ME5413 Final Project
Group 10

Cao Chenyu Li Zhangjin Wang Yuanlong
Zhao Huaiyi Zhao Xu Zhu Rong

2024-04-11

CONTENTS

01 Project Overview

02 Task 1: Mapping

03 Task2: Navigation

04 Future Work

01

Project Overview

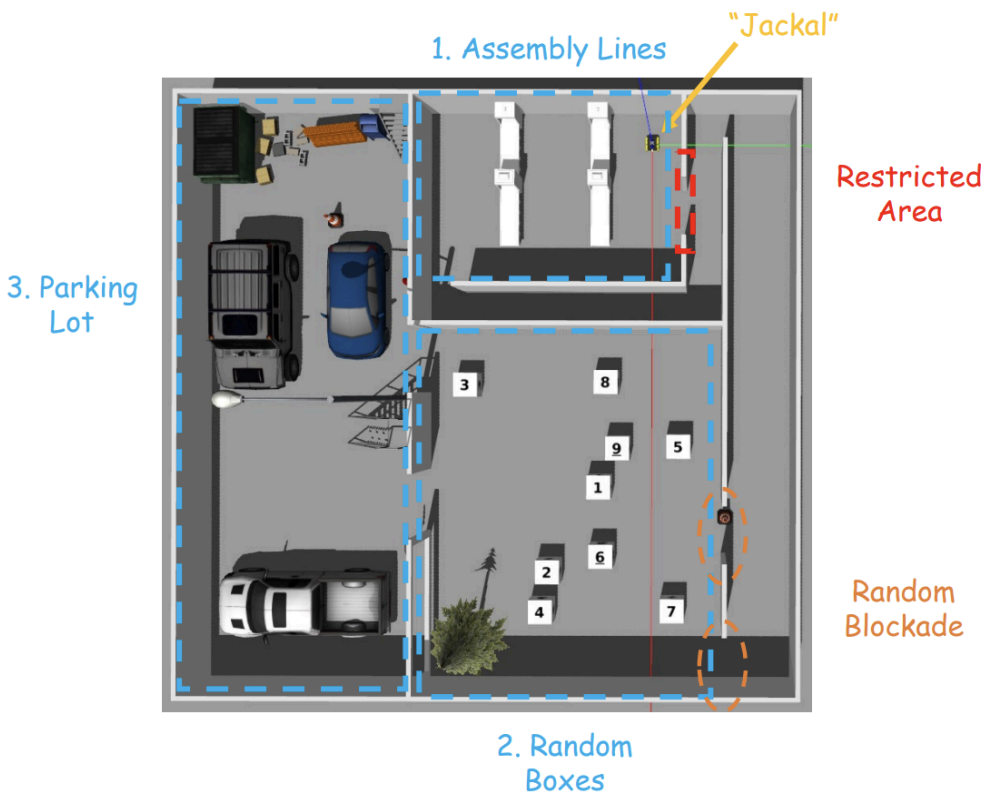


Figure 1: Schematic of mini-factory environment

Project Objective:

Mapping

- Manipulate the jackal robot to map the whole environment

Navigation

- Guide the 'Jackal' robot through a predefined sequence of locations:
  - Assembly Line sections 1 and 2
  - Random Box Locations (Number 3)
  - Delivery Vehicle stations 1 and 2
- Add prohibition zones to restrict the robot's movement.
  - Restricted Area
  - Random Blockade

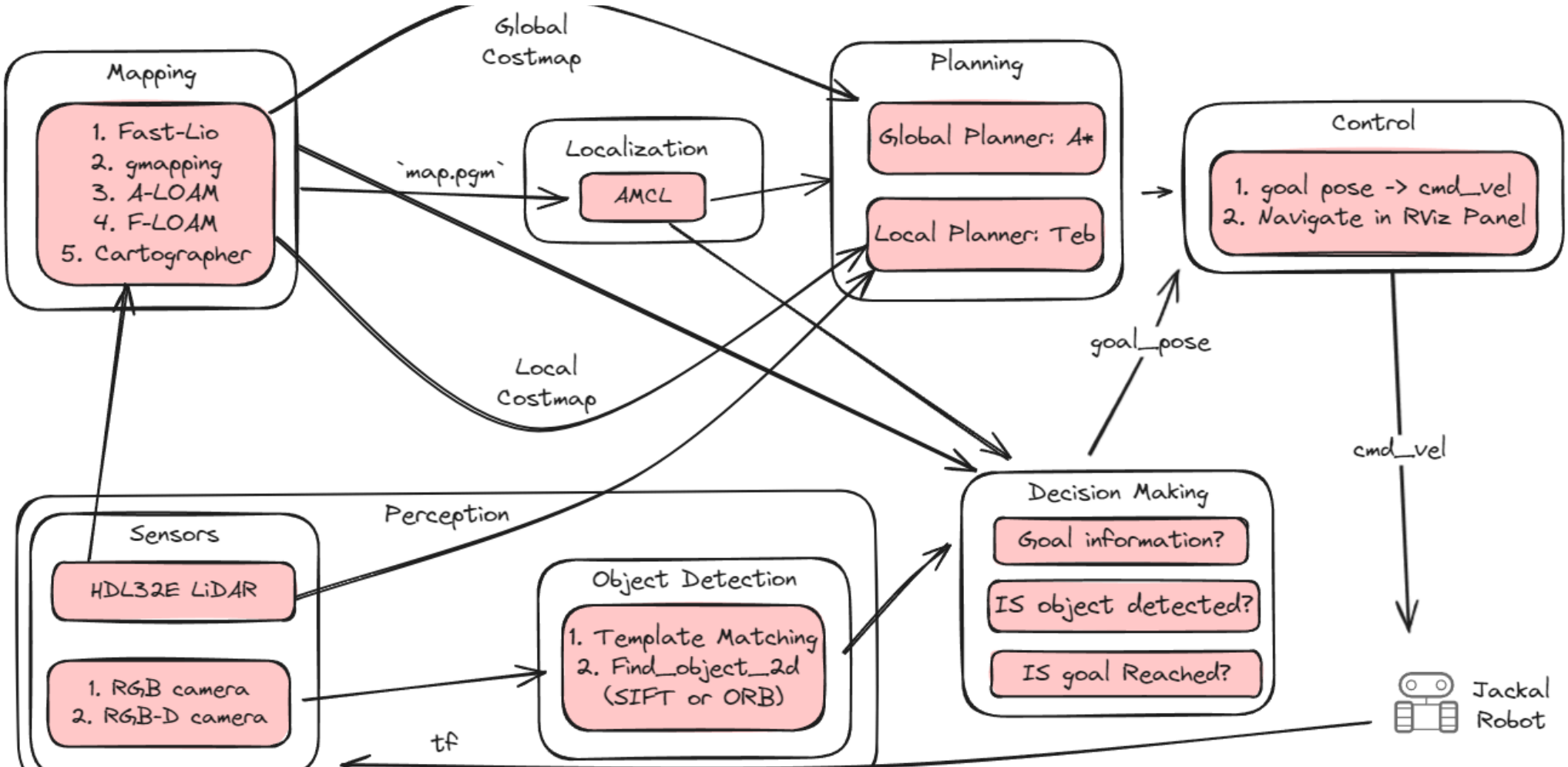


Figure 2: Schematic of mini-factory environment

The project consists of six main components:

- **Mapping:** The primary mapping method used is the `Fast-Lio` package. Other mapping methods, such as `gmapping`, `Cartographer`, `A-LOAM`, and `F-LOAM`, are also tested for performance comparison.
- **Perception (Object Detection):** Two methods are used to detect the object (number 3 on the box) using the camera information:
  - Template matching method provided by the OpenCV library.
  - The `find_object_2d` package, which uses the `ORB` (Oriented FAST and Rotated BRIEF) and `SIFT` methods for object detection.
- **Localization:** Localization is performed using the `AMCL` (Adaptive Monte Carlo Localization) method.
- **Planning:** The global planner used is `A*`, and the local planner used is `Teb` (and `DWA` as a comparison).
- **Decision Making:** The decision-making modules decides how to get to the desired goal location based on the goal information and the detected object information.

02

Task 1: Mapping

Problems in LiDAR-SLAM

- `2D cartographer` fails to detect the hollow section of walls and does not account for the robot's height.
- `3D cartographer` offers a more detailed environmental representation, but still lacks the necessary definition for complex navigation tasks.
- `gmapping` lacks a great capabilities for obstacle recognition, which could increase the risk of collision.

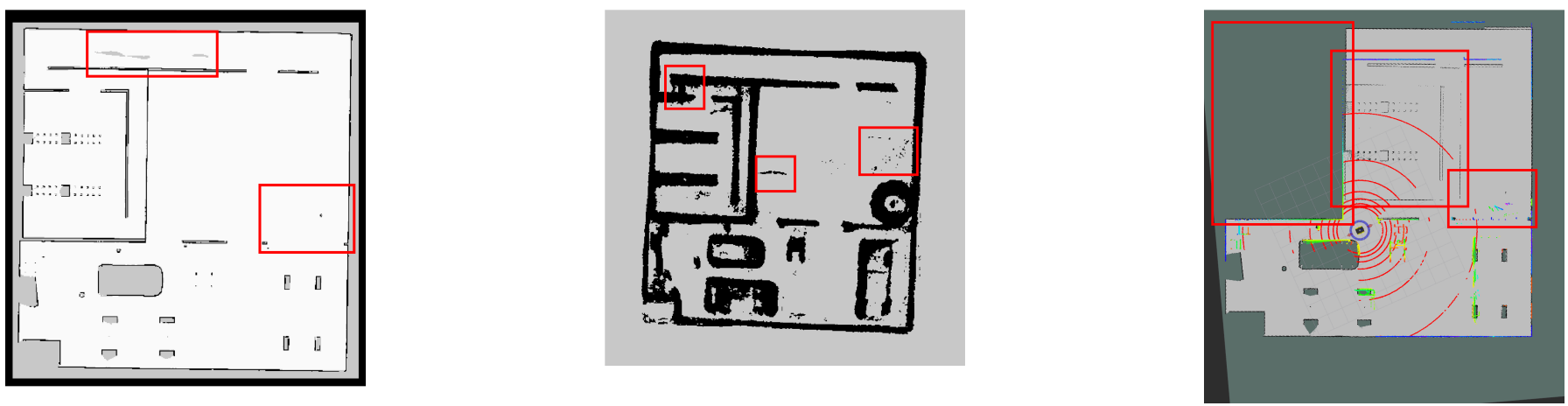


Figure 3: Mapping results by `carto_2d`, `carto_3d`, and `gmapping`

Fast-LIO

FAST-LIO (Fast LiDAR-Inertial Odometry) is a computationally efficient and robust LiDAR-inertial odometry package. It fuses LiDAR feature points with IMU data using a tightly-coupled iterated extended Kalman filter to allow robust navigation in fast-motion, noisy or cluttered environments where degeneration occurs.

The mapping sequence generates a `.pcd` file that details the surveyed environment, which can be converted into a `.pgm` file for navigation and global costmap deployment.

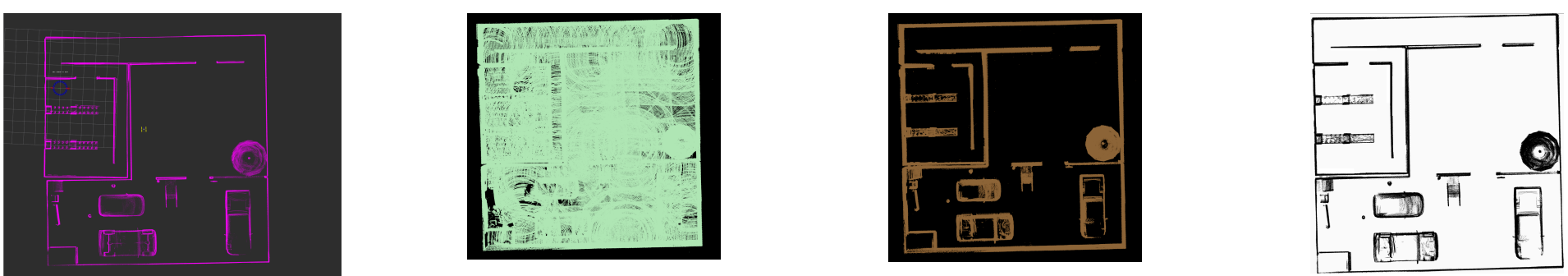


Figure 4: `Fast-lio` Running Visualizatio, `pcd` file, filtered `pcd` file, and generated `pgm` file

A-LOAM

A-LOAM advances the original LOAM system through refined feature extraction and code architecture leveraging Eigen and the Ceres Solver. Topic remappings are essential for its real-time pose estimation and mapping capabilities: `<remap from="/velodyne_points" to="/mid/points"/>`

Fast-LOAM

The Fast LiDAR Odometry and Mapping (FAST-LOAM) algorithm is an efficient LiDAR-based SLAM approach that builds upon LOAM. It extracts and utilizes features from point cloud data for odometry estimation and mapping. The mapping module refines the pose graph and optimizes the map globally. Optimizations in feature extraction and selection enable FAST-LOAM to operate effectively in computationally constrained scenarios. Topic remmapings are shown below:

```
<remap from="/velodyne_points" to="/mid/points"/>
<remap from="/odom" to="/odometry/filtered"/>
<remap from="/cam03/image_raw" to="/front/image_raw"/>
```

Visualization of A-LOAM and F-LOAM

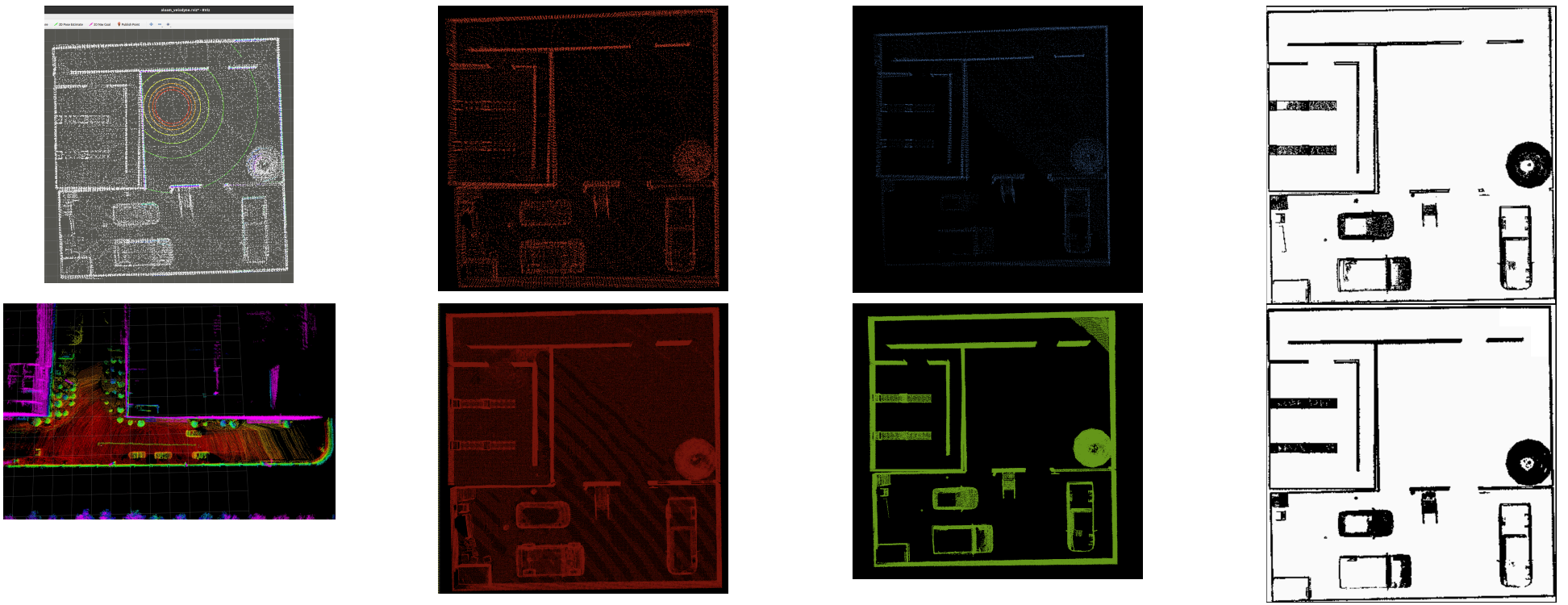


Figure 5: `A-LOAM` and `F-LOAM` Running Visualization, `pcd` file, filtered `pcd` file, and generated `pgm` file

Point Cloud Filtering

```
<param name="file_directory" value= "/home/john/ME5413_Final/src/FAST_LIO/PCD/" />
<param name="file_name" value= "map" />
<!-- minimum height-->
<param name="thre_z_min" value= "-2.0" />
<!-- maximum height-->
<param name="thre_z_max" value= "0.7" />
<!-- 0 select points in height range, 1 select points out of height range -->
<param name="flag_pass_through" value= "1" />
<!-- radius filter's radius-->
<param name="thre_radius" value= "0.3" />
<!-- radius filter's points required-->
<param name="thres_point_count" value= "10" />
<!-- resolution of gripmap-->
<param name="map_resolution" value= "0.05" />
<!-- topic of gripmap, use map_server to save -->
<param name="map_topic_name" value= "map" />
```

Maps Generation and Comparison

- **A-LOAM:** The sparsity is due to the hardware limitations of the Jackal platform's 16-beam LiDAR sensor.
- **F-LOAM:** FAST-LOAM's algorithmic design adeptly identifies and prioritizes geometrically significant data, resulting in an expeditious yet accurate cartographic depiction of the surroundings.
- **Fast-LIO:** Robust data processing capabilities, underpinning its utility for precise localization and comprehensive environmental modeling.

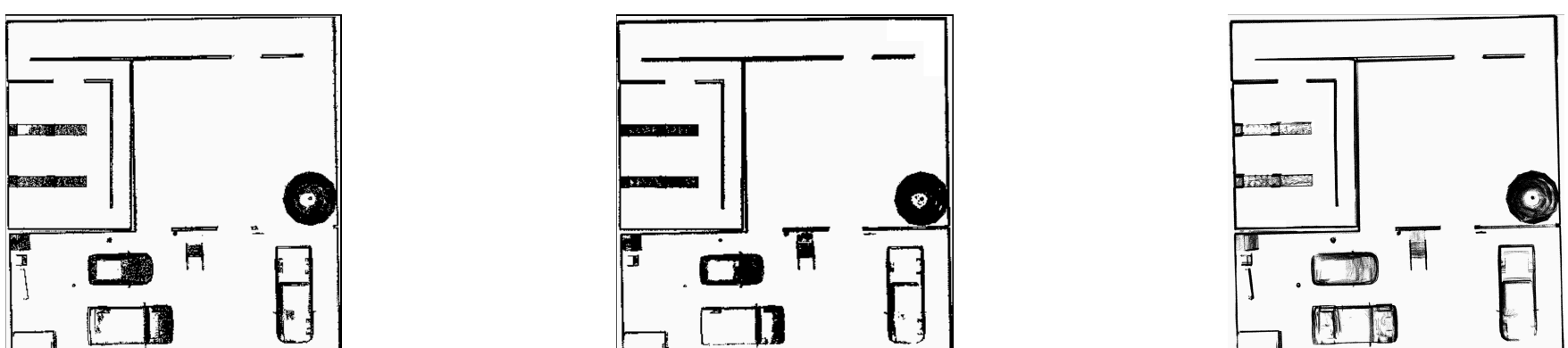


Figure 6: Filtered Point Cloud Map Generated by F-LOAM and Fast-Lio, respectively

Modifying the Original Map

In autonomous navigation, certain zones, such as control rooms, must be restricted from robotic entry. This can be achieved by modifying the `.pgm` file of the original map using image editing software like GIMP.

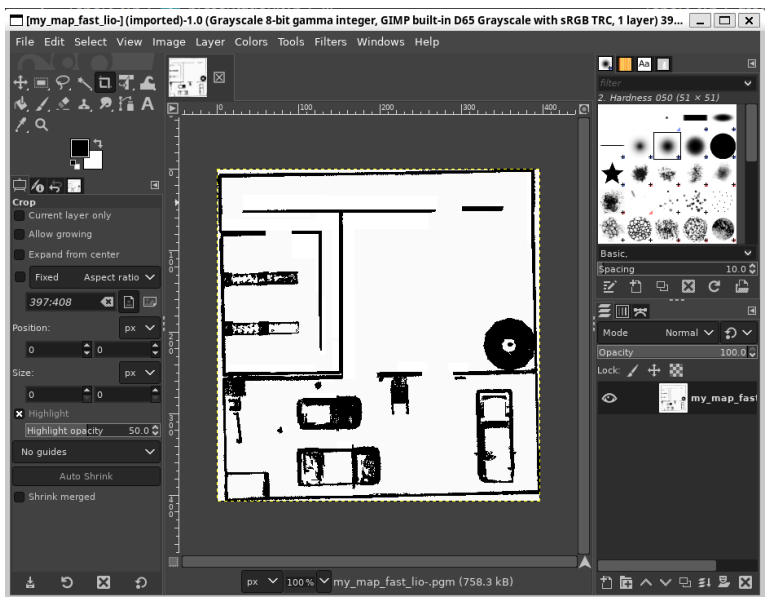


Figure 7: Editing the `.pgm` file using GIMP

Fast-LIO Evaluation

The operational data stream captured during the FAST-LIO algorithm's execution is critical for post-mission performance analysis. A ROS node within the *fast_lio.launch* file is tasked with recording this data into a bag file.

```
<node name="record_bag" pkg="rosbag" type="record" args="-O $(find me5413_world)/
result/output.bag /gazebo/ground_truth/state /Odometry" />
```

This bag file is then used for the Absolute Pose Error (APE) analysis by executing the *evo_ape* command. The command for this evaluation is as follows:

```
evo_ape bag output.bag /gazebo/ground_truth/state /Odometry -r full -va --plot --
plot_mode xy
```

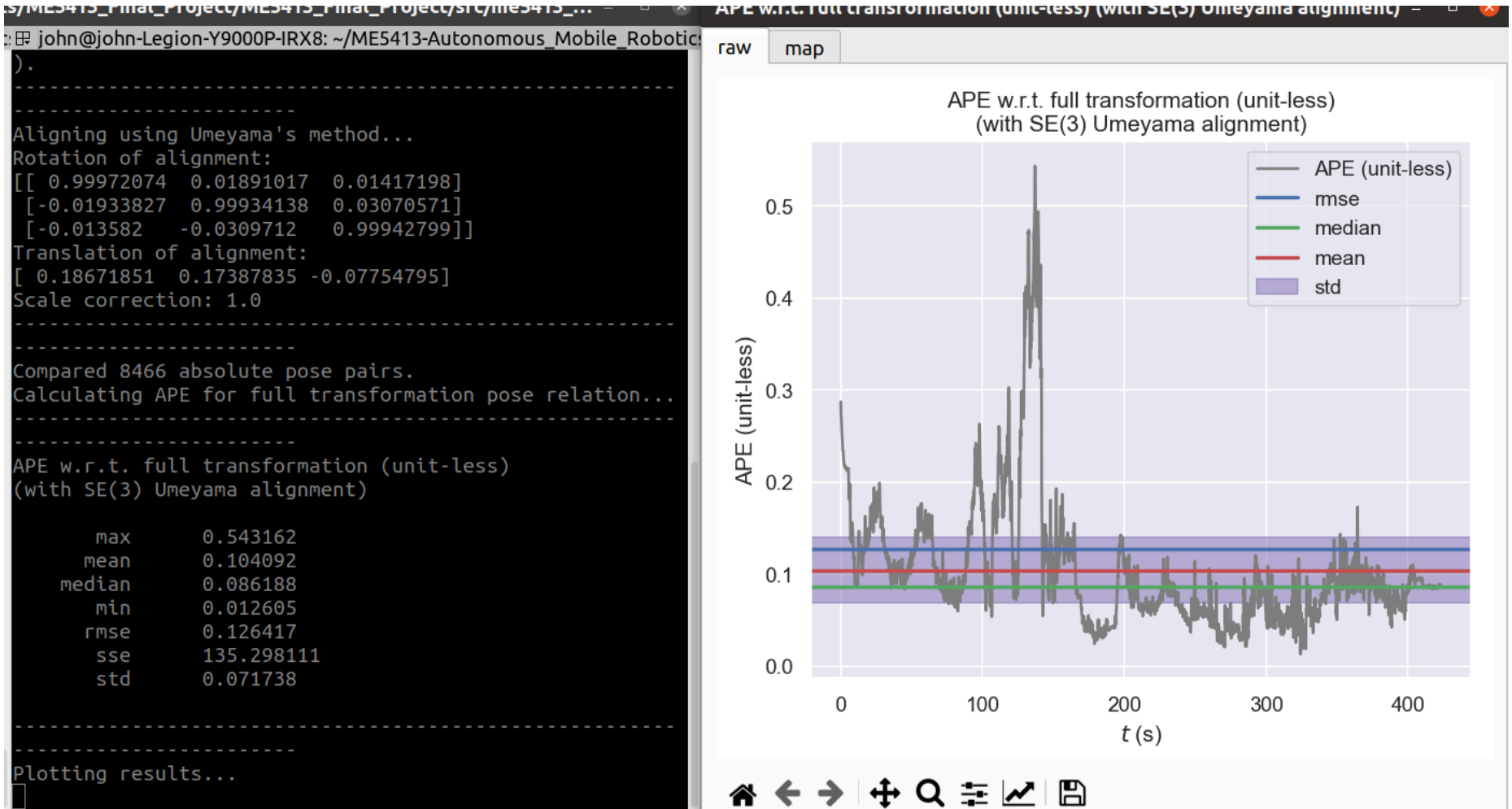


Figure 8: FAST-LIO APE Analysis 1

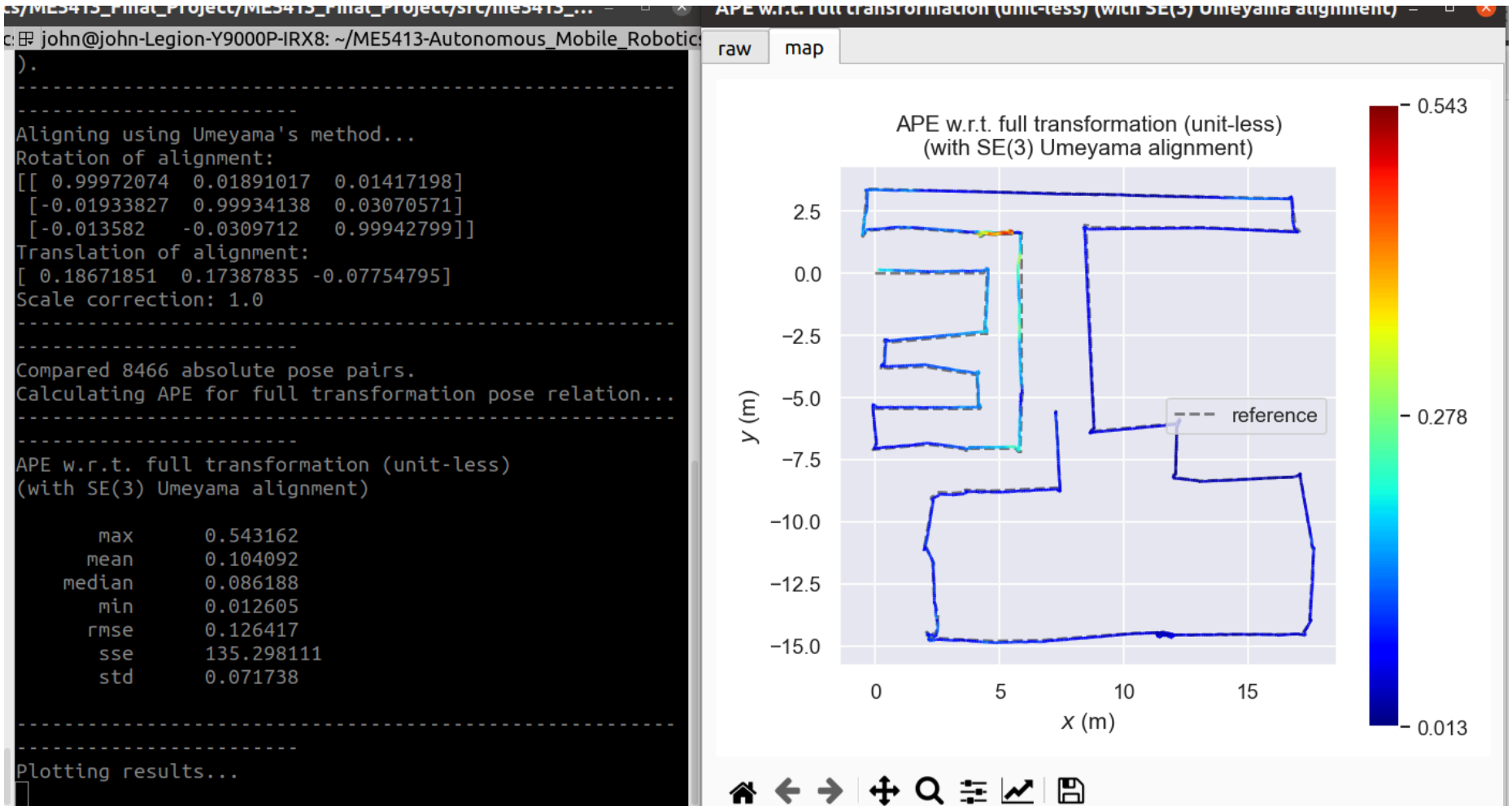


Figure 9: FAST-LIO APE Analysis 2

03

Task2: Navigation

Localization

Localization allows the robot to determine its position and orientation within a given coordinate system, which is essential for effective path planning. The robot's various coordinate frames and their relationships are represented in the TF (transformation) tree.

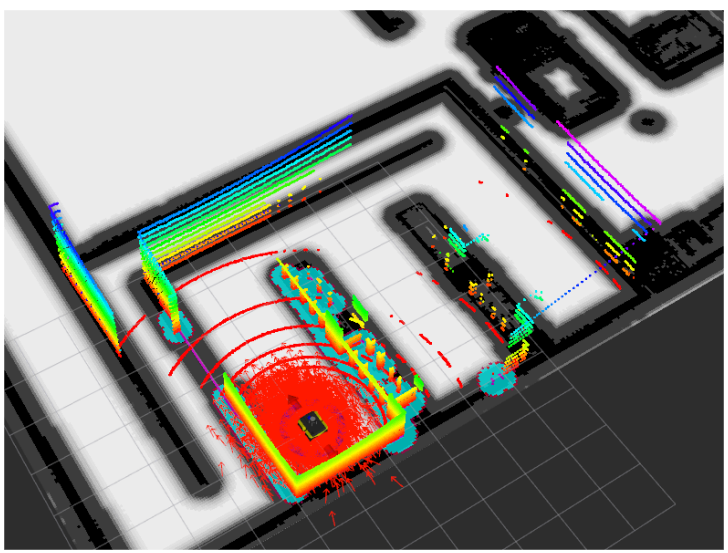


Figure 10: AMCL Localization

AMCL

For this project, we used the Adaptive Monte Carlo Localization (AMCL) algorithm from ROS. AMCL is a particle filter-based localization method that can efficiently estimate the robot's pose even in dynamic environments. Key parameters of the AMCL algorithm were configured in the `amcl.launch` file:

- `odom_model_type="diff"`: Specifies the use of the differential drive model for the robot's motion.
- Min/max number of particles set to 8000/10000 for improved accuracy.
- KLD sampling error (`kld_err`) set to 1.5 to reduce max error between true and estimated distributions.
- KLD confidence (`kld_z`) set to 0.5 for more conservative particle count adjustment.
- Likelihood field sensor model used to handle measurement errors.
- `update_min_d` and `update_min_a` set to trigger localization updates and improve robustness.

Additional parameters for the global and local cost maps were configured in

- `global_costmap_params.yaml`
- `local_costmap_params.yaml`
- `costmap_common_params.yaml`

Other Localization Methods

Besides AMCL, we explored two filter-based localization methods:

- **Extended Kalman Filter (EKF):** A classic nonlinear estimation algorithm based on Kalman filtering, which linearizes the system and measurement models. Although simple, EKF may introduce significant errors in complex nonlinear systems. EKF-based localization was implemented in the `robot_pose_ekf.launch` file.
- **Unscented Kalman Filter (UKF):** A nonlinear filtering algorithm based on the unscented transform, using carefully selected sample points to approximate the probability distribution and avoid linearization errors. UKF better handles nonlinear systems and offers advantages in localization accuracy and stability compared to EKF. The UKF-based method was configured in the `ukf_template.launch` file.

Comparative experiments showed that AMCL provided more stable and accurate localization results for the project's environment and conditions. Therefore, we chose AMCL as the primary localization method and integrated it into the entire navigation system.

Costmap

The robotic system's navigation framework uses a composite costmap divided into global and local representations.

- **The global costmap** represents the environmental model constructed by the Fast-Lio package, with its configuration parameters in the `jackal_navigation/params/costmap_common_params.yaml` file. It provides a high-level overview of the robot's operational terrain.
- **The local costmap** focuses on the robot's immediate surroundings, supporting local pathfinding and collision avoidance. It is continuously updated with real-time sensor data, such as laser scans and point clouds, and has a smaller scale but higher resolution compared to the global costmap.

As can be seen in Figure 11, the global costmap provides a bird's-eye view of the environment, highlighting the various zones and obstacles within the mini-factory. The costmap is color-coded to represent different cost values, with higher costs indicating obstacles or restricted areas that the robot should avoid during navigation.

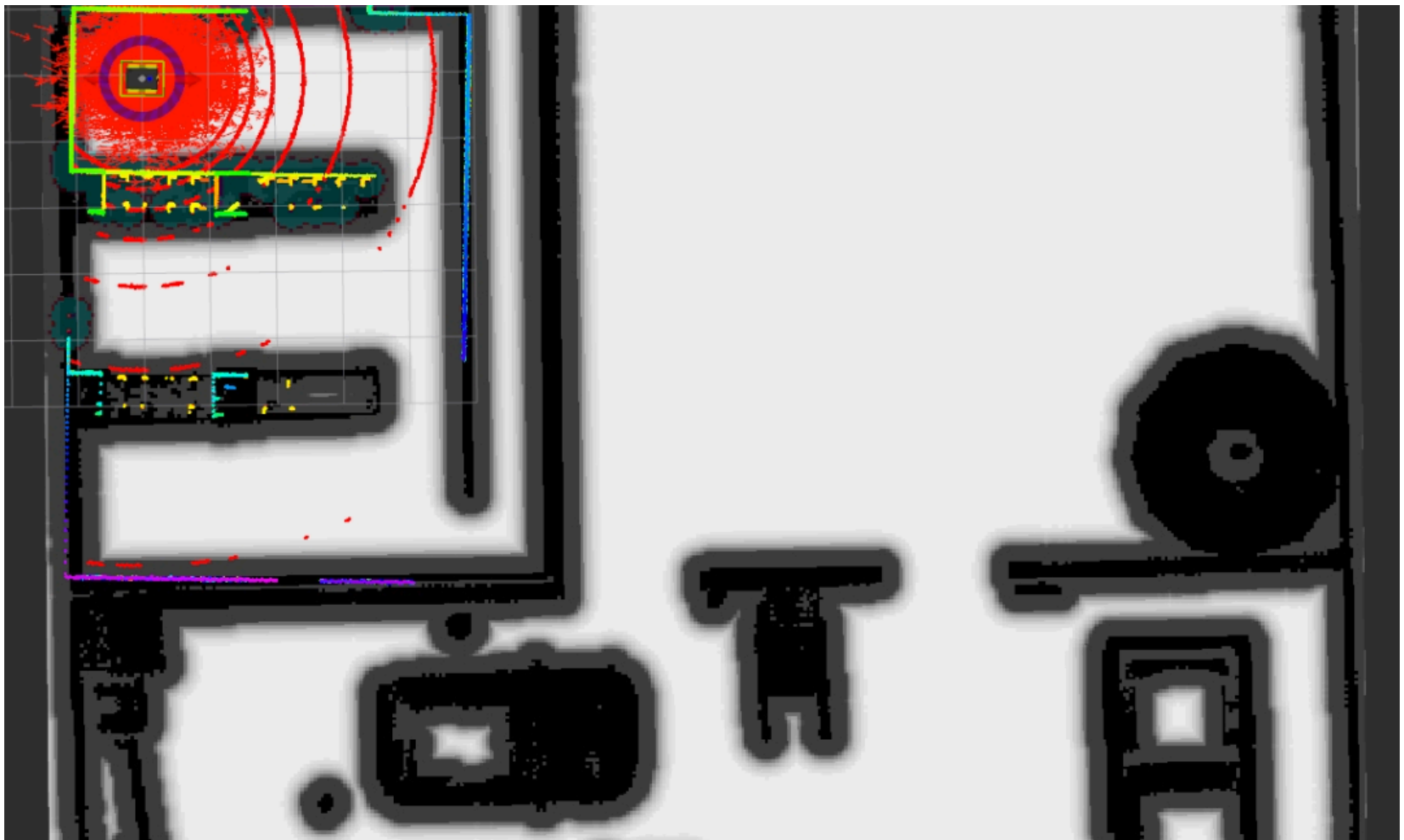


Figure 11: Global Costmap

Prohibition Area

Restricted Area

Prohit the robot from entering the restricted area:

```
prohibition_areas:
- [[4, 1], [1, 1]]
```

Random Blockade

Prohibit the robot from entering the random blockade area.

```
def cone_position_callback(msg):
  x, y = msg.point.x, msg.point.y
  rospy.loginfo("Received cone position: x = %s, y = %s", x, y)
  update_prohibition_area(x, y)
def update_prohibition_area(x, y):
  # Update the prohibition area based on the cone position
  pass
```

Global Planning

The A* algorithm is employed for global planning within the `move_base` package, calculating an optimal path from the robot's starting point to its goal while considering the mapped environment. A* is a search algorithm that combines the advantages of best-first search and Dijkstra's algorithm to efficiently find the shortest path in a known environment, making it particularly useful for robot path planning and navigation.

The global planner's behavior can be configured through the `jackal_navigation/params/global_planner_params.yaml` file, where parameters such as path cost weights, planning resolution, and obstacle handling can be adjusted. Tuning these settings allows the navigation system to be optimized for different scenarios, balancing path efficiency and computational demands.

Global Planning

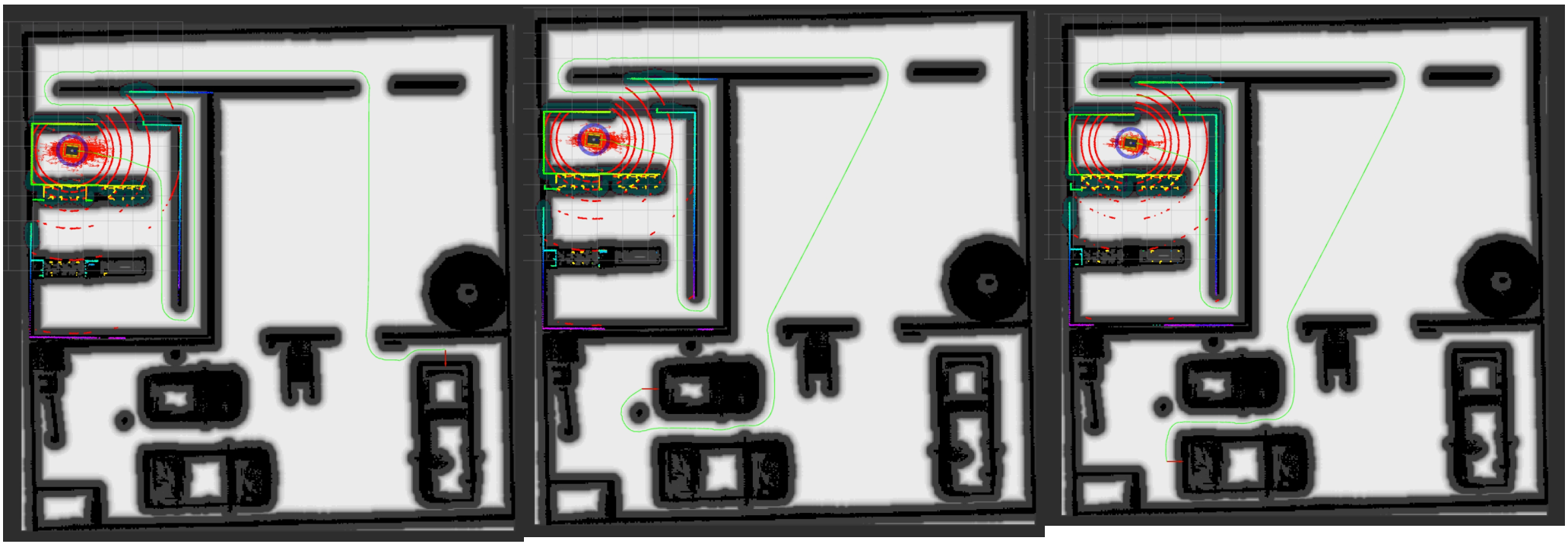


Figure 12: Global Navigation to Vehicle 1, 2, and 3

Local Planning

For local planning, the Time-Elastic Band (Teb) algorithm is employed, focusing on real-time adjustments to navigate around immediate obstacles and dynamic environmental changes. The Teb planner's configuration is managed through the `teb_local_planner_params.yaml` file, allowing for fine-tuning of parameters such as obstacle avoidance behavior, robot velocity limits, and path flexibility. These adjustments enable the local planner to adapt to varying conditions, ensuring smooth and efficient navigation.

Figure 13 demonstrates the improvement in local mapping for navigation. The thick green path line is generated based on the global plan, which would lead to a collision with box 5. The thin green line, generated by local mapping, successfully avoids the crate and prevents the collision.

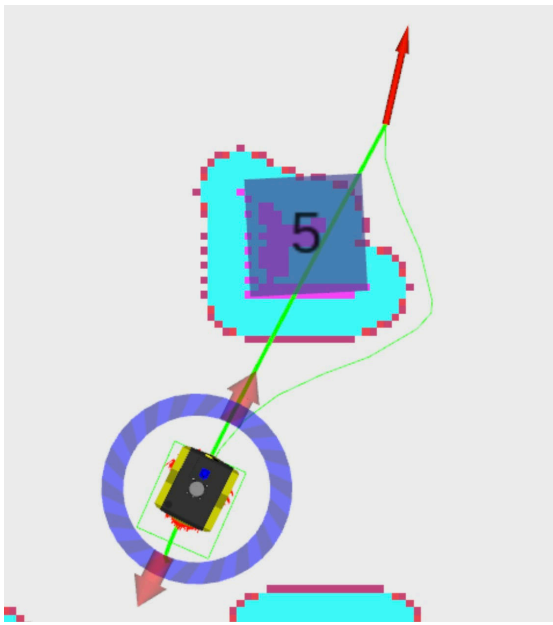


Figure 13: Local mapping for navigation

Local Planning

Several parameters in the Teb planner configuration are noteworthy:

- `min_obstacle_dist`: The minimum distance the robot should maintain from obstacles, crucial for ensuring safety during navigation. Setting this parameter requires balancing safety and path effectiveness. After thorough consideration, we chose 0.25 as its value.
- `inflation_dist`: Defines the area around obstacles deemed hazardous, where cost values are elevated, effectively expanding the obstacle's impact. The size of this buffer considers the robot's dimensions, speed, and environmental complexity. Excessive inflation can restrict mobility in tight spaces, while too little can increase collision risks.
- `dynamic_obstacle_inflation_dist`: Determines the expansion range of dynamic obstacles in the cost map, helping the robot maintain an appropriate safety distance while efficiently navigating in uncertain and dynamic environments.

Local Planning

illustrate the impact of varying the `inflation_radius` parameter on the costmap. This parameter determines the buffer zone around obstacles where the robot is prohibited from entering. A smaller `inflation_radius` results in a more conservative buffer, potentially leading to overly cautious navigation. Conversely, a larger `inflation_radius` can increase the risk of collision by allowing the robot to approach obstacles too closely. The optimal `inflation_radius` value strikes a balance between safety and efficiency, ensuring smooth and obstacle-free navigation.

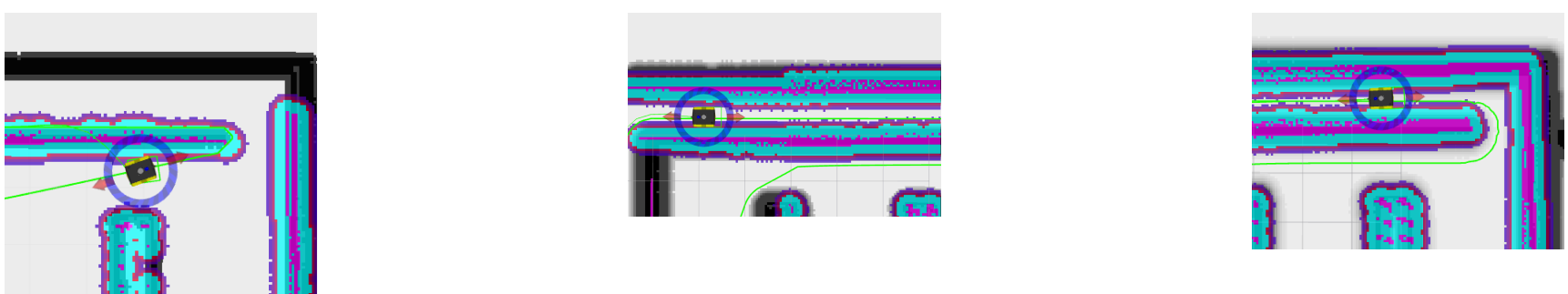


Figure 14: Effect of Inflation Radius (0.1, 0.3, and 0.5)

Comprasion and Discussion

In this section, we adapt the position error, heading error, relative position error, and relative heading error as evaluation indices to compare the performance of the robot navigating to the same location (Assembly Line 2) when applied to two different local planners: TEB and DWA. The results are shown in Figure 15, and a further comparison is presented in Figure 16.

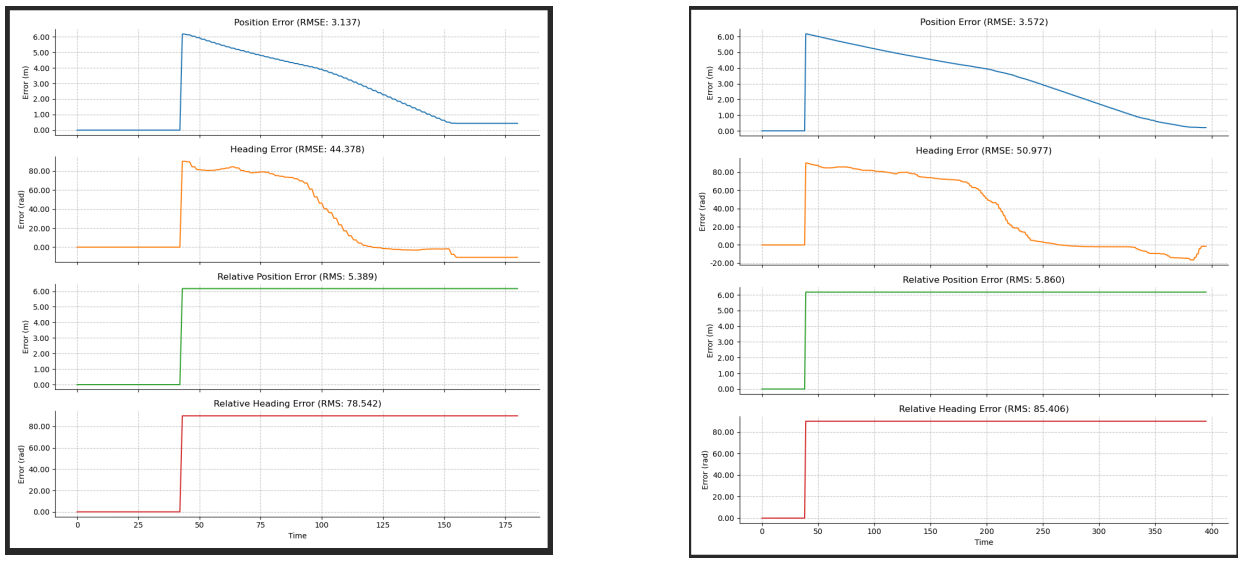


Figure 15: Errors Evaluation of DWA and TEB local Planners

Comprasion and Discussion

Metric	TEB	DWA
RMSE Position	**3.137**	3.572
RMSE Heading	**44.378**	50.977
RMS Relative Position	**5.389**	5.860
RMS Relative Heading	**78.542**	85.406

Figure 16: Comparison of TEB and DWA local planners with A* global planner

Two Methods for Object Detection

1. **Template Matching:** Compares a pre-established template image with the live camera feed to detect the designated object (digit 3 on a box). Implemented in `template_matching_node_py.py` (Python) and `template_matching_node.cpp` (C++).
2. **Feature-based Matching:** Uses the `find_object_2d` package, which employs the SIFT feature detection algorithm for image matching.

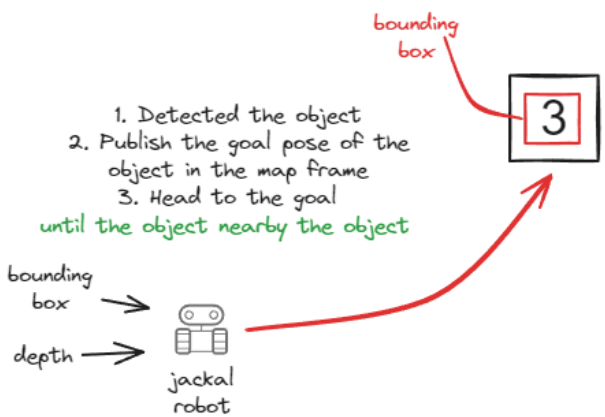


Figure 17: RGB-D Camera-based

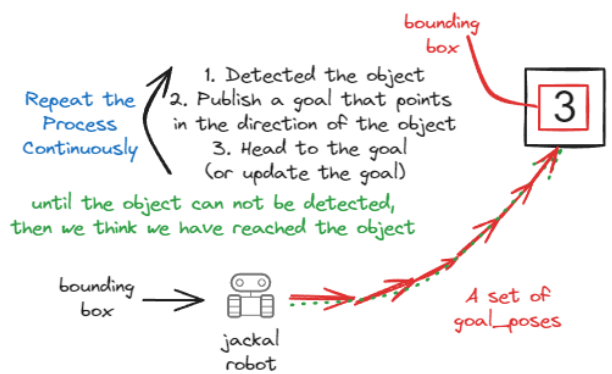


Figure 18: Non-Depth Camera-based

Visual Identification of Box3

We have used 4 templates to identify the box3 object. The templates are shown in Figure 19. The `find_object_3d` package is used to identify the box3 object. The `template_matching` method is also used for comparison.

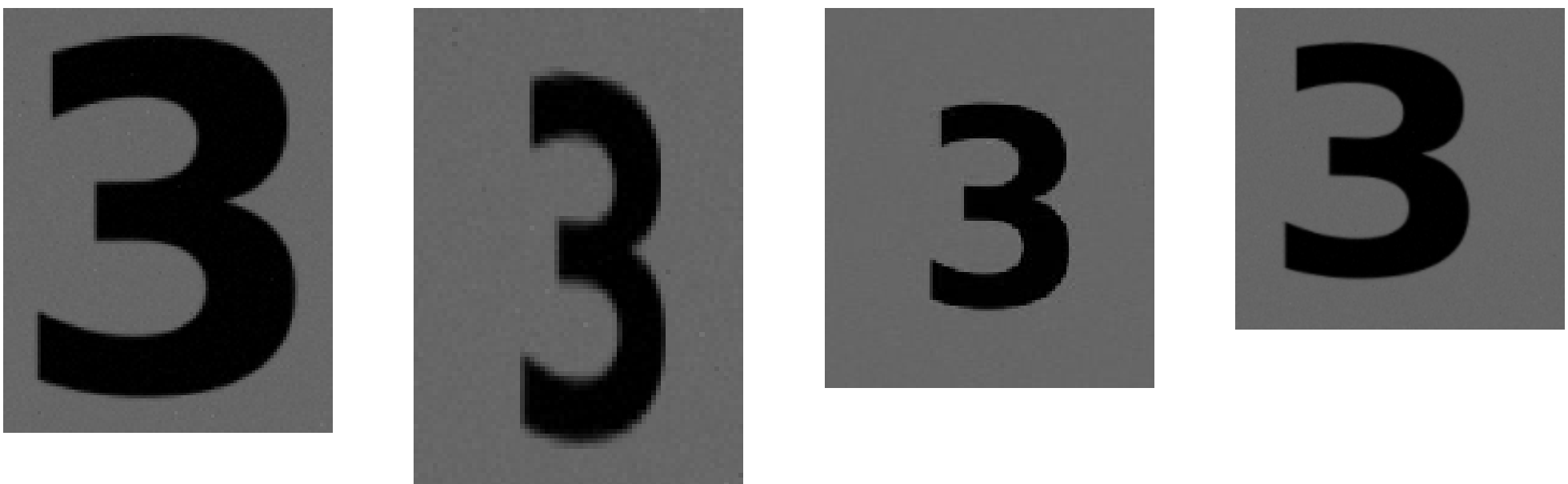


Figure 19: Templates used in Object Detection

Visual Identification of Box3

Figure 20 and Figure 21 show the identification of using `find_object_3d` and `template_matching` methods, respectively.

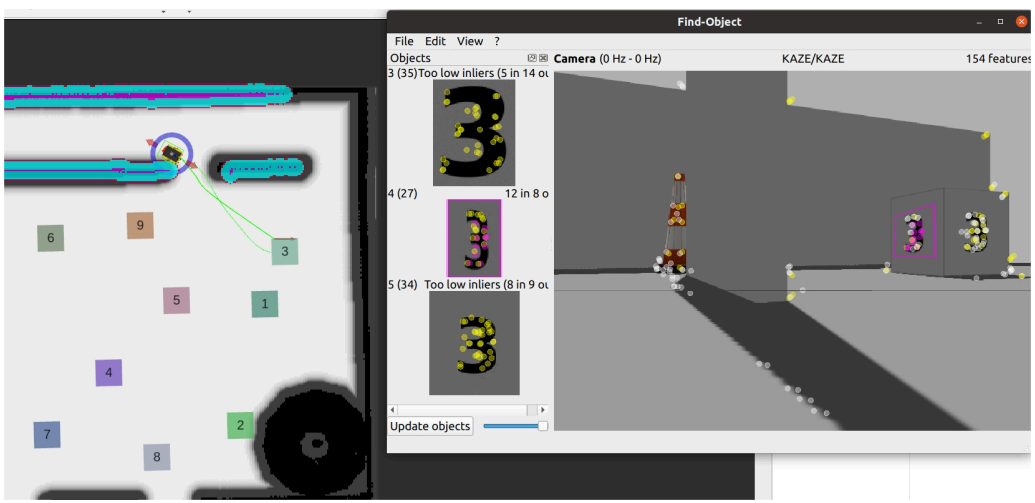


Figure 20: Box3 identification by `find_object_3d`

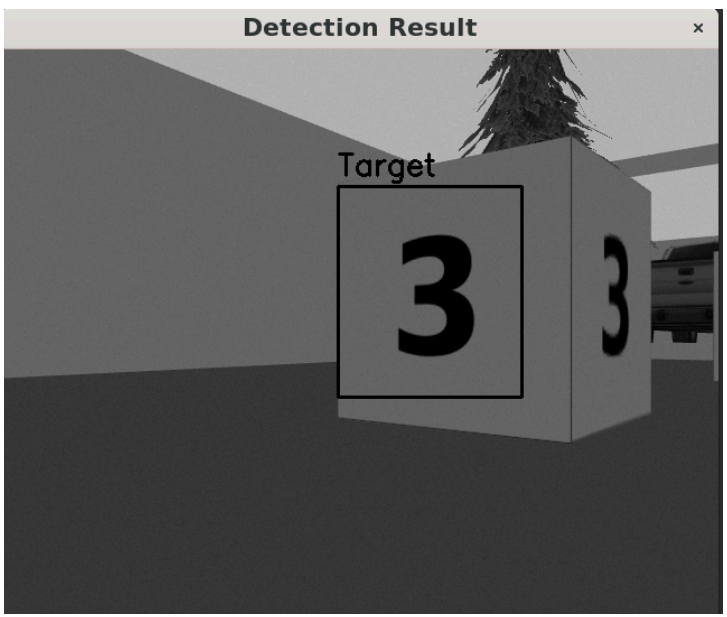


Figure 21: Box3 identification by `template matching`

Camera Calibration Policy

Firstly, the pixel coordinates (u, v) of the target within the image frame, corresponding to the center of the bounding box, are determined by:

$$u = \text{bbox.x} + \frac{\text{bbox.width}}{2}$$

$$v = \text{bbox.y} + \frac{\text{bbox.height}}{2}$$

Subsequently, the depth value Z at these coordinates is retrieved from the depth image. Thereafter, the camera intrinsic matrix K is utilized to transform the image coordinates and depth value into a three-dimensional point (X, Y, Z) in the camera coordinate system. The matrix K is expressed as:

$$K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

Camera Calibration Method

$$X = \frac{(u - c_x) \cdot Z}{f_x}$$

$$Y = \frac{(v - c_y) \cdot Z}{f_y}$$

$$Z = \text{depth}$$

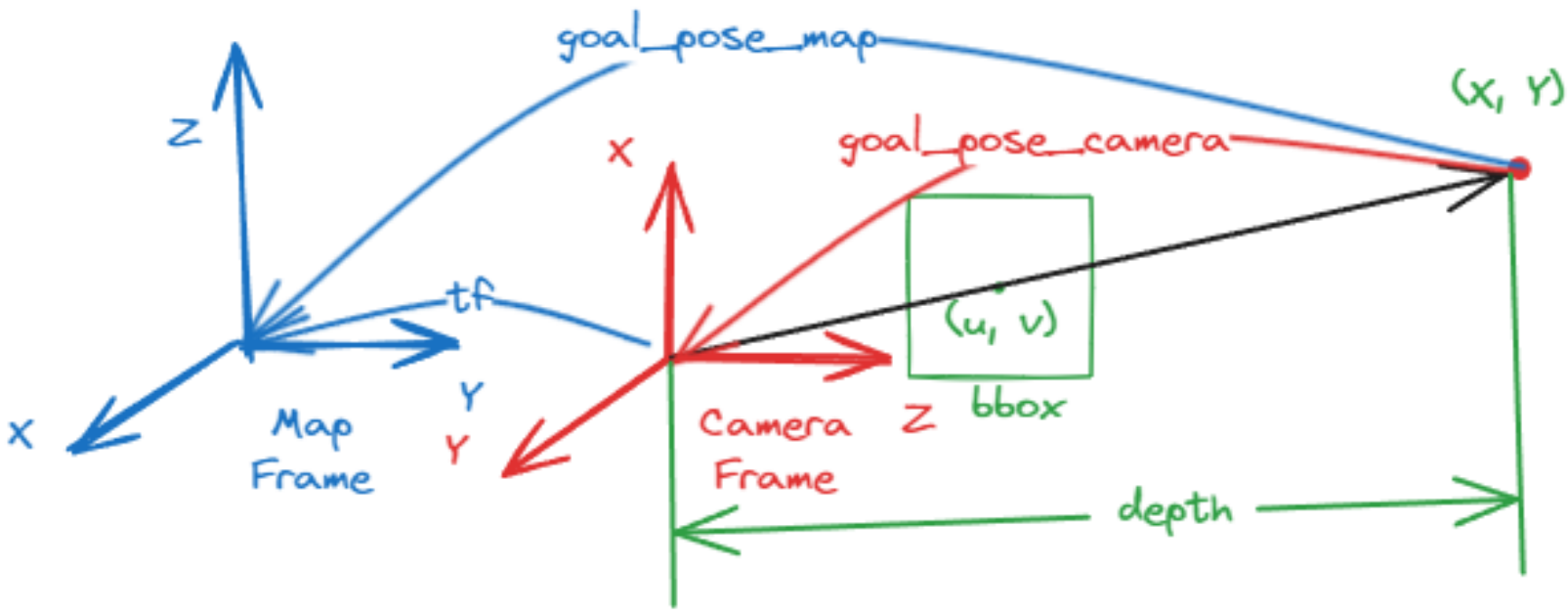


Figure 22: Camera Calibration Method

Decision Making Flowchart

The decision making logic is shown in the flowchart below:

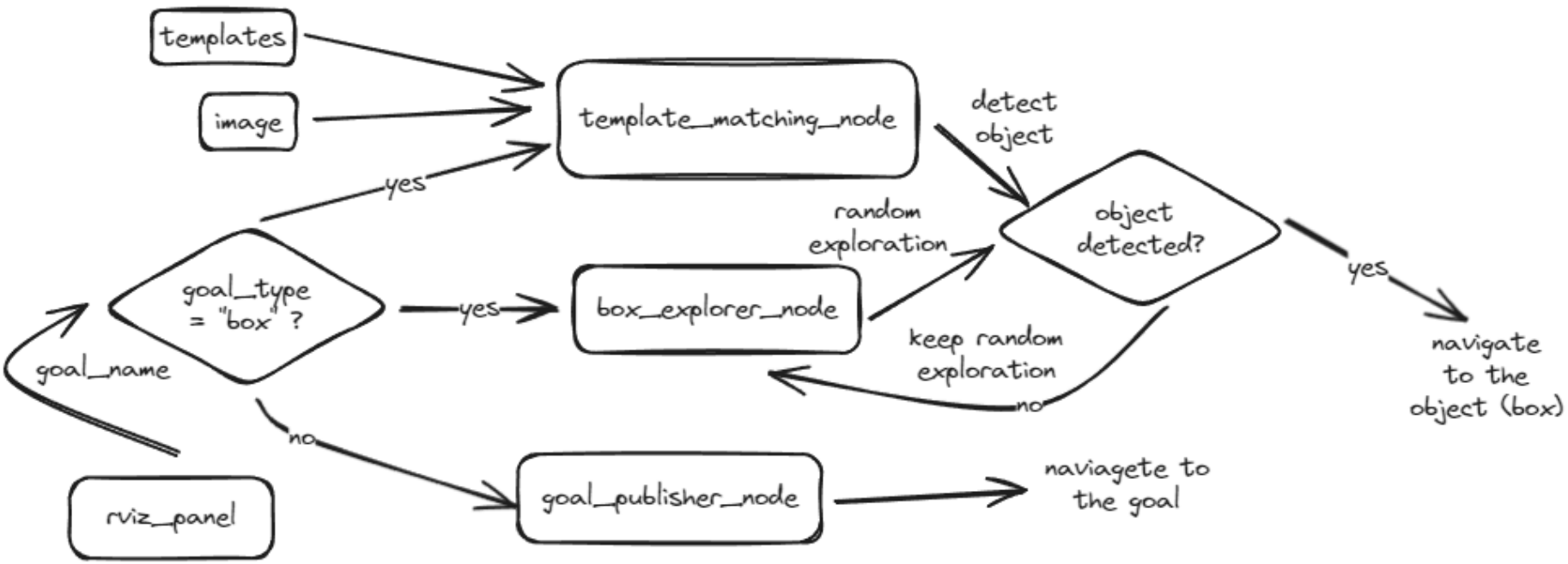


Figure 23: Flowchart of decision making

Decision Making Flowchart

We have mainly used three nodes to implement the decision (i.e. How to get to the desired goal) logic:

- `goal_publisher_node`: Publish the goal location (when the goal is not a box). The robot will navigate to the goal location directly.
- `box_explorer_node`: Explore the box location (when the goal is a box). The robot will navigate to the boxes area (where the boxes are spawned) and explore the box location randomly until the box is detected.
- `template_matching_node`: Continuously detect the object (number 3 on the box) using the template matching method. If the object is detected, the robot will navigate to the object location.

Random Exploration Policy

The random exploration policy is implemented in the `box_explorer_node.cpp` file. A basic process is shown in the figure below:

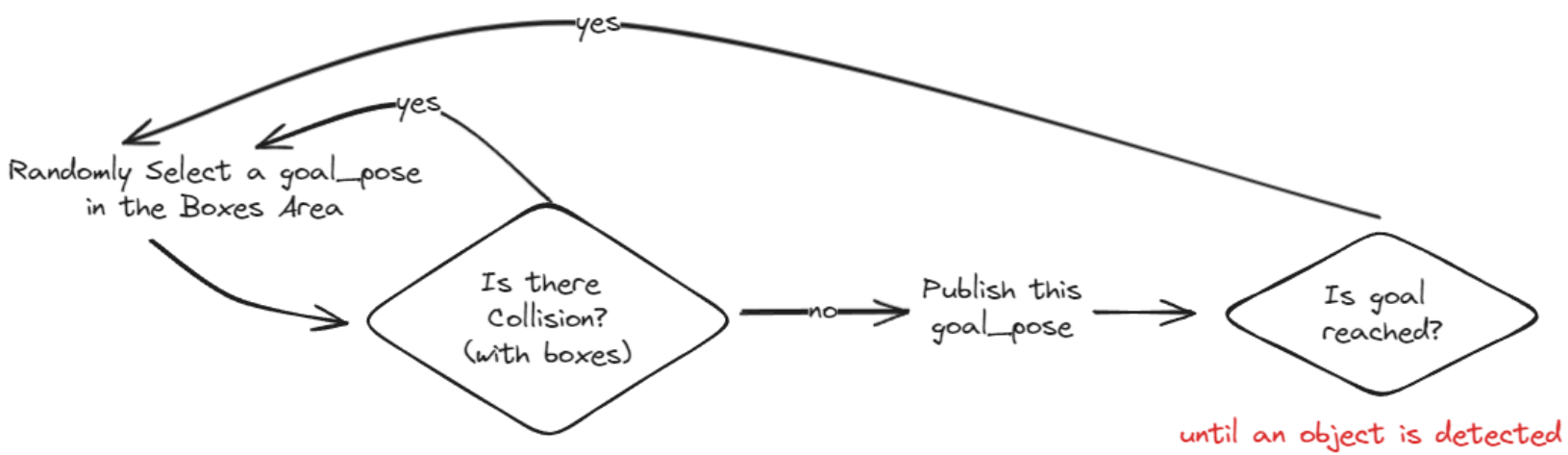
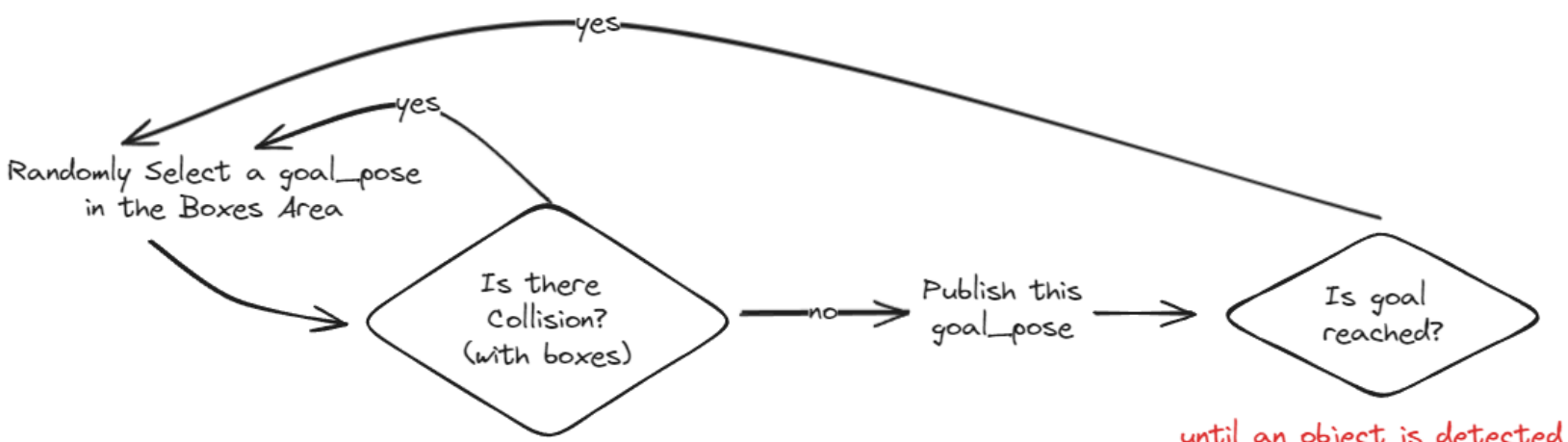


Figure 24: Flowchart of decision making

Random Exploration Policy

First, we need to randomly select a `goal_pose` in the boxes are, which is implemented in the `createWaypoints()` and `updateCurrentWaypoint()`. Then we need to check where the selected `goal_pose` has collides with the obstacles, which is implemented in the `isPointInObstacle` function. If the `goal_pose` is not in the obstacles, we can navigate to the `goal_pose` directly, if not we need to select another `goal_pose`. The robot will keep exploring the box location until the object is detected.



Collision Checking Policy

We implemented the function by subscribing the `box_markers` topic and decided whether the random selected `goal_pose` is near these `box_poses` location. The function is implemented in the `isPointInObstacle()` function (in `box_explorer_node.cpp`).

```
bool BoxExplorerNode::isPointInObstacle(const geometry_msgs::PoseStamped& point) {
  for (const auto& box_pose: box_poses_) {
    double dx = point.pose.position.x - box_pose.pose.position.x;
    double dy = point.pose.position.y - box_pose.pose.position.y;
    double distance = std::sqrt(dx*dx + dy*dy);
    if (distance < 1) {
      ROS_INFO("Point is in obstacle...");
      return true;
    }
  }
  return false;
}
```

04

Future Work

Future Work

Mapping

- [✓] Implement the `Fast-Lio`, `gmapping`, `Cartographer`, `A-LOAM`, and `F-LOAM` mapping methods.
- [✓] Save the best map as a `.pgm` file. (mapped by `Fast-Lio`)
- [ ] Record the mapping process to a ros bag file.
- [ ] Deploy or design a randomly exploration policy to autonomously map the environment.

Object Detection

- [✓] Implement the template matching method for object detection.
- [✓] Implement the `ORB` or `SIFT` method for object detection. (`find_object_2d`)
- [ ] Train a YOLO detctor for object detection. (may be trained from a small `mnist` dataset)
- [✓] Attempt to use the RGB-D camera to detect the object.
- [ ] Attempt to use the stereo camera to detect the object.
- [ ] Capture the `cone` in the gazebo world from the camera.

Localization

- [✓] Implement the `AMCL` method for localization.
- [✓] Tune the parameters of the `AMCL` package to get better localization results.
- [ ] Solve the drifting problem in localization.

Planning

- [✓] Implement the `A*` global planner.
- [✓] Implement the `Teb` local planner.
  - [✓] Tune the parameters of the `Teb` local planner to get better navigation results.
  - [✓] Avoid the dynamic object when navigating to the goal location. (boxes and cones)
- [✓] Successfully navigate the robot to the goal location.
- [ ] Change the global planner to other planners (`GBFS` or `RRT*`) and compare the performance.
- [✓] Change the local planner to other planners (`DWA` or `EBand`) and compare the performance.

Thanks For All