# Experimental Robotics Final Report

**Team 2**

Sidhardh Burre 12/6/22

Julian Lee 12/6/22

Avaneen Pinninti 12/6/22

University of Virginia

Format (/10):

Description (/15):

Results (/10):

Conclusions (/5):

# 1 Design Objectives and Constraints

The overall objective was to develop a robot that can autonomously navigate and drive to a charging plate, detect the charging plate, and dock at the charging station. Due to various time constraints in the semester, the objective was divided into three options (in increasing point worth and complexity): remote control the robot manually and detect and dock to the charging plate using a video feed, have the robot navigate to a desired location specified in rviz and approach the charging plate on its own, and have the robot be fully autonomous, including searching for and approaching the drogue, detecting the drogue, and automatically completing the charging circuit. Our group completed the first option due to limited time, but put in work to accomplish the autonomous navigation though it was incomplete.

As with the midterm, one subtask was that the robot was required to be driven into a charging station and then signify that it was charging by sequentially lighting state of charge (SoC) LED indicators as time passed. Another was that the docking mechanism must be designed and fabricated by the team. Further, it was required that the robot stream video output to the base station. The robot had to be controlled via a joystick. All components had to be integrated together to form a cohesive system accomplishing all tasks. The charging plate and AR tag also had to be detected by the robot and displayed in rviz for our task. The higher deliverables we were working towards also required navigation be done automatically when a 2D Nav Goal is specified within rviz and that the robot can detect the pose of the AR tag and automatically dock.

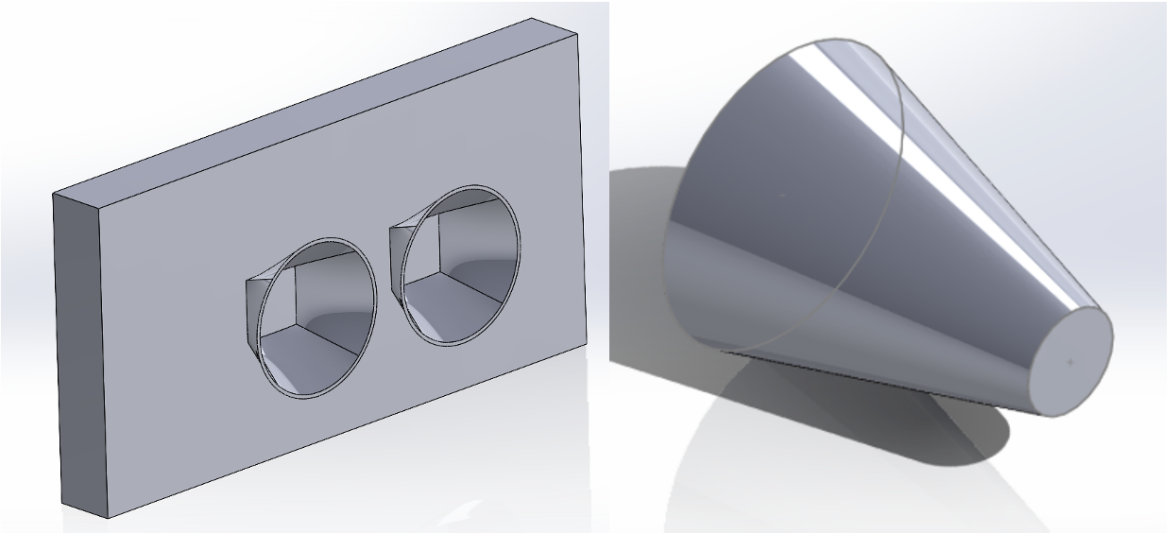
Constraints included the hardware and docking parameters. Much of the robot hardware was provided including the chassis, drivetrain, and hardware controllers. This included a pair of motor controllers (one for each side) and four motors with wheels. Each side of the robot's wheels could be controlled but capabilities for wheel-based control do not exist. Additional hardware included an Arduino Uno and basic electrical components such as breadboards, LEDs, and resistors. For imaging, two cameras (one RGBD and one odometry) were used for navigation and detection of important features. The charging dock was a rectangular aluminum plate mounted with its centerline 650 mm above the ground with a width of 180 mm. The robot was required to have two contact points with the charging dock to complete a "charging" circuit and light the SoC indicators (the charging circuit did not actually charge the battery due to safety concerns but served only to light the LEDs). For the higher deliverables, the robot also needed to be properly localized within an existing map (or use SLAM) and be able to receive navigation goals through rviz.

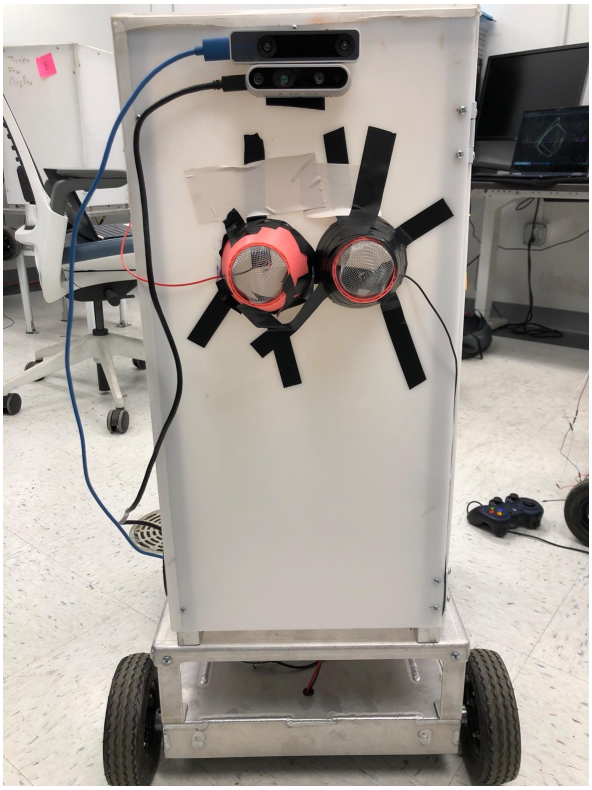
# 2 Hardware System

The hardware system of our robot consists of four parts: charging probes, status of charge LEDs, drivetrain motors/motor controllers, and our two cameras (RealSense D435i and RealSense T265) used for perception as well as navigation. Our Arduino, which is powered by

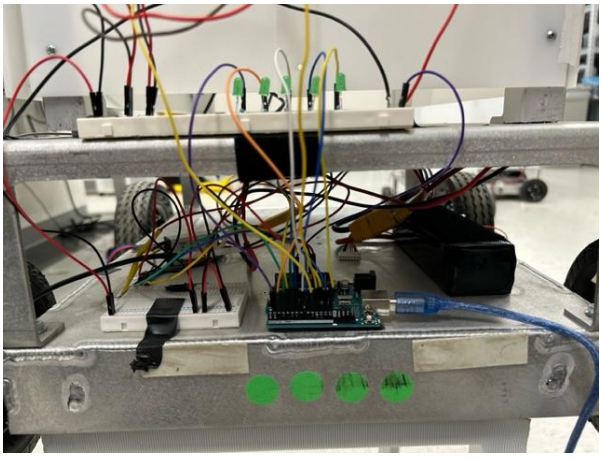
our robot computer (laptop) via USB connection, is central to these four components. Our team did not use an Intel NUC for this course as we were unable to acquire one due to shipping delays. As we did not have an Intel NUC, our robot did not require the use of a voltage splitter from our 22.2V battery. Our 22.2V battery powered our robot's four IG42 24VDC 013 RPM Gear Motors with Encoder.

Starting with our charging probes, two 3D printed conical frustums served as our probes for this project. Below is an image of the CAD model as well as how they looked when attached to our robot. We covered the tops of the frustums with a metal wire mesh which allowed for better conductivity when the two "probe" wires connected to the Arduino and the metal "charging" plate. We also planned on using drogues that would be mounted on the charging plate that would allow our frustums to easily slide in and make contact with the "charging" plate. Unfortunately, we did not implement the drogues as they were always 3D printed incorrectly. The CAD model of the drogues can be seen below. One "probe" wire is connected to pin A0 of the Arduino which was configured as a digital input pull-up resistor. The other wire was connected to a Ground output from the Arduino. This setup results in pin A0 reading a LOW whenever the probes make contact with the metal "charging" plate (as the circuit is completed and the Ground from one pin "pulls down" the signal from the input pull-up resistor), and HIGH otherwise. More of this can be seen in the circuit diagram below, specifically the wiring for the charging system is colored orange.

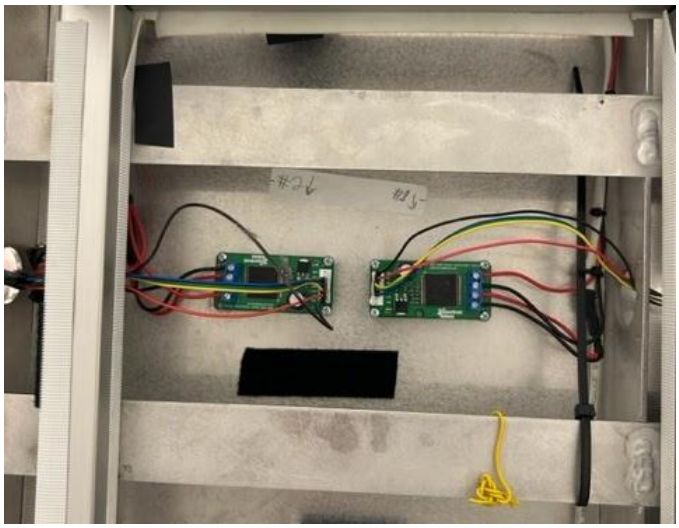




Moving onto our status of charge LEDs, these are connected to digital pins 1, 2, 7, 8, and 13 which were configured as output pins on our Arduino. When our charging probes made contact with the metal charging plate, the LEDs turned on in sequence (with a 5 second delay between each one) to indicate charging. The first LED stays lit the entire time our robot is in operation as an indication that our robot is functioning well. The image below shows the connections between our Arduino and the 5 LEDs. A circuit diagram further down shows the connections with a blue wire along with a 1k ohm resistor to ensure our LEDs do not burn due to excess current.

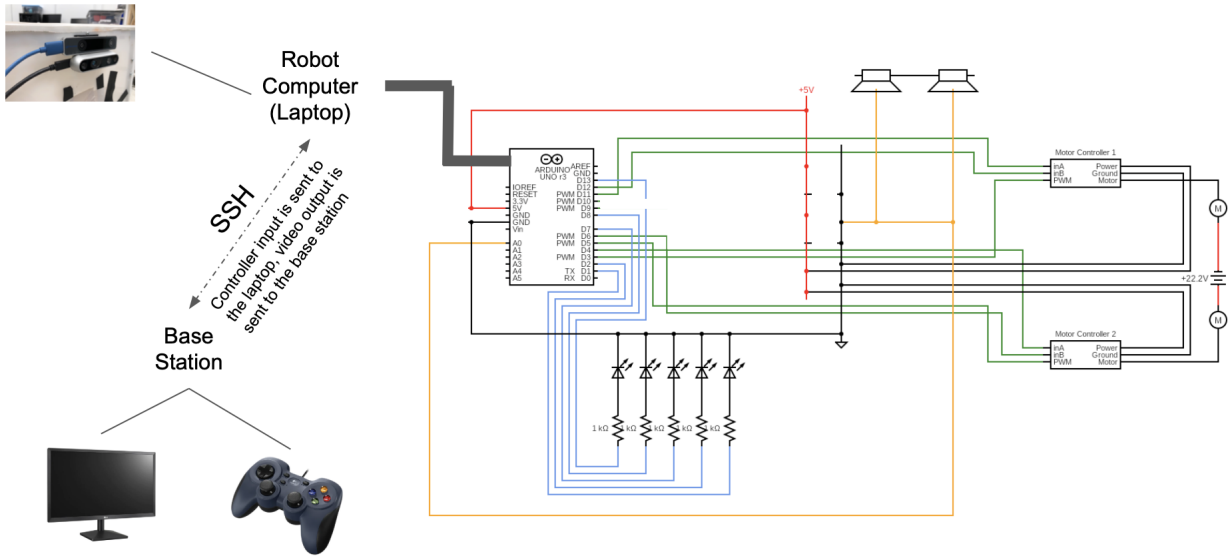


The drivetrain was controlled by four motors connected to our Arduino via two motor controllers. We used a 22.2V LiPo battery to power the motors which can be seen in the image above. The two motor controllers were powered by our Arduino (both motor controllers have an input for 5V and Ground. Each motor controller has 3 pins - inA, inB, and PWM - which control the direction and speed at which the motors move. Pins inA, inB, and PWM on one of our motor controllers are connected to digital output pins 4, 6, and 5 on our Arduino while pins inA, inB, and PWM of our second motor controller are connected to digital output pins 11, 12, and 3. Based on controls from our joystick, HIGH and LOW messages are sent to the respective inA and inB pins on our motor controllers which sets the direction the motors will turn. The signal sent from the Arduino to the PWM pins controls the speed at which the motors turn. The image below shows the motor controllers underneath our robot and the wires that connect to their respective Arduino output pins. The connections to the motor controllers can be identified on our circuit diagram as green wires.

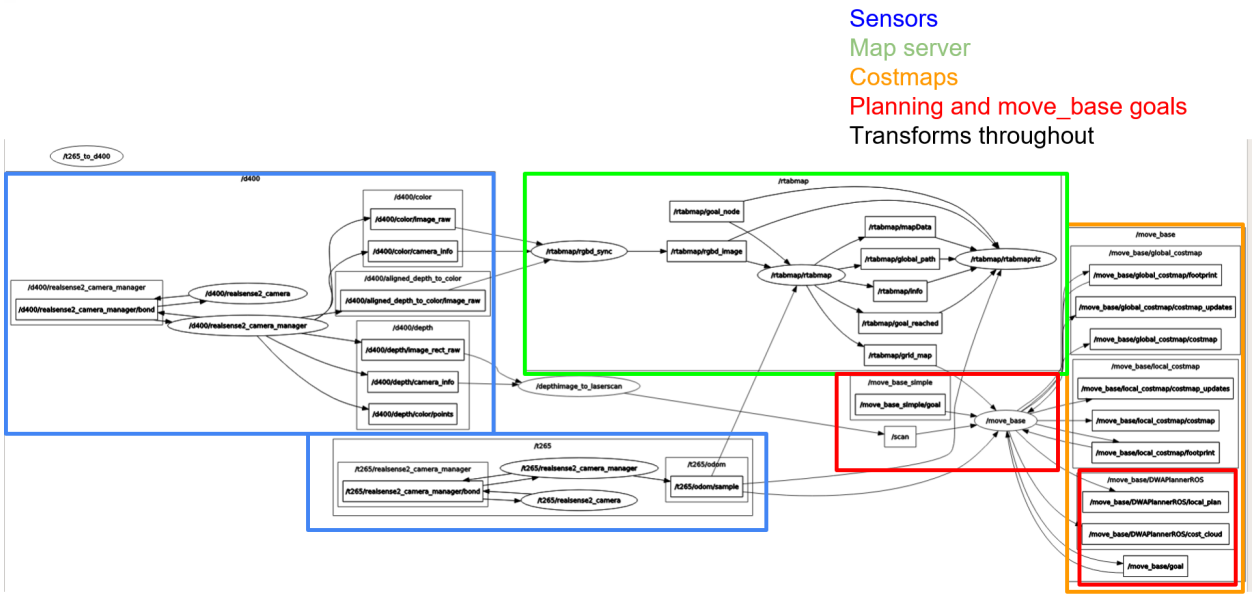


The last component, our camera system consisting of the RealSense D435i and RealSense T265, allowed us to view the robot's surroundings, detect AR tags using ar_track_alvar, detect the charging plate with with find_object_2d, access SLAM capabilities as well as use the navigation stack. The camera system was mounted to the front of our robot, unlike atop our robot which is what we did for the midterm assignment. Mounting the camera system to the front of our robot helped keep the system rigid and stable - which is very important for SLAM and navigation. We also decided to stack the cameras vertically which was once again helpful for utilizing the navigation stack and SLAM. Both cameras were connected to our robot computer via USB which was all that was needed to power the cameras and for our software to communicate data with the cameras. A picture of how we mounted the cameras can be seen below:

The circuit diagram below highlights all four components described above. The base station, joystick, and SSH connection is not a part of our hardware system and is part of our larger software and integrated system.



# 3 Software System




The figure above displays the software that was employed to control our robot. The software involved in moving and controlling our robot is split up into several parts: sensors, map server nodes, costmaps nodes (nodes related to the 2D occupancy grid), and planning/move_base goal nodes.

Starting with sensors, our robot relied on information extracted from various nodes related to the RealSense D435i and RealSense T265. Our team used the /d400/color/image_raw topic to stream the input received from the D435i camera. We used this information to control our robot around the map. Other nodes and topics related to the D435i and T265 were used by the RTAB-Map nodes to localize the robot and display its position in the rviz and RTAB-Map GUIs. Lastly the planning and move_base goals nodes/topics were used in the navigation stack to send a goal position to the robot in the rviz GUI.

One part of our software system that is excluded from the image above is the architecture used to detect AR tags. Nodes in the ar_track_alvar package used information from the RealSense D435i to locate and calculate the pose of AR tags with respect to the camera. The d400 base_link transform was especially useful in this case. Additionally nodes in the find_object_2d package would reply on information from the d400 nodes (especially the /d400/color/image_raw topic).

Similar to the software used in the midterm, the /joy_node was used to take in input from a Logitech F310 Wired Controller and published the raw controller output to the /joy topic which was then read by /teleop_twist_joy node which converted the input to a proper format to the /cmd_vel. Namely, the /teleop_twist_joy node only outputs to /cmd_vel when the right bumper on the controller is pressed, and processes the stick input such that the right stick is used for turning the robot and the left stick is for moving the robot back and forth. Translation motion with the left stick was published to the topic as the linear.x component and rotational motion

with the right stick was published as the angular.z component. These components were further processed by the /serial_node to activate the motors. The /serial_node, effectively running on the Arduino Uno, subscribes to the /cmd_vel topic. Within the Arduino code, care is taken to prioritize turning action of the robot over forward/backward action and safeguards are put on the maximum speed the robot can use. Note that the /serial_node is a wrapper node for the Arduino Uno.

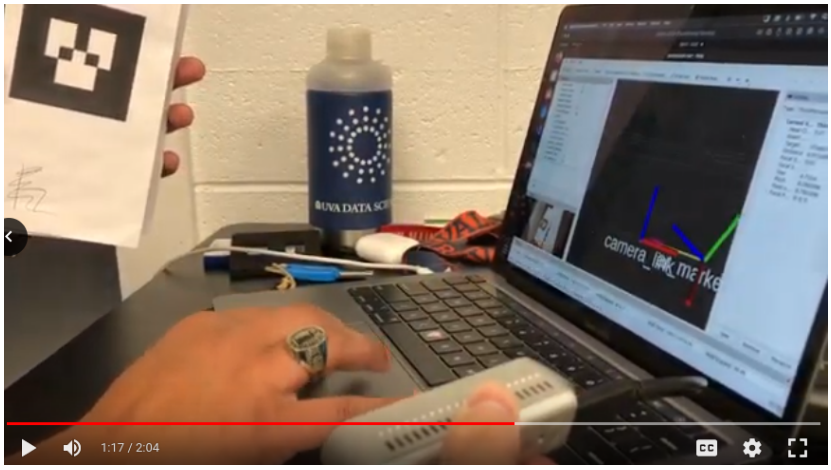
Additional logic was included in the Arduino Uno to control the charging display. The Arduino was programmed to monitor the duration of connection to the charging circuit. Every five seconds another led was programmed to turn on. Due to how our pins were configured the first bulb would always be on and the second bulb would appear to take 10 seconds to light but this was considered a feature of our design.

# 4 Approach

## 4.1 Perception

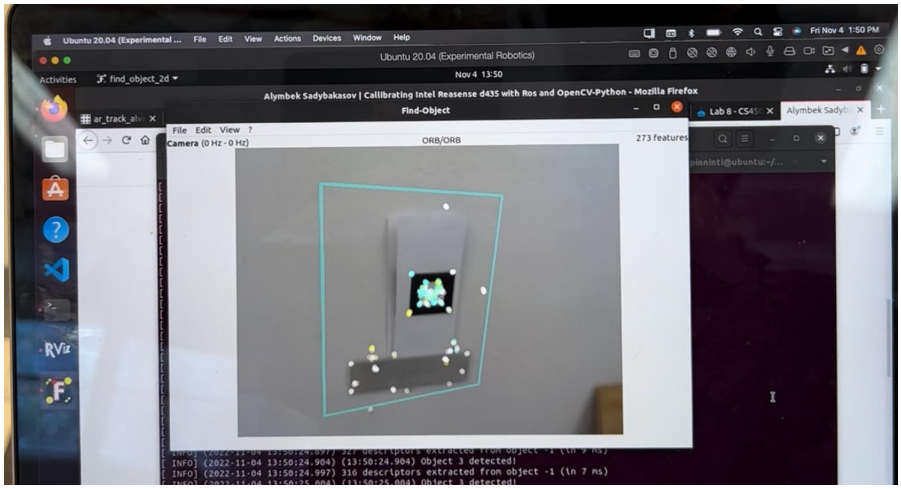
### 4.1.1 AR Tag Detection

The Intel RealSense D435i RGBD camera was for video input to detect the AR tags. We launched the camera then used the ar_track_alvar package to automatically detect an AR tag. We also launched rviz to be able to see the pose of the robot and AR tag together. Doing so simply involved adding a TF view into rviz, since ar_track_alvar would automatically detect an AR tag and publish its transform which is detected by rviz. Real time visualization of the AR tag and robot was shown in rviz; if the fixed frame is set to the “camera_link” topic, then the position of the tag relative to the robot is shown. The actual pose detected by ar_track_alvar was also published to the /ar_pose_marker topic. This was used in our final demonstration, but we also later republished the pose data from /ar_pose_marker to be used in the robot navigation by setting the pose as a move_base goal. The figure below shows an image of the basic AR tag detection process, with both the camera setup and rviz visualization shown.



### 4.1.2 Charging Dock Detection

We again used the D435i camera for detecting the charging dock. This time, we used the find_object_2d package to detect the charging plate. After launching the camera, we launched the find_object_2d GUI, which takes input from the camera and displays it directly. Detected features are automatically shown as yellow circles in the image. We saved an object by capturing an image and selecting the region of interest. Then, we loaded the object and once sufficient features were detected in find_object_2d, the charging dock was automatically highlighted on screen. An image of detecting the charging dock at a slight angle is shown below.




### 4.1.3 Environment Detection

Although we did not yet complete the fully autonomous robot navigation, our work on it involved perceiving the environment in order to navigate through it. We used the D435i camera and also an Intel RealSense Tracking Camera T265 for odometry. This allowed us to create and use a map of the lab space using the ros_rtabmap package. Details are elaborated in Section 4.2 below.

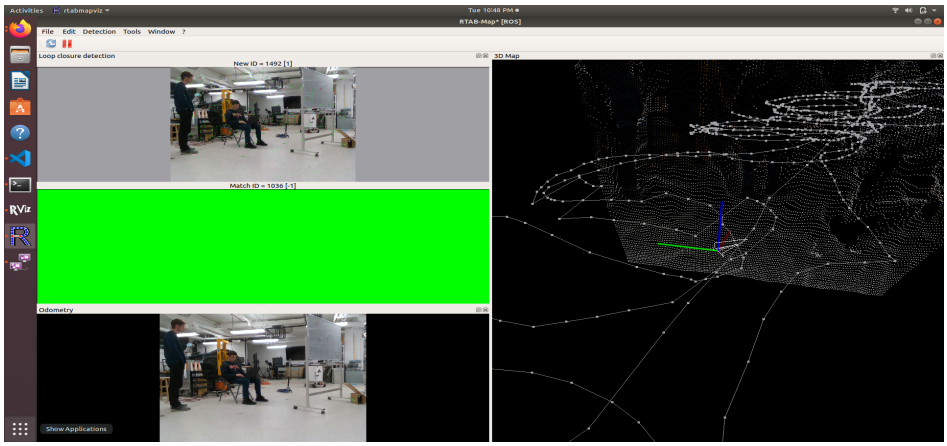
## 4.2 SLAM

Simultaneous Localization and Mapping (SLAM) is a method used by autonomous vehicles to build a map of their environment as well as localize themselves in that map at the same time. SLAM generates a map that can be saved for future use. A map allows robots to efficiently carry out their tasks such as localization and path planning. For the best SLAM results, we used both the RealSense D435i and RealSense T265 to collect RGBD (color and depth) and odometry data. This information helped us develop a more accurate and feature rich map which in turn yields a better occupancy grid map. A 2D occupancy grid map is frequently

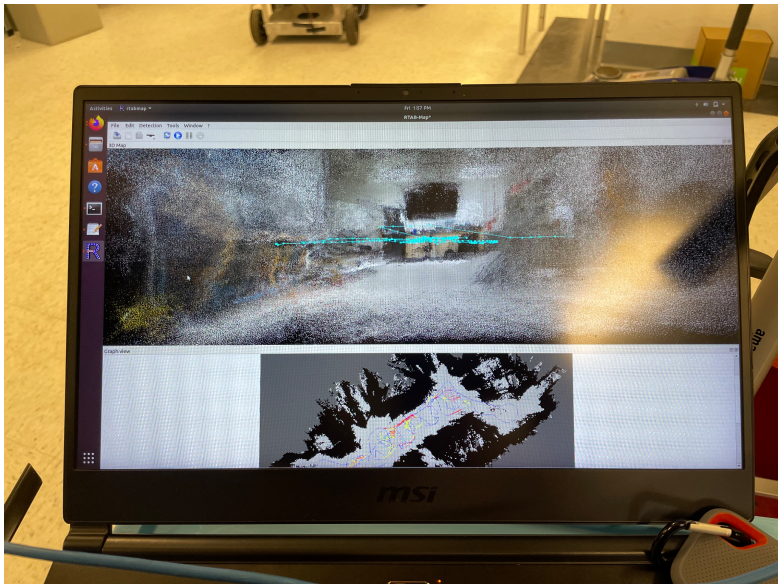
used in ROS and other robotics applications to describe an environment as an area of occupiable and non-occupiable cells.

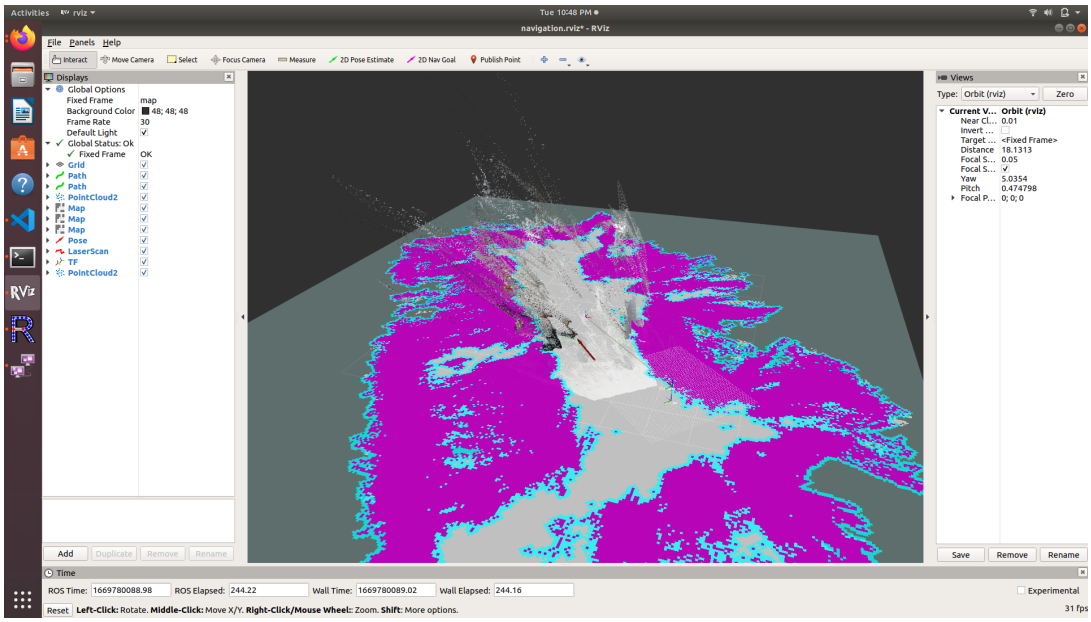
To start the SLAM process, we used the “rs_rtabmap.launch” launch file from the “realsense2_camera” package. This launch file opens the RTAB-Map GUI, RVIZ, and begins the SLAM process. Once the SLAM process has been started, one can see the map being built in the RTAB-Map GUI. We noticed several things that are important to the SLAM process and ensuring we get a high map. First, it is important to move the cameras in a slow and stable fashion around the entire area (in this case it was the lab). Moving slowly ensures that our cameras detect all necessary features. Next, loop closure is a vital part of the SLAM process. Loop closure is when a robot recognizes it has returned to a previously mapped region, and uses this information to reduce the cumulative error of the robot’s estimated pose (localization).

The picture below depicts the RTAB-Map GUI during SLAM. Various parts of the GUI include the path the robot has taken so far (used in localization), the camera feed (used to detect features and create a feature map), and a green rectangle that serves as a status indicator for mapping and localization.




The two pictures below show the 3D point cloud map generated by SLAM as well as the 2D occupancy grid map. The 2D occupancy grid map is used extensively in the navigation stack.






## 4.3 Navigation

For robot navigation, a 2D navigation system was implemented via the ROS Navigation stack with modifications to interface with our specific robot hardware. A high-level overview of the Navigation system appears as described below in Figure XYZ.

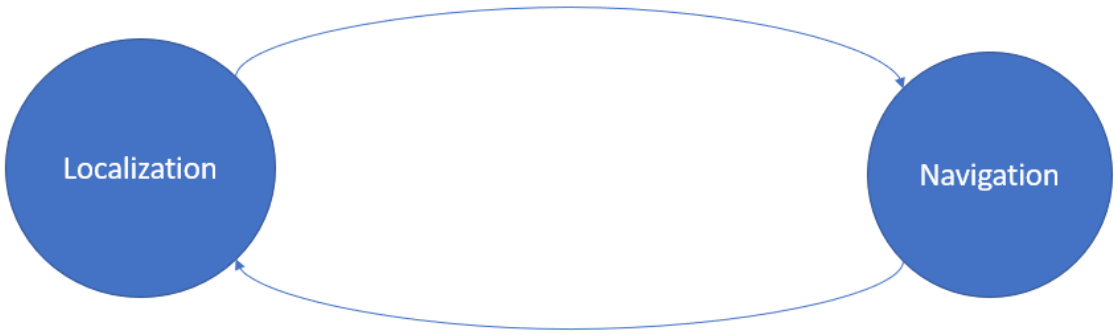



Figure XYZ. FSM of Navigation Strategy

In general, the navigation stack would attempt to localize by rotating in-place until the navigation stack is able to localize the robot within the pre-generated 3D map. Once the robot is localized within the map, it proceeds to the navigation step, simultaneously updating the map and sending movement commands to the robot hardware in real-time.

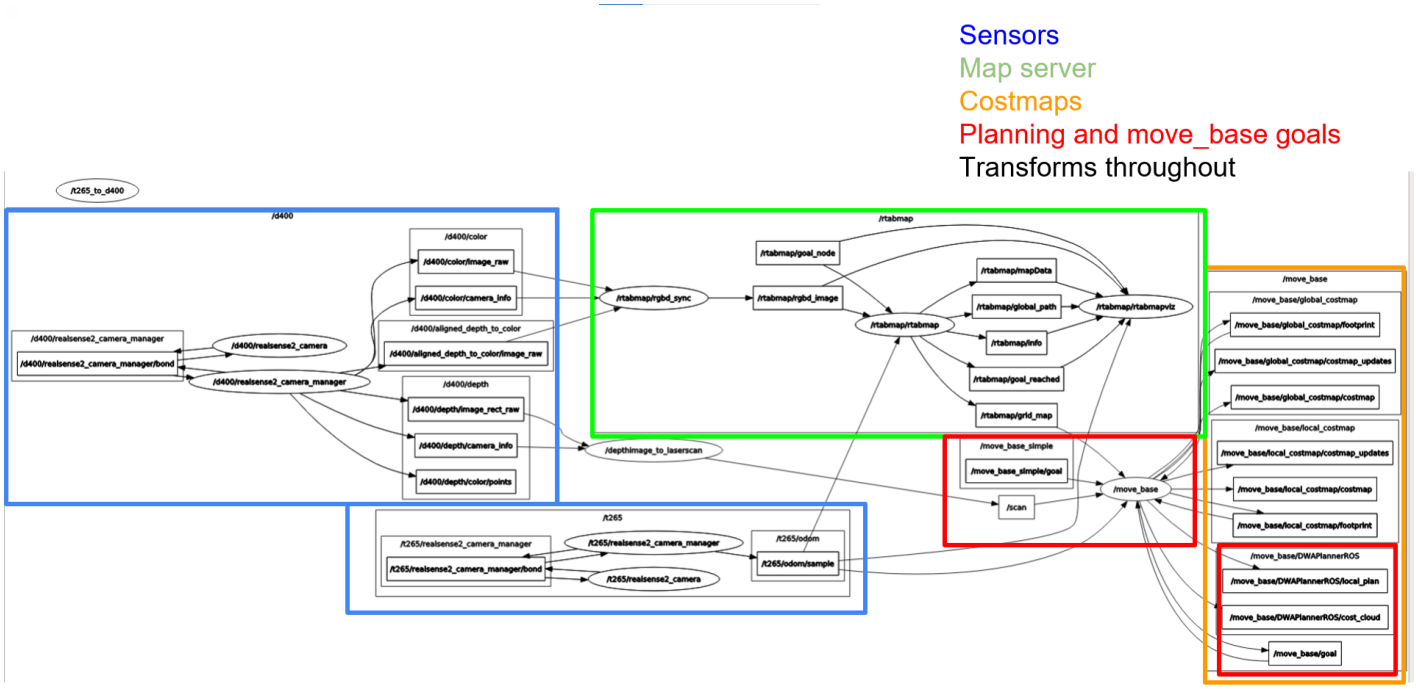
## 4.4 Autonomy

We did not complete the autonomy part of the final assignment, but were making progress towards it. Autonomy is handled by the system, which takes in various parameters and inputs and dictates which actions to take next to achieve an overall goal. The ultimate goal was to have the robot autonomously drive around the lab until the AR tag is detected, at which point it would adjust its path to dock at the charging station on its own. The charging dock's coordinates would be known in our premade map, but the robots initial position would not. When the robot is placed somewhere in the lab, it would first localize itself somehow (see the SLAM section), then begin heading towards the coordinates of the charging dock once it properly localized. From there, the perception systems (AR tag and drogue detection) would take over and send updated pose estimates as a nav goal. The navigation stack would then use the goal to plan and create a path, then ultimately drive there by sending desired velocities to the Arduino. The Arduino has internal code to translate the /move_base/cmd_vel topic velocities published from the nav stack to the desired velocity formats we implemented for motor control.

We were not able to properly localize the robot, as it would keep spinning and not properly seek out walls and other features to gain its bearings. This was likely due to some

mistuned parameters and an unclean map. However, the velocity translation and AR tag pose republication was implemented.

Shown below is an image of the node-topic graph from rqt with different sections of the navigation stack color-coded. We had all components except the costmaps and move_base parameters tuned properly, so this is why the autonomy did not work for us yet. The connection to the Arduino is not shown on this graph, but velocities published by move_base would be sent to the serial_node located on the Arduino, which would then manipulate the data to be usable by the motor controllers.




# 5 User's Manual

## 5.1 Hardware Setup

To configure the robot hardware, the following steps were taken.

1. Plug Arduino into robot computer via USB
2. Plug D435i Intel RealSense camera into the robot computer via USB
3. Plug Intel T265i Intel RealSense camera into the robot computer via USB
4. Plug battery into robot battery input
5. Power on robot
6. Run command "roscore"

## 5.1 AR-Tag Detection

To detect AR-Tags the following steps were taken.

1. Run command “roslaunch ar_track_alvar pr2_indiv_no_kinect_realsense.launch”
2. Run command “roslaunch realsense2_camera rs_camera.launch”
3. Run command “rviz”
4. Within rviz, the map frame was converted to the “camera_link” topic so that the cameras are used as the base frame
5. Display the transform topic to show the ar-tag itself

## 5.2 Drogue/Charging Plate Detection

To detect the charging plates the following steps were taken.

1. Run command “roslaunch ar_track_alvar pr2_indiv_no_kinect_realsense.launch”
2. Run command “roslaunch find_object_2d find_object_2d.launch”
3. Extract a picture of the relevant image from the find_object_2d GUI
4. Update image within GUI if necessary

## 5.3 Manual Control

To perform manual control of the robot two laptops were required and the following steps were taken.

1. Both computers were connected to the lab VICTOR-5G network
2. The computers IP’s were inspected via “ifconfig -a”
3. Both computers “.bashrc” files were edited to contain the following lines where IPADD is the corresponding machine’s IP address
    a. “export ROS_MASTER_URI=http://[base station IPADD]”
    b. “export ROS_IP=IPADD”
4. On the base station computer the following commands were run:
    a. “roscore”
    b. “rqt” with the

## 5.4 Autonomous Control

To launch the autonomous control stack, the following steps were taken

1. Ensure auto.launch parameters are correctly specified
2. Run command “roslaunch leader_2dnav auto.launch”
3. Within the rviz interface a goal pose was selected and delivered

# 6 Experimental Validation

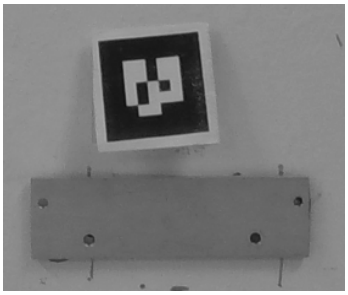
## 6.1 Drogue detection

Drogue detection was performed primarily via the find_object_2d package. This package extracted features from a pre-selected image of the charging plate that was modified and updated to improve the system’s recognition accuracy. Although the primary use-case for this system was straightforward charging plate detection, angled detection was tested as well. Furthermore, visual modifications were made to the charging plate and the corresponding accuracies were tested.

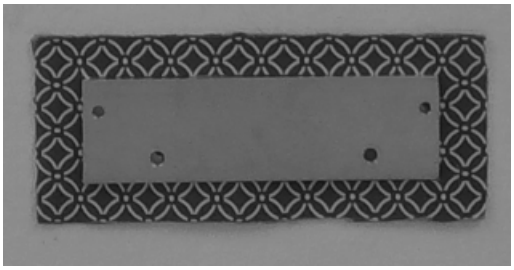
The following charging plate configurations were tested:

1. Plain charging plate (configuration type “a”)
2. Pattern-bordered charging plate (configuration type “b”)
3. AR-Tag charging plate (configuration type “c”)

(a)



(b)



**(c)**

The number of features that the software detected are listed in Table 1.

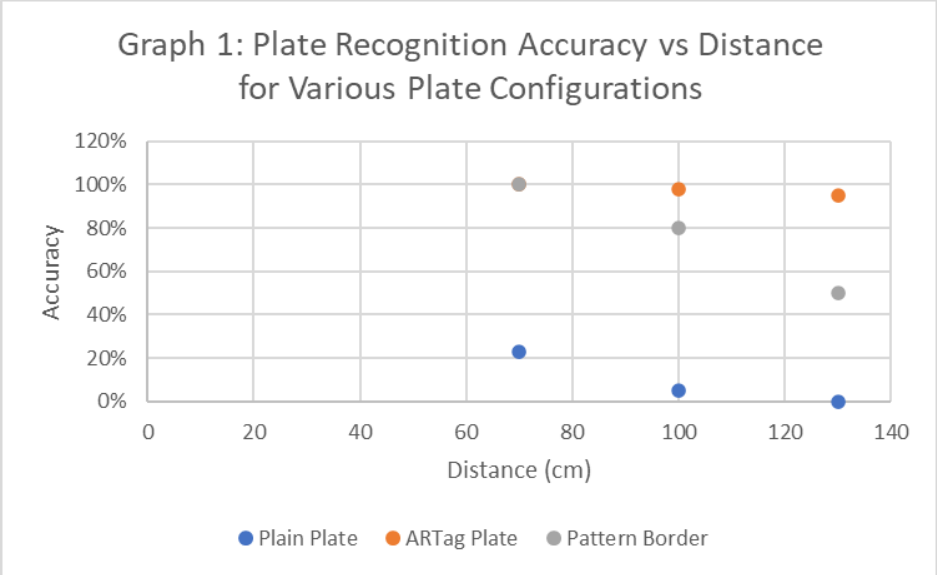
Table 1: Plate Configuration Features

| Type | Features |
|---|---|
| (a) | 17 |
| (b) | 90 |
| (c) | 127 |

From Table 1, we hypothesized that configuration type “c”, the one with the most features, would be the most reliable for all of our testing methods. Considering that features, most of the time, directly correspond to improvements in precision and recall metrics, metrics that are used internally for find_object_2d to detect objects.

We first tested straight-ahead detection of each of the configurations. Therefore, for each of the following tests, the robot was placed in-line with the plate but distance was varied, as seen in Table 2.

| Table 2: Plate Configuration Straight-Ahead Testing | | |
|---|---|---|
| Type | Distance (cm) | Accuracy |
| (a) | 70 | 23% |
| | 100 | 5% |
| | 130 | 0% |
| (b) | 70 | 100% |
| | 100 | 98% |
| | 130 | 95% |
| (c) | 70 | 100% |
| | 100 | 80% |
| | 130 | 50% |




From Graph 1, we can see that the Plain Plate configuration type "a" was not very good at maintaining the robot's attention. While configuration types "b" and "c" were far better. Interestingly the hypothesized superior type "c" was beat out by the type "b" configuration and offered superior accuracy at farther distances than the type "c" configuration.

This prompted further testing to determine the limiting angle that each plate configuration could be recognized. This testing limited distance to 100 (cm) while the angle was varied as seen in Table 3.

| Table 3: Plate Configuration Angle Testing | | |
|---|---|---|
| Type | Angle (°) | Accuracy |
| (a) | 0 | 23% |
| | 20 | 0% |
| | 40 | 0% |
| (b) | 0 | 100% |
| | 20 | 90% |
| | 40 | 80% |
| (c) | 0 | 100% |
| | 20 | 75% |
| | 40 | 40% |

Table 3 confirms the findings of Table 2 that configuration type “b” was superior to the configuration type “c” despite having fewer features and a more simplistic design. This prompted us to use only configuration type “c” throughout the remainder of our project.

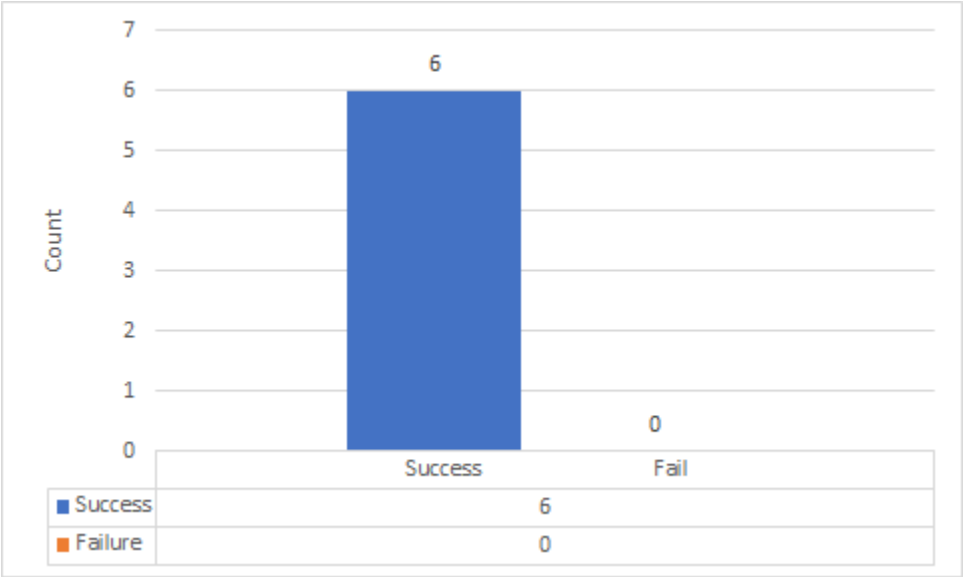
## 6.2 AR Tag Detection

To test AR Tag detection an AR-tag was positioned at various distances away from the camera in front of a noisy background and the camera’s detection and positioning accuracy was tested. Accuracy was maintained at all angles up to the 10 foot mark where afterwards, the detection accuracy diminished rapidly. Interestingly, AR Tag detection was limited to a square box in the center of the camera image instead of the entire camera’s FOV. But this detection was not standard when varying both the angle of the AR Tag and the distance the AR Tag had from the camera. In this case the fish-eye effect due to the lenses caused premature mis-detections when the AR-Tag was at extreme angles but less than 10 feet away.

## 6.3 Previous Data

Shown below are data about charging angle, charging successes and failures, and latency. This was discussed in the midterm report but since the hardware is the same, still applies here.

| Trial | Angle of Attack (°) | Docked Successfully? |
|---|---|---|
| 1 | 0 | Yes |
| 2 | 90 | No |
| 3 | 5 | Yes |
| 4 | 10 | Yes |
| 5 | 45 | No |
| 6 | 20 | No |




| Trial | Latency (s) | Connection Maintained? |
|---|---|---|
| 1 | 2.34 | Yes |
| 2 | 2.01 | Yes |
| 3 | 2.59 | Yes |
| 4 | 2.39 | Yes |

# 7 Conclusions and Future Work

For our final project, we successfully designed and developed a remote control ground vehicle using ROS as a message and signal distributor, but with additional perception capabilities for detection of AR tags and the charging dock. We already had our physical hardware set up from the midterm, so the main constraints were time and fully implementing the navigation stack and autonomy. While we were not successful in accomplishing this, we did implement most of the individual components necessary to do so.

We discussed our quantitative and qualitative conclusions from our tests in the Experimental Validation section above. There were many qualitative insights gained into the

operation of our robot, but the primary ones were about our perception metrics. We tested drogue detection at a distance of up to 130 cm (further not tested due to significantly decreased accuracy), and found that the AR tag and plate combination had the highest accuracy, maintaining 95% accuracy at 130 cm. The next best was the plate and pattern border, then finally plate alone, because there were numerous features in the first two tests with easily distinguishable features. This is why teams used AR tags in combination with the plate for the most accurate detection. As seen in the previous section, the AR tag-plate combination also had the greatest detection angle. We also found that the AR tag alone could be detected up to 10 feet away in approximately a conic volume with a 20° vertex angle. In our navigation and autonomy work, this allowed us to detect the AR tag from a reasonably far distance, navigate closer, then use detection of the plate to more accurately guide itself in.

Although we did not get the robot to be fully autonomous, we did have numerous successes. Our perception systems worked properly and we were able to view and detect the AR tags, charging plate, and lab space from sufficient distances and resolutions. We also used SLAM to create a map of the lab, even though the fidelity was not great. Navigation and autonomy were nearly complete, except for proper localization and the initial pathfinding. Overall, much of the system components worked though full system integration has yet to be realized.

Future work should involve the proper tuning of navigation parameters and remapping of the lab so that localization and the initial pathfinding can be done autonomously. Better cameras could also aid in creating a higher-fidelity map. A redesign of the Arduino motor control system to use a differential drive would also make converting the /move_base/cmd_vel velocity messages easier (compared to our current tank-style driving system).

# 8 Team Contributions

Technical
Julian - 34%
Avaneen - 33%
Sid - 33%

Effort
Julian - 95%
Avaneen - 95%
Sid - 100%