Final Project

Introductory Robot Programming

December 16, 2021

Instructors:
Z. Kootbally

Group:
Group 3

Semester:
Fall 2021

Course code:
ENPM809Y

Students:
Douglas Summerlin

Rajan Pande

Contents

1 Introduction

Urban search and rescue (USR) operations require the navigation of treacherous environments such as fallen buildings and other destroyed structures to locate trapped victims and extract them for medical attention. Victims of collapsed buildings are often trapped in large voids in the debris and require rescue operators to quickly locate and remove them from the wreckage before the building continues to collapse. The inherent dangers of these unstable structures and their susceptibility for further collapse makes the location and rescue of trapped victims an interesting area for employing autonomous robotics. By utilizing autonomous robotics to execute search and recovery missions in hazardous environments the endangerment of first responders can be mitigated, as they would only have to enter the hazardous structure if a living victim suitable for rescue was located by the navigational probe robots. The present report summarizes a simplified urban search and rescue simulation executed in ROS to illustrate the advantages autonomous robotics can provide in assisting such operations.

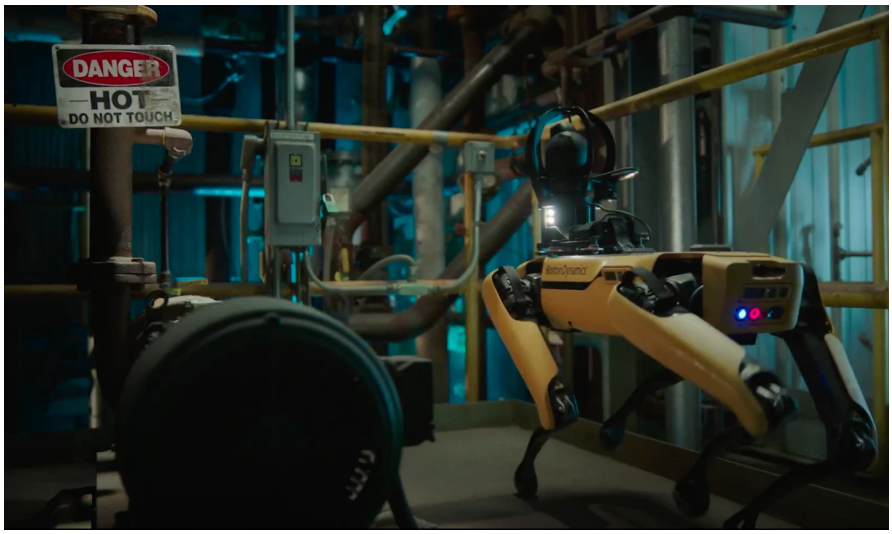

Figure 1: Boston Dynamics "Spot" robot operating remotely in a hazardous environment

A logical method of using autonomous robotics to execute USR tasks is to use a first robot, henceforth referred to as the "explorer" robot, to probe a hazardous urban environment for potential victims. The explorer robot will map the collapsed building while it navigates throughout it using mapping technologies such LIDAR, SONAR and computer vision to create a visualization of the environment for the first responders. As the explorer robot continues to search the debris, other sensing technologies such as thermal sensors or cameras can detect the presence of a trapped or otherwise unconscious person. Once a person has been detected by the explorer robot, the robot will make note of the victim's location relative to the robots starting position, so that the USR crew will know exactly where the victim and can formulate a recovery strategy for that victim that minimizes the chance of jeopardizing the first responders.

To simulate this methodology in ROS, our group was tasked with writing a program in ROS that would command an explorer robot to probe a simulated urban disaster environment to look for a variety of potential targets to be rescued. Once the explorer robot had completed its search of the environment, a second "follower" robot would travel to the locations of the discovered victims provided by the explorer robot to represent the recovery phase of the mission. The urban disaster environment is simulated in Gazebo by a map with a variety of obstacles to impede the explorer

robot's path, to make it more realistic to the unpredictable terrain encountered in actual USR situations. The victims in the simulation were represented by a series of four ArUco markers, each with a unique ID number (1-4) and location within the environment.

To find the ArUco markers, we were instructed to program the explorer robot to navigate to four discrete coordinates in the environment and complete a slow 360 degree rotation while using a ROS node called `aruco_detect` to scan the environment for a possible ArUco marker. If an ArUco marker is detected, the explorer robot would make note of its ID and location within the map reference frame to send to the follower robot. After a given ArUco marker is detected, the explorer robot is to travel to the next location in the map and repeat the scanning procedure until all of the four map targets are reached and scanned for ArUco markers. Once all of the target locations had been visited in the correct sequence, we had to instruct the robot to return to its home location so the follower robot could begin its recovery mission. The follower robot was relatively simple, as it only had to retrieve the positions of the ArUco markers stored by the explorer robot and visit them in the same sequence as the explorer robot. Once the explorer robot has visited all four target locations, it is to return back to its home location to represent removing the disaster victims from the hazardous debris zone. Once the follower robot successfully returned home, the project was complete and the program could be terminated. The Gazebo environment and a collection of ROS nodes were provided to our group such that the collection of tasks we had to complete to successfully solve the problem outlined above were greatly reduced. The present report details our approach in accomplishing the simplified USR simulation by constructing an ROS node (referred to as the `final_project` node) that executes the necessary actions to complete the mission.

2 Approach

To outline the approach our group utilized when completing the simulated USR mission in ROS, it is first important to briefly present the initial functionality in the provided ROS package at the beginning of the project. Our group was provided a setup package which included a variety of ROS programs and tools. This package would initialize the simulated urban environment in Gazebo, the explorer and follower robots (represented by ROS Turtlebots), and a visualization environment in RViz used to monitor a given robot's trajectory. All of these elements were initiated with the launch of a single ROS executable, such that our team was tasked with creating one additional executable that would complete the simplified USR mission detailed in the introduction. With these important elements of the project supplied to us from the beginning, our team was able to focus on the problems of locating the target locations in the map, moving the robots to the target locations, detecting the ArUco markers with the explorer robot, and storing ArUco marker locations for use during the recovery phase of the mission.

The first item our team needed to complete was obtaining the locations of the ArUco markers from the ROS parameter server. The explorer robot has knowledge of the size of the given map environment and can autonomously detect objects along its path to a certain location by using the included `move_base` ROS package. However, the explorer robot has no inherent knowledge of the locations of the ArUco markers. To determine the locations to search for the ArUco markers, our `final_project` node program had to possess the ability to acquire the four target locations of the map by reading ROS parameters off of the parameter server. Hardcoding the discrete coordinates for the explorer robot is not only bad programming practice, but it would also cause difficulty if the simulation was run with different ArUco marker locations. As a result, the method of extracting these parameters from the parameter server was devised while recognizing that the coordinates of the target locations could change when running the program with different ArUco marker locations on other simulation maps. There are 4 total target parameters that represent the location the explorer bot should navigate to before attempting to scan the area for ArUco markers, and not the actual position of the markers themselves. Each target parameter stores the (x,y) coordinate of the target location in an array and is also given a name, ie. `target_1, target_2`, etc. A C++ function titled `getParam()` exists to collect parameters from the ROS parameter server given a parameter name, but this function is only compatible with simple data types such as strings or integers. To collect the target parameters from the parameter server, our team utilized a special C++ class called `XmlRpc::XmlRpcValue` to extract the target location parameter arrays and store them in variables in our program. The (x,y) elements of the arrays were then extracted and stored in other variables in the program as primitive data types compatible with the C++ standard library for easy reference.

Once the target locations were stored in the `final_project` node program, our team had to devise an algorithm to make the explorer robot move from location to location and look for the ArUco markers. The locations from the parameter server are identified numerically from 1-4, and the explorer robot is required to reach each location sequentially in numerical order. However, because the explorer robot has no prior knowledge of the map before moving, it needs to plan a path to reach a given target location as quickly as possible while avoiding obstacles. The path planning, navigation, and actuation of both the explorer and follower robots is executed using the ROS package `move_base`. The `move_base` package has methods that globally and locally map optimized paths toward a given coordinate in the map, and then also moves the robot along the generated paths until it reaches the goal. However, our team still had to devise an algorithm for making the explorer and follower robots move to the right target location at the right time. We decided to take a procedural approach to the movement algorithm, creating a sequential series of commands that the program would execute only if certain conditions were met. This series began with sending the first target location to the explorer robot and waiting for it to arrive. Once the explorer successfully arrived

at the first target location, it would execute a separate marker detection algorithm. If the ArUco marker was successfully detected by the explorer robot, it would then continue to the next target location, repeating until it had visited all of the target locations. Whenever the explorer robot is done visiting all of the target locations, we command the robot to return to its starting position so the follower robot could begin its route. The algorithm pseudocode below describes the program flow for moving the explorer throughout the map:

Algorithm 1 Explorer Robot Movement Algorithm

```
data goal1                                              ▷ repeat for goals 2-4
data home
data botswitch = false
data goalsent = false
set goalheaderframe ← mapframe                          ▷ repeat for goals 2-4
set goal1coord ← target1coord                           ▷ repeat for goals 2-4
---
make buffer object
make transformlistener object
set rosRate ← 10
---
while ros node is active do
    ---
    if botswitch and goalsent are both false then
        print: sending bot to goal 1
        call movebase function                          ▷ sends bot to goal
        set goalsent ← true
    end if
    call waitforbot function                            ▷ waits for bot to arrive at goal
    if bot reaches goal 1 then
        print: hooray bot reached goal 1
        print: beginning detection algorithm
        call detection algorithm                        ▷ scans area for aruco marker
        set goalsent ← false
    end if
    ---          ▷ repeat the above procedure from "If botswitch..." for goals 2-4

    if botswitch and goalsent are both false then       ▷ now send bot home
        print: sending bot home
        call movebase function                          ▷ sends bot home
        set goalsent ← true
    end if
    call waitforbot function                            ▷ waits for bot to arrive home
    if bot reaches home then
        print: hooray bot reached home, now followers turn to move
        set goalsent ← false
        set botswitch ← true
    end if
end while
```

Whenever the explorer robot arrives at a target location, it needs to execute a detection algorithm to scan the area for ArUco markers. To do this, the robot rotates slowly in place while it autonomously scans for ArUco markers using the `aruco_detect` node. To make the explorer robot rotate in place, our team utilized a C++ package provided to us by the instruction team earlier in

the semester that could command the turtlebot to move upon command in a simulated ROS environment. This package, titled `bot_controller`, contained methods for passing linear and angular velocities to a given robot using the Linux command line. Our mission required autonomous completion of the USR simulation after the launch of a single executable file, so most of the methods in the `bot_controller` class provided little utility as the package was primarily designed for real-time user command execution within the Linux terminal. Our team utilized two methods from the package titled `m_move` and `stop` which required no input from the command line and could be used procedurally in our C++ program. The `m_move` method was used for passing angular velocities directly to the topic `explorer/cmdbot_vel`. This method was executed whenever the explorer reached a target location, as the explorer bot would reorient itself at the target frame and stop if no further command was given to it after arrival. Once the explorer had reoriented itself, we used the `m_move` command to pass a small angular velocity to the explorer robot so it would begin to rotate slowly in place. No translational velocity was sent to the explorer robot because translational movement was not desired. If the ArUco marker is present at the target location, we halt the explorer robot's rotation using the method `stop` which simply zeros all velocities of the explorer robot. The detection of the ArUco marker also breaks the detection algorithm loop so the explorer robot can then move to the next target location.

Algorithm 2 Marker Detection Algorithm

while boolean **do**
  begin rotating explorer bot CCW
  do ROS spinOnce
  if the marker has been found **then**
    stop the explorer bot from moving
    print: marker found
    print: fiducial marker id
    set *boolean* $\leftarrow$ *false*
    ———
    call listener function ▷ computes pose of marker frame in map frame
    set *markerlocation* $\leftarrow$ *markercoordinates*
    ———
    call tolerance function ▷ computes adjusted coordinates for follower robot
    set *followerlocation* $\leftarrow$ *tolerancecoordinates*
  else !boolean
    continue with rest of program
  end if
end while

While the robot is rotating, the camera node on the explorer turtlebot scans the image collected by it's camera for the presence of an ArUco marker using the `aruco_detect` node. The `aruco_detect` node publishes a message containing the position of the ArUco marker as a child frame of the explorer robot's camera to the topic `fiducial_transform`. This message contains the rotation and translation of the child marker frame to completely describe its position and orientation, as well as an identification tag denoted `fiducial_id`. Collecting all of this data from the `aruco_detect` publisher is vital for the simulated USR mission because the follower robot has no method of detecting ArUco markers or any inherent knowledge of their location. As a result, the follower robot needs to be told the exact position of the ArUco marker in the frame of the map, as well as the `fiducial_id` of the markers so the follower arrives at the markers in the correct sequential order (the numerical order of the marker IDs). Our team determined we needed to write our node's program such that it was able to extract this data from the `fiducial_transform` topic upon detection of an ArUco marker and store the data in the program for later use by the follower robot.

To gather the ArUco marker information from the `fiducial_transform` topic, our team was tasked with subscribing our `final_project` node to the topic so that our program had access to the message data published and could store the data for later use. When subscribing a ROS node to a topic, callback functions are used to interpret each individual message pulled from the topic from the subscriber, as the function is called once for each message received by the subscriber. This is important because we needed to analyze each incoming message from the topic to ensure we did not miss the detection event triggered by the `aruco_detect` node. We used a callback function (partially provided by the instructional staff) to retrieve the message sent along the `fiducial_transform` topic from the publisher and store the message data into storage variables for the follower robot to use later in the mission. The location data was sent as a series of float-type variables with unique identification IDs for translation and rotation, and the `fiducial_id` was sent as a single int-type variable.

Once subscribed to the `fiducial_transform` topic, our team used a callback function titled `fiducial_callback` to read the queue messages from from the topic and determine from the data in the message whether or not an ArUco marker had been detected. To do this, the callback function reads a specific entry in the message that is only populated when an ArUco marker has been detected. If this entry is not empty, the callback function will recognize this message as a detection event and store the data from this message into storage variables in the program. These storage variables are then loaded into an `array` element corresponding to the identification number of the ArUco marker to be used later by the follower robot to travel to the coordinates of the markers in order. The message from the `fiducial_transform` topic is passed as a reference to a pointer in the callback function, so to access its variables we needed to use the accessor operator `(->)` to extract the data from the variable for a given identification path. Our team needed to extract all of the `translation` and `rotation` elements as well as the `fiducial_id` from the detection event message to fully describe the ArUco marker for the follower bot. To do this, we defined global variables in the C++ program to temporarily store the values accessed by the assessor operator. To determine the ID path of each variable in the message, we used the linux command `rostopic info fiducial_transforms` to determine the type of message sent across the `fiducial_transforms` topic. We then used a separate linux command `rosmsg show fiducial_msgs/FiducialTransformArray` to display the name and data type of each variable in the message type sent across the topic. With the name and data type of each variable, our team then created compatible storage variables for each necessary variable to store the data from the message for later use. A brief description of the callback algorithm is detailed below in Algorithm 3.

Algorithm 3 Callback Function Algorithm

Require: *message*
  if the marker is detected **then**
    set $markerfound \leftarrow true$
    make broadcaster **object**
    make transform **object**
    ———
    set $IDstorage \leftarrow IDmessage$
    set $transformX \leftarrow messageX$
    set $transformY \leftarrow messageY$
    set $transformZ \leftarrow messageZ$
    ———
    set $transformroll \leftarrow messageroll$
    set $transformpitch \leftarrow messagepitch$
    set $transformyaw \leftarrow messageyaw$
    set $transformW \leftarrow messageW$
    ———
    broadcast transform
  else
    $markerfound \leftarrow false$
  end if

The last line of the callback function pseudocode above represents the use of a broadcaster function. The `fiducial_callback` function executes the `broadcast()` function to send the marker_frame as a child of the camera_frame to the `/tf` topic for the listener to receive later in the program. This is imperative for the proper description of the ArUco marker position because when the `aruco_detect` node detects the presence of the ArUco marker, it describes its position relative to the frame of the camera module on the explorer turtlebot. Due to the nature of a search and rescue mission, this description can be problematic because the location of a potential victim has to be described in the frame of the global environment to give rescuers the correct location of the victims relative to their position or the launch position of the searching robot. In our simulated mission, we wanted the follower robot to be able to navigate as close to the detected ArUco markers as possible (without crashing into the wall) so we needed the exact position of the ArUco marker in the frame of the Gazebo map so we could navigate directly to the marker. This could represent an autonomous robot being sent to a disaster zone to pick up a human victim for extraction in the context of urban search and rescue. To transform the pose of the marker from the camera_frame to the map_frame, our team was instructed to use the broadcaster function to broadcast the marker_frame as a child frame of the camera_frame whenever the detection event was triggered. A corresponding listener function would then receive the marker frame from the broadcast and compute the pose transformation from the marker_frame to the map_frame so the follower robot would know the exact location of each marker relative to the frame of the Gazebo map environment. This is important because the coordinates passed to the explorer and follower robots using the `move_base` node are expressed in the frame of the map, so for proper navigation goal coordinates must be expressed in the map_frame.

Once the marker_frame is broadcast to the listener function over the `/tf` topic as a child frame of the camera, we used the listener function to execute the pose transform of the ArUco marker position from the marker_frame to the map_frame. The listener function is passed the marker_frame from the broadcast function as an argument. A "`Try`" block is used in this program to define a series of statements that could possibly throw an exception. If a specific type of exception occurs, a "`Catch`" block catches the exception and sends a warning statement to the terminal. In the "`Try`"

block the listener is tasked with listening to the `/tf` topic to look for a specific possible transformation (marker to map). If the necessary frames are sent over the topic, the listener will execute a function `lookupTransform()` to execute the transformation of the marker pose in the marker_frame to the map_frame. The coordinates of this transformed pose represent the position of a given ArUco marker within the frame of the map. The coordinate outputs of the `lookupTransform()` function are stored in global variables and passed to the follower robots coordinate array in the element corresponding to the marker identification number. Once all four ArUco markers are visited, this array will be filled with the marker coordinates for the follower robot to visit once it begins its rescue mission when the explorer returns to its home location.

The follower movement algorithm is somewhat simple. To ensure the follower robot does not move when it is the explorer's turn to execute its mission, we implemented a boolean "switch" into the program as a variable to control which robot is moving. If the switch is set to `false`, the explorer robot will move and the follower robot will not. Conversely, a `true` value for the switch allows the follower to move and the explorer will not. This boolean switch is initialized with a `false` value which is then changed to `true` whenever the explorer has returned home after visiting all of the ArUco marker locations. The path planning, navigation, and actuation of the follower robot is executed using the ROS package `move_base`, just the same as the explorer. The goal coordinates for the follower robot are the transformed pose coordinates of the ArUco markers expressed in the map_frame placed in the follower coordinate storage array by the listener function. Our team programmed the follower robot to navigate to each of the four elements in the array in numerical order, which corresponds to the correct numerical order of the `fiducial_id` of each marker (0-3). However, because the coordinates give the exact position of the ArUco marker on the wall of the map, the follower robot will crash into the wall if it were to try to reach the exact coordinate of a given marker. To mitigate this issue, our team implemented a tolerance zone, where if the follower robot navigated to within approximately 0.4m of the marker's position, the follower would count the marker as visited and move to the next marker. To do this, our team wrote a function that takes the target location of a given ArUco marker and the actual position of the ArUco marker itself as arguments. The function creates a line between these two coordinate points and then creates a third coordinate point 0.4m along the line segment from the ArUco marker coordinate point. This third coordinate point is then 0.4m from the ArUco marker position and can be used as the (x,y) coordinate to send the follower robot to with the `move_base` package. This function is executed to find new coordinates with computed tolerances for all four ArUco marker locations for the follower to travel to. Once the follower has reached all four locations, it executes a `ros::shutdown` command to shutdown the node and complete the project. The pseudocode of the follower robot movement algorithm is given below.

Algorithm 4 Follower Robot Movement Algorithm

```
data goal1                                                  ▷ repeat for goals 2-4
data home
data botswitch = true
data goalsent = false
set goalheaderframe ← mapframe                              ▷ repeat for goals 2-4
set goal1coord ← tolerancecoordinates                       ▷ repeat for goals 2-4

while ros node is active do

    if botswitch is true and goalsent is false then
        print: sending bot to goal 1
        call movebase function                              ▷ sends bot to goal
        set goalsent ← true
    end if
    call waitforbot function                                ▷ waits for bot to arrive at goal
    if bot reaches goal 1 then
        print: hooray bot reached goal 1
        set goalsent ← false
    end if
    ------   ▷ repeat the above procedure from "If botswitch..." for goals 2-4

    if botswitch is true and goalsent is false then         ▷ now send bot home
        print: sending bot home
        call movebase function                              ▷ sends bot home
        set goalsent ← true
    end if
    call waitforbot function                                ▷ waits for bot to arrive home
    if bot reaches home then
        print: hooray bot reached home, project complete
        ros::shutdown                  ▷ node killed upon followers successful return home
    end if
end while
```

3 Challenges

The first challenge our team faced was the challenge of retrieving the `fiducial_id` data from the message received from the `fiducial_transform` topic. While completing this portion of the assignment, we conceptually struggled to grasp the way the message was being received by the subscriber and how the callback function was extracting and storing data from the received message. We initially misunderstood the way the values are retrieved from the message passed to the callback function, and were attempting to refer to the values from the message in the `while(ros::ok)` loop, instead of with pointers in the callback function. Once we understood that the code in the callback function should access the data of a specific variable in the message `pointer` and store that data in another variable, we were able to collect all of the necessary translation and rotation data from the published message. To obtain more information about the `fiducial_id` data, we launched the `multiple_robots.launch` executable in the terminal and used the command `rostopic echo fiducial_transform` in another terminal to see the actual message published on the `fiducial_-transform` topic. This allowed us to see the identification path and name for each data element stored in the message. We determined the correct identification path for the `fiducial_id` data to be `transforms.at(0).fiducial_id` so we accessed this element of the message to extract the value of this data into a storage variable. We defined the type of the storage variable for the `fiducial_id` data by using the command `rosmsg show fiducial_msgs/FiducialTransformsArray` to display the data types of each of the elements in the message. The data was successfully stored once we used a compatible data type (`int`) to store the `fiducial_id` data.

Our team encountered a second odd issue while conducting trial runs of our project, which was when our explorer robot arrived at the second target location and began its scan for the second ArUco marker (Fiducial ID: 1), it would instead detect the ArUco marker at the fourth target location instead (Fiducial ID: 2). We first noticed there was a problem when the explorer robot would arrive at the second marker. Whenever the explorer robot began its rotation at the second marker it would immediately detect the presence of a marker and continue its route to the third ArUco marker, despite not even actually turning to face the second ArUco marker. We at first did not realize that the fourth marker was actually the one being detected by the camera, and attempted to look back at our program to make sure the movement algorithm did not possess any errors. The movement algorithm appeared to be correct because the behavior of the explorer at all of the other target locations was as we expected it to be.

After analyzing the map from the explorer robot camera perspective at the second location, we noticed that the ArUco marker at the fourth target location had an unobstructed line of sight to the camera at the beginning of the scanning phase of the second target location, giving our team a hypothesis to test. It is worth noting at this point we had not determined how to print the `fiducial_id` data to the terminal, so we attempted to troubleshoot the issue by implementing this functionality. Once we had properly stored and printed the `fiducial_id` data to the terminal, we ran our mission program and observed the marker ID printed at the second target location. The program printed "fiducial id: 2" at the second target location, but "fiducial id: 2" is the marker at the fourth target location, so our theory gained credibility. To further confirm this suspicion, we added a large cubic block to the gazebo marker to obstruct the line of sight between the camera and the ArUco marker at the fourth target location. This fixed the problem and the mission executed exactly as intended with the inclusion of the block in the map environment, so we could confirm the source of the issue was absolutely the explorer robot erroneously detecting the wrong ArUco marker because it was visible in the camera's image. Our team suggested some possible solutions to this behavior, maybe reorienting the robot upon arrival to a certain location, changing the direction of rotation of the robot, or somehow reducing the effective range of the explorer robot's camera. Upon consultation with the instructions staff about this issue, they implied that this behavior could be programmati-

cally corrected by reading the translational distance of the marker relative to the explorer robot and disregarding the detection event of the marker if it exceeded some set distance threshold. The instructional staff also indicated that the ArUco markers in the live project demonstration would be oriented such that this issue would not occur, so our team did not prioritize implementing this functionality in the interest of time.

The final and largest challenge our team encountered was passing the correct `/map`-transformed marker coordinates from the storage variables to the follower robot. We first noticed this problem after we confirmed the explorer robot exhibited the full range of functionality required of it, including visiting each marker location in order using data from the parameter server, scanning the location for the presence of a marker, and storing the positional and identification data from the detected marker in the program. The program written by our team at this point seemed to work as intended, so we progressed to the follower phase of the USR mission simulation, which tasks the second (follower) robot with visiting the locations of the ArUco markers detected by the explorer robot. Due to the nature of urban search and rescue missions, our team was instructed to position the explorer robot as close as possible to the ArUco marker without crashing into a wall to more accurately simulate the retrieval of a human victim from a disaster environment. The instructional staff also indicates that a navigational tolerance value of .4m should be assigned to the follower robot to indicate successful arrival to the marker, because the follower robot would crash into the map wall if the robot was instructed to navigate directly to the ArUco marker's exact location. Our team decided to first design and test our initial follower movement algorithm without the inclusion of the specified tolerance just to observe the behavior of the follower robot. Upon testing our initial program for the follower robot movement algorithm, our follower robot visited the four markers in the correct order according to each marker's respective `fiducial_id` before returning home. Unfortunately, the follower robot navigated to approximately the same area as the explorer robot at each location, but never ran into the map wall at any ArUco marker target location. This behavior suggested that there was an issue with the `listener()'s lookupTransform()` function, such that the transformation between the explorer robot's camera frame to the frame of the ArUco marker had not taken place. As a result, the follower robot was going to the location that the explorer robot detected the ArUco marker, whereas it should be traveling much closer to the ArUco marker itself.

To troubleshoot this issue, our team began by printing the transformed ArUco marker location from the `listener()` function to the terminal to observe the coordinate points the follower robot was being sent to for each location. These values from the terminal were almost identical with the target location coordinates received from the parameter server. Upon consultation with the instruction staff, we determined that the reason these values were the same is because the transform was never executed because we had mistakenly inserted a `broadcast()` function with no argument into the detection algorithm. The inclusion of this extra `broadcast()` function during the detection of the ArUco marker effectively hardcoded the pose of the child_frame of the camera in the frame of the camera itself, instead of the desired frame of the marker. The inclusion of this broadcast function was obsolete because the broadcast function already existed in the callback function and was automatically executed by the callback function upon detection of the ArUco marker, so our team eliminated the extra broadcast function from our detection algorithm. Our team fundamentally misunderstood the purpose of the broadcast function and believed the calling of the empty broadcast function we included in our detection algorithm was necessary to broadcast the "map" frame to the listener function as its first argument, while the broadcast function in the callback function broadcasted the "marker_frame" to the listener function as its second argument. Upon removing the second broadcast function from our detection algorithm and the implementation of a tolerance specification for the follower robots marker navigation, our program worked as intended and the project was complete.

4 Project Contribution

Doug's Contributions:

- Wrote launch file for launching node.
- Wrote code for retrieving the target parameters from the parameter server.
- Wrote code for making the robots move to the targets.
- Implemented the `bot_controller` class to make the robot rotate.
- Worked on grabbing the `fiducial_id` from the `fiducial_transforms` topic message.
- Wrote the introduction to the report.
- Wrote the approach section of the report.
- Wrote the challenges section of the report.
- Wrote the pseudo codes

Rajan's Contributions:

- Wrote code for rotating the robot until it detected the marker
- Wrote code for storing the `fiducial_id`
- Wrote code for broadcasting the marker coordinates to the `map` frame
- Wrote code for transforming the marker coordinates with certain tolerance and storing them as goal coordinates for the `follower` robot

5 Resources

Urban search and rescue. (2021, October 18). Retrieved December 15, 2021, from https://en.wikipedia.org/wiki/Urban_search_and_rescue [1]

Kootbally, Z. (n.d.). Final_project/main.cpp at main · zeidk/final_project. Retrieved December 15, 2021, from https://github.com/zeidk/final_project/blob/main/src/main.cpp [2]

Bray, H. (2021, September 22). Boston Dynamics introduces an enhanced Spot robot - The Boston Globe. Retrieved December 15, 2021, from https://www.bostonglobe.com/2021/02/02/business/boston-dynamics-introduces-an-enhanced-spot-robot/ [3]

Wiki. (n.d.). Retrieved December 15, 2021, from http://wiki.ros.org/roscpp/Overview/Parameter Server#CA-a03bfcab2d7595a784e24298b326fdc4c76f3aee_5 [4]

How to use data from a callback function in another function edit. (n.d.). Retrieved December 15, 2021, from https://answers.ros.org/question/343938/how-to-use-data-from-a-callback-function-in-another-function/ [5]

Wiki. (n.d.). Retrieved from http://wiki.ros.org/tf/Tutorials/Writing-a-tf-listener-(C) [6]

How to read a value from a ros message? edit. (n.d.). Retrieved December 15, 2021, from https://answers.ros.org/question/158976/how-to-read-a-value-from-a-ros-message/ [7]

Techopedia. (2011, December 04). What is a Try/Catch Block? - Definition from Techopedia. Retrieved December 15, 2021, from https://www.techopedia.com/definition/25641/trycatch-block#:~:text=Try/Catch Block-,What Does Try/Catch Block Mean?,and handles try block exceptions. [8]

Writing a Simple Publisher and Subscriber (C). (2011, November 19). Retrieved December 15, 2021, from http://library.isr.ist.utl.pt/docs/roswiki/ROS(2f)Tutorials(2f)WritingPublisherSubscriber(28)c(2b2b29).html [9]

6 Course Feedback

Doug's Feedback: Overall, I think the course was a great introduction to robotic programming and programming in general. I think encouraging the students to use a Linux environment to develop their code is a great idea that actually made me learn a lot about operating systems and computer architecture too. I had no experience in ROS before this course, and while I think ROS appears to be too complex to truly grasp in 5 lectures, the class served as an effective primer on ROS and introduced the main concepts and applications very well too. I really liked that there were always plenty of resources given in the slides, which felt vital especially for a programming class because there are so many things in programming that come down to syntax/semantics and knowing where to find info online helps. The more challenging concepts of C++ such as pointers and OOP were illustrated very effectively and concisely compared to how they had been introduced to me before. I also think the last two projects were a great introduction to vital robotic programming concepts like DFS/BFS searches, robotic transformations and SLAM.

A criticism from me would be that sometimes the lectures could be a bit slow at times, which is probably a problem for most instructors in a 3 hour lecture to be fair. I also strongly recommend making the students do more of the ROS setup commands at home, so the lectures can be more focused around executing things in ROS, illustrating ROS concepts, and writing ROS code. It felt like sometimes in the ROS lectures were just us copying you by entering commands in the terminal and it was easy to get lost in the whole lecture if one step was missed or the terminal threw an error. One more suggestion is to assign a few smaller RW assignments, not meant to be especially rigorous, but to write simple scripts in C++ that just solve simple problems and make the students more comfortable working on independently writing code themselves instead of relying on group members. The quizzes were a good way to keep students honest too and ensure they were keeping up with the material. The level of difficulty was not as high as my other classes this semester, but I think it was appropriate for an introductory graduate level course. Thanks for a great semester!

Rajan's Feedback: This was a great introductory course for ROS and C++. Having never coded in C++ before, I lacked the proficiency my peers showed. Having said that, I was able to cover up a lot because of the organized and apt notes provided by the professor. There was a dramatic increase in difficulty after the first RWA. Which not only made us use our heads, but also helped us understand the programming language. The only stage where the course fell short was the response time of the TAs.