Zeid Kootbally
University of Maryland
College Park, MD
zeidk@umd.edu

Final Project: US&R Simplified (v1.1)

ENPM809Y : Fall 2021

Due **Wednesday, December 15, 2021, 4 pm (Section I and Section II)**

Contents

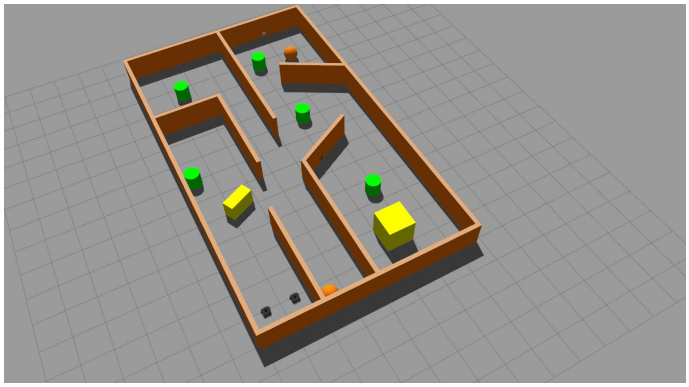

Figure 1: World used in the final project.

1 Prerequisites

- You need to be part of a group of 3 students (or less) for this assignment.
- Execute `[install.bash]` to install all necessary packages.
- Get the `final_project` package from github:

```
> cd ~/catkin_ws/src
> git clone https://github.com/zeidk/final_project.git
> catkin build final_project
```

(located here).

2 Start Application

To run the final project, you will need to run only two commands:

1. `> roslaunch final_project multiple_robots.launch`
   - This command will start Gazebo (see Figure 1) and RViz, will set parameters on the Parameter Server, spawn robots, and do topic remappings.
2. `> roslaunch/rosrun final_project launch_file/executable`
   - This command will run your node when you are done writing it.

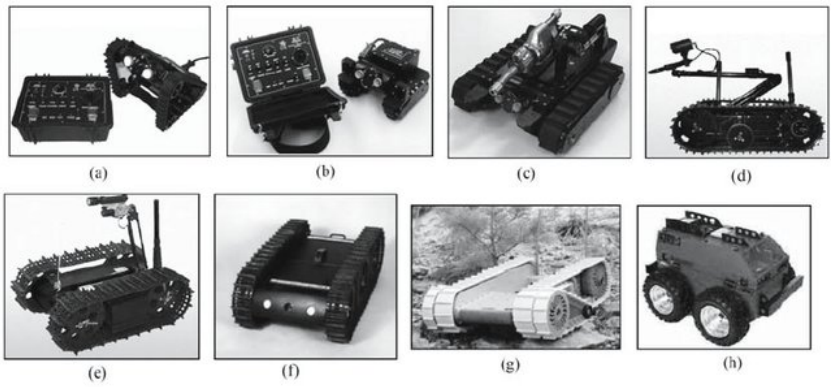

Figure 2: Search and rescue robots used in World Trade Center's rescue operation: (a) Micro VGTV; (b) Micro Traces; (c) Mini Traces; (d) Talon; (e) SOLEM; (f) Urbot; (g) Packbot; (h) ATRV

3 Objectives

- The final project is inspired by the the challenge of autonomous robotics for Urban Search and Rescue (US&R). In US&R after a disaster occurs, a robot is used to explore an unknown environment, such as a partially collapsed building, and locates trapped or unconscious human victims of the disaster (see examples of robots in Figure 2). The robot builds a map of the collapsed building as it explores, and places markers in the map of where the victims are located. This map is then given to trained First Responders who use it to go into the building and rescue the victims.
- The map has already been provided to you in the `maps` folder of the catkin package: `[final_world.pgm]` and `[final_world.yaml]`
- Instead of simulated victims, we use ArUco markers (squared fiducial markers).
- We will use a turtlebot (called ***explorer***) to use a map of the building to find victims (markers). We will then use another turlebot (called ***follower***) to fetch the victims.
- Write a ROS program which can:

  1. Find victims: Task ***explorer*** to go through four different locations. At each location, make the robot rotate until it detects an ArUco marker and store the pose of this marker in your program. Once all four markers are found, task the robot to go back to its start position.
  2. Rescue victims: To keep it simple, instead of rescuing victims, task ***follower*** to visit each ArUco marker (found by ***explorer***), based on the marker ID: 1) Visit the marker with ID 0, 2) visit the marker with ID 1, and so on. Once all markers have been visited, return ***follower*** to its start position.
  3. Exit you program by shutting down your Node with `ros::shutdown();`

4 ArUco Markers

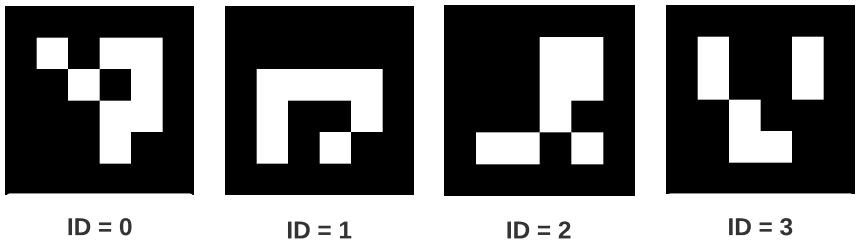

Figure 3: ArUco markers used in the final project.

ArUco markers are binary square fiducial markers that can be used for camera pose estimation. Their main benefit is that their detection is robust, fast and simple. There are a total of four ArUco markers used in this project, each one has a unique design and a unique ID. These markers were generated online (see https://chev.me/arucogen/). The markers are depicted in Figure 3.

Only *explorer* is capable of detecting those markers. In `[multiple_robots.launch]`, the Node aruco_detect is started. The markers are detected in the camera frame of *explorer* through Topic remappings:

```
<remap from="/camera/compressed" to="/explorer/camera/rgb/image_raw/compressed" />
<remap from="/camera_info" to="/explorer/camera/rgb/camera_info" />
```

5 *explorer*

The different tasks to be performed by ***explorer*** are listed below and also described in each subsequent subsections:

1. Read target locations from the Parameter Server.
2. Move to each target in a specific order.
    - Once a target is reached, look for the ArUco marker at this location.
3. Return to start position.

5.1 Read Target Locations from the Parameter Server

explorer has to go to four different locations to find ArUco markers. These locations are not locations of those markers but spots in the environment where robot will be able to find the markers. These locations are provided to you and are stored on the parameter server:

- `/aruco_lookup_locations/target_1`
- `/aruco_lookup_locations/target_2`
- `/aruco_lookup_locations/target_3`
- `/aruco_lookup_locations/target_4`

Each one of these parameters is an array with two numbers:

- `> rosparam get /aruco_lookup_locations/target_1` → `[-1.752882, 3.246192]`
- x and y coordinates for `target_1`

Warning ***explorer*** has to reach the different targets in a specific order: `target_1` first, then `target_2`, and so on.

- To-do: Retrieve parameters and store them in your program.
    - In class we used the `getParam()` function to retrieve simple parameter types (string, int, etc). To retrieve a parameter array you will need to proceed differently with the use of `XmlRpc::XmlRpcValue`. Refer to the code at the bottom of this page to learn how to read parameter arrays.

5.2 Move to a Target

To move to a target, we will use the package `move_base` as seen in class. You need to make sure you are sending the goal in the /map frame.

5.3 Detect AruCo Marker

Once a goal is reached with `move_base`, rotate ***explorer*** to find the ArUco marker. To do so, make ***explorer*** rotate very slowly (`linear.x=0.0` and `angular.z=0.1`) by publishing velocities to the Topic explorer/cmd_vel. While the robot is moving, detect the marker.

ArUco markers are detected in the ***explorer***'s camera frame: explorer_tf/camera_rgb_optical_frame

The Node aruco_detect publishes on the Topic /fiducial_transforms. Below is the Topic outputs when a marker is detected.

```
> rostopic echo /fiducial_transforms
```

```
header:
  seq: 1503
  stamp:
    secs: 461
    nsecs: 847000000
  frame_id: "explorer_tf/camera_rgb_optical_frame"
image_seq: 1506
transforms:
  -
    fiducial_id: 2
    transform:
      translation:
        x: 4.72388611875
        y: -0.154209633128
        z: 12.3464879525
      rotation:
        x: -0.922945334667
        y: 0.0186403074191
        z: 0.33582110924
        w: 0.187212795356
    image_error: 0.0618348410353
    object_error: 0.254729020252
    fiducial_area: 202.842063334
---
```

Later in this project you will need to task ***follower*** to visit markers detected by ***explorer***. Since we are sending goals to robots in the /map frame, we need to transform markers' poses from explorer_tf/camera_rgb_optical_frame to /map frame.

Approach

1. Use the output from the Topic /fiducial_transforms to publish a new frame on the Topic /tf. This new frame will be a child of ***explorer***'s camera frame. We will use a broadcaster to accomplish this.
2. With this new frame we can now transform from ***explorer***'s camera frame to /map frame (similar to the can detection scenario in the our previous lecture). We will use a listener to accomplish this.

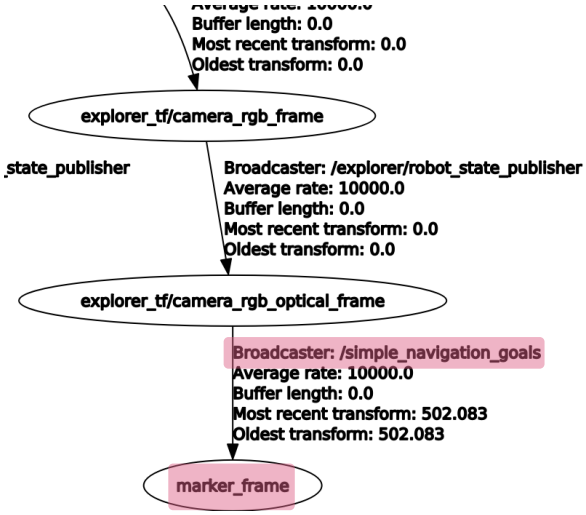

Figure 4: /marker_frame broadcasted on /tf Topic.

5.3.1 Broadcaster

Create a child frame (e.g., marker_frame) in explorer_tf/camera_rgb_optical_frame. This is done through a broadcaster. This new frame will be published on the Topic /tf. It will also appear in the TF tree (see Figure 4) and in RViz.

- To-do:
  - Subscribe to the Topic /fiducial_transforms.
  - In the callback function, broadcast the frame /marker_frame as a child of explorer_tf/camera_rgb_optical_frame. It is very important that you only broadcast this frame when the marker is detected. Below is the output of the Topic /fiducial_transforms when NO marker is detected.

```
header:
  seq: 925
  stamp:
    secs: 441
    nsecs: 766000000
  frame_id: "explorer_tf/camera_rgb_optical_frame"
image_seq: 925
transforms: []
---
```

  - Below is an excerpt for the callback function to help you out.

```
void fiducial_callback(const fiducial_msgs::FiducialTransformArray::ConstPtr& msg)
{
  if (!msg->transforms.empty()) {//check marker is detected
    //broadcaster object
    static tf2_ros::TransformBroadcaster br;
    geometry_msgs::TransformStamped transformStamped;

    //broadcast the new frame to /tf Topic
    transformStamped.header.stamp = ros::Time::now();
    transformStamped.header.frame_id = "explorer_tf/camera_rgb_optical_frame";
    transformStamped.child_frame_id = "marker_frame";
    transformStamped.transform.translation.x =
     ↪  msg->transforms[0].transform.translation.x;
    /*write the remaining code here*/
    br.sendTransform(transformStamped);
  }
}
```

5.3.2 Listener

The next step is to convert the ArUco marker's pose in the /map frame so ***follower*** can reach it. For this, you will use a /tf listener. A listener allows ROS to listen to the Topic /tf for a few seconds and compute the transform between two given frames.

Important You really need to understand that to get the transform between two frames, the two frames must be published on /tf Topic. This is why we used a broadcaster earlier to publish marker_frame on /tf Topic. Now that we have two frames, we can the transform from one frame to the other. With a listener we can ask about the pose of marker_frame in map frame.

Below is a code snippet showing how to get the pose of the frame marker_frame in the map frame.

```
//listen to /tf Topic and get the broadcasted frame in the /map frame
try {
    transformStamped = tfBuffer.lookupTransform("map", "marker_frame",
     ↪  ros::Time(0));
    ROS_INFO_STREAM("marker in /map frame: ["
        << transformStamped.transform.translation.x << ","
        << transformStamped.transform.translation.y << ","
        << transformStamped.transform.translation.z << "]"
    );
}
catch (tf2::TransformException& ex) {
    ROS_WARN("%s", ex.what());
    ros::Duration(1.0).sleep();
}
```

In the `try{..}` statement, you will also store each marker location (x and y) in your program so it

can be used later by ***follower***. I suggest you create an `std::array` of size 4 and store each marker, based on its ID, inside this array. Marker with ID 0 will be stored at index 0 in the array, marker with ID 1 will be stored at index 1 in the array, and so on. The ID of the detected marker can be retrieved from the field `fiducial_id` from data published to the Topic /fiducial_transforms.

5.3.3 Broadcaster and Listener Summary

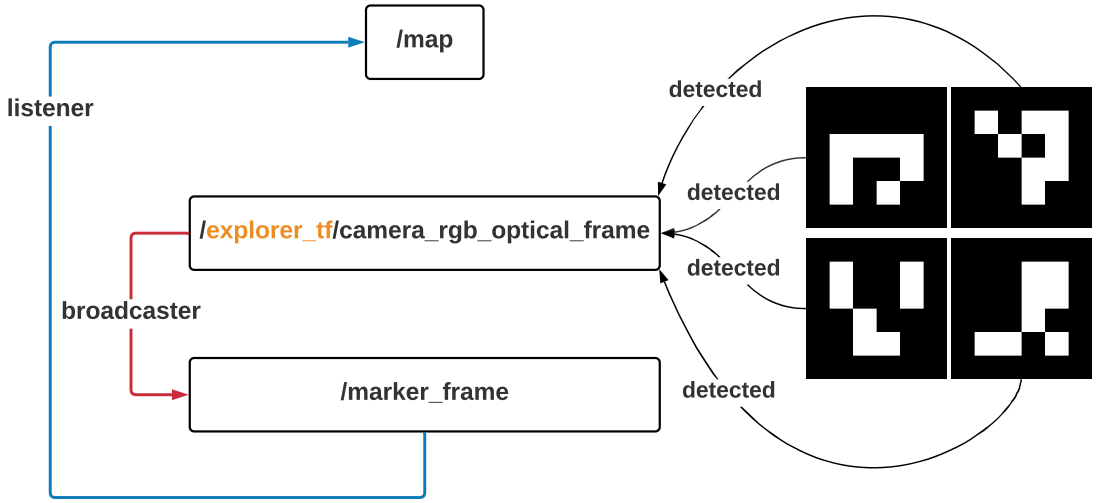

Figure 5: Broadcaster and listener.

5.4 Return to Start Position

Once all four markers have been detected, use `move_base` to task ***explorer*** to go back to its start position $(-4, 2.5)$ -- you can hardcode this goal. Once it reaches the goal, the role of ***explorer*** is done and it is now ***follower***'s turn to move.

6 *follower*

- Using the `std::array` from Section 5.3.2, task ***follower*** to go to each location in the array. The locations stored in this `std::array` are locations of ArUco markers in the /map frame and the markers are stuck to walls. The costmaps and planners used by `move_base` will not allow ***follower*** to reach those locations. I suggest you task ***follower*** to reach those markers within a tolerance ($0.4m$ seems to be good).
- If you properly stored marker locations in your `std::array`, the first marker has the ID 0, the second marker has the ID 1, etc. If everything is good, ***follower*** will visit the markers in this order: ID=0, ID=1, ID=2, and ID=3.

- Once ***follower*** has reached all markers, make it return to its start position $(-4, 3.5)$ -- you can hardcode this goal -- and terminate your program with `ros::shutdown();`

7 Deliverables

Your group's project submission consists of three deliverables:

1. ROS package written in C++ containing all code, launch files, and other content required to run your group's software. Use the package `final_project` that I provided and add your content in this package.
   Important You must use OOP for this project. You can (should) reuse part of the code provided in Lecture 13 for the `bot_controller` package. Modify the C++ class provided (add/remove attributes and methods). A few other things to do:

   - Follow the conventions and guidelines.
   - Get rid of unused code. Do not leave big chunks of commented code in your program.
   - HTML Doxygen documentation is needed this time. Generate HTML Doxygen documentation in the `doc` folder.
   - Also in the `doc` folder, create an `[instructions.txt]` file where you will include instructions on how to run your code.

2. Live demonstration of your software. Each group will be given 10 minutes to complete the task. You are allowed to restart only once if your program crashes during the first run. Remember that only the two commands described in Section 2 will be used to run the live demonstration.
   Important During the live demonstration, the Gazebo building stays the same but the location of ArUco markers will change. This is to ensure you are not hardcoding information that should be retrieved dynamically.
3. A report describing how your group solved the simplified US&R task, the challenges you faced, your program flow (sequence diagrams), and the contribution of each one in the group. The report should not be more than 15 pages (single column). If needed, a template will be provided for the report.

8 Getting Help

You are given 2 weeks for this assignment. Unless you are a ROS guru, you will not be able to complete this assignment in less than two weeks (I may be exaggerating here). You will need some help, especially when it comes to the listener and broadcaster. TAs are here to help (they are the ROS gurus). I am here to answer questions related to confusions about assignments and eventually technical questions.

9 Grading Rubric

coming soon...