Final Project

Introductory Robot Programming

December 16, 2021

Students:
Nicholas Novak(118450933)

Orlandis Smith (11849307)

Jerry Pittman, Jr. (117707120)

Instructors:
Z. Kootbally

Group:
5

Semester:
Fall 2021

Course code:
ENPM809Y

Contents

List of Figures

1 Introduction

Urban Search and Rescue (USAR) is an important and challenging facet of robotics. USAR robots must move through a dynamic environment to find victims of disasters that could not otherwise be reached. Furthermore, these robots must be robust and have a carefully constructed codebase that allows them to handle the greatly unpredictable environments they operate in.

USAR requires a map of the environment to be made in disaster environments as these environments are unpredictable and have no prior mapping. These maps are then used to guide the USAR vehicles to identified victims (identified in real-life from thermal signatures or other life signs) for rescue.

Furthermore, an adaptive algorithm for movement around the environment must be employed. Commonly, Simultaneous Localization and Mapping (SLAM) algorithms are used to obtain a map of the environment, and an Adaptive Monte-Carlo Localization (AMCL) algorithm is employed for obstacle avoidance during operation.

1.1 Problem Statement

This project will produce a simulation of autonomous robots used in USAR to explore an unknown environment to find injured victims. Two waffle turtlebots will be used to represent the rescue vehicles. The first one, (*Explorer*), will use the targets provided to find victims, indicated by ArUco markers, also called fiducial transforms (essentially special QR codes for robots), and send that data to the second turtlebot (*Follower*) to fetch or further "triage" the victims in order of their marker ID values (0 through 4) representing their health status.

Both the SLAM and AMCL algorithms will be supplied since they are out of the scope of this project. Additionally, preexisting camera and transform packages are used.

1.2 Design Overview

This project will be developed on a Linux OS platform and will be implemented using C++. The robots used are pre-configured Robot Operating System (ROS) turtlebot3 robots with associated packages for movement, visualization, transformation, and detection. Distribution-specific packages of ROS Noetic and/or Melodic will allow for accessibility of needed files and functions for providing calls to the bots interfaces.

In addition, the packages will also allow for the developed algorithms to interface with the ROS server which contains all of the required pre-configured sub-components.

By utilizing these preexisting codebases, the developed code can interact with the ROS server via calls in developed launch and header files. These can be used to influence the movement of the robot in the environment.

There will be three main algorithms developed for the simulated USAR operation. These will be the explorer algorithm, detection algorithm, and follower algorithm.

2 Approach

To solve this problem, an Object-Orientated Programming (OOP) approach was taken. We used the provided *bot_controller* package as a reference for the two classes *Explorer* **(Algorithm 1)**, and *Follower* **(Algorithm 2)**, to represent the two robots, and developed a third, *ArucoNode* **(Algorithm 3)**, to detect and save the fiducial locations and IDs.

The explorer turtlebot would drive to the detected goals from a map obtained before operation, and use *ArucoNode* to find and store the fiducial IDs and marker locations. Then, these positions would be transformed to the map frame and the follower turtlebot would drive to their locations, in order of fiducial ID number.

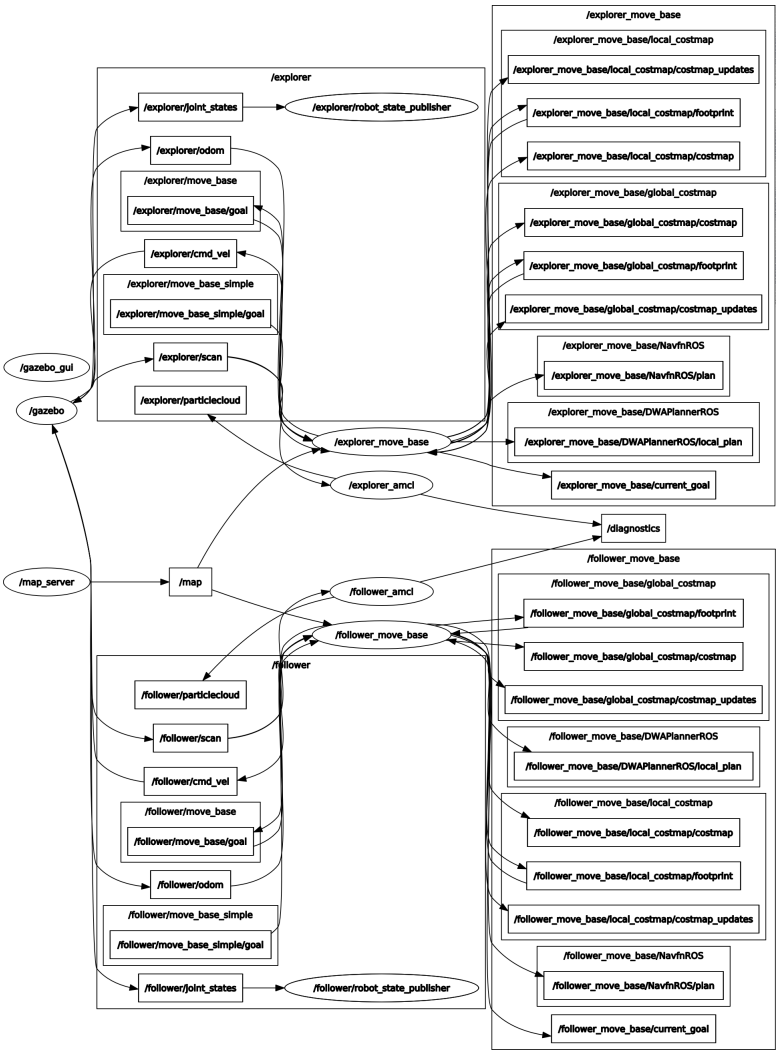


Figure 1: The topics and nodes tree during operation.

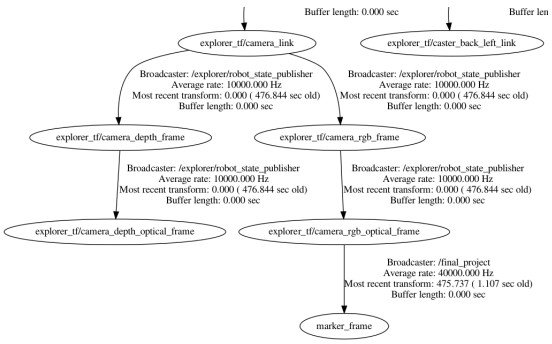


Figure 2: The *marker_frame* attached successfully to the transforms tree.

2.1 Target Environment

A simulated Gazebo environment was used to represent a collapsed building. A map of the area was performed using the laser scan prior to the search for **ArUco** markers to aid in quicker in-situ navigation.

2.2 Robot Initialization

We initialized the *Explorer*, *Follower*, and *ArucoNode* Class using the provided launch files and class header files. This allowed for the classes to communicate with each other, the XML-RPC remote server (for accessing the yaml target location files), ROS, and the Gazebo and RViz environments. This communication was done within each class or over ROS nodes and topics.

We used a ROS node handle for broadcasting and listening and to activate the publishing and subscribing actions. This allows the marker locations to be stored and translated in to the map frame. The *rqt_graph* figure in **Section 5** showcases the prescribed topics of the nodes.

The robots are configured as simulated differential drive turtle bot Waffle configuration. The bots are packaged with a camera, laser scanner, and a Proportional-Integral-Derivative (PID) controller for control of angular and linear speeds. We used the provided baseline Proportional controller code and did not tune or incorporate Integral or Derivative functions.

2.3 Marker Detection

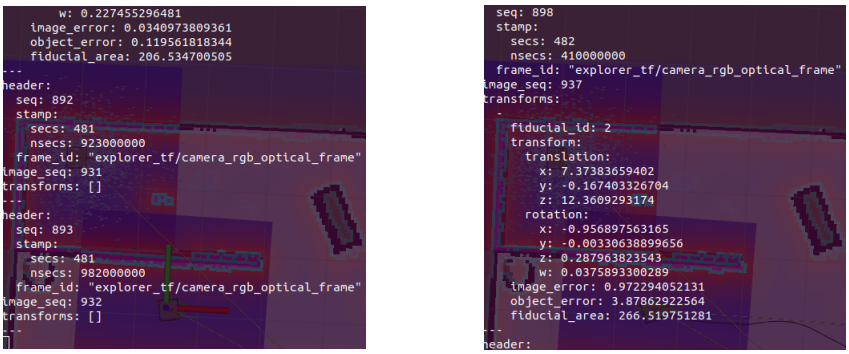
Next, the explorer travelled to the approximate victim locations, interpreted from the included *aruco_lookup.yaml* file with targets, and looked for the markers by spinning slowly and monitoring the $fiducial\_transforms$ topic, as seen in **Figure 3** and **Figure 4**.

The severity was assessed by the ID number of each fiducial marker (0, 1, 2, or 3). Then, each marker would be attached as a child frame to $explorer\_tf/camera\_rgb\_opticl\_frame$, as seen in **section 2.0.2**. This was then transformed into the *map* frame using a listener listening to the $fiducial\_transforms$ to obtain the location of each marker relative to the world.

Then, these transformed locations and their attached fiducial IDs were passed to the follower as the exact locations of victims and their injury severity, respectively.

2.4 Broadcaster Messages

The broadcaster was used within a callback function that was called whenever $fiducial\_transforms$ was updated. This broadcaster would attach the data from $fiducial\_transforms$ in the transform section of **Figure 4** to a new frame and attach it to $explorer\_tf/camera\_rgb\_opticl\_frame$ frame.



(a) No ArUco Detected (b) ArUco Detected

Figure 3: Images of **Topic:** /fiducial_transforms

Additionally, the ID of the current marker would be stored in a variable that could be called later. In other words, the broadcaster was only called when an **ArUco** marker was seen, and would pass necessary data to the map frame for use by the follower. This was instantiated in the *ArucoNode* class and called by the explorer robot during its operation.

2.5 Listener

Next, a listener was called to monitor $fiducial\_transforms$ for changes and, when a change was detected, look at the explorer camera portion of the tf tree, as seen in **Figure 2**. From there, it would obtain the transformation of the position of the newly added marker with respect to the map frame, which was a constant parent frame of all others. The the marker was defined with respect to the explorer by default, which was not useful for the follower, hence the transformation. This is useful because then the follower can grab these goals and drive to their exact locations.

2.6 Target Identification

To detect the targets, an included ROS package, *aruco_detect* was used to setup the camera on the explorer, which included presets for detected image locations. Since the markers were also predefined in the *aruco_detect* library, finding them within an image was easy and no additional code was needed, whereas a normal vision algorithm can be time-intensive to develop.

2.7 Return to Base

After each robot performs its task, it will return to its starting location, simulating the relaying of data to any rescuers in that area. The follower will not start until the explorer has performed this action, and the simulation ends when the follower performs this action.

2.8 Algorithms

Algorithm 1 Explorer

```
1: procedure READ_WAYPOINTS(waypoints_path)
2:     procedure READ_YAML(waypoints_path)
3:         waypoints_path ← loadwaypointspath
4:         try
5:         waypoints ← readwaypointsfromthefile
6:         except
7:         stdout ← exception
8:         return waypoints
9:     end procedure
10:    pose ← extractpositioncoordinates
11:    frame_orientation ← extractframecoordinates
12:    pose.extend(frame_orientations)
13:    output ← positioncoordinates
14:    return output
15: end procedure
16:
17: procedure LEADER_CLIENT(coordinates)
18:    client ← actionlib.simpleactionclient(move_base, MoveBaseAtion)
19:    client.wait_for_server()
20:    goal ← MoveBaseGoal()
21:    goal.target_position ← fromcooardinates
22:    client.send_goal(goal)
23:    wait ← client.wait_for_result()
24:
25:    if not wait then
26:    stdout ← Actionservernotavailable!
27:    stdout_shutdown ← Actionservernotavailable!
28:    else
29:    returnclient.get_result()
30: end procedure
31:
32: procedure MAIN
33:    rospy.init_node() ← initializealeadernode
34:    locationsreadwaypointsfromwaypoints_path
35:    try
36:    for location, coordinates.in locations do
37:      end
         if coordinates is not None then
38:    result ← leader_client(coordinates)
39:    if result then
40:    stdout ← locationreached
41:    except
42:    stdout ← exception
43: end procedure
```

Algorithm 2 Follower

procedure READ_ARUCO_LOOKUP_LOCATIONS(waypoints_path) **if** coordinates **is not** None **then**
    $result \leftarrow leader\_client(coordinates)$
    if distance_to_goal 0.05**then**
    $result \leftarrow \_client(coordinates)$
end procedure
$pose \leftarrow extractpositioncoordinates$
$frame\_orientation \leftarrow extractframecoordinates$
$pose.extend(frame\_orientations)$
$output \leftarrow positioncoordinates$
return output

procedure MAIN
    $rospy.init\_node() \leftarrow initializealeadernode$
    procedure FOLLOWER_CLIENT(coordinates)
        $client \leftarrow actionlib.simpleactionclient(move\_base, MoveBaseAction)$
        client.wait_for_server()
        client.wait_for_result()
        $goal \leftarrow MoveBaseGoal()$
        $goal.target\_position \leftarrow fromcooardinates$
        try
        $m\_posit \leftarrow readwaypointsfromtransformStampedoffiducial\_msgs :: FiducialTransformArray$
        $m\_fid \leftarrow readfiducial\_idfromfiducial\_msgs :: FiducialTransformArray$
        for *iterator.**in*** $fiducial\_ids$ **do**
    end
    Save m_posit array based on $fiducial\_id$ read and store fiducial IDs into class array m_fid. **if** coordinates **is not** None **then**
        Send $follower$ robot to $fiducial\_id$ waypoints using a For loop and go_to_goal function.
        Send $follower$ robot to Starting Position (-4,3.5) once reaches Fiducial ID 3 waypoint.
        ros::shutdown();
    end procedure

Algorithm 3 ArucoNode

procedure MARKER_LOCATIONS(tf2 Buffer, count number)
    $CallSUBSCRIBERS$
    if no error seen **then**
    $Calllistenerbetweenmapandmarkerframes$
    $Spinrosforsubscribercallback$
end procedure

procedure SUBSCRIBERS
    $Callsubscriberinitialization$
    $Initializesubscribertothefiducial\_transformstopic$
    $CallCALLBACK$
end procedure

procedure CALLBACK
    if marker is seen **then**
    $Initializebroadcaster$
    $Setupnewframe"makrer\_frame"aschildof"explorer\_tf/camera\_rgb\_optical\_frame"$
    $Obtainpositionandrotationdataofthemarkeronthisframe$
    $Broadcastnewframedata$
end procedure
    =0

3 Challenges

- One of the challenges we encountered was how to save the fiducial IDs where they can be accessed. We decided to go with public class variables m_fid (array for fiducial IDs) and m_posit (array for fiducial ID marker positions) to get updated in the $fiducial\_callback$ function of $Bot\_Controller$ and then called in the main function and saved into local variables for iterative use.

- We had issues with using various msgs and topics and it was determined that we didn't include the header files initially.

- We also had issues with inherited classes so shifted our $Follower$ and $Explorer$ classes to be independent classes and removed all **bot_msgs** and **bot_controller** files.

- Once we was able to get a successful build we had errors and issues with rosrun of our main.cpp on our included header files and doxy comments. This was due to use not including "$\_node$" at the end of our executable name, matching the cmakelists.txt file.

- Another issue we had was that we were initially subscribing and publishing to the same topics (i.e $/odom$) versus $/explorer/odom/$ and $/follower/odom/$.

- Another issue we had was the robot(s) would get to a location and publish their next goal locations but not move. This was since the "BOT_goal_sent was not reset to "false" after reaching a goal for the next goal to be sent in the next iteration of the respective for loop. Additionally, for this issue we encountered how **nodehandle**'s built in $sendGoalAndWait$ was not as useful (had worse results) then $sendGoal$, at times.

- We additionally had issues in that while debugging the code we needed to close Gazebo/RViz, re-build the catkin workspace, then re-run to verify vice being able to see in-situ update/changes.

- Finally, we were not able to adequately perform the listening function. While it would be called, we were calling it after a subscriber call and, as such, it was only called once and did not have

time to find and transform the detected marker. Given more time, this can be amended by simply allowing more time for the listener to listen to given transforms on the tf tree. Instead, to meet time constraints, the follower simply traverses to the explorer's goals, albeit in increasing order of fiducial ID instead of the provided order.

4 Project Contribution

- Orlandis worked on pulling the data from the camera and the m_fiducial_callback functions as well as the research into the various msgs (i.e. fiducial, geometry, odometry) used.
- Jerry worked on the *Follower* class member and tasks. He also worked on debugging the main, integration and interactions of the two robots, sorting of the fiducial IDs, and creation/updating/maintaining of the doxygen documentation. He is also the maintainer of the git repository.
- Nicholas built the *Explorer* and *ArucoNode* header and cpp files. He also worked on debugging for the *main* file in order to effectively drive either robot to a goal, as well as debugging with Jerry on the *Follower* file for attempts at a successful listener. For the explorer portion of *main*, he wrote methods to receive a goal and detect the **ArUco** marker at the goal. For the follower portion of *main*, he worked with Jerry to write methods that received fiducial IDs and to drive to the explorer goals, in order of ID, when the listener was no longer attainable. Further, he developed the listener and broadcaster applications as well as their interactions within each file.

  He also may have watched one Netflix episode over the course of the project.
- We all worked on the report and cleanup of unused code.

5 Resources

- We used the provided **bot_controller** and **final_projectmain** files to build our files from.
- Our Github repository: `https://github.com/jpittma1/809Y_final_project_Group_5.git`

6 Course Feedback

- Nicholas Novak: Now that I have completed two courses with Dr. Kootbally, I know that his classes are both challenging and rewarding. The hours spent debugging assignments really opened my eyes to more effective techniques of comprehending and modifying C++ files. I know that, come future applications, I will be well-prepared to write robot control algorithms in C++ or Python, especially for hardware integration next semester in 809T. I look forward to diving even further into these programming topics in the near future, and applying my ROS skills in a future career.

  Also, seeing Jerry's excitement for 809E over the summer, if I my degree plan extands to that time, feel free to reach out if you need a TA!
- Jerry Pittman, Jr.: I really liked the organization of the course and lecture material as well as the hands-on nature. The lecture material was so good that use of the textbooks was rarely required of me except during researching proper C++ etiquette and when working on the projects. The professor and TA were both very accessible and helpful whenever we had doubts throughout the semester making the class feel personalized and containing far less students then it actually did. I also found the explanation and teach of ROS much better than what is available on various forums or outdated web textbooks. I wish I had the ROS portion before my

ENPM662 course which was very ROS heavy and didn't seem to know what I was doing until right before turning in that final project based on what I learned in this class. The other complication was that we was using python vice C++ but I would like to be fluent and competent in both languages in addition to MATLAB. I look forward to taking 809E (Please teach it during the summer!).

- Orlandis Smith: I did not realize until I took this course that I would be in for a journey, and well it was a journey. I came into the course thinking that it would be a basic programming course in C++ but I was wrong. I actually learned a lot and had a chance to learn Linux which is really important in industry. What I like most about the course is that it challenges you and a new developer will have the opportunity to know how to deal with real world problems, that of which are needed when going into a tough career. Thanks to Dr. Zeid, he has been patient with me and has been helpful in understanding that I do not have the strongest background in coding. I really appreciate this class and look forward to how the upcoming semesters will be.