Autonomous Reconnaissance Robot for a Simulated Disaster Environment

Taoran Liu
College of Engineering
Team Donatello
liu.taora@northeastern.edu

Tae Hoon Yang
College of Engineering
Team Donatello
yang.tae@northeastern.edu

Yin Wang
College of Engineering
Team Donatello
wang.yin4@northeastern.edu

Azhar Hussain Quadri Syed
College of Engineering
Team Donatello
syed.azh@northeastern.edu

***Abstract*—In this project, Turtlebot3 is introduced and an autonomous system is developed on the robot under ROS and Ubuntu environments. The objective of the autonomous system is performing reconnaissance to assist emergency workers by creating a map of an unknown area and detecting all necessary figures in it. The reconnaissance operation is divided into two phases. In the first phase, the robot creates an occupancy grid map as soon as possible based on Frontier exploration. In the second phase, the robot starts to detect the figures ensuring the camera is always oriented to the figures. The resulting map and detected figures are provided and this verifies the accuracy and efficiency of the strategy.**

Keywords—AprilTags, TurtleBot3, localization, mapping

I. Introduction and Motivation

In disaster response, there are many risks for emergency workers to operate their rescue mission. If there is a reconnaissance robot that can survey the dangerous environment and locate victims, it would help minimizing the risks to the workers significantly. The robot should be able to perform the reconnaissance in a complete autonomous system when introduced into a close but initially unknown environment. There are two functions that the robot must have to achieve the reconnaissance objective. One is generating a complete map of the environment using an occupancy grid map and the other is locating any victims identifying their IDs and poses (represented with respect to the map frame). AprilTags are used to indicate the victims as stand-ins and each of them has an unique ID. Turtlebot3 is chosen as a robot development platform because it possesses proper specifications to implement those functions.

II. Proposed Solution

A. General strategy introduction

In order to generate the complete occupancy grid, Frontier-based exploration is suggested.

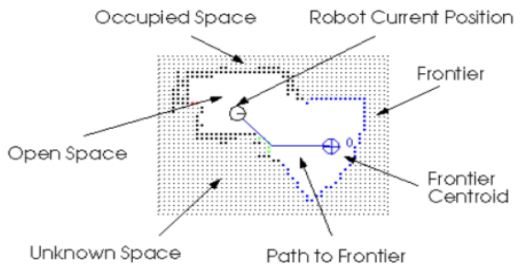

Fig. 1 Frontier-based exploration [1]

As shown in the figure above, the robot explores the environment following frontier centroids until there is no more frontier left which means every area is searched in the closed environment. SLAM is required for the exploration so as to keep updating the occupancy grid map and localize the robot. In ROS, many packages are already available to execute this exploration. One of the implementations is using the gmapping package for SLAM, explore_lite package for Frontier-based exploration and move_base package for driving the actual robot. The entire architecture is explained in the figure below.

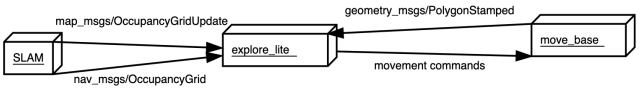

Fig. 2 ROS node architecture for Frontier-based exploration [2]

In order to identify the victims, a Pi camera on the robot is used. Also, there is an existing package called apriltag_ros, that reads images from the camera and outputs tag IDs and poses when a tag is detected. Its overview is shown in the figure below.

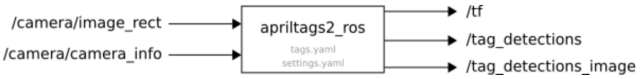

Fig. 3 Topics overview in the apriltag_ros package [3]

B. Problems

However, these two functions do not ensure that all the tags are going to be detected by the camera because the spinning lidar sensor can scan all around the robot while the camera is at the fixed position looking in the forward direction. The robot will not search every single corner of the environment with the camera as the lidar will have finished creating the map. Moreover, the tag measurement of the camera might be rather noisy because the robot is moving and the tags could be viewed in a distance which will increase the noise in the measurement. These two problems have to be resolved so as to thoroughly perform the robot's tasks. Many solutions are proposed to ensure detecting all tags. One approach is rotating the robot to scan around with the camera while creating the map. Another one is limiting the lidar's performance so that the robot could move in the range of the accurate tag measurement. The other one is focusing more on the information about the tags. It is specified that the tags are "stand-ins" for simulated victims which means they are always on walls or obstacles not just on some random ground in the environment. This is found to be an inevitable condition as the camera is not good at detecting the tags on the ground when tested. Therefore, the rough locations of the tags are informed which would be really useful when planning the robot's motion. This information ensures 100% detection rate of all tags in the environment as the robot just

needs to survey the walls and surfaces of the obstacles carefully.

C. Our strategy

Our strategy is to create the map as soon as possible to figure out where the walls and obstacles are using the full capability of the lidar sensor and the robot motion. Subsequently, the path along the walls and obstacles will be created based on the complete map. If the robot just moves straight along the path, the camera is still not oriented toward the possible tag areas. Thus, the camera is relocated to the side of the robot as shown in the figure below so that it can now follow the path and detect the tags at the same time.

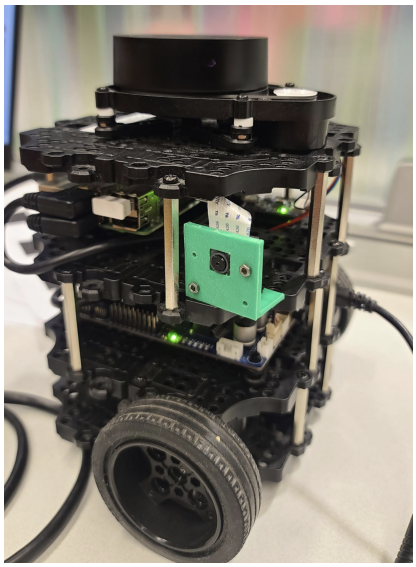

Fig. 4 Frontier-based exploration

To resolve the second problem as to the noise of the tag measurement, the camera is tested several times with different tags to determine in what distance the tags are detected accurately and this distance is taken into account when the robot plans the tag detection path after creating the complete map.

We try to use this method because we find the AprilTags can not be detected very well on the ground with a flat angle. So we dececided to put them on the wall and the side of the obstacles.

III. CONSTRUCTION OF SYSTEM

In this part we want to show how we run the code. Run *roscore* in the PC. Make sure to setup the *~/.bashrc* file with the right IP address of the PC and *TurtleBot3* and *ROS_MASTER_URI* and *ROS_HOSTNAME* as per the instructions from the guide [4]

a) In the Turtlebot3, do the following:

- To bringup the turtlebot
  run "*roslaunch turtlebot3_bringup turtlebot3_robot.launch*"
- To run the usb_cam_node
  run "*roslaunch usb_cam usb_cam-test.launch*"
- To run the apriltag detection
  run "roslaunch apriltag_ros continuous_detection.launch"

b) In the remote PC, do the following:

- To execute the move_base node
  run "*roslaunch turtlebot3_bringup turtlebot3_robot.launch*"
- To execute the gmapping node
  run " r*oslaunch turtlebot3_slam turtlebot3_slam.launch*"
- To execute the explore_lite frontier exploration,
  run "*roslaunch explore_lite explore.launch*"
- To save the map
  use command "*rosrun map_server map_saver -f ~/map*"
- To transform the camera
  use the command "*rosrun tf static_transform_publisher 0.03 0 0 1.57 0 0 base_link usb_cam 100*"

Since our camera is mounted on the right side of the robot, the camera frame has rolled 90 degrees with respect to the base frame.

- Starts to record tag IDs and poses into a bag file
  use the command "*rosbag record [tagID] [tagPose]*"
- To generate the route, and do the route following driving Algorithm 1

Input (map) Output(route) black=1 white=0 gray=2

1: Generatemap{map.png}
2: Turn into V=$\{color\}_{x*y}$
3: For every V_{ij} color == black
4: $V_{i:i\leftarrow i+k\ j:j\leftarrow j+k}.color = gray$
5: For every V_{ij} color == gray
6: $V_{ij}.color = black$
7: For every V_{ij} color == black
8: If ($V_{ij\leftarrow(-1,+1)}.color == white$)
9: $V_{ij}.color = gray$
10: For every V_{ij} color == gray
11: R: r(x,y,0)=(i,j,0)
12: Return R

Algorithm 2

1: Route following (R)
2: For every R_i
3: find the newest R_j
4: do navigation (from $R_i \rightarrow R_j$)
5: Delete R_j
6: If $R_i = \emptyset$
7: Return 0

- Once the route following is done, stops recording the tag data and print the saved data from .bag file

"*time ros_readbagfile <mybagfile.bag> [tagID] [tagPose]*"

- To view the output of the usb_cam_node and /tag_detections node
  use the command "*rqt_image_view*"

A. Move_base execution

The move base node is a ROS interface for configuring, operating, and interfacing with a robot's navigation stack. Above is a high-level view of the move base node and its interactions with other components. Blue nodes differ depending on the robot platform, gray nodes are optional but offered for all systems, and white nodes are mandatory but also provided for all systems. The move base node is a ROS interface for configuring, operating, and interfacing with a robot's navigation stack. Above is a high-level view of the move base node and its interactions with other components. Blue nodes differ depending on the robot platform, gray nodes are optional but offered for all systems, and white nodes are mandatory but also provided for all systems. The move_base node links together a global and local planner to accomplish its global navigation task. In the absence of dynamic barriers, the move base node will finally reach its goal tolerance or

report failure to the user. When the robot believes itself to be stuck, the move base node may optionally undertake recovery behaviors.

The move_base package implements an action that, given a destination in the world, attempts to attain it using a mobile base. To complete its global navigation duty, the move_base node connects a global and local planner. The move_base node additionally keeps two costmaps, one for the global planner and one for the local planner, which are used to navigate.

We chose move_base since it is the foundation of the navigation algorithm. move_base is a package which integrates several components of navigation, each of which complies to a common API provided in the ROS standard. This standard interface enables the user to simply construct their own algorithms for each component of robotic navigation (Local and Global planning, Recovery, Costmaps).

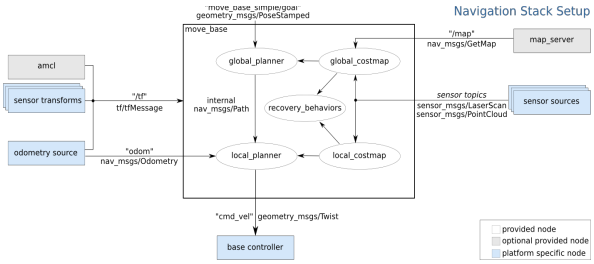

Fig. 5 Navigation Stack Setup[5]

To clear out space, the move base node will do the following steps by default: The robot's map will first be cleansed of obstructions outside of a user-specified region. The robot will then, if possible, undertake an in-place rotation to free out space. If this too fails, the robot will more aggressively clear its map, eradicating all obstructions outside of the rectangular region in which it can rotate in position. Then there will be another in-place rotation. If all of this fails, the robot will deem its goal to be impossible and will alert the user that it has aborted. The recovery behaviors option can be used to configure these recovery_behaviors, and the recovery_behavior_enabled parameter can be used to disable them.

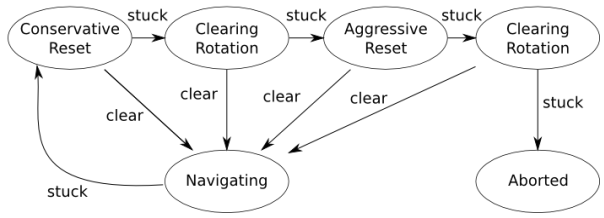

Fig. 6 Move base default recovery behaviors[6]

B. SLAM Implementation

The gmapping package includes a ROS node named slam_gmapping that performs laser-based SLAM (Simultaneous Localization and Mapping). You may produce a 2-D occupancy grid map from laser and posture data gathered by a mobile robot using slam_gmapping.

ROS can assist you in tracking coordinate frames throughout time. It comes with a distinct message type: tf/Transform and is always bound to one topic: /tf. The package for it is tf2 - the transform library. Message tf/Transform contains transformation (translation and rotation) between two coordinate frames, as well as the names of both frames and a timestamp. Publishing to /tf is done in a different method than to any other topic, we will build tf publisher in the example.

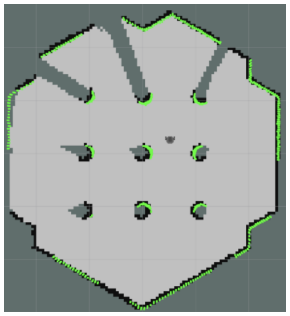

Fig. 7.1 Gmapping Slam in gazebo simulation

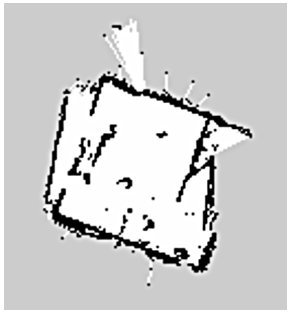

Fig. 7.2 Real World mapped using Gmapping

C. Frontier Exploration

In ROS it is possible to explore the environment with use of occupancy grid frontiers. One of the nodes that perform this task is explore_server node from frontier_exploration package. This node uses occupancy grid e.g. created by slam_gmapping and publishes goal position to /move_base/goal topic subscribed by path planner e.g. move_base node.

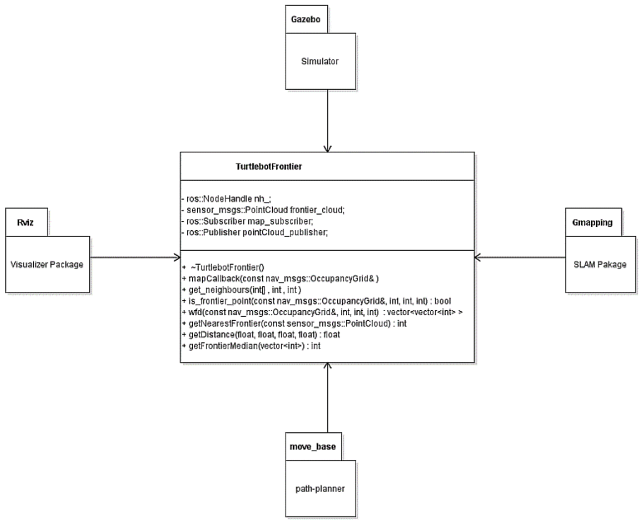

Fig. 8 Move base default recovery behaviors[7]

D. AprilTag Detection

In this part, we use an usb camera in combination with a package from ROS to detect all the AprilTags that as substitutes for our victims. When the robot detects an apriltag we can get the information that contains the tag's ID and its pose relative to the camera. Then we transform this pose to

map frame and add it to a list of all tags that have been detected already.

a) AprilTag recognization: The whole design of this detect algorithm is that we do edge detection first and find the quadrilateral, after the homography we go for the recognize the tag stage.

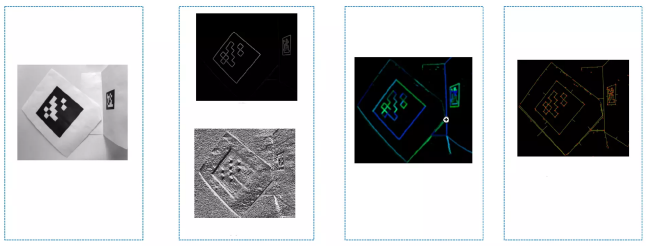

Fig. 9 Design of tag detect algorithm [8]

We will use aprilTag_ros package to realize this part. Apriltag is a visual reference library and is widely used in VR, robot, camera calibration and other fields. Through specific two-dimensional code marks, the target position can be quickly detected and the relative position can be calculated in real time. The apriltags2_ros package had already renamed by apriltag_ros but the function is same as before. In this figure, we can see this package received /camera/image_rect topic and /camera/camera_info topic. Then through its two configuration files tags.yaml and settings.yaml, it publishes /tf, /tag_detections and /tag_detections_image these three topics. The /tf topic contains the position and direction data of each detected QR code relative to the camera. /tag_detections includes a custom message tag ID, which is mainly used to detect a cluster of tag bundles. In our project, we choose tag36h11 family and custom 6 tags information in it. The information of /tag_detections_image is same as /camera/image_rect what different is that we use this to highlight the label position on the output image in real time.

b) Pose estimation of the AprilTags: Now we can get the AprilTags's information in the frame of the robot's camera, which is published to the tag_detections topic. However, for this project, our robot have to estimating the locations of the AprilTags in the environment. Therefore, we need to transform the apriltag pose with respect to the map frame. If we want the global pose of an AprilTag we should understand the relation between these two frame:

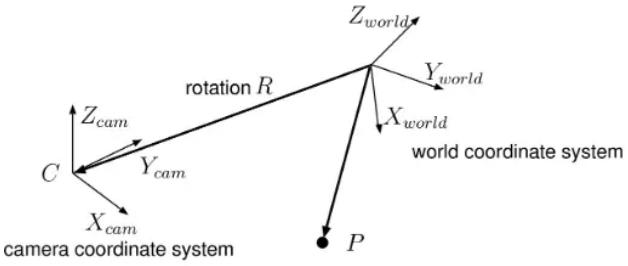

Fig. 10 Frame of two coordinate system

As this figure shows, the world coordinate system is what we want. We have already obtained a current estimate of the robot's pose from the world frame provided by gmapping. The camera coordinate system takes the camera lens (optical center) as the coordinate origin. The optical axis as the z-axis, in the right hand coordinate system, the thumb downward, pointing to the y-axis, the index finger as the z-axis, and the middle finger as the x-axis. Theoretically, the x-axis and y-axis should be parallel to the x and y in the image, and the image we selected here uses the upper left corner as the origin, the x-axis positive direction as the right, and the y-axis positive direction as the bottom and these belongs to three-dimensional data.

Rotation in 3D will help us to realize the frame change. It also be defined as linear transformations, although parameterizing them is not as simple as in 2D. The space of 3D rotations is known as the special orthogonal group $SO(3)$. A rotation in 3D can be represented by following matrix equation

$$\mathbf{p}' = R\mathbf{p} \qquad (1)$$

Here p is the original point, p' is the transformed point and R is a 3×3 rotation matrix.

$$R = \begin{bmatrix} x_x & y_x & Z_x \\ x_y & y_y & Z_y \\ x_z & y_z & Z_z \end{bmatrix} \qquad (2)$$

Here (x_x, x_y, x_z) give the coordinates of the new X axis in the old frame, (y_x, y_y, y_z) gives the coordinates of the new Y axis, and (Z_x, Z_y, Z_z) gives the coordinates of the new Z axis.

After a rotation, the coordinates of the transformed point that relative to the original axes are determined via a matrix multiplication with the rotation matrix R. The rotation matrices is to interpret each of the 3 columns as the coordinates of each of the coordinate axes after rotation. We can interpret the entries of the rotation matrix for as the formula when a coordinate frame rotates by matrix R about the origin.

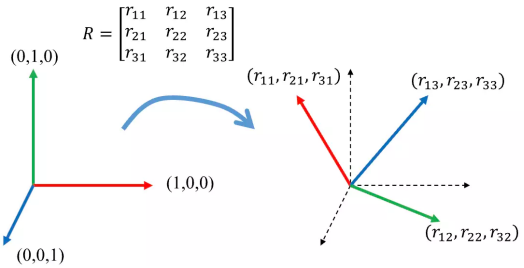

Fig. 11 Frame transform process [9]

IV. RESULT

Fig. 12 Test environment

The test environment can be checked here. Our test environment is a rectangular area around 13.3m^2. We put eleven AprilTags in our test environment. Several obstacles were placed and the tags were attached to the walls and obstacles in different heights and orientations. The robot started to move as soon as the explore_lite command was run. The video of the robot implementing the frontier based exploration can be checked here. Also, you can check what Rviz was displaying during the frontier based exploration here. The resulting occupancy grid map is shown in the figure below.

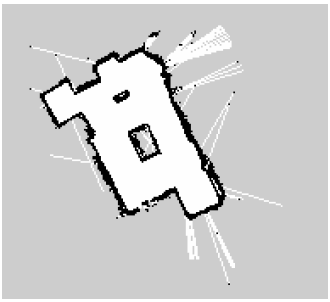

Fig. 13 Resulting occupancy grid map of the test environment

The figure below shows the rqt_graph of the robot during phase 1.

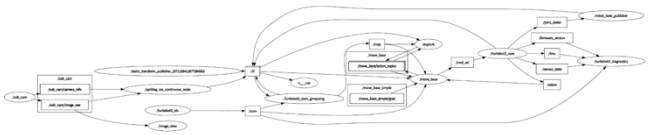

Fig. 14 RQT graph in phase 1

Once the map is created, the robot is supposed to create a path along the wall and obstacles. However, migrating the path planning algorithm to the ros framework is requiring more work and studies, so for this demonstration, the robot was controlled with the keyboard to simulate the algorithm and verify that all the eleven tags could be detected with this strategy. The demonstration of the wall/obstacle follower can be checked here. Also, what Rviz was showing during the operation can be checked here.

The poses of all eleven tags are displayed on Rviz as shown in the figure below:

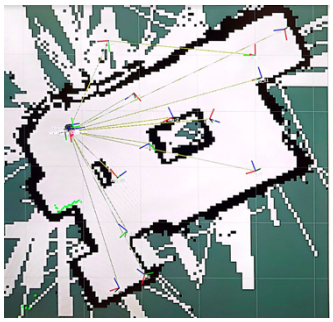

Fig. 15 Rviz about tags information

In this figure, Tag IDs and poses displayed on the occupancy grid map generated in phase 1 previously.

The exact coordinates of the tags are shown in the figure below.

Fig. 16 Pose data of the tags part 1

Fig. 17 Pose data of the tags part 2

The figure below shows the rqt_graph of the robot during phase 2.

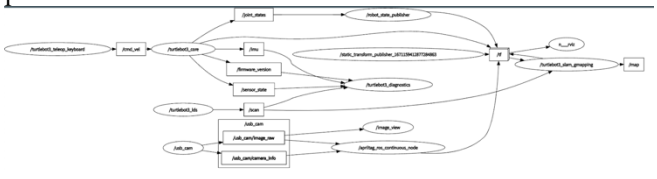

Fig. 18 RQT graph in phase 2.

V. CONCLUSION

A robust autonomous system for reconnaissance in a simulated disaster environment has been developed. It is confirmed that the robot is able to create an occupancy grid map correctly and fast using Frontier-based exploration given the resulting occupancy grid map generated by the robot in phase 1. As for phase 2, the custom node to create the path for tag detection and navigate the robot through the path isn't completed yet, but the general idea of following the walls and

obstacles was able to be demonstrated by controlling the robot with keyboard operation assuming the resulting motion is the same as that of the custom node. Obviously, the robot could detect all of the tags as our strategy ensures 100% detection rate with very accurate pose calculations based on the resulting tag data.

REFERENCES

[1] https://arxiv.org/pdf/1806.03581.pdf

[2] http://wiki.ros.org/explore_lite

[3] http://wiki.ros.org/apriltag_ros

[4] https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/#pc-setup

[5] http://wiki.ros.org/move_base?action=AttachFile&do=get&target=overview_tf.png

[6] http://wiki.ros.org/move_base

[7] https://arxiv.org/ftp/arxiv/papers/1806/1806.03581.pdf

[8] https://www.pudn.com/news/6228d09f9ddf223e1ad160a5.html

[9] http://motion.cs.illinois.edu/RoboticSystems/CoordinateTransformations.html