Final Project

Introductory Robot Programming

December 16, 2021

Students:
Kumara Ritvik Oruganti

Adarsh Malapaka

Venkata Sai Ram Polina

Instructors:
Z. Kootbally

Group:
9

Semester:
Fall 2021

Course code:
ENPM809Y

Contents

1 Introduction

The main aim of this project is to address how robots detect their goal in an unknown environment and how they coordinate with each other in a search and rescue operation. This project is inspired from the challenge of autonomous robotics for Urban Search and Rescue (US&R). [Fig. 1] shows the Gazebo world representing a building inside which the search and rescue efforts have to be carried out by two robots respectively, namely: Explorer and Follower. These robots are spawned in the lower left corner of the shown Gazebo environment (not visible in the figure) near ArUco marker 0. The objective of the explorer robot is to build the map of the building and find the ArUco markers in the environment (the victims in real world) and send these co-ordinates to the follower robot such that it reaches these markers, according to the order of the ArUco marker IDs and later heads back to it's starting position. In this project, the explorer is provided with locations to reach, from where it can rotate and detect the ArUco markers. However, the issue is that the targets provided to the explorer need not be in the order in which the follower must follow while 'rescuing'. The following report discusses about the problem, our approach towards the solution, the challenges we faced and resources we used along with the code to solve the problem both effectively and efficiently.

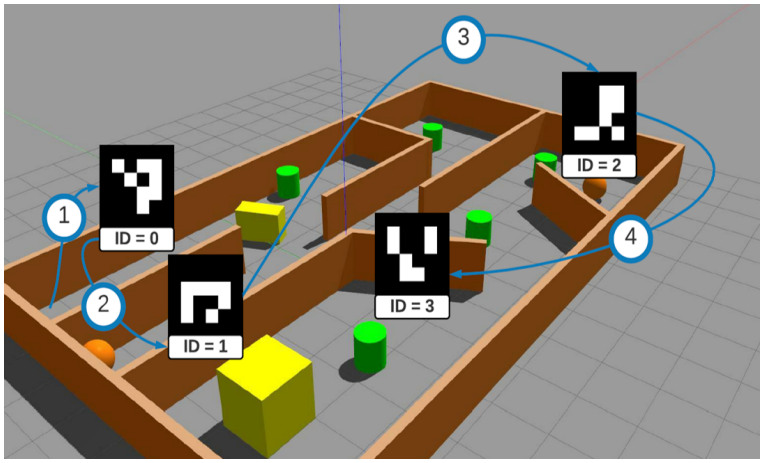

Figure 1: Gazebo World (building) with ArUco markers (victims)

2 Approach

When presented with the task as described above, our approach was to adhere to the following three phase structure to ensure both completion of the task at hand and to enable us to follow good coding practices and test for cases not described in the problem statement, as a means to replicate a typical workflow in a workplace. The three phases, in chronological order, are: *Algorithm*, *Implementation* and *Testing*. Towards the end of the Implementation phase and subsequently running in parallel with the Testing phase, optimization of the code and error debugging were performed which are not considered as separate phases but rather lumped into the above defined phases.

An important assumption was made wherein this search and rescue task was considered solely to be a planar (X-Y) problem and that the robots do/can not actuate along the vertical axis to be able to explore new places and identify 'victims'.

2.1 Algorithm Phase

In the *Algorithm* phase, considerable amount of time was spent initially to clearly understand the problem and it's objectives. For this, the provided project instructions document was thoroughly read through multiple times to understand each individual sub-task under the project. After the initial reading of this document, the project seemed daunting. However, to simplify this, from the second re-read of the document on wards, the given *final_project* ROS package and the already implemented *Bot_Controller* codes were simultaneously executed and understood. The functionalities of the two robots that have already been encapsulated and provided to us were explored, for instance, the *move_base* ROS package conveniently computes a path and moves the robot to the given desired point, provided there exists a legal path between the current pose and the desired pose. This investment of time in understanding the problem coupled with our prior experience with ROS made it easy for us to decompose a seemingly difficult task into easier sub-parts.

[Fig. 2] depicts the devised overall flowchart for how the project code must work, prior to transitioning into the *Implementation* phase. The third step of creating a class and object were not initially performed as these were subsequently added during the code optimization period in the subsequent phases. The flow can be simplistically viewed as first obtaining the target locations for the explorer robot to reach to look for 'victims'. Here, it shall undertake searching operations using the mounted camera on the robot to detect 'victims' (in this case, ArUco markers). Upon successful searching, the follower robot must undertake rescue operations by reaching the locations the explorer went to, in the specified order. At the end of this activity, the total time taken is computed and printed to be used as a metric for evaluating the performance of our program.

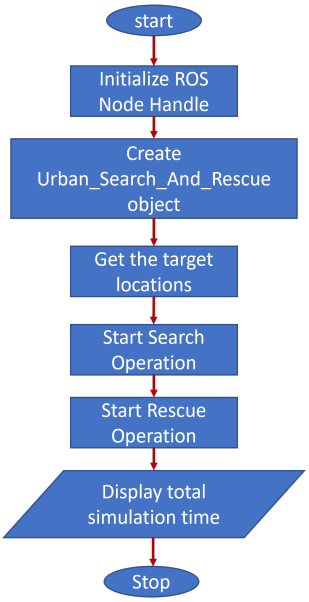

Figure 2: Overall Flowchart

2.2 Implementation Phase

Having obtained a general overview of the algorithm and working of the entire program, the next phase was the *Implementation* phase wherein each block in the [Fig. 2] flowchart was implemented using regular procedural programming. Once satisfactory results were obtained, the code was modified to include object oriented programming to leverage the various advantages of such a programming paradigm.

2.2.1 Search Operation

The search operation, as mentioned in [Fig. 2], shall be zoomed into in this section. The explorer robot takes the target locations from the ROS parameter server and gives them as input to the *move_base* navigation stack package. Once the robot reaches the target, it rotates [Fig. 5] to detect the ArUco marker and later broadcasts the marker's frame onto TF. [Fig. 6] shows the flowchart depicting the overall 'search' process.

Retrieval & Storing of Fiducial ID

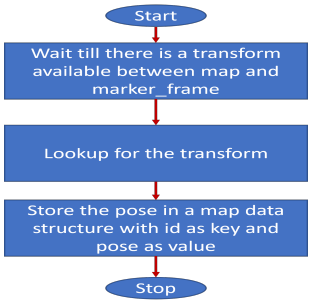

Figure 3: tf Listener flowchart

The implemented TF listener function [Fig. 3] is used to listen to the transformation of marker_frame with respect to the map frame. The fiducial ID (FID) instance variable of the class is updated by the broadcaster [Fig. 8] and is used as a key in a C++ std::map data structure with the values consisting of the pose of the ArUco marker frame with respect to map frame. This pose is a std::array of 7 elements consisting of 3 parameters for translation and 4 parameters for quaternion rotation [Fig. 4].

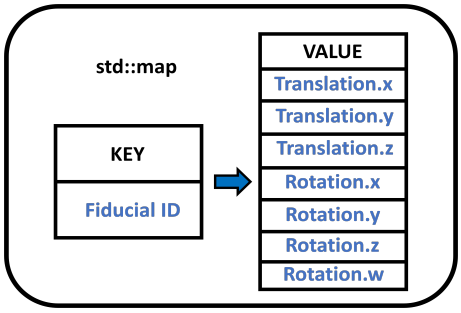

Figure 4: C++ std::map to store fiducial marker information

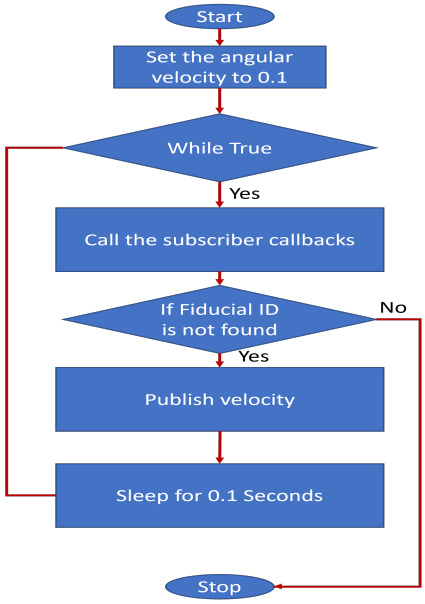

Figure 5: Rotate bot flowchart

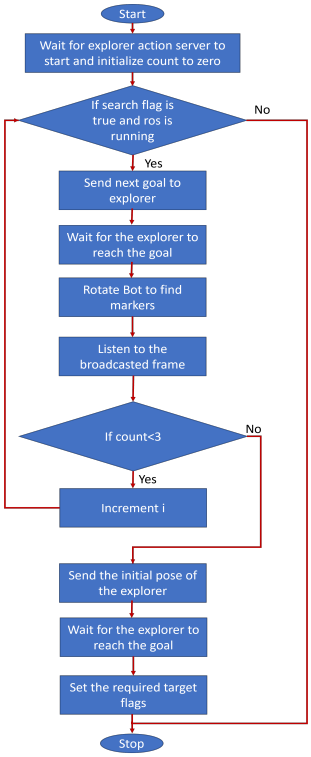

Figure 6: Search operation flowchart

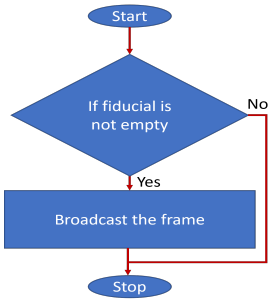

Figure 7: Fiducial transform topic callback flowchart

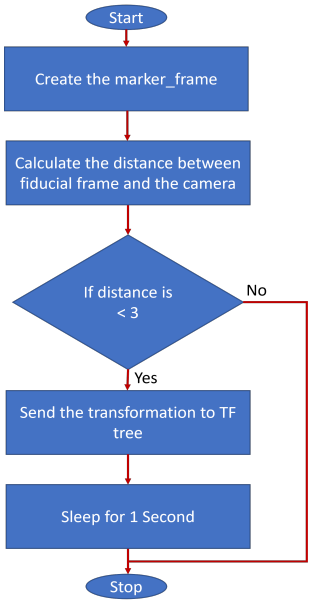

Figure 8: Broadcaster flowchart

2.2.2 Rescue Operation

The rescue operation, as mentioned in [Fig. 2], shall be zoomed into in this section. The follower robot uses the already stored listened transformation between the marker frame and the map frame to reach all the specified target poses (within a distance tolerance) and ultimately returns back to its home position after which the entire search and rescue activity terminates. [Fig. 9] shows the flowchart depicting the overall 'rescue/follow' process.

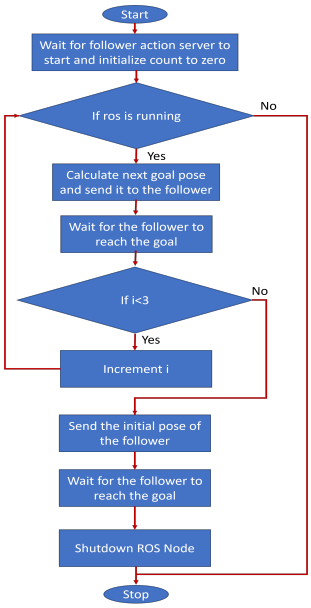

Figure 9: Rescue operation flowchart

Target Pose for Follower Robot

The follower robot is commanded to reach the ArUco marker frame poses broadcasted by the explorer, in the ascending order of the fiducial ID values. Since, the origin of these marker frames lie inside the respective wall/object the marker is mounted to, the follower robot will not be able to reach the target. To circumvent this potential issue, the robot is made to reach the target position at a distance tolerance of 0.4 meters from the marker frame origin.

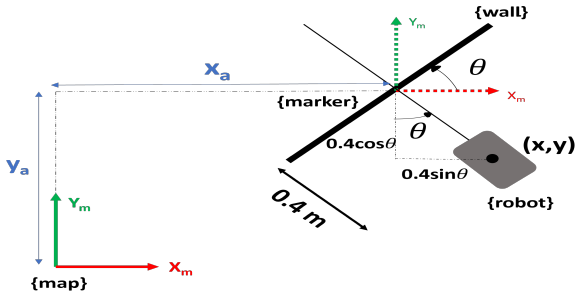

Figure 10: Follower target position w.r.t. the map frame

As groundwork to implement this distance tolerance in cases where the ArUco marker is on a wall that is at an angle with respect to the absolute horizontal/vertical, [Fig. 10] was constructed to obtain the required X and Y co-ordinates of the robot with respect to the map frame, to place it 0.4 meters away from the marker.

- Here, θ is the angle between the map frame (denoted by X_m and Y_m) and the wall/marker, defined positive using the Right-Hand rule. This angle is obtained from the listened transformation's quaternion component of the ArUco marker frame which is then converted to its corresponding Euler yaw angle and finally limiting its range to $\{0, 2\pi\}$ radians.
- x_a and y_a are the X and Y co-ordinates of the corresponding ArUco marker frame origin with respect to the map frame, obtained using the tf listener.

Resolving the distance of 0.4 into its corresponding horizontal and vertical components, gives the following equations to compute the X and Y co-ordinates of the robot in the map frame which are then fed as inputs to the *move_base* navigation stack to make the follower robot move.

$$x_{robot} = x_a + 0.4sin\theta \tag{1}$$

$$y_{robot} = y_a - 0.4cos\theta \tag{2}$$

2.2.3 Object Oriented Programming

As mentioned previously, the search and rescue programs were initially implemented without the usage of concepts from C++ object-oriented programming. After successful working of the codes, we ported all the necessary variables and functions into corresponding attributes and methods respectively of the class *Urban_Search_And_Rescue*. The attributes named with a prefix "m_" are defined with the access modifier of 'private' to implement the concept of Data Hiding. Public methods are available for the end user from the main code which implements Data Encapsulation. Only one C++ class definition [Fig. 11] was sufficient to properly organize the data for smooth working of the program.

The simulation video of running the search and rescue activities for the given Gazebo world and YAML files is given in [2].

Figure 11: Class Diagram for Urban_Search_And_Rescue

2.3 Testing Phase

As seen in the previous section, having successfully implemented and simulated the desired search and rescue operations for the given set of ArUco marker positions in the world frame, we felt the need to generate our own custom ArUco marker positions in the Gazebo environment, to verify the extent of generality of our implemented solution. A new Gazebo World file with the ArUco marker position along with a YAML file with the new X and Y co-ordinates of the marker were defined. In this case, we only shifted the first ArUco marker to a new position. The problem associated with changing more than one marker is listed in the subsequent section on Challenges.

Location	ArUco ID	X_{new} (m)	Y_{new} (m)	X_{old} (m)	Y_{old} (m)
1	0	-1.192846	2.821559	-1.752882	3.246192
2	1	-2.510293	1.125585	-2.510293	1.125585
3	3	-0.289296	-1.282680	-0.289296	-1.282680
4	2	7.710214	-1.716889	7.710214	-1.716889

Table 1: Modified and Original ArUco values in YAML file

[Table. 1] shows the X and Y co-ordinates of the new and old ArUco markers with the value of location

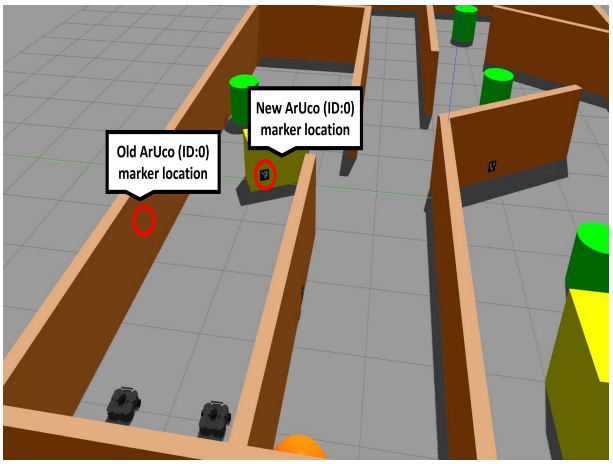

Figure 12: Visualizing the new ArUco marker (ID:0) in Gazebo

1 alone being changed from what was used earlier. [Fig. 12] shows the Gazebo World containing the marker placed on a new object with the old position marked for reference. The figure below [Fig. 13] shows a snapshot from executing the codes with the new marker location wherein the explorer robot is seen to have successfully reached the new marker target and detected it with it's camera.

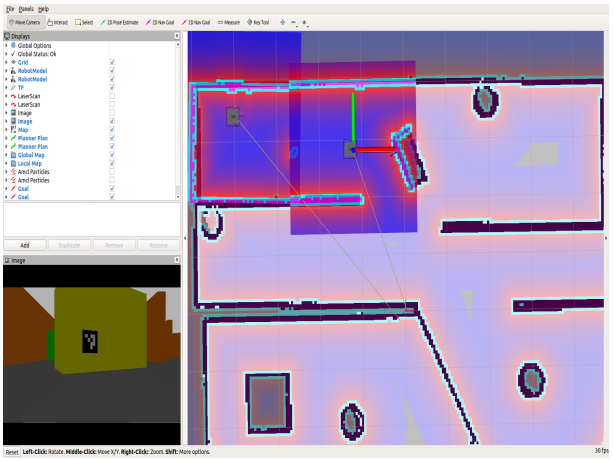

Figure 13: Explorer Robot detecting the new ArUco marker position in RViz

The simulation video of running the search and rescue activities for the generated test Gazebo world and YAML files is given in [3].

3 Challenges

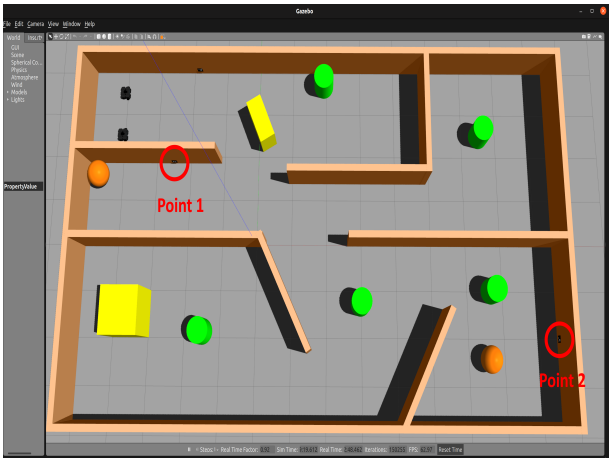

Figure 14: ArUco ID 1 and ID 2

- When the explorer robot is heading towards ArUco ID 1 (represented by Point 1 in [Fig 14]), it's camera briefly falls in the Line of Sight with ArUco ID 2 (represented by Point 2 in [Fig 14]), thereby causing the camera to accidentally detect ArUco ID 2 first despite this marker being more than 6 meters away from the camera. To address this issue, the distance of any detected ArUco marker is computed and if this value is greater than 3 meters, it is discarded. [Fig. 15] shows the position vector of the ArUco marker frame with respect to the camera frame, ${}^{c}_{a}P$. The norm of this vector gives the distance which is checked for unwanted ArUco detection.

  Ignore the detected marker if:

$$||{}^{c}_{a}P|| \geq 3.0$$

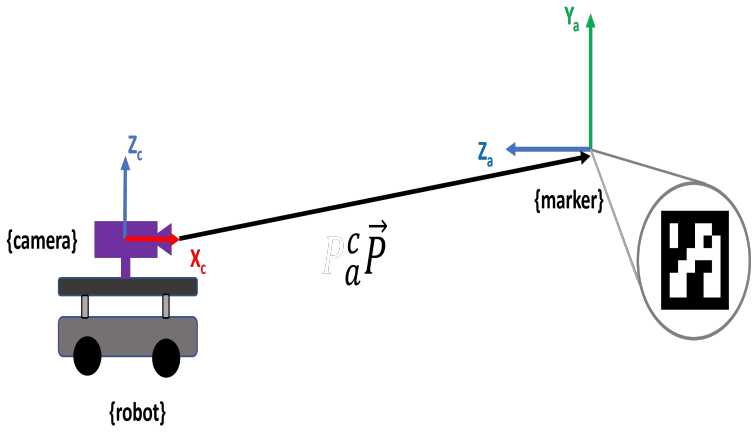

Figure 15: Visualizing the new ArUco marker (ID:0) in Gazebo

- In certain trial runs of the simulation, the explorer and/or the follower robots seem to "pushback" despite having a valid path from the current pose to the target pose. This pushback occurs for

a brief amount of time after which the robot starts following the computed path. One potential way of addressing this issue is to clean the ROS log files using the command *rosclean purge*.

- While running the explorer robot to reach the target ArUco position, on a number of occasions the *marker_frame* was not displaying under the TF dropdown in RViz for the first ArUco marker alone. However, this frame with the pose information was displayed for the other markers. This was deduced to be an issue with RViz and loading the simulation environment rather than our code. [1].
- During the creation of a custom Gazebo World with new ArUco marker positions, changing the orientation of the marker in the pose properties in Gazebo would sometimes make it continuously spin as if zero gravity. Due to this problem, we stopped with only changing one ArUco position in the *testing* phase since we were not able to address the root cause of this issue.

4 Project Contribution

- Adarsh - Worked on the documentation, generated basic code and algorithm for the application, calculated the distance from ArUco markers for the follower using trigonometric equations, did error debugging to address challenges mentioned in the previous section, also worked on broadcaster & listener implementation, wrote the code for rotating explorer after reaching the target.
- Ritvik - Point of contact between the Professor and the team, worked on implementing the tf broadcaster & listener, contributed during the implementation of the algorithm, ported the basic code to contain Object Oriented Programming, wrote ROS subscriber callback functions and made flowcharts for the functions implemented in the code.
- Sai Ram - Generated the test case's Gazebo world and YAML file, worked on retrieving the parameters and tasking explorer to reach the targets. Iteratively tested the code with different target locations and worked on tasking the follower to reach broadcasted targets. Contributed during the documentation of the simulations in videos.

References

[1] Project GitHub Repository - https://github.com/okritvik/ENPM-809Y-Final-Project.git

[2] https://www.youtube.com/watch?v=bqUnzQA4Owg

[3] https://www.youtube.com/watch?v=N_4cfe7u9H8

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

[5] http://docs.ros.org/en/api/geometry_msgs/html/msg/PoseWithCovariance.html

[6] https://githubmemory.com/repo/OpenVSLAM-Community/openvslam_ros/issues/37

[7] https://stackoverflow.com/questions/55406243/tf2-rosbuffercantransform-returning-false-for-existi

[8] https://answers.ros.org/question/359968/tf_repeated_data-error-with-robot_localization-ukf-package/