Autonomous Reconnaissance using Turtlebot3

Rucha Pendharkar
College of Engineering
Northeastern University
Boston, MA
pendharkar.r@northeastern.edu

Kaushal Sorte
College of Engineering
Northeastern University
Boston, MA
sorte.k@northeastern.edu

Girish Raut
College of Engineering
Northeastern University
Boston, MA
raut.g@northeastern.edu

Anway Shirgaonkar
College of Engineering,
Northeastern University
Boston, MA
shirgaonkar.a@northeastern.edu

Neeraj Sahasradbudhe
College of Engineering
Northeastern University
Boston, MA
sahasrabudhe.n@northeastern.edu

***Abstract*—Reconnaissance refers to the act of gathering information or intelligence regarding the terrain or a target in a particular environment. Reconnaissance with autonomous bots is a powerful tool that can help us gather information and data in various fields, enabling us to make better-informed decisions and discoveries. Over the past two decades, mobile robots have begun to play an increasingly important roles in reconnaissance, especially in disaster response applications. The use of autonomous bots in disaster management has the potential to improve response times and ultimately save lives greatly. These bots can play an essential role in disaster response efforts by providing responders with valuable information and allowing them to access difficult-to-reach areas.**

***Index Terms*—reconnaissance, disaster-response, mobile robots, SLAM, AprilTag**

I. Introduction

In this project, we designed and implemented a completely autonomous system to perform reconnaissance in a simulated disaster environment. AprilTags were used as a proxy for victims in the simulated environment. The primary goals of this exercise were to map the unknown environment, localize the robot and accurately locate the AprilTags. A Turtlebot3 Burger was deployed to accomplish these tasks. The robot equipped with an array of sensors including LDS-01 LIDAR, a Raspberry Pi camera module V2 with Sony IMX219 8-megapixel sensor, and an IMU. Cartographer package was used to implement SLAM.

II. Proposed Solution

There were mainly two functionalities that the system needed to fulfil. The turtlebot needed to autonomously explore an unknown but closed environment, and detect AprilTags. The necessary code can be found in this repository: EECE5550_Turtlebotics

A. Autonomous Exploration

We used the `Cartographer_ROS` for simultaneous localization and mapping. Cartographer generates a map through the node `occupancy_grid_node` which provides the present occupancy grid produced by the robot. As the turtlebot explores it's environment, the map is updated. In addition to this, Cartographer is essential for motion-planning and exploring. The 2D grid map obtained from Cartographer has a resolution of r = *5cm*. We used the package `explore_lite` to implement greedy frontier-based exploration. The nodes of the package `explore_lite` subscribe to the `/map` topic published by Cartographer to generate frontiers. The `move_base` package from the ROS Navigation stack was used to actuate the turtlebot. We wrote a custom node called `Tag_Tracker`. This node is used to calculate the pose of the AprilTag in the co-ordinates of the map frame
In order to line up the packages, we made a few changes to obtain the required functionality.

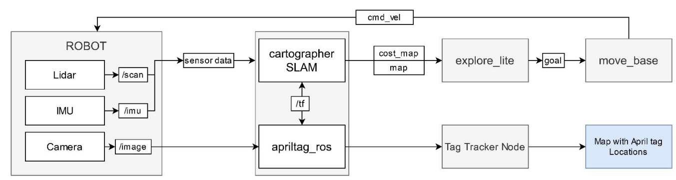

Fig. 1. Main Architecture of the System

1) Remapping: Cartographer's `/map` topic publishes data as [-1,0-100]. These values represent unknown space, and probability of occupancy. However, `explore_lite` anticipates map input as [-1,0,1] representing unknown, occupied and unoccupied space respectively. We wrote a custom node to threshold these values. This script publishes these updated values to another topic `/cmap`. Cartographer subscribes to this topic, and publishes the final occupancy grid to the `/map` topic

2) Parameter Tuning: `explore_lite` has parameters such as `min_frontier_size`, `exploration_strategy`, `cost_map` that need to be tweaked in order to achieve best performance. In order to prevent `explore_lite` from prioritizing larger frontiers, we set the `min_frontier_size` to 0.2. This would ensure

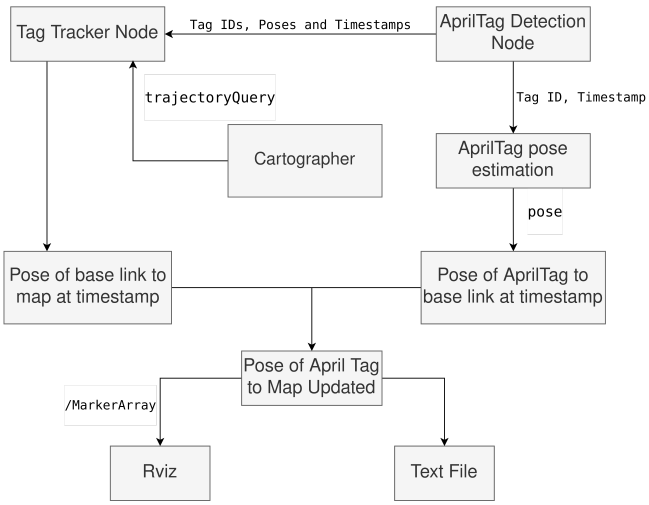

Fig. 2. Flowchart of the Tag Tracking Algorithm

better exploration, even of relatively smaller frontiers. We set `exploration_strategy` to 'frontier-based', as opposed to the default 'closest'. We assigned `/cost_map` to `/map`. Apart from these, we tuned parameters in `explore_lite` such as `inflation_radius`, `potential_gain` for optimizing the obstacle avoidance behaviour

3) Sensor Parameters: We capped the maximum angular and translation velocities. The default values of these implied faster exploration, reducing the probability of detecting all AprilTags. AprilTags can only be localized well from a certain angle and distance. Speeding past them could lead to improper localization. We also capped the maximum use-able range of the LIDAR from 3 meters to 1.5 meters. This helped compensate for the latency in transmission of images

B. AprilTag Detection

AprilTags are a visual fiducial system developed by April Robotics Lab, University of Michigan for accurate localization applications. In this project, AprilTags are detected using a Raspberry Pi camera equipped with an AprilTag detector algorithm. For the AprilTag detection, we followed the Monocular Camera Calibration using a *10 x 7* checkerboard. We used MATLAB's inbuilt camera calibration toolbox to obtain the intrinsic camera parameters. We used the apriltag_ros package, OpenCV, and the intrinsic camera parameters to determine the position and orientation i.e pose of the AprilTag in the camera frame. The script also saves the timestamp at which the AprilTag was detected.

C. Pose Estimation

For estimating the pose of the AprilTag w.r.t `/map`, we used a custom ROS node called `tag_tracker`. One of the primary roles of this node was to subscribe to the `/tf` topic to access the latest transformations between frames present in the tf tree. Secondly, the node was also used for incorporating trajectory updates as the robot's localization improved with

Algorithm 1 AprilTag Detection Algorithm

Set F_x ▷ focal length in x
Set F_y ▷ focal length in y
Set C_x ▷ principle point x
Set C_y ▷ principle point y
$K \leftarrow \begin{bmatrix} F_x & 0 & C_x \\ 0 & F_y & C_y \\ 0 & 0 & 1 \end{bmatrix}$ ▷ Intrinsic matrix of the camera

$distCoeffs$ ▷ Distortion Coefficients of the camera
$tagSize$ ▷ Tag size of AprilTag
$imgSub \leftarrow Subscriber(imageTopic)$
$posePub \leftarrow Publisher(AprilTagPoseTopic)$

while rospy is running **do**
    $image \leftarrow imageData$
    $cvImage \leftarrow ConvertToOpenCVImage(image)$
    $corners \leftarrow DetectApriltagCorners(cvImage)$
    $rotation, translation, id \leftarrow GetTagInfo(corners)$
    $quaternion \leftarrow RotationtoQuaternions(rotation)$
    $apriltaginfo \leftarrow Quaternion, Translation, ID$
    $PublishPose(posePub)$
end while

time. The final transformation from `/tag_ID` to `/map` was calculated and the translational components were extracted to get the position of the AprilTag in the `/map` frame.

1) Pose of the detected tag wrt Camera: Initially, a custom node in OpenCV was written for AprilTag detection. However, later on, the off-the-shelf package `apriltag_ros` was deemed more robust. The detector outputted transformations with Z-axis perpendicular to the camera, whereas the camera frame in the tf tree was x-front. We had to handle this transformation before post-processing tag orientation.

2) Pose of the AprilTag w.r.t. map: The pose of the AprilTag w.r.t. map is found by lining up the transformations: AprilTag w.r.t. camera, camera w.r.t. base_link of the robot, and base_link w.r.t. map. The second transformation was rigid and was added to the tf tree as a constant value. The third transformation was extracted from the pose graph data that the Cartographer stores. Algorithm 2 is used for updating the pose of the April Tag w.r.t. map for handling loop closure events. The timestamp and the tag-to-base_link transformation for a particular tag are stored in a dictionary. We keep overwriting the dictionary until a particular tag is in the camera's field of view, effectively storing the last pose that the camera detected.

Simultaneously, we employ the *trajectory_query* service to extract the updated base_link to map transform at the timestamp, rendering the AprilTag pose robust to loop closures. The key benefit of this approach is that we do not need to revisit the AprilTag, later on, to estimate its pose better if the localization during detection is poor.

III. Results

We successfully deployed our autonomous system in a real arena. The arena in Fig 3 is a conference room in ISEC,

Algorithm 2 Tag Tracker Algorithm

```
Initalize ROS node: tag_tracking_node
Define ROS Service: trajectory_query
Subscribe to topic: tag_detections
Define Publisher: MarkerArray
dict_tag_to_baselink ← None
while rospy is running do
    dict_tag_to_baselink ← Pose of AprilTags wrt CAM
    if tag in dict_tag_to_baselink then
        if tag is recently detected then
            tagToMap ← TF tree
        else if tag is out of view then
            Get timestamp from dict_tag_to_baselink
            tagToBase ← dict_tag_to_baselink
            baseToMap ← get_trajectory_query
            tagToMap ← tagToBase × BaseToMap
        end if
    end if
    points_to publish ← translational{tag_to_map}
    Publish point_to_publish on MarkerArray
end while
Save points_to_publish to text file
```

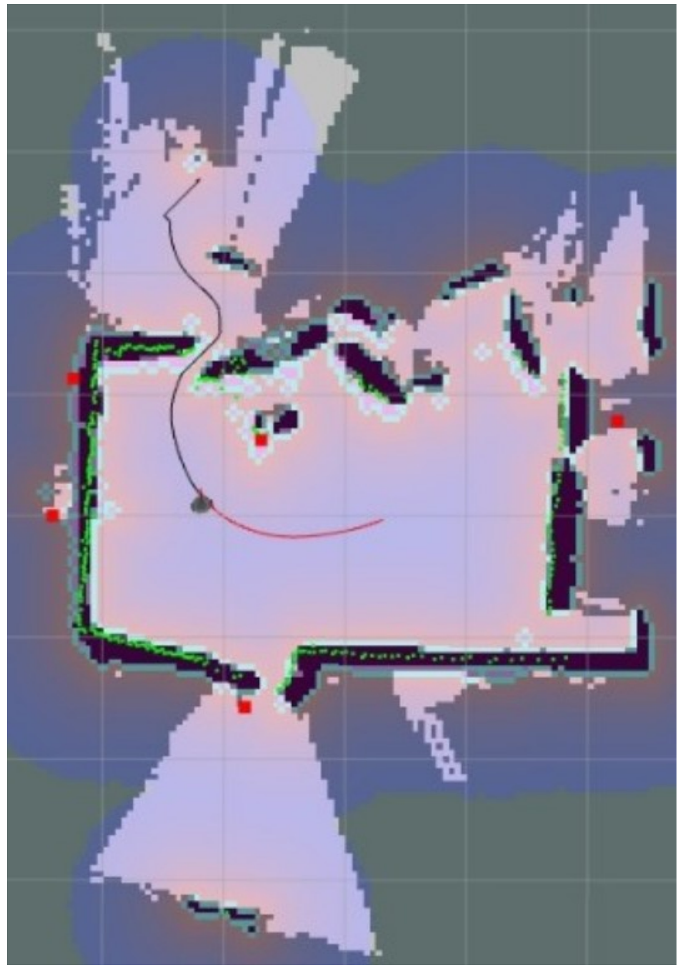

Fig. 3. Results using Cartographer in Arena 1

Northeastern University, Boston. The arena in Figure 4 is the basement beneath Snell Library. Both test setups yield a reasonable performance, with one or two tags with poor localization. The *trajectory_query* service in Cartographer successfully updated the AprilTags as the trajectory updates. Link to the video demonstarting this implementation can be found here: Video Demonstration

IV. Conclusion

This project aimed to implement a SLAM algorithm in an unknown environment, resembling disaster reconnaissance. We have successfully analyzed and implemented autonomous navigation on Turtlebot3. An arena was built to implement the same wherein AprilTags were used to simulate the subjects to be rescued. Methods used for navigation include SLAM and path-planning algorithms. As shown above, the robot successfully generated a map of the unknown environment and identified the AprilTags with minimal error without colliding with obstacles in the environment.

V. Open Challenges

It has been challenging to reconcile the various transforms starting from the tag and ending in the map. From here on, we will first focus on improving our tag measurements from the camera.

Currently, by employing the *trajectory_query* service in Cartographer, the danger of poor localization of the robot is mitigated. Thus, as long as a tag remains in the field of view of the camera, we get a steady stream of valid tag pose measurements that can be corrected as the localization improves - and this fact is not exploited right now since we overwrite the dictionary. In the coming iterations of this project, two possible methods can be implemented to use this data.

1) An averaging filter that stores and averages all the tags to map transforms. This average can be computed in real-time as measurements roll in. In this case, the average would change as the newer measurements come in or if a loop closure occurs. Alternatively, we can compute the average towards the end.
2) A Monte-Carlo Localization can be implemented by treating each measurement as a particle.

The off-the-shelf explorer `explore_lite` is unsuitable for disaster reconnaissance. The rate of exploration outpaces that of exploitation, i.e., tag detection. In the future, we plan to augment the greedy explorer with a random sampler that will launch after `explore_lite` stops getting frontiers. The sampler will improve the chances of detecting all tags.

As reported in [1], the detection performance drops dramatically as the distance and the off-axis angle rise. Thus, we will handle this in future work by registering camera measurements only if the tag lies within a certain threshold distance and off-axis angle.

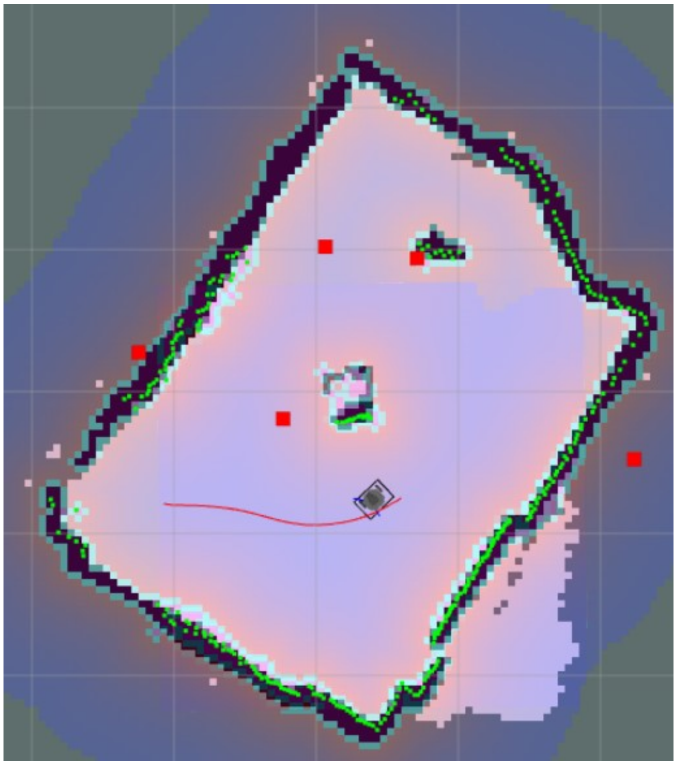

Fig. 4. Results using Cartographer in Arena 2

VI. Contribution

Rucha Pendharkar and Girish Raut worked on experimenting with different SLAM packages such as Cartographer, Gmapping and integrating them with `explore_lite` and `move_base` by tweaking the parameters to obtain optimum results. Anway Shirgaonkar worked on interfacing with the RaspiCam and April Tag detection w.r.t Camera. Neeraj Sahasrabudhe and Kaushal Sorte worked on the formulation and implementation of the tag tracking algorithm responsible for tracking and obtaining the AprilTag pose in the map frame.

VII. Acknowledgment

We thank Professor Michael Everett profusely for his valuable insights into the April Tag localization problem.

References

[1] Edwin Olson, "AprilTag : A robust and flexible visual fiducial system," 2011 IEEE International Conference on Robotics and Automation.

[2] Google Cartographer Team. (2021). *ROS Cartographer*. Retrieved from https://google-cartographer-ros.readthedocs.io/en/latest/

[3] Open Robotics. (2021). *ROS explore_lite*. Retrieved from http://wiki.ros.org/explore_lite

[4] Open Robotics. (2021). *ROS gmapping*. Retrieved from http://wiki.ros.org/gmapping