ME5413 Autonomous Mobile Robots

Group 4 Final Project

Anurag Roy	Chun Pok Lam	Jinhan Xue	Yurong Liu	Praveen Krishnapur	Bowen Hu
A0304443N	A0305085J	A0305137M	A0304449A	A0304688U	A0234942J

I. Project Briefing

This project consists of two main tasks: mapping and navigation. In **Task 1**, the goal is to implement an algorithm to map the environment and evaluate the SLAM (Simultaneous Localization and Mapping) performance by comparing the estimated odometry with ground truth. This report includes a detailed description of the mapping pipeline, a qualitative and quantitative analysis of SLAM performance and a discussion of encountered challenges along with proposed solutions.

In **Task 2**, the objective is to navigate the robot through an area with randomly spawned boxes, extract the least occurring numbered box, cross the bridge with an obstacle in the middle of it and reach the least occurring numbered box. This report describes the navigation pipeline, and how the robot is designed to perform each task, while analyzing the performance using multiple metrics, and discussing challenges faced and solutions implemented.

Project Repository: Click to go to Project (https://github.com/anuragroy2001/ME5413_Final_Project).

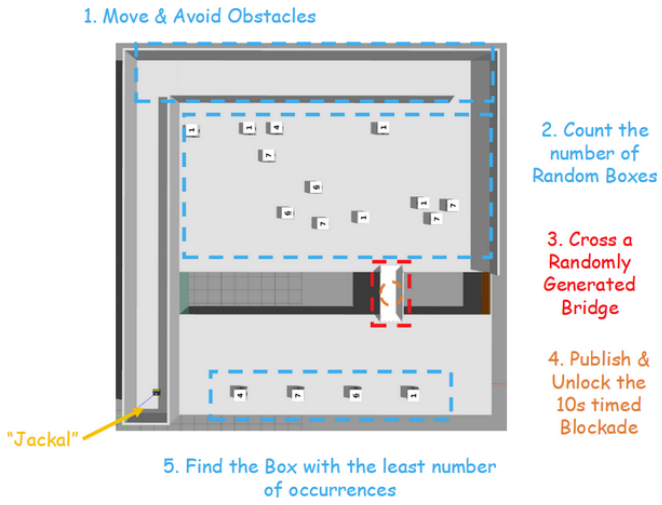


Fig. 1: Task and Map Overview

II. Task 1: Mapping

A. Cartographer

Cartographer [7] is a real-time SLAM framework developed by Google, designed to enable mobile robots to simultaneously construct a map of an unknown environment while estimating their own position within it. It supports both 2D and 3D SLAM, making it suitable for a wide range of robotic platforms, from indoor ground robots to aerial drones equipped with multi-dimensional sensors.

At its core, Cartographer operates using a graph-based SLAM approach, where the robot's trajectory and the map are represented as a pose graph. This graph comprises nodes that store robot poses and submaps—local collections of sensor data—while edges represent spatial constraints between them. The system continuously processes input data from various onboard sensors, primarily lidar (2D or 3D), IMU , and odometry. These sensors provide spatial and inertial information necessary for accurate localization and mapping.

The SLAM pipeline in Cartographer is divided into two main stages: local SLAM and global SLAM. In the local SLAM stage, the robot performs real-time scan matching, where each new lidar scan is aligned with a set of recent scans or an existing submap using algorithms such as the Ceres-based scan matcher. This allows the robot to incrementally estimate its pose and build submaps with high local accuracy.

In the global SLAM stage, Cartographer performs loop closure detection and pose graph optimization. Loop closures occur when the robot revisits previously explored areas, allowing the system to correct accumulated drift by establishing new constraints between the current and past poses. These constraints are added to the pose graph and optimized using non-linear least squares optimization. As a result, the global consistency of the map is significantly improved over time.

*1) **Cartographer 2D**:* We implemented Cartographer 2D, leveraging a front-facing 2D LiDAR (LaserScan) sensor as the primary source of range data. To enhance the accuracy and robustness of SLAM, we integrated the IMU and wheel odometry data into the system. This multi-sensor fusion allowed for more stable motion estimation, particularly during rapid maneuvers or in feature-sparse areas where lidar alone might be insufficient for reliable scan matching.

To further improve real-time scan alignment, we enabled online **correlative scan matching** within Cartographer's local SLAM pipeline. This algorithm performs an

exhaustive search around the predicted pose by evaluating multiple scan-to-submap alignments, ultimately selecting the transformation with the highest correlation score. We configured $linear\_search\_window$ to 0.1 meters and the $angular\_search\_window$ to ±35 degrees, striking a balance between computational load and alignment accuracy. These settings proved to be to be effective in environments with moderate motion uncertainty and cluttered geometry.

We also tuned several other key parameters to optimize mapping performance and computational efficiency. The $missing\_data\_ray\_length$ parameter was adjusted to ensure that unknown space behind occlusions or unobserved regions was correctly represented in the occupancy grid. The $submaps.num\_range\_data$ parameter was set to control the density and size of each submap, which affected both map detail and the responsiveness of local SLAM. To maintain a smooth and consistent global map, we set $optimize\_every\_n\_nodes$ to trigger pose graph optimization after a fixed number of trajectory nodes had been added. Additionally, scan matching weights such as $translational\_weight$ and $rotational\_weight$ were carefully calibrated to reflect the weight placed on lidar versus odometry inputs, while loop closure detection thresholds were fine-tuned to improve robustness against false positives.

As illustrated in Fig. 2, our configuration produced accurate and well-structured maps in complex, obstacle-rich environments in the initial corridor. Features such as walls, partitions, the cone, the human figure and the car were clearly resolved, enabling reliable path planning and localization. However, the system exhibited limitations in large open spaces, particularly on the far side of the river in our testing environment. In such areas, the 2D LiDAR was unable to receive sufficient reflections from distant surfaces, resulting in incomplete map coverage and gaps in the occupancy grid. This is a known constraint of 2D SLAM systems, which lack vertical sensing capabilities and thus struggle in environments where key features are elevated or sparsely distributed in height.

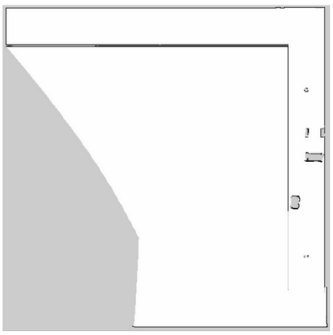


Fig. 2: Cartographer 2D Map

2) ***Cartographer 3D****:* To improve mapping coverage in open areas and achieve higher spatial resolution, we implemented Cartographer 3D, utilizing the 3D LiDAR sensor in conjunction with IMU and wheel odometry data. This configuration enabled the robot to perform robust, real-time SLAM in three-dimensional space, allowing for the accurate reconstruction of complex structural features that are typically challenging for 2D SLAM systems.

The 3D SLAM pipeline was enhanced by enabling IMU-based pose extrapolation, which uses inertial measurements to predict the robot's motion between successive LiDAR scans. This feature is particularly valuable during rapid rotations or when lidar updates are sparse. Additionally, odometry fusion was activated to incorporate wheel encoder data, further stabilizing pose estimation by providing an independent motion prior. To improve local scan alignment, online correlative scan matching was also enabled, allowing the system to evaluate a broad set of candidate transformations and select the one with the best match based on scan correlation scores.

To achieve a high update frequency and fine-grained detail, we configured the system to use $num\_accumulated\_range\_data = 1$, ensuring that each individual LiDAR scan was processed as it arrived. A fine voxel filter with a resolution of 0.05 meters was applied to the incoming point cloud, preserving fine structural details while maintaining real-time performance. In addition, adaptive voxel filters were deployed at various stages of the pipeline, dynamically adjusting resolution based on point density and map scale. This approach allowed the system to balance computation time with map fidelity, effectively managing resources while preserving important geometric features.

Key components of Cartographer 3D's optimization backend were also carefully tuned. The **Ceres-based scan matcher** was configured with adjusted scan and translation weights to better reflect sensor noise characteristics and environment complexity. We incorporated **intensity-based cost functions** to improve matching accuracy in environments with reflective surfaces. Moreover, **loop closure detection** thresholds were made stricter to prevent incorrect associations in visually ambiguous areas. These refinements collectively enhanced the robustness of the SLAM process and improved the global consistency of the generated map.

To further improve the usability of the final map output, a custom post-processing step was applied to the generated 3D occupancy grid. This involved filtering small noisy artifacts and smoothing map boundaries to produce a cleaner, more interpretable representation.

As illustrated in Fig. 3, the transition to Cartographer 3D significantly improved overall map quality. Compared to the 2D SLAM results, the 3D maps offered superior structural completeness and clarity. Features such as walls, objects in the corridors, and elevated structures were accurately reconstructed, resulting in a richer spatial understanding of the environment.

3) ***Evaluation****:* To quantitatively evaluate the mapping and localization accuracy of our Cartographer 3D configuration, we employed the evo_ape tool [6] to compute the Absolute Pose Error (APE) by comparing the estimated trajectory against a time-synchronized ground truth reference. As shown in Fig. 4 and summarized in Table I, the system achieved a maximum APE of 1.258 meters, with a mean APE of 0.461

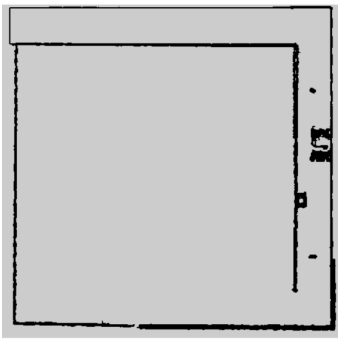


Fig. 3: Cartographer 3D Map

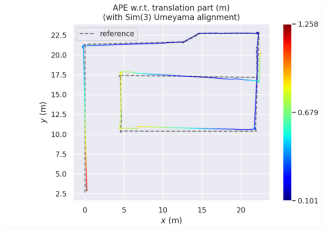


Fig. 4: Cartographer 3D Evaluation

Metric (m)	Result
Max APE	1.258
Mean APE	0.461
Median APE	0.366
Min APE	0.101
RMSE	0.536

TABLE I: Evaluation of 3D Cartographer

meters and a root mean square error (RMSE) of 0.536 meters. The median APE of 0.366 meters and minimum APE of 0.101 meters further indicate a generally stable and accurate pose estimation over the duration of the experiment.

The results demonstrate that Cartographer 3D meets the project's baseline requirements for trajectory accuracy, with the estimated path maintaining close alignment with the ground truth. Visual inspection of the trajectory overlay in Figure 3 confirms that the system performs well in capturing the overall motion of the robot, particularly in structured indoor areas with dense features and consistent scan overlap.

Despite its strengths, we observed that Cartographer 3D has certain limitations in environments with sparse or ambiguous geometric features, such as open riverbank areas or large flat surfaces. In such scenarios, the LiDAR point cloud lacks sufficient spatial constraints, leading to occasional drift or reduced loop closure reliability. Additionally, the system's global optimization relies heavily on discrete scan matching and pose graph adjustments, which may be less flexible in real-time dynamic environments or during aggressive maneuvers.

To address these challenges and improve performance in sparse or open areas, we proceeded to evaluate a more specialized LiDAR-inertial SLAM solution—Fast-LIO—which offers tightly coupled state estimation with continuous-time filtering, designed specifically for high-frequency, low-latency localization in complex 3D environments.

B. Fast-LIO

Fast-LIO [17], [18] is a computationally efficient and robust LiDAR-inertial odometry algorithm designed to address the limitations of traditional LiDAR-based SLAM systems, particularly in dynamic and resource-constrained scenarios. Unlike loosely coupled approaches, Fast-LIO employs a tightly coupled iterated Kalman filter (IKF) to fuse LiDAR feature points directly with IMU measurements. This design enables real-time compensation for motion distortion and significantly enhances robustness in high-speed motion and cluttered environments. Additionally, by optimizing the computation of the Kalman gain, Fast-LIO reduces computational complexity, making it suitable for deployment on platforms with limited onboard processing power.

In our implementation, Fast-LIO was utilized to perform 3D mapping, with the goal of reconstructing an accurate 2D projection of the environment for navigation tasks. The mapping pipeline involved four main stages: data collection and point cloud generation, point cloud processing and 2D projection, map initialization and obstacle setting, and final map generation with obstacle projection.

*1) **Data Collection and Point Cloud Generation**:* The robot was remotely operated to traverse the environment while continuously collecting 3D point cloud data using the 3D LiDAR sensor. As the robot navigated, Fast-LIO processed the LiDAR scans and IMU readings in real time, producing a spatially accurate point cloud that included walls, obstacles, and other environmental features. The real-time point cloud visualization, captured in RViz, is shown in Fig. 5. At this stage, the raw point cloud data provided a dense 3D representation of the surroundings, with the robot's position.

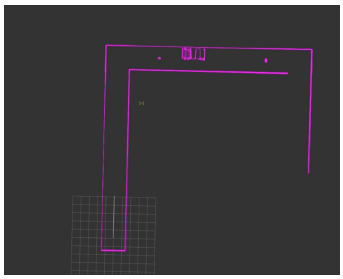


Fig. 5: Data Collection in RViz

*2) **Point Cloud Processing and 2D Projection**:* To convert the collected 3D point cloud into a 2D occupancy map, we used the pcd2pgm library, which transforms Point Cloud Data (PCD) files into grayscale image representations suitable for localization and mapping. The initial step generated a .pcd file from the Fast-LIO output, capturing the spatial structure in a standardized format. An example of the processed 3D point cloud is shown in Fig. 6, highlighting the geometry of the scanned environment.

During the map initialization phase, all grid cells in the occupancy map were initially set to a value of 100, indicating occupied or unknown regions. This conservative initialization ensures that unobserved areas—such as depressions, riverbanks, or occluded zones—are treated as impassable during navigation. Next, the processed 3D point cloud was projected onto the 2D plane, and areas corresponding to visible features such as floors and open regions were updated to a value of

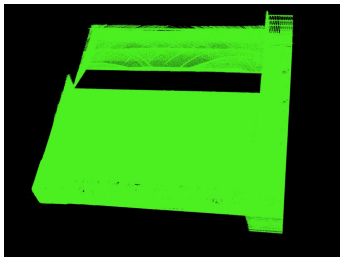


Fig. 6: Generated 3D Point Cloud Data

0, representing **free space**. Regions without corresponding point cloud data remained marked as occupied, preserving the integrity of unmapped spaces.

*3) **Obstacle Projection and Final Map Generation**:* In the final mapping stage, obstacle data from the 3D point cloud was explicitly projected onto the 2D map. Each 3D point was transformed into a corresponding 2D grid cell and marked as 100, denoting an obstacle. Conversely, open areas detected through point cloud gaps retained a value of 0. A filtering step was applied post-projection to smooth noisy regions and refine map boundaries. The result is a clean and reliable 2D occupancy grid, suitable for navigation and path planning. The final map is visualized in Fig. 7, showing clearly defined obstacles and free-space regions.

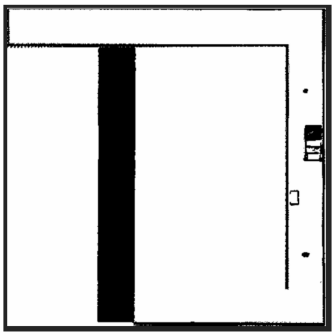


Fig. 7: Generated 2D Map

*4) **Evaluation**:* As shown in Fig. 8, the estimated trajectory from Fast-LIO closely follows the reference path, with minimal deviations. The corresponding numerical results are summarized in Table II. Fast-LIO achieved a maximum APE of 0.813 meters, with a mean APE of 0.182 meters and a median APE of 0.148 meters. The minimum APE was recorded at 0.055 meters, while the RMSE of the trajectory was 0.214 meters. These metrics represent a significant improvement over our earlier Cartographer 3D evaluation, indicating more accurate and stable trajectory estimation, particularly in areas where Cartographer exhibited drift due to sparse features or insufficient loop closures.

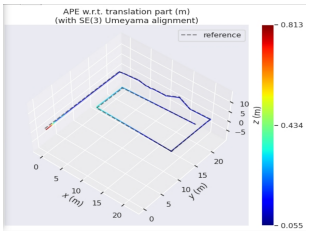


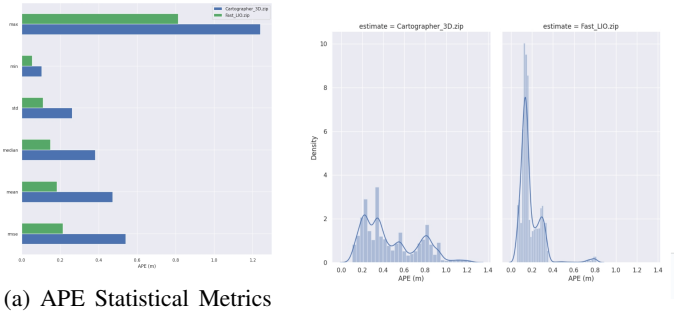
Fig. 8: Fast-LIO Trajectory Evaluation

Metric (m)	Result
Max APE	0.813
Mean APE	0.182
Median APE	0.148
Min APE	0.055
RMSE	0.214

TABLE II: Evaluation of Fast-LIO

C. Map fusion

Before proceeding to map fusion, we conducted a comparative analysis of localization accuracy between Cartographer and Fast-LIO, as shown in Fig. 9. The Absolute Pose Error (APE) distributions indicate that Fast-LIO achieves lower mean, median, and RMSE errors compared to Cartographer, demonstrating superior localization performance. This result motivated our design choice to assign higher weight to Fast-LIO outputs during map fusion.



(a) APE Statistical Metrics Comparison

(b) APE Histogram Comparison

Fig. 9: APE Analysis of Cartographer and Fast-LIO

Following the individual evaluations of Cartographer 2D, Cartographer 3D, and Fast-LIO, we implemented a map fusion strategy to integrate the strengths of each SLAM method into a single, unified occupancy grid. The goal was to create a more complete and reliable representation of the environment by leveraging the strengths of all the SLAM methods.

To achieve this, we developed a custom Python script to preprocess and combine the grayscale .pgm images generated by each mapping pipeline. These images were first standardized through histogram normalization and threshold unification to ensure consistent interpretation of free (0), occupied (100), and unknown (-1 or 205) regions. Binary masks were then extracted from each map based on defined intensity thresholds, corresponding to occupied and free-space areas.

The fusion process was conducted using a rule-based overlay strategy. Occupied cells from Fast-LIO were prioritized to preserve its high-resolution obstacle detection, especially in open and complex spaces. Structural outlines from Cartographer 3D were retained to reinforce map boundaries and ensure geometric consistency. Lastly, Cartographer 2D was used to fill in gaps and reinforce coverage in planar regions, particularly where 3D sensors failed to capture low-profile

features due to occlusion or sparsity. An example of the fused output is presented in Fig. 10, illustrating the enhanced spatial awareness achieved by combining the individual maps.

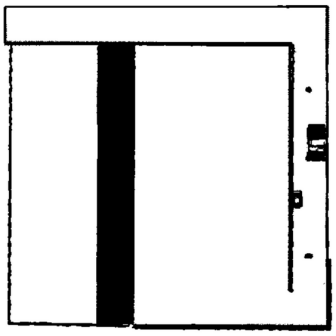


Fig. 10: Cartographer-FastLIO Fusion Map

III. Task 2: Navigation

A. Overview of Navigation Stack

The navigation system is built upon the ROS Navigation Stack [2], which provides a modular framework for integrating mapping, localization, global path planning, and local trajectory planning in mobile robots. Our system is tailored for a differential-drive robot, leveraging the map generated during the Mapping task to support autonomous goal-driven navigation.

Localization is achieved using the Adaptive Monte Carlo Localization (AMCL) algorithm, which estimates the robot's pose by matching incoming 2D LiDAR scans to the known occupancy grid. AMCL employs a particle filter that accounts for uncertainty in motion and sensor data $p(x_t \mid z_{1:t}, u_{1:t})$, where x_t is the pose, $z_{1:t}$ are sensor measurements, and $u_{1:t}$ are control inputs.

To improve localization accuracy and responsiveness, several AMCL parameters were tuned. The number of particles was increased to a range of 2000 to 5000, enhancing pose estimation precision in structured environments. The number of laser beams used for scan matching was raised to 2000, allowing more detailed alignment between LiDAR data and the map. Additionally, update thresholds were lowered (`update_min_d` = 0.1 m, `update_min_a` = 0.2 rad), enabling more frequent filter updates during robot motion. The motion model noise parameters (`odom_alpha1`–`odom_alpha4`) were refined to better suit the Jackal's differential drive kinematics.

Once a navigation goal is set, a global planner computes a collision-free path from the robot's current position to the target location using the static map. This path serves as a high-level guide for the robot. In parallel, a local planner continuously refines this path in real-time, reacting to dynamic obstacles and unexpected changes in the environment. The local planner ensures smooth, safe, and efficient motion by adjusting the robot's velocity and trajectory based on sensor feedback.

This combination of global and local planning enables the robot to navigate effectively in both structured and unstructured environments, while maintaining adaptability to real-world uncertainty and obstacle presence.

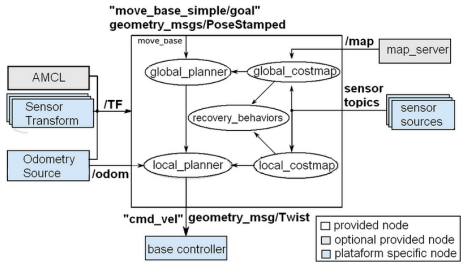


Fig. 11: Navigation Stack

B. Costmap

A costmap represents the configuration of the robot, where each cell holds a cost value indicating traversability, and indicates the location of free spaces (low cost) and obstacles (high cost), along with their surrounding areas using inflation (intermediate values). The costs can be generated by fusing sensor data, including LiDAR scans and point clouds, with static maps. This structure allows path planners to generate safe and efficient paths by avoiding high-cost regions while maintaining a buffer around obstacles. The ROS package `costmap_2d` is implemented here [10], as it allows a customized costmap based on environmental features.

`inflation_radius` is set to 0.35, and the `cost_scaling_factor` is set to 1 to allow the robot to safely navigate through the first narrow alley with multiple nearby obstacles.

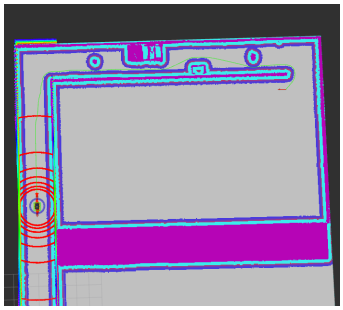


Fig. 12: Costmap and Global Planning path

C. Global Path Planning

1) ***Djikstra's Algorithm****:* Dijkstra's algorithm [3] is one of the global path planners used as it guarantees finding the optimal path with weighted graphs to the goal on static maps. The ROS package `navfn` [15] is used to implement Dijkstra's algorithm.

The map is discretized into a graph $G = (V, E)$, where V represents the grid cells and E the edges with weights corresponding to traversal costs derived from the costmap.

Dijkstra's algorithm computes the shortest path cost $f(v)$ from a source node v using:

$$f(v) = g(v)$$

where

$$g(v) = \min_{u \in \text{neighbors}(v)} [g(u) + w(u, v)]$$

, is the cumulative cost from the start node to node v.

*2) **A* Algorithm**:* The A* algorithm improves upon Dijkstra's method by incorporating a heuristic function to guide the search toward the goal more efficiently. It retains the optimality guarantees of Dijkstra's algorithm when an admissible heuristic is used. A* is implemented via the `global_planner` [14] ROS plugin.

In A*, the cost function is defined as:

$$f(v) = g(v) + h(v) \tag{1}$$

where $g(v)$ is the cost from the start node to node v, and $h(v)$ is the heuristic estimate of the cost from v to the goal. For our map, the heuristic $h(v)$ is typically the Euclidean distance to the goal. This heuristic significantly reduces the number of explored nodes compared to Dijkstra's algorithm, particularly in large open areas, making A* more computationally efficient while still producing optimal paths as shown in Figure 13.

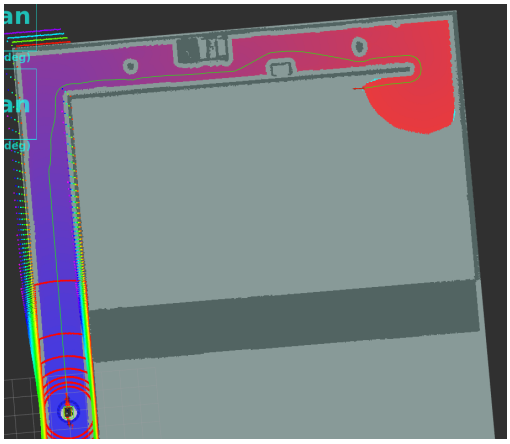


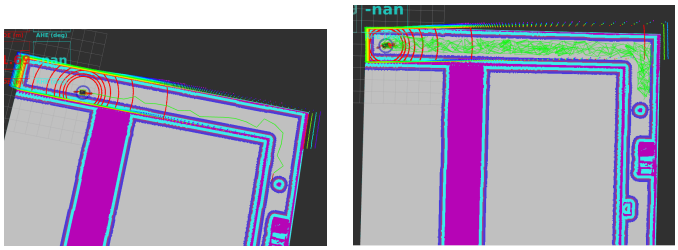
Fig. 13: A* Planner Path and Search Space

*3) **Rapidly-Exploring Random Tree (RRT)**:* RRT is a sampling-based planner designed for high-dimensional or continuous spaces, making it well-suited for unstructured or complex environments. Unlike graph-search algorithms, RRT incrementally builds a tree by randomly sampling points in the configuration space and connecting them to the nearest node in the tree if the path is collision-free.

We implemented a basic RRT planner by adapting an open-source Python implementation from Craig Daniel Miller's ROS-based RRT tutorial [8], and integrated it with our map data for global planning. . The RRT algorithm grows the tree until a node reaches the vicinity of the goal. The main steps are as follows:

1) Randomly sample a point x_{rand} in the configuration space.
2) Find the nearest node x_{nearest} in the existing tree.
3) Extend the tree toward x_{rand} to generate a new node x_{new}.
4) Add x_{new} to the tree if the connection from x_{nearest} is collision-free.

While RRT does not guarantee optimality, it is effective for generating feasible paths in large or partially-known environments. The generated path can be post-processed using smoothing or optimization techniques to improve quality and navigability.



(a) Generated RRT path (b) RRT tree visualization

Fig. 14: RRT global planning: path and tree structure in the environment

D. Local Path Planning

DWA (Dynamic Window Approach) and TEB (Timed Elastic Band) local planners are used and implemented here using the ROS packages `dwa_local_planner` [11] and `teb_local_planner` [12].

*1) **DWA planner**:* As a sampling-based algorithm, DWA [5]samples a set of possible velocities and simulates short trajectories while considering obstacles, dynamics, and the robot's limitations in action space and then selects the optimal action based on object function and constraints. At the sampling stage, DWA obtains possible velocities under constraints, and then it selects the optimal velocity (v, ω) by maximizing the following object function:

$$G(v, \omega) = \sigma\left(\alpha \cdot \text{heading}(v, \omega) + \beta \cdot \text{dist}(v, \omega) + \gamma \cdot \text{vel}(v, \omega)\right)$$

*2) **TEB planner**:* TEB [16] represents the planned path as a series of poses connected like an elastic band and continuously deforms it to minimize a cost function that considers obstacles, velocity, robot kinematics, and the shortest execution time. TEB is capable of handling tighter spaces and more complex motion constraints.

TEB optimizes a trajectory $\mathbf{X} = \{x_i, y_i, \theta_i, t_i\}$ to minimize a multi-objective cost:

$$\min_{\mathbf{X}} \sum_{i=1}^{N} \left(c_{\text{obst}}(x_i, y_i) + c_{\text{time}}(t_i) + c_{\text{kin}}(x_i, y_i, \theta_i)\right)$$

where c_{obst} penalizes proximity to obstacles, c_{time} regulates timing between poses, and c_{kin} ensures feasible motion given the robot's kinematics.

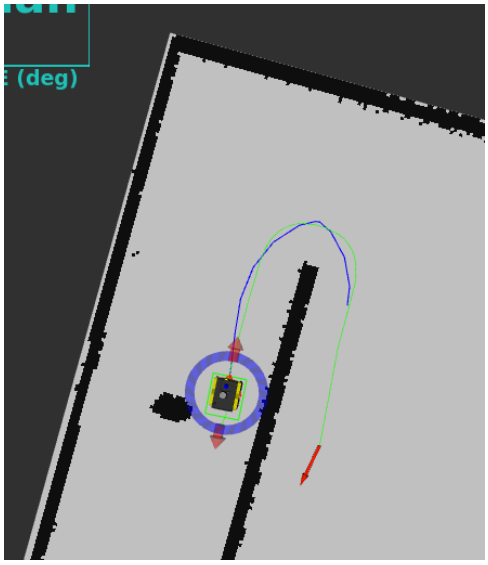


Fig. 15: TEB Local PLanner (Blue) working with A* Global Planner (Green)

E. Perception

Visual perception plays a critical role in enabling the robot to complete three main tasks in our project:

1) Detecting, classifying, and keeping track of the numbers printed on the surfaces of cubes.
2) Identifying the location of the bridge using the coordinates of cones.
3) Determining the coordinates of the final goal box that displays the number seen the least number of times.

To implement these functionalities, three perception nodes were created under the `me5413_world` package. These nodes utilize an onboard RGB-D camera to extract both color and depth information from the robot's environment.

1) Number Detection and Frequency Tracking: The objective is to explore a region scattered with numbered cubes and keep track of how often each number appears. The robot then navigates to the cube with the least frequently detected number.

To detect and classify the numbers, we used Optical Character Recognition (OCR) implemented via the open-source library **EasyOCR** [1]. This library was chosen for its out-of-the-box accuracy, ability to handle real-world images without additional training, and lightweight integration into our ROS-based pipeline.

EasyOCR performs recognition using a two-stage deep learning pipeline:

- **Text Detection:** Utilizes the CRAFT algorithm to identify regions containing text.
- **Text Recognition:** Applies a Convolutional Recurrent Neural Network (CRNN) for decoding character sequences from detected regions.

However, using EasyOCR directly introduced several challenges:

- Since numbers are printed on multiple faces of each cube, the same digit was often detected multiple times when multiple faces were visible.
- Numbers on distant cubes, including those beyond the bridge, were occasionally detected and included in frequency tracking.

To address the above issues, we designed a spatial filtering pipeline to discard noisy or duplicate detections.

Once the bounding boxes (BBs) of numbers are detected, the corresponding 3D coordinates in the camera frame are calculated using the **pinhole camera model**, as shown below:

$$X = \frac{(u - c_x) \cdot d}{f_x}, \quad Y = \frac{(v - c_y) \cdot d}{f_y}, \quad Z = d \tag{2}$$

Here, (u, v) are the pixel coordinates of the bounding box center, d is the depth value at that point, (c_x, c_y) are the principal point coordinates, and (f_x, f_y) are the focal lengths of the RGB-D camera.

These coordinates are then transformed into the map frame using the ROS `tf` package, which provides real-time transformations between coordinate frames.

We maintain a running list of detected coordinates and apply the following spatial filters:

- **Depth threshold:** Numbers detected at very small depths (too close to the camera) are likely to be repeated detections of the same cube face and are ignored.
- **Distance threshold:** Detections beyond a preset spatial limit (e.g., across the bridge) are discarded to prevent counting unreachable cubes.

This filtered set of detections ensures that the robot keeps an accurate count of unique cube numbers within a valid spatial context, significantly improving the robustness of the frequency tracking mechanism.

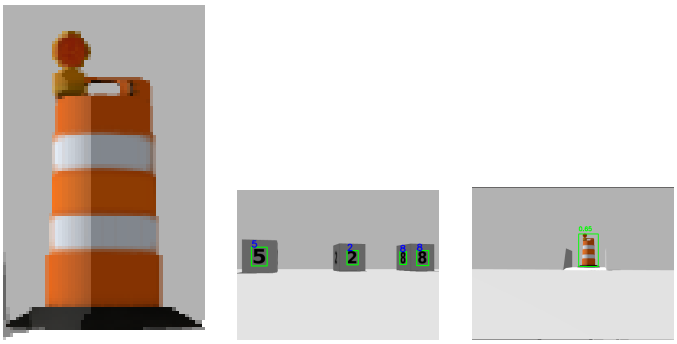
2) Bridge Localization via Cone Detection: In our environment, the bridge is randomly initialized in each episode. To proceed toward the final goal, the robot must first identify the location of the bridge. Fortunately, a bright orange traffic cone is placed at the entrance of the bridge, serving as a visual cue.

To detect the cone in the robot's camera feed, we used **template matching in the HSV color space**. The cone's saturated orange hue provides strong contrast against the predominantly gray background of the world, making it ideal for color-based detection. Both the input image and the template were converted to HSV space to increase robustness to lighting changes and to isolate color information more effectively.

We implemented the detection using OpenCV's `cv2.matchTemplate()` function. A cropped image of the cone (shown in Fig. 16a) was used as the template. Multiple scaled versions of the template were matched against the input image to account for perspective distortions and varying distances.

Once the bounding box of the cone was identified, the center of the bounding box was used to extract the corresponding depth value from the RGB-D camera. The 3D coordinates in the camera frame were then computed using the pinhole

camera model, and transformed into the map frame using ROS `tf`. This final position served as a reliable waypoint, allowing the robot to navigate toward and cross the bridge.



(a) Cone Template (b) OCR on Cubes (c) Cone Detection

Fig. 16: Perception tasks: (a) Template used for cone detection, (b) OCR results on cube faces, (c) Cone detected using template matchinge

3) Task 3: Final Goal Box Localization: The final task involves identifying and navigating to the cube that displays the number seen the least number of times during exploration.

To achieve this, the frequency information tracked in **Task 1** was continuously published as a `std_msgs/Int32MultiArray` message to the `/percep` topic. This array holds the count of each number detected so far. By subscribing to this topic, we extracted the number with the lowest frequency at runtime.

Once the bridge was crossed (as determined by reaching the cone waypoint), the robot began searching the area beyond the bridge for a cube displaying the target number. We once again leveraged the EasyOCR-based detection pipeline to identify visible digits in the camera feed. As in previous tasks, the 3D coordinates of each detection were computed using the pinhole camera model and transformed into the map frame.

When a cube displaying the correct number was found, its coordinates were published as a navigation goal, enabling the robot to approach the final target.

F. Finite State Machine

With the tasks and methods defined, a Finite State Machine (FSM) is implemented to provide a structured framework for executing operations. The FSM operates by transitioning through states based on a designed state-transition structure. Each state is initialized with specific inputs, output tokens, and termination conditions, enabling logical and stable execution while preventing task interference and conflicts.

In this project, we implement the FSM using the SMACH library for ROS to control the Jackal robot and perform autonomous navigation through a sequence of defined actions. These actions are divided into five tasks with each task corresponds to a state in the FSM and is executed sequentially.:

Task 1: Navigate to the room
Task 2: Explore the room
Task 3: Navigate to the bridge
Task 4: Cross the bridge
Task 5: Reach the boxes

The system begins with Task 1, where the robot navigates to a fixed goal position while avoiding obstacles. This state utilizes ROS's move_base action server for path planning and continuously monitors the navigation status. Upon successful navigation, the FSM transitions to Task 2, where the robot performs a lawnmower scanning pattern to explore the environment. During this phase, custom perception scripts are executed for box number detection and cone coordinate localization.

After the room is explored, the FSM proceeds to Task 3, where it captures cone coordinates published by the localization script via a subscribed ROS topic. Using the cone's y-coordinate (with x fixed at 9), the robot navigates to a position near the bridge. Upon reaching this location, it transitions to Task 4, where it moves straight across the bridge and stops in front of the cone. It then publishes a bridge-opening command to a designated ROS topic and continues across the bridge.

Upon successful bridge crossing, the FSM enters Task 5, where the robot visits predefined waypoints in front of the four boxes. Simultaneously, it activates the box coordinate recognition script and subscribes to the ROS topic where the final box goal coordinates are published. The FSM terminates once the robot reaches the final goal point.

The overall workflow of the FSM is illustrated in Figure 17.

G. Evaluation

*1) **Global Planner**:* To evaluate the performance of different global planners, we compared the behavior of A* and RRT in the various Tasks.

In all tasks, the A* algorithm consistently outperformed RRT in terms of path reliability and computation time. A* was able to generate **smooth, collision-free paths** even through narrow passages as shown in Figure 13, and was advantageous for our map since there were numerous small gaps to pass through in the initial corridors and also during exploration where some blocks would be very close to each other.

In contrast, the RRT planner exhibited difficulty in consistently finding feasible paths through narrow gaps, primarily due to its random sampling strategy and lack of fine resolution in sparse environments as shown in Figure 14. Moreover, RRT demonstrated significantly higher computation time compared to A*, especially for Task 1 where it pass through multiple obstacles to reach the end of the corridor, and the path it produced was **often jagged leading to longer path lengths**. While RRT eventually succeeded in generating a path for the other tasks, the resulting trajectories were often suboptimal.

Overall, A* was determined to be the more effective global planner for our application, providing reliable path planning with faster performance and better suitability for our map.

*2) **Local Planners**:* For local trajectory generation and obstacle avoidance, we evaluated the performance of the Dynamic Window Approach (DWA) and the Timed Elastic Band (TEB) planner. The evaluation considers task completion time and overall task completion efficiency in the environment.

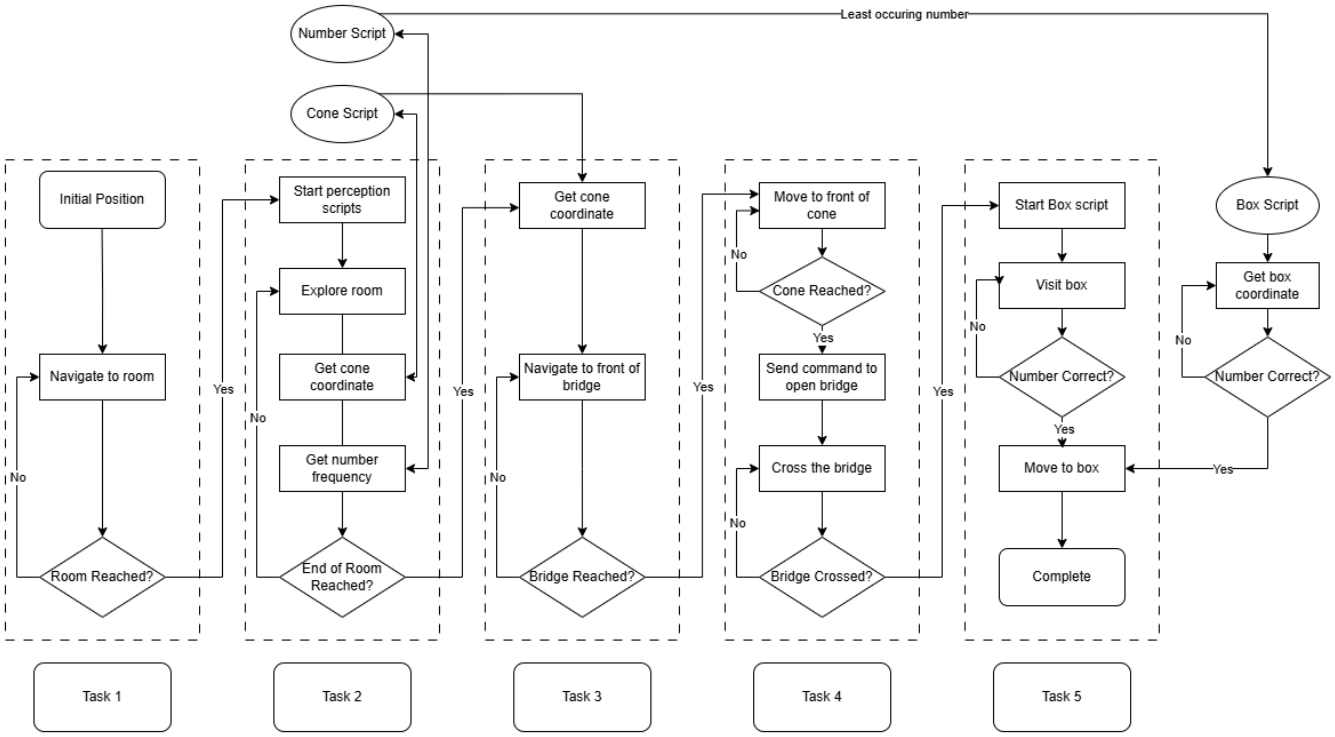


Fig. 17: Finite State Machine Workflow

In practice, the TEB planner consistently outperformed DWA, particularly in scenarios involving **tight spaces** such as **corridors** or **narrow gaps between walls and obstacles**. TEB's ability to optimize the full trajectory over time, while accounting for dynamic and kinematic constraints, enabled it to produce **smoother and more flexible trajectories**. This allowed the robot to closely follow walls and navigate through cluttered areas with minimal clearance, where DWA frequently struggled.

DWA, which samples velocity commands in the robot's local space, often became **stuck or oscillatory** near tight passages. This resulted in longer planning times and even occasional failures to proceed through narrow paths, especially around closely spaced obstacles. In contrast, TEB was able to maintain progress and adapt its trajectory to fit within the available free space. This is shown in Table III and Table IV, where in Task 1 and 2, TEB planner consistently outperformed the DWA planner.

Furthermore, across all navigation tasks, **TEB achieved consistently faster path completion times**. Its predictive and optimization-based approach allowed it to minimize detours and recover quickly from obstacle encounters, whereas DWA required more time to resolve suboptimal local decisions, getting stuck in the narrow gaps in Task 1 and also taking a lot of time to navigate around the blocks in Task 2, ultimately getting stuck.

Overall, TEB proved to be the more robust and efficient local planner for our project, particularly in the tasks requiring **precise maneuvering** and **tight navigation**.

TABLE III: Path Completion Time for DWA and TEB across Tasks at V=1.0

Task No.	DWA (s)	TEB (s)
Task 1	201.0s	118.2s
Task 2	395.0s	340.7s
Task 3	27.1s	24.1s
Task 4	10.0s	10.0s
Task 5	93.1s	74.7s

TABLE IV: Navigation Success Rate for DWA and TEB across Tasks for 5 attempts

Task No.	DWA Success Rate	TEB Success Rate
Task 1	1/5	5/5
Task 2	2/5	4/5
Task 3	5/5	5/5
Task 4	5/5	5/5
Task 5	5/5	5/5

IV. Future Improvements

While the current system demonstrates effective performance in both SLAM and navigation tasks, several enhancements can be made to improve accuracy, efficiency, and autonomy. Future work will explore advanced algorithms and more robust frameworks across the mapping and planning pipeline.

Improved SLAM with A-LOAM

To enhance the quality and consistency of 3D mapping, we plan to integrate **A-LOAM (Advanced LiDAR Odometry and Mapping)**, a feature-based LiDAR SLAM system optimized for real-time performance. A-LOAM extracts edge and planar features from point clouds and uses them for robust pose estimation and map construction. Compared to Cartographer [7] and Fast-LIO [17], [18], A-LOAM can offer improved accuracy in outdoor or high-speed environments where scan matching becomes unreliable. Its tight LiDAR odometry loop and refined map optimization make it a promising candidate for future deployments.

Enhanced Navigation with Theta and MPC*

For global path planning, we propose evaluating the **Theta* algorithm** [9], an any-angle extension of A*, which allows path smoothing through line-of-sight checks between nodes. This would reduce unnecessary turns and produce shorter, more natural-looking paths compared to grid-constrained planners like A* and Dijkstra.

In local planning, we aim to replace reactive planners like DWA and TEB with a model-based control approach such as **Model Predictive Control (MPC)** [4]. MPC plans optimal control inputs over a finite horizon while explicitly accounting for dynamic constraints, resulting in smoother and more precise trajectories—particularly beneficial in complex or cluttered spaces.

Autonomous Mapping with `explore_lite`

To improve system autonomy, we plan to incorporate the `explore_lite` package [13] to enable frontier-based autonomous exploration. This module identifies unexplored frontiers in the occupancy map and dynamically generates navigation goals, allowing the robot to map unknown environments without manual intervention. When combined with a robust SLAM backend, this approach supports fully autonomous coverage of unknown areas and is well-suited for long-term deployments.

REFERENCES

[1] Jaided AI. EasyOCR: Ready-to-use OCR with 80+ languages. https://github.com/JaidedAI/EasyOCR, 2020.

[2] David Coleman, Ioan Sucan, Sachin Chitta, and Nikolaus Correll. Reducing the barrier to entry of complex robotic software: a moveit! case study. *arXiv preprint arXiv:1404.3785*, 2014.

[3] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[4] Paolo Falcone, Francesco Borrelli, H. Eric Tseng, Javad Asgari, and Davor Hrovat. Predictive active steering control for autonomous vehicle systems. *IEEE Transactions on Control Systems Technology*, 15(3):566–580, 2007.

[5] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, volume 1, pages 1–5. IEEE, 1997.

[6] Michael Grupp. evo: Python package for the evaluation of odometry and slam. https://github.com/MichaelGrupp/evo, 2017.

[7] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. Real-time loop closure in 2d lidar slam. In *2016 IEEE international conference on robotics and automation (ICRA)*, pages 1271–1278. IEEE, 2016.

[8] Craig Daniel Miller. Rrt global planner for ros. https://craigdanielmiller.com/2021/10/23/rrt-global-planner-for-ros/, 2021. Accessed: 2025-04-06.

[9] Andrew Nash, Kenny Daniel, Sven Koenig, and Ariel Felner. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, 39:533–579, 2010.

[10] Open Source Robotics Foundation. costmap_2d. http://wiki.ros.org/costmap_2d. Accessed: 2025-04-06.

[11] Open Source Robotics Foundation. dwa_local_planner. http://wiki.ros.org/dwa_local_planner. Accessed: 2025-04-06.

[12] Open Source Robotics Foundation. teb_local_planner. http://wiki.ros.org/teb_local_planner. Accessed: 2025-04-06.

[13] Brian Pomerleau. explore_lite: Ros frontier exploration. https://github.com/hrnr/m-explore, 2016. Accessed: 2025-04-06.

[14] ROS Navigation Stack Contributors. global_planner: A* and dijkstra's global planner plugin. http://wiki.ros.org/global_planner, 2023. Accessed: 2025-04-06.

[15] ROS Navigation Stack Contributors. navfn: Dijkstra-based global planner plugin for ros. http://wiki.ros.org/navfn, 2023. Accessed: 2025-04-06.

[16] Christoph Rösmann, Maren Hoffmann, and Torsten Bertram. Trajectory modification considering dynamic constraints of autonomous robots. In *Proceedings of the 7th German Conference on Robotics (ROBOTIK)*, pages 1–6. VDI Verlag, 2012.

[17] Wei Xu, Yixi Cai, Dongjiao He, Jiarong Lin, and Fu Zhang. Fast-lio2: Fast direct lidar-inertial odometry. *IEEE Transactions on Robotics*, 38(4):2053–2073, 2022.

[18] Wei Xu and Fu Zhang. Fast-lio: A fast, robust lidar-inertial odometry package by tightly-coupled iterated kalman filter. *IEEE Robotics and Automation Letters*, 6(2):3317–3324, 2021.