Autonomous Disaster Response & Reconnaissance using TurtleBot 3

Deep Kotadiya
College of Engineering
Northeastern University
Boston, USA
kotadiya.d@northeastern.edu

Vishnu Rohit Annadanam
College of Engineering
Northeastern University
Boston, USA
annadanam.v@northeastern.edu

Hussain Kanchwala
College of Engineering
Northeastern University
Boston, USA
kanchwala.h@northeastern.edu

Nikhil Chowdary Gutlapalli
College of Engineering
Northeastern University
Boston, USA
gutlapalli.n@northeastern.edu

Aakash Singhal
College of Engineering
Northeastern University
Boston, USA
singhal.aak@northeastern.edu

Satvik Tyagi
College of Engineering
Northeastern University
Boston, USA
tyagi.sa@northeastern.edu

***Abstract*—Mobile robots have become essential in disaster response, providing emergency teams with better situational awareness and physical control in hazardous environments. This paper explores the use of TurtleBot3, a popular mobile robot platform, in disaster response and reconnaissance. The proposed solution implements ROS (Robot Operating System) packages such as *explore_lite*, *move_base*, *cartographer_ros*, and *apriltag_ros*, along with custom nodes, to achieve the project's objectives. The system design incorporates a reliable and efficient TCP/IP protocol for data transmission between the TurtleBot3 and a master PC, enabling real-time control and data processing. The paper discusses the challenges associated with deploying mobile robots in complex environments, including communication and control, and how ROS technology can streamline rescue operations and improve the safety of emergency responders.**

- GitHub Codebase: Click_here.
- YouTube Video: Click_here.

I. Introduction and Motivation

In recent years, mobile robots have gained significant attention for their role in disaster response. These robots have proven to be valuable assets for emergency response teams, providing them with improved situational awareness and physical control in hazardous environments. One particular application that has garnered significant interest is the use of mobile robots for reconnaissance. In this scenario, emergency responders deploy mobile robots to quickly survey hazardous environments, such as the interior of a collapsed building. These robots can identify potential hazards and the location of victims, enabling emergency workers to develop an effective action plan that minimizes the risk to themselves and others.

This paper focuses on the role of mobile robots in disaster response and their application in reconnaissance. We explore the capabilities of mobile robots in identifying hazards and locating victims in hazardous environments. We also examine the challenges associated with deploying these robots, including communication and control in complex environments. Additionally, we discuss the advancements in mobile robot technology by implementing existing ROS packages such as *explore_lite* [1], *move_base* [2], *cartographer* [3], and *apriltag_ros* [4] etc. Furthermore, we have designed custom nodes to achieve the project's objectives and discussed the use of GTSAM (Georgia Tech Smoothing and Mapping Library) [5] to improve the pose of the robot and April tags on the map. By implementing such systems, rescue operations can be streamlined, and the risks to rescuers can be significantly reduced. Mobile robots equipped with ROS can quickly and efficiently survey hazardous environments, identifying potential hazards and the location of victims. This information can then be used to develop an effective action plan that minimizes the risk to emergency workers.

Overall, the development and implementation of ROS with custom nodes for mobile robots (For this project, we have used Turtlebot3) in disaster response can improve the efficiency and safety of rescue operations. This, in turn, can help to save lives and reduce the costs associated with rescue efforts.

II. Proposed Solution

TurtleBot3 is a popular and versatile mobile robot platform used for research and educational purposes. The communication between the TurtleBot3 and the Master PC is established using a router with a 5GHz frequency to minimize latency issues. By leveraging the TCP/IP protocol for data transmission, this setup ensures a reliable and efficient connection between the two devices, which is essential for real-time control and data processing in robotic applications.

On the TurtleBot3, hardware drivers for components such as LIDAR, camera, IMU, and motors are implemented directly on the robot. Meanwhile, the Remote PC handles other nodes responsible for tasks such as occupancy map

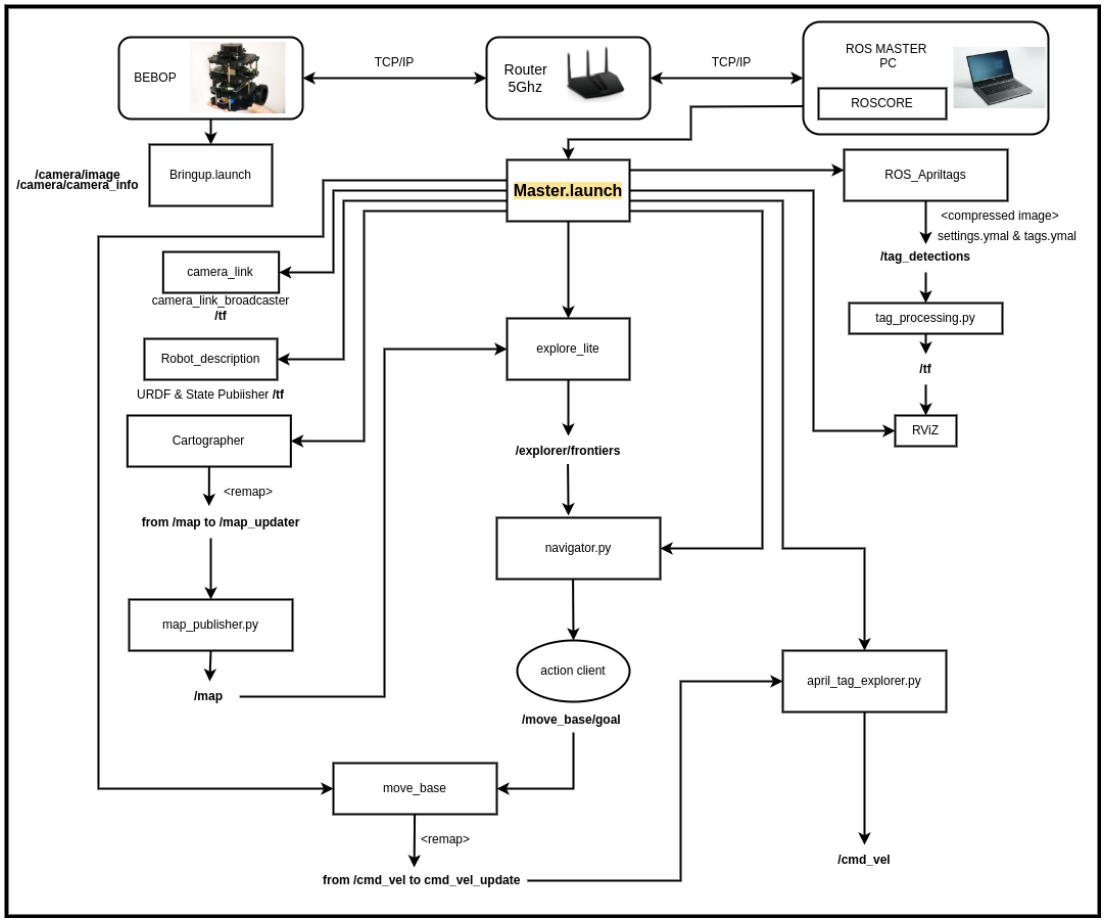

Fig. 1. High Level Flowchart

generation, exploration and navigation, as well as AprilTag [] detection and transformation. The division of responsibilities between the TurtleBot3 and the Remote PC allows for more efficient operation and better resource allocation. To streamline the process of launching various nodes, a master launch file is created. This launch file initiates all the required nodes in a specific sequence, ensuring that each component starts up correctly and in the proper order. This organized approach simplifies the process of starting the entire system, improving overall efficiency and reducing the potential for errors during initialization.

The occupancy map generated by Cartographer [3] is modified by the *map_republisher* node to produce an occupancy grid map with cells having discrete values of [100, 0, -1]. This modified map is then utilized by the *explore_lite* node for frontier detection. A custom node named *navigator* is implemented, which evaluates a set of conditions to switch between a wall-following algorithm and *explore_lite* for goal generation. These goals are subsequently passed to the *move_base* goal client.

To facilitate the detection of AprilTags while the robot is navigating, an additional node called *april_tag_explorer* is launched. This node periodically interrupts the command velocity generated by *move_base* to rotate the robot at its stationary position, enabling tag detection. Before launching the navigation and exploration nodes, the *apriltag_ros_continuous_node* [4] and the custom *tag_processing_node* are initiated to detect AprilTags and obtain their transforms in the global map frame.

III. System Design

A. Turtlebot 3 and Camera setup

The Turtlebot3 is a differential drive robot designed for versatile performance in various environments. This compact robot is equipped with two parallel wheels that share a common instantaneous center of curvature, enabling smooth and precise movement. Each wheel has a radius of r = 0.033 meters, while the track width, the distance between the wheels, measures w = 0.16 meters. We have incorporated a static transformation for the camera into Turtlebot's URDF (Unified Robot Description Format) file, enabling the necessary transformations for detecting and identifying April tags in the robot's environment. By adding the camera_joint and camera_link to the Xacro.urdf file, we have successfully

integrated the camera with the robot's kinematic chain. The improved URDF file ensures seamless integration of the camera's coordinate frame with the robot's coordinate system, allowing for accurate and reliable transformations between the camera frame and the robot frame. Consequently, this addition enhances the Turtlebot's overall performance in tasks requiring visual perception and spatial awareness.

The ROS tutorial for Monocular Camera Calibration [6] was followed to begin the calibration process. A 9 × 7 checkerboard was printed and an ost.yaml file was obtained containing the calibration data. The calibration process resulted in the generation of an ost.yaml file, which contains critical calibration data, such as the camera matrix, distortion coefficients, and rectification parameters. This file is instrumental in correcting any distortion present in the camera's images and improving the accuracy of the visual data used for tasks like object detection, tracking, and pose estimation.

B. Occupancy Grid Map

The Cartographer [3] generates a occupancy grid map with each cell having a probability of being occupied or unoccupied in a range of 0 to 100 in addition to -1 for the unknown space. But when explore_lite [1] subscribes to the map, it expects an occupancy grid map having discrete cell values of [100,0,-1] denoting occupied, unoccupied and unknown cells. A custom node *map_republisher* is built that converts the occupancy grid map from the cartographer to the one that can be interpreted by *explore_lite*.

The idea is to set particular thresholds for known and unknown areas in the occupancy grid map in order to convert the cell values to a binary representation of occupied and unoccupied cells along with a value of -1 for unknown cells. For the sake of this project, the OBSTACLE_THRESHOLD and UNKNOWN_THRESHOLD are set to 75 and 50 respectively.

Algorithm 1 Map Republisher

- Set: OBSTACLE_THRESHOLD,UNKNOWN_THRESHOLD
- Initialize: rospy node, subscriber, publisher

function PROCESS_MAP_DATA(data, width, height)
- Initialize: processed_data

for each grid cell (x, y) in the occupancy grid **do**
- Compute: index i based on x, y, and width
- Get: cell_value from data at index i

if cell_value $\geq$ OBSTACLE_THRESHOLD **then**
- Append: 100 to processed_data

else if $0 \leq$ cell_value $<$ UNKNOWN_THRESHOLD **then**
- Append: 0 to processed_data

else
- Append: -1 to processed_data

end if

end for
- **return** processed_data

end function

function CALLBACK(map_carto)
- Call: process_map_data with map_carto's data, width, and height
- Update: map_carto.data with processed_data
- Publish: map_carto to '/map' topic

end function

function RUN
- Start: rospy.spin()

end function

C. Exploration and Navigation

In order to explore the environment, we primarily utilized the greedy frontier exploration algorithm provided by the explore_lite [1] package. This algorithm effectively identifies and navigates towards the unexplored regions of the environment. Additionally, to ensure efficient exploration, we implemented a custom node called *navigator*, which incorporates a wall following algorithm. The navigator node seamlessly switches between the explore_lite and wall following algorithms based on specific conditions encountered during exploration.

Specifically, the node utilizes data from the robot's sensors, including the 360° Lidar and IMU, to detect the proximity of obstacles and the robot's orientation with respect to the environment. Based on this information, the node makes decisions about when to switch between the two algorithms to maximize the efficiency and accuracy of the exploration process. Overall, this approach provides a robust and effective means of exploring complex environments and locating potential victims which in our case are represented by April Tags.

The node subscribes to *map, scan* and *odom* topics from where it receives occupancy grid map, laser data and odometry data. The *action client* is then initialized to send commands to *move_base* once the goal position is determined and the *min_obstacle_distance* parameter is also initialised. A check parameter named *exploring* is initialized with a *false* value. The *GENERATE_GOAL* function consists of the main algorithm. The laser scan data is divided into regions *right, front_right, front, front_left* and *left*. The Odometry data is then used to calculate the orientation of the Bot in Euler system. The following conditions are checked:

- Case1: If the minimum range of laser scan in all the regions is greater than the *min_obstacle_distance*, then the *explore_lite* frontier-based detection is used. The *exploring* flag is set to *True*.
- Case2: If the minimum range of laser scan in the front region is less than the *min_obstacle_distance*, then the robot is made to rotate 90° left from its current orientation so as to avoid collision.
- Case3: If the minimum range of laser scan in the front_right region is less than the *min_obstacle_distance* then we consider this to be a wall and then set $x_{goal} =$

Algorithm 2 Navigator

procedure INITIALIZATION
- Initialize map, scan, and pose data
- Set exploring flag to False
- Set default values for ROS parameters
- Subscribe to topics and create action client

end procedure
procedure MAP_CALLBACK(msg)
- Update map data

end procedure
procedure SCAN_CALLBACK(msg)
- Update scan data

if not exploring **then**
- Publish goal

end if
end procedure
procedure ODOM_CALLBACK(msg)
- Update current pose

end procedure
procedure GENERATE_GOAL
if scan and pose data available **then**
- Classify regions based on distance
- Generate goal pose based on region conditions
- Check if goal is within map boundaries
- Return goal if valid, else switch to exploration mode

else
- Return None

end if
end procedure
procedure PUBLISH_GOAL
if not exploring **then**
- Generate and send goal

end if
end procedure
procedure RUN
while not rospy.is_shutdown() **do**
if not exploring **then**
- Publish goal

end if
- Sleep for the specified rate

end while
end procedure

Algorithm 3 AprilTag Explorer

procedure INITIALIZATION
- Initialize ROS node, cmd_pub, move_start_time, move_duration, rotation_duration, and scan_data
- Subscribe to topics

end procedure
procedure GET_SCAN_DATA(msg)
- Update scan data

end procedure
procedure GET_COMMAND(msg)
if Time now $\geq$ move_start_time + move_duration **then**
- Create a new Twist message with rotational velocity
- Set rotation start and end times

while Time now $\leq$ rotation_end_time **do**
- Log info and publish rotation command
- Sleep for 0.1 seconds

end while
- Update move_start_time

else
- Publish received command

end if
end procedure
procedure RUN
- ROS spin

end procedure

$x_{current} + cos(yaw)$ and $y_{new} = y_{current} + sin(yaw)$.

- Case4: Any case other than the above three mentioned would trigger the *explore_lite* and set *exploring* flag to *True*.

The *GENERATE_GOAL* Function also checks whether the goal generated by Case2 or Case3 is in the map boundaries or not. In the case where goal is not in the map boundary, the *exploring* flag is set to True and *explore_lite* is activated to generate goals. The *PUBLISH_GOAL* Function checks if the *exploring* flag is set to *False*, then the goal position which is generated by either Case2 or Case3 is within the map boundaries and the goal positions are passed to the *move_base Action Client* which generates path as well as command velocity to reach the goal.

To increase the chances of the April Tags being detected in the environment a custom node is written with the name *april_tag_explorer*, this node interrupts the exploration at specific time intervals, and rotates the bot at a stationary position for a specified time interval so that the April Tag is seen if it is in the camera range. In this case, the exploration is interrupted at an interval of every 30 seconds. To achieve our goal of rotating the robot around its z-axis, we calculate the necessary angular velocity, denoted as $\theta_{z,Command}$. Since we only need rotation about the z-axis, all other angular velocities are set to zero. We can determine $\theta_{z,Command}$ by dividing the desired amount of rotation by the desired time to complete this phase. For example, if we want to rotate 360 degrees in 20 seconds, then:

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

The commands from *move_base* are remapped to *cmd_vel_update* which is subscribed by the *AprilTagExplorer* alongside *scan* topic. The *GET_COMMAND* Function checks if the current system time is greater than *move_start_time* + *move_duration* then a new twist message with the angular velocity of $\frac{\pi}{10}$ rad/sec is generated. If the current system time is less than *rotation_end_time* then the new twist message that was created is published on *cmd_vel topic* or the twist message from *move_base* is published to *cmd_vel* topic.The

RUN Function starts the node and spin until the node is shutdown.

D. April Tag Detection and Transformation

To detect the apriltags, we have utilized the *apriltag_ros* package that takes feed from the onboard Raspberry PI camera of the TurtleBot. To ensure that the camera can detect the AprilTags, the *settings.yaml* and *tags.yaml* files were edited with updated parameters of the printed Apriltags. These configuration files can be found in the config folder, which can be accessed from the terminal using *roscd apriltag_ros*. The edited configuration files specify the family of tags used, which is *36h11*, the IDs of all expected tags, as well as the physical sizes of the tags present in the environment.

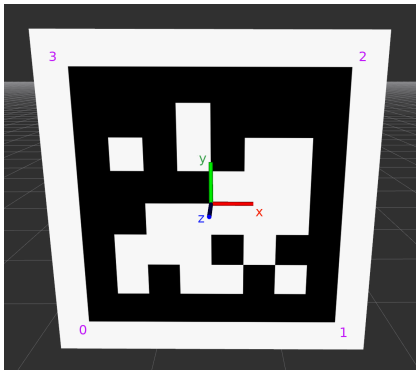

Fig. 2. April Tags family 36h11

By using the video feed of that camera, *apriltag_ros* provides the positional transforms of the detected apriltags with respect to the camera. But in the context of detecting people in a disaster environment, it was necessary to obtain transforms with respect to the global map. To achieve this, we implemented an custom node which converts the position relative to the origin frame using transformation matrix as well as broadcasts the detected april tags continuously to *tf2* topic so that it is continuously represented on the map.

The *TagTracker* class is designed to transform the tags relative to global map origin,track tags and broadcast their pose information in a ROS system. It initializes global variables, sets up a file path for saving tag data, and initializes tf buffer, listener and broadcaster. It subscribes to topic *tag detections* and create a timer callback for periodic updates. When a new tag detection message is received, it computes the transformation matrices between tags, camera and origin frames, and updates the tag dictionary accordingly. It also provides functions for looking up and publishing *tf* information, as well as saving tags to a file. The formula [7] below transforms the april tag orientation and position from camera frame to Map origin frame.

$$T_{AO} = T_{AC} \cdot T_{CO} \tag{1}$$

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

Algorithm 4 TagTracker

- Initialize: filepath, DT, tags, TF_ORIGIN, TF_CAMERA
- Create: tf_buffer, tf_listener, tf_broadcaster
- Generate: filepath based on datetime
- Subscribe: to ”/tag_detections” topic
- Initialize: timer callback

function GET_TAG_DETECTION(tag_msg)
  if tag_msg.detections is empty **then**
    return
  end if
  for each detection in tag_msg.detections **do**
    - Extract: tag_id, tag_pose
    - Compute: T_AC

    if T_AC is None **then**
      - Print: ”Found tag, but cannot create global transform.”

      return
    end iftag_id in self.tags.keys()
    - Print: ’UPDATING TAG:’, tag_id
    - Set: L = 0.9
    - Update: self.tags[tag_id] using weighted sum of current and new T_AO

    - Print: ’FOUND NEW TAG:’, tag_id
    - Add: new tag to self.tags with T_AO value

  end for
end function
function GET_TRANSFORM(TF_TO, TF_FROM)
  - Compute: pose by looking up transform
  - Print: ”Transform not found.”

  return None
  - Compute: transformT, transformQ

  return Transformation Matrix
end function
function TIMER_CALLBACK(event)
  - Call: publish_tf()
  - Call: save_tags_to_file(tags)

end function
function PUBLISH_TF
  for each tag_id, T_AO in tags **do**
    - Create: TransformStamped t
    - Set: t’s attributes
    - Send: Transform t with tf_broadcaster

  end for
end function
function SAVE_TAGS_TO_FILE(tags)
  if tags is empty **then**
    return
  end if
  - Prepare: data_for_file
  - Save: data_for_file to filepath

end function
function MAIN
  - Initialize: rospy node
  - Create: tag_tracker
  - Start: rospy.spin()

end function

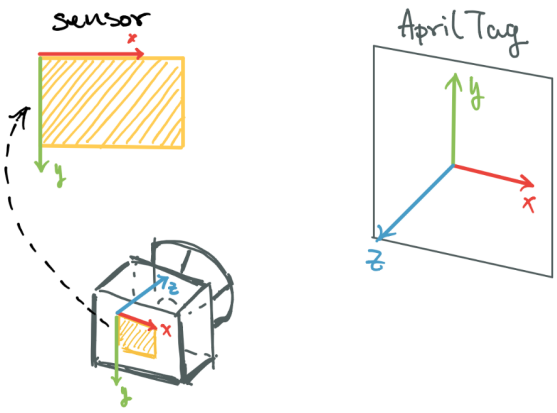

Fig. 3. April Tags in environment w.r.t. sensor co-ordinates

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}. 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 below this acts like a complimentary filter.

$$\text{tags\_id} = L \odot \text{tags\_id} + (1 - L) \odot T_{\text{AO}} \tag{2}$$

where L is the complimentary coefficient that we set to 0.9.

IV. Results

A. Environment Setup

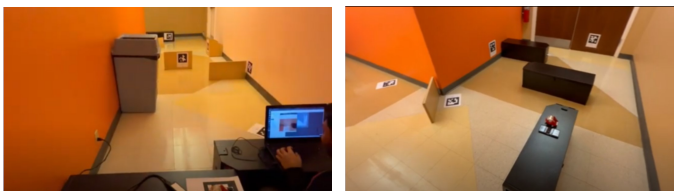

Fig. 4. Test Environment Setup

In order to conduct a comprehensive evaluation of the system, two arenas were set up with different levels of complexity. Trash cans and wooden boards placed in the arena as the obstacles for the bot. A total of 11 April Tags with unique tag IDs ranging from 0 to 10 were randomly placed on walls, obstacles, and the floor at different orientations and heights. The purpose of these obstacles was to obstruct the direct line of sight of the camera and Lidar, allowing for testing of the system's performance to map and detect April Tags under various conditions.

To further test the limits of the bot and the algorithm, the second arena was made even more challenging by adding additional obstacles, further obstructing the direct line of sight of the camera and making the environment more difficult to maneuver.

B. Testing

During the comprehensive evaluation of the system, it was observed that the system successfully detected all 11 April Tags in the first arena. The robust perception system enabled it to detect the April Tags even when they were placed on the floor and obstructed by wooden boards, demonstrating its reliability. However, on the second arena, it failed to detect one out of the 11 tags in the arena. Although the tag was detected in the first round, in second arena however, had an obstacle blocking the light falling on the tag that was placed on the ground. The system's ability to operate effectively in low lighting conditions can be further improved by upgrading to a better camera that can handle low-light conditions better.

The custom exploration node, optimized for efficient exploration of unknown environments, was highly effective in enabling the bot to navigate through the challenging testing environment, and capture camera feed for April Tag detection, even those that were not immediately visible. The occupancy map generated by the system was highly accurate, providing a precise representation of the orientation and location of the April Tags. Overall, the system demonstrated high performance and reliability in this challenging testing environment.

For a visual demonstration of the system's capabilities, please refer to the video for which a link is provided at the start of this paper.

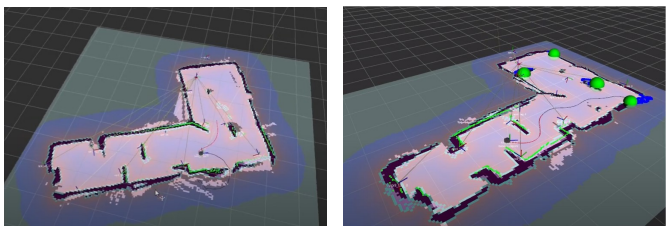

Fig. 5. Map of the Environment with April Tags

V. Conclusion

In this work, we presented our approach to obtaining a map of the environment and acquiring the poses of all the AprilTags present using the Turtlebot3. We first implemented the ROS package *cartographer_ros* for mapping and localization. To effectively explore the environment, we used the ROS package *cartographer_ros*, which utilizes frontier-based exploration. However, due to the limited field-of-view of the Raspi camera, there was a possibility of missing AprilTags during exploration. To address this, we implemented a ROS node that switches between the wall follower algorithm and

explore_lite for efficient exploration along with a custom node to make the Turtlebot spin after a set interval of time. We also utilized the ROS package *tag_detections* to detect and transform tag positions and orientations to a global pose. During our runs, we were able to detect all 11 tags, and the bot took between 6-7 minutes to complete the mission.

VI. Future Scope

A. GTSAM

Currently we are working on implementing the GTSAM [5] library over cartographer in order to obtain more accurate representations of the occupancy grid maps as well as the poses of the April Tags. Cartographer uses EKF for state estimation while GTsam library uses factor graphs for state estimation. because factor graphs provide a more flexible and efficient way to represent and solve nonlinear optimization problems.

EKF is a recursive algorithm that updates the state estimate based on the measurement and system models. EKF assumes that the error terms are Gaussian and linearizes the system model around the current state estimate. This linearization can lead to errors in the state estimate when the system is highly nonlinear or when the errors are non-Gaussian.
On the other hand, factor graphs are a more general representation of the optimization problem that can handle nonlinear and non-Gaussian error terms. A factor graph represents the optimization problem as a set of nodes and edges, where the nodes represent variables and the edges represent the constraints between the variables. The nodes can be connected to multiple edges, allowing for more complex and flexible modeling of the optimization problem.
GTSAM provides a set of factors that represent the constraints between the variables, such as odometry measurements, loop closures, and landmark observations. GTSAM can efficiently solve the optimization problem by marginalizing over the variables and propagating the constraints through the factor graph.

The idea is to create a custom node that employs GTSAM to estimate and correct poses on the map generated by Cartographer. The node subscribes to both the map and trajectory published by Cartographer, executes GTSAM-based pose estimation and correction, and republishes the modified map on the */map* topic for further use by the system.

Algorithm 5 GtsamSLAMNode

- Initialize: rospy node, odom_sub, apriltag_sub, pose_pub, map_sub, updated_map_pub
- Initialize: factor_graph, initial_estimates, odom_noise, apriltag_noise, pose_counter, landmark_set, map_data

function MAP_CALLBACK(msg)
- Update: map_data

end function
function ODOM_CALLBACK(msg)
- Process: odometry data and add odometry factors to factor_graph
- Increment: pose_counter

end function
function APRILTAG_CALLBACK(msg)
- Process: AprilTag detections and add observation factors to factor_graph

end function
function OPTIMIZE
- Optimize: factor_graph
- Return: result

end function
function UPDATE_MAP(optimized_poses)
- Transform: occupancy grid
- Publish: transformed map

end function
function UPDATE_POSE(optimized_pose)
- Publish: updated pose of the robot

end function
function RUN
  while not rospy.is_shutdown() **do**
    - Call: optimize, update_pose, and update_map functions
    - Sleep: for a specified duration

  end while
end function

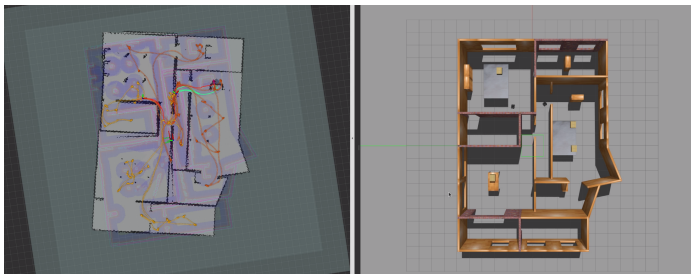

Fig. 6. Results of testing swarm on gazebo

B. Implementing Swarm Robotics

In our future work, we aim to extend our research on April tag detection to swarm robotics, a promising field that involves coordinating multiple robots to work collaboratively towards a common goal. The vision is to implement April tag detection in swarm robots, creating a system where multiple robots can work together in a coordinated manner to quickly and accurately detect and localize April tags in their environment. This could lead to more efficient and robust system in complex and dynamic disaster environments, where swarm robots can adapt and react to changes in the environment while also performing the reconnaissance operation in a distributed manner. We plan to investigate different strategies for swarm robots to communicate, coordinate, and share information to collectively detect April tags, and evaluate the performance of our proposed approach through extensive experiments in real-world swarm robotics scenarios.
As a preliminary step, we have simulated two robots in

a gazebo, where these robots explore the environment and map the areas using the *explore_lite* package and the *multi-robot_map_merge* package in ROS to combine the maps from the individual robots and create a global map for exploration. This approach allows for scalability, as distributing the workload among multiple robots enables them to cover a larger area and potentially reduce the time needed for April tag detection, which can be particularly beneficial in large-scale or time-critical applications. Additionally, swarm robots offer robustness in the system, as multiple robots can detect the same April tag from different angles, enhancing the reliability and accuracy of the detection process.

REFERENCES

[1] X. Chen, "Explore lite." https://github.com/xianyi/ExploreLite, 2018. Accessed on April 23, 2023.

[2] M. Quigley, B. Gerkey, and W. Smart, "Ros navigation stack." https: //github.com/ros-planning/navigation, 2021. Accessed on April 23, 2023.

[3] W. Hess, S. Kohlbrecher, H. Rapp, and G. Andrade, "Cartographer ROS: Real-time 2d and 3d slam." https://github.com/cartographer-project/cartographer ros, 2021. Accessed on April 23, 2023.

[4] P. Steffen, "apriltag ros: Ros wrapper for the apriltag visual fiducial detection algorithm." https://github.com/AprilRobotics/apriltag ros, 2021. Accessed on April 23, 2023.

[5] F. Dellaert, "Factor graphs and gtsam: A hands-on introduction." https: // www.cc.gatech.edu/~dellaert/Factor-Graphs-Tutorial.pdf, 2012. Accessed on April 23, 2023.

[6] U. Robotics, "Raspberry pi_camera node calibration." https://github.com/UbiquityRobotics/raspicam node#calibration, 2021. Accessed on April 23, 2023.

[7] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. MIT press, 2005. Accessed on April 23, 2023.

–