Autonomous Mobile Robotics Final Project Group 10

Cao Chenyu
Mechanical Engineering
National University of Singapore
Singapore, Singapore
haug@typst.app

Laurenz Mädje
Co-Founder
Typst GmbH
Berlin, Germany
maedje@typst.app

***Abstract*— This project focuses on the development and implementation of an autonomous navigation system for a Jackal robot in a simulated mini-factory environment using the Gazebo framework. The primary objective is to guide the robot through a predefined sequence of locations while mapping the environment, localizing the robot, and detecting and navigating to a specific object (a box with the number 3). The system integrates state-of-the-art algorithms such as FAST-LIO for mapping, Adaptive Monte Carlo Localization (AMCL) for localization, A* for global path planning, and the Time Elastic Band (Teb) approach for local planning. Additionally, the project incorporates object detection using template matching and the find_object_2d package, as well as a random exploration policy and collision checking for efficient decision-making. The performance of the navigation system is evaluated in terms of its ability to accurately map the environment, localize the robot, and navigate to the target locations while avoiding obstacles. The project's outcomes demonstrate the effectiveness of the selected algorithms and provide insights into the challenges and potential improvements in autonomous robot navigation for industrial applications.**

***Index terms*—Scientific writing, Typesetting, Document creation, Syntax**

A. Project Description

In the rapidly evolving field of mobile robotics, which is crucial for technological advancement, this project focuses on a simulated mini-factory environment meticulously designed in the Gazebo framework. The setup is divided into three target zones and one restricted area, each designed to evaluate the effectiveness of various mapping and navigational algorithms.

The primary challenge of this project revolves around the development and implementation of an advanced robot navigation software stack. The goal is to guide our designated Jackal robot through a predetermined sequence of locations within the simulation. Starting from the Assembly Line sections labeled 1 and 2, the journey continues through Random Box zones 1 to 4, and concludes at Delivery Vehicle stations 1, 2, and 3, as illustrated in Figure 1. This sequence serves to assess the robot's navigational proficiency and to examine the robustness and adaptability of the underlying algorithms in the dynamic conditions of the simulated mini-factory.

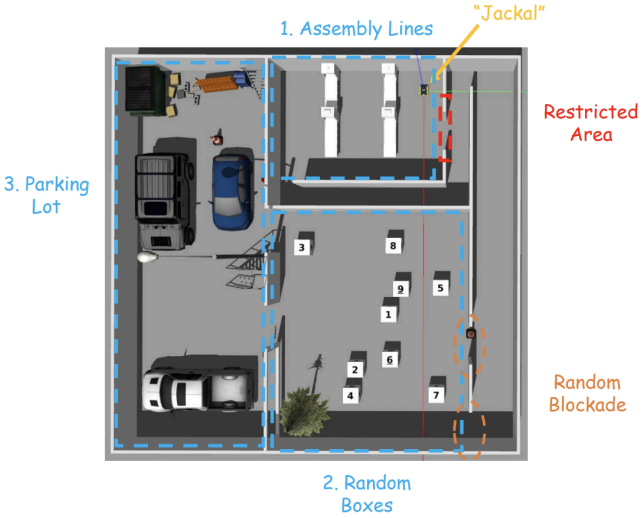

Figure 1: Schematic of mini-factory environment

B. Task 1: Mapping

1) *Problems in LiDAR SLAM Algorithms:*

Figure 2 (a) demonstrates that the 2D Cartographer algorithm fails to detect the hollow sections of walls and does not account for the robot's height, potentially leading to path planning issues. Figure 2 (c) highlights the limitations of the gmapping algorithm, where its scanning capabilities are limited in scope, and its obstacle recognition is poor. This can be problematic when considering elements like glass walls, which are only outlined but not fully captured, and stairs, which are only partially mapped, increasing the risk of collisions.

Figure 2 (b) displays the pixel clustering issue inherent in the 3D Cartographer algorithm, which, despite offering a more detailed environmental representation, still lacks the necessary definition for complex navigation tasks. These observations motivate the exploration of advanced 3D SLAM techniques, specifically FAST-LIO, FAST-LOAM, and A-LOAM, which are designed to overcome the dimensional limitations

of 2D SLAM by providing detailed environmental mapping and improved obstacle recognition.

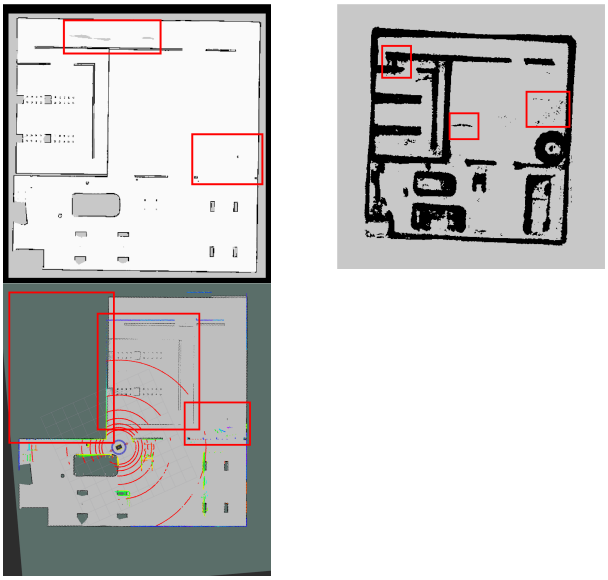

Figure 2: Mapping results by carto_2d, carto_3d, and gmapping

2) *Algorithms Introduction:*

FAST-LIO:

The mapping pipeline in this project is built upon the Fast-Lio package, which is selected for the primary mapping phase. FAST-LIO effectively mitigates drift by fusing dynamic motion detection with static environmental scanning to refine pose accuracies through iterative state estimation processes that resemble Kalman Filter methodologies. The mapping sequence generates a .pcd file that details the surveyed environment, which is then converted into a .pgm format for navigation and global costmap deployment.

The 16-beam LiDAR system for the robotic platform is initialized through the launch file specified in `src/me5413_world/launch/fast_lio.launch`, with the relevant line as follows:

```
<rosparam command="load" file="$(find fast_lio)/
config/velodyne.yaml" />
```

The `velodyne.yaml` file contains the configuration for the FAST-LIO package's `laserMapping` node. It specifies the subscribed topics for LiDAR (`/mid/points`) and IMU (`/imu/data`) data, and the node is configured to publish the resulting aligned point cloud to the `/cloud_registered` topic. These settings define the data flow and processing pipeline for the SLAM system.

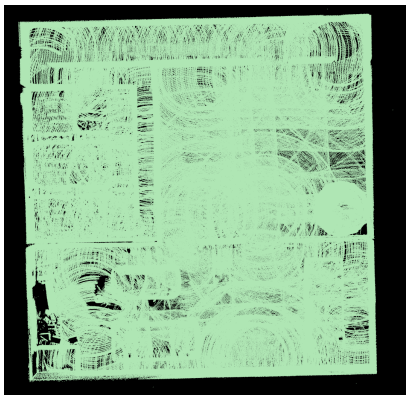

Figure 3: Unfiltered Point Cloud by FAST-LIO

Figure 3 depicts a point cloud generated by the FAST-LIO algorithm, illustrating the sophisticated spatial mapping achieved through LiDAR and IMU data fusion. The point cloud represents a three-dimensional structure of the scanned environment, with each point's position corresponding to a coordinate system anchored to the sensor's location.

The varying densities of points within the image indicate the algorithm's capability to discern between different surfaces and objects. Areas of higher point density suggest closer proximity or more reflective surfaces, while sparser areas indicate distant or less reflective materials. The nuanced differences in the point cloud formations demonstrate FAST-LIO's precision in capturing the environment's geometry.

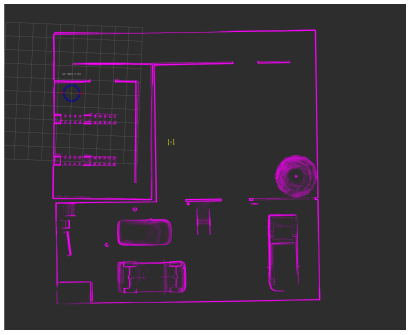

Figure 4: FAST-LIO Running Visualization

Figure 4 showcases the advanced visualization capabilities of the FAST-LIO algorithm, where the multi-colored point cloud exhibits the depth and detail achievable through the fusion of LiDAR and IMU data. This image serves as a visual testament to the algorithm's high-resolution spatial mapping and its ability to discern subtle variances in an environment's topography and texture.

FAST-LOAM: The Fast LiDAR Odometry and Mapping (FAST-LOAM) algorithm is an enhanced LiDAR-based SLAM approach that builds upon the foundations of LOAM (LiDAR Odometry and Mapping). FAST-LOAM's core principle lies in its efficient extraction and utilization of features from the point cloud data, which are pivotal for both odometry estimation and map construction.

The mapping module in FAST-LOAM refines the pose graph to account for drift and optimizes the map globally, ensuring a high-fidelity representation of the environment. The "fast" aspect of FAST-LOAM is achieved through optimizations in feature extraction and selection, enabling it to operate effectively even in computationally constrained scenarios, making it suitable for robotic applications where efficiency and speed are essential.

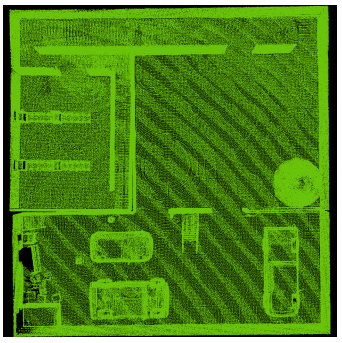

Figure 5: Unfiltered Point Cloud by FAST-LOAM

Figure 5 depicts the environment's point cloud generated by FAST-LOAM, elucidating its efficacy in navigating the specified areas within the mini-factory. By synthesizing a coherent navigational map from raw sensor inputs, FAST-LOAM supports the Jackal robot's mandate to navigate efficiently and accurately, as mandated by the project's goals.

A-LOAM: Within the A-LOAM framework, which advances the original LiDAR Odometry and Mapping (LOAM) system through refined feature extraction and code architecture leveraging Eigen and the Ceres Solver, specific topic remappings are essential for its operation. These remappings facilitate the algorithm's real-time pose estimation and environmental mapping capabilities:

```
<remap from="scan" to="/front/scan"/>
<remap from="odom" to="/odometry/filtered"/>
<remap from="imu" to="/imu/data"/>
<remap from="fix" to="/navsat/fix"/>
<remap from="/velodyne_points" to="/mid/points"/>
```

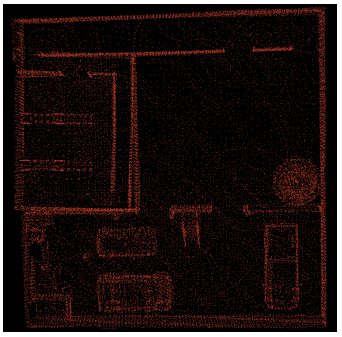

Figure 6: Unfiltered Point Cloud by A-LOAM

Figure 6 shows A-LOAM's output, a refined version of LOAM, featuring a selectively sparse point cloud.

3) *Map Generation:*

Point Cloud Filtering:

Through the execution of three algorithms, we have generated three raw point cloud maps. To extract more meaningful regions from the raw point cloud, we need to apply filtering.

We developed a *pcd_to_map.launch* file to implement this functionality.

The dimensional filtering was performed by setting *thre_radius = 0.3* and *thres_point_count = 10*. This ensures that only regions with a minimum of 10 points within a 0.3 meter radius are considered for the final map. The height filtering was implemented by setting *thre_z_min = −2.0* and *thre_z_max = 0.7* as shown in Figure 7.

pcd_to_map.launch

```
<launch>
    <node pkg="pcd_to_map" name="pcd_to_map" type="pcd_to_map" output="screen">
    <!-- filepath to save .pcd file -->
    <param name="file_directory" value= "$(find fast_lio)/PCD/" />
    <!-- .pcd file name-->
    <param name="file_name" value= "map" />
    <!-- minimum height-->
    <param name="thre_z_min" value= "-2.0" />
    <!-- maximum height-->
    <param name="thre_z_max" value= "0.3" />
    <!--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" />
    </node>
</launch>
```

Figure 7: Pcd TO Map Launch File

Additionally, the *flag_pass_through* was set to 1, which selects the points outside of the specified height range. This helps to remove unwanted points, such as those belonging to buildings or other tall structures, and focus on the desired regions of interest.

Finally, the *map_resolution* was set to 0.05 to generate the occupancy grid map at a resolution of 5 centimeters per cell. This level of detail is essential for capturing the fine-grained features of the environment and providing a high-quality representation for the subsequent algorithm evaluation and comparison.

Maps Generated and Comparison:

The raw point cloud maps and the filtered point cloud maps generated by the three algorithms are shown below.

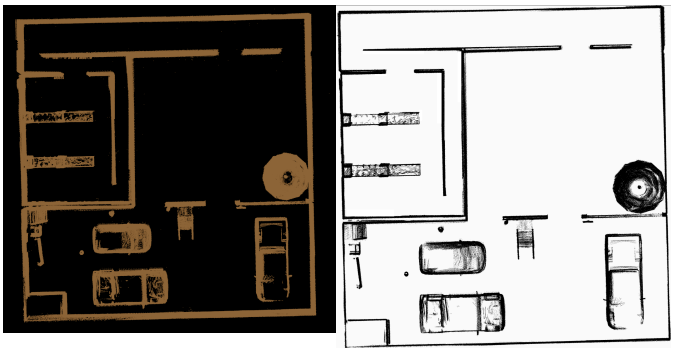

Figure 8: Filtered Point Cloud by FAST-LIO and Map Generated by FAST-LIO

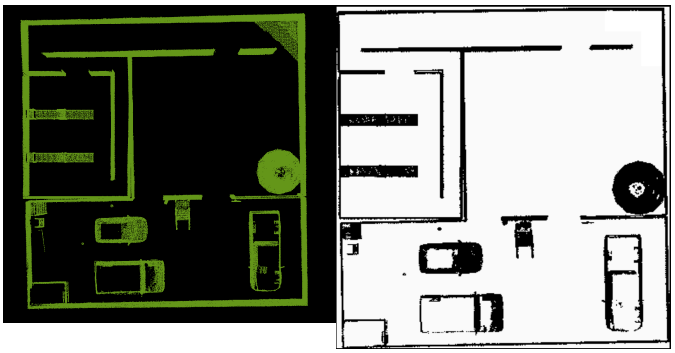

Figure 9: Filtered Point Cloud by FAST-LOAM and Map Generated by FAST-LOAM

Figure 8 exemplifies the efficacy of the FAST-LIO algorithm, illustrating its prowess in refining the raw point cloud to eliminate noise while retaining critical structural details within the mini-factory environment. The resultant high-resolution map elucidates FAST-LIO's robust data processing capabilities, underpinning its utility for precise localization and comprehensive environmental modeling.

Figure 9 showcases the output of FAST-LOAM, where the filtration process meticulously highlights salient environmental features from the point cloud. FAST-LOAM's algorithmic design adeptly identifies and prioritizes geometrically significant data, resulting in an expeditious yet accurate cartographic depiction of the surroundings.

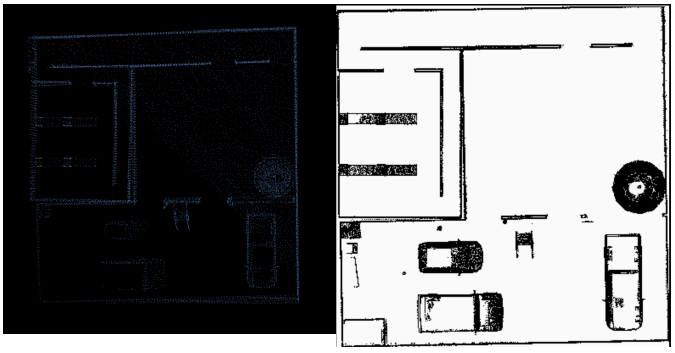

Figure 10: Filtered Point Cloud by A-LOAM and Map Generated by A-LOAM

Figure 10 presents the mapping outcome of A-LOAM. The sparsity is due to the hardware limitations of the Jackal platform's 16-beam LiDAR sensor.

In the process of enhancing the A-LOAM algorithm's scanning capabilities, an adjustment in the sensor configuration is mandated. Specifically, the substitution of the 3D LiDAR model from VLP-16 to HDL-32E within the Jackal robot's Unified Robot Description Format (URDF) files. This alteration is conducted in the accessories.urdf.xacro file located at *src/jackal_description/urdf*. The modification is succinctly encapsulated in the following XML property definition:

```
<xacro:property name="lidar_3d_model" value=
"$(optenv JACKAL_LASER_3D_MODEL vlp16)"/>
```

To effectuate the desired change for enabling the HDL-32E LiDAR configuration, the value parameter *vlp16* is to be replaced with *hdl32e*.

Executing this modification primes the system to leverage the increased resolution and scanning range offered by the HDL-32E model, thereby augmenting the environmental scanning efficacy of the A-LOAM algorithm.

In this evaluation of SLAM algorithms, FAST-LIO distinguishes itself as the superior choice for continued project development, attributed to its integrated LiDAR-inertial approach which ensures robust pose estimation even amidst sensor noise and dynamic environmental factors. FAST-LIO's efficient data handling, coupled with its real-time processing capability, further substantiates its selection.

4) *Evaluate FAST-LIO Performance:*

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
```

This assessment provides quantitative insights into the FAST-LIO algorithm's positional accuracy by comparing the estimated trajectory with the ground truth using visual plots. Such rigorous analysis is essential for verifying the algorithm's performance and ensuring that it meets the navigational requirements of the project's mini-factory environment.

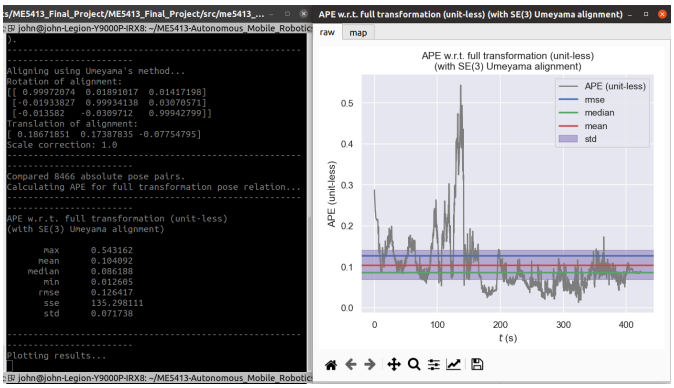

Figure 11: FAST-LIO APE Analysis

Figure 11 presents the APE results as a time series, allowing us to observe the deviation of the algorithm's estimated poses from the ground truth over the duration of its operation. The plot reveals that the FAST-LIO algorithm largely maintains a low APE, with the majority of the data points clustering around the lower end of the error spectrum.

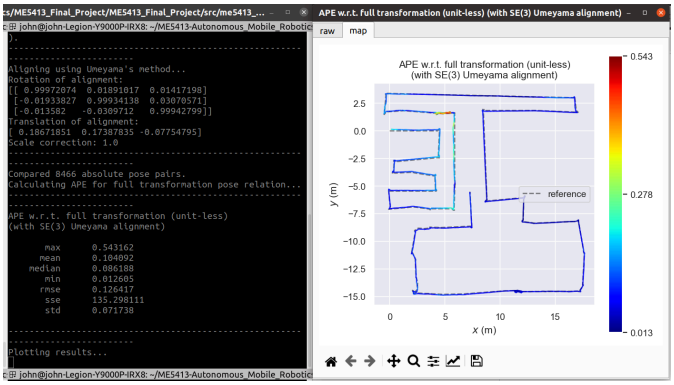

Figure 12: FAST-LIO APE Analysis

Figure 12 complements this analysis by plotting the APE across the XY plane, providing a spatial representation of the pose errors. The overlay of the FAST-LIO trajectory over the ground truth highlights the algorithm's precision in tracing the actual path taken by the robotic platform.

Upon meticulous evaluation, FAST-LIO emerges as the superior algorithm among its counterparts, FAST-LOAM and A-LOAM, for the project at hand. FAST-LIO's integration of LiDAR and IMU data facilitates a higher resolution and fidelity in environmental mapping, allowing for more accurate and consistent pose estimation.

C. *Task 2: Navigation*

Robot localization refers to the capability of a robot to ascertain its position and bearing within a given coordinate system. It serves as the foundation for path planning, wherein the robot not only identifies its current locale but also computes the trajectory to a designated target within the same spatial framework. Localization and path planning are interdependent, the former establishing a basis for the latter within the context of navigation using an extant map.

In the context of Task 2, these functions are embodied within a hierarchical structure delineating the robot's navigation assignment. This structural hierarchy outlines the robot's transformation framework, known as the TF tree. The TF tree represents the various coordinate frames and their interrelations, essential for the robot's spatial orientation and movement planning in the environment.

1) *Localization:*

AMCL: In this project, we utilized the Adaptive Monte Carlo Localization (AMCL) algorithm from ROS for the robot's localization task. AMCL is an adaptive particle filter-based localization algorithm that efficiently estimates the robot's pose in dynamic environments.

The key parameters of the AMCL algorithm were configured in the `amcl.launch` file:

- `odom_model_type="diff"`: Differential drive model to describe the robot's motion characteristics.
- Minimum and maximum number of particles set to 8000 and 10000, respectively, to improve localization accuracy.
- KLD sampling error parameter (`kld_err`) adjusted to 1.5 to reduce the maximum error between the true and estimated distributions.
- KLD confidence parameter (`kld_z`) set to 0.5, making the algorithm more conservative in deciding whether to increase the number of particles.
- Likelihood field sensor model adopted to handle measurement errors in dynamic environments.
- Minimum translation and rotation thresholds (`update_min_d` and `update_min_a`) set to trigger localization updates and improve robustness.

Additionally, the `global_costmap_params.yaml`, `local_costmap_params.yaml`, and `costmap_common_params.yaml` files were used to configure the parameters of the global and local cost maps, providing important environmental information for path planning and navigation tasks.

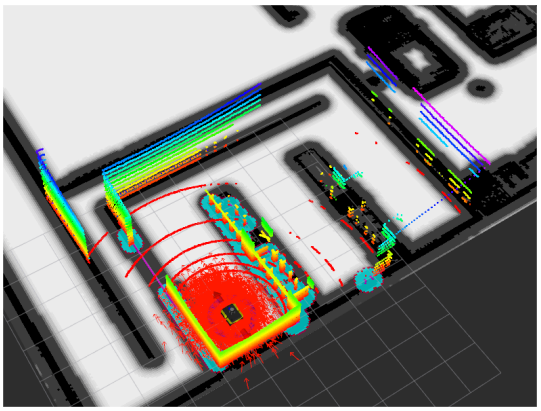

Figure 13: AMCL Localization

Other Localization Methods: In addition to AMCL, we explored two filter-based localization methods: the Extended Kalman Filter (EKF) and the Unscented Kalman Filter (UKF).

- **EKF:** A classic nonlinear estimation algorithm based on Kalman filtering, which linearizes the system and measurement models. Although relatively simple, EKF may introduce significant errors in complex nonlinear systems. EKF-based localization was implemented in the `robot_pose_ekf.launch` file.
- **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 can better handle 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.

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

2) *Planning:*

Costmap: The robotic system's navigation framework utilizes a composite costmap that is divided into global and local representations.

The global costmap represents the environmental model constructed by the Fast-Lio package, with its configuration parameters accessible in the `jackal_navigation/params/costmap_common_params.yaml` file. It provides a high-level overview of the robot's operational terrain.

On the other hand, 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.

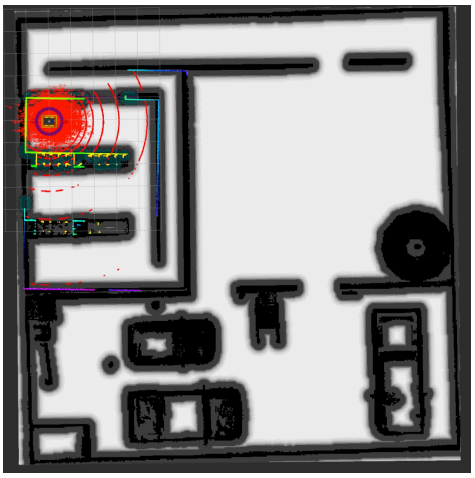

Figure 14: Global Costmap

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. The pixels corresponding to the restricted area are painted with a solid, obstacle-indicative color (usually black), effectively communicating the presence of insurmountable obstacles to the robot's costmap generation process. However, such direct manipulations alter the map's integrity and may misrepresent the actual layout of the environment. Therefore, this method, while effective, is generally discouraged in favor of more dynamic and reversible approaches.

Plugin Layer: The `move_base.launch` framework conceptualizes the costmap as comprising multiple layers, each serving a distinct function in the robot's navigation system:

- **Static Layer:** Responsible for integrating pre-existing static maps.
- **Obstacle Layer:** Manages dynamic obstacle inflation during the robot's movement, ensuring that transient obstacles are effectively incorporated into the navigation strategy.
- **Inflation Layer:** Extends the representation of obstacles within the navigation plane, effectively enlarging the obstacles' footprint on the map.

To implement spatial constraints and prevent the robot from entering designated zones, the `costmap_prohibition_layer` package is integrated. The parameters delineating the restricted zones are defined within the `jackal_navigation/params/prohibition_layer.yaml` file, demarcating the 'Restricted Area'. The `dynamic_obstacle_updater.py` script dynamically modifies the restricted zones by subscribing to the `/gazebo/cone_position` topic, ascertaining the cone's location, and updating the prohibition parameters to reflect the shifting blockade area.

In the Rviz visualization of the Global Costmap, these designated areas are represented in gray (Figure 13), indicating their status as high-cost routes that the navigation algorithm is conditioned to avoid. These modifications preserve the integrity of the underlying static map while overlaying dynamic navigation constraints.

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.

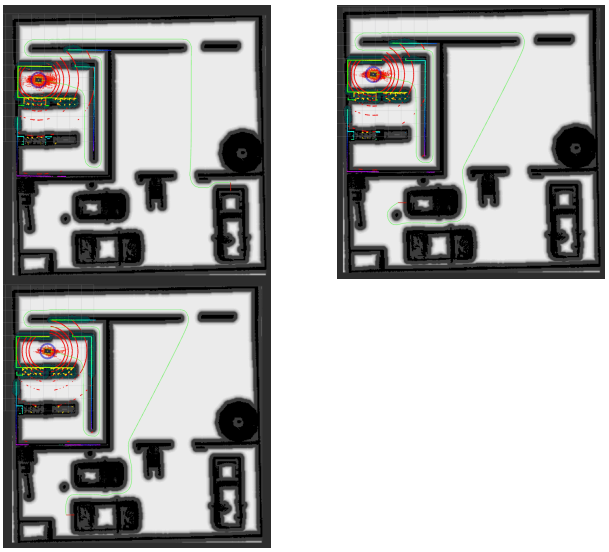

Figure 15: Global Navigation to Vehicle 1, 2, and 3

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.

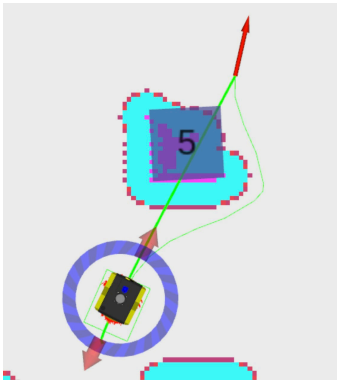

Figure 16: Improvements in local mapping for navigation

Figure 16 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.

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.

These parameters affect the representation of obstacles on the cost map, indirectly influencing robot path planning.

The Teb planner also includes optimization-related parameters. In this project, we have utilized six internal optimizations and five external optimizations, which together control the depth and detail of the optimization process.

The `weight_kinematics_nh` parameter in the TebLocalPlannerROS configuration is a weight value used to consider the robot's non-holonomic constraints during optimization. A value of '1000' indicates that compliance with non-holonomic constraints is considered very important, having a higher priority compared to other optimization objectives. Increasing this weight encourages the path planner to gener-

ate paths that are compliant with the robot's non-holonomic motion characteristics, ensuring that the generated paths are safe and feasible for the robot to execute.

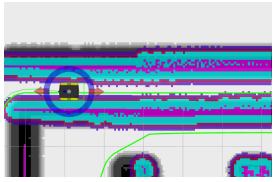

Figure 17: Normal inflation_radius = 0.3

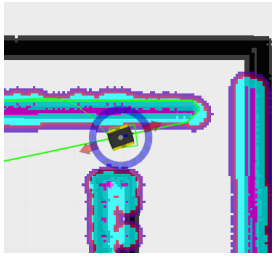

Figure 18: Smaller inflation_radius = 0.1

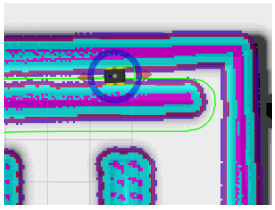

Figure 19: Bigger inflation_radius = 0.5

Figure 17, Figure 18, and Figure 19 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` (Figure 18) results in a more conservative buffer, potentially leading to overly cautious navigation. Conversely, a larger `inflation_radius` (Figure 19) 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.

Comparison 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 20, and a further comparison is presented in Figure 21.

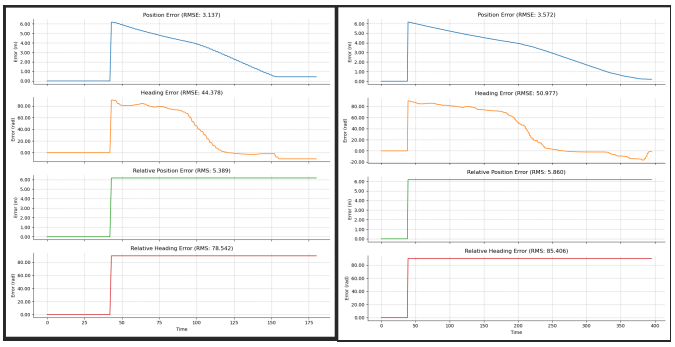

Figure 20: Errors Evaluation of DWA and TEB local Planners

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

3) *Object Detection:*

Visual Identification of Box3:

After generating random objects (boxes 1-9 and a cone), setting the box1, box2, box3, or box4 buttons on the simple panel will generate a random target point in the box room area. The random target point subscribes to `box_markers` to load the box's position and size (0.8, 0.8, 0.8) for detecting the box's area. The target point is generated 1 unit away from the `box_marker` to ensure it doesn't appear inside the box. When the Jackal vehicle moves towards the random target, if box3 is not detected, another random target point will be generated in the box room, allowing the Jackal to roam the area.

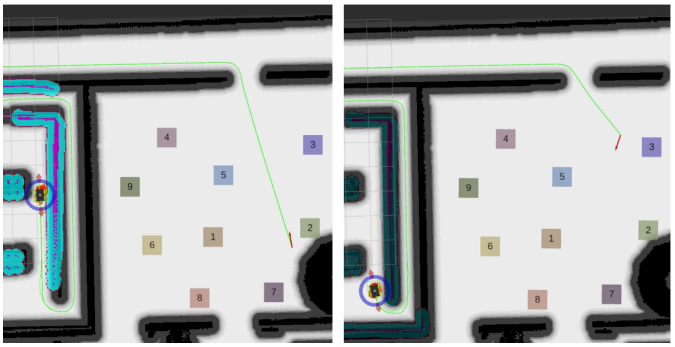

Figure 22: rondom point in box room

We employed the `find_object_2d` package, which uses the SIFT feature detection algorithm for image matching. The

find object node is added to the `navigation.launch` file and remaps and subscribes to the following topics:

```
<node   name="find_object_3d"   pkg="find_object_2d"
type="find_object_2d" output="screen">
  <param name="gui" value="true" type="bool"/>
      <param   name="settings_path"   value="~/.ros/
find_object_2d.ini" type="str"/>
      <param   name="subscribe_depth"   value="true"
type="bool"/>
       <param   name="objects_path"   value="$(find
me5413_world)/pictures" type="str"/>
      <param   name="object_prefix"   value="object"
type="str"/>
  <remap from="rgb/image_rect_color" to="/front/rgb/
image_raw"/>
   <remap from="/depth_registered/camera_info" to="/
front/depth/camera_info"/>
    <remap  from="/depth_registered/image_raw"  to="/
front/depth/image_raw"/>
</node>
```

The package loads images of numbers on box3 from different angles as a dataset.

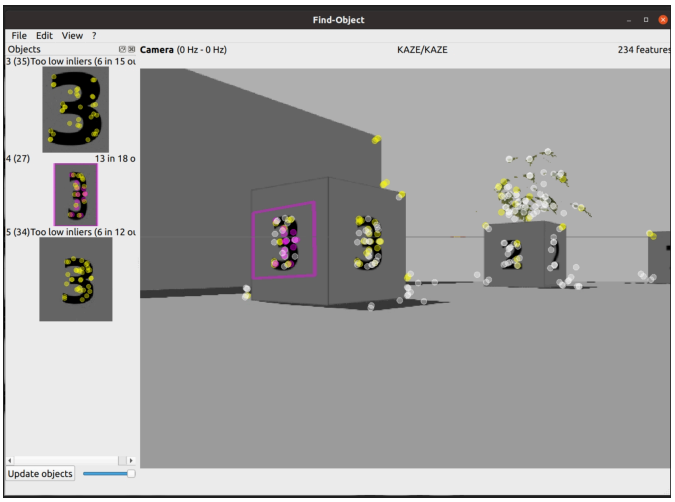

Figure 23: box3_detection

When box3 is detected, the `/objects` topic is published, including the detected target ID, width, height, and a 3x3 homography matrix containing the image's location and pose.

To calculate the distance from the Jackal to the target box, depth information is needed. We modified the vehicle's camera to a Kinect depth camera with a 60-degree field of view, a minimum measurement distance of 0.05 meters, and a maximum of 8 meters. The camera publishes:

- RGB Image Data: `rgb/image_raw`
- Depth Image Data: `depth/image_raw`
- Point Cloud Data: `depth/points`
- Camera Information: `rgb/camera_info`, `depth/camera_info`

We created a ROS node script, `cal_target_point.py`, which subscribes to:

- Camera Information: `/front/rgb/camera_info`
- Depth Image: `/front/depth/image_raw`
- RGB Image: `/front/rgb/image_raw`
- Target Detection Results: `/objects`

Based on the subscribed topics, the script publishes:

- Navigation Target Points: `/move_base_simple/goal`
- Target Name for Display in RViz: `/rviz_panel/goal_name`
- Calculated Target Position: `/target_goal`

To obtain the depth value of the target center point, the script uses the target center coordinates (`center_x`, `center_y`) from the detection results to retrieve the depth value from the depth image, representing the distance from the target to the camera lens.

The pinhole camera model is used to convert 2D image coordinates to 3D spatial coordinates using the camera's intrinsic parameters (focal lengths `fx`, `fy`, and principal point `cx`, `cy`) obtained from the `CameraInfo` message. The 3D coordinates (X, Y, Z) of the target point in the camera coordinate system are calculated as follows:

$$X = \frac{(\text{center}_x - c_x) \cdot Z}{f_x} \tag{1}$$

$$Y = \frac{(\text{center}_y - c_y) \cdot Z}{f_y} \tag{2}$$

$$Z = \text{depth} \tag{3}$$

To transform the target point from the camera coordinate system to the map coordinate system, the `tf` library is used. A `PointStamped` message containing the 3D coordinates and reference frame (`front_frame_optical`) is created. The `waitForTransform` method is used to wait for the transformation from camera to map coordinate system, and the `transformPoint` method performs the transformation.

Finally, a `PoseStamped` message is constructed with the transformed point's position and default orientation (upwards) and published to `/move_base_simple/goal` to direct the navigation system to move the robot to the target position.

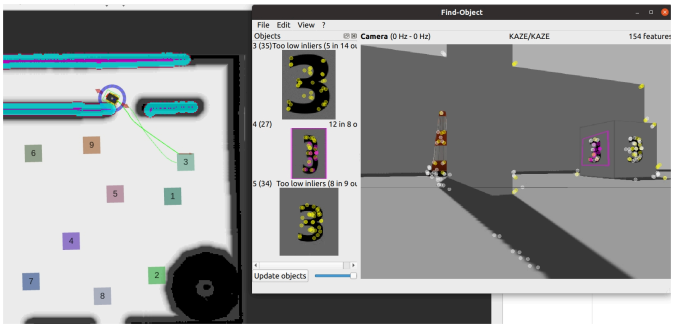

Figure 24: box3_identification

Two Methods for Object Detection:

Object recognition is performed using two methods:

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.

Choice of Camera:

Two approaches were considered for robotic navigation to a specific object:

1. **RGB-D Camera:** Augments the conventional non-depth camera with an RGB-D camera to acquire direct depth information, enabling precise distance computation from the robot to the object, enhancing navigation accuracy and efficiency.
2. **Non-Depth Camera with Simulated Depth:** Retains the original camera setup and imputes a nominal depth to the identified object, navigating based solely on 2D visual data. Less precise and may increase collision risk due to the absence of true depth perception.

Figure 25 and Figure 26 illustrate the RGB-D camera-based and non-depth camera-based navigation processes, respectively.

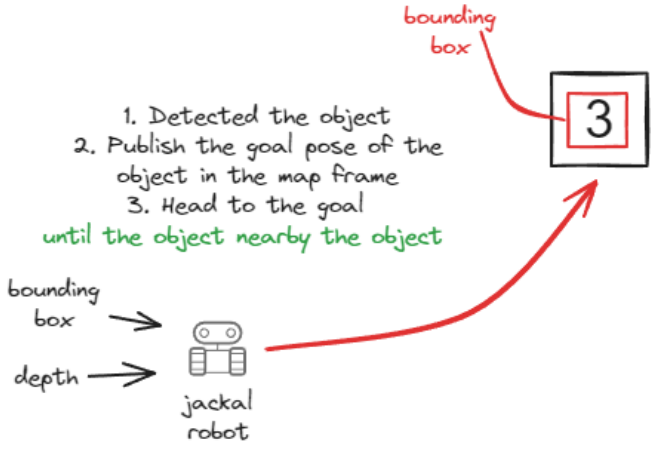

Figure 25: RGB-D Camera-based Navigation Process

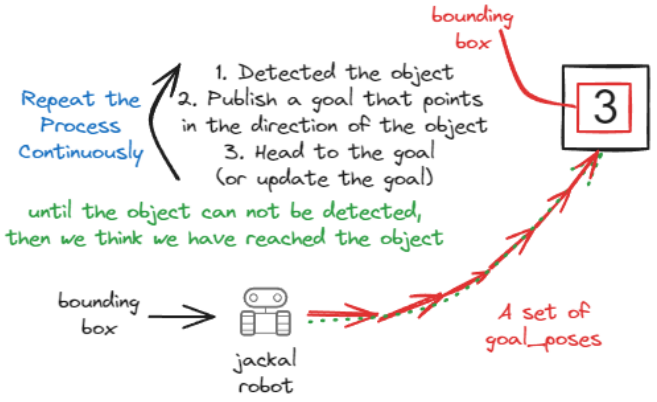

Figure 26: Non-Depth Camera-based Navigation Process

Camera Calibration Policy:

Upon the detection of an object, it becomes essential to compute the distance from the robot to the object as well as to ascertain the object's pose within the map frame. The camera calibration process integral to these computations is conducted through the calculate_target() function. Once the robot captures a bounding box, this function is invoked to calculate the distance and pose of the object. The process is depicted in Figure 27 and is outlined as follows:

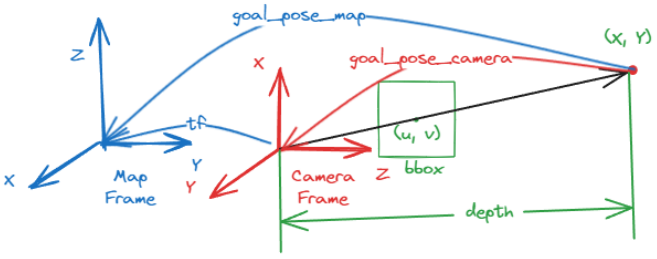

Figure 27: Camera Calibration Method

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

$$u = \text{bbox.x} + \frac{\text{bbox.width}}{2} \tag{4}$$

$$v = \text{bbox.y} + \frac{\text{bbox.height}}{2} \tag{5}$$

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:

$$K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \tag{6}$$

Employing the camera intrinsic matrix K, the 3D point in the camera frame is calculated using the formulas:

$$X = \frac{(u - c_x) \cdot Z}{f_x} \tag{7}$$

$$Y = \frac{(v - c_y) \cdot Z}{f_y} \tag{8}$$

pagebreak()

4) *Decision making:*

The decision making logic is shown in the flowchart below:

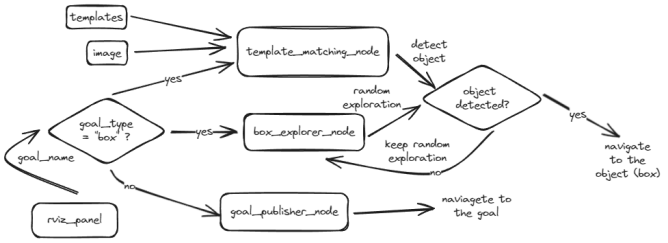

Figure 28: Flowchart of decision making

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:

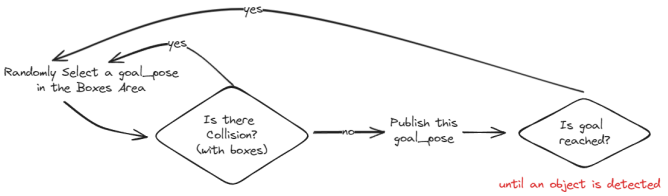

Figure 29: Flowchart of decision making

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: In the random exploration process, we need to check whether the selected `goal_pose` is in the obstacles. Initially, we had an idea to subsribe the `/gazebo/global_costmap` and use a circular collision checking method to to check whether the `goal_pose` is in the obstacles. However, we found that the global costmap does not contain the information of the spawned boxes, so the method can not work properly.

Then 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`).

I. Appendix

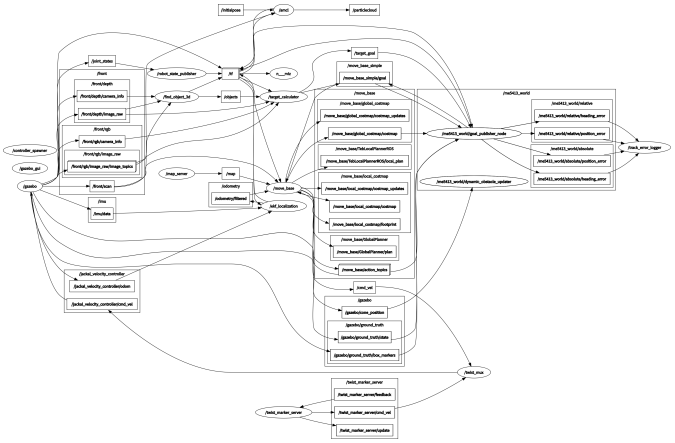

Figure 30: rosgraph

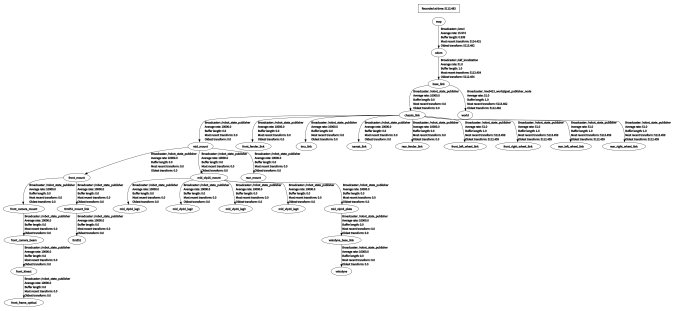

Figure 31: rosgraph

References