College of Design and Engineering

ME5413 Autonomous Mobile Robotics Final Project

Group 13

GAO LINYI	A0303944E	E1373032@u.nus.edu
GAO XIANG	A0304316R	E1373404@u.nus.edu
HE KEHAN	A0303738A	E1372826@u.nus.edu
HUANG PEILONG	A0313748Y	E1499143@u.nus.edu
YANG HAOCHUAN	A0304617J	E1373705@u.nus.edu
WANG LINJIE	A0303587Y	E1372675@u.nus.edu

https://github.com/HcBlackYang/jackal_autonomy

Apr 6, 2025

Contents

1 Project Description

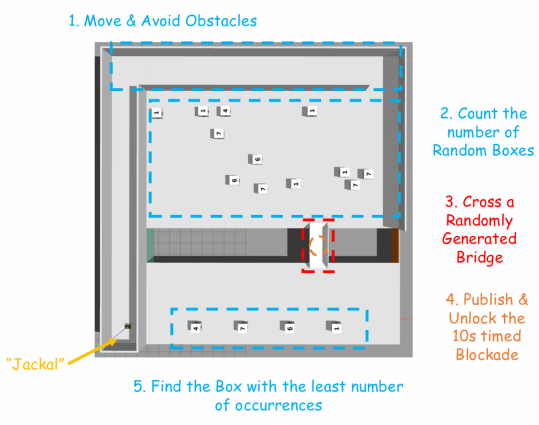

Fig. 1: Overview of the Task Objectives

As shown in Fig. 1. In this project, a simulated environment is created in Gazebo, featuring a procedurally generated world containing multiple random boxes, a bridge with a timed blockade, and four destination points. The objective is to develop a complete autonomous navigation system using the "Jackal" robot as the mobile platform to accomplish a series of perception-driven and navigation-intensive tasks.

The project consists of two main components: mapping and navigation. In the mapping stage, the robot is required to autonomously explore the unknown environment and construct an accurate 2D occupancy grid map using SLAM (Simultaneous Localization and Mapping). Mapping performance is assessed through both qualitative map evaluation and quantitative comparison between estimated and ground truth odometry data. In the navigation stage, the robot performs a sequence of tasks, including obstacle avoidance, box detection and counting, bridge crossing after unlocking a timed blockade, and locating the box with the lowest number of occurrences. The robot must complete these tasks autonomously and in a specific order, adapting to the dynamic structure of the environment.

2 Task1: Mapping

Mapping is a critical first step in autonomous robot navigation, as it provides a spatial representation of the environment that allows the robot to localize itself and plan safe, efficient paths. In this project, we implemented a SLAM solution to enable the Jackal robot to explore and build a 2D occupancy grid map of an unknown environment. The map is later used as the foundation for all subsequent navigation tasks, including object detection, bridge traversal, and goal selection. A reliable and accurate map is essential for ensuring that the robot can perceive obstacles, detect structural features, and operate robustly in dynamic or partially structured surroundings [1].

2.1 SLAM Algorithm: Gmapping and Cartographer

In selecting a SLAM algorithm for this project, we conducted a literature-based comparison between two commonly used 2D SLAM methods in ROS: GMapping and Cartographer. Both algorithms are widely adopted and well-supported in robotic applications, particularly for indoor mapping tasks using LiDAR and odometry data.

TABLE I: GMapping vs. Cartographer [2], [3]

Criteria	GMapping	Cartographer
Mapping Accuracy	Good in structured environments	High accuracy with reduced drift
Loop Closure	Basic support; often less reliable	Robust global loop closure with pose graph
Real-time Performance	Lightweight, performs well in real-time	Real-time capable but more CPU-intensive
Computational Cost	Lower resource usage	Higher, due to continuous optimization
Map Consistency	May degrade over long runs without loop closure	Maintains consistency through graph optimization
ROS Integration	Stable and widely used	Actively maintained with good documentation

The table I summarizes the theoretical strengths and limitations of each algorithm, as reported in existing studies and developer documentation.

Although both algorithms are suitable for 2D SLAM, Cartographer provides several advantages in terms of global consistency, robust loop closure, and scalability in more complex environments. GMapping remains a solid choice for simpler or resource-constrained systems, but Cartographer is generally favored in modern applications due to its graph-based approach and improved accuracy over extended trajectories.

Based on these considerations, we selected Cartographer as the SLAM solution for this project. While both algorithms offer comparable performance under our experimental conditions, Cartographer's theoretical advantages and broader capabilities make it a more future-proof and reliable choice for our navigation pipeline.

2.2 Map Representation and Processing

Cartographer is a state-of-the-art SLAM framework developed by Google, known for its high accuracy, robust loop closure detection, and real-time performance. Unlike traditional SLAM methods that rely solely on odometry and laser scan matching, Cartographer builds a pose graph composed of local submaps and continuously performs scan-to-submap matching and pose graph optimization to maintain global consistency. This results in globally accurate maps with minimal drift, even in large indoor environments [3].One of Cartographer's key advantages is its real-time loop closure, which enables the system to automatically correct accumulated trajectory errors by detecting and aligning overlapping submaps. Additionally, Cartographer supports 2D and 3D mapping, integrates seamlessly with ROS, and provides stable performance without heavy parameter tuning. These features make it particularly suitable for mobile robots operating in complex or partially structured environments.

The mapping pipeline is designed to enable the robot to autonomously explore and construct a 2D occupancy grid map of the environment using sensor fusion and real-time SLAM. In this project, we adopt Cartographer as the SLAM solution, configured to operate in 2D mode while taking input from a 3D LiDAR. This setup allows us to utilize richer point cloud data while generating a lightweight 2D map compatible with downstream navigation components.

The system processes data from three main sources: a 3D LiDAR publishing on the *"/mid/points"* topic, an IMU on

"/imu/data", and filtered odometry on *"/odometry/filtered"*, which fuses information from wheel encoders, IMU, and GPS. These data streams are fed into Cartographer's front-end, which performs online scan matching and pose estimation. Vertical filtering is applied to the LiDAR data, excluding points below 0.15 meters along the Z-axis, and the maximum scan range is limited to 30 meters. Online correlative scan matching is enabled to improve alignment in areas with limited geometric features.

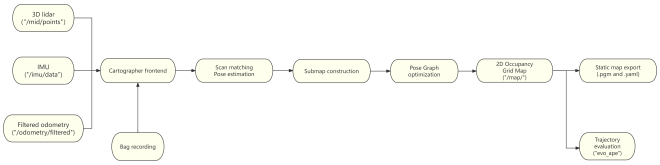

Fig. 2: Mapping pipeline using Cartographer SLAM

Pose estimation combines LiDAR scan matching with IMU and odometry data to refine the robot's localization. Cartographer incrementally builds submaps from incoming scans, which are later used to form a globally consistent map. The back-end performs pose graph optimization every five nodes. The generated occupancy grid is published in real time via the "`/map`" topic during SLAM execution. To extract a static map for later use, the "`map_saver`" utility from the "`map_server`" package is used. This tool exports the occupancy grid as a "`.pgm`" image file along with a "`.yaml`" metadata file containing information such as resolution and map origin. These files are later loaded in the navigation pipeline to support AMCL localization and global path planning.

To preserve the best SLAM result, the entire mapping session is recorded using the command "`rosbag record -a`", producing a "`.bag`" file that contains all sensor data, odometry, and Cartographer outputs. This file is then replayed offline using "`rosbag play --clock`" to extract a clean and complete map under controlled conditions. The same "`.bag`" file is also used in the SLAM evaluation phase, where the estimated trajectory is compared against ground truth using the "`evo_ape`" tool.

Ground truth information is not involved in the mapping pipeline itself, and is used exclusively for post-processing and performance evaluation [4] .

2.3 Evaluation

After completing the SLAM pipeline using Cartographer, we evaluated the performance of the system through both visual inspection and quantitative trajectory analysis. The objective of this evaluation is to verify that the generated 2D occupancy grid map is consistent with the actual simulated environment and that the estimated robot trajectory aligns closely with the ground truth.

To quantitatively evaluate the accuracy of the generated SLAM trajectory, we used `evo`, a widely adopted Python-based tool for benchmarking odometry and SLAM systems. The tool allows comparison between the estimated trajectory and ground truth using several metrics, including Absolute Pose Error (APE) and Relative Pose Error (RPE). In our case, "`evo_ape`" was applied to measure the global deviation of the robot's estimated path from the simulated ground truth (obtained from "`/gazebo/model_states`"), while "`evo_traj`" and "`evo_rpe`" provided visualization and local error metrics respectively. The trajectory alignment was performed using SE(3) Umeyama alignment, ensuring frame consistency. This approach provides an objective and reproducible method for trajectory evaluation in SLAM systems [5] .

After completing the SLAM pipeline using Cartographer, we evaluated the performance of the system through both visual inspection and quantitative trajectory analysis. The objective of this evaluation is to verify that the generated 2D occupancy grid map is consistent with the actual simulated environment and that the estimated robot trajectory aligns closely with the ground truth.

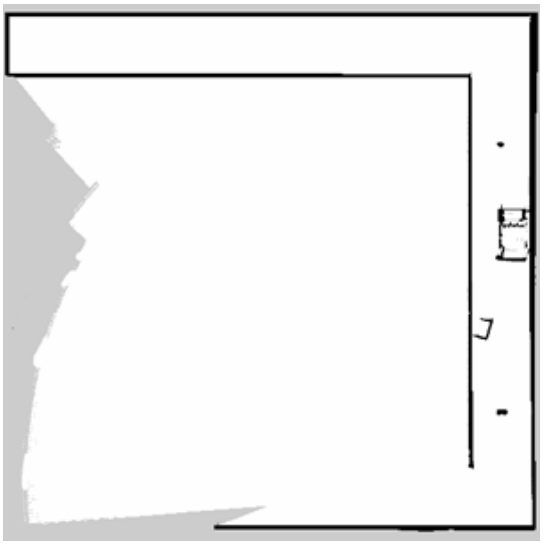

Fig. 3: Cartographer Map

As shown in Fig. 3, from a qualitative perspective, the 2D occupancy grid map produced by Cartographer accurately reflects the static structure of the simulated environment. The outer walls and initial fixed obstacles were clearly captured, and the overall layout appears complete and topologically consistent. As dynamic elements such as boxes and the bridge are only introduced after mapping is completed, they do not appear in the SLAM-generated map. The estimated trajectory also appears smooth and continuous, with no abrupt discontinuities or misalignments. These observations suggest that Cartographer effectively handled scan matching, maintaining geometric consistency throughout the exploration.

To quantitatively assess the system's accuracy, we used the "`evo_ape`" tool to compare the SLAM-estimated trajectory with the ground truth trajectory published under the "`/ground_truth`" topic in the simulation. The estimated poses were obtained from the transformation between the

"map" and "base_link" frames. To account for both translational and rotational discrepancies, the two trajectories were aligned using Umeyama alignment in SE(3) space.

TABLE II: APE Evaluation Metrics

Metric	Value (m)
Max APE	0.1565
Mean APE	0.0540
Median	0.0467
Min	0.0018
RMSE	0.0609
STD	0.0282

The Absolute Pose Error (APE) was computed over the entire sequence, and the results are summarized in Table II. The final Root Mean Square Error (RMSE) was 0.0609 m, indicating that on average, the estimated trajectory deviated less than 6.1 cm from the ground truth. The mean and median APE were 0.0540 m and 0.0467 m, respectively, while the maximum error observed was 0.1565 m. The low standard deviation of 0.0282 m confirms that the errors were tightly distributed and stable throughout the mapping session.

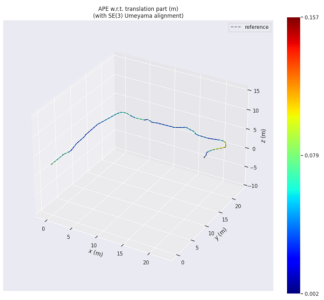

Fig. 4: APE over time using SE(3) Umeyama alignment.

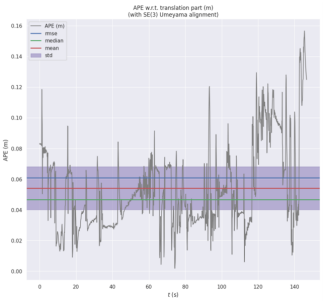

Fig. 5: 3D trajectory colored by APE values after alignment.

These results are further supported by the visual outputs shown in Fig. 4 and Fig. 5. Fig. 4 displays the estimated and ground truth trajectories overlaid in 3D space, with color coding indicating the magnitude of APE. The close alignment of the two trajectories throughout the run confirms that Cartographer maintains a high degree of global consistency. Fig. 5 presents the APE over time, illustrating that the error remains consistently low across the trajectory, with no abrupt spikes or divergence from the reference.

Given the visual quality of the map and the low APE values obtained from the trajectory analysis, the SLAM system can be considered reliable for this specific environment. The estimated trajectory follows the ground truth closely throughout the run, and the resulting map captures all necessary static structures needed for navigation. While dynamic elements like the bridge and boxes are not included at this stage, the generated map provides a sufficient foundation for the subsequent localization and path planning tasks.

2.4 Problems and Solutions

During the implementation of the SLAM system, two major issues were encountered that significantly affected the quality of the mapping results. These challenges were related to sensor configuration and the limitations of LiDAR data in specific environmental conditions. The following summarizes the problems and the corresponding solutions.

2.4.1 Issue of Mapping Without Odometry Input

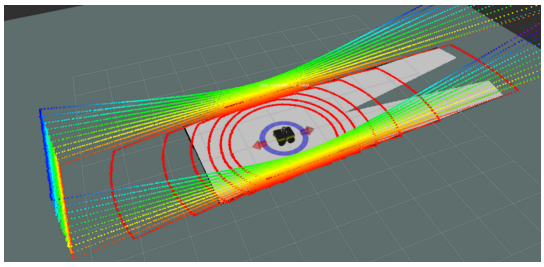

Fig. 6: Mapping without Odometry

The first issue we encountered during the mapping process occurred when attempting to use GMapping without incorporating odometry data. As shown in Fig. 6, the robot appeared to move forward in a straight corridor but the SLAM system consistently estimated it as stationary. This led to no meaningful change in the estimated pose, and thus the map was not updated. Upon further investigation, we found that the system was only using LiDAR scan data, without any reference to wheel odometry or IMU information.

Fig. 7: Gmapping Map without Odometry

In feature-sparse environments such as long corridors, LiDAR scan data from consecutive frames can appear nearly identical. Without additional motion input, GMapping's particle filter has no reliable way to detect small translations or rotations. This caused the robot's pose estimate to stagnate, and scan matching failed to introduce new map updates. As a result, the map

remained incomplete and distorted, with some areas incorrectly duplicated or missing altogether, as shown in Fig. 7.

To address this issue, fused odometry was integrated into the mapping pipeline by subscribing to the "`/odometry/filtered`" topic, which combines data from the IMU, wheel encoders, and GPS. This integration provided smoother and more reliable motion estimates for guiding the particle filter. As a result, the system was able to maintain accurate localization even in low-feature environments, enabling the mapping process to resume effectively. The estimated trajectory became continuous, and the occupancy grid map exhibited significant improvements in both completeness and consistency. Fig. 3 is the map generated using the odometry.

2.4.2 Issue of using a 2D LiDAR for mapping

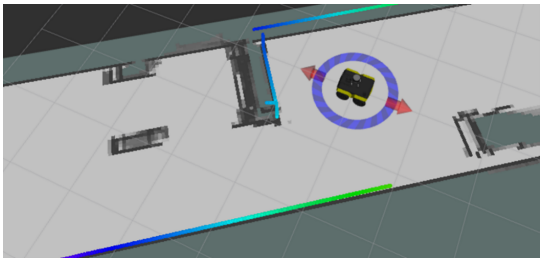

Fig. 8: Using a 2D LiDAR for Mapping

The second issue we encountered occurred when initially using a 2D LiDAR for mapping with Cartographer. During early tests, we observed that the LiDAR could only capture a few points near the robot's base—specifically the wheels of nearby objects—as shown in Fig. 8. As a result, larger obstacles such as walls or tall boxes were either missing from the map or rendered with incomplete outlines. This significantly affected the mapping quality and obstacle representation.The root cause was that the 2D LiDAR, which scans in a fixed horizontal plane, was not well-aligned with the most relevant parts of the environment. In our simulation setup, many structural features were either above or below the LiDAR scan plane, making them invisible to the sensor. Consequently, Cartographer had insufficient scan data to perform reliable scan matching and map updating, especially when the robot moved near tall or irregular obstacles.

To address this, we replaced the 2D LiDAR with a 3D LiDAR as the primary scan input while continuing to build a 2D occupancy map. This was accomplished by configuring Cartographer to consume the 3D LiDAR's point cloud and applying a Z-axis filter to extract points near ground level, effectively projecting the 3D data onto a 2D plane. Simultaneously, we enabled `use_trajectory_builder_2d` in Cartographer's configuration and incorporated IMU data along with fused odometry from "`/odometry/filtered`" to improve pose estimation.

We also fine-tuned several parameters to improve performance: the maximum laser range was set to 30 meters, Z-axis filtering excluded points below 0.15 meters, and each scan was processed immediately without accumulation. Loop closures were enforced by adjusting the minimum match score to 0.65, and the pose graph optimizer was configured to run every 5 nodes using a Huber loss function. These settings helped ensure robustness and scan matching quality.

After applying these changes, the system was able to correctly identify walls and static obstacles with well-defined contours. The final 2D occupancy grid was significantly clearer and more complete than before, allowing for successful downstream localization and planning.

These adjustments significantly improved the consistency and completeness of the generated map, and ensured the system could function reliably across both sparse and cluttered environments.

3 Task2: Navigation

Autonomous navigation is a core capability for mobile robots, enabling them to plan and execute motion within a mapped environment while avoiding obstacles and completing high-level tasks. In this project, after the environment was successfully mapped using SLAM, the robot was required to navigate through a sequence of objectives, including box detection, bridge crossing, and locating the target with the least frequent digit.

Our navigation system was built upon the ROS Navigation Stack, integrating several key components including localization, a global planner a local planner and a task controller to coordinate the execution sequence. Each subtask was modularized to ensure robustness, reusability, and clarity [6]. By combining these modules, the robot is expected to complete a series of mission-specific actions such as reaching designated areas, passing through constrained environments, and responding to dynamic changes in the scene.

A robust navigation pipeline is essential for achieving reliable autonomous behavior in complex indoor settings. In this project, we implemented and tested such a pipeline using the ROS navigation stack, evaluating its performance across various tasks and scenarios.

3.1 Localization: AMCL

Accurate localization is a critical component of autonomous navigation, particularly in environments where GPS is unavailable. In this project, we employed the Adaptive Monte Carlo Localization (AMCL) algorithm, which is widely used in ROS-based robotic systems for estimating a robot's 2D pose (x, y, θ) on a known static map. AMCL is based on the principles of particle filter localization, where a set of weighted samples (particles) represent the belief distribution over possible robot poses [7].

Each particle encodes a potential pose hypothesis, and the algorithm updates these particles in two stages: prediction and correction. During prediction, motion updates (typically from odometry) are used to propagate particles forward using a probabilistic motion model that accounts for uncertainty in control signals. In the correction phase, incoming laser scan data are compared to the static map to adjust particle weights based on their likelihood, i.e., how well the predicted pose explains

the observed scan. The resampling process emphasizes high-likelihood particles and discards low-probability ones, thereby converging toward the true pose. AMCL adapts the number of particles dynamically based on localization confidence—when uncertainty is high (e.g., after global relocation), it increases the particle set to cover more hypotheses. When confidence improves, it reduces the particle count to save computational resources. This makes it particularly effective for real-time localization in large maps or dynamic indoor environments [7], [8] .

In our system, AMCL utilized fused odometry from "`/odometry/filtered`" and LiDAR measurements to maintain a stable pose estimate throughout navigation. The static map used for localization was generated from the Cartographer SLAM pipeline and served as the global reference. The combination of motion and sensor models allowed AMCL to accurately track the robot's pose, even during turns, obstacle avoidance, and complex task sequences.

The static map used by AMCL was obtained from the Cartographer SLAM pipeline and exported in "`.pgm`" and "`.yaml`" format. It was loaded at runtime via the "`map_server`" node and served as the global reference frame for localization and planning. During operation, AMCL continuously aligned incoming LiDAR scans with the map to refine the robot's estimated position.The motion input for AMCL was provided through the "`/odometry/filtered`" topic, which fuses data from wheel encoders, IMU, and GPS using an Extended Kalman Filter. This fused odometry improved the robustness of localization, particularly in feature-sparse areas or during short-term scan mismatch.

To further enhance localization performance, especially during the navigation phase, we replaced the default LiDAR with a UTM-30 2D laser scanner. The UTM-30 operates at 50 Hz and provides a longer maximum range of up to 30 meters, which is particularly useful for detecting distant objects, such as randomly generated boxes in later task stages. The increased range also improved the accuracy of scan matching in AMCL, enabling more reliable localization even in large open spaces or when approaching distant goals.

3.2 Global Planning

Global planning is responsible for generating a collision-free path from the robot's current position to a specified goal location on the static map. This high-level path is planned over a global costmap that encodes environmental constraints, inflated obstacles, and travel costs. The global planner outputs a reference trajectory, which is subsequently followed and refined by the local planner during execution.

In this project, we evaluated and compared two classical algorithms for global path planning: "`Dijkstra`" and "A^*". Both were implemented within the "`move_base`" framework by configuring the "`global_planner`" plugin. The planning process was based on the same static map and costmap configuration, including an "`obstacle layer`" that incorporated both 2D and 3D LiDAR inputs. The inflation radius was set to 0.3 m for both the global and local costmaps to ensure safe navigation around obstacles.

To compare the performance of the two algorithms, we designed a test scenario focusing on the initial segment of the navigation task. Specifically, the robot was required to move from the starting location to the end of the first zone ("Move and Avoid Obstacles"). Both planners were assigned the same start and goal positions and were used in combination with the same local planner (TEB) to isolate the impact of global planning. The comparison was based on path geometry, total planned distance, and time taken to complete the route.

The following subsections present the implementation details and results for each planner, followed by an evaluation and selection of the preferred global planning strategy.

3.2.1 Global Planning - Dijkstra

Dijkstra's algorithm is a classical single-source shortest path algorithm that operates on a weighted graph. It computes the minimum-cost path from a given start node to all other reachable nodes by incrementally selecting the node with the smallest cumulative cost. When applied to a discretized costmap, such as those used in robot navigation, it guarantees to find the globally optimal path based on travel cost while avoiding obstacles [9].

In our project, both Dijkstra and A^* algorithms were implemented using the "`global_planner`" plugin within the ROS "`move_base`" navigation framework. This planner operates over a 2D costmap generated by combining the static occupancy grid from SLAM with real-time sensor data. The costmap encodes traversability by assigning higher costs to regions near obstacles through inflation layers.

3.2.2 Global Planning - A^*

A^* is a heuristic-based extension of the Dijkstra algorithm. It incorporates an estimated cost-to-go to prioritize nodes that appear closer to the goal. This heuristic function, typically based on Euclidean or Manhattan distance, helps A^* reduce the number of explored nodes and accelerate pathfinding, especially in large-scale environments or long-range planning tasks [10]. Unlike Dijkstra, which expands uniformly outward from the start node, A^* focuses on directed search and balances optimality with efficiency. As noted by Hart et al., when the heuristic is admissible, A^* remains both complete and optimal. Although A^* is well-suited for high-dimensional search spaces, in our project we selected Dijkstra due to its deterministic behavior and better performance in the constrained simulation environment.

3.2.3 Evaluation

To compare the performance of "`Dijkstra`" and "A^*", both planners were tested on the same navigation task segment, with identical start and goal positions and using the same local planner ("`TEB`"). The comparison focused on two key metrics: the total path length generated by the global planner and the time taken to compute the path.

As shown in Fig. 9 and Fig. 10, both planners produced feasible paths to the target location while avoiding static obstacles.

As can be seen from Table III, although $A*$ is theoretically faster due to its heuristic-driven search, the actual planning times

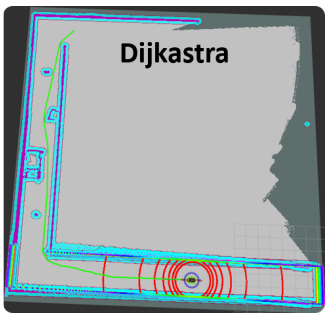

Fig. 9: Planned path using Dijkstra algorithm

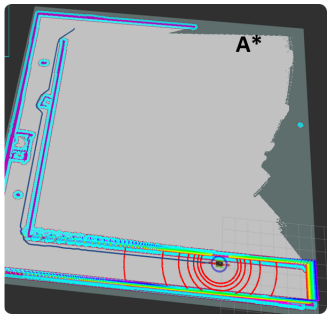

Fig. 10: Planned path using A^* algorithm

TABLE III: Comparison of Global Planner Performance

Global Planner	Distance (m)	Time (ms)
A*	44.30	8
Dijkstra	41.84	6

observed in our experiments were comparable, likely due to system-level timing variability. Specifically, *Dijkstra* completed the global planning in 6 ms, while *A** required 8 ms. This slight difference is negligible and had no noticeable impact on runtime performance.

Interestingly, *Dijkstra* produced a slightly shorter path of 41.84 m, compared to 44.3 m generated by *A**. This difference can be attributed to Dijkstra's conservative and uniform cost expansion, which tends to yield smoother paths around obstacles. In contrast, *A** may favor diagonal shortcuts guided by its heuristic, which can result in longer total paths due to the effects of grid discretization.

Based on these results, "`Dijkstra`" was selected as the preferred global planner for the remainder of the project. Its slightly shorter paths and stable performance made it more suitable for our task, especially considering that the time difference was negligible and did not affect responsiveness.

3.3 Local Planning

Local planning plays a critical role in enabling the robot to follow a global path in real time while avoiding dynamic and static obstacles. Unlike global planning, which operates on a static map, local planning must account for the robot's kinematic constraints, real-time sensor inputs, and short-term environmental changes. It continuously generates velocity commands to guide the robot safely and smoothly to the goal.

In this project, we compared the performance of three local planning algorithms: DWA, TEB, and TrajectoryPlanner. These were implemented in ROS through the "`DWAPlannerROS`", "`TEBLocalPlannerROS`", and "`TrajectoryPlannerROS`" plugins, respectively. Each planner uses a different approach to trajectory generation and optimization. Our goal was to evaluate their practical performance and select the most suitable one for use in the final navigation pipeline.

All three planners were tested under the same global planner (Dijkstra) and identical costmap settings, including the "`obstacle layer`" and inflation radius. The only differences lay in the local planner configuration. To ensure that each planner could operate effectively, we individually tuned their maximum velocity, acceleration limits, and obstacle inflation parameters. Without this tuning, some planners would get stuck or fail to complete the route. The comparison focused on navigation smoothness, obstacle avoidance ability, responsiveness, and goal-reaching success.

The following subsections present each planner's characteristics and behavior, followed by a final evaluation and selection.

3.3.1 Local Planning - TrajectoryPlanner

TrajectoryPlannerROS is one of the earliest local planners developed for the ROS navigation stack, and is based on a velocity space sampling approach. At each control cycle, it samples a set of linear and angular velocity pairs, simulates their resulting trajectories over a fixed time horizon, and evaluates each candidate using a cost function. The cost function typically includes three main components: proximity to the global path, obstacle distance, and heading alignment with the goal. The trajectory with the lowest total cost is selected for execution [11].

Although `TrajectoryPlannerROS` is simple and fast, its performance is limited in highly dynamic or narrow environments, where finer trajectory optimization is needed. In our project, it was implemented through the `base_local_planner` package under the plugin name "`TrajectoryPlannerROS`", and evaluated using the same costmap and global path generated by Dijkstra. Its behavior served as a baseline for comparison against more advanced local planners such as TEB.

3.3.2 Local Planning - DWA

The Dynamic Window Approach (DWA) is a local collision avoidance algorithm that integrates the robot's dynamic constraints—such as velocity and acceleration limits—into the trajectory planning process. At each control cycle, DWA samples a set of velocity commands that are admissible within the robot's physical limits and simulates short-term trajectories accordingly. These candidates are then evaluated based on a multi-term cost function, which includes obstacle distance, alignment with the global path, and estimated progress toward the goal. Compared to earlier local planners, DWA offers better dynamic feasibility and real-time performance, especially for differential-drive robots in structured indoor environments [12].

In our implementation, the DWAPlannerROS plugin from the ROS navigation stack was used. It allows for flexible tuning of forward simulation time, goal tolerance, and cost function weights, making it suitable for responsive navigation under varying conditions.

3.3.3 Local Planning - TEB

The Timed Elastic Band (TEB) planner is a state-of-the-art local planner that formulates navigation as a nonlinear optimization problem. Unlike traditional local planners such as DWA or TrajectoryPlanner that sample discrete velocity commands, TEB continuously optimizes a time-parameterized trajectory by minimizing a cost function that incorporates

obstacle avoidance, trajectory smoothness, time efficiency, and dynamic feasibility [13]. The trajectory is represented as a sequence of poses with associated time intervals, forming an "elastic band" that can deform in both spatial and temporal dimensions to avoid collisions and satisfy robot kinematics.

In ROS, the TEB planner is implemented via the `teb_local_planner` plugin and integrated with `move_base`. It supports both holonomic and non-holonomic robots, allows user-defined velocity and acceleration limits, and includes features like via-points, obstacle inflation, dynamic re-planning, and penalty functions. As shown by Rösmann et al. [13], [14], the TEB planner is well-suited for narrow environments and dynamic obstacle scenarios, offering smoother and more adaptive paths compared to sampling-based planners.

3.3.4 Evaluation

To evaluate the performance of the three local planners—TrajectoryPlanner, DWA, and TEB—we conducted a set of controlled experiments on the same navigation segment. Each planner was tested under the same environmental conditions, costmap configuration, and global path (generated by Dijkstra). The evaluation metrics included navigation success, path length, total navigation time, average linear velocity, and maximum angular velocity.

TABLE IV: Local planner performance comparison

Planner	Suc.	Len (m)	T (s)	Lin (m/s)	Ang (rad/s)
Traj.ROS	Y	44.07	102.77	0.43	1.00
DWA	Y	41.84	54.64	0.77	0.78
TEB	Y	42.51	44.79	0.95	1.57

From Table IV, among the three planners, TEB demonstrated the best overall performance. Although its path length was slightly longer than DWA's, it completed the navigation task in the shortest time (44.79 s), with the highest average linear velocity (0.95 m/s) and angular flexibility (1.57 rad/s). This indicates that TEB was able to generate efficient, smooth, and responsive trajectories.

In contrast, TrajectoryPlannerROS resulted in the longest path and slowest movement, indicating a more conservative and less efficient approach. While DWA performed better than TrajectoryPlanner, it exhibited oscillations in tight spaces and required more frequent adjustments, leading to slower response in certain scenarios.

Based on this comparative evaluation, TEB was selected as the final local planner in our navigation system. Its optimization-based trajectory planning allowed the robot to navigate smoothly and efficiently even in complex environments with dynamic changes.

3.4 Navigation Pipeline and Task Execution

The navigation component was responsible for coordinating mapping, localization, perception, global and local planning to guide the robot through a sequence of tasks in a simulated mini-factory environment. The robot followed a predefined mission sequence based on the group number, and each stage required a combination of robust perception and reliable navigation. The overall navigation pipeline is illustrated in figures and summarized as follows:

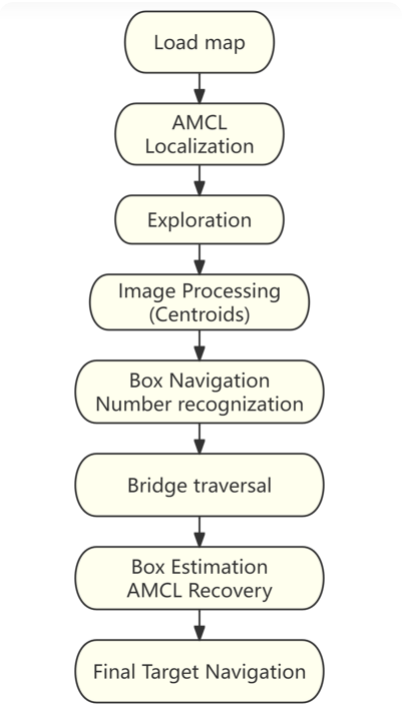

Fig. 11: Task flow chart

3.4.1 Move and Avoid Obstacles

At the beginning of the task, the robot had access to a partial 2D map generated from the previous SLAM phase. Although this map lacked information about random boxes and the bridge, it included all static walls and background obstacles. Using this map, the robot successfully navigated from the start location to the random box region, avoiding all obstacles using *Dijkstra* for global planning and *TEB* for local trajectory optimization. This stage verified the reliability of the planner in known, static environments.

3.4.2 Exploration of the Random Box Regions

Upon arrival at the random box region, the robot performed a serpentine sweeping motion to explore the area. During this process, the system collected a series of local costmaps reflecting real-time observations of the environment. These local maps were continuously fused to construct a more complete occupancy grid that captured all runtime-generated elements, including the randomly placed boxes and the bridge structure. This fused map served as the basis for downstream image processing and goal extraction.

3.4.3 Target Point Extraction via Image Processing

After exploration, the fused occupancy grid map was used for image-based analysis. We applied contour detection techniques to identify the bounding shapes of all detected boxes and the bridge structure. For each identified object, the centroid was

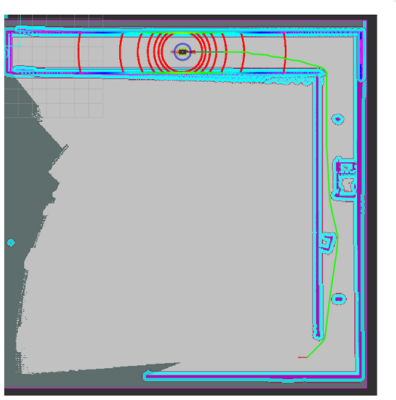

Fig. 12: Move and Avoid Obstacles

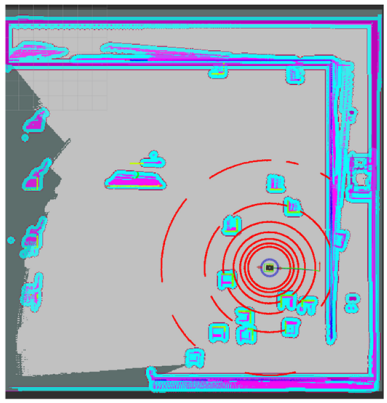

Fig. 13: Exploration of the Random Box Regions

computed to represent its approximate location. However, rather than using these centroids directly as navigation goals, we evaluated the surrounding area of each centroid to determine a reachable and safe goal point.

In the case of the bridge, this process also allowed us to estimate its approximate size and orientation, and to select an appropriate pre-bridge entry point for later traversal based on map geometry and available clearance.

3.4.4 Navigation to Boxes and Number Recognition

Using the extracted box centroids, the robot navigated to each box to record the labeled number. A simple nearest-goal selection algorithm ensured that the robot always traveled to the closest unvisited box. This stage continued until all boxes in the area had been visited and labeled.

3.4.5 Bridge Crossing and Localization Recovery

After completing the box visits, the robot navigated to the entry point of the bridge and waited for the temporary obstacle to open. Since the bridge was only accessible for 10 seconds, we used the previously estimated bridge length from the exploration map and commanded the robot to move forward by a fixed distance immediately after the obstacle was cleared.

Due to the lack of static features on the bridge surface, the robot's AMCL-based localization frequently became inaccurate

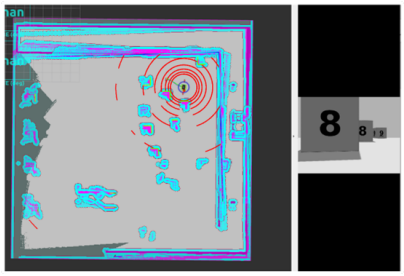

Fig. 14: Navigation to Boxes and Number Recognition

or unstable during crossing. To address this, we computed an approximate target pose on the opposite side based on the robot's last known location and the estimated travel distance. After reaching the expected position beyond the bridge, the robot remained stationary to allow AMCL to gradually recover its localization using new LiDAR observations and its built-in periodic update mechanism. This automatic relocalization enabled the robot to continue with the next stage of navigation without manual pose resetting.

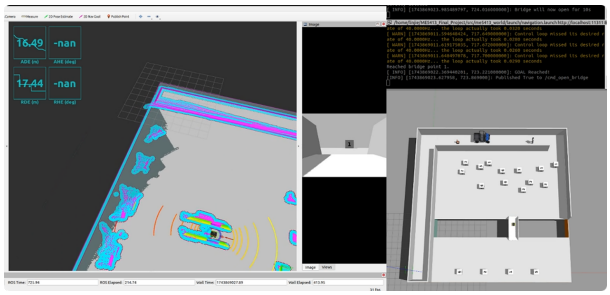

Fig. 15: Bridge Crossing and Localization Recovery

3.4.6 Final Navigation to the Least Frequent Number

Once AMCL localization was restored after bridge traversal, the robot proceeded to identify the box labeled with the least frequent number observed earlier. Although the positions of the four boxes on the opposite side were fixed and known in the map frame, their associated digit labels remained unknown.

Therefore, the robot sequentially navigated to each box's position, stopping at a predefined observation point to detect the number on the box. If the detected number matched the least frequent one recorded earlier, the robot terminated the search and approached the box directly, stopping in front of it to complete the task. Otherwise, it continued navigating to the remaining candidate boxes until the correct one was found.

This pipeline demonstrates the integration of perception, path planning, real-time control, and localization recovery mechanisms under realistic conditions. The Jackal robot successfully completed all Task 1 and Task 2 objectives. The outcomes of the mapping and navigation processes were clearly demonstrated in the video.

3.5 Problems and Solutions

During the development of the navigation pipeline, two major challenges emerged in the process of identifying the positions

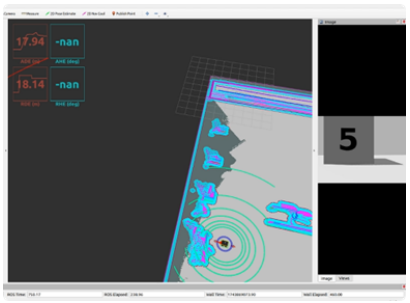

Fig. 16: Final Navigation to the Least Frequent Number

of randomly generated boxes. These issues directly affected the accuracy of goal setting and the reliability of subsequent navigation behavior.

3.5.1 Unstable Box Center Estimation via Normal Vector Fitting

The first challenge arose during our early attempt to estimate box center positions by analyzing the structure of each box as it appeared on the occupancy grid map. After the robot approached a box, a portion of the box surface became visible in the map due to laser scan accumulation. Based on this, we extracted multiple edge points corresponding to the observed face of the box. The goal was to compute a surface normal vector using least-squares fitting, and then project along this normal to estimate the actual center of the box.

However, in practice, this method was found to be highly sensitive to noise and map quality. Slight variations in edge sampling, limited resolution, and partial observations often led to significant deviations in the computed normal vector. Even small geometric distortions caused by incomplete scanning or pose error could mislead the fitting process, resulting in inaccurate goal estimation. Although we tried to improve robustness by increasing the number of measurement points and filtering outliers, the method remained both fragile and computationally inefficient, and was therefore eventually abandoned.

3.5.2 Centroid Splitting Caused by Partial Scan Coverage

The second challenge was encountered after switching to a centroid-based method for estimating box positions. In this approach, we used the updated local occupancy grid to extract connected components via image processing, then calculated the centroid of each detected object as a navigation target. This method was simpler and generally more robust. However, we observed that when the exploration was insufficient or the scan coverage was incomplete, a single box could appear as two disjoint regions on the map. This led to the same box being mistakenly recognized as two separate objects, resulting in duplicated centroids and potential navigation errors.

To address this issue, we analyzed the physical layout and spacing of boxes in the environment. It was observed that the minimum distance between two distinct boxes was never smaller than 1.131 meters. Based on this prior knowledge, we introduced a post-processing step that examined all centroid pairs and merged those whose separation was below a specified threshold. This effectively eliminated redundant detections and improved the reliability of box position estimation, especially in cases with limited scan coverage.

These two challenges highlighted the importance of robust geometric reasoning and map quality in object-level navigation. Through iterative testing and refinement, we were able to converge on a method that balanced accuracy and computational efficiency, forming a reliable foundation for downstream navigation tasks.

4 Conclusion

This project presented the design and implementation of a complete autonomous navigation pipeline for a simulated factory scenario. Based on a static map generated via Cartographer SLAM, we integrated AMCL localization, Dijkstra global planning, TEB local planning, and perception modules to enable the robot to complete a multi-stage task. These tasks included obstacle-aware navigation, exploration of a random object region, image-based extraction of navigation targets, bridge traversal, and goal-directed motion planning.

The navigation system was carefully evaluated at both the global and local levels, with Dijkstra and TEB selected as the final planning combination based on quantitative performance and trajectory quality. The perception module evolved from a surface-normal estimation approach based on box contours in the map to a more robust centroid-based method, which—after spatial filtering—offered greater consistency in box localization. Bridge traversal presented a significant challenge due to the lack of reliable features, and was addressed by combining prior pose information with estimated distance traveled, allowing AMCL to recover localization automatically.

Looking forward, several aspects of the system could benefit from further refinement, particularly in terms of execution efficiency and real-time integration. While the current task pipeline operates in a predefined and sequential manner—first exploring, then locating, then observing—this structure results in relatively long task durations and underutilization of available runtime information.

A more efficient approach would involve integrating object detection and pose estimation directly into the exploration phase. For example, using a depth camera, the system could apply object detection algorithms to identify digits and simultaneously extract 3D box or bridge locations, eliminating the need to revisit each object later. If using LiDAR and a monocular camera instead, the LiDAR data could be used for geometric localization, while the camera handles semantic recognition. In either case, perception and mapping would be conducted in parallel, allowing for real-time updates to both environmental structure and object information.

Another potential area of improvement lies in the exploration coverage itself. The current serpentine pattern provides basic region coverage, but it is not dense enough to guarantee complete box segmentation in all cases. Increasing the coverage density could improve object detection accuracy, but at the cost of significantly longer exploration time. An adaptive frontier exploration strategy could help balance this trade-off

by dynamically adjusting exploration density based on local map completeness and obstacle complexity.

Overall, enhancing the integration between perception and planning, and optimizing the exploration strategy, could substantially reduce task time and improve the robustness of the navigation pipeline.

References

[1] Durrant-Whyte, Hugh, and Tim Bailey. "Simultaneous localization and mapping: part I." *IEEE robotics & automation magazine* 13.2 (2006): 99-110.

[2] Grisetti, Giorgio, Cyrill Stachniss, and Wolfram Burgard. "Improved techniques for grid mapping with rao-blackwellized particle filters." *IEEE transactions on Robotics* 23.1 (2007): 34-46.

[3] Hess, Wolfgang, et al. "Real-time loop closure in 2D LIDAR SLAM." *2016 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2016.

[4] Cadena, Cesar, et al. "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age." *IEEE Transactions on robotics* 32.6 (2016): 1309-1332.

[5] M. Grupp, "evo: Python package for the evaluation of odometry and SLAM," [Online]. Available: https://github.com/MichaelGrupp/evo

[6] Rösmann, Christoph, Frank Hoffmann, and Torsten Bertram. "Integrated online trajectory planning and optimization in distinctive topologies." *Robotics and Autonomous Systems* 88 (2017): 142-153.

[7] Thrun, Sebastian. "Probabilistic robotics." *Communications of the ACM* 45.3 (2002): 52-57.

[8] Bailey, Tim, and Hugh Durrant-Whyte. "Simultaneous localization and mapping (SLAM): Part II." *IEEE robotics & automation magazine* 13.3 (2006): 108-117.

[9] Dijkstra, Edsger W. "A note on two problems in connexion with graphs." *Edsger Wybe Dijkstra: his life, work, and legacy*. 2022. 287-290.

[10] Hart, Peter E., Nils J. Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths." *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968): 100-107.

[11] Gerkey, Brian, Richard T. Vaughan, and Andrew Howard. "The player/stage project: Tools for multi-robot and distributed sensor systems." *Proceedings of the 11th international conference on advanced robotics*. Vol. 1. 2003.

[12] Fox, Dieter, Wolfram Burgard, and Sebastian Thrun. "The dynamic window approach to collision avoidance." *IEEE Robotics & Automation Magazine* 4.1 (1997): 23-33.

[13] Rösmann, Christoph, Frank Hoffmann, and Torsten Bertram. "Integrated online trajectory planning and optimization in distinctive topologies." *Robotics and Autonomous Systems* 88 (2017): 142-153.

[14] Rösmann, Christoph, et al. "Trajectory modification considering dynamic constraints of autonomous robots." *ROBOTIK 2012; 7th German Conference on Robotics*. VDE, 2012. *Robotics and Autonomous Systems* 88 (2017): 142-153.