WPI

Worcester Polytechnic Institute

Major Qualifying Project

Pedestrian Avoidance for Indoor Robots

Authors:

Jialin Song

Yichi Xu

Zhongchuan Xu

Lile Zhang

Advisors:

Prof. Xinming Huang

Prof. Jianyu Liang

Abstract

The goal of this project is to design and implement a pedestrian detection and avoidance solution for indoor robots. The team utilized a TurtleBot robot running ROS, with onboard sensors of RealSense Camera and Velodyne Lidar. The robot system is able to maneuver autonomously in an indoor environment, recognizing pedestrians and automatically generating new routes to avoid the moving pedestrians. The team conducted a number of experiments to evaluate the functionality and reliability of the prototype system. The project will create a safer environment for human robot interactions.

Acknowledgements

Our team would like to thank Professor Xinming Huang, Professor Jianyu Liang, and Ozan Akyildiz for their supports and guidance in this project.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Problems Defined

Today an increasing number of manual jobs are, completely or partially, replaced by robots such as the mobile robots at Amazon's fulfillment centers. After each warehouse receives online orders, robots are programmed to pick up ordered products from designated locations and move them to the distribution stations. As the robots move at high speed within the warehouse, there is a major safe concern that the robots may hit workers. The project aims to enhance the functionality of these robots by allowing them to detect and subsequently avoid collision with pedestrians.

1.2 Project Statement

The goal of this project is to implement pedestrian detection and avoidance on an indoor robot, conduct the test on the prototype and provide recommendations.

1.3 Summary

As an experimental prototype, the team uses a TurtleBot2 to imitate a warehouse robot and test it in the hallways to simulate the real environment in the warehouse. The focus of this project is to test the functionality of sensors, TurtleBot and algorithms involved in pedestrian and detection and avoidance.

1.4 Additional Requirements

This is an interdisciplinary MQP project. Chapters 1 through 7 are the main report. Chapter 8 is to meet the additional requirements in computer science. Chapter 9 is to meet the additional requirements in mechanical engineering.

2 Background

2.1 Current Status of Warehouse Robots

This section introduces the warehouse robots employed at Amazon fulfillment centers, the corresponding international regulations, and the existing safety measures to prevent incidents occurring on human associates.

2.1.1 Warehouse Robots in Amazon Fulfillment Centers

As a leader of warehouse robots, Amazon Robotics has developed autonomous robots that could work collaboratively with humans since 2012. In 2019, out of 175 fulfillment centers with working warehouse robots were built globally, 26 centers achieved the collaboration of humans and robots [1]. The majority of warehouse robots in the centers are autonomous mobile robots called "drive units," which are 2 feet by 2.5 feet in dimension. By scanning 2D barcodes located on the floor, "drive units" calculates an efficient path to carry the required pallets to humans by moving horizontally or vertically [2].

2.1.2 Safety Standards of Warehouse Robots

According to ISO, the International Organization for Standardization, collaborative robotics was defined as the autonomous robots that "share the same workspace with humans." Therefore, the robot units that work collaboratively with humans in fulfillment centers ensemble this definition. In ISO/TS 15066 established in 2016, safety requirements for collaborative robotics were explicitly presented. Multiple measurements were considered, including limitations of power and force applied to humans, the maximum speed of robots, the minimum separation distance between robots and humans, and avoidance of protrusions utilized on robot bodies [3].

2.1.3 Current Approach to Improve Safety of Associates

To improve the associates' safety when working with robots, Amazon Robotics created the "Robotic Tech Vest," a wearable technology that allowed the robots to avoid collisions by recognizing the associates, calculating their movements, and keeping distance [4]. However, as a critical concern in the collaboration of associates and robots, the safety of associates requires more attention. According to Raz Osman, a Senior Health and Safety Manager at

Amazon, vision technology would possibly be implemented to lower the risks of collisions and injuries [5].

2.2 Sensors

"Sensing and intelligent perception in robotic applications are crucial because many essential features greatly depend on the performance of sensors that provide critical data to these systems." [6] In this project, the ability to detect the environment, the ability to locate itself in the map, the ability to follow the path generated by algorithms are all closely related to the sensors.

2.2.1 LiDAR

"LiDAR," the acronym of Light Detection and Ranging, is a range measurement approach with pulsed lasers [7]. Lidar uses electromagnetic waves in the optical and infrared wavelengths. It is an active sensor, sending out an electromagnetic wave and receiving the reflected signal. It uses a much shorter wavelength compared to microwave radar, which results in higher angular resolution and better accuracy [8]. In robotics, this sensor provides real-time data for map generation, object monitoring and detection, and localization.

2.2.2 Camera

Camera is one of the most wide-known sensors nowadays because of its inexpensive price and versatile fields of implementation. In robotics, cameras are broadly used for object recognition and depth sensing. In pedestrian avoidance of robots, for example, camera provides inputs for human detection and distance estimation, facilitating the robot to avoid collisions with pedestrians.

2.3 Algorithm

"Algorithm is the procedure for addressing the task as operationalized: steps for aggregating those assigned values efficiently or making the matches rapidly" [9]. For pedestrian avoidance, a path-planning algorithm is needed for the robot to reach the destination; an object recognition algorithm is needed to identify the pedestrians; an obstacle avoidance algorithm is needed to navigate around the obstacle.

2.3.1 Path Planning Algorithm

Path planning algorithms generate a geometric path, from an initial to a final point, passing through pre-defined waypoints, either in the joint space or in the operating space of the robot, while trajectory-planning algorithms take a given geometric path and endow it with the time information [10]. There are multiple popular path planning algorithms for robot navigation such as Dijkstra, A*, and greedy algorithms.

2.3.2 Object Recognition Algorithm

"In computer vision, deep learning based object recognition models have become more and more influential in recent years" [11]. Object recognition allows a robot to analyze and classify designated objects through the images from the camera. The speed and accuracy of this algorithm affects the ability of pedestrian avoidance of a robot.

2.3.3 Object Avoidance Algorithm

Obstacle avoidance is one of the essential tasks in local path planning, which guarantees human and vehicle safety. Though multiple theoretical approaches are brought up from different researchers, most of which failed to perform accurately in real systems. In robotics, an object avoidance algorithm includes object recognition, detouring path generation, and new navigation path update [12].

2.4 Existing Robots with Similar Functionality

In the field of autonomous mobile robotics, researchers have developed multiple collision-free navigation approaches. Therefore, various types of robotics contain the pedestrian avoidance functionality as a rudimentary basic [13]. Commercial robotics such as Spot from BostonDynamics can avoid obstacles in 360 degrees with its stereo cameras [14]; robot vacuum cleaner Roborock S6 MaxV applies machine learning technology to recognize and avoid obstacles [15]. Vehicles such as Audi A8 and Tesla Model 3 contain collision avoidance system to detect the surrounding traffic and assist drivers to take proper reactions [16][17]. ABB provides collision detection option in its controller software, RobotWare, for certain industrial robotics to protect both the robot and its work pieces [18].

3 Methodology

3.1 System Overview

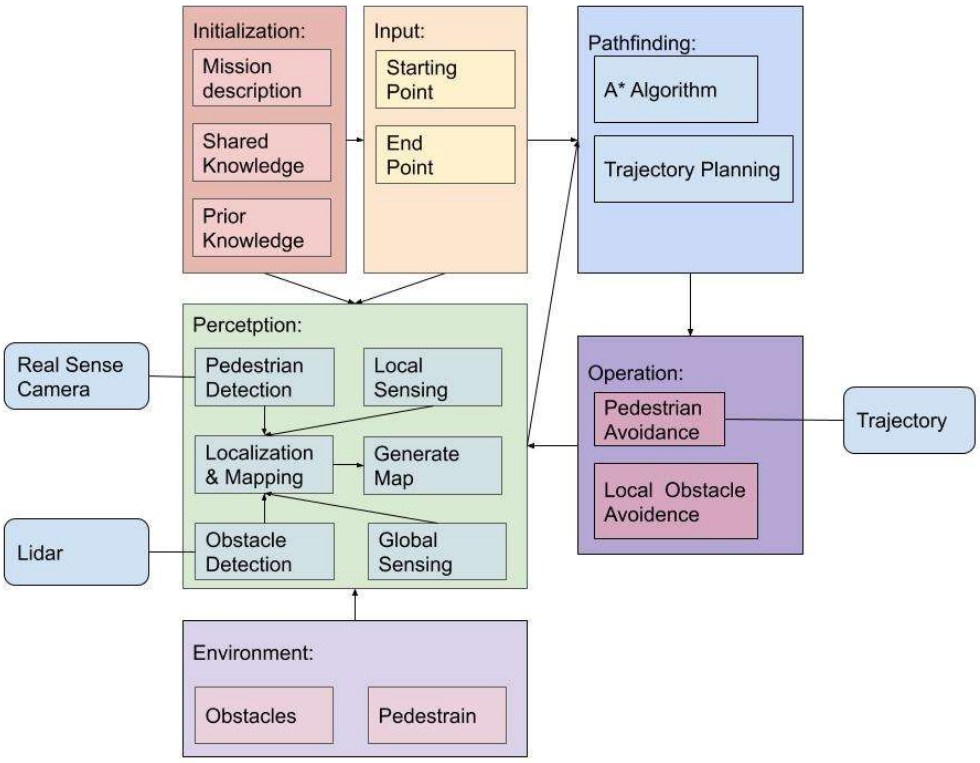

Figure 3.1: System layout of the project

This project consists of pathfinding, perception for pedestrian detection, and navigation for obstacle avoidance. These sections are mutually dependent, and the overview of the system is shown in Figure 3.1.

3.2 Design Specification

3.2.1 Robot

The first aspect that required consideration was the size of the robot. Since this project was targeting for the warehouse, a robot with similar size was preferred. Secondly, the robot needed to be easily programmable, thus a robot system with high quality open-source SDKs and many open-source software were preferred. The robot chosen at last was the TurtleBot2.

TurtleBot is a low-cost, personal robot kit with open-source software. The TurtleBot kit consists of a mobile base that is 14 inches by 14 inches in dimension, 2D/3D distance sensor, laptop computer or SBC (Single Board Computer), and the TurtleBot hardware mounting kit. In addition to the TurtleBot kit, users can download the TurtleBot SDK from the ROS wiki. The core technology from TurtleBot is SLAM and Navigation. TurtleBot can run SLAM algorithms to build a map and then move around using the included resources. Moreover, it is convenient for the team to test the robot because it can be controlled remotely from a laptop.

3.2.2 Mother Board

For the central information processor, a compact and power-efficient board was preferred, though it was not expected to complete heavy computation. The team selected the existing Jetson TX2 processor platform that was available in the lab.

Jetson TX2 is an embedded Artificial Intelligence (AI) computing device developed by Nvidia Corporation. It contains professional Graphics Processing Units (GPU) with 256 NVIDIA CUDA cores and 32GB of onboard storage. TX2 supports the power modes of 7.5W or 15W, for users to configure and apply [19].

3.2.3 Camera

Camera was the primary input device on the robot for pedestrian detection. For requirement, the camera must have a depth sensing function because the distance between pedestrian and the robot was necessary for performing avoidance. The camera chosen for this project was the Intel RealSense camera D435i.

The main advantage of this model is the depth vision, which is crucial for position calculation. This camera owns a wide field of the view with 86 degrees by 57 degrees, which supports versatile applications such as “robotics and augmented/virtual reality” within the range of 10 meters. Moreover, with Inertial Measurement Unit (IMU) involved, this camera can fulfill the applications with better depth awareness in motion. This unit is especially helpful during SLAM and tracking application by improving point-cloud alignment in this project [20].

3.2.4 LiDAR

LiDAR was the primary input for robot localization. Since the team preferred a low-cost, lightweight yet efficient LiDAR, the model Velodyne Puck-16 was chosen.

Velodyne Puck-16 LiDAR is a real-time depth sensor that can detect three-dimensional distance comprehensively. With a long detection range of 100 meters and a featherweight body of 830 grams, this compact sensor serves a broad field of view with 360 degrees horizontally and 30 degrees vertically, with 15 degrees up and down [21].

3.2.5 Path Finding Algorithm

A path finding algorithm was necessary to ensure the time efficiency in robot navigation. Existing shortest path algorithms were mostly similar, such that they all yielded the correct path towards the destination. Despite low time complexity, this algorithm must be convenient to implement for robot systems; hence the team chose A* as the path finding algorithm.

A* is implemented in versatile fields. This algorithm combines the property of Dijkstra's Algorithm, "favoring vertices that are close to the starting point," and Greedy Best-First-Search, "favoring vertices that are close to the goal." Therefore, it plans and optimizes the path ahead of the movement of robot [22].

3.2.6 Object Recognition Model

After obtaining information from the camera, the system needed to recognize the images. A computer vision algorithm was needed. A pre-trained model readily to be integrated with the robot system was preferred in this project. The pre-trained model chosen was YOLO.

YOLO is an efficient neural network for object detection on a full image. This network divides the image into regions and predicts bounding boxes and probabilities for each region. These bounding boxes are weighted by the predicted probabilities. YOLO has several advantages over classifier-based systems. It looks at the whole image so its predictions are informed by global context in the image. It also makes predictions using a single neural network unlike systems like R-CNN which generates thousands of candidates first followed by a classifier. This makes YOLO extremely fast, which is more than 1000x faster than R-CNN and 100x faster than Fast R-CNN [23].

3.2.7 Object Avoidance Algorithm

Once a pedestrian was detected and classified by the robot, the system needed to react to it dynamically. However, adopting an existing object avoidance algorithm from sources was not viable because nearly all the existing methods required different sensors that this robot did not contain. Thus, a simple object avoidance algorithm was designed. This algorithm took the human position as input. It created another layer in RViz based on the 3-dimensional coordinates generated by the algorithms. Then it marked the position of the pedestrian in the map.

4 Simulation

Simulation plays an essential role in this project by providing an opportunity to verify the feasibility of the objectives. It additionally helps the team to understand how the TurtleBot senses and operates visually.

4.1 Simulation Platform

Before testing on the real robot, an attempt to simulate the robot and its environment virtually on a computer was made using ROS on the Ubuntu system. The simulation platforms used were Stage and Gazebo.

4.1.1 Ubuntu System

Ubuntu develops and maintains a cross-platform, open-source operating system based on Debian, where Debian is a volunteer project that has developed and maintained a GNU/Linux operating system for well over a decade. The focuses of the Ubuntu system are release quality, enterprise security updates and leadership in key platform capabilities for integration, security and usability.

4.1.2 ROS

ROS is the acronym of the Robot Operating System. This “flexible framework of writing robot software” was built to simplify tasks for users. By creating tools, libraries, and conventions under the same platform, the complexity of connecting various robotic platforms after using each feature provided will now be lessened [24]. Throughout the distribution releases published by ROS, different versions are primarily targeted for various platforms. For example, the twelfth distribution release, ROS Melodic Morenia, was targeted at Linux Ubuntu 18.04 [25]. Moreover, ROS supports a wide range of robot hardware, including TurtleBot2 [26].

There are approximately more than 3000 packages in the ROS ecosystem, contributed from the ROS maintenance teams and the public. The core components of ROS include communications infrastructure, which provides inter-process communication at the low level; robot-specific libraries, such as Robot Geometry Library and Robot Description Language; and tools, such as Command-Line Tools and Rviz, which provides “three-dimensional visualization of many sensor data types and any URDF-described robot” [27].

4.1.3 Stage

Stage is a robot simulator. It provides a virtual world populated by mobile robots, sensors, and various objects for the robots to sense and manipulate. Stage provides several sensors and actuator models, including sonar or infrared rangers, scanning laser rangefinder, color-blob tracking, fiducial tracking, bumpers, grippers, and mobile robot bases odometry for global localization.

4.1.4 Gazebo

Gazebo 3D simulator allows users to "rapidly test algorithms, design robots, perform regression testing, and train AI systems using realistic scenarios" [28]. To implement Gazebo simulator with ROS, the package named gazebo_ros_pkgs is required. This package utilizes ROS messages and services to perform simulations in Gazebo [29].

4.2 Simulation: Stage

4.2.1 Simulation Objectives

The first simulation goal was relatively simple: performing a 2D simulation on the software Stage, particularly for map recognition and path planning, as Table 4.1 shown below. A map with perfect information was given to the robot. This preliminary simulation did not include the detection of local obstacles or random pedestrians.

Table 4.1: Aspect and achievement for Stage simulation

Aspects	**Achievements**
Map recognition	Able to transfer.png file into map
Path Planning	Able to path planning with known map

The preliminary simulation laid the ground for adding more features that were complex later, such as non-static objects and imported object locations.

4.2.2 Simulation System layout

The overall system layout for this simulation is shown in Figure 4.1.

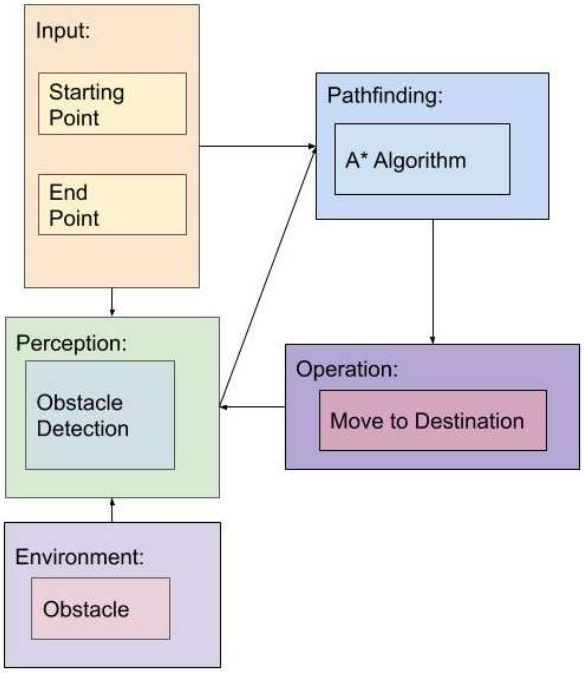

Figure 4.1: System layout for Stage simulation

Specifically, the system only required a starting point and an end point as inputs. Users only needed to specify the endpoint to drive the robot. The A* algorithm also took the map and obstacle information to generate the route automatically, then the chassis would be driven in the simulated environment with chassis data in TurtleBot SDKs.

4.2.3 Simulation Summary

The first step needed was map recognition to let the robot know what environment it was in. An image was given to be used as a map for the robot to navigate (see Figure 4.2).

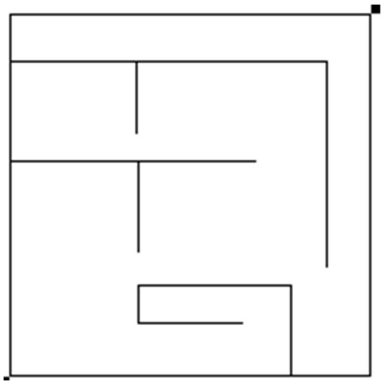

Figure 4.2: The map for Stage simulation

Then it allowed the simulator to generate a map as shown below in Figure 4.3 that was recognizable for the robot in Stage simulator. Now the robot could be controlled via RViz, as shown in Figure 4.4.

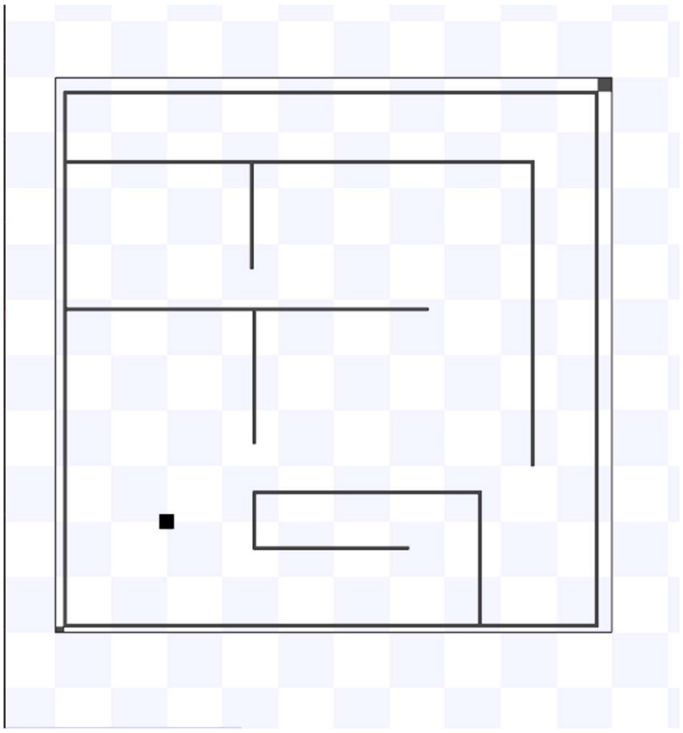

Figure 4.3: 2D simulation in the Stage simulator

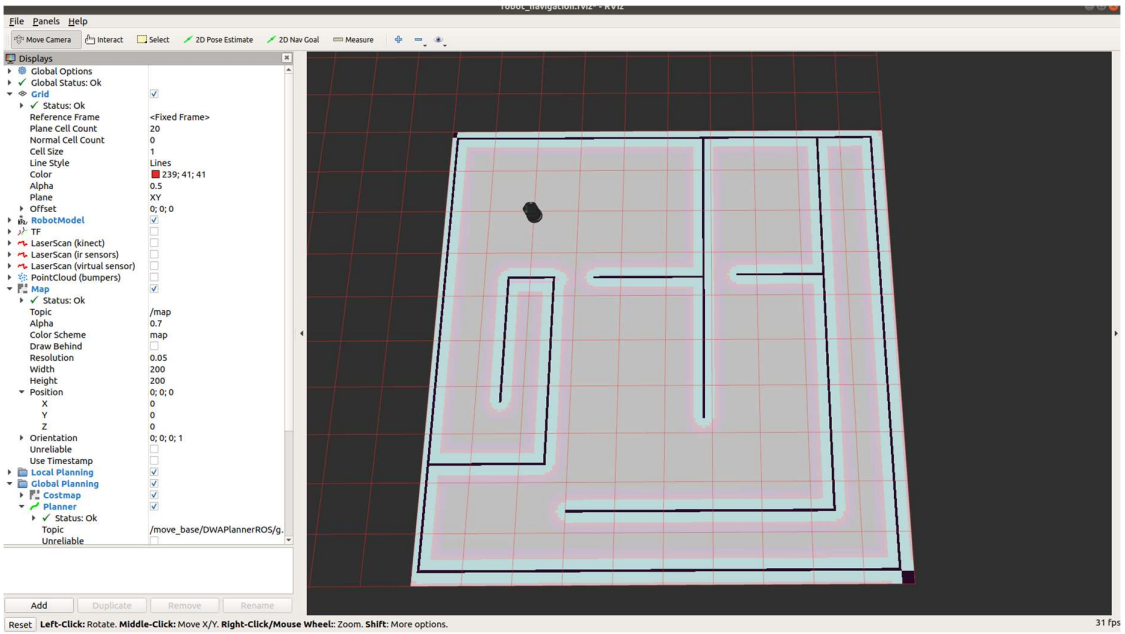

Figure 4.4: RViz control panel for the simulation

Our initial plan was to use the algorithm A* for path planning, included in the package Navfn. However, after the installation of Navfn, which included both A* and Dijkstra, it appeared that A* was not working; the developer of Navfn has stopped updating the package. Consequently, a new package, global_planner, was found, which contains various algorithms, including Dijkstra and A*. This version of A* was working, hence it was adopted.

4.3 Simulation: Gazebo

4.3.1 Simulation Objectives

The second simulation objective was to perform an upgraded version of the first simulation: a 3D simulation on Gazebo Simulator. In addition, for this simulation, a map input was not included. The robot needed to explore the local environment and generate a map on its own. Moreover, the environment was designed to be more challenging as random pedestrians and local obstacles were added. The overall objectives are shown in Table 4.2.

Table 4.2: Aspects and achievements for Gazebo simulation

Aspects	**Achievements**
Object Detection	Able to recognize obstacle
Object Reaction	Able to react correctly to the object
Exploring	Able to drive and explore in unknown area
SLAM	Generate map while moving

4.3.2 Simulation System layout

The overall system layout for this simulation is shown in Figure 4.5.

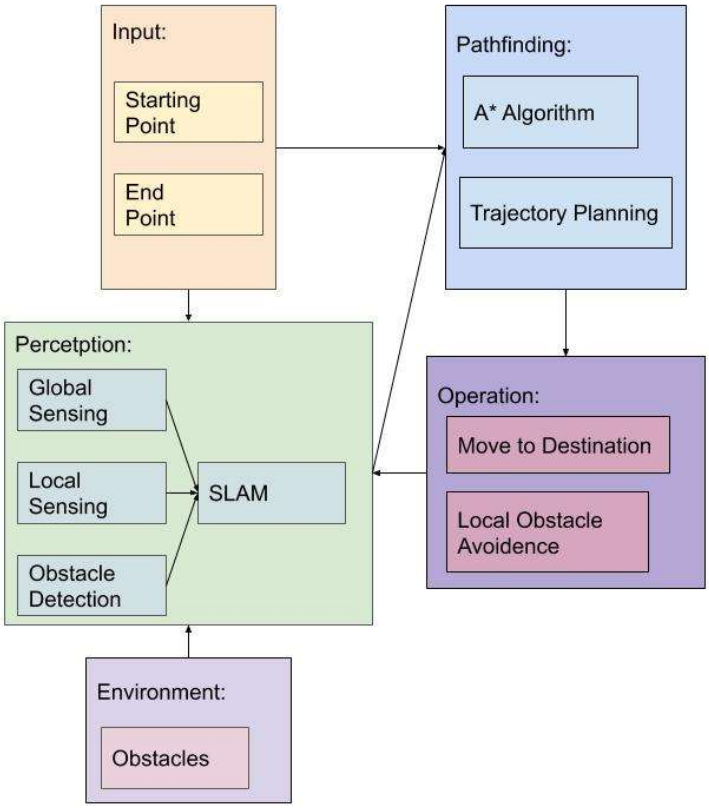

Figure 4.5: System layout for Gazebo simulation

In addition to the preliminary system, the new system added two features including trajectory planning and local obstacle avoidance, which utilized the information from local/ global sensing. The robot in the simulator would generate a path from user input, and then maneuver following the path. The system could generate a new path when it encountered a new local obstacle, and then follow the updated path on the movement.

4.3.3 Simulation summary

After accomplishing basic functionalities in the Stage simulation, this project moved on to the second simulation with more complex features, which presented a more realistic situation with pedestrians and obstacles. Gazebo allows robots to explore in a 3-D environment, which is practically similar to the real-world situations.

The first step was importing a gazebo.world file that set up the 3D environment for the robot. The gazebo.world file included playground.world, a file creating random obstacles and the robot. The robot would first perform gmapping, during which the robot would walk around, collect topographic information via camera, and transform learned information into a 2D map.

In the actual simulation, the robot was controlled by keyboards when exploring its environment. Through the camera, every image was transformed into a complete 2D map of the surrounding. Then the 2D map was exported into RViz, in which everything about the simulation could be manually controlled.

Therefore, the set objects were successfully achieved in this phase of the project: the robot could explore around an unknown area and transform everything it saw in the 3D world into a 2D map. The robot could detect and react to objects according to the map it generated.

For local obstacle detection, since local obstacle location was generated from an actual camera and human detection algorithm, it was not possible to simulate in the gazebo virtual environment. The local obstacles in the simulation were placed as several dummy points randomly generated based on the robot's position. The actual integration of human detection algorithm and local path planning were validated in the phase of on-site testing.

5 Coding and Testing

The integration of each component, testing of existing ROS packages and custom-written code consist mainly of two sections: TurtleBot and sensors. The following sections detail the coding process, individual testing, and integration testing for TurtleBot and sensors.

5.1 TurtleBot

There were two significant coding and testing parts of TurtleBot, including bringing up the TurtleBot and navigating from a start location to the desired location.

5.1.1 TurtleBot Coding

A ROS package called turtlebot_bringup was required to launch. It offered roslaunch scripts to start the functionality of the TurtleBot base. To be more specific, the base launch file called minimal.launch was utilized to start some essential nodes such as kobuki_node, robot_state_ publisher, and robot_ pose_ekf. There was mainly a robot model difference between the default setup of TurtleBot and that of the real one, so the default robot model was modified according to the real robot model. Notice that the sensors and control boards were different.

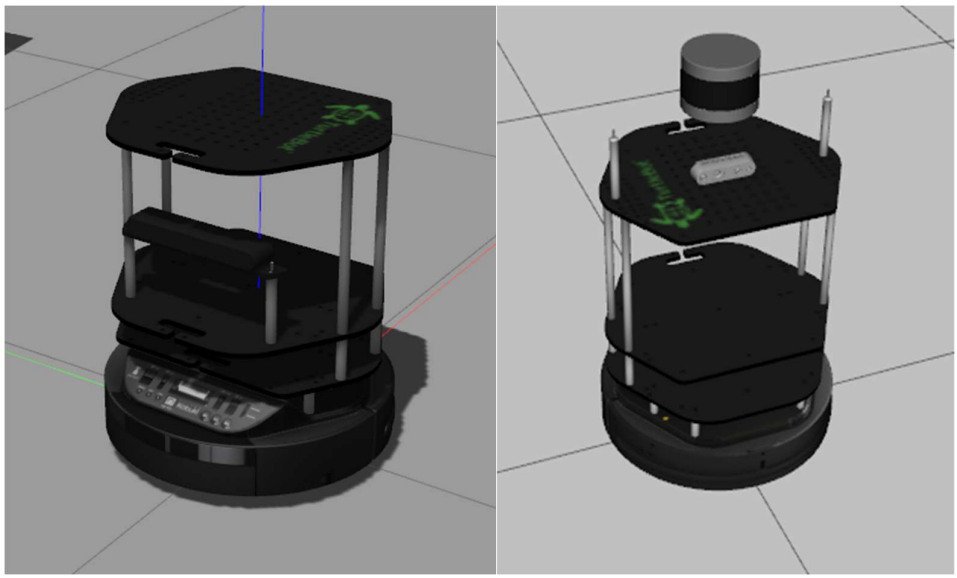

Figure 5.1: Default (left) and actual (right) TurtleBot models

As shown in Figure 5.1, the left model was the default TurtleBot model, while the right one was our actual robot model. Essentially, a default camera Microsoft Kinect camera supported by two poles was removed. Two poles were moved and attached to the top hexagon stack. They could support a rectangle plate used as a base to mount lidar and camera. The URDF files of lidar and camera were found online and integrated into minimal.launch file. There were several coordinate adjustments of these components after measuring their position. Ideally, using this modified launch file could start the TurtleBot and show the correct robot model in RViz.

The other ROS package that the team integrated and modified was called turtlebot_navigation. This package had a wide variety of functions, including costmap, local planner, global planner, map server, etc. The team decided to use A* algorithm to make path planning. However, by default, the global planner uses a package called Navfn, which created a path plan using Dijkstra's algorithm. There was a new version of the Navfn package called global_planner, which utilized the more reliable A* algorithm to generate a path by solving the optimization problem. For some launch files, they would find the defined parameters when they launch. Therefore, the modifications were completed on the parameters that defined what navigation package and path planning algorithm needed to be used.

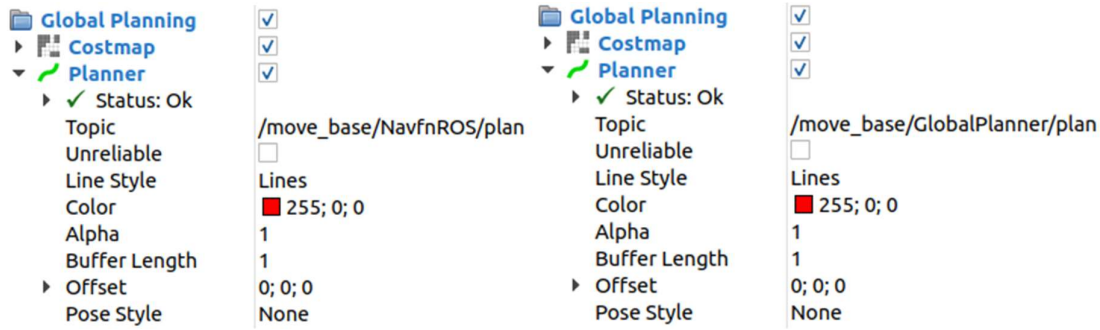

Figure 5.2: Default (left) and actual (right) navigation algorithms

As shown in Figure 5.2, after setting the new parameter, the global planner could access the global_planner package's nodes and topics. In addition to the global planner, the costmap had adjustable parameters such as frame, update frequency, and transform tolerance. These parameters could be modified as needed when testing the navigation for the robot.

5.1.2 TurtleBot Testing

The first step was to power Jetson TX2 and the kobuki base of TurtleBot. A power bank powered the Jetson TX2 while the kobuki base was using its designated battery. After pressing their buttons, the green lights indicated the base and Jetson TX2 are on.

The second step was to launch the kobuki base. When running the modified minimal.launch in the terminal, the terminal showed that the kobuki was detected and launched successfully. At the same time, a tinkle was made by the kobuki base.

Before testing the navigation, an initial testing of the kobuki base was conducted by using keyboard teleoperation. The keyboard teleoperation was a basic ROS package that allows people to remote control TurtleBot through the keyboard. After launching the keyboard teleoperation file, TurtleBot could move and rotate based on the key pressed.

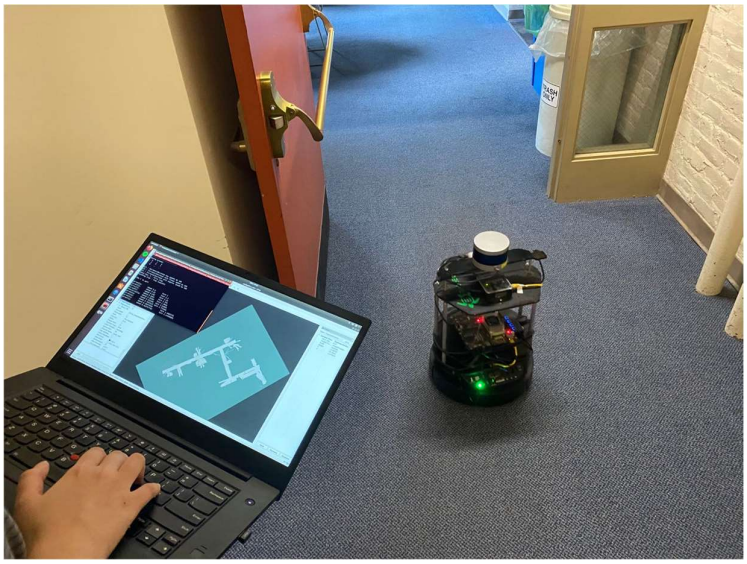

Figure 5.3: Remote control of TurtleBot through keyboard

The keys “i” and “,” were used to command TurtleBot to move forward and backward; keys“j” and “l” were used to turn TurtleBot to left and right, and key “k” was to stop TurtleBot. As shown in Figure 5.3, a team member was teleoperating TurtleBot to move forward on the second floor of Atwater Kent Laboratories.

5.2 Sensors

There were two primary coding and testing parts for the lidar, including launching the lidar and using the data from the lidar to build the floor map and localize the robot. For the camera, in addition to the launch file for initializing the camera, the launch files for human detection and map layer were also required.

5.2.1 Lidar Coding

To launch the lidar to generate the sensor data, the ROS package called veloydne_pointcloud was utilized. This package provided ROS nodelets and sensor_msgs/PointCould2 messages. However, since the gmapping package only created the map from sensor_msgs/LaserScan messages, a package called pointcloud_to_laserscan was required. This package offered a ROS node which took a pointcloud2 message and converted it to a 2D laser scan message.

Another way was found on how to generate the sensor_msgs/LaserScan messages. The velogyne_laserscan node could convert a ring of a Velodyne pointcloud2 to a sensor_msgs/LaserScan message and published it. It was noticed that some users reported the issues of the velogyne_laserscan node. A testing was done to see whether this node works better than pointcloud_to_laserscan node or not.

Lidar was responsible for Adaptive Monte Carlo Localization (AMCL) when a map was given. A ROS package called amcl was included in the launch file. This package took three required messages, including a laser-based map, laser scans and transform messages, and an optional message called initalpose produced the estimated pose of the robot on the map. The map would be built from gmapping, and the laser scans were generated from either pointcloud_to_laserscan node or velodyne_laserscan node.

5.2.2 Lidar Testing

The first step was to power and launch the kobuki base. Then, a gmapping node and a laserscan node were launched to build the map. Packages such as velodyne_laserscan and pointcloud_to_laserscan were tested individually.

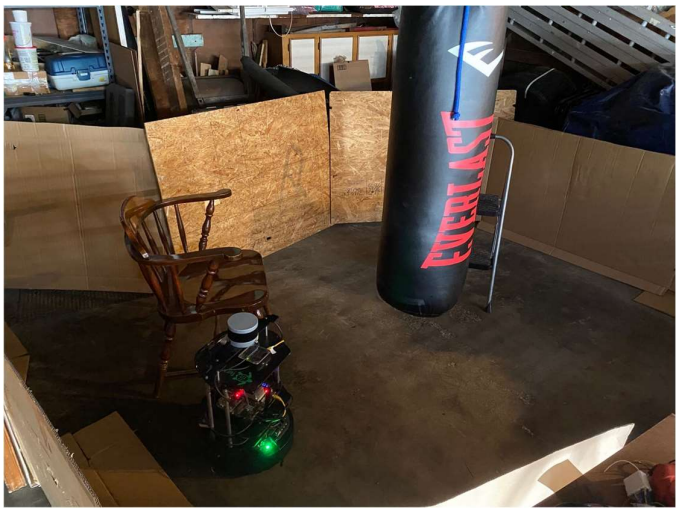

Figure 5.4: TurtleBot was building map in garage

An initial testing of building the map was done in a garage. TurtleBot was placed in an area enclosed by the cardboard (see Figure 5.4). The team remotely drove the TurtleBot to scan the surroundings.

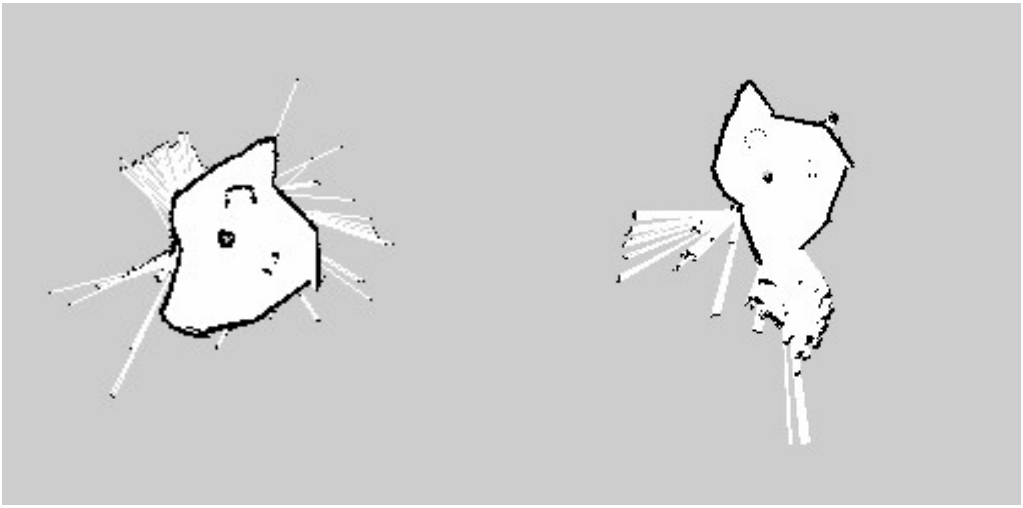

Figure 5.5: The map created by pointcloud_to_laserscan (left) and velodyne_laserscan (right)

As shown in Figure 5.5, the gmapping built two maps by using pointcloud_to_laserscan node and velodyne_laserscan node, respectively. The lidar could do a 360-degree scanning, so two maps were built in 30 seconds. Using pointclould_to_laserscan node could build a more distinct map, so later on, building the map relied on pointclould_to_laserscan node.

A further testing was conducted on the second floor of Atwater Kent Laboratories (AK lab). The goal was to build the floor map using the lidar. Following the same steps as taken during the initial testing, the team obtained

the building floor map (see Figure 5.6). TurtleBot started from the position marked in red, moved around the floor, and finally returned to the starting position.

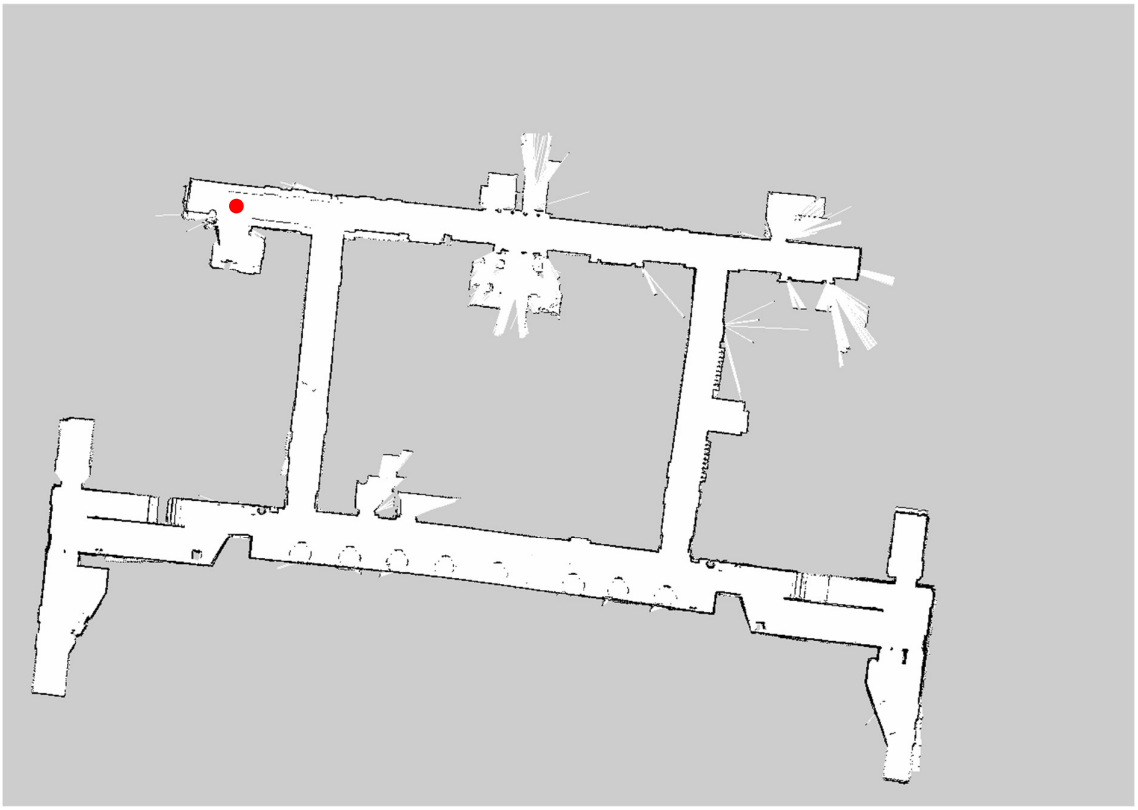

Figure 5.6. The second floor map of AK lab

After obtaining a static map, TurtleBot could navigate from a start position to a request position. AMCL would help TurtleBot to localize the robot. Since the tests of AMCL and navigation could be done together, a combined test is outlined in Section 5.3.

5.2.3 Camera Coding

The primary function of the camera in this project was to detect humans. The first step was to launch the RealSense Camera D435i. A ROS package called realsens2_camera was used to launch the RealSense Camera and publish the topics. Some settings of the RealSense Camera, such as image width, height, and frames per second, were modified. This would be discussed later in section 7.

The next step was to use YOLO to detect humans. An existing ROS package called YOLO ROS was developed for object detection using camera images. Although this pre-trained YOLO ROS could detect different objects, in this project, the team were only interested in pedestrian detection. Therefore, the

object detection class was modified to focus on detecting humans only. To access the data from RealSense Camera, YOLO's default subscriber was changed from /camera/rgb/image_raw to /camera/color/image_raw.

Moreover, the implementation of human detection capability was not sufficient since the robot needs to know not only a person's existence but also the exact location of a person for collision avoidance. Thus, a new ROS package called coordinate_map was written to use the depth information and point cloud data generated by the RealSense Camera to calculate the person's location in the world coordinate with respect to the camera.

The final implementation was to update the person's location on the map. This would let the path planner know the existence and location of the person. Based on the given person's location, the path planner could create a new path to avoid the person. The costmap consisted of several layers, such as the obstacle and inflation layer. The idea was that a new layer used for merely displaying the person's information could be added to the map. Therefore, a new ROS plugin was coded to read the position information from coordinate_map and project the position information into the new costmap layer, then display the person dynamically on the map based on the person's calculated location. Within this layer, a marker was used to display the person as a dark square on the map.

5.2.4 Camera Testing

The RealSense Camera was connected to the Jetson TX2 installed on the robot. Running the camera and YOLO launch file was able to launch the camera and detect the person. A window was popping on. Within the window, the camera's real-time image was shown, and a bounding box indicated the person's position.

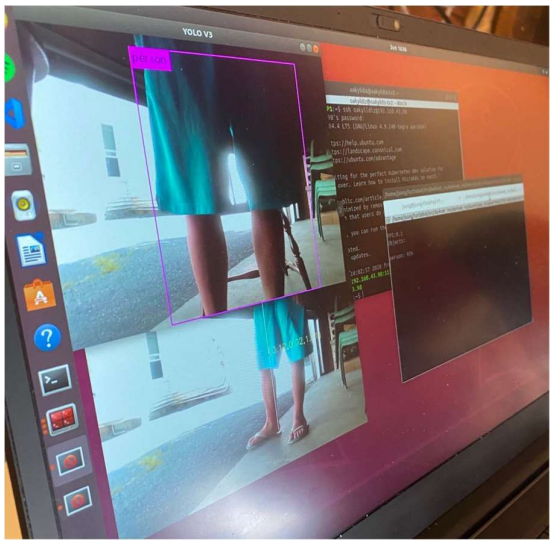

Figure 5.7: The testing of YOLO detection

As shown in Figure 5.7, a tester was standing closely in front of the camera. Even though the camera only captured the lower part of the body, YOLO could still detect the person successfully.

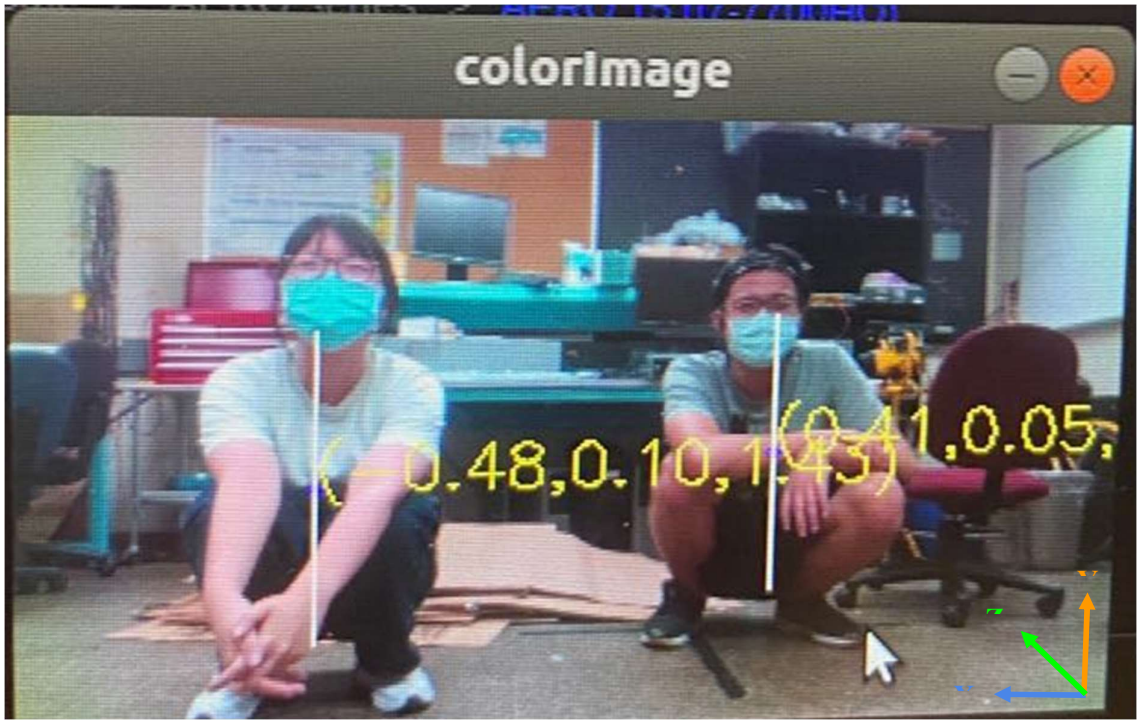

Figure 5.8: The coordinate of two people

The next part was to test if the program could give precise location information of the person. The launch file was running to calculate the coordinate of the person. As a human was detected, a pop-up window drew a

line to indicate the center axis of the person and displayed three numbers in parentheses. As shown in Figure 5.8, two testers crouched in front of the camera. The program calculated and displayed the coordinate of these two testers. Three numbers in the parentheses represented the distance from the camera to the person in X, Y, and Z direction, respectively. More tests were carried out to verify the correctness of coordinate. For example, the object moved forward and backward, leftward and rightward, stood up, and sat down to verify if coordinates were changed as expected.

After obtaining the location information of the person, the human layer was updated to the map. The layer node was launched to display the person on the map. Two testers crouched in front of the camera (see Figure 5.8). The map was updated, as shown in Figure 5.9. The hexagon was TurtleBot, and two little squares represented the locations of two testers. These two squares would be constantly updated based on the actual positions of human objects and could be erased entirely when the person moved out of the frame.

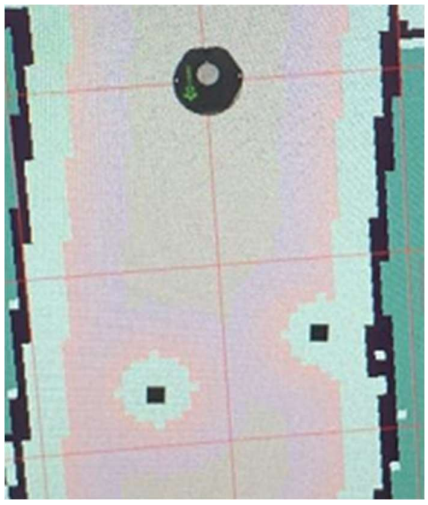

Figure 5.9: The costmap of two testers

5.3 Combined Test

Followed by the individual test of TurtleBot and sensors, a combined test was conducted in the following scenarios listed from easy to hard:

The general command for the robot was to navigate from the start position to a request location.

1. On the way to the destination, there were no pedestrians.
2. On the way to the destination, there was a standing pedestrian.
3. On the way to the destination, there were two standing pedestrians.

4. On the way to the destination, a walking pedestrian is passing by the robot.

To make the pedestrian detection and avoidance more convincing, the pedestrians were standing or walking in the hallway where the robot was navigating towards its target destination. The demonstration video can be viewed using this YouTube link:
https://www.youtube.com/watch?v=6usXNrI7-Fk

6 Results

Turtlebot was able to generate maps for the surrounding environment, such as a garage and the second floor of Atwater Kent Laboratories, using keyboard teleoperation, map generation, and localization packages. The map presented high clarity for larger objects at the height of LiDAR, whereas lower clarity for smaller objects, such as the round tables that are shown as unclearly defined circles on the map (Figure 5.6).

YOLO could detect humans from parts of the body, such as limbs (Figure 5.7), or the entire figure (Figure 5.8) when the human was walking, crouching or standing. The map could calculate the coordinate and mark humans' positions as squares on the updated map (Figure 5.9). However, the speed of detecting humans and updating coordinates was related to various factors, such as computer computing capacity, network data bandwidth, people's moving speed, and the camera's refreshing rate. Therefore, the team often observed short delays when refreshing images from the camera or updating humans' coordinates.

Turtlebot could navigate from its current point to a designated point, but there existed low possibilities that the navigation algorithm failed to calculate the path and aborted the process. When humans were detected, TurtleBot would adjust the navigation path to avoid the obstacle updated on the map. In successful cases, the robot would generate a new path in the unoccupied spaces, moving around or away from the obstacles, and proceed to its goal. TurtleBot could successfully reach the goal when there were no pedestrians, or when there were multiple standing pedestrians or one walking pedestrian passing by the robot. However, the team sometimes observed failure cases caused by unsuccessful human detections when testers were standing too close to the camera or moving too quickly, due to the time delay for publishing the pedestrian coordinates and updating its navigation path.

7 Discussion

Piecing together the whole system raised some unexpected problems. This section elaborates on the potential flaws that can be solved by taking different approaches, the error exposed during testing, and suggestions for future work.

7.1 Improving System Processing Speed

When launching all components, the system would occasionally terminate itself automatically to prevent over-computing due to the multitude of different tasks.

To reduce the work from the computing device, various attempts were made:

1. Change the resolution of the generated image to the minimum value. For RealSense camera, the minimum resolution of the image is 320 * 180. However, this may adversely lower the detection accuracy of the YOLO algorithm.
2. Change the FPS of the camera from 20 to 6.
3. Optimize the architecture of the system by shutting down all unrelated branches.
4. Disable the graphic display for Jetson TX2.
5. Use another laptop to SSH into it. Launch control panel Rviz on that laptop to reduce tasks.
6. Change to another computer vision model (see section 7.2).

After a series of different attempts, the speed of the whole system increased. In terms of memory usage tracked in the memory manager of the processor, a summary of each attempt is shown below.

- Attempt 1 and 2 were the most effective way to improve memory performance, releasing about 60% of memory space from 2 cores.
- Attempt 3, 4 were not as effective. They barely made any differences in the memory managers.
- Attempt 5 produced some improvements, but they were outweighed by its resulting problems. Details will be discussed in detail in section 7.2.

- Attempt 6 barely resulted in any changes in the memory managers. However, there was not much delay on the control panel RViz anymore.

The experiment also yielded some warnings and recommendations for processing speed:

- Real-time human detection consumed much computing power. If the budget allows, it is recommended to purchase a device with more computing power.

7.2 The Choice of Computer Vision Model

The most power consuming part of the system was the real-time human detection model. The model chosen for this project is YOLO. It was a pre-trained object-classifying model. Since the system was overloaded with tasks, another model called tiny-YOLO was tested as an attempt to improve processing speed. The main difference between these two models was that tiny-YOLO did not have as many neural network layers as YOLO to process the image taken in, which indicated that the tiny-YOLO would have a much faster processing speed. Nevertheless, the trade-off was a decrease in accuracy. Rounds of testing with tiny-YOLO leaded to the observation that although the processing speed was faster than the regular YOLO, the robot failed to detect humans in some cases.

Even though the tiny-YOLO model only took up 50% of the memory that the regular YOLO model used, it could not detect humans when only seeing small parts of humans. An attempt to adjust the threshold was made. If the threshold was set too high, it would not detect humans unless 90% of the human body was revealed in front of the camera. If the threshold was set too low, it would think most objects with similar contour were human.

For the use of computer vision model, a series of recommendations are made here:

1. In this scenario, where the camera was set at a height of 7.5 inches, accuracy was more important than processing speed.
2. If we have more time, training a human detection model is preferred because considering solely the capability of human detection could lessen the layers in the neural network model.

3. The team implemented and tested two additional packages for human detection and tracking, “leg_detector” and “multiple_object_tracking_lidar,” attempting to compare and check if more efficient computer vision or the less computational ability of the computer could be achieved. Details for the two additional packages will be discussed in section 8.

7.3 The Accuracy of Navigation

Repeated testing revealed another problem, accuracy of navigation, which indicated the coherence between the physical robot navigation and the simulated robot navigation on RViz. This test was carried out in two different locations: in the garage and at the second floor of an academic building.

In the garage, the navigation accuracy seemed lower than expected, which meant that sometimes when the RViz robot had already reached the destination, the actual robot was still about half a meter away, and this half-meter difference fluctuated constantly.

In the building, the navigation accuracy seemed higher than that in the garage. A series of observations leaded to a hypothesis that the difference between the real-life robot and the RViz robot was proportional to the total distance traveled. One possible reason was due to the lidar localization accuracy in different building environments.

This experiment raised the assumption that the robot in real life was affected by the nature of the floor. The flooring in the garage was uneven, which might result in inconsistent differences. On the contrary, the building floor was carpeted.

For this observation, a series of recommendations are made:

1. It is essential to pay attention to the differences between the real-life robot and the robot in the control panel. The differences will keep increasing as time goes on.
2. Synchronizing the robot's location in real-life and the corresponding robot in RViz periodically is preferred for long-term work.

8 Additional Packages for Human Detection and Tracking

Two other packages for human detection and tracking were implemented and tested to compare with the darknet_ros package utilized in two aspects: the efficiency of human detection and tracking, and the delay caused by algorithm calculation and processing.

8.1 Background

In this section, the team introduces the two ROS packages used for human detection and tracking.

8.1.1 leg_detector

This package subscribes to laser scans as a topic, calculates the possible laser scans as legs, and potentially pairs the legs to show as a person. One of the package's published topics, "visualization_marker," can be used in RViz to show the detected legs and persons. Both markers of legs and persons are presented as spheres inside RViz, while persons' markers appear larger than legs'.

Three of the vital parameters provided for users to adjust are:

- connection_threshold: maximum meters of separation for lasers to be considered as a group
- min_points_per_group: minimum points in a laser scan group
- leg_reliability_limit: minimum reliability to consider input as a leg [30].

8.1.2 multiple_object_tracking_lidar

This package subscribes point-cloud as the topic, extracts the possible clusters of person or objects, and tracks through Kalman Filters. The published topic "viz" provides a marker array with cubes in different colors for users to visualize inside RViz. Since six objects will be tracked at once, the cubes will be presented at the origin point if fewer objects are tracked in RViz [31].

8.2 Implementation

Implementation of the Gazebo simulation environment and RViz visualization are discussed in this section.

8.2.1 Gazebo Simulation Environment

The environment was built with multiple human-like animated models. Each animated model was created to walk between two designated points repeatedly. The walking speed of the animated models can be changed by controlling the time for each walking segment.

TurtleBot Model was added with a simulated LiDAR, Velodyne VLP-16 sensor, that was able to publish a topic containing information of point-cloud.

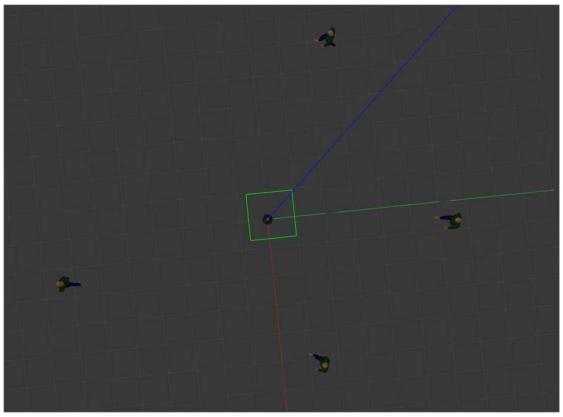

Figure 8.1: TurtleBot model

8.2.2 RViz Visualization

RViz was used to present point-cloud laser scans from the simulated LiDAR and show the markers published from human detection and tracking ROS packages. Laser scans were generated by transforming point-cloud with pointcloud_to_laserscan ROS package.

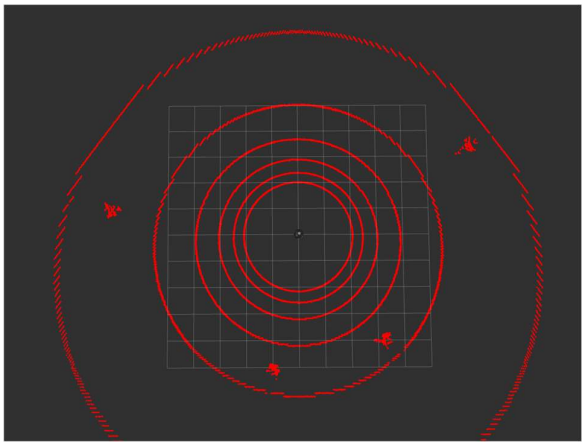

Figure 8.2: RViz Visualization with point-cloud of animated models

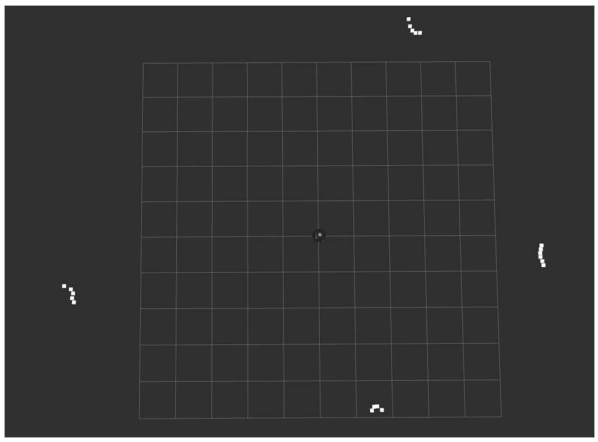

Figure 8.3: RViz Visualization with laser scans of animated models

8.3 Testing

During testing, various parameters and limitations were changed to achieve better detection results and tracking for both packages. For leg_detector, parameters were revised when launching the node; for multiple_object_tracking_lidar, parameters and limitations were revised inside its “main.cpp” file.

8.3.1 leg_detector

With the default parameter settings, leg_detector was able to detect legs and attempt to pair legs into a person when the model was within approximately 1.8 meters away from the TurtleBot, measuring with the built-in tool in RViz.

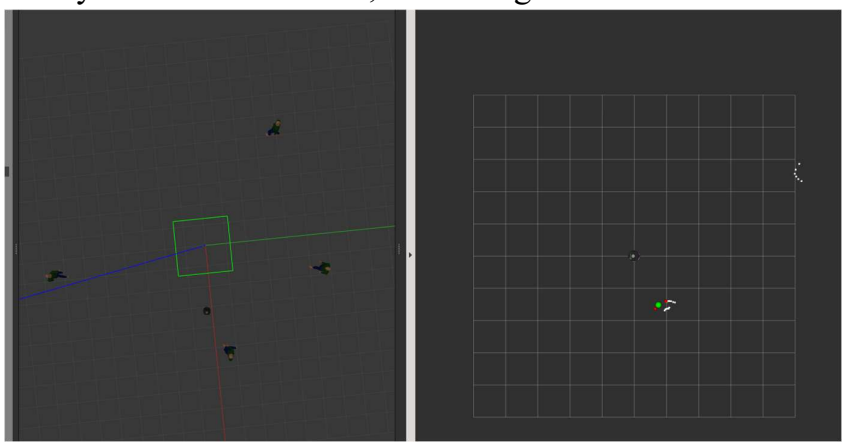

Figure 8.4: RViz(right) simulation of leg_detector: farthest distance of detection with default parameters

By lowering the value of min_points_per_group, leg_reliability_limit, and increasing connection_threshold, it was able to detect legs and potentially pair legs to persons within approximately 5.3 meters away from the TurtleBot, measuring with the built-in tool in RViz.

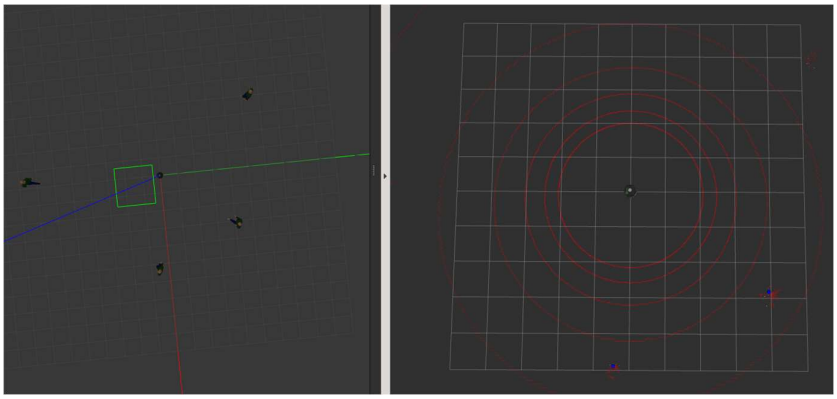

Figure 8.5: Gazebo(left) and RViz(right) simulation of leg_detector: farthest distance of detection with changed parameters

The optional input topic called people_tracker_filter was not utilized since this topic should be published from a face detection algorithm that was not implemented in this testing.

8.3.2 multiple_object_tracking_lidar

It was difficult to track the animated models if the default package version was utilized as all markers appeared to move quickly and randomly in RViz. It was able to detect some animated models for a short time but failed to track them consistently.

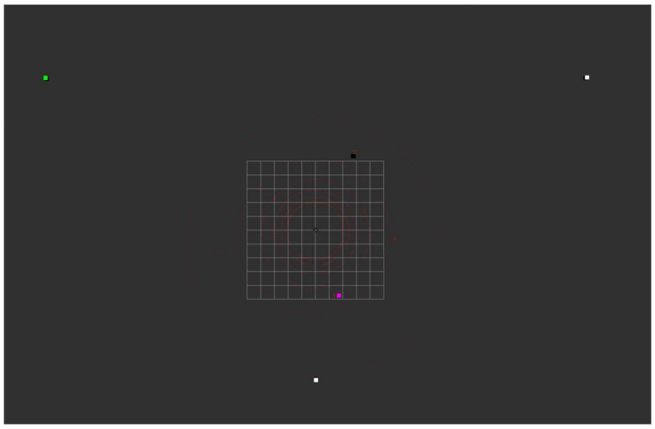

Figure 8.6: RViz simulation of multiple_object_tracking_lidar: two models out of four detected, other markers scattered

The team improved the accuracy of detection and tracking on animated models by adding a limitation of centroids calculated from extracted clusters. The cluster could only be tracked if the centroids' x and y distances are smaller than 10 meters. Additionally, parameters used in cluster extraction functions were adjusted by increasing the cluster tolerance and broadening the limitation of cluster size in point-cloud. It resulted in a faster detection and a more stable pattern of tracking.

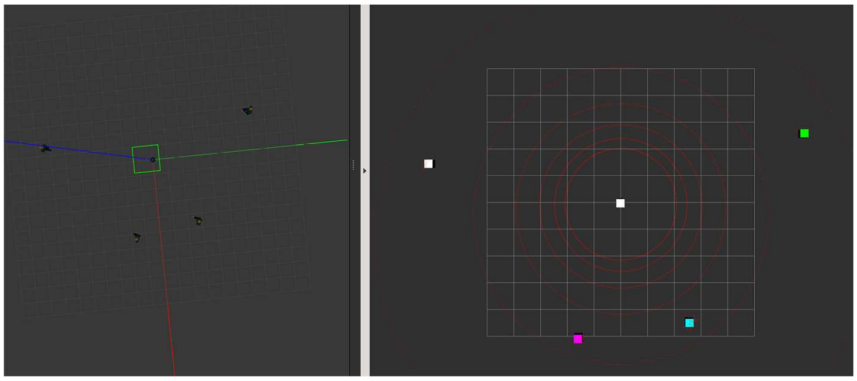

Figure 8.7: RViz(right) simulation of multiple_object_tracking_lidar: successful detection and tracking of four animated models, markers without detection target at the origin

If the limitations of the maximum cluster size were adjusted too high, detection would fail, and markers would present at the origin all together. Therefore, the ideal distance for the animated models to be detected and tracked within the limitation of cluster size ranged approximately from 3 meters to 10 meters.

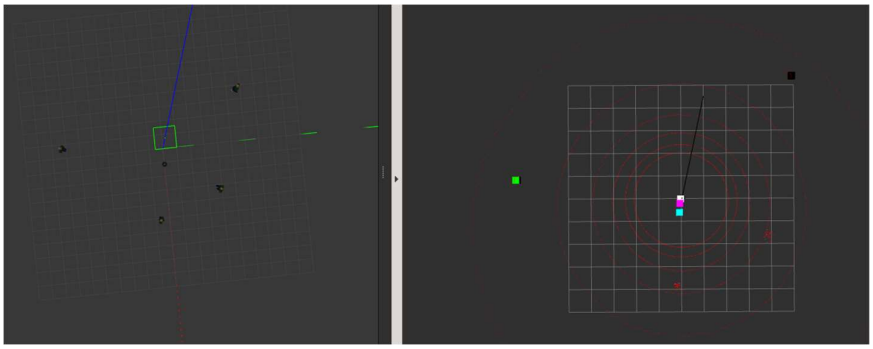

Figure 8.8: Gazebo(left) and RViz(right) simulation of multiple_object_tracking_lidar: failed detection and tracking of two animated models with large maximum cluster size

8.3.3 Running Both Packages Simultaneously

When running both packages simultaneously, leg_detector was able to detect models closer to the TurtleBot, and multiple_object_tracking_lidar was

able to detect models further to the Turtlebot by controlling each parameter. However, overlaps in detection were observed, which increased the complexity.

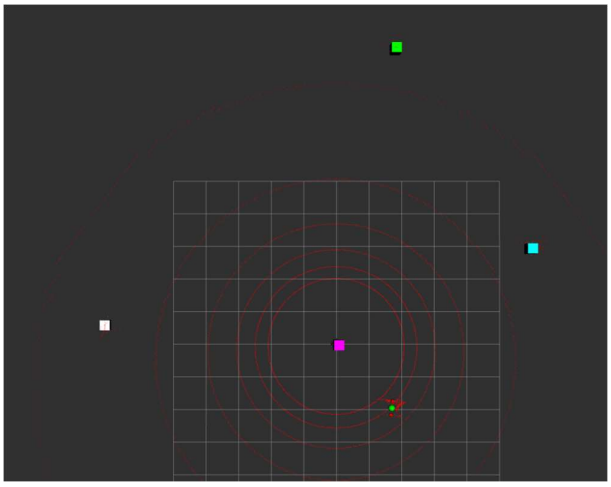

Figure 8.9: RViz of running simultaneously: Successful complement in distance

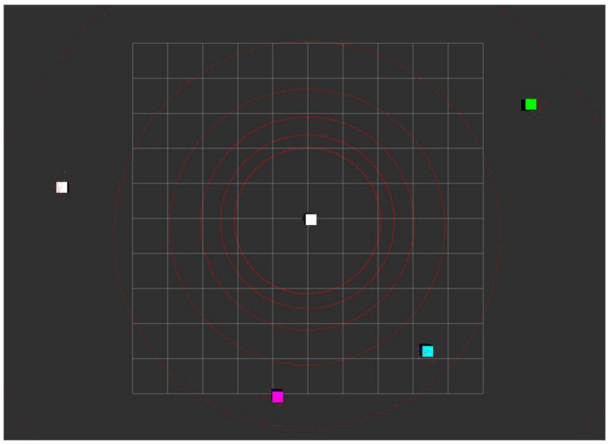

Figure 8.10: RViz of running simultaneously: Overlap in bottom two models' detection, having sphere and cube on the same model

8.4 Result

Both ROS packages were able to detect and track the animated models, and their accuracy can both be adjusted with the parameters discussed in section 8.3. With the built-in simulation environment, both packages can run together to complement the limitation of distance with some overlaps (Figure 8.10).

The package darknet_ros uses images from the camera, which has limited degree of view, but the two packages we tested used the data from LiDAR with 360 degree of view (Figure 8.2). Therefore, leg_detector and multiple_object_tracking_lidar can detect a much larger area than darknet_ros. Additionally, these two packages had fewer requirements for the computing

device. Both packages were tested on a laptop with Nvidia Quadro P1000 smoothly, while darknet_ros failed to process in real-time because the laptop appeared to be frozen. However, darknet_ros had a higher accuracy in human detection when the target was more than 10 meters away.

8.5 Discussion

8.5.1 Parameter Changes in leg_detector

Since the simulation environment contained only animated models, a higher detection possibility with longer distance away from the sensor could be achieved by varying the parameters as described in Section 8.3. However, in the real environment, objects with similar appearances in laser scans such as chairs or shelves might confuse the leg_detector. Therefore, the parameters would have to be adjusted accordingly. The parameter fixed_frame in leg_detector should be changed according to the relative frame name that the user's robot contains.

9 Enable the Camera to Rotate to Detect Pedestrian

The following section outlines a mechanical design to resolve a problem encountered during the testing.

9.1 Problem Statement and Design Objectives

During the testing, a human detection problem was found. To be specific, the TurtleBot could only detect the pedestrians and update the path to avoid them when the TurtleBot was facing the pedestrians. It was observed that when the TurtleBot was passing a pedestrian, the camera could not detect the pedestrian since the camera was mounted on the robot with the viewing angle fixed, which led to the limited field of view. Therefore, the path planner would not recognize the pedestrian on the side as an object to avoid. If the destination is on the same side of the pedestrian, the new shortest path generated by the A star path planner may cause the TurtleBot to hit the pedestrian. Hence, it was needed to design a way to rotate the camera. This function could enable the camera to detect pedestrians who are outside the field of view of the fixed camera and thus lower the possibility of collision with pedestrians.

9.2 Nomenclature

Table 1 details the variables used throughout this section of the report to develop the design.

Table 9.1: Nomenclature

Variable	Meaning	Variable	Meaning
h	Euclidean Distance from the from a Robot to a Pedestrian (m)	a	Vertical Distance from a Robot to a Pedestrian (m)
r	Horizontal Distance from a Robot to a Pedestrian (m)	θ	Angle (degree)
I	Moment of Inertia (kg $\cdot$cm^2)	m	Mass of Camera (kg)
L	Width of Camera (m)	V_R , V_P	Velocity (m/s)
τ	Torque (kg $\cdot$cm^2 /s^2)	α	Angular Acceleration (rad/s^2)

9.3 Known Parameters

Table 2 summarizes parameters obtained from the official datasheet of Intel RealSense Camera D435 that have been used in the design.

Table 9.2: Specification of Intel Realsense Camera D435 [32]

Dimension	Min	Nominal	Max	Unit
Width		90		mm
Height		25		mm
Depth		25		mm
Mass		72		gr

Parameter	Camera Sensor Properties
Image Sensor	OV9282
Active Pixels	1280 X 800
Sensor Aspect Ratio	8:5
Format	10-bit RAW
F Number	f/2.0
Focal Length	1.93mm
Filter Type	None
Focus	Fixed
Shutter Type	Global Shutter
Signal Interface	MIPI CSI-2, 2X Lanes
Horizontal Field of View	91.2°
Vertical Field of View	65.5°
Diagonal Field of View	100.6°
Distortion	<=1.5%

From Table 2, the camera's mass m is 72 grams and width L is 90 mm, which were used to determine the moment of inertia of the camera. In addition, the camera's horizontal field of view is 91.2 degrees, which was used to determine the required rotational speed for the camera to detect a moving pedestrian.

9.4 Design Assumptions

Following assumptions have been made in developing the design:

- The constant moving speed V_R of TurtleBot is 0.6 m/s and V_P of the pedestrian is 1.4 m/s.
  The moving direction of TurtleBot and pedestrians is opposite.
- The horizontal distance r between TurtleBot and pedestrian is 0.5 meter.

- The moment of inertia other than camera and turntable is negligible.
- The camera is a slender rod and the turntable is a flat plate.

9.5 Design Schematic and Justification

In this section, a solution to the problem and justification are provided based on the previously stated assumptions and known values.

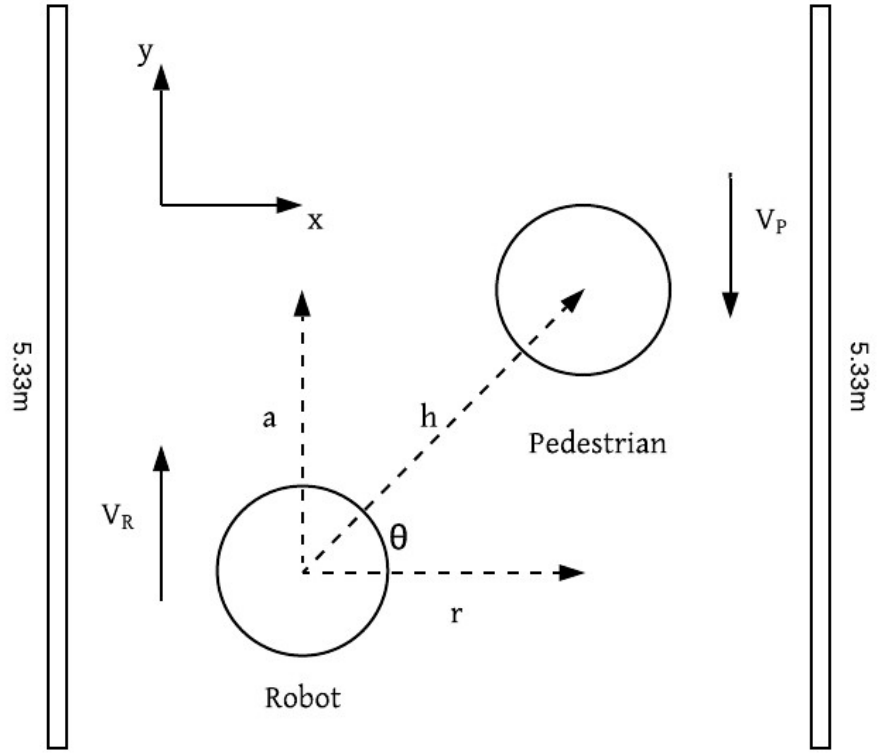

Figure 9.1: The Schematic of Pedestrian Detection

Based on Figure 9.1, equations can be derived as shown in the following:

$$tan\,(\theta) = \frac{a}{r}$$

$$\frac{d}{d\theta}(tan\,(\theta)) = \frac{1}{r}\frac{da}{d\theta}$$

$$sec^2(\theta) = \frac{1}{r}\frac{da}{dt}\frac{dt}{d\theta}$$

$$\frac{d\theta}{dt}sec^2(\theta) = \frac{1}{r}\frac{da}{dt}$$

$$\frac{d\theta}{dt} = \frac{cos^2(\theta)}{r}\frac{da}{dt} \quad (1)$$

Based on assumptions and known parameter, two parameters can be obtained as the following:

1. θ is the complementary angle of the half horizontal field of view.

$$\theta = 90 - \frac{1}{2} \times 91.2 = 44.4\, degs$$

2. $\frac{da}{dt}$ is the relative speed between TurtleBot and pedestrian.

$$\frac{da}{dt} = V_p + V_R = 1.4 + 0.6 = 2.0\, m/s$$

Therefore, the required rotational speed for camera to detect a moving pedestrian is:

$$\frac{d\theta}{dt} = \frac{cos^2(44.4°)}{0.5} \cdot 2 = 2.04\, rad/s = 19.5\, rpm \quad (2)$$

Furthermore, the maximum angular acceleration of the camera and the torque required to rotate the camera were calculated:

$$\frac{d\theta}{dt} = \frac{cos^2(\theta)}{r}\frac{da}{dt}$$

$$\frac{d^2\theta}{dt^2} = \frac{d}{dt}\left(\frac{cos^2(\theta)}{r}\frac{da}{d\theta}\right)$$

$$\frac{d^2\theta}{dt^2} = \frac{1}{r}\left(\frac{d^2a}{dt^2} \cdot cos^2(\theta) + (-2) \cdot \frac{da}{dt} \cdot cos(\theta) \cdot sin(\theta) \cdot \frac{d\theta}{dt}\right)$$

$$\frac{d^2\theta}{dt^2} = \frac{1}{r}\left(0 + (-2) \cdot \frac{da}{dt} \cdot cos(\theta) \cdot sin(\theta) \cdot \frac{cos^2(\theta)}{r}\frac{da}{dt}\right)$$

$$\frac{d^2\theta}{dt^2} = (-2) \cdot \frac{1}{r^2} \cdot \left(\frac{da}{dt}\right)^2 \cdot cos^3(\theta) \cdot sin(\theta)$$

$$\frac{d^2\theta}{dt^2} = 2 \cdot \frac{1}{0.5^2} \cdot 2^2 \cdot 0.325$$

$$\frac{d^2\theta}{dt^2} = 10.4\, rad/s^2$$

According to assumption, the camera is considered as a slender rod and the turntable as a flat square plate which would have moment of inertia I_c and I_t as respectively shown below:

$$Ic = \frac{1}{12} \cdot m \cdot L^2$$

$$I_c = \frac{1}{12} \cdot \frac{72}{1000} \cdot \left(\frac{90}{10}\right)^2 = 0.486\ kg \cdot cm^2$$

$$It = \frac{1}{6} \cdot m \cdot a^2$$

$$I_t = \frac{1}{6} \cdot \frac{30}{1000} \cdot \left(\frac{76.2}{10}\right)^2 = 0.290\ kg \cdot cm^2$$

$$I_{tot} = I_t + I_c = 0.776\ kg \cdot cm^2$$

$$\tau = I_{tot} \cdot \alpha = 0.776 \cdot 10.4 = 8.07\ kg \cdot cm^2 \cdot s^{-2} \quad (3)$$

From (2) and (3), the required rotational speed is 19.5 rpm and the maximum torque required during the rotation of the camera is 8.07 $kg \cdot cm^2 \cdot s^{-2}$. Based on the calculated rotational speed and torque, in the next section, a suitable motor and gearbox were selected in order to rotate the camera.

9.6 Motor Selection and Gearbox Design

The selected motor is a Pololu metal gearmotor. This motor consists of a 12 V brushed DC motor combined with a 99:1 metal gearbox [33]. It also provides a quadrature encoder on the motor shaft, which can be used to control the output rotational speed of the motor.

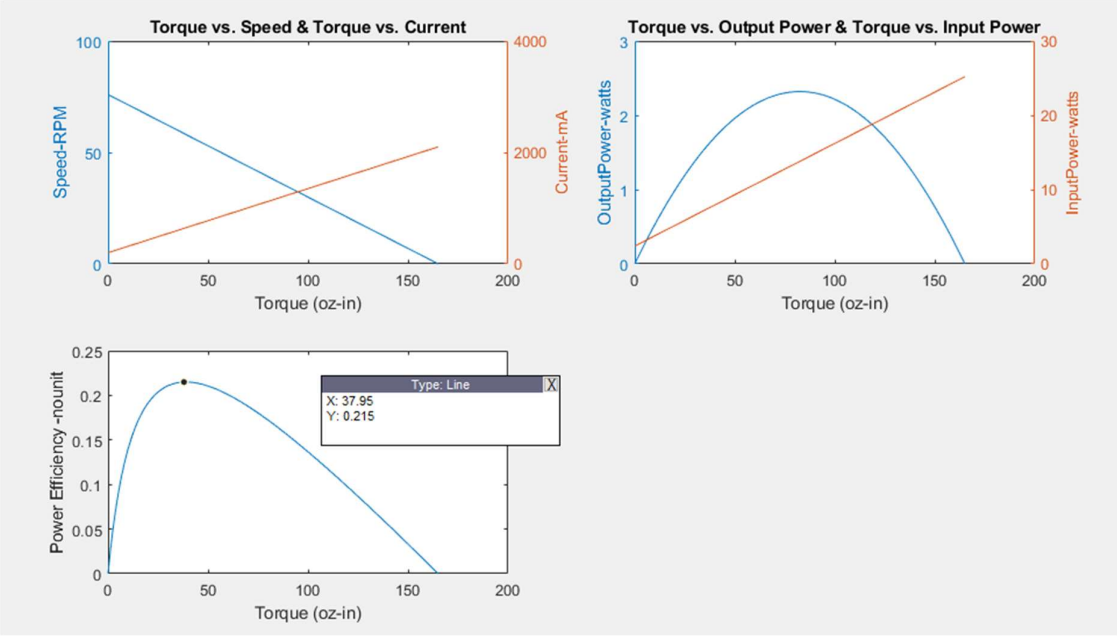

Figure 9.2: The Pololu Motor Performance Curve

Figure 9.2 is a MATLAB generated performance curve of the selected motor showing the no load and stall performance. The justification was made under the assumption that the efficiency of the motor was maximized.

The maximum efficiency of the motor is 0.215 and the corresponding torque is about 38 oz-in. Then, the rotational speed at max efficiency, gear ratio of gearbox and actual output were calculated as shown below.

$$Max\ efficiency\ torque\ =\ 38\ oz \cdot in\ =\ 2.7\ kg \cdot cm^2 \cdot s^{-2}$$
$$=\ 38 \div 165\ =\ 23\%\ stall\ torque$$
$$Max\ efficiency\ rotational\ speed\ =\ (1\ -\ 0.23)\ *\ 76\ rpm\ =\ 59\ rpm$$
$$Gear\ ratio\ =\ Win\ /\ Wout\ =\ 59\ rpm \div\ 19.5\ rpm\ =\ 3$$
$$Acutal\ output\ torque\ =\ 2.7\ kg \cdot cm^2 \cdot s^{-2}\ *\ 3$$
$$=\ 8.1\ kg \cdot cm^2 \cdot s^{-2}$$
$$Acutal\ output\ rotational\ speed\ =\ 59\ rpm \div 3\ =19.7\ rpm$$

The calculated gear ratio of the gearbox is three. Both actual output torque and rotational speed is enough to spin the camera to detect the pedestrians.

9.7 Camera Turntable Assembly CAD Design

With the specific model of motor being selected based on calculation, a housing for the motor, a holder for camera, and a turning structure are designed using SolidWorks. The isometric view of the design is shown below in Figure 9.3. Additionally, to better appreciate the design, a section view of the design is also shown in Figure 9.4.

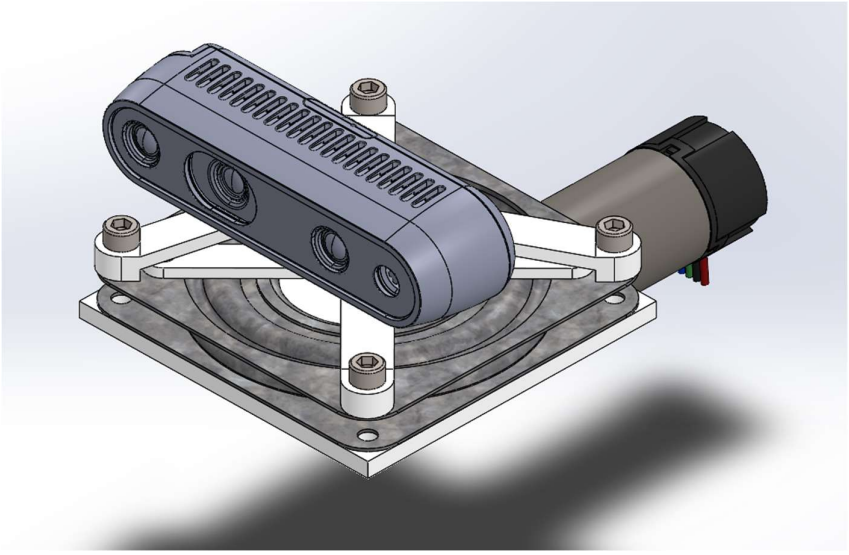

Figure 9.3: Isometric view of camera turntable assembly

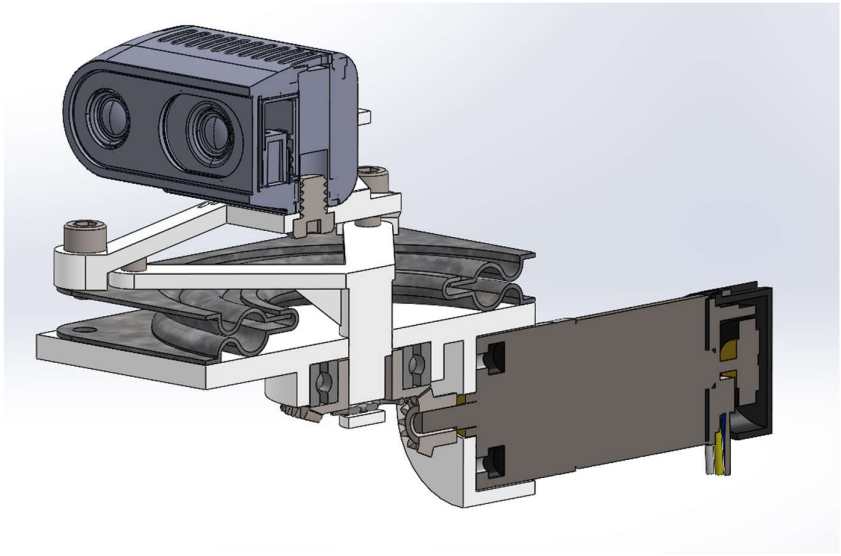

Figure 9.4: Section view of the assembly

The overall design goal is an assembly which can steadily and smoothly rotate the camera to different angles with a high rate of acceleration, driven by a motor. To accomplish the design goal, the component breakdown is introduced below.

As shown in Table 9.3 and Figure 9.5, the design consists of seven components.

Table 9.3: The List of Components in the Assembly

Number shown on drawing	Parts Name
1	Camera Holder
2	Gear-Holder Linkage
3	Turntable
4	Bevel Gear System
5	Camera
6	Transfer Case
7	Motor

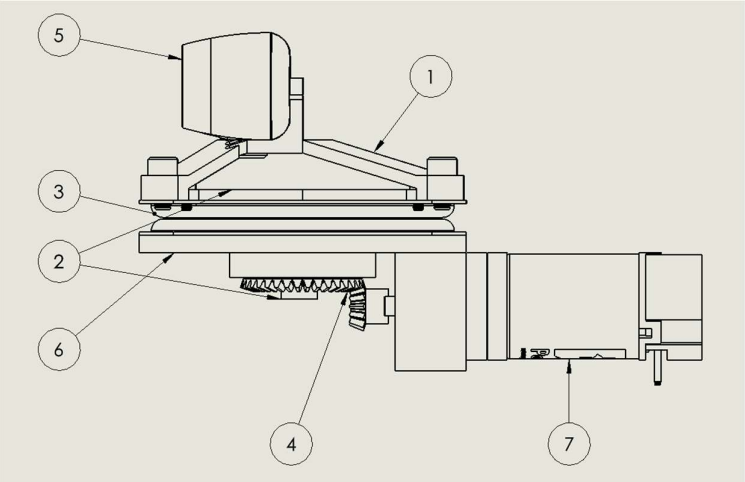

Figure 9.5: Seven components in the design

Camera holder

The design requirement of the camera holder is to lock the camera in place at a rapid acceleration. The included tripod in the box with the camera uses a friction pad and a 1/4”-20 screw to lock the camera. However, based on the experience with the included tripod, its design failed to secure the camera in place at rapid acceleration, so an improved design is needed.

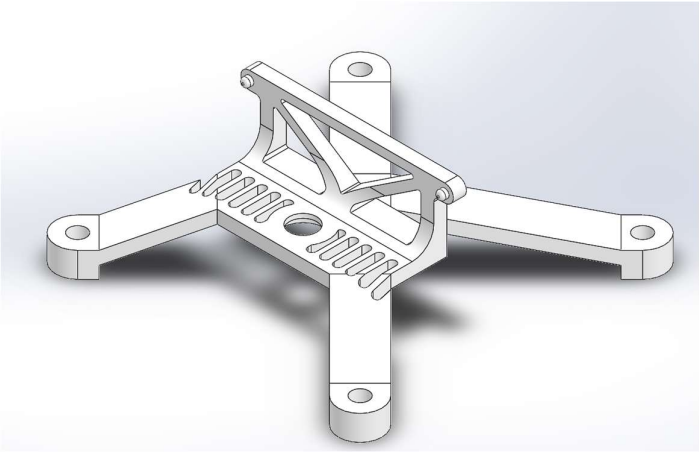

Figure 9.6: Design of a camera holder

One more detail to notice is that the D435i camera generated lots of heat during operation, and there are arrays of ventilation holes for passive convective cooling. Thus, ventilation holes were designed on the camera holder so that hot air from the bottom of the camera can move freely. The camera holder was also lifted with sloped legs to avoid blocking the field of view of the camera. The camera holder was bolted onto the turntable with four #8-32 screws in each leg for securing.

Gear-Holder Linkage

The task of the linkage is to transmit the torque from bevel gear to the turntable/camera holder assembly. The linkage is locked with bevel gear using a D-shaped key. It is bolted to the turntable with four #4-40 screws in each corner.

The linkage is inserted to a hole on the transfer case. For better alignment and more stable turning, the linkage is inter-locked by bevel gear, which latter was secured by a bearing. Finally, the linkage is also locked with the gear with cotter pins for improved security in horizontal direction.

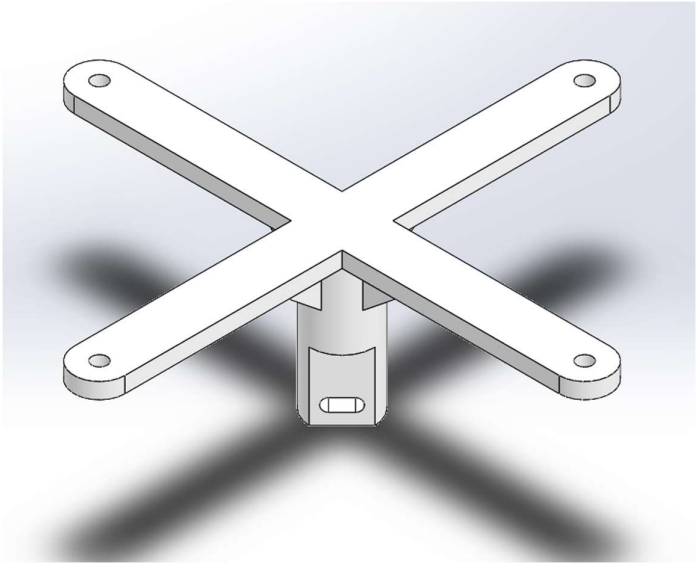

Figure 9.7: Design of a gear-holder linkage

Turntable

The turntable was chosen to be a part from McMaster-Carr. It is 3” by 3” square with balls and lubrication for smoother turning. The part was chosen because it has an adequately sized base and a hole in the middle and the hole allows a simple linkage design to transmit torque and rotate without complicated gearboxes.

Bevel gear/ bevel pinion

The bevel gear was chosen to change the direction of torque 90 degrees, with a 3:1 gear ratio. The gear and pinion have a 0.8 module, 20-degree pressure angle, and 36 and 12 teeth respectively. The change of direction of transition allows a horizontal placement of the motor, for better installation and lower overall height. Both gear and pinion have a D-shaped key to lock with respective axles and held in place with constraints of transfer case and linkage assembly.

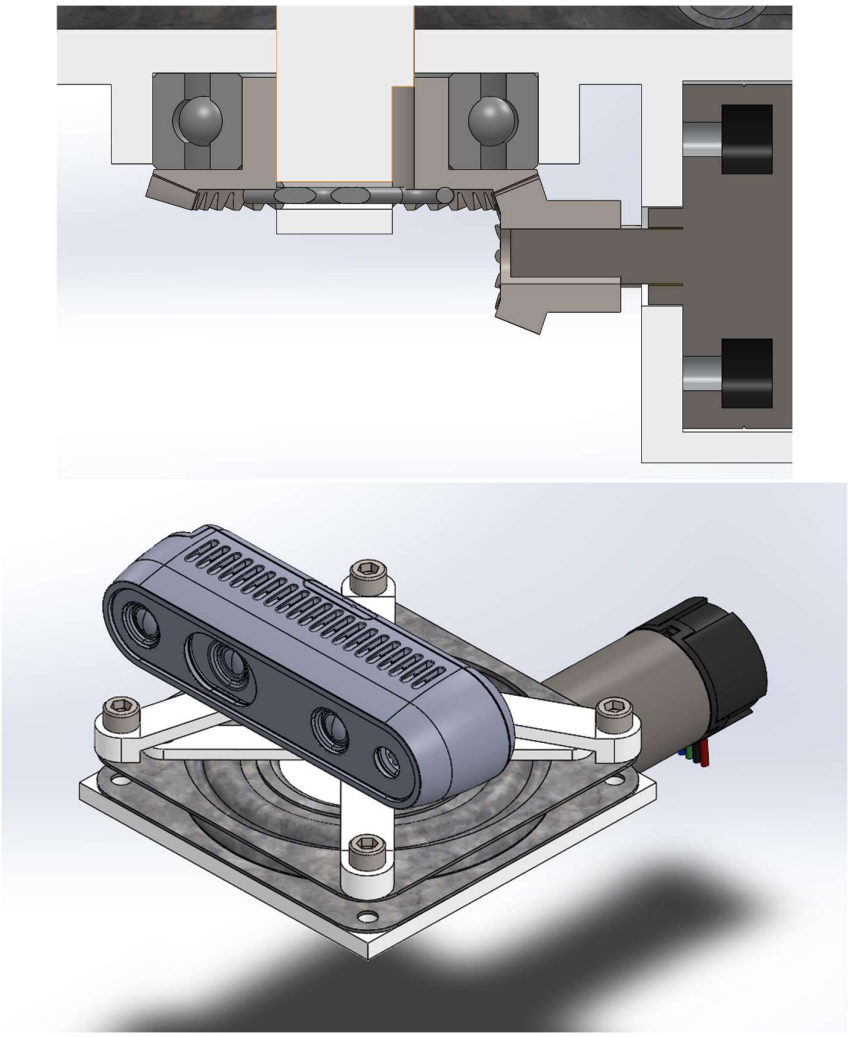

Figure 9.8: Bevel gear system and bearing

9.8 Simulation and Verification

A motion study was conducted in SolidWork using the Motion Analysis toolbox. The input of the motor shaft is given using an expression of angle in the time domain. In other words, the angle of the shaft at a given time is given to the simulation. A sensor was used to monitor the torque value on the motor shaft, which is used to verify the theoretical calculation and feasibility of design. The expression of angle in time domain was derived by solving differential equation (1), and yielding the expression (4).

$$\frac{d\theta}{dt} = \frac{cos^2(\theta)}{r}\frac{da}{dt} \quad (1)$$

$$sec^2(\theta)d\theta = \frac{1}{r}\frac{da}{dt}dt$$

$$\int sec^2(\theta)d\theta = \int \frac{1}{r}\frac{da}{dt}dt$$

$$tan(\theta) = \frac{1}{r}\frac{da}{dt}t + C$$

$$\theta = tan^{-1}\left(\frac{1}{r}\frac{da}{dt}t + C\right) (4)$$

An initial condition is necessary to solve constant c, since an $\theta = \pi/2$ is unsolvable at any time, so another initial condition is needed. The initial condition was solved by assigning the encountering duration, which is the time duration the camera took to turn from its initial pose to $\theta = 0$. The encounter duration was set to 5 second, because it gives a good overview of the turning process without too much time spent that camera is looking nearly straight ahead. It yields a c = 20. Result in the following expression of $\theta$with variable t.

$$\theta = tan^{-1}(4t + 20) \ (5)$$

Using the expression (5) as motor position input, setting two gears as solid contacting bodies for analyzing gear contact, and applying a gravitational field, the simulation setup is completed.

9.9 Result

As shown in Figure 9.9, The analyzing result was given by SolidWorks, and a calculated result was plotted along with the simulated result.

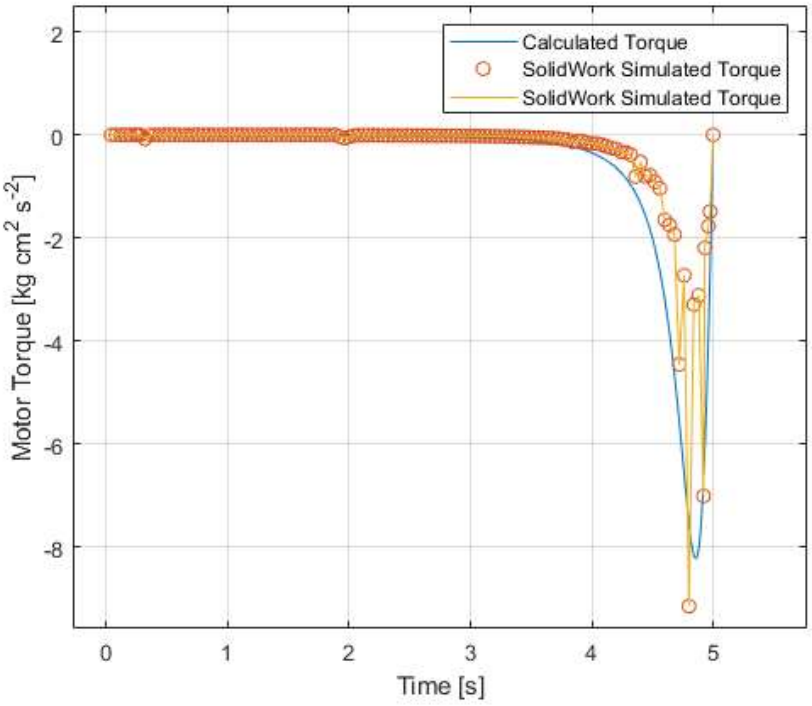

Figure 9.9: The simulated result from SolidWorks

Since the force analysis resolution from SolidWorks is limited by computational power of the computer, the simulated result is not a smooth line. Nevertheless, , consistency with the calculated result is obvious.

However, it is noticeable that the peak motor torque was higher in the simulated result. One possible reason is simulation accuracy and resolution could be improved. Another possible reason is the frictional force between each moving part, which is neglected in calculation but simulated in SolidWorks. The potential solution could be to add lubrication to the turntable and the linkage to minimize the frictional force in the system.

Since the speed of rotation of the motor, and hence the turntable, can be controlled by PWM signal, the motion analysis is needed to verify whether the motor can provide sufficient torque at a certain rpm range with a current draw restriction. From the simulated result, the design requirement is met that the motor can provide a sufficient torque and turning speed, thus being able to track the person in the hypothetical encounter scenario.

10 Conclusions

The team successfully utilized TurtleBot2 with RealSense Camera and Velodyne Lidar to achieve building maps, navigation with existing maps, and pedestrian detection and avoidance. The robot could successfully detect a walking pedestrian or multiple standing pedestrians and update its navigation path accordingly. Problems such as system processing speed and the navigation accuracy occurred during the testing and sometimes affected the robot's accuracy. Potential solutions such as altering the computer vision models, updating a faster computing platform, and adding a spinning platform for the camera to ameliorate the robot's achievement were suggested.

11 Appendix A

MATLAB Code for Generating Motor Performance Curve

```
function pololuMotorPlotGenAAMv2
   clc;
   discreteBins = 500; %We will use this number of bins for plotting and calculating all
functions such as Torque, speed etc.
% Input part of the main function
   StallTorque = input('Please enter the stall torque in oz-inch [17]: ');
   StallCurrent = input('Please enter the stall current in mA [700]: ');
   RatedVoltage = input('Please enter the rated voltage in Volts [6]: ');
   NoLoadCurrent = input('Please enter the free run currennt in mA [40]: ');
   NoLoadSpeed = input('Please enter the free run speed in RPM [290]: ');

   %Some basic input error checking is here.
   if or(not(isfloat(StallTorque)), isempty(StallTorque))
      StallTorque = 17;
      fprintf('\nUsing default value for StallTorque');
   end
   if or(not(isfloat(StallCurrent)), isempty(StallCurrent))
      StallCurrent = 700;
      fprintf('\nUsing default value for StallCurrent');
   end
   if or(not(isfloat(RatedVoltage)), isempty(RatedVoltage))
      RatedVoltage = 6;
      fprintf('\nUsing default value for RatedVoltage');
   end
   if or(not(isfloat(NoLoadCurrent)), isempty(NoLoadCurrent))
      NoLoadCurrent = 40;
      fprintf('\nUsing default value for NoLoadCurrent');
   end
   if or(not(isfloat(NoLoadSpeed)), isempty(NoLoadSpeed))
      NoLoadSpeed = 290;
      fprintf('\nUsing default value for NoLoadSpeed');
   end
   %

   %Here we calculate basic stuff to get all the variables and outputs.
   Resistance = RatedVoltage / (StallCurrent/1000);
   %Torque line
   TorqueLine = 0:(StallTorque/discreteBins):StallTorque;
   %Current Line
   CurrentLine = NoLoadCurrent:(StallCurrent-NoLoadCurrent)/discreteBins:StallCurrent;
```

```
    %Speed Line
    SpeedLine = NoLoadSpeed: (0-NoLoadSpeed)/discreteBins : 0;
    % Torque Constant in Torque per current is
    SlopeOfTorqueVsCurrent = (StallCurrent - NoLoadCurrent) / (StallTorque);


    %Output Mechanical Power in watts is Torque * Speed * 0.00074 watts
    OutputPower = 0.00074 * TorqueLine .* SpeedLine;
    %Input Electrical Power to the motor is Voltage * Current
    InputPower = CurrentLine * RatedVoltage / 1000; %We are dividing by 1000 as the input
was in mA and we need power in Watts.

%Plot part of the functions
    subplot(2,2,1)
    [hAx, hLine1, hLine2] = plotyy([0 StallTorque], [NoLoadSpeed 0], [0 StallTorque],
[NoLoadCurrent StallCurrent]); %This is the TorqueLoad vs. Motor Speed graph

    title('Torque vs. Speed & Torque vs. Current');
    xlabel('Torque (oz-in)');
    ylabel(hAx(1), 'Speed-RPM');
    ylabel(hAx(2), 'Current-mA');

    %This is the plot of the Output Mechanical power in watts vs. Input
    %Electrical power in Watts.
    subplot(2,2,2);

    [h2Ax, h2Line1, h2Line2] = plotyy(TorqueLine, OutputPower, TorqueLine, InputPower);
    xlabel('Torque (oz-in)');
    ylabel(h2Ax(1), 'OutputPower-watts');
    ylabel(h2Ax(2), 'InputPower-watts');
    title('Torque vs. Output Power & Torque vs. Input Power');

    %This is the plot of the Power Efficiency of the motor.
    subplot(2,2,3);
    PowerEff = OutputPower ./ InputPower;
    plot(TorqueLine, PowerEff);
    xlabel('Torque (oz-in)');
    ylabel('Power Efficiency -nounit');

    %Output information part of the function
    fprintf('\n\n\nSlope of TorqueVsCurrent is %f. The recprocal is %f\n',
SlopeOfTorqueVsCurrent, (1/SlopeOfTorqueVsCurrent));
    %Max Output Power is at
    [V,I] = max(OutputPower);
```

```
    fprintf('Maximum output mechanical power is %f(watts).\nThis happens at the Torque
load of %f(oz-in), with Current %f(mA)\n', OutputPower(I), TorqueLine(I), CurrentLine(I));
    fprintf('Resistance of the motor is %f (ohms)\n', Resistance);

end
```

12 Bibliography

[1] A. A. Staff, “What robots do (and don't do) at Amazon fulfilment centres,” *UK About Amazon*, 31-Jul-2019. [Online]. Available: https://www.aboutamazon.co.uk/amazon-fulfilment/what-robots-do-and-dont-do-at-amazon-fulfilment-centres. [Accessed: 04-Aug-2020].

[2] D. O. Team, “Bots by the numbers: Facts and figures about robotics at Amazon,” *UK Day One Blog*, 14-Jan-2019. [Online]. Available: https://blog.aboutamazon.co.uk/bots-by-the-numbers-facts-and-figures-about-robotics-at-amazon. [Accessed: 05-Aug-2020].

[3] M. Lazarte, “Robots and humans can work together with new ISO guidance,” *ISO*, 08-Mar-2016. [Online]. Available: https://www.iso.org/news/2016/03/Ref2057.html. [Accessed: 05-Aug-2020].

[4] D. O. Team, “Recognising World Day for Safety and Health at Work 2019,” *UK Day One Blog*, 03-May-2019. [Online]. Available: https://blog.aboutamazon.co.uk/recognising-world-day-for-safety-and-health-at-work-2019. [Accessed: 05-Aug-2020].

[5] D. O. Team, “Meet Amazon's ergonomics expert, Raz Osman,” *UK Day One Blog*, 28-Nov-2019. [Online]. Available: https://blog.aboutamazon.co.uk/working-at-amazon/meet-amazons-ergonomics-expert-raz-osman. [Accessed: 05-Aug-2020].

[6] Mattieu Chevrier, “How sensor data is powering AI in robotics,” TEXAS INSTRUMENTS, Jan-2019. [Online]. Available: https://www.ti.com/lit/wp/sszy036/sszy036.pdf. [Accessed: 06-Aug-2020].

[7] N. O. and A. A. US Department of Commerce, “What is LIDAR,” *NOAA's National Ocean Service*, 01-Oct-2012. [Online]. Available: https://oceanservice.noaa.gov/facts/lidar.html. [Accessed: 05-Aug-2020].

[8] Paul F. McManamon, “Introduction to LiDAR,” in *LiDAR Technologies and Systems*, 2019. [Online]. Available: https://www.spiedigitallibrary.org/eBooks/PM/LiDAR-Technologies-and-Systems/Chapter1/Introduction-to-LiDAR/10.1117/3.2518254.ch1

[9] Tarleton Gillespie, “Algorithm” in *Digital Keywords*, 2016. [Online]. Available: https://www.jstor.org/stable/j.ctvct0023?turn_away=true

[10] Alessandro Gasparetto et al. Path Planning and Trajectory Planning Algorithms: A General Overview. DOI: 10.1007/978-3-319-14705-5_1. [Online]. Available: https://www.researchgate.net/publication/282955967_Path_Planning_and_Trajectory_Planning_Algorithms_A_General_Overview

[11] V. Blanz et al. “Comparison of view-based object recognition algorithms using realistic 3D models” in *International Conference on Artificial Neural Networks*, 2005. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-61510-5_45

[12] Jang-Ho Cho et al. “A Real-Time Obstacle Avoidance Method for Autonomous Vehicles Using an Obstacle-Dependent Gaussian Potential Field”. Journal of Advanced Transportation, 2018. [Online]. Available: https://www.hindawi.com/journals/jat/2018/5041401/. [Accessed: 16-Aug-2020].
[13] Yutaka Hiroi and Akinori Ito, “A Pedestrian Avoidance Method Considering Personal Space for a Guide Robot,” *MDPI*, Sep-19-2019. [Online]. Available: https://www.mdpi.com/2218-6581/8/4/97. [Accessed: 16-Aug-2020].
[14] “Technology,” *BostonDynamics*. [Online]. Available: https://www.bostondynamics.com/spot/technology. [Accessed: 16-Aug-2020].
[15] “S6 MaxV,” *Roborock*. [Online]. Available: https://us.roborock.com/pages/roborock-s6-maxv. [Accessed: 16-Aug-2020].
[16] “Audi adaptive cruise assist,” *Audi*. [Online]. Available: https://www.audiusa.com/models/audi-a8?tile=driver-assistance. [Accessed: 16-Aug-2020].
[17] “Model 3,” *Tesla*. [Online]. Available: https://www.tesla.com/model3. [Accessed: 16-Aug-2020].
[18] “Collision Detection,” ABB. [Online]. Available: https://library.e.abb.com/public/d4708d0240be2966c125772f00528ca9/Collision%20det%20PR10044EN_R2.pdf. [Accessed: 16-Aug-2020].
[19] “Jetson TX2 Module,” *NVIDIA Developer*, 14-Aug-2019. [Online]. Available: https://developer.nvidia.com/embedded/jetson-tx2. [Accessed: 05-Aug-2020].
[20] “Depth Camera D435i,” *Intel® RealSense™ Depth and Tracking Cameras*, 16-Jun-2020. [Online]. Available: https://www.intelrealsense.com/depth-camera-d435i/. [Accessed: 05-Aug-2020].
[21] “Velodyne Puck VLP-16 Sensor: LiDAR,” *AutonomouStuff*. [Online]. Available: https://autonomoustuff.com/product/velodyne-puck-vlp-16/. [Accessed: 05-Aug-2020].
[22] Introduction *to A**. [Online]. Available: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html. [Accessed: 06-Aug-2020].
[23]J Redmon and A Farhadi, “YOLOv3: An Incremental Improvement,” 2018. [Online]. Available: https://pjreddie.com/publications/
[24] “About ROS,” *ROS.org*. [Online]. Available: https://www.ros.org/about-ros/. [Accessed: 05-Aug-2020].
[25] “Wiki,” *ros.org*. [Online]. Available: http://wiki.ros.org/melodic. [Accessed: 05-Aug-2020].
[26] “TurtleBot2,” *robots.ros.org*. [Online]. Available: https://robots.ros.org/turtlebot/. [Accessed: 05-Aug-2020].
[27] “Core Components,” *ROS.org*. [Online]. Available: https://www.ros.org/core-components/#:~:text=In addition to the core,Standard Message Definitions for Robots. [Accessed: 05-Aug-2020].

[28] Osrf, “Why Gazebo?,” *gazebo*. [Online]. Available: http://gazebosim.org/. [Accessed: 05-Aug-2020].
[29] Orff, “ROS overview,” *gazebo*. [Online]. Available: http://gazebosim.org/tutorials?tut=ros_overview. [Accessed: 05-Aug-2020].
[30] Caroline Pantofaru, “leg_detector,” *ROS.org.* [Online]. Available: http://wiki.ros.org/leg_detector. [Accessed: 05-Aug-2020].
[31] Praveen Palanisamy, “multi_object_tracking_lidar,” *ROS.org.* [Online]. Available: http://wiki.ros.org/multi_object_tracking_lidar. [Accessed: 05-Aug-2020].
[32] “Intel RealSense Depth Camera D400-Series Datasheet.” RealSense Technology, Sep-2017.
[33] “Pololu - 99:1 Metal Gearmotor 25Dx69L mm MP 12V with 48 CPR Encoder,” *Pololu Robotics & Electronics*. [Online]. Available: https://www.pololu.com/product/4867. [Accessed: 09-Aug-2020].