Maze Solving Robot Project Report

Nathaniel Hanson, Ethan Marcello, and Charles Fortner

Abstract— **Group 8 successfully completed Tasks 1-4 as outlined by project B. We incorporated our combined knowledge of ROS, localization and mapping algorithms, control design, and software design to deliver a robust and repeatable solution to the problem statement. The team overcame significant challenges running two robots with the ROS navigation stack, such as the association of each named robot transform to the proper node. In spite of this, all of the project goals were satisfied, from mapping an unknown environment, to navigation, and coordination of two independent robots in the same environment.**

I. PROBLEM DEFINITION

Project B, the maze solver, has three major objectives. The first is to enter a previously unknown maze and generate a map of the environment. The second is to navigate through the maze and find the exit path of the maze from any internal location using our generated map. Finally, the robot must be able to search the maze to rescue another robot spawned inside the maze, and lead it out of the maze. These goals will be achieved through solving the following derivative problems: Control of differential drive robot motion, mapping of the environment, localization of the robot in the world, navigation of the robot to a desired location in the maze, communication between two robots via ROS topics, and enabling robots to follow each other.

II. TASK 1 - MAPPING OF THE MAZE BY ROBOT M

Mapping of the maze primarily relied on the Simultaneous Localization and Mapping (SLAM) algorithm with the ROS package *gmapping* to map the environment. SLAM was used in conjunction with a movement python script which provides robot M with the ability to both explore the entire maze without colliding with any obstacles, and terminate movement and save the map when the mapping is complete. The movement script used a proportional-integral-derivative (PID) controller to ensure that the robot stayed a safe distance away from obstacles at all times while moving. A "left-wall following" algorithm was implemented in the movement script to ensure that robot M explores the entire maze. Finally, the script was designed to terminate when the robot returns to the entrance of the maze by comparing its current location to its starting location.

Real-time visualization of the mapping results were done using *rviz* as shown in Fig. 3, and the *map-server* package was used to save the generated map as .pmg and .yaml files. The *map-server* is called in the movement script after the mapping is determined to be complete in order to save the fully mapped maze for use in later tasks.

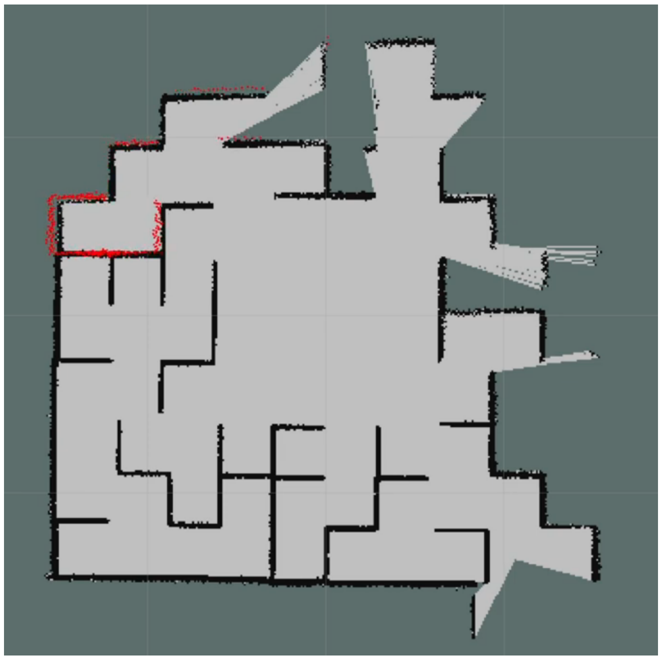

Fig. 1. Generation of a map by robot M of the maze environment, displayed in *rviz*. Red points represent the current laser scan data projected onto the world map.

The most challenging part of Task 1 was to command the robot to successfully navigate the entire maze while not crashing into any obstacles. In adhering to a "left-wall following" algorithm, outlined as a flowchart in Fig. 2, this problem was solved by cycling through a series of state conditions to command the robot in any situation it could face in the maze. These state conditions guide the robot to always follow the left wall. While the robot sees no wall on its left, it drives forward until it sees a wall in front of it. It then turns in place to the right, placing the wall on its left side. It will then continue to drive forward until an opening is discovered on its left, large enough to drive through. The robot will then drive through the left turn, and continue checking for a wall on the left. If there is no wall, then the robot completes the hairpin turn to the left, and drives forward. Finally, if the robot has a wall both on its left and in front, then it is in a corner and the robot makes a 90 degree turn to the right. With the movement algorithm complete, the final step was to ensure a clear and accurate generation of the map. Creation of an accurate map was delayed through difficulties faced in two major aspects. The first issue was regarding major errors in the map due to low resolution rendering. We were able to solve this problem by increasing the map resolution of the *gmapping* package, which gave us much more consistent results. The second issue was that the robot would see through walls when it drove too close to them. To solve this problem, we changed the minimum scan distance of the scanner so it could see nearby obstacles.

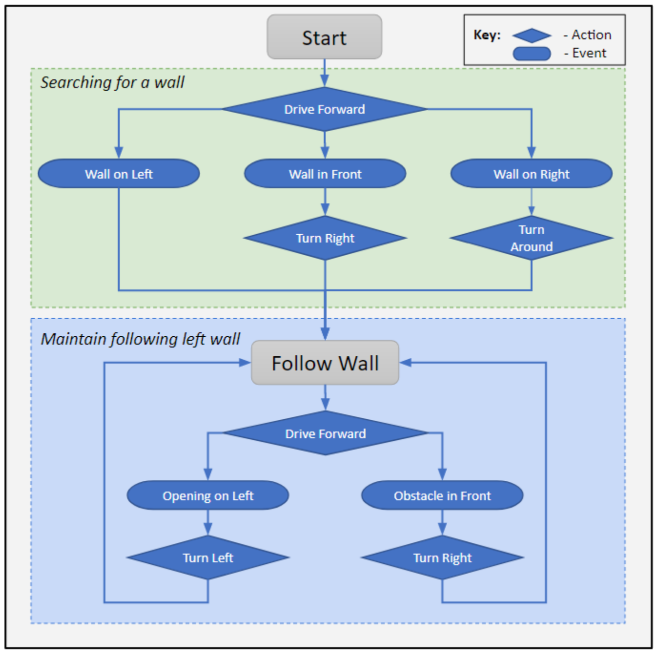

Fig. 2. The robot movement "left-wall following" algorithm used to explore the entire maze, displayed in a flowchart.

Task 1 was completed successfully, allowing the robot to completely map the maze while avoiding all obstacles, and generate this map in high-resolution for use in the additional tasks.

III. TASK 2 - NAVIGATION

The navigation task asks for robot M to navigate to a desired position in the maze, and return on the reverse path back to its starting position. Navigation with a known map was directly achieved using the Adaptive Monte Carlo Localization algorithm by incorporating the ROS package *amcl* for localization. In order to achieve localization of the robot in the world frame, AMCL uses a particle filter to track the robot pose in the known map environment. Initially, the robot's position in the world is highly uncertain, as evidenced by the wide spread of particles across the map. With such a degree of uncertainty, we found it was difficult for the robot to successfully plan paths if it does not know where it currently is. In this task before setting a goal, we rotate the robot in place around the z-axis, by approximately 2π. Performing this step allows the robot to view more of the map and condenses the probability field for a stronger estimation of robot position. When initializing the robot, it began listening the */goal* topic, which listened for a custom defined message containing $\{x, y, \theta\}$. Once the robot received a message, it leveraged the global path planner provided by the ROS navigation stack. If the point received was valid in the global frame, the robot generated a global plan using the A* algorithm, considering the lethal portions of the costmap. With a global map in hand, the robot then generated a local odometry plan, which dictated how the robot should behave until the next local plan is generated. In practice, the robot will often deviate from these local plans. The ROS navigation stack takes as input a world position passed as a Tuple, as well as the known map generated in Task 1, and is able to calculate and execute the shortest path to that point while avoiding any obstacles. In order to ensure the robot returned on the reverse path, as the robot was traveling to the goal point it was programmed to save readings of its position from the odometry at regular intervals. These points were then fed back to the navigation stack in reverse order to get the robot to follow its reverse path back to the exit of the maze.

The largest challenge of this portion of the project was configuring the various launch files dictating the parameters of AMCL and the Move Base planner. The Fira Maze environment was designed to be challenging for a Turtlebot Burger to navigate. Thus the default parameters for generating the costmap, and path plan were inaequate. Using the default values of the inflation radius and gradient caused large portions of the maze to be unnavigable. The allowable margin of error in navigation was extremely small; therefore, minimizing oometry drift was of the utmost importance. The robot's speed linear an rotational speeds were limited to 10% of it's normal maximum. Additionally, the local path planner was tuned to adhere more strictly to the global path plan, which with tuning we were able to draw cleanly down the middle of corridors. This section provided an excellent opportunity to leverage a methodical debugging strategy. As the launch files provided nearly 50 variables that needed to be changed in concert to positively affect the robot's performance.

Task 2 was completed successfully, enabling the robot to navigate freely to any given point inside the world map, and return on the reverse path that it traveled to get to the goal. Because of the tightness of the corridors in the maze, and the relatively small inflation radius, the robot occasionally became stuck on corners, but tuning the planner to have a greater degree of patience resulted in the robot being able to successfully navigate through the maze.

IV. TASK 3 - RESCUE OPERATION

The rescue operation problem required the coordination of two robots working together to navigate the maze. The first robot, robot M, had the map of the maze generated in Task 1 and was tasked with searching the maze for the second robot, robot S, which was spawned somewhere in the maze. While robot M was driving through the maze, robot S searched for robot M using its laser scanner, but does not move. Once robot S detected robot M, robot S sent a message to robot M using the ROS topic *'/comm'*. From this message, robot S determined how far the robot M was from it. Robot S continued to inform robot M of its relative position until the two were nearly adjacent. At this point the robot sent a stop message to robot M to indicate it was close enough to follow it out of the maze. Robot M then engaged its global planner to find the shortest path out of the maze. As it navigated, Robot M also transmitted breadcrumbs in the form of odometry positions. Using the *gmapping* stack, the robot was able to successfully transform these points into positional goals. The frequency of these positional message had to be fine tuned along with the speed at which the robot navigated to ensure the robot would not deviate of course as it had a n incomplete map of the world.

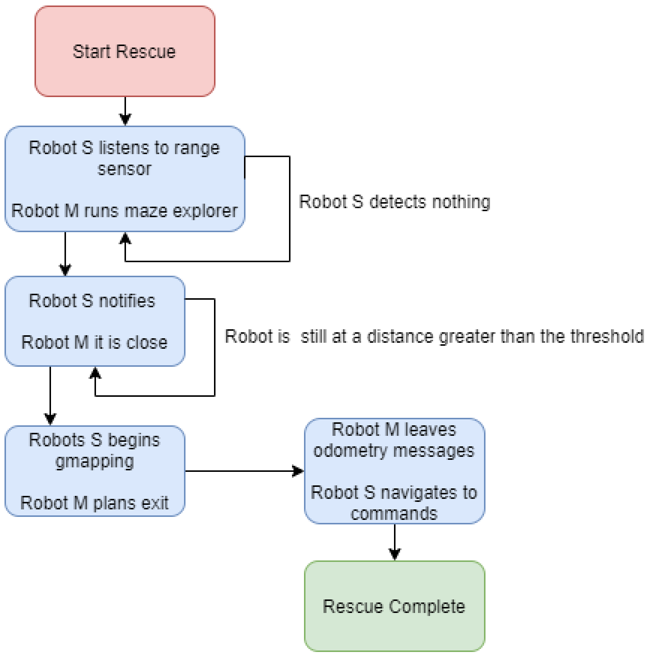
Start Rescue
Robot S listens to range sensor
Robot M runs maze explorer
Robot S detects nothing
Robot S notifies
Robot M it is close
Robot is still at a distance greater than the threshold
Robots S begins gmapping
Robot M plans exit
Robot M leaves odometry messages
Robot S navigates to commands
Rescue Complete

Fig. 3. Robot state diagram for rescue operation.

The rescue operation task presented many challenges in ROS, as using two different robots requires them to have different namespaces in ROS in order to distinguish between the two when sending and receiving data in simulation between different nodes. The ROS navigation stack is intended to work with only one robot, and not multiple, so it is not configured to access the robot namespaces. This makes associating different ROS nodes–e.g. *gmapping*–with the robot namespace a rather difficult task. In addition to this, many of the transforms that define the linkages between two different spacial orientations change in name when a robot is associated with a certain namespace. This causes the ROS navigation stack to look for transforms that don't exist, since these nodes don't expect to have to search for a specific namespace when looking for the necessary transforms.

V. Task 4 - Multi-Robot Mapping

For our creativity task, we sought to expedite the long process of generating a map of the environment by using multiple robots to generate a singular world map. We leveraged an existing ROS package, Multi Robot Map Merge, to accomplish this task. The package located robot name spaces and listened to their individual map topics. To demonstrate the efficacy of this task, we initialized two turtlebots, each with their own *gmapping* stack. Each robot independently generated a map which was read by the Map Merge node. This node under the hood leverages the C++ implementation of OpenCV to detect and correlate features in the map. When the node received both nodes, it first detects features in both images, given by corners. With known features, the package then leveraged the AKAZE Algorithm (Accelerated-KAZE) to detect nonlinear transforms between features from one map to the next [1]. These correlations were as a series of vectors denoting how the image should be rotated best correlate the features. Once the maps are rotated and correlated, a new image was created and published on the */map* topic for viewing in *rviz*.

VI. Individual Contributions

A. Nathaniel Hanson

Serves as team lead, coordinating goals and providing initial project setup. Established GitHub repository and workflow for maintaining usable code amongst team members. Developing path planning strategy for robot, as an derivative of the SLAM map. Has helped with architecting ROS solutions using existing TurtleBot navigation stack as well as providing general software engineering advice and debugging assistance.

B. Ethan Marcello

Majority of work has been in contributing to experimentation with the Turtlebot3 ROS launch files. Through testing has been able to run SLAM in conjunction with a basic movement script while viewing with *rviz*. Developed mapping capability of determining when the robot has finished mapping the maze, and terminating movement when this condition has been met. Completed initial work in spawning two robots in gazebo and running a movement script on one robot independently. Also contributed in discussion of controller development, and made significant contributions in writing both reports, and hosting three team meetings.

C. Charles Fortner

Primarily worked on the controllers for Part 1 and the follower robot in Part 3. Developed the maze explorer algorithm to map out the complete maze. Investigated and solved issues with the map generation in Part 1, tweaking the SLAM parameters to make a more accurate map.

VII. Results

The team successfully completed Tasks 1-4 as outlined by the project. We incorporated our combined knowledge of ROS, localization and mapping algorithms, control design, and software design to deliver a robust and repeatable solution to the problem statement. All of the project goals were satisfied, from mapping an unknown environment, to navigation, and coordination of two independent robots in the same environment.

VIII. Appendix: Discussion on Debugging ROS

Nota Bene: This section is provided as an explanation for potential shortcomings in the project after consultation with the professor, and as a detailed guide on how to debug issues within the ROS.

Our progress on this project was severely impacted by some deprecated features in the core ROS library. As we were building our environment to develop the maze solver, we made the technical decision to utilize Python3, Ubuntu 20.04 Focal Fossa, and ROS noetic. This technology stack provided us with the latest capabilities ROS had to offer; additionally, we could leverage some of the builtin syntactical features of python3 to make the code more efficient. However, ROS noetic is still in a state of relative infancy, with approximately 6 months of time as a public release. In making the jump from Melodic to Neotic, we also discovered that several core packages in ROS deprecated some of their parameters silently. For example, the *robot state publisher* library which is used to publish the joint states of the robot to the *tf2*, the transform library, normally has the option for a *tf prefix* which can be used to clearly delineate the namespace for multiple robots. In ROS noetic this parameter has been silently deprecated. There still is no clear work around for the depcreation as many libraries still presume the presence of a tf prefix for working in multi-robotic environments. We were only able to discover the change in the code after coming across a similar question on ROS Answers. Once we had identified the issue, our procedure to develop a work around was as follows:

1) **Using the tf package, visually inspect the transform trees**
   In this step, we first noticed that only a single robot state publisher was present, even though two had been initialized in our launch file.
2) **Reread the documentation on the ROS Wiki page**
   When in doubt there is likely some defect on the end user, as community maintained libraries tend to be stable, and have bugs fixed quickly. A reread of the documentation confirmed the tf prefix parameter should still be valid for the ROS noetic distribution
3) **Search for a similar issue on Google**
   ROS has a wide area of community support, especially when working with Turtlebots. Issues are unlikely to be siloed; other are likely to have experienced similar setbacks. Leveraging knowledge on ROS answers can also be fruitful.
4) **Use ROS WTF**
   This package can help users inspect potential environment issue. It is particularly useful in debugging launch files.
5) **Listening to ROS topics**
   Using the command line to listen to topics published by various nodes can yield information about what is or is not present in the messages. Listing the information on a given topic can tell you who which nodes are subscribed to a current topic, and which ones are publishing to it. This can inform you or erroneous connections or resolution of namespace.
6) **Check the Git Repo**
   If you are having issues with a particular package, it may be useful to look at the GitHub page where the source code is posted. Each repository has an ”issues” tabs where users can report various bugs or suggest improvements. This is how we ultimately discovered the namespace issue.
7) **Check for Unmerged Pull Requests**
   If there is a well documented issue, someone is likely working on a branch to fix that issue. One penitential resolution might be to pull and build that particular branch. However, this runs the risk of being unstable. Some repositories are connected to a continuous integration pipeline which can give you a better grasp of how reliable a branch is likely to be. If there are no other pull requests opened, one might consider reverting the repository to a previous commit when things were functioning as expected.
8) **Build Package From Source**
   Once a target branch is identified, the source code can be used to rebuild a specific version of the package in the catkin workspace. Once the source code is pulled, the target can be built using
```
rosdep init
rosdep update
catkin_make_isolated
```
9) **Try a Different Version of ROS**
   Ultimately this is the solution we settled on. Fixing the dependency sprawl from this singular package proved too costly. We had to revert to using ROS Melodic and Ubuntu 18.04. This step is definitely a last resort as it involves reverting an operating system. Ubuntu 20.04 cannot run any other version of ROS without extensive reworking.

Like all forms of software engineering, debugging is an art form that requires rigor if it is to be successful. Following a process where things are tried and recorded is likely to have the highest chance of success.

References

[1] R. Siegwart, I. Nourbakhsh, D. Scaramuzza, Introduction to Autonomous Mobile Robotics, Cambridge: MIT Press, 2011.

[2] Pablo F Alcantarilla, Jesús Nuevo, and Adrien Bartoli. Fast explicit diffusion for accelerated features in nonlinear scale spaces. Trans. Pattern Anal. Machine Intell, 34(7):1281–1298, 2011.

[3] S. Thrun, W. Burgard, D. Fox. Probabilistic Robotics, Cambridge: MIT Press, 2005.