ME5413 Autonomous Mobile Robot

Final Project Report

Group 8

Master of Science in Robotics

COLLEGE OF DESIGN AND ENGINEERING

Apr, 2024

CONTENTS

I. Project Overview

In this project, a simulated mini-factory environment is set up in Gazebo, consisting of three target areas and one restricted zone. The tasks involve guiding the robot to specific poses within each area, adhering to a specialized sequence determined by the group number. Our objective is to develop a robust navigation system using the "Jackal" robot as the hardware platform to efficiently accomplish these tasks.

To achieve this goal, we extensively explore and compare SLAM (Simultaneous Localization and Mapping) and path planning algorithms to identify effective and efficient options. By integrating the promising algorithms into our system, we aim to enable seamless navigation of the robot within the simulated environment.

Given our group number, '**8**', representing luck, the mission target sequence is defined as: **Assembly Line 1, Random Box 1,** and **Delivery Vehicle 3**.

II. Task1: Mapping

In this task, several SLAM algorithms are adopted and evaluated with the EVO tool.

A. Cartographer

Cartographer is an open-source SLAM algorithm developed by Google. It utilizes grid-based mapping along with a Ceres-based scan matcher to reconstruct the environment across different sensor configurations [1]. Renowned for its high accuracy and efficiency, Cartographer excels in producing detailed maps.

The algorithm is designed to generate both 2D and 3D maps of an environment by integrating data from various sensors using graph optimization. It continuously optimizes and updates the robot's location estimation and map construction by minimizing the error between all observed data points. This approach is characterized by efficient loop closure handling, enabling the algorithm to effectively handle large-scale and complex environments.

1) Cartographer Implementation: After compiling Cartographer_ROS following our GitHub repository *"README.md"* file, it becomes capable of estimating robot pose under the map frame. By default, Cartographer maintains and publishes the relationship between the tracking frame *"base_link"* and the map frame *"map"* during mapping, aligning with the simulation coordinate definition of this project. Thus, there is no need for additional robot link configuration in this regard.

To fully utilize Cartographer's multi-sensor support, the default topics should be modified to correspond with those published in *me5413_world*:

- 2D LaserScan: "/scan" → "/front/scan"
- 3D PointCloud: "/points2" → "/mid/points"
- Odometry: "/odom" → "/odometry/filtered"
- IMU: "/imu" → "/imu/data"
- GPS: "/fix" → "/navsat/fix"

Here we use an Extended Kalman Filter (EKF) fused with robot wheel odometry and IMU data as the odometry input for Cartographer. The kinematic models of vehicles can be found at Appendix A. With these modifications incorporated into our project setup, both 2D and 3D Mapping by Cartographer can be one-shot launched after activating the simulation environment by running *"roslaunch final_slam mapping_carto_2d"* or *"roslaunch final_slam mapping_carto_3d"*.

2) Cartographer Tuning: A user-friendly parameter tuning mechanism is offered by Cartographer_ROS through a *"lua"* file, facilitating easy adjustments to trajectory builder, pose graph, and other parameters. To enhance SLAM performance, we conducted experiments exploring various parameter configurations, with statistical results summarized in Tab. VIII.

Experimentation revealed that odometry prediction accuracy is notably influenced by local SLAM parameter *"submaps.num_range_data"*, which determines how range data is aggregated to form submaps in the 2D SLAM process. Adjusting this parameter has a direct impact on map resolution and computational load. Lower values result in more frequent submap creation, while higher values reduce noise but may overlook detailed environmental features. The benefits of increasing this parameter diminish over time.

The parameter *"optimize_every_n_nodes"* for global SLAM also significantly affects odometry accuracy by determining the sparse pose adjustment optimization frequency. Lower values result in more frequent optimization, particularly beneficial when *num_range_data* is small, enhancing precision.

Additional parameters related to online correlative scan matching, such as search windows and cost weights, have a minor effect on SLAM performance. It is worth noting that for 3D Mapping, the value of *"optimize_every_n_nodes"* needs to be twice the value of *"submaps.num_range_data"*, otherwise the *".pgm"* format map files cannot be exported from the *".pbstream"* files correctly.

3) 2D Cartographer: Fig. 22 illustrates a typical high-quality map built by 2D Cartographer. Four different sensor configurations are tested for 2D Cartographer. Pure LiDAR-based mapping is not considered due to the existence of feature-lacking long corridors in the task scenarios, which can result in significant LiDAR-based localization failure. The selected configurations are as follows:

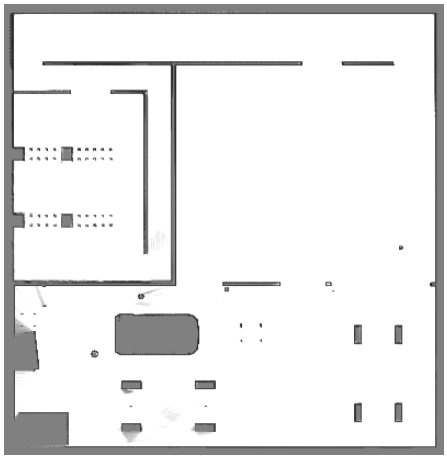


Fig. 1: Cartographer map

- IMU + 2D LiDAR (I2)

- Odometry + 2D LiDAR (O2)
- Odometry + IMU + 2D LiDAR (OI2)
- Odometry + IMU + GPS + 2D LiDAR (OIG2)

For different configurations, the corresponding built map and Absolute Pose Error (APE) are shown in Fig. 16, 18, 20, 22, and Fig. 17, 19, 21, 23, respectively. Table I provides 2D mapping performance comparison of different configurations in the same scenario. It can be observed that the OIG2 configuration, which utilizes the most sensors, yields the best result.

TABLE I: Evaluation of 2D Cartographer Configurations

Metric (m)	I2	O2	OI2	OIG2
Max APE	7.066	0.410	0.851	0.401
Mean APE	3.427	0.157	0.380	0.123
Median APE	1.670	0.105	0.360	0.109
Min APE	0.064	0.064	0.064	0.064
RMSE	4.261	0.181	0.433	0.134

4) 3D Cartographer: In addition to 2D Cartographer, we also evaluate the performance of 3D Cartographer with the following setup. Unlike its 2D laser scan, the point cloud obtained by the simulated Velodyne VLP-16 3D LiDAR is rendered using GPU, resulting in significantly faster and smoother visualization compared to CPU rendering. The render mode can be adjusted by modifying the */opt/ros/noetic/share/velodyne_description/urdfVLP-16.urdf.xacro* file.

For 3D Cartographer, we test three different sensor configurations:

1) Odometry + 3D LiDAR (O3)
2) Odometry + GPS + 3D LiDAR (OG3)
3) Odometry + IMU + GPS + 3D LiDAR (OIG3)

The reason for not including IMU+3D LiDar is because the 2D experiments indicated that the IMU data resulted in drifting and higher noise levels, making the measurements relatively unreliable. The maps generated are depicted in Fig. 24, 26, and 28, with corresponding APE values illustrated in Fig. 25, 27, and 29, respectively. Table II provides a performance comparison of different 3D mapping configurations in the task scenario. It is observed that the OIG3 configuration, which utilizes the most sensors, yields relatively better results. However, despite the additional information available, the 3D Cartographer does not achieve robust localization and complete map building. Consequently, it does not perform as well as the 2D mode. Therefore, it will not be utilized for localization in this work.

TABLE II: Evaluation of 3D Cartographer Configurations

Metric (m)	O3	OG3	OIG3
Max APE	3.465	2.363	2.843
Mean APE	0.180	0.167	0.161
Median APE	0.176	0.158	0.147
Min APE	0.099	0.093	0.089
RMSE	0.194	0.179	0.176

Additionally, while rendering the point cloud with GPU results in faster and smoother visualization compared to CPU rendering, running the 3D LiDAR algorithm on GPU causes the map to not be built properly. Rendering the point cloud in CPU mode allows the map to be generated, but the processing is significantly slower. This may be due to CPU rendering causing the system to lose some ROS messages for the robot, thus compromising the successful building of the map.

Furthermore, although GPU rendering enhances visualization speed and smoothness compared to CPU rendering, executing the 3D LiDAR algorithm on GPU results in improper map generation. Conversely, rendering the point cloud in CPU mode permits map generation, albeit with significantly slower processing. This discrepancy may be attributed to CPU rendering potentially causing the system to miss some ROS messages for the robot, thereby hindering successful map construction.

B. Fast-LIO2

FAST-LIO2 is an advanced LIO framework that delivers enhanced speed by forgoing feature extraction and incorporating an efficient mapping process. It introduces the ikd-Tree data structure for dynamic and efficient point cloud management, leading to significant improvements in kNN search essential for LiDAR odometry. Extensive tests confirm its superior performance in speed and accuracy, particularly in high-speed and low-feature scenarios [2].

While modifying Fast-LIO2, we encountered significant performance issues. Throughout our experiments, we aimed to enhance performance incrementally:

1) **Configuration Adjustment**:Initially, we adjusted the parameters considering the synchronization requirement for IMU and lidar frequency consistency. We increased the scan rate in the *"velodyne.yaml"* file to 50Hz (originally 10Hz). Additionally, adapting to the simulation scenario's Velodyne VLP-16 (16-line lidar), we set the *'scan_line'* parameter to 16 and adjusted the blind parameter to 1.
2) **Speed Reduction**:Upon observing continuous upward drift in the map construction period, we attributed it to the tight coupling of Lidar and IMU in Fast-LIO2. Inaccurate z-axis measurements of the IMU could lead to incorrect point cloud construction. We discovered that excessive vehicle speed caused inaccurate z-axis measurements of the IMU. Addressing this issue, we reduced the speed to 0.6, resulting in a close fit between the robot and the built map with the ground. Compared to the fast speed configuration, which only used points higher than 1.5m filtered by a path-through filter, the adjusted configuration only required the removal of point clouds below 0.3 meters, as they closely adhered to the ground surface. Fig. 32 and 31 indicates the variation of point clouds on the z-axis before and after reducing speed.
3) **Queue Reduction**:Despite introducing GPU and previously successful configurations, performance stuttering persisted, primarily due to the long queue length for publishing *"PointCloud2"* messages in ROS. To address this, we reduced the queue length for publishing *"PointCloud2"*2 messages and the execution frequency of the

callback function to one-tenth of the original, resulting in a significant speed improvement. Additionally, further reducing the rate to one-fifth of the original and focusing only on feature-rich point clouds below while ignoring upper point clouds scanning the walls, allowed us to reduce computational costs while ensuring performance. Finally, these adjustments led to a significant speed improvement.

Additionaly, we also Conducted mapping tests in various test-worlds to check whether reflections exist or not in this scenario by removing stop signs and windows. The result in indicating no reflection issues. Table III provides the comparison of results of Fast-LIO by different modification.

TABLE III: Localization performance of Fast-LIO

Metric (m)	Config. Adjustment	Speed Reduction	Queue Reduction
Max APE	1.850	1.322	0.936
Mean APE	0.837	0.619	0.399
Median APE	0.828	0.678	0.381
Min APE	0.043	0.034	0.041
RMSE	0.983	0.693	0.453

The high-quality point cloud captured by modified Fast-LIO and its coresponding map are shown in Fig. 2 and 3

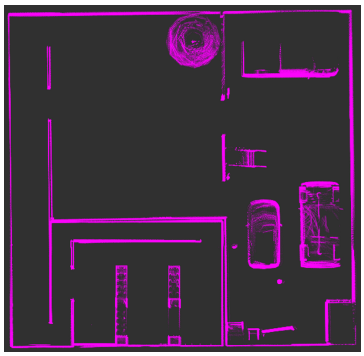


Fig. 2: Final point cloud by Fast-LIO2

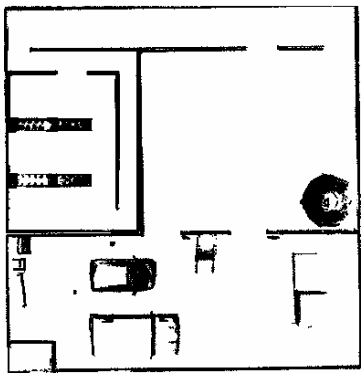


Fig. 3: Final map by Fast-LIO2

C. Localization

1) AMCL: Among localization algorithms, the Adaptive Monte Carlo Localization (AMCL) algorithm is applied most often in robot localization, a two-dimensional environment probabilistic localization system to improve the problems existing in the traditional MCL method [3]. AMCL adapts the number of particles used in the process according to the requirement for computational resources and the need for precision. It works effectively in dynamic environments and can recover from localization failures due to its ability to maintain multiple hypotheses about the robot's position. We implement the AMCL in the navigation package of ROS. The final map is shown in the Fig. 35.

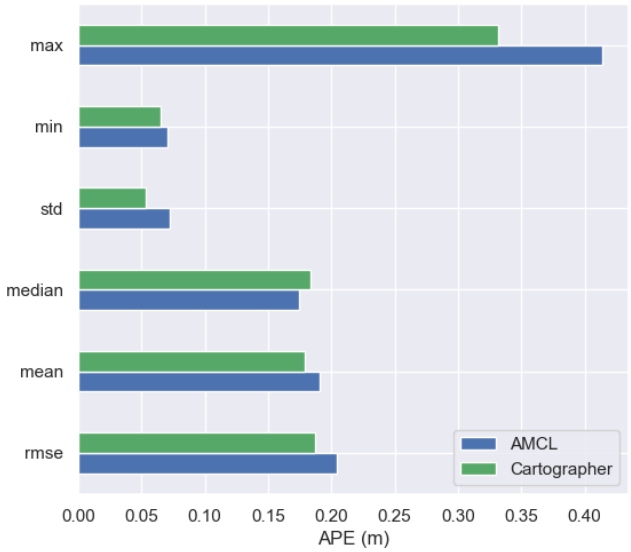


Fig. 4: Localization comparison

2) Cartographer Localization: To initiate Cartographer in its localization mode, the ".pbstream" file recorded during mapping is essential. In our implementation for 2D Cartographer localization, the pbstream file used is generated by the OIG2 configuration. We customized the original Cartographer_ROS package to ensure that incremental mapping is omitted in localization mode, focusing solely on pure localization.

Regarding Cartographer 3D localization, its implementation is not unnecessary as mapping in 3D mode is not feasible.

D. Evaluation

The EVO tool is employed to compare the pose output generated by the aforementioned mapping methods with the odometry ground truth values based on the *"base_link"* (published by */gazebo/ground-truth/state*). Overall, the 2D Cartographer with a configuration combining Odometry + IMU + GPS + 2D LiDAR achieves the highest performance and is selected as the localization algorithm for Task 2 Navigation.

III. Task2: Navigation

When a robot starts to navigation, self-localization is the first task to be solved for a mobile robot, which must identify its own position and direction as well as the obstacles in the environment. Locating the robot on its own is one of the critical challenges of mobile robots, which is the basis for the subsequent of path planning.

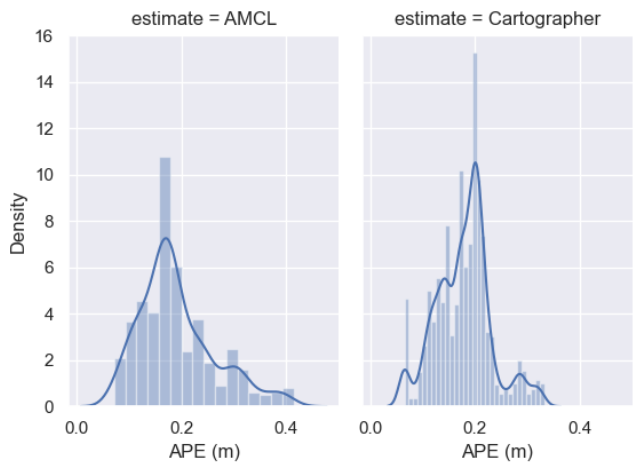


Fig. 5: APE evaluation

TABLE IV: Comparison of best mapping results of each algorithm

Metric (m)	Cartographer 2D	Cartographer 3D	Fast-LIO2
Max APE	0.401	2.843	0.936
Mean APE	0.123	0.161	0.399
Median APE	0.109	0.147	0.381
Min APE	0.064	0.090	0.041
RMSE	0.134	0.176	0.453

A. Global Planning

1) Costmap: A costmap is a representation of a robot's configuration space that facilitates smooth path generation while maintaining a safe distance from obstacles. The costmap_2d package in ROS allows customization of the costmap attributes. It builds a 2D or 3D occupancy grid from sensor data and inflates costs based on the occupancy grid and a user-specified inflation radius [4]. The inflater_layer optimizes the costmap by adding new values around lethal obstacles to represent the robot's configuration space.

The costmap consists of two components: the configuration space and the buffer zone. In costmap_2d, the buffer zone is modeled as a gradient descent function. The costmap is obtained by setting the following parameters:

- **inflation_radius**: This parameter refers to the robot's configuration space. In this work, it is set to 3.
- **cost_scaling_factor**: This is a scaling factor applied to cost values during inflation. The cost function is computed as follows for all cells in the costmap further than the inscribed radius distance and closer than the inflation radius distance away from an actual obstacle:

$$\begin{aligned}\exp -(&\text{cost\_scaling\_factor}\\ &\times(\text{distance\_from\_obstacle}\\ &-\text{inscribed\_radius})\\ &\times(\text{inflated\_obstacle\_param}-1)\end{aligned}$$

In this project, the value of **cost_scaling_factor** is set to 5.

2) Path Searching: The global planning process involves computing a static trajectory from the current pose to the designated goal pose. In this work, we chose the Theta* algorithm for global path planning. Theta* is an any-angle path planning algorithm that combines the strengths of A* and Dijkstra's algorithms while addressing their limitations [5]. It allows paths to propagate information along grid edges without constraining them to grid edges [6]. The key idea behind Theta* is to check the line-of-sight between the parent of the current node and each neighbor, allowing the path to deviate from grid edges when there are no obstacles.

Mathematically, let s be the start node, $g(n)$ be the actual cost from s to node n, and $h(n)$ be the heuristic estimate of the cost from n to the goal. Theta* updates the cost of each neighbor n' of the current node n as follows:

$$\begin{aligned}g(n') = &\min(g(n'), g(parent(n))\\ &+c(parent(n), n'))\end{aligned} \tag{1}$$

where $c(parent(n), n')$ is the cost of the straight line path from $parent(n)$ to n', if the line-of-sight exists.

We chose Theta* for our global path planning algorithm due to several advantages it offers over other classical algorithms:

- Generates shorter and more natural-looking paths compared to grid-based algorithms like A* [6].
- Guarantees finding the optimal path if one exists and is complete [5].
- Has a similar time complexity to A* but typically explores fewer nodes, making it more efficient in practice [5].
- Generates paths suitable for autonomous driving scenarios [7].

To enhance planning efficiency and conserve computational resources, we take global planning only upon receiving the planning request. In our setup, planning is triggered by sending the goal pose to the */move_base_simple/goal* topic.

B. Local Planning

The local planning can be divided into four parts: path segmentation, local re-planning, path interpolation, and tracking.

1) Path Segmentation: As the global path is generated only once, the robot can merely utilize it as a reference. Supposing the robot's current pose and the position of each point on the global path are known, the nearest point on the trajectory to the robot's current location can be determined first. Subsequently, given the size of the local search window, the intersection point along the global path can be identified. The segmented path consists of the points from the robot's current pose to this intersection point along the global path. Fig. 8 depicts an example of this procedure.

2) Local Re-planning: To prevent collisions with dynamic obstacles, a new path needs to be planned from the current pose to the intersection point on the global path segmented by the local search window. Besides, a smoother trajectory can significantly reduce the burden of the ensuing optimization

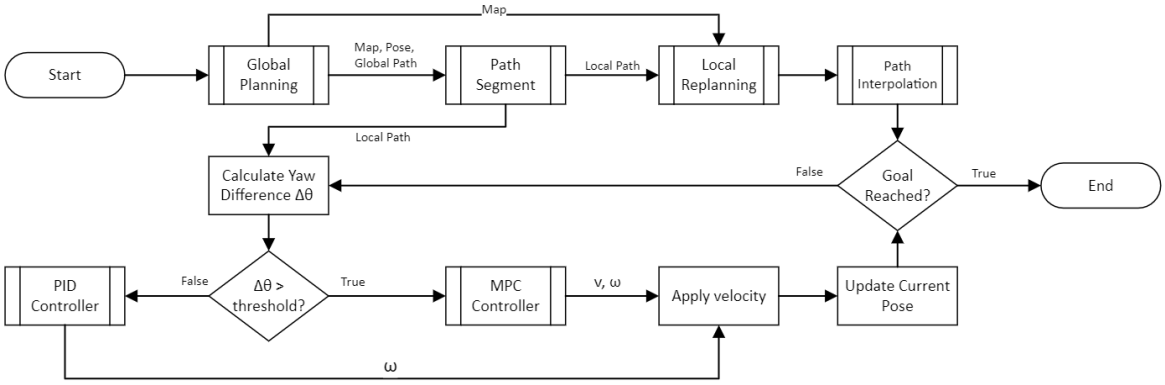


Fig. 6: Navigation pipeline

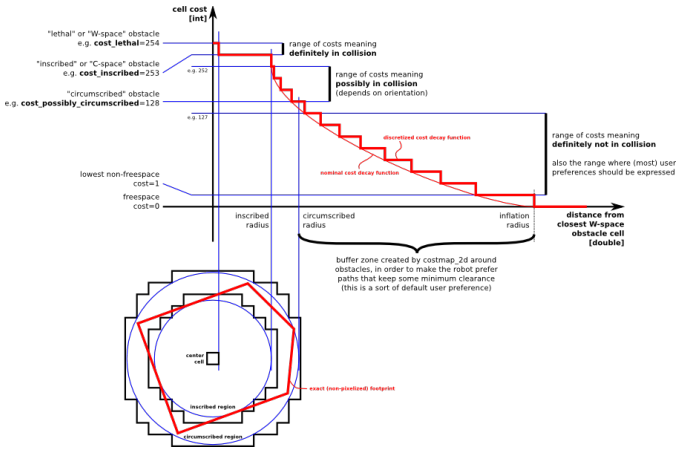


Fig. 7: Illustration of the cost calculation in the inflation layer [4].

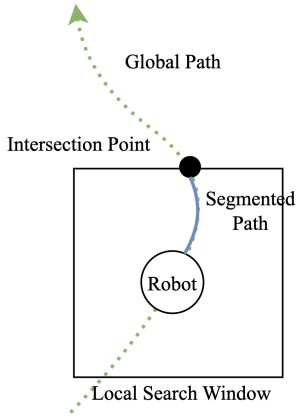


Fig. 8: Segment the global path based on local search window.

process and prevent conflicts between smoothness and collision avoidance. Therefore, the re-planned path should be as smooth as possible. In this case, we incorporate the costmap during local planning, resulting in a refined A* algorithm. The corresponding cost function of A* can be rewritten as:

$$f(n) = g(n) + h(n) + c(n) \tag{2}$$

where $c(n)$ is the value obtained from the costmap, $g(n)$ represents the actual cost from the starting point to the current point, and $h(n)$ is the estimated cost from the current point to the target node (heuristic function). As mentioned earlier, $c(n)$ is calculated in advance using a smooth approximation function, which guarantees that the cost value is continuous and smooth, guiding A* to find a path that minimizes the costmap, thus generating a smoothed path.

3) Path Interpolation: The local re-planned path points are discretely distributed based on the grid map resolution. Yet, in the moving horizon-based control algorithm, the time step and horizon length can significantly differ from the grid map path, leading to a prediction gap. To address this issue, we utilized linear interpolation to extend the global path segments to adapt to the controller's preference.

Let $\mathbf{p}(t) = (x(t), y(t), \theta(t))$ represent a 2D trajectory with orientation, where $t \in [0, 1]$ is the parameter, and $\theta(t)$ represents the orientation angle. Given two points on the trajectory, $\mathbf{p}_0 = \mathbf{p}(0) = (x_0, y_0, \theta_0)$ and $\mathbf{p}_1 = \mathbf{p}(1) = (x_1, y_1, \theta_1)$, the interpolated point $\mathbf{p}(t)$ at any parameter value $t \in [0, 1]$ can be expressed as:

$$\begin{aligned} x(t) &= (1-t)x_0 + tx_1 \\ y(t) &= (1-t)y_0 + ty_1 \\ \theta(t) &= \mathrm{Slerp}(\theta_0, \theta_1, t) \end{aligned} \tag{3}$$

where $\mathrm{Slerp}(\theta_0, \theta_1, t)$ represents the spherical linear interpolation between angles θ_0 and θ_1 at parameter t, defined as:

$$\mathrm{Slerp}(\theta_0, \theta_1, t) = \frac{\sin((1-t)\Omega)}{\sin(\Omega)}\theta_0 + \frac{\sin(t\Omega)}{\sin(\Omega)}\theta_1 \tag{4}$$

where Ω is the angle between θ_0 and θ_1, computed as:

$$\Omega = \arccos(\cos(\theta_1 - \theta_0)) \tag{5}$$

4) Tracking: Once the local reference path is obtained, the actual command needs to be calculated in order for robot to execute. In this task, the Model Predictive Controller (MPC) was adopted to realize trajectory tracking.

At each timestamp t, the MPC receives a state measurement and calculates the optimal control action sequence that drives the predicted system output to the desired reference. The

optimal control problem is converted into a nonlinear programming (NLP) problem using the direct multiple shooting method [8], as shown in Equation 6.

$$\begin{aligned} \min_{x_k, u_k} \quad & \sum_{k=0}^{N-1} \mathcal{L}(x_k, u_k) + V_f(x_N) \\ \text{s.t.} \quad & x_{k+1} = f(x_k, u_k), k = 0, \ldots, N-1 \\ & x_0 = \hat{x} \\ & x_k \in \mathcal{X}, \quad k = 1, \ldots, N \\ & u_k \in \mathcal{U}, \quad k = 0, \ldots, N-1 \\ & \dot{x}_k \in \dot{\mathcal{X}}, \quad k = 1, \ldots, N \end{aligned} \tag{6}$$

The objective function is defined as the quadratic error form:

$$\mathcal{L}(x_k, u_k) = \|x_k - x_k^r\|_Q + \|u_k - u_k^r\|_R \tag{7}$$

where $||\cdot||_\chi$ represents the quadratic form, and χ is a positive semi-definite diagonal matrix. $\mathbf{x}^r(k)$ denotes the reference state to be tracked. The first constraint represents the discrete system dynamics equation. The second constraint indicates the initial condition with x_0 being the real-time measured state value. The last two constraints denote the variable value limitations.

In our implementation, CasADi [9] is utilized as the MPC code formulation framework, and IPOPT [10] is utilized as the underlying non-linear optimization solver.

However, the MPC is not suitable for tracking the entire trajectory. One critical issue is that the MPC controller will try to meet the velocity and acceleration constraints, so the generated control command will cause the robot to turn in a large circle if the difference between the robot's yaw angle and the trajectory's tangential direction is large, as illustrated in Fig. 9.

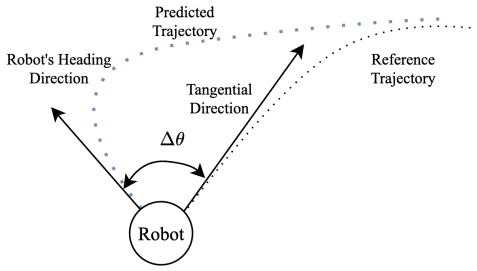


Fig. 9: High curvature trajectory caused by MPC at the beginning of tracking.

To address this issue, we implemented a PID controller to adjust the robot's yaw if $\Delta\theta$ is larger than a specified angle. The corresponding control law can be written as:

$$\omega^c(t) = k_p \Delta\theta(t) + k_i \int \Delta\theta(t)dt + k_d \Delta\dot{\theta}(t) \tag{8}$$

where $\omega^c(t)$ is the command angular velocity, and k_p, k_i, k_d are the tunable parameters. In the yaw tuning phase, the linear velocity is set to zero.

To summarize, the persudo of the proposed local planning algotithm is shown as Alg. 1.

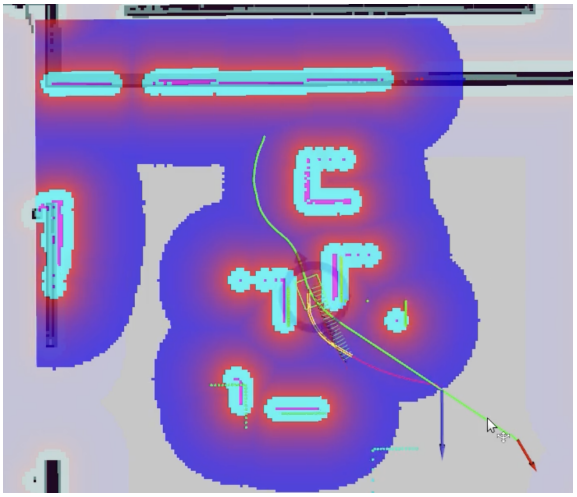


Fig. 10: An example of Local Path Planning. The green line shows the global path, the red line shows the locally re-planned path. The yellow line shows the interpolated path and the set of axes represent the predicted trajectory of the MPC.

An illustration of the proposed local path planning algorithm is shown in Fig. 10.

C. Evaluation

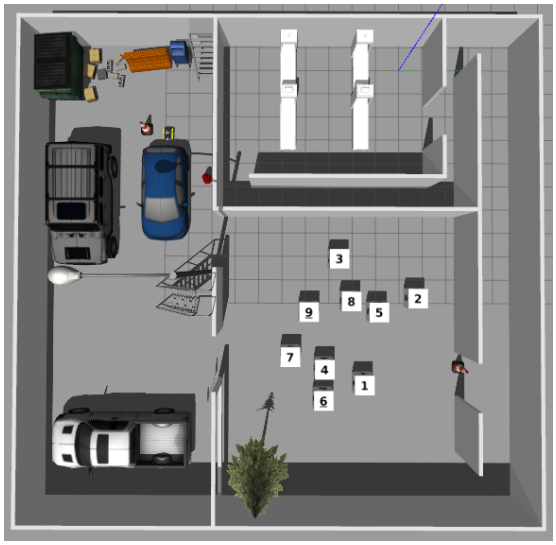
We conducted a series of experiments to rigorously evaluate the effectiveness of our designed algorithms. In our experiments, we maintained a constant simulation scenario (Fig. 11) with fixed number boxes.

First, we investigated the performance of different global path planning algorithms while keeping the local planner fixed to our proposed algorithm (Alg. 1). Tab. IX presents the experiment results, revealing that Theta* outperforms other algorithms in terms of explored cells, indicating superior search efficiency. Furthermore, Theta* generates smoother paths compared to A*, as illustrated in Fig. 12. This suggests that Theta* is more effective at finding optimal paths while minimizing unnecessary turns or maneuvers.

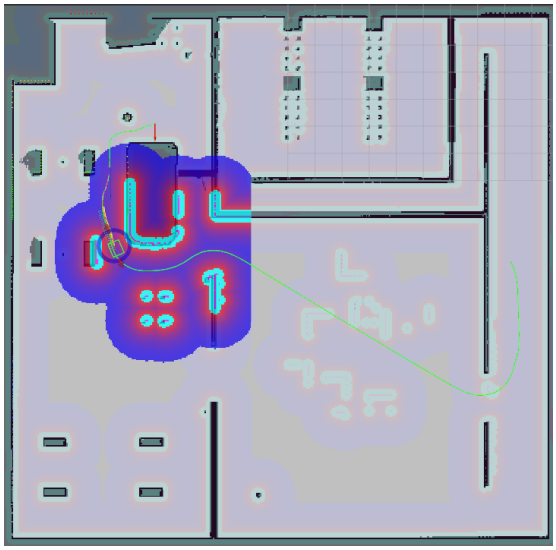
To thoroughly assess the effectiveness of our proposed local planning algorithm, we performed a comparative analysis against the classical Dynamic Window Approach (DWA) and Timed Elastic Band (TEB) local planners.

The specific parameter configurations of the Model Predictive Control (MPC) algorithm employed in our task are detailed in Tab. V. Here, f represents the control loop frequency, T indicates the time step, and N signifies the prediction horizon length. $v_{\max}$ and $\omega_{\max}$ denote the maximum allowable linear and angular velocities, while $a_{\max}$ and $\alpha_{\max}$ correspond to the maximum linear and angular accelerations, respectively.

Tab. VI presents the task completion times of different local planners under varying maximum linear velocity constraints while keeping other dynamic parameters constant. The results demonstrate that our method significantly outperforms the other planners across all speed configurations. Notably, TEB can successfully complete the planning task at $1.5\,\text{m/s}$, whereas DWA can only succeed at velocities below $1\,\text{m/s}$.



(a) Simulation scenario: the positions of boxes are predetermined.

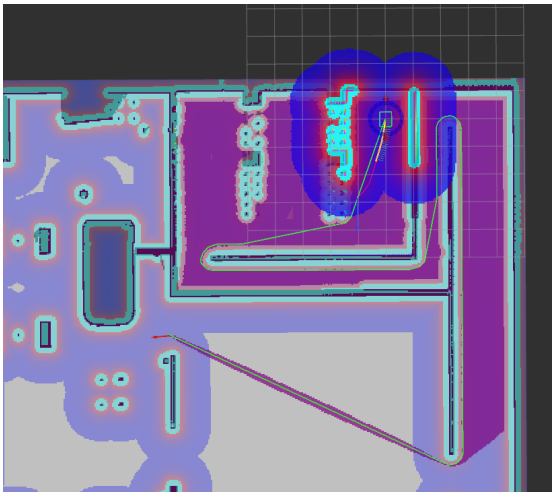


(b) Task Demonstration

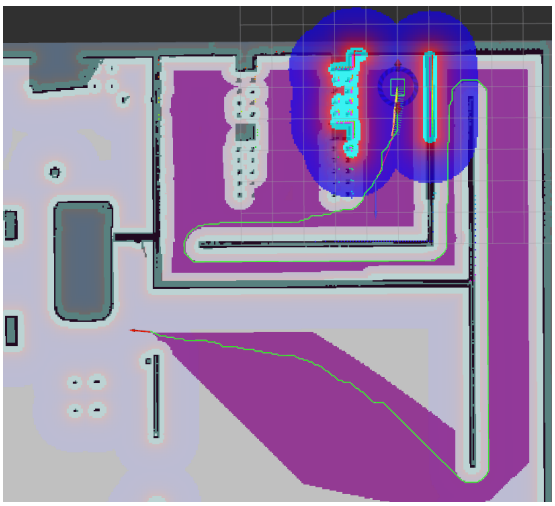
Fig. 11: Illustration of the benchmark task for local planner evaluation.

TABLE V: Parameter settings of MPC.

Parameter	Value	Parameter	Value
Q	diag(5,5,0)	$v_{\max}$	2 m/s
R	diag(0.01,0.2)	$\omega_{\max}$	2 rad/s
f	50 Hz	$a_{\max}$	15 m/s^2
T	0.05 s	$\alpha_{\max}$	10 rad/s^2
N	20		



(a) Theta*



(b) A*

Fig. 12: Global paths generated by different algorithms. The purple pixels indicate the visited cells during the search process.

This finding highlights the robustness and efficiency of our proposed algorithm in handling higher speed scenarios.

TABLE VI: Task completion times (in seconds) for different local planners. '-' indicates unsuccessful task completion.

$v_{\max}$	DWA	TEB	Ours
2.5	-	-	32.6
2	-	51.5	36.1
1.5	-	62.7	44.8
1	149.35	82.1	69.5

Furthermore, we compared the navigation errors of TEB and MPC in high-speed scenarios. The results, shown in Tab. VII, clearly indicate the superiority of MPC in maintaining lower pose and heading errors across different velocity settings. This suggests that MPC is more capable of precise trajectory tracking and maintaining stability even at higher speeds, which is crucial for safe and efficient navigation in dynamic environ-

ments. The corresponding dynamics smoothness comparisons are shown as Fig. 37.

TABLE VII: Comparison of navigation errors between TEB and MPC in high-speed scenarios.

v_{max}	Error Type	Values	
		TEB	MPC
2m/s	Pose	0.181	0.12
	Heading	0.004	0.074
1.5m/s	Pose	0.192	0.11
	Heading	0.019	0.05

IV. Task State Machine

A. Finite State Machine

For any automatic system, a finite state machine (or FSM) provides a framework for modeling operational modes of the system. It encapsulates the logic for state transitions in response to external and internal stimuli, facilitating a deterministic approach to the system's control logic.

In this project, the robot is required to navigate between the assembly line room, the random box room, and the parking lot. The terrain is complex and different for each room, especially in the random box room where visual and exploration modules are needed. To manage better the robot's current state, we create a state machine.

Based on the division of rooms, we designed three main states to manage the behavior of the robot within and between different rooms. For ease of control, these three states can be manually triggered through rviz. Additionally, we also designed several hidden states to manage operations within states and transitions between states.

- For convenience, we will refer to the three rooms as room 1, room 2, and room 3 respectively.

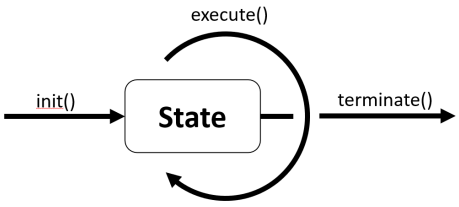


Fig. 13: An instance of a state

1) Implementation methodology: When designing the FSM, each state needs to have the following three functions: *init()*, *execute()*, and *terminate()* as shown in Fig. 13. At the beginning of the state, the *init* function is activated to publish the corresponding topics and activate the corresponding functionalities (such as road obstacle detection, exploration, etc.). In the main loop, the *execute()* function is continuously executed to determine if the robot has reached the next state or to repeat certain operations within the state. The *terminate* function is called when the state exits to terminate certain services and topics, thereby improving overall computational efficiency.

Fig. 14 is a demonstration of how the states in room 2, the most complex room, is managed.

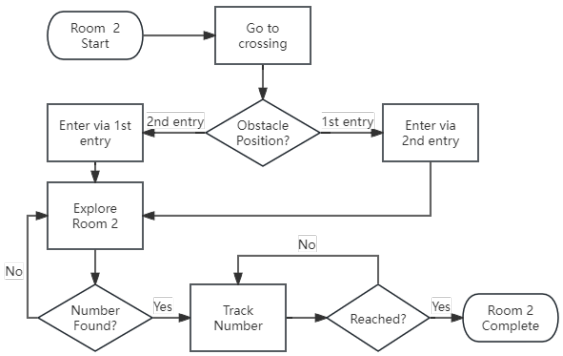


Fig. 14: Pipeline for room 2. Each process in the graph is implemented using a state.

B. Visual Perception

The field of visual perception has seen a significant amount of research. Over the years, a wide range of methods have been developed, ranging from simple template matching to sophisticated deep neural networks. In this task there are 2 specific usage of visual perception, including detecting the traffic cone and identifying numbers on the boxes. The first involves the detection of traffic cones, which can be accomplished through a relatively straightforward color-based detection approach. The second task focuses on identifying numbers printed on boxes, for which multiple techniques have been employed. These include optical character recognition (OCR) methods as well as multi-sensor position estimation algorithms.

1) Cone identifying: The first function of the visual perception system is to detect the presence of a traffic cone at a specific street corner as shown in Fig. 42. This task is simplified by the fact that the surrounding environment at the corner is predominantly gray, making the red components of the traffic cone more readily identifiable.

When the vehicle reaches a designated position near the corner, the FSM sends a command to the visual perception node. Upon receiving this command, the node captures an image from the camera and converts it to the HSV color space. The hue component of HSV represents the actual color, independent of brightness, unlike the RGB color space where the color is a combination of the intensity in all three channels. This separation allows for more efficient isolation of specific color ranges, regardless of variations in brightness or intensity.

In contrast, gray colors in the RGB color space always contain non-zero intensity in all three channels, complicating the process of detecting the traffic cone against the gray background.

2) Number detection: The other task of visual perception is to identify the number on the boxes within the designated region.

Under ideal conditions, this task could be broken down into several steps: image segmentation to isolate the objects from the background, object pose estimation to determine the orientation of each box, and number detection to identify the digits on the boxes. This comprehensive approach would

provide a detailed understanding of the entire environment and the specific information associated with each box. However, such a multi-component system would require several complex neural networks, which can be computationally intensive and challenging to train effectively.

For instance, the robot may need to detect numbers from various viewpoints, requiring the ability to recognize digits of different sizes and orientations. Additionally, the specific environmental conditions of this task may not be well-represented in existing training datasets, while creating a custom dataset could be a complicated and uneven process, potentially leading to overfitting.

Therefore, instead of training neural networks from scratch, the team opted to utilize Optical Character Recognition (OCR) for number identification. Similarly, for box pose estimation, a simple and straightforward method was applied, rather than relying on neural networks or point cloud clustering techniques.

The OCR solution employed in this application utilizes the open-source library, EasyOCR (Fig. 43. EasyOCR leverages deep learning models to perform text detection and recognition on images or documents. Its workflow encompasses several key steps: preprocessing, text detection, text recognition, and postprocessing.

The core components of the EasyOCR system are the text detection and recognition models. For text detection, EasyOCR employs the CRAFT algorithm, which is a CNN-based network that provides bounding boxes around potential text regions. The text recognition model is a CRNN (Convolutional Recurrent Neural Network) architecture, composed of feature extraction (using ResNet and VGG), sequence labeling (LSTM), and decoding (CTC).

After detecting the numbers on the boxes, the next step is to determine which specific box each number is associated with. Instead of attempting to identify and estimate the pose of each individual box, a more subtle and straightforward approach is adopted in this implementation.

Rather than obtaining the full pose of the boxes, determining the goal position for the robot to approach is sufficient. To find this goal position, depth information is required. The solution leverages the complementary capabilities of the camera and LiDAR sensors - the camera provides the text information, while the LiDAR provides the depth data.

Specifically, when a number is detected in the current camera frame, the corresponding planar LiDAR scan result is also stored. The direction of the detected number, relative to the camera, is calculated based on the bounding box position in the image and the camera parameters. This direction is then transformed into the LiDAR frame of reference. Within the LiDAR frame, the point with the smallest angle from the calculated direction is considered the position of the detected number. Once detecting the goal number, the node will publish its pose but only position in which works. In practice, a filter is applied to the detection result to smooth out any inconsistencies. The final detected position is then published as the goal position for the robot to approach. A visualization of the result can be found at Fig. 44.

C. Exploration

Autonomous exploration is essential for robots to navigate unfamiliar environments without human intervention. Currently, there are three main different approaches. Frontier-based methods prioritize unexplored areas, adapting strategies based on updated data. Probabilistic techniques, such as occupancy grid mapping, aid decision-making by constructing maps from sensor data. Machine learning, including reinforcement and deep learning methods, enhances exploration by learning efficient policies from data. In our case, we used a altered version of *explore_lite*, a frontier-based exploration algorithm.

1) Algorithm description: The *explore_lite* package uses a Greedy frontier-based exploration algorithm. This is a popular strategy utilized by robots to autonomously explore unknown environments. In this approach, the robot prioritizes navigating towards unexplored areas, known as frontiers, in a greedy manner, aiming to maximize the coverage of the environment (Fig. 38). By continually updating the frontier information and selecting the closest frontier as the next target, the robot efficiently explores the environment while avoiding unnecessary backtracking. This method proves effective in our scenario, where we would have to explore an unknown room full of unknown numbers. By traversing the entire room, the correct box position can be found.

2) Alterations: In order to make *explore_lite* more suitable for our project pipeline, we implemented the following changes to the original source code: Firstly, we adjusted how *explore_lite* updates itself, allowing it to explore only when prompted by the state machine, thus remaining idle at other times to reduce performance overhead. Secondly, we modified how exploration targets are published and how maps are received to ensure seamless integration into the project without affecting the navigation module, where a global map is maintained. Finally, we introduced a new Python script to process the OccupacyGridMap produced by the mapping algorithm, thresholding the map so that the exploration module can function better.

D. Incremental mapping

In the preceding text, we utilized a map exploration mechanism to actively search for target boxes. However, for effective exploration, we need to re-map the local area of room 2 based on the map constructed in the mapping section.

Common 2D mapping algorithms like Cartographer and Gmapping typically create a map-to-odometry transform during mapping, facilitating simultaneous localization. However, our experiment with different localization methods necessitates a mapping-only approach utilizing the tf tree published by the localization algorithm for map transform. Integration of localization within SLAM algorithms leads to conflicts in the tf tree, and would disrupt the whole system.

In our case, we utilized, modified and tuned a simple mapping repository named *occ_grid_mapping*.

1) Algorithm description: occ_grid_mapping or occupancy grid mapping, uses probabilistic models to account for uncertainty in sensor measurements. By incorporating sensor noise

and uncertainties, these algorithms can make informed decisions about the occupancy status of grid cells. The probabilistic models used in this algorithm is the log-odds representation, which maintains a log-odds ratio of the probability that a cell is occupied versus free.

The main difference between this algorithm and Gmapping or Cartographer is that this algorithm uses the direct updates based on the probabilities of the occupancy grid map, rather than updating based on extracted features like other SLAM algorithms. This fundamental distinction is the primary reason for us choosing this algorithm.

2) Alterations: To enable it to read pre-built maps and accept the transform of the tf tree as the pose of the LiDAR, we made extensive modifications to the source code of this library. These modifications include but are not limited to rewriting global map data back into probability map data, modifying tf, and adjusting publisher and subscriber topics.

Additionally, the default mapping performance of this algorithm is extremely poor, as it tends to label obstacles as free space (Fig. 40). By adjusting parameters of the Bayesian filter in the source code, modifying grid confidence settings after each scan, and adjusting the threshold for processed exploration map, we successfully improved the mapping accuracy of the algorithm (Fig. 41). This enhancement allows it to provide stable and reliable information about explored, unexplored, and obstacle areas for our exploration algorithms.

E. Discussions

Due to the tight project schedule, many features were implemented in a relatively rudimentary manner. Originally, we planned that after completing the exploration process, if the target box location was not found, we would perform rectangle detection in the map of room 2. Subsequently, the robot would directly navigate to the positions of those boxes that did not have their corresponding numbers obtained, in order to get the corresponding numbers for every box. However, in practice, the robot almost always found the boxes corresponding to their goal number before the mapping was completed. Therefore, we deemed this feature to be of little significance and did not implement it in the end.

Regarding the exploration aspect, we initially aimed to use fully automatic exploration during the mapping task of task 1 and made some attempts. However, compared to manually operating the robot, automatic exploration provided overly random target points, often requiring the robot to rotate more than 90 degrees in place to follow the local path. Additionally, the 2D lidar cannot detect low obstacles, so the robot could get stuck by such obstacles during exploration, As a result, we only kept partial mapping videos as a feasibility demonstration and did not succeed in allowing the SLAM algorithm to automatically complete the entire map.

Lastly regarding to perception, the position of the detected number is determined now by directly selecting the LiDAR point that has the smallest angular deviation from the calculated direction of the number. However, a more refined approach to estimating the goal position would be to use interpolation. Instead of relying on a single LiDAR point, a line can be drawn between the two LiDAR points closest to the calculated direction. The intersection of this line and the detected direction vector can provide a more accurate estimation of the number's position. This interpolation-based approach leverages the additional spatial information available from the surrounding LiDAR points, rather than solely relying on the closest point. By considering the local geometry of the LiDAR data, the estimated position of the detected number can be improved, potentially leading to a more precise goal point for the robot to approach.

REFERENCES

[1] A. Dwijotomo, M. A. Abdul Rahman, M. H. Mohammed Ariff, H. Zamzuri, and W. M. H. Wan Azree, "Cartographer slam method for optimization with an adaptive multi-distance scan scheduler," *Applied Sciences*, vol. 10, no. 1, p. 347, 2020.

[2] W. Xu, Y. Cai, D. He, J. Lin, and F. Zhang, "Fast-lio2: Fast direct lidar-inertial odometry," *IEEE Transactions on Robotics*, vol. 38, no. 4, pp. 2053–2073, 2022.

[3] M.-A. Chung and C.-W. Lin, "An improved localization of mobile robotic system based on amcl algorithm," *IEEE Sensors Journal*, vol. 22, no. 1, pp. 900–908, 2021.

[4] "ROS costmap_2d," ROS Wiki. [Online]. Available: http://wiki.ros.org/costmap_2d

[5] A. Nash, S. Koenig, and C. Tovey, "Theta*: Any-angle path planning on grids," in *AAAI*, vol. 7, 2007, pp. 1177–1183.

[6] K. Daniel, A. Nash, S. Koenig, and A. Felner, "Theta*: Any-angle path planning on grids," *Journal of Artificial Intelligence Research*, vol. 39, pp. 533–579, 2010.

[7] H. Niu, Y. Lu, A. Savvaris, and A. Tsourdos, "An efficient path planning algorithm for unmanned surface vehicles," *IFAC-PapersOnLine*, vol. 51, no. 29, pp. 121–126, 2018.

[8] C. Kirches, *The Direct Multiple Shooting Method for Optimal Control*. Wiesbaden: Vieweg+Teubner Verlag, 2011, pp. 13–29. [Online]. Available: https://doi.org/10.1007/978-3-8348-8202-8_2

[9] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "CasADi – A software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation*, 2018.

[10] A. Wächter and L. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical Programming*, vol. 106, no. 1, pp. 25–57, 2006. [Online]. Available: https://doi.org/10.1007/s10107-004-0559-y

APPENDIX

A. Task 1

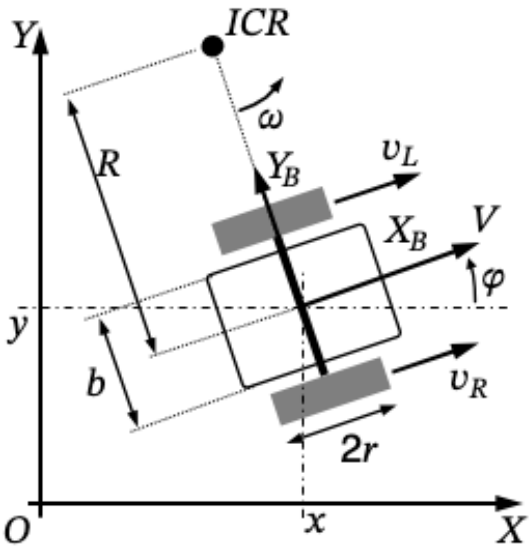


Fig. 15: Wheel odometry model.

1) Kinematics Model: Given:

- r: radius of the driven wheels
- L: wheelbase (distance between the centers of the driven wheels)
- ω_{left}: rotation rate of the left wheel (in radians per second)
- ω_{right}: rotation rate of the right wheel (in radians per second)
- Δt: time interval between measurements

The linear velocity of each wheel.

$$\begin{aligned} v_{\text{left}} &= r \times \omega_{\text{left}} \\ v_{\text{right}} &= r \times \omega_{\text{right}} \end{aligned} \tag{9}$$

The linear and angular velocities of the car.

$$\begin{aligned} v &= \frac{v_{\text{left}} + v_{\text{right}}}{2} \\ \omega &= \frac{v_{\text{right}} - v_{\text{left}}}{L} \end{aligned} \tag{10}$$

The linear and angular displacements of the car.

$$\begin{aligned} \Delta s &= v \times \Delta t \\ \Delta\theta &= \omega \times \Delta t \end{aligned} \tag{11}$$

Update the position and orientation of the car.

$$\begin{aligned} x_{\text{new}} &= x_{\text{old}} + \Delta s \times \cos(\theta_{\text{old}} + \frac{\Delta\theta}{2}) \\ y_{\text{new}} &= y_{\text{old}} + \Delta s \times \sin(\theta_{\text{old}} + \frac{\Delta\theta}{2}) \\ \theta_{\text{new}} &= \theta_{\text{old}} + \Delta\theta \end{aligned} \tag{12}$$

Where x_{old}, y_{old}, and θ_{old} are the previous position and orientation of the car.

B. Task2

C. Finate State Machine

TABLE VIII: Cartographer parameter tuning experiments. NRD: *num_range_data*, OENN: *optimize_every_n_nodes*, MS: *constraint_builder.min_score*, LSW: *linear_search_window*, ASW: *angular_search_window*, TDCW: *translation_delta_cost_weight*, RDCW: *rotation_delta_cost_weight*

Index	APE Max	APE Mean	APE RMSE	NRD	OENN	MS	LSW	ASW	TDCW	RDCW
1	2.635	0.358	0.668	35	35	0.65	-	-	-	-
2	3.098	0.452	0.860	35	10	0.65	-	-	-	-
3	1.627	0.197	0.373	140	35	0.65	-	-	-	-
4	1.512	0.198	0.353	140	35	0.65	0.1	35	10	0.1
5	0.479	0.154	0.178	280	35	0.65	0.1	35	10	0.1
6	0.223	0.151	0.160	1000	35	0.65	0.1	35	10	0.1
7	0.224	0.151	0.160	2000	35	0.65	0.1	35	10	0.1
8	0.223	0.151	0.160	1000	5	0.65	0.1	35	10	0.1
9	3.997	0.467	0.963	35	5	0.65	0.1	35	10	0.1
10	3.990	0.488	0.998	35	1	0.65	0.1	35	10	0.1
11	3.302	0.937	1.139	10	5	0.65	0.1	35	10	0.1
12	2.314	0.429	0.721	70	5	0.65	0.1	35	10	0.1
13	1.077	0.175	0.257	140	5	0.65	0.1	35	10	0.1
14	4.751	1.331	1.968	140	5	0.1	0.1	35	10	0.1
15	1.405	0.192	0.301	140	5	1.2	0.1	35	10	0.1
16	4.725	1.314	1.956	140	5	0.8	0.1	35	10	0.1
17	4.756	1.278	1.911	140	5	0.65	0.5	35	10	0.1
18	4.756	1.304	1.941	140	5	0.65	0.2	35	10	0.1
19	4.750	1.337	1.972	140	5	0.65	0.025	35	10	0.1
20	4.753	1.319	1.957	140	5	0.65	0.1	70	10	0.1
21	4.748	1.339	1.976	140	5	0.65	0.1	15	10	0.1
22	1.079	0.175	0.258	140	5	0.65	0.1	35	20	0.1
23	1.077	0.174	0.255	140	5	0.65	0.1	35	1000	0.1
24	1.925	0.242	0.441	140	5	0.65	0.1	35	20	1.0
25	0.223	0.151	0.160	1000	35	0.65	0.1	35	20	0.1
26	0.223	0.151	0.160	1000	5	0.65	0.1	35	20	0.1
27	**0.223**	**0.151**	**0.160**	1000	5	0.5	0.1	35	20	0.1

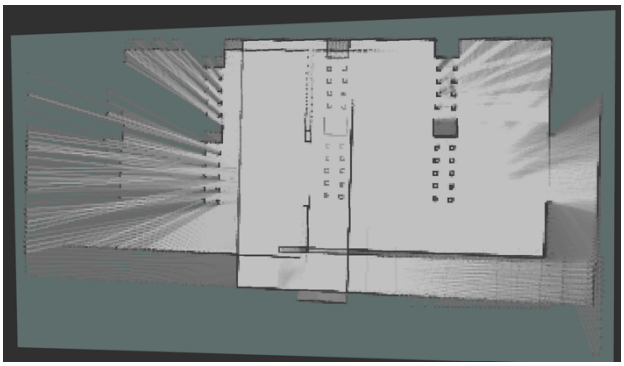


Fig. 16: 2D Cartographer map, IMU + 2D LiDAR

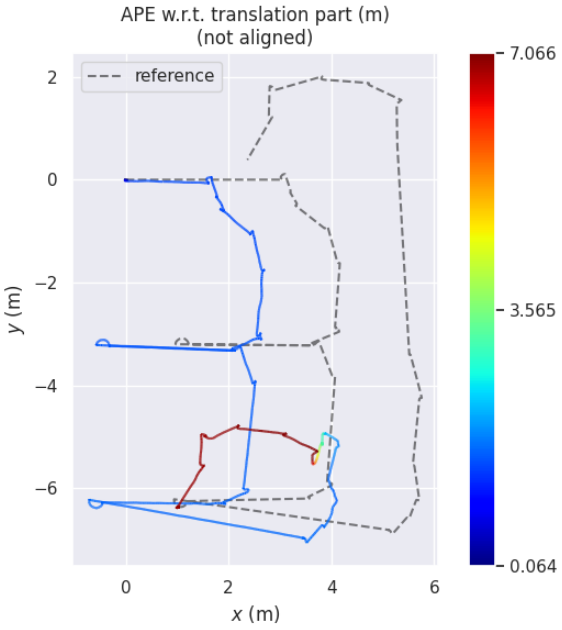


Fig. 17: APE of 2D Cartographer map, IMU + 2D LiDAR

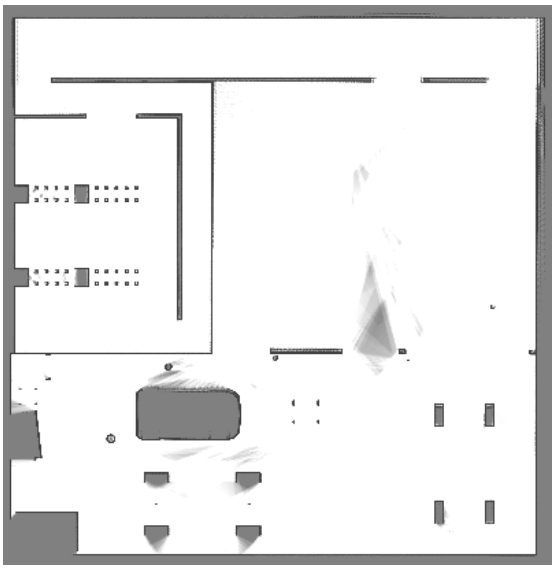


Fig. 18: 2D Cartographer map, Odometry + 2D LiDAR

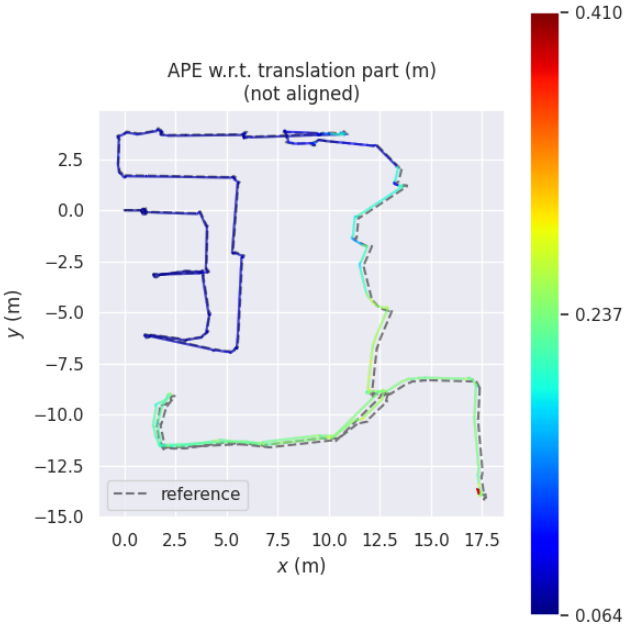


Fig. 19: APE of 2D Cartographer map, Odometry + 2D LiDAR

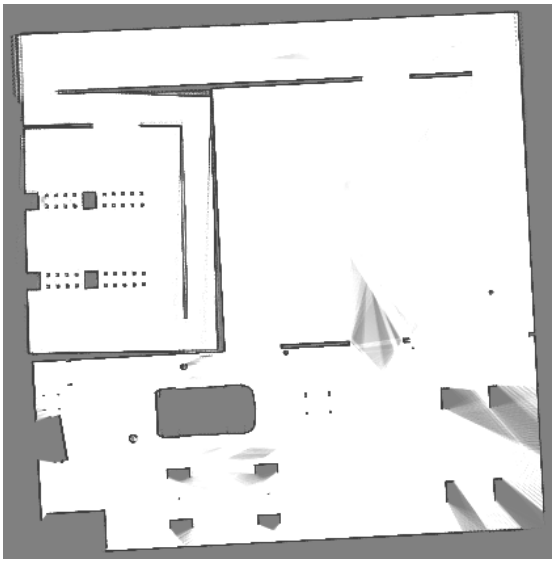


Fig. 20: 2D Cartographer map, Odometry + IMU + 2D LiDAR

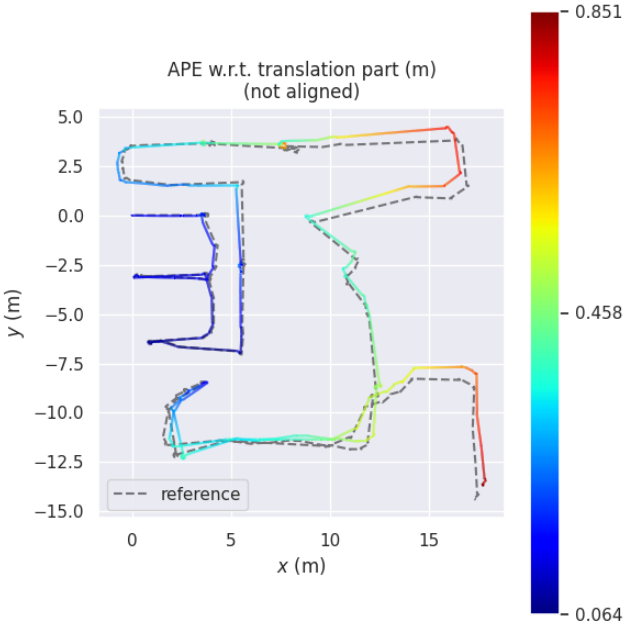


Fig. 21: APE of 2D Cartographer map, Odometry + IMU + 2D LiDAR

TABLE IX: Comparison of global path planning algorithms.

Algorithm	Pose error	Heading error	Visited cells	Computation Time (s)	Task Duration (s)
A*	0.829	0.082	137648	0.40	35.59
D*	0.724	0.082	151987	0.48	34.87
Dijkstra	0.907	0.079	180236	0.54	35.49
Theta*	0.770	0.081	122785	0.39	35.14

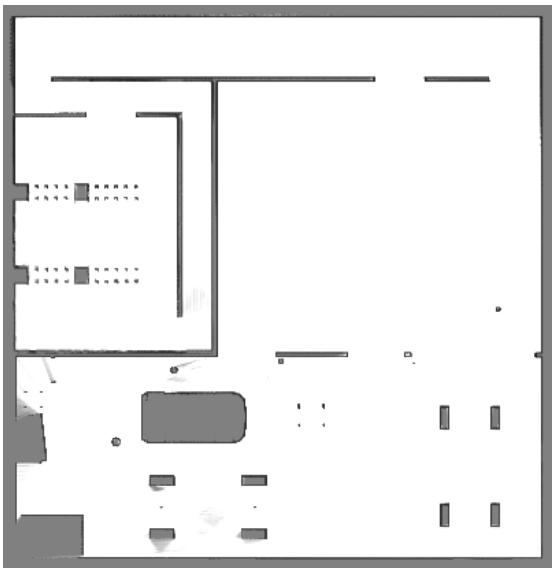


Fig. 22: 2D Cartographer map, Odometry + IMU + GPS + 2D LiDAR

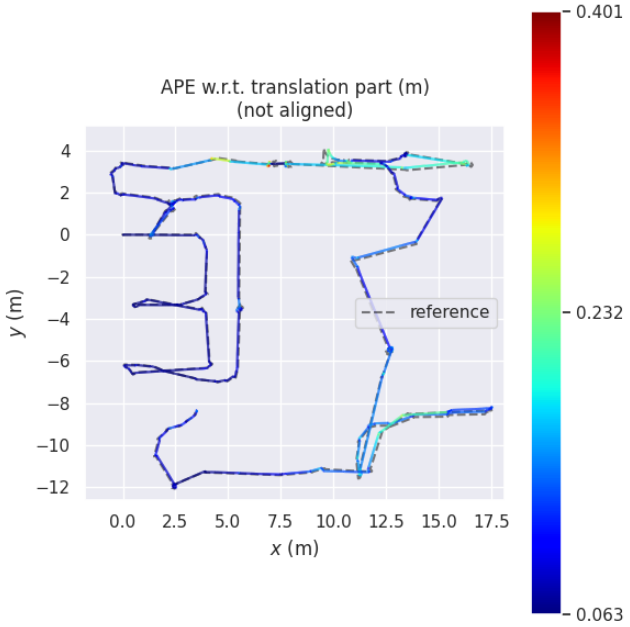


Fig. 23: APE of 2D Cartographer map, Odometry + IMU + GPS + 2D LiDAR

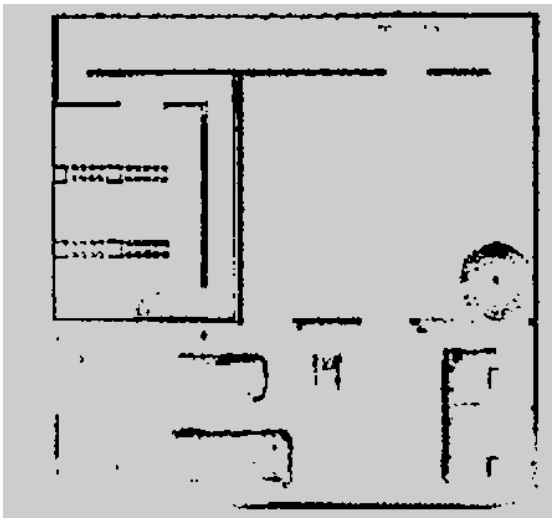


Fig. 24: 3D Cartographer map, Odometry + 3D LiDAR

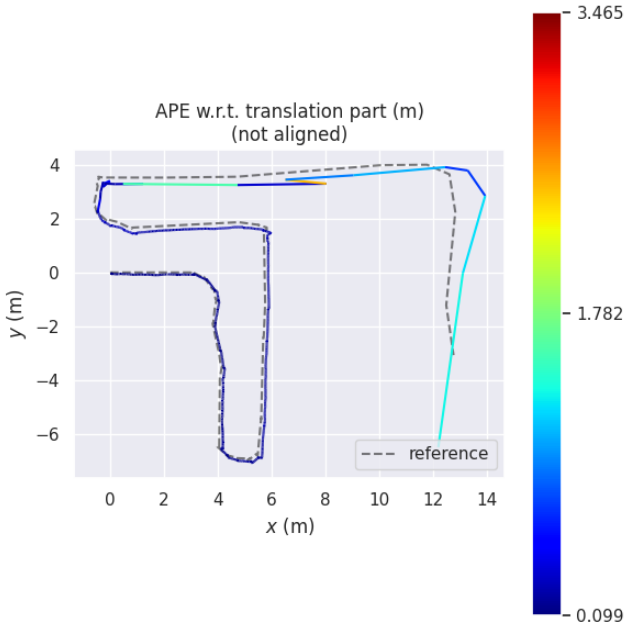


Fig. 25: APE of 3D Cartographer map (GPU), Odometry + 3D LiDAR

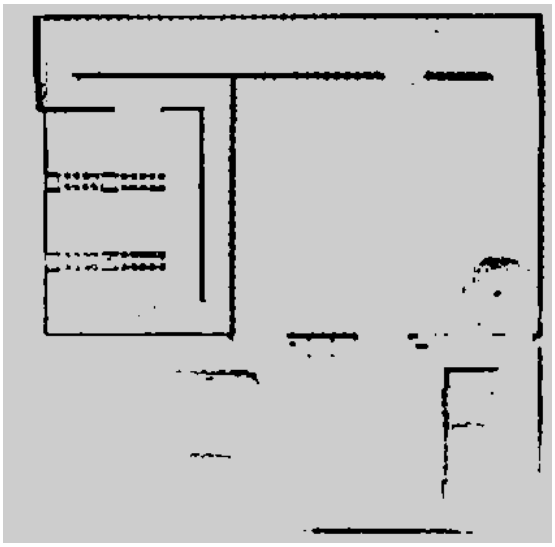


Fig. 26: 3D Cartographer map, Odometry + GPS + 3D LiDAR

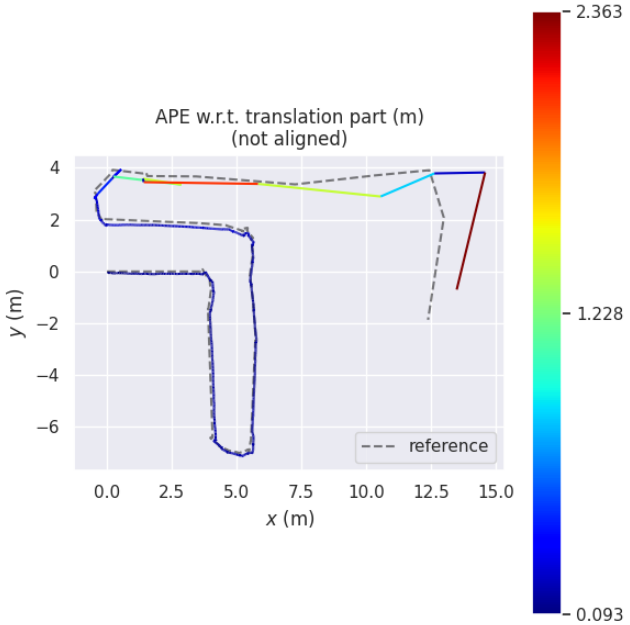


Fig. 27: APE of 3D Cartographer map, Odometry + GPS + 3D LiDAR

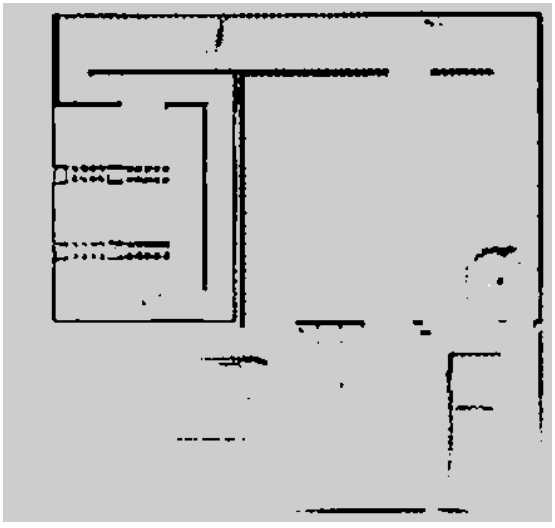


Fig. 28: 3D Cartographer map, Odometry + IMU + GPS + 3D LiDAR

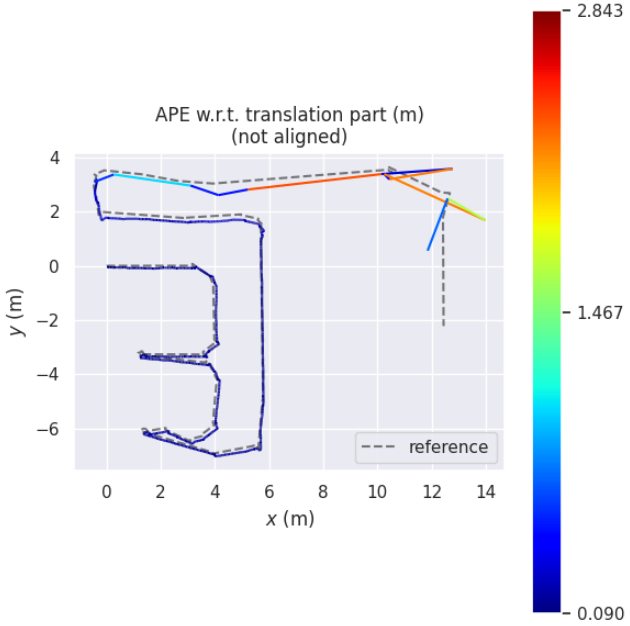


Fig. 29: APE of 3D Cartographer map, Odometry + IMU + GPS + 3D LiDAR

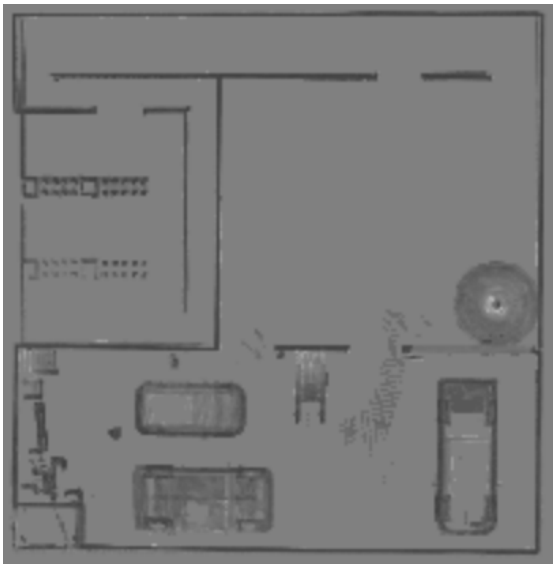


Fig. 30: 3D Cartographer map (CPU), Odometry + 3D LiDAR

Fig. 31: Point cloud by Fast-LIO2 in low speed

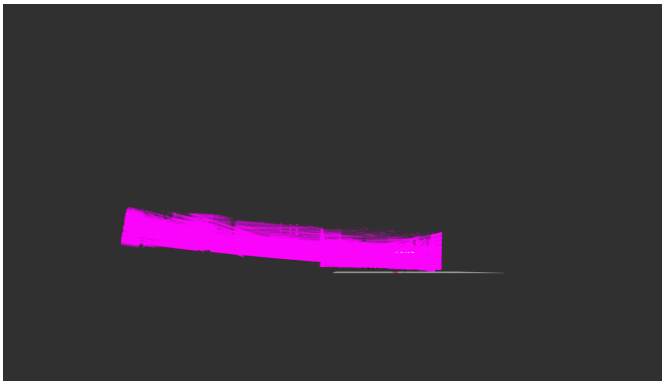


Fig. 32: Point cloud by Fast-LIO2 in high speed

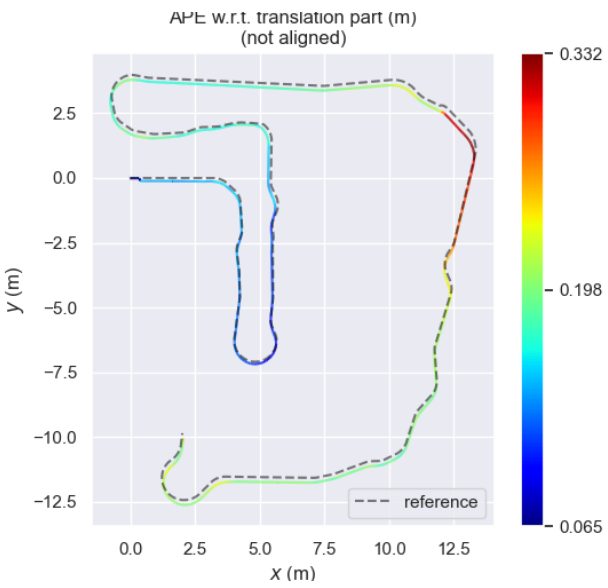


Fig. 33: Cartographer-map

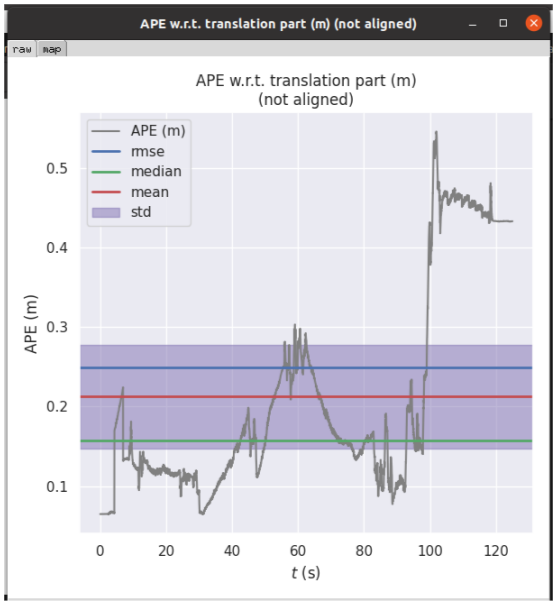


Fig. 34: Cartographer-APE

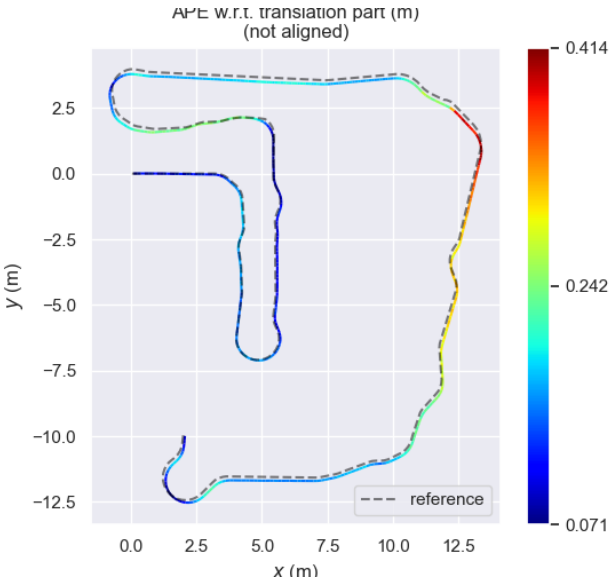


Fig. 35: AMCL-map

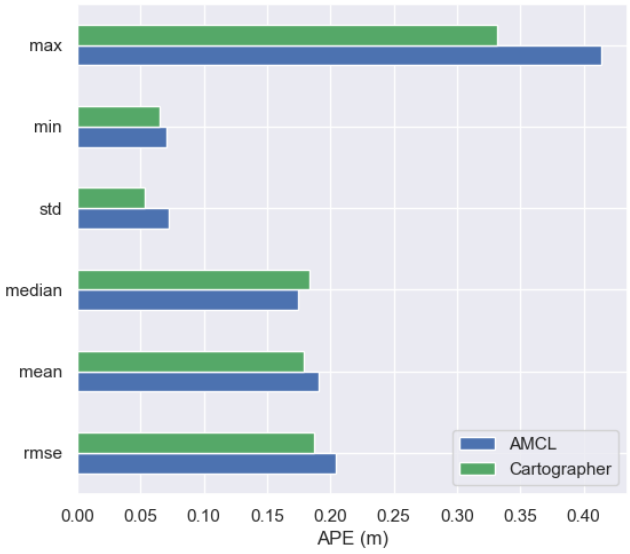
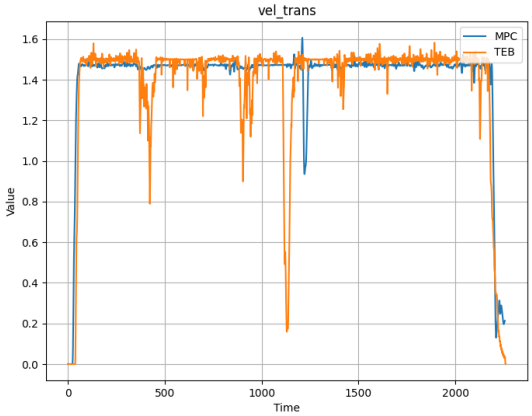
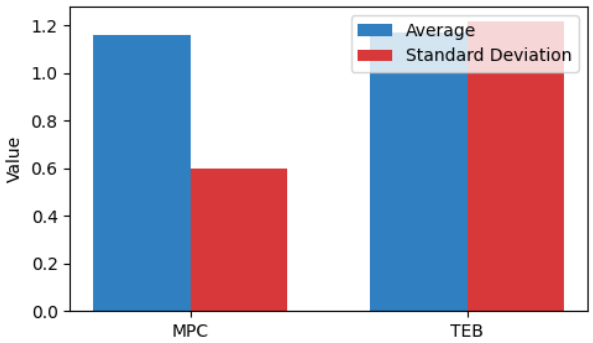


Fig. 36: Comparison AMCL with Cartograph



(a) Velocity



(b) Acceleration

Fig. 37: Comparison between TEB and MPC on dynamics smoothness.

Algorithm 1 Local Planning Algorithm

procedure LOCALPLANNING(G_P, C_P, C_M)
  G_P: Global Path, C_P: Current Pose, C_M: Current Map
  $L_P \leftarrow$ PathSegmentation(G_P, C_P)
  $L_P \leftarrow$ LocalReplan(L_P, C_M)
  $L_P \leftarrow$ PathInterpolation(L_P)
  $StartFlag \leftarrow$ True
  while ¬GoalReached() **do**
    $\Delta\theta \leftarrow$ CalculateYawDifference(C_P, L_P)
    if $\Delta\theta > \theta_{th}$ **and** StartFlag **then**
      $\omega \leftarrow$ PIDController($\Delta\theta$)
      ApplyAngularVelocity(ω)
    else
      $StartFlag \leftarrow$ False
      $(v, \omega) \leftarrow$ MPCController(C_P, L_P)
      ApplyVelocities(v, ω)
    end if
    $C_P \leftarrow$ UpdateCurrentPose()
  end while
end procedure
function PATHSEGMENTATION(G_P, C_P)
  $L_P \leftarrow$ SelectSegment(G_P, C_P) **return** L_P
end function
function LOCALREPLAN(L_P, C_M)
  $L_P \leftarrow$ RefinedAStarPlanner(L_P, C_M) **return** L_P
end function
function PATHINTERPOLATION(L_P)
  $L_P \leftarrow$ InterpolatePath(L_P) **return** L_P
end function

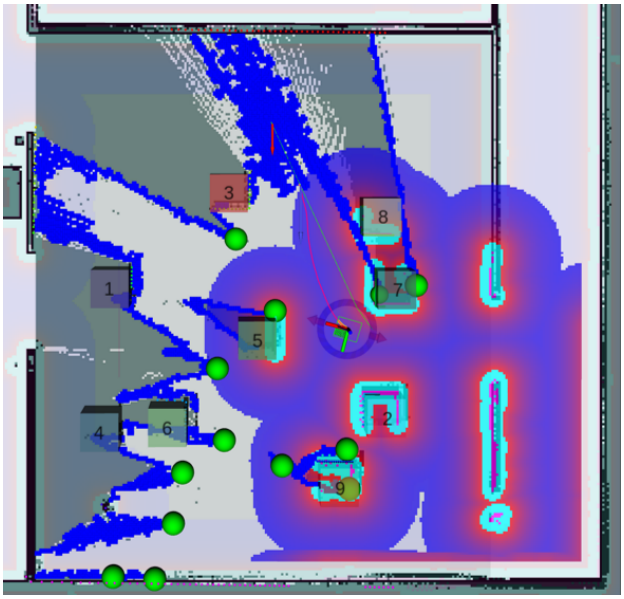


Fig. 38: Exploration frontiers (deep blue) and explore candidates (green)

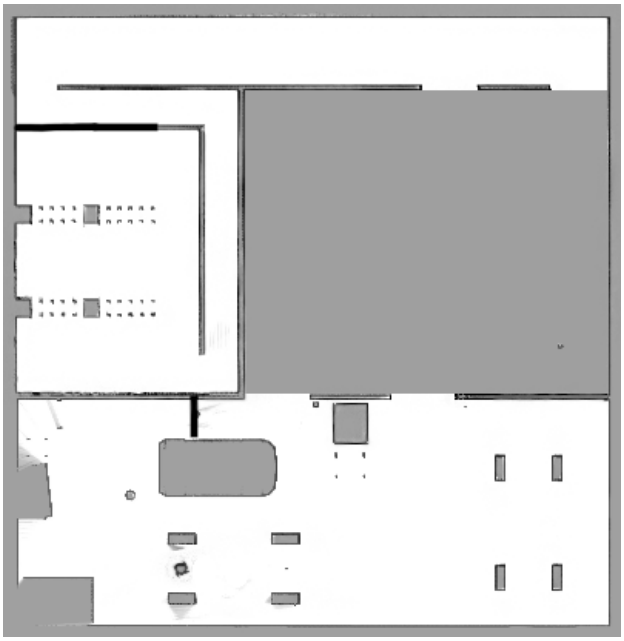


Fig. 39: Map used for exploration as reference for mapping algorithms

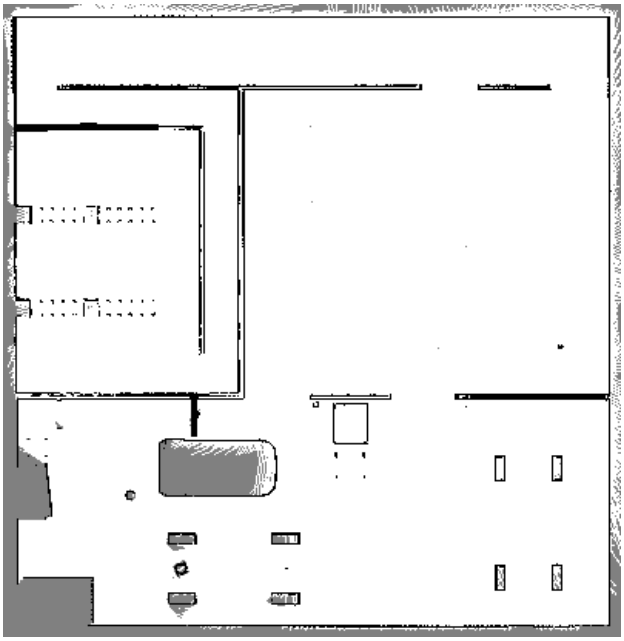


Fig. 40: Map built after exploration without parameter tuning. Note that though there are obstacles in the unknown space, the mapping incorrectly mapped them all as free space

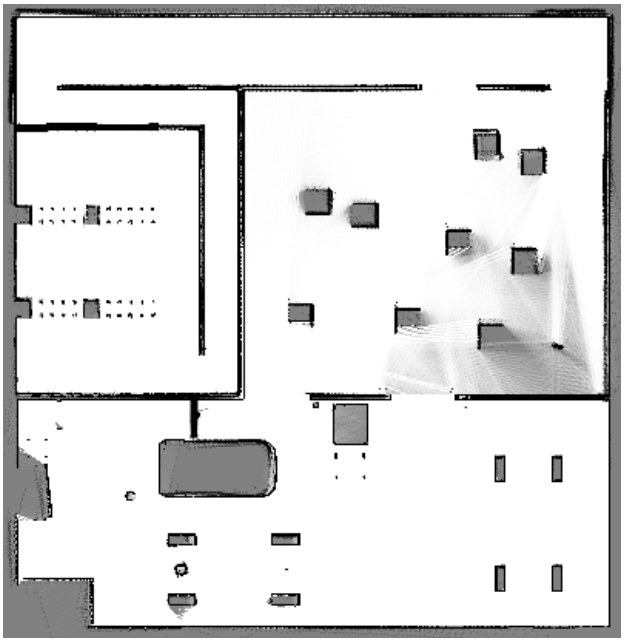


Fig. 41: Map built after exploration. Note that the boxes are now correctly mapped as obstacles

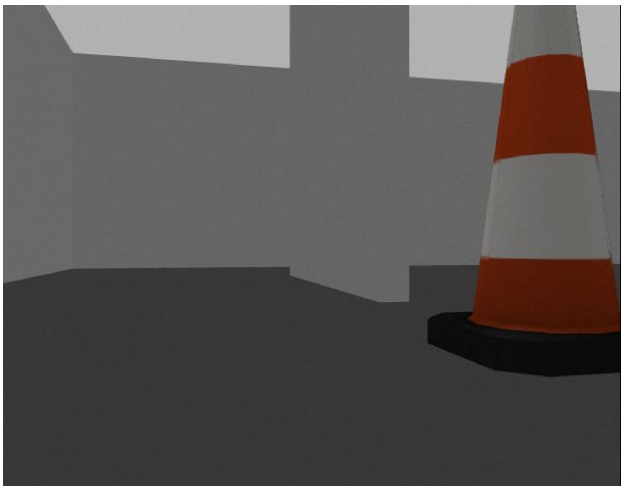


Fig. 42: Cone for Detection

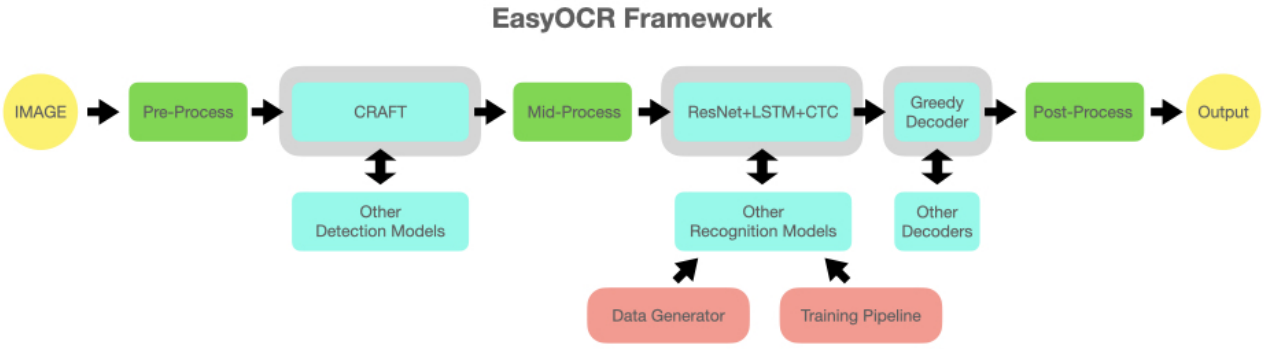


Fig. 43: Framework for EasyOCR

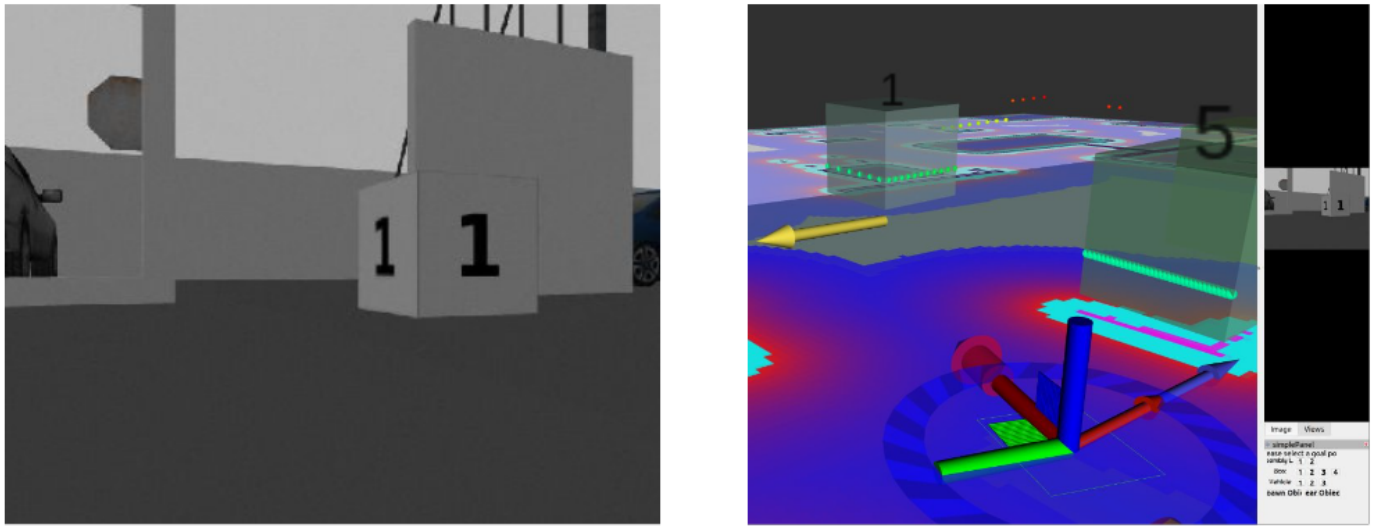


Fig. 44: Visual Detection and Result