PROJECT REPORT

Introduction to Robotics

MEIC 2024/2025

Group 4

- Rúben André Silva Cruz - IST1111005
- David Filipe Oliveira Pereira - IST1111232
- Vincenzo Filangeri - IST1110837

Mini-Project 1: Mobile Robot Localization Using ROS

---

1. Introduction

Autonomous Navigation, localization and interaction with the environment are crucial aspects of mobile robotics. This project, divided into two projects, focused on developing and testing localization and path planning techniques using the ROS framework and the TurtleBot3 Waffle Pi robot [1].

In **Mini-Project 1**, the goal was to implement and analyze localization algorithms: the Extended Kalman Filter (EKF) and Adaptive Monte Carlo Localization (AMCL). These methods were compared and tested using a baseline scenario where no localization method was instead only the direct data from the odometry.

Mini-Project 2, instead, expands the work done for the localization in the previous phase, concentrating on path planning and trajectory tracking for the TurtleBot3 Waffle Pi. In this phase, we made use of ROS packages like move_base and costmap_2d where a custom Rapidly Exploring Random Trees (RRT) was used for global path planning and a Dynamic Window Approach (DWA) planner was used for the local path planning . Gazebo was used to simulate a test environment in order to adjust parameters given by the previously mentioned packages. In

the end a test with the real robot was performed to verify the performance of the RRT implementation and chosen parameters.

Together, these mini-projects offer practical insights into mobile robot localization and navigation, focusing on optimizing parameters, ensuring data accuracy, and implementing robust algorithms for efficient robot autonomy.

2. Mini Project 1: Localization

The objective of Mini-Project 1 was to familiarize with practical aspects of mobile robot localization, using filters to handle uncertainty in perception and action effects. We utilized the TurtleBot3 Waffle Pi robot [1], equipped with a laser scanner, IMU and odometry, for data acquisition. Additionally a rosbag was provided which contains a high precision mocap sensor while the real world test made in the class lab does not. The algorithms were implemented using ROS packages such as robot_localization for EKF and amcl for AMCL [2], [3], [4]. The aim was to collect and process the data and compare the results, focusing, among the others, on: parameter choices, evaluation metrics and error study.

2.1. Experimental Setup and Parameter Choice

For the implementation, the following setup and parameters choice was used:

- **EKF (Extended Kalman Filter)**: obtained combining odometry's x, y, yaw, x velocity and y velocity data with IMU's yaw, yaw velocity and x acceleration data. The inclusion of x and y data in the odom confers information about the robot's pose obtained with the mocap sensor as such we chose to only run EKF with 1Hz.
- **AMCL (Adaptive Monte Carlo Localization)**: Utilized the amcl package with laser scan and odometry inputs. The number of particles was set between a minimum of 500 and a maximum of 2000 for a balance between accuracy and efficiency. The same reasoning was why the error and probability threshold for Kullback-Leibler divergence (KLD) sampling was set for 0.05 and 0.99.
- **No Localization**: A baseline scenario relying on raw odometry data to show the impact of having no localization correction to highlight the results of the other methods.

Two experimentation environments were used, namely a provided rosbag captured using a mocap sensor and other topics necessary for localization, and a recorded rosbag recorded in the lab class, both using a Turtlebot3 Waffle Pi robot, but the recorded rosbag doesnt possess any mocap data, making it impossible to compare the estimated path results with the real path, such comparison was made for the provided rosbag and is presented below.

2.2. Results and Comparisons

2.2.1. Data Smoothing and Outlier Removal

For the following presented data for error, smoothing and outlier removal techniques were removed to better highlight the trends and eliminate data inconsistencies resulting from noise in the mocap data of the rosbag or any missing data for a timestamp. Namely:

Smoothing: A moving average filter with a window size of 10 was applied to all the datasets to reduce noise.

Outlier Removal: The z-scores were calculated and data points beyond a threshold of 2.5 standard deviations were removed in the AMCL dataset to deal with extreme fluctuations caused by sensor noise or data inconsistencies. This technique was not applied to the other datasets.

2.2.2. Analysis of Methods

- **No Localization**: The original error, in fig. 1 shows a significant drift, exceeding 175mm. Without any corrective mechanism, the robot's accuracy declines over time. Even after smoothing, the increasing error trend remains, confirming that no localization correction results in unreliable position estimates.
- **AMCL**: AMCL achieves the lowest average error, remaining mostly below 100mm and an average of 53mm, except for a spike around 80 seconds caused by missing rosbag data. As shown in Figure 3, applying smoothing and outlier removal removes this spike, demonstrating AMCL's robustness.
- **EKF with 1HZ**: looking at the fig. 2, the EKF method reduces error compared to the baseline (even if the trend stays almost identical), remaining generally below 120mm and an average of 53mm. We can see big fluctuations around the 80 and 110 second mark corresponding to fast and tight turns of the robot. To remedy this adding extra sensors is a reasonable approach but if there are cost limitations on the quality and quantity of the sensors and you have sufficient computational capacity AMCL could be preferred. Yet we do not oppose extra fine tuning of parameters to increase its performance.
- **EKF with 10HZ:** As seen in fig. 4 we can infer that increasing the frequency resulted in an even worse error than our baseline, this we believe is due to higher noise propagation of the considered sensors resulting in a drastic underperformance.

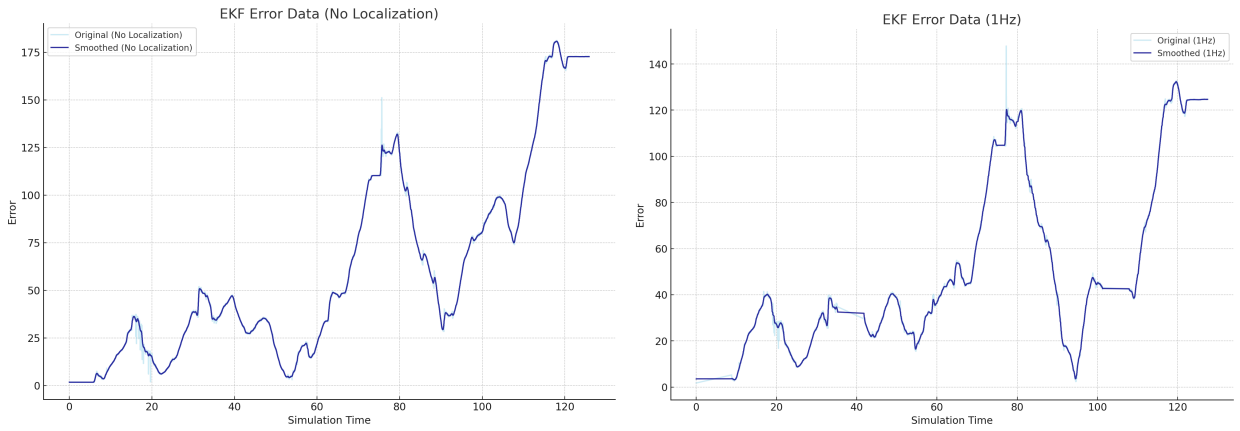

Figure 1 and 2: Comparison of error data for each localization method: No Localization and EKF 1HZ showing the original and smoothed data.

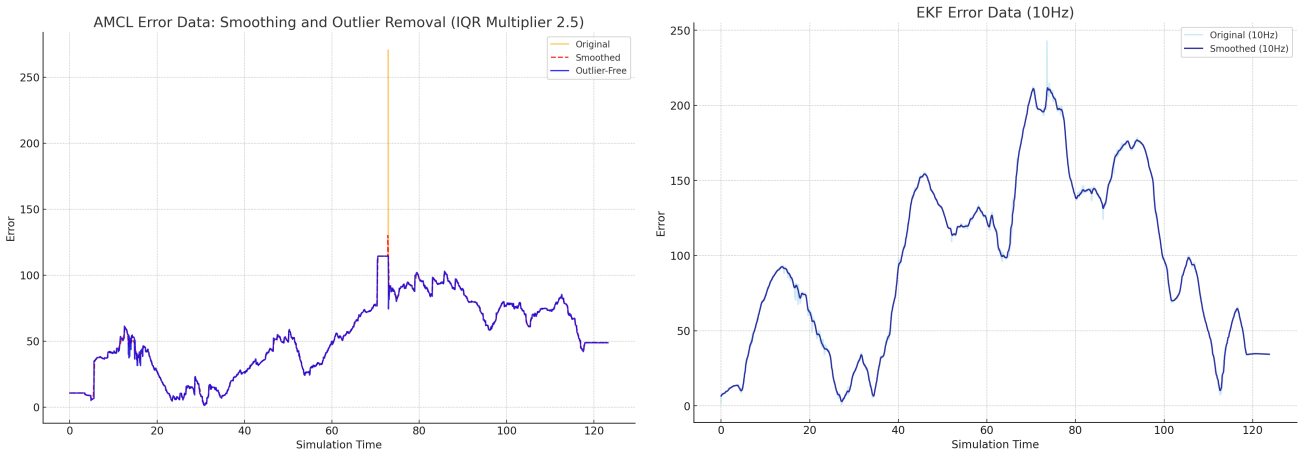

Figure 3 and 4: Comparison of error data for each localization method: AMCL and EKF 10Hz, showing the original, smoothed, and outlier-free data.

	Method	Maximum error (in mm) before Smoothing and Outliers Removal (for AMCL)	Maximum error (in mm) after applying Smoothing and Outliers Removal (for AMCL)	Mean Error (in mm)
1	No Localization	181.17	180.77	60.70
2	EKF (1 Hz)	147.82	132.32	52.88
3	EKF (10 Hz)	242.90	211.67	95.90
4	AMCL	270.53	117.80	53.14

Table 1: Summary of the average and maximum errors before and after applying smoothing and outlier removal techniques for each localization approach.

2.2.3. Uncertainty Study of Localization Methods

In this section, we present a uncertainty study that compares AMCL and EKF 1HZ (we omit the results for 10HZ as it is the same but with higher fluctuation frequency) based on simulation data, focusing on the trends, stability.

The uncertainty variance observed in the experiments are the following:

- **AMCL**: The uncertainty trend for AMCL reveals a descending pattern over time. Initially, the uncertainty is high starting at 0.4m, but with the progression of the simulation the AMCL accumulates more consistent data, for this reason, the variance decreases stabilizing at around 0.05m, this indicates an improvement in the localization accuracy over time as its expected of this method as it needs a brief calibration period.
- **EKF at 1 Hz**: The EKF at 1 Hz configuration shows a very stable trend at around 0.06m but has a very dramatic fluctuation of the uncertainty, this is because the data of the mocap is being inserted into the localization, without this injection the uncertainty would only increase overtime but after its real position is passed its uncertainty is reduced to almost 0, only to increase again explaining this variance.

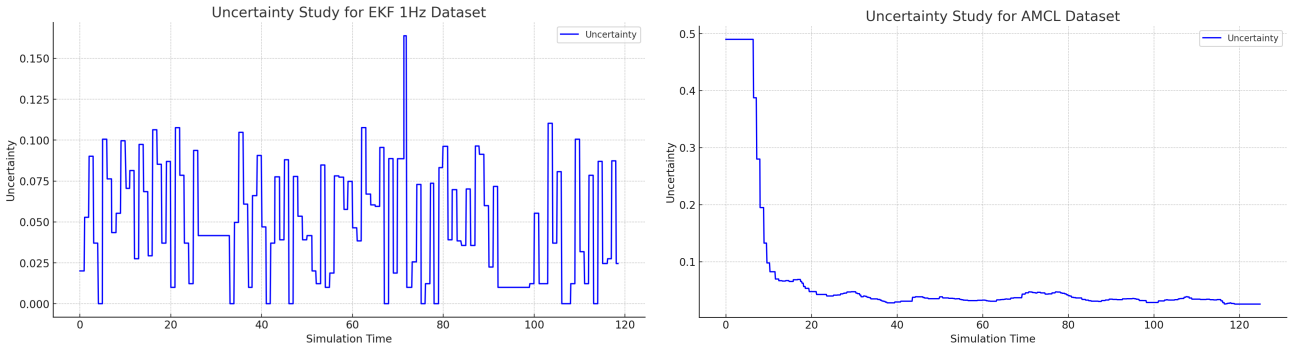

Figure 5 and 6: Uncertainty evolution plot for the EKF (left) and AMCL (right) methods, displaying its evolution over the duration of the simulation.

2.2.4. Real and Estimated Paths comparison

In Figures 7, 8, 9 and 10 displayed below, we can see a comparison between the real path represented in green, obtained with the mocap data and the estimated paths using EKF (left) and AMLC (right) being represented by the red paths. Specifically, two different scenarios and environments are shown: figure 7 and 8 displays the obtained paths from the given rosbag captured using mocap, while figure 9 and 10 were obtained using a real Turtlebot3 waffle pi robot running in the lab class. Previously we saw that AMCL has a lower error but looking at the obtained paths seen in fig. 7 and 8 we can see that in practice even in the areas where the error is larger the positioning isn't drastically different from the real path, the same can be said from

fig 9 and 10 as both paths are fairly similar yet it's worth to point out that in fig. 9, EKF shows a minor misalignment in the robot's pose, that could be caused, among the other reasons, by the drifting of the wheels, while AMCL in fig. 10, instead, presents a high degree of noise in the estimated path, especially in areas of the room in which the robot moved faster or where several obstacles like chairs, tables or even people were encountered.

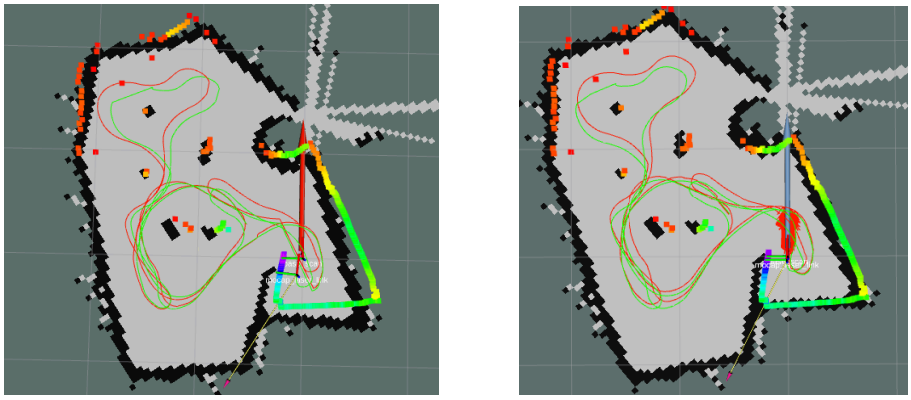

Figure 7 and 8: Real and Estimated Paths Comparison in the Gazebo TurtleBot3 simulation environment using EKF Localization (left) and AMCL (right).

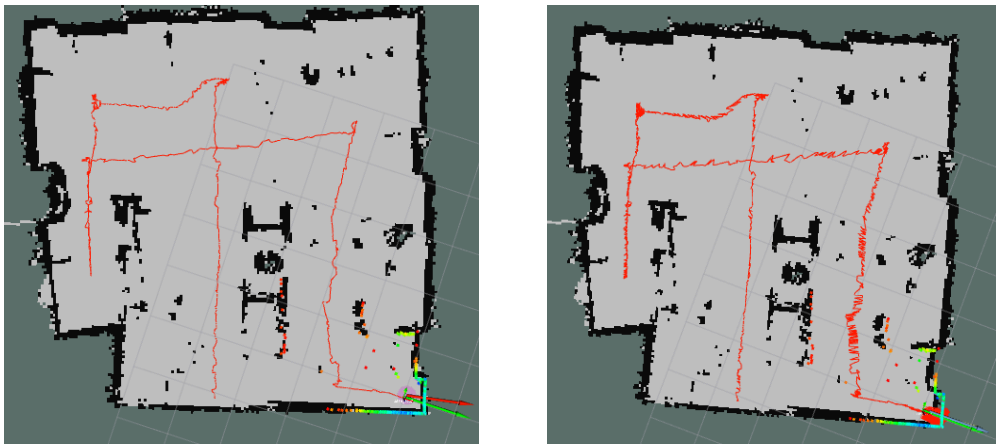

Figure 9 and 10: Estimated Paths Comparison in the Robotic LAB using EKF Localization (left) and AMCL (right).

3. Mini-Project 2: Path Planning

As previously stated, the objective of Mini-Project 2 was focused on the implementation of an RRT plugin for ROS with the majority of the code given by the professors. For this goal, two main ROS packages were used: **move_base** for path planning and **costmap_2d** for collision detection. A DWA planner was used as the local path planner while the developed RRT plugin was used as the global path planner [5], [6], [7]. Gazebo simulations were used to fine tune the parameters given by the move_base and costmap_2d packages, concluding in real life testing using a real TurtleBot3 Waffle Pi robot inside the Lab classroom. The implementation was saved inside the rrt_planner package (IST-rrt_planner folder)

3.1. Methodology and Experimental Setup

The project was divided into two phases:

Simulation Phase: Using the Gazebo TurtleBot3 simulation environment with the standard world here we finished the implementation of the RRT algorithm followed by a series of trials to refine our parameters.

Real-World Phase: After we were satisfied with the path planning capabilities of the RRT algorithm and the obstacle avoidance of the DWA planner we moved to testing on a real TurtleBot3 Waffle Pi robot inside the LAB classes to ascertain the real world capacities of our implementation. For this task, the robot 15 was used.

We started every experiment by launching rrt_turtlebot.launch inside the rrt_planner package to bring up all the required nodes for the path planning, namely AMCL for localization, map publishing, move_base, taken path and rviz.

After this, using rviz we selected the initial position using the 2D Pose Estimate tool to define the initial position followed by the use of teleop to make the robot make a full 360º turn to calibrate its location. Then, we used the rviz 2D Nav Goal tool to define a goal for the planner to decide on a path towards.

3.2. Parameter Adjustments and Optimization

The ROS packages **move_base** and **costmap_2d** offer a set of parameters to refine the pathing behavior, as such with the use of the Gazebo simulation environment the below parameters were defined to achieve better performance [6], [7].

3.2.1. DWA Local Planner Adjustments

Parameters are inside **ISt-rrt_planner/config/dwa_local_planner_params_waffle_pi.yaml**

xy_goal_tolerance: was increased to **0.2**, allowing the robot a bigger acceptable radius consider having reached the target, minimizing minor movements to try and obtain the perfect position

path_distance_bias: we found the paths for the global plan were more reliable so we decided to adjust it from 32 to **38.0** making the robot prefer sticking to the global path

occdist_scale: experimentally we found that lower values would cause the robot's local planner to get too close to obstacles and get stuck, as such from the original 0.02 we increased it to **0.7** resulting in better local planner obstacle avoidance performance

3.2.2. Costmap Configuration Adjustments

Parameters are inside **ISt-rrt_planner/config/costmap_common_params_waffle_pi.yaml**

inflation_radius: was reduced to **0.5m** (default was 1.0), allowing the robot to navigate tighter spaces, such as narrow passages in Gazebo, without collisions while also maintaining a wide enough safe zone around obstacles for the robot to avoid them.

cost_scaling_factor: Adjusted to **1.2** (default was 3.0). This reduction in the scaling factor increased the rate at which the cost around obstacles escalated, enabling a faster increase in cost to account for the smaller radius allowing for a circle of around 0.25m of diameter with high cost that the robot will avoid at all costs and the rest of the 0.5m is and area of lower cost the robot can move on but should avoid.

3.2.3. Move base

The parameters for the move_base configuration can be found in the file located at **lSt-rrt_planner/config/move_base_params.yaml.**

planner_frequency: This parameter was reduced to 0 as through testing we found that the replanning of paths were causing the local planner to sometimes become unsure what to follow causing it to get stuck so we chose to only allow the global planner to only generate 1 path at the start.

recovery_behaviors: We added a recovery behavior to replan the global path plan in case the robot gets stuck due to some dynamic obstacle or unknown at the time of the plan (possibly because it was behind a corner and it wasn't considered for the global costmap), this is to directly address the decision to set the Planner frequency to 0.

Every other parameter in the move base package was left as default.

3.3. RRT Implementation

When developing the custom RRT planner, several key functions were implemented namely getNearestNodeId, createNewNode, sampleRandomPoint, extendTree and inFreeSpace, all remaining functions were left as given to us.

The **extendTree** function simply computes the normalized direction vector between the nearest node and a random sample given by **sampleRandomPoint** that simply creates a random x and y important to mention that it would be possible to add goal bias here. extendTree also scales the vector by a given parameter and returns it. The **inFreeSpace** makes use of the costmap_2d package to compute a cost of a point and defines the threshold to be considered free space to be 200. This stricter criterion made the path planning more effective, allowing the RRT algorithm to define paths that traverse further away from obstacles. **getNearestNodeId** simply iterates through all the nodes in the tree and finds the closest node to the input point. **createNewNode** only appends a new node to the node array. Finally **planPath** is the function that integrates all these components and makes them useful by sampling a random point with **sampleRandomPoint,** then obtaining the nearest node to the point and extending the tree from that node in the direction of the point using **getNearestNodeId** and **extendTree** and only

creating a new node if the node is in free space with the use of **createNewNode** and **obstacleBetween (**which calls **inFreeSpace**), repeating this operation until the goal is reach or the number of iterations exceeded the max.

3.4. Experimental Results and Discussion

The final fine tuned results enable us to produce 2 videos, one for the Gazebo simulation [9] and the other for the real life robot [10]. We invite you to see these videos as the following discussion will focus heavily on the results and behaviors presented in the videos as well as reading the description as they also have a detailed discussion on the video.

Looking at the videos we can see that the RRT is working and effectively planning around the obstacles with a big enough distance to clear them without colliding with the obstacles although we recognize that in both experimentation environments it isn't taking the optimal path towards the goal, that is to say that while the path is not entering inside the red areas of the costmap due the values of inflation_radius and cost_scaling_factor in tandem with the modification made in the collision_detector.cpp namely reducing the maximum cost a grid position can have before it's considered an obstacle, as with an higher value it would very often pass way too close to the obstacle resulting in collision in spite of this the robot is taking longer paths when planning for a route this we believe is due to intrinsic qualities of RRT and some optimizations that weren't taken in this project for example its radom nature that prioritizes finding a path instead of finding a short path, this inherent randomness is also the reason why we choose to set the global planner frequency to 0 as the selected path would vary greatly causing a loss of performance on the local planner resulting in collision due to its indecision and some ways to mitigate this issues would be to introduce some bias towards the goal into the RRT implementation as well as path smoothing post processing steps could greatly increase its performance.

On the gazebo simulation we can see some artifacting on the global costmap, this was introduced in the setup of the environment when using the 2d Pose Estimate, in this specific example it causes no issues but it's possible that artifacts or missing data in the global costmap makes a goal impossible to reach by making a barrier of excessive cost for the RRT to consider free space, or in least concerning cases it makes the path longer.

Though we see that in the real life robot the quality of the global costmap greatly affects its performance, take the waypoint 1 (around 30s onward) the global costmap says its free but when getting close some obstacles appear making it getting dangerously close to them and possibly get stuck, this phenomenon and also the flicker of obstacles seen at around 1min we believe that is cause due to the chair's legs as they are slim they are harder to detect by the sensor but due to its higher path bias even thought occdist_scale was set pretty high the robot would still pass extremely close to the chairs. Yet again problems that could be fixed if we allowed for global path replaning, but this could cost local planning performance.

At around the 1:35 min mark the robot gets stuck because the goal was defined too close to an obstacle but this makes evident that the values used for obstacle inflation cause the robot to not

be able to navigate in tight spaces as although the chokepoint can fit tow robots we can see that the navigable area is very small being only the section painted in blue.

As such we recognize our approach to not be the most optimal, although it performs its task of reaching the goals without getting stuck there are some caveats that makes it not the best it could be, given more time we would further refine the parameters, implement the RRT optimizations previously mentioned and even test other path planning algorithms such as RRT* for example, to try and find a way to make the global planner able to replan without costing local planner performance, minimize the path length and increase the ability of navigating in tight spaces.

4. Conclusions

In sum we analyzed the effectiveness of EKF and AMCL [8] when applied to robot localization where we verified that the AMCL being a particle filter-based localization method presents the best results in general as the fact it can consider the belief about the robot's position with a set of particles creating a more stable and less noisy error while EKF being a probabilistic filter is more unstable although the covariance fluctuates wildly it remains between 2 values throughout all the experiments while AMCL needs an initial calibration at the start as its initial covariance is significantly higher but considering the error is always smaller than we can conclude if the robot has an opportunity to calibrate before having to execute its task AMCL is preferable, although AMCL is more computationally expensive than EKF, so for simpler and more static environments or when computational power is a constraint EKF could be preferable.

While AMCL and EKF are largely effective with minimal fine tuning of the params the same can't be said for the implemented work in mini project 2, where careful consideration of the application is crucial to define the parameters, for example if path length needs to be minimized to reduce operational costs RRT is not the best choice as its random approach focuses on finding a path and not the best, while if your robot is meant to operate in thigh spaces or needs to get close to walls you need the reduce inflation radius but it can't be so small the robot gets stuck in a obstacle as such, or if your environment is highly dynamic you must play around with the path, goal and obstacle distance biases to allow more control of the local planner, and even allow replanning with the global planning. As we saw, parameters that worked in Gazebo were causing some problems in the real world.

As such a deep understanding of the application as well as the options available to be used is paramount for attaining the best possible efficiency of your robot, either localization or navigation consideration of the target robot, environment and task is essential to match the best approaches.

For future work better consideration of parameters is essential and implementation of RRT optimizations could drastically increase performance of the given tasks or the use of better pathing algorithms. For localization integration of more sensors or fewer higher precision sensors could also improve the performance of localization, and could also improve pathing by allowing the creation of more accurate costmaps.

5. Bibliography

1. Robotis.“TurtleBot3 Waffle Pi.”: https://emanual.robotis.com/docs/en/platform/turtlebot3/features/
2. ROS Wiki. “robot_localization.” http://wiki.ros.org/robot_localization
3. ROS Documentation. “robot_localization API.” https://docs.ros.org/en/melodic/api/robot_localization/html/index.html
4. ROS Wiki. “Adaptive Monte Carlo Localization (AMCL).” http://wiki.ros.org/amcl
5. ROS Navigation Tutorials. “Writing A Global Path Planner As Plugin in ROS.” http://wiki.ros.org/navigation/Tutorials/Writing%20A%20Global%20Path%20Planner%20As%20Plugin%20in%20ROS
6. ROS Navigation Stack Documentation. “move_base.” http://wiki.ros.org/navigation/Tutorials/SendingSimpleGoals
7. ROS Wiki. “Costmap_2d.” http://wiki.ros.org/costmap_2d
8. Comparison of Particle Filter and Extended Kalman Filter Algorithms for Monitoring of Bioprocesses: https://www.sciencedirect.com/science/article/abs/pii/B978044463965350249X
9. Gazebo Simulation for mini project 2: https://youtu.be/LemBuY0D7wE
10. Real world pathing for mini project 2: https://youtu.be/rJzGoPSmc04