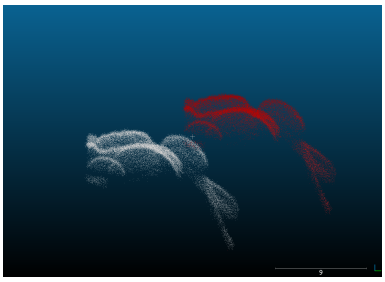
1 Task 1: Writing the ICP Algorithm for Point Cloud Registration

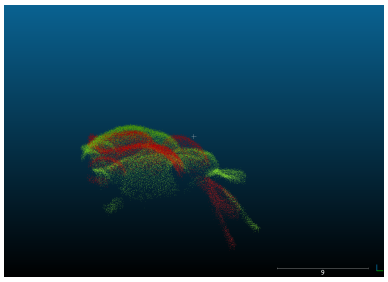
1.1 Algorithm and Implementation

Iterative Closest Point (ICP) is an algorithm designed to align two point clouds by minimizing the distance between them through iterative refinement. A core component of ICP is Singular Value Decomposition (SVD), which computes the optimal rotation matrix by decomposing the covariance matrix, ensuring accurate and valid transformations.

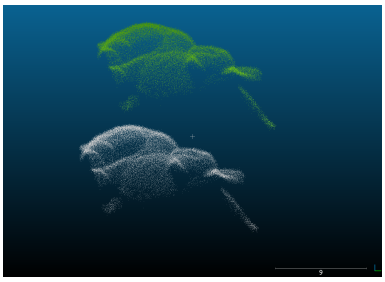
To improve accuracy and simplify the calculation of the covariance matrix, decentralization ensures that the calculated rotation matrix is independent of the absolute position of the point cloud by subtracting the center of mass of the point cloud. The effect of decentralization is shown in Figure 1.



(a) Original (white) vs. decentralized target point cloud (red).



(b) Position comparison of decentralized.



(c) Original (white) vs. decentralized source point cloud (green).

Figure 1: Position comparison of decentralization for target (red) and source (green) point clouds.

Another key aspect of ICP is the handling of point correspondences: if the correspondences are known, the transformation matrix can be directly computed (Task 1.1); if they are unknown, an iterative process is required to estimate and refine these correspondences (Task 1.2). For the second case, KD-Tree is utilized to efficiently find the nearest neighbors between point clouds, as shown in Figure 2. By organizing point data into a spatial indexing structure, KD-Tree efficiently finds nearest neighbors, accelerating the iterative estimation of correspondences. Figure 9 in the Appendix illustrates the direct computation approach with known correspondences in the left, while the right side demonstrates the iterative process for handling unknown correspondences.

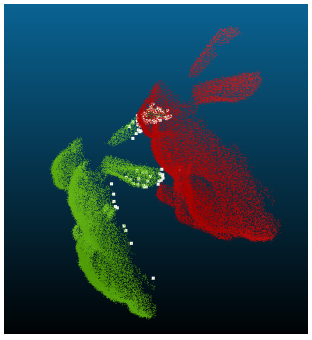


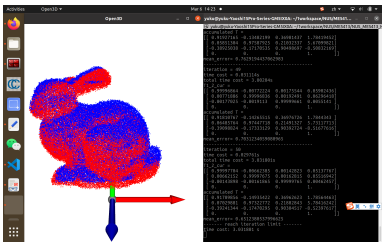
Figure 2: KD-Tree reordered point cloud registration (white): target (red) vs. source (green).

1.2 Results and Analysis

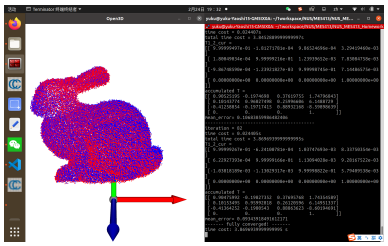
1.2.1 Qualitative Results

Figure 3 shows that Task 1.1 successfully aligns the point clouds with minimal error, as evidenced by the well-overlapped transformed point cloud. For Task 1.2, the alignment improves progressively, demonstrating effective refinement of the transformation matrix. The transformation matrices printed in the terminal shown in Figure 4 confirm that the iterative process gradually converges to a solution similar to that of Task 1.1, indicating successful point cloud registration even without known correspondences.

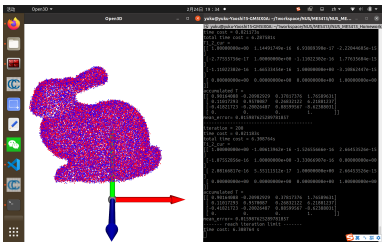
Figure 3: Task 1.1 Visualization result for source (green), target (red), and transformed source (blue) point clouds.



(a) Visualization result with iteration = 50.



(b) Visualization result with iteration = 100.



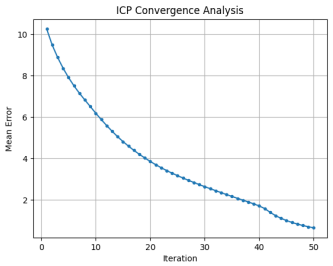
(c) Visualization result with iteration = 200.

Figure 4: Task 1.2 Visualization result for transformed source (blue) and source (red) point clouds.

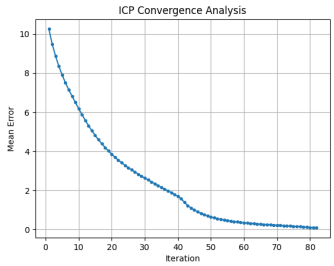
1.2.2 Quantitative Results

The convergence analysis in Figure 5 shows that Task 1.2 can achieve convergence with sufficient iterations. The 100-iteration case stopped early at 82 (Figure 4 (b)) iterations due to reaching the error threshold, explaining its lower time cost compared to the 200-iteration case, which showed convergence stagnation. The similar time costs for the 50-iteration and 82-iteration cases can be explained by two factors: first, early iterations involve higher computational costs due to larger adjustments, while later iterations primarily make minor refinements; second, the nearest neighbor search dominates time consumption, especially in early stages, making the total time difference minimal.

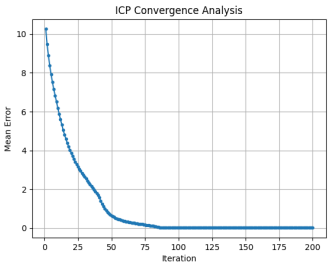
As shown in Table 1, Task 1.1 achieves the lowest mean error with minimal computation time due to the direct calculation of the transformation matrix. Although Task 1.2 requires iterative computation, it can achieve almost identical accuracy once converged, as seen in the 200-iteration case. This demonstrates that Task 1.2 is capable of matching Task 1.1's accuracy despite higher computational costs, making it a viable option when known correspondences are unavailable.



(a) ICP convergence analysis in 50 iteration.



(b) ICP convergence analysis in 100 iteration..



(c) ICP convergence analysis in 200 iteration.

Figure 5: ICP convergence analysis in different iterations.

	Task 1.1	**Task 1.2 (iter=50)**	**Task 1.2 (iter=82)**	**Task 1.2 (iter=200)**
Time Cost (s)	0.000536	3.031801	3.869694	6.308764
Mean Error	0.015988	0.651238	0.093439	0.015988

Table 1: Comparison of ICP Results with Known and Unknown Correspondences

The transformation matrix calculated from Task 1.1 is:

$$T = \begin{bmatrix} 0.90163997 & -0.20982829 & 0.37817648 & 1.7658544 \\ 0.11017046 & 0.95700854 & 0.26832281 & 6.21876911 \\ -0.41821983 & -0.20026669 & 0.88599403 & -8.62390124 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The final accumulative transformation matrix with 200 iterations calculated from Task 1.2 is:

$$T = \begin{bmatrix} 0.90164088 & -0.20982929 & 0.37817376 & 1.76589631 \\ 0.11017293 & 0.9570087 & 0.26832122 & 6.21881237 \\ -0.41821723 & -0.20026487 & 0.88599567 & -8.62388031 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2 Task 2: Running a SLAM Algorithm in ROS

2.1 Introduction, Implementation, and Methodology

Simultaneous Localization and Mapping (SLAM) is a fundamental technique in mobile robotics that enables a robot to build a map of an unknown environment while simultaneously determining its location within that map. In this task, Google Cartographer, an open-source library widely used for 2D and 3D SLAM, was employed to construct a map based on data collected by a Jackal Unmanned Ground Vehicle (UGV) equipped with a 2D laser and a 3D Velodyne sensor. Due to the simplicity of the environment and computational considerations, only 2D LiDAR data was utilized.

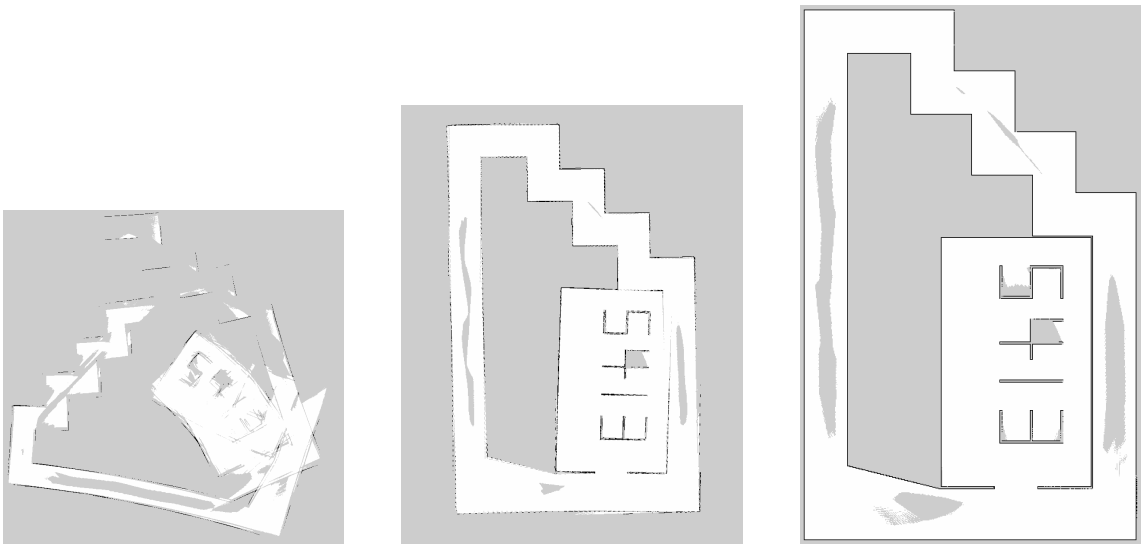
The performance of the generated map and the accuracy of the robot's trajectory were evaluated using the evo tool. Table 4 provided in Appendix summarizes the evo Absolute Pose Error (APE) metrics, where lower values indicate better performance.

The frames and tf tree generated during the execution are provided in the Appendix Figure 10 and Figure 11, respectively.

2.2 Parameter Tuning and Results

2.2.1 Default and Origin Parameters Performance

The original Lua file set `TRAJECTORY_BUILDER_2D.num_accumulated_range_data` = 10, which led to significant mapping inaccuracies as shown in Figure 6 (a) and Table 3. Commenting this parameter and using default values improved results slightly but was still suboptimal, as depicted in Figure 6 (b).



(a) Original parameters map. (b) Default parameters map. (c) Tuned parameters map.

Figure 6: Mapping results: Original vs. Default vs. Tuned parameters.

evo APE	Origin	Default	Tuned	Case 1	Case 2
max	19.406266	0.759563	**0.099321**	0.821805	0.809161
mean	6.553552	0.252703	**0.038960**	0.269606	0.322801
median	4.862016	0.219900	**0.038367**	0.207566	0.293443
min	0.189023	0.003878	**0.000798**	0.000993	0.091765
rmse	8.315716	0.291734	**0.099321**	0.340979	0.350111
sse	199985.077574	246.560771	**5.199136**	336.823609	355.108517
std	5.118797	0.145774	**0.016656**	0.208756	0.135563

Table 3: Comparison of evo APE results for different parameter settings.

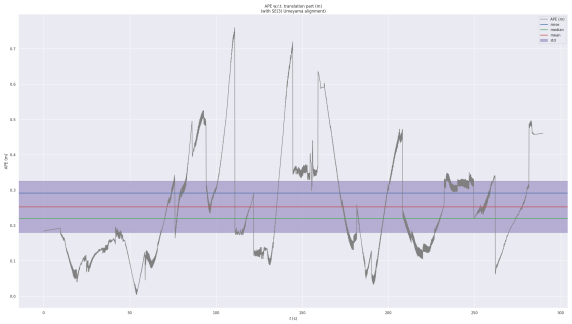
Case 1: Only set `TRAJECTORY_BUILDER_2D.use_online_correlative_scan_matching` = false.
Case 2: Only set `TRAJECTORY_BUILDER_2D.submaps.num_range_data` = 100.

2.2.2 Tuned Parameters Performance

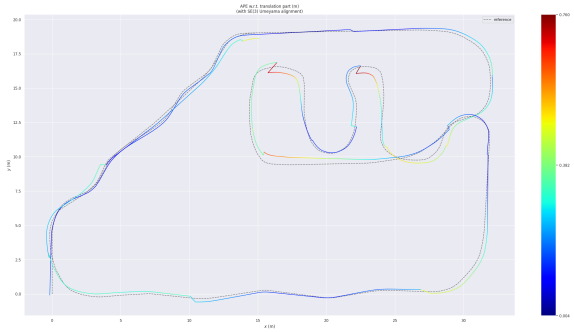
To enhance accuracy, two key parameters were adjusted:

- `TRAJECTORY_BUILDER_2D.use_online_correlative_scan_matching` = true:
  Improved real-time scan alignment accuracy, significantly reducing RMSE and SSE.
- `TRAJECTORY_BUILDER_2D.submaps.num_range_data` = 5000:
  Increased submap size, minimizing stitching errors and enhancing map consistency.

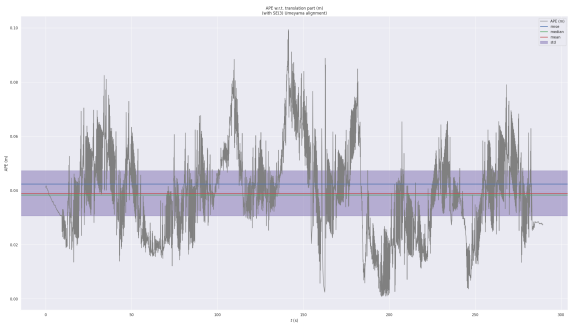
The tuned parameters led to a great improvement in trajectory accuracy and smoothness compared to the default settings. The default configuration exhibited noticeable zig-zag patterns (Figure 7 (b)) and abrupt deviations (Table 3) due to frequent submap stitching and limited real-time scan matching precision, as reflected in the high max error, RMSE, and SSE. In contrast, the tuned settings effectively reduced all error metrics, indicating a more continuous and consistent path (Figure 7 (d)). This improvement is primarily due to the elimination of frequent submap transitions and enhanced real-time matching accuracy, which mitigated sensor noise and motion uncertainties. The results are illustrated in Figure 6 (c) and Table 3.



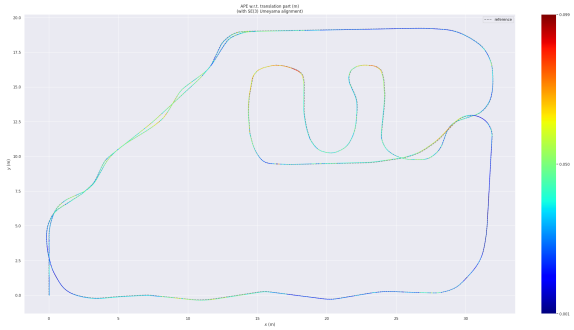
(a) Default: evo APE raw trajectory.



(b) Default: trajectory comparison map between ground truth and computed output.



(c) Tuned: evo APE raw trajectory.

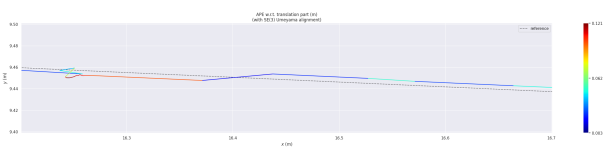


(d) Tuned: trajectory comparison map between ground truth and computed output.

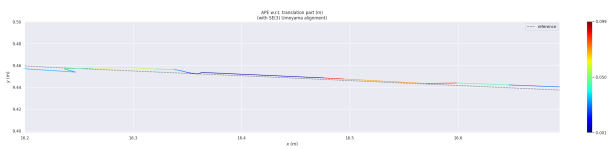
Figure 7: Comparison of evo APE results with default and tuned parameter settings.

2.2.3 Other Parameters Attempted

Other adjustments, such as increasing `TRAJECTORY_BUILDER_2D.real_time_correlative_scan_matcher.translation_delta_cost_weight` and `TRAJECTORY_BUILDER_2D.real_time_correlative_scan_matcher.rotation_delta_cost_weight`, led to degraded performance due to overemphasis on small deviations, as shown in Figure 8. Thus, they were not included in the final configuration.



(a) Increased weight leads to drift.



(b) Default weight with stable trajectory.

Figure 8: Impact of different weight settings on trajectory stability.

A Appendix

A.1 Task 1

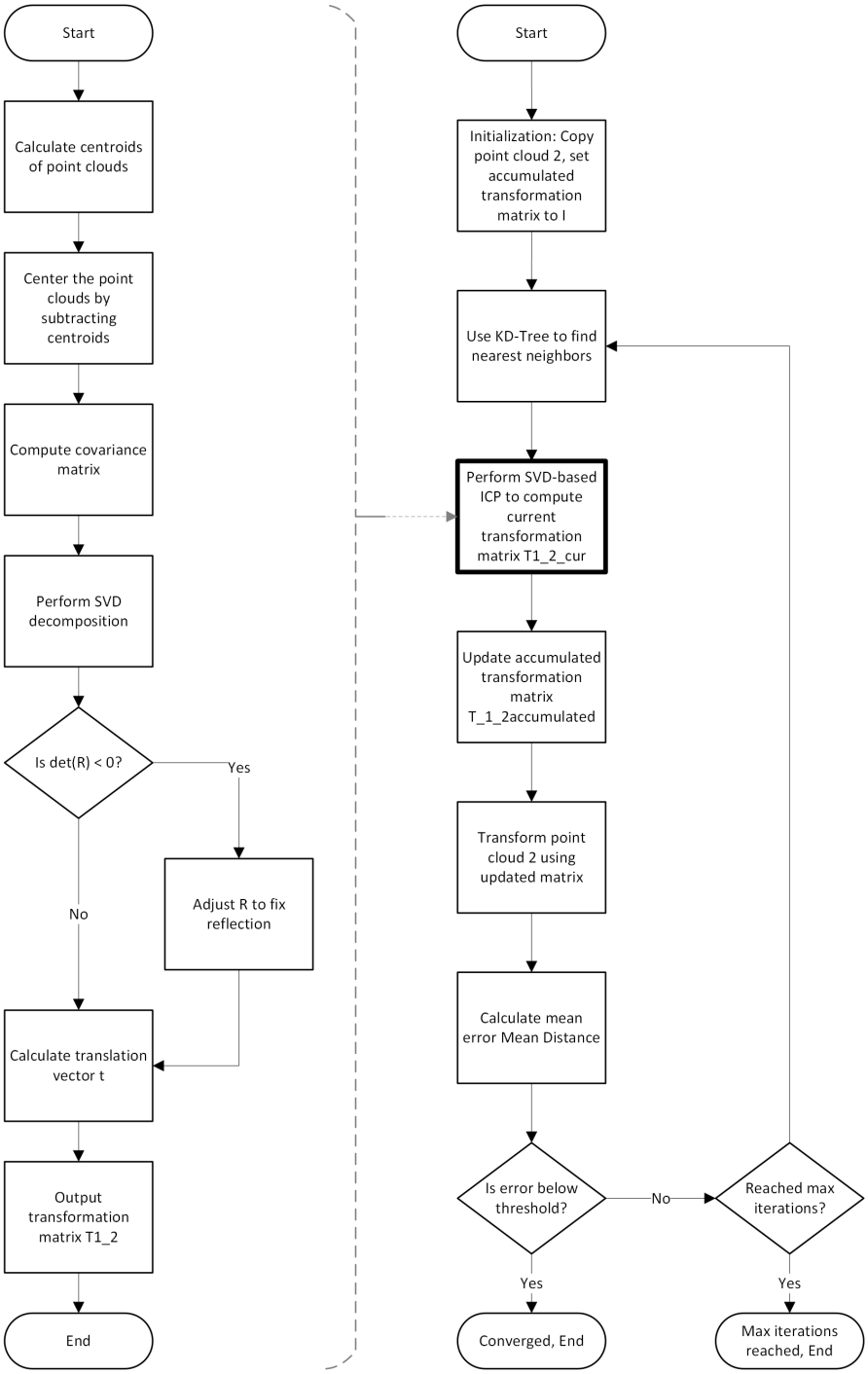


Figure 9: Task 1 flowchart, with Task 1.1 in left, and Task 1.2 in right.

A.2 Task 2

Metric	Description	Optimal Value
max	Maximum error observed in the trajectory.	Lower is better
mean	Average error across the entire trajectory.	Lower is better
median	Median error value, representing the central tendency.	Lower is better
min	Minimum error observed in the trajectory.	Lower is better
rmse	Root-mean-square error, emphasizing larger errors.	Lower is better
sse	Sum of squared errors, highlighting cumulative error.	Lower is better
std	Standard deviation of errors, indicating consistency.	Lower is better

Table 4: Metrics in evo APE.

The following are simplified key commands used for executing Cartographer:

Listing 1: Key commands executed for Cartographer.

```
#!/bin/bash
# Start roscore
roscore
# Run CartograpAbsolute Pose Errorher SLAM with the ROS bag file in a new terminal
roslaunch cartographer_ros demo_jackal_2d.launch bag_filename:=task2.bag
# Record ROS topics during SLAM execution
rosbag record -a -O task2_cartographer_result.bag
# Save the generated map using map_server
rosrun map_server map_saver --occ 70 --free 30 -f task2 map:=/map
# Evaluate SLAM trajectory against ground truth using evo
evo_ape bag task2_cartographer_result.bag /ground_truth /tf:map.base_link --plot --align
    --plot_mode xy
```

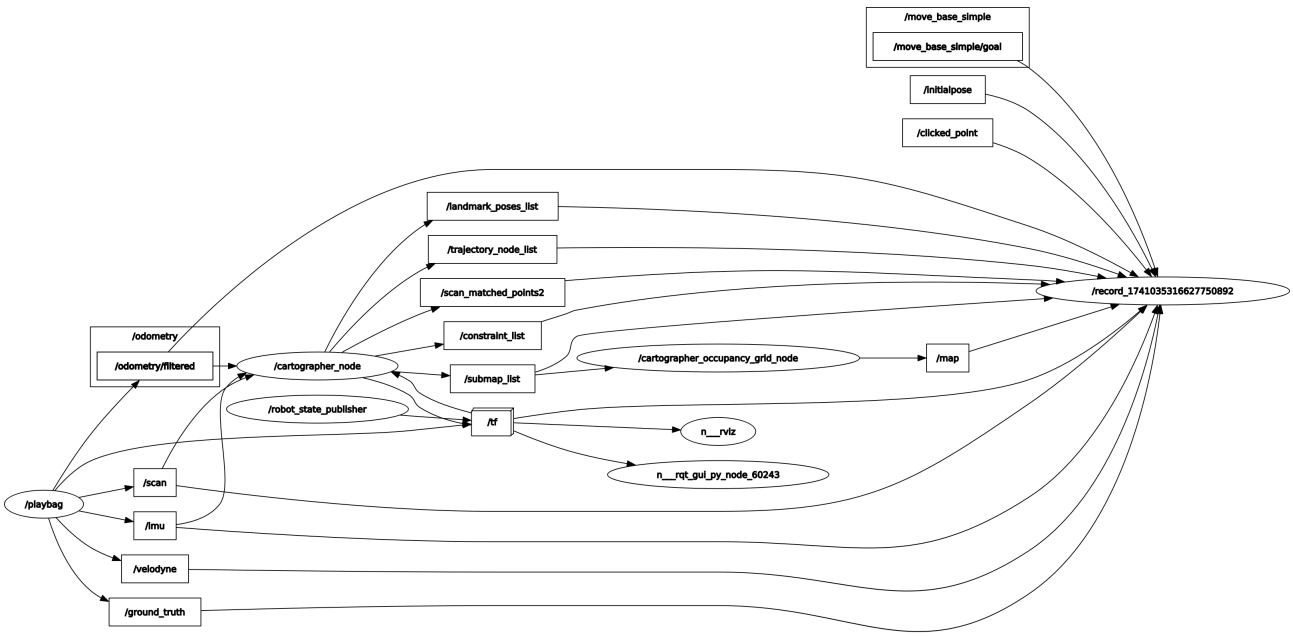


Figure 10: Task 2 Cartographer ROS graph.

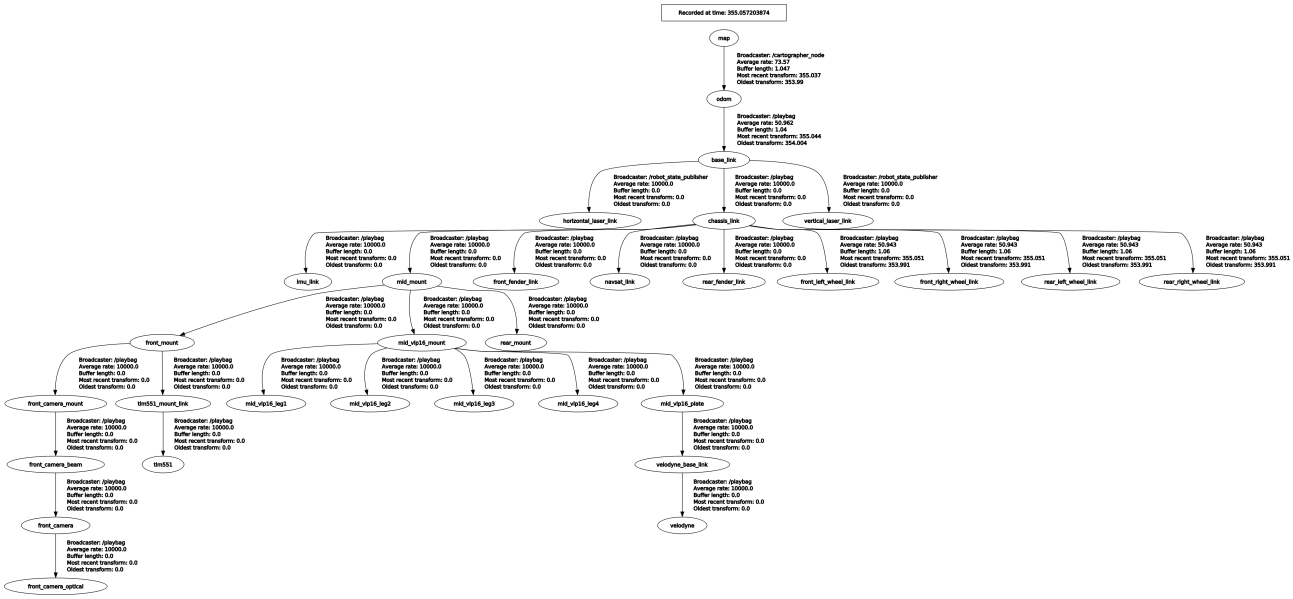


Figure 11: Task 2 Cartographer /tf tree graph.