EECE 5550 Final Project Report

Team Bebop

February 9, 2022

Team Members

- Abhinav Gupta
- Kaushik Venkatesh
- Kevin Robb
- Ricky Kaufman
- Sarthak Gupta

Figure 1: Team photo after first successful full demo.

1 Introduction

In this project we apply the concepts of mobile robotics to perform autonomous reconnaissance in a simulated disaster environment. More specifically, our task entails placing a TurtleBot3 in an unexplored environment populated by AprilTags, which act as stand-ins for the simulated victims. The Turtlebot must generate a complete map of the environment in addition to a comprehensive list of the AprilTags present. This list must include the AprilTag's absolute pose with respect to the map generated and its ID number. To complete this task, the Turtlebot is equipped with a 360° LiDAR scanner, necessary for localization and mapping, and a Rasberry Pi Camera to detect the Apriltags. A successful reconnaissance operation will require our team to implement techniques such as mobile robotic kinematics and sensing, feature extraction, simultaneous localization and mapping (SLAM), and motion planning.

2 Procedure

2.a General Outline

1. Place the robot in a new, unknown, environment.
2. Simultaneously:
    - Via Cartographer, perform SLAM to create a map of the environment and track the position of the robot.
    - Use frontier exploration to autonomously drive the robot around the environment.
    - Detect AprilTags and store their global poses.
3. End the search after all the tags are detected and a complete map of the environment is developed

2.b To Run Our Code

Before starting, clone our git repo to both the robot and the host PC, and ensure that it is sourced on both.

1. Establish an SSH connection between the robot and the host PC. This can be done with the following command in the terminal:

    ```
    ssh ubuntu@IP_ADDRESS_OF_RASPI_ON_ROBOT
    ```

    Update the `.bashrc` with the IP addresses of both devices. On the robot, the following two lines should be in the `.bashrc`:

```
export ROS_MASTER_URI=http://IP_ADDRESS_OF_REMOTE_PC:11311
export ROS_HOSTNAME=IP_ADDRESS_OF_RASPI_ON_ROBOT
```

while the following two lines should be in the `.bashrc` of the host PC:

```
export ROS_MASTER_URI=http://IP_ADDRESS_OF_REMOTE_PC:11311
export ROS_HOSTNAME=IP_ADDRESS_OF_REMOTE_PC
```

2. Run `roscore` on the host PC.

3. Run the two raspicam commands on the robot via SSH.

```
roslaunch raspicam_node camerav2_1280x960_10fps.launch enable_raw:=true
rosrun tf static_transform_publisher 0.03 0 0.1 0 1.57 0 base_link raspicam 100
```

4. Run the AprilTag detection node on the robot via SSH. This will use the camera images to identify AprilTag poses. We use the provided launch file from Lab 3, with some modifications to the camera information.

```
roslaunch bebop apriltag_gazebo.launch
```

5. On the host PC, run the node `tag_tracking_node.py`, a custom node to track of all tags detected.

```
rosrun bebop tag_tracking_node.py
```

6. Run turtlebot bringup on the robot.

```
roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

7. Add this line to remap from `/map` topic to `/cmap` in `turtlebot3_cartographer.launch` (occupancy grid node) file in `turtlebot3_slam` package.

```
<remap from="/map" to="/cmap" />
```

8. Run SLAM on the host PC. We use Cartographer.

```
roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=cartographer
```

9. Remap the Cartographer occupancy grid for explore_lite with our node `cartographer_remapping.py`. Run on the host PC.

```
rosrun bebop cartographer_remapping.py
```

10. Change the topic to which move_base publishes the command velocity in `move_base.launch` in the `turtlebot3_navigation` package from "`/cmd_vel`" to "`/cmd_vel_intermediary`".

11. Run our node (on the host PC). This intercepts motion commands and forwards them to the robot.

```
rosrun bebop cmd_interrupt_node.py
```

12. Run our motion planner on the host PC. (Cartographer will automatically run move_base by default)

```
roslaunch turtlebot3_navigation move_base.launch
roslaunch bebop explore_lite_custom.launch
```

Once the turtlebot completes its task, you can save the map by running

```
rosrun map_server map_saver <FILEPATH>
```

The list of tags' global poses is automatically saved to your working directory under the filename "`tags_DATETIME.txt`".

2.c Deliverables

- A complete map of the environment.
- A list of all AprilTags present in the environment with their global poses.
- A screen recording alongside a video of the robot exploring a full environment for demonstration.

3 Design Process

3.a Motion Model

The Turtlebot3 is a differential drive robot. It's equipped with two parallel wheels with a common instantaneous center of curvature. Both wheels have a radius $r = .033$m and a track width $w = .16$m. A schematic of a differential drive robot is displayed in 2.

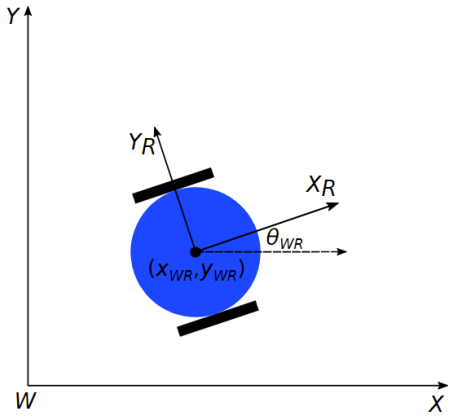

Figure 2: Schematic of Differential Drive Robot

Given the TurtleBot3 design, the robot's position is constrained to a 2-Dimensional plane, and there is no lateral motion allowed in the Y_R direction. We start the motion model design by outlining the robot's equations of motion. Note that the TurtleBot3 allows control of the left and right wheel angular velocities, $\dot{\phi}_l$ and $\dot{\phi}_r$ respectively. The robot's linear and angular velocities in the world frame, given the wheel angular velocities and body orientation, can be determined as follows:

$$\begin{bmatrix} \dot{X}_{WR} \\ \dot{Y}_{WR} \\ \dot{\theta}_{WR} \end{bmatrix} = \begin{bmatrix} \cos(\theta_{WR}) & -\sin(\theta_{WR}) & 0 \\ \sin(\theta_{WR}) & \cos(\theta_{WR}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{r}{2}(\dot{\phi}_r + \dot{\phi}_l) \\ 0 \\ \frac{r}{w}(\dot{\phi}_r - \dot{\phi}_l) \end{bmatrix} \tag{1}$$

With these velocity definitions, we can define the equations of motion on the Lie Group SE(2). We start with the pose of the robot, obtained by mapping the orientation, $R(\theta) \in \text{SO}(2)$, and position, $t \in \mathbb{R}^2$, of the robot to SE(2). Here, the elements of t represent X_{WR} and Y_{WR}, the x and y position of the robot in the world frame. $R(\theta)$ represents the orientation of the robot with respect to the world frame, defined below:

$$R : \mathbb{R}^2 \to \text{SO}(2)$$

$$R(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$

Given our definitions for t and $R(\theta)$, the pose of the robot is as follows:

$$\psi(t, \theta) : \mathbb{R}^2 \times \mathbb{R} \to \text{SE}(2)$$

$$\psi(t, \theta) = \begin{bmatrix} R(\theta) & t \\ 0 & 1 \end{bmatrix} \tag{2}$$

Note that the identity $I \in \text{SE}(2)$ is obtained at $\psi_{(0,0)}$. Next, an element of the Lie Algebra of SE(2), the tangent space at the identity, is obtained from the closed form expression for the derivative map at the identity, $d\psi_{(0,0)}$.

$$d\psi_{(0,0)} : T_{(0,0)}(\mathbb{R}^2 \times \mathbb{R}) \to \text{Lie}(\text{SE}(2))$$

$$d\psi_{(0,0)} = \begin{bmatrix} 0 & -\dot{\theta} & \dot{x} \\ \dot{\theta} & 0 & \dot{y} \\ 0 & 0 & 0 \end{bmatrix} \tag{3}$$

Considering the linear and angular velocities of the TurtleBot3 are determined by the two wheel angular velocities, we can derive a map which sends the wheel velocities $(\dot{\phi}_r, \dot{\phi}_l)$ to the Lie algebra element $\dot{\Omega}_{(\dot{\phi}_r, \dot{\phi}_l)}$.

$$\dot{\Omega} : \mathbb{R}^2 \to \text{Lie}(\text{SE}(2))$$

$$\dot{\Omega}_{(\dot{\phi}_r, \dot{\phi}_l)} = \begin{bmatrix} 0 & -\frac{r}{w}(\dot{\phi}_r + \dot{\phi}_l) & \frac{r}{2}(\dot{\phi}_r + \dot{\phi}_l) \\ \frac{r}{w}(\dot{\phi}_r + \dot{\phi}_l) & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{4}$$

Finally, the Lie algebra element for a given set of left and right motor commands can be utilized to determine the robot's trajectory. This forward kinematics model is defined using the exponential map. Given an initial pose of the robot $X_o \in \text{SE}(2)$, and motor commands $(\dot{\phi}_r, \dot{\phi}_l)$, we determine the pose of the robot at time t, $\gamma(t)$, as:

$$\gamma(t) = X_o exp_m(t\dot{\Omega}_{(\dot{\phi}_r, \dot{\phi}_l)}) \tag{5}$$

Using equation 5, one can also conduct inverse kinematics analysis. Here, motor commands $\dot{\phi}_r$ and $\dot{\phi}_l$ may be determined from a feasible desired pose.

3.b SLAM Implementation

We utilize the package Cartographer_ROS for simultaneous localization and mapping. Developed by Google in 2016, it supplies real-time simultaneous localization and mapping capabilities with ROS integration to the turtlebot. The cartographer system provides a novel methodology of reducing mapping and loop closure computational requirements.

The 2D grid map obtained by Cartographer has an effective resolution of $r = 5cm$. Particle filters are not deployed in this method. Instead, pose optimization is run at regular intervals, with laser scans inserted onto a sub-map at the best estimated pose. Finished sub-maps do not include laser scans, and actively contribute to the scan matching phase. Sufficient scan matches for an estimated current pose are added as a *loop closing constraint* to the optimization problem. Optimization is performed every few seconds, and loops are closed when a location is revisited. Consequently, loop closures are implemented more frequently than scan additions. A GPU accelerated amalgamation of the finished and current sub-maps presents the operator/client with an updated preview of the map.

Regarding to our task, Cartographer provides us with a means to achieve our two primary objectives. Firstly, we need a complete occupancy grid of the unknown environment. Cartographer publishes to the topic `/map` through the node `occupancy_grid_node`, which provides the present occupancy grid produced by the robot and its package. As the turtlebot obtains new information, this topic's data continuously updates and expands on its existing grid. In addition to creating the map necessary to the assignment, this topic is also essential to the motion planning and frontier exploration phase of the project. As described in section 3.a, the nodes of package `explore_lite` subscribe to `/map`, using the occupancy grid to generate frontiers.

In addition to building the map and searching for frontiers, Cartographer is also necessary to log the absolute positions of the AprilTags. The `apriltag_ros` package alone can only obtain the pose of the AprilTags with respect to the Robot's current pose. The localization offered by Cartographer allows the robot's pose to be depicted with respect to a static base frame on the map. This pose corresponds to the initial pose of the turtlebot when Cartographer is launched, and is made available in the "`/tf`" topic "`map`". With this base frame provided by Cartographer, we can obtain the absolute pose of each AprilTag we discover. The process for doing so is outlined in AprilTag Detection section, 3.c.

3.c AprilTag Detection

AprilTags are detected utilizing a Rasberry Pi Camera in combination with the apriltag_ros package. We obtain a current estimate of the robot's pose from the `map` frame in the `/tf` topic, provided by Cartographer. When an AprilTag is in frame of the robot's camera, a message is published to the "tag_detections" topic. This message contains the tag's ID and its pose relative to the camera. We calculate its global pose and add it to a list of all tags detected thus far, indexed by their IDs.

We start by following the ROS tutorial for Monocular Camera Calibration. Here, we printed a 10×7 checkerboard and obtained an `ost.yaml` file with the calibration data. To make sure that the camera detects the AprilTags, `settings.yaml` and `tags.yaml` were edited with updated parameters of the printed Apriltags. The files can be found in the config folder (which can be navigated to from the terminal using `roscd apriltag_ros`). These files specify the family of tags we are using, `36h11`, the IDs of all tags to expect, as well as the physical sizes of the tags in our environment.

To better understand how the Raspi Cam works with AprilTag detection, we determine the relative range of poses that we'll be able to detect with the raspicam. We start by reviewing the specifications of the Raspi Camera V2, finding that the field of view is $\approx 70°$. Our tags printed for practice are 0.17×0.17 m, and we calculate the minimum distance at which the tag will remain fully in view as ≈ 0.24 m. In our testing, by moving the tags around and using the `/tag_detections/image` topic, we determined a distance range of ≈ 0.26 m to ≈ 2.5 m. We also found that in most cases a tag can be detected at an angle offset from normal by as much as 80°.

A custom python node is used to find the global poses of the tags. The global pose of an AprilTag can be computed as:

$$T_{AO} = T_{AC} \cdot T_{CO}$$

where

$$\begin{cases} T_{AO} = \text{Pose of AprilTag relative to Origin} \\ T_{AC} = \text{Pose of AprilTag relative to Camera} \\ T_{CO} = \text{Pose of Camera relative to Origin} \end{cases}$$

We use a TF buffer and listener from the `tf2_ros` python package to obtain and update T_{CO} every so often. T_{AC} is obtained from the "`tag_detections`" topic.

These global poses are stored using a python dictionary with the tag IDs as keys, and saved to a file every so often. This ensures that we will still have them saved if the node exits unexpectedly. We update the pose estimate with new measurements using equation 6

$$\texttt{tags[id]} = L \cdot \texttt{tags[id]} + (1 - L) \cdot T_{AO} \tag{6}$$

where L is a learning rate that we set to 0.9. Setting this to 0 will cause old data to be thrown away and replaced by each new measurement. The pseudocode for the ROS node is additionally available below:

Algorithm 1 tag_tracking_node

$rospy.Subscriber(\text{"}/tag\_detections\text{"}, AprilTagDetectionArray, get\_tag\_detection)$ ▷ Subscriber

$T_{CO} \leftarrow tf\_listener('raspicam', 'map')$ ▷ TF frames service
$L \leftarrow 0.9$ ▷ Learning rate
$tags \leftarrow \{\}$ ▷ create a dictionary to store the tags

if $len(tag\_msg.detections) \neq 0$ **then** ▷ tag_msg is info from tag_detections topic
  $T_{AC} \leftarrow tag\_pose \leftarrow tag\_msg.detections.pose.pose.pose$
  $tag\_id = tag\_msg.detections.id$
  $T_{AO} \leftarrow np.matmul(T_{AC}, T_{CO})$
  if tag_id in $tags.keys()$ **then**
    $tags[tag\_id] \leftarrow np.add(L * tags[tag\_id], (1 - L) * T_{AO})$ ▷ Update pose of tag
  else
    $tags[tag\_id] \leftarrow T_{AO}$ ▷ New tag found
  end if
end if

Note about tag updating: our method of updating the tags in Eq. (6) assumes vector operands, which we do not have in this case. If we were to implement this project again, we would use a more correct approach for SE(3) pose estimation, which we discuss in Section 5.

3.d Motion Planning

3.d.1 Frontier Exploration

For frontier exploration we utilize the ROS package explore_lite, which promotes greedy frontier based exploration. Here, the robot subscribes to an occupancy grid map, identifies the frontiers, and then sends the movement goals to the move_base node in order to explore the frontier. The occupancy grid map utilized by this package is obtained from Cartographer. Note, however, that explore_lite anticipates a grid with three possible values: $[0, 1, -1]$, representing unoccupied, occupied, and unknown cells. From Cartographer, the \map topic provides an occupancy grid map where cells may be assigned values $[0 - 100]$, as a range of probability for occupancy, in addition to -1 for unknown space. To address this, we reassign the cells with values $[0 - 100]$ to values of either 0 or 100 based on a thresholds M and N. Below we show how we mapped Cartographer's occupancy grid to a new topic for explore_lite to subscribe to:

Algorithm 2 Cartographer_remapping

$M \leftarrow 75$
$N \leftarrow 50$
if $Cell \geq M$ **then**
    $Cell \leftarrow 100$
else if $Cell \geq 0$ and $Data \leq N$ **then**
    $Cell \leftarrow 0$
else
    $Cell \leftarrow -1$
end if

With a suitable occupancy grid, `explore_lite` searches for unknown space that borders empty space. This is designated as a frontier space, and is visualized as blue frontier points. Additionally, the package displays green spheres in `Rviz` corresponding to each frontier. The size of these spheres designates the evaluated cost associated with the pursuit of that frontier. In simple terms, a frontier's cost is determined by its minimum distance from the turtlebot in combination with its size. As a greedy frontier based exploration package, `explore_lite` will typically prioritize exploration of large frontiers. The extent to which the frontier size affects the frontier cost is determined by the `gain_scale` and `potential_scale` parameters. When the gain scale is high, the frontier size is weighted heavily in determining the cost. When zero, the frontier cost is dependent on its proximity to the robot alone. In contrast, a higher `potential_scale` value increases the weight ascribed to the distance from the frontier to the turtlebot. When high, the lowest cost frontiers will be those closest to the robot. To avoid the turtlebot altering its trajectory to pursue a larger but out of the way frontier, we may increase `potential_scale` to a value lower greater than its default of 3. In consequence, the turtlebot will more frequently finish exploring a path before taking on a new one.

Another important parameter we consider is `min_frontier_size`. When `explore_lite` is used in combination with a SLAM generated map, it recommends specification of a minimum frontier size. This accounts for how the mapping in SLAM may overlook parts of certain obstacles. Specifying a minimum frontier size will prevent the turtlebot from pursuing frontiers that are inaccessible. For our demonstration (4), the primary obstacles for the turtlebot to navigate around are tables boxes. We chose a minimum frontier size to ignore any frontiers developed on the walls between the gaps of the border table obstacles. The final parameter we tune is `progress_timeout`. This determines the point at which the robot will give up on a frontier it has made no progress on. Given the potential for the turtlebot to choose and get stuck on a frontier it cannot further explore, we decrease this period. Below we display a table outlining our choices for selected `explore_lite` parameters.

The final component of `explore_lite` is the movement commands. When a priority frontier is identified, a goal pose is sent to the move_base node, which is part of the move_base package. Next, the move_base node publishes velocity commands with `cmd_vel(geometry_msgs/Twist)`. These velocity commands are chosen in order to achieve the desired goal pose, and are determined using

Parameter	Values	Units
gain_scale	1	–
potential_scale	4	–
min_frontier_size	.4	meters
progress_timeout	15	seconds

Table 1: explore_lite Parameter Definitions

the motion model outlined in section 3.a.

3.d.2 Ensuring All AprilTags are Seen

To ensure all AprilTags come into the camera's view while exploring the environment, we make some additions to the robot's motion plan. More specifically, we change the topic to which our motion planner publishes from "/cmd_vel" to "/cmd_vel_intermediary". Additionally, we create a node cmd_interrupt_node.py, which subscribes to the "/cmd_vel_intermediary" topic and decides whether to forward it along to "/cmd_vel" or to replace with our own motion command.

Our basic approach to improve exploration of the environment entails stopping the robot every specified number of seconds, and spinning it a full 360° revolution in place. We do this slowly, at one revolution per 20 seconds. Doing so ensures that the camera will acquire good data to use for tag detection and prevents SLAM from improperly creating frontiers along walls. This improves the chance that the Raspi camera will view every surface in the environment.

To determine the motor commands necessary to achieve this goal, we refer to our motion model in section 3.a. Velocity commands of the message type Twist() are published to the "/cmd_vel" topic. This comes in the form of two arrays of length 3: one for linear velocity commands and one for angular velocity commands. For our goal, we only need to incite rotation about the robot's z-axis, meaning all other values will be left at zero. To determine the necessary angular velocity about the z-axis, θ_z, we divide the amount of rotation we want by our desired time to complete this phase:

$$\theta_{z,Command} = \frac{2\pi}{20} = \frac{\pi}{10}\text{rad/s}$$

With our angular velocity command defined, we include this in our ROS node cmd_interrupt_node.py. Here, we define a move phase and pirouette phase. During the former, the turtlebot will receive the motion commands from explore_lite as normal. During the latter period, the turtlebot recieves the angular velocity command we describe above. Below we display the pseudocode for this ROS node:

Algorithm 3 cmd_interrupt_node

$cmd\_pub \leftarrow rospy.Publisher('/cmd\_vel', Twist, queue\_size = 1)$ ▷ Publisher
$rospy.Subscriber('/scan', LaserScan, get\_scan\_data, queue\_size = 1)$ ▷ Subscribers
$rospy.Subscriber('/cmd\_vel\_intermediary', Twist, get\_command, queue\_size = 1)$

$MoveDuration \leftarrow rospy.duration(30)$
$PirouetteDuration \leftarrow rospy.duration(20)$
$MoveStartTime \leftarrow rospy.Time.now()$
$MoveEndTime \leftarrow MoveStartTime + MoveDuration$
if $rospy.Time.now() \geq MoveEndTime$ **then**
    $msg\_new \leftarrow Twist()$
    $msg\_new.linear.x \leftarrow 0$
    $msg\_new.angular.z \leftarrow 0.31415$
    $PirouetteStartTime \leftarrow rospy.Time.now()$
    $PirouetteEndTime \leftarrow PirouetteStartTime + PirouetteDuration$
    while $rospy.Time.now() \leq PirouetteEndTime$ **do** ▷ Perform a full 360°spin in place
        $cmd\_pub.publish(msg\_new)$
        $rospy.sleep(0.1)$ ▷ Prevent bot from being flooded with commands
        $MoveStartTime \leftarrow rospy.Time.now()$
        $MoveEndTime \leftarrow MoveStartTime + MoveDuration$
    end while
else
    $cmd\_pub.publish(msg)$ ▷ Forward along command from motion planner
end if

4 Demonstration and Findings

4.a RQT Diagram

With SLAM, motion planning, and AprilTag detection set up, we are ready to begin testing project design. In figure 3, we outline the active ROS topics as our project runs.

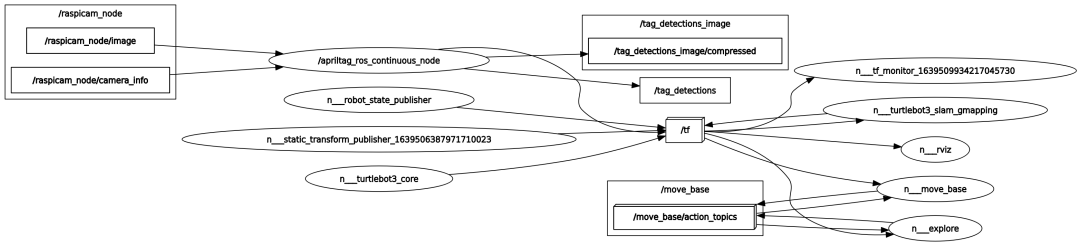

Figure 3: RQT graph of active rostopics as our project runs.

4.b Simulated Environment for Testing

A Gazebo world was created with an AprilTag to simulate the Search and Rescue operation. The reference world used was turtlebot3_house as shown in Fig 4.

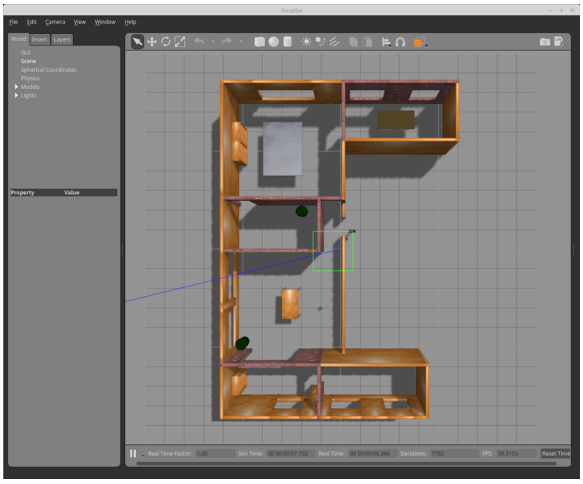

Figure 4: Layout of the simulated environment.

We introduced an AprilTag to a random location to ensure the turtlebot navigation and exploration design would be adaptive to differing scenarios. We used a closed-off map to ensure exploration is limited to just the bounds of the modeled house.

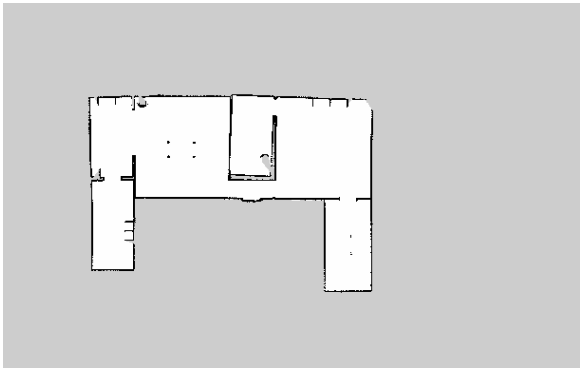

Figure 5: Map of the simulated environment (made using Gmapping).

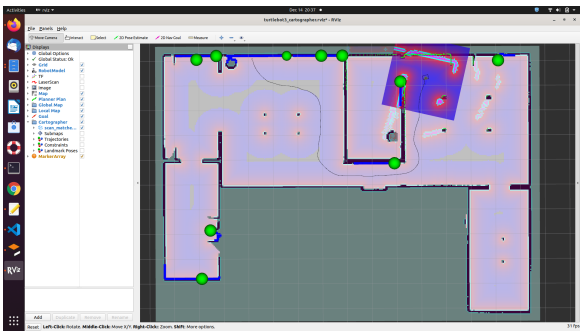

Figure 6: Vastly improved Map of the simulated environment made using cartographer.

4.c Environment Setup and Demonstration

A total of 15 AprilTags were printed of size .16 × .16 meters. Each tag is ascribed a unique ID numbered $0 - 14$, and all are members of the 36h11 family. The turtlebot is initially placed at a corner as shown in Fig. 7. Three demo runs were performed with the `minimum_frontier_size` parameter in `explore_lite` set to 20cm for the first run, and 40cm for second and final run.

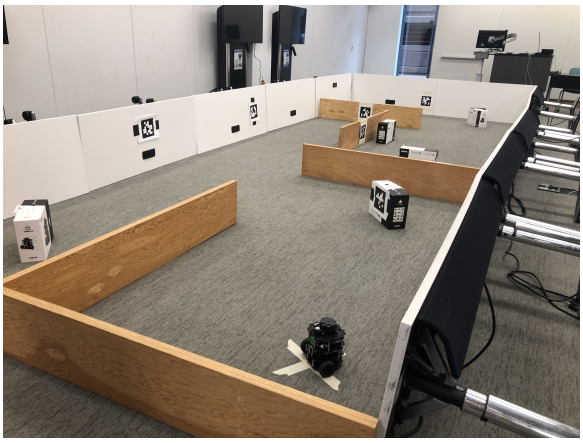

Figure 7: Search environment setup

As displayed in Fig. 8, 14 out of 15 tags were detected in all of the runs (Run 3 is displayed in Fig. 8. The final runtime was 7:23 minutes.

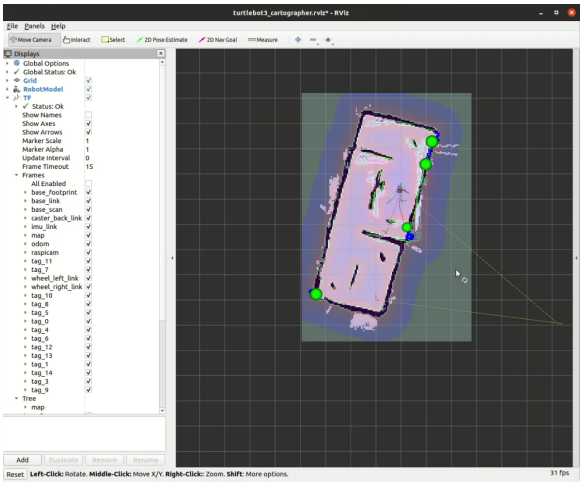

Figure 8: Rviz visualization of the generated map.

5 Discussion

This assignment has two primary goals: obtain a map of the environment and acquire the poses of all the AprilTags present. To tackle this problem, we started by obtaining the motion model for the turtlebot (3.a), allowing us to better understand how the turtlebot moves through its environment. Next, we researched and implemented the ROS package `cartographer_ros`, allowing us to map the environment and localize the robot (3.b). To move the turtlebot through the environment in a way

that would effectively create a full map and detect the AprilTags, we implemented the ROS package `explore_lite`. Using frontier-based exploration, this package ensured we create a comprehensive map. A glaring obstacle to our goals, however, was that the tags may be missed during the process of exploration, due to the limited field-of-view of the raspi camera. We combated this by writing a ROS node to make the robot spin after a set interval of time (3.d). Finally, we found the list of tag poses utilizing the ROS package `tag_detections`, along with our own ROS node to transform these tag positions and orientations to a global pose.

The bot was able to catch at most 14 tags during our runs at the official demo session. The 1 tag that was missed each time was not accessible due to our minimum frontier size constraint, and was in a location that prevented camera detection. Moreover, there was no position on the map where the robot could both have it in its field of view and be sufficiently far to include the entire tag in frame. For each trial the bot took between 6-7 minutes to complete the mission. Unfortunately recording the full rosbag while running slows everything down to the point of breaking functionality, so we could not get the full data saved.

Our methods for the motion planning and AprilTag detections have room for improvement, as we illustrate in the subsections below.

5.a Issues and Potential Future Improvements

5.a.1 Run Time and Map Improvements

We can improve the efficiency of the search and rescue mission by disabling the spinning when we are in a known part of the map and enabling it when we are close to new frontiers and exploring unknown spaces. This way, we reduce unnecessary pirouettes and can reduce the time of the mission. Another reason to reduce the number of pirouettes is to produce a better map, as spinning in place is not optimal for SLAM. A combination of translation and rotation of the robot produces more robust LiDAR data for Cartographer to utilize when updating the map.

5.a.2 Tag Pose Estimates

Our method of averaging tag poses to remove the noise is flawed; a linear averaging such as this would work fine for vector-like operands, but here we have poses in SE(3). This was an oversight on our part, as we only really considered the translation component of our poses as we checked them for accuracy. To end up with correct final estimates of the global poses of all tags, we would need to use a more abstract method such as least squares minimization of error, or utilize GTSAM.

5.a.3 Ensure All AprilTags are detected

Under our current method, there is no plan for a contingency where the tutlebot maps the full environment but does not detect all the AprilTags present. Since the LiDAR scanner has a range greater than the RaspiCam, and a 360°view, there's potential for Cartographer to completely map a

set of obstacles without the camera detecting the tags mounted. In this scenario, the `explore_lite` would not develop any frontiers in this area and consequently the robot would never approach them closely enough to detect these tags.

One approach to alleviate this issue would be to create an intermediary on the LiDAR data that would strip out all data that corresponds to a data footprint not visible by the camera. This would involve setting all measurements outside the camera's distance range of $(0.12, 2.5)$ to 0, and setting all LiDAR measurements outside the camera's FOV of $\pm \approx 35^\circ$ to 0. We chose not to implement this as it involves throwing away data, but it would force exploration by only allowing the map to be built using data the camera would have access to.

Finally, the method we would most likely implement were we to do this project again is to keep track of a second version of the map which is used to classify surfaces (boundaries between occupied and unoccupied cells) as having been seen in the camera's footprint or unseen. We would then need to modify our motion planner to actively seek out not only unexplored frontiers, but also to attempt to mark all surfaces as seen by the camera. This seems to us to be the best approach for guaranteeing all tags in an environment which are possible to detect will be recorded.

5.a.4 Driving Backwards and the Motion Model

The turtlebot is substantially more effective at driving forwards, with the bearing behind the wheels, than backwards. This discrepancy is aggravated on a carpeted or otherwise rough environment. In our demonstration, the bearing's housing scraped the carpet and did not rotate adequately. Additionally, driving backwards is not ideal for AprilTag detection, as the camera is oriented towards the front view of the robot. Initially, we attempted to alleviate the problem by setting the parameter `move_forward_only` to "`true`". However, this will not work with `explore_lite`, as this sends desired poses for `move_base` to calculate the proper motor commands. The poses it sends however, are assumed to be achievable from forward or backward movement. Therefore, some of the poses set by `exlpore_lite` will not be achievable when this parameter is set to "`true`", causing the robot to get completely stuck in a corner, unable to execute its computed trajectory.

We could avoid this problem by catching the motor commands with another intermediary ROS node. When a path would force the turtlebot to drive backwards, we can rotate the turtlebot 180 degrees and allow `explore_lite` to compute a new path, which will likely now be achievable via only forwards motion.