Disaster Response

1st Srinivas Peri
Electrical & Computer Engineering
Northeastern University
Boston, USA
peri.sr@northeastern.edu

2nd Sriram Pandi
Electrical & Computer Engineering
Northeastern University
Boston, USA
pandi.s@northeastern.edu

3rd Rishabh Singh
Electrical & Computer Engineering
Northeastern University
Boston, USA
singh.risha@northeastern.edu

4th Shruti Pasumarti
Electrical & Computer Engineering
Northeastern University
Boston, USA
pasumarti.sh@northeastern.edu

5th Sivashankar Venkatchalam
Electrical & Computer Engineering
Northeastern University
Boston, USA
venkatchalam.s@northeastern.edu

6th Srinidhi Pattala
Electrical & Computer Engineering
Northeastern University
Boston, USA
pattala.s@northeastern.edu

***Abstract*—This project delves into the growing significance of mobile robots in disaster response over the past two decades. By enhancing situational awareness and control in potentially dangerous environment (such as the interior of a collapsed building), these robots prove invaluable to emergency response teams. One major application of mobile robots is reconnaissance, where they are deployed to explore hazardous environments and identify potential threats and survivors (victims). The data gathered by mobile robots facilitates the formulation of effective action plans that minimize risks to emergency personnel.**

***Index Terms*—SLAM, ROS, AprilTags detection, gmapping, explore_lite**

Here are the required links of entire project :-

Platform	Links
Github	Github Repository
Youtube	SLAM/AT detection RViz Simulation
Youtube	SLAM/AT Detection
Youtube	Real Environment SLAM/AT Detection

I. Introduction

The project involves the concept an autonomous system using mobile robotics to perform reconnaissance in a potentially dangerous environment. The system will generate a complete map of the unexplored environment and locate any victims present in the form of AprilTags. The TurtleBot3, is equipped with a LiDAR scanner and a Raspberry Pi Camera, will be used to localization, mapping and AprilTags detection. The system will require the implementation of techniques such as mobile robotic kinematics, camera calibration, feature extraction, AprilTags detection, SLAM, and path planning. A successful system will provide a list of AprilTags with their ID numbers and poses with respect to the generated map.

II. Procedure

A. Outline

Refer to Fig.1

1) Place the robot in any new unknown environment.
2) Using gmapping, perform SLAM, to create a map of the environment and also track the position of the robot.
3) Using greedy frontier-based exploration technique autonomously drive the robot around the environment.
4) We will be using explore_lite package to generate the whole map of the unknown environment.
5) Detect AprilTags using "apriltag_ros" package and store their global poses.
6) Now, end the search after traversing the mobile robot in whole environment along with AprilTags detection.

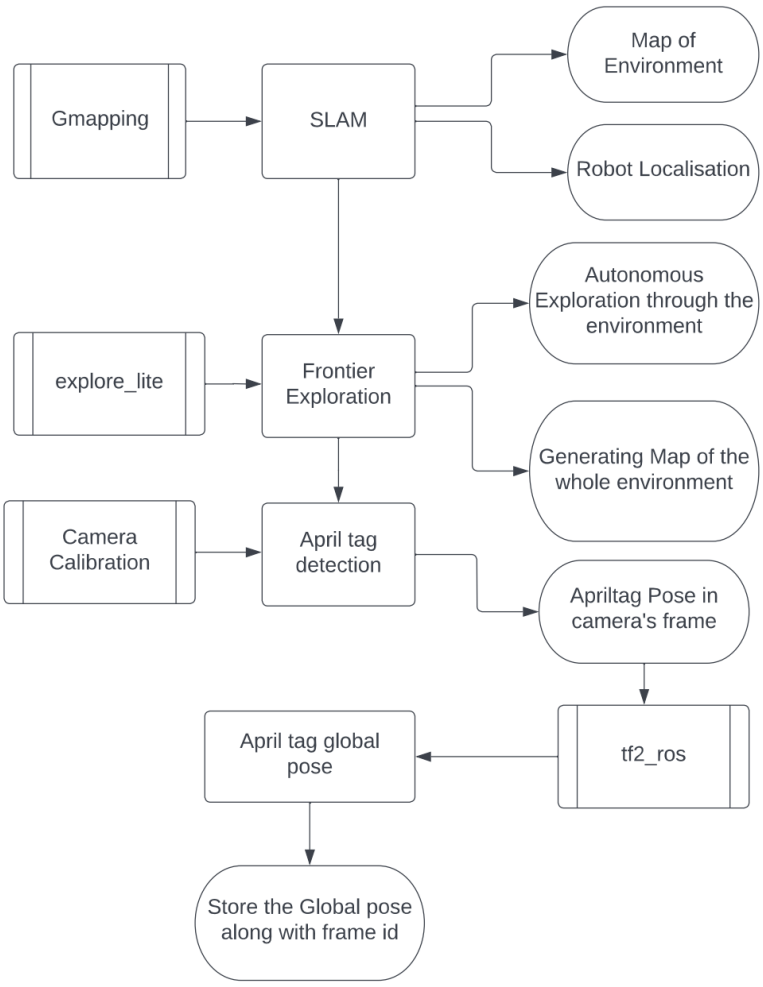

Fig. 1. Block Diagram

B. Steps to run the project

1) Firstly we need a personal computer operating on Ubuntu 20.04 and has ROS Neotic installed on it. The ROS master

Fig. 2. Architecture

node runs on the PC, which allows access to all nodes in the ROS environment. To establish a connection between the PC (as the master) and the turtlebot (as the slave), all devices need to be connected to the same local network provided by a phone hotspot. Additionally, the IP addresses of both the PC and the turtlebot need to be configured correctly to facilitate message exchange between them. Refer to Fig.2

2) Clone our Github Repository to both robot and the host PC. Command to connect the turtlebot via SSH:

```
$ ssh ubuntu@raspberrypi_IP
```

3) To launch all the required nodes on the turtlebot3:

```
$ export TURTLEBOT_MODEL=burger $ roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

4) To run the raspicamera on turtlebot3: SSH in a new terminal and run the below command to start raspberry pi camera.

```
$ roslaunch raspicam_node camerav2_1280x960_10fps.launch enable_raw:=true
```

5) Run SLAM on the host PC. We are using gmapping.

```
$ roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
```

This will enable the turtlebot to scan the room using gmapping.

6) To enable the AprilTag_ros pkg to subscribe to required topics and publish other topics such as /tf and /tag_detection_image.

```
$ roslaunch apriltag_ros continuous_detection.launch
```

7) To initiate the exploration_lite along with move_base and AMCL nodes.

```
$ roslaunch my_nav exploration.launch
```

8) Run a python script to store the detected apriltag pose in respect with the maps frame of reference.

```
$ rosrun my_nav static_transform.py
```

C. Deliverables

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

D. ROS Packages Utilized

1) gmapping : The gmapping package is a ROS wrapper for OpenSlam's Gmapping and it offers laser-based SLAM (Simultaneous Localization and Mapping). The package includes a ROS node called slam_gmapping that allows the creation of a 2-D occupancy grid map using laser and pose data obtained from a mobile robot. This map can be compared to a floorplan of a building.
2) explore_lite : This package provides a greedy frontier-based exploration method for a mobile robot. When the node is running, the robot will explore its environment in a greedy manner. The greedy approach prioritizes exploration of the nearest frontier first before moving on to others. This helps in faster exploration of the environment. It allows the robot to continue exploring until all frontiers have been explored. Once no frontiers remain, the robot will stop moving. The movement commands are sent to the move base, which is responsible for controlling the robot's movements. By using this package, the mobile robot can efficiently explore an unknown environment and create a map of the area for further analysis.
3) move_base : The move_base package enables a mobile robot to navigate towards a goal in the environment. It uses a global and local planner to achieve this task and can work with any planner that follows the nav_core package's BaseGlobalPlanner and BaseLocalPlanner interfaces. The package also maintains two costmaps for the global and local planner.
4) apriltag2_ros : This package subscribes to two input topics, camera_rect/image_rect and camera_rect/camera_info, to obtain an undistorted image and camera calibration matrix, respectively. The package's behavior is defined by two configuration files, and it outputs information about the detected tags in the environment through the tf and tag_detections topics. The tag_detections_image topic publishes the input image with the detected tags highlighted.
5) camera_calibration : camera_calibration package provides an easy way to calibrate monocular or stereo cameras with a checkerboard target. It uses OpenCV camera calibration and produces various calibration parameters. This package does not have a supported code API but uses the plumb_bob or rational_polynomial distortion model for pinhole cameras based on the number of parameters used.

E. Motion model

The Turtlebot3 is a differential drive robot with two parallel wheels that share an instantaneous center of curvature. Each wheel has a radius of 0.033 meters, while the track width (the space between the wheels) is 0.16 meters. This design restricts the robot's location to a two-dimensional plane, with no lateral movement in the YR direction. The robot's motion model is based on equations of motion that define the link between the angular velocities of the left and right wheels ($\dot{\phi}_l$ and $\dot{\phi}_r$, respectively) and the robot's linear

and angular velocities in the world frame, given its body orientation. Fig.3 depicts a differential drive robot concept. The robot's linear and angular velocities in the world frame, given the wheel angular velocities and body orientation, can be determined as follows:

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

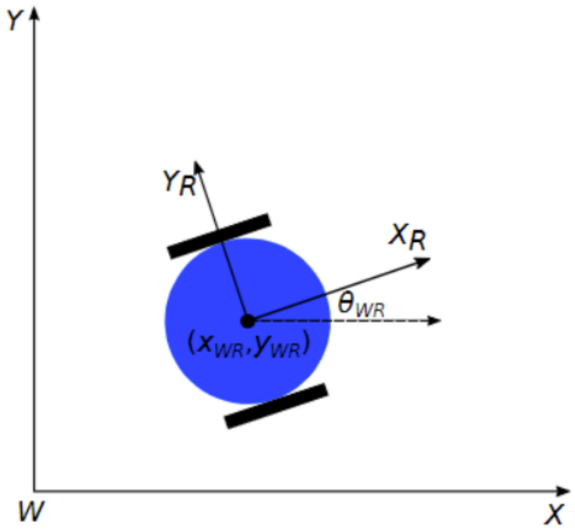

Fig. 3. Schematic of a differential drive robot

III. Implementation

A. SLAM Implementation

SLAM (Simultaneous Localization and Mapping) is a technique used in robotics to build a map of an unknown environment while simultaneously estimating the robot's position and orientation within that environment. The goal of SLAM is to enable a robot to navigate through an unknown environment without prior knowledge of its surroundings.The SLAM process typically involves using sensors, such as LiDAR or cameras, to collect data about the environment and the robot's movements. The data is then processed using algorithms to estimate the robot's pose and create a map of the environment. There are various approaches to SLAM, including filter-based methods such as the Extended Kalman Filter (EKF) and the Particle Filter, and graph-based methods such as the GraphSLAM algorithm.

We utilized the gmapping package for simultaneous localization and mapping. It is a widely used SLAM package with ROS integration that has the ability to handle loop closures.

Refer to Fig.4

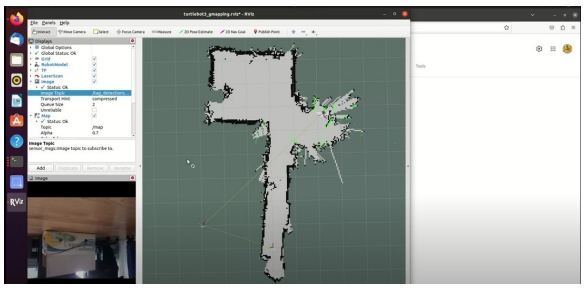

Fig. 4. Baseline implementation of SLAM

In the GMapping package, the Particle Filtering technique is employed to enable laser-based SLAM in the robot. This technique uses a Bayesian approach to update the probabilities of each cell in the map, indicating whether it is occupied or free. The package provides a ROS node called slam_gmapping, which transforms each incoming scan into the odometry (odom) tf frame and combines them to build a map of the robot's environment. To estimate the robot's position and orientation in the map, the node requires laser scan data and odometry data. As the TurtleBot moves through the environment, the node continuously updates the map and saves the resulting map to a file which can be visualized in RViz.

Since GMapping only utilizes laser scan data and odometry information for SLAM, it has certain limitations when it comes to AprilTag detection. As a result, we explored the Cartographer package to integrate data from multiple sensors, such as LIDAR, IMU, and cameras, resulting in more accurate and reliable SLAM. This multi-sensor fusion approach was particularly well-suited for detecting and tracking AprilTags, which require robust loop closure detection to accurately estimate the robot's position and orientation in complex environments.

B. Frontier Exploration

Frontier exploration is the most common method we use in mobile and autonomous robots to explore unknown spaces. In this, we utilize the explore_lite ROS package to implement frontier exploration. When the explore_lite node executes, the robot will greedily explore the unknown environment until no frontiers can be found. The occupancy map utilized by this package is obtained from the map. The explore_lite uses a greedy-based frontier approach and maximizes the exploration in unknown environments. The basic working principle of explore_lite mainly depends on the three possible outcomes: unoccupied, occupied, and unknown cells. Here frontiers are defined as boundaries between the explored and unexplored areas. The robot navigates towards these frontiers and explores them while constantly updating the map of the environment. Refer to Fig.5. When the node runs, the bot identifies

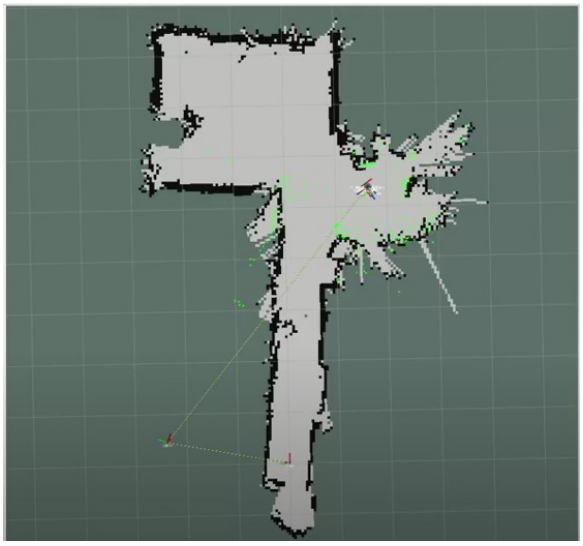

Fig. 5. Exploration Visualization in RViz

the unoccupied neighbors and decides the potential path. This can be done by using a cost map to determine which frontier the robot should navigate through and plan paths for the bot. The cost map assigns cost to each cell based on the obstacles, proximity, and other features in the environment, which navigates the robot to

avoid any obstacles. Then the bot will detect the frontier which has the least cost. Once the node detects a frontier, it sends movement commands to the bot's move_base, which controls its motion. The movement commands are sent in a way that ensures the bot moves toward the frontier, following a greedy approach. There are various parameters we tuned as a part of the explore_lite implementation which are explained here. min_frontier_size is used to filter out small frontiers.If the size of a frontier is less than min_frontier_size value the bot will not detect small frontiers and detect only large frontiers progress_timeout is maximum amount of time that the robot can spend trying to make progress towards a frontier before giving up and searching for a new frontier.If the robot spends more than the progress_timeout value trying to make progress towards a frontier, it will give up on that frontier and search for a new one.

Parameters	Initial Values	Final Values
planner frequency	0.33	0.33
progress timeout	15.0	30.0
potential_scale	4.0	3.0
orientation_scale	0.0	0.0
gain_scale	1.0	1.0
transform_tolerance	0.3	0.3
min_frontier_size	0.85	0.75

C. AprilTag Detection & pose estimation

To detect AprilTags, a Raspberry Pi Camera and the apriltag ROS package are used in combination. The apriltag ROS package processes these images to identify and track AprilTags in real-time. To enable the detection of AprilTags using the apriltag ROS package, the camera feed is provided as input to the software. The package then processes this feed and generates a list of detected tags along with their corresponding 3D locations. However, in order to accurately detect these tags, the software must be configured with information on the specific tags it is looking for and their respective sizes. This information is provided in two configuration files, namely config/settings.yaml and config/tags.yaml.
Refer to Fig.6. The apriltag ROS package utilizes default input

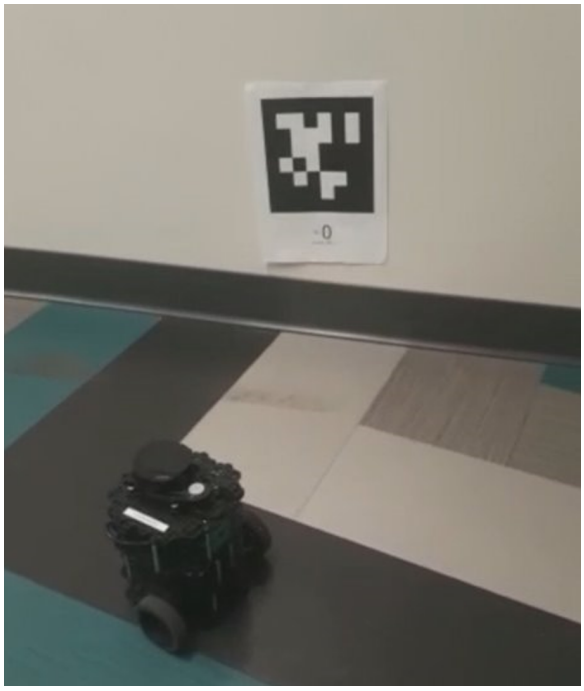

Fig. 6. April Tags Detection

topics to detect and track AprilTags.The default input topics are:

1) /camera_rect/image_rect: This is a sensor_msgs/Image topic that contains the undistorted image captured by the camera. It is assumed that the camera produces a pinhole image without distortion.
2) /camera_rect/camera_info: This is a sensor_msgs/CameraInfo topic that contains the camera calibration matrix in the /camera/camera_info/K parameter. This calibration matrix can be obtained by performing camera intrinsics calibration using tools such as camera_calibration.

When the ROS wrapper detects a tag or tag bundle, it outputs information on two topics: /tf and /tag_detections.

1) The /tf topic publishes the relative pose between the camera frame and each detected tag's or tag bundle's frame using tf (the transform library in ROS). This topic is only published if publish_tf is set to true in the settings.yaml file.
2) The /tag_detections topic provides the same information as the /tf topic, but in the form of a custom message that carries the tag ID(s), size(s), and geometry_msgs/PoseWithCovarianceStamped pose information. This topic is always published.
3) Finally, the /tag_detections_image topic publishes the same image as input by /camera/image_rect, but with the detected tags highlighted. This topic is only published if publish_tag_detections_image is set to true in the continuous_detection.launch file.

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
T_{AO} = Pose of AprilTag relative to Origin
T_{AC} = Pose of AprilTag relative to Camera
T_{CO} = Pose of Camera relative to Origin
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.

Baseline Implementation:
Initially, we utilized the gmapping package and the explore_lite package to establish baseline results. The output yielded a complete map of the environment, with 5 out of the 7 tags being detected in the map, then we tried it again with different environment and it detected 5 out of 5 tags. The following is the output of the of all the tags in homogeneous coordinate form stored in a text file. Refer to Fig.7 :

```
id: (1,)
[0.05621208631195284, -0.05590965810820254, -0.9963670177068269, 0.020914533923570541
[-0.18787179095605022, 0.774358432669091, -0.06898301465246216, 0.313311669430615]
[0.7752832230337277, 0.1912401925785385, 0.04026572271238579, 0.15406060343858868]
[0.0, 0.0, 0.0, 1.0]
----------------------------------------
id: (2,)
[-0.5804882170792418, -0.6994286154662839, -0.01585880850083376, -0.2976901446817279]
[0.015640098788129058, 0.004195744105379116, -0.9975381035765698, 0.33947777844811583]
[0.7009930595301760, -0.5819673360400949, -0.0026653220481262976, -0.41279171199647554]
[0.0, 0.0, 0.0, 1.0]
----------------------------------------
id: (17,)
[0.9559545071079593, 0.28933144056286597, -0.04908299041804792, 0.40828510233395271
[-0.051284828215461679, 4.744689632021282e-05, -0.9986747816169369, 0.3894943861198822]
[-0.2889446185271886, 0.957221648246843, 0.0148690723636053111, 2.8841386849900372]
[0.0, 0.0, 0.0, 1.0]
----------------------------------------
id: (3,)
[0.34450515923690556, 0.04171527583364762, 0.11259270252328189, -0.22670395678484664]
[0.04976246453109256, 0.07583019833058673, -0.9895370744165537, 0.5117942461111401]
[-0.0502513387017869, 0.34664935693809296, 0.022682434681153024, 1.957434801364849]
[0.0, 0.0, 0.0, 1.0]
----------------------------------------
id: (0,)
[0.33120578141704831, 0.9328368465678257, -0.022114557767020364, 1.7500188314763716]
[-0.14239366375908186, 0.027393126039592161, -0.9892983212370108, 0.5023279827632832]
[-0.9221840167303754, 0.33088056712521921, 0.14408786788071193, 2.1809128381142187]
[0.0, 0.0, 0.0, 1.0]
----------------------------------------
```

Fig. 7. Stored tag-id: Position and Orientation

IV. Demonstration & Findings

A. rqt diagram

Rqt graph is a very useful tool to see what's going on in our ROS graph. It's a GUI plugin from the Rqt tool suite. With rqt graph we can visualize the ROS graph of your application.
On one window you can see all our running nodes, as well as the communication between them. The nodes and topics will be displayed inside their namespace.
When we develop with ROS we usually organize our work into packages and nodes. Refer to Fig.8, We'll have more and more nodes, with more and more communication between them (topics, services, actions). Using rqt graph will help us mostly for those two things:

- We'll get a global overview of our system. This is really useful so we can take better decisions for the future new parts of our application.
- When we have a bug somewhere due to communication between nodes, we'll be able to easily spot the problem.

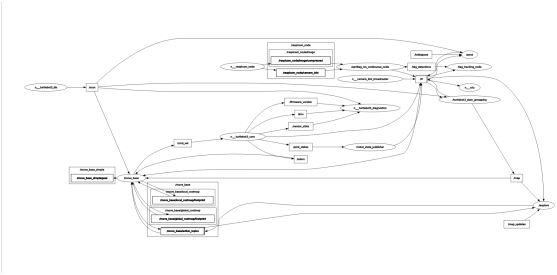

Fig. 8. Rqt Diagram

B. Simulated Environment for testing

Initially, we used off-the-shelf packages without modifying their parameters and tested them in the Gazebo Simulator. We created a Gazebo world with the seven Apriltags randomly located to simulate a Search and Rescue operation. The turtlebot3_house environment was used as a reference, as shown in Fig 9. In order to restrict the exploration to only the boundaries of the modeled house, we utilized a map that was closed off.

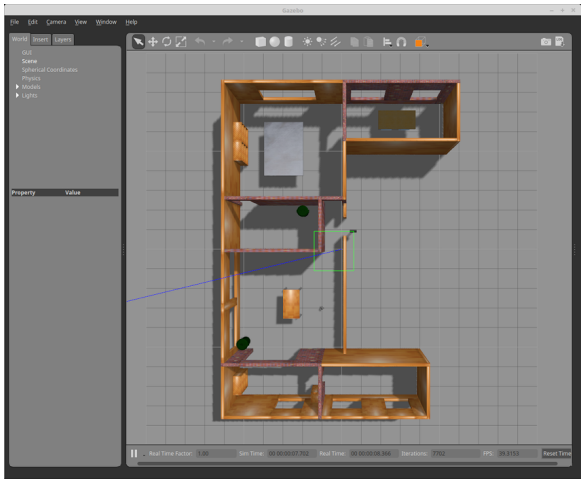

Fig. 9. Simulated Environment

C. Real Environment setup & demonstration

Seven AprilTags, each belonging to the 36h11 family and measuring 0.16 x 0.16 meters, were positioned at various heights and orientations, each having a distinct ID number from 0 to 6. The starting position of the turtlebot was at the entrance. In addition to the AprilTags, a total of 5 obstacles were placed during the test. Refer to Fig.10

Fig. 10. Panoramic view of Environment (a) Setup

We tested our bot in 2 different unknown environments, below is the final map generated of both the environments. Refer to Fig.11 and Fig.12

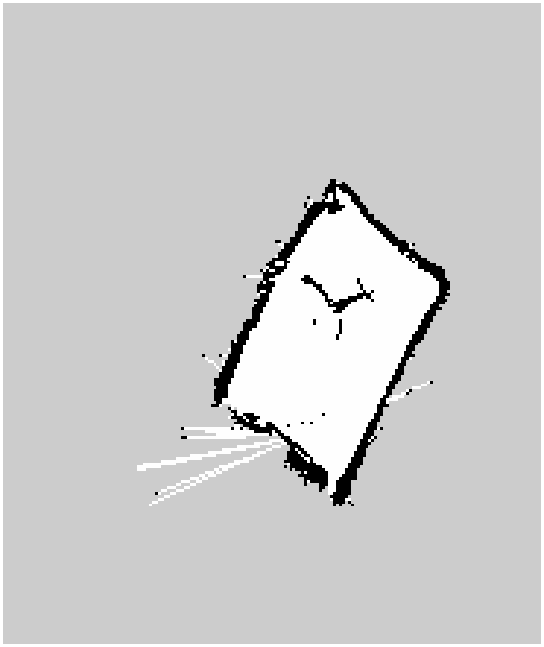

Fig. 11. Final Map generated in Environment (a)

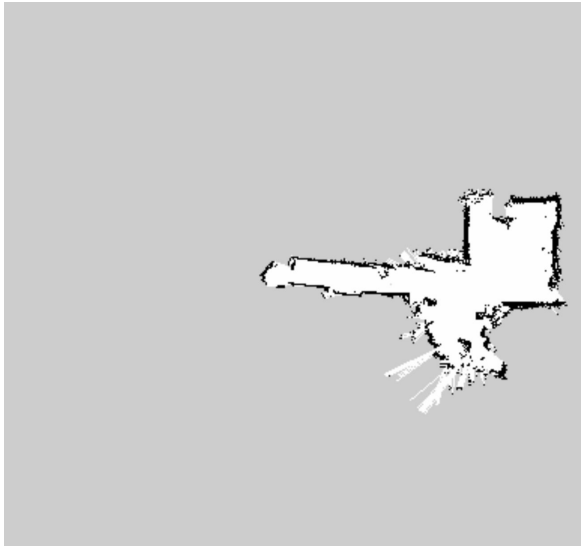

Fig. 12. Final Map generated in Environment (b)

Below is the presentation of RViz simulation when our bot started detecting and storing the AprilTags ids. Refer to Fig.13.

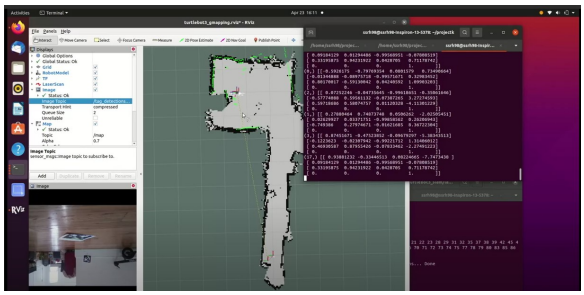

Fig. 13. RViz Visualization with Tag id detection

D. Challenges & limitations

1) Synchronization problem with raspi_cam caused difficulty in obtaining accurate and timely data from the camera, which affected the detection and tracking of AprilTags. We had to experiment with different settings and parameters to achieve reliable synchronization.
2) Default settings of the Lidar HLDS were not sufficient for accurate detection and mapping of the environment. We had to optimize the parameters of the Lidar to improve its performance and ensure that all obstacles and objects in the environment were detected and mapped correctly.
3) To estimate the pose of the AprilTags with respect to the base link or map's reference frame, we used the AprilTag ROS package to detect the tags and obtain their transforms. We then converted these transforms from the camera's reference frame to the base link's reference frame using a node written with the tf2_ros package. Initially, we used a Python script to estimate the pose, but later found alternative methods including measuring the length, width, and height (translation) of the camera from the base link and passing these values to obtain the position and pose of the AprilTags in the base link's reference frame.
4) When the LiDar emits a laser beam, it encountered transparent object (glass), the beam got refracted or bent, rather than reflecting back to the LiDar. So, this made it difficult for the LiDar to accurately detect the glass as obstruction, as it relies on the beam bouncing back to measure the distance. Refer to Fig.14

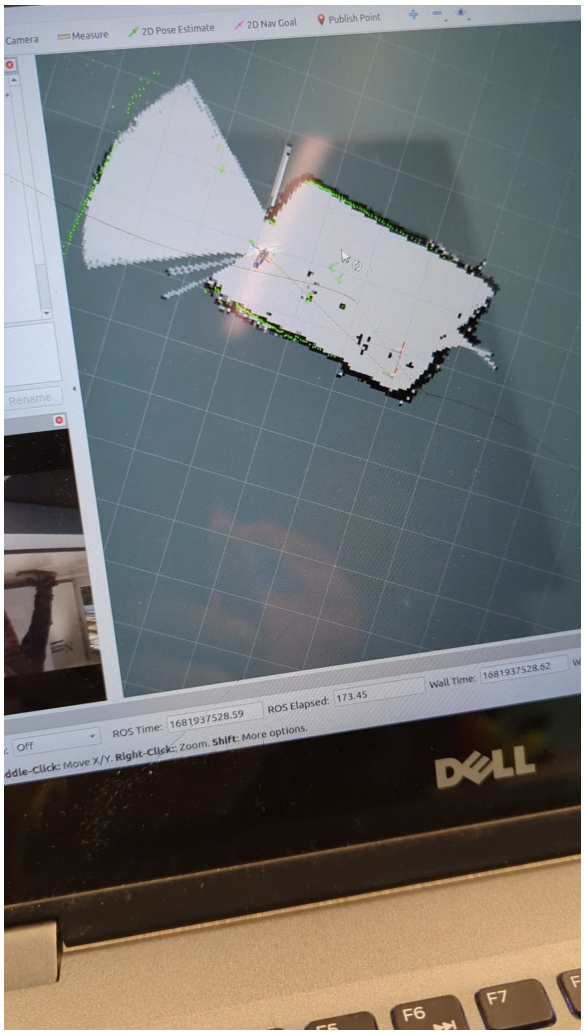

Fig. 14. LiDar not detecting glass as obstruction

V. Conclusion

Our primary objectives in this assignment were to generate a complete map of the environment and report the ID and pose of the detected Apriltags. Initially, we utilized the gmapping and explore_lite ROS packages to guide the turtlebot and map the environment. However, due to the default LIDAR settings and the robot's limited field-of-view, only five out of the seven Apriltags were detected. To improve the accuracy of our system, we explored the cartographer package as a solution but it is a recent development in our project. With this improvement, we will be able to successfully achieve both objectives and generate a complete map of the environment while detecting all seven Apriltags.

VI. Future Plans

1) Sensor fusion: To overcome the lidar's difficulty in detecting transparent objects like glass, alternative sensors like depth cameras, ultrasonic sensors, or radar can be used in conjunction with lidar, or multi-sensor fusion can be applied to combine data from different sensors and obtain a more accurate and complete view of the environment
2) 360° camera : Using a 360-degree camera can be a possible solution for the limited view of lidar. A 360-degree camera can capture images from all directions, providing a more comprehensive view of the environment. By combining the data from the 360-degree camera and the lidar, a more accurate and complete map of the environment can be created. The camera can also help in detecting transparent objects like glass, which can be difficult for the lidar to detect.
3) ML techniques: We can use Machine learning techniques to teach the robot to recognize transparent objects like glass based on their properties, such as their shape, size, and refractive index. This can help the robot to detect and avoid glass obstacles more accurately and effectively.

Acknowledgment

First and foremost, we would like to thank our Prof. Michael Everett for his invaluable guidance and support throughout the project. His class teachings knowledge, expertise, and insights have been instrumental in shaping the direction of the project and helping us overcome challenges along the way. We would also like to extend our heartfelt thanks to our TAs, Kishore Reddy Pagidi and Arvinder Singh who have assisted us and solved our doubts during

this project. Their collaboration, ideas, and support have been vital in achieving the objectives of the project. Finally, We express our gratitude to all the resources such as libraries, online resources, and research papers, which we referred to during this project.

TABLE I
ROLES & RESPONSIBILITIES

Names	Responsibilities
Srinidhi Pattala	Configuration and SBC setup
Shruti Pasumarti	Report, SLAM (gmapping) implementation
Sivshankar Venkatchalam	Report, SLAM (gmapping) implementation
Rishabh Singh	Report, PPT, AprilTags Detection
Sriram Pandi	PPT, AprilTags Detection
Srinivas Peri	SLAM and AprilTags detection

REFERENCES

[1] John Wang, and Edwin Olson, "AprilTag 2: Efficient and robust fiducial detection," IEEE International Workshop on Intelligent Robots and Systems (IROS), 10.1109/IROS.2016.7759617, 01 December 2016. **https://ieeexplore.ieee.org/document/7759617/authors#authors**

[2] Haileleol Tibebu,Jamie Roche,Varuna De Silva,and Ahmet Kondoz, "LiDAR-Based Glass Detection for Improved Occupancy Grid Mapping,"Institute of Digital Technologies, Loughborough University London, 3 Lesney Avenue, London E20 3BS, UK,10.3390/s21072263,24 March 2021. **https://www.mdpi.com/1424-8220/21/7/2263**

[3] Github:Kevin-Robb,Repository:eece-5550 **https://github.com/kevin-robb/eece-5550**

[4] Github:YuehChuan,Repository:tb3_aprilTag **https://github.com/YuehChuan/tb3_aprilTag**

[5] John Wang, and Edwin Olson, "Simulation of Indoor Localization and Navigation of Turtlebot 3 using Real Time Object Detection," IEEE, 10.1109/CENT-CON52345.2021.9687937, 03 February 2022. **https://ieeexplore.ieee.org/document/9687937**

[6] **https://blog.jethro.dev/posts/camera_calibration_apriltag/**

[7] Anirudh Topiwala,Pranav Inani,and Abhishek Kathpal, "Frontier Based Exploration for Autonomous Robot,10.13140/RG.2.2.34130.40641,January 2018. **https://arxiv.org/ftp/arxiv/papers/1806/1806.03581.pdf**

[8] rqt-diagram **https://arxiv.org/ftp/arxiv/papers/1806/1806.03581.pdf**