Semi-Autonomous iRobot Navigation

Group members:

Akef, Selim, *53241782*
da Cunha Vasco, Thiago, *57250771*
Dakin, Nicholas, *30958557*
Venkateshwaran, Akash, *36411338*
Youm, Daniel, *99238206*

ABSTRACT:
For our project, we originally set out to accomplish two goals. Our first goal was to find a way to navigate an iRobot to any reachable pose within a 2D grid environment, while our second goal was to program the iRobot to reach a pose while avoiding collisions with objects within its actual environment. While our first goal was accomplished almost instantly by our discovery of the pycreate2 library package, we did not achieve our second goal of setting up a semi-autonomously navigating iRobot due to issues with stereo camera calibration, and the resulting complication which arises when trying to convert a disparity map into a point cloud. Nevertheless, this report will go through the work we've been able to do on stereo camera calibration, communicating sensor and actuator signal between the iRobot and a processor, and simulating the iRobot in a 3D environment using Gazebo and ROS2.

KEYWORDS: iRobot, ROS2, Nav2

Contents

1 INTRODUCTION

With the prevalence of industrial robotics in the modern world, the growing need to equip robots with sophisticated mapping and pathfinding functionality arises from the dynamism, complexity, and diverse operational requirements of the environments which industrial robots can be applied to. These environments require autonomous systems to be capable of navigating with precision and agility while concurrently mapping their surroundings. The significance of such capabilities also encompasses other important features such as workplace safety, resource optimization, and adaptability to evolving operational tasks. However, engineering a robotic system to be able to operate in a complex and dynamic environment involves many challenges across several domains. Some of these challenges include sensing and actuation, formulating the environment dynamics, vision, path planning, and finally, integrating all of these moving pieces together.

In this project report, we outline what progress we've made in our attempt at creating an iRobot system which is capable of sensing and autonomously navigating a cluttered 3D environment. More specifically, we will touch on stereo camera calibration, communicating sensor and actuator signal between the iRobot and a processor, and simulating the iRobot in a 3D environment using Gazebo and ROS2.

1.1 Features of the iRobot

The iRobot Create 2 used for this project is a programmable robot based on the Roomba vacuum cleaner iRobot (2024). It consists of a circular body with two driving wheels on the bottom. The other critical components of the iRobot are its sensors, primarily its bumper sensor which detects collisions from obstacles, and its cliff sensors to avoid any sudden drops. Furthermore, we planned on mounting a stereo camera on top of the iRobot which would allow the iRobot to map its surroundings and avoid colliding with obstacles. Lastly, the iRobot has an internal processor which allows us to send control signals and receive sensor input through the iRobot Create 2 API.

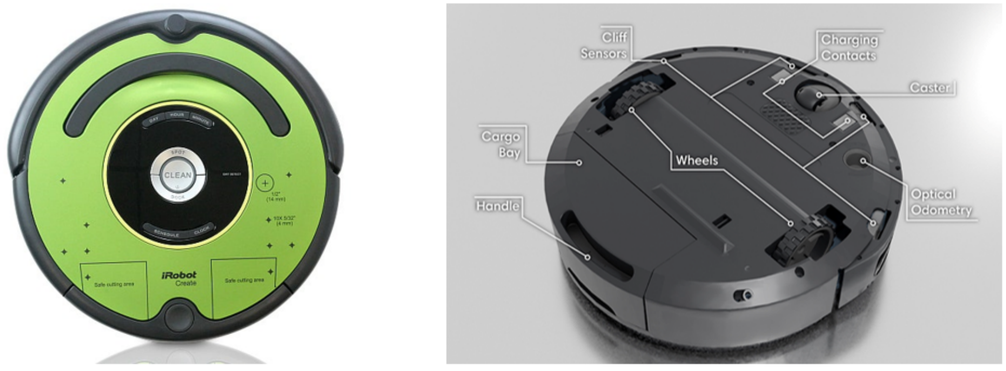

Figure 1: A diagram of the iRobot.

1.2 The stereo camera

We aimed to enable the iRobot to sense its environment by connecting it to a stereo camera (namely, the LI-OV580-STEREO). These are a type of camera with two or more lenses, which in our case is just two lenses. These get mounted together to create binocular vision, where the images of each lens can be combined to create a 3D image, which we use for range imaging and obtaining distance measurements. At each timestep, our system would use images taken by both the left and right camera and create a disparity map. Using further information about the camera, we should then be able to derive a point cloud with coordinates within the iRobot's coordinate frame, which we could then port over to NAV2.

2 METHODOLOGY

2.1 The iRobot control loop

Our original strategy for controlling the iRobot can be found in figure 2. One would begin by picking a point using Gazebo for the iRobot to travel to. Then NAV2 would subsequently lay out an approximately optimal trajectory which would follow this path. For every iteration of the outer-loop we calculate a new trajectory, whereas for every iteration of the inner-loop we apply PD control to make our iRobot travel a fixed trajectory. We exit the control loop once the iRobot's current position (x_0, y_0) is sufficiently close to the destination (x_d, y_d). The iRobot keeps track of its current position using SLAM.

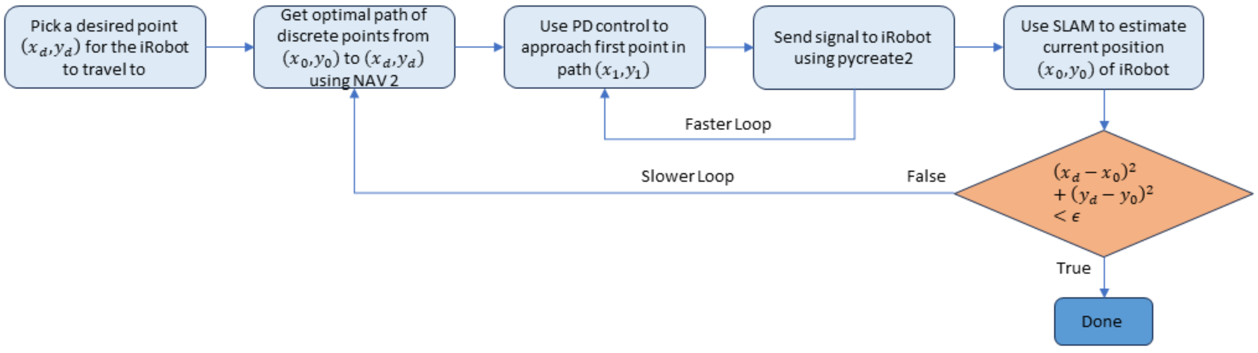
Pick a desired point (x_d, y_d) for the iRobot to travel to
Get optimal path of discrete points from (x_0, y_0) to (x_d, y_d) using NAV 2
Use PD control to approach first point in path (x_1, y_1)
Send signal to iRobot using pycreate2
Use SLAM to estimate current position (x_0, y_0) of iRobot
Faster Loop
Slower Loop
False
$(x_d - x_0)^2 + (y_d - y_0)^2 < \epsilon$
True
Done

Figure 2: The iRobot control loop.

2.2 Dynamics System of iRobot

We use the kinematic design of a similar robot derived by Abbasi et al. (2020) in order to model the dynamics of the iRobot.

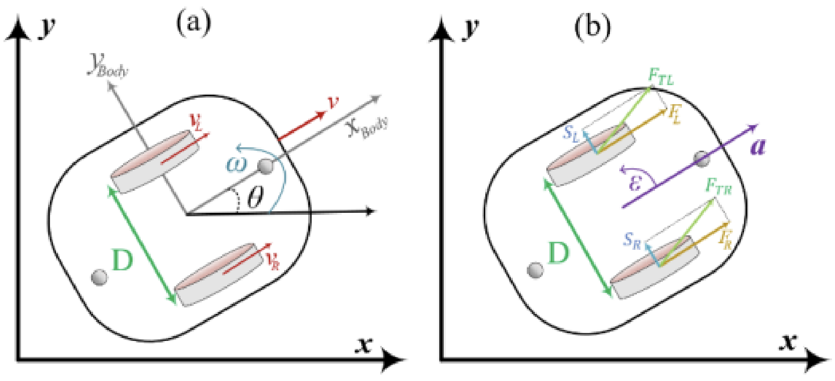

Figure 3: iRobot kinematics.

The iRobot utilizes two motors that drive individual wheels which allows for movement in front and backwards direction in addition to rotation of the robot. A kinematic diagram of the iRobot can be found in figure 3 above. We have that v_R and v_L correspond to the tangential velocity of the left and right wheel while ω is the angular velocity of the iRobot's body with respect to the global coordinate frame, about the center of mass of the iRobot. Furthermore, D is the distance between the two wheels and r is the radius of the individual wheels. The robot's equations of motion are controlled by the angular velocity of each wheel. The relationship between translational and tangential velocities are given by

$$\begin{aligned} v_R &= r * \omega_R \\ v_L &= r * \omega_L \\ v &= \frac{(v_L + v_R)}{2} \\ \omega &= \frac{(v_L - v_R)}{D}. \end{aligned} \tag{1}$$

Let x, y, and θ be the coordinates and rotation of the iRobot in the global coordinate frame. Then we have that

$$\begin{aligned} \dot{\theta}(t) &= \omega(t) \\ \dot{x}(t) &= v(t) * \cos(\theta(t)) \\ \dot{y}(t) &= v(t) * \sin(\theta(t)). \end{aligned} \tag{2}$$

It's important to note that we're assuming the environment in which the iRobot lives in is entirely flat, and provides constant friction. Further, we assume that the iRobot's wheels do not have a moment of intertia which is significant.

Let F_L and F_R be the tangent forces applied to the left and right wheels from the wheels acting on the floor due to a change in velocity (acceleration or breaking). We can further derive the robot's translational acceleration to be equal to $a(t) = \frac{(F_L + F_R)}{m}$, and the angular acceleration to be equal to $\epsilon(t) = \frac{D}{2*I}(F_L - F_R)$ where J is the intertia coefficient of the iRobot. Finally, we can use Lagrangian mechanics to derive the equations of motion for the iRobot which are

$$\begin{aligned} u_L &= J * \epsilon_L(t) + F_L * \omega_L(t) + F_L * r \\ u_R &= J * \epsilon_R(t) + F_R * \omega_R(t) + F_R * r \end{aligned} \tag{3}$$

where ϵ_L and ϵ_R are the angular velocities of the left and right wheels respectively. We can now define a linear state-space dynamics model with a state vector defined as $\mathbf{x} = [v\ \omega\ \omega_L\ \omega_R]^T$, with its time derivative being $\dot{\mathbf{x}} = [a\ \epsilon\ \epsilon_L\ \epsilon_R]^T$. Lastly, define the control input to be $\mathbf{u} = [F_L\ F_R\ U_L\ U_R]^T$. We can now rewrite our dynamics as

$$\dot{\mathbf{x}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{F}{J} & 0 \\ 0 & 0 & 0 & -\frac{F}{J} \end{bmatrix} \mathbf{x} + \begin{bmatrix} 1/m & 1/m & 0 & 0 \\ \frac{-D}{2J} & \frac{D}{2J} & 0 & 0 \\ -\frac{r}{J} & 0 & \frac{1}{J} & 0 \\ 0 & -\frac{r}{J} & 0 & \frac{1}{J} \end{bmatrix} \mathbf{u}. \tag{4}$$

Originally we planned to apply a control algorithm on top of the iRobot's dynamics model in order to get it to follow a trajectory planned by NAV2. Unfortunately we never reached this part.

2.3 Stereo Camera and Image Processing

We aimed to enable the iRobot to detect objects in its environment by connecting it to a stereo camera (namely, the LI-OV580-STEREO). At each timestep, our system would use images taken by both the left and right camera and create a disparity map. Using further information about the camera, we should then be able to derive a point cloud with coordinates within the iRobot's coordinate frame, which we could then port over to NAV2.

2.3.1 Rectification and calibration

In order to obtain accurate point cloud information and a cleaner disparity map, it was necessary for us to calibrate the stereo camera. The procedure for doing so involves taking dozens of pictures of a checkerboard at varying poses and distances from the camera, and using the two sets of pictures in order to estimate several parameter matrices. In order to accomplish this we used the Python StereoVision library which is mostly based off of OpenCV. For stereo camera calibration in particular, the StereoVision library more or less just uses the openCV methods cv2.stereoRectify() and cv2.stereoCalibrate(). One of the matrices computed through rectification is the 4x4 perspective transformation matrix Q, which is crucial for obtaining a point cloud downstream. Some other matrices that are estimated include

1. cameraMatrix1 & cameraMatrix2 - Input/output camera intrinsic matrix for both cameras
2. distCoeffs1 & distCoeffs2 - Input/output vector of distortion coefficients for both cameras
3. R & T - A rotation matrix and translation vector for mapping points in the first camera's coordinate system to the second camera's coordinate system
4. E - The essential matrix, which describes that geometrically relates the same corresponding point found within the two images taken by cameras 1 and 2
5. F - The fundamental matrix, which is similar to the essential matrix but in pixel coordinates
6. R1 & R2 - Rectification transform matrices for images from cameras 1 and 2 respectively. These matrices map points in an unrectified camera's coordinate system to the rectified camera's coordinate system
7. P1 & P2 - Projection matrices for cameras 1 and 2 respectively. Projects points given in rectified camera's coordinate system into the rectified camera's image.

2.3.2 Disparity map and preprocessing

We used the StereoSGBM algorithm to compute a disparity map using two grayscale images given by the stereo camera. StereoSGBM works by matching uniformly-sized blocks of pixels between two cameras, and computes the disparity of a given block by measuring the difference in location of that block within the two corresponding image arrays. In order to get this to work we had to do a fair amount of hyperparameter tuning in order to get a smooth map which presents closer objects as having higher disparity values. By far the greatest improvement in the quality of the disparity map came from applying SGBM to a downsampled version of the two images, and upsampling the disparity map to the original image size. Lastly, a Gaussian blur was applied to the disparity map which seemed to reduce speckle noise. We attempted to use erosion and dilation in order to address the issue of speckle noise, but it didn't appear to be effective. We visualized the difference in disparity maps before rectification/preprocessing and after in figures 4 and 14 respectively.

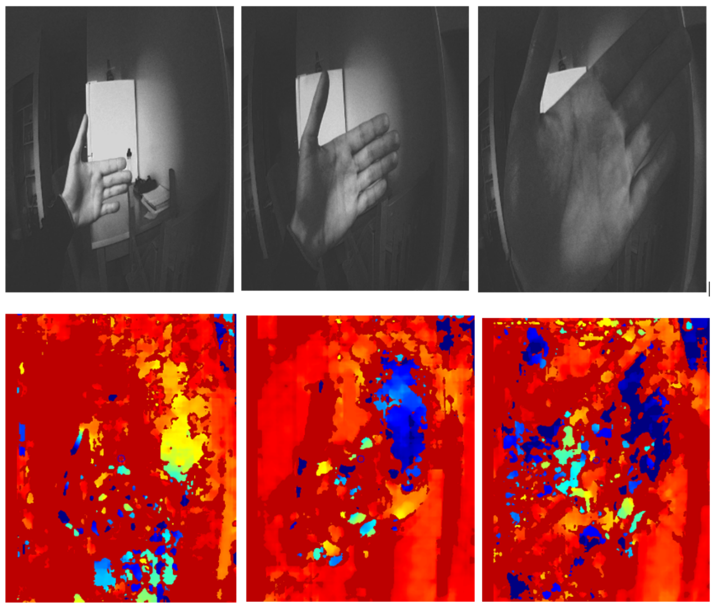
Figure 4: The disparity map prior to camera rectification and preprocessing.

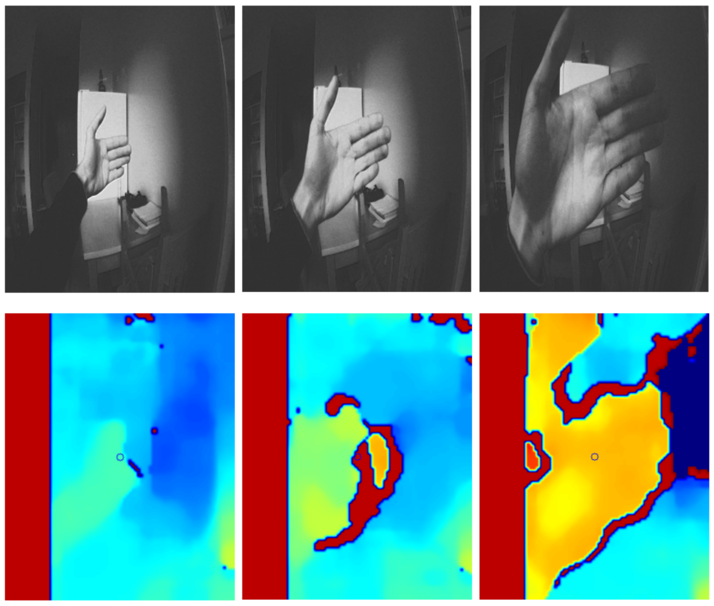
Figure 5: The disparity map after camera rectification and preprocessing.

2.3.3 Point cloud retrieval and failure

Calibrating the stereo camera turned out to be incredibly unwieldy; we took 100 checkerboard pictures with the stereo camera (per each lens), and the calibration algorithm only accepted a little over a dozen pictures out of the 100; just by looking at the pictures one could sense that the image quality of the stereo camera was poor and extremely sensitive to light intensity. The issue of poor calibration spilled over into our aim of using the disparity map to generate a point cloud, since the calibration scheme was unable to find an accurate estimate of the perspective transformation matrix. This leads to very unrealistic point cloud estimates. The creator of the repository understood this to be a common issue, and so provided a "generic" perspective transformation matrix matrix of the form

$$Q = \begin{bmatrix} 1 & 0 & 0 & -\frac{1}{2} * \text{width} \\ 0 & -1 & 0 & \frac{1}{2} * \text{height} \\ 0 & 0 & 0 & -\text{focal length} \\ 0 & 0 & 1 & 0 \end{bmatrix}, \tag{5}$$

where “width” and “height” refer to the number of pixels along a row and column of an image respectively. This results in point cloud estimates which now lie in a realistic range relative to the camera coordinate system, but which at best yields mediocre accuracy, and only for objects that are within 40 cm of the camera. We did not figure out how to port point cloud data over to NAV2.

2.4 pycreate2

The pycreate2 library Python Package Index (2024) is a python package designed specifically for use and control of the iRobot via an open interface protocol. It works by establishing a serial connection between the robot and the computer via USB and allows the user to communicate with the robot by controlling its movements, reading sensor data, and program predefined operations for the robot to follow, all done using python scripts. Some of the functionality it offers includes moving and stopping the robot, rotation by specifying wheel speeds for differential driving as well as reading data from the various infrared and bump sensors located on board.

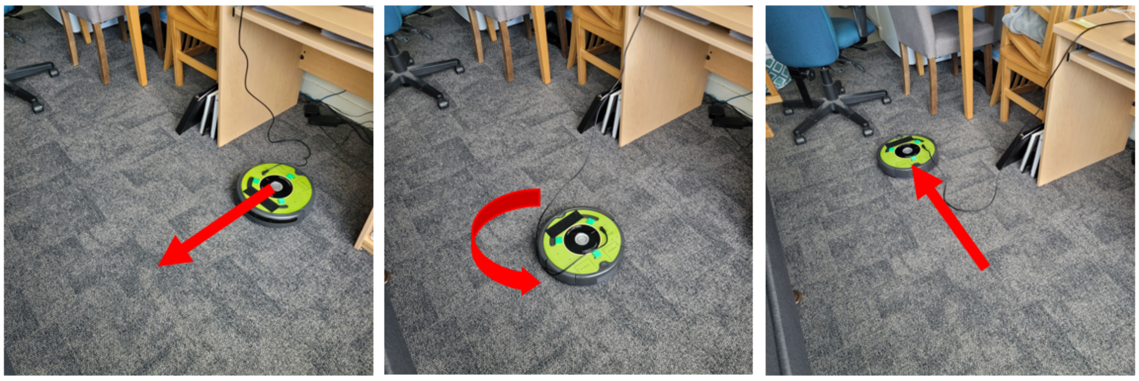

Figure 6: The iRobot traverses a simple L-shaped path.

2.5 Simulation of iRobot

In this section, we describe the methodology used for simulating the iRobot using the Gazebo simulation environment. Gazebo allows for the creation of complex three-dimensional scenarios on a computer, featuring robots, obstacles, and various other entities. It incorporates a physics engine to simulate real-world physical phenomena such as lighting, gravity, and inertia. This capability enables time-efficient evaluation and testing of robotic behaviors in challenging or hazardous conditions without risking damage to the physical robot.

Gazebo functions as a 3D simulation platform, whereas ROS operates as the interfacing system for robotic control. This methodology describes constructing a virtual environment wherein the iRobot will operate (Gazebo world), as well as developing a precise model of the iRobot. Upon completion of these models, individual robot joints can be controlled through teleoperation using ROS nodes, which transmit the necessary commands to the robot. Additionally, the entire process is visualized and monitored using a visualization tool known as RViz. Fig. 7 shows the framework we used for simulation in this project.

2.5.1 iRobot world

Figure 8 depicts the virtual environment constructed for the simulation. The world consists of an enclosed space, with walls serving as obstacles. The robot is initially spawned at a designated start state, as illustrated in the figure. Subsequently, a goal state is established, which directs the robot toward its destination.

2.5.2 iRobot model

The Universal Robotic Description Format (URDF) is an XML file format used in ROS to describe the various components of a robot. We created a URDF file comprising links and joints, constructing a model of an iRobot. The chassis is represented by a cylinder with appropriate dimensions. Two driving wheels and two caster wheels are then attached to the

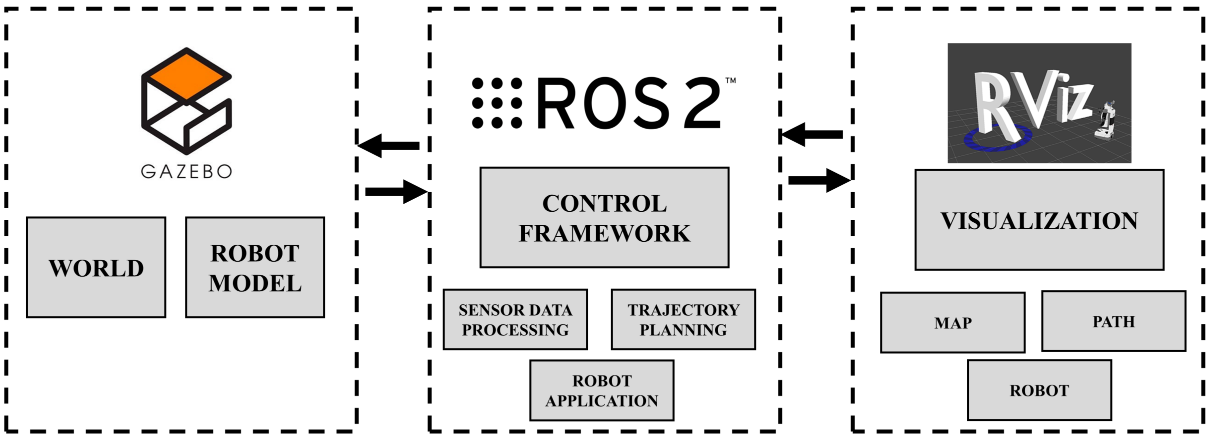

Figure 7: Simulation framework for iRobot.

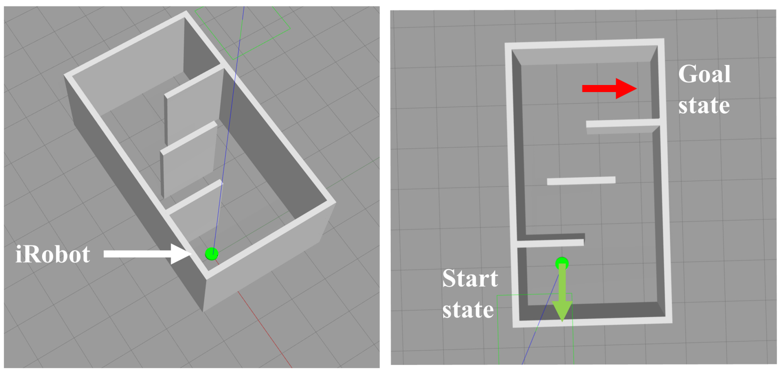

Figure 8: iRobot test world with the spawned robot.

chassis, with distances and radii approximating those of the actual iRobot. Additionally, a camera is defined as a simple cuboid affixed to the front of the chassis. The weights of each component are estimated and integrated into the model. Fig. 9 shows the model we used in this project.

We use the following Gazebo plugins for the sensors: (i) libgazebo_ros_diff_drive for the differential drive, and (ii) libgazebo_ros_camera for the stereo camera, which captures the depth image. Detailed configurations of these plugins can be found in the appendix. The differential drive plugin receives cmd_vel commands, allowing the robot to navigate within the virtual environment. The camera plugin publishes both raw images and depth images in the PointCloud2 format. These outputs are further processed for navigation purposes.

Odometry is a method used to estimate the position and orientation of a differential drive robot based on the movement of its wheels. The odometry is calculated using the Gazebo plugin that leverages the wheel velocities to compute the robot's change in position and orientation.

Assume the left and right wheels have angular velocities ω_L and ω_R, respectively. The distances covered by each wheel during a time interval Δt can be computed as follows:

$$\Delta d_L = R\omega_L \Delta t \tag{6}$$

$$\Delta d_R = R\omega_R \Delta t \tag{7}$$

where R is the radius of the wheels.

The linear velocity v and angular velocity ω of the robot can then be calculated using the following equations:

$$v = \frac{\Delta d_L + \Delta d_R}{2\Delta t} \tag{8}$$

$$\omega = \frac{\Delta d_R - \Delta d_L}{b\Delta t} \tag{9}$$

where b is the distance between the two wheels.

From these velocities, the robot's change in position $(\Delta x, \Delta y)$ and change in orientation $\Delta\theta$ can be calculated:

$$\Delta\theta = \omega\Delta t \tag{10}$$

$$\Delta x = v\cos(\theta + \frac{\Delta\theta}{2})\Delta t \tag{11}$$

$$\Delta y = v\sin(\theta + \frac{\Delta\theta}{2})\Delta t \tag{12}$$

These incremental changes can be added to the robot's current state (x, y, θ) to update its position and orientation:

$$x_{\text{new}} = x_{\text{old}} + \Delta x \tag{13}$$

$$y_{\text{new}} = y_{\text{old}} + \Delta y \tag{14}$$

$$\theta_{\text{new}} = \theta_{\text{old}} + \Delta\theta \tag{15}$$

Thus, by continuously tracking wheel rotations and computing these incremental changes, odometry provides an estimate of the robot's trajectory over time.

The following section details the navigation of this model within the virtual environment we have made.

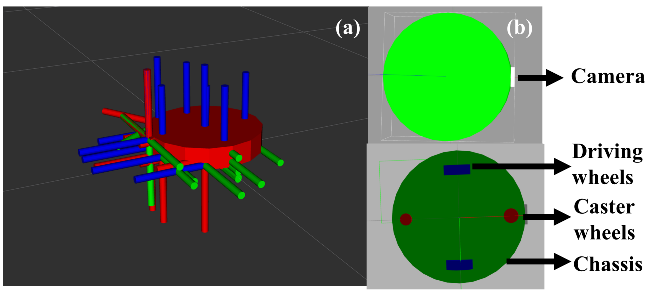

Figure 9: (a) iRobot URDF, and (b) iRobot.

2.6 Navigation for iRobot

The ROS 2 Navigation Stack is a comprehensive package designed to facilitate autonomous navigation for mobile robots. It provides tools and algorithms that enable robots to move seamlessly within complex environments, avoiding obstacles and reaching designated goals. In this project, we used Nav2 package using the depth camera sensor.

The navigation process for the iRobot in the test environment consists of the following steps:

1. The robot is manually driven in the test world to generate a map of the environment. This map is created using the OctoMap package Hornung et al. (2013), representing the environment's geometry, and is stored locally for future reference.

2. Once the map is generated, the ROS 2 Navigation Stack (Nav2) is utilized, incorporating the Adaptive Monte Carlo Localization (AMCL) algorithm Xiaoyu et al. (2018) to localize the robot within the map.

3. With the robot's position established, a goal state is defined. The Navigation Stack ROS 2 Navigation Project (2024) then employs Dijkstra's algorithm to calculate an optimal path from the current position to the specified goal, allowing the robot to navigate effectively.

2.6.1 Mapping using Octomap

OctoMap Hornung et al. (2013) is a 3D mapping framework for robotic navigation and exploration. It models occupied and free spaces, with unknown areas implicitly encoded. The map can be continuously updated, incorporating new sensor data to adapt to changes in the environment and allowing for contributions from multiple robots. OctoMap also dynamically expands as needed, offering multi-resolution capabilities for both coarse and fine-grained planning and visualization. Additionally, it efficiently stores map data in memory and on disk, supporting compressed files for easy exchange between robots.

In this project, we use the depth camera as our sensor input in the format of PointCloud2 and create 3D voxels based on the height of the points in the cloud. To prevent the floor from being considered an obstacle, we filter out all points with a zero Z-axis value relative to the robot's base frame. Additionally, we remove points from the point cloud that are located more than 2 meters away from the camera, effectively filtering out distant obstacles.

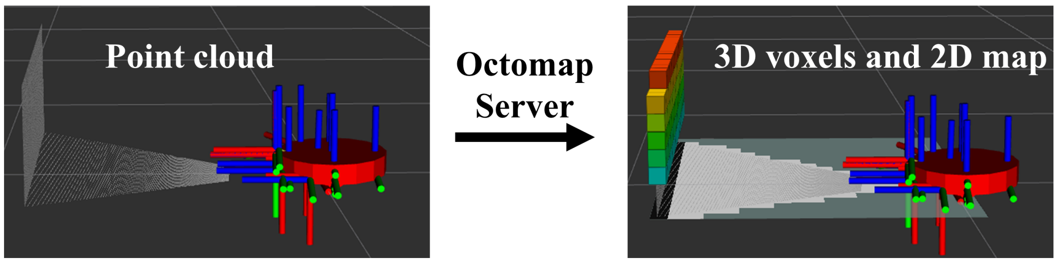

Figure 10: Mapping using Octomap.

2.6.2 Localization using AMCL

Adaptive Monvte Carlo Localization (AMCL) Xiaoyu et al. (2018) is a probabilistic localization method widely used in robotics, particularly in ROS. AMCL works by integrating point cloud sensor data with an existing map of the environment, using a particle filter approach to estimate the robot's position and orientation. The algorithm continuously updates its estimates by sampling particles and weighting them based on their consistency with the observed sensor data. This process allows AMCL to provide accurate and robust localization even in dynamic environments. By effectively combining sensor readings and map information, AMCL ensures precise positioning, facilitating autonomous navigation, mapping, and various robotic tasks. Fig. 11 demonstrates this.

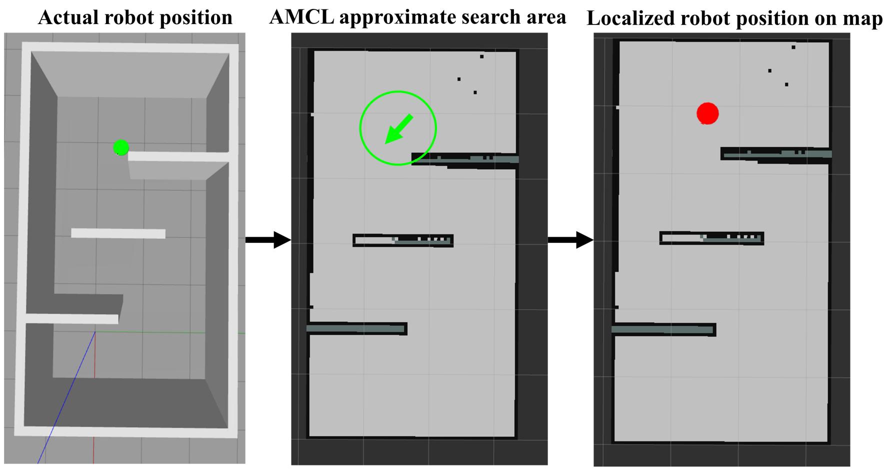

Figure 11: Localization demonstrated using AMCL.

2.6.3 Navigation using Nav2

Once localized, the navigation stack publishes global and local cost maps. The global cost map is generated using the pre-existing static map, with parameters such as robot_radius and buffer layers adjusted to suit our needs. The navigation stack utilizes this cost map to compute an optimal path using Dijkstra's algorithm. Additionally, a local cost map is generated, which acts as a controller, guiding the robot to follow the global path.

3 RESULTS

The results encompass the stereo camera image processing, mapping of the test world in simulation, and navigation of the iRobot.

As outlined in the methodology, the mapping is performed using the OctoMap package, with manual control via keyboard. Fig. 12 shows the resulting map of the test world. The 3D voxels are accurately positioned according to the received point cloud data. A 2D map is generated as an image, where white pixels denote free space and black pixels indicate obstacles. This map is subsequently exported as YAML and PGM files, which are utilized by the navigation stack.

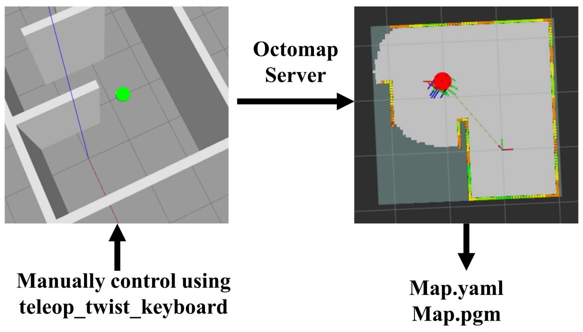

Figure 12: Mapping of test environment using iRobot. The mapping video can be accessed here.

Subsequently, we launch the navigation stack, providing the map YAML file as a parameter. This initiates a series of nodes responsible for publishing both the global and local cost maps. Additionally, the Behavior Tree server is activated, which determines the actions for the robot. Fig. 13 shows the resulting global and local maps.

However, we encountered an issue with completing the stack. Despite the navigation stack functioning correctly, providing the goal pose causes the Behavior Tree Navigator to fail and crash with a segmentation error. We used the GDB utility to monitor the logs, but a solution to this problem has yet to be identified.

Furthermore, we managed to do the best we could in obtaining point-cloud data from the stereo camera using OpenCV, although it was quite poor in quality. And lastly, we managed to send signals and gather sensor data from the iRobot using pycreate2.

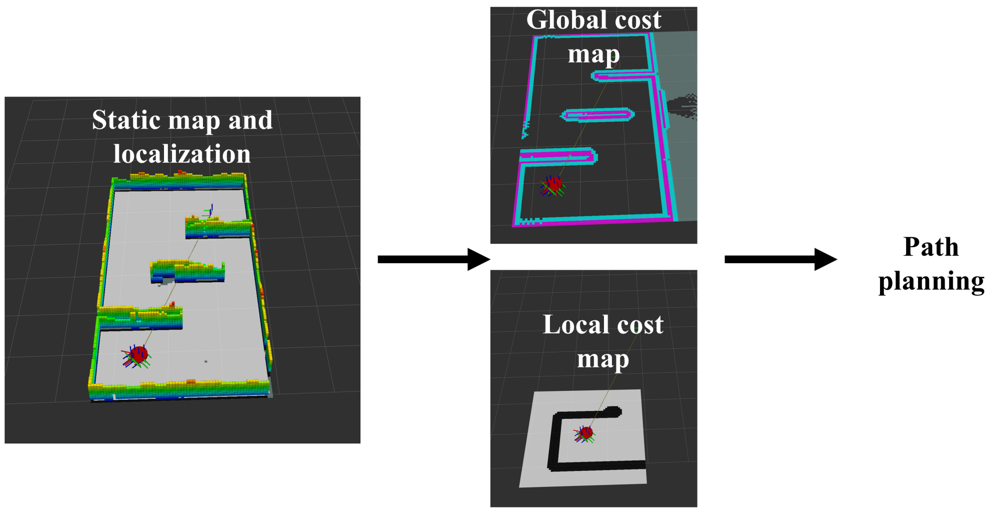

Figure 13: iRobot navigation.

4 DISCUSSION AND CONCLUSION

This project on iRobot's semi-autonomous navigation was largely successful. We achieved numerous milestones, yet encountered challenges both in our code implementation, image processing and overall approach.

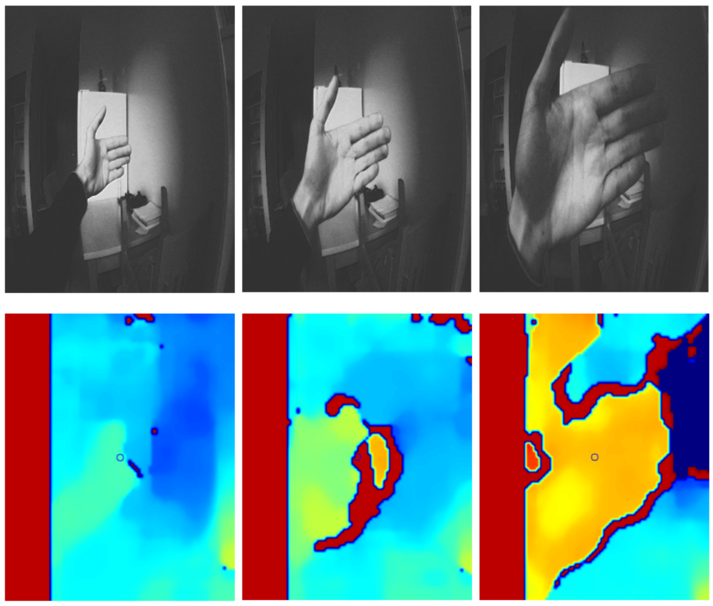
Figure 14: The disparity map after camera rectification and preprocessing.

4.1 Achievements:

1. **Dockerized Workspace:** We created a Dockerized workspace and posted it on GitHub (Link) for public access. This workspace can be run on any system, with setup instructions provided in the README file.
2. **iRobot URDF Model:** We developed an iRobot model in URDF, incorporating the necessary links, joints, and sensors through Gazebo plugins.
3. **Custom Test World:** We designed a custom test world to run our simulations and implement our algorithms.
4. **Mapping with OctoMap:** We coded custom launch files for mapping the environment using OctoMap, utilizing point cloud data to represent the environment.
5. **Navigation Stack Parameters:** We tuned and modified the parameters of the navigation stack to suit our robot, replacing conventional LaserScan data with point cloud data from the camera.
6. **Navigation Launch File:** We developed a launch file for the navigation of the iRobot within a static map, enabling autonomous movement along pre-defined paths.
7. **Controlling the iRobot:** We successfully used the pycreate2 package for controlling the iRobot, including testing its functionality and sensors.
8. **Stereo Camera:** We managed to significantly improve the disparity map quality for the stereo camera over its initial performance.
9. **Model Dynamics:** We found a kinematics model which accurately models the iRobot and which could be used for control.

4.2 Shortcomings:

1. **Camera Quality:** We encountered difficulties translating the camera's images into point cloud data, primarily due to the camera's quality. The stereo images were of low quality (including noise and distorted images), resulting in unwieldy calibration and unusable point cloud data.
2. **Bumper Sensor as Point Cloud:** Initially, we attempted to create a point cloud using the bumper sensor, implementing a node that converts the `bumper_state` message into a point cloud for mapping the environment. However, we realized this approach was inadequate for localization and abandoned it.
3. **Raspberry Pi Issues:** We faced difficulties with the iRobot's Raspberry Pi, which wouldn't boot up. After buying a new SD card, new cables, and accessories, we still didn't couldn't manage to install the OS onto the Pi and get it to run. Consequently, we were unable to design a fully wireless autonomous system as envisioned. This issue was overcome by connecting the robot via a USB connection to a portable computer.

4. **Lidar Sensor Access:** We didn't have access to a LiDAR sensor, which would have significantly eased the mapping and localization processes.

4.3 Alternative Approaches and Suggestions:

1. **ORB-SLAM with a Monocular Camera:** Rather than relying on the stereo camera, we could use a monocular camera, such as a smartphone camera, to implement ORB-SLAM (Oriented FAST and Rotated BRIEF Simultaneous Localization and Mapping) Mur-Artal et al. (2015). This approach would enable mapping and localization using a single camera sensor, simplifying the hardware setup.
2. **LiDAR and Camera Integration:** If access to a LiDAR sensor is available, it can be employed for both localization and mapping, providing a standardized and reliable solution. Furthermore, incorporating a camera in addition to the LiDAR would enhance functionality, allowing for a more comprehensive perception of the environment.

5 STATEMENT OF CONTRIBUTION

Thiago: Stereo camera setup and results, dynamics model, control loop, pycreate2, pictures and diagrams, raspberryPi troubleshooting.

Akash: Developed the irobot_workspace with Docker; Created iRobot urdf model; implemented depth camera, bumper sensor, and motor driver; created Gazebo custom world; developed code for mapping using octomap; implemented localization using point cloud data and navigation using Nav2; Made video simulation of mapping; Wrote report sections including the simulation, navigation, results, and discussion and conclusion; Made figures for methodology

Selim: iRobot setup and control using pycreate2, stereo camera setup.

Nicholas: Various sections of the report such as stereo camera calibration, iRobot features, results, and discussion.

Daniel: Dynamics model, project report.

6 APPENDIX I - Running iRobot Simulation

The code for our project can be accessed on our GitHub workspace (Link). Our workspace is dockerized, allowing you to create an image and build it using the provided Dockerfile. This installs ROS 2, the navigation stack, and all necessary packages. The installation instructions can be found in our documentation available at this Google Drive link.

To launch the simulation, use the following command:

```
ros2 launch create_bringup create_2_gazebo.launch.xml
```

This launch file first starts the Gazebo simulation with the test world passed as an argument. Next, it runs the `robot_state_publisher` and `joint_state_publisher`, which publish the transformation matrices of the robot. The robot is then spawned into the world. Finally, RViz2 is launched to visualize all components.

7 APPENDIX II - Running iRobot OctoMapping

This section outlines the steps for mapping the environment using the iRobot. Ensure that the iRobot simulation is running as described in the previous section. Next, clone the OctoMap server package into the `src` directory of the workspace and build it. Then, launch the OctoMap server using our custom launch file:

```
ros2 launch create_bringup create_2_octomap.launch.py
```

This launch file ensures that the point cloud topic names are consistent and that topic remapping is performed for convenience. To map the entire environment, run the following command to control the robot:

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

Once the map is complete, we use another custom launch file to save the map as YAML and PGM files using the `nav_map_server`. Since the OctoMap saver node only stores 3D voxels as a `.bt` file, we created our own launch file for saving the map. This can be launched with the following command:

```
ros2 launch create_bringup create_2_savemap.launch.py
```

8 APPENDIX III - OpenCV Scripts for Stereo Camera

The Python scripts for calibrating and retrieving a disparity map on a real-time stereo camera feed can be found at

`https://github.com/Thiagodcv/stereo-camera`

Please note that this implementation is partly based off of

`https://github.com/Matchstic/depthmapper/tree/main`

References

Abbasi, A., MahmoudZadeh, S., Yazdani, A., and Moshayedi, A. J. (2020). Feasibility assessment of a cost-effective two-wheel kian-i mobile robot for autonomous navigation.

Hornung, A., Wurm, K. M., Bennewitz, M., Stachniss, C., and Burgard, W. (2013). OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*. Software available at `https://octomap.github.io`.

iRobot (2024). iRobot Create 2: Programmable Robot. Accessed April 28, 2024.

Mur-Artal, R., Montiel, J. M. M., and Tardos, J. D. (2015). Orb-slam: a versatile and accurate monocular slam system. *IEEE transactions on robotics*, 31(5):1147–1163.

Python Package Index (2024). pycreate2: Python API for the iRobot Create 2. Accessed April 28, 2024.

ROS 2 Navigation Project (2024). ROS 2 Navigation Documentation. Accessed April 28, 2024.

Xiaoyu, W., Caihong, L., Li, S., Ning, Z., and Hao, F. (2018). On adaptive monte carlo localization algorithm for the mobile robot based on ros. In *2018 37th Chinese Control Conference (CCC)*, pages 5207–5212. IEEE.