and holonomic controller. . . . 54
4.11 SLAM Toolbox Components Overview. 55
4.12 General architecture of the Nav2 stack. Adapted from [82] 57
4.13 Navigation workflow. 58
4.14 Floor 0 of the ENP research center reconstructed in Isaac Sim. 59
4.15 Occupancy grid map of Floor 0. 60
4.16 Prior graph for the Floor 0 environment. 60
4.17 Eulerianised prior graph for the Floor 0 environment. 61
4.18 Eulerian circuit extracted by the CPP algorithm on the prior graph. 61
4.19 ROS 2 multi-robot architecture using namespaces. 62
4.20 Multi-robot simulation environment in Isaac Sim. 63
4.21 Partitioning of the prior graph into two subgraphs. 63
4.22 Simultaneous exploration performed by two robots 64
4.23 Map merging architecture. 64
4.24 Execution flow of navigation goal handling with invalid goal skipping. 65

5.1 Custom Robot Platform Design. 67
5.2 GB37Y3530-12V-251R Gear Motor with Encoder. 68
5.3 Initial Prototype of the Platform's Wheels. 68
5.4 Final Prototype of the Platform's Wheels. 69
5.5 Computation Components. 69
5.6 RPLIDAR A2M12. 69
5.7 MPU 6050 IMU Module. 70
5.8 Power Components. 70
5.9 Diagram of the custom robot hardware. 71
5.10 First Prototype of the Platform. 72
5.11 Final Prototype of the Platform. 72
5.12 Low-level control loop architecture. 73

5.13 Two-wheeled differential drive model. 73
5.14 Operating principle of a Hall effect encoder. 75
5.15 Comparison of PID controller step responses before and after tuning. 76
5.16 STM32 CubeIDE. 76
5.17 STM32CubeIDE pin configuration view for the STM32F429ZITx. 77
5.18 High-level architecture for the custom robot. 78
5.19 Docker Compose Architecture. 79
5.20 UART Communication between High-Level and Low-Level Layers. 80
5.21 TF tree structure following REP 105 for mobile robot navigation 85
5.22 LiDAR laser scan visualization in RViz 85
5.23 Festo Robotino Platform. 86
5.24 Robotino Full Architecture, Adapted from [87]. 87
5.25 Mechanical Diagrams of the Robotino. 87
5.26 GR 42x25 DC Motor with PLG 42 S Planetary Gearbox. 88
5.27 Robotino Wheel. 88
5.28 Robotino View Interface on Computer. 89
5.29 Kinematic schematic of the three-wheeled omnidirectional robot platform. 90
5.30 Established architecture for the Robotino. 92
5.31 Robotino View block diagram implementing the GPIO velocity command decoder. 93
5.32 Wireshark packet capture. 94
5.33 Annotated hex dump of a 144-byte REC frame. 94
5.34 RPLIDAR A2M8. 96
5.35 Final assembled Robotino platform. 96
5.36 RPLidar A2 frame. 97
5.37 Goal tolerance effect on robot navigation. 98
5.38 Inflation radius effect on robot navigation. 99
5.39 AC-SLAM platform architecture. 101
5.40 AC-SLAM Mission Control Platform Interface. 102
5.41 Map visualization panel. 102
5.42 Teleoperation mode of the AC-SLAM platform for Robot 1 and Robot 2. 103
5.43 Partitioner mode of the AC-SLAM platform. 104

6.1 Map generated during the single-robot simulation experiment 108

6.2 Map generated during the single-robot real-world experiment 108

6.3 Individual maps generated by the two robots in the simulation environment . . . 109

6.4 Merged map generated in simulation 109

6.5 Overlap between the generated map in the simulation and the ground truth map, where the green indicates the ground truth and the red the generated map with slam 110

6.6 Individual maps generated by the two robots in the real environment 111

6.7 Merged map generated in the real environment 111

6.8 Overlap between the generated map in the real world and the ground truth, where the green indicates the ground truth and the red the generated map with slam 112

6.9 Navigation path replanning after obstacle detection 113

List of Abbreviations

SLAM	Simultaneous Localization and Mapping
AC-SLAM	Active Collaborative SLAM
A-SLAM	Active SLAM
C-SLAM	Collaborative SLAM
EKF	Extended Kalman Filter
EKF-SLAM	Extended Kalman Filter SLAM
AMR	Autonomous Mobile Robot
ROS2	Robot Operating System 2
Nav2	Navigation 2
BFS	Breadth-First Search
TF2	Transform Library 2
DDS	Data Distribution Service
IMU	Inertial Measurement Unit
LiDAR	Light Detection and Ranging
GPU	Graphics Processing Unit
CPU	Central Processing Unit
GPIO	General-Purpose Input/Output
PWM	Pulse-Width Modulation
UART	Universal Asynchronous Receiver-Transmitter
USART	Universal Synchronous/Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
TCP	Transmission Control Protocol
PID	Proportional-Integral-Derivative
RPM	Revolutions Per Minute
HAL	Hardware Abstraction Layer
REST	Representational State Transfer
YAML	Yet Another Markup Language

General Introduction

Mobile robotics has progressively shifted from single-platform deployments toward coordinated fleets of autonomous agents capable of jointly perceiving, mapping, and navigating shared environments. At the core of this transition lies Simultaneous Localization and Mapping (SLAM), the process by which a robot builds a representation of an unknown environment while simultaneously estimating its own pose within it. While SLAM has reached a considerable level of maturity for single-robot systems, extending it to teams of robots introduces new challenges related to communication, map consistency, and coordinated decision-making.

This thesis addresses these challenges by designing, simulating, and deploying a **Multi-Agent Mobile Robot System for Active Collaborative SLAM (AC-SLAM)**. The proposed system enables multiple robots to autonomously explore an unknown environment, build individual maps, merge them into a consistent global representation, and actively plan their trajectories to minimize mapping and localization uncertainty, which is all validated through both simulation and real hardware deployment.

Motivations

The growing demand for autonomous mobile robots in warehouses, hospitals, and other semi-structured environments has driven the need to reduce operational costs, increase throughput, and remove humans from repetitive or hazardous tasks. Fleets of cooperating robots are increasingly preferred over isolated platforms, as coordinated teams cover larger areas faster and provide redundancy against individual failures.

Despite this, the vast majority of Active Collaborative SLAM research remains confined to simulation. Real-world validation is rare, making replication and extension by the broader community difficult. The gap between simulation and reality remains a key challenge, as algorithms that perform well in simulation often struggle with sensor noise, communication delays, and real-world robot imperfections. So this thesis is motivated by two aims:

- **Real-world realisation:** To design, build, and instrument physical robot platforms capable of Active Collaborative SLAM and to validate that the full approach operates reliably on these real, heterogeneous systems, not only in simulation.
- **Platform-Independent Modularity:** To design the Active Collaborative SLAM pipeline such that it is decoupled from any specific hardware, making it deployable on any mobile robot platform that satisfies a minimal set of requirements.

Problem Context

Simultaneous Localization and Mapping (SLAM) is one of the most fundamental problems in mobile robotics, enabling a robot to build a map of an unknown environment while estimating its own position within it. It's important in every autonomous navigation application, from warehouse logistics to search and rescue. Despite its maturity, classical SLAM remains reactive: the robot moves and updates its map, but has no mechanism to decide *where to go next*. Active SLAM closes this loop by coupling mapping with deliberate exploration decisions, guiding the robot toward areas that maximally reduce map uncertainty.

Single-robot SLAM is also inherently fragile. Odometry drift and sensor noise can corrupt the estimated trajectory in ways a single platform cannot self-correct [1]. Collaborative SLAM addresses this by distributing the task across multiple robots, whose merged observations cover larger areas faster and allow individual errors to be corrected through inter-robot loop closures.

Most existing Active Collaborative SLAM solutions are, however, evaluated purely in simulation, where idealized sensors and perfect communication mask the difficulties of real deployment [2]. Exploration strategies built on a prior connectivity graph are rarely tested beyond a single simulated platform, and map partitioning, region assignment, and map merging are often left theoretically sketched rather than implemented.

This thesis tackles this gap directly. A multi-robot AC-SLAM pipeline is built and validated in NVIDIA Isaac Sim, using ROS2, SLAM Toolbox for pose-graph mapping, and Nav2 for navigation, with a prior environment graph partitioned via BFS to assign regions and merge occupancy maps. The same pipeline is then deployed on two heterogeneous physical platforms coordinated through a dedicated AC-SLAM Mission Control Platform.

Objectives and Contributions

This work aims to bridge the gap between theoretical Active Collaborative SLAM and real-world deployment by designing a complete, modular, and reproducible multi-robot system built entirely on open-source software. The following objectives and contributions structure this effort.

1. Design and build two physical robot platforms capable of autonomous SLAM and navigation: a custom-built differential-drive robot and an adapted Festo Robotino.
2. Develop a multi-robot Active Collaborative SLAM pipeline using ROS2, SLAM Toolbox, and Nav2, validated first in NVIDIA Isaac Sim before deployment on physical hardware.
3. Build a dedicated AC-SLAM Mission Control Platform for monitoring and coordinating the robot fleet in real time.
4. Ensure full portability and reproducibility by relying exclusively on open-source software, containerized via Docker for cross-platform deployment.
5. Release the entire system from software stack, configuration files, and documentation as an open-source project on GitHub, making the framework freely accessible, reproducible, and extensible by the wider robotics community.

Thesis Structure

The thesis is structured into five main chapters:

Chapter 1 reviews the state of the art in SLAM, from probabilistic and algorithmic approaches to AI-based methods, and introduces multi-agent systems, Collaborative SLAM, Active SLAM, and their combination.

Chapter 2 develops the theoretical foundations of the proposed system, including the graph-based SLAM formulation, connectivity graphs as exploration priors, BFS-based partitioning, and map merging.

Chapter 3 presents the simulation environment built around ROS2 and NVIDIA Isaac Sim, detailing the SLAM Toolbox and Nav2 configurations and the multi-robot AC-SLAM implementation.

Chapter 4 describes the deployment on two physical platforms, a custom-built robot and a Festo Robotino, covering hardware, software layers, and the AC-SLAM Mission Control Platform.

Chapter 5 presents the experimental results, evaluating SLAM and navigation performance in simulation and real-world conditions, and comparing collaborative versus single-robot exploration.

Chapter 2

State of the Art

2.1 Introduction

Simultaneous Localization and Mapping (SLAM) consists of the concurrent construction of a representation of the environment (mapping) and the estimation of the robot's pose within it (localization). SLAM relies on sensor data acquired from both proprioceptive sensors, such as wheel odometry and inertial measurement units (IMU) and exteroceptive sensors, including cameras, LiDAR and radar systems.

The central role of SLAM in robotics research stems from the fact that robot poses are rarely known in practical applications. While technologies such as differential GPS or motion capture systems can provide accurate localization, they are often costly and limited to constrained environments, making them unsuitable for large-scale or indoor deployments. Over the past decades, the SLAM community has made significant progress, evolving from early probabilistic formulations to robust and scalable solutions capable of operating in complex real-world scenarios. More recently, research has also expanded toward collaborative approaches, where multiple robots share information to improve mapping efficiency and scalability, further accelerating the transition of SLAM into industrial applications.

This chapter presents the evolution of SLAM from classical problem formulations to modern algorithmic and AI-based approaches. It also explores the extension of SLAM toward collaborative and active paradigms, highlighting how these approaches improve efficiency, autonomy and scalability in multi-robot systems.

2.2 General SLAM

The development of SLAM is divided in three stages: classical age, algorithmic analysis, and robust perception [3].

- **Classical Age (1989-2004):** placed the theoretical framework foundations and mathematical model of SLAM.
- **Algorithmic Age (2004-2015):** focused on observability, consistency, convergence, and algorithm efficiency of SLAM.

- **Robust Pereception Age (2015-present):** focuses on the robustness of SLAM and ability to understand high-level environmental information.

2.2.1 Probabilistic Approaches

In the earliest works, the SLAM problem was formulated as a **state estimation problem** which was instigated by Smith and Cheeseman's work in 1986 [4] and further developed by Durrant-Whyte (1988) [5], focusing on the representation and propagation of spatial uncertainty. Then in 1991, Leonard et al.[6] introduced one of the first probabilistic implementations, employing an Extended Kalman Filter (EKF) to estimate the robot pose while tracking geometric landmarks.

The formalization of the SLAM problem and the introduction of the term "SLAM" appeared in a robotics survey presented at the 1995 International Symposium on Robotics Research [7]. This led to the development of EKF-SLAM in the early 2000s [8], which became the first widely adopted solution. However, EKF-SLAM suffers from scalability limitations due to its quadratic computational complexity [9].

To address these limitations, Montemerlo et al. [10] introduced the FastSLAM algorithm, based on Rao-Blackwellized particle filters. This approach represented a major conceptual shift by separating robot trajectory estimation from landmark estimation, significantly improving scalability.

2.2.2 Algorithmic SLAM

A key disadvantage of filter-based SLAM techniques is that sensor data is processed sequentially and then discarded, making it impossible to revisit past observations during map building. This limitation reduces the ability to achieve globally consistent maps and has driven the development of more robust algorithmic SLAM methods that retain and jointly optimize data over time.

Within this evolving landscape, SLAM algorithms can be broadly categorized based on the sensors they use, with the two primary classes being **visual SLAM** and **LiDAR SLAM**. We can see in figure 2.1 that the path to SLAM divided into LiDAR-based (blue blocks) and visual SLAM (orange blocks).

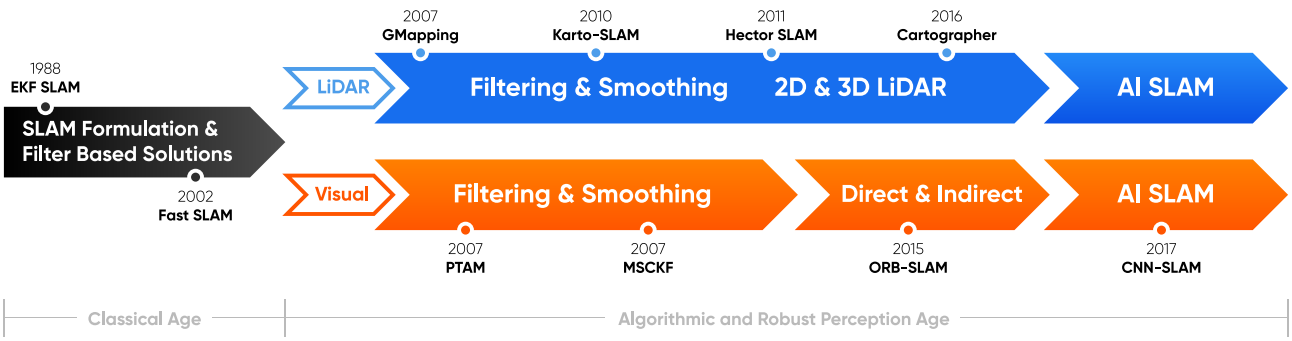


Figure 2.1: Key milestones of general SLAM.

Filtering-Based Methods

While filtering approaches were introduced in the probabilistic era, the algorithmic age saw their extension to adapt for sensor modalities giving rise to a new generation of filtering methods.

In SLAM, filtering methods use a mathematical model of the system and sensory information to estimate the state of the system at a particular point in time. Many algorithms were developed such as Parallel Tracking and Mapping (PTAM) in 2007 [11] and the Multi-State Constraint Kalman Filter (MSCKF) [12] for Visual SLAM. In LiDAR SLAM, GMapping relies on particle filters combined with scan matching techniques [13].

Scan matching techniques are often integrated within filtering frameworks to improve pose estimation in larger environments. Olson introduced the real-time correlation-matching algorithm (CSM) [14], where matching real-time frames is formulated as an optimal pose search within a restricted range. Similarly, Kohlbrecher et al. proposed the Hector SLAM algorithm [15], which uses a bilinear difference method to decompose probability values into grid maps. This approach was later improved by incorporating inertial measurement unit (IMU) data, allowing for more accurate pose estimation and improved matching performance [16].

Smoothing and Graph-Based Methods

In contrast to filtering approaches, smoothing-based methods estimate the entire robot trajectory by optimizing a global objective function over multiple poses and observations [17]. These methods improve global consistency by considering all measurements jointly rather than incrementally.

Graph-Based SLAM is a prominent realization of smoothing techniques, where the SLAM problem is modeled as a graph of robot poses connected by spatial constraints. The work of Thrun and Montemerlo (2006) [18] demonstrated the effectiveness of this formulation, enabling the use of optimization techniques such as Pose Graph Optimization (PGO) to minimize global trajectory errors. Approaches such as Karto-SLAM [19] and later Cartographer [20] adopt graph-based optimization techniques, enabling improved scalability and global consistency through loop closure detection and pose graph optimization.

Beyond algorithmic development, a growing line of research addresses the computational demands of deploying SLAM on resource-constrained platforms. [21] present an FPGA-based SoC implementation of SLAM using OpenCL, achieving nearly twice the speed of a pure software approach while meeting real-time constraints.

Direct and Indirect Visual SLAM

Visual SLAM methods can be broadly classified into direct and indirect (feature-based) approaches, depending on how visual information is processed.

Indirect methods rely on extracting and matching distinct image features, such as keypoints and descriptors, to estimate camera motion [17]. These approaches first detect salient features in the images and then establish correspondences between consecutive frames to compute the relative pose. In contrast, direct methods use the raw pixel intensity values of images without relying on explicit feature extraction [22]. They estimate motion by minimizing the photometric error between consecutive frames.

A notable example of an indirect method is ORB-SLAM, introduced by Mur-Artal et al. [23]. ORB-SLAM utilizes Oriented FAST and Rotated BRIEF (ORB) features, known for their efficiency and robustness in real-time applications. It combines feature-based tracking with optimization techniques such as Bundle Adjustment to achieve high accuracy in pose estimation and mapping. Recent work has also explored hybrid approaches; H-SLAM [24] combines feature-based and photometric information to improve robustness and efficiency while maintaining competitive accuracy.

AI-Based SLAM

Advancements in deep learning have led researchers to explore replacing traditional point cloud matching and loop closure detection with convolutional neural networks (CNNs) [25]. For instance, Jiaxin Li et al. converted consecutive two-frame data into polar coordinates as input and proposed a multi-layer CNN, ConvNet, which outputs both matching pose and loop closure detection results [26].

Furthermore, through reinforcement learning algorithms, the robots can adapt their navigation techniques dynamically since they can learn from using the algorithms in real situations [27].

Incorporating AI with SLAM allows to deal with uncertainties of the sensor measurements more efficiently to provide better navigation techniques in different applications such as automobiles and robots moving in unstructured environments [28].

2.2.3 SLAM Software Frameworks

To support the development and evaluation of SLAM algorithms, many software implementations have been made publicly available. These frameworks vary in sensor support and programming environment. Early tools were primarily MATLAB-based for simulation, while more recent implementations in C++ target real-time deployment on physical robots. Table 2.1 provides a summary of representative open-source tools.

Table 2.1: Open-source SLAM software. Adapted from [7].

Author	**Description**	**Link**
Kai Arras	CAS, a MATLAB simulation toolbox for robot localization and mapping.	https://openslam-org.github.io/cas-rnt.html
Tim Bailey	MATLAB simulators for EKF-SLAM, UKF-SLAM and FastSLAM 1.0 and 2.0.	https://openslam-org.github.io/bailey-slam.html
Andrew Davison	Scene, a C++ library for map-building and localization. Facilitates real-time single camera SLAM.	http://www.doc.ic.ac.uk/~ajd/Scene/index.html
José Neira	MATLAB EKF-SLAM simulator that demonstrates joint compatibility branch-and-bound data association.	http://webdiis.unizar.es/~neira/software/slam/slamsim.htm

OpenSLAM

An initiative called OpenSLAM aimed at promoting open-source implementations of SLAM algorithms within the robotics community [29]. Established in 2006, it provides a platform where researchers can share and access publicly available SLAM software, facilitating reproducibility and comparison between different approaches. Over time, OpenSLAM has become a reference repository for widely used implementations and it is currently maintained through a GitHub-based infrastructure. The project was initiated by Cyrill Stachniss, Udo Frese and Giorgio Grisetti.

Many of the algorithms mentioned above like GMapping or ORB-SLAM are available on the OpenSLAM platform.

Benchmark datasets

In addition, several benchmark datasets have been collected from real sensors operating in real-world environments. Benchmark datasets are essential for evaluation of SLAM systems, as they allow different methods to be compared under identical conditions. Table 2.2 presents a snippet of the "*Awesome SLAM datasets*" [30] which is an open-source list of widely used datasets in the SLAM community.

Table 2.2: *Awesome SLAM datasets* snippet [30]

Author	**Description**	**Link**	**Year**
Radish	Indoor datasets from multiple large-scale environments.	http://radish.sourceforge.net/	2003
MRPT-Malaga	Urban driving dataset collected with a multi-sensor car.	https://www.mrpt.org/robotics_datasets	2014
NCLT Dataset	Long-term campus dataset with vision and LiDAR from a Segway robot.	https://robots.engin.umich.edu/nclt/	2016
Cartographer Dataset	2D LiDAR backpack data collected at the Deutsches Museum.	https://google-cartographer-ros.readthedocs.io/en/latest/data.html	2016
Symphony Lake	121 surveys of a lakeshore over three years in Metz, France.	https://dream.georgiatech-metz.fr/datasets/#symphony-lake-dataset	2017
NTU VIRAL	Multi-modal UAV dataset with dual LiDAR, stereo cameras, IMU, and UWB ranging.	https://ntu-aris.github.io/ntu_viral_dataset/	2021

2.3 Multi-Agent Robot Systems

Mobile robots operating as part of a coordinated group form what is known as a Multi-Agent Robot System (MARS), a concept that combines the physical mobility of Multi-Robot Systems (MRS) with the distributed intelligence of Multi-Agent Systems (MAS) [31]. Research interest in such systems has grown steadily since the late 1980s, driven by the recognition that cooperative agents can collectively solve problems that are intractable for a single platform [31].

Multi-Agent Systems (MAS)

MAS is a branch of distributed artificial intelligence that sits at the intersection of AI and distributed computing [32]. It consists of a set of intelligent agents that communicate with one another and with their environment to collectively achieve goals that would be difficult or impossible for a single agent.

As illustrated in Figure 2.2, a single-agent system relies on one agent to centrally handle all interactions with the user, actions and data. A multi-agent system, by contrast, distributes these responsibilities across several interconnected agents that communicate bidirectionally, enabling parallel execution and improved fault tolerance.

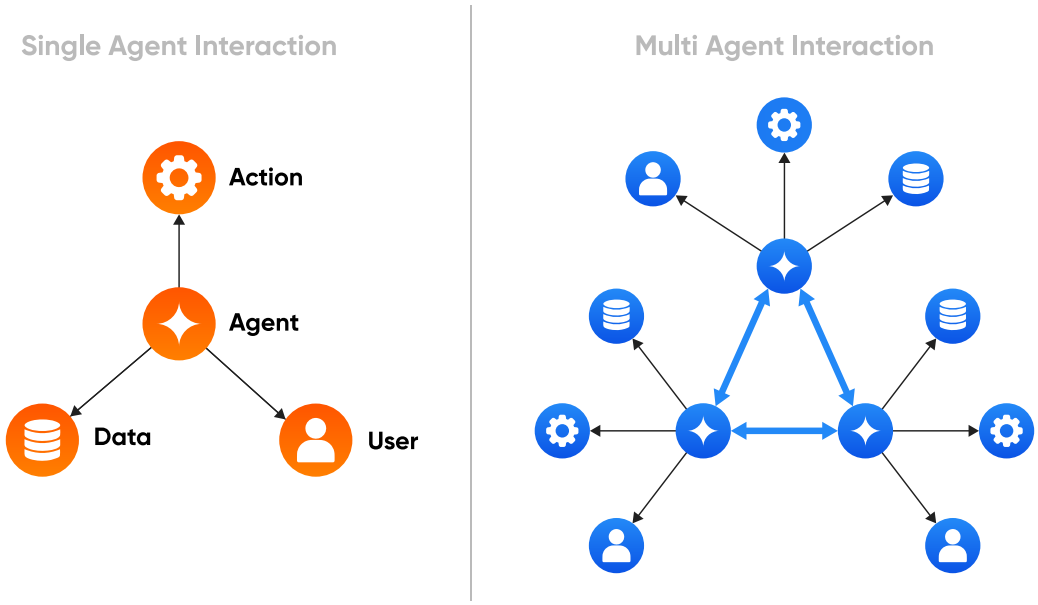


Figure 2.2: Single-Agent System vs. Multi-Agent System Interaction.

There are two principal architectures on how agents are organized, process data make decisions [33]:

- **Centralized**: All data is processed by a single leader agent which coordinates all others.
- **Decentralized**: Removes any central authority, allowing agents to act and process data based on local information alone.

Beyond robotics, MAS has found applications in e-learning platforms [34], cyber-physical systems [35] and IoT environments [35], demonstrating its versatility as a coordination framework.

Multi-Robot Systems (MRS)

MRS extends the concept of a single autonomous robot to a team of physically embodied agents that navigate, sense and act in shared environments [36]. Two broad categories of mobile robots are distinguished in the literature:

- **Autonomous Guided Vehicles (AGVs)**: follow pre-defined routes using physical guidance and are well-suited to repetitive, structured tasks such as assembly line logistics, but lack the decision-making capability to adapt to changing environments [37].
- **Autonomous Mobile Robots (AMRs)**: equipped with intelligent control systems that allow them to perceive their surroundings, plan routes dynamically and operate safely in unstructured or cluttered spaces [38].

Challenges in Multi-Agent Settings

Despite their promise, autonomous mobile robots operating in multi-agent settings face significant challenges. The most fundamental of these include localization, navigation, path planning and inter-robot communication [39].

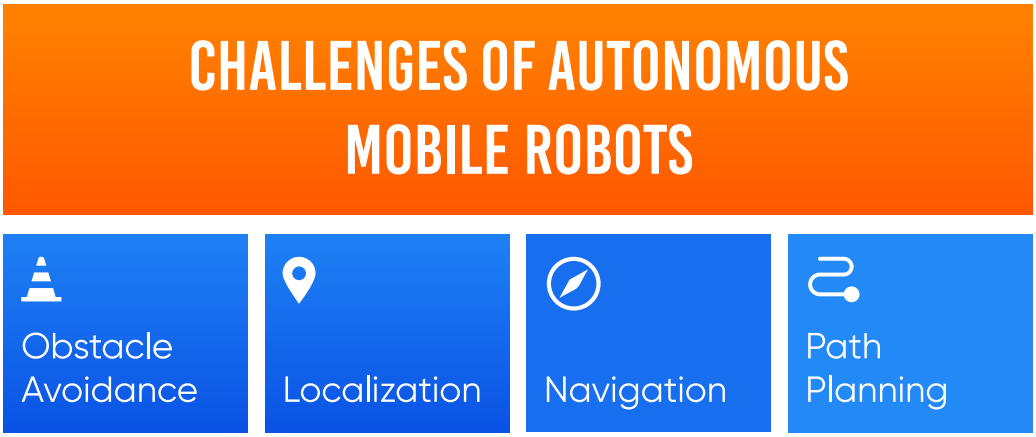


Figure 2.3: Challenges of autonomous mobile robots.

Path planning becomes particularly complex in multi-robot scenarios, as each robot must find collision-free trajectories in an environment that is continuously modified by the movement of its teammates. Communication between agents further compounds these difficulties, as intermittent or delayed data transmission can degrade coordination performance and lead to inconsistent system states.

In dynamic or unstructured environments, the problem is further aggravated by the fact that robots cannot always achieve precise localization, making reliable pose estimation a persistent open challenge [40]. Reliable localization in unknown or dynamic environments remains a critical prerequisite for any higher-level coordination. Without it, robots cannot reason about their position relative to one another or to the environment.

2.3.1 Collaborative SLAM

The challenges outlined above naturally lead to the integration of SLAM into multi-robot frameworks. One way of mapping a large-scale environment is to deploy multiple robots performing

SLAM simultaneously, dividing the scenario into smaller areas each mapped by a different robot.

This approach is called Collaborative Simultaneous Localization and Mapping (C-SLAM), also known as multi-robot SLAM.

Communication in C-SLAM

Communication is a fundamental component of collaborative SLAM, as robots must exchange information to build a consistent global map.

Early collaborative SLAM systems relied on centralized architectures, where all robots transmit raw sensor data to a central unit for processing. For example, Thrun [41] considered sharing laser and odometry data among robots. While this simplifies data fusion, it introduces a single point of failure and limits scalability.

More recent approaches adopt decentralized strategies, where robots perform local computations and exchange only selected information. Birk et al. [42] proposed map merging techniques to align and fuse independently built maps, reducing communication load but introducing challenges in map consistency. The choice of the method depends on several factors such as the application and the available resources.

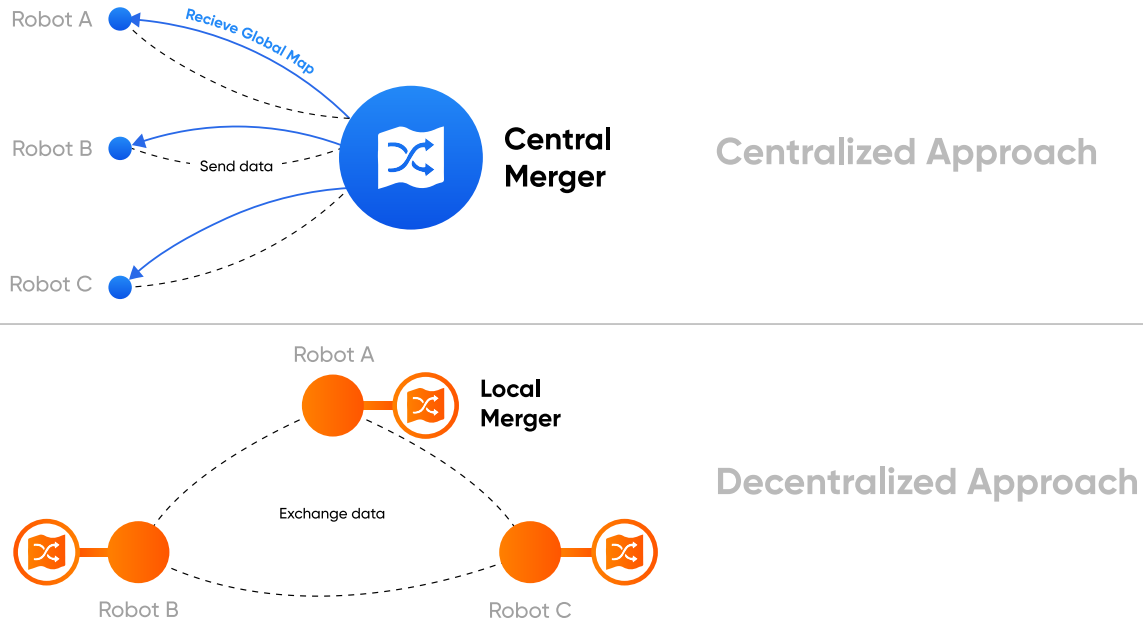


Figure 2.4: Communication approaches in C-SLAM

Map Merging

Map merging is a core component of collaborative SLAM, as independently built local maps must be aligned into a consistent global representation. This requires estimating the relative transformation between robots and ensuring consistency across overlapping regions.

Early approaches focused on geometric alignment of occupancy grid maps. For instance, Birk et al. [42] proposed methods based on scan matching and correlation techniques to align maps without prior knowledge of relative poses. Other approaches rely on feature-based and graph-based formulations:

- **Feature-based** use place recognition and descriptor matching to detect inter-robot loop closures, enabling more robust alignment in large-scale environments [43].
- **Graph-based** integrate inter-robot constraints into a global pose graph and optimize it using techniques such as pose graph optimization [18].

Overall, geometric methods are simpler and require fewer assumptions, but are less robust in complex environments. In contrast, feature-based and graph-based methods offer higher accuracy and scalability, at the cost of increased computational complexity.

Problems in Multiple-robot SLAM

Despite significant progress, collaborative SLAM systems still face several limitations [44].

- **Relative Poses of Robots:** Estimating the relative transformation between robots is essential for map fusion. Solutions include rendezvous strategies, map merging and relative localization techniques [45, 42, 46].
- **Uncertainty of the Relative Poses:** The uncertainty in relative pose estimates propagates to the global map and robot trajectories. This is typically handled through uncertainty propagation methods [47, 45].
- **Line-of-sight Observations:** Direct observations between robots can improve localization accuracy. These are often modeled using manifold representations [48].
- **Closing Loops:** Detecting inter-robot loop closures is critical for reducing drift. Solutions include filtering and optimization-based methods [49, 41].
- **Complexity:** The computational and memory complexity increases with the number of robots. Information filters and efficient optimization techniques are used to address this issue [50].
- **Heterogeneous Vehicles and Sensors:** Differences in robot platforms and sensors complicate data fusion. Solutions include multi-modal frameworks and cross-platform calibration [51, 52].
- **Synchronization:** Accurate time synchronization is required for consistent data fusion. Protocols such as Network Time Protocol are commonly used [53].

These limitations highlight that as the number of robots and the scale of the environment grow, the efficiency and reliability of SLAM systems become increasingly difficult to maintain. This can motivates research into more autonomous approaches to robot navigation and mapping.

2.4 Active SLAM

General SLAM algorithms are mostly passive, whereby the robot is controlled manually or goes toward predefined waypoints and the navigation or path-planning algorithm does not actively take part in robot motion or trajectory.

Active SLAM (A-SLAM) addresses this by solving the optimal exploration problem of an unknown environment, proposing a navigation strategy that generates future goal positions to minimise pose uncertainty, enabling fully autonomous navigation and mapping [2].

Its roots trace back to the 1980s, drawing from ideas in artificial intelligence and robotic exploration [54]. The first algorithmic implementation was presented by Feder et al. [55], while the term "Active SLAM" was formally coined in [56].

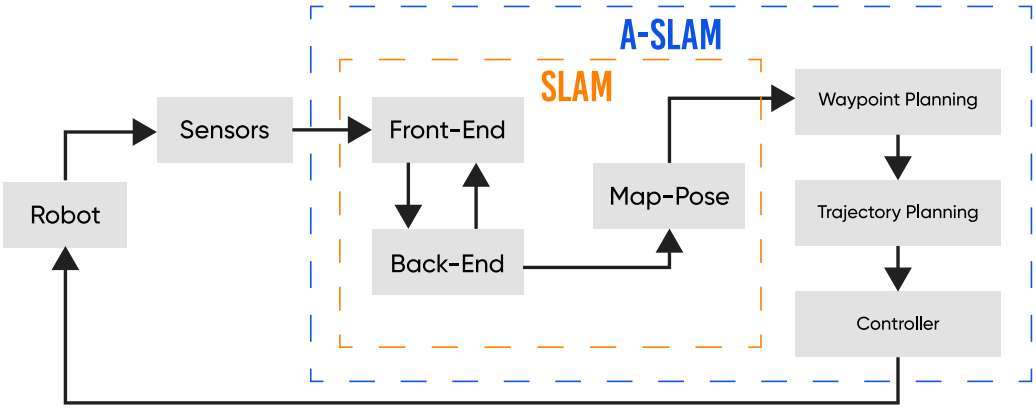


Figure 2.5: Architecture of SLAM and A-SLAM. Adapted from [2].

Active SLAM is a decision-making problem and there are several general frameworks for decision making that can be used as backbone for exploration-exploitation decisions. The main frameworks used in Active SLAM can be summarized as follows:

- **Theory of Optimal Experimental Design (TOED):** Applied to A-SLAM in [57], this framework selects future robot actions based on predicted map uncertainty, providing a principled mathematical foundation for decision-making under uncertainty.
- **Information-Theoretic Approaches:** Shannon and Rényi entropy have been integrated into A-SLAM to maximise information gain while minimising map and pose uncertainty [58].
- **Frontier-Based Exploration:** The robot navigates toward the boundaries between explored and unexplored space. Mammolo [59] extended this approach to dynamic environments with multiple ground robots, combining frontier exploration with graph-based iSAM optimization [60] as the backend and introducing a path utility function based on Shannon and Rényi entropy.
- **Geometric Approaches:** A-SLAM is formulated as an optimization task in which the robot selects the optimal path and trajectory to reduce pose and map uncertainty for efficient autonomous navigation [2].
- **Trajectory Optimization with Prior Map Information:** Soragna et al. [61] assumed the availability of prior topological map information, which A-SLAM exploits to actively close loops and improve global map consistency.

2.4.1 Active Collaborative SLAM

Active Collaborative SLAM (AC-SLAM) extends Active SLAM to multi-robot systems, where multiple agents jointly aim to minimize both map and pose uncertainty while coordinating their actions [2]. All the approaches using A-SLAM mentioned above are also applicable on AC-SLAM with additional constraints of managing the communication and parameter exchange between robots as mentioned above.

Network Topology of AC-SLAM

Building upon the multi-agent system architectures introduced previously, AC-SLAM systems are typically implemented using the following structures:

- **Centralized:** All robots send their data to a central unit responsible for map fusion and decision-making.
- **Decentralized:** Each robot performs local SLAM and exchanges only selected information with others.

Application Scenarios

AC-SLAM has been applied to several collaborative robotics scenarios, where coordination between agents improves overall mapping efficiency and robustness:

- **Collaborative Localization:** Robots cooperate to improve localization accuracy, particularly when some agents experience high uncertainty. Approaches such as belief-space planning [62] enable robots to actively assist others, while chained localization strategies [63] allow robots to sequentially refine their poses through inter-robot observations.
- **Exploration and Exploitation Tasks:** A key challenge is balancing the exploration of unknown areas with revisiting known regions to reduce uncertainty. Frontier-based methods like [64] are commonly extended to multi-robot systems, often combined with relocalization phases to improve global consistency.
- **Trajectory Planning:** Robots jointly plan their trajectories to maximize global information gain while minimizing redundancy. Long-horizon planning approaches, such as decentralized RRT*-based methods like [65] and entropy-based strategies like [66], allow efficient coordination in large-scale environments.

Open Challenges

Despite recent advances, several challenges remain in AC-SLAM. First, communication constraints such as limited bandwidth and delays significantly affect system performance and consistency. Moreover, scalability remains a critical issue, as the complexity of coordination and optimization increases with the number of robots [44]. Finally, there exists a significant gap between simulation and real-world deployment, as only a limited portion of AC-SLAM works are validated on physical robot platforms (approximately 37%) [2].

2.5 Real-World Applications

The development of SLAM algorithms, particularly in their collaborative and active forms, has enabled a wide range of real-world applications. These systems are now deployed in structured and semi-structured environments where multiple robots operate simultaneously to improve efficiency, robustness and scalability.

Warehouse Automation

One of the most prominent applications of multi-robot systems is in warehouse automation. Modern fulfillment centers rely on large fleets of mobile robots to transport goods, optimize storage and accelerate order processing.

An example of such application is Mecalux's AMRs which uses LiDAR scanners to perform SLAM mapping in order to orient themselves. Their robots are able to navigate complex spaces with people, goods, storage systems and other machinery avoiding all kinds of stationary and moving obstacles [67].



Figure 2.6: Mecalux's AMR Series [67]

Another well-known example is the use of autonomous mobile robots in Amazon fulfillment centers, originally developed by Kiva Systems, renamed later as Amazon Robotics [68]. These robots operate collaboratively to move storage shelves to human operators, reducing travel time and increasing efficiency [69].

The original Kiva robots, shown in Figure 2.7a, navigate beneath storage pods and transport them to human operators, carrying loads up to 450 kg while coordinating across hundreds of units [69]. Subsequent platforms such as Pegasus and Xanthus, shown in Figure 2.7b, introduced reduced size, higher payload capacity and modular architectures supporting multiple attachments for tasks ranging from parcel sorting to tote transportation.

Healthcare

Another application of SLAM is in hospital service robots. In 2024, Richtech Robotics announced their latest innovation, MedBot, an elevator-capable delivery robot designed to streamline pharmacy operations by providing uninterrupted, 24/7 medication delivery [70].



(a) Amazon's Kiva Robots [69].



(b) Amazon's Xanthus Robots [69].

Figure 2.7: Autonomous robots in a warehouse environment.

MedBot is equipped with LiDAR and two cameras that allow it to map unknown environments [71], enhancing obstacle detection and avoidance. This capability is especially critical in hospital settings, where precision and reliability are essential.



(a) MedBot in idle position [71].



(b) MedBot in operation [70].

Figure 2.8: MedBot by Richtech Robotics.

2.6 Conclusion

This chapter has traced the evolution of SLAM from early probabilistic methods to modern collaborative and active approaches. While general SLAM has reached a considerable level of maturity, extending it to multi-robot systems introduces additional challenges in communication, map merging and scalability. Active SLAM further advances autonomy by integrating navigation decisions into the mapping process, with its multi-agent extension enabling coordinated exploration. Despite significant progress, the gap between simulation and real-world deployment remains an open challenge.

Chapter 3

Active Collaborative SLAM

3.1 Introduction

This chapter develops the theoretical foundations of the Active Collaborative SLAM system proposed in this project. It begins with the probabilistic formulation of the SLAM problem and its graph-based representation, establishing the mathematical framework upon which the rest of the system is built. The Active SLAM problem is then formalised, casting navigation decisions as an optimisation over an information-theoretic objective, and the coverage-path-planning framework of Soragna *et al.* is presented in detail, including the online graph adaptation mechanism that makes it robust to prior inaccuracies. Finally, the chapter introduces the collaborative layer that extends single-robot Active SLAM to a multi-robot setting: skeleton extraction to construct a prior graph from a floor plan, BFS-based workload partitioning to distribute the exploration task across robots, distributed single-robot Active SLAM for the exploration phase, and a coordinate-aligned occupancy grid merging procedure that produces a unified global map at the end of the mission.

3.2 Pose Graph SLAM

Simultaneous Localization and Mapping (SLAM) is a fundamental problem in mobile robotics. Robots are deployed in an unknown environment and must simultaneously build a map of the environment while estimating their own positions within it.

3.2.1 Mathematical Formulation of SLAM

Simultaneous Localization and Mapping (SLAM) is the task of building an incremental map of an unknown environment while simultaneously estimating the robot's pose within that map.

Formal statement

Consider a mobile robot moving through an environment, taking relative observations of several unknown landmarks using an onboard sensor, as illustrated in Figure 3.1. The figure depicts the central difficulty of SLAM: neither the robot's true trajectory nor the true landmark locations are ever directly known. At each time step k, the robot occupies a true pose x_k (shown as a faded triangle) that differs from its estimated pose (solid triangle), and each landmark m_i has a true position (faded star) that differs from its estimated position (solid star). The robot moves under control inputs u_k, accumulating error along its trajectory, while landmark observations $z_{k,i}$ are made relative to the current (uncertain) robot pose. Because every observation is corrupted by both sensor noise and pose uncertainty, errors in landmark estimates become correlated through their shared dependence on the robot's location history.

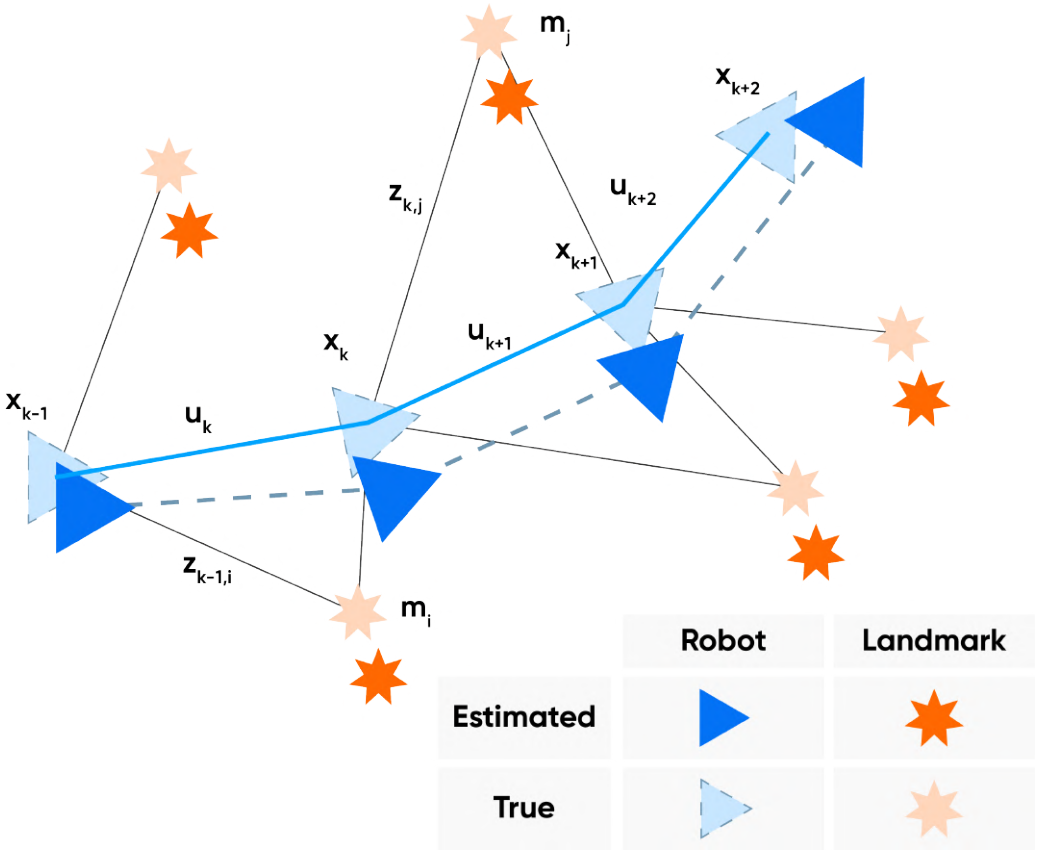


Figure 3.1: SLAM Problem Illustration. Adapted from [1]

At a discrete time step k, the following variables are defined:

- $\mathbf{x}_k$: the robot state vector at time k, representing its position and orientation.
- $\mathbf{u}_k$: the control input applied between time steps $k-1$ and k, which drives the robot from state $\mathbf{x}_{k-1}$ to state $\mathbf{x}_k$.
- $\mathbf{m}_i$: the position vector of the ith landmark in the environment.
- $\mathbf{z}_{i,k}$: the observation of landmark $\mathbf{m}_i$ acquired by the robot at time k. When the specific landmark is not relevant or multiple observations are considered simultaneously, the notation $\mathbf{z}_k$ is used.

In addition, several sets are defined to represent the robot's trajectory, control history, map, and observations. The robot trajectory up to time k is denoted by $\mathbf{X}_{0:k} = \{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_k\}$, while the sequence of control inputs is represented by $\mathbf{U}_{0:k} = \{\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_k\}$. The map is described by the set of all landmarks, $\mathbf{m} = \{\mathbf{m}_1, \mathbf{m}_2, \ldots, \mathbf{m}_n\}$, and the collection of all observations acquired up to time k is denoted by $\mathbf{Z}_{0:k} = \{\mathbf{z}_0, \mathbf{z}_1, \ldots, \mathbf{z}_k\}$.

The SLAM problem can be formally stated as follows: given a sequence of sensor measurements $\mathbf{Z}$ and (optionally) control inputs $\mathbf{U}$, estimate the **robot trajectory X** and the **map m** of the environment [1].

Bayesian formulation

Solving the full SLAM problem consists of estimating the **posterior distribution** of the robot's trajectory and the map of the environment given all the measurements plus an initial position $\mathbf{x}_0$ which is written as: $p(\mathbf{x}_k,\, \mathbf{m} \mid \mathbf{Z}_{0:k},\, \mathbf{U}_{0:k},\, \mathbf{x}_0)$

Bayes' theorem states that, for any two events A and B:

$$p(A \mid B) \;=\; \frac{p(B \mid A)\; p(A)}{p(B)}. \tag{3.1}$$

Identifying $A \equiv (\mathbf{x}_k, \mathbf{m})$ and $B \equiv (\mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{x}_0)$ we get:

$$p(\mathbf{x}_k, \mathbf{m} \mid \mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{x}_0) \;=\; \frac{p(\mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{x}_0 \mid \mathbf{x}_k, \mathbf{m})\;\; p(\mathbf{x}_k, \mathbf{m})}{p(\mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{x}_0)}. \tag{3.2}$$

The denominator $p(\mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{x}_0)$ is a normalisation constant that does not depend on the unknowns $(\mathbf{x}_k, \mathbf{m})$, so:

$$p(\mathbf{x}_k, \mathbf{m} \mid \mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{x}_0) \;\propto\; p(\mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{x}_0 \mid \mathbf{x}_k, \mathbf{m})\;\; p(\mathbf{x}_k, \mathbf{m})\,. \tag{3.3}$$

Sensor measurements $\mathbf{Z}_{0:k}$ depend on both the trajectory and the map, whereas control inputs $\mathbf{U}_{0:k}$ are commands chosen by the robot and are therefore statistically independent of $\mathbf{m}$. Using the product rule:

$$p(\mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{x}_0 \mid \mathbf{x}_k, \mathbf{m}) \;=\; p(\mathbf{Z}_{0:k} \mid \mathbf{x}_k, \mathbf{m})\;\; p(\mathbf{U}_{0:k} \mid \mathbf{x}_k)\,. \tag{3.4}$$

Since $p(\mathbf{U}_{0:k} \mid \mathbf{x}_k)$ is constant with respect to the unknowns $(\mathbf{x}_k, \mathbf{m})$, it can be dropped from the expression, leaving only $p(\mathbf{Z}_{0:k} \mid \mathbf{x}_k, \mathbf{m})$.

The joint prior over the trajectory and the map factorises as:

$$p(\mathbf{x}_k, \mathbf{m}) \;=\; p(\mathbf{x}_k \mid \mathbf{U}_{0:k}, \mathbf{x}_0)\; p(\mathbf{m}), \tag{3.5}$$

where:

- $p(\mathbf{x}_k \mid \mathbf{U}_{0:k}, \mathbf{x}_0)$ is the **motion prior**: the probability of the trajectory given the sequence of control inputs and the known starting pose, governed by the robot's kinematic and dynamic model.
- $p(\mathbf{m})$ is the **map prior**: typically taken to be uniform, reflecting the absence of any preference over maps before sensor data are observed.

Combining (3.3), (3.4), and (3.5) we get the SLAM posterior:

$$p(\mathbf{x}_k, \mathbf{m} \mid \mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{x}_0) \;\propto\; \underbrace{p(\mathbf{Z}_{0:k} \mid \mathbf{x}_k, \mathbf{m})}_{\text{observation likelihood}} \quad \underbrace{p(\mathbf{x}_k \mid \mathbf{U}_{0:k}, \mathbf{x}_0)}_{\text{motion prior}} \quad \underbrace{p(\mathbf{m})}_{\text{map prior}} . \tag{3.6}$$

- **Observation likelihood** $p(\mathbf{Z}_{0:k} \mid \mathbf{x}_k, \mathbf{m})$ measures how well the sensor readings match the current estimates of the trajectory and the map [1]. The single-step **observation model** $p(z_k \mid \mathbf{x}_k, \mathbf{m})$ describes the probability of making measurement z_k given the current vehicle pose and the map [1]. Assuming observations are conditionally independent given the vehicle state and the map, the likelihood factorises as:

$$p(\mathbf{Z}_{0:k} \mid \mathbf{x}_k,\, \mathbf{m}) \;=\; \prod_{i=0}^{k} p(z_i \mid \mathbf{x}_i,\, \mathbf{m})\,. \tag{3.7}$$

- **Motion prior** $p(\mathbf{x}_k \mid \mathbf{U}_{0:k}, \mathbf{x}_0)$ encodes how likely the trajectory is given the sequence of control inputs applied and the known initial pose [1]. The single-step **motion model** $p(\mathbf{x}_k \mid \mathbf{x}_{k-1}, u_k)$ describes the probability of transitioning to state $\mathbf{x}_k$ given the previous state and control input u_k. Under the Markov assumption, the motion prior factorises as:

$$p(\mathbf{x}_k \mid \mathbf{U}_{0:k},\, \mathbf{x}_0) \;=\; \prod_{i=1}^{k} p(\mathbf{x}_i \mid \mathbf{x}_{i-1},\, u_i)\,. \tag{3.8}$$

- **Map prior** $p(\mathbf{m})$ encodes any prior knowledge about the environment. When no such knowledge is available it is set to a uniform distribution and can be dropped from the optimisation.

Maximum A Posteriori Estimate

As mentioned in 2.2.2 two broad approaches were taken to solve the SLAM problem: filter-based methods (Filtering) and graph-based methods (Smoothing). The latter have become dominant in modern implementations due to their scalability at large scale [1]. They do not track the full distribution; instead they compute the *Maximum A Posteriori* (MAP) estimate:

$$\hat{\mathbf{x}},\, \hat{m} \;=\; \underset{\mathbf{x},\, m}{\arg\max}\; p(\mathbf{x}_k, \mathbf{m} \mid \mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{x}_0) \tag{3.9}$$

Computing this MAP estimate requires evaluating the posterior of Eq. (3.6). Taking the negative logarithm and dropping constant terms, maximising the posterior is equivalent to minimising:

$$-\log\, p(\mathbf{x}_k, \mathbf{m} \mid \mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{x}_0) \;\propto\; -\sum_i \log\, p(z_i \mid \mathbf{x}_i, \mathbf{m}) \;-\sum_i \log\, p(\mathbf{x}_i \mid \mathbf{x}_{i-1}, u_i) \tag{3.10}$$

Assuming that the noise on both the motion and observation models is zero-mean Gaussian, each measurement $\mathbf{z}_{ij}$ between two poses $\mathbf{x}_i$ and $\mathbf{x}_j$ can be modelled as the predicted transformation $\hat{\mathbf{z}}_{ij}$ with additive Gaussian noise whose covariance is $\mathbf{\Sigma}_{ij}$. The residual $\mathbf{e}_{ij}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{z}_{ij} - \hat{\mathbf{z}}_{ij}(\mathbf{x}_i, \mathbf{x}_j)$ therefore follows a zero-mean Gaussian distribution with the same covariance, and its negative log-likelihood reduces to [72].

Since $\mathbf{e}_{ij} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma}_{ij})$, the likelihood of the measurement is given by the multivariate Gaussian:

$$p(\mathbf{z}_{ij}) \;=\; \frac{1}{\sqrt{(2\pi)^n \det(\mathbf{\Sigma}_{ij})}} \exp\!\Big(-\frac{1}{2}\,\mathbf{e}_{ij}^{\top}\,\mathbf{\Sigma}_{ij}^{-1}\,\mathbf{e}_{ij}\Big) \tag{3.11}$$

Taking the negative logarithm:

$$-\log\, p(\mathbf{z}_{ij}) \;=\; \frac{1}{2}\log((2\pi)^n \det(\mathbf{\Sigma}_{ij})) \;+\; \frac{1}{2}\,\mathbf{e}_{ij}^{\top}\,\mathbf{\Sigma}_{ij}^{-1}\,\mathbf{e}_{ij} \tag{3.12}$$

The first term depends only on the covariance $\mathbf{\Sigma}_{ij}$, which is fixed and does not depend on the poses. It can therefore be absorbed into a constant C. Substituting $\mathbf{\Omega}_{ij} = \mathbf{\Sigma}_{ij}^{-1}$:

$$-\log\, p(\mathbf{z}_{ij}) \;=\; \frac{1}{2}\,\mathbf{e}_{ij}^{\top}\,\mathbf{\Omega}_{ij}\,\mathbf{e}_{ij} \;+\; \mathrm{C} \tag{3.13}$$

where $\mathbf{\Omega}_{ij} = \mathbf{\Sigma}_{ij}^{-1}$ is the information matrix of the measurement, encoding its uncertainty. Summing over all constraints and minimising, the MAP estimate reduces to the following nonlinear least-squares problem:

$$\hat{\mathbf{x}} \;=\; \arg\min_{\mathbf{x}} \sum_{(i,j)\,\in\,\mathcal{E}} \mathbf{e}_{ij}(\mathbf{x}_i, \mathbf{x}_j)^{\top}\,\mathbf{\Omega}_{ij}\,\mathbf{e}_{ij}(\mathbf{x}_i, \mathbf{x}_j) \tag{3.14}$$

3.2.2 Graph-Based SLAM Formulation

In graph-based SLAM the robot trajectory is represented as a **pose graph** $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where each node $v_i \in \mathcal{V}$ represents a robot pose $\mathbf{x}_i$ visited along the trajectory, and each edge $e_{ij} \in \mathcal{E}$ encodes the relative spatial constraint between poses $\mathbf{x}_i$ and $\mathbf{x}_j$, as illustrated in Figure 3.5.

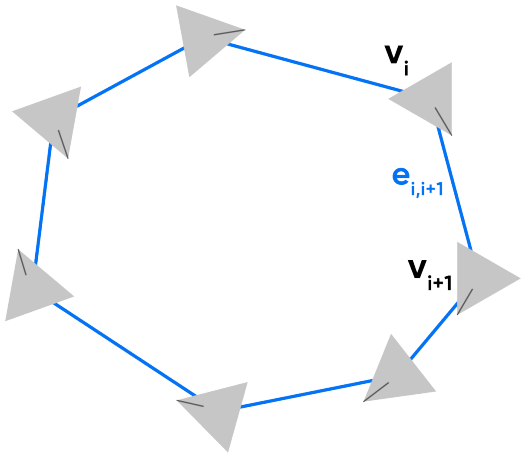


Figure 3.2: Pose-graph representation of a SLAM process

In the context of graph-based SLAM, one typically considers two different problems:

- The first one is to **identify the constraints** based on sensor data. A solution to this problem is often referred to as the **SLAM front-end** and it directly deals with the sensor data [19].

- The second problem is to **correct the poses** of the robot to obtain a consistent map of the environment given the constraints. This part of the approach is often referred to as the optimizer or the **SLAM back-end** [19].

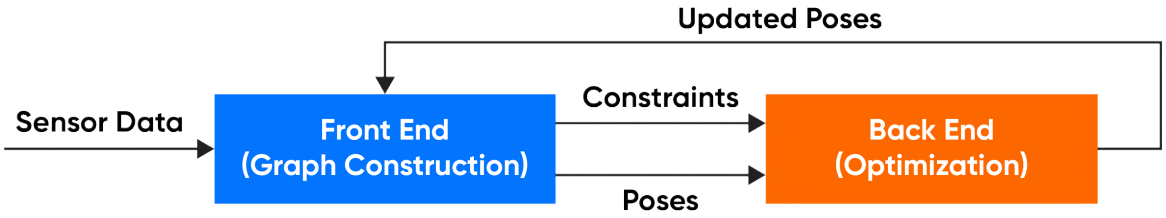


Figure 3.3: Typical graph-based SLAM system.

Front-End: Scan Matching and Loop Closure

The front-end is responsible for **generating the constraints** that populate $\mathcal{E}$. Two categories of edges populate the pose graph: **Odometry edges** that connects two consecutive poses (i, $i+1$) at each time step, derived from robot sensors'. And **loop closure edges** that connects two non-consecutive poses (i, j) when the robot revisits a previously explored area. The front-end periodically attempts to detect these **loop closures** by matching the current sensor scan against stored scans at previous poses using a two-stage process:

- **Candidate selection:** any previously visited pose within a configurable search radius is flagged as a candidate, provided it belongs to a chain of at least a minimum number of linked scans.

- **Correlation scan matching:** the current scan is projected onto a grid and its correlation score is evaluated over the search space of translations and rotations in a coarse pass to identify the rough alignment then a fine pass to refine the relative pose estimate.

In Eq.3.10, each term in this sum corresponds to one constraint in the pose graph: odometry edges contribute terms from the **motion model**, while **loop closure edges** contribute terms from the observation model.

Figure 3.4 illustrates a pose graph representation. The grey nodes represent robot poses and the pink edges are the sequential odometry constraints that chain them together, accumulating drift over time. The cyan edges represent the loop-closure constraint added when the robot detects that it has returned to a previously visited location.

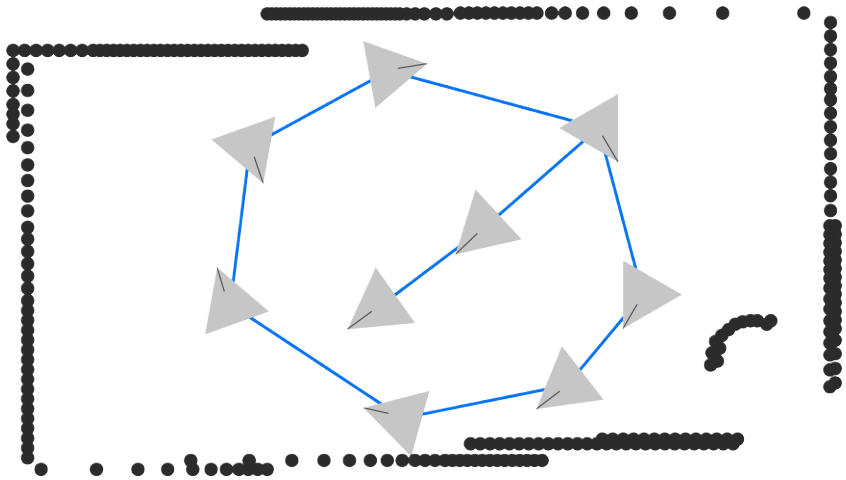


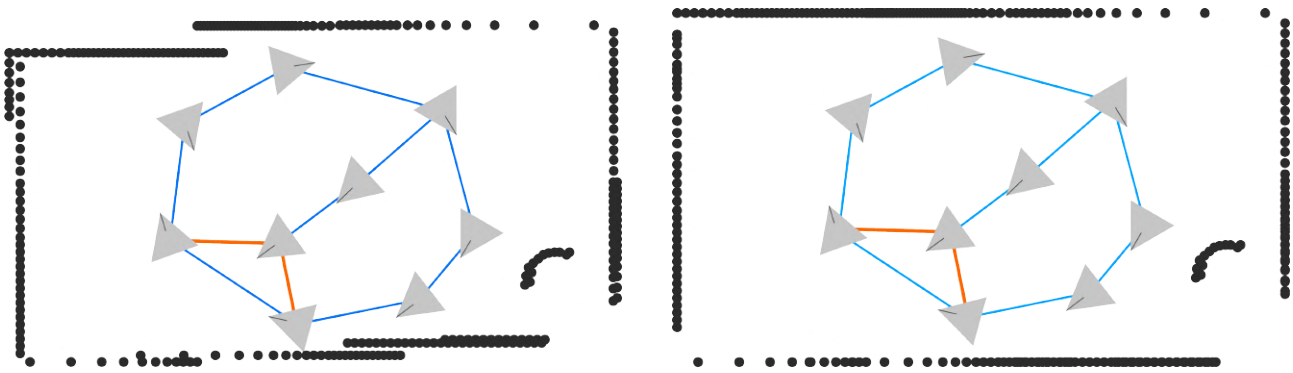
Figure 3.4: Pose graph representation with a loop-closure constraint.

Back-End: Graph Optimization

Given the pose graph, the back-end seeks to optimize the configuration of all node poses that best explains the observed constraints. Formally, it minimizes Eq. 3.14 which is the nonlinear least-squares objective function.

Since the error function is nonlinear in the poses, the problem is solved iteratively. At each iteration, **the error function is linearised** around the current pose estimate, reducing the problem to the sparse linear system. This system is solved efficiently via sparse Cholesky factorisation of $\mathbf{H}$: $\mathbf{H} = \mathbf{L}\mathbf{L}^\top$. The pose estimates are then updated as $\hat{\mathbf{x}} \leftarrow \hat{\mathbf{x}} + \Delta\hat{\mathbf{x}}$ and the process repeats until convergence [72].

Figure 3.5 illustrates a pose graph before and after optimisation. The two loop-closure constraint are optimized when the robot detects that it has returned to a previously visited location, resulting a more accurate map representation.



(a) Before optimisation. (b) After optimization.

Figure 3.5: Pose graph before and after optimization.

3.3 SLAM using Connectivity Graphs as Priors

Standard SLAM assumes a *passive* robot: its trajectory is determined by an external operator and the algorithm simply processes whatever observations arrive. Active SLAM removes this assumption by letting the robot itself decide where to move in order to minimise map and pose uncertainty while still achieving complete coverage of the environment.

Formally, let $\mathbf{x}_t$ be the robot pose at time t and let $\mathcal{G}_t$ be the pose graph maintained by the SLAM front-end. The Active SLAM policy π selects the next goal pose

$$\mathbf{x}^*_{t+1} = \pi(\mathcal{G}_t,\, \mathcal{M}_t) \tag{3.15}$$

such that some measure of expected information gain (or uncertainty reduction) is maximised, subject to the kinematic constraints of the robot and the connectivity of the navigation space.

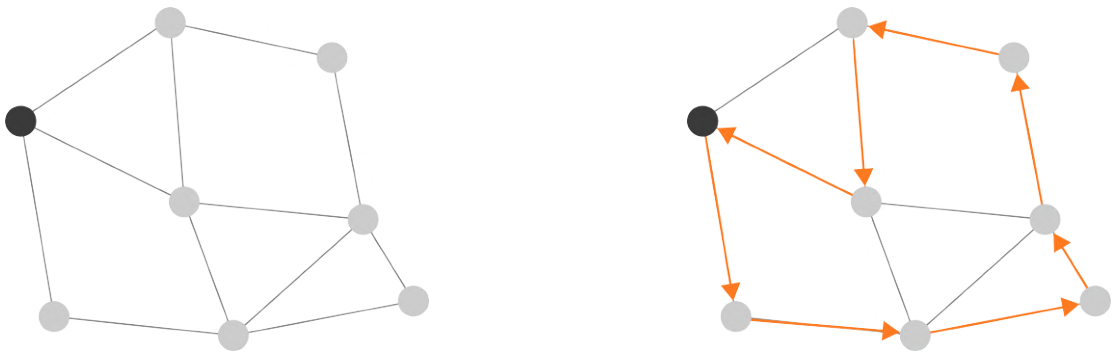
As discussed in Section 2.4, greedy frontier-based exploration is simple but often suboptimal, as it ignores prior environmental knowledge and provides limited control over path structure. To address these limitations, Soragna *et al.* [61] incorporate prior information through a topo-metric graph and formulate global exploration as a Chinese Postman Problem.

Topo-Metric Prior Graph & The Chinese Postman Problem

Let $G_0 = (V_0, E_0)$ be the prior topo-metric graph provided before exploration begins. Each vertex $v^{[i]} \in V_0$ carries a 2D position $\mathbf{x}^{[i]}_{\text{ref}} \in \mathbb{R}^2$, and each edge $(i, j) \in E_0$ represents a traversable corridor between the two vertices, with a reference length $L^{[i,j]}_{\text{ref}}$. In our system, the length is calculated as the Euclidean distance between the positions of the connected vertices:

$$L^{[i,j]}_{\text{ref}} = \left\| \mathbf{x}^{[i]}_{\text{ref}} - \mathbf{x}^{[j]}_{\text{ref}} \right\|_2 . \tag{3.16}$$

This reference graph serves as a prior representation of the environment topology before the exploration process begins. The exploration task consists of navigating every edge of G_0 at least once, modelled as a Chinese Postman Problem (CPP) [73]. Given a graph $G = (V, E)$, the CPP finds the minimum-cost closed walk that traverses all edges, producing an ordered sequence of nodes that defines the robot's exploration schedule, as illustrated in Figure 3.6.



(a) Topo-metric connectivity graph. (b) Optimal CPP tour.

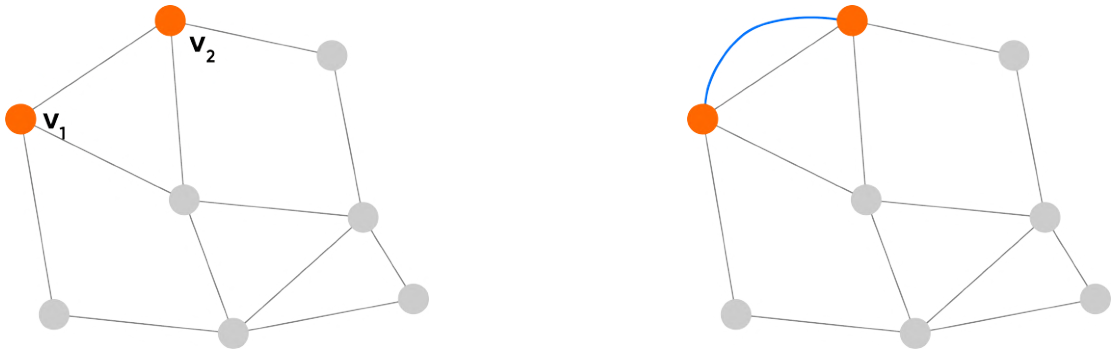
Figure 3.6: CPP-based exploration on a topo-metric graph.

Graph Eulerianisation

An Eulerian circuit exists if and only if every vertex has even degree. When this condition is not met, the graph is augmented through the following steps:

1. Identify the set V_{odd} of all odd-degree vertices.
2. Compute the shortest path between every pair of vertices in V_{odd} using algorithm like Djikstra's Algorithm.
3. Find a perfect matching on V_{odd}, which is a set of pairs that covers every vertex in V_{odd} exactly once, such that the sum of the matched pair distances is minimised.
4. For each matched pair (u, v), duplicate the edges along their shortest path in G, raising the degree of every odd vertex by one.

After augmentation all vertices have even degree and the graph becomes Eulerian, as illustrated in Figure 3.7. In this example, v_1 and v_2 are the only odd-degree vertices (degree 3 each); duplicating the single edge v_1–v_2 raises both degrees to 4, satisfying the Eulerian condition across the entire graph.



(a) Graph before Eulerisation. (b) Graph after Eulerisation.

Figure 3.7: Eulerianisation by odd-degree matching.

Eulerian Circuit Extraction

Once the graph is Eulerian, the optimal circuit is extracted as formalised in Algorithm 1, through the following steps:

1. Start from an arbitrary vertex and follow unused edges greedily until the walk returns to its origin, forming an initial cycle.
2. If unused edges remain incident to any vertex on the current cycle, select such a vertex as a new starting point.
3. Extend the tour by splicing in a new sub-cycle from that vertex, following unused edges until returning to it.
4. Repeat steps 2–3 until all edges have been consumed, at which point the concatenated cycles form a single Eulerian circuit covering the entire graph.

Algorithm 1 Chinese Postman Problem Algorithm

Require: Graph $G = (V, E)$
Ensure: Minimum-cost closed walk covering every edge at least once

▷ *Step 1: Check Eulerian condition*

$V_{\text{odd}} \leftarrow \{v \in V \mid \deg(v) \text{ is odd}\}$
if $V_{\text{odd}} = \emptyset$ **then**
    go to Step 3
end if

▷ *Step 2: Eulerianisation*

for all $(u, v) \in V_{\text{odd}}$ **do**
    $\text{dist}(u, v) \leftarrow \textsc{Dijkstra}(G, u, v)$
end for
$M \leftarrow$ minimum-weight perfect matching on V_{odd}
for all $(u, v) \in M$ **do**
    duplicate every edge of $P(u, v)$ in G
end for

▷ *Step 3: Eulerian circuit extraction*

tour $\leftarrow \textsc{GreedyWalk}(G, s)$
while $\exists\, w \in$ tour with unused edges **do**
    splice $\textsc{GreedyWalk}(G, w)$ into tour at w
end while
return tour

3.4 Active Collaborative SLAM System

The proposed AC-SLAM system extends the single-robot Active SLAM framework of Section 3.3 to a team of N robots operating in the same environment. The overall pipeline is divided into three sequential phases:

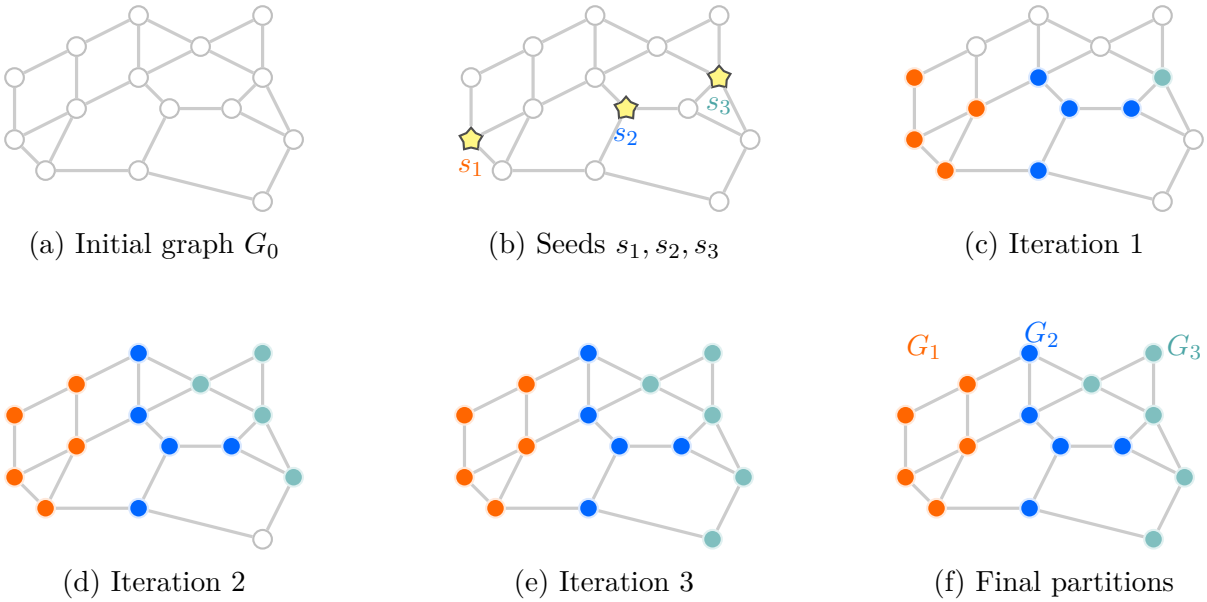
1. **Pre-exploration:** The topo-metric graph is partitioned into N balanced sub-graphs using a BFS partitioning algorithm; each sub-graph is assigned to one robot.
2. **Exploration:** Each robot independently executes the Active SLAM framework within its assigned sub-graph partition, running SLAM Toolbox for local mapping and the CPP planner for path selection.
3. **Post-exploration:** Once all robots have completed their assigned partitions, their individual occupancy grid maps are merged into a single global map using the MapMerge Algorithm.

3.4.1 BFS Graph Partitioning

Once the graph $G_0 = (V_0, E_0)$ has been established, it must be divided into N sub-graphs $\{G_1, \ldots, G_N\}$ to be distributed among the robots. The partitioning criterion is twofold: each sub-graph should be:

- **Connected:** so that the assigned robot can navigate its region without crossing into another robot's territory.
- **Balanced:** so all robots have equal territory to navigate, which is approximated by the total edge length

A Breadth-First Search (BFS) approach is used to divide G_0 into N balanced sub-graphs, one per robot. As illustrated in Figure 3.8, N seed nodes $s_1, \ldots, s_N$ are placed at spatially separated locations across G_0 (b), each serving as the starting point of a frontier that expands outward by claiming unclaimed neighbouring nodes (c)–(e). The frontiers grow simultaneously in round-robin fashion, and the expansion of frontier i is paused once its assigned sub-graph G_i reaches the target size $k = \lfloor |E_0|/N \rfloor$, preventing any single robot from claiming a disproportionate share of the graph. Once all nodes have been assigned, the final balanced partition $\{G_1, \ldots, G_N\}$ is extracted (f), with each connected region handed to its corresponding robot as its exploration area.



(a) Initial graph G_0 (b) Seeds s_1, s_2, s_3 (c) Iteration 1

(d) Iteration 2 (e) Iteration 3 (f) Final partitions

Figure 3.8: BFS partitioning of G_0 into three sub-graphs $\{G_1, G_2, G_3\}$.

3.4.2 Distributed Exploration

Each robot executes the following loop:

1. Compute the initial CPP plan on G_i using the modified cost.
2. Navigate to the next goal pose following the plan.
3. Update the SLAM Toolbox pose graph and occupancy map $\mathcal{M}_i$ with incoming LiDAR scans.
4. Perform the online graph adaptation step to correct G_i against $\mathcal{M}_i$.
5. Re-plan if the corrected graph deviates significantly from the current plan.
6. Repeat until all edges of G_i have been traversed.

3.4.3 Map Merging

After the partitioning step, each robot i receives its sub-graph G_i and operates independently. No inter-robot communication is required during the exploration phase, which makes the system robust to communication failures and bandwidth limitations. At the end of this phase, robot i holds a local occupancy grid map m_i and a SLAM pose graph covering its assigned partition.

Once all N robots have finished exploring their partitions, the N local occupancy grid maps $m_1, \ldots, m_N$ must be assembled into a single global map $\mathcal{M}$. Since the robots operate independently and may start from different positions, their local maps are expressed in different reference frames with unknown relative transformations. The map merging approach of Birk and Carpin [42] addresses this problem without requiring any knowledge of the initial relative poses.

Occupancy grid map

An occupancy grid map is a discretisation of a continuous environment into a finite two-dimensional grid of cells, where each cell $\mathcal{M}[x, y]$ stores a value representing the occupancy state of the corresponding region of physical space. Formally, a map $\mathcal{M}$ of dimensions $W \times H$ is defined as a function

$$\mathcal{M} : \{0, \ldots, W-1\} \times \{0, \ldots, H-1\} \longrightarrow \{-1,\, 0,\, 1, \ldots, 100\}, \tag{3.17}$$

where the cell value encodes one of three semantic states:

$$\mathcal{M}[x, y] = \begin{cases} -1 & \text{unknown (never observed)}, \\ 0 & \text{free space (no obstacle detected)}, \\ 100 & \text{occupied (obstacle detected)}. \end{cases} \tag{3.18}$$

The map is further characterised by a metric origin $(o_x, o_y) \in \mathbb{R}^2$, which anchors the grid to a global coordinate frame, and a resolution $r > 0$ (in metres per cell), which defines the physical size of each cell.

Each robot constructs such a map incrementally from its onboard sensor readings, yielding a partial representation of the environment that covers only the region explored so far.

Transform-Based Map Merging

Because each partial map is localised in a common coordinate frame, the relative transformation between any two maps is fully determined by their metric origins. Given two maps $\mathcal{M}_1$ and $\mathcal{M}_2$ with origins $(o_x^{(k)}, o_y^{(k)})$, resolutions r_k, and dimensions $W_k \times H_k$, the merged map $\mathcal{M}^*$ has origin and dimensions determined by the bounding box of both maps:

$$o_x^* = \min(o_x^{(1)}, o_x^{(2)}), \quad o_y^* = \min(o_y^{(1)}, o_y^{(2)}), \tag{3.19}$$

$$W^* = \left\lceil \frac{x_{\max} - o_x^*}{r^*} \right\rceil, \qquad H^* = \left\lceil \frac{y_{\max} - o_y^*}{r^*} \right\rceil, \tag{3.20}$$

where $r^* = \min(r_1, r_2)$. Each cell (x, y) of $\mathcal{M}_k$ is projected into $\mathcal{M}^*$ by

$$\tilde{x} = \left\lfloor \frac{o_x^{(k)} + x\, r_k - o_x^*}{r^*} \right\rfloor, \qquad \tilde{y} = \left\lfloor \frac{o_y^{(k)} + y\, r_k - o_y^*}{r^*} \right\rfloor. \tag{3.21}$$

$\mathcal{M}^*$ is initialised to -1; cells from $\mathcal{M}_1$ are written unconditionally, and cells from $\mathcal{M}_2$ only where $\mathcal{M}^*$ remains unknown:

$$\mathcal{M}^*[\tilde{x}, \tilde{y}] = \begin{cases} \mathcal{M}_1[x, y] & \text{(first pass)}, \\ \mathcal{M}_2[x, y] & \text{if } \mathcal{M}^*[\tilde{x}, \tilde{y}] = -1. \end{cases} \tag{3.22}$$

Finally, the known initial position $\mathbf{p}^{(k)} = (p_x^{(k)}, p_y^{(k)})$ of each robot is projected via (3.21) and verified to lie within the spatial extent of its own partial map,

$$o_x^{(k)} \le p_x^{(k)} \le o_x^{(k)} + W_k r_k, \qquad o_y^{(k)} \le p_y^{(k)} \le o_y^{(k)} + H_k r_k, \tag{3.23}$$

providing a geometric consistency check before the merge is performed.

Multi-Robot Extension

For $N > 2$ robots, the pairwise merge operation defined by equations (3.22) is extended to the full set of partial maps $\{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_N\}$ via a binary reduction tree. At each level ℓ of the tree, adjacent map pairs are merged according to

$$\mathcal{M}_k^{(\ell+1)} = \text{merge}\Big(\mathcal{M}_{2k-1}^{(\ell)},\ \mathcal{M}_{2k}^{(\ell)}\Big), \qquad k = 1, 2, \ldots, \left\lceil \tfrac{N}{2^\ell} \right\rceil, \tag{3.24}$$

where $\mathcal{M}_k^{(0)} = \mathcal{M}_k$ are the original partial maps. If N is odd at any level, the unpaired map is carried forward unchanged to the next level. The process terminates after $\lceil \log_2 N \rceil$ levels, yielding a single global map $\mathcal{M}^*$.

The global merge is performed exactly once, after all N robots have completed their individual exploration phases. At that point, the partial maps $\{\mathcal{M}_1, \ldots, \mathcal{M}_N\}$ are submitted simultaneously as inputs to the reduction tree. Since no inter-robot map exchange occurs during the exploration phase, the merging computation is entirely decoupled from the mission execution and imposes no communication overhead on the robots during operation.

3.5 Conclusion

This chapter presented the theoretical foundations of the Active Collaborative SLAM system developed in this project, covering the graph-based SLAM formulation, the Active SLAM planning framework, and the collaborative layer comprising prior graph construction, workload partitioning, distributed exploration, and post-exploration map merging. The following chapters describe how this architecture is implemented in simulation (Chapter 4) and evaluated on real robotic hardware (Chapter 5).

Chapter 4

Simulation

4.1 Introduction

Bridging the gap between theoretical design and real-world deployment requires a rigorous simulation phase in which algorithms can be tested, tuned, and validated under controlled yet realistic conditions. This chapter describes the simulation environment built in Isaac Sim to validate the complete Active Collaborative SLAM system developed in the previous chapter.

The chapter begins with an overview of the simulation tools used, covering the Robot Operating System (ROS2) middleware and the Isaac Sim platform. It then details the construction of the simulated environment, modelled after Floor 0 of the ENP research center using real metric measurements, and the configuration of the robot platforms. The implementation of each system component is then described, including SLAM, navigation, prior graph construction, workload partitioning, active exploration, and map merging. Finally, the multi-robot simulation setup is presented alongside the results obtained, providing a comprehensive evaluation of the system's behavior before deployment on physical hardware.

4.2 The Robot Operating System

The Robot Operating System (ROS) is an open-source middleware framework for robotic software development, providing tools, libraries, and abstraction layers that afford scalability, flexibility, and modularity [74]. Maintained by the Open-Source Robotics Foundation (OSRF), it has become a pivotal framework in robotics research. ROS is structured around the concept of a **Node** — a single-purpose process executing a specific task — where nodes communicate through topics, services, and actions, and can operate independently across multiple machines.

There are two versions of ROS: ROS1 (released 2007) and ROS2 (released 2017). ROS1 carries notable limitations including the lack of real-time support, limited OS compatibility, no built-in security, and a centralized architecture with a single point of failure via `roscore`. To address these, Open Robotics released **ROS2** in 2018 as a complete redesign, switching from TCP/UDP to the **Data Distribution Service (DDS)** — a peer-to-peer protocol that enhances security, fault tolerance, and support for real-time, embedded, and multi-robot systems. A comparison is summarized in Table 4.1.

Table 4.1: Differences between ROS1 and ROS2.

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

The ROS2 distribution used in this work is **ROS2 Humble Hawksbill** (released 2022), the eighth ROS2 release. It is based on Ubuntu 22.04 (Jammy Jellyfish) and remains officially supported until at least May 2027, making it a stable and well-supported choice for long-term robotics development projects [75].



Figure 4.1: ROS2 Humble Hawkbill Logo.

4.2.1 ROS Filesystem

ROS employs a structured approach to organizing code and resources, centered around **packages**, the fundamental unit of organization. A ROS **workspace** is a directory where packages are developed and compiled, containing four sub-directories: `src`, which holds the packages; `build`, for compiled code; `install`, for installed packages; and `log`, for logging information [76].

Each package is further organized into sub-directories such as `config`, `include`, `launch`, and `src`, and contains two key files. The `package.xml` file holds metadata and dependency declarations, ensuring correct construction and incorporation of packages into the broader ROS ecosystem. The `CMakeLists.txt` file contains instructions for building the package, including compilation, linking, and required dependencies.

Launch files are used to start multiple nodes concurrently, automating complex system startup by specifying nodes, parameters, and additional configurations [76]. **Parameter and configuration files**, written in YAML format, store parameters that customize node behavior without requiring code modifications. Source files implementing node functionality are written in Python or C++ and stored within the `src` directory of each package.

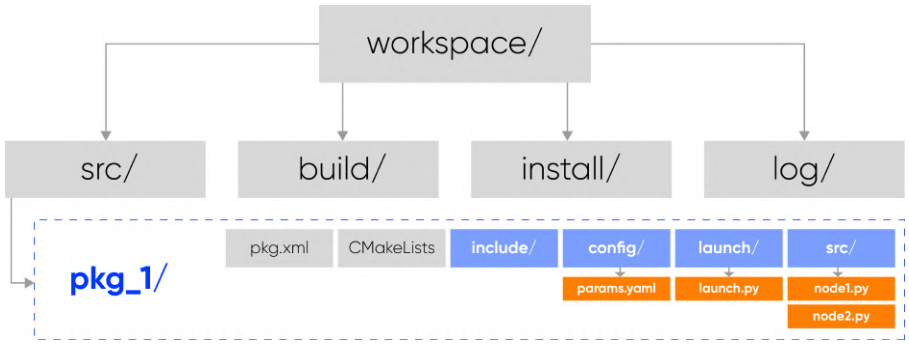


Figure 4.2: ROS2 workspace filesystem structure.

4.2.2 ROS Communication System

The ROS communication system can be visualized as a computational graph in which nodes are vertices and the communication channels between them are edges [76]. The framework offers three primary communication patterns:

- **Topics** facilitate asynchronous communication using a publisher–subscriber paradigm with a strongly typed interface. A node wishing to send data takes on the role of **publisher**, while nodes wishing to receive that data become **subscribers**. This model allows multiple nodes to publish and subscribe to the same topic simultaneously.
- **Services** represent a synchronous request–response communication pattern, employed for tasks that require confirmation of completion or the receipt of specific information. A service is structured into two sections, request and response, and a calling node is not required to block during the call.
- **Actions** are designed for long-running, goal-oriented tasks that require periodic feedback. An action is composed of three messages, a goal, feedback, and result.

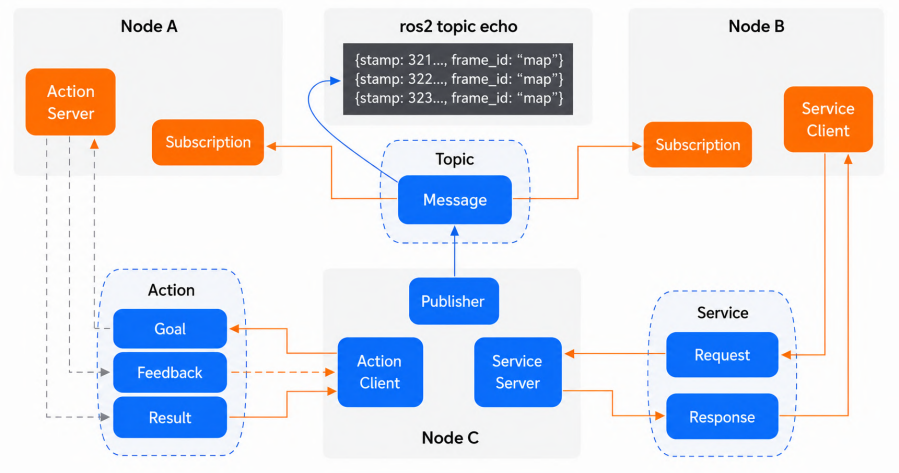


Figure 4.3: ROS 2 node interfaces: topics, services, and actions. Adapted from [74]

4.2.3 ROS2 Tooling

ROS2 provides several tools for robot development, visualization, and debugging, notably RViz2 for real-time visualization and TF2 for coordinate frame management.

RViz2

RViz2 or ROS Visualization 2 is a three-dimensional visualization tool developed for ROS2. It allows developers to visualize sensor data, robot models, and a wide range of information pertaining to a robot's environment in real time [77]. RViz2 supports point clouds, laser scans, robot states, maps, and more, making it invaluable for debugging and developing complex robotic systems in both simulation and real-world deployments. Users can interactively configure displays, adjust parameters, and monitor robot status.

In the context of SLAM, RViz2 provides real-time visual feedback on map construction and robot localization, facilitating the fine-tuning of algorithms by highlighting potential issues in the robot's trajectory, landmark detection, and obstacle representation [77].

TF2

TF2 is the second generation of the Transform Library in ROS, which automates the management of coordinate frame transformations in robotic systems [78]. As robots grow more complex and rely on data from multiple sensors distributed across multiple compute nodes, manually managing reference frames becomes cruicial. TF2 addresses this by providing two key components:

- **Broadcasters**, which disseminate transformation information between frames on a regular basis.
- **Listeners**, which receive and store these messages in a chronologically ordered list, enabling queries about the transformation between any two frames at any given time, with interpolation in cases of partial information.

4.3 Simulation Environment

ROS2 supports a wide range of simulation environments such as Webots, Gazebo, Isaac Sim and even Unity or Unreal Engine. The key differences between these simulation environments are summarized in Table 4.2.

Table 4.2: Comparison of robotics simulation platforms.

Feature	**Gazebo**	**Isaac Sim**	**Webots**	**Unity**
Physics & sensors realism	Medium	High	Medium	Medium
ROS2 integration	Native	Native	Native	Plugin
GPU acceleration	No	Yes	No	Partial
Reinforcement learning	Limited	Yes	Limited	Yes
Digital twin support	No	Yes	No	Partial
GUI configuration	Limited	Yes	Yes	Yes
Resource requirements	Low	High	Low	Medium

Isaac Sim was selected for this project as it is becoming a leading standard in modern robotics simulation, offering more realistic physics and accurate sensor simulation than alternatives. Unlike simulators that abstract low-level behavior, Isaac Sim requires implementing critical components such as controller tuning, bringing the simulation closer to real-world deployment. Its integration with the Omniverse GUI further simplifies robot setup, sensor configuration, and environment creation without the need for complex configuration files.

4.3.1 NVIDIA Isaac Sim

NVIDIA Isaac Sim is a robotics simulation platform built on NVIDIA Omniverse and designed for developing, testing and validating robotic systems in realistic virtual environments [79]. It supports importing robot models using formats such as URDF and CAD models and provides realistic physics simulation with GPU acceleration. Isaac Sim also supports ROS 2 integration, sensor simulation, reinforcement learning workflows and digital twin applications.

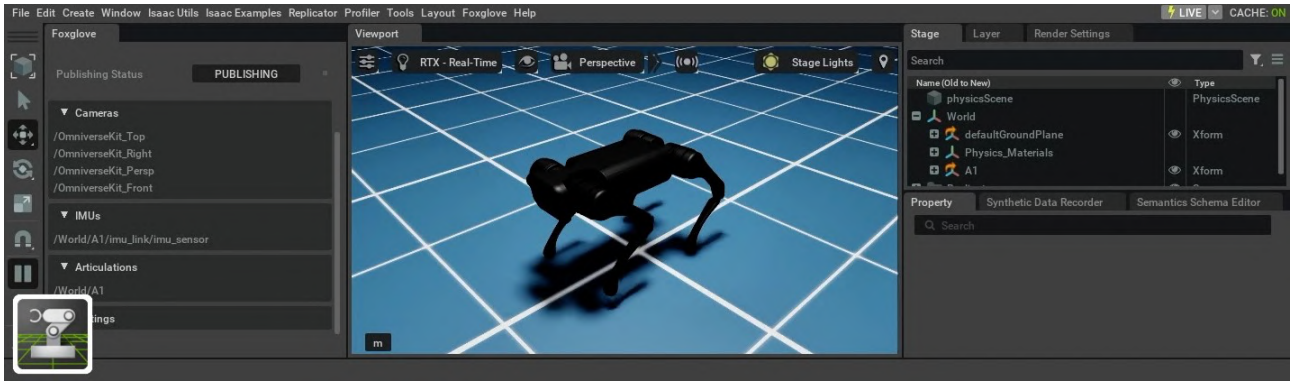
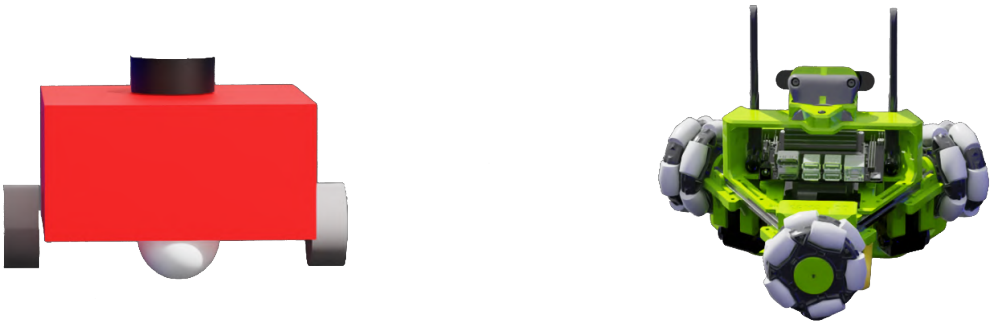


Figure 4.4: NVIDIA Isaac Sim's Interface.

Robot Model

Two different types of robots were used in the simulation. The first is a differential-drive robot modeled directly in Isaac Sim using shapes like cubes, spheres, and cylinders assembled to form the robot body, two actuated wheels, and two passive caster wheels, one at the front and one at the rear, ensuring stability during both linear motion and rotation. A 2D LiDAR sensor was added on top of the model, and the complete assembly was exported as a `.usd` file for use in the simulation. The second robot is the NVIDIA Kaya, an open-source three-wheel omnidirectional robot available directly from Isaac Sim's asset library. Unlike the differential-drive robot, this configuration allows movement in all directions without changing the robot orientation. The Kaya uses three omnidirectional wheels placed at 120° from each other, giving it full planar mobility, which is useful for testing holonomic SLAM and navigation and for evaluating the behavior of the designed system across multiple robot types. Figure 4.5 shows both robot models as configured in Isaac Sim.



(a) Differential-drive robot (b) NVIDIA Kaya omnidirectional robot

Figure 4.5: Robot models used in Isaac Sim

Action Graphs

Isaac Sim uses a visual scripting system called **Action Graphs**, built on top of the Omniverse OmniGraph framework [79]. Action Graphs allow users to define robot behavior and control logic through a node-based interface, without requiring manual code implementation for every component. Each node in the graph represents a specific operation, such as reading sensor data, computing velocities, or sending commands to actuators, and nodes are connected through typed ports that define the flow of data and execution.

This approach is particularly useful for configuring low-level robot controllers directly within the simulator. For instance, a differential drive robot can be set up by connecting an `On Playback Tick` node which triggers execution at every simulation step to a `Differential Controller` node that computes wheel commands from linear and angular velocity inputs. The resulting velocity commands are then passed to an `Articulation Controller` node, which applies them to the robot's joints. Joint names are supplied through `Constant Token` nodes combined into an array via a `Make Array` node, as illustrated in Figure 4.6.

These action graphs can be established in three ways, either using the UI or scripting inside the extension workflow or the standalone Python workflow

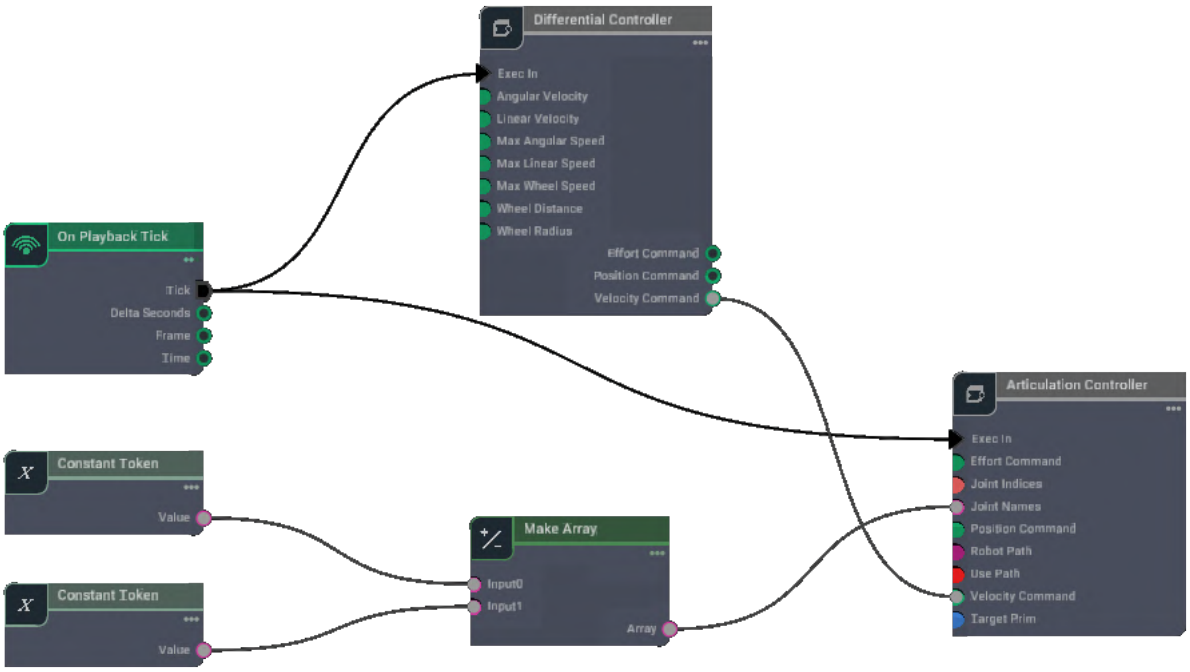


Figure 4.6: Simple differential control action graph.

4.3.2 ROS2 Integration with Isaac Sim

In this project, Isaac Sim was used to validate the Active Collaborative SLAM system before real-world deployment. The simulation included robot motion, LiDAR sensing, SLAM, navigation, and multi-robot exploration within a custom indoor environment. ROS2 communication was implemented through Action Graphs and the Isaac ROS2 Bridge.

Three Action Graphs were configured for each robot:

- **LiDAR Scan Publishing:** An `Isaac Read LiDAR Beams Node` reads the simulated scan data and forwards it to a `ROS2 Publish Laser Scan` node that publishes to the `/scan` topic. This graph is identical for both robot types since both are equipped with the same LiDAR sensor.
- **Odometry and TF Publishing:** An `Isaac Compute Odometry Node` computes the robot's position and velocity at each simulation step, and the results are forwarded simultaneously to a `ROS2 Publish Odometry` node publishing to `/odom`, a `ROS2 Publish Raw Transform Tree` node, and a `ROS2 Publish Transform Tree` node, ensuring that the full transform tree is available to the ROS2 system.
- **Velocity Command Subscription:** A `ROS2 Subscribe Twist` node receives commands from `/cmd_vel`. Here the two robots differ: for the differential-drive robot, the commands are passed to a `Differential Controller` and finally to an `Articulation Controller` that drives the wheels; for the omnidirectional robot, the controller is replaced by a `Holonomic Controller` combined with a `Usd Setup Holonomic Robot` node that configures the wheel geometry, enabling full planar motion.

These Action Graphs are illustrated in Figures 4.7, 4.8, 4.9, and 4.10.

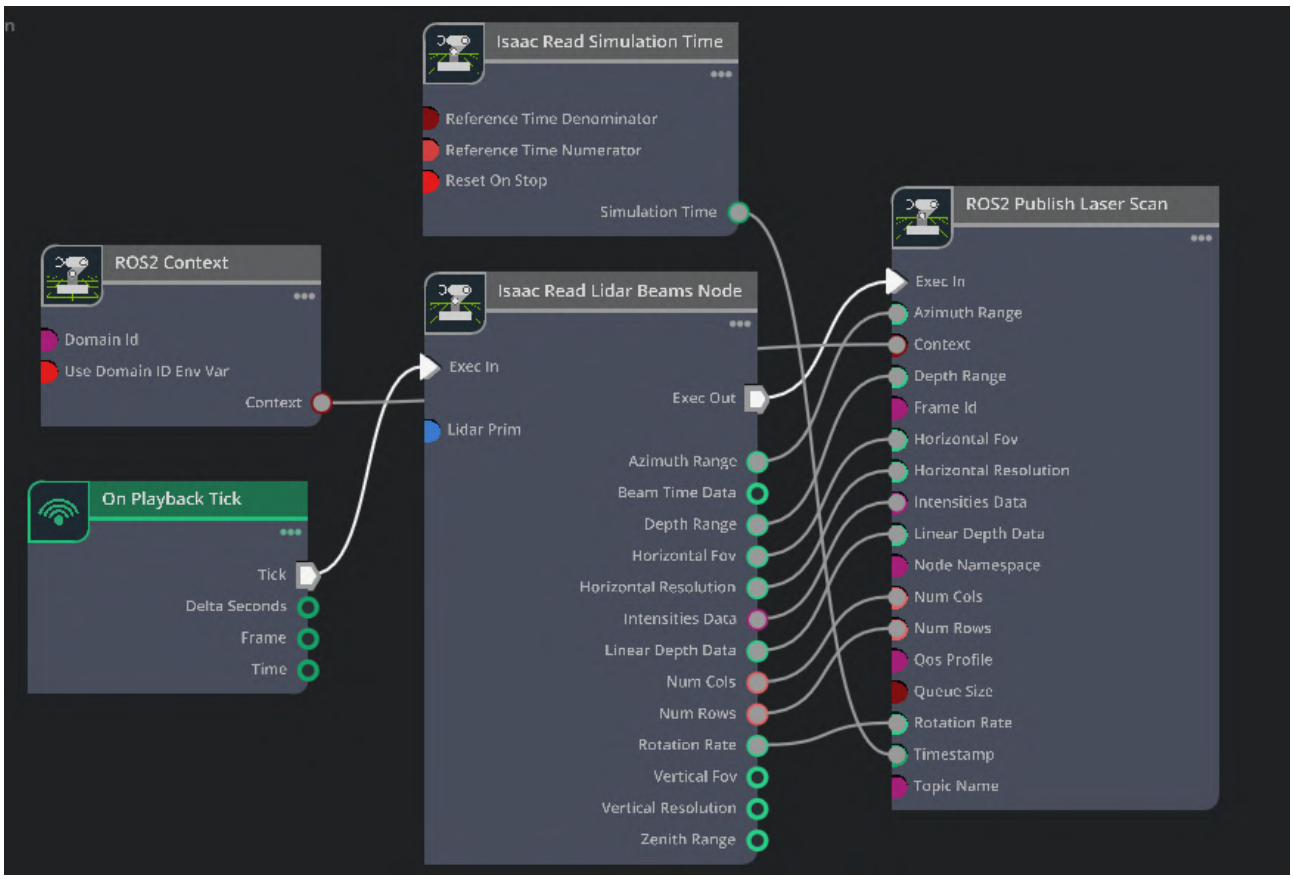


Figure 4.7: Action Graph for LiDAR scan publishing via the `/scan` topic.

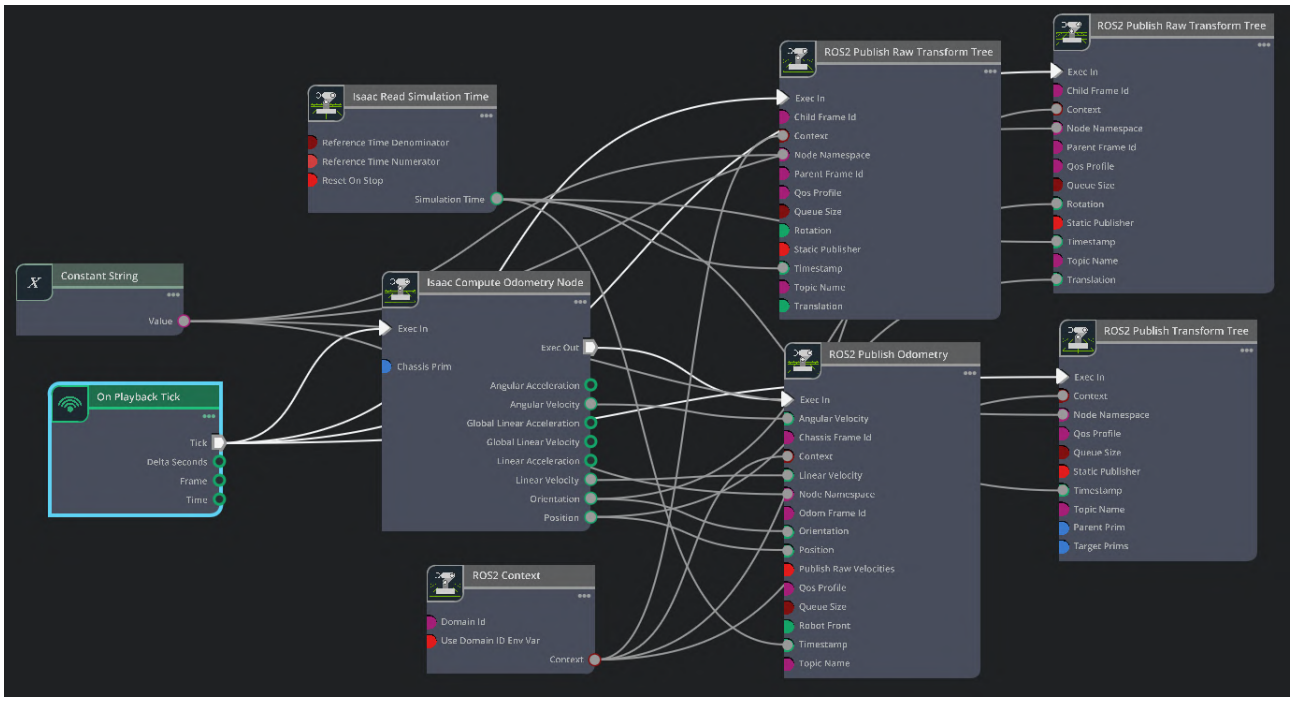


Figure 4.8: Action Graph for odometry and transform tree publishing via `/odom` and `/tf`.

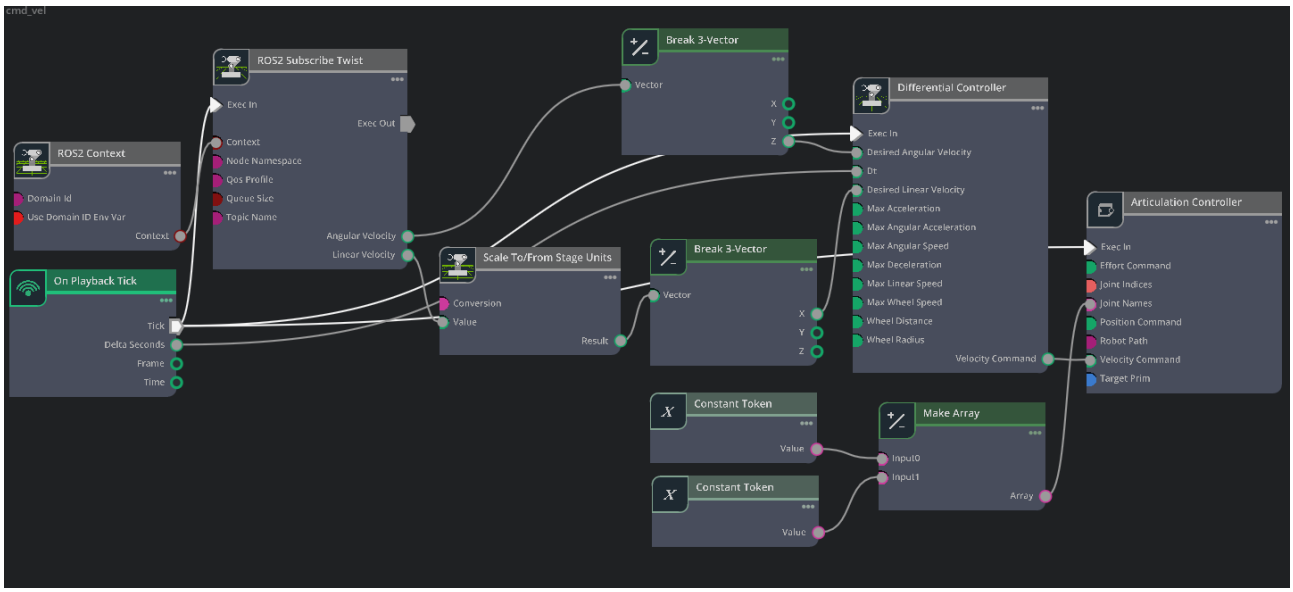


Figure 4.9: Action Graph for velocity command subscription and differential drive controller.

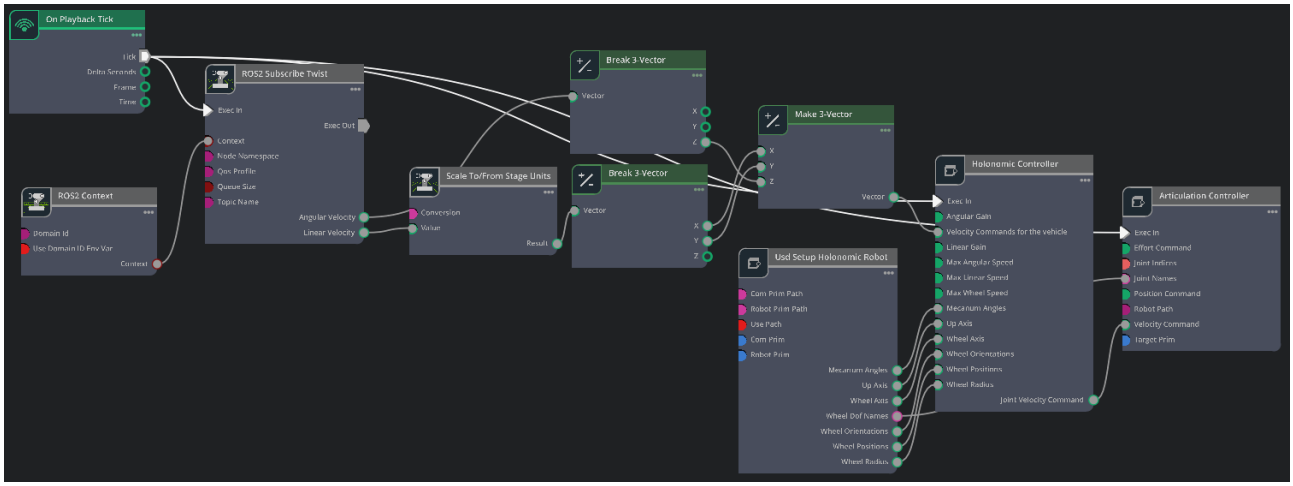


Figure 4.10: Action Graph for velocity command subscription and holonomic controller.

4.4 SLAM Toolbox

SLAM Toolbox is a set of tools and functionalities designed for 2D SLAM in ROS 2. It allows the robot to build a map of the environment while localizing itself at the same time using LiDAR scans, odometry and TF transforms. The package provides several useful functionalities such as map generation, map saving, pose-graph serialization, localization mode, map continuation, graph optimization and RViz interaction tools [80].

The package provides a function to save the complete pose graph and mapping data. This allows the mapping session to be loaded again later to continue mapping, refine the generated map, or perform localization using previously recorded data without losing the original information. SLAM Toolbox also supports synchronous and asynchronous mapping modes. In this project, the `online_async` mode was used because it will be later used on real hardware and reduces the computation load on the system. Instead of processing every incoming LiDAR scan, this mode processes the scans using a best-effort approach and updates the map only when the robot moves enough to produce meaningful changes in the environment.

4.4.1 SLAM Toolbox Components

SLAM Toolbox is organized into several interconnected components, each responsible for a specific stage of the mapping pipeline. A full overview of these components and their interactions is illustrated in Figure 4.11.

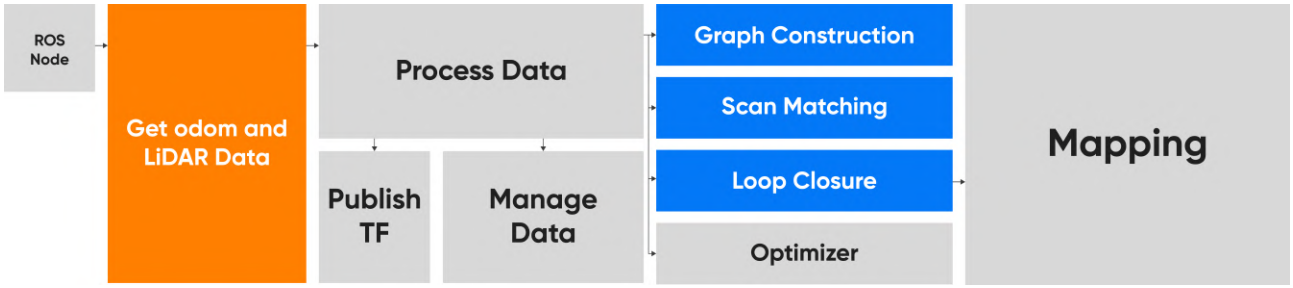


Figure 4.11: SLAM Toolbox Components Overview.

LiDAR Scan Acquisition

The pipeline begins with odometry and LiDAR data acquisition. Raw scans and odometry are received, wrapped into posed scan objects, and queued for processing. Laser metadata and sensor properties are also resolved at this stage, handling cases such as an upside-down mounted sensor. The resulting scans are passed downstream to the scan matching stage.

Odometry Estimation

Odometry estimation tracks the robot motion between consecutive scans using wheel velocities and motion models, providing an initial pose guess before scan matching is applied. The map-to-odometry transform is kept up to date and published continuously for use by downstream components.

Scan Matching

At the core of the processing stage, each new scan is matched against previously stored scans to refine the pose estimate. The result is added to the pose graph as a localized scan. Scan matching improves pose estimation beyond raw odometry alone [81], minimizing the alignment error between scans:

$$e = \sum_{i=1}^{n} \|p_i - T(q_i)\|^2 \tag{4.1}$$

where p_i and q_i are corresponding points between scans and T is the transformation between them.

Graph Construction

Each accepted scan is inserted into the pose graph as a new node, with edges encoding the spatial constraints derived from scan matching and odometry. Edge covariance information is updated before the constraints are forwarded to the optimizer.

Loop Closure

When the robot revisits a previously mapped area, the system computes the distance to candidate loop closure nodes and performs scan matching to obtain a fine alignment. If the response exceeds a threshold, a new constraint is added to the graph and all affected poses are corrected. Loop closure minimizes the pose inconsistency between revisited locations:

$$e_{loop} = z_{ij} - h(x_i, x_j) \tag{4.2}$$

where z_{ij} is the measured relation between two poses and $h(x_i, x_j)$ is the predicted relation.

Mapping

The mapping stage converts the corrected pose graph into an occupancy grid. All processed scans are collected and ray casting is applied to mark grid cells as free or occupied. The resulting map is then converted to a ROS navigation map and published periodically.

4.5 Navigation Stack

The Navigation2 (Nav2) stack is the ROS 2 navigation framework used to allow the robot to move autonomously inside the environment. It is considered the successor of the ROS navigation stack and is designed for autonomous mobile robots with different kinematic configurations such as differential-drive and omnidirectional robots. Nav2 allows the robot to move from one position to another while avoiding obstacles and continuously updating its trajectory according to the environment. The navigation system is composed of several servers, where each server is responsible for a specific task during the navigation process [82]. Figure 4.12 shows the general architecture of the Nav2 stack.

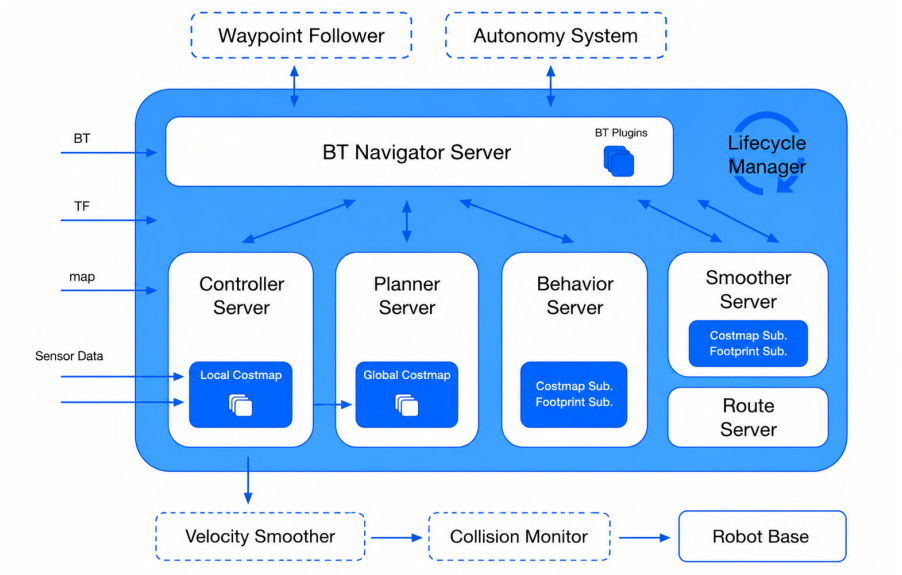


Figure 4.12: General architecture of the Nav2 stack. Adapted from [82]

4.5.1 Nav2 servers

Nav2 is organized into several servers, each handling a specific aspect of the navigation pipeline. They communicate through action interfaces coordinated by a central behavior tree navigator, forming a modular and recoverable navigation stack [82].

- **Planner Server.** The planner server is responsible for generating a global collision-free path between the robot's current position and the target goal, using the environment map and global costmap. In this project, the NavFn planner is used, which applies the A* search algorithm to find an optimal route while avoiding occupied and inflated obstacle regions. The resulting path is forwarded to the controller server for execution.
- **Controller Server.** The controller server follows the global path by continuously computing velocity commands that move the robot along the trajectory. The DWB (Dynamic Window Approach) controller is used, evaluating multiple velocity samples and selecting the best command based on robot kinematics, obstacle proximity, and trajectory alignment. The selected command is published as velocity output to drive the robot.
- **Behavior Server.** The behavior server handles recovery actions when navigation fails, for example when the robot becomes stuck or the planner cannot find a valid path. Recovery behaviors include clearing the costmap, rotating in place, backing up, and replanning. These actions improve navigation robustness in difficult or unexpected situations.

- **BT Navigator.** The BT Navigator coordinates the entire navigation process using a behavior tree that sequences interactions between the planner, controller, and behavior servers. It manages the flow of computing a path, following it, detecting failures, triggering recovery behaviors, and replanning when needed. This tree-based structure makes the navigation pipeline modular and straightforward to extend or modify.

4.5.2 Costmaps

The navigation stack relies on two costmaps that provide spatial representations of the environment at different scales, each built from multiple layers.

- **Global Costmap.** The global costmap represents the complete known environment based on the generated map. It is used by the planner server to compute a collision-free global path from the robot's current position to the target goal.
- **Local Costmap.** The local costmap represents the immediate surroundings of the robot using real-time LiDAR data. It is continuously updated during navigation and is primarily used by the controller server for obstacle avoidance and local replanning.
- **Inflation Layer.** The inflation layer enlarges obstacle boundaries within the costmap to maintain a safety margin around walls and objects, preventing the robot from planning paths that pass too close to them.
- **Obstacle Layer.** The obstacle layer dynamically updates the costmap using live sensor measurements. Obstacles detected by the LiDAR are inserted into the local costmap in real time as the robot moves.

4.5.3 Navigation Workflow

The navigation process begins when a goal is sent to the robot via RViz2 or an autonomous system. The planner server then computes a global collision-free path using the global costmap, which the controller server follows by continuously issuing velocity commands while monitoring the local costmap for nearby obstacles. If a new obstacle appears, the local costmap is updated from LiDAR data and a new local trajectory is computed on the fly. Should the planner fail entirely, the behavior server triggers recovery actions and restarts the navigation attempt. Figure 4.13 illustrates this workflow.

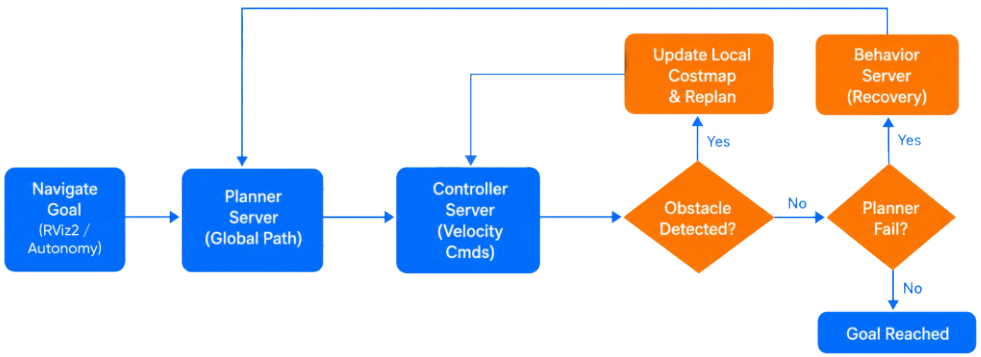


Figure 4.13: Navigation workflow.

4.6 Active Collaborative SLAM Simulation

The proposed exploration framework follows the Active SLAM approach using prior connectivity graphs, as introduced by Soragna et al [61]. In this approach, a topological representation of the environment is available before exploration starts and is used to guide the robot during the exploration process. While the robot moves through the environment, SLAM is performed simultaneously to incrementally generate the occupancy map and estimate the robot pose.

A custom indoor environment was created in Isaac Sim to test the complete Active Collaborative SLAM system. The environment is based on Floor 0 of the research center at the École Nationale Polytechnique, reconstructed using real metric measurements to accurately replicate the actual spatial layout. It features a main corridor of approximately 30 meters, multiple interconnected rooms, narrow passageways, dead-end branches, and asymmetric wall configurations, providing sufficient complexity to evaluate SLAM stability, navigation performance, obstacle avoidance, and autonomous exploration behavior in both single-robot and multi-robot scenarios, as shown in Figure 4.14.

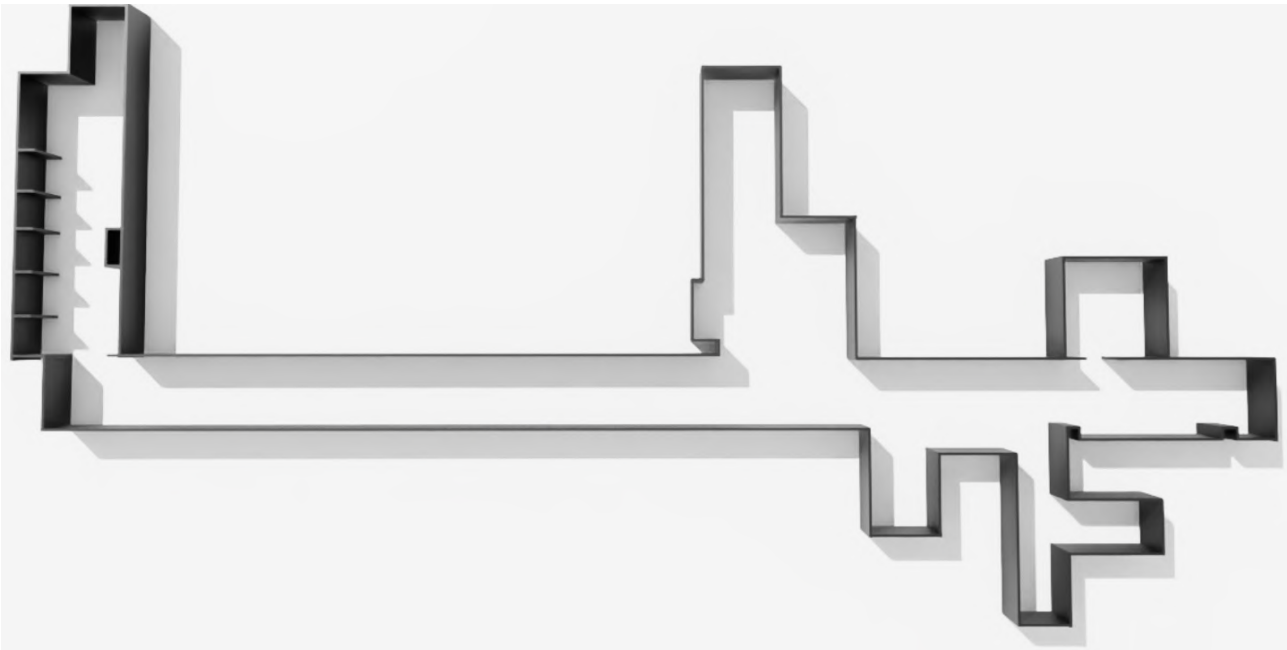


Figure 4.14: Floor 0 of the ENP research center reconstructed in Isaac Sim.

4.6.1 Prior Environment Graph Construction

To obtain a reference map of the environment, Isaac Sim's built-in occupancy grid generation tool was used to export a PGM file representing the floor plan, as shown in Figure 4.15. In this representation, black pixels correspond to occupied cells (walls and obstacles), white pixels denote free navigable space, and gray pixels indicate unknown or unobserved regions. This ground truth map serves as a reference baseline for evaluating the accuracy of the SLAM-generated maps, enabling direct comparison between the expected layout and the maps produced during both simulation and real-world deployment.

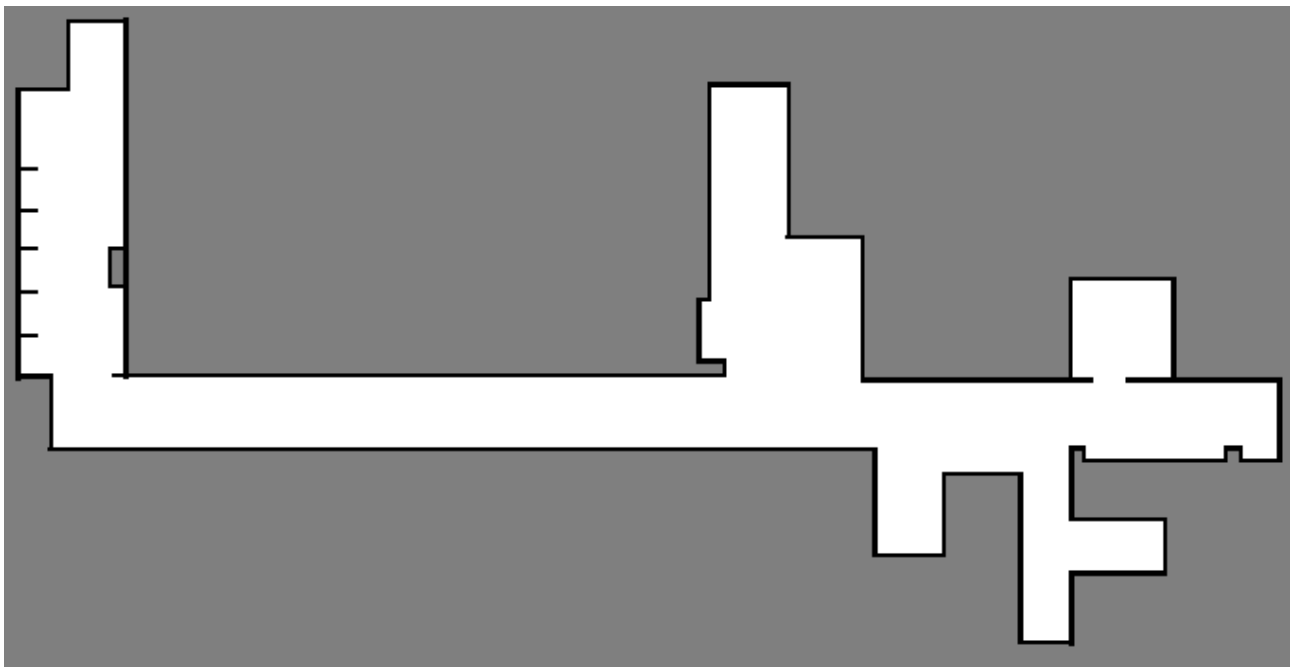


Figure 4.15: Occupancy grid map of Floor 0.

After that, a prior topological representation of the workspace is constructed in the form of a graph. This graph serves as an abstract description of the environment and is later used as a priori information for coverage path planning and exploration. Figure 4.16 shows the prior graph for this environment.

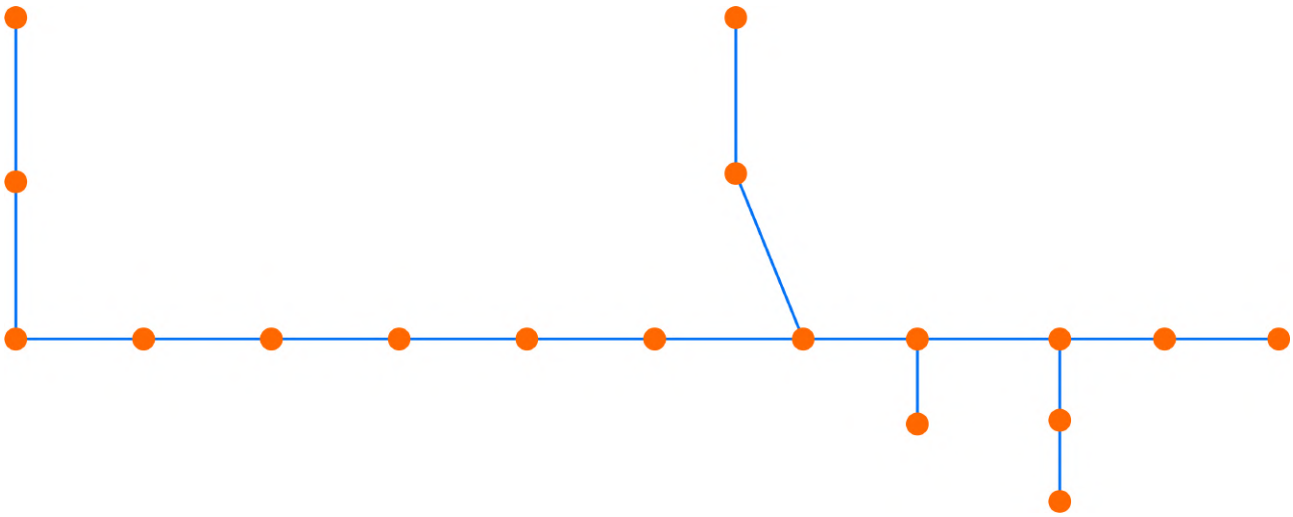


Figure 4.16: Prior graph for the Floor 0 environment.

Each node in the graph corresponds to a physical location in the environment and is associated with planar coordinates (x, y) expressed in the map reference frame. These coordinates are later used to generate navigation goals for the robot. The graph structure is then stored in a YAML file containing the node coordinates and connectivity information. This representation simplifies graph loading and allows easy modification of the environment topology when required.

The constructed graph is also designed to be reusable and scalable. In later stages of the project, the same graph can be partitioned and assigned to multiple robots, allowing collaborative exploration while maintaining independent navigation and SLAM processes for each robot.

Eulerianisation & Node Sequencing

Once the prior connectivity graph is constructed, a coverage planning strategy is applied to determine the optimal visitation order of the graph nodes. However, as discussed in Section 3.3, the prior graph is not Eulerian, meaning some vertices have an odd degree, which prevents a complete traversal without revisiting edges. To address this, the graph must first be Eulerianised by computing a minimum-weight perfect matching over the set of odd-degree vertices and augmenting the graph with duplicate edges connecting each matched pair, as illustrated in Figure 4.17.

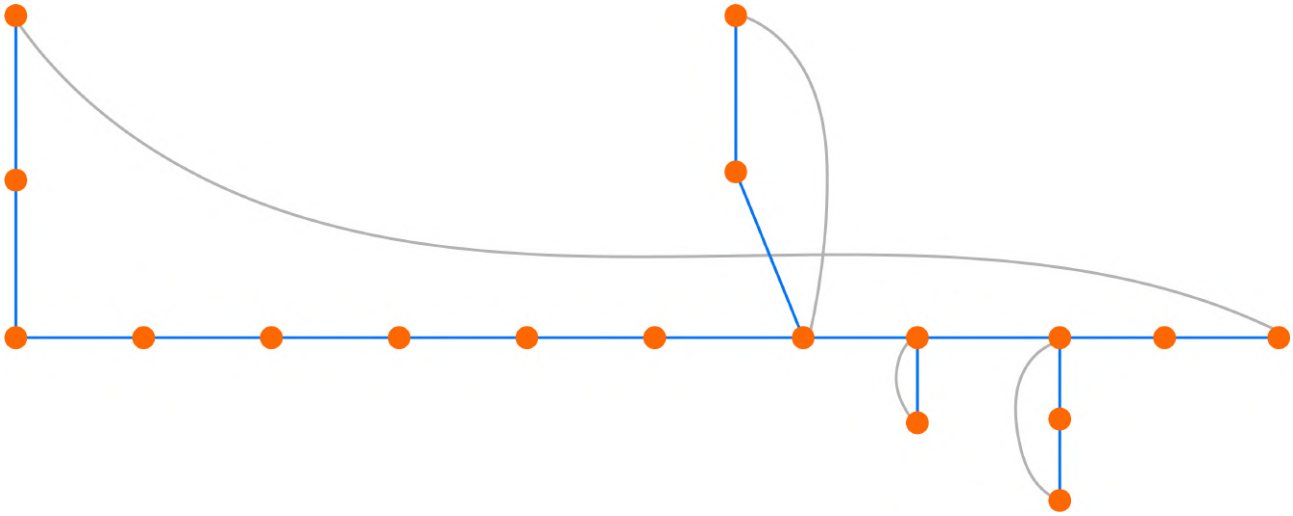


Figure 4.17: Eulerianised prior graph for the Floor 0 environment.

With the Eulerian property satisfied, the Chinese Postman Problem (CPP) algorithm is applied to extract the minimum-cost Eulerian circuit, guaranteeing that every edge is traversed at least once while minimizing redundant travel. The resulting node visitation sequence defines the coverage path assigned to the robots, as shown in Figure 4.18.

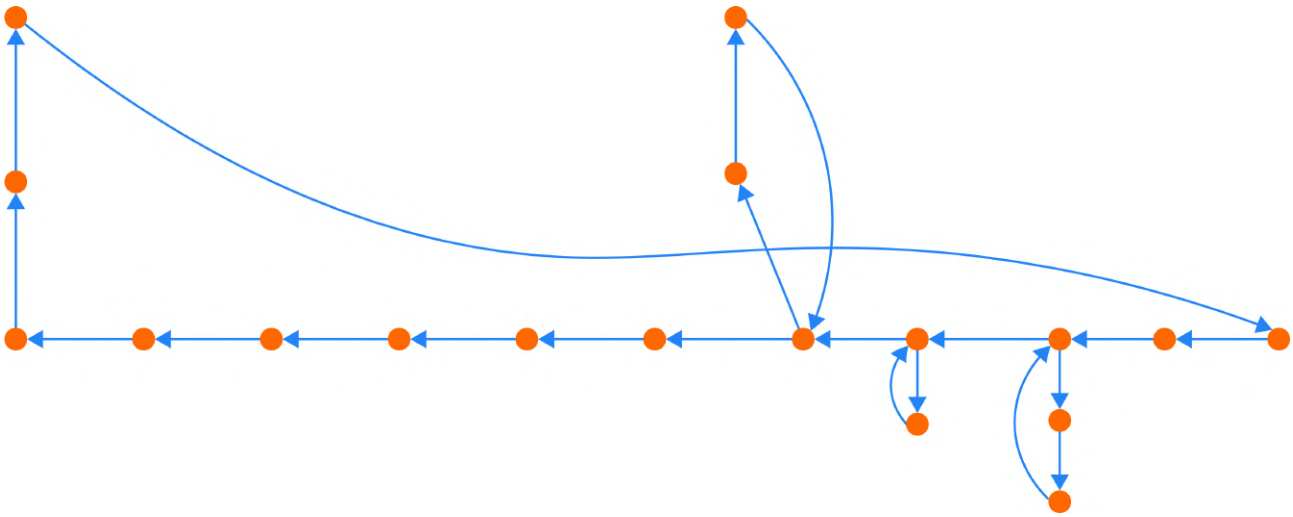


Figure 4.18: Eulerian circuit extracted by the CPP algorithm on the prior graph.

4.6.2 Multi-Robot ROS2 Architecture

In this work, a two-robot configuration is considered. Both robots operate concurrently in the same simulated environment and rely on the same exploration principles introduced in the single-robot case, while maintaining fully independent sensing, localization and navigation pipelines. Each robot in the simulation runs its own ROS 2 nodes and publishes its own sensor data. To avoid conflicts between topics and transforms, namespaces are used for every robot. A prefix is added to all robot topics using the format:

$$/robotX \tag{4.3}$$

where X identifies the robot (e.g., `robot1`, `robot2`). Each robot publishes its own namespaced ROS topics — including `/robotX/scan`, `/robotX/odom`, and `/robotX/cmd_vel` — ensuring fully isolated data streams that allow each robot to run its own navigation stack, localization, and SLAM process independently. The TF tree is also separated for each robot to avoid transform conflicts between frames such as `base_link`, `odom`, and `laser`. Each robot therefore maintains its own independent coordinate frames and localization information. Figure 4.19 illustrates the ROS 2 architecture used for the multi-robot system.

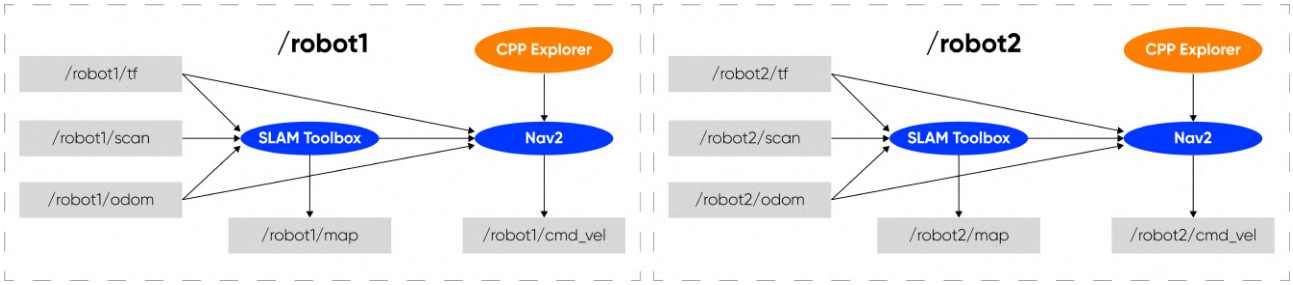


Figure 4.19: ROS 2 multi-robot architecture using namespaces.

Shared Environment

Although the robots operate independently, they share the same simulated environment, exploring it simultaneously while maintaining separate navigation and sensing pipelines. Each robot navigates autonomously, performing obstacle avoidance and building its local map from its own LiDAR scans, without any direct coordination at the navigation level. This setup effectively evaluates multi-robot exploration behavior, autonomous navigation, and collaborative mapping strategies under realistic simulation conditions. Figure 4.20 illustrates the two robots deployed in the Isaac Sim environment, where the NVIDIA Kaya robot is positioned at the far end of the corridor, and the differential drive robot is located midway along the same corridor.

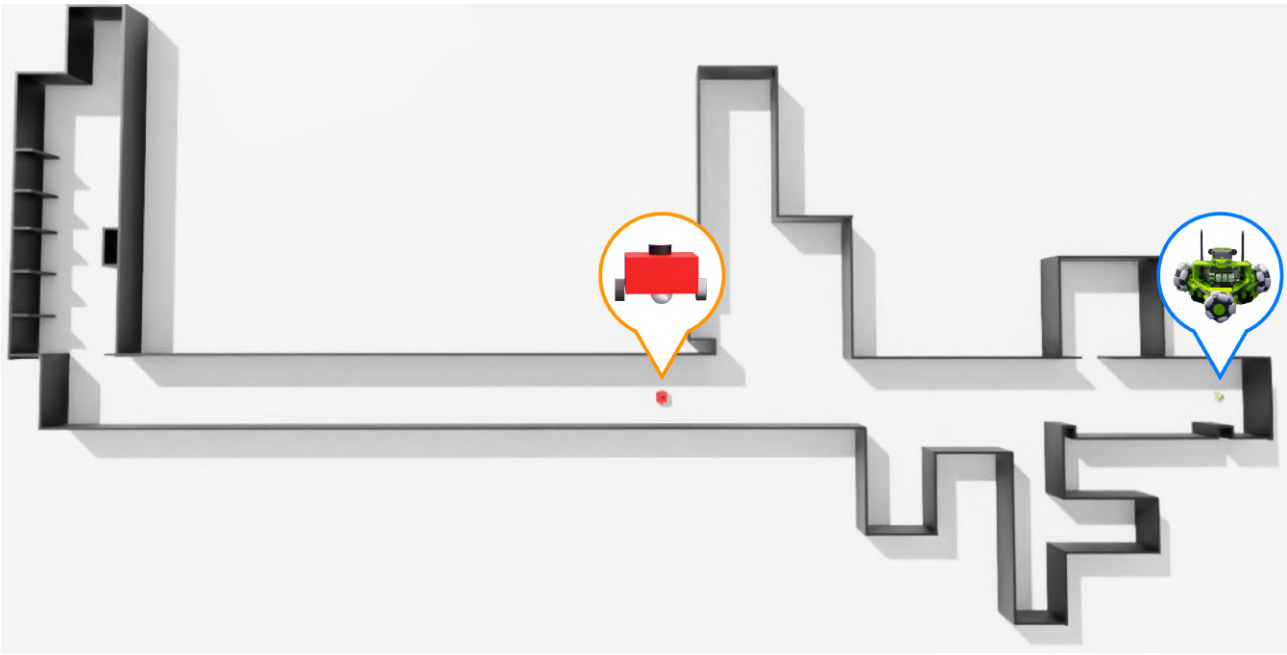


Figure 4.20: Multi-robot simulation environment in Isaac Sim.

Graph Partitioning

The objective of graph partitioning is to divide the environment into multiple regions and assign a different subset of nodes to each robot. As a result, each robot focuses on exploring a specific portion of the environment while minimizing overlap with the other robots. As explained in 3.4.1, graph partitioning is performed using a Breadth-First Search (BFS) strategy. Each subgraph is then converted to an eulerian graph, ready for the CPP Algorithm exploration as shown in Figure 4.21.

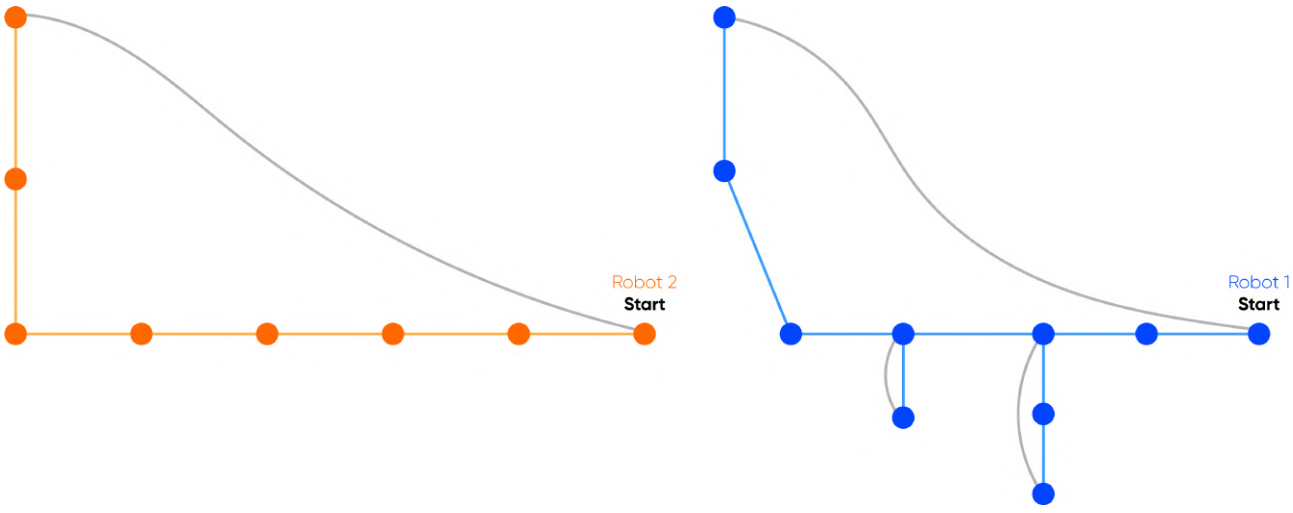


Figure 4.21: Partitioning of the prior graph into two subgraphs.

Independent Exploration and Mapping

Once the graph has been partitioned, each robot executes its own exploration process independently. The assigned graph is first processed by the CPP-based coverage planner to generate an ordered sequence of navigation goals. The generated goals are then sent to the local Nav2

stack of each robot. As the robots move through the environment, they simultaneously perform localization, obstacle avoidance, and map generation using their onboard LiDAR sensors. Since each robot operates within its own namespace and maintains an independent TF tree, navigation and SLAM computations remain isolated while exploration is performed in parallel.

As exploration progresses, each robot generates its own local map while covering the nodes assigned to its partition. The resulting maps will later be combined to obtain a more complete representation of the environment. Figure 4.22 shows both robots exploring different regions of the environment simultaneously.

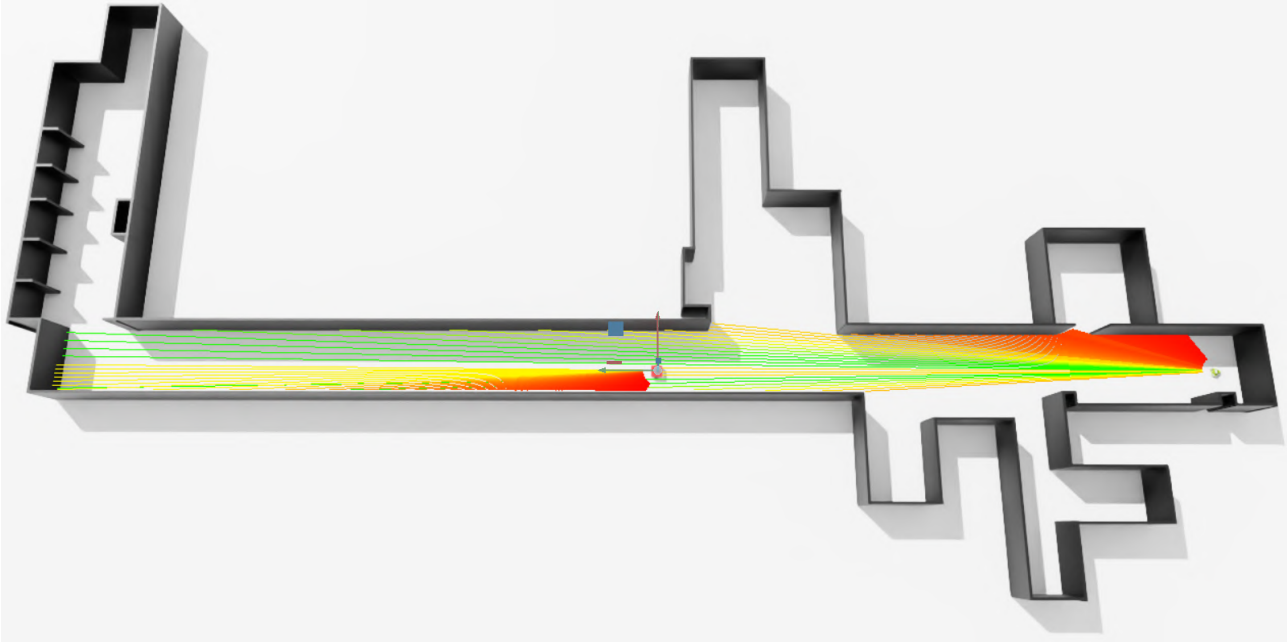


Figure 4.22: Simultaneous exploration performed by two robots

Map Merging Architecture

As explained in 3.4.3, once each robot has completed its exploration phase, the individually built maps must be combined into a single consistent global map. Each robot's SLAM Toolbox instance continuously publishes its local occupancy grid to its own namespaced topic (i.e., `/robot1/map` and `/robot2/map`). These partial maps are then consumed by the `map_merger` node, which subscribes to all robot map topics and fuses them into a unified representation published on the `/merged_map` topic, as illustrated in Figure 4.23.

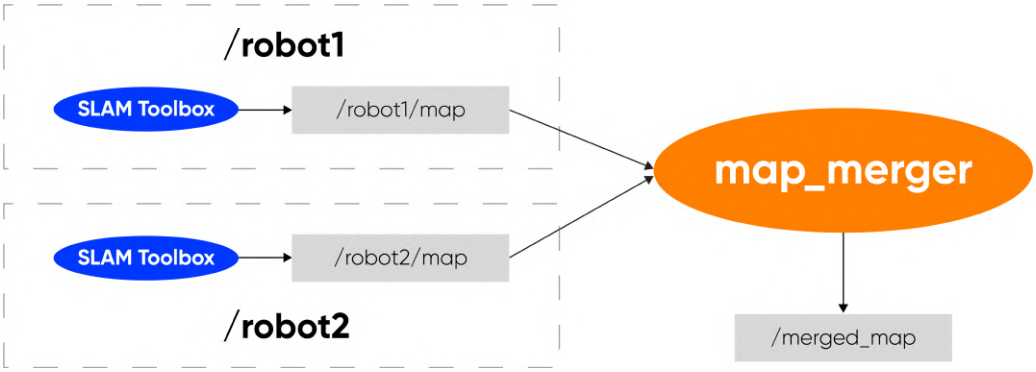


Figure 4.23: Map merging architecture.

The merging process aligns the individual maps by estimating the relative transformations between them, resolving overlapping regions, and combining free space and obstacle information from all sources. The resulting global map provides a complete view of the explored environment, which can subsequently be used for path planning, localization, or mission evaluation.

Handling Invalid Goals and Blocked Edges

In practical applications, some graph nodes may correspond to locations that become unreachable during execution. Similarly, some edges may be blocked by obstacles that were not represented in the prior graph. When Nav2 fails to compute a valid path toward a navigation goal, the goal is considered unreachable. This can happen when the target lies outside the navigable map, is located inside an obstacle, or when no valid trajectory can be generated.

Instead of stopping the exploration process, the current goal is automatically aborted and skipped. The explorer node then proceeds to the next node in the exploration sequence. This mechanism prevents exploration deadlocks and allows the system to continue operating even when errors exist in the prior graph representation. the execution flow is illustrated in Figure 4.24.

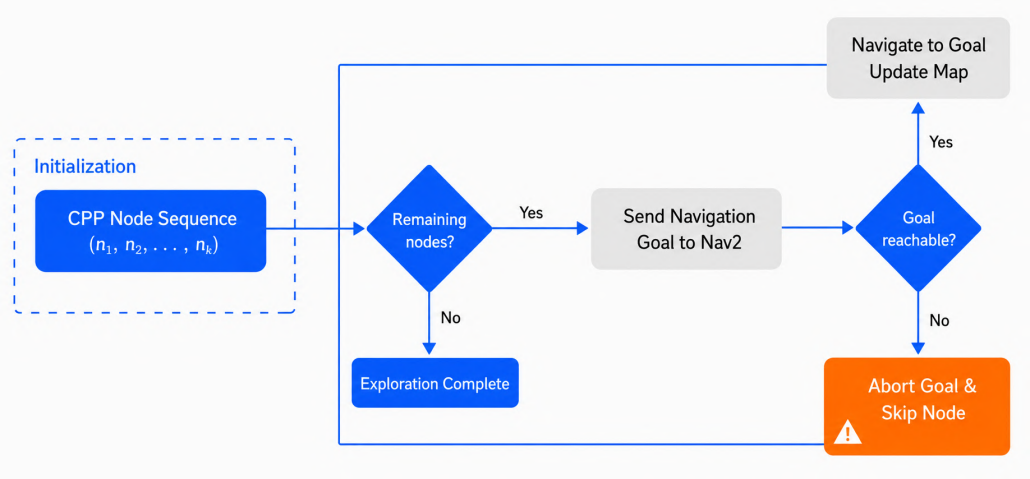


Figure 4.24: Execution flow of navigation goal handling with invalid goal skipping.

4.7 Conclusion

This chapter presented the full simulation pipeline of the Active Collaborative SLAM system within a custom Isaac Sim environment faithfully reconstructed from Floor 0 of the ENP research center. The ROS2-based architecture was validated across all system layers, from individual robot navigation and SLAM to collaborative workload partitioning, active frontier exploration, and post-exploration map merging. The simulation results confirm that the proposed system operates correctly in a realistic multi-robot setting, establishing a solid foundation for deployment on physical robotic hardware, as described in the following chapter.

Chapter 5

Deployment on Real Robots

5.1 Introduction

While simulation provides a controlled and repeatable environment for validating algorithms, deploying robotic systems on real hardware introduces a new layer of complexity that cannot be fully reproduced in software. Sensor noise, mechanical tolerances, communication latencies, and computational constraints all contribute to a significant gap between simulated and real-world behavior.

This chapter presents the complete deployment of the Active Collaborative SLAM system on two physical robot platforms:

- A custom-built differential drive robot designed and assembled from individual components.
- Festo Robotino v2.1, a commercial omnidirectional platform that required custom hardware and software extensions to integrate into the ROS2-based architecture.

For each platform, the hardware design, low-level control, and high-level software stack are described in detail. The chapter then covers the deployment of the SLAM and navigation pipelines on real hardware, including the parameter tuning required to adapt the system to physical constraints, the challenges encountered during testing, and the strategies used to address them. Finally, the Active SLAM and collaborative exploration frameworks are deployed and evaluated on the real robots, completing the transition from simulation to physical operation.

5.2 First Robot: Custom Made

The first platform was designed from the ground up, with each component selected to meet the specific requirements of indoor autonomous navigation.

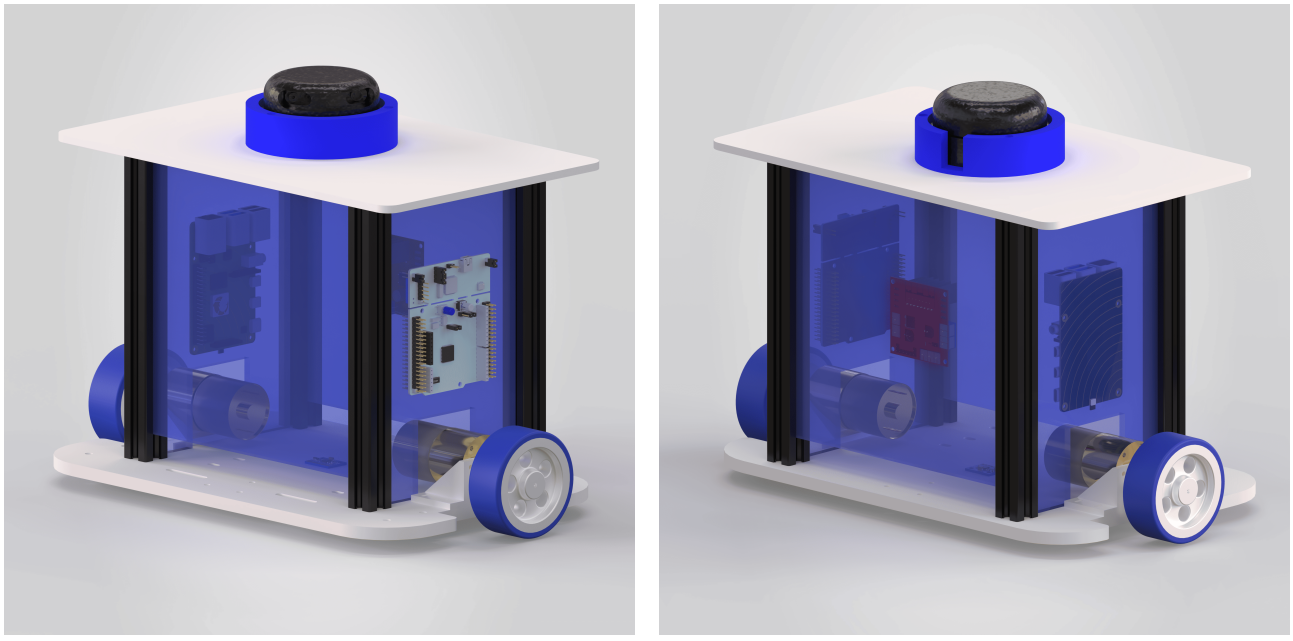
5.2.1 Hardware

Building a functional mobile robot requires careful selection and integration of mechanical, electronic, and computational components; what follows covers each of these layers in turn.

Mechanical Design

The custom robot is a differential-drive platform designed for indoor navigation. The chassis consists of two wooden plates connected by four aluminum standoffs at each corner, providing structural rigidity while allowing easy access to internal components.

The chassis was modelled in SOLIDWORKS prior to fabrication and follows the same design as the simulation platform, with two caster wheels at the front and back to replicate the conditions used in simulation, as shown in Figure 5.1.



(a) Front View (b) Back View

Figure 5.1: Custom Robot Platform Design.

Actuation

When selecting the actuation system, the following criteria were considered:

- **Torque:** sufficient to move the platform under full load on indoor surfaces.
- **Encoder resolution:** high enough for accurate odometry computation.

- **Driver simplicity:** easy integration with the STM32 microcontroller via PWM signals.

The **GB37Y3530-12V-251R** gear motor was selected as it satisfies all of the above requirements and was available in the FABLAB. Its key specifications are listed in Table 5.1.

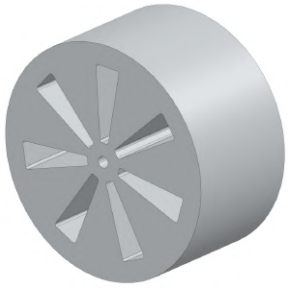
Table 5.1: Motor Specifications.

Parameter	**Value**
Gear Ratio	43.8:1
No-load Speed	251 ± 10% RPM
No-load Current	350 mA
Stall Torque	18 Kg.cm
Insulation Resistance	20 MΩ
Encoder Type	Hall Effect
Encoder Voltage	5 V
Encoder Resolution	1900 tick/revolution
Weight	205 g



Figure 5.2: GB37Y3530-12V-251R Gear Motor with Encoder.

The wheels have a diameter of **6 cm** and went through two design iterations. The first prototype was modeled in SOLIDWORKS and fabricated via 3D printing, as shown in Figure 5.8. However, this version presented a poor mechanical coupling with the motor shaft, which caused slippage under load.



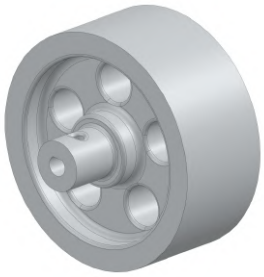
(a) 3D Model using SOLIDWORKS



(b) 3D Printed Result

Figure 5.3: Initial Prototype of the Platform's Wheels.

To address this, a second prototype was designed with a dedicated shaft extension to ensure a rigid connection to the motor. This version was machined in aluminum for improved structural robustness, as shown in Figure 5.8. A silicone coating was added to the wheel surface to enhance traction on indoor floors.



(a) 3D Model using SOLIDWORKS

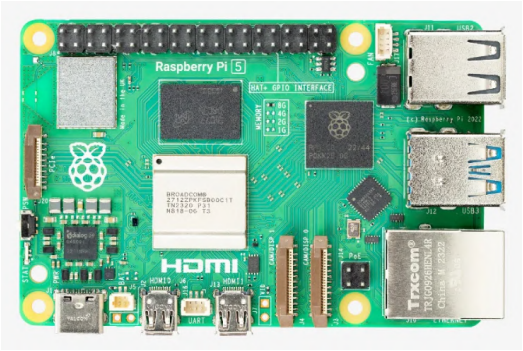


(b) Machined Result

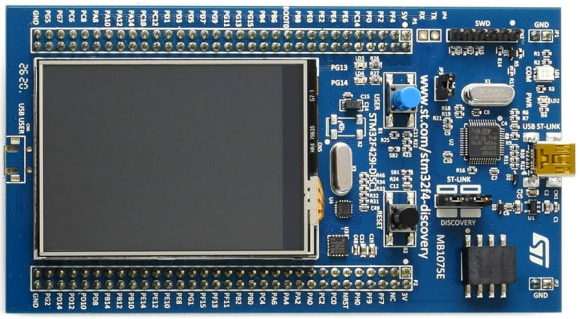
Figure 5.4: Final Prototype of the Platform's Wheels.

Computation & Communication

The system uses a two-layer architecture: an **STM32F429 Discovery board** handles low-level tasks (motor control, encoder acquisition), while a **Raspberry Pi 5** (8 GB) runs the ROS2 stack (SLAM, navigation, exploration). The two layers communicate via **UART** at a baudrate of **921600**.



(a) Raspberry Pi 5 Model B



(b) STM32F429 Discovery

Figure 5.5: Computation Components.

Sensing

The sensing system covers two complementary needs: environment perception for SLAM and orientation estimation for state fusion.

Environment perception is provided by the **RPLIDAR A2M12** provides 360° planar scanning, which is essential for the SLAM Toolbox to build consistent occupancy maps. Its specifications are listed in Table 5.2.

Table 5.2: RPLIDAR A2M12 Specifications.

Parameter	**Value**
Measuring Range	0.2 – 12 m
Sampling Frequency	16,000 points/s
Rotational Speed	10 Hz (5 – 15 Hz configurable)
Angular Resolution	0.225°



Figure 5.6: RPLIDAR A2M12.

For state estimation, an **MPU6050** inertial measurement unit (IMU) is used. As it combines a 3-axis gyroscope and a 3-axis accelerometer in a single module. It provides orientation data that complements wheel odometry. Its specifications are listed in Table 5.3.

Table 5.3: MPU-6050 Specifications.

Parameter	**Value**
Communication	I^2C
ADC Resolution	16-bit
Gyroscope Range	±250 / 500 / 1000 / 2000 °/s
Accelerometer Range	±2 / ±4 / ±8 / ±16 g

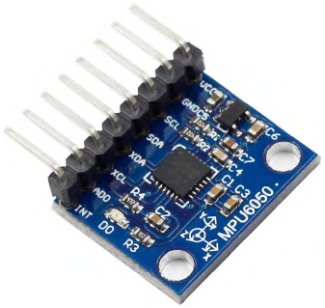


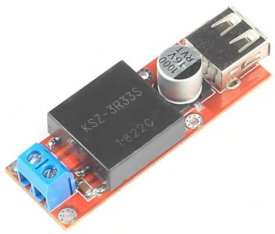
Figure 5.7: MPU 6050 IMU Module.

Power

The robot is powered by a **12 V - 4200 mAh LiPo battery**. A **KIS-3R33** voltage regulator distributes stable power across all components. An emergency stop button is mounted on the top plate for immediate shutdown in case of malfunction.



(a) LiPo Battery



(b) KIS-3R33 Voltage Regulator Module

Figure 5.8: Power Components.

Hardware Summary

Table 5.4 summarizes all hardware components along with their key specifications and individual costs.

System Block Diagram

The STM32 forms the low-level control layer, interfacing with the motors through the L298N driver and processing encoder feedback. The Raspberry Pi communicates with the STM32 via UART and acquires data from the LiDAR and IMU for ROS2. Power is distributed from the LiPo battery through the voltage regulator to the different subsystems. Figure 5.9 presents the overall hardware architecture of the robot.

Table 5.4: Hardware components summary.

Component	Model	Cost (DZD)
Microcontroller	STM32F429 Discovery	8,000.00
Single-board PC	Raspberry Pi 5 Model B	45,000.00
LiDAR	RPLIDAR A2M12	82,000.00
IMU	MPU6050	800.00
Motor driver	L298N	1,000.00
DC Motors (×2)	GB37Y3530-12V-251R	3,850.00
Battery	12V 4200mAh LiPo	9,900.00
Voltage regulator	KIS-3R33	1,400.00
Chassis	Custom frame	XX
Total		**151,950.00**

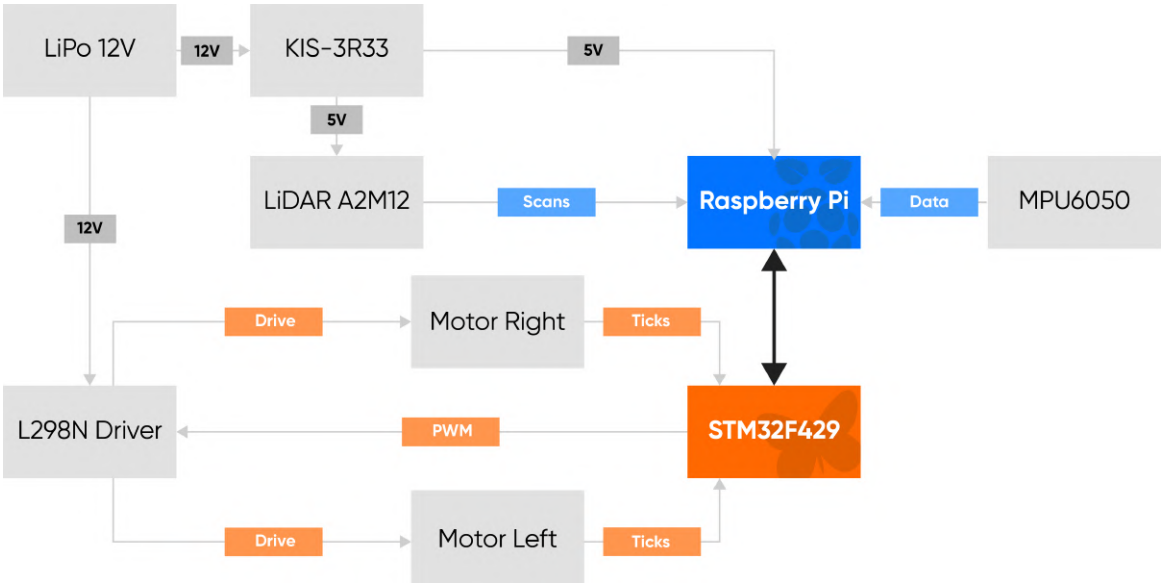
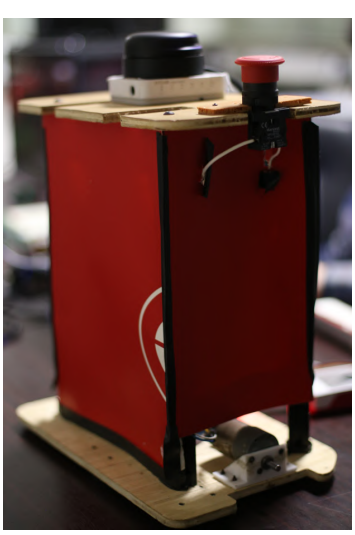


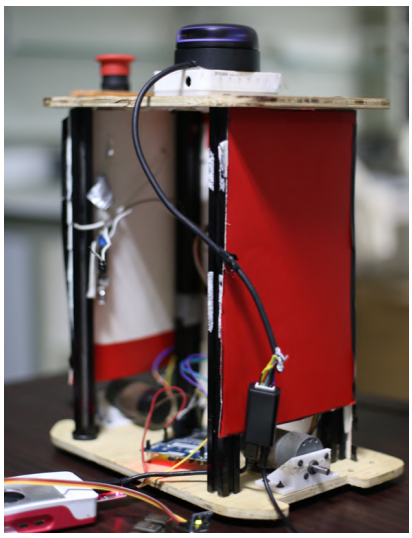
Figure 5.9: Diagram of the custom robot hardware.

Prototype

The robot went through two physical prototypes before reaching its final form. The first prototype, shown in Figure 5.10, served as a functional testbed to validate the hardware integration and the communication between the STM32 and the Raspberry Pi. At this stage, the structure was partially open, allowing easy access to the wiring and components during debugging. The motors were already mounted in their final positions, while the electronics remained exposed for rapid iteration.



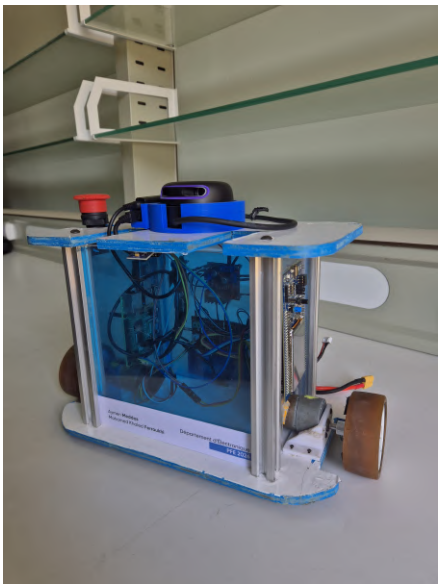
(a) Front View



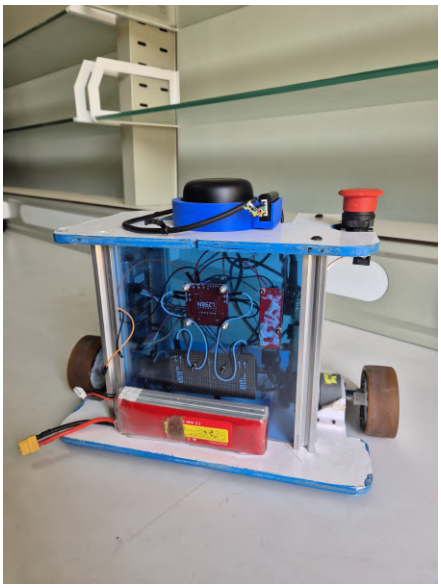
(b) Back View

Figure 5.10: First Prototype of the Platform.

The final prototype consolidates all components into a clean and self-contained platform similar to the original mechanical design, with proper cable management and a closed chassis. It is shown in Figure 5.11.



(a) Front View



(b) Back View

Figure 5.11: Final Prototype of the Platform.

Overall the robot was with the dimensions of **32 × 20 × 32 cm** and total weight is ≈ **8 kg**.

5.2.2 Low-Level Layer

The low-level layer is executed on the STM32 and is responsible for direct interaction with the motors, including real-time control and feedback processing.

Control Loop

The control loop is responsible for executing velocity commands received from the Raspberry Pi and ensuring that each wheel tracks its target speed accurately. The overall control loop is illustrated in Figure 5.12.

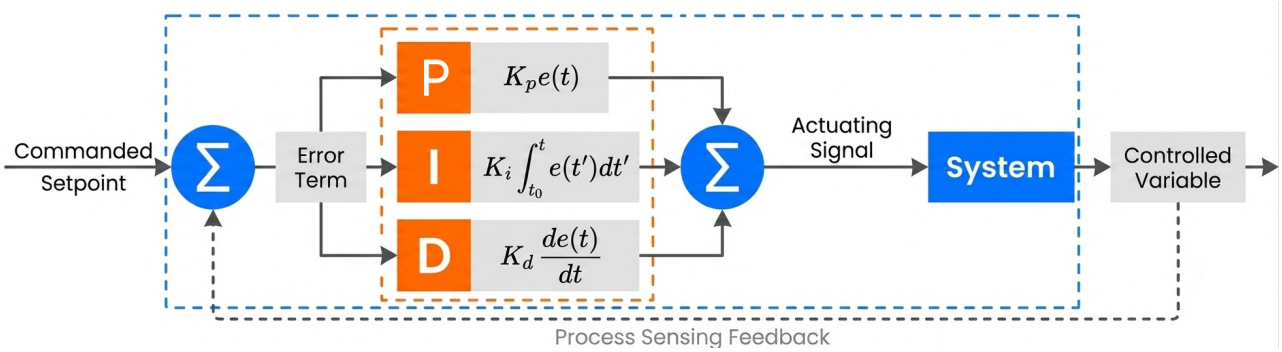


Figure 5.12: Low-level control loop architecture.

The robot follows a differential drive model, uses two independently driven wheels on a common axis, allowing motion by varying the relative rotational velocities of each wheel. The differential drive model is illustrated in the Figure 5.13

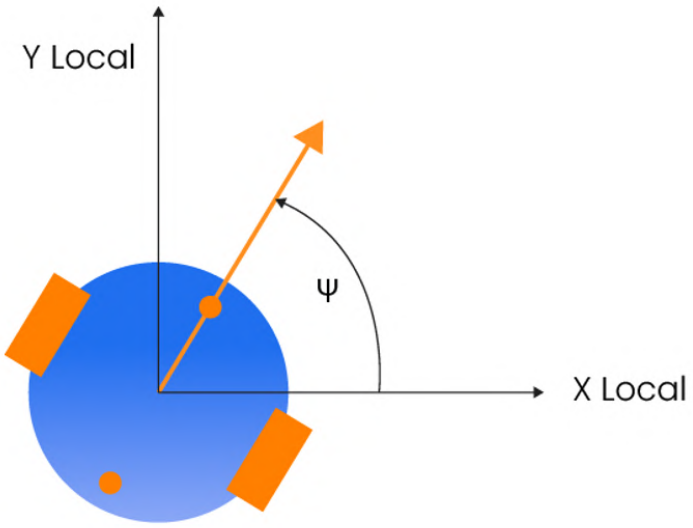


Figure 5.13: Two-wheeled differential drive model.

Given the wheel radius $\mathbf{r = 0.0365\,m}$ and the wheelbase (the distance between the two wheels) $\mathbf{L = 0.345\,m}$, the forward kinematics map wheel velocities to robot body velocities as follows:

$$v = \frac{v_r + v_l}{2} \tag{5.1}$$

$$\omega = \frac{v_r - v_l}{L} \tag{5.2}$$

where v is the linear velocity of the robot, ω is its angular velocity and v_l, v_r are the left and right wheel linear velocities respectively.

The inverse kinematics, used to convert a desired (v, ω) command into individual wheel velocity targets, are given by:

$$v_r = v + \frac{\omega \cdot L}{2} \tag{5.3}$$

$$v_l = v - \frac{\omega \cdot L}{2} \tag{5.4}$$

The Raspberry Pi publishes velocity commands as (v, ω) pairs over UART. Upon reception, the STM32 applies Equations 5.3 and 5.4 to compute the target linear speed for each wheel, converts them to target RPM and feeds them into the PID controllers. The measured RPM from the encoders closes the loop and the resulting PWM signals drive the motors via the L298N driver.

Sensor Block

The GB37Y3530 motors are equipped with Hall effect encoders. A Hall effect encoder consists of a rotating magnetic disk and two Hall effect sensors (channels A and B) placed 90° apart, as illustrated in Figure 5.14. As the disk rotates, each sensor outputs a digital pulse when it detects a magnetic pole transition. The quadrature arrangement of the two channels allows both speed and direction to be determined: the frequency of the pulses encodes speed, while the phase relationship between channels A and B encodes the direction of rotation.

The STM32 reads encoder pulses using its hardware timer in encoder mode, which counts both rising and falling edges of channels A and B, giving 4× the base resolution. The wheel RPM is then computed from the tick count Δn accumulated over a sampling period Δt (in seconds):

$$\text{RPM} = \frac{\Delta n}{N_{ticks}} \times \frac{60}{\Delta t} \tag{5.5}$$

To validate the theoretical tick count, a physical calibration was performed by marking a reference point on the wheel and manually rotating it through exactly one full revolution while counting the encoder pulses recorded by the STM32. The measured value was found to be consistent with the theoretical value of 1900 ticks/revolution, confirming the datasheet parameters.

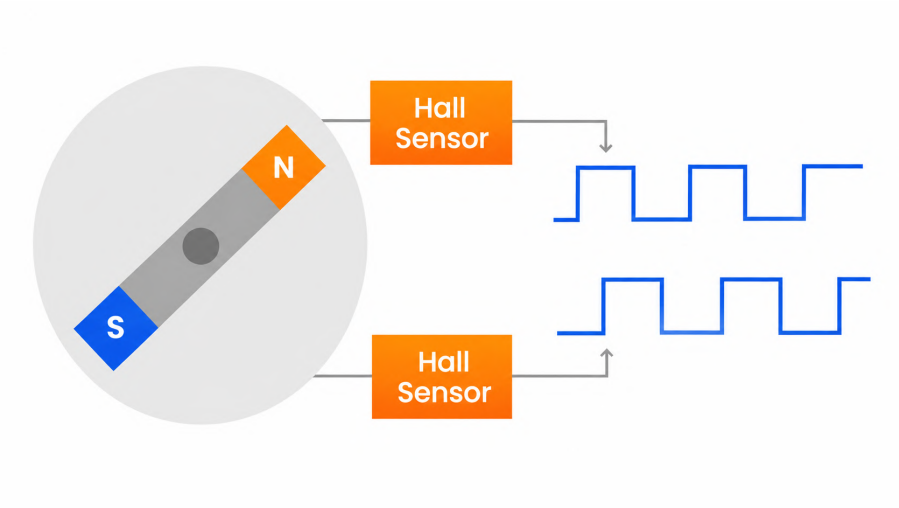


Figure 5.14: Operating principle of a Hall effect encoder.

PID Block

A PID controller is implemented independently for each wheel to track the target RPM computed from the inverse kinematics. The continuous-time PID control law is:

$$u(t) = K_p\, e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{de(t)}{dt} \tag{5.6}$$

where $e(t) = \text{RPM}_{target} - \text{RPM}_{measured}$ is the velocity error and $u(t)$ is the control output mapped to a PWM duty cycle.

Since the controller runs on the STM32 at a fixed sampling period Δt, the discrete velocity form of the PID is used:

$$u[k] = u[k-1] + K_p\, \Delta e[k] + K_i\, e[k]\, \Delta t + K_d \frac{\Delta e[k]}{\Delta t} \tag{5.7}$$

where $\Delta e[k] = e[k] - e[k-1]$.

Before tuning, the PID gains were empirically set to $K_p = 1$, $K_i = 0.5$, and $K_d = 0.5$, resulting in a fast rise time but significant overshoot and oscillations around the reference.

The gains were determined experimentally through iterative testing. The tuning process started with $K_i = K_d = 0$ and an increasing K_p until the system responded adequately, followed by the introduction of K_i to eliminate steady-state error. The final values are listed in Table 5.5.

Table 5.5: PID controller gains for the left and right motors.

(a) Left motor

Parameter	K_p	K_i	K_d
Value	1.35	0.064	0.8

(b) Right motor

Parameter	K_p	K_i	K_d
Value	1.5	0.08	0.8

After tuning, the response became slightly slower but achieved accurate setpoint tracking with negligible overshoot and no oscillations, providing a better compromise between responsiveness and stability. as illustrated in Figure 5.15.

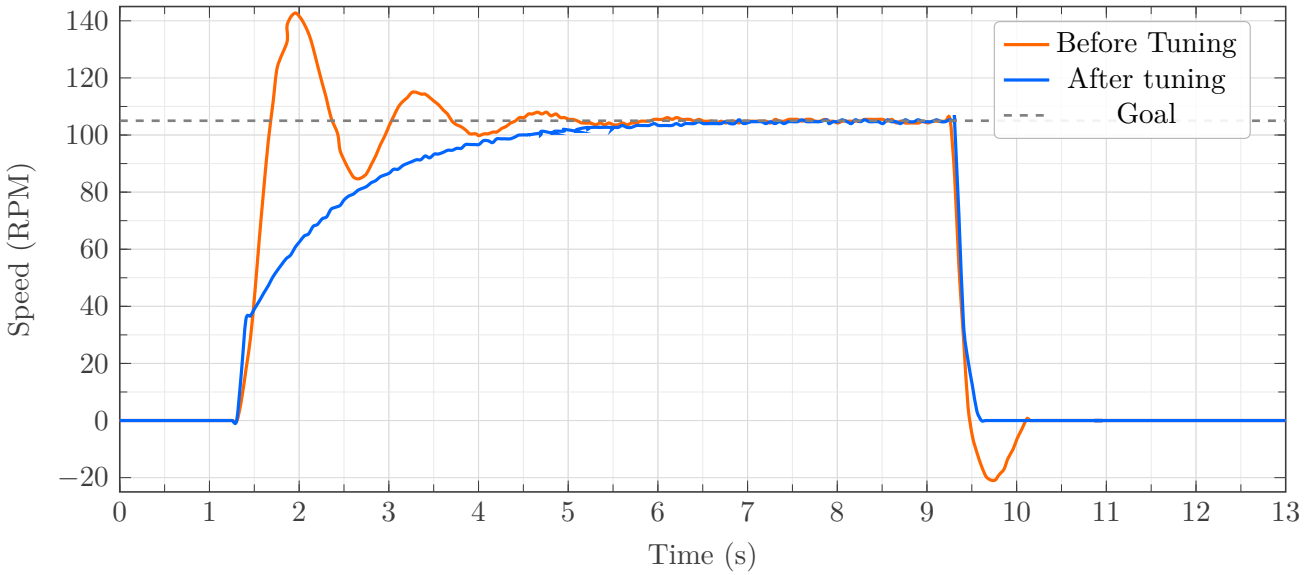
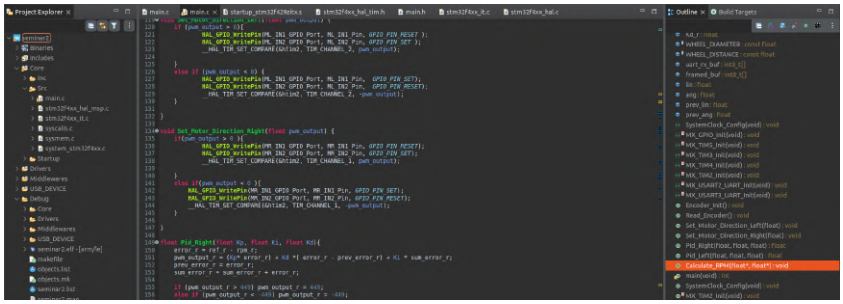


Figure 5.15: Comparison of PID controller step responses before and after tuning.

HAL Configuration & Pin Mapping

The firmware was developed using **STM32CubeIDE**, an all-in-one development environment provided by STMicroelectronics. It integrates a graphical pin configuration tool, code generation and a GCC-based compiler and debugger in a single interface, as shown in Figure 5.16a.



(a) CubeIDE Interface.

(b) CubeIDE Logo.

Figure 5.16: STM32 CubeIDE.

The development workflow consists of three main steps:

- First, the target MCU is selected and its pins are assigned graphically using the **Pinout & Configuration** view, where each peripheral is enabled and mapped to physical pins.
- Second, CubeIDE automatically generates the full HAL initialization code based on the configured settings.
- Finally, the application logic is written on top of the generated code.

Figure 5.17 shows the pin configuration view for the STM32F429ZITx used in this project.

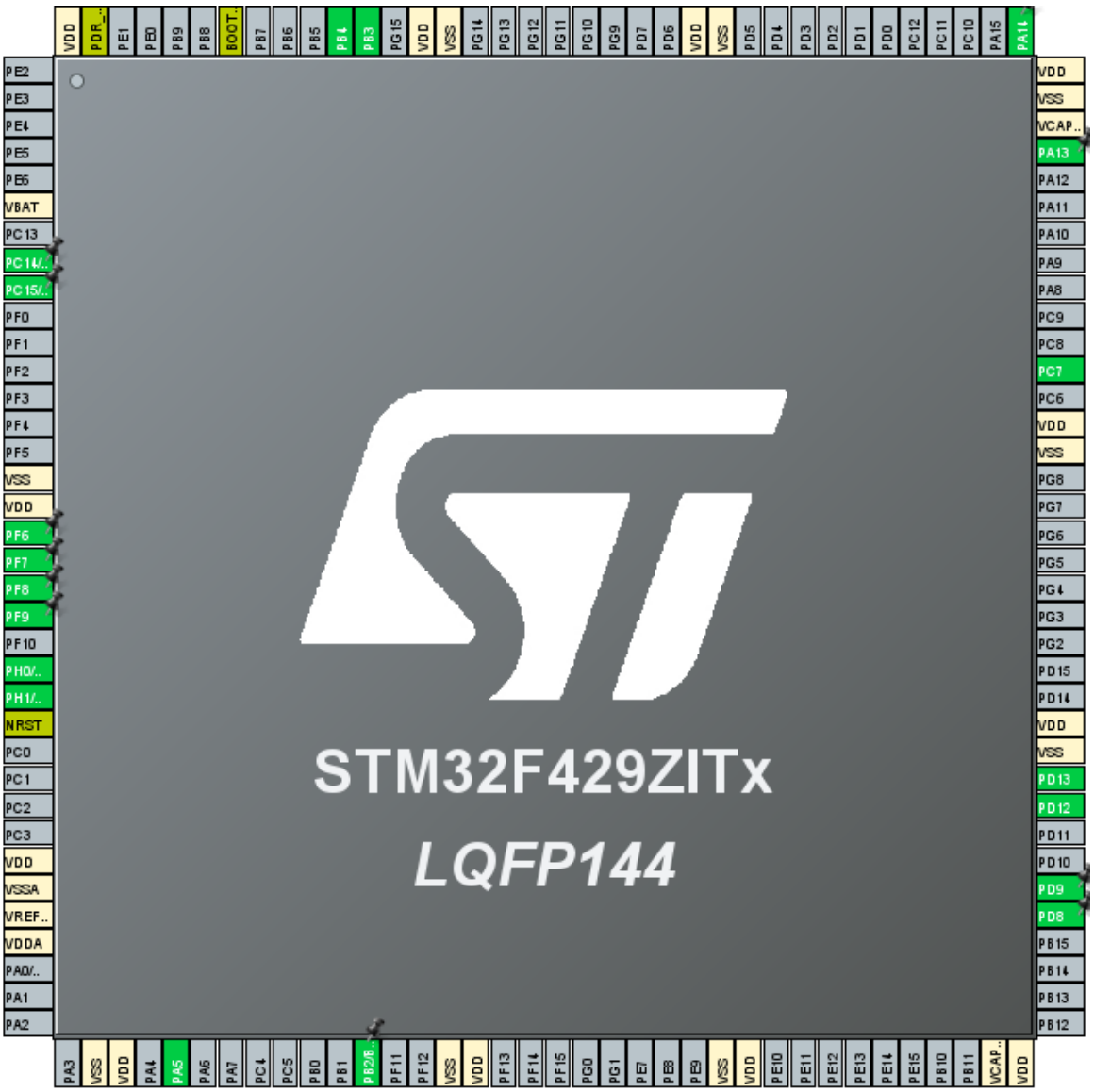


Figure 5.17: STM32CubeIDE pin configuration view for the STM32F429ZITx.

The system clock is configured using the external clock source at the maximum frequency of **180 MHz**. The peripheral assignments are as follows:

- **PWM outputs:** TIM2 is configured in PWM mode on channels CH1 and CH2 to drive the left and right motors respectively via the L298N driver.
- **Motor direction pins**: GPIO pins configured to control each motor's direction based on the sign of the PWM value.
- **Encoder inputs:** TIM4 and TIM3 are configured in encoder interface mode to read quadrature signals from the left and right wheel encoders respectively.
- **UART:** USART3 is configured at a baud rate of **921600** for communication with the Raspberry Pi.

The complete pin mapping is provided in Table 5.6.

Table 5.6: STM32F429 Pin Mapping.

Signal	Pin	Peripheral
PWM Left	PB3	TIM2 CH1
PWM Right	PA5	TIM2 CH2
MR Forward	PF6	GPIO
MR Backward	PF7	GPIO
ML Forward	PF8	GPIO
ML Backward	PF9	GPIO
Encoder Left A	PD12	TIM4 CH1
Encoder Left B	PD13	TIM4 CH2
Encoder Right A	PB4	TIM3 CH1
Encoder Right B	PC7	TIM3 CH2
UART TX	PD8	USART3
UART RX	PD9	USART3

The STM32 continuously streams wheel speeds (RPM) to the Raspberry Pi for odometry computation, while the Raspberry Pi sends back velocity commands (m/s) that the STM32 converts into motor PWM signals.

5.2.3 High-Level Layer

As mentioned previously, the high-level layer is handled using the Raspberry Pi 5 with 8GB of RAM. This layer is responsible for odometry computation and reading IMU data to implement sensor fusion between wheel odometry and IMU measurements. It also runs the SLAM and navigation systems. In short, it is responsible for the heavy computation tasks of the robot. Figure 5.18 illustrates the overall architecture of the high-level layer with Docker containers holding Ubuntu and ROS2 Humble and all the main functions and nodes running on the robot.

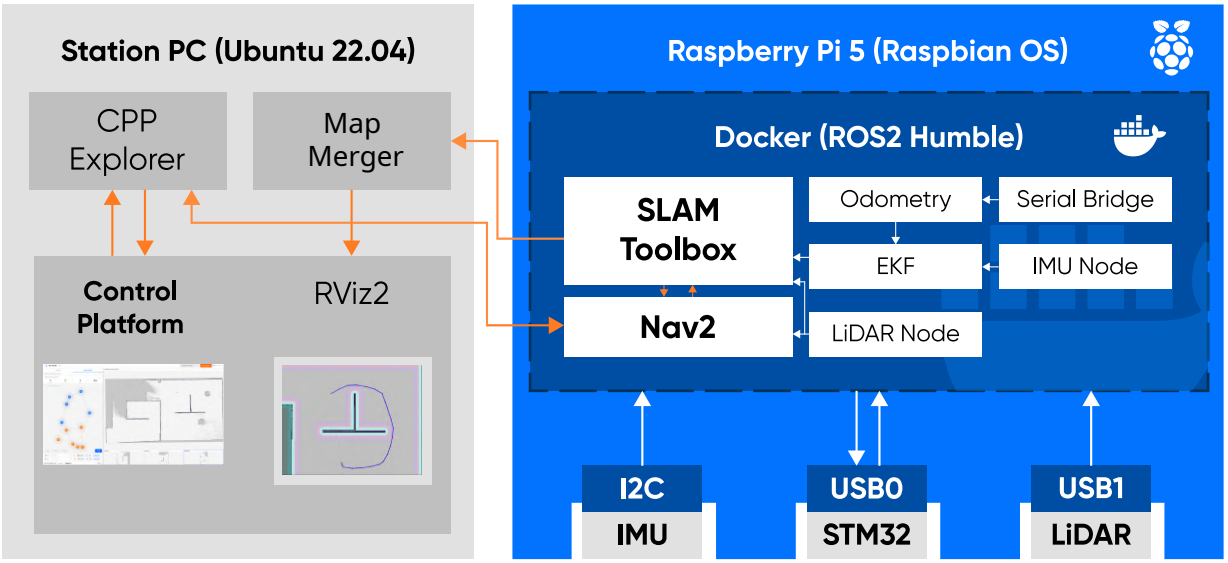


Figure 5.18: High-level architecture for the custom robot.

Data Distribution Service (DDS)

ROS 2 relies on the **Data Distribution Service (DDS)** middleware for communication between nodes. DDS provides a decentralized publish-subscribe architecture that allows ROS 2 topics, services and actions to be exchanged transparently across multiple devices connected to the same network.

To reduce the computational load on the Raspberry Pi, the navigation stack was executed on a personal computer while the remaining ROS 2 nodes continued running on the robot. Communication between both devices was achieved through DDS over the local network. The Raspberry Pi and the PC were configured with the same `ROS_DOMAIN_ID`, allowing them to automatically discover each other and exchange ROS 2 messages. As a result, sensor data such as odometry, LiDAR scans and TF transforms published by the Raspberry Pi became directly accessible to the navigation nodes running on the PC, while velocity commands generated on the PC were transmitted back to the robot through the `/cmd_vel` topic.

Docker & Docker Compose

Docker is an open-source platform that uses OS-level virtualization to build, ship and run applications including the Robot Operating System (ROS) in lightweight, isolated packages called containers [83].

Instead of deploying everything directly on the Raspberry Pi operating system, a Docker container with **Ubuntu 22.04** and **ROS2 Humble** was used. This makes dependency management easier and avoids conflicts between system libraries and ROS2 packages. It also makes the project easier to **reproduce on another machine** since the same environment can be rebuilt using the same Docker configuration.

The architecture used is based on **Docker Compose**, which was used to configure the container settings such as device access, mounted volumes, network configuration and the scripts executed when the container starts. The `docker-compose.yml` file defines the main container configuration and **allows privileges** to hardware interfaces such as GPIOs, serial ports and USB connections required by the robot. Figure 5.19 shows the overall docker compose archetecuture.

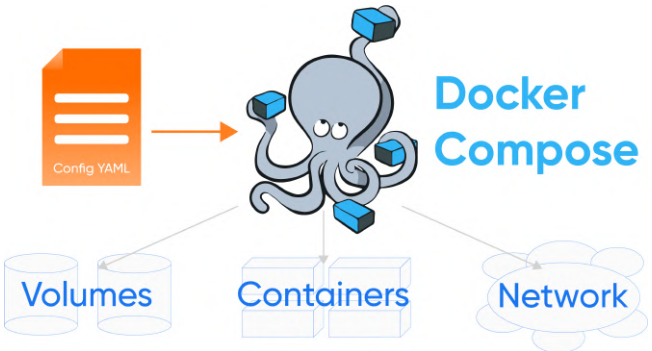


Figure 5.19: Docker Compose Architecture.

A `Dockerfile` was also used to build the container image and install all necessary packages and dependencies including ROS2 packages, navigation tools, SLAM packages and communication utilities. A shell script is then executed when the container starts automatically sourcing the ROS workspace and launches the required ROS nodes. This helps **simplify the startup**

process and avoids running multiple commands manually each time. Finally, the `src` folder contains all the ROS2 packages developed and used in the project.

UART Communication

The communication between the Raspberry Pi and the STM32 is done using UART serial communication through the UART pins of the Raspberry Pi connected directly to the UART pins of the STM32. This communication is used to **exchange the required data** between the high-level and low-level layers of the robot.

As illustrated in figure 5.20 velocity commands can come from two different sources depending on the operating mode of the robot:

- **Manual mode**: The command is generated using teleoperation.
- **Autonomous Mode**: The command is generated by the navigation stack during path planning and obstacle avoidance.

A ROS2 node is responsible for handling this serial communication. It subscribes to the `/cmd_vel` topic to receive the linear and angular velocity commands, then converts and sends these values to the STM32 through UART. At the same time, the STM32 sends back the measured wheel RPM values based on encoder feedback. The same node receives these values and publishes them in a dedicated topic used by the odometry node to compute the robot position and orientation.

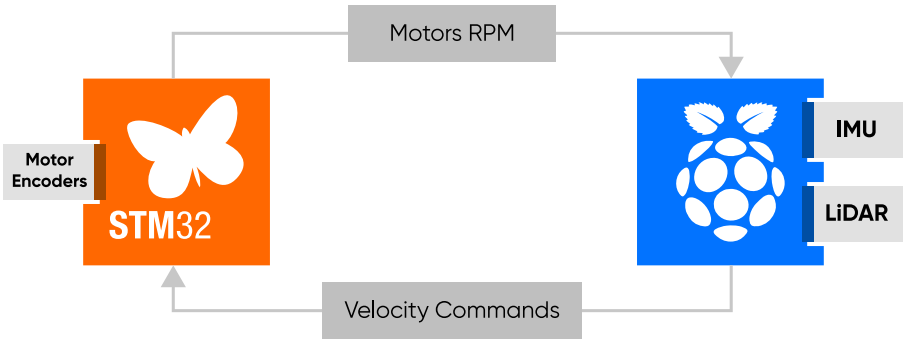


Figure 5.20: UART Communication between High-Level and Low-Level Layers.

Odometry

Odometry is responsible for **localizing the robot** by estimating how much it has moved and calculating its current position and orientation. This is done using the wheel encoder feedback. For accurate odometry computation, two important robot parameters must be known: the **wheel radius** and the **wheelbase**.

For the differential drive robot, an odometry node was established. It subscribes to the `rpm_l` and `rpm_r` topics published by the serial communication node. These values represent the measured rotational speed of the left and right wheels in revolutions per minute (RPM). Next, the RPM values is converted into linear wheel velocities in meters per second (m/s) using Equation (5.8).

$$v = \text{RPM} \times \frac{2\pi r}{60} \tag{5.8}$$

Then, equations (5.1) and (5.2) of the forward kinematic model are used to compute the robot's linear velocity v_x and angular velocity ω. Next, the robot pose is updated by estimating the small displacement during each sampling period dt. The variations in position Δx and Δy and orientation $\Delta\theta$ are calculated as follows:

$$\Delta x = v\cos(\theta) \cdot dt \tag{5.9}$$

$$\Delta y = v\sin(\theta) \cdot dt \tag{5.10}$$

$$\Delta\theta = \omega \cdot dt \tag{5.11}$$

Then, the global robot pose is updated by adding the variation to the current values:

$$x_i = x_{i-1} + \Delta x = x_{i-1} + v\cos(\theta) \cdot dt \tag{5.12}$$

$$y_i = y_{i-1} + \Delta y = y_{i-1} + v\sin(\theta) \cdot dt \tag{5.13}$$

$$\theta_i = \theta_{i-1} + \Delta\theta = \theta_{i-1} + \omega \cdot dt \tag{5.14}$$

The odometry node is also responsible for publishing the corresponding transforms between the `odom` frame and the `base_footprint` frame. These frames are required for robot localization and navigation, since they represent the robot position relative to its local reference frame. The complete TF tree and frame structure will be explained in section 5.2.3.

This process is repeated continuously to estimate the robot position in real time and provide the necessary localization information for navigation and SLAM.

Extended Kalman Filter

The Extended Kalman Filter (EKF) is essential in modern control systems, particularly in robotics and unmanned vehicles, due to its ability to provide accurate state estimates from noisy sensor measurements. This accuracy is crucial for stability and performance, enabling precise control and navigation.

Wheel odometry is useful for estimating the robot position, but it suffers from **accumulated drift over time**, especially in rotation estimation. Small errors in encoder measurements can produce large errors in orientation after long movements. On the other hand, the Inertial Measurement Unit (IMU) provides **better orientation estimation**, especially for angular motion, but when used alone it also suffers from drift and noise.

To reduce the weaknesses of both sensors and improve the precision of robot localization, sensor fusion is applied between wheel odometry and IMU data using the Extended Kalman

Filter (EKF). The goal is to combine the advantages of both sources: the wheel odometry provides good short-term position estimation, while the IMU improves orientation estimation and rotational stability.

The standard discrete-time Kalman filter is the optimal linear estimator for systems that can be described by the following linear model [84]:

$$x_k = F_{k-1}\, x_{k-1} + G_{k-1}\, u_{k-1} + w_{k-1} \tag{5.15}$$

$$z_k = H_k\, x_k + v_k \tag{5.16}$$

where F_{k-1} is the state transition matrix, G_{k-1} is the input matrix, H_k is the observation matrix and w_k, v_k are zero-mean white noise processes with covariance matrices Q_k and R_k, respectively. The filter alternates between a **prediction step**, which propagates the state estimate forward in time using the system model and a **correction step**, which incorporates new sensor measurements to refine the estimate. The Kalman filter is optimal in the sense that it minimises the mean squared estimation error when the noise is Gaussian.

However, a mobile robot's motion model is **nonlinear**. The kinematic equations involve trigonometric functions of the heading angle θ, making direct application of the standard Kalman filter impossible. And when the system does not behave linearly even over a small range of operation, linear estimators no longer give satisfactory results.

The Extended Kalman Filter addresses this by **linearising the nonlinear functions locally** at each time step using a first-order Taylor series expansion. Instead of constant matrices F and H, the EKF computes their **Jacobian** counterparts evaluated at the current state estimate.

The robot dynamics are governed by the following nonlinear system and measurement models [85]:

$$x_k = f(x_{k-1},\, u_k) + w_{k-1} \tag{5.17}$$

$$z_k = h(x_k) + v_k \tag{5.18}$$

where f is the nonlinear motion model, h is the nonlinear measurement model, u_k is the control input, $w_k \sim \mathcal{N}(0, Q_k)$ is the process noise and $v_k \sim \mathcal{N}(0, R_k)$ is the measurement noise.

The state vector contains the robot's pose and velocity:

$$\mathbf{x} = \begin{bmatrix} x & y & \theta & v_x & v_y & \omega \end{bmatrix}^T \tag{5.19}$$

where x and y represent the robot position, θ is the heading angle, v_x and v_y are the linear velocities (provided by wheel odometry) and ω is the angular velocity (provided by the IMU). The measurement vectors selected from each sensor are:

$$z_{\text{odom}} = \begin{bmatrix} v_x & v_y \end{bmatrix}^T, \qquad z_{\text{IMU}} = \begin{bmatrix} \omega \end{bmatrix} \tag{5.20}$$

At each time step the filter first **predicts** the state and its uncertainty based solely on the motion model. The Jacobian of f with respect to the state, evaluated at the previous a posteriori estimate $\hat{x}_{k-1}^{+}$, is:

$$F_{k-1} = \left.\frac{\partial f}{\partial x}\right|_{\hat{x}_{k-1}^{+}} \tag{5.21}$$

The predicted state estimate and its covariance are then computed as follows:

$$\hat{x}_k^{-} = f\left(\hat{x}_{k-1}^{+},\, u_k\right) \tag{5.22}$$

$$P_k^{-} = F_{k-1}\, P_{k-1}^{+}\, F_{k-1}^{T} + Q_{k-1} \tag{5.23}$$

where P_k^{-} is the predicted estimation-error covariance matrix and Q_{k-1} is the process noise covariance. The superscripts $-$ and $+$ denote a priori (before measurement) and a posteriori (after measurement) quantities, respectively.

When a new sensor measurement z_k becomes available, the filter **corrects** its prediction. The Jacobian of h with respect to the state, evaluated at the a priori estimate $\hat{x}_k^{-}$, is:

$$H_k = \left.\frac{\partial h}{\partial x}\right|_{\hat{x}_k^{-}} \tag{5.24}$$

The **innovation** is the difference between the actual measurement and the predicted measurement:

$$\tilde{y}_k = z_k - h\left(\hat{x}_k^{-}\right) \tag{5.25}$$

The **Kalman gain** K_k determines how much weight is given to the innovation relative to the prediction. A large gain trusts the sensors more; a small gain trusts the model more:

$$K_k = P_k^{-}\, H_k^{T}\left(H_k\, P_k^{-}\, H_k^{T} + R_k\right)^{-1} \tag{5.26}$$

Finally, the a posteriori state estimate and covariance are updated:

$$\hat{x}_k^{+} = \hat{x}_k^{-} + K_k\, \tilde{y}_k \tag{5.27}$$

$$P_k^{+} = (I - K_k\, H_k)\, P_k^{-} \tag{5.28}$$

An important consequence of the nonlinear nature of f and h is that the Jacobian matrices F_k and H_k depend on the current state estimate, which in turn depends on the noisy measurements. Therefore, unlike the linear Kalman filter, the EKF gain and covariance **cannot be computed offline** before the system operates; they must be recomputed at every time step.

In ROS2, this process is implemented using the `robot_localization` package [86], which provides a ready-to-use EKF node configurable through a YAML parameter file. The node runs at a frequency of 5 Hz and if a sensor times out, the filter continues to operate in prediction-only mode with a timeout threshold of 0.2 s.

Two sensor sources are then configured as follows:

- **Wheel odometry**: The topic `odom` is fused using the `odom0_config` vector, which selects only the x and y positions and the yaw angle from the full 15-element state vector.
- **IMU**: The topic `imu_filtered` is configured via `imu0_config` to contribute only the angular velocity $\dot{\psi}$ around the z-axis, which is the dominant rotational degree of freedom for a differential-drive robot operating in a planar environment.

Outlier rejection is applied to both sources using Mahalanobis distance thresholds: pose measurements from odometry are rejected beyond a distance of 5.0 and twist measurements beyond 1.0, preventing corrupted sensor readings from destabilising the filter.

The process noise covariance matrix Q is specified as a full 15×15 diagonal matrix, with representative values of 0.05 for position states (x, y), 0.06 for the yaw angle ψ, 0.025 for linear velocities and smaller values of 0.01–0.02 for angular rates and accelerations. These values reflect the relative confidence in the motion model: larger diagonal entries cause the filter to weight incoming measurements more heavily during correction, which is useful for variables that are difficult to model accurately. The initial estimation-error covariance is set to 10^{-9} for all states, indicating high confidence in the initial pose, which is assumed to be known at startup.

This process runs continuously at each sensor update cycle, allowing the robot to maintain an accurate and consistent estimate of its position and orientation — significantly more reliable than using wheel odometry or IMU data in isolation.

Transforms & Namespacing Setup

In ROS2, transforms (TF) are used to represent the relation between the different components of the robot and how they move relative to each other and to the fixed reference frame. These transforms are necessary for localization, navigation and sensor integration, since each sensor and robot component must be placed correctly in the global frame.

For mobile robots, both `slam_toolbox` and Nav2 follow the REP 105 standard (*Coordinate Frames for Mobile Platforms*), which defines the main reference frames used in robot navigation. Since this project is based on a multi-agent architecture, each robot must maintain its own set of reference frames while coexisting within the same network. To avoid frame name collisions, all robot-related frames are prefixed with a unique namespace (`robot1/`, `robot2/`). In our implementation, a fixed `world` frame is used as the root of the TF tree and serves as a common reference for all robots. Under this frame, the `map` frame represents the global reference used for long-term localization. The `odom` frame provides a locally continuous reference generated from odometry and state estimation, while the `base_link` frame defines the physical body of the robot, and all sensor frames, such as `laser_frame` and `imu_link`, are attached relative to it.

Since an Extended Kalman Filter (EKF) is used for sensor fusion, it is responsible for publishing the transform between `robot1/odom` and `robot1/base_link`. The odometry node does not publish this transform, as only a single source should provide a given TF relation; otherwise, conflicting transforms may cause instability and TF jitter.

The SLAM node publishes the transform between `robot1/map` and `robot1/odom`, completing the TF chain required by the navigation stack. Although the frames themselves are namespaced, all transforms are published on the standard ROS 2 `/tf` topic rather than on a namespaced topic such as `/robot1/tf`. Figure 5.21 shows the resulting TF tree structure used in the system.

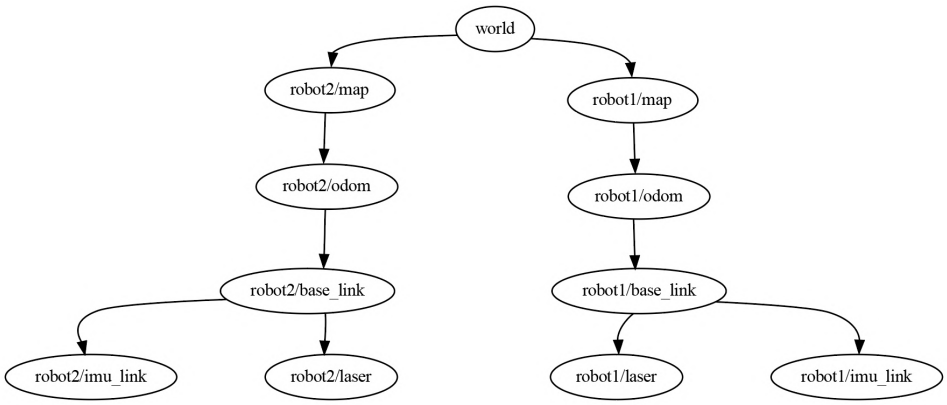


Figure 5.21: TF tree structure following REP 105 for mobile robot navigation

Since the system is composed of multiple robots, namespaces are used to prevent conflicts between topics, nodes and TF frames. Each robot is assigned a unique prefix (e.g., `/robot1`, `/robot2`), ensuring that topics such as `/cmd_vel`, `/odom` and `/scan` remain independent across the network.

LiDAR Integration and Laser Scan

The LiDAR also plays an important role in map generation. Since it provides accurate distance measurements, it is used by the SLAM system to build a precise map of the environment while the robot moves. In addition, the navigation stack uses the same scan data for obstacle detection and local path correction during autonomous movement.

The integration with ROS2 is simplified thanks to the SDK provided by Slamtec, which allows direct communication with the sensor and easy publishing of scan data.
The LiDAR node publishes the laser scan measurements in the `/scan` topic using the standard `sensor_msgs/LaserScan` message type. This `/scan` topic is then used by the `slam_toolbox` node to generate the map of the environment and by the Nav2 stack to detect obstacles and avoid collisions during robot navigation.

Figure 5.22 shows the laser scan visualization in RViz. The scan points are color-coded according to their distance from the sensor: red points correspond to nearby obstacles, while pink points represent obstacles located farther away.

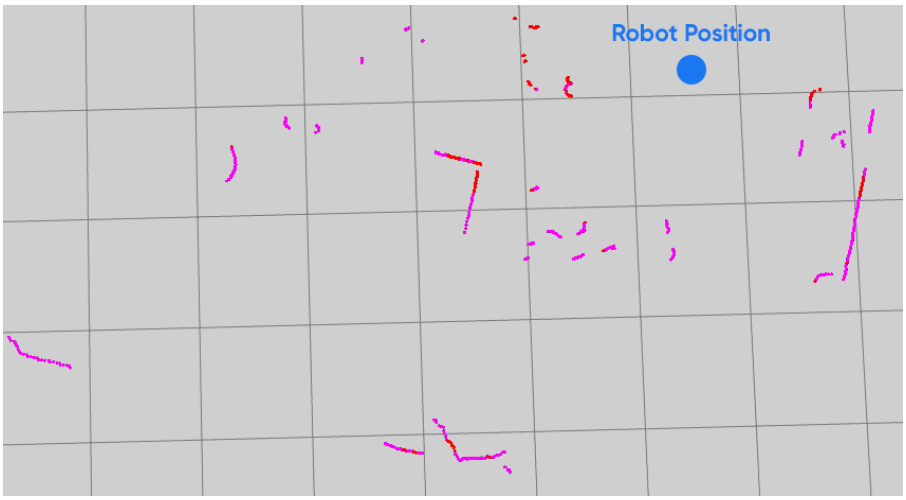


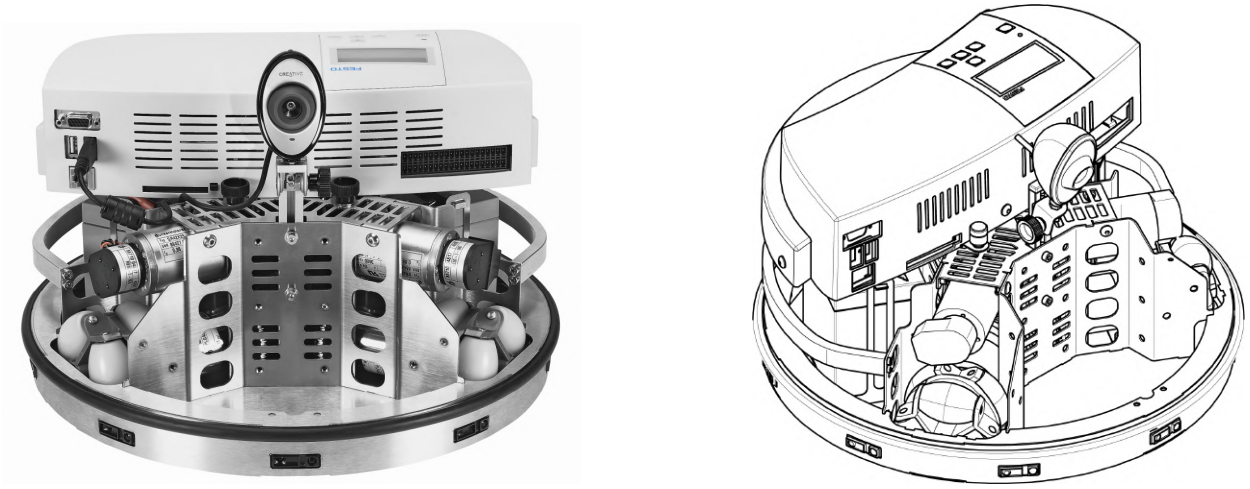
Figure 5.22: LiDAR laser scan visualization in RViz

5.3 Second Robot: Festo Robotino

The Festo Robotino is a commercial omnidirectional mobile robot platform developed by **Festo Didactic**, designed primarily for research and education in autonomous robotics [87].

5.3.1 Robot Description

Robotino evolved through four generations. Robotino 1 was released in 2005, followed by Robotino 2 in 2008 with additional USB webcam support and safety bumpers. Robotino 3, launched in 2013, introduced modular accessory slots, integrated PC systems and improved REST/ROS support. Finally, Robotino 4, released in 2019, upgraded the hardware and operating system for more demanding computations and added a high-performance lithium-ion battery system supporting up to four battery packs [87]. **Robotino v2.1 was used** in the scope of this project. The platform is shown in Figure 5.23 with front view and isometric diagram.



(a) Front View. (b) Isometric Diagram.

Figure 5.23: Festo Robotino Platform.

Hardware Specifications

The Robotino v2.1 architecture consists of an embedded PC based on a PC/104 system featuring an **AMD Geode LX800** processor operating at 500 MHz with 128 MB SDRAM memory and a Compact Flash (CF) card reader for storage. The robot runs **Debian GNU/Linux 5.0 Lenny** and provides several communication interfaces including Ethernet, USB, VGA and parallel ports. Networking capabilities are ensured through either Ethernet port or using the integrated **Air Live WL-5480PAP** for WLAN connectivity.

The low-level control system is built around a 32-bit **NXP LPC2378** microcontroller and a **Xilinx XC3S50A** FPGA connected through an 8-bit data bus. The I/O board provides access to internal sensors as well as 8 user analog inputs, 8 user digital inputs, 8 digital outputs and 2 relay outputs. Additional communication interfaces include CAN bus, Ethernet, RS232 and JTAG. The motion system supports three drive motors and one accessory motor output, while encoder feedback is processed through the FPGA for motor control and localization tasks.

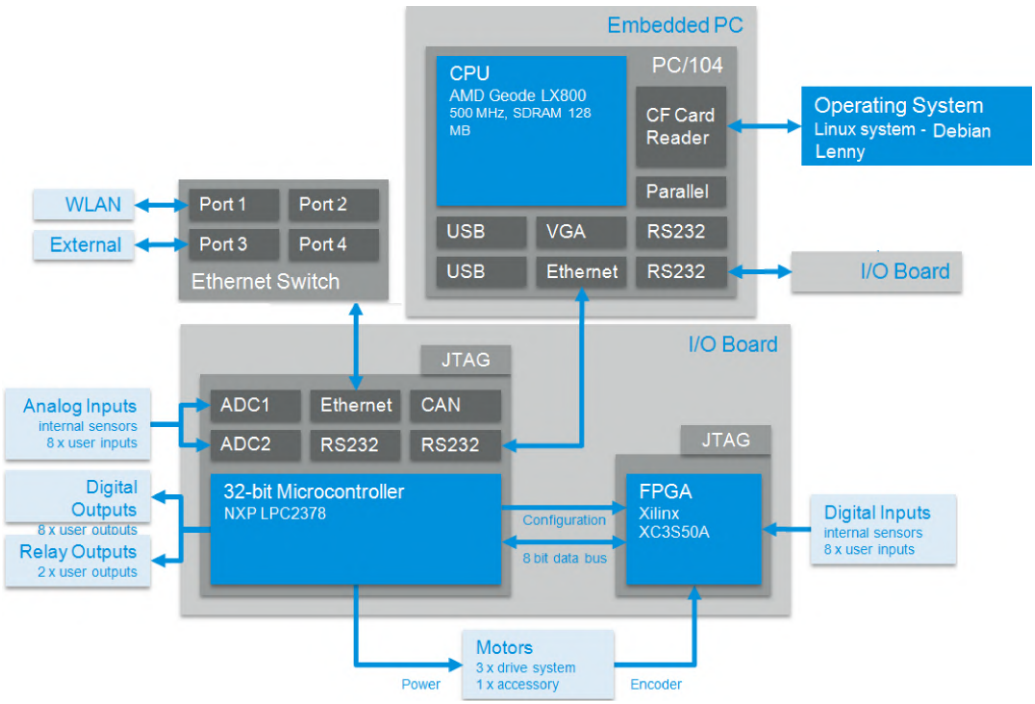
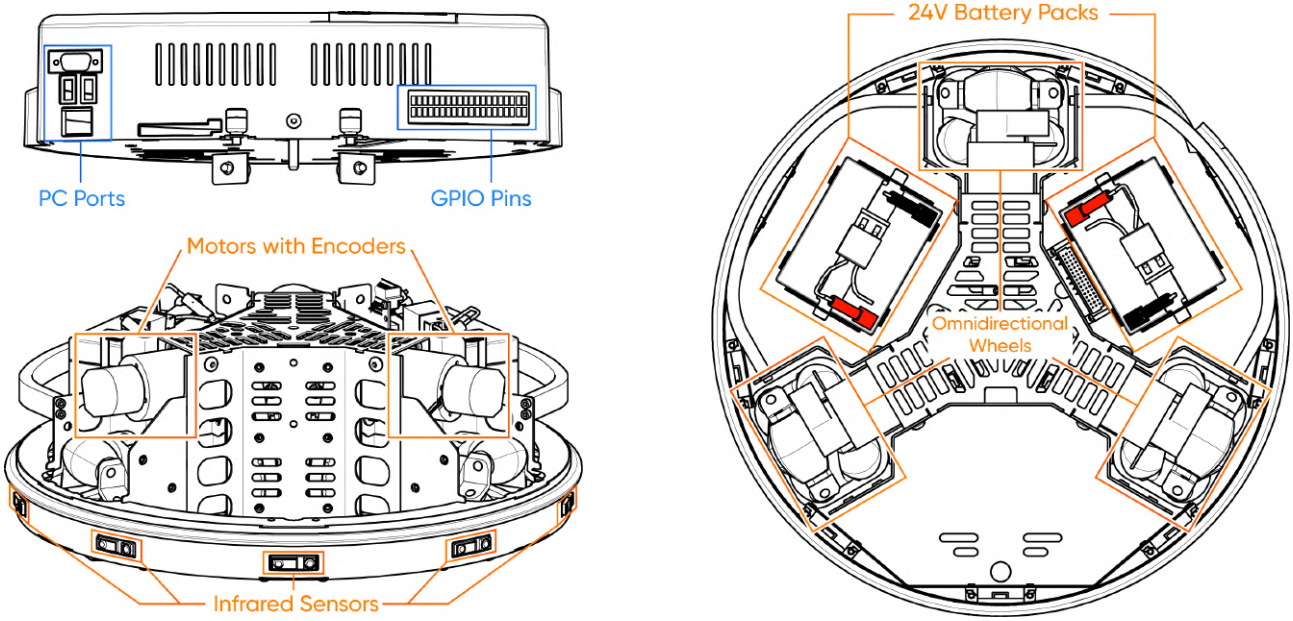


Figure 5.24: Robotino Full Architecture, Adapted from [87].

Robot Components

Other than the onboard computing unit, the robot features a holonomic omni-wheel base driven by **three 24V DC motors** with incremental encoders, as well as a suite of built-in sensors including **nine infrared distance sensors** arranged radially around the chassis. An optional webcam and USB-accessible I/O extension interface are also available, making the platform well suited to sensor-rich navigation experiments. Figure 5.25 shows the various parts of the robot in a mecanical diagram form.



(a) Disassembled View. (b) Top View.

Figure 5.25: Mechanical Diagrams of the Robotino.

Actuation

The Robotino uses **GR 42x25 DC motor** combined with the **PLG 42 S planetary gearbox** due to its high torque capability, compact design and compatibility with the required operating conditions. Its main specifications are summarized in Table 5.7.

Table 5.7: Motor and Gearbox Specifications.

Parameter	Value
Gearbox Type	Planetary PLG 42 S
Gearbox Ratios	4:1 – 512:1
No-load Speed	4200 RPM
No-load Current	170 mA
Starting Torque	20 Ncm
Starting Current	4 A
Nominal Voltage	24 V
Nominal Speed	3600 RPM
Nominal Torque	3.8 Ncm
Nominal Current	0.9 A
Maximum Gearbox Torque	14 Nm
Motor Weight	390 g

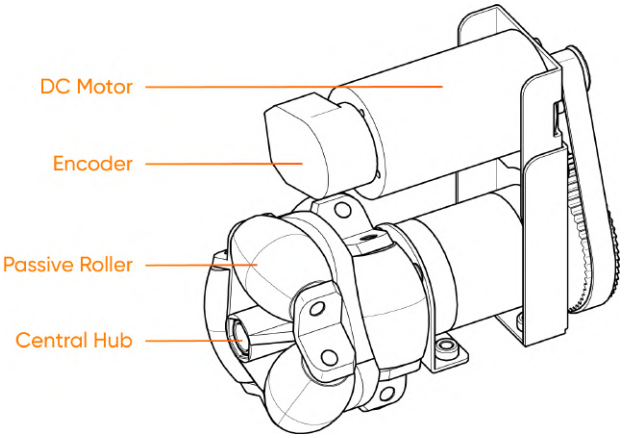


Figure 5.26: GR 42x25 DC Motor with PLG 42 S Planetary Gearbox.

The robot uses omnidirectional wheels. It consists of a **central hub** surrounded by **three passive rollers** mounted around its circumference at an angle so they enable lateral motion with low friction. These rollers are typically coated with rubber to improve traction and reduce vibration. These wheels are connected to the motors using toothed belt in order to save space in the robot as shown in Figure 5.27



(a) Omnidirectional Wheel with Three Rollers.



(b) DC Motor attached to a Wheel.

Figure 5.27: Robotino Wheel.

Robotino View

To program the platform, a dedicated software environment called **Robotino View** is used. It is an interactive graphical programming interface specifically designed for the Robotino platform. The software supports the simultaneous control of multiple robots and represents hardware components through modular function blocks, including motors, sensors, I/O interfaces, cameras, odometry, grippers, manipulators, power outputs and encoder inputs. It also provides advanced function blocks for image processing tasks such as line recognition, color range detection and marker recognition, in addition to navigation functionalities including position control, path following, obstacle avoidance and factory integration. Furthermore, Robotino View supports communication and data exchange through UDP, TCP/IP client-server protocols and OPC interfaces.

Robotino View can be used in two different ways:

- **Directly from the Robotino:** By connecting a monitor to the robot through the VGA port, it is possible to access the onboard operating system and launch Robotino View directly from the embedded computer. Programs can then be created, executed and stored locally on the robot for later standalone operation without requiring the monitor connection.
- **From an external computer:** The Robotino View software can also be installed on a separate PC. By connecting to the Robotino through its WLAN network, the robot can be programmed remotely and the generated programs can be uploaded directly to the platform. The main interface of the Robotino View on a computer can be seen in Figure 5.28.

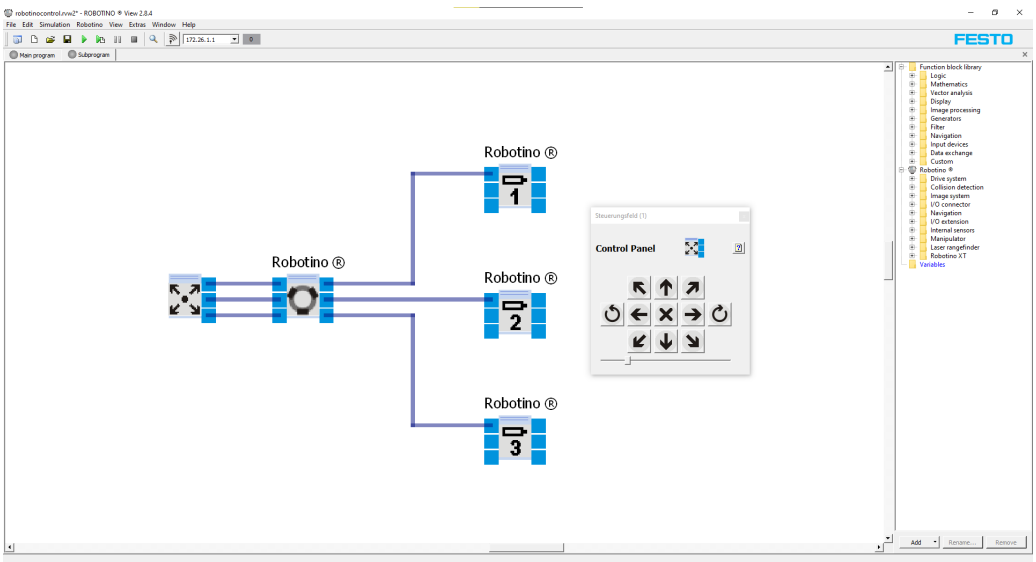


Figure 5.28: Robotino View Interface on Computer.

5.3.2 Holonomic Drive

The Robotino uses a holonomic drive system consisting of three omni-wheels arranged symmetrically at 120° intervals around the chassis. Unlike a differential drive system, the robot can move independently in the x and y directions and rotate about its vertical axis simultaneously, without any reorientation of the chassis.

Kinematic Model

Figure 5.29 shows the kinematic schematic of the three-wheeled platform. Wheels W1, W2 and W3 are separated by 120° and their driving direction vectors are oriented tangentially to the circle of radius L centred on the robot's centre of mass, where L is the distance from the centre to each wheel contact point. The linear velocity of wheel i is V_{wi} and r denotes the radius of each omni-wheel. All velocity components are resolved into a horizontal component along the X-axis (cosine projection) and a vertical component along the Y-axis (sine projection), with $\alpha = 60°$ being the angle used in the projection [88].

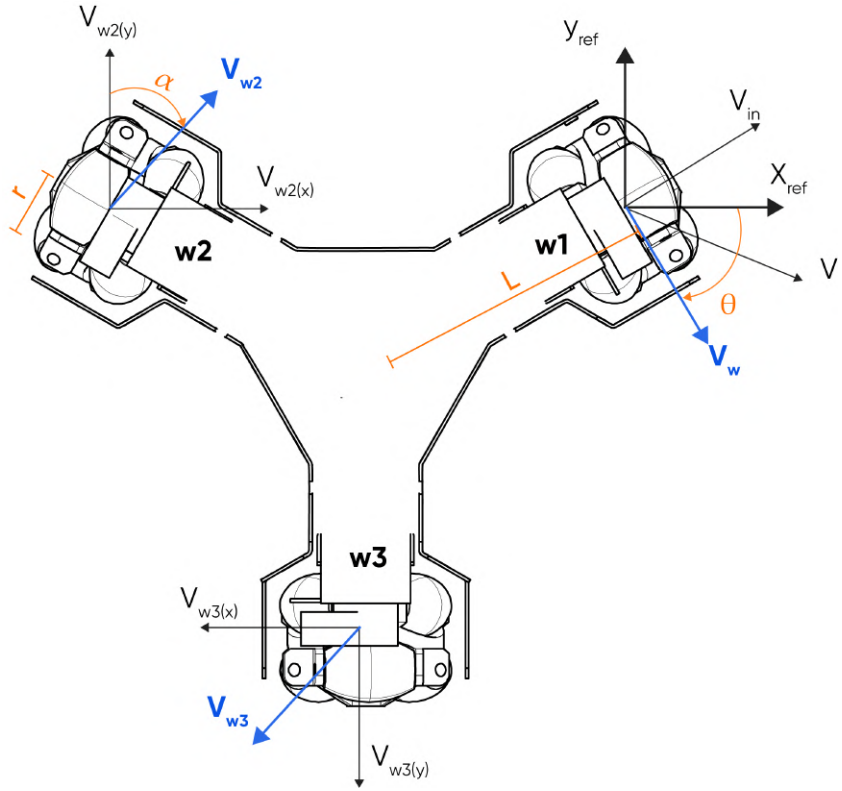


Figure 5.29: Kinematic schematic of the three-wheeled omnidirectional robot platform.

Velocity Decomposition

The contribution of each wheel to the platform body velocity is obtained by projecting its linear velocity onto the body-frame axes. For wheels W1 and W2, with projection angle $\alpha = 60°$:

$$V_{w1(x)} = -V_{w1} \cdot \cos\alpha, \qquad V_{w1(y)} = +V_{w1} \cdot \sin\alpha \tag{5.29}$$

$$V_{w2(x)} = -V_{w2} \cdot \cos\alpha, \qquad V_{w2(y)} = +V_{w2} \cdot \sin\alpha \tag{5.30}$$

Summing contributions from all three wheels gives the resultant body-frame velocity components:

$$V_{w(x)} = V_{w3} - V_{w1}\cos\alpha - V_{w2}\cos\alpha \tag{5.31}$$

$$V_{w(y)} = V_{w1}\sin\alpha - V_{w2}\sin\alpha \tag{5.32}$$

Substituting $\alpha = 60°$ (i.e. $\cos 60 = 1/2$, $\sin 60 = \sqrt{3}/2$):

$$V_{w(x)} = V_{w3} - \frac{V_{w1}}{2} + \frac{V_{w2}}{2} \tag{5.33}$$

$$V_{w(y)} = \frac{\sqrt{3}}{2} V_{w1} - \frac{\sqrt{3}}{2} V_{w2} \tag{5.34}$$

The rotational velocity of the platform about its vertical axis is obtained by summing the wheel speed contributions weighted by the inverse of the moment arm L:

$$V_\theta = \frac{V_{w1}}{L} + \frac{V_{w2}}{L} + \frac{V_{w3}}{L} \tag{5.35}$$

Forward Kinematics

Collecting Equations (5.33), (5.34) and (5.35) into matrix form yields the forward kinematics of the platform:

$$\begin{bmatrix} V_{w(x)} \\ V_{w(y)} \\ V_\theta \end{bmatrix} = \begin{bmatrix} -\frac{1}{2} & -\frac{1}{2} & 1 \\ \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} & 0 \\ \frac{1}{L} & \frac{1}{L} & \frac{1}{L} \end{bmatrix} \begin{bmatrix} V_{w1} \\ V_{w2} \\ V_{w3} \end{bmatrix} \tag{5.36}$$

where $V_{w(x)}$ and $V_{w(y)}$ are the body-frame translational velocity components along the X and Y axes respectively and V_θ is the platform's angular velocity. The resultant translational speed and instantaneous heading angle θ are then:

$$V_r = \sqrt{V_{w(x)}^2 + V_{w(y)}^2}, \qquad \theta = \tan^{-1}\left(\frac{V_{w(y)}}{V_{w(x)}}\right) \tag{5.37}$$

Inverse Kinematics

The inverse kinematics decompose a desired body velocity command ($V_{w(x)}$, $V_{w(y)}$, V_θ) into individual wheel speed targets and are used directly by the velocity command interface. Each wheel linear velocity relates to the motor angular velocity ω_i via the wheel radius r:

$$V_{wi} = \omega_i \cdot r \tag{5.38}$$

Inverting Equation (5.36) gives:

$$\begin{bmatrix} V_{w1} \\ V_{w2} \\ V_{w3} \end{bmatrix} = \begin{bmatrix} -\frac{1}{2} & \frac{\sqrt{3}}{2} & L \\ -\frac{1}{2} & -\frac{\sqrt{3}}{2} & L \\ 1 & 0 & L \end{bmatrix} \begin{bmatrix} V_{w(x)} \\ V_{w(y)} \\ V_\theta \end{bmatrix} \tag{5.39}$$

Each row computes the wheel speed required to jointly satisfy the requested translational and rotational components. The $L\,V_\theta$ term appears in every row because each wheel contributes equally to platform rotation, scaled by the common moment arm L.

Compared to the differential drive model of the custom robot, the holonomic drive removes trajectory constraints: the robot can reach any target pose from any starting configuration without intermediate reorientation manoeuvres. This is particularly advantageous for navigation in confined spaces and for Active SLAM, where the planner benefits from the full range of reachable velocities at every step.

5.3.3 Added Extensions

Unlike the newer versions, **Robotino v2.1 does not natively support ROS2** or advanced programming frameworks. Therefore, custom hardware and software extensions were developed to integrate the Robotino into the proposed ROS2-based system. A Raspberry Pi was added as the main computation unit of the robot, responsible for running the ROS2 nodes and high-level algorithms. The original internal hardware of the Robotino was retained as a low-level control layer, responsible for receiving velocity commands and providing odometry feedback to the Raspberry Pi. Figure 5.30 illustrates the overall architecture of the Robotino system.

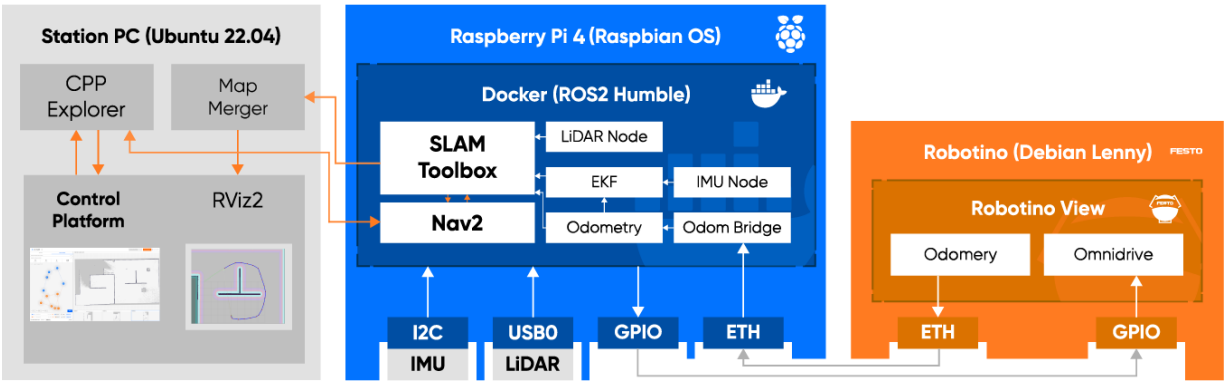


Figure 5.30: Established architecture for the Robotino.

Velocity Command Interface via GPIO

Velocity commands are transmitted from the Raspberry Pi to the Robotino through the I/O interface connector using GPIO pins. Each velocity component (v_x, v_y and ω) is encoded as a PWM signal and sent to a dedicated analog input channel of the Robotino.

A Robotino View program running on the onboard controller decodes these signals, smooths the PWM input and scales the resulting voltage to the velocity range required by the motor controllers, as shown in Figure 5.31.

Since the Raspberry Pi PWM outputs can only generate voltages in the range of 0–3.3 V, negative velocities cannot be represented directly. One possible solution is dividing the PWM voltage range into two segments: one for positive velocities and another for negative velocities. However, this reduces the available resolution for each range. To avoid this problem, an additional GPIO pin is assigned to each velocity channel to represent the sign of the command, while the PWM signal is used only for the velocity magnitude.

The Robotino digital inputs require a switching threshold of approximately 10 V, which exceeds

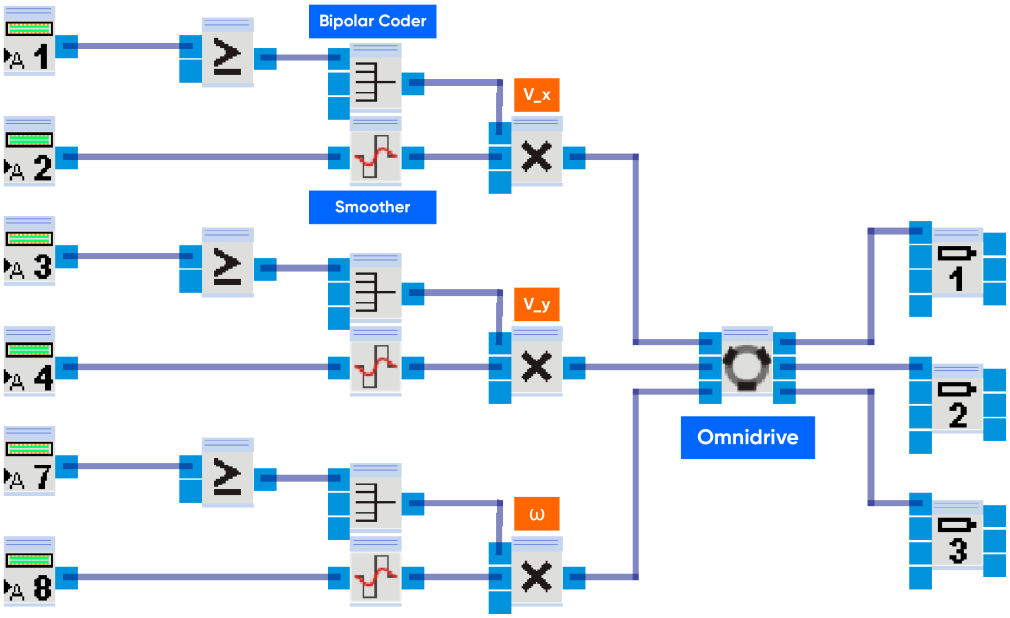


Figure 5.31: Robotino View block diagram implementing the GPIO velocity command decoder.

the Raspberry Pi GPIO output level of 3.3 V. Instead of adding external boost converter, another pwm pin was used: a high-frequency PWM signal is applied such that depending on the velocity sign a pwm value in sent to the robotino via the analogue pins and compared with a constant value to determine if the velocity converted to a negative value or kept as it is.

The decoded sign is then applied to the scaled velocity value. A smoothing block is used to filter the PWM signals before decoding and a multiplication factor ($\times 50$) is applied to map the normalized input signals to the actual velocity range expected by the motor controllers. This provides a final command range of $[-v_{\max}, +v_{\max}]$ for each velocity component.

Odometry Interface

The Festo Robotino 2 exposes its sensor data through a proprietary binary protocol on TCP port 80. Official REST and SOAP endpoints can not be used for Robotino v2.1. The only confirmed working path to live odometry data was the binary stream consumed by **Robotino View**. The approach taken here was to **reverse-engineer the binary protocol** by capturing the traffic between Robotino View and the robot with Wireshark, then parsing the capture to identify the byte offsets carrying x, y and θ.

Wireshark is an open-source network protocol analyzer used to capture, visualize and examine packets exchanged over a network in detail. It provides a clear representation of the different layers of the OSI model and helps identify protocols, IP and MAC addresses and the communication mechanisms being used. Thanks to its filtering and deep packet inspection features, Wireshark is an essential tool for learning, diagnosing and analyzing computer networks especially for a reverse engineering task.

The capture was performed by connecting the host PC to the Robotino's own Wi-Fi hotspot and recording all traffic to and from `172.26.201.1` using Wireshark's capture filter `host 172.26.201.1`. Robotino View was opened in parallel with its *Odometry* block active, providing ground-truth (x, y, θ) values visible on screen during the capture.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	172.26.201.4	172.26.201.1	TCP	66	49202 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
2	0.000269800	172.26.201.1	172.26.201.1	TCP	66	49203 → 8180 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
3	0.000933000	172.26.201.4	172.26.201.1	TCP	66	49204 → 8280 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
4	0.001170800	172.26.201.4	172.26.201.1	TCP	66	49205 → 8380 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
5	0.003617600	172.26.201.1	172.26.201.4	TCP	66	80 → 49202 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=1460 SACK_PERM WS=4
6	0.003731100	172.26.201.4	172.26.201.1	TCP	54	49202 → 80 [ACK] Seq=1 Ack=1 Win=65280 Len=0
7	0.003960800	172.26.201.1	172.26.201.4	TCP	60	8180 → 49203 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
8	0.004525300	172.26.201.1	172.26.201.4	TCP	60	8280 → 49204 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
9	0.004857400	172.26.201.1	172.26.201.4	TCP	60	8380 → 49205 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
10	0.006820800	172.26.201.1	172.26.201.4	TCP	1514	80 → 49202 [ACK] Seq=1 Ack=1 Win=5840 Len=1460
11	0.007528600	172.26.201.1	172.26.201.4	TCP	598	80 → 49202 [PSH, ACK] Seq=1461 Ack=1 Win=5840 Len=544
12	0.007613000	172.26.201.4	172.26.201.1	TCP	54	49202 → 80 [ACK] Seq=1 Ack=2005 Win=65280 Len=0
13	0.028978900	172.26.201.4	172.26.201.1	TCP	133	49202 → 80 [PSH, ACK] Seq=1 Ack=2005 Win=65280 Len=79
14	0.030565500	172.26.201.1	172.26.201.4	TCP	60	80 → 49202 [ACK] Seq=2005 Ack=80 Win=5840 Len=0
15	0.030620500	172.26.201.4	172.26.201.1	TCP	141	49202 → 80 [PSH, ACK] Seq=80 Ack=2005 Win=65280 Len=87
16	0.031158800	172.26.201.1	172.26.201.4	TCP	198	80 → 49202 [PSH, ACK] Seq=2005 Ack=80 Win=5840 Len=144
17	0.072348100	172.26.201.1	172.26.201.4	TCP	60	80 → 49202 [ACK] Seq=2149 Ack=167 Win=5840 Len=0
18	0.076500300	172.26.201.4	172.26.201.1	TCP	54	49202 → 80 [ACK] Seq=167 Ack=2149 Win=65280 Len=0
19	0.076863200	172.26.201.4	172.26.201.1	TCP	133	49202 → 80 [PSH, ACK] Seq=167 Ack=2149 Win=65280 Len=79
20	0.077781500	172.26.201.1	172.26.201.4	TCP	198	80 → 49202 [PSH, ACK] Seq=2149 Ack=167 Win=5840 Len=144

Figure 5.32: Wireshark packet capture.

Reassembling all TCP payloads originating from port 80 on the robot revealed that, after an initial ASCII status block terminated by a dashed separator line, the stream transitions to a continuous sequence of **144-byte binary frames** delivered at approximately **25 Hz**. Every frame begins with the two-byte sequence `0x01 0x8D` followed, at offset 4, by the four-byte magic `\x00REC`, identifying the proprietary *REC protocol*. A sequence counter at offset 3 alternates between `0x36` and `0x37` on successive frames.

To locate the odometry bytes, each of the 543 motion-capture frames was compared byte-by-byte against the first (stationary) frame. Offsets whose value *never* changed were immediately discarded. The remaining candidate offsets were then interpreted as `float32` and their time-series were plotted against the known motion profile. Three fields matched the expected odometry signatures unambiguously:

- **Offset 125** (`float32 LE`): increased monotonically from 0 m to −0.627 m during the forward push, then stabilised during rotation ⇒ x **position, unit: metres**.
- **Offset 129** (`float32 LE`): rose from 0 m to 2.010 m over the full motion sequence, with negligible variation during the pure forward push ⇒ y **position, unit: metres**.
- **Offset 133** (`float32 LE`): was zero during the forward push, then rose to 169.841° counts at the end of the rotation ⇒ **heading θ, unit: degrees**.

A fourth field at **offset 36** (`uint32 LE`) incremented at exactly 1 000 counts per second across the entire capture, confirming it is the robot's internal **millisecond timestamp**.

```
0x00:  01 8d 00 37 00 52 45 43 00 00 f7 03 00 01 00 00
0x10:  00 00 50 0a 01 02 00 00 00 00 01 00 00 80 19 95
0x20:  00 00 00 0f 37 56 20 00 00 14 03 00 00 00 00 06
0x30:  00 00 01 00 c2 0e 00 00 0f 38 56 20 00 05 aa ff
0x40:  00 00 00 00 76 00 00 01 17 b2 83 ff ff 0f 39 56
0x50:  20 00 00 00 00 01 00 01 01 02 00 00 01 00 00 00
0x60:  00 00 0f 00 00 00 00 72 65 63 00 00 00 00 00 00
0x70:  00 00 00 00 00 00 00 00 00 00 00 00 00 e0 f8 92
0x80:  43 66 8d 51 40 c3 34 67 43 00 00 00 00 00 00 00
```

Protocol magic Timestamp x y θ

Figure 5.33: Annotated hex dump of a 144-byte REC frame.

Table 5.8 summarises all fields.

Offset	Type	Raw unit	Field
36	`uint32 LE`	ms	Robot timestamp
125	`float32 LE`	m	Position x
129	`float32 LE`	m	Position y
133	`float32 LE`	deg	Heading θ

Table 5.8: Confirmed field offsets in the 144-byte REC frame.

The Robotino controller does not treat the TCP connection as a passive data pipe: it actively monitors whether the client is still present and closes the connection within a few seconds if no traffic is received. Analysis of the Wireshark capture showed that Robotino View sends two distinct request packets to the robot at regular intervals; `REC_79` and `REC_87`.

Both packets share the same `REC` protocol magic and the same *rec* subsystem identifier, but differ slightly in their header prefix. `REC_79` is the initial subscription request: it carries the field `0x0090 = 144` which instructs the robot to send back 144-byte sensor frames and it sets an update-rate factor. `REC_87` is an extended variant with an additional 8-byte prefix that appears to act as a session-continuation or acknowledgement token; the remaining 79 bytes are identical to `REC_79`. During normal operation, Robotino View alternates between the two packets, sending one every ≈ 30 ms, for a combined round-trip period of ≈ 60 ms. If neither packet arrives within the controller's internal timeout, the TCP stream is silently dropped.

The integration node therefore dedicates a separate daemon thread (`_keep_alive`) solely to this task. It runs an infinite loop that sends `REC_79`, sleeps 30 ms, sends `REC_87`, sleeps another 30 ms and repeats. This thread is completely decoupled from the odometry reading loop and the motor command thread so that a slow consumer on the main thread cannot inadvertently starve the keep-alive and cause the connection to drop.

ROS2 Integration

To integrate the Robotino v2.1 into ROS2, a dedicated software interface was developed and deployed inside a **Docker container** running on the Raspberry Pi. This interface is implemented as a custom ROS 2 package that consists of two main nodes responsible for odometry acquisition and motion control, respectively:

- **Odometry Node:** This node acquires odometry from the Robotino controller through a TCP connection, maintains the keepalive communication, parses incoming binary packets, extracts the robot pose (x, y, θ), converts the heading to a quaternion, and **publishes** standard `nav_msgs/Odometry` messages on the `/odom` topic.
- **Velocity Control Node:** This node **subscribes** to the `/cmd_vel` topic and receives velocity commands in the form of v_x, v_y, and ω. The received commands are translated into PWM signals generated by the Raspberry Pi GPIO interface and transmitted to the Robotino controller.

Sensor Fusion

The same Extended Kalman Filter approach described in the previous section is applied to the Robotino. Wheel odometry is fused with IMU measurements to correct accumulated heading drift. The EKF node is configured with the same state vector $\mathbf{x} = [x, y, \theta]^\top$ and covariance structure as the custom robot. The primary difference lies in the sensor topic names, which are remapped to the Robotino's namespace and minor covariance retuning to account for the different odometry characteristics of the holonomic drive system compared to the differential drive.

LiDAR Integration

For SLAM's environment perception, an **RPLIDAR A2M8** LiDAR was used on the Robotino which provides 360° planar scanning. Its specifications are listed in Table 5.9.

Table 5.9: RPLIDAR A2M8 Specifications.

Parameter	**Value**
Measuring Range	0.2 – 12 m
Sampling Frequency	8,000 points/s
Rotational Speed	10 Hz (5 – 15 Hz configurable)
Angular Resolution	0.45°



Figure 5.34: RPLIDAR A2M8.

Robot Assembly

The final Robotino platform after all hardware modifications is shown in Figure 5.35. The original Robotino v2.1 hardware was preserved as the low-level control layer. The RPLIDAR A2M8 and the IMU were mounted on the top of the platform, and the Raspberry Pi was interfaced with the Robotino I/O board and Ethernet to exchange velocity commands and odometry data.

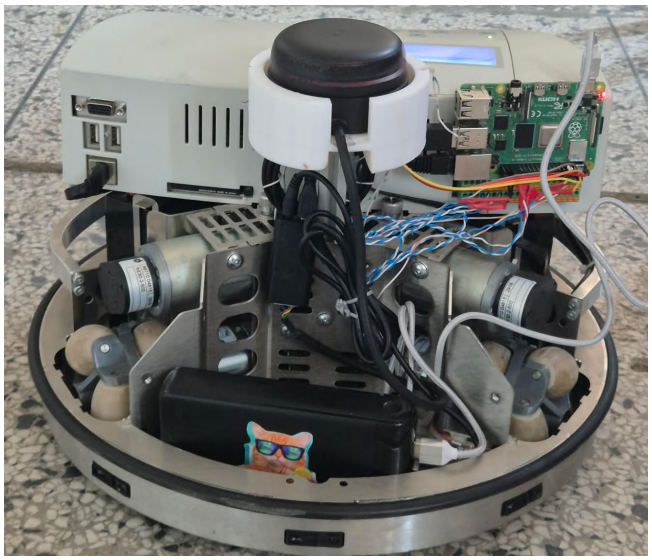


Figure 5.35: Final assembled Robotino platform.

5.4 SLAM and Navigation Deployment

This section presents the SLAM and Navigation implementation on the custom robot and the Robotino, the steps and the rules that must be followed to insure high quality map generation.

5.4.1 SLAM Configuration

The SLAM node takes as input the odometry data, the TF transforms between frames and the LiDAR scans. Based on these inputs, it generates and continuously updates the map of the environment. The `slam_toolbox` package is used to perform SLAM and generate the map of the environment in real time. The same SLAM pipeline described in the simulation is reused on the real robot.

Sensor Initialization

After ensuring that the fused odometry and the LiDAR is working properly then verifying that the TF tree is correctly set up following REP 105, the SLAM process to generate the map of the environment can be started. The LiDAR should be oriented toward the front of the robot. If not, it can cause map distortion or overlapping during robot motion, which affects the quality of the generated map. Figure 5.36 shows the frame of the LiDAR for the Slamtec A2 series.

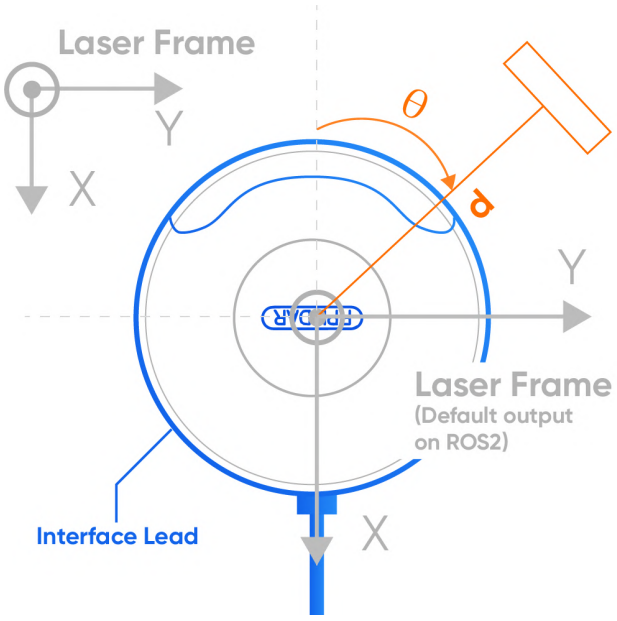


Figure 5.36: RPLidar A2 frame.

Mapping Process Using Teleoperation

Before moving to Active SLAM, it is important to verify that the SLAM system is working correctly. This could be done using teleoperation, by manually controlling the robot and observing the map generation in real time.

The robot is controlled using the `teleop_twist_keyboard` package, which allows sending velocity commands using keyboard inputs. This package works for both holonomic and non-

holonomic robots, making it suitable for this system. During this step, the robot is moved around the environment to allow the SLAM system to build the map. The generated map is visualized in RViz to check its quality, consistency and correctness.

5.4.2 Navigation Parameter Tuning

The default Nav2 parameters were not directly suitable for the target platform because the system was designed for autonomous exploration rather than precise goal reaching. Several parameters were adjusted to improve exploration efficiency while reducing the computational load on the Raspberry Pi.

Global Planner Selection

The **NavFn planner** was selected as the global planner due to its low computational complexity and stable behavior in indoor environments. Since the objective of the robot is autonomous exploration and map generation rather than highly optimized trajectory planning, the NavFn planner provided sufficient performance while keeping the computation load relatively low.

Goal Tolerance Adjustment

The navigation goal tolerances were increased to avoid unnecessary corrective movements near the target position. The `xy_goal_tolerance` parameter was increased to allow the robot to stop near the target instead of requiring perfect positioning. In addition, the `yaw_goal_tolerance` was set to π radians, meaning that the final robot orientation is not constrained.

This configuration is suitable for Active SLAM applications because the robot continuously moves toward new exploration goals and does not require precise orientation alignment at each target point. Reducing the orientation constraint also minimizes oscillatory corrections near the goal and produces smoother navigation behavior. Figure 5.37 shows the influence of the goal tolerance radius on navigation. A larger tolerance allows the robot to stop within an acceptable region around the target, reducing corrective movements and improving navigation smoothness.

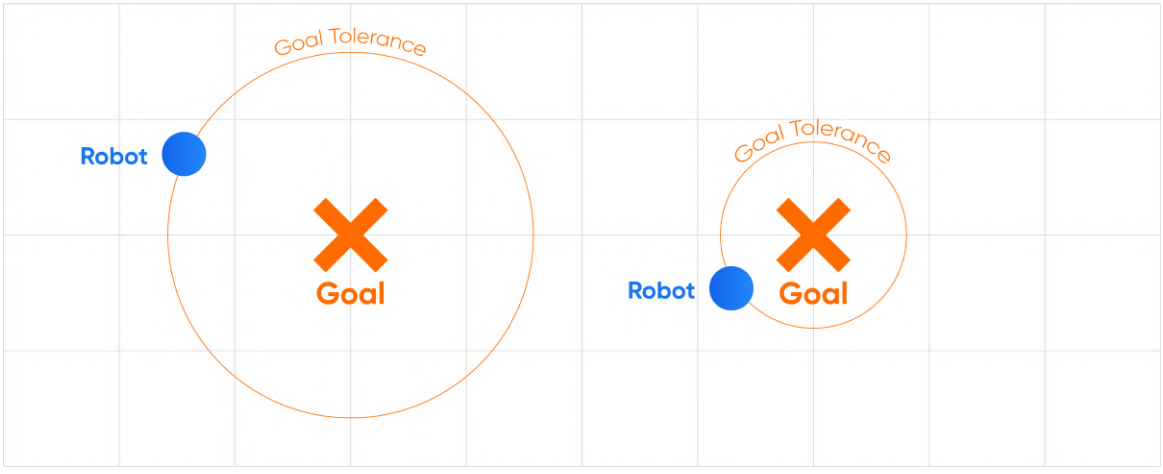


Figure 5.37: Goal tolerance effect on robot navigation.

Inflation Radius Reduction

The inflation radius of both the local and global costmaps was reduced because the robot platform is physically small and operates inside narrow indoor environments.

A smaller inflation radius allows the robot to navigate closer to walls and obstacles, which improves mobility inside tight corridors and reduces unnecessary detours. This also increases exploration efficiency by allowing the planner to generate shorter and more direct paths. However, reducing the inflation radius also reduces the safety margin around obstacles, which requires reliable localization and obstacle detection. Figure 5.38 shows a comparison of path planning with different inflation radii around an obstacle. A smaller inflation radius (right) allows the robot to follow a shorter path closer to obstacles, while a larger inflation radius (left) increases the safety margin but results in a longer and more conservative trajectory.

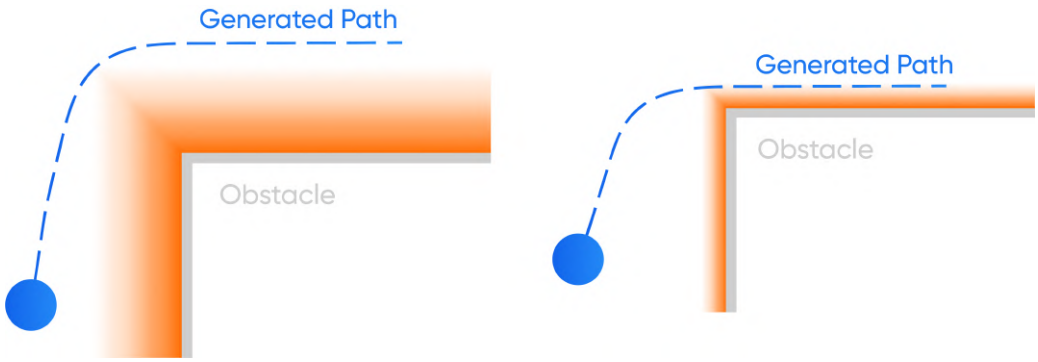


Figure 5.38: Inflation radius effect on robot navigation.

Local Controller Critic Tuning & Controller Frequency

The local trajectory critics were tuned to prioritize exploration progress rather than strict trajectory tracking. The weight of the `PathAlign` critic was increased to improve trajectory consistency and reduce unstable local paths. At the same time, the `FollowPath` critic weight was reduced to give the controller more freedom to locally deviate from the global path when necessary. This tuning allows the robot to prioritize reaching the exploration target efficiently instead of strictly following the exact global trajectory, which is more suitable for SLAM and exploration tasks.

The local controller frequency was reduced to $10\,Hz$ in order to decrease CPU utilization on the Raspberry Pi. Higher controller frequencies increase trajectory update rates but also significantly increase the computational load due to more frequent costmap updates and trajectory evaluations. Since the robot operates at moderate speed in indoor environments, a frequency of $10\,Hz$ was sufficient to maintain stable navigation while reducing processor usage.

Omnidirectional local planner

The same previous parameters were applied on the Robotino platform with few modifications compared to the custom robot because the omnidirectional motion already provides smoother maneuverability in narrow environments.

The Robotino supports lateral motion along the y-axis. Therefore, the local planner was configured to use the y velocity component during navigation. In addition, rotational alignment

constraints were reduced because the omnidirectional robot does not need to rotate before changing direction. This allowed smoother motion and reduced unnecessary rotational movements during exploration and navigation tasks.

5.4.3 Navigation Challenges

During the implementation and testing of the navigation system on real hardware, several challenges were encountered that affected the robot localization and navigation performance.

- **Wheel Slipping:** Wheel slipping can affect odometry accuracy since encoder measurements no longer represent the actual robot displacement. This effect becomes more significant during fast motions or abrupt direction changes. To mitigate this issue, the robot was operated at relatively low speeds, resulting in smoother and more stable motion.
- **Odometry Drift and Localization Errors:** Wheel odometry accumulates errors over time, particularly during long trajectories and sharp rotations. These errors were reduced by filtering the odometry data and fusing it with IMU measurements through an Extended Kalman Filter (EKF), providing more reliable localization estimates.
- **TF Timing Issues:** Timing differences between the Raspberry Pi and the Docker container caused TF synchronization problems, leading to inconsistent transform timestamps. This issue was mitigated by synchronizing the clocks of both systems to use the same time reference.
- **CPU Limitations on Raspberry Pi:** Running SLAM and navigation simultaneously significantly increased CPU usage, particularly due to continuous costmap updates and path planning. To reduce the computational load, ROS 2 DDS communication was used to distribute processing across devices, with the navigation stack executed on a personal computer while the Raspberry Pi handled the remaining tasks.

5.5 Active Collaborative SLAM Deployment

The Active SLAM framework deployed on real hardware reuses the same architecture presented in Section 4.6. From prior graph representation to Eulerianisation, CPP-based node sequencing, and Nav2 integration, without structural modifications. The only notable adaptation concerns prior graph construction: since exact environment dimensions were not always available, node positions were estimated from the building layout and manual measurements, then refined iteratively to ensure all navigation goals were physically reachable. The invalid-goal handling strategy is preserved. For the collaborative case, the full multi-robot architecture from Section 4.6 was deployed directly, with each robot maintaining its own namespace, TF tree, SLAM process, and navigation stack; BFS-based graph partitioning assigns each robot an exclusive sub-graph, and the `map_merger` node fuses `/robot1/map` and `/robot2/map` into a unified grid on `/merged_map` upon completion. Real-world challenges such as odometry drift, wheel slip, and communication latency were mitigated through the parameter tuning, confirming the portability of the proposed architecture from simulation to real hardware.

5.6 AC-SLAM Mission Control Platform

To facilitate interaction with the robots during both testing and real-world deployment, a dedicated web-based control platform was developed. Rather than relying solely on terminal commands or RViz, the platform provides a unified graphical interface that integrates teleoperation, graph partitioning, exploration control, and live map visualization into a single browser-accessible application.

5.6.1 Architecture Overview & ROS Bridge Integration

The platform follows a client-server architecture composed of two layers, as shown in Figure 5.39.

- The **frontend** is a single-page HTML/CSS/JavaScript application that uses `roslibjs` to communicate with the ROS 2 network via `rosbridge_server` by subscribing to `/robot1/map`, `/robot2/map`, and `/merged_map` for live map rendering, and publishing `Twist` messages to `/robotX/cmd_vel` for teleoperation.
- The **backend** is a lightweight Python HTTP server (`server.py`) exposing three endpoints: `POST /launch` to write a YAML graph file and spawn a `cpp_explorer` process, `POST /stop` to terminate it, and `GET /logs` to stream output to the browser.

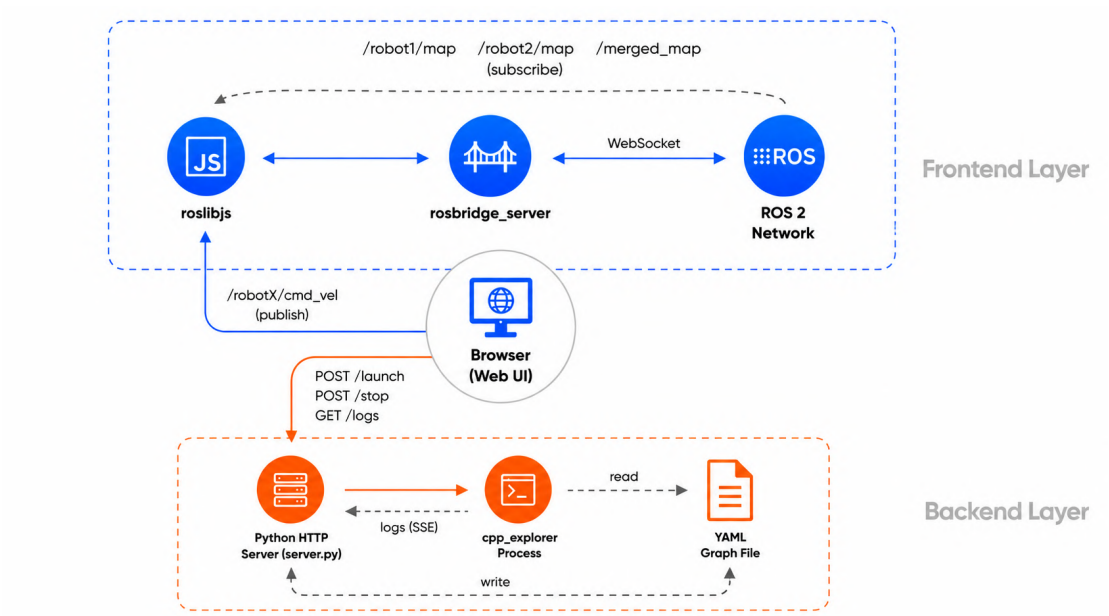


Figure 5.39: AC-SLAM platform architecture.

5.6.2 User Interface

The platform interface is divided into two main areas: a left control panel and a right map visualization area. The connection bar at the top allows the operator to specify the `rosbridge` WebSocket URL and connect to the ROS 2 network with a single click, with a live status indicator reflecting the connection state.

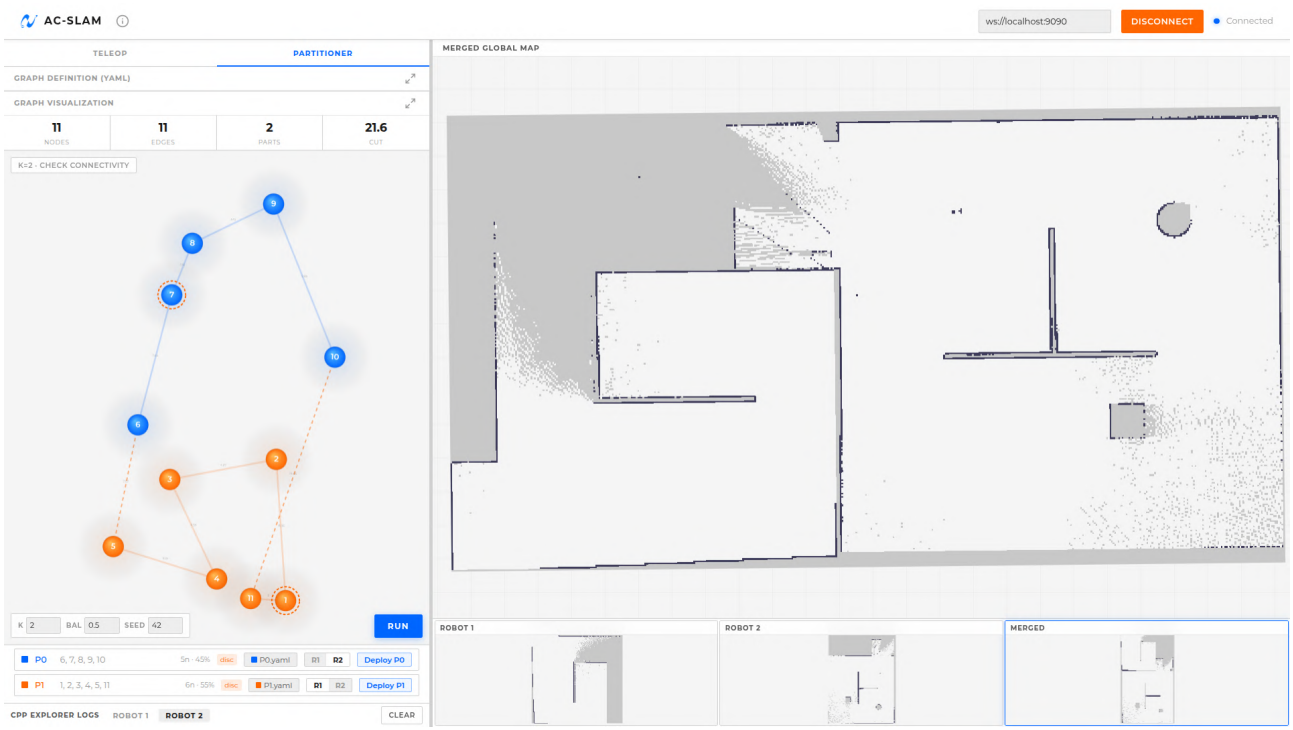


Figure 5.40: AC-SLAM Mission Control Platform Interface.

Map Visualization

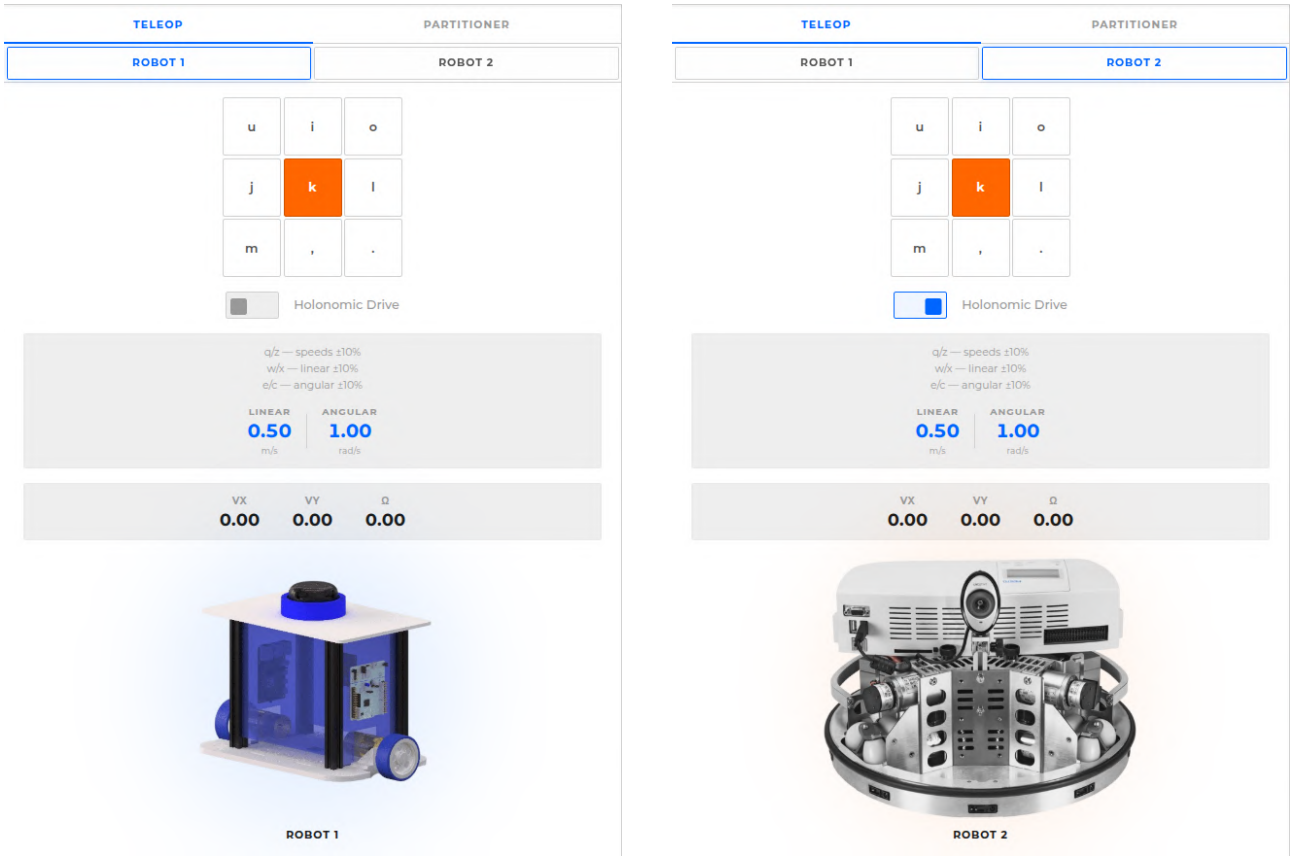
The right panel of the interface provides continuous live map visualization. Individual occupancy grids from `/robot1/map` and `/robot2/map` are rendered on separate canvas thumbnails, while the merged global map from `/merged_map` is displayed in a larger main view. The operator can click any thumbnail to promote it to the main view, enabling quick inspection of individual or combined maps during exploration.



Figure 5.41: Map visualization panel.

Teleop Mode

The Teleop mode, shown in Figure 5.42, provides manual control of either robot. The operator selects the active robot via the `ROBOT 1` / `ROBOT 2` tab, then drives it using an 8-direction D-pad (keys `u`, `i`, `o`, `j`, `k`, `l`, `m`, `,`, `.`) directly from the keyboard or on-screen buttons. A holonomic toggle switches the velocity model between non-holonomic (differential drive) and holonomic (omnidirectional) kinematics. Linear and angular speeds are displayed and adjustable incrementally, and a live velocity readout shows the current v_x, v_y, and ω values being published to the robot's `cmd_vel` topic.



(a) Robot 1 teleoperation interface

(b) Robot 2 teleoperation interface

Figure 5.42: Teleoperation mode of the AC-SLAM platform for Robot 1 and Robot 2.

Partitioner Mode

The Partitioner mode, shown in Figure 5.43, provides the complete workflow for launching collaborative exploration. The left panel is organized into three sections. The *Graph Definition* section contains a YAML editor where the operator defines the prior connectivity graph — nodes with their coordinates and edges with their costs. The *Graph Visualization* section renders the graph live on an interactive canvas and displays summary statistics including the number of nodes, edges, partitions, and the cut cost. Partitioning parameters — number of sub-graphs k, balance factor, and random seed — are configurable via overlay inputs.

Clicking *RUN* triggers the BFS-based partitioning algorithm, which splits the graph into color-coded sub-graphs assigned to each robot and generates the corresponding YAML files. Each partition entry shows the assigned node list, estimated coverage time, and deploy buttons to send the graph to `Robot 1` or `Robot 2`, which issue a `POST /launch` request to the Python

server to start the `cpp_explorer` node for the corresponding namespace. The *CPP Explorer Logs* section at the bottom streams the live output of each robot's exploration process via Server-Sent Events, with per-robot tabs for isolated log inspection.

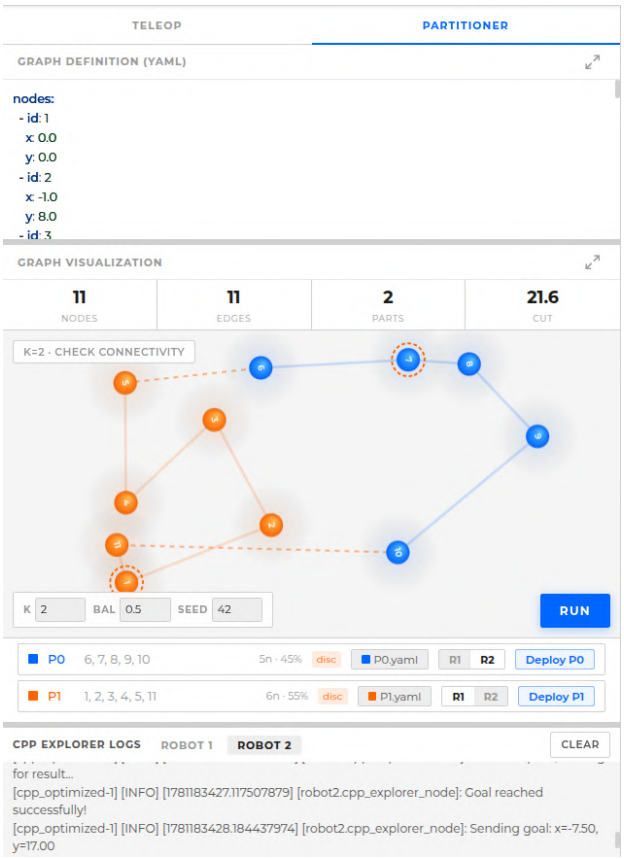


Figure 5.43: Partitioner mode of the AC-SLAM platform.

5.7 Conclusion

This chapter presented the complete deployment of the Active Collaborative SLAM system on two real robotic platforms: a custom-built differential drive robot and a Festo Robotino v2.1. Starting from hardware design and low-level control, through SLAM and navigation configuration, to full autonomous exploration, each layer of the system was successfully adapted and validated on physical hardware. The AC-SLAM Mission Control Platform further provided a unified interface for teleoperation, graph partitioning, and live map visualization, completing the operational deployment of the system. The following chapter presents the experimental results obtained from both simulation and real-world trials, evaluating the performance of the proposed framework across all deployment scenarios.

Chapter 6

Results & Discussions

6.1 Introduction

This chapter presents the experimental results obtained from the implementation of the proposed Active Collaborative SLAM framework in both simulation and real-world environments. The objective of these experiments is to evaluate the performance of the system in terms of mapping accuracy, autonomous navigation, and collaborative exploration.

The evaluation is carried out using a set of quantitative metrics that compare the generated occupancy maps with reference ground-truth maps. These metrics provide an objective assessment of the quality of the maps produced during the exploration process. In addition, the navigation performance of the robots is analyzed by evaluating their ability to reach assigned exploration goals, avoid obstacles and recover from navigation failures.

The experiments are conducted using both single-robot and multi-robot configurations. This allows the impact of collaborative exploration on mapping performance and exploration robustness to be evaluated. Furthermore, a comparison between simulation and real-world deployments is presented in order to evaluate the transferability of the proposed framework from controlled environments to practical robotic platforms.

6.2 Experimental Results and Evaluation

6.2.1 Evaluation Metrics

To evaluate the performance of the proposed exploration and mapping framework, a set of quantitative metrics is utilized to measure the fidelity of the generated maps. These metrics compare the occupancy grid produced by the SLAM process with a ground-truth map of the environment.

The generated map corresponds to the occupancy grid produced by SLAM Toolbox and saved in the `.pgm` format after the exploration process is completed. Before computing the metrics, the generated map is aligned with the ground truth map in the same reference frame to ensure a fair comparison.

For the real world experiments, obtaining an exact ground truth occupancy map is considerably

more difficult. Therefore, the same ground truth map used in the simulation will be used. The generated occupancy grid is then compared against this reference map using the same evaluation methodology.

Four distinct metrics are considered in this work to evaluate the mapping accuracy across both global alignments and local structural boundaries:

Correlation Similarity

The global alignment and matching consistency between the maps is evaluated using the Normalized Cross-Correlation (NCC). It assesses the structural pattern similarity of the map layouts and is formulated as:

$$C_N(y) = \frac{\langle I_t T \rangle - \langle I_t \rangle \langle T \rangle}{\sigma(I_t)\sigma(T)} \tag{6.1}$$

where (I_t) denotes the generated map to be evaluated and (T) represents the reference (ground truth) map. The operator $(\langle \cdot \rangle)$ corresponds to the average value computed over the overlapping region of the two maps, while $(\sigma(\cdot))$ denotes the standard deviation of the corresponding map values within the same region. The numerator measures the covariance between the generated map and the reference map, whereas the denominator normalizes this value using the standard deviations of both maps. As a result, the normalized correlation coefficient $(C_N(y))$ provides a measure of similarity between the two maps. A value close to 1 indicates a strong similarity, a value close to 0 indicates little correlation, and a negative value indicates an inverse correspondence between the maps.

Map Score

The map score evaluates the absolute point-wise difference between the occupancy probabilities of the generated map and the ground-truth map. It treats the maps as rigid spatial arrays and calculates the global absolute mismatch, defined as:

$$\textit{map-score} = \sum_{i,j} \left(P(m_{i,j}) - P(n_{i,j})\right)^2 \tag{6.2}$$

where $P(m_{i,j}), P(n_{i,j}) \in \{0, 1\}$ denote the binary occupancy state (free or occupied) of cell (i, j) in the generated map m and the ground-truth map n, respectively. A lower map score indicates fewer mismatched pixels. However, this metric is highly sensitive to minor wall thickness variations and pixel-level discretizations noise.

Signed Distance Field (SDF) Score

To reduce the vulnerability of the evaluation to minor wall thickness variations, a Signed Distance Field (SDF) representation is adopted. Instead of directly comparing binary occupancy values, the distance of each cell to its nearest structural obstacle boundary is computed via an Euclidean Distance Transform (EDT).

The SDF score sums the squared differences of these distance fields across the map grid and is defined as:

$$\textit{SDF-score} = \sum_{i,j} \left(s_{i,j} - t_{i,j}\right)^2 [\text{px}^2], \tag{6.3}$$

where $s_{i,j}$ and $t_{i,j}$ represent the signed pixel-distance values of cell (i, j) in the generated map and the ground-truth map, respectively. Because distances are computed relative to grid coordinates, the units are expressed in squared pixels (px^2).

Relaxed SDF Score

Although the standard SDF score scales smoothly with distance, small local deviations and sensor discretized boundaries may still contribute disproportionately to the total accumulated error. For this reason, a relaxed SDF score is implemented to filter out low-level discretization noise.

The relaxed SDF score introduces a threshold dead-band of exactly 1 pixel (which corresponds to a physical dimension of $\delta = 0.05$ m under our configured map resolution). It is mathematically defined as:

$$\textit{rel.SDF-score} = \sum_{i,j} \begin{cases} \left(|s_{i,j} - t_{i,j}| - 1\right)^2 & \text{if } |s_{i,j} - t_{i,j}| > 1 \text{ px}, \\ 0, & \text{otherwise.} \end{cases} \tag{6.4}$$

This metric completely disregards minor structural alignments under 5 cm and focuses entirely on macro-level mapping errors and geometric structural distortions, making it highly representative of navigation safety and route planning viability.

6.2.2 SLAM Performance Evaluation

This subsection evaluates the performance of the proposed SLAM framework in both simulation and real-world environments. The objective is to verify that the robots can successfully build a consistent map of the environment while autonomously exploring the assigned graph nodes.

Single-Robot Simulation Results

Before evaluating the proposed collaborative exploration framework, a preliminary experiment was conducted using a single robot. The objective of this experiment was to validate the Active SLAM architecture and establish a baseline for comparison with the multi-robot approach.

The robot was assigned the complete prior graph and sequentially navigated through all exploration nodes while simultaneously performing SLAM. Figure 6.1 shows the occupancy map generated during the exploration process.

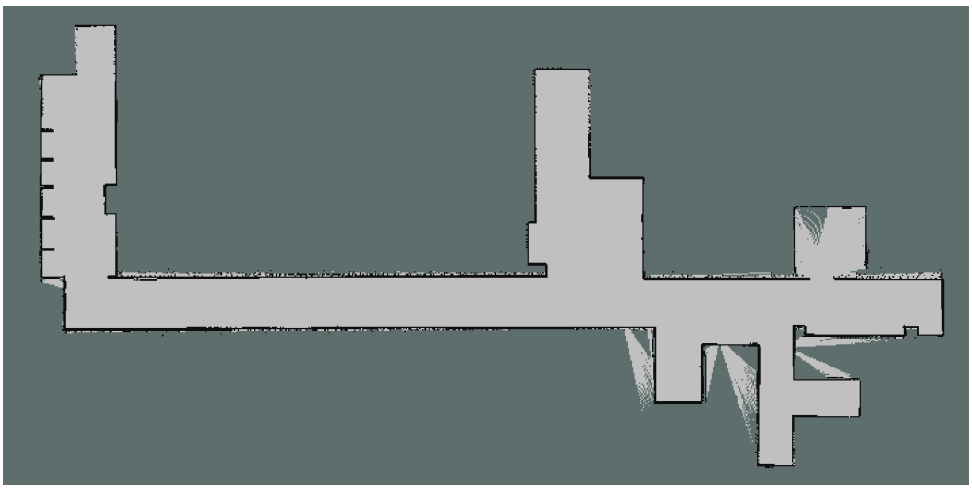


Figure 6.1: Map generated during the single-robot simulation experiment

Single-Robot Deployment Results

A preliminary experiment was also conducted using a single robot in the real environment. The objective was to evaluate the performance of the Active SLAM framework under real operating conditions before deploying the collaborative multi-robot system.

The robot was assigned the complete exploration graph and autonomously navigated between the generated exploration nodes while simultaneously performing SLAM. Figure 6.2 shows the map generated during the experiment.

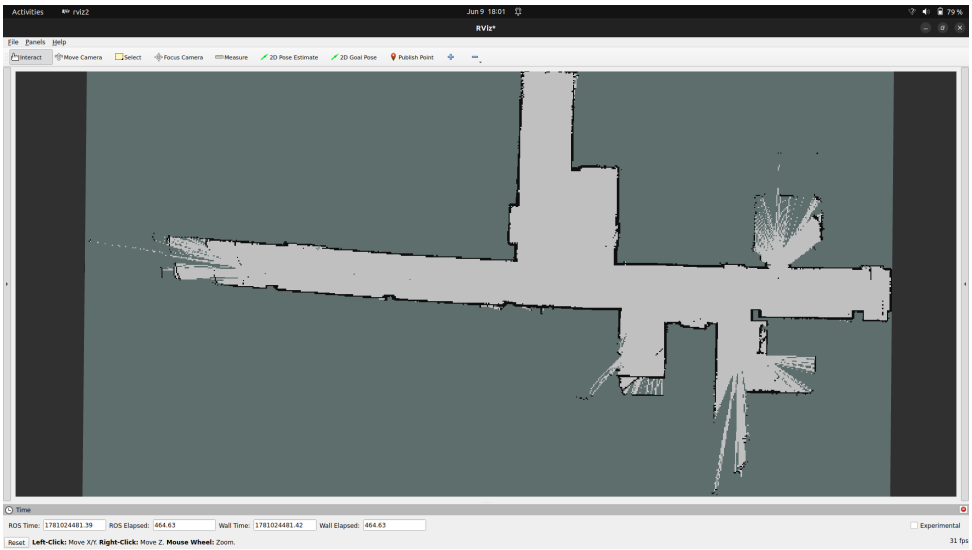


Figure 6.2: Map generated during the single-robot real-world experiment

The robot successfully explored most of the environment and generated a consistent occupancy map. However, unlike the simulation experiment, the navigation and localization processes were affected by odometry drifts.

As the robot progressed through the environment, the drifts gradually accumulated and caused a slight distortion of the generated map. Near the end of the exploration mission, two navigation goals became unreachable because their estimated positions no longer corresponded to valid free-space locations in the map. Consequently, the exploration process was terminated before

all graph nodes could be visited.

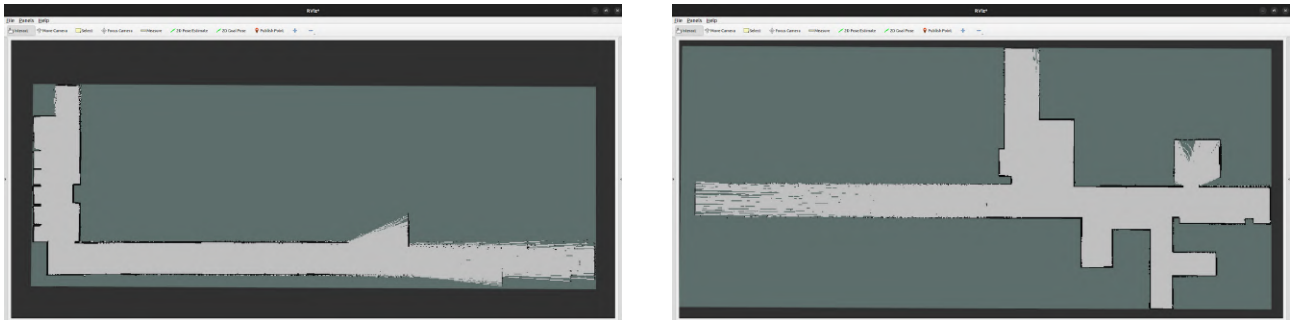
Despite these limitations, the resulting map accurately captured the main structure of the environment and confirmed the feasibility of deploying the proposed Active SLAM framework on real robotic hardware.

The observations obtained from this experiment further motivated the use of a collaborative multi-robot strategy. By distributing the exploration task among multiple robots, the distance traveled by each robot is reduced, which limits odometry drift and improves the overall robustness of the exploration process.

Multi-Robot Simulation Results

The first evaluation was performed in the Isaac Sim environment using two robots operating simultaneously. During the exploration process, each robot generated its own local map while navigating through the environment using the proposed Active SLAM framework.

Figure 6.3 shows the individual maps generated by each robot during the simulation experiment. Since the exploration graph was partitioned between the robots, each robot explored a different region of the environment.



(a) Robot 1 map (b) Robot 2 map

Figure 6.3: Individual maps generated by the two robots in the simulation environment

The local maps are then merged using the proposed map merging framework to generate a global representation of the environment. Figure 6.4 shows the final merged map.



Figure 6.4: Merged map generated in simulation

The resulting map successfully reconstructs the environment geometry and preserves the main structural features of the simulated world. Both robots completed their assigned exploration trajectories and reached all navigation goals generated from the prior graph.

To quantitatively evaluate the map quality, the merged map was compared with the ground-truth map extracted from the simulation environment. Figure 6.5 shows the overlap between the generated map and the reference map after registration.

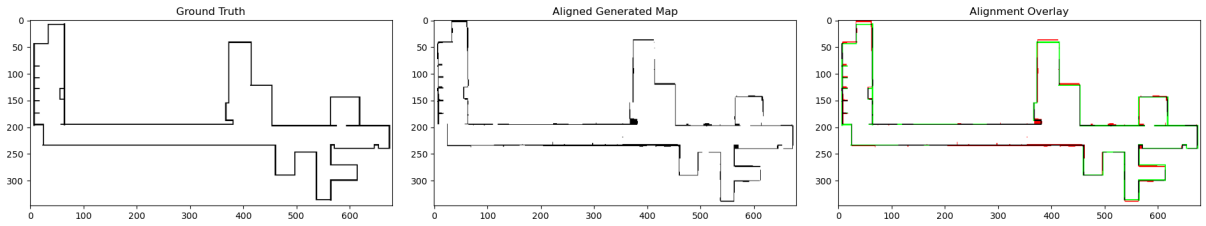


Figure 6.5: Overlap between the generated map in the simulation and the ground truth map, where the green indicates the ground truth and the red the generated map with slam

The obtained metrics are summarized in Table 6.1.

Table 6.1: SLAM evaluation metrics in simulation

Metric	**Value**	**Unit**
Correlation Similarity	49.80	%
Map Score	5428.00	px
SDF Score	4.86×10^5	px^2
Relaxed SDF Score	1.75×10^5	px^2

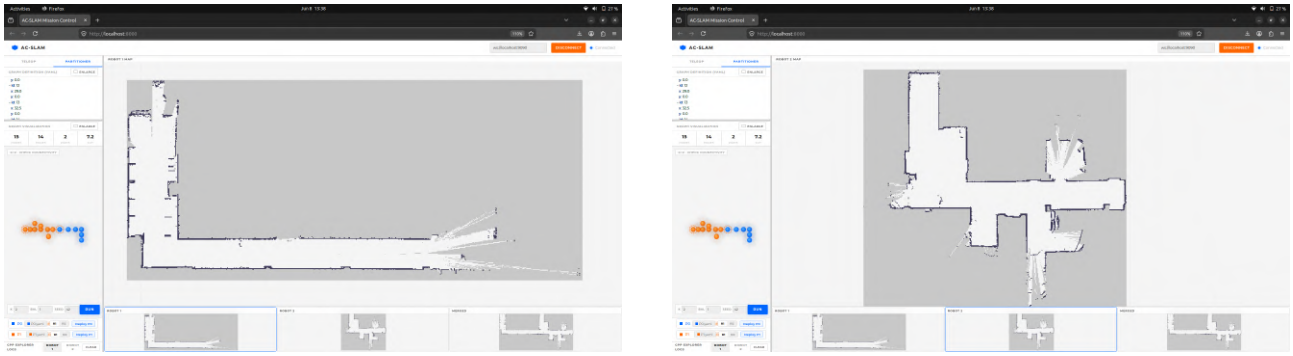
The quantitative results summarized in Table 6.1 demonstrate that the generated map closely aligns with the ground-truth simulation layout. Although the Correlation Similarity registers at 49.8%, this performance does not mean that the map is not usable. In grid mapping evaluations, pixel-wise cross-correlation metrics are heavily penalized by minor wall thickness discrepancies, distinct sensor occupancy probabilities, and sub-pixel shifts across extended boundaries. Even when two maps share an identical topological layout, minor pixel offsets along extensive parallel walls drastically lower the correlation coefficient.

For this reason, the Map Score and the SDF metrics provide a more descriptive assessment of structural fidelity. The low relative Map Score combined with the significant drop in the Relaxed SDF score indicates that macro-level environmental features were correctly captured, and that residual errors are merely artifacts of minor local alignment discretization rather than structural defects.

Multi-Robot Deployment Results

The same framework was then deployed on the real robotic platform. Similar to the simulation experiments, two robots explored the environment simultaneously while independently performing SLAM and navigation.

Figure 6.6 shows the local maps generated by the two robots during the real-world exploration experiment, while Figure 6.7 presents the final merged map.



(a) Robot 1 map

(b) Robot 2 map

Figure 6.6: Individual maps generated by the two robots in the real environment

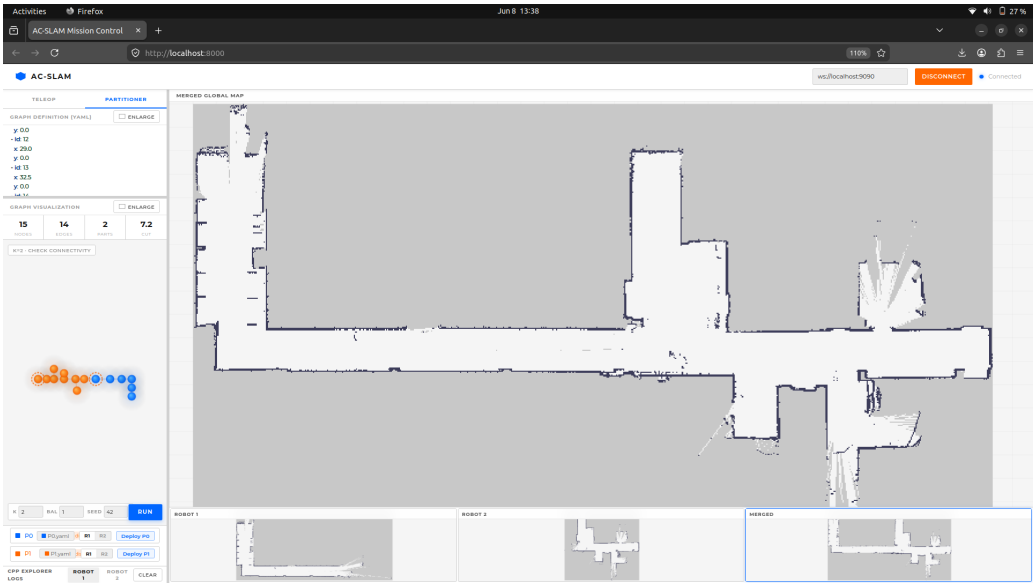


Figure 6.7: Merged map generated in the real environment

Unlike the simulation environment, the real-world experiments are affected by wheel slippage, floor irregularities, encoder noise and localization errors. Despite these challenges, both robots successfully completed their assigned exploration trajectories and generated a complete map of the environment.

The map merger was modified to take into account the initial positions of the robots and the distance separating them. This modification allowed the individual maps to be aligned correctly and produced a consistent global map.

Figure 6.8 shows the overlap between the generated map and the reference map constructed from the measured dimensions of the experimental environment.

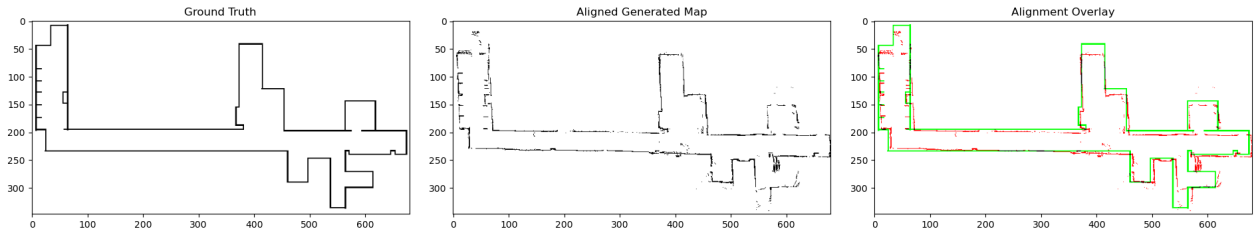


Figure 6.8: Overlap between the generated map in the real world and the ground truth, where the green indicates the ground truth and the red the generated map with slam

The obtained metrics are summarized in Table 6.2.

Table 6.2: SLAM evaluation metrics on real hardware

Metric	**Value**	**Unit**
Correlation Similarity	32.22	%
Map Score	8691.00	px
SDF Score	6.99×10^6	px^2
Relaxed SDF Score	5.55×10^6	px^2

The results presented in Table 6.2 confirm that the framework remains effective when transitioned to real hardware. As expected, a nominal degradation across all metrics is observed when compared to ideal simulation bounds.

The increase in the Map Score (8691) and the baseline SDF score (6.99×10^6 px^2) reflects the geometric distortions introduced by hardware uncertainties. In a physical environment, small angular errors from wheel slippage propagate over distance, thickening wall profiles and introducing rotational shifts.. The overlay in Figure 6.8 confirms that the topological structure remains accurate and safe for autonomous path planning.

6.2.3 Navigation Performance Evaluation

This subsection evaluates the performance of the navigation system during the exploration missions in both simulation and real-world environments. The objective is to verify that the robots can successfully reach the navigation goals generated from the prior graph while avoiding obstacles and adapting to changes in the environment.

Goal Reaching Performance

The navigation framework was evaluated by measuring the ability of the robots to reach the graph nodes assigned during the exploration process.

In simulation, all generated goals were successfully reached by both robots. The navigation stack was able to compute valid trajectories for all graph nodes, allowing the robots to complete the exploration mission without failures.

A similar behavior was observed during the real-world experiments. In the collaborative exploration scenario, both robots successfully reached all assigned goals and completed the exploration process. In contrast, during the single-robot experiment, two goals were not reached because localization drift caused the generated exploration points to gradually move outside the valid map area. As a result, no valid trajectory was computed and the corresponding goals were aborted.

These results demonstrate that the proposed multi-robot framework improves the robustness of the exploration process while reducing the impact of localization drift.

Path Following and Obstacle Avoidance

During the experiments, the robots successfully followed the trajectories generated by Nav2. The global planner generated collision-free paths toward the desired goals, while the local planner continuously produced velocity commands to track the path.

Increasing the goal tolerance proved beneficial for the exploration task. Since the objective is to generate a map rather than reach an exact final pose, relaxing the position and orientation constraints reduced oscillations near the goal and prevented unnecessary corrections.

Obstacle avoidance was also successfully validated. When obstacles were detected by the LiDAR, the local costmap was updated and Nav2 generated a new path around the obstacle. Figure 6.9 shows an example of trajectory replanning around an obstacle.

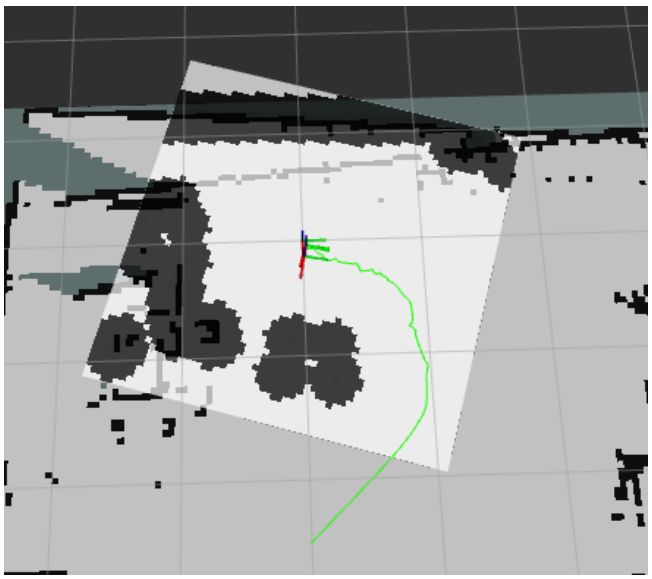


Figure 6.9: Navigation path replanning after obstacle detection

The robots were able to continue their exploration missions without collisions while maintaining smooth navigation behavior.

Recovery Behaviors

The recovery mechanisms provided by Nav2 were observed during the experiments. These behaviors are triggered when the robot cannot make progress toward the goal or when a valid trajectory cannot be generated.

For the omnidirectional Robotino platform, no recovery behaviors were required during the experiments. The robot was able to reach all assigned goals without becoming blocked or losing localization.

For the differential-drive robot, a recovery behavior was triggered once during the exploration process. This situation was mainly caused by wheel slippage and localization drift. The recovery action allowed the robot to recover its navigation state and continue the mission successfully.

The limited number of recovery events confirms the overall stability of the navigation system.

6.2.4 Comparative Analysis

Differential Drive vs Omnidirectional Navigation

The experiments also highlighted the differences between the differential-drive platform and the omnidirectional Robotino platform.

The differential-drive robot successfully completed the navigation tasks but required more parameter tuning and was more sensitive to wheel slippage and localization errors. Since the robot must rotate before moving in a new direction, navigating in narrow areas was sometimes more difficult. In some situations, the robot slightly overshot the goal position while performing turning maneuvers.

In addition, the computational limitations of the Raspberry Pi required reducing some Nav2 parameters to maintain stable operation. As a result, the robot occasionally exhibited a stop-and-go motion during navigation.

The omnidirectional Robotino platform demonstrated better overall navigation performance. Since the robot can move directly along both the x and y directions, orientation constraints are less critical and fewer corrective maneuvers are required. This allowed the robot to reach goals more smoothly and with fewer localization disturbances.

Furthermore, the omnidirectional platform navigated more efficiently in confined spaces and generally completed trajectories faster than the differential-drive robot.

Overall, the results indicate that both platforms are capable of performing autonomous exploration and navigation. However, the omnidirectional robot provides smoother motion, improved maneuverability and greater robustness in narrow environments, making it particularly suitable for Active SLAM applications.

Single vs Multi Robot

A comparison between the single-robot and multi-robot experiments highlights the advantages of distributing the exploration task among multiple robots.

In the single-robot configuration, the robot was responsible for exploring the entire environment while simultaneously performing SLAM and navigation. As the exploration progressed, localization errors accumulated due to odometry drift. These errors became more noticeable in long corridor sections, where the generated map gradually deviated from the expected geometry. As a result, the final exploration nodes were projected outside the valid free space of the map and two goals could not be reached.

In contrast, the collaborative exploration strategy divided the prior graph among two robots. Each robot was therefore responsible for exploring only a subset of the environment. This reduced the distance traveled by each robot and consequently reduced the accumulation of localization errors.

The most significant difference between the two approaches is the exploration completion rate. While the single robot failed to reach all assigned nodes, the multi-robot configuration successfully completed the entire exploration mission and reached all assigned graph nodes. This demonstrates that distributing the exploration workload improves the robustness of the system and reduces the impact of localization drift.

From a mapping perspective, the collaborative approach also produced a more complete representation of the environment. Since both robots explored different regions simultaneously, the generated local maps contained fewer distortions caused by long trajectories. After merging, the resulting global map provided a more consistent representation of the workspace.

Another important advantage of the multi-robot framework is fault tolerance. In the single-robot case, any localization degradation directly affects the entire exploration mission. In the collaborative configuration, the exploration process is distributed among several robots, reducing the impact of local navigation and localization errors on the overall system.

Although the multi-robot experiment was completed in less time than the single-robot experiment, a direct comparison of exploration duration is not entirely meaningful because the single robot did not complete all assigned goals. Therefore, exploration completion and successful goal execution provide a more representative comparison between the two approaches.

Overall, the experimental results demonstrate that the multi-robot framework provides better exploration completion, improved robustness against localization drift and a more reliable map generation process compared to the single-robot configuration.

Table 6.3: Single-robot versus multi-robot exploration performance

Metric	**Single Robot**	**Two Robots**
Assigned Goals Reached	All except 2	All goals reached
Exploration Completion	Partial	Complete
Localization Drift	Higher	Lower
Map Quality	Good	Better
Map Merging	Not Required	Successfully Performed

Simulation vs Real

A comparison between the simulation and real-world experiments highlights the impact of real operating conditions on the performance of the SLAM system. Although the same Active Collaborative SLAM framework was used in both cases, the real environment introduces additional sources of error that are not fully present in simulation.

From a qualitative perspective, both experiments produced complete maps of the environment. In simulation, the generated map closely matches the ground-truth map, as shown in Figure 6.5. Similarly, the real-world experiments produced a consistent global map after merging the local maps generated by both robots, as shown in Figure 6.8. In both cases, the robots successfully completed their exploration trajectories and reached all assigned graph nodes.

The quantitative results, however, reveal the effect of real-world uncertainties. The simulation experiment achieved a correlation similarity of 49.8 %, compared to 32.22 % on real hardware. This decrease is expected since the real environment introduces wheel slippage, encoder noise, floor irregularities, sensor measurement noise, and small localization errors. These effects accumulate over time and slightly distort the generated occupancy grid.

The same thing can be observed in the map score and metrics. The simulation map obtained a map score of 5428, whereas the real-world map obtained a map score of 8691. Although the results are higher for the real-world experiment, they remains relatively low, indicating that most structural features of the environment were reconstructed correctly.

The difference becomes more pronounced when examining the SDF-based metrics. The simulation experiment achieved an SDF score of 4.86×10^5, while the real-world experiment produced an SDF score of 6.99×10^6. The same behavior is observed for the relaxed SDF score. These larger values are mainly caused by small wall misalignments, localization drift, and slight geometric distortions introduced during exploration. Nevertheless, the overlap shown in Figure 6.8 confirms that the overall geometry of the environment was reconstructed successfully.

Overall, the simulation results provide an upper performance bound under ideal operating conditions, while the real-world experiments validate the robustness of the proposed framework in the presence of practical challenges. Although the real-world metrics are naturally lower than those obtained in simulation, the generated maps remain accurate and usable for autonomous navigation and exploration. These results demonstrate that the proposed Active Collaborative SLAM framework can be successfully transferred from simulation to real robotic platforms while maintaining reliable mapping performance.

6.3 Conclusion

This chapter presented a comprehensive evaluation of the proposed Active Collaborative SLAM framework in both simulation and real-world environments. The obtained results demonstrated the ability of the system to autonomously explore unknown environments, generate consistent occupancy maps and successfully navigate between exploration goals.

The SLAM evaluation showed that the generated maps closely matched the reference environments in both simulation and real-world experiments. Although the real-world results exhibited lower performance, the generated maps remained sufficiently accurate for autonomous navigation and exploration tasks.

The navigation experiments confirmed the reliability of the proposed framework. The robots successfully reached their assigned goals, avoided obstacles, and recovered from occasional navigation failures. The use of Nav2 provided stable path planning and obstacle avoidance capabilities throughout the exploration process.

The comparative analysis highlighted the advantages of collaborative exploration over single-robot operation. By distributing the exploration workload among multiple robots, the system achieved better exploration completion, reduced the impact of localization drift and generated more reliable maps. The experiments also showed the benefits of omnidirectional robots in terms of maneuverability and navigation efficiency when compared to differential-drive platforms.

Overall, the experimental results validate the effectiveness of the proposed Active Collaborative SLAM framework and demonstrate its successful deployment in both simulated and real robotic environments.

General Conclusion

This thesis has presented the design, implementation, and experimental validation of an Active Collaborative Simultaneous Localization and Mapping (AC-SLAM) framework for heterogeneous multi-agent mobile robot systems. Driven by the need to overcome the fundamental limitations of single-robot mapping—namely, cumulative localization drift, extended operational duration, and vulnerability to single points of failure—this work leveraged coarse environmental topological priors to orchestrate coordinated spatial exploration under severe communication constraints.

The proposed architectural pipeline successfully decoupled the multi-agent problem into three distinct, robust operational phases. In the pre-exploration phase, an environmental connectivity graph was utilized as an actionable spatial prior. By implementing a Breadth-First Search (BFS) graph expansion strategy, the workspace was partitioned into balanced, connected subgraphs, effectively isolating each agent's territory and eliminating the need for continuous inter-robot communication during active mapping. During the independent exploration phase, individual agents resolved the Chinese Postman Problem (CPP) over their assigned local topologies to compute optimal node-visitation paths. By establishing independent ROS2 navigation (`Nav2`) and mapping (`SLAM Toolbox`) pipelines inside fully isolated namespaces and transform trees (`tf`), parallel execution was achieved across a heterogeneous fleet comprising a custom-built differential-drive platform and a commercial holonomic Festo Robotino. Finally, a post-exploration map merging phase fused local occupancy grids into a structurally consistent global representation via a pairwise binary reduction tree utilizing known initial robot configurations. This entire workflow was monitored and orchestrated through a centralized, web-based Mission Control Platform that aggregated telemetry data and streamed real-time system logs via Server-Sent Events (SSE).

The framework was rigorously evaluated through parallel benchmarks inside the *NVIDIA Isaac Sim* photorealistic environment and physical deployments within the research facilities of the École Nationale Polytechnique (ENP). The experimental outcomes provided critical insights into multi-robot coordination under real hardware constraints:

- **Resilience to Odometric Drift:** While single-robot baseline configurations failed to achieve full spatial coverage due to severe open-loop wheel slippage and geometric degeneracy along uniform corridor structures, the collaborative strategy achieved a 100% exploration completion rate. Distributing the graph topology directly minimized individual robot travel distances, effectively keeping localization errors within safe bounds.
- **Metric vs. Topological Fidelity:** Quantitative evaluation against architectural floor plans yielded Normalized Cross-Correlation (NCC) scores of 49.80% in simulation and 32.22% on real hardware. While these absolute values reflect real-world sensor noise,

moving obstacles, and discretization artifacts along parallel boundaries, qualitative spatial overlays confirmed that the macro-level topological structure and free-space navigation corridors were preserved with high fidelity.

Despite these successful results, several technical limitations remain, offering fertile ground for future research directions:

1. **Environment-Adaptive Chinese Postman Routing:** The current exploration framework assumes a rigid, static prior connectivity graph. Future work will extend this pipeline into an online, adaptive CPP formulation. Under this approach, the routing algorithm will dynamically modify, expand, or repair the underlying graph structure in real time based on incoming sensor feedback. If the initial topological prior contains structural inaccuracies or geometric discrepancies, the algorithm will autonomously realign or regenerate missing nodes and edges to conform to the true layout of the physical environment, guaranteeing robust execution despite imperfect topological priors.
2. **Robust Feature-Based Grid Registration:** The post-exploration map merging pipeline is currently sensitive to initial pose calibration errors. Integrating an image-space registration step—such as an automated Hough Transform, Fourier-Mellin alignment, or ORB feature matching along overlapping grid boundaries—would allow the system to dynamically refine transformation matrices without relying on strict initial configuration constraints.
3. **Decentralized Peer-to-Peer Map Fusion:** Transitioning the map merging pipeline from a centralized web platform to an ad-hoc, decentralized peer-to-peer network protocol would completely eliminate central processing bottlenecks, allowing robots to asynchronously share and merge partial sub-maps whenever they encounter one another within communication range.

Open-Source Contributions and Reproducibility

To contribute in future research in multi-agent systems and ensure complete scientific reproducibility, the entire software stack developed over the course of this project has been fully open-sourced. The codebase is distributed across three dedicated public repositories, corresponding to the primary layers of our framework:

- **Simulation Stack:** https://github.com/Aymen-Jocef/Sim-AC-SLAM
- **Real Hardware Deployment:** https://github.com/Aymen-Jocef/Real-AC-SLAM
- **Mission Control Platform:** https://github.com/TheHiCore/Platform-AC-SLAM

In conclusion, the system developed in this thesis successfully demonstrates that anchoring active multi-robot exploration to coarse topological priors offers a reliable and communication-efficient solution for complex mapping tasks. The successful bridge from high-fidelity physics simulations to physical heterogeneous hardware validates the engineering viability of the architecture, providing a scalable baseline for future autonomous multi-agent industrial and inspection applications.

Bibliography

[1] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part I. *IEEE Robotics & Automation Magazine*, 13(2):99–110, June 2006.

[2] Muhammad Farhan Ahmed, Khayyam Masood, Vincent Fremont, and Isabelle Fantoni. Active SLAM: A Review on Last Decade. *Sensors*, 23(19):8097, September 2023.

[3] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, Jose Neira, Ian Reid, and John J. Leonard. Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age. *IEEE Transactions on Robotics*, 32(6):1309–1332, December 2016.

[4] Randall C. Smith and Peter Cheeseman. On the Representation and Estimation of Spatial Uncertainty. *The International Journal of Robotics Research*, 5(4):56–68, December 1986.

[5] H.F. Durrant-Whyte. Uncertain geometry in robotics. *IEEE Journal on Robotics and Automation*, 4(1):23–31, February 1988.

[6] J.J. Leonard and H.F. Durrant-Whyte. Mobile robot localization by tracking geometric beacons. *IEEE Transactions on Robotics and Automation*, 7(3):376–382, June 1991.

[7] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part II. *IEEE Robotics & Automation Magazine*, 13(3):108–117, September 2006.

[8] M.W.M.G. Dissanayake, P. Newman, S. Clark, H.F. Durrant-Whyte, and M. Csorba. A solution to the simultaneous localization and map building (SLAM) problem. *IEEE Transactions on Robotics and Automation*, 17(3):229–241, June 2001.

[9] S.B. Samsuri, Hairi Zamzuri, Mohd Azizi Abdul Rahman, Saiful Mazlan, and Abdul Hadi Abd Rahman. Computational cost analysis of extended Kalman filter in simultaneous localization and mapping (EKF-SLAM) problem for autonomous vehicle. 10:7764–7768, January 2015.

[10] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. FastSLAM: a factored solution to the simultaneous localization and mapping problem. In *Eighteenth national conference on Artificial intelligence*, pages 593–598, USA, July 2002. American Association for Artificial Intelligence.

[11] Georg Klein and David Murray. Parallel Tracking and Mapping for Small AR Workspaces. In *Proceedings of the 2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, ISMAR '07, pages 1–10, USA, November 2007. IEEE Computer Society.

[12] Anastasios I. Mourikis and Stergios I. Roumeliotis. A Multi-State Constraint Kalman Filter for Vision-aided Inertial Navigation. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 3565–3572, April 2007. ISSN: 1050-4729.

[13] G. Grisetti, C. Stachniss, and W. Burgard. Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 2432–2437, April 2005. ISSN: 1050-4729.

[14] E.B. Olson. Real-time correlative scan matching. In *2009 IEEE International Conference on Robotics and Automation*, pages 4387–4393, Kobe, May 2009. IEEE.

[15] 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*, pages 155–160, Kyoto, Japan, November 2011. IEEE.

[16] Xing Wei, Changchun Yang, Lingcheng Kong, and Peng Sun. Improved Hector-SLAM Algorithm Based on Data Fusion of LiDAR and IMU for a Wheeled Robot Working in Machining Workshop. In *2022 China Automation Congress (CAC)*, pages 2126–2131, Xiamen, China, November 2022. IEEE.

[17] Kaiwen Wang, Lammmert Kooistra, Ruoxi Pan, Wensheng Wang, and João Valente. UAV-based simultaneous localization and mapping in outdoor environments: A systematic scoping review. *Journal of Field Robotics*, 41(5):1617–1642, 2024. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rob.22325.

[18] Sebastian Thrun and Michael Montemerlo. The Graph SLAM Algorithm with Applications to Large-Scale Mapping of Urban Structures. *The International Journal of Robotics Research*, 25(5-6):403–429, May 2006.

[19] 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*, pages 22–29, October 2010. ISSN: 2153-0866.

[20] 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)*, pages 1271–1278, May 2016.

[21] Salah Bouhoun, Rabah Sadoun, and Mourad Adnane. OpenCL implementation of a SLAM system on an SoC-FPGA. *Journal of Systems Architecture*, 111:101825, December 2020.

[22] Rajaa Ali, Heba Hakim, and Mohammed Al-Ibadi. A Full Overview of Visual SLAM Algorithms. *Journal of Computer Science and Engineering Research*, pages 27–41, June 2025.

[23] Raul Mur-Artal, J. M. M. Montiel, and Juan D. Tardos. ORB-SLAM: A Versatile and Accurate Monocular SLAM System. *IEEE Transactions on Robotics*, 31(5):1147–1163, October 2015.

[24] Georges Younes, Douaa Khalil, John Zelek, and Daniel Asmar. H-SLAM: Hybrid direct–indirect visual SLAM. *Robotics and Autonomous Systems*, 179:104729, September 2024.

[25] Yingying Ran, Xiaobin Xu, Zhiying Tan, and Minzhou Luo. A Review of 2D Lidar SLAM Research. *Remote Sensing*, 17(7):1214, March 2025.

[26] Jiaxin Li, Huangying Zhan, Ben M. Chen, Ian Reid, and Gim Hee Lee. Deep learning for 2D scan matching and loop closure. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 763–768, Vancouver, BC, September 2017. IEEE.

[27] Zheming Wang and Xinguang Zhang. Artificial Intelligence-Driven Innovations in SLAM for Robotics and Autonomous Navigation. In *2025 9th International Conference on Robotics, Control and Automation (ICRCA)*, pages 156–159, March 2025.

[28] Junfu Qiao, Jinqin Guo, and Yongwei Li. Simultaneous localization and mapping (SLAM)-based robot localization and navigation algorithm. *Applied Water Science*, 14(7):151, June 2024.

[29] OpenSLAM: Home. https://openslam-org.github.io/.

[30] Cho Younggun. Awesome SLAM Datasets, May 2018. https://sites.google.com/view/awesome-slam-datasets/home.

[31] Wagner Tanaka Botelho, Maria Das Graças Bruno Marietto, Eduardo De Lima Mendes, Daniel Rodrigues De Sousa, Edson Pinheiro Pimentel, Vera Lúcia da Silva, and Tamires dos Santos. Toward an interdisciplinary integration between multi-agents systems and multi-robots systems: a case study. *The Knowledge Engineering Review*, 35:e35, January 2020.

[32] Jorge Rocha, Inês Boavida-Portugal, and Eduardo Gomes. Introductory Chapter. In *Multi-agent Systems*. IntechOpen, September 2017.

[33] Mikuláš Hajduk, Marek Sukop, and Matthias Haun. *Cognitive Multi-agent Systems*, volume 138 of *Studies in Systems, Decision and Control*. Springer International Publishing, Cham, 2019.

[34] Issam Matazi, Rochdi Messoussi, Salah-Eddine Bellmallem, Ilham Oumaira, Abdellah Bennane, and Raja Touahni. Development of Intelligent Multi-agents System for Collaborative e-learning Support. *Bulletin of Electrical Engineering and Informatics*, 7(2):294–305, June 2018.

[35] Davide Calvaresi, Kevin Appoggetti, Luca Lustrissimini, Mauro Marinoni, Paolo Sernani, Aldo Franco Dragoni, and Michael Schumacher. Multi-Agent Systems' Negotiation Protocols for Cyber-Physical Systems: Results from a Systematic Literature Review. January 2018.

[36] Dr. Anish Pandey. Mobile Robot Navigation and Obstacle Avoidance Techniques: A Review. *International Robotics & Automation Journal*, 2:1–12, May 2017.

[37] Murat Köseoğlu, Orkan Murat Çelik, and Ömer Pektaş. Design of an autonomous mobile robot based on ROS. In *2017 International Artificial Intelligence and Data Processing Symposium (IDAP)*, pages 1–5, September 2017.

[38] Spyros G. Tzafestas. Mobile Robot Control and Navigation: A Global Overview. *Journal of Intelligent & Robotic Systems*, 91(1):35–58, July 2018.

[39] Mary B. Alatise and Gerhard P. Hancke. A Review on Challenges of Autonomous Mobile Robot and Sensor Fusion Methods. *IEEE Access*, 8:39830–39846, 2020.

[40] Wei Zheng, Hong-Bin Wang, Zhi-Ming Zhang, Ning Li, and Peng-Heng Yin. Multi-layer Feed-forward Neural Network Deep Learning Control with Hybrid Position and Virtual-force Algorithm for Mobile Robot Obstacle Avoidance. *International Journal of Control, Automation and Systems*, 17(4):1007–1018, April 2019.

[41] Sebastian Thrun. A Probabilistic On-Line Mapping Algorithm for Teams of Mobile Robots. *The International Journal of Robotics Research*, 20(5):335–363, May 2001.

[42] A. Birk and S. Carpin. Merging Occupancy Grid Maps From Multiple Robots. *Proceedings of the IEEE*, 94(7):1384–1397, July 2006.

[43] Sooraj Sunil, Saeed Mozaffari, Rajmeet Singh, Behnam Shahrrava, and Shahpour Alirezaee. Feature-Based Occupancy Map-Merging for Collaborative SLAM. *Sensors*, 23(6):3114, January 2023.

[44] Sajad Saeedi, Michael Trentini, Mae Seto, and Howard Li. Multiple-Robot Simultaneous Localization and Mapping: A Review. *Journal of Field Robotics*, 33(1):3–46, 2016. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rob.21620.

[45] Xun S. Zhou and Stergios I. Roumeliotis. Multi-robot SLAM with Unknown Initial Correspondence: The Robot Rendezvous Case. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1785–1792, October 2006. ISSN: 2153-0866.

[46] D. Fox, J. Ko, K. Konolige, B. Limketkai, D. Schulz, and B. Stewart. Distributed Multi-robot Exploration and Mapping. *Proceedings of the IEEE*, 94(7):1325–1339, July 2006.

[47] Sajad Saeedi, Liam Paull, Michael Trentini, and Howard Li. Neural Network-Based Multiple Robot Simultaneous Localization and Mapping. *IEEE Transactions on Neural Networks*, 22(12):2376–2387, December 2011.

[48] A. Howard. Multi-robot mapping using manifold representations. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, volume 4, pages 4198–4203 Vol.4, April 2004. ISSN: 1050-4729.

[49] Andrew Howard. Multi-robot Simultaneous Localization and Mapping using Particle Filters. *The International Journal of Robotics Research*, 25(12):1243–1256, December 2006.

[50] Sebastian Thrun, Yufeng Liu, Daphne Koller, Andrew Y. Ng, Zoubin Ghahramani, and Hugh Durrant-Whyte. Simultaneous Localization and Mapping with Sparse Extended Information Filters. *The International Journal of Robotics Research*, 23(7-8):693–716, August 2004.

[51] Teresa A. Vidal-Calleja, Cyrille Berger, Joan Solá, and Simon Lacroix. Large scale multiple robot visual mapping with heterogeneous landmarks in semi-structured terrain. *Robotics and Autonomous Systems*, 59(9):654–674, September 2011.

[52] Been Kim, Michael Kaess, Luke Fletcher, John Leonard, Abraham Bachrach, Nicholas Roy, and Seth Teller. Multiple relative pose graphs for robust cooperative mapping. In *2010 IEEE International Conference on Robotics and Automation*, pages 3185–3192, May 2010. ISSN: 1050-4729.

[53] Keith YK Leung, Yoni Halpern, Timothy D Barfoot, and Hugh HT Liu. The UTIAS multi-robot cooperative localization and mapping dataset. *The International Journal of Robotics Research*, 30(8):969–974, July 2011.

[54] A. G. Barto and R. S. Sutton. Goal Seeking Components for Adaptive Intelligence: An Initial Assessment. Technical Report ADA101476, Massachusetts Univ., Amherst. Dept. of Computer and Information Science.; Air Force Wright Aeronautical Labs., Wright-Patterson AFB, OH., 1981. Num Pages: 521.

[55] Hans Jacob Sverdrup Feder. *Simultaneous stochastic mapping and localization.* Thesis, Massachusetts Institute of Technology, 1999. Accepted: 2005-08-22T18:09:40Z.

[56] Cindy Leung, Shoudong Huang, and Gamini Dissanayake. Active SLAM using Model Predictive Control and Attractor based Exploration. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5031, Beijing, China, October 2006. IEEE.

[57] Henry Carrillo, Ian Reid, and José A. Castellanos. On the comparison of uncertainty criteria for active SLAM. In *2012 IEEE International Conference on Robotics and Automation*, pages 2080–2087, May 2012. ISSN: 1050-4729.

[58] Henry Carrillo, Philip Dames, Vijay Kumar, and José A. Castellanos. Autonomous robotic exploration using occupancy grid maps and graph SLAM based on Shannon and Rényi Entropy. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 487–494, May 2015. ISSN: 1050-4729.

[59] Dario Mammolo. *Active SLAM in Crowded Environments.* PhD thesis, Autonomous Systems Lab, ETH Zurich, March 2019.

[60] Michael Kaess, Ananth Ranganathan, and Frank Dellaert. iSAM: Incremental Smoothing and Mapping. *IEEE Transactions on Robotics*, 24(6):1365–1378, December 2008.

[61] Alberto Soragna, Marco Baldini, Dominik Joho, Rainer Kümmerle, and Giorgio Grisetti. Active SLAM using Connectivity Graphs as Priors. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 340–346, Macau, China, November 2019. IEEE.

[62] Vadim Indelman. Towards multi-robot active collaborative state estimation via belief space planning. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4620–4626, September 2015.

[63] Zhaoyi Pei, Songhao Piao, Mohammed El Habib Souidi, Muhammad Zuhair Qadir, and Guo Li. SLAM for Humanoid Multi-Robot Active Cooperation Based on Relative Observation. *Sustainability*, 10(8):2946, August 2018.

[64] B. Yamauchi. A frontier-based approach for autonomous exploration. In *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97. 'Towards New Computational Principles for Robotics and Automation'*, pages 146–151, July 1997.

[65] Marie Ossenkopf, Gastón Castro, Facundo Pessacg, Kurt Geihs, and Pablo De Cristóforis. Long-Horizon Active SLAM system for multi-agent coordinated exploration. In *2019 European Conference on Mobile Robots (ECMR)*, pages 1–6, September 2019.

[66] Michail Kontitsis, Evangelos A. Theodorou, and Emanuel Todorov. Multi-robot active SLAM with relative entropy optimization. In *2013 American Control Conference*, pages 2757–2764, June 2013. ISSN: 2378-5861.

[67] Mecalux. SLAM in robotics: Industrial applications. https://www.mecalux.com/blog/slam-robotics.

[68] Peter Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. *AI Magazine*, 29:9–20, March 2008.

[69] The Amazon Robotics Family: Kiva, Pegasus, Xanthus, and more... - AllAboutLean.com, March 2020. Section: Industry 4.0.

[70] MEDBOT Medical Delivery Robot. https://www.servicerobotics.ai/richtech/medbot.html.

[71] Richtech Robotics Launches Elevator-Enabled Medbot: A Revolutionary Robot to Improve Healthcare Facility Efficiency. https://www.richtechrobotics.com/resources/medbot-launch.

[72] Giorgio Grisetti, Rainer Kümmerle, Cyrill Stachniss, and Wolfram Burgard. A Tutorial on Graph-Based SLAM. *IEEE Intelligent Transportation Systems Magazine*, 2(4):31–43, 2010.

[73] M Kwan. Graphic programming using odd or even points. 1(1962):110, 1962.

[74] Steve Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot Operating System 2: Design, Architecture, and Uses In The Wild. 2022.

[75] ROS 2 Documentation — ROS 2 Documentation: Humble documentation. https://docs.ros.org/en/humble/index.html.

[76] ROS: Home. https://ros.org/.

[77] rviz2 — RViz 1.9.0 documentation. https://docs.ros.org/en/humble/p/rviz2/.

[78] tf2 — ROS 2 Documentation: Humble documentation. https://docs.ros.org/en/humble/Tutorials/Intermediate/Tf2/Tf2-Main.html.

[79] Isaac Sim Omnigraph Tutorial — Isaac Sim Documentation. https://docs.isaacsim.omniverse.nvidia.com/4.5.0/omnigraph/omnigraph_tutorial.html.

[80] Steve Macenski and Ivona Jambrecic. SLAM Toolbox: SLAM for the dynamic world. *Journal of Open Source Software*, 6(61):2783, May 2021.

[81] SteveMacenski/slam_toolbox: Slam Toolbox for lifelong mapping and localization in potentially massive maps with ROS.

[82] Nav2 — Nav2 1.0.0 documentation. https://docs.nav2.org/.

[83] What is Docker?, 0530. https://docs.docker.com/get-started/docker-overview/.

[84] Dan Simon. The discrete-time Kalman filter. In *Optimal State Estimation: Kalman, H, and Nonlinear Approaches*, pages 121–148. John Wiley & Sons, Ltd, 2006. Section: 5 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/0470045345.ch5.

[85] Dan Simon. Nonlinear Kalman filtering. In *Optimal State Estimation: Kalman, H, and Nonlinear Approaches*, pages 393–431. John Wiley & Sons, Ltd, 2006. Section: 13 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/0470045345.ch13.

[86] Thomas Moore and Daniel Stouch. A Generalized Extended Kalman Filter Implementation for the Robot Operating System. In Emanuele Menegatti, Nathan Michael, Karsten Berns, and Hiroaki Yamaguchi, editors, *Intelligent Autonomous Systems 13*, pages 335–348, Cham, 2016. Springer International Publishing.

[87] RobotinoWiki. https://wiki.openrobotino.org/Main_Page.html.

[88] Rutvik Mehenge, Niket Satoskar, Anukool Shidhore, and N. Arshath. Kinematic and optimal design of a three-wheeled holonomic omnidirectional robot platform for mobile manipulation tasks. *Global Journal of Engineering and Technology Advances*, 17:045–054, October 2023.