University of South Florida

ORB_SLAM2 integration with Robobulls Robot

Final Report

Justin Rodney

CIS4900 Independent Study in Robotics

Professor. Alfredo Weitzenfeld

Summer 2020

Index

I. Preface

The objective of this study was to configure ***ORB_SLAM2,*** a simultaneous localization and mapping package, to work with the ***RoboBulls differential drive robot*** via ***ROS*** in order to use the camera for localization. The map and pose published by ***ORB_SLAM2*** could be used for autonomous exploration of an unknown environment to generate maps, as well as autonomously navigate to a goal in a known map. There are several pre-existing ***ROS*** packages that provide these autonomous navigation functions, such as Exploration_lite[1] for autonomous exploration of an unknown environment and the navigation stack[2] which can be used for path planning. However, after conducting research and testing these packages, it is clear that they are not compatible with ***ORB_SLAM2*** because of the lack of ***odometry*** and ***tf***(transform) information being published. Additionally, ***ORB_SLAM2*** is the only existing ***SLAM*** package in ***ROS*** that is able to use only a monocular camera for its input, making it the only compatible SLAM package for the ***RoboBulls robot*** equipped with a camera, IR distance sensors, servos, and encoders as of 2020.

With the current configuration of the robot, ***ORB_SLAM2*** can be used for localization in a known map as demonstrated in the bug algorithm demo program. Map generation with ***ORB_SLAM2*** was tested in four different rooms, with only one of the rooms being able to produce a map accurate enough for localization throughout the entirety of the map. As found in a previous study by Yu-Ting Chung, that is uploaded to the ***biorobaw*** github, points cannot be

[1] http://wiki.ros.org/explore_lite
[2] http://wiki.ros.org/navigation

generated on objects in an image frame if they are too dark or too bright[3]. This leads to difficulties mapping the environment, as a white wall or black footstool would not be captured in the map and this would lead to the robot being lost when facing these objects, resulting in an unusable map. The robot must avoid obstacles using its ***distance sensors,*** as the performance of localization and quality of the maps that are generated are currently only accurate enough to provide the robot with its current location and does not provide information regarding its distance to obstacles for path planning. In order to utilize the more advanced ***SLAM*** packages, the robot would need to be updated as they require **IMUs**, **stereo cameras**, or **laser scanners**. Upgrading the robot would provide a more accurate and reliable mapping and localization, as well as the ability to use features like **RGBD** processing or the ability to use the pre-existing ***ROS*** packages used for autonomous navigation via path planning in a known map(***navigation stack***) and exploration in an unknown environment while generating a map(***exploration_lite***).

[3] https://github.com/biorobaw/SLAM-S2018/blob/master/docs/Reports/final%20report.pdf

II. Introduction

ORB_SLAM2 is a simultaneous localization and mapping package that uses a monocular, stereo, or RGBD(RGB with depth) camera for its input. The ***ROS*** implementation of the program outputs Pose, PointCloud2, and a debug image to ***ROS***. The Pose contains the x, y, and z coordinates as well as quaternion information which can be used to calculate the yaw that the camera, or for the use of the project the robot, is located in the environment. Using ORB_SLAM2 with Stereo or RGBD cameras provide a map of "true scale", however, using a mono camera does not provide a map that is true to scale[4].

Rviz is a visualization tool for displaying commonly used ROS messages such as Pose, PointCloud2, map, and Image. This can be used for displaying the Pose of robot, the PointCloud2 points, and debug Image that is sent from ***ORB_SLAM2***

Telop allows you to manually control the robot by sending ***Twist*** messages over ***ROS***. This can be used for controlling the robot when generating maps for ***ORB_SLAM2***.

pi3_slam_nav.py is a program that uses Pose information sent from ORB_SLAM2, allows input coordinates to be input as a goal, and moves the robot to the goal using a bug algorithm. The program uses the x, y, and yaw information as well as the x and y coordinates of the goal to calculate the required yaw for the robot to be facing the goal.

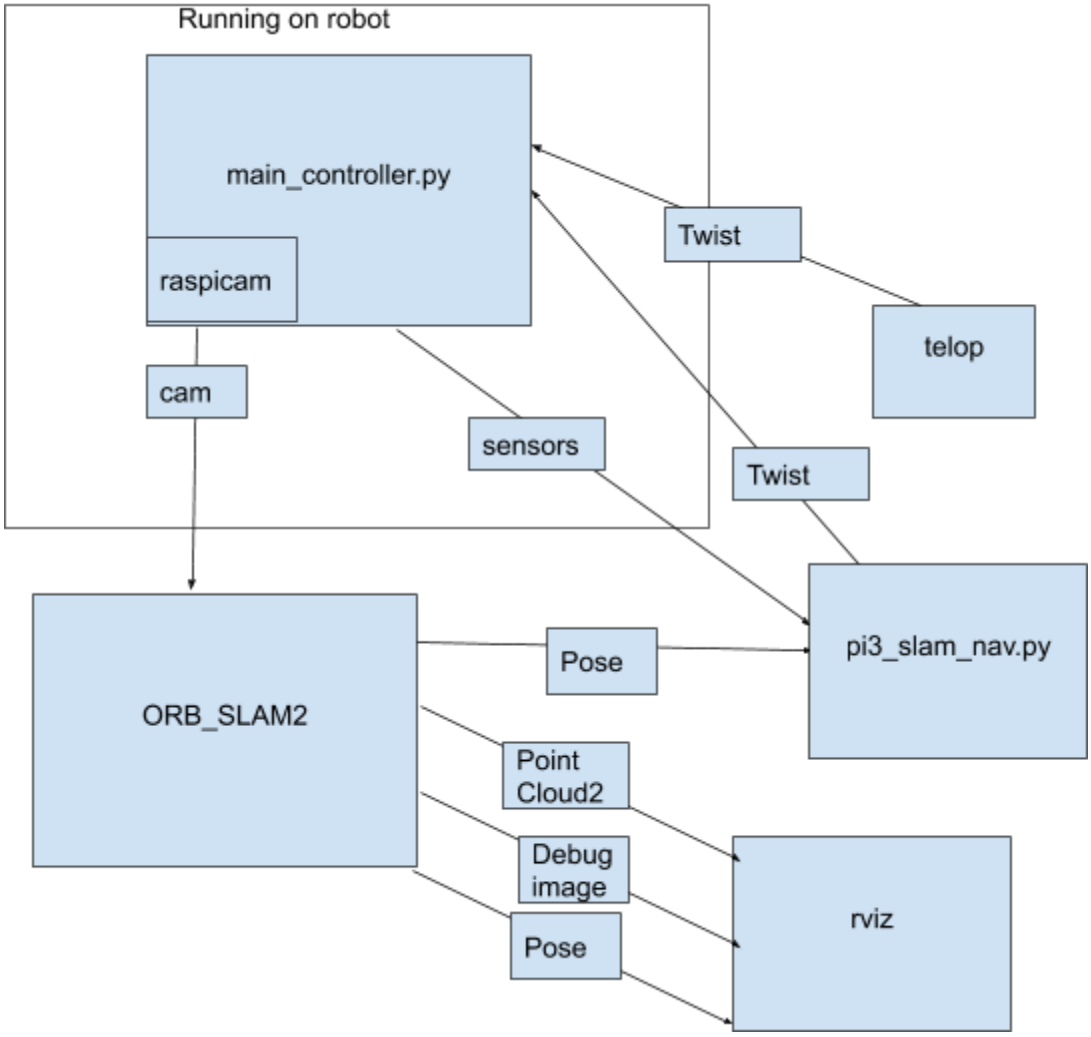
Desired angle = atan2(goalY- currentY, goalX-currentX)

[4] http://wiki.ros.org/orb_slam2_ros

Distance = (goalX-currentX)^2 + (goalY-currentY)^2

The program will then adjust the robot's angle until it is facing the goal, travel towards the goal and use wall following to avoid any obstacles along the way.

Figure 1. Diagram of the software system



III. Methodology

A. Requirements & Installation

The utilization of ***ORB_SLAM2*** with the robot requires the corresponding github branches to be installed and set-up on both the robot and your computer. If you have not done so, you will need to follow the installation documentation, see footnote below.[5]

This package includes the libraries required for controlling the robot by sending speed messages, reading from its distance sensors for obstacle avoidance, receiving encoder information to retrieve the distance traveled by the robot, and obtaining the video feed from the camera. The launch files for starting ***ORB_SLAM2*** as well as the bug algorithm program that utilizes the pose information are also included.

Additionally you will need to install:

ORB_SLAM2(ROS):https://github.com/appliedAI-Initiative/orb_slam_2_ros (you only need to install, you do not need to configure anything)

ORB_SLAM2 is a simultaneous localization and Mapping algorithm that uses input from monocular, stereo, and RGBD cameras, and publishes Pose, PointCloud2, and debug Image topics[6].

[7]***Rviz:http://wiki.ros.org/rviz***

Rviz is a visualization tool for displaying commonly used ROS messages such as Pose, PointCloud2, map, and Image.

Recommended:

[8]***Telop***(**highly** recommended): http://wiki.ros.org/teleop_twist_keyboard

Telop allows you to manually control the robot by sending ***Twist*** messages over ***ROS***.[9]

---

[5]https://github.com/biorobaw/pi3_robot_2019/blob/python_client/docs/Control%20of%20Mobile%20Robots%20Installation%20Doc.pdf
[6] http://wiki.ros.org/orb_slam2_ros
[7] Included in the full installation of ROS-Kinetic
[8] Included in the full installation of ROS Kinetic
[9] http://wiki.ros.org/geometry_msgs

B. Running the Programs

Step 1. Running camera node

An initial set up of ***ROS*** to allow communication between your computer and the robot as detailed in the documentation for installing the ***robot_client*** and ***ros_controller*** is necessary as well. In addition to running the primary launch file on the robot, you will want to run an instance of ***raspicam***(included in ***ros_kinetic***) in a separate terminal. In my testing, I had run into issues where the camera had to be restarted in order for ***ORB_SLAM2*** to receive the images for processing. The settings I had most success with were:

rosrun raspicam_node raspicam_node _width:=640 _height:=480 _framerate:=30 _quality:=100 _ISO:=200 _shutter_speed:=100000 _saturation:=50 _awb_mode:=horizon __name:=cam _enable_raw:=true _hFlip:=true _vFlip:=true

These are the parameters that I had found and modified from Pablo Scleidorovich's ***main_controller.py*** program in the ***ros_controller*** branch on the ***biorobaw*** github[10].

The camera appears to be mounted upside-down on my robot, so I had to flip the image. However, if yours is properly oriented you may set the ***hFlip*** and ***vFlip*** parameters to false.

Additionally, the **640x480** resolution worked best for me, however, you may wish to experiment with **320x240** or **1280x960**. Higher resolution images seem to work better with ***ORB_SLAM2*** for generating the points in a map, but may slow down the rate of camera frames being received through the network, hindering the robot's ability to maintain localization while moving, as the time between two image frames are too significant. Whereas, lower resolutions have smoother frame rate, but seems to take more time and generate less accurate maps.

[10] https://github.com/biorobaw/pi3_robot_2019/blob/ros_controller/scripts/main_controller.py

Step 2. Launching ORB_SLAM2

In the ***robot_client*/*launch*** folder, there exists two launch files:

pi3_orb_slam2.launch

pi3_orb_slam2_localization_only.launch

These launch files will launch ***ORB_SLAM2*** with the settings configured to work with the robot. You would run it with the command:

roslaunch robot_client pi3_orb_slam2.launch

Recommended: The camera should be calibrated as detailed here.[11]

After running the calibration program you can update the camera calibration parameters in the launch files:

```
<!-- Camera calibration parameters -->

    <!--If the node should wait for a camera_info topic to take the camera calibration data-->
    <param name="load_calibration_from_cam" type="bool" value="false" />
    <!-- Camera calibration and distortion parameters (OpenCV) -->
   <param name="camera_fx" type="double" value="583.01740710559432" />
   <param name="camera_fy" type="double" value="583.01740710559432" />
   <param name="camera_cx" type="double" value="320" />
   <param name="camera_cy" type="double" value="240" />
    <!-- Camera calibration and distortion parameters (OpenCV) -->
   <param name="camera_k1" type="double" value="0.20483552665926258" />
   <param name="camera_k2" type="double" value="-0.42414204428032326" />
   <param name="camera_p1" type="double" value="0.0" />
   <param name="camera_p2" type="double" value="0.0" />
   <param name="camera_k3" type="double" value=".0095177708663656737" />
```

If the RoboBulls robot you have is equipped with the same camera as used in this paper, you may not need to update the parameters. However, if you are having problems with ***ORB_SLAM2*** you may need to update these parameters.

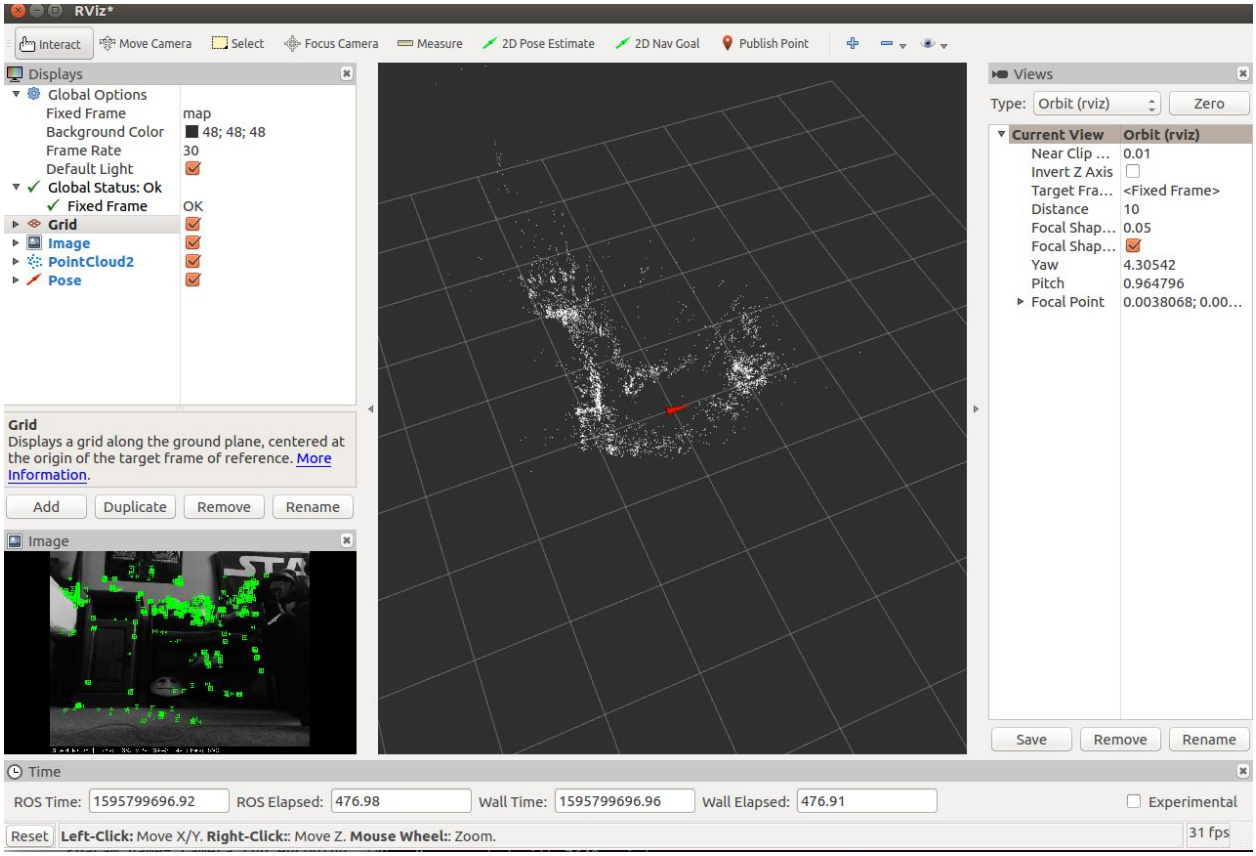
[11] https://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html

Step 3. Running Rviz

In order to view the map generated by ***ORB_SLAM2*** and the robot's position in the map, you will need to run rviz:

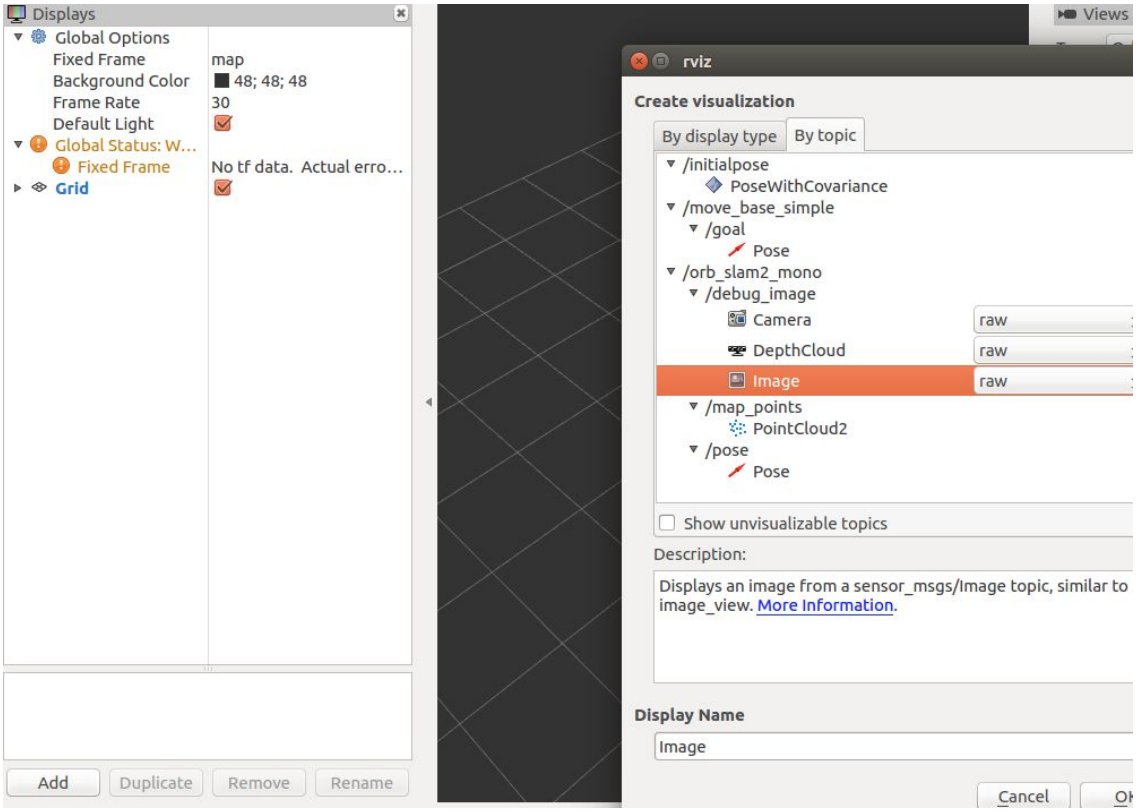
rosrun rviz rviz

Figure 2. Rviz fully configured and showing a map of a bedroom



You will want to add the ***debug_image***(Image), ***map_points***(PointCloud2), and ***pose***(Pose) topics that are located in /orb_slam2_mono.

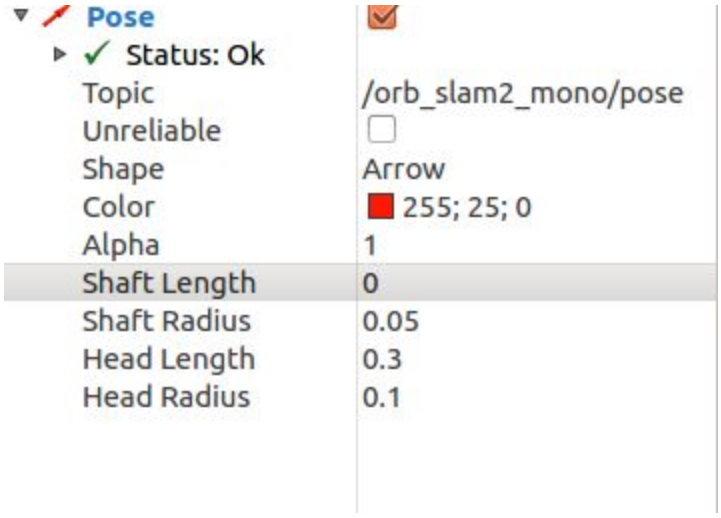
Figure 3. Setting Rviz parameters



The ***debug_image*** topic allows you to view ***ORB_SLAM2*** generating points from the image frames as well as whether the robot is currently localized or lost.

The ***map_points*** topic shows you the map points in a three dimensional space, with ***pose*** showing the location of the robot. ***Rviz*** generates a model to represent the robot using the robot's pose in the environment. I recommend that you set the ***Shaft Length*** of the pose display to 0 for a more accurate visual representation of the robot in the environment, as the robot does not have a significant body length that needs to be represented by the model.

Figure 4. Rviz pose parameters



You can move the camera around by clicking and scrolling on the 3D environment, while holding shift will allow you to change the point the camera is centered around. You may save these settings in Rviz so that you do not need to reapply the changes upon start up.

Step 4. Map generation using telop

Now that you have ***ORB_SLAM2*** generating a map from the video, the next step is generating the map. In order to do this, you will want to manually control the robot using ***telop***.

rosrun teleop_twist_keyboard teleop_twist_keyboard.py cmd_vel:=/pi3_robot_2019/r1/speed_vw

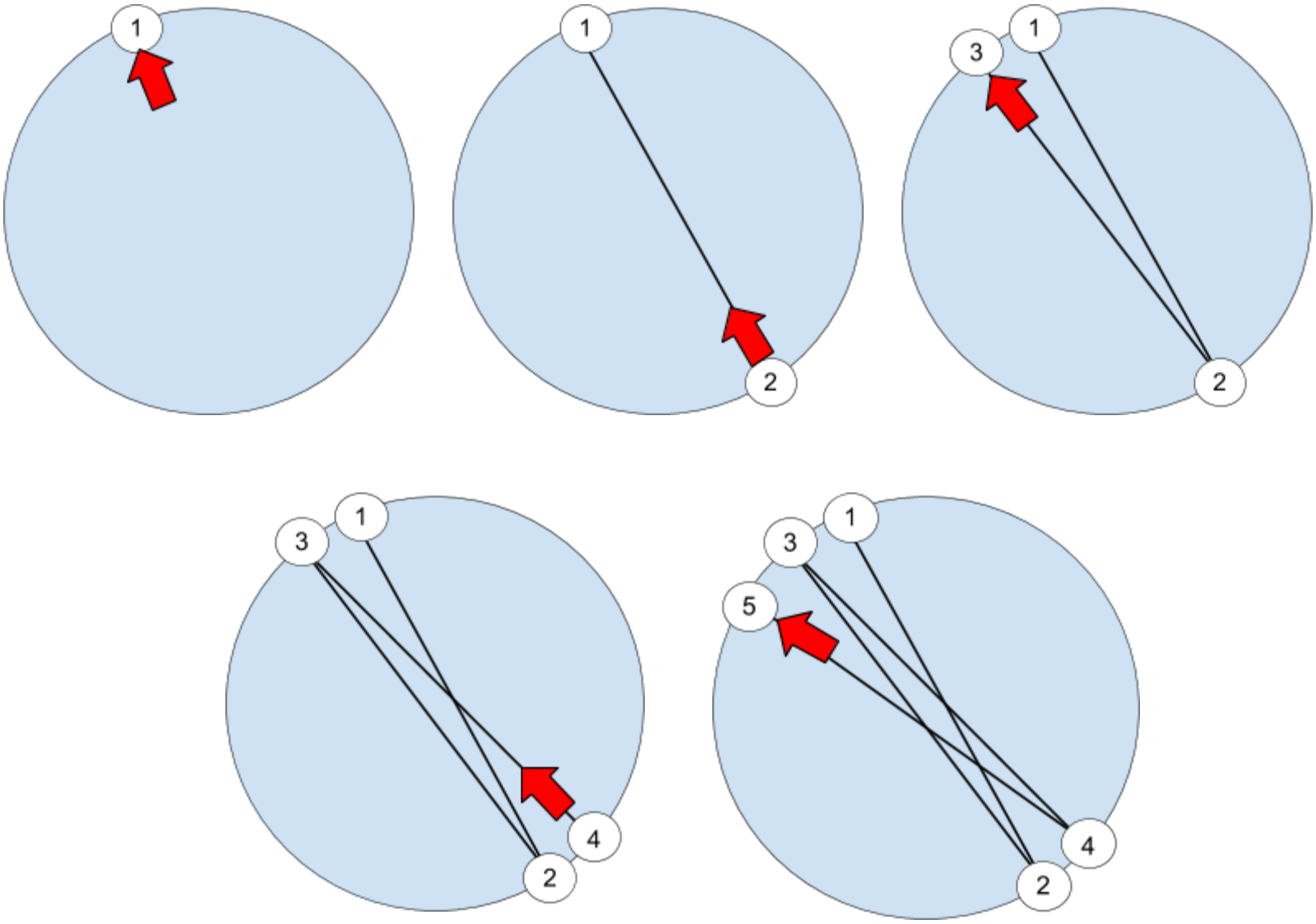
Depending on the complexity of the environment the robot is in, generating a map will take a varied amount of time and may involve different methods of navigating within it. For instance, generating points when the robot is facing a white wall will not work very well, and will require you to position the robot in a way that there are some distinguishable features in the captured image so that points can be generated. On the other hand, a cluttered room with colorful tapestry may generate a lot of points that are not accurately located. Keep in mind that this will

play a part in localization as well, as if the robot is not able to recognize points, it will not know where it is located.

In order to test the system, I conducted tests in four different environments; a kitchen, two living areas, and a bedroom. I used methods of wall-following during map generation, having the robot rotate in the middle of the room during map generation, and manually navigating the robot throughout the environment.

The methodology I used for manually navigating the robot in the environment while creating maps is as followed; positioning the robot in one corner or side of the room. You will want to place the robot (camera) facing the corner and navigate the robot in reverse, away from the corner, until ***ORB_SLAM2*** picks up enough unique points to start initializing the map. Then you will want to start a process of moving the robot forwards and backwards the full length of the room, while slowly adjusting the angle of the robot, until the robot fully turns and captures 360 degrees. During this process, the robot will likely lose its location, so you will need to reposition the robot towards objects that it has already seen. After traveling a full circle, if the generated map appears to be mostly accurate, you will want to then manually navigate the robot throughout the environment, mapping all of the objects individually, to ensure that the finer details of the map are captured. You should test that the map is complete by navigating throughout the entirety of the map and ensuring that the robot stays localized. If the robot loses localization, you should repeat the motion as detailed above until the robot can stay localized at that point.

Figure 5. Diagram showing the motion of the mapping methodology



Additionally, maps can be saved using:

```
rosservice call /orb_slam2_mono/save_map <name>.bin
```

Maps can be opened for localization only(where the map is no longer being built and only used for localization) or a map can continue being built. Modify ***pi3_orb_slam2_localization_only.launch*** to contain the name of the map you wish to save.

<!-- static parameters -->

<param name="load_map" type="bool" value="true" />

<param name="map_file" type="string" value="<name>.bin" />
//enter the name of the map file

Additionally if you want to continue building a map, modify the same parameters in the ***pi3_orb_slam2.launch*** file

<!-- static parameters -->

<param name="load_map" type="bool" value="false" /> ***//set this to true***

<param name="map_file" type="string" value="<name>.bin" />

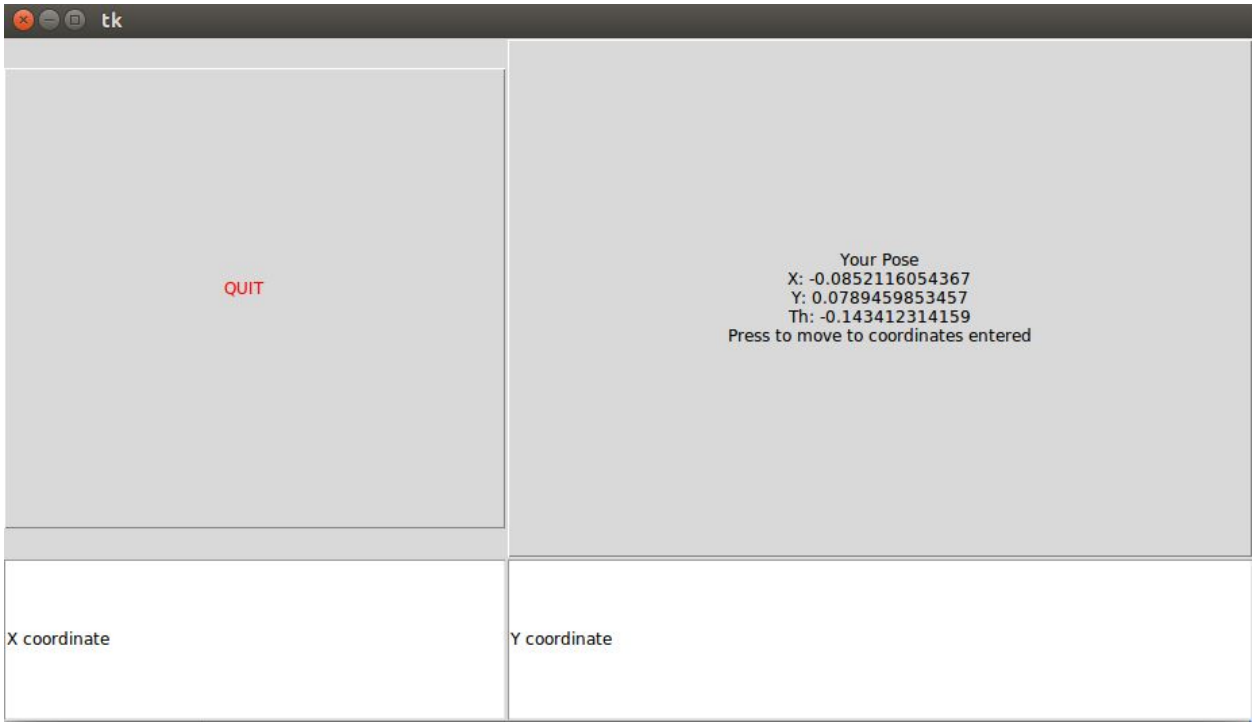
Step 5. Running bug algorithm

Once you have created a map that is detailed enough that the robot can maintain localization throughout the entirety of the environment, you can now run the bug algorithm

rosrun robot_client pi3_slam_nav.py

The GUI will display the robot's position (x, y, theta) in the map, and allow you to enter the (x, y) coordinates of the goal position. You will want to use Rviz to view the map and determine your goal point because, as aforementioned, the coordinates of the Pose from ***ORB_SLAM2*** are not to true scale, and are not represented by inches or meters. ***ORB_SLAM2*** appears to approximate the depth of the image differently each time the map is initialized, so there does not seem to be a way to convert these coordinates to a unit of measurement.

Figure 6. pi3_slam_nav.py GUI



IV. Discussion and Results

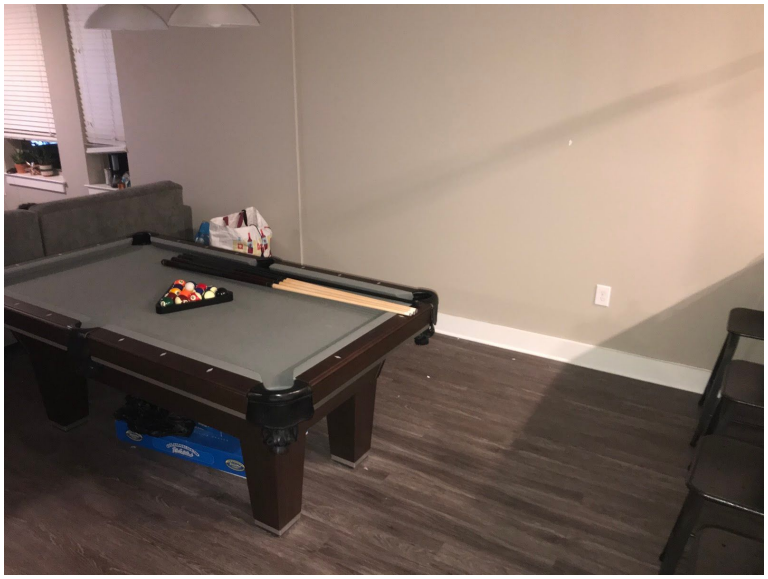
One factor that hinders this from being a viable option for localization is the performance of the camera over ***ROS***. As aforementioned, using a higher resolution for the images provides lower FPS, making it difficult for ***ORB_SLAM2*** to maintain localization. Unfortunately, ***ORB_SLAM2*** does not work with compressed image topics. The bottleneck for the performance appears to be on either the CPU or Wifi card of the robot, as I have tried running ***ROSCORE*** on both robot, and my client computer with the same results, with my computer's CPU and Wifi card running significantly under max load. The system was additionally tested under 3 different wifi networks with the same results as well.

Because of these issues, autonomous exploration for mapping does not seem feasible and did not work with tests using wall following or moving the robot in a circle. Depending on the complexity of the frame, the time it takes to generate points widely differs. Therefore, according to my research, it is best to manually navigate the robot using ***telop*** to map the environment.

During manual map generation there are still significant issues while generating a map, such as the inability to generate points on objects that are too dark or light. This issue was also documented by Yu-Ting Chung who was working on integrating ***ORB_SLAM2*** onto a raspberry pi with a camera[12]. This results in parts of the environment not being mapped, and leaving holes in the PointCloud2 map, which in turn leads to sections of the environment that the robot cannot maintain localization in. I tested the map generation in a kitchen, two living areas, and a bedroom, and only the map generated in the bedroom was usable.

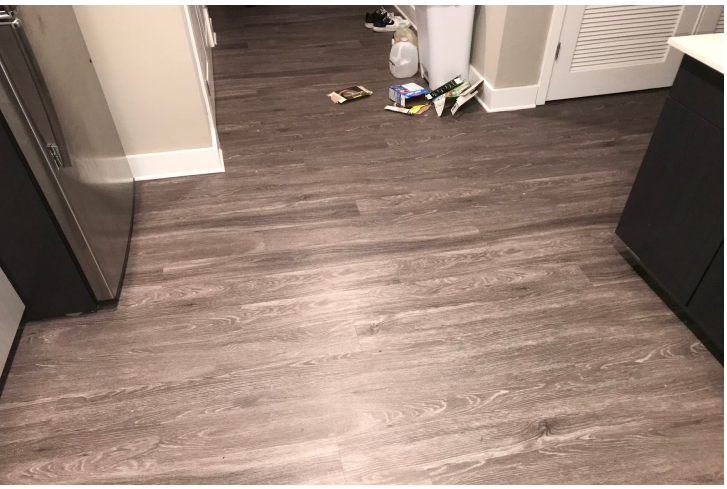
[12] https://github.com/biorobaw/SLAM-S2018/blob/master/docs/Reports/final%20report.pdf

Figure 7. Living area #1



One of the living areas had white walls which would not be mapped by the environment

Figure 8. Kitchen

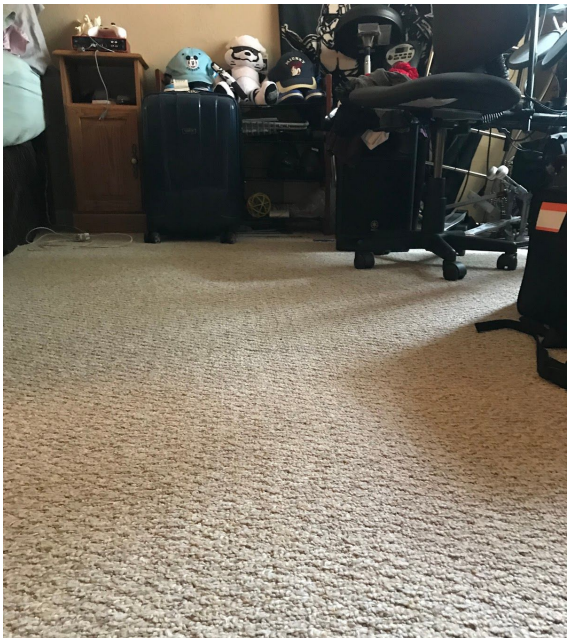


The kitchen, in addition to the black cupboards which would not generate points, had many reflective surfaces which either would not get picked up, or would generate points in incorrect locations related to their physical location in the environment.

Figure 9. Living area #2

The other living area had a black footstool which would not be picked up, in addition to a colorful carpet that also generated points in the environment that did not correspond to any physical counterpart.

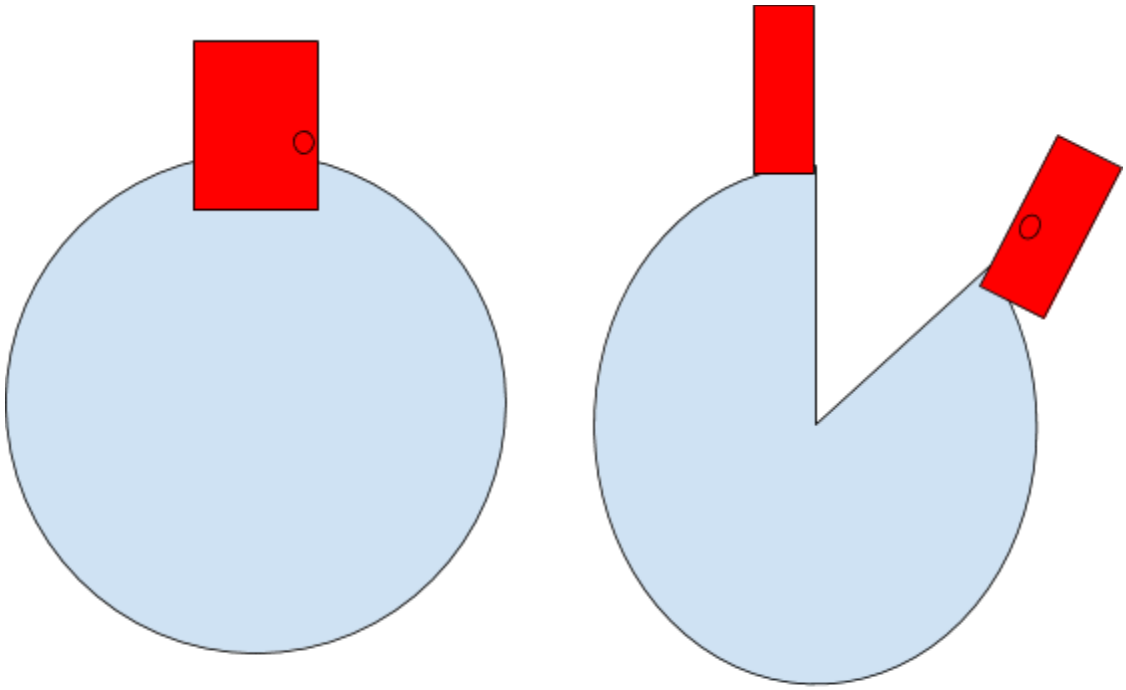
Figure 10. Bedroom



The last environment, which ***ORB_SLAM2*** was capable of generating a usable map in, was a bedroom that had enough unique objects in the environment to generate the points needed to allow the robot to maintain localization.

Another issue from map generation is that mapping finer details of an environment before capturing the entire environment through a panoramic view would result in the corners of the map not meeting. For example, if the robot was inside of a circle with a red door, instead of capturing the full 360 degrees of the environment, the map would have a portion of the map left blank with the rest of the environment skewed.

Figure 11. Diagram of skewed map



Throughout the research and development of this project, I had discovered a plethora of existing ***ROS*** packages used for autonomous navigation and path planning such as Exploration_lite[13] and Frontier_exploration[14] that used the ***navigation stack(http://wiki.ros.org/navigation***). However, through further research or trying to run these programs, I ran into the common issue that the robot did not have sensors that met the requirements of these packages. The ***navigation stack*** specifically requires a ***tf***(transform) linking the ***map***, ***odometry***, and ***base_link*** coordinate frames together, however, ***ORB_SLAM2*** only publishes the link ***tf*** for the ***map*** and the ***camera_link***. Throughout researching ways to insert the odom information into the already existing map->camera_link ***tf*** tree, there was a lack of resources detailing this process. There were a handful of discussions in the ***ROS*** community where users urged others in this predicament to simply use another ***SLAM*** package that does provide the proper ***tf*** information, such as ***rtabmapping*** and ***gmapping***, as this would be more trouble than it is worth and likely yield inaccurate results[15]. However, these packages require ***Laser Scanners*** or ***RGB-D*** cameras.

Testing ***ORB_SLAM2*** and researching ***navigation stack*** and ***SLAM*** alternatives has suggested that using a single mono-camera, which ***ORB_SLAM2*** also does not support RGB processing for, does not provide high quality results for complex usage. Mapping can take an upwards of an hour to generate a map in a 11.5'x13' bedroom in certain instances, and in other environments a complete map will not be able to generate at all. Once an accurate map is correctly generated, the coordinate frame of the map does not correspond to actual units of

[13] http://wiki.ros.org/explore_lite
[14] http://wiki.ros.org/frontier_exploration
[15] https://answers.ros.org/question/351167/transformation-between-odom-and-camera_link/

measurement, so the encoders cannot be used to approximate the location of the robot in the event that the robot loses its track.

Once the map is generated, a package called ***Octomap,*** a ***ROS*** package that can generate occupancy grids from ***LaserScanner*** or ***PointCloud2*** topics, can be used to generate an occupancy grid[16]. However, due to inaccuracies of the pointmap generated by ***ORB_SLAM2***, ***Octomap*** gets several false negatives where cells are marked as occupied even when there are no physical obstacles in the location. An example of this would be points generated from a colored carpet which would not obstruct the robot's movement, but is picked up by ***ORB_SLAM2*** and marked as an obstruction by ***Octomap***. Perhaps it is possible to tweak the launch files to reject obstacles unless it is between certain z coordinates(***height***), this could be resolved, however, currently the occupancy grids generated are not accurate enough for path planning. Additionally, in my experience, running ***Octomap*** simultaneously significantly decreases performance. I believe there is a bottleneck in the robot's networking speed, because the amount of image frames sent by the camera drops, decreasing the reaction time for the localization.

[16] http://wiki.ros.org/octomap

V. Conclusion

While ***ORB_SLAM2*** can provide us with pose information inside of its depth approximated coordinate frame that can be used for bug algorithms and other simple applications, using ***ORB_SLAM2*** as the primary form of localization has several limitations.

The performance of ***ORB_SLAM2*** on the robot appears to be bottlenecked by the robot's wifi transfer rates, as transmitting images at ideal resolutions results in low frame rates, making localization inaccurate. While increasing this transfer rate, perhaps by getting a faster wifi card, may be a solution, I believe that better results would be achieved by acquiring additional sensors such as an IMU or laser scanner. Having these sensors would allow us to generate better maps and have more accurate localization, as well as broaden the compatibility with ***ROS*** packages used for autonomous navigation, which I believe would provide a system with a more practical use.

Works Cited

Camera Calibration. (n.d.). Retrieved July 26, 2020, from https://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html

Chung, Y. (2018, July 18). Final Report. Retrieved July 30, 2020, from https://github.com/biorobaw/SLAM-S2018/blob/master/docs/Reports/final%20report.pdf

Gmapping. (n.d.). Retrieved July 26, 2020, from http://wiki.ros.org/geometry_msgs

Gmapping. (n.d.). Retrieved July 26, 2020, from http://wiki.ros.org/gmapping

Navigation. (n.d.). Retrieved July 26, 2020, from http://wiki.ros.org/navigation

Octomap. (n.d.). Retrieved July 26, 2020, from http://wiki.ros.org/octomap

Orb_slam2_ros. (n.d.). Retrieved July 26, 2020, from http://wiki.ros.org/orb_slam2_ros

Rtabmap_ros. (n.d.). Retrieved July 26, 2020, from http://wiki.ros.org/rtabmap_ros