ENPM 809 Y

Introduction to Robot Modeling

Final Project Report

US&R Simplified

By

Mothish Raj

Shelvin Pauly

Jai Sharma

Table of Contents

Introduction

The final project is inspired by a real-world problem. Search and rescue robots are often used in disaster-struck zones to assist first-responders like paramedics and firefighters. These robots can carry out several important tasks like reconnaissance, mapping, and rescue operations.

For the project, we use a ROS setup consisting of a map, 2 turtlebots(model = waffle), and a set of 4 randomly positioned ArUco markers. We use one turtlebot (explorer) for mapping the environment, and the other (follower) to fetch the victims. The victim, in this case, means the ArUco marker. Hence, our program should be able to carry out the following tasks:

1. Find victims: the *explorer* must visit the location of each marker, detect the marker, and store the pose in the program. It should return to the start position once the 4 ArUco markers are visited and read.
2. Rescue victims: the *follower* must visit the ArUco markers using the marker ID. It should return to the start position once the 4 ArUco markers are visited and read.
3. Exit the program by calling ros::shutdown ( )

The explorer retrieves approximate locations of the markers from the Parameter Servers. Once the explorer reaches these areas in the map, it rotates by 1 degree to locate the exact location of the ArUco marker on the wall. The marker ID is stored along with the marker's pose. So, the follower simply needs to use this stored information to visit all 4 ArUco markers on the map.

Approach

The team has implemented object oriented programming to solve this problem. The code is divided into three files:

- bot_controller.h
- bot_controller.cpp
- main.cpp

We have created a class called Bot_Controller. We are fetching all methods and its functionalities from the bot_controller.cpp file. The three main methods that we have created are the broadcaster, the listener and

the sorter. You can also locate bot_controller.h file, header file, in the include folder. The file contains all attributes that we need for this project. In the main.cpp file, you will see an object of the class Bot_Controller. The object is called a controller. Our approach to incorporate OOPs to solve the project is as follows:

1. Initialize all required variables, arrays, vectors, classes, and methods.
2. Create methods called ‘broadcast’, ‘listen’, and ‘sort’ under Bot_Controller class
3. Fetch ArUco marker locations from ‘broadcast’ using an object called controllers.
4. Call the ‘listen’ method to transform the marker frame of reference to map’s frame of reference.
5. Ask explorer to go to each Target Locations and detect ArUco Markers
6. Use bubble sort to sort marker location arrays based on marker ID.
7. Ask follower to go to each ArUco Marker in the order of marker ID

The purpose and the architecture of each of these blocks will be explained in the coming sections.

Variables, arrays, classes and methods

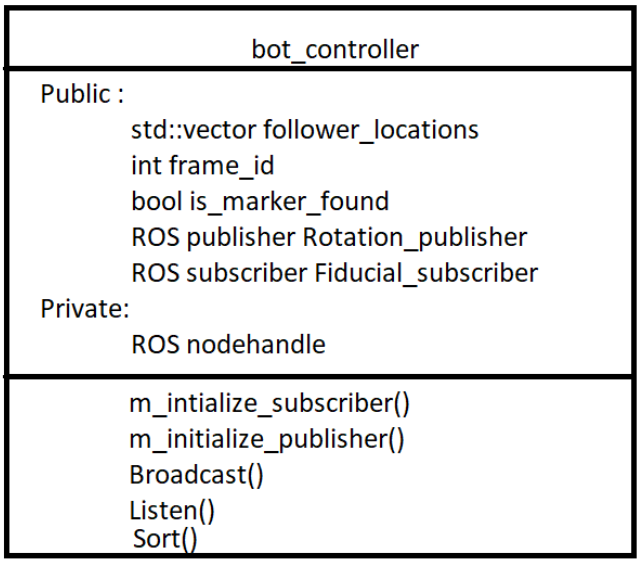
Bot_controller class:-

Public Members :

- Parameterized constructor Bot_controller : Calls methods m_intialize_publishers and m_intialize subscribers
- Data Members:-
    - Vector of array(double) follower_locations: To store the locations for the follower to visit [x,y and orientation w]
    - Integer frame_id : To store the frame ID for that location
    - Boolean is_marker_found : Flag to keep checking if the marker is found
    - Ros publisher Rotation_publisher : Publishes the angular rotation
    - Ros Subscriber Fiducial_subscriber : Subscribes to fiducial transforms
- Class Methods :-
    - Broadcast (Explained elaborately later in the document)
    - Listen (Explained elaborately later in the document)
    - m_initialize_subscribers : Publishes the cmd_vel at 1000 Hz for bot rotation
    - m_initialize_publishers : Subscribe to fiducial_transform cmd_vel at 10000 Hz

Private Members :-

- Ros Nodehandle



Main C++ file :-

- Movebaseclient : Typedef for Client service. Used to spin threads for the explorer and follower

- locations : vector of type xmlrpcvalue. Here, we store the 4 locations that we get from the .yaml file. Basically, location vector for the explorer.

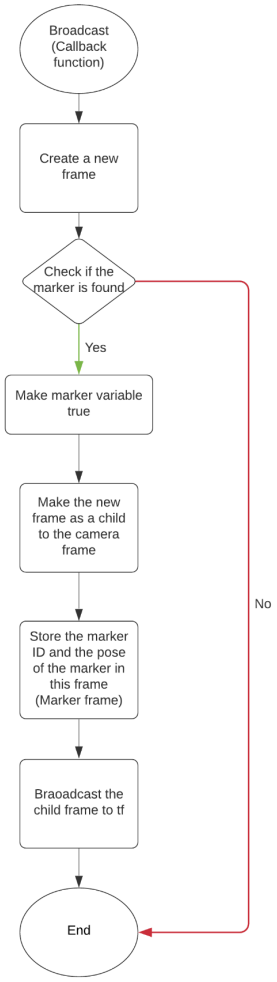
- nh : ROS Nodehandle
- controller : To create a controller_bot object to use all its methods
- explorer_goal : movebase for explorer
- follower_goal : movebase for follower
- is_explorer_work : Flag to check if work is completed by explorer
- is_follower_work : Flag to check if work is completed by follower

Broadcaster

The broadcaster method is essentially a callback function. We initialize the method as void Bot_Controller::broadcast(const fiducial_msgs::FiducialTransformArray::ConstPtr& msg). This function is called in the *int main* to generate a new frame each time a marker is detected.

To get the transform between two frames, the two frames must exist and be published on /tf Topic. Hence, we need to build a frame at the location of the ArUco marker and publish this new frame as a new Topic. It is a child frame to the explorer_tf/camera_rgb_optical_frame and we call it ‘my_frame.’ The translation and rotation information passed through ‘msg’ ensures the new frame is built at the position of the marker. TransformBroadcaster imports the sendTransform class, which we use to broadcast the transform on /Tf topic.

In the int main function, the command to call the broadcaster is placed such that the child frame is only created when the marker is detected.



Flowchart for broadcast

Listener

We initialize the listener method as void Bot_Controller:: listen(tf2_ros::Buffer& tfBuffer). The listener function has two primary goals:

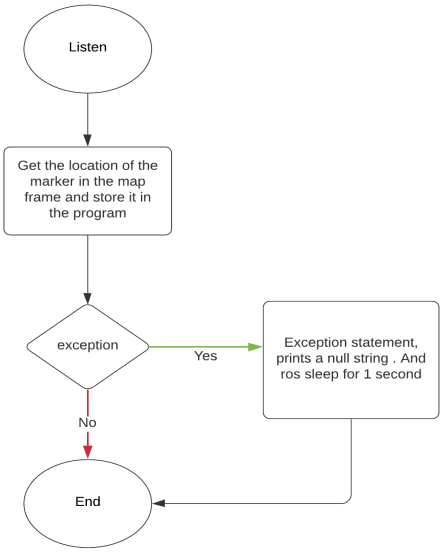
1. convert the ArUco marker's pose from the explorer's frame of reference to the map's frame of reference
2. Store the marker's pose in the array named follower_locations

The function is written using a **try** and **catch** method. It allows us to see what the errors are as soon as they come up. This prevents the code from blowing up or entering a computationally demanding process.

We use a function called lookup.transform to build a relation between the map frame and my_frame. This output is stored using variables trans_x, trans_y, and trans_z. These variables are eventually used to build an array called fiducial_location.

The follower reads information about the marker's location using an array called follower_location. This array is built using a push_back function. Each time a marker is detected, its position, in the map's reference frame, gets stored in fiducial_locations array, which is eventually pushed into the follower location array. The ArUco ID information is also passed using a variable c, although the marker information is stored in the order in which it is detected.

If any errors are encountered in the 'listen' function, the catch component outputs a warning message in form of a null string. After pausing for a second, the code leaves the 'listen' function.



Flowchart for listen

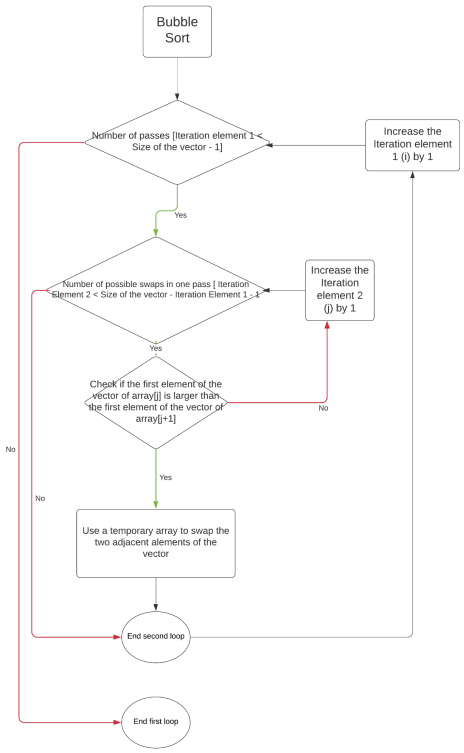
Sorting Function

The marker locations are stored in the array follower_locations. The follower reads data in a specific order, column by column. Hence, to ensure that the followers go to all the markers in the right order, we have to sort the data inside the array. This method is initialized as void Bot_Controller:: sort()

We use the technique of bubble sorting to do this. The first element of the array contains the ArUco marker IDs. Hence, the sorting algorithm only uses the first element of each column to swap data around. In each pass, adjacent values are compared and the higher is pushed ahead in the array.

The resulting follower_locations array would be expected to look like the following:

Array Index	0	1	2	3
Stored ID Number	0	1	2	3



Flowchart for bubbelsort

Int main function

The initial part of the int main function contains several lines of initializations. A few bool variables are used, like explorer_goal_sent and is_explorer_work. They are mainly used as conditions to stay inside ‘if’ loops, ensuring that that specific code runs till the required task is complete. Four location variables are defined using XmlRpc::XmlRpcValue data type. They eventually store the Target Locations when the data is retrieved using nh.getParam function. The arrays are then compiled into a single vector called ‘locations’ using the push_back function.

The move_base action servers, associated with explorer_client and follower_client, need some buffer time to come up and get started. This buffer is provided by using a while loop that adds 5 seconds each time explorer_client and follower_client cannot be reached.

In the next block of code inside the int main function, a while loop is used to launch ROS. This loop contains all the implementation code required by the explorer and the follower to complete

their tasks. First, the explorer goal is built using id, x, y, w information. The x and y coordinate values are in xml format (string), so they are passed as double using 'static_cast' method. The explorer_goal is next sent to explorer_client and implemented move base. A while loop is used to ensure that the explorer keeps moving as long as the Target Location is not found. If the location is reached and if the marker is not found, the rotation_publisher command is published using the object ‘controller.’ This asks the explorer to rotate in place and look for the ArUco Marker. While doing this, the fiducial_transforms are also being subscribed simultaneously. ros::spinOnce is used right after to execute the broadcasts. When the marker is found, msg.angular.z is set to 0, stopping the explorer from rotating any more. Bool variables controller.is_marker_found and is_explorer_work are redefined to leave the if loops.

The sorting algorithm is run by calling the method through controller.sort() command . Next code relating to the follower is written. For each follower_locations index, the pose of the ArUco marker is retrieved and stored as an array. This array is sent to follower_client that implements the move_base package. Similar to the explorer, the follower goes to each of the target locations and directly detects the ArUco markers. Thanks to the sorting function, the follower visits all markers in the order of its ID numbers. Is_follower_work is redefined to be true, and consequently, the if loop breaks. This concludes the code we wrote to solve the problem addressed in this project. To exit the program ros::shutdown ( ) was called.
This program solves the US&R problem in a simplified situation
In the main function

Firstly, get all the locations from the parameter server and check the values
Secondly, Push all the locations into a vector

1.while loop(ROS is running)

Check if explorer task is not finished

A. Provide the location to movebase(explorer) one by one using for loop

2. While loop(Till the location is reached by explorer)

Check if explorer reached the goal

Check if the robot is not in the initial position

3. while loop(the Marker is not found)

Rotate the explorer and keep scanning

Check if the marker is detected

Stop the explorer

Store the marker location

Marker flag becomes false

break

Else

Make the flag for explorer work true

Sort the locations according to the frame ID

Check if follower work is pending

Provide the location to movebase(follower) one by one using for loop

5. while loop(till the marker location is reached)

Check if marker location is reached

Turn the flag true and continue the task

Challenges

Problem 1

We used a broadcaster as our fiducial subscriber callback and when data is available we used the 'listen' function but we kept getting the following error:
Error Message: *"Lookup would require extrapolation at time 882.563000000, but only time 882.585000000 is in the buffer, when looking up transform from frame [my_frame] to frame [map]".*

Problem 2

At times the robot is not detecting the data provided in the first spin and we would get this error:

Error Message: *"my_frame" passed to lookupTransform argument source_frame does not exist"*

We realized that the publisher and subscriber are not working at the same rate. Sometimes the listener was subscribing data from an older timestamp, not the current timestamp, and this was causing the issue. To resolve this, we added the following line: ros::Duration(1.0).sleep() to the int main function. We command the ros to sleep for a duration of 1 second and this allows the publisher to 'catch up' and the subscriber can read the most updated value. Adding the line was able to solve both the problems.

Project Contribution

Writing the program and the report was fairly divided between the three team members.

Team Member	Contributions
Mothish Raj	Publisher, subscriber, OOPS implementation;Debugging
Shelvin Pauly	Listen,Sort;Debugging;Doxygen;Report
Jai Sharma	Code without OOPs implementation, getParam server. Report.

Resource

- *roscppOverviewParameter Server*. ros.org. (n.d.). Retrieved December 10, 2021, from http://wiki.ros.org/roscpp/Overview/Parameter%20Server#CA-a03bfcab2d7595a784e24298b326fdc4c76f3aee_5
- *Bubble sort*. GeeksforGeeks. (2021, September 24). Retrieved December 11, 2021, from https://www.geeksforgeeks.org/bubble-sort/
- *Wiki*. ros.org. (n.d.). Retrieved December 15, 2021, from http://wiki.ros.org/roscpp/Overview/Callbacks%20and%20Spinning

Course Feedback

- The course was well-structured and got a lot to learn from the course structure. Especially the project based approach gave hands on experience.
- It will be helpful for students if a few more ros based projects are included to learn more.