POLITECNICO DI TORINO

Master's Degree in Mechanical Engineering

Master's Thesis

Validating Autonomous Ground Vehicle Navigation Across Diverse Environments: A Comprehensive Assessment of Nav2 Planners from Simulation to Hardware Implementation

Supervisors

Prof. Alessandro RIZZO

Orlando TOVAR ORDOÑEZ

Edoardo TODDE

Student

Arefeh AZAD

July 2025

Abstract

The continuous advancement of autonomous systems has significantly accelerated the evolution of industrial automation, aligning with the objectives of Industry 4.0. This thesis investigates the hardware implementation of navigation systems for Autonomous Ground Vehicles (AGVs), with a particular focus on Hardware-in-the-Loop (HIL) testing. The research employs state-of-the-art Simultaneous Localization and Mapping (SLAM) techniques and navigation algorithms within the Robot Operating System 2 (ROS2) framework, aiming to enhance AGVs' localization, mapping, and obstacle avoidance capabilities in both indoor and outdoor environments.

Multiple SLAM methods were explored, including SLAM Toolbox, Cartographer, and LIO-SAM, to identify the most effective algorithm for accurate mapping and localization. Additionally, 3D SLAM was employed to generate a three-dimensional map of the environment, particularly useful in scenarios with uneven or bumpy terrain where 2D SLAM may fail. The study evaluates multiple navigation algorithms based on the Nav2 stack controllers and planners to identify the most effective approach considering the AGV's wheel type, driving mechanism, maneuverability, and responsiveness. Simulation environments created in Gazebo were used to test and compare different controllers and planners, assessing their performance in navigating static and dynamic obstacles. Moreover, the integration of GPS technology further enhanced localization accuracy for large-scale applications.

The validated SLAM and path planning algorithms were successfully deployed on both the Scout and Bunker mobile robot models in simulation, with tests conducted in both office and outdoor environments. Additionally, autonomous navigation was implemented on the hardware of the Mini Bunker model, demonstrating its capabilities to navigate autonomously in real-world indoor and outdoor environments, marking a significant step toward practical AGV applications in industrial and agricultural settings.

To my family,
Thank you for all your love and support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Industry 4.0 has driven significant advancements in industrial automation, integrating smart technologies to improve manufacturing efficiency and adaptability. A key component of this shift is the use of autonomous mobile robots (AMRs), which enable flexible and dynamic operations in environments where traditional systems, such as Autonomous Ground Vehicles (AGVs), fall short. While AGVs depend on fixed infrastructure like magnetic tapes or predefined paths, AMRs use sensors and control algorithms to navigate autonomously. Central to this capability is Simultaneous Localization and Mapping (SLAM), a technique that allows robots to build maps of unknown environments while simultaneously localizing themselves. SLAM enables AMRs to operate safely, adapt to changes, and perform tasks in complex and dynamic industrial settings.

The versatility of AMRs has made them critical assets across industries, particularly in sectors like logistics and warehousing, where production layouts evolve rapidly. Equipped with LiDAR, depth-sensing cameras, and inertial measurement units, these robots dynamically detect obstacles, reroute paths, and perform tasks ranging from precision material transport to quality inspections. Their ability to function without infrastructure modifications makes them cost-effective solutions for industries aiming to meet evolving demands.

1.2 Contribution

This work is part of the FIXIT project, shown in Figure 1.1, led by Competence Industry Manufacturing 4.0 (CIM 4.0). The FIXIT system combines an AMR and an Unmanned Aerial Vehicle (UAV) to perform coordinated tasks. The AMR acts as a mobile base for the UAV, enabling collaborative operations such as inspections in hard-to-reach areas. A major challenge is ensuring reliable AMR navigation in unmapped or dynamic environments. To address this, we propose a SLAM solution implemented using the ROS 2 (Robot Operating System 2) framework to improve the AMR's ability to explore, map, and localize effectively while supporting UAV integration.

Figure 1.1: FIXIT Platform

In addition to the FIXIT project, this research extends to developing robust localization algorithms for AgileX's Bunker robot, a multi-purpose tracked platform designed for rugged environments. The Bunker robot features a high climbing ability, enabling it to navigate stairs and uneven terrain. Its adaptability makes it suitable for inspection, exploration, and agricultural applications. For outdoor localization in agricultural scenarios, we developed GPS-based localization algorithms.

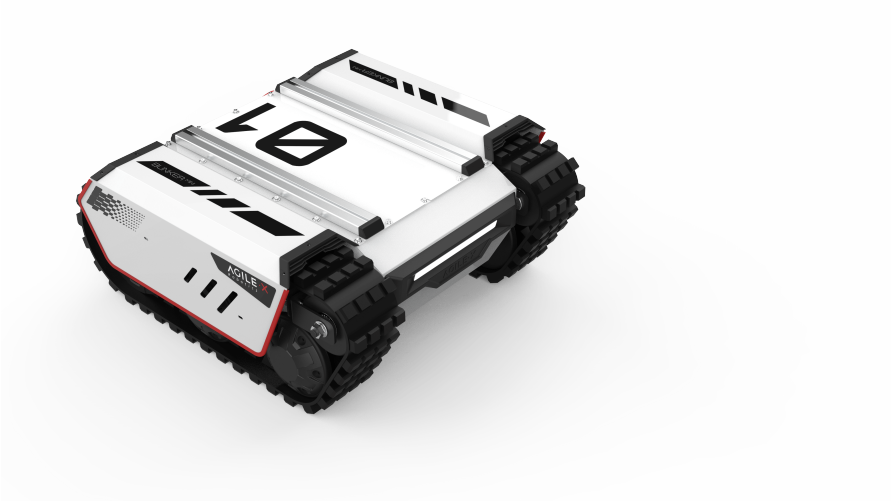


Figure 1.2: Bunker Mini AMR

ROS 2 serves as the unified framework for both projects, offering advantages such as real-time performance, security, and support for distributed systems. Using ROS 2's slam_toolbox and Nav2 packages, we implemented an Active SLAM system for the FIXIT AMR, combining mapping, localization, and adaptive path planning. For the Bunker robot, GPS-based localization was integrated with the Nav2 stack to ensure reliable outdoor navigation in agricultural environments. These systems were validated through simulations (Gazebo, RViz) and real-world testing on both platforms.

By pioneering advancements in AMR navigation and outdoor robotic systems, this work directly advances CIM 4.0's goals of intelligent, interconnected manufacturing and agricultural automation. The proposed solutions enhance operational efficiency, reduce downtime during reconfigurations, and improve adaptability across diverse environments. This paper details the theory and implementation of SLAM and GPS-assisted localization, as well as ROS 2's modular architecture, highlighting the role of mobile robotics in driving Industry 4.0 innovation.

Chapter 2

State of the Art

2.1 Overview

This chapter provides a comprehensive overview of the current state of the art in autonomous robot navigation, organized around the key components necessary for a robot to perceive, understand, and navigate effectively through complex environments. Beginning with the foundational task of SLAM, the discussion covers methods that allow robots to estimate their location and build reliable maps simultaneously. It then explores path planning and motion control, addressing the strategies robots employ to generate and follow safe and efficient trajectories using the maps and localization information obtained from SLAM. Additionally, the chapter highlights the emerging trends and integrated frameworks designed to enhance robustness, accuracy, and real-time adaptability in dynamic scenarios.

2.2 SLAM Foundations

SLAM algorithms must simultaneously estimate robot pose (localization) and construct or update a map of the environment (mapping). These tasks form a tightly coupled estimation problem, as localization accuracy depends on the map's precision, while mapping requires reliable pose estimates. Sensor noise, environmental dynamics, and computational constraints further complicate this challenge.

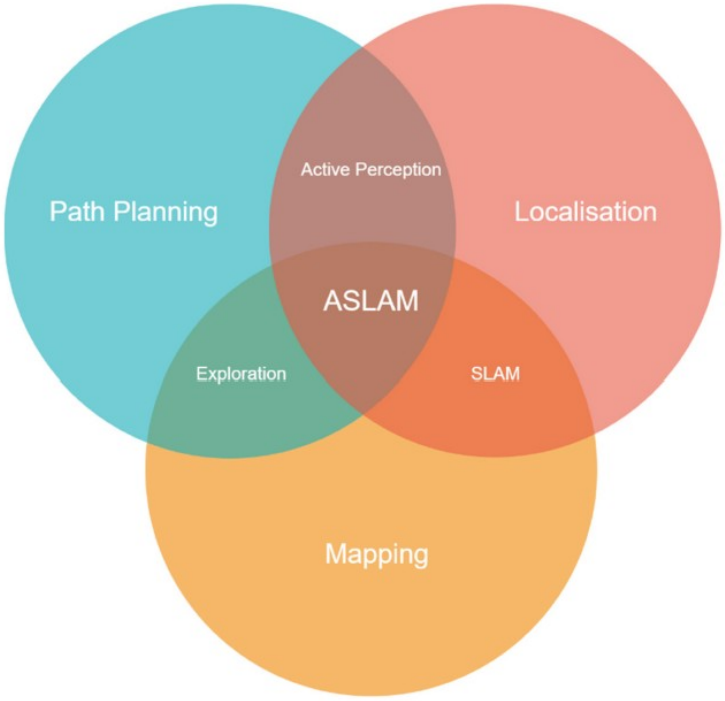


Figure 2.1: Outline of the SLAM algorithm [1]

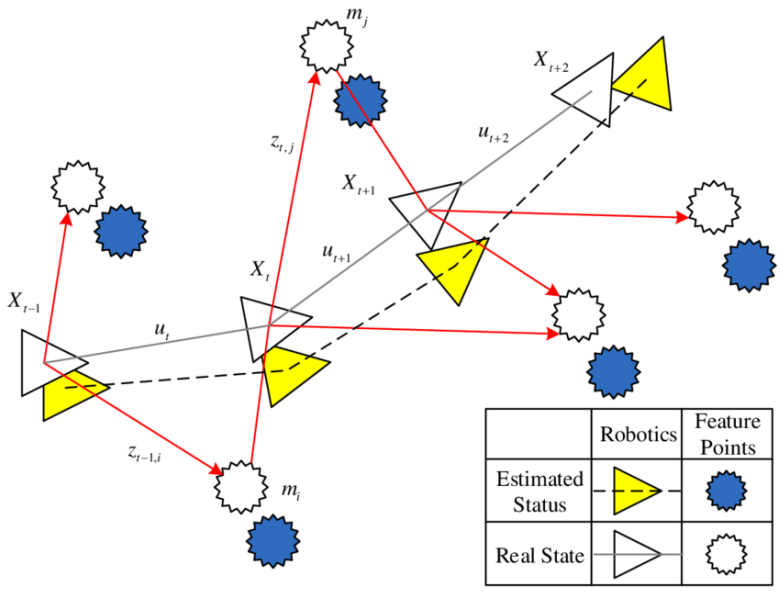


Figure 2.2: SLAM Problem Modeling Process [2]

2.2.1 Localization, Mapping, and Sensor Fusion

Localization

It aims to determine the robot's position and orientation within a coordinate system based on a known map, and the primary task is to track or refine the robot's pose. In SLAM, this is typically achieved through recursive Bayesian estimation:

$$p(\mathbf{x}_t|\mathbf{y}_{1:t}, \mathbf{u}_{1:t}) = \eta\, p(\mathbf{y}_t|\mathbf{x}_t) \int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t) p(\mathbf{x}_{t-1}|\mathbf{y}_{1:t-1}, \mathbf{u}_{1:t-1}) d\mathbf{x}_{t-1}, \tag{2.1}$$

where $\mathbf{x}_t$ are states, η is the Normalization constant, $\mathbf{y}_t$ are sensor measurements and $\mathbf{u}_t$ are control inputs.

Mapping

It involves constructing or refining an environmental model based on sensor observations across time. Common approaches include:

1. **Feature-Based Mapping:** Feature-based mapping is a technique where a map is constructed using distinct, recognizable features (landmarks) extracted from sensor data instead of dense occupancy grids or raw point clouds. These features, such as corners, edges, or key points from LiDAR or camera images, serve as reference points for localization and map updates. Feature-based mapping is particularly efficient in environments with well-defined landmarks, reducing memory and computational requirements compared to dense mapping approaches.

2. **Grid-Based Mapping:** Grid-based mapping represents the environment as a discrete occupancy grid, where each cell stores the probability of being occupied. Sensor measurements update the grid using Bayesian filtering or log-odds models, refining occupancy estimates over time. While memory-intensive, this method enables precise localization, path planning, and obstacle avoidance in robotics.

3. **Dense/Surface-Based Mapping:** Dense/Surface-Based Mapping represents environments as continuous surfaces using dense sensor data, such as LiDAR point clouds or RGB-D depth maps. Methods like TSDFs (Truncated Signed Distance Functions), voxel grids, and mesh reconstructions enable precise 3D modeling, facilitating high-accuracy navigation and manipulation. While computationally intensive, this approach excels in unstructured environments where fine surface details are critical, offering high-resolution maps for dense mapping or efficient surface-based representations for real-time performance. The choice between dense and surface-based methods depends on balancing detail and computational efficiency.

In SLAM, localization and mapping occur simultaneously, complicating matters because the map itself is initially unknown or partially known. To robustly link new sensor measurements to existing map elements, additional mechanisms for loop closure and data association are required.

Sensor Fusion

Modern SLAM systems integrate heterogeneous sensors to mitigate uncertainty and enhance robustness. Each sensor contributes unique information, enabling complementary data fusion:

- **Wheel Encoder:** Provides relative motion estimates. While computationally efficient, it accumulates drift over time due to integration errors and wheel slippage.
- **Inertial Measurement Unit:** Measures linear acceleration and angular velocity at high frequencies. IMU data is prone to bias and drift, requiring calibration and fusion with other sensors for long-term stability.
- **LiDAR or RGB-D Camera:** Captures precise depth information through time-of-flight (LiDAR) or structured light/depth sensing (RGB-D). These sensors enable geometry-based methods like iterative closest point (ICP) for scan alignment and dense 3D reconstruction.
- **Monocular/Stereo Camera:** Monocular cameras rely on feature extraction and structure-from-motion (SfM) for pose estimation but lack scale information without additional constraints. Stereo cameras provide direct scale estimation through triangulation, improving metric accuracy.

These measurements are fused within probabilistic frameworks such as Kalman filters, particle filters, or factor-graph optimization. These methods handle noise, partial observability, and sensor inconsistencies, ensuring accurate pose estimation and map reconstruction.

2.2.2 Components of SLAM

SLAM is typically divided into two main components called the **front-end** and the **back-end** as shown in Figure 2.3. Both are critical to the overall performance and accuracy of the system.

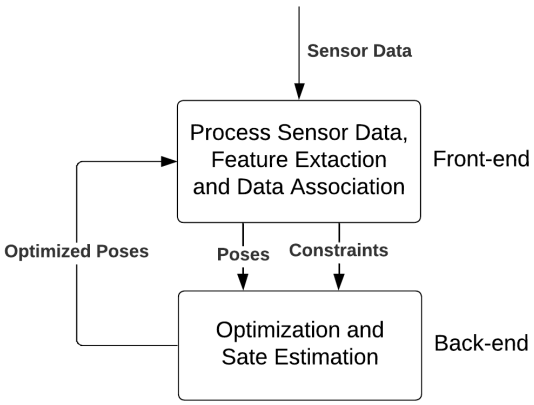


Figure 2.3: SLAM Structure

Front-End of SLAM

This component is responsible for processing raw sensor data and extracting meaningful information about the environment and the robot's pose. It focuses on real-time performance and local consistency, handling tasks such as feature extraction, data association, and odometry estimation.

- **Sensor Data Processing**: It begins by processing data from sensors such as LiDAR, cameras, and IMUs. Key tasks include:
  - **Feature Extraction**: The front-end identifies distinctive features in the environment, such as corners, edges, or planar surfaces. These features serve as landmarks for mapping and localization. For example, in visual SLAM, feature detectors like SIFT or ORB are used to extract keypoints from images.
  - **Data Association**: This step matches observed features with previously mapped landmarks. Correct data association is critical for maintaining map consistency. Techniques such as nearest-neighbor search or more advanced methods like joint compatibility branch and bound (JCBB) are often employed.
- **Odometry Estimation**: It computes the robot's relative motion using sensor data which can be achieved through:
  - **Wheel Odometry**: Uses encoder data from the robot's wheels to estimate motion. However, this method is prone to drift over time due to wheel slippage.
  - **Visual Odometry (VO)**: Estimates motion by analyzing the displacement of features in consecutive camera frames.
  - **LiDAR Odometry**: Uses point cloud data from LiDAR sensors to infer the robot's movement. Iterative Closest Point (ICP) is a common algorithm for this purpose.
  - **Inertial Odometry**: Integrates data from IMUs to estimate motion. While IMUs provide high-frequency data, they suffer from drift due to noise and bias.
- **Loop Closure Detection**: It identifies when the robot revisits a previously mapped area. This is crucial for correcting accumulated drift in the map and trajectory. Techniques include:
  - **Feature-Based Matching**: Compares current features with previously stored features to detect revisits.
  - **Appearance-Based Methods**: Uses place recognition algorithms, such as bag-of-words or deep learning-based approaches, to identify previously visited locations.

Back-End of SLAM

The back-end focuses on optimizing the map and the robot's trajectory to ensure global consistency. It addresses the errors and uncertainties introduced by the front-end, particularly drift and incorrect data associations.

- **Graph-Based Optimization**: It represents the SLAM problem as a graph, where nodes correspond to robot poses or landmarks, and edges represent constraints (e.g., odometry or loop closures). The goal is to find the configuration of the graph that minimizes the error across all constraints. This is typically formulated as a non-linear least squares problem:
- **Bundle Adjustment**:In visual SLAM, bundle adjustment refines the 3D structure of the map and the robot's trajectory by minimizing reprojection errors. This involves optimizing both the camera poses and the positions of the observed landmarks.
- **Global Consistency**: It ensures global consistency by incorporating loop closure constraints into the optimization process. Techniques such as **Pose Graph Optimization** and **Incremental Smoothing and Mapping (iSAM)** are commonly used to efficiently update the map and trajectory as new data arrives.
- **Uncertainty Management**:It models and propagates uncertainty in sensor measurements and robot poses. Probabilistic frameworks, such as Bayesian inference or covariance matrices, are used to represent and manage this uncertainty.

  Key Differences Between Front-End and Back-End

  - **Front-End**: Focuses on real-time processing, feature extraction, and local consistency. It is computationally lightweight but prone to drift and errors in data association.
  - **Back-End**: Focuses on global optimization, correcting drift, and ensuring long-term consistency. It is computationally intensive but significantly improves the accuracy and reliability of the SLAM system.

2.3 Probabilistic Approaches to SLAM

Noise in motion and sensor data require the use of probabilistic methods in SLAM to manage uncertainty and ensure consistent state and map estimates. These methods are broadly categorized into filter-based approaches and optimization-based techniques. Both rely on robust statistical frameworks to handle noise, partial observability, and non-linearities inherent in real-world environments.

2.3.1 Gaussian Filter–Based Approaches

Gaussian filters, such as the Kalman filter, estimate a robot's state by recursively applying prediction and update steps. The prediction step propagates the state estimate and its

covariance matrix using a motion model, while the update step refines the estimate using sensor measurements. The covariance matrix quantifies uncertainty in the state estimate, evolving based on process and measurement noise. Under Gaussian noise and linear dynamics, the Kalman filter provides an optimal and computationally efficient solution for real-time localization.

Extended Kalman Filter (EKF) SLAM

The EKF was foundational in early SLAM research and remains conceptually straightforward:

- **State Representation:** A state vector combines the robot pose and landmark positions.
- **Prediction Step:** The motion model (often non-linear) is linearized around the current estimate.
- **Update Step:** Sensor observations (e.g., landmark distances) are linearized, and the filter adjusts the state mean and covariance.

Pros: Real-time feasibility in small-scale maps and intuitive handling of correlated uncertainties.
Cons: Computational scalability issues (covariance matrix grows quadratically with landmarks) and potential divergence under strong non-linearities.

Unscented Kalman Filter (UKF) SLAM

The UKF improves upon the EKF by using sigma points to better capture non-linear behavior:

- **Pros:** More robust to strong non-linearities compared to the EKF.
- **Cons:** Still requires an $N \times N$ covariance matrix, limiting scalability for large maps.

2.3.2 Particle Filter–Based SLAM

Particle Filter-based SLAM estimates a robot's state and environment map by maintaining a set of pose hypotheses (particles), each representing a possible state. The particles are propagated using a motion model and weighted based on how well the predicted state matches sensor observations. Resampling selects particles with higher weights, improving the accuracy of the estimate. This method allows for robust performance in non-Gaussian or ambiguous environments, but can be computationally expensive due to the need for a large number of particles.

FastSLAM

FastSLAM [3] factorizes the SLAM problem into separate estimations for robot trajectory (using a particle filter) and landmarks (using individual EKFs or similar). This technique can be efficient, but the computational load grows with the number of landmarks and particles. Particle depletion and data association errors can also pose challenges in ambiguous or dynamic environments.

DP-SLAM

Dp-SLAM (Dynamic Pose SLAM) [4] extends traditional methods to dynamic environments by jointly estimating the robot's pose. It focuses on 2D occupancy grids, maintaining a distinct map for each particle while sharing common grid portions to limit memory overhead. This allows multiple map hypotheses to coexist, addressing ambiguity or noise in sensor data. Optimization is achieved through probabilistic frameworks like factor graphs or EM, balancing computational efficiency and robustness in dynamic, uncertain environments.

GMapping

GMapping [5] extends Particle Filter–based SLAM by using occupancy grid mapping to create a 2D map of the environment. Each particle represents a hypothesis of the robot's pose and maintains its own grid map, updated using sensor data such as laser scans. By leveraging the Rao-Blackwellized Particle Filter framework [6], GMapping decouples the estimation of the robot's trajectory from the map, reducing computational complexity. This approach allows GMapping to handle non-linearities and noise effectively, producing accurate maps in static environments. However, it assumes no dynamic objects and requires careful tuning of parameters like the number of particles and resampling strategies to balance performance and computational cost.

2.3.3 Graph Optimization–Based SLAM

Graph Optimization–Based SLAM models the robot's trajectory as a graph where nodes represent robot poses and edges encode spatial constraints derived from sensor data, such as odometry or landmark observations. The optimization process adjusts the poses to minimize errors in the graph, ensuring accurate localization and mapping. When the robot revisits prior locations, loop-closure edges are added to correct accumulated drift, enabling global consistency across the map. This approach is computationally efficient and scalable, making it well-suited for large-scale environments, as it focuses on refining the pose graph rather than processing raw sensor data repeatedly.

GraphSLAM

GraphSLAM is a classical implementation of graph optimization–based SLAM, where nodes represent both robot poses and map landmarks, and edges encode constraints from odometry and landmark observations. After collecting sufficient constraints, including

loop closures to address drift, it employs non-linear least squares optimization to find the configuration of poses and map elements that best satisfies all measurements. This batch optimization process ensures globally consistent solutions, making GraphSLAM particularly effective for offline applications requiring precise and consistent maps. While computationally intensive, it provides a comprehensive approach to achieving high-quality mapping and localization [7].

Optimization Frameworks

GraphSLAM and similar methods rely on optimization frameworks to solve the underlying non-linear least squares problem. Two widely used tools are:

- **g2o (general graph optimization):** A specialized framework for graph-based SLAM, providing efficient implementations of sparse linear solvers and robust kernels to handle outliers caused by incorrect loop closures or noisy measurements. It is optimized for pose graph and landmark-based SLAM, offering high performance for large-scale optimization tasks.
- **Ceres Solver:** A general-purpose non-linear least squares library widely used in computer vision (e.g., bundle adjustment) and adaptable to factor-graph SLAM. It supports a wide range of optimization problems, including those involving visual or LiDAR-based constraints, and is known for its flexibility and ease of use.

These frameworks enable efficient and scalable optimization of large graphs, ensuring globally consistent solutions for robot trajectories and maps.

Loop Closure

Loop closure is a critical mechanism in SLAM for correcting cumulative drift caused by incremental odometry errors. It detects when the robot revisits a previously mapped location and enforces spatial constraints between the current and historical poses, thereby optimizing the graph for global consistency. This process reduces uncertainty in the trajectory and map by aligning overlapping observations. However, robust loop closure faces significant challenges:

- **False Loop Closures:** Incorrectly associating dissimilar scenes introduces catastrophic distortions in the graph, requiring outlier rejection mechanisms or robust cost functions.
- **Computational Complexity:** Real-time loop detection in large-scale environments demands efficient place recognition, often achieved through scalable methods such as bag-of-words models, learned descriptors or submap-based indexing.
- **Dynamic Environments:** Transient or permanent scene changes (e.g., moving objects, lighting variations) degrade feature matching reliability. Solutions include leveraging persistent landmarks, temporal consistency checks, or semantic segmentation to filter dynamic elements.

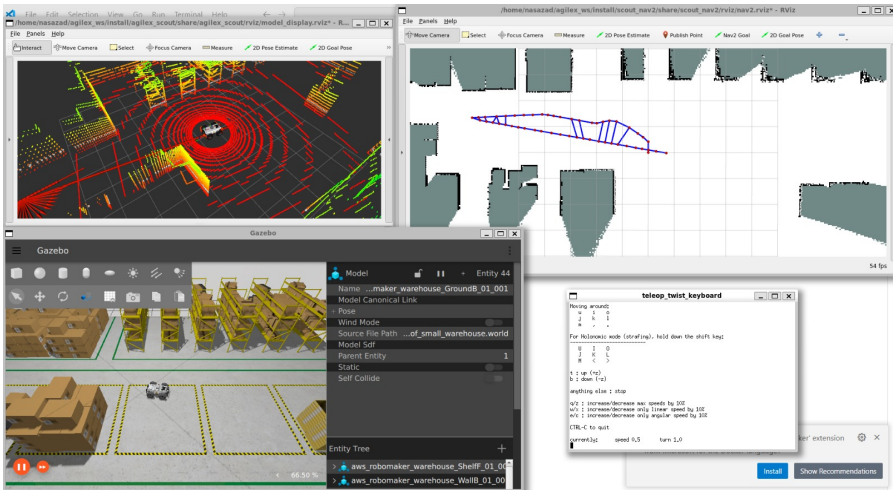


Figure 2.4: Loop closure of the AgileX Scout robot in simulation, enhancing map consistency.

2.4 Representative SLAM Algorithms

Modern SLAM frameworks often merge concepts from Gaussian filters, particle filters, and graph optimization, leveraging specialized sensors (e.g., LiDAR, IMU, camera arrays) to increase robustness and accuracy. This section details several noteworthy SLAM algorithms and their key design features, strengths, and limitations.

2.4.1 Hector SLAM

Hector SLAM is a 2D LiDAR-based SLAM system recognized for its high-frequency scan matching and independence from odometry. Originally developed for applications such as UAV indoor navigation and ground-based robots with unreliable wheel encoders, it uses real-time alignment of new scans with a continuously updated occupancy grid.

- **Core Principle:** Hector SLAM applies a Gauss-Newton approach directly to the gradient of the map for scan matching. Each incoming laser scan is iteratively aligned against the existing grid, estimating the robot's pose at a rate often exceeding standard filter-based methods.
- **Advantages:**
  1. **High Update Rates:** By directly matching scans to the grid, Hector SLAM can operate at the scanning frequency of modern LiDARs (e.g., 10–40 Hz). This makes it suitable for fast-moving robots or platforms where instantaneous pose estimates are crucial.
  2. **Odometry Independence:** Many SLAM methods use odometry as a prior; Hector SLAM does not rely on it, making it attractive for vehicles with poor or no wheel encoder data.

- **Limitations:**

  1. **2D Restriction:** Hector SLAM assumes a planar environment. While suitable for indoor spaces or flat outdoor terrains, it cannot handle significant variations in elevation without modifications.
  2. **No Global Pose Graph:** Lacking an explicit pose graph or loop-closure mechanism, large-scale drift corrections are limited. Over extensive mapping sessions, small alignment errors can accumulate if the environment lacks rich geometric features.

2.4.2 Cartographer

Cartographer [8] is a state-of-the-art SLAM system designed for high-precision mapping in both 2D and 3D environments. It combines a local scan-matching module with a global pose-graph optimization framework to achieve robust loop closure and drift correction. Cartographer supports a variety of sensor configurations, including 2D and 3D LiDARs and IMU data, making it highly adaptable to different applications. Its modular architecture and efficient computational design make it suitable for use in robotics, autonomous vehicles, and augmented reality.

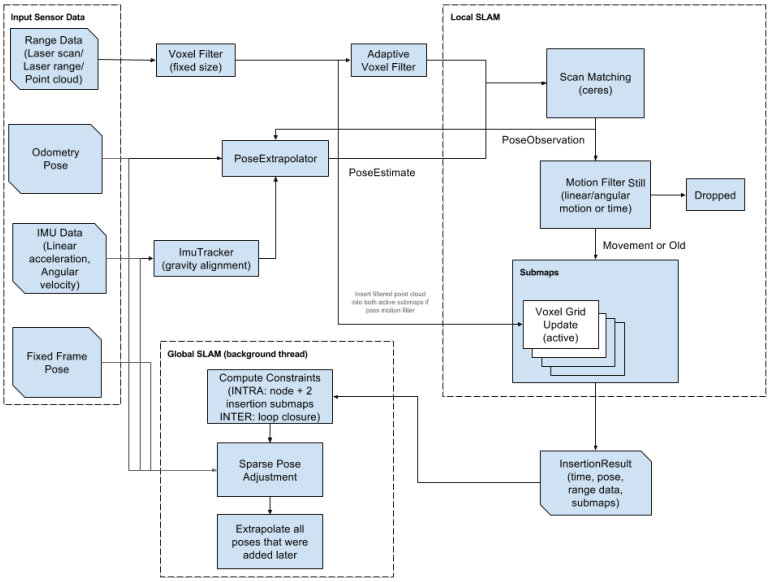


Figure 2.5: Cartographer Flowchart: A high-level overview of the Cartographer SLAM pipeline, illustrating the integration of local SLAM, global SLAM, and multi-sensor fusion.

Local SLAM

The local SLAM module in Cartographer focuses on real-time pose estimation and submap construction. It operates on a smaller scale, ensuring low-latency feedback for robot control while maintaining accuracy.Incoming LiDAR scans are incrementally inserted

into a local submap, which represents a small section of the environment. The system employs a probabilistic scan-matching algorithm, such as Ceres Scan Matcher, to align the current scan with the submap. This process refines the robot's pose estimate in real-time, ensuring accurate local mapping even in dynamic environments. By processing data in small submaps, Cartographer minimizes the computational complexity of scan matching. This approach reduces latency and ensures efficient operation on resource-constrained hardware. The local nature of the submaps allows for quick feedback, which is critical for real-time robot navigation and obstacle avoidance.

Global SLAM

The global SLAM module addresses long-term drift and ensures global consistency of the map by detecting and correcting loop closures.Cartographer periodically checks for overlaps between the current submap and previously constructed submaps. When an overlap is detected, the system adds a loop-closure edge to the global pose graph, which represents a spatial constraint between the two submaps. Once a loop closure is identified, Cartographer performs a non-linear optimization on the global pose graph using algorithms such as the Ceres Solver. This optimization adjusts all poses in the graph to minimize the error introduced by drift, resulting in a globally consistent map.

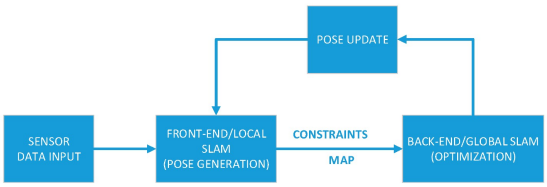


Figure 2.6: Local and Global SLAM interaction [8].

Advantages

- **Multi-Sensor Fusion:** Cartographer's ability to integrate data from multiple sensors, such as LiDARs and IMUs, significantly enhances its robustness. IMU data is particularly useful for compensating for motion distortions in LiDAR scans, especially during rapid movements or in environments with uneven terrain.

- **High-Quality Mapping:** The combination of local and global SLAM techniques enables Cartographer to produce highly accurate and globally consistent maps. Unlike systems that rely solely on local scan matching, Cartographer's global optimization ensures that long-term drift is corrected, resulting in maps that are reliable for navigation and planning.

- **Scalability:** Cartographer is designed to handle large-scale environments efficiently. Its hierarchical approach, which combines local submaps with global optimization, allows it to scale to complex and expansive environments without significant degradation in performance.

Limitations

- **Parameter Sensitivity:** The performance of Cartographer is highly dependent on the choice of parameters, such as submap size, scan-matching thresholds, and loop-closure detection criteria. Selecting optimal parameters can be challenging and often requires tuning based on the specific environment and application.
- **Overlap Requirements:** Successful loop closure in Cartographer relies on sufficient overlap between consecutive submaps. In environments with long, featureless corridors or repetitive structures, achieving this overlap can be difficult, leading to missed loop closures and increased drift over time.
- **Computational Complexity:** While Cartographer is designed to be efficient, the global pose-graph optimization process can be computationally demanding, particularly in large-scale environments. This may require powerful hardware to maintain real-time performance.

2.4.3 Karto SLAM

Karto SLAM is a 2D mapping framework that constructs a pose graph from odometry and laser scan matching. It is known for its simplicity and robustness, making it a popular choice for indoor robotics applications. Karto SLAM's loop closure detection mechanism enables it to correct accumulated drift, ensuring globally consistent maps.

Algorithmic Flow

Karto SLAM operates through a structured pipeline that combines odometry data, laser scan matching, and graph optimization to build a consistent map of the environment.

1. **Node Creation:** As the robot moves, Karto SLAM dynamically adds nodes to the pose graph. A new node is created whenever the robot travels a predefined distance or rotates by a certain angle. Each node is associated with a laser scan captured at that pose, which serves as a snapshot of the environment at that location.
2. **Edge Formation:** Edges in the pose graph represent constraints between nodes. Consecutive nodes are connected by edges based on odometry data, which provides an initial estimate of the relative transformation between poses. Additionally, when the system detects a loop closure, new edges are added to the graph to enforce consistency between the revisited location and its previous representation.
3. **Scan Matching:** Karto SLAM uses a highly optimized scan-matching algorithm to align laser scans with the existing map. This process refines the robot's pose estimate by minimizing the discrepancy between the current scan and the map.
4. **Graph Optimization:** The pose graph is periodically optimized to minimize the error introduced by odometry drift and scan-matching inaccuracies. This optimization process adjusts the positions of all nodes in the graph to ensure global consistency.

Advantages

- **Simplicity and Robustness:** Karto SLAM is known for its straightforward implementation and ease of use. Its algorithmic design is relatively simple compared to more complex SLAM frameworks, making it accessible to users with varying levels of expertise. Despite its simplicity, Karto SLAM delivers robust performance in standard indoor environments.
- **Efficient Scan Matching:** The scan-matching algorithm used by Karto SLAM is highly efficient, enabling real-time operation even on low-power hardware. This efficiency is particularly important for applications requiring rapid feedback, such as autonomous navigation and obstacle avoidance.
- **Drift Correction through Loop Closure:** Karto SLAM's loop closure detection mechanism effectively corrects accumulated drift, ensuring that the resulting map is globally consistent.

Limitations

- **2D Focus:** Karto SLAM is primarily designed for 2D mapping, which limits its applicability in environments requiring 3D spatial understanding. Extending Karto SLAM to 3D mapping is non-trivial and often requires significant modifications or integration with external libraries.
- **High-Speed Motion:** The performance of Karto SLAM can degrade in scenarios involving high-speed robot motion. Rapid movement can cause distortions in laser scans, making it difficult for the scan-matching algorithm to accurately align scans.
- **Feature-Sparse Environments:** In environments with limited distinctive features, Karto SLAM may struggle to detect loop closures. This can lead to increased drift and reduced map accuracy over time.
- **Parameter Sensitivity:** Karto SLAM's performance is sensitive to the choice of parameters, such as the distance and angle thresholds for node creation, scan-matching parameters, and loop-closure detection criteria. Tuning these parameters for optimal performance can be challenging.

2.4.4 LOAM (LiDAR Odometry and Mapping)

LOAM (LiDAR Odometry and Mapping) [9] is a 3D LiDAR-based SLAM framework that decouples the SLAM process into two parallel modules: *odometry* and *mapping*. This separation allows LOAM to achieve real-time performance while maintaining high accuracy. The odometry module operates at a high frequency to provide fast but approximate pose estimates, while the mapping module runs at a lower frequency to refine these estimates and build a globally consistent 3D map. LOAM is particularly well-suited for applications involving 3D LiDARs, such as autonomous vehicles, drones, and mobile robots operating in complex environments.

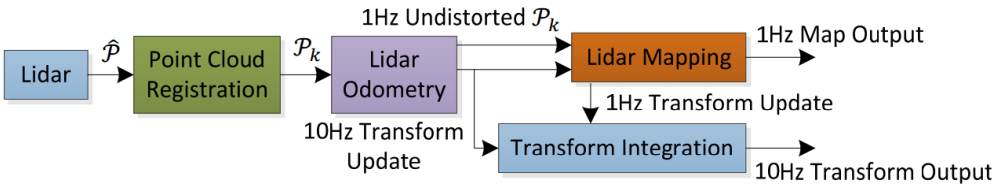


Figure 2.7: LOAM Flowchart: A high-level overview of the LOAM SLAM pipeline

LiDAR Odometry (High-Frequency)

The high-frequency odometry module is responsible for real-time motion estimation using incoming 3D LiDAR scans and performs the following tasks:

1. **Feature Extraction:** Distinctive geometric features are extracted from each 3D LiDAR scan, such as sharp edges and planar surfaces. These features are selected based on their curvature, with edges representing high-curvature points and planar surfaces representing low-curvature regions. This process reduces the computational complexity of scan matching by keeping track of the most informative parts of the environment.

2. **Incremental Motion Estimation:** Consecutive LiDAR scans are aligned by matching their extracted features, providing an incremental estimate of the robot's motion between scans. While this estimate is fast and suitable for real-time applications, it is slightly approximate due to the high-frequency nature of the odometry module. The output of this module is used for immediate feedback in robot control and navigation.

LiDAR Mapping (Low-Frequency)

The low-frequency mapping module refines the pose estimates generated by the odometry module and builds a globally consistent 3D map and performs the following tasks:

1. **Global Refinement:** The mapping module operates at a slower rate, allowing it to perform more computationally intensive optimization steps. It refines the robot's pose estimates over a larger time window by aligning the current scan with the global map. This process corrects any drift accumulated by the high-frequency odometry module, ensuring long-term accuracy.

2. **Consistent 3D Map:** The refined poses are used to update the global map, which consists of the environment's geometric features. The resulting map is highly accurate and suitable for tasks such as path planning, obstacle avoidance, and localization.

Advantages

- **Real-Time 3D Performance:** LOAM is highly efficient and capable of handling modern 3D LiDAR data rates (e.g., Velodyne, Ouster) on standard CPUs. Its decoupled architecture ensures real-time performance without sacrificing accuracy.

- **High Accuracy:** The combination of high-frequency odometry and low-frequency mapping effectively mitigates drift over long distances. This two-level design makes LOAM one of the most accurate 3D LiDAR-based SLAM frameworks available.
- **Scalability:** LOAM's modular design allows it to scale to large and complex environments, making it suitable for applications such as autonomous driving and large-scale mapping.

Limitations

- **Reliance on Geometric Features:** LOAM's performance depends on the availability of distinct geometric features in the environment. In sparse or uniform environments (e.g., featureless tunnels or open fields), the lack of features can weaken the scan-matching process, leading to increased drift.
- **Computational Overheads:** Although LOAM is well-optimized, its mapping module can be computationally intensive, particularly in large-scale environments. This may require careful resource management on embedded systems or lower-end CPUs.

2.4.5 LIO-SAM (LiDAR Inertial Odometry with State Augmentation)

LIO-SAM [10] extends the LOAM framework by integrating IMU data into a factor graph-based SLAM system. It corrects motion distortion in LiDAR scans and enhances robustness in dynamic environments by leveraging IMU preintegration and loop closure constraints.

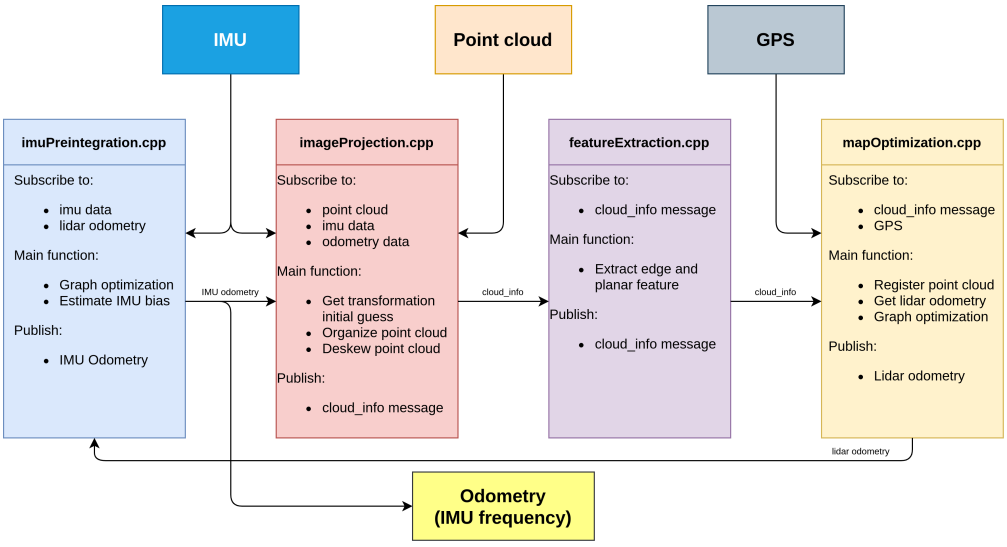


Figure 2.8: LIO-SAM Flowchart: A high-level overview of the LIO-SAM SLAM pipeline

Algorithmic Details

- **Factor Graph Architecture:** LIO-SAM constructs a factor graph where nodes represent state variables, including robot pose, velocity, and IMU biases. Edges encode constraints between states, derived from IMU preintegration, LiDAR odometry, and loop closure detections.
- **Key Factors in the Graph:**
  - **IMU Preintegration Factor:** IMU data between LiDAR frames is accumulated to estimate relative motion, reducing drift and improving state estimation.
  - **LiDAR Odometry Factor:** LiDAR scan matching constraints are incorporated to refine pose estimates.
  - **Loop Closure Factor:** ScanContext descriptors detect loop closures, which are refined using ICP before being added to the factor graph to maintain map consistency.
- **Motion Compensation** IMU angular rates and linear acceleration are used to correct for motion distortion in LiDAR scans. Each point cloud is transformed based on interpolated motion estimates.
- **Sliding Window Marginalization** To ensure computational efficiency, LIO-SAM maintains a limited history of recent poses and marginalizes older states using mathematical optimization techniques, reducing the size of the problem without losing significant information.

Advantages

- **Tight Sensor Fusion:** IMU integration enhances robustness in environments with rapid motion, such as UAV navigation.
- **Global Consistency:** The factor graph framework and loop closures prevent drift over long trajectories.
- **Multi-Sensor Flexibility:** GPS data can be incorporated for large-scale outdoor mapping applications.

Limitations

- **Computational Overhead:** Optimization-based methods require significant processing power.
- **Parameter Sensitivity:** Proper tuning of noise models and extrinsic calibration is necessary for optimal performance.
- **Feature Dependency:** Performance may degrade in environments with sparse geometric features.
- **Initialization Requirements:** Precise calibration of the LiDAR-IMU transform is essential for reliable operation.

2.4.6 FAST-LIO (Fast Lightweight LiDAR-Inertial Odometry)

FAST-LIO [11] is designed for real-time SLAM on resource-constrained platforms. It utilizes an iterated Kalman filter (IKF) to achieve high-frequency state estimation, making it suitable for applications like UAVs and small robots.

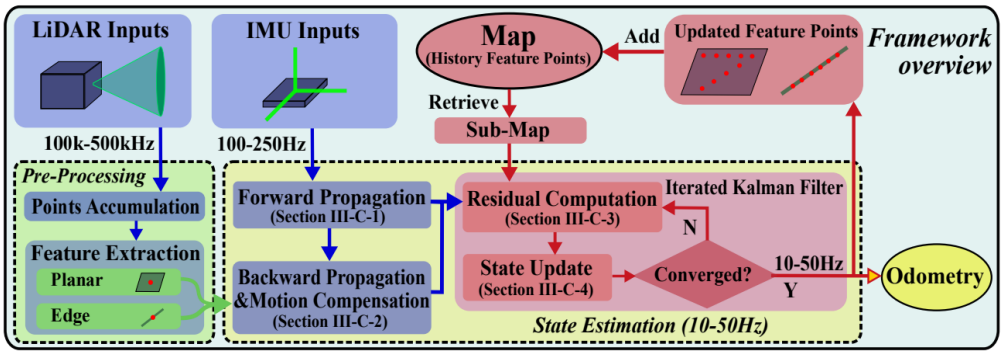


Figure 2.9: FAST-LIO Flowchart: A high-level overview of the FAST-LIO SLAM pipeline

Technical Innovations and Algorithmic Details

State Representation and Propagation FAST-LIO maintains a state vector consisting of pose, velocity, and IMU biases. IMU data is used to propagate the state forward in time between LiDAR updates.

Efficient Measurement Update The iterated Kalman filter (IKF) refines state estimates using LiDAR measurements, with automatic differentiation used to compute measurement Jacobians for improved accuracy.

Incremental k-D Tree (ikd-Tree) FAST-LIO employs an adaptive k-D tree structure to manage point cloud data efficiently. It allows for real-time insertion, deletion, and nearest-neighbor searches, optimizing map updates.

Forward-Backward Motion Compensation IMU-based motion compensation corrects for distortions in LiDAR scans by applying transformations in both forward and backward directions, ensuring accurate mapping.

Advantages

- **Computational Efficiency:** Runs at high frequencies with minimal processing power requirements.
- **Low Drift:** Achieves precise localization with minimal error accumulation.
- **Lightweight Design:** Optimized for embedded systems and small-scale robotics.

Limitations

- **Simplified State Model:** Does not account for advanced sensor biases, potentially reducing accuracy in dynamic environments.
- **Limited Loop Closure:** Lacks global map correction through loop closure detection.
- **Sparse Feature Use:** Reduces the number of extracted features to maintain computational efficiency, potentially affecting map resolution.
- **Initialization Sensitivity:** Requires precise initialization of LiDAR-IMU parameters for stable operation.

2.5 Path Planning

Once the robot acquires a reliable map and accurate localization from SLAM, it must plan a safe, kinematically feasible, and preferably optimal route to its goal. Path-planning is therefore divided into two tightly-coupled layers:

- **Global Path Planning (GPP)** — an *offline / low-frequency* process that searches the static (or slowly-changing) map for a collision–free *reference path*.
- **Local Path Planning (LPP)** — an *online / high-frequency* process that refines or entirely re-plans short segments of the reference path while the robot is moving to handle dynamics, moving obstacles, and actuation limits.

The remainder of this chapter details the algorithms, theoretical foundations, and practical trade-offs of each layer, followed by hybrid frameworks that bridge the two.

2.5.1 Global Path Planning (GPP)

A global planner computes a *reference path* from the current robot pose $\mathbf{x}_{\text{start}}$ to the goal pose $\mathbf{x}_{\text{goal}}$ on a static or slowly-changing map. These methods usually rely on an internal representation of the environment (e.g., occupancy grids, topological graphs).

Environment Representations

- **Multi-Layer Costmaps:** Global planners consume a costmap instance that fuses separate layers (static occupancy grid, inflation, semantic classes, elevation). Each layer publishes a cost $c_{ij} \in [0,255]$ where higher values encode traversal risk. A meta-layer applies user-defined cost fusion.

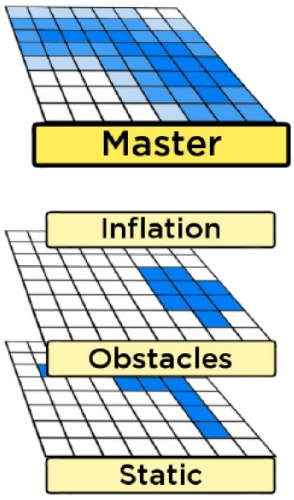


Figure 2.10: Layered Costmap

- **Traversability Grids:** Outdoor AGVs often replace binary occupancy with probabilistic traversability $p_{\text{free}} \in [0,1]$ derived from LiDAR roughness, RGB vegetation indices, or DEM slope, enabling planners to penalise soft soil or steep inclines without forbidding them outright.

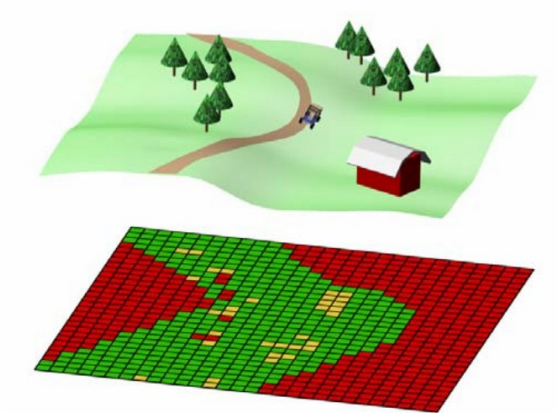


Figure 2.11: Traversability Grids [12]

- **Topological Graphs:** In very large facilities a metric map is first thinned into a topology (Voronoi graph or generalized Voronoi skeleton), allowing kilometre-scale queries.

Because the search must be rerun every time the goal changes—or the map is significantly updated—an effective GPP algorithm strikes a balance between **optimality**, **memory**,

and **runtime**. Three families dominate modern mobile-robot stacks and are reviewed below: graph-search, sampling-based, and meta-heuristic methods.

Graph Search

These methods model the environment as a graph, where nodes represent discrete states (e.g., grid cells, waypoints, or landmarks), and edges represent feasible transitions between states with associated movement costs. These methods search for an optimal or near-optimal path from the start node to the goal node.

Dijkstra's Algorithm: Dijkstra's algorithm is a best-first, *uniform-cost* search technique that finds the optimal path from a start vertex to every other reachable vertex—or to a specified goal—in a weighted graph with non-negative edge costs. Introduced by Edsger W. Dijkstra in 1959 [13], it remains a cornerstone of modern path-planning.

The algorithm selects the next vertex to expand using the evaluation function

$$f(n) \;=\; g(n), \tag{2.2}$$

where

- $g(n)$ is the exact path cost accumulated from the start vertex to n.

Because vertices are expanded in non-decreasing order of g, the first time the goal vertex is removed from the priority queue the path found is guaranteed to be optimal.

Complexity. With a binary-heap priority queue, the worst-case running time is $O((|V| + |E|) \log |V|)$; a Fibonacci heap lowers the bound to $O(|E| + |V| \log |V|)$ at the expense of larger constant factors. Memory usage is $O(|V|)$ for the distance array, predecessor map, and queue entries.

Advantages:

- **Optimality.** Always returns the least-cost path in graphs whose edges satisfy $c(e) \geq 0$.
- **Single-source solution.** A single run yields distances to *all* vertices, permitting inexpensive queries to multiple goals.
- **Simplicity and maturity.** Straightforward to implement and extensively studied, with many efficient variants such as Δ-stepping and bidirectional search.

Limitations:

- **Isotropic expansion.** The wavefront grows evenly in all directions, causing many unnecessary expansions in large open regions when only one goal is relevant.
- **Edge-weight restriction.** Cannot handle negative cost edges; algorithms like Bellman–Ford are required in such cases.
- **Priority-queue overhead.** For very dense graphs ($|E| \approx |V|^2$), all-pairs methods (e.g. Floyd–Warshall) may be simpler and comparable in cost.

A* Algorithm: A* is a best-first search algorithm that efficiently finds the optimal path from a start node to a goal node in a weighted graph. Introduced by Hart, Nilsson, and Raphael in 1968 [14], A* extends Dijkstra's algorithm by incorporating heuristic guidance, balancing exploration and exploitation.

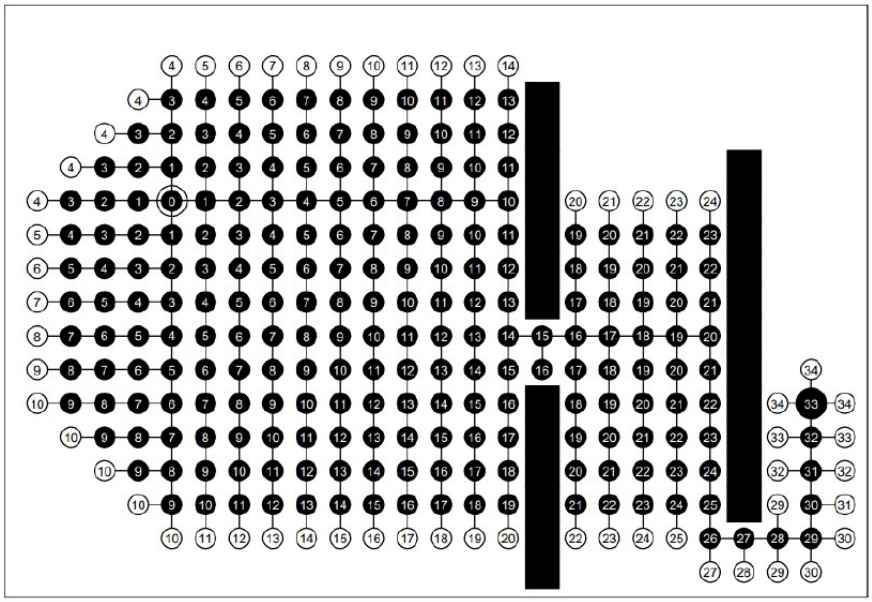


Figure 2.12: A* Algorithm [15]

This algorithm is one of the most widely used search methods in robotics. It selects the next node to expand based on the evaluation function:

$$f(n) = g(n) + h(n), \tag{2.3}$$

where:

- $g(n)$ is the exact cost from the start node to node n.
- $h(n)$ is a heuristic estimate of the cost from n to the goal.
- $f(n)$ represents the total estimated cost through n.

By integrating heuristic knowledge, A* reduces unnecessary exploration compared to Dijkstra's algorithm, which only considers $g(n)$. A heuristic is *admissible* if $h(n) \leq h^*(n)$, where $h^*(n)$ is the true minimal cost to achieve the goal. If the heuristic is *consistent* (i.e., for every node n and its successor n', $h(n) \leq c(n, n') + h(n')$, with $c(n, n')$ being the cost to move from n to n'), then A* guarantees optimality while efficiently guiding the search.

The performance of A* heavily depends on the choice of heuristic. A commonly used heuristic is the Euclidean distance for continuous spaces or the Manhattan distance for grid-based graphs. The time complexity of A* in the worst case is exponential, but in

well-structured graphs, it performs significantly better than uninformed search methods such as Dijkstra's algorithm.

Advantages:

- Guarantees optimality under admissible and consistent heuristics.
- Straightforward implementation on discrete grids or graphs.

Limitations:

- Can be computationally expensive in high-dimensional spaces and very large maps.
- Performance is highly sensitive to the choice and accuracy of the heuristic.

D* Algorithm: The D* (Dynamic A*) algorithm, introduced by Anthony Stentz in 1994 [16], is an extension of A* designed for environments where the cost of traversing edges may change dynamically. Unlike A*, which searches from the start node to the goal, D* performs a backward search from the goal towards the start. Each node maintains a backpointer indicating the next node in the optimal path, enabling efficient path updates.

D* assumes an initially unknown or partially known environment. It starts by planning an optimal path using a heuristic, assuming unknown areas are traversable. As the robot moves and acquires new information (such as detecting obstacles), it updates the cost of affected edges and propagates changes efficiently using a priority queue. If a new obstacle is discovered, D* efficiently replans the shortest path from the current position to the goal without recomputing the entire solution from scratch.

Therefore, for environments that may change during navigation, algorithms like D* and its variant D* Lite are more suitable. A simplified and memory-efficient version of D* exist called D* Lite which uses a priority queue-based approach, reducing the complexity of re-evaluations. Each node maintains two values: the current cost $g(n)$ and a one-step lookahead cost, often denoted as $rhs(n)$:

$$rhs(n) = \min_{n' \in succ(n)} \{c(n, n') + g(n')\}, \tag{2.4}$$

where $succ(n)$ represents the set of successor nodes of n. When the environment changes, only the affected portions of the graph are updated. D* and its variants are widely used in mobile robotics, particularly for real-time navigation in dynamic and unknown environments. Compared to A*, D* significantly reduces redundant computations by focusing updates only on affected regions of the graph.

Advantages:

- Efficient re-planning by incrementally updating the search space.
- Well-suited for scenarios where changes are localized (e.g., planetary rovers, indoor mobile robots).

Limitations:

- More complex to implement compared to standard A*.
- In rapidly changing or extensive maps, the computational overhead can be significant.

Hybrid A*

Concept. Hybrid A* embeds the vehicle heading into the state (x, y, θ) and expands motion primitives that satisfy non-holonomic constraints (Dubins or Reeds–Shepp arcs), producing curvature-bounded paths directly.

The planner first quantises the vehicle's heading into N_θ discrete sectors (typically 12–16); from each state it then explores only a compact set of forward and reverse motion primitives. Whenever an expanded state enters a small neighbourhood of the goal, it attempts a closed-form Dubins or Reeds–Shepp analytic shot, and if this shortcut is collision-free the search terminates immediately.

Advantages

- Generates kinematically feasible paths—no extra smoothing.
- Handles forward and reverse for skid-steer and car-like AGVs.

Limitations

- 5–10 × slower than planar A* on the same grid.
- Memory rises with the number of heading bins.

Sampling-Based Planning

Sampling-based methods are ideal for high-dimensional configuration spaces where grid discretization is computationally infeasible. These approaches randomly sample the free space and build structures (trees or graphs) that capture the underlying topology of the environment.

RRT / RRT* Rapidly-exploring Random Trees (RRT) build a tree rooted at the start configuration by randomly sampling the state space and extending the tree towards these samples, while simultaneously checking for collision. The RRT* variant refines this approach by incorporating a *rewiring* step that reconnects nodes when a lower-cost path is found. This results in:

- **Probabilistic Completeness:** Given enough samples, an RRT will almost surely find a path if one exists.
- **Asymptotic Optimality (RRT*):** As the number of samples tends to infinity, the cost of the solution converges to the optimum.

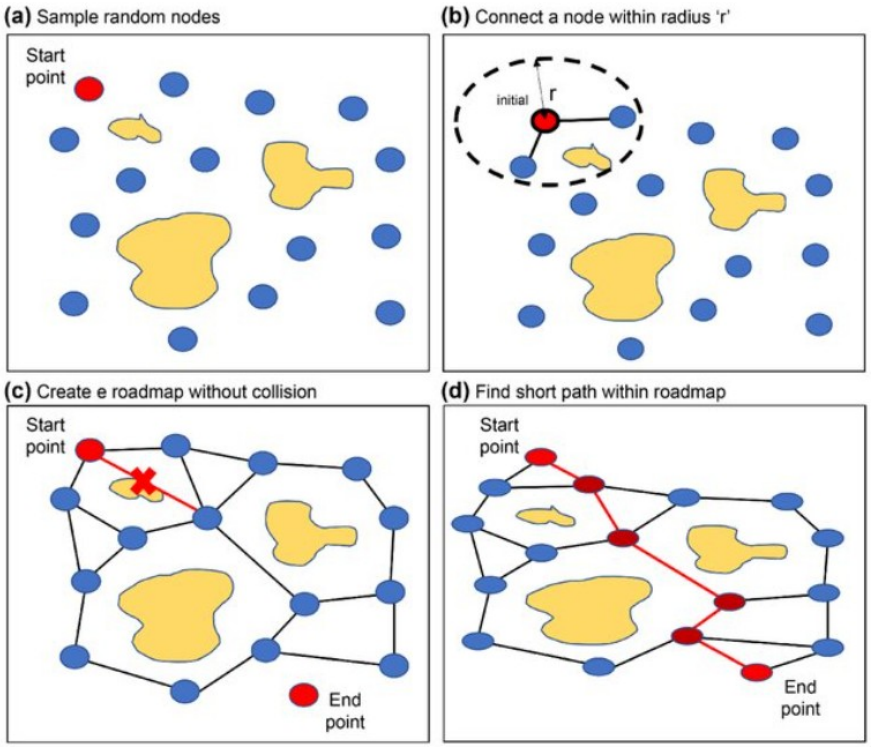


Figure 2.13: The PRM algorithm's search process: (a). Sample random nodes, (b) Connect a node within a radius, (c) Create a roadmap without collision, and (d) Find shortest path within the roadmap. [17]

Advantages:

- Handles high-dimensional spaces effectively.
- The sampling process can be parallelized.

Limitations:

- Initial paths are often suboptimal and require additional refinement.
- In environments with narrow passages, purely random sampling may lead to slow convergence.

PRM (Probabilistic Roadmap) PRM method involves two phases:

1. **Learning Phase:** Randomly sample the configuration space and connect nodes with collision-free paths to form a roadmap.
2. **Query Phase:** For a given start and goal, connect these points to the roadmap and perform a graph search (e.g., using Dijkstra or A*) to compute the path.

PRM is particularly efficient in multi-query scenarios, where the roadmap is built once and reused.

Advantages:

- Effective in high-dimensional spaces.
- Rapid query resolution once the roadmap is established.

Limitations:

- Roadmap construction can be computationally heavy and sensitive to the sampling density.
- The quality of the solution depends on the connectivity and distribution of the samples.

Metaheuristics

Metaheuristic algorithms have been widely adopted in path planning due to their ability to efficiently explore large search spaces, handle non-convex optimization problems, and adapt to dynamic environments. These algorithms provide near-optimal solutions within reasonable computational time by balancing exploration and exploitation strategies.

Ant Colony Optimization (ACO) Algorithm ACO is a bio-inspired metaheuristic algorithm that mimics the behavior of ants. Ants communicate indirectly using pheromone trails to find the shortest path between their nest and a food source. This concept is leveraged for solving combinatorial optimization problems, including path planning.

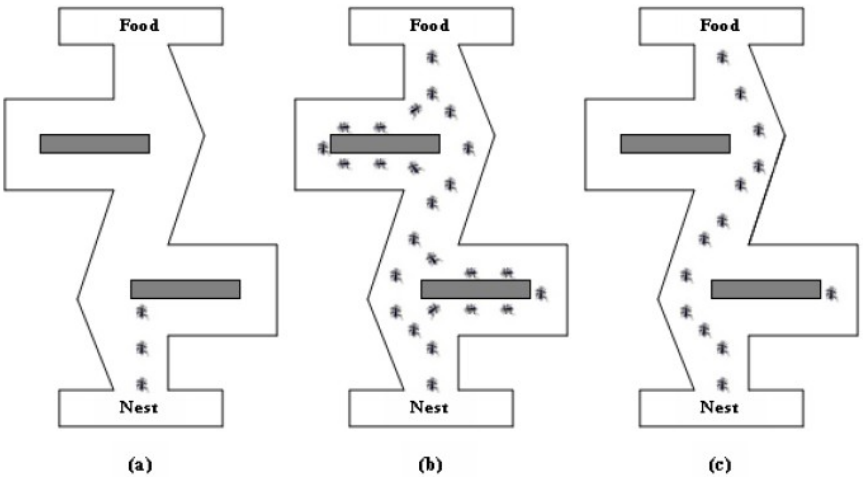


Figure 2.14: Ant colony optimization. [18]

Each ant in the algorithm constructs a path by selecting the next node based on the pheromone intensity and heuristic desirability. The probability of selecting a path is determined by:

$$p_{xy}^{(k)} = \frac{(\tau_{xy}^{\alpha})(\eta_{xy}^{\beta})}{\sum_{z \in allowed_x}(\tau_{xz}^{\alpha})(\eta_{xz}^{\beta})} \tag{2.5}$$

where:

- τ_{xy} is the pheromone level on the path between nodes x and y,
- η_{xy} represents a heuristic factor (e.g., inverse distance),
- α and β control the influence of pheromones and heuristics, respectively.

Once all ants complete their paths, pheromone trails are updated based on path quality. A common update rule is:

$$\tau_{xy} \leftarrow (1-\rho)\tau_{xy} + \sum_{k=1}^{m} \Delta\tau_{xy}^{(k)} \tag{2.6}$$

where ρ is the pheromone evaporation rate, and $\Delta\tau_{xy}^{(k)}$ is the pheromone deposited by the kth ant:

$$\Delta\tau_{xy}^{(k)} = \begin{cases} \frac{Q}{L_k}, & \text{if the ant used path } xy \\ 0, & \text{otherwise} \end{cases} \tag{2.7}$$

where L_k is the total length of the path taken by the kth ant, and Q is a constant.
Advantages:

- Suitable for dynamic and complex environments.
- Can handle large search spaces efficiently.
- Scalable to multi-agent robotic applications.

Limitations:

- Convergence speed depends on parameter tuning.
- Risk of stagnation if pheromone updates are not well-balanced.

Genetic Algorithm (GA) GAs are evolutionary optimization techniques inspired by natural selection and genetics. They are widely used in global path planning due to their ability to explore complex search spaces and optimize solutions efficiently. GAs do not require gradient information, making them suitable for nonlinear, high-dimensional problems.

In path planning, a potential path is represented as a **chromosome**, which is a sequence of **waypoints** connecting the start point to the goal point. Each waypoint represents a location in the environment, and the chromosome is encoded as a vector of coordinates. For example, in a 2D environment:

$$\text{Chromosome} = [(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)]$$

where (x_i, y_i) are the coordinates of the i-th waypoint.

The merit of a chromosome is quantified by a fitness function that blends several criteria: short overall length, collision-free clearance from obstacles, smooth turning behaviour, safe margins, and, when relevant, estimated energy consumption. Because GA relies only on comparisons of fitness values, it handles discontinuous or highly non-linear cost landscapes in which gradient-based optimisers fail.

The GA process consists of the following steps:

1. **Initialization**: A population of candidate paths (chromosomes) is randomly generated.
2. **Selection**: Paths with higher fitness scores are selected for reproduction. Common methods include:
   - **Roulette Wheel Selection**: Probabilistic selection based on fitness.
   - **Tournament Selection**: A subset of chromosomes competes, and the best is selected.
   - **Rank Selection**: Chromosomes are ranked, and selection probability is assigned accordingly.
3. **Crossover**: Two parent chromosomes are combined to create offspring. Common methods include:
   - **Single-point Crossover**: Genes are exchanged at a randomly chosen point.
   - **Two-point Crossover**: A segment between two points is swapped.
   - **Uniform Crossover**: Each gene is randomly selected from either parent.
4. **Mutation**: Random changes are introduced to maintain diversity. For example, a waypoint may be shifted or a new waypoint added.
5. **Evaluation**: The fitness of each offspring is evaluated.
6. **Replacement**: The new generation replaces the previous one.
7. **Termination**: The process stops when a termination condition is met (e.g., maximum generations or satisfactory fitness).

Advantages:

- **Global Search Capability**: Explores the entire search space to find global optima.
- **No Gradient Requirement**: Suitable for non-differentiable or discontinuous fitness functions.
- **Handling Complex Environments**: Effective in high-dimensional environments with multiple obstacles.
- **Parallelism**: Evaluates multiple solutions simultaneously.

- **Flexibility**: Easily adaptable to additional constraints or objectives.

Limitations:

- **Computational Cost**: Can be expensive for large populations or complex environments.
- **Premature Convergence**: May converge to suboptimal solutions if diversity is lost.
- **Parameter Tuning**: Performance depends on carefully chosen parameters.
- **Scalability**: Performance may degrade in extremely high-dimensional search spaces.

2.5.2 Local Path Planning (LPP)

Where the global planner produces a single reference path, the local planner must *continually reshape* that reference so the vehicle stays safe when people, pallets, or sensor noise invalidate the original assumptions. To do so it runs an inner control loop that operates directly in the robot's control space. At this rate, a planner must satisfy three constraints: it has to finish within a few milliseconds, obey the platform's kinematic and dynamic limits, and remain robust to the range and localisation noise that exist in real-world perception.

Dynamic Window Approach (DWA)

DWA regards motion commands as points in the plane of linear and angular velocity (v, ω). Because a robot cannot jump instantaneously to any command, only those pairs attainable within a short horizon Δt form the *dynamic window* $\mathcal{V}_d$. The bounds in (2.8) follow directly from the platform's maximum forward acceleration $a_{\max}$ and yaw acceleration $\alpha_{\max}$:

$$\mathcal{V}_d = \{(v, \omega) \,|\, v \in [v_c - a_{\max}\Delta t,\ v_c + a_{\max}\Delta t],\ \omega \in [\omega_c - \alpha_{\max}\Delta t,\ \omega_c + \alpha_{\max}\Delta t]\}. \quad (2.8)$$

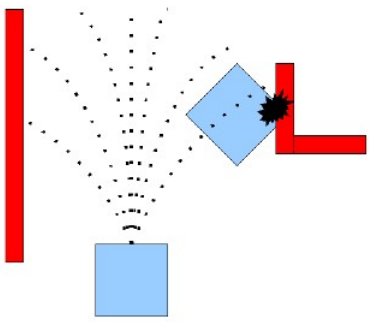


Figure 2.15: Dynamic Window Approach Algorithm. [19]

A second filter rejects commands that cannot be brought to a halt before hitting the nearest obstacle. Rearranging the work–energy principle gives the familiar braking

inequality $v \leq \sqrt{2a_{\max} d_{\text{obs}}}$, yielding the admissible set $\mathcal{V}_a$. The effective search space is their intersection $\mathcal{V}_r = \mathcal{V}_d \cap \mathcal{V}_a$.

Each candidate command is forward–simulated for Δt with the differential–drive model; the resulting trajectory is scored by (2.9). Four terms dominate practical tunings:

$$J = \underbrace{\alpha\, d_{\text{obs}}^{-1}}_{\text{clearance}} + \underbrace{\beta\, \|\theta_{\text{goal}} - \theta_{\text{end}}\|}_{\text{heading}} + \underbrace{\gamma\, (v_{\max} - v)}_{\text{progress}} + \underbrace{\delta\, |\omega|}_{\text{smooth turn}} \quad . \tag{2.9}$$

Good commands maximise obstacle clearance, point the robot roughly toward the goal, keep the linear speed high, and avoid unnecessary spinning. The best-scoring (v, ω) is executed for one control period before the process repeats.

- **Key Features:**
  1. Directly accounts for the robot's dynamic constraints via $\mathcal{V}_d$ and $\mathcal{V}_a$.
  2. Provides real-time adaptability to dynamic environments through velocity space sampling.
  3. Inherently handles non-holonomic constraints through kinematic simulation.
- **Applications:**
  1. Widely used in differential-drive and car-like robotic platforms.
  2. Integrated in popular navigation frameworks such as the ROS Navigation Stack.

Potential Field Methods

APF interprets navigation as movement along the negative gradient of a scalar potential $U(\mathbf{x})$. A quadratic well centred on the goal attracts the robot, $U_{\text{att}} = \frac{1}{2} k_{\text{att}} \|\mathbf{x} - \mathbf{x}_g\|^2$, whereas each obstacle raises a repulsive hill that grows steep when the robot comes closer than a threshold ρ_0.

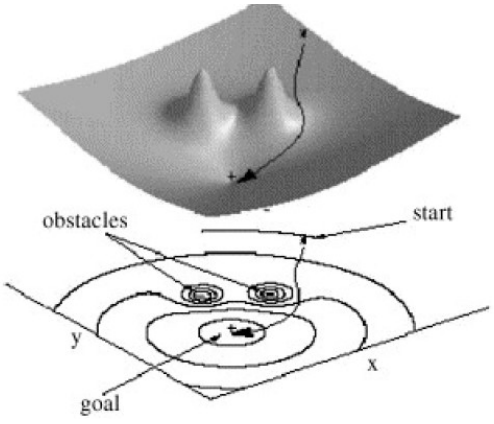


Figure 2.16: Artificial Potential Field Method. [20]

Summing the gradients produces a continuous force field $\mathbf{u} = -\nabla U_{\text{att}} - \sum_i \nabla U_{\text{rep},i}$ that can be fed straight into a velocity or acceleration controller. In open spaces the attractive term dominates, yielding graceful motion; near obstacles the repulsive gradient pushes the robot away.

To mitigate local minima, harmonic potential fields replace U_{rep} with solutions to Laplace's equation $\nabla^2 U = 0$, ensuring no spurious minima. Alternatively, randomized escape strategies temporarily add virtual repulsive forces.

- **Advantages:**
  1. Conceptually simple with continuous control laws.
  2. Computationally light, making it suitable for fast reactive control.
- **Limitations:**
  1. *Local Minima:* The robot may become trapped in areas where the net force is zero.
  2. *Narrow Passage Handling:* Standard potential fields may require modifications (e.g., harmonic potentials) to navigate tight spaces.

Vector Field Histogram (VFH)

Instead of working in Cartesian space, VFH converts a laser scan into a one-dimensional polar histogram that encodes *how much free space* lies in each steering direction. A valley in this histogram marks a potentially safe heading.

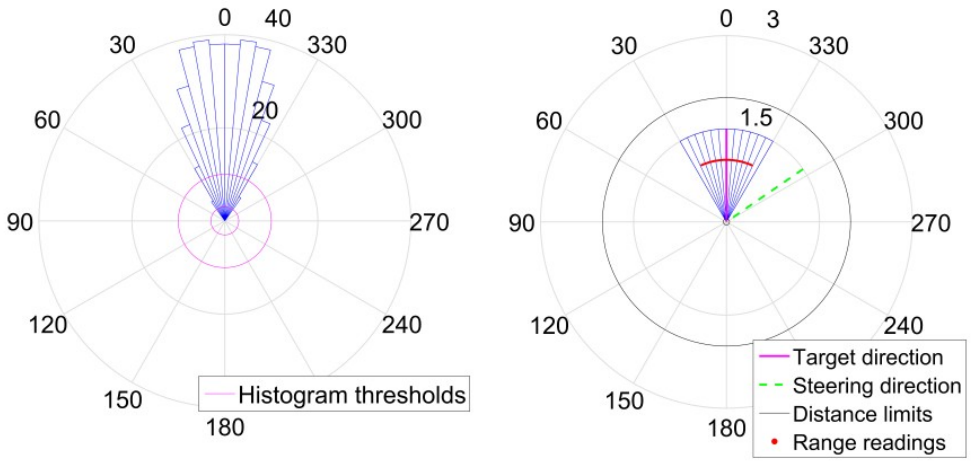


Figure 2.17: Vector Field Histogram Examples. [21]

Distances within every angular sector $\theta_k = 2\pi k/n$ are mapped to a density value c_k a smoothing kernel suppresses sensor noise. Candidate valleys are those sectors whose smoothed density $\tilde{H}(\theta_k)$ falls below a threshold. The heading that minimises the cost,

balances goal alignment, continuity with the previous steering command, and raw obstacle density.

- **Advantages:**

  1. VFH avoids forward simulation and therefore executes in *sub-millisecond time*, easily meeting the cycle times of real-time controllers.
  2. Its computational lightness makes it attractive for *low-speed indoor robots* and other resource-constrained platforms.

- **Limitations:**

- Steering commands can *oscillate* when adjacent histogram sectors alternate rapidly between free and occupied; this artifact is usually mitigated with *hysteresis thresholds* or additional smoothing of the histogram.

Timed Elastic Band (TEB)

Where APF and VFH react over a single control step, TEB formulates local planning as a constrained optimisation of the whole short-horizon trajectory. The path is an "elastic band",a string of poses $\mathbf{q}_i = (x_i, y_i, \theta_i, t_i)$, whose virtual springs penalize stretching and bending, while obstacle and dynamic-limit terms keep it safe and feasible.

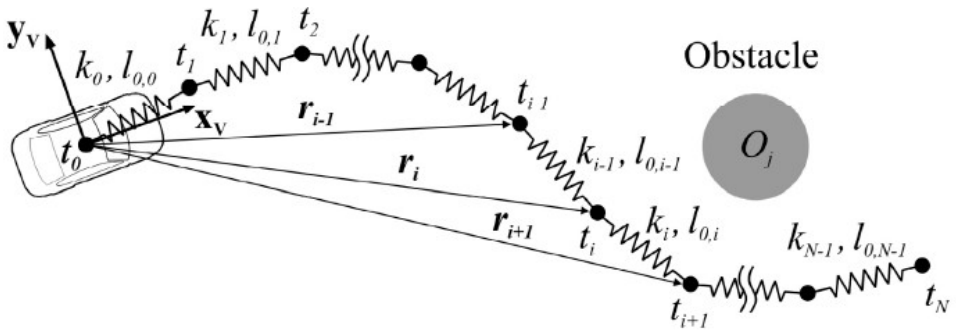


Figure 2.18: Single elastic band described in a vehicle fixed reference frame and an obstacle $\mathbf{O}_i$. [22]

The cost sums quadratic penalties on translational and angular velocity, acceleration, and proximity to obstacles. By incorporating the pose timestamps t_i as decision variables, TEB obtains a time-optimal schedule *and* a geometrically smooth curve in one solve. Because the underlying graph has only local couplings, Gauss–Newton with an analytic Jacobian solves each Newton step in a few hundred microseconds on a modern CPU.

- **Advantages:**

  1. Generates smooth and dynamically feasible trajectories.

2. Can adapt to moving obstacles by incorporating predictive models into the cost function.

- **Limitations:**
  1. Higher computational cost compared to simpler reactive methods.
  2. Requires careful tuning of multiple parameters to achieve the desired behavior.

Chapter 3

ROS 2 Middleware Overview

ROS 2 represents a significant evolution in robotic software frameworks, offering enhanced capabilities for SLAM and autonomous navigation. This chapter provides a comprehensive overview of ROS 1 and ROS 2's architecture, tools, and workflows for SLAM and navigation, with a focus on practical implementation details taken directly from the official documentation [23] and [24].

3.1 ROS Architecture

The first generation of the Robot Operating System, also known as ROS 1, emerged from Willow Garage in 2010 to provide a thin, portable middleware layer that is inserted between low-level real-time control loops and high-level application software for robotic applications.

At its core, ROS 1 organises a robot's software into independent Unix processes called *nodes*. During start-up the node opens an XML-RPC connection to the `roscore` master, registers its own URI, and advertises the names of every *topic*, *service*, or *parameter* it intends to provide or consume. Node names and topic names form a hierarchical namespace whose syntax mirrors POSIX paths.

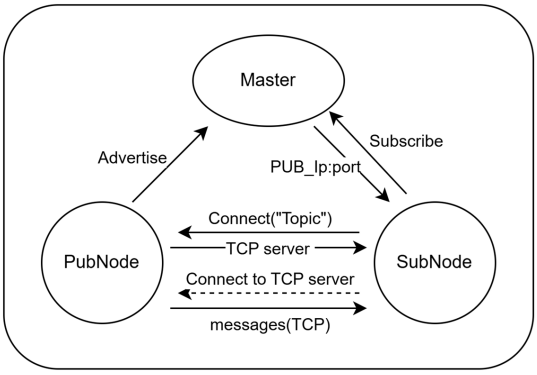


Figure 3.1: Establishing a topic connection in ROS 1.

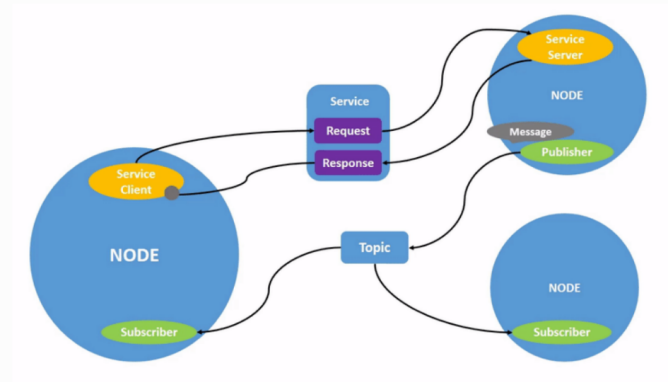
This abstraction quickly resulted in an increased level of code reuse. Because the message definition language is separated from the implementation languages, the same topic could be produced in C++ and consumed in Python without recompilation. This resulted in the development of reusable packages for perception, mapping, and manipulation at a pace that traditional laboratory-specific frameworks could not match. The ecosystem enables rapid integration and testing: a LiDAR-based SLAM algorithm can be downloaded, its odometry streamed into the canonical `move_base` navigation stack, and the outcome visualised in `rviz` with minimal effort. By easing the burden of custom code, ROS 1 broadened access to robotics research and swiftly became academia's standard tool for rapid development.

The same decisions that made ROS 1 highly accessible also ultimately constrained its effectiveness for large-scale, safety-critical, or real-time applications. The centralized *master* node represented a single point of failure where the crash of the `roscore` would result in the loss of all topic connections. Communication was based on ad hoc TCP or UDP transports that lacked deterministic latency and formal Quality-of-Service (QoS) guarantees, limitations that became apparent over unreliable wireless networks or when handling high-bandwidth sensor data such as 3D LiDAR. Security was largely overlooked: messages were sent in plaintext, there was no authentication, and parameter namespaces did not isolate multiple robots operating on the same network. Additionally, the build system and runtime environment assumed POSIX compliance, which complicated efforts to port ROS 1 to real-time operating systems or Windows.

3.1.1 Main Components of ROS

Nodes

A node is a fundamental ROS element that serves a single, modular purpose in a robotics system. Each node in ROS should be responsible for a single, modular purpose, e.g. controlling the wheel motors or publishing the sensor data from a laser range-finder. Each node can send and receive data from other nodes via topics, services, actions, or parameters. A full robotic system is comprised of many nodes working in concert, and a single executable (C++ program, Python program, etc.) can contain one or more nodes.



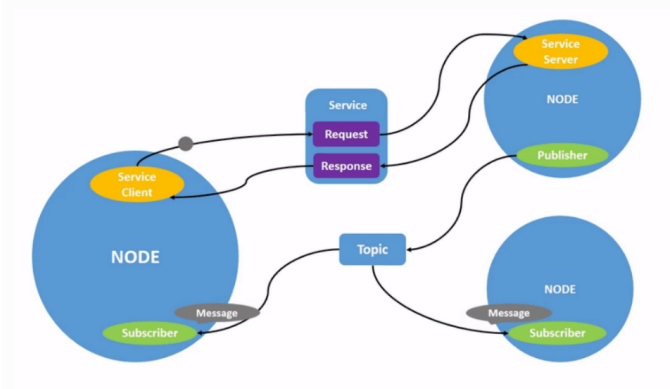
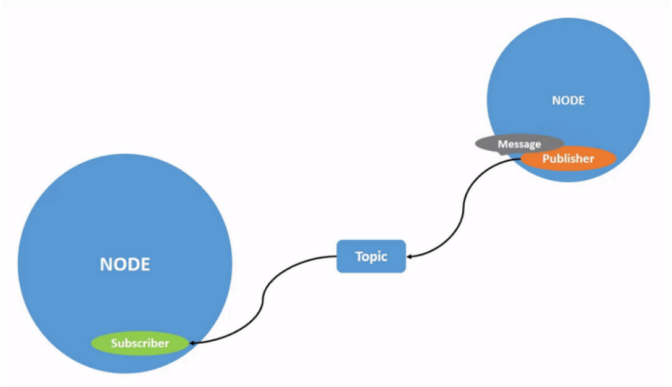


Figure 3.2: A full robotic system is comprised of many nodes working in concert.

Topics

A topic is an anonymous many-to-many bus that streams messages of a single, strongly-typed class. The type is defined in a `.msg` specification, compiled ahead of time into language-specific structs together with a compile-time MD5 checksum. When a publisher and subscriber negotiate a connection they exchange this checksum to guarantee wire compatibility, then choose either the TCPROS or UDPROS transport. Messages are serialised in a zero-terminated, length-prefixed binary format. No framing packets traverse `roscore` once the peer-to-peer socket is established, so nominal throughput approaches raw TCP/UDP bandwidth. Because any number of publishers can coexist on the same topic, sensor fusion pipelines can multiplex LiDAR, radar, or visual feeds without an intermediary broker. ROS 2 breaks complex systems down into many modular nodes. Topics are a vital element of the ROS graph that act as a bus for nodes to exchange messages. A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics.



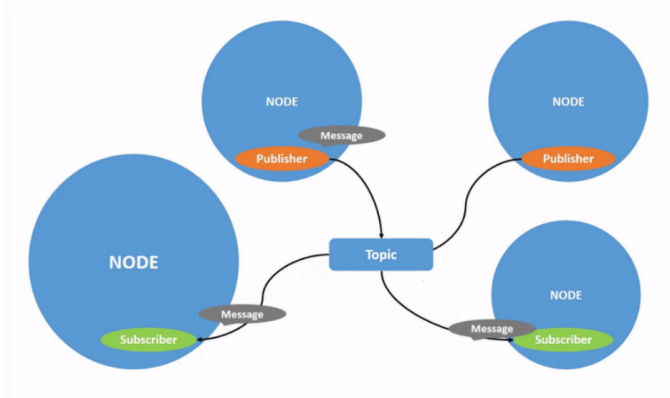


Figure 3.3: Topics are one of the main ways in which data is moved between nodes.

Services

Services are another method of communication for nodes in the ROS graph. Services are based on a call-and-response model versus the publisher-subscriber model of topics. While topics allow nodes to subscribe to data streams and get continual updates, services only provide data when they are specifically called by a client. A service is declared in a `.srv` file that splits the message definition into a request part and a response part separated by `---`. The client blocks until the provider returns, making services ideal for parameter look-ups, database queries, or single-shot motion planning calls where timing is not hard-real-time.

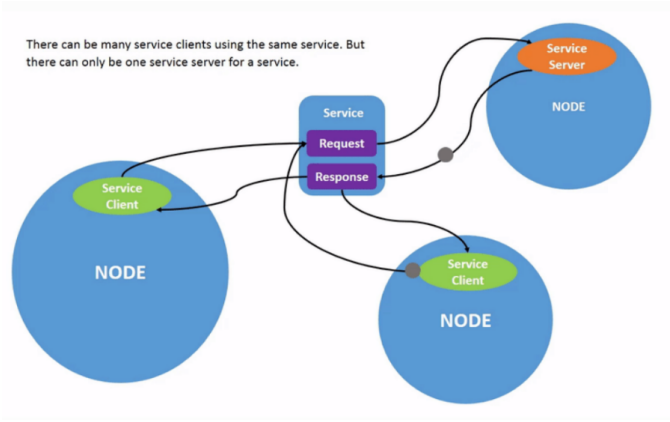


Figure 3.4: There can be many service clients using the same service.

Actions

Long-running, pre-emptible tasks such as mobile-base navigation require richer semantics than a simple RPC. The **actionlib** protocol therefore layers three auxiliary topics on

top of a logical *action name*: `<name>/goal`, `<name>/feedback`, and `<name>/result`. A client submits a goal, receives streaming feedback, and may cancel the goal at any time; when the server finishes, it publishes a result. This pattern is used in canonical stacks such as `move_base`, `moveit_action`, and `grasp_execution`. Actions are built on topics and services. Their functionality is similar to services, except actions can be canceled. They also provide steady feedback, as opposed to services which return a single response. Actions use a client-server model, similar to the publisher-subscriber model. An "action client" node sends a goal to an "action server" node that acknowledges the goal and returns a stream of feedback and a result. A robot system would likely use actions for navigation. An action goal could tell a robot to travel to a position. While the robot navigates to the position, it can send updates along the way (i.e. feedback), and then a final result message once it's reached its destination.

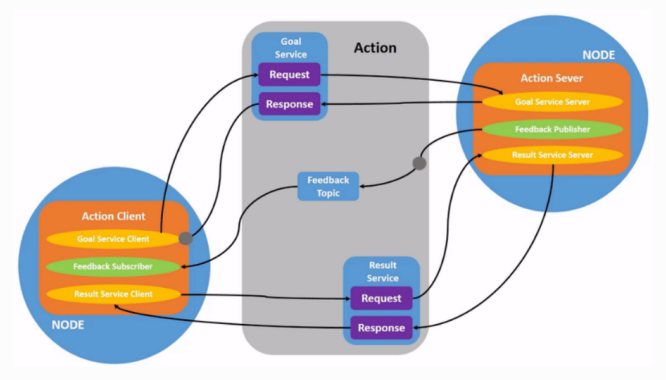


Figure 3.5: Actions are like services that allow the execution of long running tasks.

`roscore` and the global graph

The program `roscore` bundles three services: the *ROS Master*, a hierarchical *Parameter Server*, and the `/rosout` logging aggregator. The Master is responsible solely for name resolution and connection brokering. After it delivers the peer URIs, data flow bypasses it entirely. The Parameter Server exposes a process-safe, YAML-typed dictionary that nodes query via XML-RPC calls, enabling run-time reconfiguration without rebuilds. Taken together, the Master's registry plus all active nodes, topics, services, and action channels make up the *ROS computation graph*: A directed graph that developers can inspect with tools such as `rosnode info` and `rqt_graph`.

3.1.2 ROS 2 Improvements

ROS 2 introduces several architectural improvements over ROS 1, making it more suitable for modern robotic applications. It is a full middleware rewrite that replaces the centralized

design of the original framework with a standards-based, real-time-capable, and security-aware architecture. The key changes that directly benefit SLAM, closed-loop control, and the HIL workflows used in this study are outlined below.

Data Distribution Service (DDS)

ROS 1 established point-to-point TCP or UDP sockets through a single `roscore` master. In ROS 2, that bespoke layer is replaced by the OMG Data Distribution Service (DDS) and its Real-Time Publish–Subscribe (RTPS) wire protocol. The middleware provides the following features:

- **Quality of Service (QoS) Policies**: Configurable communication reliability, durability, and deadline constraints. This is critical for SLAM and navigation systems, where sensor data must be delivered reliably and on time. For example:
  - `RELIABLE`: Ensures message delivery for critical data like LiDAR scans.
  - `BEST_EFFORT`: Suitable for high-frequency odometry data.
  - `TIME_BASED_FILTER`: Synchronizes sensor data from multiple sources.
- **Decentralized Communication**: Eliminates the need for a central ROS Master, enabling peer-to-peer communication. This improves fault tolerance and scalability, as nodes can communicate directly without a single point of failure.
- **Real-Time Support**: Ensures deterministic message delivery for time-critical applications. ROS 2 supports real-time scheduling and priority-based execution, making it suitable for applications requiring precise timing, such as sensor fusion and control loops.

The *ROS Middleware Interface* (`rmw`) lets developers swap the underlying Data Distribution Service (DDS) implementation without touching application code. Several engines are maintained as *Tier-1* options:

- **Fast DDS** (eProsima) ships as the default since the *Foxy* distribution. It achieves sub-millisecond latency on Gigabit Ethernet and provides built-in `SHM` (shared-memory) transports that remove kernel context-switch overhead—useful when the LiDAR front-end and the Nav 2 controller run on the same CPU.
- **Cyclone DDS** (Eclipse) focuses on deterministic discovery timing and steady latency under heavy multicast traffic. From the *Iron Irwini* release onward it is the default for micro-ROS, where a deterministic RMW agent on the SBC proxies traffic for multiple Cortex-M micro-controllers.
- **Zenoh** is an emerging `rmw` back-end optimised for lossy, high-latency links such as Wi-Fi 6 mesh and 5G URLLC. Zenoh performs *query-able pub/sub*: if a packet is missed, the subscriber can re-query the key space—an attractive feature for outdoor SLAM where transient drop-outs are inevitable.

Lifecycle Nodes

Where ROS 1 nodes had no formal state machine, ROS 2 introduces *lifecycle nodes* that progress deterministically through the following states:

- `Unconfigured`: The node is created but not configured.
- `Inactive`: The node is configured but not actively processing data.
- `Active`: The node is fully operational and processing data.
- `Shutdown`: The node is deactivated and resources are released.

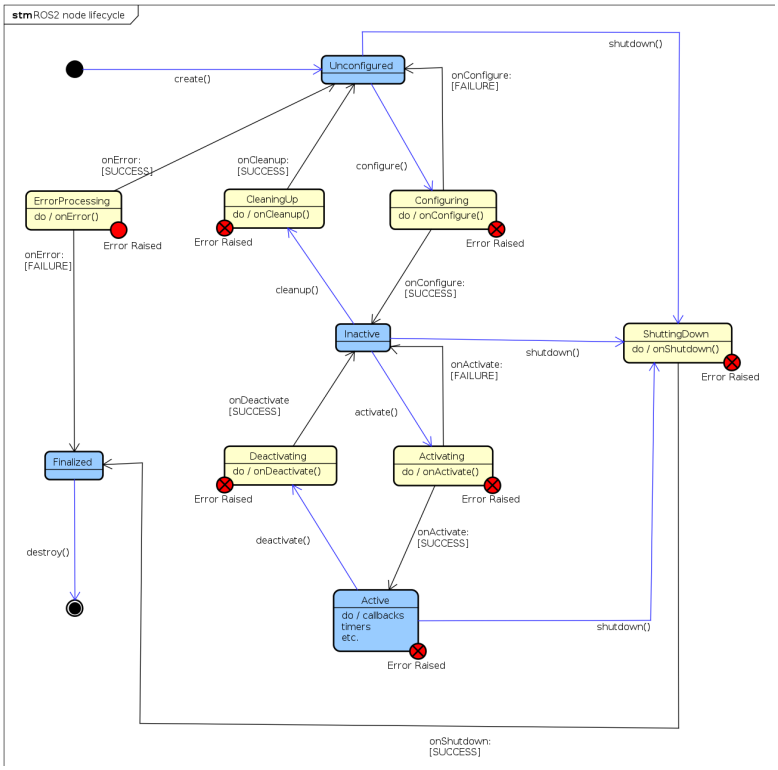


Figure 3.6: Overview of the Life Cycle Finite State Machine.

Node lifecycles give finer control over a ROS 2 system's state. It enables roslaunch to verify that all components are properly instantiated before any begin executing their behavior. Additionally, lifecycle management ensures that nodes can be started, stopped, and recovered gracefully, which is critical for long-running systems like SLAM. For example, the `slam_toolbox` node can be configured to transition to an inactive state if sensor data is unavailable, and reactivate once data is restored.

Performance Optimization

ROS 2 provides several mechanisms for optimizing SLAM and navigation systems. First, *intra-process communication* allows a publisher and subscriber running inside the same process to exchange a pointer to a *loaned* message rather than serializing the data into a full ROS message over shared memory. Second, the executor API ships with a *static real-time executor* that can bind callbacks to threads and lock memory pages, ensuring deterministic wake-up even under CPU load. Finally, the `rclcpp` executor may spin in multi-threaded mode, letting independent callback groups—scan matching, TF broadcasts, and cost-map updates—run in parallel on separate cores.

Multi-Robot Support

ROS 2 supports multi-robot systems through namespaces and domain IDs, enabling concurrent SLAM and navigation processes for multiple robots. This is achieved by:

- **Namespaces**: Each robot operates within its own namespace, isolating its topics, services, and parameters.
- **Domain IDs**: Robots can operate in separate communication domains, preventing interference between systems.
- **TF2 Frame Prefixing**: Each robot's TF2 frames are prefixed with its namespace, ensuring proper coordinate transformations in multi-robot environments.

Improved Security

ROS 2 integrates DDS Security, providing encryption, authentication, and access control for robotic systems. This is particularly important for industrial and commercial applications where data integrity and system security are critical. Key features include:

- **Encryption**: Protects sensor data and control commands from interception.
- **Authentication**: Ensures that only authorized nodes can communicate.
- **Access Control**: Restricts access to sensitive topics and services.

3.2 Autonomous Navigation with ROS 2

ROS 2 delivers an integrated software pipeline that covers every stage of mobile robot autonomy. First, an online SLAM module constructs a metric (or metric–topological) map of the workspace. Second, a localisation filter maintains a real-time estimate of the robot's pose with respect to that map. Finally, the Nav 2 navigation stack converts a target pose into velocity commands that comply with the vehicle's kinematic and dynamic constraints. Frame consistency is enforced by TF 2, whose microsecond-timestamped buffers ensure that asynchronous sensor packets are fused without geometric error which is a prerequisite for HIL tests.

3.2.1 SLAM in ROS 2

Every ROS 2 SLAM package follows the same standard process: it subscribes to raw sensor feeds such as `/scan`, `/imu`, and `/odom`, publishes pose and map updates on `/map` and related topics, and broadcasts the transform chain `map` → `odom` → `base_link`. Within that interface, the following two packages are usually implemented for this purpose.

SLAM Toolbox focuses on 2-D LiDAR and runs efficiently even on low-power ARM boards. The front-end uses a point-to-line ICP aligned with a multi-resolution branch-and-bound search. Its back-end maintains an incremental pose graph solved by a Levenberg–Marquardt routine on the SE(2) manifold; loop-closure constraints that add little new information are pruned, which keeps optimization time roughly constant as the map grows. Because the full pose graph—including sub-map tiles and covariance blocks can be serialized to a `.pbstream` file and restored without restarting the node, robots can extend or refresh large-scale maps during off-hours and resume normal operation the next morning.

Cartographer extends the same ideas to three dimensions and to heterogeneous sensor suites. Incoming LiDAR scans are first batched into local sub-maps that store a truncated signed-distance function (TSDF) or a probability grid, depending on whether dense or sparse mapping is required. A GPU-accelerated branch-and-bound search aligns each incoming scan to the nearest sub-map; the resulting relative poses feed an error-state Kalman filter that fuses inertial, wheel-odometry, and optionally GNSS measurements at several hundred hertz. Global consistency is enforced by a sparse pose-graph optimiser running in a parallel thread. Constraints older than a configurable age are marginalised so that memory remains bounded even during day-long mapping sessions, and the whole pipeline reaches real-time performance on a desktop-class GPU, which makes it suitable for UAVs and legged robots that need online 3-D perception.

Transform and Timing Considerations

TF 2 stores static and dynamic transforms in a lock-free circular buffer; each lookup completes in constant time and supports interpolation or short extrapolation windows of approximately ±100 ms. In practice, transforms that link the odometry frame to the robot base should be broadcast at 50–100 Hz so that the navigation stack, whose default transform timeout is 0.5 s, never encounters a gap. The same buffer lets the SLAM back-end compensate for moderate hardware-level time skews between LiDAR and IMU packets without additional synchronisation circuitry.

3.2.2 Navigation Stack (Nav 2)

Nav 2 is the ROS 2-native successor to the monolithic `move_base` node of ROS 1. The redesign decomposes navigation into life-cycle managed servers that communicate through DDS Actions and Services and are orchestrated at run time by a behaviour-tree (BT) interpreter. This separation of concerns yields three advantages that are central to hardware-in-the-loop (HIL) testing:

1. Each server can be restarted or re-parameterised without halting the others, enabling live tuning on the bench;

2. Every capability—global planning, local control, recovery—is delivered as a plug-in, so alternative algorithms can be swapped in without changing message contracts; and

3. The BT layer makes the robot's decision logic explicit, which simplifies fault injection and trace analysis during HIL validation.

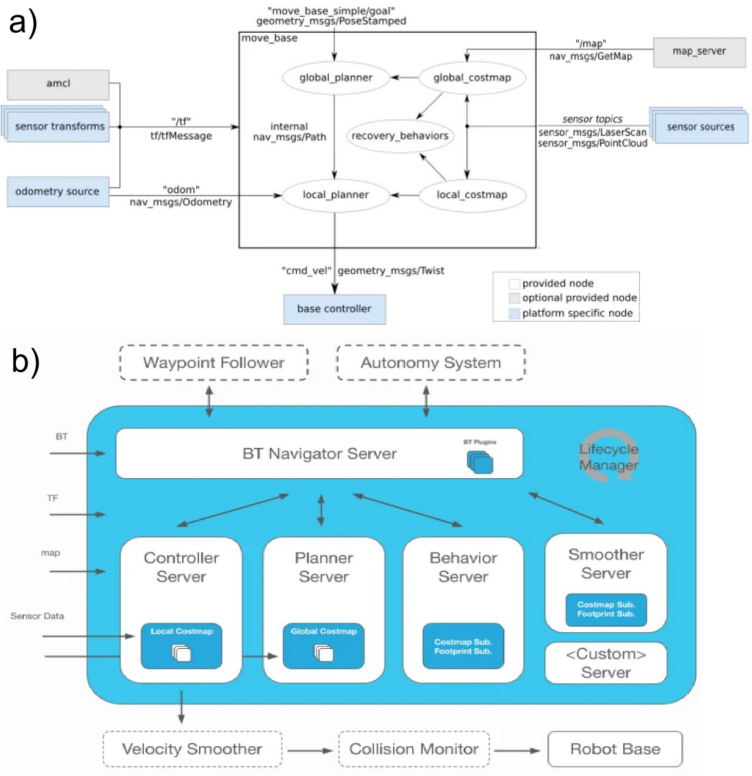


Figure 3.7: Comparison of the Navigation Stack in ROS 1(a) and ROS 2(b).

Reference Frames and the `tf2` Transform Tree

Navigation starts with a clear definition of how every part of the robot, its sensors, and the world relate to one another. In ROS 2 these relations are stored in a `tf2` tree, a time-stamped graph that connects all coordinate frames without loops. For a mobile robot the essential chain is

- **`map` → `odom`** – kept up to date by a global localiser (GPS, AMCL, or SLAM). Its job is to remove long-term drift.

- **`odom` → `base_link`** – produced at high rate by the odometry system (wheel encoders, IMU, visual/LiDAR odometry). It must be smooth so that controllers get stable feedback.
- **`base_link` → sensors** – fixed offsets taken from the URDF/Xacro model and published once by `robot_state_publisher`.

The tree is acyclic, so every pair of frames has one and only one path. Dynamic links (`map`→`odom` and `odom`→`base_link`) are broadcast at 50–100 Hz; static links are sent once and cached by all nodes.

A single error in this tree, such as wrong frame name, wrong rate, or wrong time stamp, can lead to warped maps and unstable motion. For this reason a TF visual check in RViz is a standard part of every system bring-up.

Global Localisation: GPS, AMCL, and SLAM Toolbox

The first dynamic link in the TF tree, `map`→`odom`, is maintained by a *global localiser*. Its task is to align the robot's odometry with an external reference so that long-term drift is removed.

Outdoor: When satellites are visible, a GNSS receiver can publish the `map`→`odom` transform directly, giving metre-level accuracy without further processing.

Indoor: In a known building or warehouse, Nav 2 uses **Adaptive Monte-Carlo Localisation** (AMCL). A set of weighted particles is propagated with the robot's motion and scored against a static occupancy map; the mean pose is broadcast at 10–30 Hz. AMCL is light on CPU and works well if the map is reliable and the environment changes little.

Unknown sites: If no map exists, **SLAM Toolbox** builds one on the fly while also publishing the same `map`→`odom` transform. Laser scans are added as nodes in a pose graph; loop closures are optimized in the background, keeping the estimated pose smooth enough for real-time navigation.

Both AMCL and SLAM Toolbox output a stamped TF message; no other part of the stack talks to them directly. Because the interface is only a transform, any other global source, such as motion-capture, UWB beacons, and visual-inertial SLAM, can be swapped in if it publishes the same link.

Local Odometry Fusion with `robot_localisation`

Navigation controllers need pose updates faster than a global localiser can provide. Nav 2 therefore derives the `odom`→`base_link` transform from onboard sensors only, aiming for a smooth 50–100 Hz signal.

Typical sources are wheel encoders, an IMU, and either visual or 2-D LiDAR odometry. Each sensor reports a subset of the state (pose, velocity, or acceleration) and carries its own covariance. The **`robot_localisation`** package fuses these streams in an Extended

or Unscented Kalman Filter. A YAML mask selects which of the 15 state variables each sensor owns.The filter publishes:

- a stamped transform `/tf`: `odom`→`base_link`;
- a `nav_msgs/Odometry` message whose `twist` field gives the current velocity, used by many controllers.

If the global localiser also produces pose estimates, a second, slower EKF instance can run in the `map` frame. The local EKF delivers the high-rate odometry; the global EKF injects drift corrections without causing jumps in control.
Process noise should be on the same order as the sensor covariances; too small gives sluggish motion, too large adds jitter. Publishing the TF at the controller's update rate is usually sufficient.

With both dynamic links now in place, all sensor frames have a clear, time-aligned path back to `map`. The stack can safely transform measurements, build costmaps, and plan paths.

URDF/Xacro and the Robot State Publisher

Static transforms, such as the fixed offsets between the robot chassis and its sensors, are derived from the robot description. Nav 2 relies on the *Unified Robot Description Format* (URDF), usually written as a Xacro macro file to keep it short and parametric.
Every rigid part of the robot is a `<link>` element; each spatial relation is a `<joint>`.
At launch, `robot_state_publisher` parses the URDF and emits all `base_link`→`sensor` transforms exactly once. Because `tf2` caches static messages, there is no run-time cost.
Adding `<visual>` tags lets RViz show a textured model; `<collision>` and `<inertial>` tags allow Gazebo or Ignition to simulate physics with the same file. Xacro constants (e.g. wheel radius, sensor height) make it easy to reuse the description between variants.
Many planners assume a 2-D "shadow" of the robot called `base_footprint`. This link is mass-less and has a fixed joint to `base_link`, centred at the projected contact patch.

Layered Costmaps

Planning and control need a fast answer to one question: *is a given footprint pose free, risky, or blocked?* Nav 2 answers by building a 2-D costmap, a grid where each cell stores an 8-bit cost:

$$0 = \text{unknown}, \quad 1 = \text{free}, \quad 253 = \text{inscribed}, \quad 254 = \text{lethal}.$$

Intermediate values come from the inflation layer and provide a smooth safety margin around obstacles. The costmap is the point-wise maximum of several `pluginlib` layers, each running in its own thread:

Obstacle layer: Ray-traces 2-D LiDAR or projects depth images, marking hits lethal and clearing freespace.

Voxel layer: Extends the same idea into height slices, with decay timers so doors that re-open are not treated as walls.

Static layer: Loads permanent keep-out zones from a YAML map.

Inflation layer: Convolves the merged obstacle grid with an exponential kernel, turning hard walls into a gradient useful for optimisation-based controllers.

Developers can add custom layers, for example, to raise costs near people detected by vision, without touching the planner or controller code. To avoid locking, the master costmap alternates between two grids: one is being updated by layers, the other is read by planners and controllers.
Each layer compares the stamp of incoming sensor messages with the ROS clock, and packets older than a threshold are ignored, preventing stale data from corrupting the map when bags are replayed or networks lag.

Nav 2 Action Servers and Behaviour Tree

Nav 2 organises long-running navigation tasks around ROS 2 *actions*. A behaviour tree (BT) at the top level coordinates four persistent action servers.

1. **Planner Server** : computes a global path whenever the goal or costmap changes.
2. **Controller Server** : converts that path into velocity commands at 20–50 Hz.
3. **Smoother Server** : (optional) refines new paths to reduce jagged turns and enlarge clearance.
4. **Recovery Server** : executes fallback manoeuvres when the BT detects a stall.

Each server loads its algorithms as `pluginlib` plugins, so changing planner or controller means editing a YAML file, not recompiling Nav 2. The BT ticks its child nodes at 10 Hz, passes goals to the servers, and reacts to their feedback. If the controller reports no progress, the BT switches to a recovery node; if recoveries fail, it aborts the mission and can notify an operator.
All four servers share a common pattern:

- `goal` : target pose or path
- `feedback`: percentage of task complete, time spent, distance remaining
- `result` : success flag and final pose

This uniform interface simplifies logging and testing, and allows external clients—fleet managers, tele-op stations—to monitor or pre-empt navigation with a single API.

Planner Plugins

The Planner Server decides where the robot should drive by generating a path from the current pose to the goal inside the global costmap. Nav 2 ships with several interchangeable plugins, each suited to different environments and vehicle constraints.

NavFn (Dijkstra/A*): A grid-based Dijkstra search computes the cost-to-go from every free cell to the goal then an A* back-trace extracts the path. Results are smooth and wide, which makes NavFn a dependable default for service robots in cluttered indoor spaces. Runtime is linear in the number of free cells.

Theta*: Theta* modifies A* to allow any-angle shortcuts between grid vertices, reducing both path length and number of waypoints. It excels in long corridors or open warehouses where straight lines minimize wheel wear and travel time. The algorithm runs slightly slower than NavFn because each expansion checks line-of-sight, but the difference is negligible for typical map sizes.

Smac Hybrid-A*: For non-holonomic bases, such as forklifts, ackermann cars, and long robots, paths must be *drivable*. Hybrid-A* searches in SE(2): each node stores x, y, θ and expands using vehicle kinematics, guaranteeing that every arc respects minimum turning radius and direction of travel. The plugin includes analytic Reeds–Shepp boosts for rapid convergence and supports heuristics based on a 2-D Dubins estimate. Planning time scales with heading resolution; 5° steps give sub-second plans on embedded ARM CPUs.

All three plugins can switch from point-to-point to coverage mode, producing a path that sweeps every free cell (e.g. floor-cleaning). The choice of algorithm, such as grid spiral, boustrophedon, and voronoi, follows the same plugin pattern and can be configured at launch.
Because the Planner Server reloads plugins at runtime, users often combine algorithms: NavFn for nominal goals, Hybrid-A* for tight parking, and a coverage planner for nightly cleaning. The BT selects the appropriate plugin through a blackboard variable, giving per-task flexibility with no code changes.

Controller Plugins

The goal of the Controller Server is to allow the robot to reliably follow the chosen path, while keeping it safe. Each cycle, it receives the latest global path, samples a window of the local costmap, and returns a velocity command in the `base_link` frame. Nav 2 offers four main plugins that cover the most common drive bases and operating conditions.

Table 3.1: Nav 2 controller plugins key characteristics

Plugin	Description	Robot Types / Drivetrain	Task
DWB	Highly configurable DWA with plugin interfaces.	Differential, Omni, Legged	Dynamic obstacle avoidance
MPPI	Predictive MPC with modular cost functions.	Diff., Omni, Ackermann, Legged	Dynamic obstacle avoidance
RPP	Adaptive, industrial variant of pure-pursuit.	Ackermann, Legged, Differential	Exact path following
Rotation Shim	"Shim" controller: rotates to path heading before tracking.	Diff., Omni (in-place)	Rotate to rough heading
Graceful (VP)	Pose-following control law yielding smooth trajectories.	Differential, Ackermann, Legged	High-speed path tracking

DWB (Dynamic Window Approach): DWB samples thousands of velocity pairs $(v,\ \omega)$ that respect the robot's limits over the controller time step. Each short trajectory is scored by a set of plug-in *critics*—distance to path, distance to goal, obstacle cost, heading error—and the lowest-cost command is sent to the motors. By adjusting critic weights or adding new critics, DWB can be tuned from slow, conservative motion in tight aisles to faster, more direct motion in open spaces. Default settings run comfortably at 20–30 Hz on embedded hardware.

MPPI (Model Predictive Path Integral): MPPI treats control as an optimisation problem. Starting from the previous best sequence of commands, it samples random perturbations, predicts the resulting motions with a simple kinematic model, scores each motion with the same critic set used by DWB, and updates the command sequence through a soft-max. The method handles non-convex cost landscapes and moving obstacles well, making it suitable for busy factory floors. Compute time scales with sample count..

Regulated Pure Pursuit (RPP): RPP extends the classic pure-pursuit algorithm with two safety rules: Velocity is reduced as path curvature increases, and a forward time check halts the robot if any obstacle will be reached within a set horizon. RPP is a good default when exact path tracking is more important than active obstacle avoidance. Parameter tuning focuses on look-ahead distance and the collision-check time window.

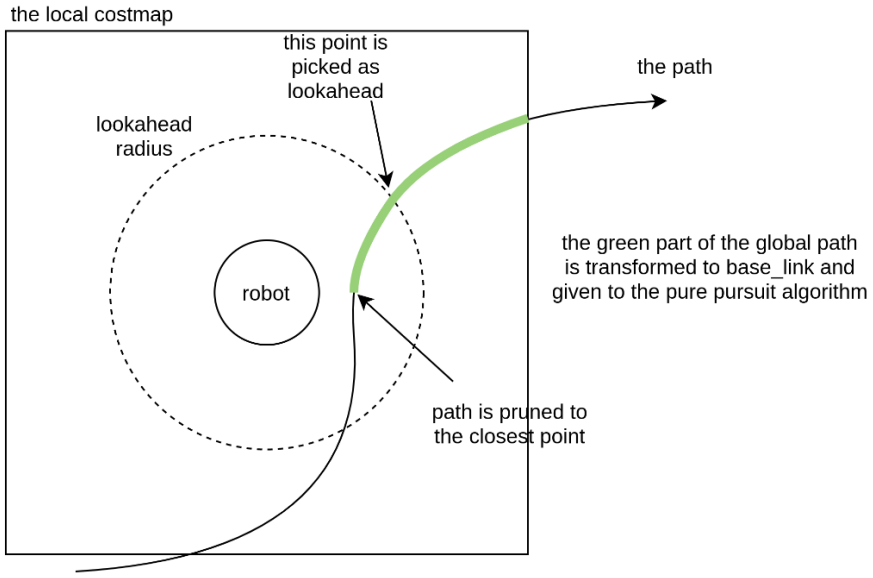


Figure 3.8: Regulated Pure Pursuit Controller.

Rotation Shim: Many global planners ignore the robot's initial heading. If a new path starts behind the current orientation, DWB or MPPI may waste time on awkward spirals. The Rotation Shim plugin solves this by rotating the robot in place until its heading

aligns with the first path segment, then handing control to the primary controller. It adds negligible CPU cost but greatly improves behaviour in cramped sites.

All controllers share three input topics, namely global path, local costmap slice, and odometry, and one output topic, `/cmd_vel`. Switching controllers at run-time is as simple as changing a BT blackboard value or reloading a YAML file, allowing a robot to cruise with MPPI in open areas, drop to DWB in crowded corners, and finish with a precise RPP dock.

Recovery Behaviours and Path Smoothing

Even well-tuned planners and controllers can stall when the environment changes faster than sensors update, when a human blocks a corridor, or when wheel slip pushes the robot off its path. Nav 2 mitigates these situations through two plugin groups managed by the Recovery Server and the Smoother Server.

A recovery is a short, self-contained action that tries to restore progress. The default set includes

1. **Clear costmap** : reset unknown or suspect cells that may hide a false obstacle;
2. **Back up** : reverse a fixed distance to escape a dead end;
3. **Spin** : rotate in place to search for a new gap in the costmap.

Each recovery is a plugin, so other tactics, such as sending a notification, switching to teleop, and requesting help,can be added without touching core code. The behaviour tree triggers a recovery when the controller reports no forward progress for a configurable time or distance. If all recoveries fail, the BT returns `FAILURE`, and higher-level supervision can decide what to do next.

Global planners produce paths on a discrete grid, which often leads to stair-step corners or close passes to obstacles. Before the first control command is sent, the Smoother Server can refine the path:

- **Gradient-based smoothers** pull waypoints away from high-cost cells while shortening the route.
- **Spline fitters** replace sharp turns with cubic curves, reducing angular jerk.
- **Clearance maximisers** push the path outward until a target cost is met—useful for wide pallets or fragile loads.

Smoothing is optional and inexpensive, yet it can halve controller effort and improve passenger comfort on social robots.

The BT orders the servers as *plan* → *smooth* → *control*. If the controller later requests a re-plan, because a new obstacle blocks the route, the cycle repeats where the planner generates a fresh path, the smoother refines it, and control resumes. This loop continues until the goal checker declares the target pose reached.

Waypoint Following

Many applications require a robot to visit a series of poses rather than a single goal. Nav 2 provides the `nav2_waypoint_follower` node for this purpose.

A user or supervisor sends an ordered list of waypoints $\{\mathbf{p}_1, \ldots, \mathbf{p}_k\}$ to the node. For each waypoint $\mathbf{p}_i$ the node forwards a `NavigateToPose` action to the BT navigator, and monitors the feedback. If the goal is reached, advances to $\mathbf{p}_{i+1}$, However, if the navigator aborts, the waypoint follower aborts the entire list and reports the last valid pose.

The same plugin interface used by planners and controllers lets developers attach additional behaviours, such as taking a picture, reading an RFID tag, and waiting for user input, between waypoints with minimal code.

Two deployment models can be utilized:

Smart dispatcher, simple robot: A cloud or edge server assigns waypoint sets that encode one unit of work. The robot executes the list and reports completion; the dispatcher handles task scheduling, battery limits, and traffic rules.

Smart robot, light dispatcher: On-board autonomy carries more logic. A custom behaviour tree built with `nav2_behavior_tree` can mix navigation, battery checks and user interaction, requesting new work only when local goals finish. This model reduces network traffic and keeps the robot productive even during brief communication outages.

3.2.3 Supporting Tools for SLAM and Navigation

An effective SLAM–navigation workflow in ROS 2 depends not only on algorithm nodes but also on an ecosystem of modelling, visualisation, and simulation utilities that shorten the design–test cycle. The most prominent instruments are RViz 2 for on-line visual analytics, Gazebo for closed-loop physics simulation, and the Unified Robot Description Format (URDF)—together with its macro language `xacro`—for kinematic and inertial modelling of the vehicle itself.

RViz 2: on-line visual analytics. RViz 2 is a Qt/OGRE application that subscribes to arbitrary DDS topics and renders their contents in real-time. A typical SLAM session overlays the raw LiDAR point cloud, the incrementally built occupancy grid, the pose graph, and the robot footprint, all anchored by TF 2 so that any frame of reference can be inspected on demand. Interactive markers allow the operator to reset global localisation by dropping a new `/initial_pose`, to send a goal pose that triggers the Nav 2 behaviour tree, or to edit the costmap by erasing false positives. Because RViz 2 stores all display parameters in a YAML config file, an identical dashboard can be committed to version control and re-used across HIL and field tests, ensuring that anomalies observed on the bench are reproduced when the robot is deployed in the warehouse aisle.

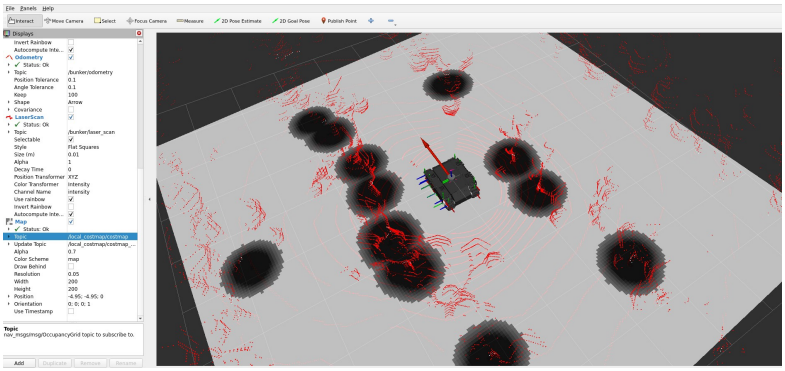


Figure 3.9: RViz2 Visualization of the Agilex Bunker Mini in an Orchard Field.

Gazebo and Ignition-Gazebo: deterministic physics in the loop. Before a ground vehicle drives through an indoor corridor or an outdoor yard, its SLAM and navigation stack can be stress-tested in Gazebo Classic or the new Ignition-Gazebo simulator. The physics engine—ODE, Bullet, or DART—solves rigid-body dynamics at a fixed time step, while sensor plugins inject realistic latency and noise into virtual LiDAR, IMU, and camera messages. The `gazebo_ros2_control` bridge exposes the simulator's joint interfaces as ROS 2 controllers, so the same velocity commands that drive real wheel motors actuate the virtual chassis. Recording the simulated run with `ROS 2 bag` produces a time-synchronised data set that can be replayed offline through the mapping pipeline, giving deterministic A/B comparisons of parameter sets. In HIL experiments, Gazebo can operate in *hardware* mode, where the physical wheel encoders and IMU feed the state estimator while synthetic LiDAR scans come from the simulator, allowing gradual ramp-up from pure simulation to full hardware deployment.

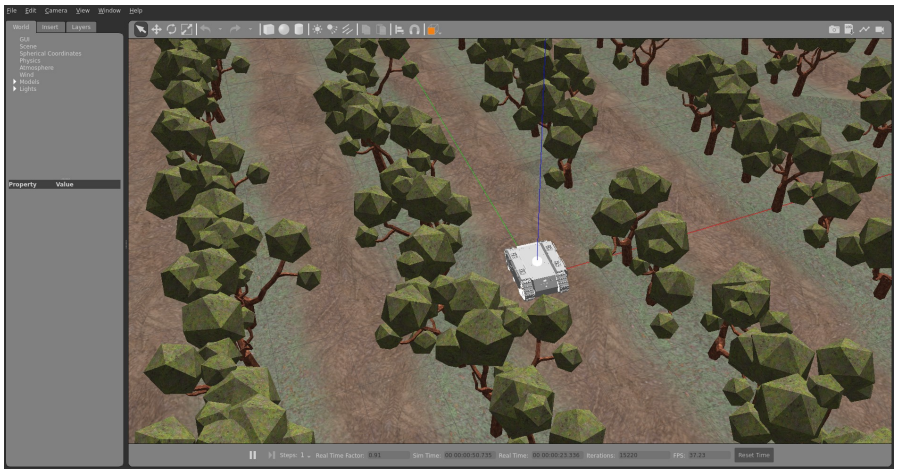


Figure 3.10: Gazebo Ignition Simulation in an Orchard Field.

URDF and `xacro`: a single source of kinematics. Accurate navigation demands an accurate model of the robot. ROS 2 adopts the Unified Robot Description Format, an XML schema that encodes link geometry, joint limits, inertial tensors, collision hulls, and visual meshes. The model is parsed by both Gazebo and RViz 2, ensuring that the simulated dynamics and the visualised footprint match the actual hardware. Large robots often reuse sub-assemblies, so URDF files are usually generated from `xacro` macros; parameterised macros let the same description instantiate, for example, a four-wheel differential platform for indoor trials and a track-drive variant for outdoor tests, simply by toggling a launch argument. During run time the `robot_state_publisher` reads the URDF and broadcasts the fixed transforms between links, closing the TF chain required by SLAM and Nav2. Because the inertial properties in URDF feed directly into Gazebo's dynamics solver, errors in mass or inertia reveal themselves as pose-graph drift or controller saturation in the simulator long before the robot reaches the factory floor.

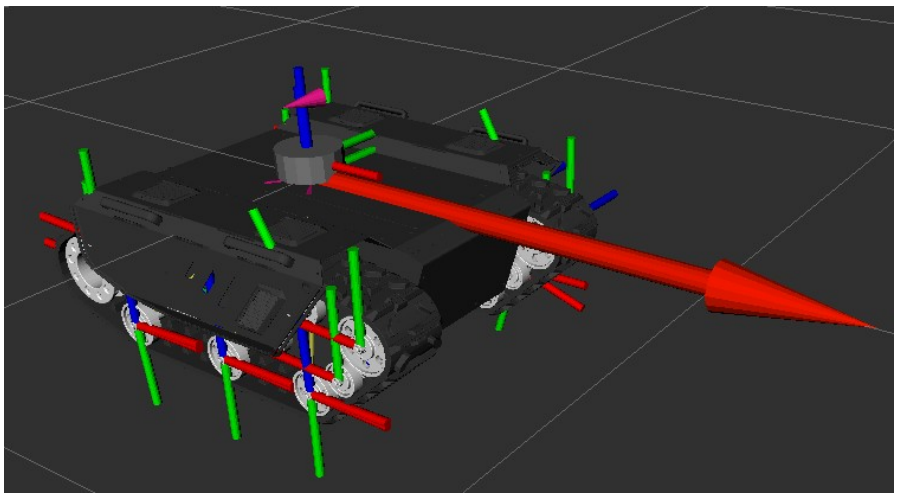


Figure 3.11: URDF Description of the Agilex Bunker Mini.

Chapter 4

Hardware Architecture

4.1 AgileX Scout Mini

The AgileX Scout Mini [25] is a compact and high-performance unmanned ground vehicle (UGV) developed by AgileX Robotics. It is widely used in research and development, robotics education, autonomous navigation, and inspection tasks. Its relatively small dimensions and lightweight build make it easy to transport and deploy, while its high maneuverability and robust chassis allow it to operate effectively in complex terrain.

The robot typically weighs between 18–25 kg, depending on its configuration, and can support a payload of approximately 10 kg. It measures around 500 mm long, 450 mm wide and 250 mm high. The *Scout Mini* can achieve speeds of up to 3–4 m/s and can operate continuously for about 1.5–2 h on a single battery charge. It is powered by a lithium-ion battery pack, usually rated at 24 V with a capacity ranging from 15–30 Ah. Charging the battery takes approximately 2–4 h.

Figure 4.1: The Agilex Scout Mini AMR.

In terms of locomotion, the Scout Mini is equipped with a four-wheel differential drive system. Each of its four wheels is driven by a high-torque DC gear motor and is equipped

with encoders to provide odometry data. The wheels are typically made of solid rubber or pneumatic material, offering a deep tread that enables traction on various surfaces. The chassis incorporates a passive suspension system that provides each wheel with independent damping, ensuring stability and shock absorption during movement across uneven terrain.

The Scout Mini is sensor-ready and supports the integration of a wide variety of sensors depending on the user's needs. In a typical setup, it includes a 3D LiDAR sensor for mapping and obstacle detection. An Inertial Measurement Unit (IMU) is also included for estimating orientation and aiding in localization. Wheel encoders provide odometry feedback, while GPS modules—often with Real-Time Kinematic (RTK) capabilities—are used for precise global positioning in outdoor applications. Additional sensors such as stereo cameras, depth cameras (e.g. Intel RealSense or ZED), and ultrasonic or time-of-flight sensors may also be integrated depending on the specific application.

The robot utilizes a range of communication protocols to enable seamless interaction between its components. The internal motor controllers and embedded systems communicate primarily through the Controller Area Network (CAN) bus. For peripheral sensors such as GPS and IMU modules, UART or other serial interfaces are used. High-bandwidth sensors like LiDAR and cameras typically use Ethernet or USB interfaces. For external communication and remote control, the system may include Wi-Fi or 4G modules. The Scout Mini is fully compatible with the Robot Operating System (ROS), including both ROS 1 and ROS 2 distributions, which allows developers to leverage a wide ecosystem of packages for navigation, perception, and simulation.

The robot includes multiple I/O ports and connectors to facilitate hardware integration and debugging. These typically include regulated power outputs at 5 V and 12 V, USB ports, Ethernet ports, CAN bus terminals, and UART interfaces. Some models also provide general-purpose input/output (GPIO) pins via expansion boards. An emergency stop (E-stop) button is also included for safety during operation and testing.

Figure 4.2: Sensor Modules.

Regarding computational capability, the Scout Mini is often equipped with powerful onboard computers tailored for AI and robotics applications. The NVIDIA Jetson series—such as the Jetson Nano, Xavier NX, or Jetson Orin—is commonly used to provide GPU acceleration for machine-learning tasks. Alternatively, some configurations utilize Intel NUCs or similar x86-based mini PCs for higher general-purpose processing power. The onboard computer typically runs Ubuntu Linux and ROS, and it interfaces with the motor controller, sensors, and communication modules through standard protocols.

From a software perspective, AgileX provides an open-source software development kit (SDK) along with support for ROS integration. This allows users to control the robot through ROS nodes, implement SLAM algorithms, perform autonomous navigation, and integrate object detection and tracking using AI models. Visualization tools such as `rviz` and `rqt` can be used for debugging and monitoring robot performance in real time. Furthermore, the Scout Mini has support for simulation in environments such as Gazebo, Ignition, and Webots, allowing researchers to develop and test algorithms in a virtual space before deploying them to the physical platform.

4.2 AgileX Bunker Mini

The AgileX Bunker Mini [26] is a compact and rugged tracked unmanned ground vehicle (UGV) developed by AgileX Robotics. It is engineered specifically for environments where high traction, stability, and the ability to traverse challenging terrain are essential. As a member of the *Bunker* series, the Bunker Mini is characterized by its robust construction and its differential tracked drive system, which grants it exceptional maneuverability in both indoor and outdoor settings, including sand, gravel, snow, mud, and stairs.

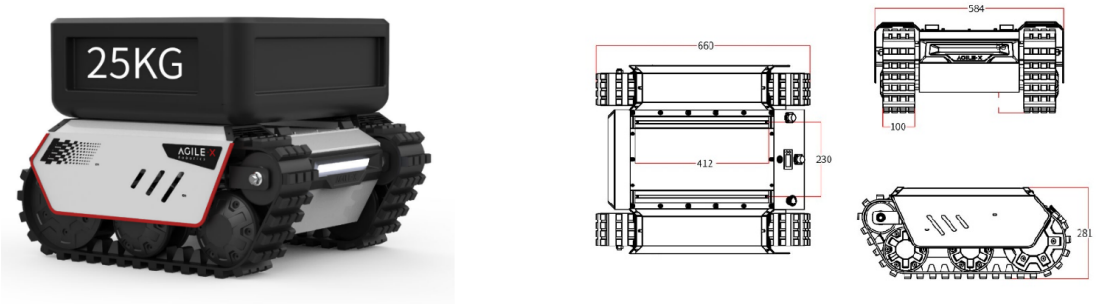


Figure 4.3: The Agilex Bunker Mini Drawings and Payload.

This mobile platform has been optimized for research, inspection, exploration, and autonomous robotic applications in environments where wheeled platforms might encounter limitations. Its form factor is compact, with dimensions typically around 700 mm in length, 570 mm in width, and 270 mm in height. The vehicle weighs approximately 40-50 kg, depending on the specific configuration and sensors used, and it can carry a payload of up to 20 kg, making it suitable for transporting additional sensors, computing units, or mission-specific tools.

One of the primary mechanical distinctions of the Bunker Mini is its tracked locomotion system. It features a pair of high-strength rubber tracks driven by powerful brushless DC motors with integrated encoders. This configuration enables the robot to perform zero-radius turning and maintain excellent traction on irregular surfaces. The tracks are tensioned and supported by a suspension mechanism that helps absorb shocks from rough terrain, improving both stability and the safety of sensitive onboard equipment. The robot is capable of climbing slopes up to 36° and can overcome vertical obstacles such as curbs or low steps up to approximately 15 cm in height.

Figure 4.4: The Agilex Bunker Mini.

The Bunker Mini is designed to support a wide array of sensors and onboard computational hardware. Standard configurations typically include a GPS module for outdoor localization, an IMU for orientation estimation, and quadrature encoders for motion feedback. The robot also supports 2D or 3D LiDAR sensors for environmental mapping, as well as RGB-D cameras or stereo vision systems for perception tasks. These sensors enable applications such as SLAM, object detection, and autonomous navigation. Depending on the user's requirements, other sensors—such as thermal cameras, ultrasonic sensors, or radar units—can also be integrated.

In terms of communication protocols, the Bunker Mini is equipped with multiple interfaces to allow seamless integration and data exchange between various system components. Internal communication between the motor controllers and the onboard microcontroller is managed through a Controller Area Network (CAN) bus, which offers high reliability and fault tolerance. Peripheral sensors may connect through UART, USB, or Ethernet ports, depending on their data bandwidth and power requirements. Wireless communication can be established via Wi-Fi or optional 4G modules, allowing for remote operation and monitoring. The robot is fully compatible with ROS, supporting both ROS 1 and ROS 2, which significantly simplifies sensor integration, algorithm development, and deployment.

The onboard computing system of the Bunker Mini typically consists of an embedded AI-capable computer such as an NVIDIA Jetson Nano, Xavier NX, or Jetson Orin, providing GPU acceleration for machine-learning applications and real-time data processing. These computing units run Ubuntu Linux and are pre-configured with drivers and software interfaces for motor control, sensor data acquisition, and system diagnostics.

Software support for the Bunker Mini includes open-source ROS packages, simulation models (`URDF` and `Xacro` files), and drivers for common sensors. This facilitates the deployment of algorithms such as autonomous waypoint navigation, obstacle avoidance, visual SLAM, and multi-agent coordination. The robot can also be simulated in environments like Gazebo or Ignition, allowing developers to prototype and validate their systems before deploying them on real hardware.

In addition to autonomous control via onboard computing and network communication, the Bunker Mini also includes a dedicated wireless remote controller for manual teleoperation. This remote allows a human operator to directly command the robot's movement, which is particularly useful during testing, transportation, or when autonomy is not active. The controller communicates with the robot via a radio-frequency (RF) link, offering a stable, low-latency channel for direct motion commands. This manual control option is essential for safe startup, diagnostics, or emergency intervention in the field, especially when operating in unstructured or unpredictable environments.

Figure 4.5: Schematic Diagram of the Remote Controller for Manual Teleoperation.

4.3 Sensors and Communication Interface

4.3.1 RoboSense - Helios LiDAR

The RoboSense RS-Helios [27] is a high-performance three-dimensional LiDAR sensor developed by RoboSense Technology, designed primarily for autonomous vehicles, mobile

robotics, and intelligent perception systems. It belongs to the Helios series, which represents a class of compact, reliable, and cost-effective multi-beam LiDARs tailored for both industrial applications and academic research in robotics, SLAM (Simultaneous Localization and Mapping), and environment perception.

The RS-Helios employs a rotating mechanical structure that uses multiple laser channels to generate a dense and accurate 3D point cloud of its surroundings. The standard model typically features 32 laser beams arranged vertically, although variations with fewer or more channels also exist depending on the specific product version. The sensor operates by emitting laser pulses and measuring the time-of-flight (ToF) of the reflected signal from surrounding objects, which allows for precise distance measurement and three-dimensional scene reconstruction.

In terms of performance, the RS-Helios offers a vertical field of view of approximately 31.9° with a vertical angular resolution of around 1 degree. Its horizontal field of view spans a full 360° due to its continuous rotation, with selectable rotation speeds ranging from 5 Hz to 20 Hz. At a standard rotation rate of 10 Hz, the sensor can achieve an effective output of up to 640,000 points per second. The maximum detection range is approximately 150 m for high-reflectivity targets and around 70 m for 10% reflectivity objects under typical conditions, which makes it suitable for both close-range mapping and long-range obstacle detection.

Figure 4.6: The RoboSense Helios LiDAR.

The sensor outputs data via a Gigabit Ethernet interface, which provides a high-bandwidth connection suitable for real-time point cloud streaming. The data packets are transmitted over UDP datagrams and follow a well-documented protocol provided by Robosense, allowing seamless integration with robotics frameworks such as ROS. Robosense provides official ROS drivers and visualization tools for both ROS 1 and ROS 2, along with APIs for C++ and Python, making it accessible to researchers and developers.

A key advantage of the RS-Helios is its built-in calibration and synchronization features. The sensor is factory-calibrated and includes time synchronization options via GPS or Precision Time Protocol (PTP), which is essential for fusing LiDAR data with IMU, GNSS,

and other perception sensors in SLAM and sensor fusion tasks.

Figure 4.7: The Agilex Bunker Mini with RS-Helios LiDAR.

In robotics applications, the RS-Helios has proven to be an effective tool for three-dimensional environmental mapping, obstacle avoidance, terrain analysis, and localization. It is frequently employed in outdoor autonomous ground vehicles (AGVs), unmanned aerial vehicles (UAVs), and industrial automation systems. Its fine angular resolution and high-density point cloud output make it suitable for detecting small or complex objects in cluttered scenes, while its long-range capability ensures the robot can perceive and plan in large, open environments.

4.3.2 Intel RealSense Depth Camera

The Intel RealSense depth camera series [28] is a family of advanced 3D vision sensors developed to provide high-resolution depth perception capabilities for applications in robotics, computer vision, augmented reality, and autonomous systems. These cameras are compact, lightweight, and equipped with active stereo or time-of-flight depth-sensing technology, making them particularly suitable for mobile robotic platforms, drones, and human-machine interaction systems.

Figure 4.8: Intel RealSense Depth Camera.

The RealSense cameras connect to the host system via a USB 3.0 interface, ensuring high-bandwidth data transmission required for streaming synchronized RGB and depth frames. They are compatible with a wide range of operating systems, including Windows, Linux, and Android. Intel provides an open-source RealSense SDK 2.0, which supports C++, Python, and ROS (Robot Operating System) environments. The SDK includes tools for accessing raw and processed data streams, aligning depth with color images, and generating point clouds. Official ROS and ROS 2 drivers are available, making integration with robotic systems straightforward.

RealSense cameras are also equipped with motion tracking capabilities through optional IMU sensors, depending on the model. This feature is particularly beneficial for mobile robots, drones, and SLAM applications where understanding the spatial orientation and movement of the camera is crucial. The data from the depth sensor, RGB camera, and IMU can be fused to perform tasks such as visual odometry, obstacle avoidance, object detection, and semantic scene understanding.

4.3.3 CAN Protocol

The Controller Area Network (CAN) protocol is a robust and widely adopted communication standard designed for reliable real-time data exchange between electronic control units (ECUs) or microcontrollers within embedded systems. CAN has become a standard communication protocol in a broad range of industries including robotics, industrial automation, medical equipment, and aerospace due to its high reliability, deterministic timing, and efficient message handling.

The fundamental design goal of CAN is to enable multiple devices to communicate over a shared two-wire bus without the need for a central host computer. This bus-based architecture is well-suited for distributed control systems where numerous subsystems must coordinate efficiently. CAN is a multi-master protocol, meaning that any node connected to the bus can initiate communication if the bus is free. This makes it highly suitable for decentralized systems such as autonomous robots, where multiple components including motor controllers, sensors, and safety modules must exchange data independently and in real time.

In the context of robotics, CAN is frequently used to connect motor drivers, encoders, IMUs, and other real-time components to the central processing unit. Because of its real-time capabilities, it allows for low-latency and deterministic control signals, which are crucial in applications requiring precise motion control and synchronized sensor feedback. Many modern robotic platforms, including those developed by AgileX Robotics, rely on CAN for internal communication between drive systems, power management boards, and low-level control units.

Figure 4.9: CAN communication connector.

One of the key advantages of CAN in robotics is its scalability. Up to 110 nodes can be connected on a single CAN bus (depending on the physical layer and system layout), and the wiring is relatively simple, consisting of just two main wires with termination resistors at each end. The protocol supports broadcast communication, allowing all nodes to receive messages simultaneously, which is advantageous for synchronized tasks or centralized monitoring.

4.4 Computer boards and Router

4.4.1 NVIDIA Jetson Nano

The Jetson Nano is an entry-level module in the Jetson series that delivers GPU-accelerated computing at a low cost and with minimal power consumption. It is designed to enable real-time AI inference on compact and power-constrained robotic platforms. The Nano features a Quad-core ARM Cortex-A57 CPU and a 128-core Maxwell GPU, capable of running parallel computations and deep learning models with modest complexity.

Despite its compact size and affordability, the Jetson Nano can run full versions of Ubuntu Linux and supports the JetPack SDK, which includes libraries and tools for AI, computer vision, and GPU-accelerated computing. The system is compatible with popular AI frameworks such as TensorFlow, PyTorch, OpenCV, and ROS. This makes it an accessible platform for researchers and developers who are working on tasks such as object detection, image segmentation, and gesture recognition in real time.

The Jetson Nano includes multiple I/O interfaces including USB 3.0, HDMI, MIPI CSI-2 camera interfaces, Gigabit Ethernet, I2C, SPI, and UART, which facilitate integration with a wide array of sensors and peripherals. It typically consumes between 5 to 10 watts of power, making it highly suitable for battery-powered mobile robots where energy efficiency is a key consideration.

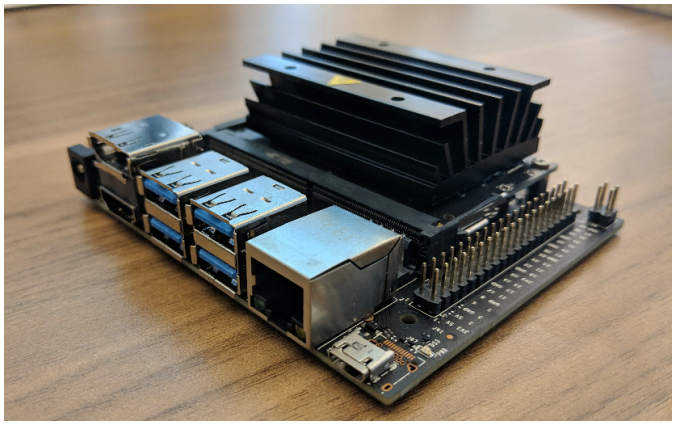


Figure 4.10: Nvidia Jetson Nano.

4.4.2 NVIDIA Jetson Xavier NX

The Jetson Xavier NX is a significantly more powerful module within the same Jetson family, offering performance suitable for more demanding robotics applications such as high-speed autonomous navigation, real-time SLAM, and AI-based perception in dynamic environments. It features a 6-core ARM Cortex-A57 processor and a 384-core Volta GPU with 48 Tensor Cores, allowing it to perform deep learning inference and high-performance computer vision tasks with exceptional speed and accuracy. In addition, it includes 8 or 16 GB of LPDDR4x memory and supports high-speed NVMe SSD storage through PCIe interfaces.

The Xavier NX is capable of achieving up to 21 TOPS (trillions of operations per second) of AI inference performance while consuming as little as 10-15 W of power, depending on the configuration. This balance between performance and energy efficiency is crucial for field-deployed autonomous robots that must process high-bandwidth sensor data—such as LiDAR point clouds, high-resolution video streams, and IMU readings—in real time.

Similar to the Nano, the Xavier NX is fully compatible with the JetPack SDK and supports major AI and robotics frameworks. It offers a wide variety of I/O interfaces including USB 3.1, MIPI CSI, Gigabit Ethernet, PCIe, UART, and CAN bus, making it versatile for integration with advanced sensors, motor controllers, and communication systems. The module also supports hardware acceleration for deep learning, image processing, and video encoding, which allows it to manage complex multi-modal sensor fusion in autonomous systems.

The Xavier NX is frequently used in research platforms, commercial robotic systems, and autonomous mobile vehicles that require high computational performance at the edge. Its small form factor and rugged design also make it suitable for deployment in harsh environments where real-time decision-making and low-latency processing are essential.

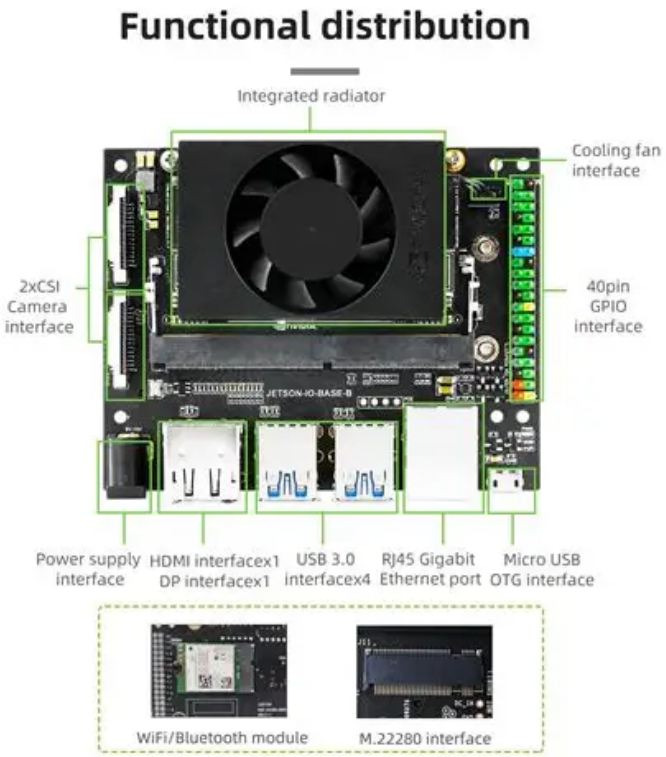


Figure 4.11: Nvidia Jetson Xavier NX Pinout.

4.4.3 Teltonika RUTX50 Router

The Teltonika RUTX50 [29] is a high-performance industrial-grade cellular router engineered to provide robust and secure internet connectivity alongside real-time positioning capabilities for advanced applications such as robotics, autonomous systems, smart transportation, and industrial automation. As part of Teltonika's flagship RUTX series, the RUTX50 offers cutting-edge 5G mobile communication, combined with multi-constellation GNSS support, ensuring both high-speed data exchange and precise geographic localization in a single, compact device.

The standout feature of the RUTX50 is its support for 5G NR (New Radio) cellular networks, with backward compatibility to 4G LTE and 3G, thereby ensuring seamless operation even in regions with evolving infrastructure. The router supports both Standalone (SA) and Non-Standalone (NSA) 5G architecture. These capabilities make the RUTX50 ideal for bandwidth-intensive robotic applications that involve real-time streaming, cloud interaction, or remote monitoring.

Complementing its high-speed communication features, the RUTX50 includes an integrated GNSS receiver compatible with multiple satellite constellations, including GPS, GLONASS, Galileo, and BeiDou. This enables the device to offer high-accuracy positioning, fast satellite fix times, and enhanced signal stability, particularly in environments where signal obstructions or reflections may occur such as urban canyons or forested areas. The

GPS module supports active antennas through an external SMA connector, ensuring better signal amplification and sensitivity.

The GNSS receiver in the RUTX50 provides real-time geographic coordinates, altitude, speed, and heading data, output in the standardized NMEA 0183 format. This stream can be accessed through the router's internal web interface or transmitted to external processors over serial interfaces, USB, or TCP/UDP sockets, depending on the application needs. In robotic systems, this GPS data is often used in conjunction with IMU readings and odometry sensors to perform sensor fusion, enabling precise localization, mapping, and path planning.

When paired with RTK correction services, which the router can receive over the cellular network via an NTRIP client, the GNSS module is capable of achieving centimeter-level accuracy. This level of precision is particularly critical in outdoor autonomous mobile robots, agricultural robots, and industrial inspection systems, where navigation errors must be minimized to avoid collisions or mission drift.

The RUTX50 also supports a broad array of I/O and interface options, including 5 Gigabit Ethernet ports, dual-SIM support, USB 2.0, digital I/Os, RS232/RS485 serial communication, and Bluetooth Low Energy. These features allow the router to act not only as a communication gateway but also as an integration hub, interfacing with motor controllers, onboard computers, and external sensors on robotic platforms. The router runs RutOS, a secure and flexible Linux-based operating system that supports advanced networking features such as VPN tunneling, firewall protection, remote configuration, cloud connectivity, and protocol support including MQTT, Modbus, and SNMP.

Figure 4.12: Teltonika RUTX50 Router.

Chapter 5

Methodology

This section outlines the procedures followed to implement SLAM and autonomous navigation in both simulated and real-world environments. The initial phase focused on simulation, involving two robot models: the Scout and the Bunker. The Scout model was tested exclusively in an indoor environment modeled after an Amazon warehouse, while the Bunker model was utilized in a simulated outdoor agricultural field. Following the successful implementation of these simulation scenarios, the next phase involved the hardware setup and real-world deployment, which was carried out exclusively on the Bunker Mini platform. The subsequent subsections detail the hardware configuration and operational procedures used in this real-world application.

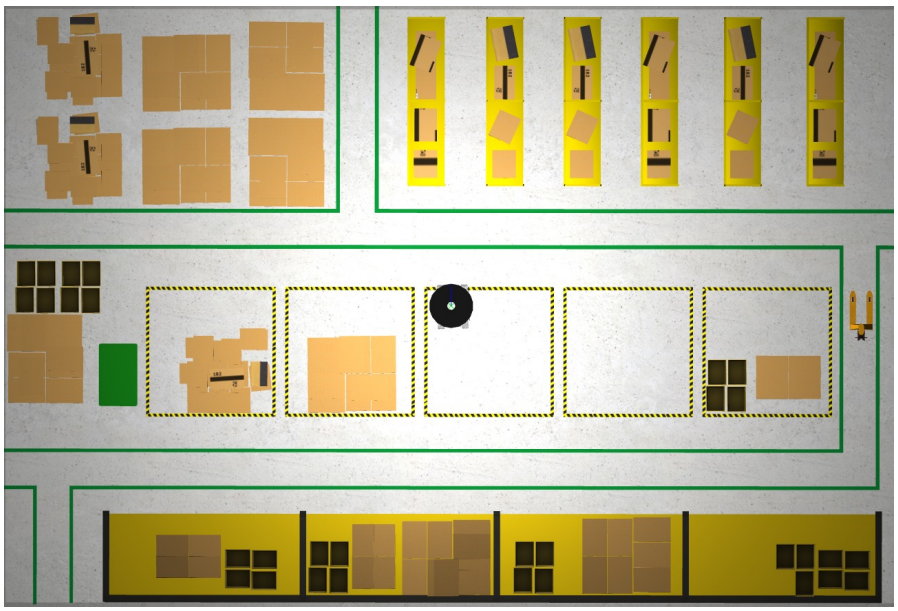


Figure 5.1: Amazon Warehouse Simulation Environment in Gazebo Ignition.

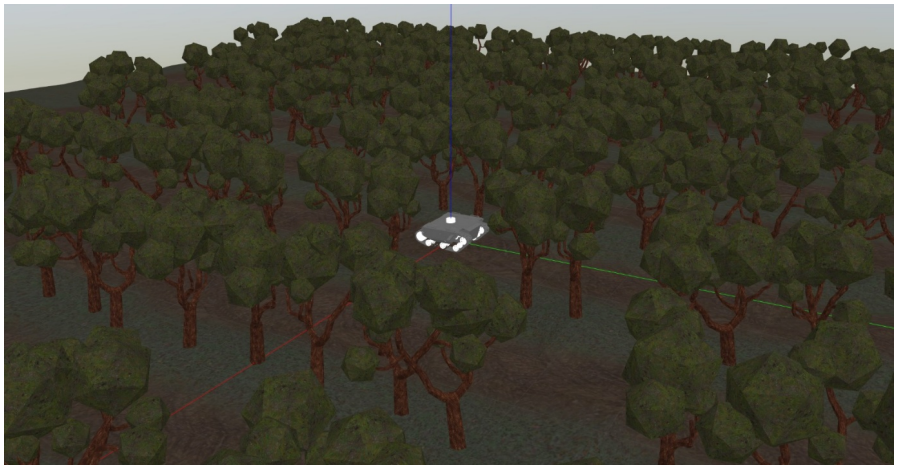


Figure 5.2: Orchard Field Simulation Environment in Gazebo Classic.

5.1 Software in the Loop (SIL)

At the initial stage of the project, the simulation of the Agilex Scout robot was conducted within a virtual Amazon warehouse environment. The primary objective was to enable the robot to navigate autonomously using the Nav 2 stack. This was achieved through a two-phase process: first, a map of the warehouse was generated using the SLAM Toolbox; then, the generated map was utilized for navigation with the Adaptive Monte Carlo Localization (AMCL) algorithm. This setup allowed the robot to follow user-defined waypoints on the map while effectively avoiding obstacles. The localization process relied on essential onboard sensors such as a LiDAR and wheel encoders, which are critical for AMCL-based localization accuracy.

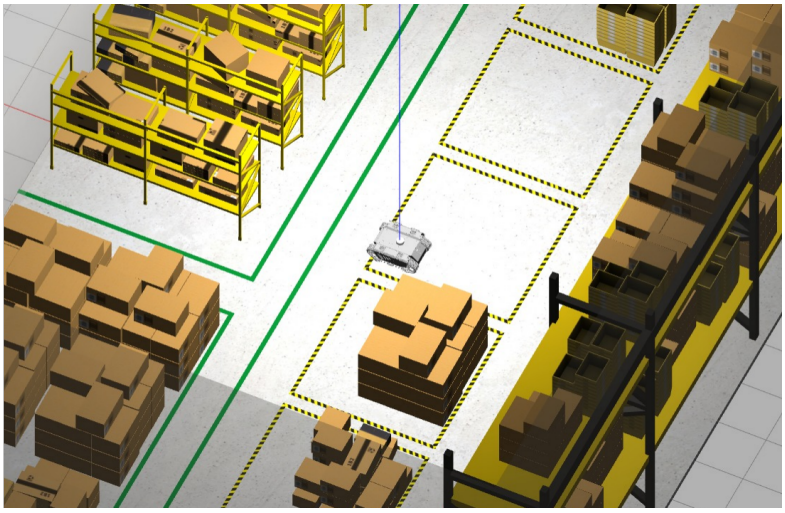


Figure 5.3: The Agilex Bunker Mini in Amazon Warehouse Simulation Environment.

As illustrated in the following figures, the process of generating the warehouse map is presented step by step. The sequence begins with the initial stages of mapping, followed by the completion of the full map. Subsequently, the generated global and local costmaps are shown, highlighting the robot's perception of the environment for effective navigation.

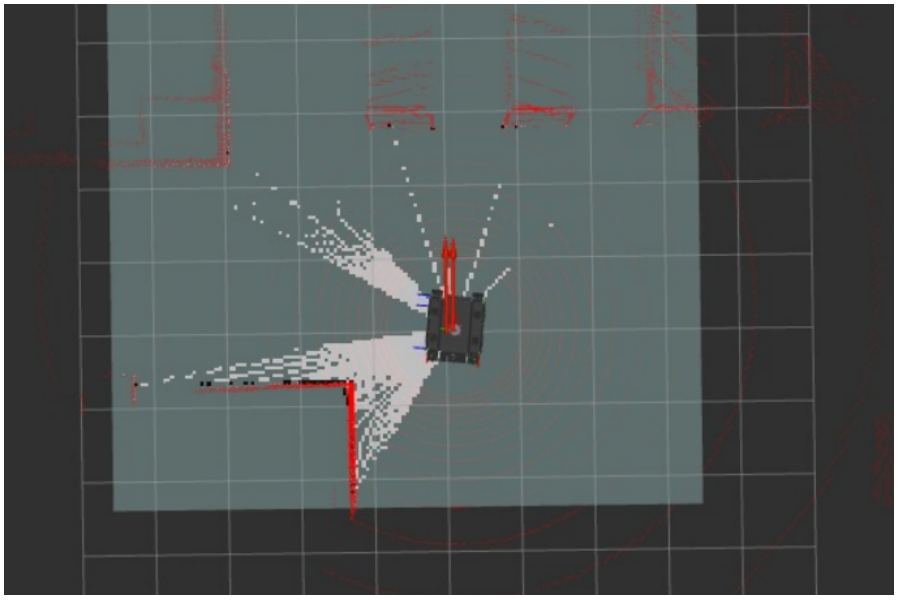


Figure 5.4: Initial Map generation using SLAM Toolbox.

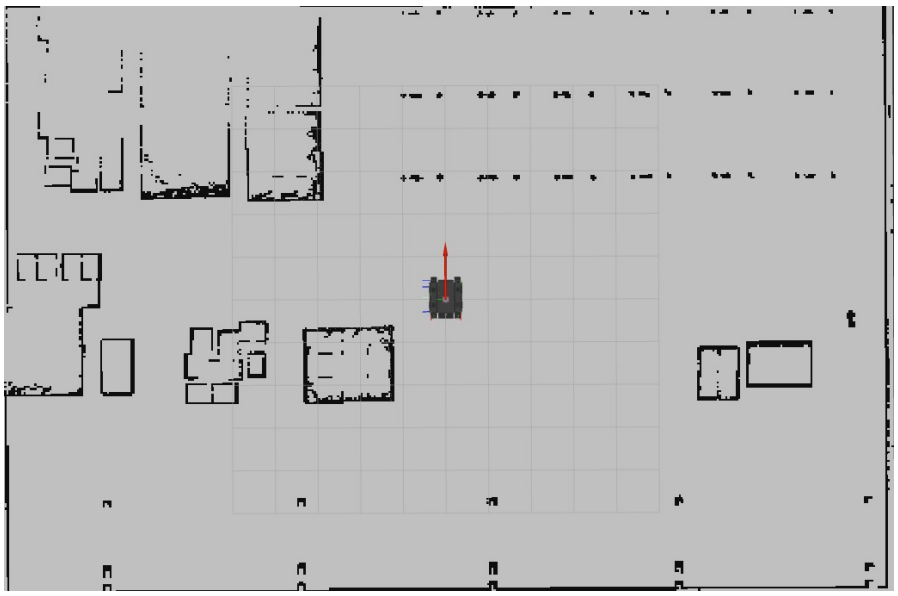


Figure 5.5: Final Map of the Amazon Warehouse generated using SLAM Toolbox.

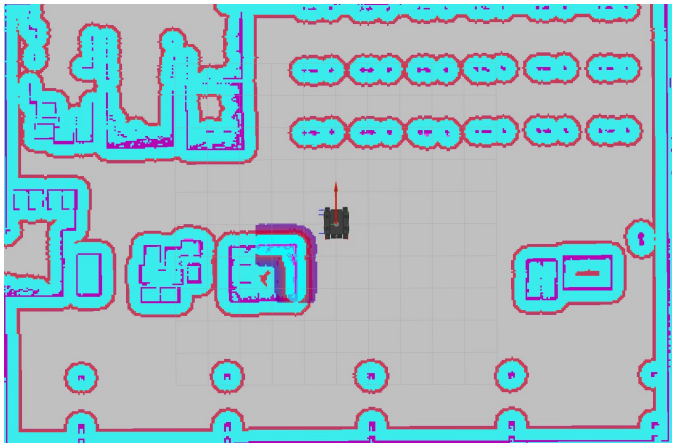


Figure 5.6: Final Map showing both Global and Local Costmaps.

Following the successful implementation in the warehouse scenario, the next phase focused on simulating the Agilex Bunker model in an outdoor environment, specifically in a modeled orchard field. Although the general workflow remained similar to the indoor case, notable differences emerged. The outdoor field's larger scale introduced challenges in maintaining reliable map generation, as the SLAM process would occasionally fail after extended operation. To address this, several parameter adjustments were made within the SLAM Toolbox configuration to improve performance. In addition, the Cartographer SLAM algorithm was also employed to generate the map of the environment, which proved to be more robust in the outdoor scenario, successfully producing a complete and usable map for navigation purposes. Additionally, due to the nature of the outdoor environment, GPS-based localization was considered and integrated as a complementary approach. Further functionalities from the Nav 2 stack such as waypoint following and behavior tree customization were employed to enhance autonomous navigation capabilities, as previously discussed in earlier chapters.

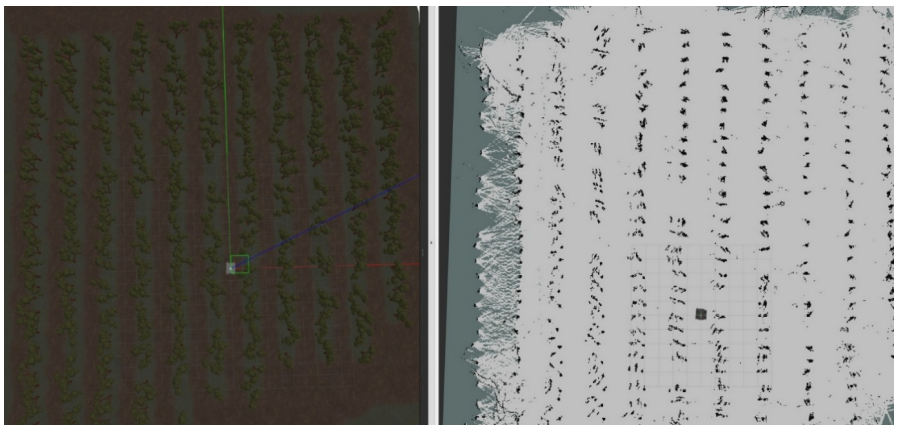


Figure 5.7: Final Map of the Orchard Field generated using SLAM Toolbox.

As illustrated in the figures, the map of the orchard field was first generated and then refined using a graphical editing tool. This post-processing step was essential to ensure that the robot could accurately detect obstacles and navigate around them effectively. The final two images display the global and local costmaps, each presented in different layouts, highlighting the robot's perception of the environment for safe and efficient path planning.

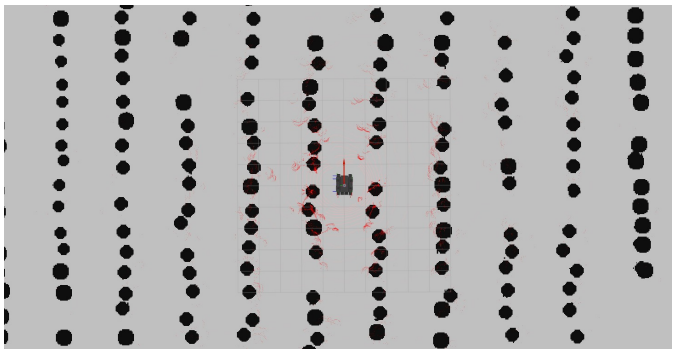


Figure 5.8: Refined Final Map of the Orchard Field.

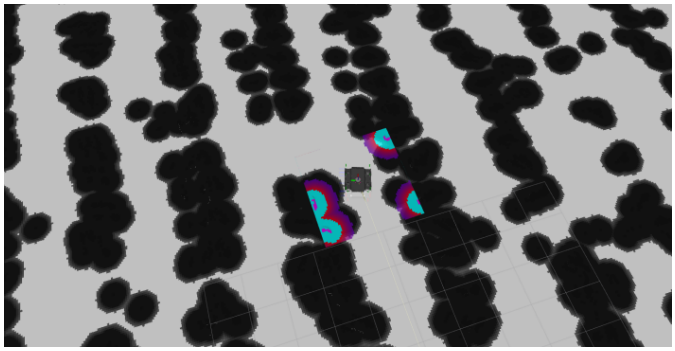


Figure 5.9: Final Map of the Orchard Field with Global and Local Costmaps.

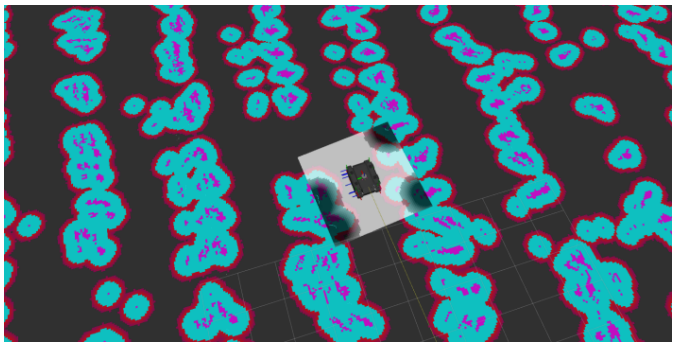


Figure 5.10: Final Map of the Orchard Field with Global and Local Costmap layouts.

In both scenarios, the robot was able to navigate autonomously through user-defined waypoints while effectively avoiding obstacles. This capability was demonstrated even in the more challenging orchard environment, which featured a large-scale map and a high density of obstacles.

5.1.1 LIO-SAM Implementation

In addition to using the SLAM Toolbox for generating 2D maps, a real-time LiDAR–inertial odometry package was employed to produce a 3D map of the environment within the simulation. As illustrated in the following figures, the progression of the 3D map is shown from the robot's initial position to the end of the row, capturing the spatial evolution of the environment in detail. It is important to note that the LiDAR sensor used in the simulation was the default model provided by Gazebo Classic.

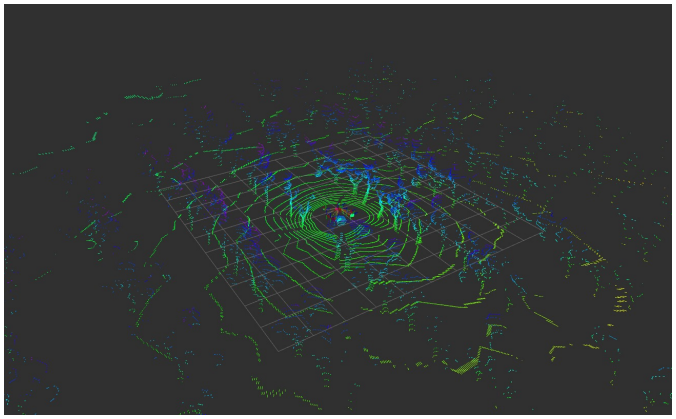


Figure 5.11: Initial 3D Map generation using LIO-SAM.

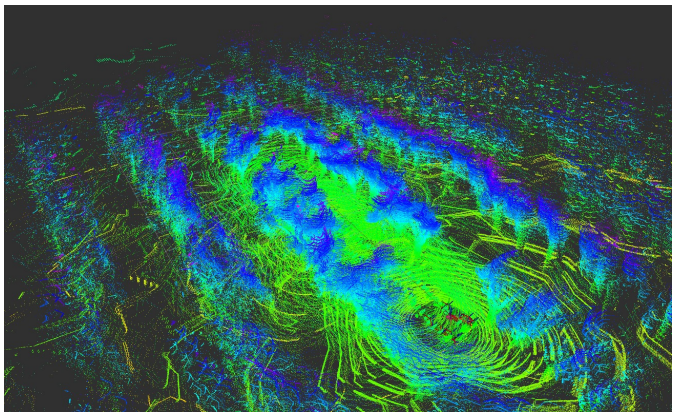


Figure 5.12: Progression of 3D map generation using LIO-SAM.

To achieve more accurate and realistic mapping results, future work should consider

integrating simulation-ready models of real-world LiDARs, such as those from Velodyne or Ouster. Incorporating these models would enhance the performance of the simulation and bring it closer to a real-world scenario.

5.2 Hardware in the loop (HIL)

In the real-world implementation, autonomous navigation was tested for the first time on the Bunker Mini model in both indoor and outdoor environments. The initial indoor mapping was conducted within an office space at the CIM4.0 company using the SLAM Toolbox, followed by autonomous navigation utilizing the AMCL algorithm. This configuration performed as expected, providing reliable localization and accurate path planning within the indoor setting.

Figure 5.13: Final map of the CIM4.0 office environment.

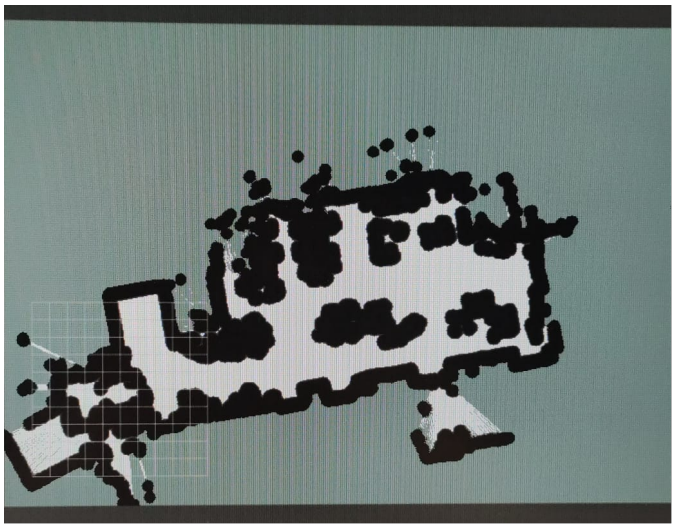


Figure 5.14: Final map of the CIM4.0 office environment with global costmap.

The figures illustrate the 3D point cloud data acquired from the LiDAR sensor as visualized in RViz2, along with the defined coordinate frames of the mobile robot.

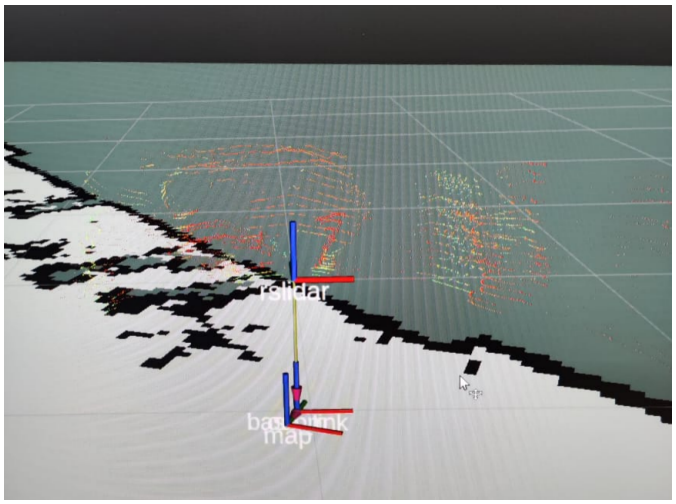


Figure 5.15: Defined coordinate frames of the robot in a real-world scenario.

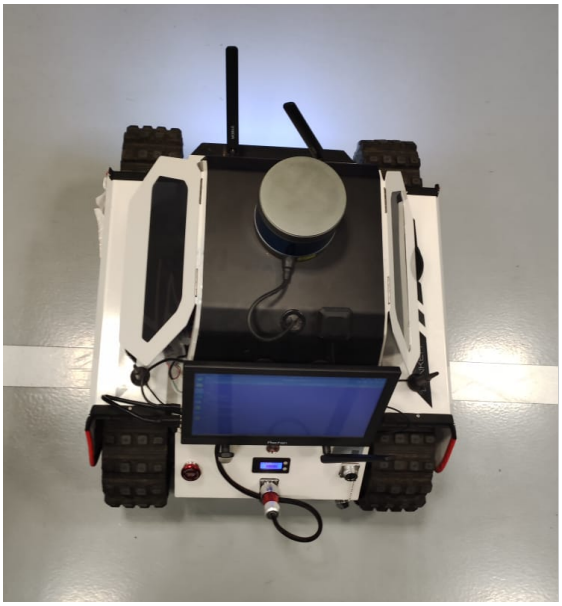


Figure 5.16: HIL Implementation of the AgileX Bunker Mini robot at the CIM4.0 office.

Moreover, mapping was extended to the outdoor area surrounding the building to evaluate GPS-based navigation in combination with the waypoint-follower approach. The initial outdoor map required refinement, so this was accomplished using the graphical

editor GIMP to enhance its accuracy and usability for detecting obstacles. It is noteworthy that the SLAM Toolbox successfully generated a large-scale outdoor map without any system failures, demonstrating its robustness and scalability for real-world applications.

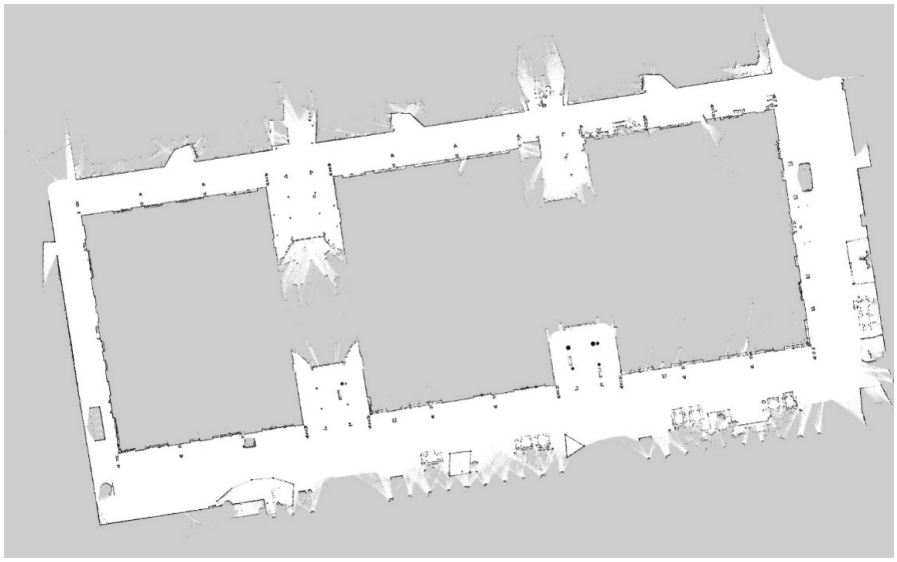


Figure 5.17: Final map of the outdoor environment generated using SLAM Toolbox.

In this case, different controllers available within the Nav 2 stack such as the DWB Controller, the MPPI Controller, and the Regulated Pure Pursuit (RPP) Controller were evaluated and compared. Among these, the RPP Controller demonstrated superior overall performance. For instance, it enabled the robot to execute turns more efficiently, whereas the other controllers, particularly the DWB Controller, exhibited difficulty in handling sharp turns. Additionally, in obstacle avoidance scenarios, the RPP Controller produced smoother and more stable trajectories around obstacles compared to the DWB and MPPI Controllers, resulting in more fluid and reliable navigation behavior.

To implement GPS-based navigation, it is essential to first obtain accurate and continuous GPS data from satellite signals. This data was received via a router as shown in the picture. In our setup, the available router was only capable of transmitting GPS data at a low frequency, limited to a maximum of 1 Hz, with a positional error of approximately 2.5 meters. As a result, the GPS-based navigation was not feasible in this configuration due to both the low update rate and insufficient positional accuracy.

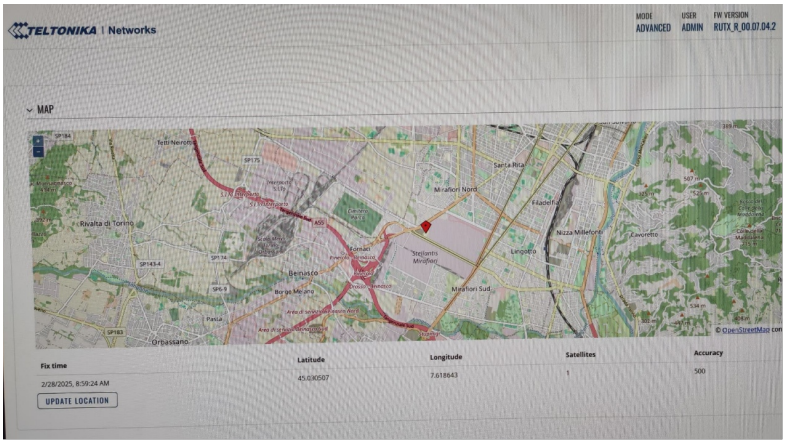


Figure 5.18: Verification of received GPS data, including latitude and longitude.

For robust and precise GPS navigation, a Real-Time Kinematic (RTK) GPS system is required. RTK GPS can provide data at much higher frequencies, up to 20 Hz, and offers significantly enhanced accuracy, with positional errors reduced to as little as 0.001 meters. Such performance is critical for real-time localization and reliable path tracking, especially in outdoor robotic applications.

Figure 5.19: HIL Implementation of the AgileX Bunker Mini robot in the outdoor environment.

Chapter 6

Conclusion and Future Work

Autonomous navigation in unknown, dynamic environments remains a central challenge in deploying mobile robots for industrial and agricultural tasks. This thesis addressed that challenge by designing, implementing, and experimentally validating a complete navigation pipeline for AGVs within the ROS 2 framework.

This was achieved through the implementation of the SLAM Toolbox for mapping and the AMCL algorithm for localization. The results demonstrated that the robot was capable of constructing a map of an unknown environment and autonomously navigating within it using the Nav 2 stack.

Another key goal of the work was to evaluate and compare the performance of different controllers within the Nav2 framework, including the DWB Controller, MPPI Controller, and RPP Controller. Experimental results in real-world scenarios revealed that the RPP Controller outperformed the others in terms of smoother obstacle avoidance and improved turning behavior, particularly when compared to the DWB Controller, which struggled with sharp turns.

In addition, the project aimed to implement waypoint-following navigation based on GPS data. While this was successfully done in the simulation environment, limitations in the real-world setup, specifically the lack of access to an RTK GPS, prevented high-accuracy, high-frequency GPS-based navigation. The standard GPS module used provided data at a low frequency (1 Hz) with a positional error of approximately 2.5 meters, which was insufficient for reliable autonomous navigation.

6.1 Future Work

To build on the results of this project, the following directions are recommended:

1. **Evaluation of Additional Controllers and Planners**
   Further evaluation of the additional controllers available within the Nav2 framework is recommended to determine the most effective option for specific use cases. In parallel, a comparative analysis of global planners focusing on criteria such as computational efficiency, path optimality, and smoothness would provide valuable insights for

selecting the most suitable planning strategy tailored to different environmental conditions and navigation requirements.

2. **Integration of RTK GPS for Outdoor Navigation**
   For reliable GPS-based navigation in outdoor scenarios, such as agricultural fields, it is strongly recommended to utilize an RTK GPS system. This would enable high-frequency (up to 20 Hz) and high-accuracy (centimeter-level) positioning, making the waypoint follower algorithm viable for practical deployment in environments where LiDAR data may be unreliable or unavailable.

3. **Addition of an IMU Sensor**
   Incorporating an Inertial Measurement Unit (IMU) into the system would significantly enhance localization accuracy through improved state estimation. This would also enable the implementation of advanced algorithms such as LIO-SAM or Fast-LIO, allowing for high-precision 3D mapping and robust sensor fusion using GPS, IMU, and encoder data.

4. **Testing in a Real Agricultural Environment**
   Field testing in a real agricultural setting would provide practical insights into the performance of AMCL-based localization versus GPS-based waypoint following. Such comparisons would help determine the most effective approach in conditions with variable terrain, sensor occlusion, or limited LiDAR visibility. Evaluating different controller and planner combinations in this context would further refine system performance.

5. **Incorporation of a Camera for Visual Perception**
   Integrating a vision sensor (e.g., RGB or RGB-D camera) could provide additional data for state estimation, semantic understanding, or visual SLAM. Leveraging camera input would enhance perception and open up opportunities for more advanced navigation tasks such as object recognition, terrain classification, or visual obstacle detection.

Bibliography

[1] Michal Mihálik, Branislav Malobický, Peter Peniak, and Peter Vestenický. «The New Method of Active SLAM for Mapping Using LiDAR». In: *Electronics* 11.7 (2022). Cited on p. 12, p. 1082. DOI: 10.3390/electronics11071082 (cit. on p. 5).

[2] Hao Qiu, Weifeng Chen, Aihong Ji, and Kai Hu. «Research on unmanned vehicle obstacle avoidance technology based on LIDAR and depth camera fusion». In: *Applied Mathematics and Nonlinear Sciences* 9 (Feb. 2023). DOI: 10.2478/amns.2023.2.00575 (cit. on p. 5).

[3] Michael Montemerlo, Sebastian Thrun, Daphne Roller, and Ben Wegbreit. «FastSLAM 2.0: an improved particle filtering algorithm for simultaneous localization and mapping that provably converges». In: *Proceedings of the 18th International Joint Conference on Artificial Intelligence*. IJCAI'03. Acapulco, Mexico: Morgan Kaufmann Publishers Inc., 2003, pp. 1151–1156 (cit. on p. 11).

[4] Austin I Eliazar and Ronald Parr. «DP-SLAM: fast, robust simultaneous localization and mapping without predetermined landmarks». In: *International Joint Conference on Artificial Intelligence*. 2003. URL: https://api.semanticscholar.org/CorpusID:267802622 (cit. on p. 11).

[5] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. «Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters». In: *IEEE Transactions on Robotics* 23.1 (2007), pp. 34–46. DOI: 10.1109/TRO.2006.889486 (cit. on p. 11).

[6] A Doucet, Nando de Freitas, Kevin P Murphy, and Stuart J Russell. «Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks». In: *Conference on Uncertainty in Artificial Intelligence*. 2000. URL: https://api.semanticscholar.org/CorpusID:2948186 (cit. on p. 11).

[7] Giorgio Grisetti, Rainer Kümmerle, Cyrill Stachniss, and Wolfram Burgard. «A Tutorial on Graph-Based SLAM». In: *IEEE Intelligent Transportation Systems Magazine* 2.4 (2010), pp. 31–43. DOI: 10.1109/MITS.2010.939925 (cit. on p. 12).

[8] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. «Real-time loop closure in 2D LIDAR SLAM». In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 1271–1278. DOI: 10.1109/ICRA.2016.7487258 (cit. on pp. 14, 15).

[9] Ji Zhang and Sanjiv Singh. «LOAM: Lidar Odometry and Mapping in Real-time». In: *Robotics: Science and Systems*. MIT Press Journals, 2014. DOI: 10.15607/RSS.2014.X.007 (cit. on p. 17).

[10] Tixiao Shan, Brendan Englot, Drew Meyers, Wei Wang, Carlo Ratti, and Daniela Rus. «LIO-SAM: Tightly-coupled Lidar Inertial Odometry via Smoothing and Mapping». In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020, pp. 5135–5142. DOI: 10.1109/IROS45743.2020.9341176 (cit. on p. 19).

[11] Wei Xu and Fu Zhang. *FAST-LIO: A Fast, Robust LiDAR-inertial Odometry Package by Tightly-Coupled Iterated Kalman Filter*. 2021. arXiv: 2010.08196 [cs.RO]. URL: https://arxiv.org/abs/2010.08196 (cit. on p. 21).

[12] Robert Touchton, Daniel Kent, Tom Galluzzoa, Carl III, David II, Nicholas Flann, Jeff Wit, and Phil Adsit. «Planning and modeling extensions to the Joint Architecture for Unmanned Systems (JAUS) for application to unmanned ground vehicles». In: Mar. 2005. DOI: 10.1117/12.603631 (cit. on p. 23).

[13] E W Dijkstra. «A note on two problems in connexion with graphs». In: *Numerische Mathematik* 1.1 (1959), pp. 269–271. ISSN: 0945-3245. DOI: 10.1007/BF01386390. URL: https://doi.org/10.1007/BF01386390 (cit. on p. 24).

[14] Peter E Hart, Nils J Nilsson, and Bertram Raphael. «A Formal Basis for the Heuristic Determination of Minimum Cost Paths». In: *IEEE Transactions on Systems Science and Cybernetics* 4 (2 1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136 (cit. on p. 25).

[15] Antti Autere et al. *Extensions and Applications of the A Algorithm*. Tech. rep. Cited on p. 7. Helsinki, Finland: Helsinki University of Technology, 2005 (cit. on p. 25).

[16] A Stentz. «Optimal and efficient path planning for partially-known environments». In: *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*. 1994, 3310–3317 vol.4. DOI: 10.1109/ROBOT.1994.351061 (cit. on p. 26).

[17] Da-un Jang and Joo-Sung Kim. «Development of Ship Route-Planning Algorithm Based on Rapidly-Exploring Random Tree (RRT*) Using Designated Space». In: *Journal of Marine Science and Engineering* 10 (Nov. 2022), p. 1800. DOI: 10.3390/jmse10121800 (cit. on p. 28).

[18] Complex Systems and AI. *Ant Colony*. Cited on p. 11. 2025. URL: https://complex-systems-ai.com/en/algorithms-desaims/antcolony/ (visited on 06/20/2025) (cit. on p. 29).

[19] ROS Wiki contributors. *base_local_planner Package — Local Path Planning*. Cited on p. 40. 2023. URL: http://wiki.ros.org/base_local_planner?distro=noetic (visited on 06/20/2025) (cit. on p. 32).

[20] Jean Bosco Mbede, Xinhan Huang, and Min Wang. «Fuzzy Motion Planning among Dynamic Obstacles Using Artificial Potential Fields for Robot Manipulators». In: *Robotics and Autonomous Systems* 32.1 (2000). Cited on p. 16, pp. 61–72. DOI: 10.1016/S0921-8890(00)00073-7 (cit. on p. 33).

[21] MathWorks. *Vector Field Histogram*. 2024. URL: https://www.mathworks.com/help/nav/ug/vector-field-histograms.html (visited on 06/20/2025) (cit. on p. 34).

[22] Thorsten Brandt and Thomas Sattel. «Path Planning for Automotive Collision Avoidance Based on Elastic Bands». In: *IFAC Proceedings Volumes* 38.1 (2005). Cited on p. 14, pp. 210–215. DOI: 10.3182/20050703-6-CZ-1902.01245 (cit. on p. 35).

[23] Open Robotics. *ROS-Wiki– ROS-1 Documentation*. https://wiki.ros.org/Documentation. Accessed-14-July-2025. 2025 (cit. on p. 37).

[24] Open Robotics. *ROS-2 Documentation*. https://docs.ros.org/en/rolling/index.html. Accessed-14-July-2025. 2025 (cit. on p. 37).

[25] AgileX-Robotics. *SCOUT-MINI: High-Speed Mini Mobile Robot*. Product page, accessed 14-July-2025. 2025. URL: https://global.agilex.ai/products/scout-mini (visited on 07/14/2025) (cit. on p. 56).

[26] AgileX-Robotics. *BUNKER-MINI: Compact Tracked Mobile Robot*. Product page, accessed-14-July-2025. 2025. URL: https://global.agilex.ai/products/bunker-mini (visited on 07/14/2025) (cit. on p. 58).

[27] RoboSense Technology-Co.,Ltd. *Helios Series LiDAR*. Product page, accessed-14-July-2025. 2025. URL: https://www.robosense.ai/en/IncrementalComponents/Helios (visited on 07/14/2025) (cit. on p. 60).

[28] RealSense-AI. *Stereo-Depth-Cameras*. Product overview page, accessed-14-July-2025. 2025. URL: https://realsenseai.com/stereo-depth-cameras/ (visited on 07/14/2025) (cit. on p. 62).

[29] Teltonika Networks. *RUTX50 Routes*. Documentation wiki page, accessed-14-July-2025. 2025. URL: https://wiki.teltonika-networks.com/view/RUTX50_Routes (visited on 07/14/2025) (cit. on p. 66).