Autonomous Exploration and Detection of Apriltags

Ravina Lad
lad.ra@northeastern.edu

Christopher Anton Dominic
dominic.c@northeastern.edu

Dipak Sairamesh
sairamesh.d@northeastern.edu

***Abstract*—The problem statement of the final project for Mobile Robotics incorporates a search and tag mission wherein a robot, deployed in an unknown environment is able to autonomously explore and detect AprilTags. Navigation combined with knowledge of search algorithms, mobile robot navigation and mapping, the Robot Operating System, and the TurtleBot3 platform are used to create a program to autonomously explore and map an unknown region. The goal of this project was a mobile robot capable of generating a complete map of an unknown region and detecting maximum AprilTags as stand-ins for simulated victims.**

Keywords—Turtlebot3, Apriltag, Frontier Exploration, SLAM, Gmapping, Raspicam.

I INTRODUCTION

The goal of this project is to program a TurtleBot3, 'burger' model to autonomously navigate and map a closed space. The space is totally unknown to the robot - it must generate the map as it moves around the space. This task was accomplished through the implementation of a frontier-based exploration process. The (estimated) pose associated with an AprilTag detection can be rather noisy, especially when the tag is viewed from a distance.

II BACKGROUND

In order to map the desired area, the robot uses the frontier based exploration method first postulated in A Frontier-Based Approach for Autonomous Exploration [1]. The central idea in this approach is as follows: in order to gain the most new information about the world, move to the boundary between open space and uncharted territory. Specifically, possible frontier cells are identified by looking for occupancy grid cells that are 'unvisited', border unknown space, and have at least one free neighbor. These frontier cells are combined into frontier regions, which are stored as a Frontier message. These messages contain a size, minimum distance from the robot, and travel point. This travel point is the centroid of the frontier region. An example of this is shown in Figure 1 below.

Once frontier regions are identified, they are stored in a queue. Breadth-First Search is then used to determine the closest frontier region. Navigation to this frontier region is accomplished by converting the frontier centroid to an 'ActionGoal'. This 'ActionGoal' is then passed to the built-in ROS navigation stack, which moves the robot while implementing dynamic obstacle avoidance. The navigation stack takes in sensor data, a goal position and utilizes a global and local planner to generate smooth command velocities for the robot.

As seen in Figure 1, the sensor transforms comes from the base minimal launch and defines the transformations between the base of the robot and its various sensors. Since we are creating a map as we are driving, we are not loading a static map but rather having GMapping[2] constantly update the map topic as the robot navigates. The global planner uses NAVFN, an instance of Dijkstra's algorithm to create a smooth path between its current location and the goal position from the Frontier exploration package. The local planner used is the Dynamic Window Approach (DWA) algorithm which listens to the local costmap and reroutes around any obstacles that appears in front of the robot that was not previously accounted for. The navigation stack listens to the scan data published from the Lidar and converts the data into a global and local costmap. The local planner then publishes command velocities on the 'cmd_vel' topic which then drives the robot.

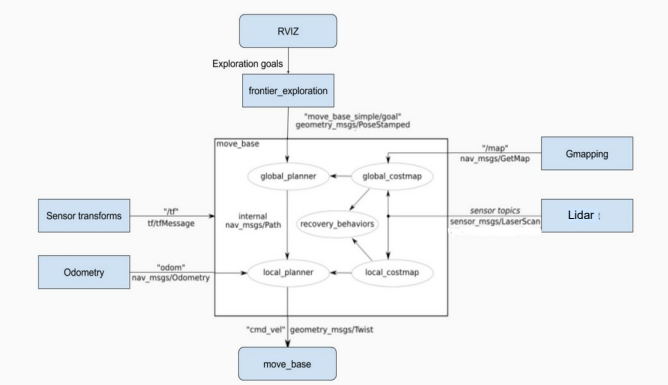

Fig.1 Navigation Stack block diagram

III FLOW OF IMPLEMENTATION

The given problem statement requires us to use a mobile ground robot capable of sensing the environment around it and performing maneuvres. Hence the Turtlebot3 is a suitable robot for the task. For autonomous exploration the Simultaneous Loalization and Mapping (SLAM) algorithm is used. To enable this via a turtlebot, few other packages are deployed in conjunction. The Apriltags fixed in the map are of type '36h11' and are detected via a camera fitted to the raspberry pi. The entire flow and nodes used are highlighted in the rqt graph in fig. 6. The functions of each of the packages used and the method of execution is explained in the following sections.

III.A Turtlebot3

The Turtlebot3 is a robot that is popular in the field of robotics. It's modular design and compact footprint enables users to test and prototype various algorithms used in the field of autonomous exploration. But with it's modular size comes a drawback- minimal processing power. Hence for computationally expensive processes a remote PC (ROS Host) is required. Therefore we use a package titled 'turtlebot3_bringup' which establishes a communication link

between the raw data published by the turtlebot and the host PC. Some of the sample topic published are 'odom', 'tf', 'cmd_vel', 'imu', 'sensor_state' and the subscribed topics are 'imu', 'scan', 'sensor_state'. The camera fitted to the turtlebot is a Raspberry Pi camera called 'raspicam'. It captures video at 1280p resolution which is useful in computer vision based tasks. Due to the large amount of data being sent between the turtlebot and the remote PC, a considerable delay in computation was noticed while reading data.

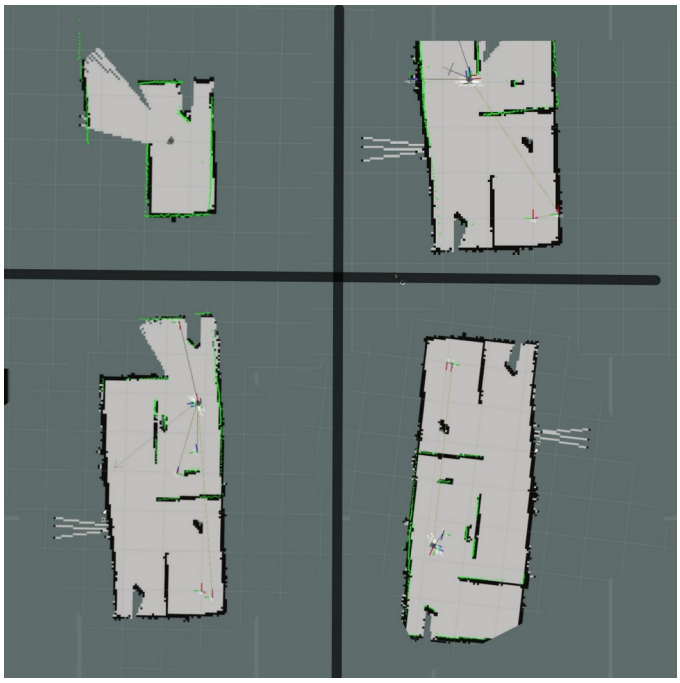

Fig.2. Map formation

III.B Apriltags

For the given problem statement, AprilTags[1] were used in representing victims. AprilTag is a visual fiducial system, useful for a wide variety of tasks including augmented reality, robotics, and camera calibration. Targets can be created from an ordinary printer, and the AprilTag detection software computes the precise 3D position, orientation, and identity of the tags relative to the camera. The AprilTag library is implemented in C with no external dependencies. It is designed to be easily included in other applications, as well as be portable to embedded devices. Real-time performance can be achieved even on cell-phone grade processors. AprilTags are simple AprilTags are conceptually similar to QR Codes, in that they are a type of two-dimensional bar code. However, they are designed to encode far smaller data payloads (between 4 and 12 bits), allowing them to be detected more robustly and from longer ranges. Further, they are designed for high localization accuracy and to compute the precise 3D position of the AprilTag with respect to the camera.

For the given problem we used the 'roslaunch raspicam_node camerav2_1280x960_10fps.launch enable_raw:=true' command to launch the camera node. The RAW feature is enabled to obtain most data from the camera. In order to overcome the lag in transmission, the AprilTags package was installed in the Raspberry Pi in order to detect the tags directly from the camera and transmit the detection rather than the entire raw image itself. This proved an efficient and quick transfer of data between the robot and the remote PC. The command used was 'roslaunch turtlebot3_mr apriltag_gazebo.launch'. The launch file was edited to subscript to the raspicam_node rather than the defalt camera node.

III.C GMapping

For mapping the environment the GMapping package is used. The GMapping package [2] contains a ROS wrapper for GMapping [3]. GMapping is a highly efficient Rao-Blackwellized particle filer to learn grid maps from laser range data. The GMapping package provides laser-based SLAM. Using GMapping, we can create a 2-D occupancy grid map from laser and pose data collected by a mobile robot. The map is created incrementally and in real-time during the robot motion.

III.D Move/Base

For navigation purposes the move_base package [4] provides an implementation to move the robot to a goal point. The move_base node links together a global and local planner to accomplish its global navigation task. The global and local planner can be substituted by other planner implementations; they just have to adhere to interfaces laid out in the move_base package. The global planner provides a path and the local planner will send appropriate velocity and angular commands to move the robot over the current segment of the global path taking into account incoming sensor data.

III.E Frontier Exploration :

The explore package[3] is a frontier-based exploration library; it generates goals for the move_base package. It actively builds a map of its environment by visiting unknown areas. The explore package returns a list of frontiers, sorted by the planners estimated cost to visit each frontier. The frontiers are weighted by a simple cost function, which will prefer frontiers that are large and fast and easy to travel to. The cost of a goal frontier depends on several factors: the distance between the robot and the frontiers, change in orientation to face the frontier and expected information gain of the frontier. cost = weight1.distance + weight2.orientationChange – weight3.frontierSize.

```
while(!map_covered &
      !no_minimum frontier_to_be_traversed)

      M ←Current map
      R ← Robot Position
      C← Camera input from Turtlebot
      A← Find Apritags(C)
      F← Find Frontier points(M)
      Fc← Components with undefined boundary(F)

      For each Fc
            E← Explore until boundary defined(Fc)
            R← Record positions of Apriltags(A)
            M← Update map and Apriltags positions(E,R)
            M← Update map and Apriltags positions(E,R)
```

Fig.3 Pseudo-code of the algorithm

The goal of exploration is to produce a map of an unknown environment. The method consists out of finding frontiers. Frontiers are the boundary between explored and unexplored space. During exploration, robots navigate to frontiers to extend the known area until the complete (reachable) environment is explored. Which frontier to go to next can be controlled based on the cost of a frontier. The path is planned to the next goal frontier. The robot moves towards this goal autonomously while avoiding obstacles on its way. As the robot continues to move, new information about the environments comes in, leading to the formation of new frontiers. If there are no frontiers left, exploration is complete and the area is mapped. Here, occupancy grids are used as an input. In order to get new information, one must go to a frontier that separates known from unknown regions, see attempted figure.4 Such a frontier is a cell in the occupancy grid that is marked as free but has a neighboring cell that is marked unknown. A segment of adjacent frontier cells is considered as a potential target if it is large enough so that the robot could pass it. If more than one potential target is detected in the occupancy grid, then the frontier which has the lowest cost associated with it is selected.

III.F Localization

The first step was determining the robot's pose within its work space. The pose is both the location of the robot and its orientation. After the robot was initialized, it would begin by rotating in place 360°, using a Lidar sensor to scan its environment. This first scan would allow the robot to place itself within the work space, as well as create the first frontiers for it to explore. With knowledge of its pose and a list of frontiers, the robot could generate a path from its current location to a goal destination.

III.G Path Generation

The robot determined its path using the ROS navigation stack, as shown in the diagram above. In implementation, we opted to use the built-in ROS navigation stack because it provided smooth acceleration and arc-based path planning. These features made the robot's navigation both faster and more reliable. The navigation stack uses Dijkstra's algorithm to plan a route from the robot's current position to the goal position. (not sure) The route planning algorithm uses the local costmap generated by the Lidar sensor scans to avoid obstacles.

III.H Obstacle Avoidance

Obstacles are inflated by a constant amount, in our case by 0.22 meters, to ensure that the robot does not navigate too closely to them. This inflation creates an increased 'cost' for the grid cells near obstacles, which in turn incentives the route planning algorithm to pick paths that are further from the wall when they are available.

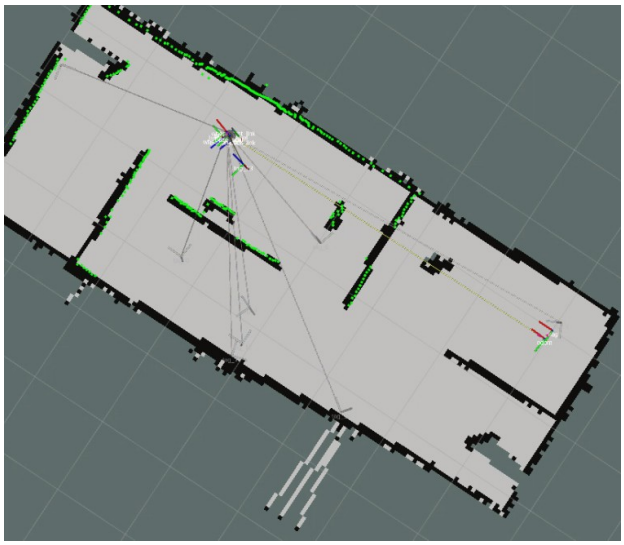

Fig.4 Map after first Iteration

IV. MODIFICATIONS

ROS packages that involve algorithms generally include param/config files that contain limits and constants used in essential functioning of the package. During the initial run of the robot, the default param/config files were used. The output of the map from Rviz is shown in the above picture.

As seen, the upper and lower was not completely mapped and was left unexplored. Hence several parameters were changed to get a better output.

1. We started with the 'explore.launch' file. The "min_frontier_size" was defined as '0.75'. this parameter defined the minimum frontier to be explored. By lowering the value to 0.25 , the robot was forced to explore more frontiers, resulting in a more complete map.
2. Secondly, the turtlebot3_navigation package includes a 'frontier_exploration.yaml' file consisting of several parameters that define the movement of the robot in the map. By altering the resolution to 0.10, enabling 'rolling_window = true' and enabling 'frontier_travel_point = closest' we were able to move the robot more closer to the edges to explore tight spaces. By doing this, the Apriltags located in these tight spaces were also detected.
3. A further problem was observed with the detection of the Apriltags. The 'explore' package terminates its exploration after the map is explored and boundaries are set. This action heavily relies on Lidar data which has high range. The camera's range, although being similar, was not sufficient in detecting the tags. Moreover, the lidar has 360 degree functionality, whereas the camera needs to in front and in its range of vision. By reducing the range of the Lidar this problem was partially solved, as the robot had to traverse closer to the edges to obtain boundary data.
4. The problem with the robot's stability on the map floor was addressed by altering the velocity parameters. During exploration, the robot had to move in reverse directions. The tall structure of the robot caused it to wobble violently upon reverse

maneuvers. Hence by lowering the velocity of the robot along the x direction in 'dwa_local_planner_params_burger.yaml' the robot traversed smoothly in the map.

5. During the testing of our navigation program we encountered some issues with the robot being unable to determine a path to its goal, even when there was enough room for the robot to traverse its path, due to the high cost incurred from traveling in close proximity to an obstacle. Through iterative testing of our program we were able to reduce the inflation constant from 0.5 meters to 0.22 meters, allowing the robot to successfully navigate the environment while avoiding obstacles.

The map after the updated parameters in shown in fig<5> As seen, the gray areas in the previous map was explored efficiently,

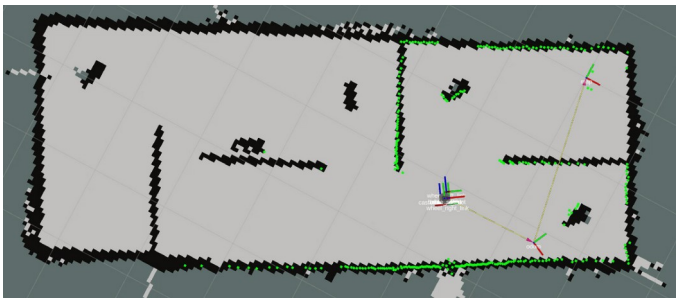

Fig. 5 Map after final Iteration

V. Results

The implementation of the autonomous searching and mapping program was completed by successfully executing frontier exploration and drive control. The TurtleBot was able to search and map the entire workspace within four minutes. Navigation goals were generated autonomously using the frontier exploration package. The goals were selected from the frontier queue using a breadth first search pattern. Using breadth first search, the robot prioritized completing the map in its local area before moving on to new areas. This increased the time efficiency of the search by reducing backtracking. The frontier goals were visualized on Rviz. Occasionally the TurtleBot would gather scan data from outside the workspace through gaps in the walls. This would place an unreachable cell on the frontier queue. The frontier exploration node would eliminate unreachable cells from the frontier once the only path to the cell was too small for the robot as defined in its parameter files.

Path planning and drive base control was accomplished using the ROS navigation stack. The built in navigation stack was used instead of the base controller code because it provided features such as smooth acceleration and arc-based path planning. These features made the navigation stack more reliable and faster than the lab four base controller code. Unlike the search algorithm developed for previous labs, the navigation stack uses Dijkstra's algorithm to plan a route from the robots current position to the goal node. Paths were displayed as green lines on Rviz, as shown in the figure above. The map of the workspace was generated using ROS Gmapping. A costmap of obstacles was generated simultaneously as the TurtleBot scanned its environment, allowing it to reliably generate a path that avoided obstacles. The cost map is visualized as the color gradient from bright blue to pink. Bright blue areas have the highest cost associated with transiting them, so the robot will avoid creating paths in these areas. Gray areas have no cost associated with transit. Occasionally the Gmapping algorithm would encounter scan matching issues, but overall the map was consistent with reality. When the frontier queue no longer contained cells to investigate, the robot would stop exploring. An example of a map generated by a successful run of the exploration program is shown in the figure below.

Of the 15 Apriltags, 13 were successfully detected. This was possible after updating the parameters during the second run. Two Apriltags were not detected due the robot ending its exploration upon complete mapping of the field and the tags being located in certain blind spots. Some methods on how this can be improved is addressed in the section below.

VI. Future Implementations

Although the goal of the problem statement was mostly achieved in the experiment, we found that there were many possible steps that could be incorporated to enable a more efficient outcome.

Since the number of tags were known initially, an algorithm to monitor the tags detected and to instruct the robot to further explore the vertical walls for the remaining tags can be implemented. This algorithm could function over the 'explore' package which terminates the map is complete.

By using machine learning techniques such as RRT and RRT* in path planning, the robot could have been made to move quickly and efficiently through the space.

A 360 degree camera setup would eliminate the limited viewing range of a single camera and could enable an accurate detection of Apriltags upon the first run

VII. Conclusion

The goal of this project was achieved by using the frontier exploration package, navigation stack, Apriltags, Rviz and custom parameter files. The robot was able to detect the tags, navigate the area without hitting any obstacles and construct a two-dimensional map of the environment in under four minutes. This final project built on the experience gained in previous lab assignments, using our knowledge of search algorithms, the ROS middle-ware, and the Turtlebot platform to create a vehicle capable of autonomous exploration.

References

1 Edwin Olson,"AprilTag: A robust and flexible visual fiducial system", Proceedings of the {IEEE} International Conference on Robotics and Automation (ICRA), May 2011Edwin Olson,"AprilTag: A robust and flexible visual fiducial system", Proceedings of the {IEEE} International Conference on Robotics and Automation (ICRA), May 2011

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

3 B. Yamauchi, “A Frontier-Based Approach for Autonomous Exploration”, in Proc. IEEE Int. Symp. Computational Intelligence in Robotics and Automation (CIRA), July 1997

4 Keidar, Matan & Kaminka, Gal. (2012). Robot exploration with fast frontier detection: Theory and experiments. 1. 113-120.

5 Uslu, F. Çakmak, M. Balcılar, A. Akıncı, M. F. Amasyalı and S. Yavuz, "Implementation of frontier-based exploration algorithm for an autonomous robot," 2015 International Symposium on Innovations in Intelligent SysTems and Applications (INISTA), Madrid, 2015, pp. 1-7.

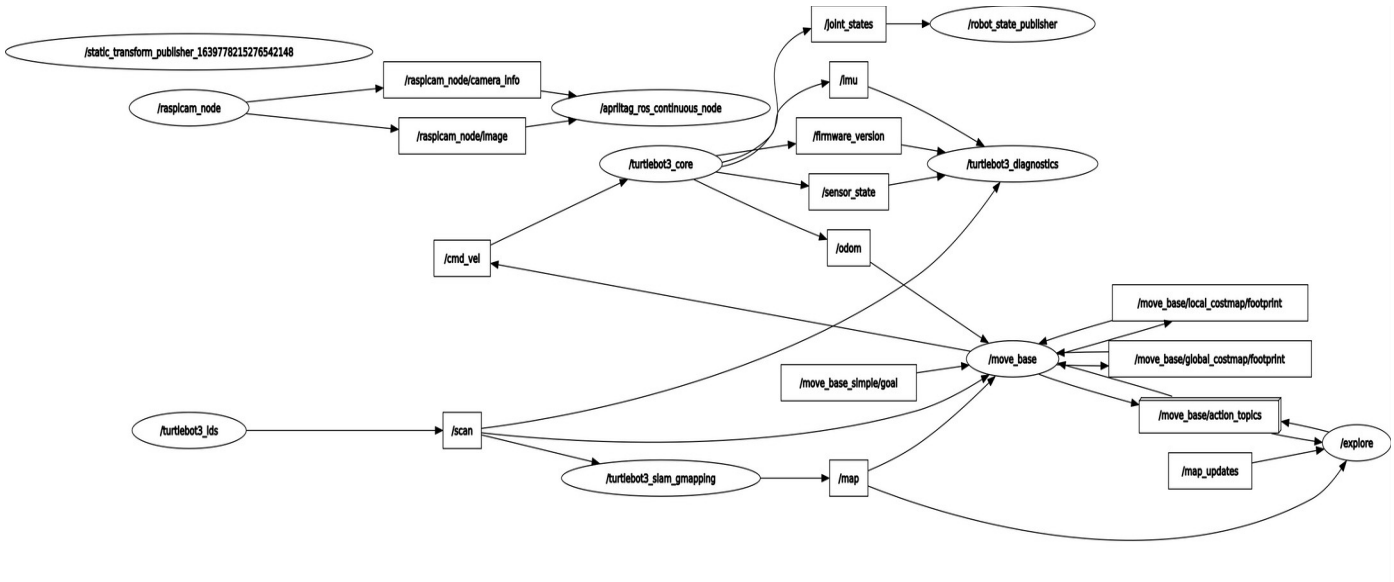

Fig 6. RQT Graph