WORCESTER POLYTECHNIC INSTITUTE
ROBOTICS ENGINEERING PROGRAM

Lab 4: SLAM

SUBMITTED BY
Sam Appiah Kubi
Sakshi Gauro
Bryce McKinley

Date Submitted: December 14, 2023
Date Completed: December 7, 2023
Course Instructor: Professor Greg Lewin
Lab Section: RBE 3002 B'23

Abstract

The project was aimed to guide the robot through an unknown map using SLAM technology, creating a map of the environment. Diddy autonomously explored, mapped, and navigated back using GMapping for mapping and AMCL for localization within the map. The project demonstrated successful maze navigation and exploration using robotics techniques.

Introduction

The purpose of this project was to navigate and explore an unknown maze with a TurtleBot3 Burger, known as Diddy. Simultaneous Localization And Mapping, also known as SLAM was used for the success of this project. SLAM is a technological mapping method that is used to build a map of the environment while simultaneously localizing the robot.

To construct the control architecture, Robot Operating System Noetic (ROS) served as a foundation, offering a framework for the modular development of custom code and facilitating the stimulation of code based on data from real (or stimulated) sensors. ROS includes “pre-made” drivers for a number of common sensors and robot platforms as well as tools for visualization and debugging. In the development process, ROS nodes such as *odom, diddy_drive, map, scan, turtlebot3_slam_gmapping, robot_state_publisher, AMCL,* and *path_planner* were implemented. These nodes aided the communication process and enhanced the coordination of different parts of the system.

Python scripts, launch files, and functions were developed to implement the different phases of the final project. The implementation of the provided GMapping package was used to help Diddy build the map of the arena using LiDAR (Light Detection and Ranging). In Phase 1, Diddy drove around the map while avoiding obstacles. An optimal path was planned using the A* algorithm to navigate to unexplored areas, known as frontiers to expand the map. Phase 1 was completed after the completion of the mapping process. In Phase 2, Diddy navigated back to the starting location while avoiding obstacles. For Phase 3, the AMCL package was used for localization. Localization is the process of determining the location and the orientation of the robot. Diddy localized itself in the generated map. Subsequently, it created and followed a path to a point on the map.

Methodology:

When beginning this project, a rough outline was created of the challenges that needed to be overcome to complete each phase. This list was placed at the start of the main file. This provided a rough guideline of the steps needed to be taken for the completion of the project as it was divided into more manageable subtasks.

```
#### TODO #########################################################################
"""
PHASE 1:
    Map what is seen so far (SLAM)
        figure out the sensor_msg/LaserScan message
        give this message to slam_gmapping node -> returns an Occupancy Grid
    C-space
    find best fontier (Distance and size)
        edge detection
        dilation
        erosion
    drive optimal path to frontier while avoiding obsticles
    repeat until map is complete
    save map to file
    Test using Simulation

Phase 2:
    Find optimal path to start
    Drive path

Phase 3:
    Find robot location (Localaziation, Bayes)
    C-space
    Plot path
    Drive to location avoiding obs


**Figure out G-Mapping

"""
```

Figure 1: The team's initial breakdown of tasks to complete in each phase

The approach to this project split the code into two main classes, PathPlanner and DiddyDevilDriver. PathPlanner was the navigation class, which could be activated with a ROS service known as plan_path. This service would plot the best course for the robot to follow between two points, and then drive that path. Although PathPlanner was built and implemented over the course of previous labs, there were some considerable changes to improve its functionality during this project. DiddyDevilDriver was the planning class, which made the calculations to determine where the robot needed to go and then called the plan_path service to achieve those points. It was created specifically for the final project and could be run in three different configurations, one for each phase.

Creating a new launch file:

Launch files offered an effective way to run the simulation, navigation, and actuation nodes simultaneously while calling on them to reference one another. The *lab4_turtle.launch* file was the base project file and was structured after launch templates in the TurtleBot3 dependencies. Referencing the *turtlebot3_remote* and *gmapping* launch files from the TurtleBot3 dependency the master device was

connected to Diddy while initializing the use of SLAM and opening the customized RViz file. Allowing the use of these files without having to replicate their contents. When including files, “find” was used to identify their path where a ROS package held the desired file, whenever the desired file was not within a ROS package it was moved into the lab4 package. This allows the launch to be run on any PC barring whether they had the necessary ROS packages.

For Phases 2 and 3, the launch file uses the map_server node to navigate after Phase 1 generates it. It also references the AMCL launch file instead of GMapping.

Updating PathPlanner:

This project presented new challenges, so it was necessary to update PathPlanner. The most considerable change implemented was changing the drive function from turn-drive-turn to pure pursuit. Pure pursuit is a path-tracking algorithm that uses a “look ahead distance” to find intersections between a virtual circle surrounding the robot and lines connecting the nodes in the path. It consists of four main parts: finding the intersections, choosing the best intersection to drive to, finding the necessary linear and angular speeds needed to achieve that point, and using those speeds to drive. Tuning the pure pursuit values proved to be rather tedious, however, these values were finally chosen:

> look ahead distance= 0.2, driving threshold = 0.1, linear speed Kp = 0.3, angular speed Kp = 0.7

These values kept the robot relatively slow, but allowed for precise path following, which was imperative for fitting through some of the tight gaps found on the fields. Additionally, the robot would drive to the first point in the path and turn to face the second using proportional control before it began pure pursuit to make sure it didn’t ever have to turn large angles and potentially swing out into walls. Finally, because pure pursuit required a point on the path ahead of it to target, when the robot detected that the look-ahead had reached the second to last point in the path, proportional control was used to drive to the final point. While driving around the arena, each time a new cell was reached in the path, a new map was requested, c-space was updated, and checked to make sure the next point in the path and the final point was still in free-space.

In addition to the new driving method, some changes were made in the A* path planning algorithm to have two different modes. The first mode, known as “Fast” mode was the original implementation of A* but with the addition of a counter which would terminate the method and return an exception known as “A-star Time out” if more than 1000 cells were explored. This mode was used when checking if frontiers were reachable in Phase 1 and the time out allowed to tune how quickly the search would terminate if a frontier was unreachable. The second mode, known as the “Slow” mode, was used when plotting the path of the robot whenever the path_plan service was called. This mode would check the distance from each cell to the nearest c-space, and add the value multiplied by 15 to the priority of the cell in the queue, essentially prioritizing paths that drove far away from walls. The priority was calculated in the following way:

> priority = (cost to reach the cell) + 2*(distance from the cell to the goal) + 15 * (4 - distance to c-space, capped at 4)

This allowed the robot to drive in the middle of paths and take corners wide when possible but would also allow it to drive through narrow gaps if no other path was available.

Finally, changes were also made to the methods in PathPlanner to be static so that they could be accessed in DiddyDevilDriver. These included A*, calculating C-space, requesting the map, and sending speeds to the robot.

Phase 1: Mapping the Field

The first phase required the robot to navigate through an unknown field and map it using ROS's gmapping node. Although the mapping itself was done, it was necessary to identify where to go and navigate there while avoiding obstacles. This challenge was broken up into four steps:

Step 1: **Finding the reachable frontiers**

To know where the robot needed to drive on the map, frontiers between free space and unknown space need to be recognized. In the code, borders were defined as the x and y coordinates of any free cell that neighbored an unknown and frontiers as lists of connected borders. To find all of the places Diddy needed to explore, the map was updated to include the c-space as walls and created a master list to contain all of the frontiers, then iterated through all of the free cells in the map grid. If a free cell bordered an unknown, which was determined by looking at its eight neighbors, then it was marked as a border and its neighbors were checked again to see if it bordered a pre-established frontier. If it bordered a frontier, the cell's coordinates were added to that frontier's list. In the case where a cell found that it was bordered by two different frontiers, the second frontier would be added to the first and then deleted, essentially combining them into one large frontier. Finally, if the cell was not bordered by any pre-established frontiers then it became the start of a new frontier in the master list.

Once there was a complete list of frontiers, it was cleaned and updated to a list of frontiers the robot could reach. To do this, the A* function in "Fast" mode was used. If a path could be made through c-space to the first border in the frontier, that frontier was marked as being reachable and tagged it with the length of the path A* took to reach it. If there was not a path, then the next border was checked until either a path was found or the entire frontier was searched. This served to eliminate all of the frontiers that could appear outside of the map, or inside of objects, due to the LiDAR inconsistencies.

Step 2: Choosing the best point to navigate to

Once the list of reachable frontiers was formed, it was time to decide which frontier was worth visiting first, and which specific point the robot had to drive to to reach it. To find the best frontier, a value was assigned based on the distance that A* had to travel to find the frontier and its size. After some testing, the following value was finalized.

value = (A* path length) - (frontier length)*(0.8)

The centroid of the frontier with the lowest value as the best point. If this point was in free space, it was returned as the point Diddy should target, but if it was not all of the points surrounding it was searched for a valid free cell, originally in a 3x3 square. If no free cell was found, the search was expanded outward around the original point until one was free. Once the closest free cell to the original centroid was found, it was returned as the target. If there were no reachable frontiers, then the best target was returned as (-1, -1), indicating that mapping was complete.

Step 3: Driving to the frontier

After finding the target point to drive to, the path_plan service was called with the start at the robot's current location and the end at the target, and PathPlanner took over. A* found the most efficient path between the points in "Slow" mode, to avoid c-space as much as possible and the robot attempted to follow that path with pure pursuit, but with a few caveats. Since the robot was driving through a partially unknown map, it had the possibility of colliding with walls it found while traveling. To fix this, three exceptions were implemented that would force the robot to return to DiddyDevilDriver and recalculate its path. The first exception was if the next point in the path changed to be within the c-space, meaning that the robot was approaching a newly discovered wall. The second was if the final point in the path became part of the c-space, meaning that the frontier had become a wall. The final exception was thrown if the robot completed its navigation and reached the final point, indicating that the frontier had led to a new opening in the map. After an exception was thrown, DiddyDevilDriver would update the map and repeat steps 1-3 until there were no more reachable frontiers.

Step 4: Saving the map

Once the field was completely mapped, the OccupancyGrid of the map was saved to diddyMap.pgm and diddyMap.yaml with the map_saver command from ROS's map_server.

Phase 2: Returning to Home Base

Phase 2 consisted of navigating back to the starting location of the robot after the mapping of the field was complete. Many of the necessary functions to complete this phase were already implemented during the creation of Phase 1. The main challenge of creating Phase 2 was determining a method to efficiently switch between the phases. This issue was solved by adding a parameter to the launching of the DiddyDevilDriver.py file. This extra parameter was either a 1, 2, or 3 to indicate the phase and was collected with "rospy.myargv(argv=sys.argv[1])", and then passed into the class as a parameter. A simple check of the parameter in the run method allowed the node to change functionality. For example, to run the second phase, the terminal command used was:

```
$ rosrun lab4 diddy_devil_driver.py 2
```

Once the robot knew it was in Phase 2, navigating to the start consisted of calling path_planner with the end being the world point (0,0,0), the origin of the map. If the origin happened to be in the c-space, then the closest free point was chosen instead, using a search method similar to that of Phase 1, step 2.

Phase 3: Localizing

The third phase was localizing the robot after it was "kidnapped". This is to say that the robot was picked up and moved to a random location on the map, then it would need to find out where it was and navigate to a given point. To complete this task the ROS AMCL (Adaptive Monte Carlo Localization) method was used. To begin this phase a console command was used, as deceived in Phase 2. A unique launch file was designed for this phase that included the AMCL node and a map server which published the map saved in Phase 1. Upon running the launch file, AMCL generated a swarm of 3000 particles around the robot, indicating potential robot locations. The global_localization service was then used to spread the particles randomly across the free space of the field. To tell when the robot was localized, DiddyDevilDriver

subscribed to the “amcl_pose” topic, which contained messages of the type PoseWithCovarianceStamped. By taking the covariance matrix out of the message, the x variance (covariance[0]), y variance (covariance[7]), and theta covariance (covariance[35]) could be extracted. In order to localize the robot, it was instructed to spin around at a constant speed, which caused AMCL to calculate the probability of each particle receiving the data being received from the LiDAR based on its location. In each generation of particles, less likely ones were removed until the true location of the robot was found. The robot was considered localized if the x, y, and theta variances were all less than 0.002. Once the robot was localized, it waited for an input on the topic “/move_base_simple/goal”, which would indicate that an end goal was selected in RViz. Upon receiving a goal, the robot would save its current location as the start and the location of the goal as the end, then drive to its target using path_planner.

Results:

Phase 1 was completed in 2 minutes and 33 seconds. Phase 2 took 58 seconds and Phase 3 took 47 seconds. The entire demonstration was completed in 4 minutes 18 seconds, with zero collisions with walls. Rviz was used to visualize the environment and debug. The following are images from Rviz and real life.

Phase 1: Creating map using GMapping

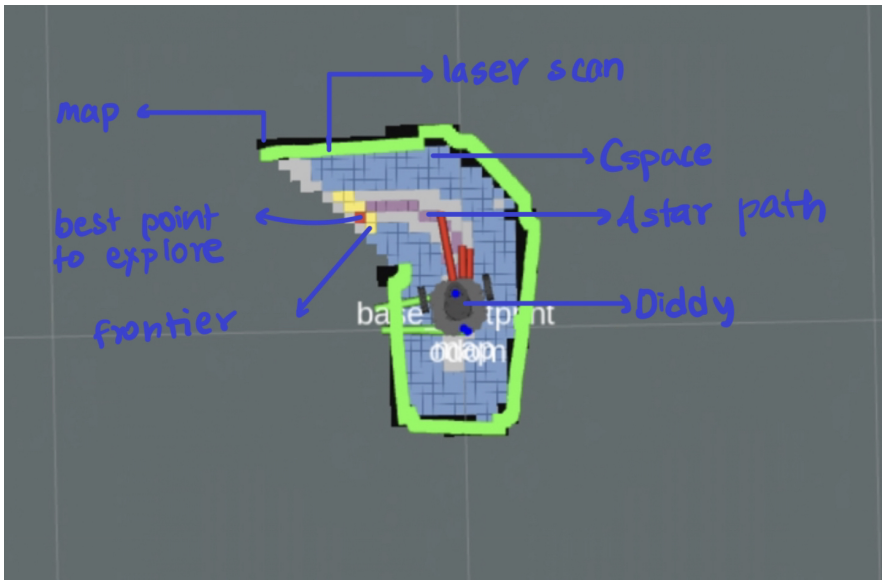

Figure 2a: Diddy starting the gmapping with labeled grid cells, home location (Rviz)

Figure 2b: Diddy starting the gmapping, home location (Real life)

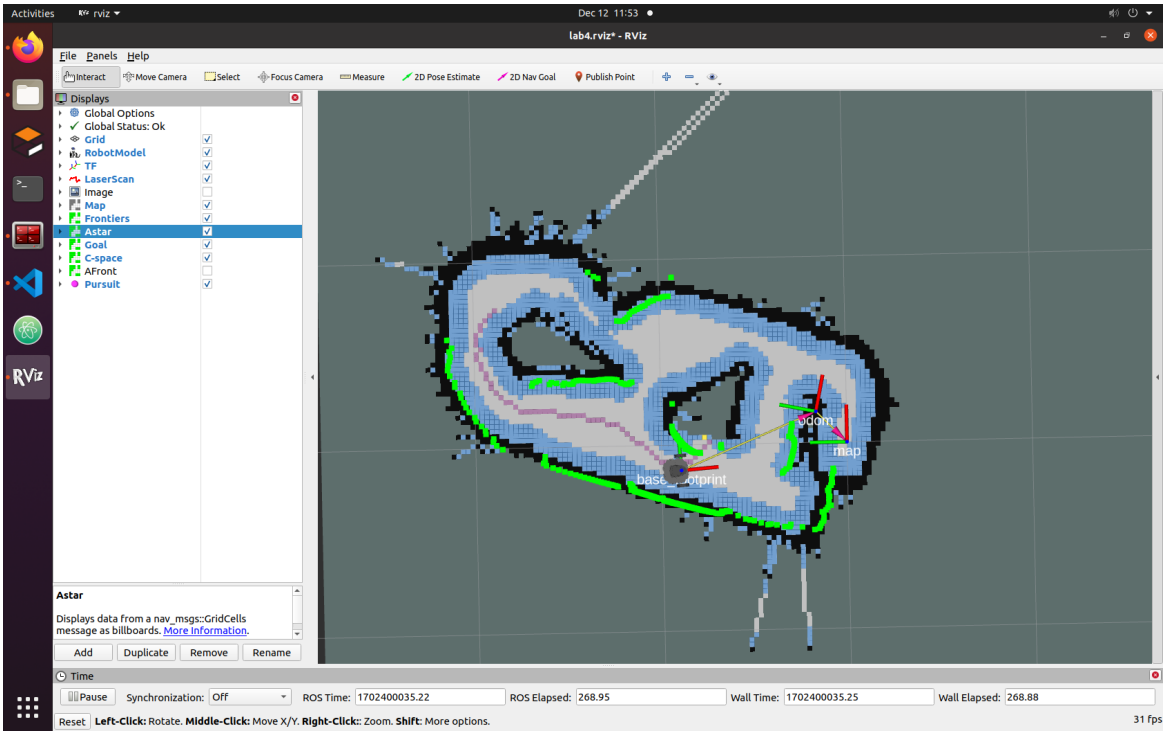

Figure 3a: Diddy exploring the last frontier (Rviz)

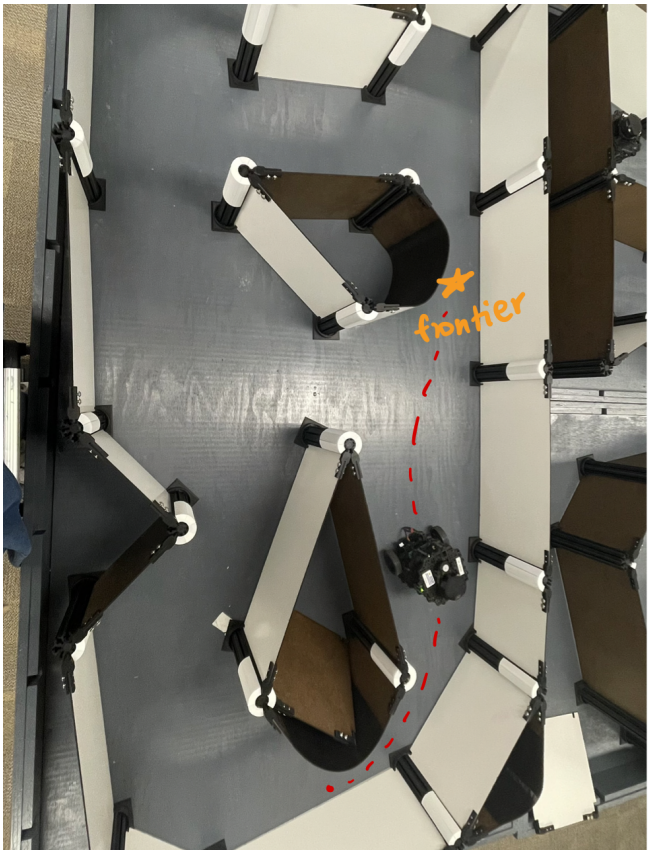

Figure 3b: Diddy exploring the last frontier (Real life)

Phase 2: Navigating to home while avoiding obstacles

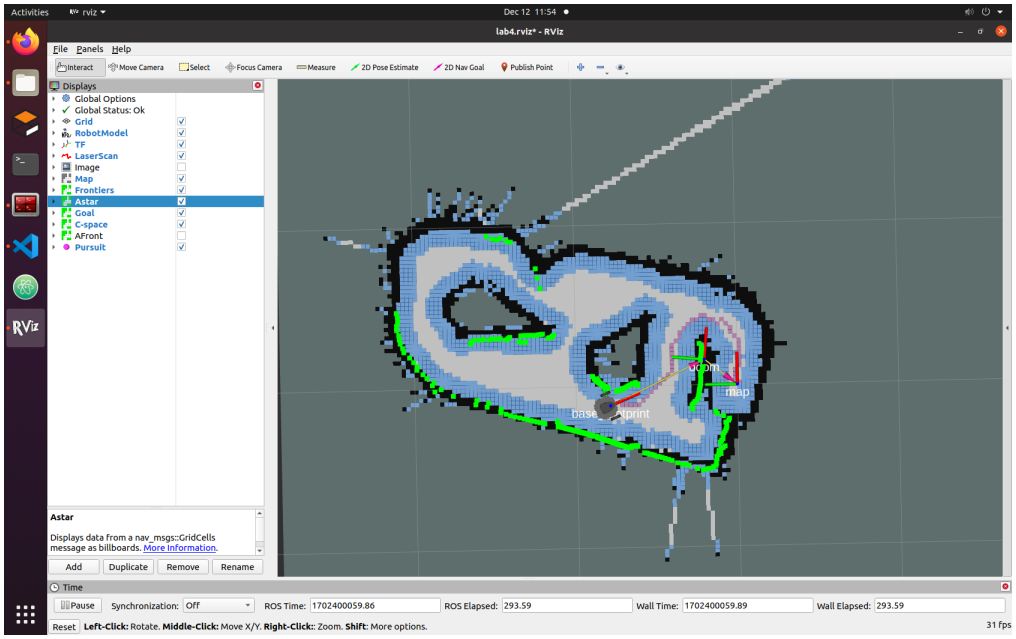

Figure 4a: Diddy navigating back to home (Rviz)

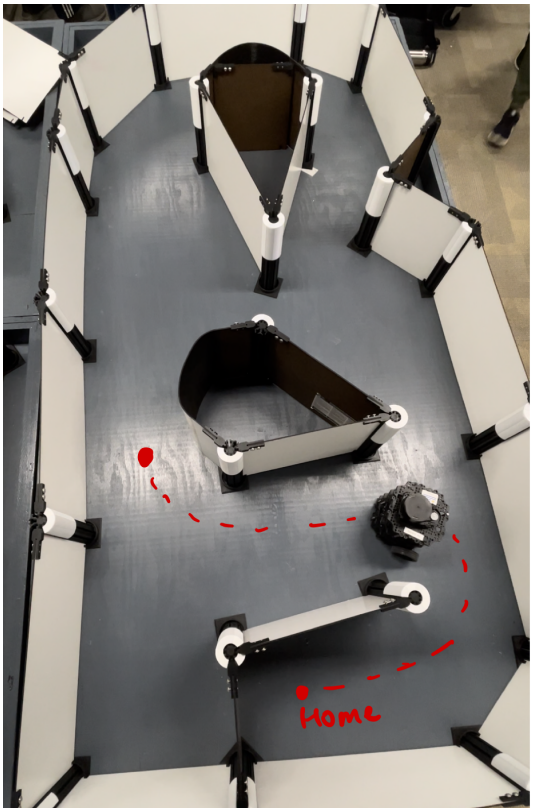

Figure 4b: Diddy navigating back to home (Real life)

Phase 3: Navigate to a specific goal position

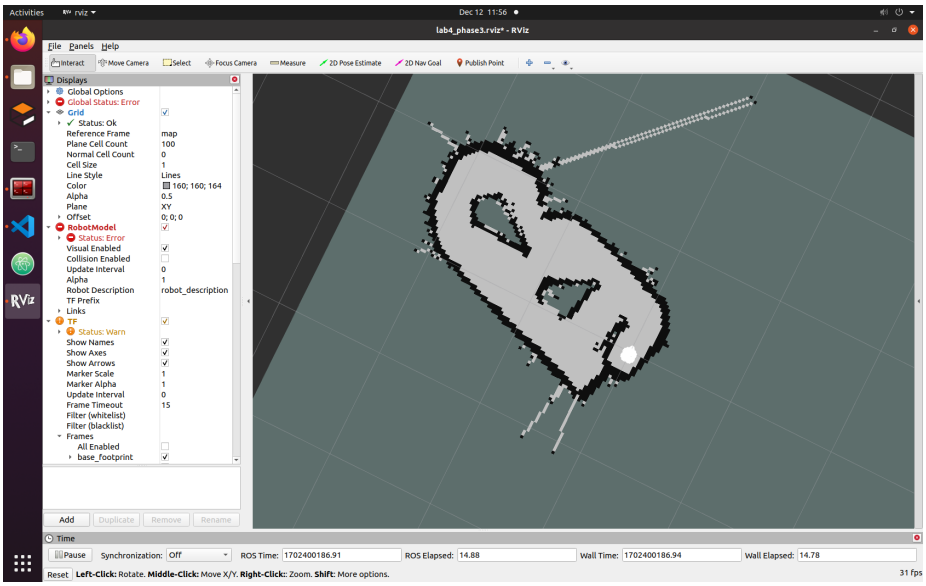

Figure 5: Map obtained from gmapping (Rviz)

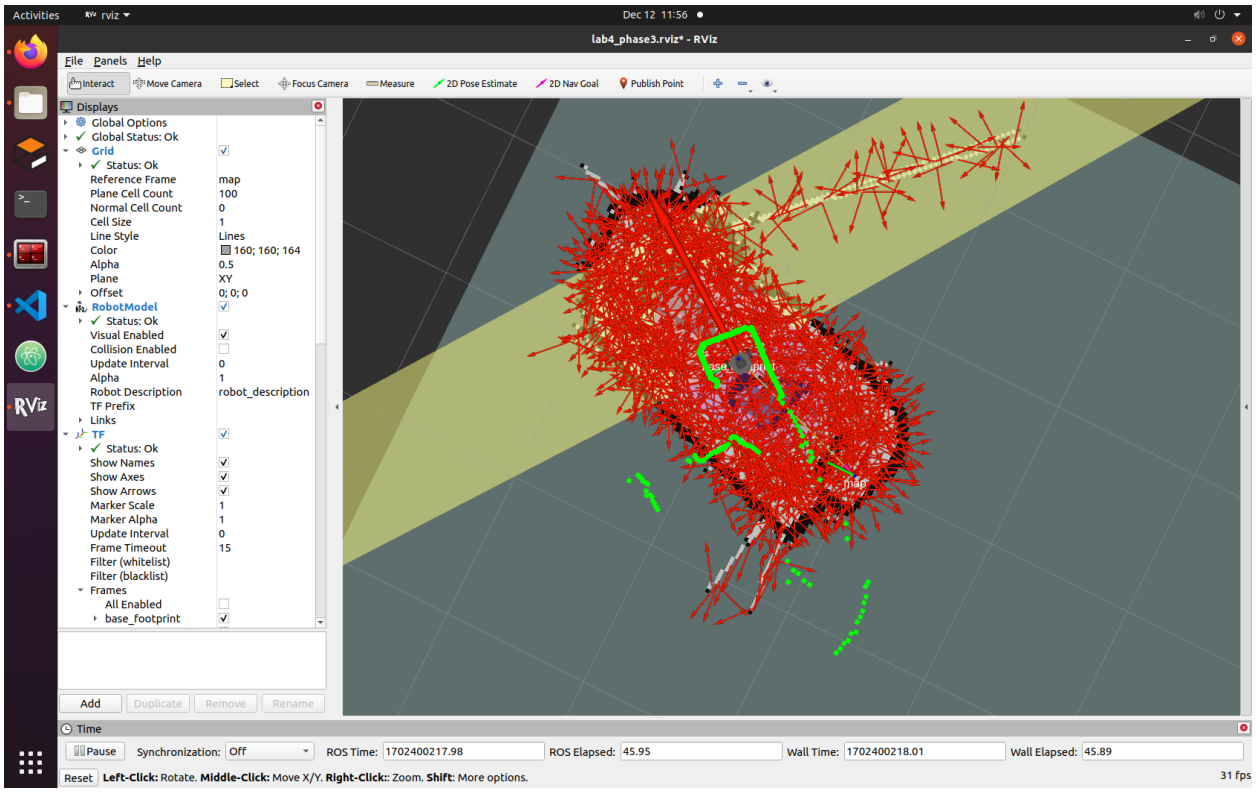

Figure 6: The particle cloud distributed all over the map (Rviz)

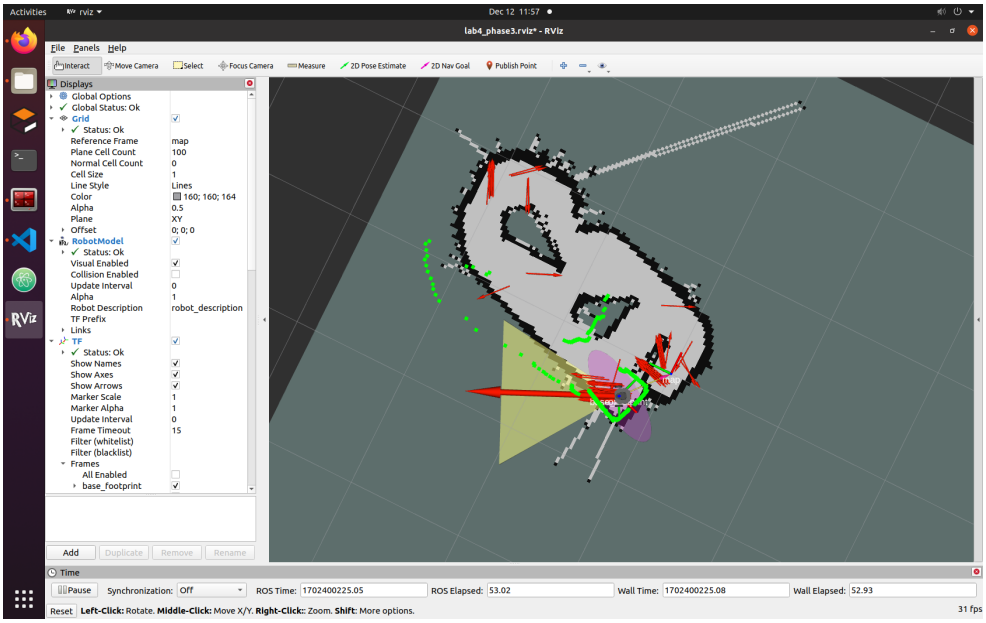

Figure 7a: Diddy localizing using AMCL (Rviz)

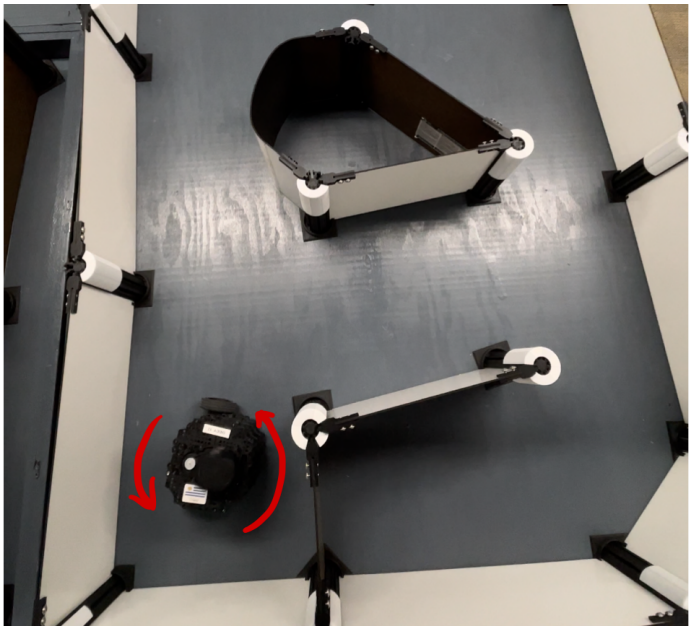

Figure 7b: Diddy localizing using AMCL (Real life)

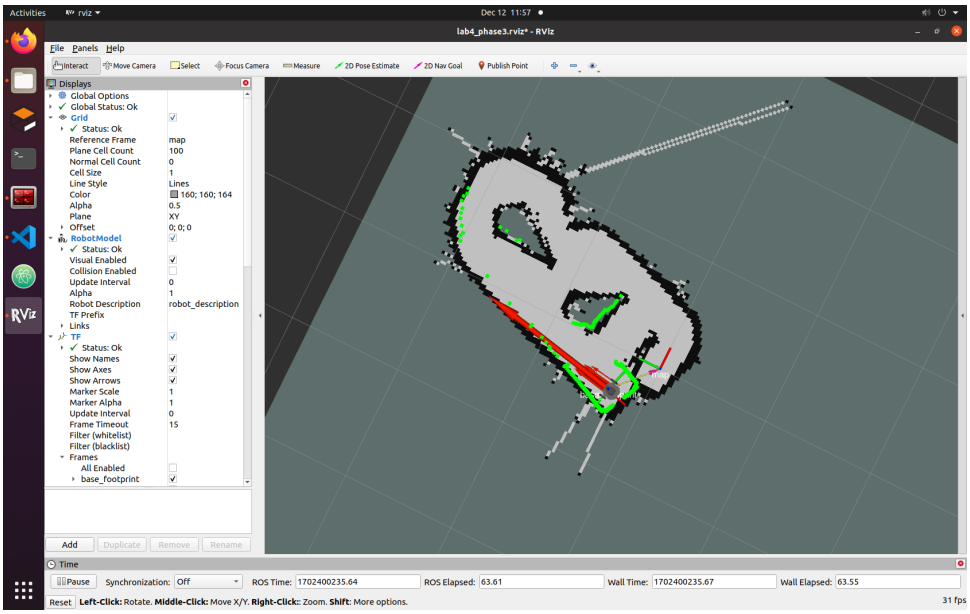

Figure 8: Diddy after done localizing (Rviz)

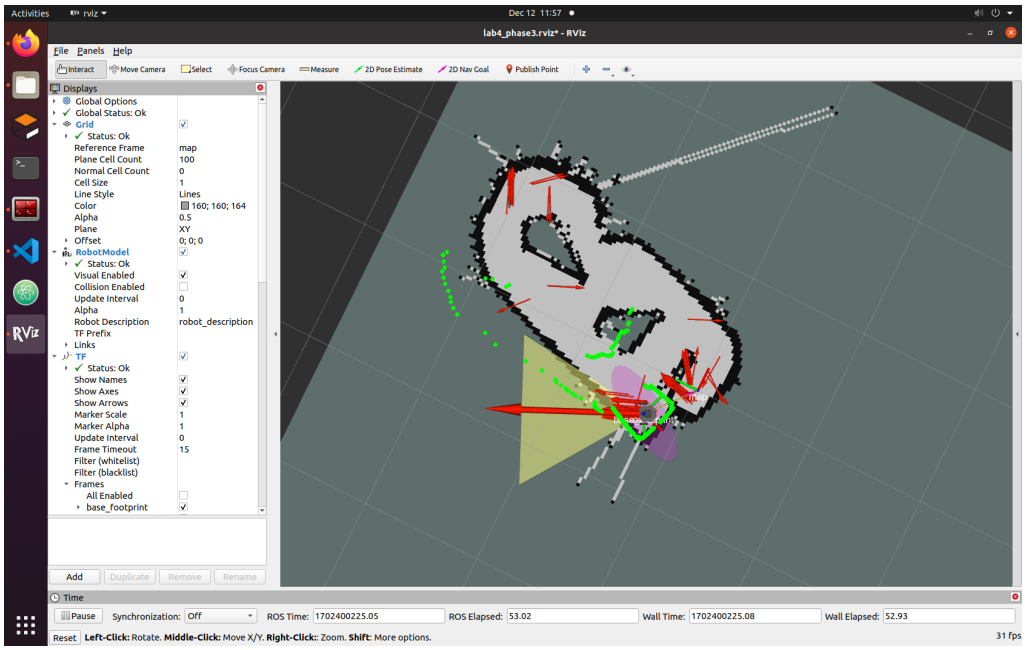

Figure 8b: Diddy failing to localize (Rviz)

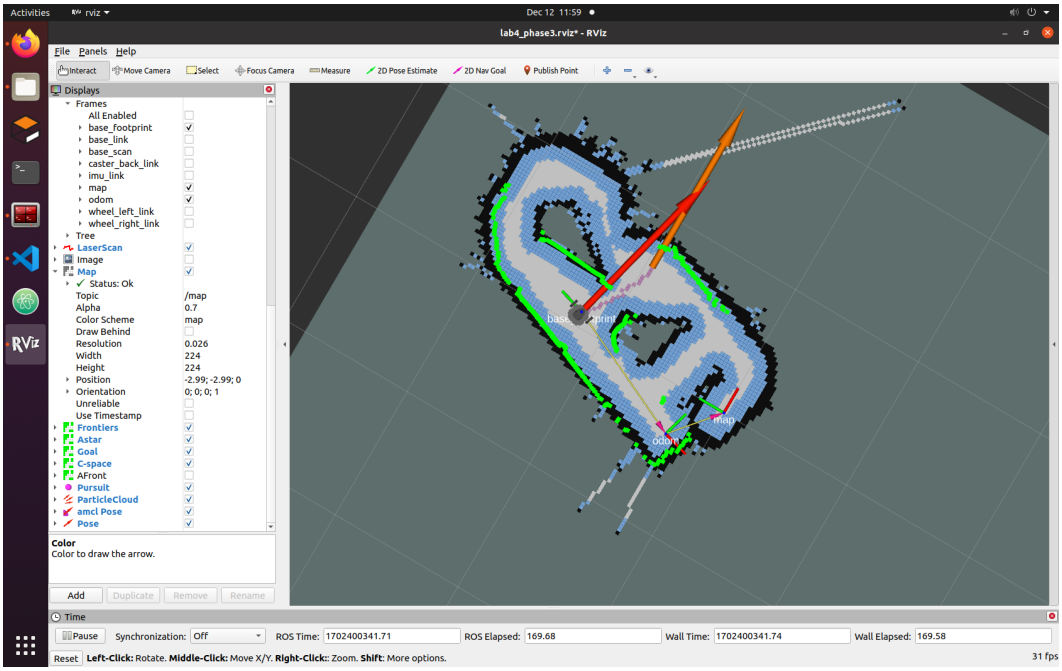

Figure 9a: Diddy navigating to the final goal location (Rviz)

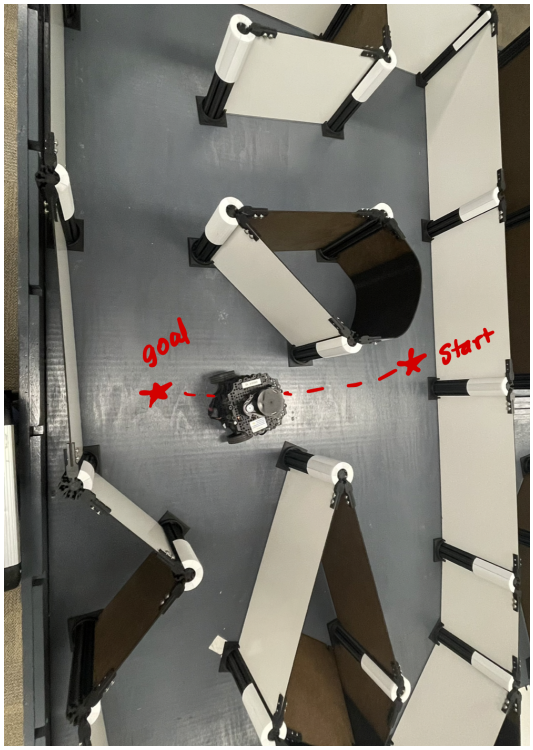

Figure 9b: Diddy navigating to the final goal location (Real life)

Discussion

Using SLAM localization, Diddy was tasked with navigating and mapping the presented maze-like structure with the quickest path possible. To verify that Diddy was able to effectively complete this task the project was given three main objectives, which were then completed through three phases of movement.

In the first phase, the assignment was to create a map of the environment using the GMapping particle filter with Diddy's lidar range data. As demonstrated in Figure 2a, a C-Space is made on the"PathPlanner" node with a padding of 4. This padding size was relative to the resolution of the grid cells produced through GMapping.

One main factor to account for was finding the right balance between the padding size of the C-Space and the resolution of the map. On one hand, a higher c-space would make sure Diddy would not crash into any obstacles during navigation, but making the c-space too high would result in the A* algorithm believing that there were no viable paths for Diddy to move through. Contrarily, a low resolution uses more processing power on the master controller when path planning, since path planning iterates through each map grid cell and lower resolutions mean more grid cells to map through. The balance was found at 4 in c-space padding and a resolution (or delta) of 0.026 when GMapping. While this worked for the presented mazes, spaces with higher densities of obstacles may require a lower resolution and thus a master device with more processing power.

Another issue that could carry on to affect Phase 2 was that the lidar was not entirely accurate. At the top of Figure 3a, the lidar returned that wall values and free space were found outside of the grid map. Since phases 2 and 3 navigate using the generated map, if the free space beyond the actual walls were too large, Diddy would attempt to navigate there. To account for this a check was done to ensure that the free space was viable using the A* algorithm.

In Phase 3: Localization, Diddy was effectively "kidnapped" and placed in a random position in the maze. Utilizing Adaptive Monte Carlo Localization (AMCL) and the map generated during navigation in Phase 1, Diddy was to navigate the maze toward a non-predefined point. The particle cloud in Figure 6 is a set of randomized positions within the previously generated map where Diddy believes it may be located. To localize, it rotates in place (Figure 7b) while increasing the certainly in its pose. This posed (get it? :]) a challenge as without a distinct set of landmarks while localizing, Diddy would not always be able to localize correctly, especially when specifying its heading. Consequently, Diddy would understand where it was located within the map, but fail to orient itself correctly.

Though this issue may seem small, improper orientations would lead to Diddy believing the map was skewed, with this error accumulating the farther it moved from its localized position. Figure 8b shows an example where there are 3 distinct positions within the map that map similarly to Diddy's true position, and while it was able to correctly find its x and y position, the offset orientation makes Diddy believe that the path in the northeast is open when in reality there is a wall there. When navigating to a given pose, Diddy would believe it was driving on the correct path while in reality driving into a wall. To further exemplify this issue, after encountering a wall Diddy would believe its position to be outside of the map's bounds, making it almost impossible to relocalize. To counter this the mapdata was updated at a higher rate, negatively affecting processing power while providing Diddy with more time to correct its localization while navigating the map.

Conclusion

An essential quality of robotics is the ability to sense and make decisions in the real world based on the environment. Using the Robot Operating System to tie algorithmic path planning with real-world actuation, the method of path planning with GMapping offers a look into the effectiveness of using particle filters to map spaces 2 dimensionally. While requiring a degree of processing power, the A* algorithm and the navigation implementation can create optimal paths within a grid space while allowing a degree of freedom when it comes to moving between waypoints. Though noise is always an issue when it comes to real-world navigation, the use of particle filters helps actively reduce the degree to which these errors may affect world navigation.

References

1. "Basic Pure Pursuit." *Basic Pure Pursuit - BLRS Wiki*, wiki.purduesigbots.com/software/control-algorithms/basic-pure-pursuit. Accessed 14 Dec. 2023.

2. *Nav_msgs MSG/SRV Documentation*, docs.ros.org/en/noetic/api/nav_msgs/html/index-msg.html. Accessed 14 Dec. 2023.

3. "Wiki." *Ros.Org*, wiki.ros.org/amcl. Accessed 14 Dec. 2023.

4. "Wiki." *Ros.Org*, wiki.ros.org/gmapping. Accessed 14 Dec. 2023.

5. *[PDF] Implementation of the Pure Pursuit Path Tracking Algorithm ...*, www.semanticscholar.org/paper/Implementation-of-the-Pure-Pursuit-Path-Tracking-Coulter/ee756e53b6a68cb2e7a2e5d537a3eff43d793d70. Accessed 14 Dec. 2023.

Appendix A:

- **GitHub release:** https://github.com/RBE300X-Lab/RBE3002_B24_Team12/releases/tag/lab4-team12

Appendix B: Authorship

Section	Author
Introduction	Sakshi
Methodology	Bryce
Results	Sakshi
Discussion	Sam
Conclusion	Sam

Appendix C: Relative Effort

Team Member	Percentage Effort
Sam Appiah Kubi	33.3
Sakshi Gauro	33.3
Bryce McKinley	33.3