Autonomous Service Robot – A Simulation

Khoa Dang Do
Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC, USA
kddo@ncsu.edu

***Abstract*—This paper serves as a final report for the Spring 2023 capstone project in ECE 591 – Software Engineering for Robotics. We conduct a full software simulation of an autonomous service robot that is capable of autonomously navigating an environment while avoiding obstacles for a service delivery task using Simultaneous Localization and Mapping (SLAM) [1] and other available packages in the Robot Operating System (ROS) [2]. The purpose of this project is to gain a better understanding of ROS and various packages available within the ROS ecosystem, so that we can utilize and integrate them together into the aforementioned software simulation before and/or without deploying the software stack on a physical robot for live testing and performance measurements, which can potentially detect bugs early and reduce the cost of fixing them. In the simulation, ROS GMapping [3] and Adaptive Monte Carlo Localization (AMCL) [4] are deployed on a TurtleBot2 [5] for SLAM and autonomous navigation and obstacle avoidance respectively. The robot is simulated in a customized Gazebo world where it picks up a virtual object at a known location and drops off the same object at a designated destination. Testing shows we have successfully put different software components together to produce a generally working simulation in Gazebo and visualized the entire process in RViz. However, there are certain scenarios where we identify potential bugs that can potentially lead to software failures during live testing, and we will discuss these later in section V of this paper.**

Keywords—SLAM, ROS, ROS navigation stack, autonomous navigation, obstacle avoidance, robotics simulation, robotics testing, robotics software bugs, robotics testing, robotics software engineering, TurtleBot, service robot, mobile robots, TurtleBot, Gazebo, RViz, GMapping, Adaptive Monte Carlo Localization.

I. INTRODUCTION

Designing and building a robot from ground up is a non-trivial engineering problem, especially with modern robots which have increased complexity, that require many people with different areas of expertise and long hours to work together towards the common goal. However, the problem is that we do not want to spend expensive resources – humans, time, and materials on building robots that completely fail in the end. Fortunately, we can bring such systems online much faster and launch production with fewer errors by using simulation. Simulation plays a key role in robotics because it permits experimentation that would be expensive and/or time-consuming if it had to be conducted on actual robots. It allows engineers to try ideas and construct manufacturing scenarios in a dynamic virtual environment, collecting virtual response data that accurately represents the physical responses of the control system [7]. Modern ROS-based robots have many other advantages over conventional automation engineering approaches beside simulation because ROS provides existing libraries and packages for autonomous navigation, obstacle avoidance, communication, and many more for code reuse and easy integration instead of developing software components and applications from scratch. Overall, there are many positives motivating us to simulate a ROS-based robot as the main work of this project but the following five are key:

- Exploring, learning, and integrating ROS and its existing libraries/packages/tools with our own contributions into a successful robotics software simulation.
- Gaining a better understanding of the above.
- Saving time from developing software from scratch, thus having a reasonable project scope for a team of one person.
- Having more freedom for different experiments without the robot's physical hardware constraints.
- Potentially identifying software errors in early stages of the robot development before finalizing the robot's hardware and software design and bringing the application online on a physical robot, thus reducing the cost of fixing issues occurring during live testing.

As previously mentioned, we would like to explore, learn, and gain a better understanding of ROS while completing the project; therefore, we want to combine existing software components available in ROS with our own contributions to produce and demonstrate our own customized capstone project. The main contributions in the project that we present this paper are as follows:

- Designing our own Gazebo 3D world (environment) where the TurtleBot2 operates in.
- Using *pgm_map_creator* ROS package [8] to generate the 2D map from the 3D world for robot navigation.
- Creating a ROS package for the virtual object to be picked up and dropped off.
- Creating a ROS package for the robot's service delivery task.
- Writing shell scripts to launch and run multiple ROS nodes at once for the simulation.
- Running the simulation under different scenarios to detect and discuss potential bugs.
- Making our work open to the community for reference and/or further case study. The project as well as this paper are available on our GitHub[1].

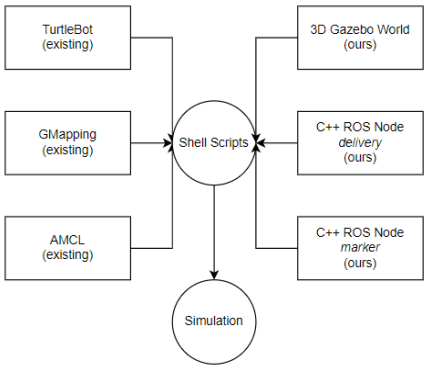
The rest of the paper is structured as follows. Section II briefly describes our project on a high level. Section III introduces the existing work and concepts that we believe are important, helpful, and closely related to our work. For this section, we aim for an entry to intermediate-level introduction of relevant studies and concepts to the readers, avoiding being "too" theoretically technical on terminologies that might confuse the readers, so that they can easily convey our work and conduct further studies on their own. Section IV

[1]https://github.ncsu.edu/kddo/course-project

discusses our starting assumptions, technicalities in approaching and completing the software simulation. We also address and discuss problems faced and possible solutions in this section. Section V discusses the experiments and results obtained. Section VI concludes, including some self-reflections throughout the process of completing the capstone project. Section VII discusses and provides directions for our future work.

II. About the Capstone Project

This capstone project is entirely software-based, meaning no physical robots or involvement of hardware-related to the robots for live testing and performance measurement purposes. We simulate a ROS-based autonomous service robot that is capable of SLAM. The robot navigates a given environment and avoids obstacles while performing a service delivery task. To mimic the real-life pick-up and drop-off task, we make the robot go to a known location where the virtual object first appears, pauses for five seconds which implies the pick-up operation. Then, the robot goes to and stops at the known destination where the virtual object reappears. We run the simulation under different scenarios to observe possible bugs that can potentially cause failures in live testing on a physical robot. We aim to utilize the existing libraries and packages available in ROS, integrate them with whatever we need to accomplish on our end to produce the final working capstone project, avoiding developing everything from scratch.



III. Related Work

A. Relevant Studies

This particular type of project is extremely popular. A simple Google search "TurtleBot SLAM simulation" will show many similar tutorials or past work similar to ours. However, few to none of these conveyed the same experiments, observations, and discussions that we would carry out in this paper. The authors in their publication [9] simply executed the examples from the TurtleBot ROS packages and had no other derived experiments or other discussions. The authors in [10] used a customized Gazebo environment and Astra depth camera instead of the default laser sensor in their simulation. However, they did not experiment with dynamic obstacle avoidance nor convey potential software bugs at all. The conducted study in [6] briefly showed that simulation could effectively and automatically discover bugs in a variety of robot application domains and remained discussing mainly on addressing challenges of using robotics simulators for testing. [23] and [24] also concluded that simulation could be used to trigger and detect bugs. In general, the majority of online tutorials, blogs, and past literary work that we find closely related to our project are either primitive, static obstacle avoidance experiments, and/or exact examples that come with the TurtleBot ROS package. They did not convey their studies nor discuss/analyze potential bugs in the same particular simulation that we would.

B. Relevant Concepts

1) Robotic Operating System (ROS)

ROS is an open-source framework that helps researchers and developers build and reuse code between robotics applications. It is supported by a global community of engineers, developers, and hobbyists who contribute to making robots better, more accessible, and available to everyone [2]. It is a middleware ecosystem designed to maximize collaborations between experts in different domains to come together and build robots as well as to maximize code reuse, implemented on the pub-sub architecture.

2) ROS Nodes, Packages, and Messages

The concept of ROS nodes is more involved. They are basically processes that perform computations, being executable programs running inside applications [17]. One or multiple nodes are put in the same or different ROS packages [18] depending on the application. Nodes communicate with one another by subscribing/publishing to ROS topics [19] and getting/sending data (so-called messages [20]) from/to respectively. We will not go into the details of these concepts since this is beyond our scope, but the readers are encouraged to conduct their own reading and studying following some of the references we provide.

3) ROS Markers

Markers display, a type of ROS node, allows the programmatic addition of various primitive shapes to the 3D view in, interacting with other objects through ROS messages without actually and physically interacting with other objects in the simulation. The main purpose of markers is for visualization in RViz [16].

4) ROS Navigation Stack

As the name suggests, ROS Navigation Stack takes in information from odometry, sensor streams, and a goal pose and outputs safe velocity commands that are sent to a mobile base for 2D autonomous navigation and obstacle avoidance task [13]. It is where the SLAM magic happens. The stack includes GMapping and AMCL packages in the shared library, but there is also an option to download and build the packages from source.

5) Gazebo and RViz

Gazebo [21] is a physics simulator which we use along with RViz [22], a visualization tool, to observe our simulation where things interact with each other.

6) TurtleBot

TurtleBot is a low-cost, personal robot with open-source software and SDK available in ROS for development [5]. The well-defined TurtleBot2 robot is used in this project to perform autonomous navigation, obstacle avoidance, and service delivery tasks.

7) Simultaneous Localization and Mapping (SLAM)

SLAM is a probabilistic method that allows robots and other autonomous vehicles to build a map and localize itself on that map at the same time. Using a wide range of algorithms, computations, and other sensory data such as

from cameras or LiDAR, SLAM allows a robot or other vehicle—like a drone or self-driving car—to plot a course through an unfamiliar environment while simultaneously identifying its own location within that environment [1]. There are many different SLAM algorithms such as RTAB-Map [11], and ORB-SLAM2 [12]. However, we will not use any other SLAM algorithms besides GMapping in this project.

8) GMapping

GMapping is a SLAM technique that introduces Rao-Blackwellized particle filters as effective means to SLAM problem. This approach uses a particle filter in which each particle carries an individual map of the environment. Accordingly, a key question is how to reduce the number of particles. The authors present adaptive techniques to reduce the number of particles in a Rao- Blackwellized particle filter for learning grid maps. They propose an approach to compute an accurate proposal distribution considering not only the movement of the robot but also the most recent observation. This drastically decreases the uncertainty about the robot's pose in the prediction step of the filter. Furthermore, they apply an approach to selectively carry out re-sampling operations which seriously reduces the problem of particle depletion [14][15]. In short, it uses AMCL for localization under the hood while performing mapping. GMapping is available as a ROS package and is widely used and integrated with the TurtlelBot robot family [3]. We will not cover the mathematical details of GMapping and AMCL in this paper but readers are encouraged to conduct their own in-depth studies on these two concepts.

9) Adaptive Monte Carlo Localization

AMCL is a probabilistic localization system for a robot moving in 2D. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach, which uses a particle filter to track the pose of a robot against a known map [4]. The AMCL ROS package used in this project, by default, utilizes the Dijkstra's algorithm for path planning.

IV. TECHNICAL APPOACH AND IMPLEMENTATION

A. Starting Assumptions

For this project in general, the complexity of an end-to-end robotic simulation depends on how customized we want the simulation to be. For example, we want to ask questions such as do we want to design our own robotic system or use some existing systems? How many robots? Do we want to model our own virtual environment or using existing ones? How complex is the environment? What kind of tasks do we want the robot to achieve in the simulation? Or do we want to develop our own software and tools from scratch, to entirely use existing software and tools in ROS, or to use existing pieces of work and twist them for our own use case? In other words, the more we want to customize our simulation, the more complex the simulation is. Using a previous simulation work done by someone else is the easiest way to get this project done and in contrast, to build everything from scratch is the hardest way to achieve the same result. The former statement might not be always true because not all previously successful works are easy to be reproduced due to missing instructions, different hardware and/or software requirement, knowledge of the person who reproduces them, or simply that they are no longer supported. Different levels of complexity will pose different challenges.

Our goal is to fuse existing packages with our own development to produce our own project, avoiding developing everything from scratch and thus, having an appropriate project scope. Being said, our starting assumptions are as follows:

1. We use ROS1 Kinetic and Ubuntu Linux 16.04. At the time of completing this project, we do not have access to a good computer with a native Ubuntu installed, and Oracle VM Virtual or VMware has a bad rendering and significantly slows down our progress. Therefore, we use an online workspace with limited GPU usage hours. This workspace is Linux 16.04 and only ROS Kinetic is compatible with this Linux version. The same Linux and ROS1 version should be used to reproduce our work. Other Linux-ROS1 compatible versions can also be used to reproduce our work. The reproduction process is the same though might have some package dependency and package version compatibility issues. However, it is trivial to resolve these issues as this type of project – "SLAM with TurtleBot" is well-defined.
2. We know and use necessary existing packages available on ROS, and they are TurtleBot, GMapping, and AMCL packages instead of developing our own SLAM algorithm and robot. In addition, we use packages' default parameters and configurations such as launching TurtleBot2 with the default laser sensor or using default inflation for the AMCL unless otherwise specified.
3. We have a perfect 2D mapped environment for the robot navigation generated from the Gazebo world file using *pgm_map_creator*, so that we can skip the time-consuming mapping task that uses GMapping. In reality, the correct process is to manually control the robot to map the environment using GMapping. There are some differences in the 2D map generated by *pgm* and GMapping, and we will discuss them later in section IV. C.
4. We neglect the 3D map (z dimension – i.e., the height dimension) for the simplicity of the simulation since 2D maps are generally sufficient for the navigation task. However, we will attempt to force the robot to go under the table (i.e., involving the z dimension) to observe the robot's behavior.
5. The service delivery task is simplified. The object is virtual (i.e., a ROS marker is used) and that we assume the robot does not have a robotic arm with actuators and an end-effector because this is beyond our scope as we will need to modify the TurtleBot robot descriptions to interface with the gripper, perform path planning and motion control for the robotic arm, and so on. The robotic arm itself is a second robot and has its own software stack on

top of the TurtleBot2. We mimic the pick-up operation by stopping the robot at the pick-up location for five seconds and resume its movement toward the drop-off location.

B. Technical Approach

As a high-level approach to this project after taking the aforementioned assumptions into account, we

1. Created and initialized a ROS workspace.
2. Installed additional necessary packages and their dependencies that the online workspace did not have. They were *slam_mapping*, *turtlebot*, *turtlebot_interaction*, *turtlebot_simulator*, and *pgm_map_creator*.
3. Designed our own 3D environment where the TurtleBot2 operated in using the Gazebo Editor.
4. Used *pgm_map_creator* to generate the 2D map from the 3D world in the previous step.
5. Created a *marker* node for the virtual object to be picked up and dropped off.
6. Created a *delivery* node for the robot's service delivery task.
7. Wrote shell scripts to launch and run the TurtleBot2, *marker*, *delivery*, and navigation stack all at once in both Gazebo and RViz.
8. Conducted experiments on the simulation under different scenarios to detect potential bugs, which we would discuss the experiments as well as the result in section V.

The TurtleBot2 would be manually controlled by the computer keyboard (*teleop* package in *turtlebot* shown in figure 1) to perform the 2D mapping using GMapping. However, we replaced this time-consuming process with the 2D map generation by *pgm_map_creator* as said in our assumptions. Then, the 2D map was used by the AMCL for autonomous navigation and obstacle avoidance while performing the service delivery task. The readers should also be aware that the */src* directory is said to be under *catkin_ws* unless otherwise stated.

C. Implementation

1) Workspace and Additional Packages

We first created a ROS catkin workspace directory and then initialized it using *catkin_init_workspace* where all packages are this project reside in under the */src* directory. The overall structure under */src* is shown in figure 1. We tried to turn as many folders into ROS packages as possible with *catkin_create_pkg*, so that we could utilize the *rospack find* when needed and avoided using the absolute path.

Fig. 1. Final workspace structure under the *catkin_ws/src/* directory. All folders shown are ROS packages except *scripts*, *sim_env*, and *some_images*.

We downloaded additional packages in step 2 from source by

git clone https://github.com/ros-perception/slam_gmapping
git clone https://github.com/turtlebot/turtlebot
git clone https://github.com/turtlebot/turtlebot_interactions
git clone https://github.com/turtlebot/turtlebot_simulator

then, we builded them under */src* using *catkin_make* and *sourced* the terminal with *source devel/setup.bash* before running built-in examples as initial tests.

Some dependency issues prevented us from running the examples. The problem was resolved by checking and installing the missing libraries or packages.

rosdep -i install gmapping
rosdep -i install turtlebot_teleop
rosdep -i install turtlebot_rviz_launchers
rosdep -i install turtlebot_gazebo

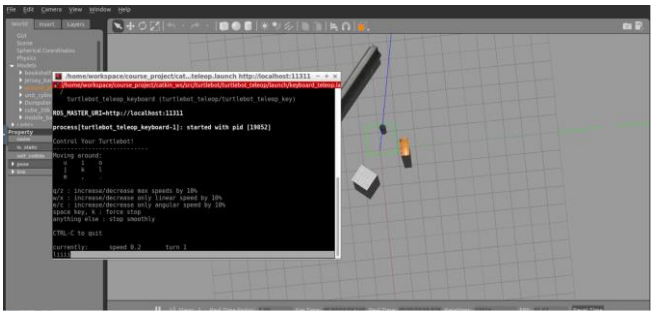


Fig. 2. Launching and testing TurtleBot2 that is manually controlled by computer keyboard in an example Gazebo world given in the *turtlebot* package.

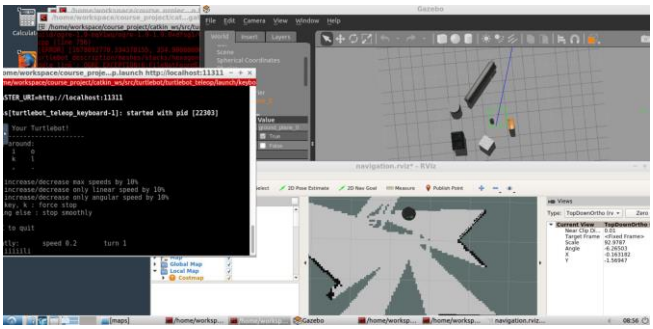


Fig. 3. Launching and testing an example to perform SLAM using GMapping.

These examples required to run multiple nodes in multiple different terminals. This inspired us to write a shell script later to launch everything at once.

2) Customized 3D World and 2D Map Generation

We created our own customized 3D environment using Gazebo Editor which was used throughout our experiments. This part of the project was simple, but it was time-consuming. The environment's configuration was saved inside *sim_env* as a *khoa.world* file.

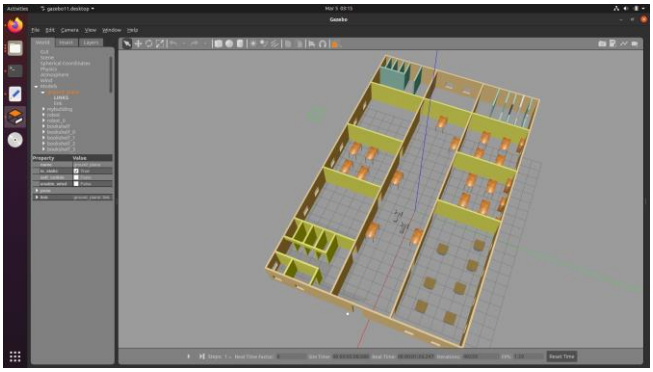


Fig. 4. Our customized 3D world created by using Gazebo Editor.

From the 3D Gazebo world, the 2D map was generated by *pgm_map_creator*. *khoa.world*, with a plugin added to the end of the file and was copied to */src/pgm_map_creator-/world/*.

gzserver src/pgm_map_creator/world/khoa.world
roslaunch pgm_map_creator request_publisher.launch

The above two commands produced the *map.pgm* shown in figure 5. Then, a *map.yaml* was created to store the 2D map's metadata and used by AMCL.

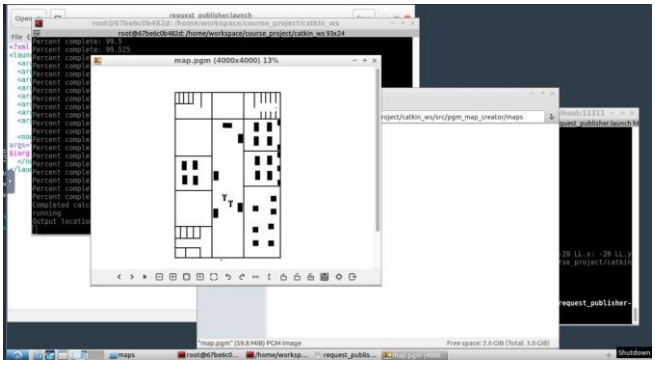


Fig. 5. 2D *map.pgm* generated *by pgm_map_creator* from the 3D Gazebo world file.

There were some problems with the above map. Firstly, the map was created from a 90º top view on the z-axis. This was not what the robot, on the ground, would see with its laser scanner that resulted in the generated 2D map shown in figure 6.

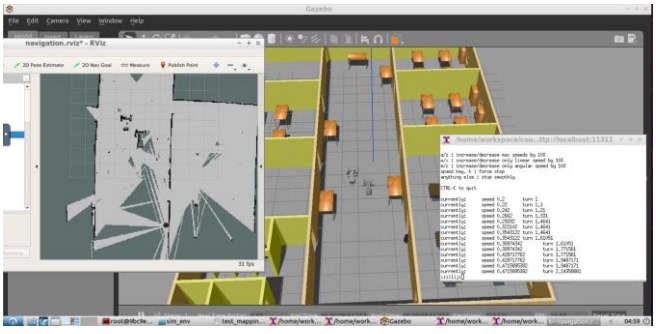


Fig. 6. Correct 2D map scanned by the robot's laser scanner. The map can be saved using map_server.

There were no open spaces where the doors were. The robot saw tables as completely occupied spaces instead of four small, occupied circles (i.e., table legs) in figure 5 as compared to figure 6. Therefore, we used *gimp* to erase occupied spaces where the doors were and kept the tables as being occupied spaces.

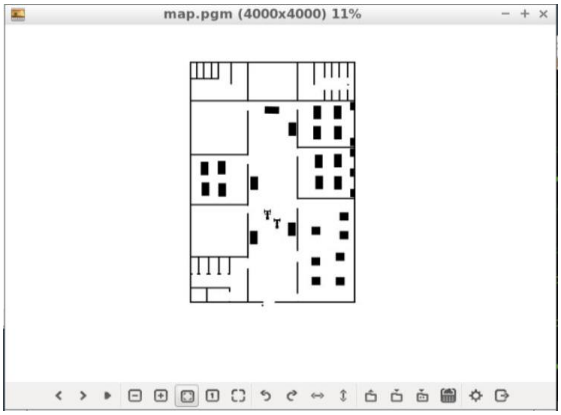


Fig. 7. Correcting the map shown in fig. 5 using *gimp*.

Secondly, the first few generated maps were partially occluded (left out of the image), and we had to change the x-y min/max values in the *request_publisher.launch* to obtain the complete map above. We also had to fix some deprecated syntax error in the *collision_map_creator.cc*. Lastly, TurtleBot2 was spawned outside of the map and its orientation with respect to the map on the z-axis was not correct. These two issues were fixed by changing the origin in map's metadata *yaml* file and the *initial_pose_a* in the *amcl_demo.launch* under the *turtlebot_gazebo/launch/* of the *turtlebot_simulator* package respectively. There seemed to be a pose (translation and orientation) mismatch between Gazebo and RViz which later on caused a problem to identify different correct coordinates of the pick-up and drop-off location for the robot movements as well as for the virtual object' spawning. Understanding why this happened was beyond our scope. However, one possible solution would be finding the homogeneous transformation matrix between a set of coordinates in Gazebo and a set of corresponding coordinates in RViz.

3) Marker and Delivery ROS Node.

Two C++ ROS nodes *marker* and *delivery* were implemented inside two ROS packages of the same names respectively. The packages were created by *catkin_create-_pkg* with some ROS message dependency arguments. The *marke*r node first spawned a virtual cube in blue color in RViz at the pick-up location. It disappeared once the robot reached within some distance away from the pick-up location and reappeared at the drop-off location once the robot reached within some distance away from the destination. On the other hand, the *delivery* node sent the coordinates of the pick-up and drop-off location to the robot. The robot stopped for five seconds after reaching the pick-up zone, mimicking the gripper gripping the object or waiting for a human to place the object inside its compartment, and resumed movement toward the destination after five seconds.

The above showed that there was an interaction between the virtual and the TurltleBot2 because the object knew when to disappear and reappear with respect to the robot's location. In fact, the *marker* tracked the robot's position through its AMCL pose message and the *delivery* sent the coordinates to robot as well as tracked its position, so that the robot knew when to stop, resume movement, and stop again. The robot also printed on the terminal whether it arrived, did not arrive, or failed to reach either pick-up or drop-off location. Figure 8 describes the overall flow of the algorithm.

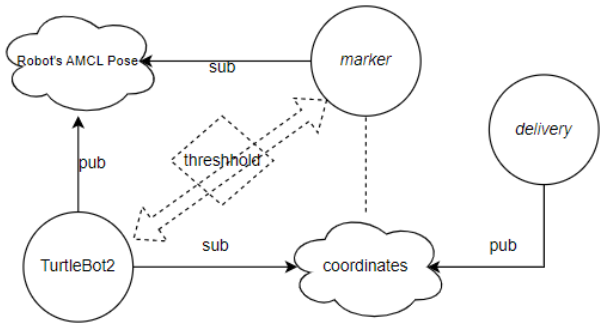


Fig. 8. Service delivery algorithm.

To ensure the complete development of these two packages, their corresponding *CMakeLists.txt* were modified, allowing a successful building process at the *catkin_ws* level using *catkin_make*.

4) Shell Scripts and Simulation

From running examples for testing packages to putting things together for our own final simulation setup, we realized that multiple processes had to be launched/run in multiple terminals separately. Therefore, we wrote multiple shell scripts to launch everything at once conveniently. For example, the example shown in figure 3 requires launching and run the following four processes: TurtleBot2 in our customized Gazebo world, GMapping node for SLAM, RViz, and *teleop* that manually controls the robot with keyboard in each different terminal separately and respectively using four different commands. The shell script running this specific simulation included these four commands in one single file and started them all up with just one command line *./file_name.sh*. Similarly, we named the shell script *final_demo.sh* to run the final simulation setup (or demo) and this final setup had different configurations for our experiments, which we will discuss in section V. Figure 9 shows the inside of the *final_demo.sh* shell script.

```
final_demo.sh - Notepad
File  Edit  Format  View  Help
#!/bin/sh

xterm -e " roslaunch turtlebot_gazebo turtlebot_world.launch world_file:=$(rospack find sim_env)/worlds/khoa_obj.world" &
sleep 5

xterm -e " roslaunch turtlebot_gazebo amcl_demo.launch map_file:=$(rospack find maps)/map.yaml" &
sleep 5

xterm -e " roslaunch turtlebot_rviz_launchers view_navigation.launch" &

xterm -e " rosrun marker marker" &
sleep 5

xterm -e " rosrun delivery delivery"
```

Fig. 9. final_demo.sh.

V. EXPERIMENTS AND RESULTS

A. Experiment Setup

We modified the final simulation setup to have different configurations by adding static obstacles and dynamic obstacles that were not initially mapped by the robot). In general, the configurations were grouped into five types with the following characteristics:

1. **Type I:** static obstacles, easy navigation and obstacle avoidance where the pick-up and drop-off points were located at open spaces (i.e., clear of obstacles or out of obstacles' inflation zones).
2. **Type II:** static and/or dynamic obstacles, easy navigation and obstacle avoidance where the pick-up and drop-off points were located at open spaces (i.e., clear of obstacles or out of obstacles' inflation zones).
3. **Type III:** static and/or dynamic obstacles, easy to medium navigation and obstacle avoidance where the pick-up and drop-off points were located near obstacles' inflation zones.
4. **Type IV:** static and/or dynamic obstacles, medium navigation and obstacle avoidance where the pick-up and drop-off points were located inside obstacles' inflation zone (i.e., under the tables specifically)
5. **Type V:** static and/or dynamic obstacles that forced the robot to go under the tables (tables appeared to be occupied spaces on the 2D map), medium to hard navigation and obstacle avoidance where the pick-up and drop-off points were located at open spaces, near, or inside obstacles' inflation zones (i.e., under the tables specifically).

We ran each of the five simulation types 10 times, observed the robot's behavior, and recorded the results. All parameters related to the robot and AMCL were kept as default. The robot succeeded when it completed the service delivery task with or without making absurd behavior and fails otherwise. Any absurd behavior occurring during the simulation may indicate potential software bugs that may lead to system failures during live testing. The reason why we were particularly interested in having to force the robot to go under the tables or having the pick-up/drop-off locations at the tables because the tables were occupied spaces in the original 2D map but the robot actually saw four table legs as mentioned in section IV. C. 2, figure 5 and 6). The robot was also shorter than the tables in order to go under and this would trigger some awkward behaviors in the TurtkeBot2.

B. Results

Table 1 sums up the results from our experiments.

TABLE I. OBSERVATIONS FROM EXPERIMENTS.

Type	***Success Rate***	***Typical Absurd Behaviors***
I	10/10 (100%)	none
II	10/10 (100%)	none
III	7/10 (70%)	Robot model circles around or jumps around the map, stuck under the table(s), and/or does not prompt that it fails to reach the locations
IV	2/10 (20%)	Same as type III
V	3/10 (30%)	Same as type III

As we expected, type I and II had a 100% success rate since these were straightforward scenarios for the robot to navigate around. For type III, we would expect to see a 100% success rate or a 100% failure rate depending on how the developers implemented the algorithms (i.e., how much they wanted to tolerate the robot being near the inflation

zones). Conveying such an aspect of the algorithm was beyond our project scope. We placed the locations near the inflation zones where we considered safe for the robot to be in. However, the results showed a mix of success and failure, meaning for the same exact locations being simulated 10 times, the robot failed three times and succeeded seven times. When the robot failed, it informed the system that it failed to reach the location once and did not do this twice. We expected to see the same outcome for type III and IV – 100% success rate or 100% failure rate. However, the results were again a mix, and the failure rate was significantly higher in type IV and V with the same observed absurd behaviors found in type III. We found that these behaviors happened when the robot failed to reach the designated locations, especially when it was forced to go under the tables. Regardless, the *2D Pose Estimate* in RViz sometimes helped the TurtleBot2 recover from its weird behaviors and resume its normal operation.

This inconsistency in success/failure of type III, IV, and V and the associated weird behaviors may have implied some underlying potential software bugs that would be worth looking into in-depth for our future work. But for the completeness of our result analysis, we assumed there were two probable causes:

1. The developers did not account for all edge cases when they developed the AMCL algorithm. For example, in our case, the robot was forced to go under the tables where the 2D map showed the tables as occupied spaces but the robot saw them as free spaces and thus causing confusion to the robot. Regardless, this suggested that the algorithm may not have been designed to clearly inform what the robot needed to do in this case – spinning around or stopping its operation. If this was the case, further development on top of the original AMCL algorithm would be needed to specify what the robot should do in this scenario.
2. Given the popularity and long history of the AMCL, the algorithm may have accounted for the above. If this is the case, the end user would need to tune its parameters to avoid such a situation, in which in this project, we used default parameters as stated in our starting assumptions.

It was also worth noting that our simulation was run for a limited amount of time, meaning the obtained results might not reflect our observations correctly until more data becomes available in our future work. The following figures demonstrate some snapshots of the overall simulation in this capstone project.

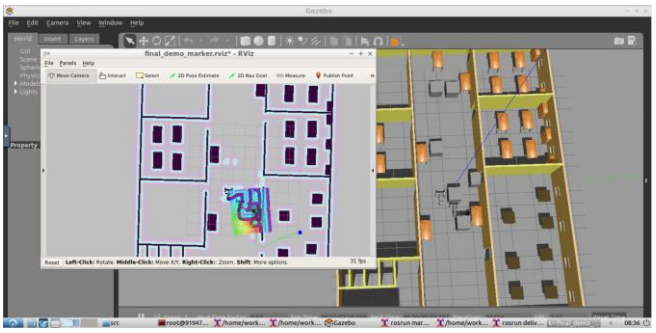


Fig. 10. TurtleBo2, being forced to go under a table, finds its way to the pick-up location.

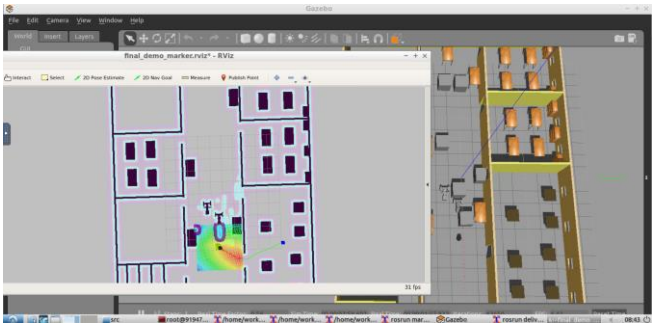


Fig. 11. *2D Pose Estimate* in RViz helps the robot recover sometimes.

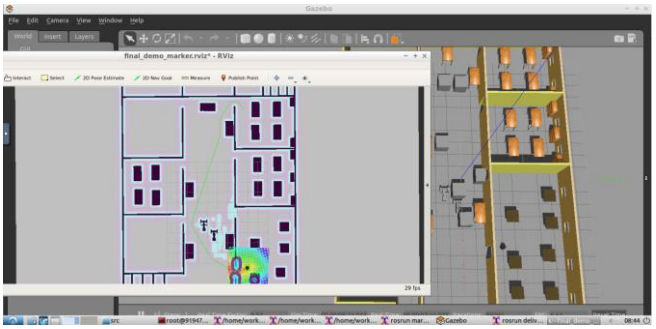


Fig. 12. *Going to the destination after picking up the object*.

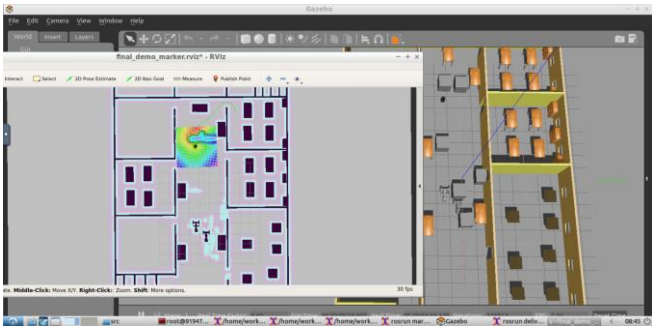


Fig. 13. *Avoiding dynamic obstacles before reaching the destination.*

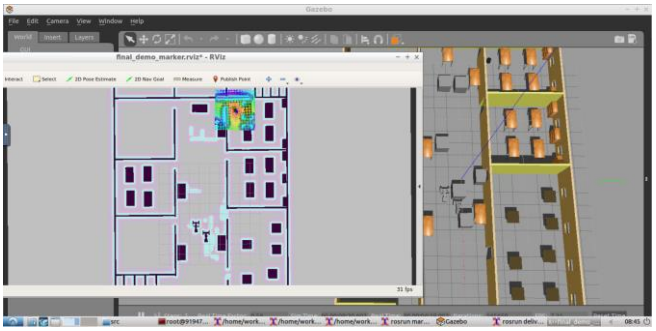


Fig. 14. *Dropping off the object at the destination, finishing the task.*

VI. Conclusion

In this project, we used the existing ROS packages (TurtleBot, GMapping, and AMCL) and integrated them with our own developments such as 3D Gazebo world and ROS nodes (*delivery* and *marker*) to simulate an autonomous service robot. The TurtleBot2 robot was capable of SLAM using GMapping and autonomous navigation and obstacle avoidance using AMCL. Experiments showed that we successfully put different software components together to produce a generally working simulation in Gazebo and visualized the entire process in RViz. However, there were certain simulated configurations where we identified, or at least the obtained results suggested potential bugs that could potentially lead to software failures during live testing by observing the robot's absurd behaviors in these cases. Although we did not convey in-depth what may have caused these behaviors and what the exact bugs were, we provided some insights in our discussion and analysis that could be helpful to our own future work or anyone who would like to conduct a further study on our subject. Throughout the process of completing this project, we encountered several problems but were fortunate to overcome. We also learned that integrating different existing software components available in ROS or even just reproducing a past work could be sometimes a hassle due to missing instructions, different hardware and/or software requirement, knowledge of the person who reproduces them, or simply that software components were no longer supported.

VII. Future Work

There are several limitations in this work of ours. Addressing them in this section inspires us to provide further directions for our future work. To simplify our simulation as a part of reducing our project to an appropriate scope, our starting assumptions did not tune the GMapping's and AMCL's parameters, use the z dimension for mapping and navigation, and use an actual end-effector for the service delivery task. Therefore, our future work will account for these three important aspects because tuning the parameters might help avoid such absurd behaviors in the robot shown in section V, accounting for the z dimension can also be helpful in preventing the robot to go under the tables, and using the end-effector to simulate the actual pick-off and drop-off action. In addition, we will reduce our dependence on existing ROS packages and develop our own algorithms instead in the future. There was a coordinate mismatch between Gazebo and RViz and to place the virtual object at the same place in both applications, we had to perform trial and error and this is time-consuming. We will investigate the underlying issue that causes the mismatch and probably resolve this with a simple homogeneous transformation matrix. We plan to generate more data for a better and reliable result analysis and conclusion by increasing the amount of time the simulation is run, as well as to convey the GMapping and AMCL in depth. We finally want to inspect and use other SLAM algorithms such as ORB-SLAM and RTAB-Map and compare them against GMapping/AMCL in finding such potential software bugs in our simulation.

References

[1] "What is Simultaneous Localization and Mapping (SLAM)?" [Online]. Available: Here. [Accessed: 20-Feb-2023].

[2] "Robot Operating System." [Online]. Available: Here. [Accessed: 20-Feb-2023].

[3] "Gmapping," 04-Feb-2019. [Online]. Available: Here. [Accessed: 20-Apr-2023].

[4] "Amcl," 27-Aug-2020. [Online]. Available: Here. [Accessed: 20-Apr-2023].

[5] "TurtleBot," 30-Jan-2020. [Online]. Available: Here. [Accessed: 20-Feb-2023].

[6] Afzal, Afsoon et al. "A Study on the Challenges of Using Robotics Simulators for Testing." ArXiv abs/2004.07368 (2020): n. pag.

[7] "What is Advanced Robotics Simulation?" [Online]. Available: Here. [Accessed: 20-Feb-2023].

[8] "Pgm_map_creator," 13-Nov-2018. [Online]. Available: Here. [Accessed: 20-Apr-2023].

[9] Thale, Sumegh & Prabhu, Mihir & Thakur, Pranjali & Kadam, Pratik. (2020). ROS based SLAM implementation for Autonomous navigation using Turtlebot. ITM Web of Conferences. 32. 01011. 10.1051/itmconf/20203201011.

[10] K. Li and H. Tu, "Design and Implementation of Autonomous Mobility Algorithm for Home Service Robot Based on Turtlebot," 2021 IEEE 5th Information Technology,Networking,Electronic and Automation Control Conference (ITNEC), Xi'an, China, 2021, pp. 1095-1099, doi: 10.1109/ITNEC52019.2021.9587138.

[11] M. Labbé and F. Michaud, "RTAB-Map as an Open-Source Lidar and Visual SLAM Library for Large-Scale and Long-Term Online Operation," in Journal of Field Robotics, vol. 36, no. 2, pp. 416–446, 2019.

[12] Raúl Mur-Artal and Juan D. Tardós. ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras. IEEE Transactions on Robotics, vol. 33, no. 5, pp. 1255-1262, 2017.

[13] "Navigation," 14-Sep-2020. [Online]. Available: Here. [Accessed: 20-Feb-2023].

[14] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard: Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters, IEEE Transactions on Robotics, Volume 23, pages 34-46, 2007.

[15] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard: Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling, In Proc. of the IEEE International Conference on Robotics and Automation (ICRA), 2005.

[16] "Markers: Sending Basic Shapes (C++)," 25-Aug-2018. [Online]. Available: Here. [Accessed: 20-Feb-2023].

[17] "Nodes," 12-April-2018. [Online]. Available: Here. [Accessed: 20-Feb-2023].

[18] "Packages," 14-Apr-2019. [Online]. Available: Here. [Accessed: 20-Apr-2023].

[19] "Topics," 20-Feb-2019. [Online]. Available: Here. [Accessed: 20-Apr-2023].

[20] "Msg," 31-Jan-2023. [Online]. Available: Here. [Accessed: 20-Apr-2023].

[21] "Gazebo Package Summary," 13-Feb-2021. [Online]. Available: Here. [Accessed: 20-Feb-2023].

[22] "Rviz Package Summary," 16-May-2018. [Online]. Available: Here. [Accessed: 20-Feb-2023].

[23] Sotiropoulos, Thierry & Waeselynck, Helene & Guiochet, Jérémie & Ingrand, François. (2017). Can Robot Navigation Bugs Be Found in Simulation? An Exploratory Study. 150-159. 10.1109/QRS.2017.25.

[24] Timperley, Christopher & Afzal, Afsoon & Katz, Deborah & Hernandez, Jam & Goues, Claire. (2018). Crashing Simulated Planes is Cheap: Can Simulation Detect Robotics Bugs Early?. 331-342. 10.1109/ICST.2018.00040.