FACULTY OF ENGINEERING AND APPLIED SCIENCE

METE 4300U: Introduction to Mobile Robotics
Winter 2021

Instructor: Scott Nokleby, Ph.D., PEng

TA:Lillian Goodwin

Group #: 6

Milestone #: 4

Requirements: SLAM and Package Retrieval for TurtleBot3 WafflePi

LAB GROUP MEMBERS				
#	Surname	Name	ID	Signature
1	Hunt	Myles	100623925	M.H
2	Jong	Connor	100621435	Connor Jong
3	Hulcoop	Hudson	100621819	Hudson Hulcoop
4	Medri	Evan	100620149	Evan Medri
5	Weber	Kenneth	100618149	Kenneth Weber

Table of Contents

Abstract

The following report outlines the design, construction and software development of a TurtleBot3 and corresponding ROS programs. Before explaining the design, the requirements and prerequisite background information for this project are discussed to provide a clear goal for our solution. After establishing the goals of the project and establishing an in depth breakdown of the functional requirements, brainstorming for concepts used for Milestone Four are evaluated. This evaluation determined the most effective gripper from the team's designs, a custom K'nex gripper, made of mostly OEM parts. Along with the custom built gripper design the previous control algorithms were implemented to support the new functionality of the gripper. The updated control algorithms for SLAM, Path Planning and Aruco Detection are then discussed, in addition to how they function with the gripper. After the implementation of new algorithms and a gripper, the test plan was created in order to evaluate the TurtleBot's performance at completing Milestone Four. After rigorous testing, it was determined that the robot was in a good state, and was able to complete all necessary tasks during the live demo. Following the results, the final CAD drawings and required diagrams such as BOM and circuit diagrams are documented at the end. These provide the necessary tools and dimensions to recreate the results achieved across the four Milestones.

Introduction

Milestone Four involves the programming of a TurtleBot3 ROS on Ubuntu 16.04. The program must allow the TurtleBot3 to autonomously navigate a room that is roughly 15 m^2, while mapping its environment through the use of SLAM. After mapping its environment, and any packages along the way, it is then required to navigate back to its starting location with as much accuracy as possible. All of these tasks have been successfully implemented and demonstrated in the previous three milestones. The new part for Milestone Four, is that the TurtleBot3 must be able to pick up a package and bring it home.

Requirements

The TurtleBot3 is required to explore an unknown area and create a two dimensional SLAM map. The TurtleBot3 must also be able to identify packages on the map. Any method is acceptable for accomplishing this and the sensors provided can be used or new ones purchased. Once a package is identified the position it is in must be properly marked on the 2D SLAM map being created. The TurtleBot3 must then retrieve this package with an acceptable manipulator and lift it off of the ground. After picking up the package and all frontiers have been explored, the robot must then navigate home, to its initial starting point.

Identify customers

The customers targeted by this developmental project include e-commerce giants such as Amazon, as well as any company that may be interested in package retrieval and delivery, even in an industrial setting. The small scale prototype assembled by the team serves as a proof of concept that the current hardware and software architecture can be expanded to address customer needs in a factory environment. There are a variety of other industries and hence corresponding customers which utilize mobile robots with mapping capabilities, however recent progress such as the addition of a robotic gripper onto the mobile bot to acquire and hold packages furthers attractability to logistic or manufacturing operations.

Customer requirements

The customers requirements for this project includes the following. The bot must be able to accurately and efficiently map its operating environment by using SLAM techniques controlled by a ROS program. The ability to locate and mark specific packages effectively in the operating area while varying obstacles are present. A final customer requirement is the ability to pick and relocate packages back to a starting position. Fulfillment of the above requirements will signify a successful project design and implementation. These customer requirements, as well as engineering requirements stemming from them, are listed in the table below.

Customer Requirement	Engineering Requirement	Acceptance Criteria
The robot must be able to map surroundings accurately.	Must run SLAM using robot sensor feed as ROS node.	Map passes a visual quality inspection.
The robot must move autonomously.	Explore lite package runs on TurtleBot sensor feed.	Final position after a test run is <5cm from origin (home) and orientation is within 5 degrees.
Packages must be detectable, and marked on the map.	Robust fiducial marker detection at various angles of approach to the package.	No false positives, minimal number of impactful false negatives (such that runs fail).
Robot must have the ability to pick up, hold, and drop off packages.	Design of a gripper capable of securing and lifting the boxes.	Gripper picks up box from a variety of approach angles without fail.

Table 1: Customer and Engineering Requirements

Engineering Specifications

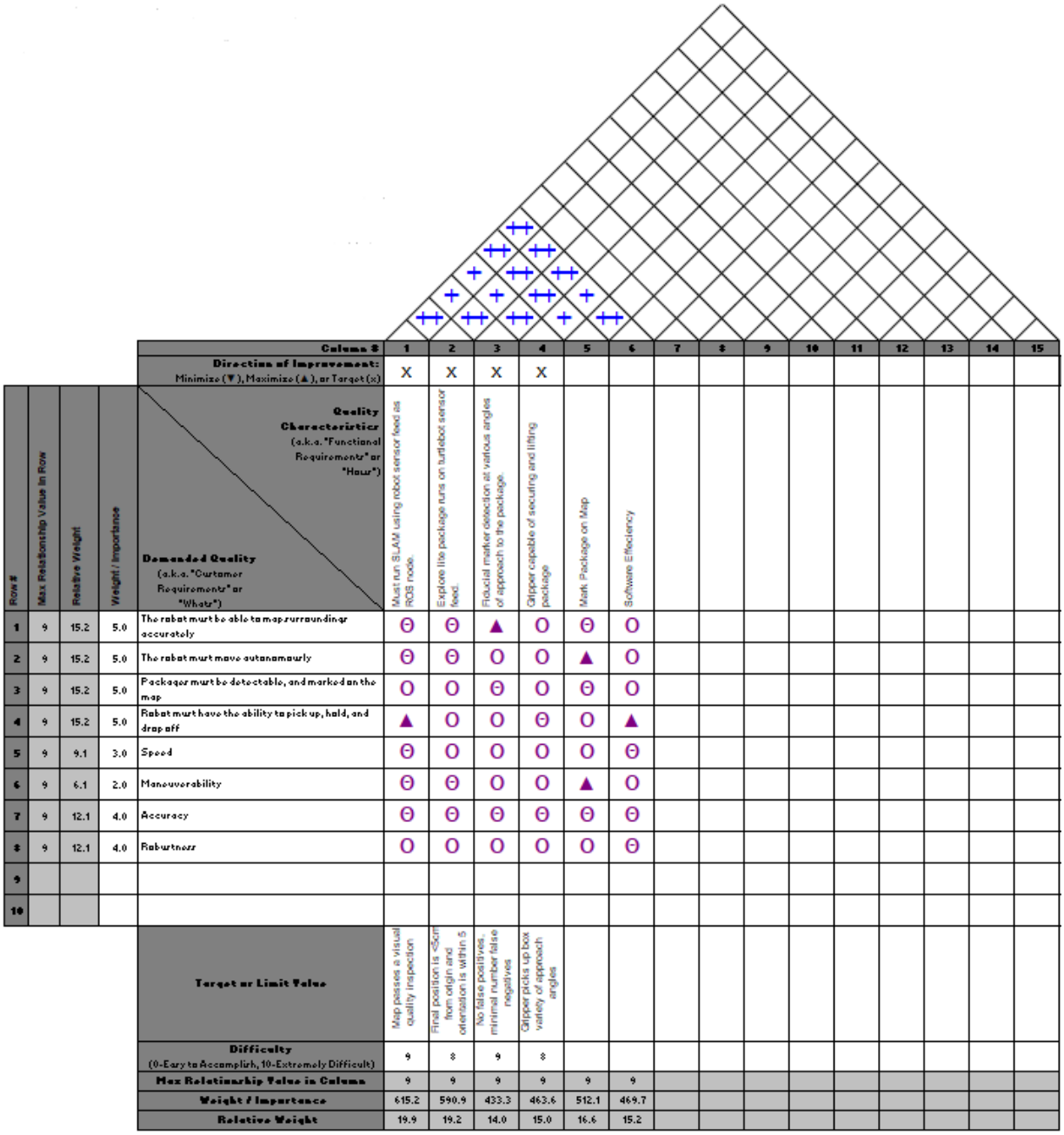
In order to meet the requirements listed above, the TurtleBot3 needs to have at least two independent motors to allow for basic driving, pivots, and turning. The robot must also be equipped with a sensor and camera that is capable of interpreting environmental data. This sensor is necessary to perform SLAM operations. For this project, a top mounted Lidar will be used as well as a Raspberry Pi camera module. Another physical requirement of the project was to be able to wirelessly communicate with an external computer to allow for remote starting, as well as programming of the robot. Additionally, the robot must be small enough to navigate through doorways and around obstacles such as table legs. If the robot was too large, it would not be able to drive to all of the points of interest, and therefore making a complete map of its surroundings may be impossible.

In terms of software, the robot must be running a SLAM program capable of producing a 2D map from the Lidar information. The program must also be able to use camera vision to recognize ArUco markers in order to determine the pose of packages within the map. The use of an exploration package is needed to navigate through the environment while avoiding obstacles and looking for the packages. For this project, the first package chosen was “explore lite”. It uses a frontier based exploration algorithm to mark areas that need to be mapped. After all of these

frontiers have been explored and packages have been properly marked, the robot must then navigate to its starting point. The next package selected was “aruco detect”, which is a part of “Fiducial SLAM”. Aruco detect allows a monocular camera to automatically recognize ArUco markers and place a bounding box around them. It also marks the pose of the marker, and has the ability to look for different types of ArUco markers if necessary.

Lastly, there is the problem of picking up objects of known position, orientation, and size. The most critical design decision pertaining to this challenge is the selection of the mechanism responsible for securing the object. The gripper must not impede the movement of the robot, and additionally, when the package is retrieved and being held, should not block critical sensors required for localization on the return to the starting location.

A House of Qualities was created to properly record and determine the customer and technical requirements and the correlation between them.



Row #	Max Relationship Value in Row	Relative Weight	Weight / Importance	Demanded Quality (a.k.a. "Customer Requirements" or "Whats") \ Quality Characteristics (a.k.a. "Functional Requirements" or "Hows")	1 Must run SLAM using robot sensor feed as ROS node.	2 Explore life package runs on turtlebot sensor feed.	3 Fiducial marker detection at various angles of approach to the package.	4 Gripper capable of securing and lifting package	5 Mark Package on Map	6 Software Efficiency	7	8	9	10	11	12	13	14	15
				Direction of Improvement: Minimize (▼), Maximize (▲), or Target (x)	X	X	X	X											
1	9	15.2	5.0	The robot must be able to map surrounding accurately	Θ	Θ	▲	O	Θ	O									
2	9	15.2	5.0	The robot must move autonomously	Θ	Θ	O	O	▲	O									
3	9	15.2	5.0	Packages must be detectable, and marked on the map	O	O	Θ	O	Θ	O									
4	9	15.2	5.0	Robot must have the ability to pick up, hold, and drop off	▲	O	O	Θ	O	▲									
5	9	9.1	3.0	Speed	Θ	O	O	O	O	Θ									
6	9	6.1	2.0	Maneuverability	Θ	Θ	O	O	▲	O									
7	9	12.1	4.0	Accuracy	Θ	Θ	Θ	Θ	Θ	Θ									
8	9	12.1	4.0	Robustness	O	O	O	O	O	Θ									
9																			
10																			
				Target or Limit Value	Map passes a visual quality inspection	Final position is <5cm from origin and orientation is within 5	No false positives, minimal number false negatives	Gripper picks up box variety of approach angles											
				Difficulty (0-Easy to Accomplish, 10-Extremely Difficult)	9	8	9	8											
				Max Relationship Value in Column	9	9	9	9	9	9									
				Weight / Importance	615.2	590.9	433.3	463.6	512.1	469.7									
				Relative Weight	19.9	19.2	14.0	15.0	16.6	15.2									

Figure 1: House of Qualities

Background Search

The background research conducted for each Milestone will be presented below as well as some preliminary information.

SLAM

SLAM stands for Simultaneous Localization and Mapping. The two main concepts behind SLAM are mapping, and localization. SLAM is a type of program that makes use of sensors to gather information about the environment, and then uses that information to generate a map. In the case of this project, a Lidar is used to create a 2-D point cloud that represents the TurtleBot3's surroundings. The point cloud gets converted into a map, which can be used for the robot to understand where it is (localization), allowing for the use of path planning and obstacle avoidance.

ROS

ROS is the abbreviation for Robot Operating System. It is used to provide access to various existing robotics related code, removing the need to create programs from scratch. It allows the user to download, run, and modify existing programs with ease. Another advantage to utilizing ROS is that it allows the use of multiple programming languages between the packages. This is a large convenience when trying to design a system using packages from various developers.

Component Background

TurtleBot3

The TurtleBot3 is a customizable, modular, pre-built robot that is designed to provide a platform for performing autonomous navigation, mapping and exploration using ROS. It is equipped with a Raspberry Pi 3 for on board processing and communication, a Lidar to detect its surroundings and a camera for more robust feature identification [1].

Lidar

The Lidar that comes with the TurtleBot3 is a 360 Laser Distance Sensor LDS-01. This sensor is mounted on top of the turtle bot in the center, and spins rapidly, getting feedback from the laser as it reflects. Based on this information, the Lidar is used to judge the distance and mark the location of anything detected in its surroundings. The Lidar will not map anything that is lower than its vertical mounting location, so objects that are close to the floor may still be run into [1].

RaspberryPi 3

The Raspberry 3 Model B is the onboard processor of the TurtleBot3. It comes preinstalled with Ubuntu and ROS Kinetic. The Raspberry Pi is used as the onboard control system and processor for the TurtleBot. It allows for transmission to an external computer, to handle more computationally heavy tasks, and display important information [1].

Camera

The TurtleBot 3 comes equipped with a Raspberry Pi Camera Module v2.1. This is an 8 megapixel camera capable of streaming 1080p HD video at 30 fps. It has the option to lower the quality in exchange for higher FPS streams. The camera is normally mounted on the front of the robot, and enables the ability to view the environment from the perspective of the TurtleBot [1].

Milestone 1

The main concept behind this milestone is to utilize SLAM and navigation ROS packages to achieve autonomous exploration and mapping. Some existing examples of similar robots will be discussed in this section. Relevant real world examples, and existing ROS packages considered can be seen below, categorized by Milestone.

ROS SLAM Robot

The ROS SLAM Robot is an autonomous, programmable robot that is made by SuperDroid Robots. This robot is almost identical in function to the requirements for Milestone One. The robot uses an Odroid XU4, with ROS Melodic on Ubuntu 18.04. It has a top mounted YDLIDAR G4, allowing it to perform SLAM mapping in a similar manner to the TurtleBot3. The ROS SLAM Robot retails for $4260, it can be seen below in Figure 2 [2].

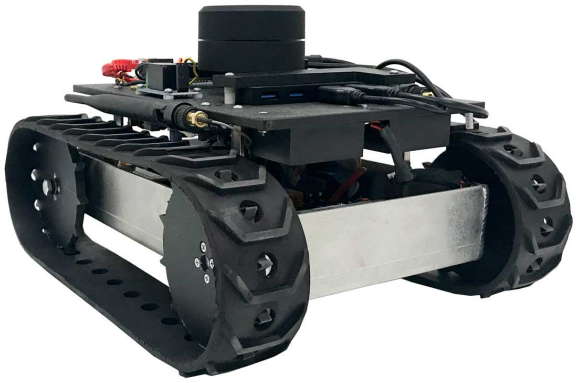


Figure 2: ROS SLAM Robot by SuperDroid Robots

Ecovacs DeepBot T8

The Ecovacs DeepBot is a consumer friendly take on an autonomous robot that can perform 2D SLAM. This is an autonomous cleaning robot that can mop and vacuum a house without user interference. It will avoid all obstacles, including harder to detect objects such as wires. The Deepbot also has many advanced features that are not seen explored in Milestone One. For example, it has additional sensors to detect ledges, is capable of multi floor mapping, is self emptying and automatically recharges. The Deepbot T8 sells for $799 on Amazon [3]. This is a good example of a practical application of the skills learned from Milestone One.

Figure 3: Ecovas Deepbot T8, Autonomous mop and vacuum with mapping and obstacle avoidance

Alternative ROS Packages:

Frontier_Exploration

This ROS package enables a robot to explore through the use of frontiers. The difference between frontier_exploration and explore_lite provides a costmap, while explore_lite does not. This makes it more difficult to change parameters for the package, making it less customizable. Moreover, having its own costmap causes it to be slightly more resource intensive, potentially slowing down performance [4]. For these reasons, explore_lite was chosen as the package to use for this project.

Milestone 2

The only difference between Milestone One and Milestone Two is that the robot has to identify and show the location of packages it has detected. The research from Milestone One is

still applicable for the core functionality of this Milestone, but new research was conducted for finding ArUco identification ROS packages.

Alternative ROS Packages:

Find_Object

This package uses object detection and various feature tracking algorithms to determine what is in view of the camera. It can determine the pose of objects, and has both 2D and 3D applications. The main reason that this package was not selected for the project was because it was deemed as overkill, and could slow down processing. This is because the feature detection algorithms run on everything in view of the camera. For this project, it would only be necessary to detect the packages, not the whole environment. Therefore the extra capability is simply wasted and could make it more difficult to extract important information.

Milestone 3

Considering that the work done for this milestone was mostly related to the surveying node and camera calibration, those will be the focus of the research. Unfortunately, no existing surveying nodes could be found that were usable for this application. For this reason, this section will discuss any resources that were found that assisted in the creation of the program.

ROS Wiki: Costmap 2D

This page on the ROS Wiki contains all necessary information about the Costmap 2D. One key term here is “cost”, which is important to both the occupancy grid and the costmaps. For the occupancy grid, the cost value ranges from 0 - 100, 0 meaning free space, and 100 meaning an obstacle is present. As the cost increases, the probability of colliding with an obstacle increases. The range 0 - 50 can usually be considered as free space, or likely free. A cost above 50 indicates that the robot is close to an obstacle, and could collide with it, depending on footprint size, orientation etc [5].

Camera Calibration Script

In past milestones, it was noted that if a package came into view at the edge of the input frame from the camera, the radial distortion of the lens would cause error in the placed location of the package on the map, To combat the effect lens have on inferring information about the external world, a standard extrinsic camera calibration was performed on the TurtleBot's onboard camera. Extrinsic calibrations serve as a method to extract parameters used in a pixel remapping function, which is applied to every received image from the camera sensor. These parameters (k_1, k_2, p_1, p_2, k_3) are generally referred to as distortion coefficients, and are used to model distortion as demonstrated in the equations below [7].

$$x_{corrected} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$
$$y_{corrected} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Camera lenses take all forms of shapes and sizes, and most introduce distortion separately in the image x and y direction. In addition to forming the model to correct radial distortion, the camera sensor has some other key parameters which are critical to correcting an incoming image. An intrinsic calibration, which is performed as a part of the full extrinsic calibration process, captures these key physical parameters of the sensor in a matrix referred to as the camera matrix, represented below.

$$camera\ matrix = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

With the function of the distortion and intrinsic camera parameters understood, the challenge of obtaining these parameters remains. For this project, a checkerboard pattern of known size was printed on a piece of paper, and then affixed to a rigid clipboard. Because the width of the internal squares of the checkerboard is known, the discrepancies between the known real world checkerboard square are compared to how it appears in the image. The OpenCV community has automated the process checkerboard pattern based extrinsic calibration, a modified version of the script which can be located in the formal opencv documentation was utilized to calibrate the camera.

Milestone 4

The focus of this Milestone was on creating and programming a gripper for the TurtleBot3. Various initial mechanical designs were explored before a standard two armed gripper design was selected. The TurtleBot3 would then need to search for a package, pick it up and return home. In order to assist this process, research was done for the design and programming of a gripper. In addition to this, the team needed to create a node that was capable of accurately driving close to the package, so that the gripper could pick it up reliably. Using previous experience and knowledge gained through the research process of the previous milestones, no additional information was required to develop the program for the robot to approach the package. However, research was required to control the gripper through the GPIO pins on the Raspberry Pi. This section will explain the resource used to help develop this program.

Servo.py

The script servo.py contains necessary information regarding the GPIO pins on the Raspberry Pi 3. Furthermore, it comes with pre-built functions that allow for accurate servo control, to a desired angle. This function is called angle_to_percent. When an angle is input, it performs calculations to control the PWM of the servo to adjust the position of the head.

Design Plan

Develop Tasks

Milestone 1

The goal of Milestone one is to research, test and select a SLAM program that is compatible with Ubuntu 16.04 and ROS Kinetic. After ensuring functionality of the SLAM program, the team wished to create a return home node that will run when mapping is completed, sending the TurtleBot to the starting location. In addition, some additional aspects of the TurtleBot and relevant ROS packages needed to be evaluated such as: camera quality, camera position, node functionality and node customizability. The work was split amongst multiple

teams for the following tasks: ROS package research, simulated tests, parameter research, future milestone research and physical testing.

Milestone 2

Milestone 2 required the TurtleBot3 to be able to identify and mark two packages on the SLAM map, without the use of the Lidar. In order to accomplish this task, the group decided to make use of ArUco markers. Therefore it was necessary to find usable ROS packages that could identify the pose of an ArUco marker. In addition, this pose needed to be communicated to the costmap, in order to be marked as an obstacle, so that the TurtleBot3 does not collide with the packages. To communicate the package positions to the costmap it was decided to create a virtual laser scan that would place obstacles where the packages are marked. To achieve these tasks, teams were assigned to ArUco package research, obstacle detection research, package construction, physical testing and simulated testing.

Milestone 3

Milestone 3 did not require any additional functionality to complete the task, so this time was used to optimize the current software and plan for Milestone 4. The most notable feature that was planned to be optimized was the calibration of the camera, to improve consistency of package marking. After verifying a successful calibration, the team was to rigorously test the existing TurtleBot3 in order to isolate and remove any unexpected outcomes.

Milestone 4

Milestone four requires the TurtleBot3 to pick up a package and bring it back to the starting location. This requires a gripper to be built and programmed, and new nodes to be created that can accurately move to the package in order to lift it off the floor. In order to finish this work, teams were assigned as follows: develop a node to move towards a package, develop a node that picks up the package when in range, construct a gripper, create a gripper control node, camera repositioning, physical testing.

Project Management

As this project was completed by a design team, a rough project schedule and work breakdown was created to help workflow and establish timelines (Tables 2, 3, 4, and 5).

Assigned Date	Jan 15, 2021
Milestone 1 (4 Weeks)	
Background Research and ROS Environment Setup Familiarization	Jan 15 - Jan 30th
Begin Testing and Building Solutions	Jan 30th - Feb 10th
Milestone 1 Final Testing	Feb 10th - Feb 11th
Demonstration Date	**Feb 12, 2021, 12:40 PM**
Group Report Writing Sessions	Feb 18th, 20th, 21st
PR #1 Due	**Feb 22, 2021, 5:00 PM**

Table 2: Milestone One Plan

Milestone 2 (4 Weeks)	
Build Physical Packages with Aruco Markers	Feb 15th, 2021
Background Research	Feb 18th, 2021
Aruco detection working	Feb 27th, 2021
Packages marked on map Improvements to previous code	March 4th, 2021
Milestone 2 Final Testing	March 5th, 2021
Demonstration Date	**March 5, 2021, 12:40 PM**
Group Report Writing Sessions	March 6th, 11th
PR #2 Due	**March 12, 2021, 5:00 PM**

Table 3: Milestone Two Plan

Milestone 3 (4 Weeks)	
Build Physical Obstacles	Feb 15th, 2021
Aruco detection improvements	March 17th, 2021
Improve obstacle avoidance	March 17th, 2021
Milestone 3 Final Testing	March 18th, 2021
Demonstration Date	**March 19, 2021, 12:40 PM**
Group Report Writing Sessions	March 6th, 11th
PR #3 Due	**March 26, 2021, 5:00 PM**

Table 4: Milestone Three Plan

Milestone 4 (4 Weeks)	
Build and Program Gripper Actions	March 30th, 2021
Attach and Program Package Retrieval	April 9th
Improve Gripping Action	April 10th,11th 2021
Milestone 4 Final Testing	April 11th, 2021
Demonstration Date	**April 12, 2021, 12:40 PM**
Group Report Writing Sessions	April 10th, 11th, 18th
PR #4 Due	**April 19, 2021, 5:00 PM**

Table 5: Milestone Four Plan

Functional Decomposition

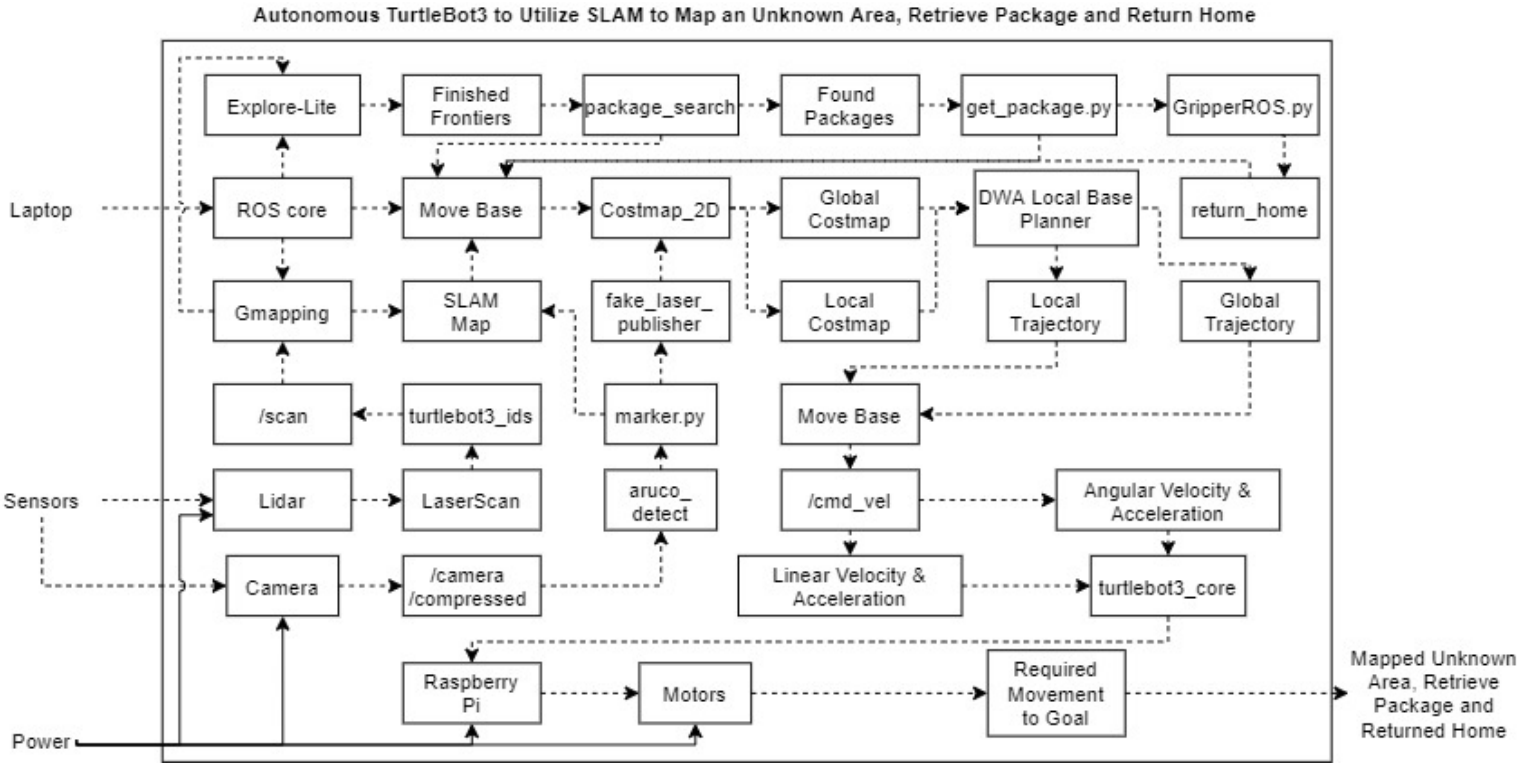


Figure 4: Functional Decomposition

UML Diagram

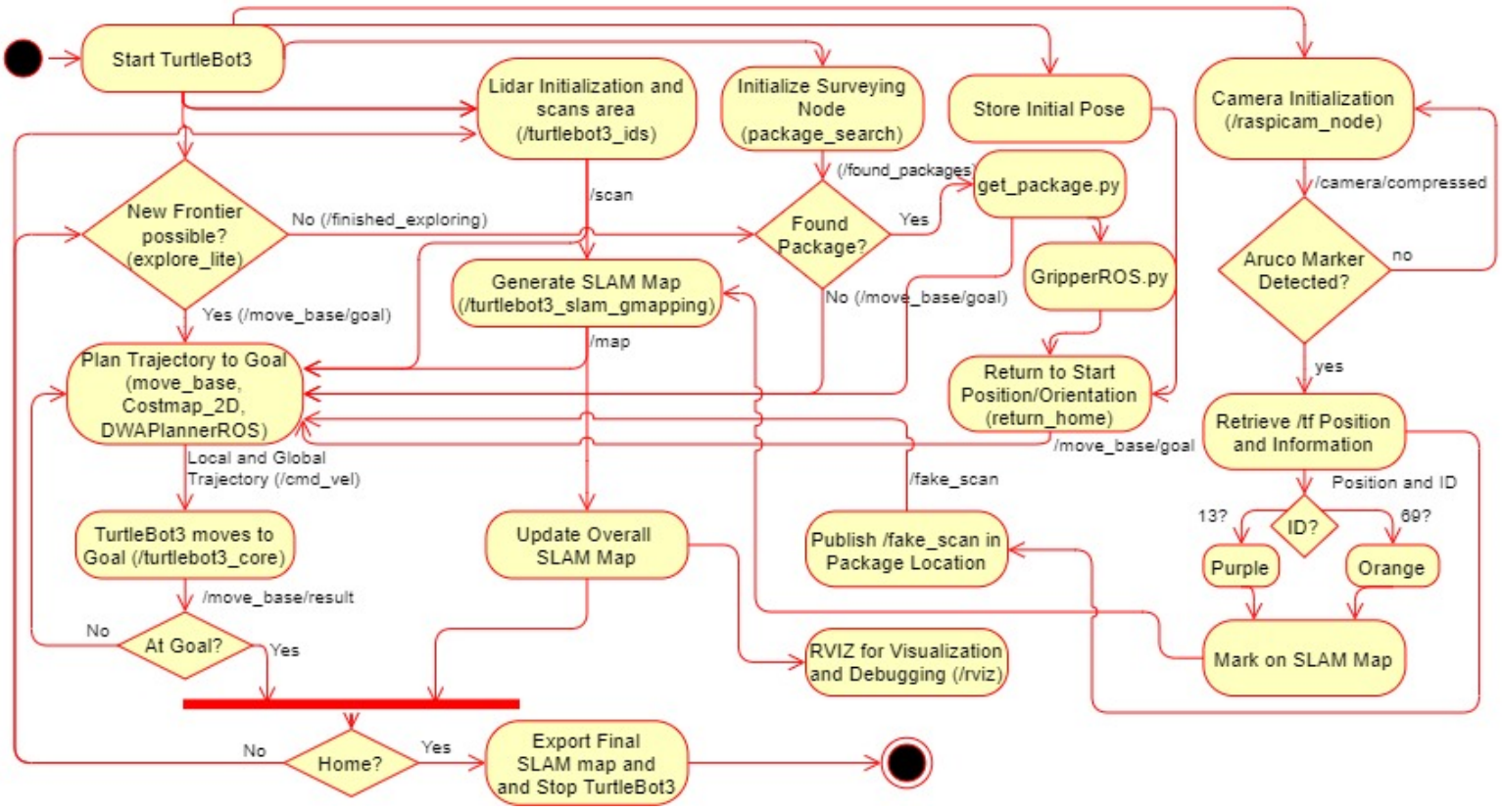


Figure 5: UML Diagram

Brainstorming

Conceptual Design: Milestone 4

The following chart is a table depicting all concepts that were discussed and an evaluation of their feasibility for the completion of Milestone four. Figures 6, 7, and 8 features sketches of some of the brainstormed concepts.

Concepts	Pros	Cons
Kinex Custom Gripper	Cheap, simple, modular, light, can drop/pickup	Requires precision
Off the shelf Gripper	Pre built, simple installation, can drop & pickup	Extra expenses, not easily modified
Magnets	Simple, cheap, small	Electrical interference, low range, can't drop, requires package modification
Velcro	Cheap, simple, light	Unreliable, can't drop, requires package modification
Vacuum Style (suction based)	Easy alignment	Complex, expensive, large

Table 6: Concept Generation / Pros & Cons

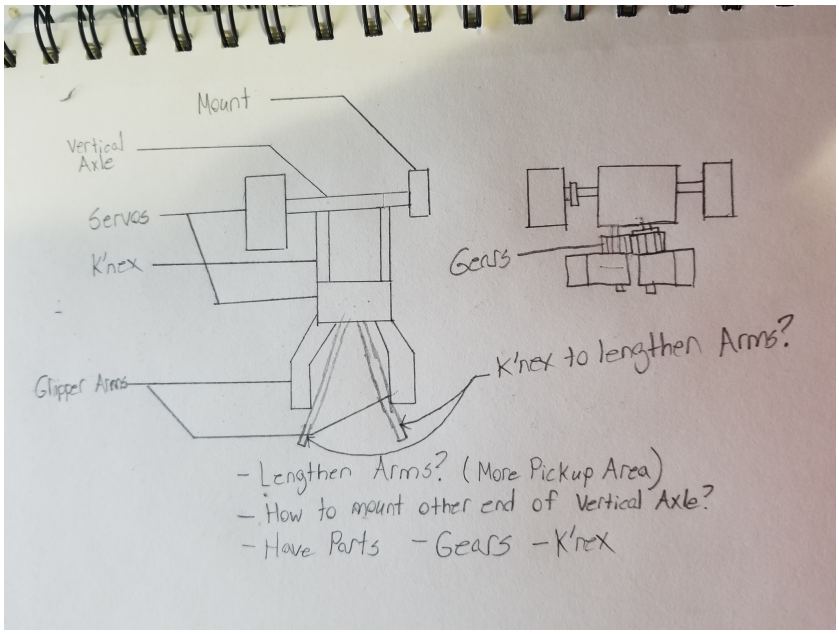


Figure 6: K'nex Gripper Concept Sketch

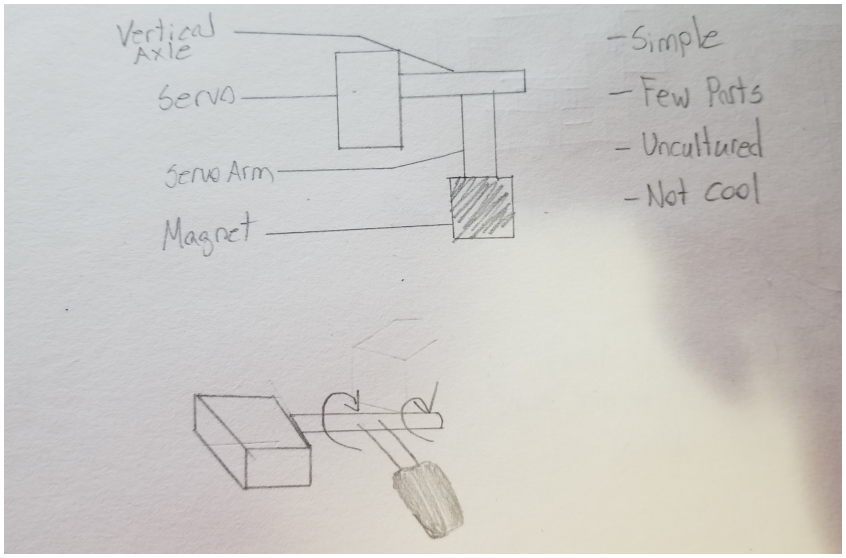


Figure 7: Magnet Concept Sketch

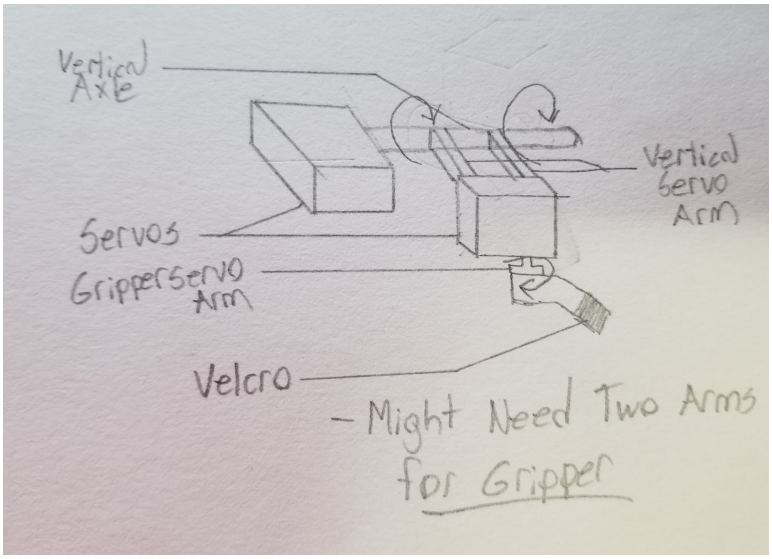


Figure 8: Velcro Concept Sketch

Conceptual Selection & Sketches

Criteria	Weight	Magnets	Velcro	Vacuum	Store Bought	Custom K'nex Gripper
Simplicity	3	1	1	-1	1	1
Reliability	5	0	0	1	1	1
Affordability	3	1	1	-1	-1	1
Robustness	4	-1	-1	1	1	1
Unique	1	1	1	1	-1	1
Manufacturability	3	1	1	-1	1	1
Total	*NA*	*6*	*6*	*1*	*11*	*19*

Table 7: Gripper Selection Pugh Matrix

In order to properly evaluate each of the concepts made for the gripper, a Pugh Matrix was utilized. Based on the matrix, it is clear that the Custom K'nex Gripper is significantly better than every other option. This concept earns full points in every category, and has no significant downside. One thing that can be said about this design compared to most others, is that it requires a longer construction and manufacturing process. This is not a large con due to the fact that K'nex pieces are highly modular and easy to assemble. Another benefit that is not clearly evaluated is that the gripper is capable of being dismounted or mounted in a matter of seconds.

When evaluating the other components, there were clear flaws in many of the designs that were undesirable. The magnet concept would require the robot to drive extremely close to the package to pick it up. Additionally, this is not a realistic solution as it requires modification to the package, and also does not provide the ability to drop the package. The velcro concept had similar issues to the magnet, and could potentially be less consistent when attempting to pick up packages. This is because attaching to velcro requires a specific amount of pressure, which may end up moving or knocking over the package instead.

The vacuum concept was a robust solution that would be reliable, however it was too complex to be a feasible solution in the allotted time. It would require significant research, money, and time to properly work, and would also be difficult to manufacture.

A store bought, pre-made gripper is a decent option, however it does create extra expenses, and is less unique because it requires no design process.

For all the reasons mentioned above, the final gripper design was selected to be the "Custom K'nex Gripper.

Form Design and Engineering Analysis

Component	Form of Interfaces	What-If Scenarios	Material Selection
Gripper Mounts	● Provides Mounting surface for gripper, attaches directly to servo motor	● Too big ● Too heavy ● Too small ●	3D Printed Plastic (ABS)
Gripper	● Picks up package when close to it	● Poor grip ● Too small ● Too large ● Too heavy ● Weak ● Crushes package	OEM K'nex parts
Gripper Arm	● Lifts gripper up/down	● Can't hold gripper & package ● Poor range of motion	OEM K'nex parts
Gripper Arm Servo	● Allows gripper arm to rotate	● Too weak ●	OEM Servo motor
Wheels	● Transfers	● Slip	OEM

	motor motion to the ground	• Can't hold TurtleBot	TurtleBot3 wheels
Wheel Motors	• Turns the wheels to move the TurtleBot	• Not enough power	OEM TurtleBot3 DC motors
Rollers	• Provides a rolling surface to act as rear wheels	• Not enough friction to roll smoothly • Small objects stuck in roller	OEM TurtleBot3 Rollers
Lidar	• Used to map the environment	• Not enough range • Use too much power • Blocked vision by gripper	OEM TurtleBot3 Lidar
Camera	• Used for package identification	• Cant see over gripper • Small FOV	OEM TurtleBot 3 Camera
	-	-	

Table 8: DFM and DFA Table

Motion Simulation

To demonstrate the motion of the gripper assembly that is mounted onto the TurtleBot3 a motion simulation in NX was completed. The motion simulation is available in the CAD folder that was submitted. A screen recording of the motion simulation is also featured in the 3D renders folder in the submission folder.

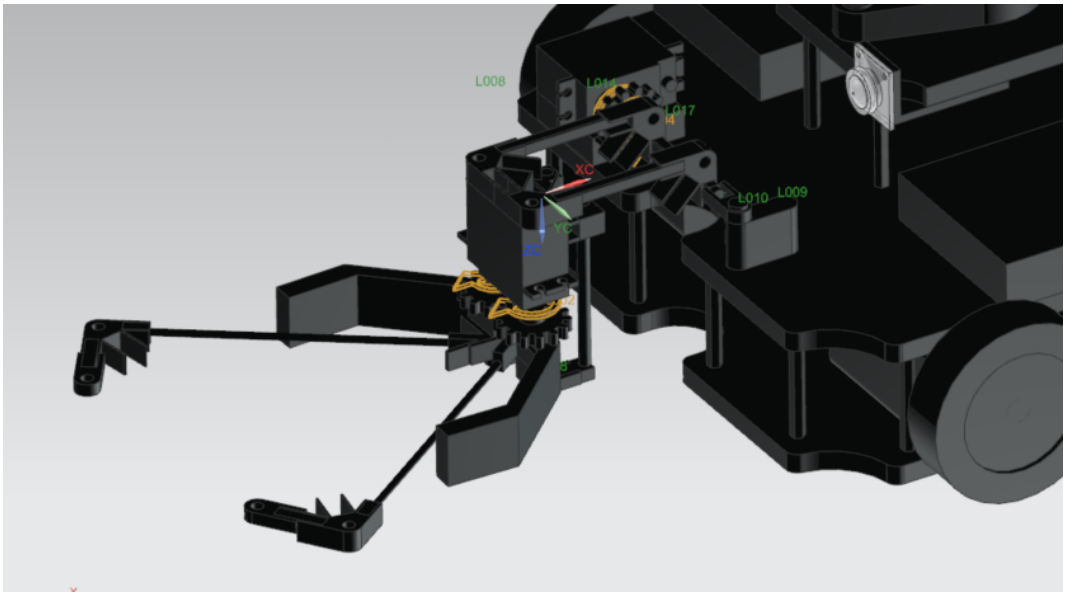


Figure 9: NX Motion Simulation on Main TurtleBot3 Assembly

Design for Manufacturing

Considering that the gripper was designed using OEM parts that can be purchased easily, it is quite simple to manufacture. The servo motors and K'nex parts require some modifications to work together, but the configuration can be recreated with common household materials, such as hot glue and cardboard. The one part that is custom is the mounting arms that attach to the gripper servo. These are 3D printed, and designed to specifically work with the servos that were available to the group. Specific dimensions and CAD information with regards to recreating these mounts will be provided in the CAD section of the report. It is worth mentioning that the arms could be replaced by any part that fits the size requirements of the servo motor shaft hole, allowing for customization based on available materials.

In terms of the overall design, it is relatively easy to manufacture if all the materials listed are available. The K’nex configuration is rather simple, requiring a few slight cuts or modifications such as the use of hot glue for mounting and reinforcement. A different more durable adhesive could be substituted for providing reinforcement to the gripper assembly structure as well. Only one fastener, which in this case was a TurtleBot3 screw, was needed to mount one end of the vertical axle to the TurtleBot3.

Design for Safety

To design for safety, a Failure Mode and Effect Analysis needs to be conducted. This analysis evaluates all risks, potential causes of failure, and how to eliminate them. This process involves the following considerations [8]:

- Determine all the ways a function can fail (with causes), and their impacts on other systems.
- Assess the severity (S) of the failure from 1- 10 (10 is catastrophic).
- For each cause assign occurrence rating (O) from 1 - 10 (1 means unlikely to occur).
- What has been added to prevent or reduce the effect of each failure (control).
- How can these problems be detected? Assign a detection rating (D) from 1- 10 (10 is undetectable).
- Risk priority number (RPN) = S x O x D - determines severity of failure, a higher number is a higher priority to control.
- Discuss the recommended actions to take based on the RPN. The results from this plan can be seen below.

Functions	Failures	Causes	Severity (S)	Occurrence (O)	Detection (D)	Control	Risk Priority Number (SxOxD)
SLAM	Inaccurate size	Modified parameters	3	1	3	Tweak Parameters	9
Exploration	Hits obstacle	Lost connection	8	2	1	Ensure stable connection	16
Package Marking	Package Not marked	Gripper Blocked camera	6	1	1	Raise Camera	6
Package Pickup	Misses Package	Misalignment	10	3	1	Retry pickup	30
Return Home	Orientation Off	PID error	1	2	1	Tweak PID control	2
Package Search	Package not found	Camera did not see package	4	3	1	Explore until packages are seen	12

Table 9: FMEA Table

Recommended Actions

Based on the above analysis of RPN (risk priority number), the first issue that should be solved is the consistency of package pickup. This could potentially be improved by developing a controller that checks to ensure the package is in the center of the frame. This could be done using PID for example.

The next issue to solve would be hitting an obstacle due to lost connection. This could be resolved by increasing the strength of the wifi connection in the testing area, or switching the testing area entirely.

The other issues mentioned have already been identified and solved. For example, a surveying node is used to ensure packages are always found in the test area. Additionally to prevent the gripper from obstructing the vision of the TurtleBot, the camera was raised to a higher mounting position.

Robot Construction Process

No construction was done to complete Milestone One through Three. This is due to the pre-built TurtleBot3 having the required components to perform the tasks. The only change made to the TurtleBot3 during Milestone One was repositioning the camera to the center of the TurtleBot3. This was so that in later Milestones, the camera would be centered for marking the packages on the SLAM map. For Milestone Four, the TurtleBot3 was required to pick up a package and return home with it. To do this, a method of picking up the package had to be selected, designed and built. After generating multiple concept ideas and comparing them, a gripper setup with two servos was decided upon. After selecting the materials to construct the gripper through FDEA (Table 8), the construction process of the gripper began.

The first step of the construction process was to mount the 3D printed gripper arm supports to the servo motor for the gripper. The gripper arm mounts can be seen in Figure 10 below prior to being attached to the servo.

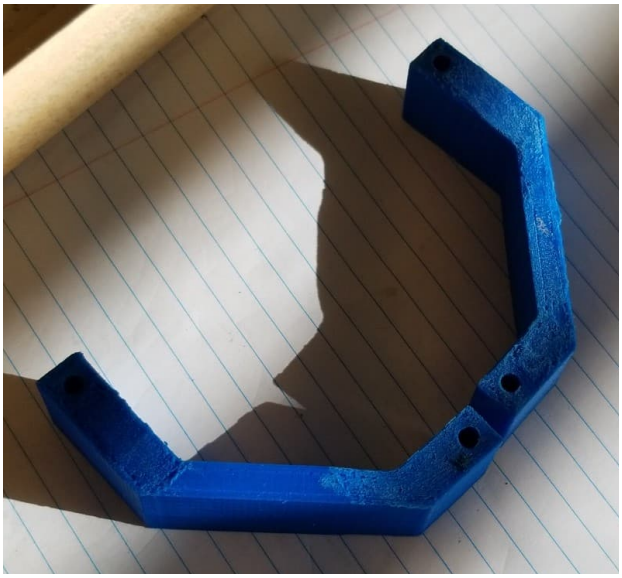


Figure 10: Gripper Arm Mounts

Two gears were attached to the top of both the gripper arm mounts to translate the rotational movement from one to the other. The left gripper's gear was then glued onto an OEM circular servo arm for the servos being utilized. This OEM circular servo arm could then be attached to the servo motor shaft. Note that this servo motor mount is de-attachable which makes it easy to make modifications to the gripper setup. It can be viewed in its attached state in Figure 11 below.

The gripper servo had to be attached to the vertical servo such that it could be moved up and down. To do this an axle was made out of K'nex and attached to a circular servo arm which would then mount onto the vertical servo shaft. Two red K'nex pieces were placed on the axle and positioned using spacers which would hold them in place. A piece was clipped on the end to prevent the spacers from sliding off the axle. Two blue K'nex pieces were attached to the red K'nex mounts on the axle. On the gripper servo, two more red K'nex mounts were attached and spaced properly for the two blue K'nex pieces to be fitted into. These blue pieces were attached to these mounts and reinforced with adhesive to prevent detachment. The results of this part of the assembly can be viewed in the following Figure 11.

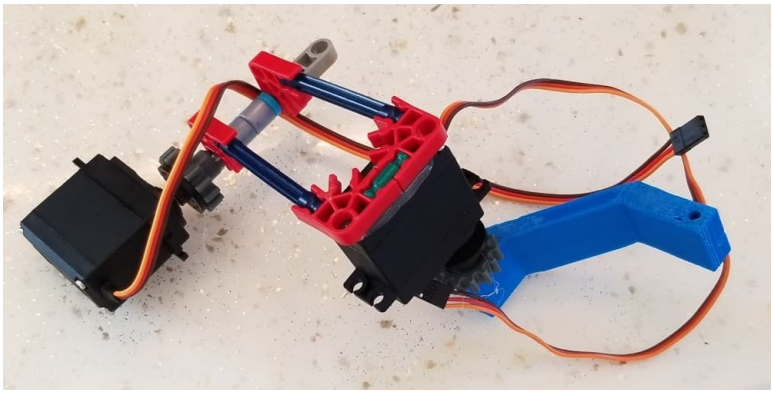


Figure 11: Mounting of Gripper Servo to Vertical Servo

The second servo arm mount now needed to be attached so that the gears aligned and it could rotate freely. An axle is attached directly to the servo motor which the second servo arm mount could slide onto and rotate. A blue K'nex piece was sanded down to fit such that the servo arm mount could spin freely on it. The gears were then aligned properly and an adhesive was then applied to keep the axle in place. The aligned gears can be seen in Figure 12.

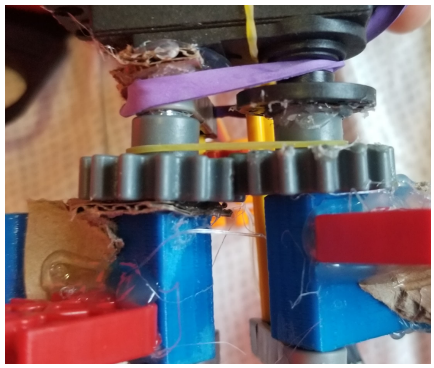


Figure 12: Gear Alignment

Next, supports were added to the bottom of the servo arm mounts to keep them in place and prevent tilting when under force. If tilting occured, the gears would become unaligned and it would cause slippage. This would cause the gripper servo arms to be out of position and not able to pick up packages. Support mounts were added to the back of the gripper servo, allowing a K'nex piece to be attached to the end of the axles of the arm mounts. One of the supports is featured in Figure 13, the second support was mounted in the same manner.

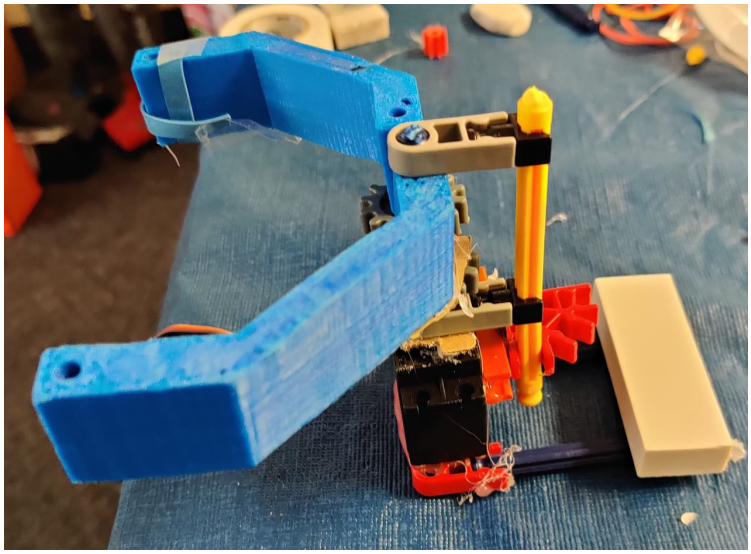


Figure 13: Adding Support to Gripper Arm Axles

The gripper then had to be positioned on the TurtleBot3 and attached, this was done using adhesive and fasteners. On the left in Figure 14, the vertical servo can be seen attached with adhesive on top of some cardboard spacers to align it properly. The far side of the axle that is attached to the vertical servo was fastened in with a screw to the OEM black mounts that came with the TurtleBot3.

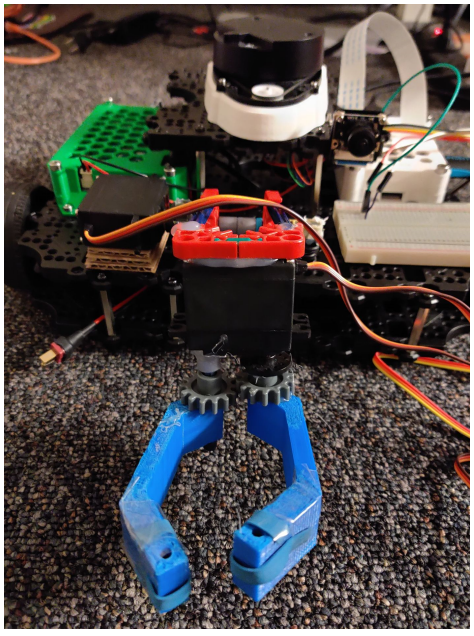


Figure 14: Mounting Gripper onto TurtleBot3

With the gripper assembly being detachable from the vertical servo axle, it could be removed to perform the remaining steps of construction. The final modification was to attach the gripper arms onto the gripper arm mounts. As can be seen in Figure 15, two red K'nex pieces were attached to the mounts using adhesive. The red K'nex pieces were positioned so that the protruding K'nex arms would rest against the ends of the gripper arm mounts. This would make the arms more steady when closing to pick up a package. Two red cylindrical K'nex pieces were attached as the gripper arms to the red K'nex mounts. On the end of the arms, a claw was made using more K'nex and foam was attached using adhesive, acting as a cushion on the package and providing more grip. The gripper assembly was now ready to be tested for functionality and to verify if further modifications were required.

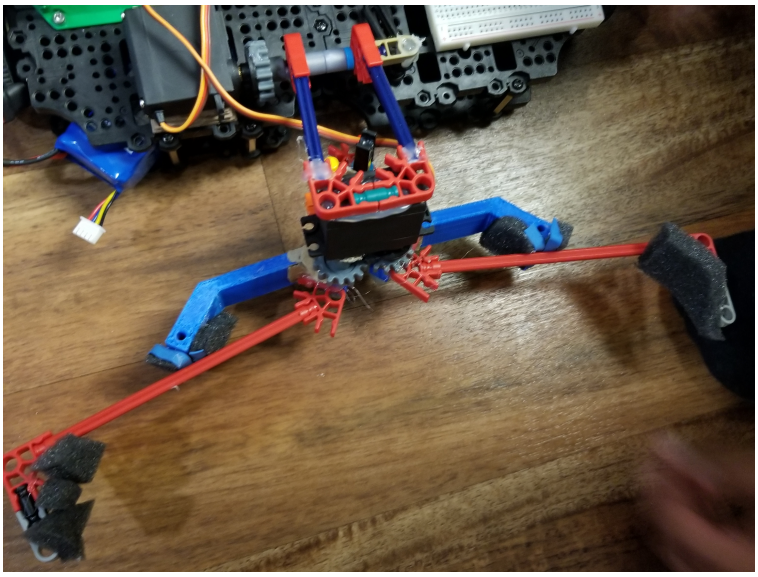


Figure 15: Gripper Arm Attachment

Finally the gripper assembly was tested in isolation to see if it could pick up the package consistently with a firm hold on the package. The wiring of the gripper required a small breadboard with a resistor and seven wires to connect the servos to the Raspberry Pi. After various tests which involved adjusting the starting and ending angles of the pick up motion in the Python code on the Raspberry Pi the gripper would successfully pick up the package one

hundred percent of the time when it was in position. The gripper can be seen picking up the package in Figure 16.

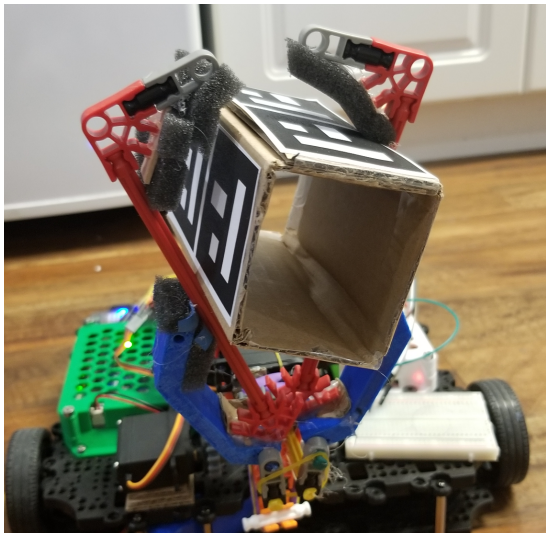


Figure 16: Gripper Arm Attachment

A top down view of the finished gripper assembly can be seen in Figure 17. The TurtleBot3 began performing full run tests after verification of the gripper operation.

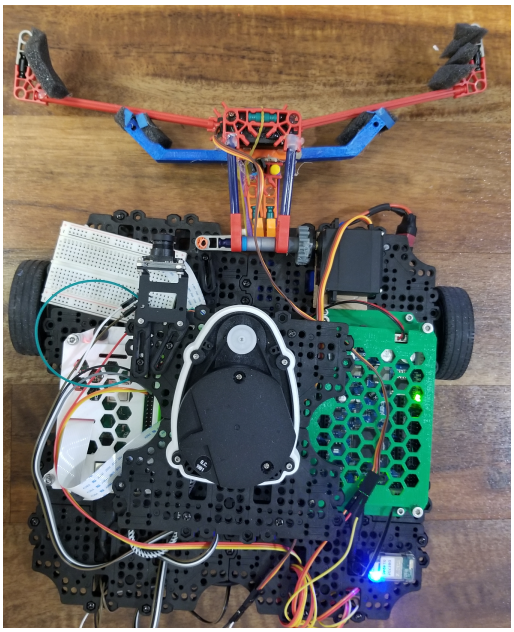


Figure 17: Completed and Mounted Gripper Assembly

Control Algorithm Design

SLAM

A map must be generated using the Lidar mounted on the TurtleBot3 as it is necessary for exploration. The Lidar data is published as a /scan ROS topic by the TurtleBot3_core node. The /scan ROS topic is subscribed to by TurtleBot3_slam_gmapping which utilizes it with the transform frames to publish a map with a message type of nav_msgs/OccupancyGrid. This map is the two-dimensional area that has been explored by the TurtleBot3 and can be displayed in RVIZ. The parameters of the TurtleBot3_slam_gmapping are loaded upon launch from the .yaml file "gmapping_params.yaml". This map is used by the explore-lite node to select frontiers. In addition to this, the SLAM map is also used by the global costmap published by costmap_2D. An example of a completely finished SLAM map after exploration in RVIZ can be seen in Figure 18.

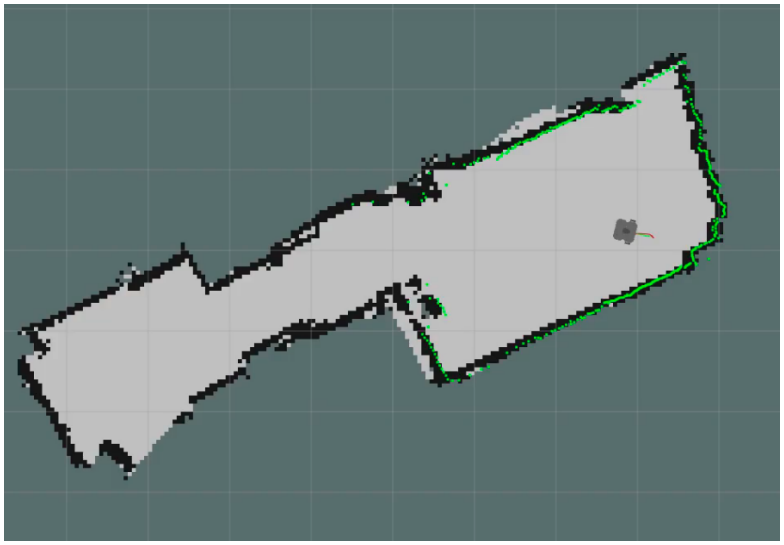


Figure 18: TurtleBot3_slam_gmapping SLAM map example

Path Planning

In order to navigate between goals, a path planning algorithm is needed. For this, an open-source path planning algorithm called DWA planner was used. DWA stands for dynamic window approach and works by sampling the velocity in the robot's control space. It then simulates those velocities, on the position of the robot, for a short amount of time. Following this, it evaluates the trajectory results of the robot and gives each one a score. The program then chooses the trajectory with the best score and selects it as the desired path for the robot [9]. This is repeated until the robot reaches its desired destination.

Package Location and Avoidance

Three primary programs are responsible for package avoidance. The first program, aruco_detect.cpp is used to identify the packages using the camera. The second program, marker.py is used to transform the data from arucodetect.cpp in order for the location of the package to be placed on the SLAM map. The final program, fake_laser_scan.cpp, is used to add the location of the package to the cost map in order to properly avoid the package while the TurtleBot is still exploring. The following sections break down these specific programs in more detail.

aruco_detect.cpp

Aruco_detect.cpp is a program that can identify arUco markers and extract their pose. This method was chosen since computer vision is capable of consistently recognizing arUco markers with a low percentage of error. The package was then outfitted with eight of the same arUco marker, two on each side with the exception of the top and bottom. Once an arUco marker is seen by the camera and recognized, aruco_detect.cpp then extracts the pose data and ID of the arUco marker. The ID is based on a predetermined library of arUco markers manually loaded into aruo_detect.cpp. The pose and the ID of the aruco marker is then published to a topic called "fiducial_transforms" in the format of a fiducial transform array. Marker.py is a subscriber of "fiducial_transforms" and receives any data published to that topic. After the information is published, marker.py then processes the data [10].

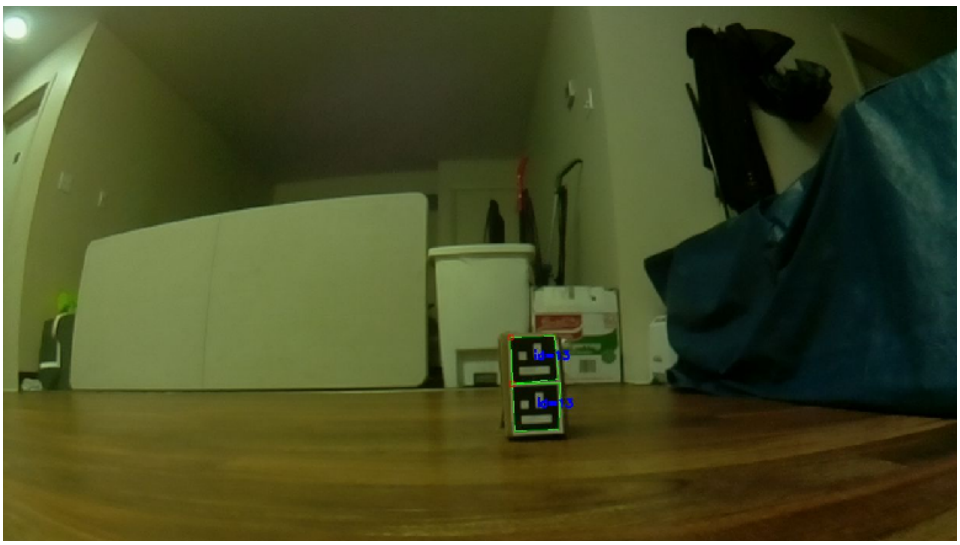


Figure 19: Aruco Marker Detection on Package

marker.py

Once the package is located, marker.py saves the location of the package and publishes that data to the 2D SLAM map. Marker.py is also responsible for sending the location of the package, with respect to the map, to the node fake_laser_scan.cpp. The data received from aruco_detect.cpp via the topic “fiducial_transform” is in the form of a fiducial transform array. Furthermore, the pose data of the package is taken with respect to the camera of the TurtleBot, not the map. In order for the pose of the object to be with respect to the map instead, a pose transform needs to be done. However, since the data is being received in the form of a fiducial transform array, it must first be converted back into a pose. Package_transform is the function used to transform the pose. It then publishes the transformed pose to the topic “obstacle_location” for fake_laser_scan.cpp to receive.

To represent the obstacles location on the 2D SLAM map, marker.py publishes a marker type message to the topic “fiducials”. Rviz, the visualization tool, is subscribed to the topic “fiducials” and places the visual marker on the map. Marker.py is featured in the appendix in Figure A4.

Figure 20: Marking Package on SLAM Map

fake_laser_scan.cpp

In order to properly treat identified packages as obstacles, fake_laser_scan.cpp is used. It does this by creating a simulated laser scan to mark objects based on given coordinates. This program subscribes to the topic "obstacle_location", published by marker.py, and receives the coordinates of the package, relative to the map, and the fiducial ID. It then creates fake laser scan data at the coordinates of the package, as if a real lidar collected the data. This data is then processed by the costmap and treats the coordinates as an obstacle. The custom code created for this function was made based on an open source code found on Github [11]. The fake_laser_scan.cpp is included in the appendix as Figure A5.

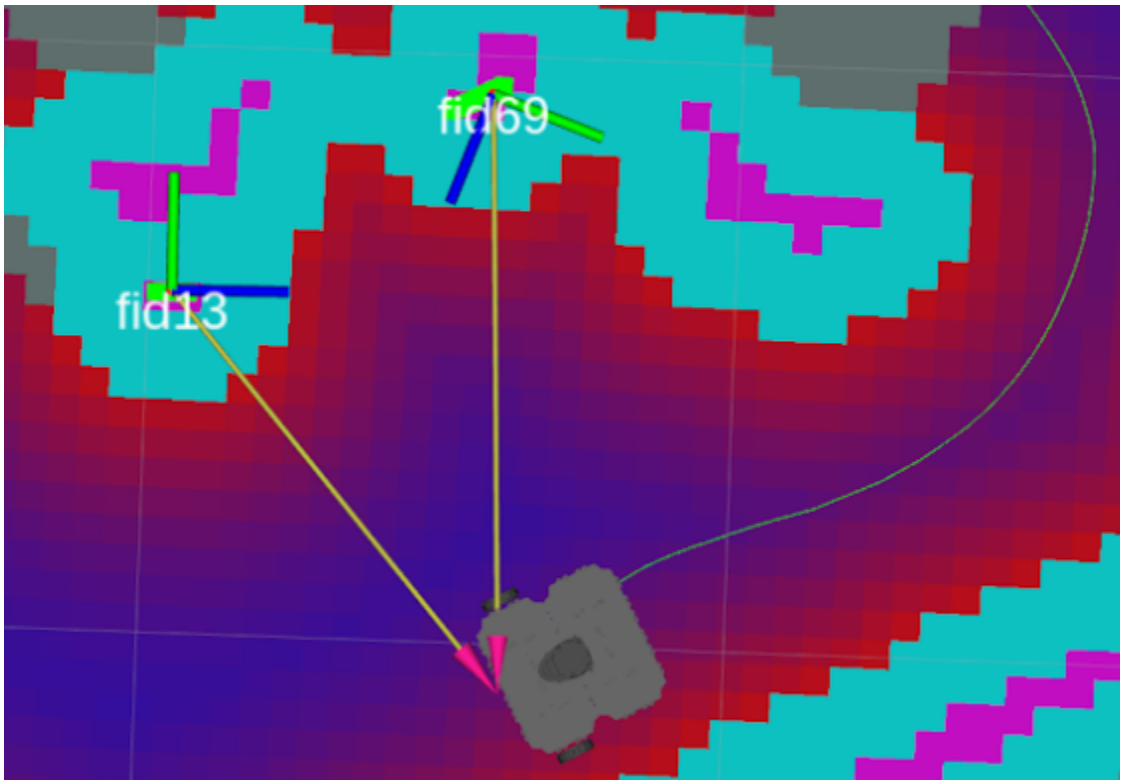


Figure 21: /fake_scan (Green) Placed in Package Locations

Exploring and Surveying

When the robot is navigating around the test area, prior to finding the package, it has two distinct modes: exploring and surveying. Exploring is performed using the program explore_lite.cpp and surveying is performed using package_search.py (Figure A6).

Explore_lite.cpp uses a frontier based exploration algorithm. Frontiers are determined by the amount of unknown space in a given area. The frontiers are then set as goals for the TurtleBot to navigate towards [12]. Once all frontiers have been explored, the program publishes a message to the topic “finished_exploring”. Package_search.py is subscribed to that topic and initiates once it receives a message. Package_search.py uses a library called occupancy_grid.py. This library is used to simplify the process of checking the cost of various cells on the generated map through pre built functions. Package_search is used to survey the map, once exploration is completed, in order to locate the package if it has not been found. The surveying prioritizes the corners of the map. It checks the cost of the coordinates, prior to setting a goal, to ensure the space is not occupied and relatively close to a wall. This is because it was found that it would often have trouble finding packages close to walls. Once the robot is within 20 cm of its surveying goal, a new goal is set. It keeps performing the surveying, approaching all 4 corners of the map, until the package is detected. Once the package is located it publishes a message to the topic “go_to_pacakge” to begin package retrieval.

Package Retrieval

The package retrieval phase consists of three primary parts: approaching the package, picking-up the package, and returning home. These three parts are split into three separate programs respectively: get_package.py (Figure A7), gripperROS.py (Figure A8), and return_home.py (Figure A2). Get_pacakge.py sets the navigation goal of the robot to the location of the package. It then approaches the package using a forward facing biased. When the TurtleBot is within 10 cm of the package and its pose is aligned with the coordinate of the package, it sends a message to the topic “grab_em” and stops in place. GripperROS.py is subscribed to this topic and is notified to initiate when it receives a message. When the program is initialized, it simply powers the servos that control the gripper through the GPIO pins, grabbing the package and lifting it above the robot. The reason it lifts the package above the robot is to reduce the size of the footprint of the TurtleBot. Once the package has been successfully lifted, the program then publishes a message to the topic “found

_pacakges” in order to initiate return_home.py return home function. Return_home.py first initializes at startup. It begins by recording the initial location of the TurtleBot. It then sets this

location as the TurtleBot's "home" for after the TurtleBot picks up the package. When it receives a message from "found_packages" the "home" location is set as a navigation goal and the TurtleBot drives towards the goal. Once it reaches its goal, it will orientate itself to match its starting pose and stop in place.

Test Plan, Results, & Validation

The task of locating, going close to and picking up a package requires significant accuracy and testing to achieve a reliable success rate, and can be impacted by various factors. In order to remove as many issues as possible, a significant portion of this milestone was dedicated to testing, debugging and optimizing the performance of the TurtleBot.

Following the creation of the gripper and its respective control node, it was necessary to evaluate its performance. The initial tests for this were simple, place the package in front of the TurtleBot, and run the pickup script. This test identified a variety of issues, such as package slip, weak grip strength, flimsy gripper design and poor gear meshing. These issues led to large problems such as the gripper not picking up the package, or the arms becoming misaligned due to gears skipping, causing potential damage to the gripper or servo motors.

In order to combat these issues, major improvements to the programming and design of the gripper were performed. To combat package slip and weak grip, foam was added to the gripper and the code was tweaked to close the arms further. This allowed more force to be exerted on the package over a larger surface, and prevented it from falling. The next major concern was that the gripper appeared flimsy, and would shake significantly upon slight movement of the TurtleBot. To remedy this, the gripper was reinforced with additional components such as K'nex, hot glue, and cardboard. This greatly improved the rigidity of the overall design.

At this point in time, the gripper was able to pick up packages, so testing of the new nodes could begin. During the software tests, it was immediately apparent that the addition of the gripper obstructed a significant amount of the camera's vision. This led to the packages sometimes being missed, despite being directly in front of the TurtleBot. To solve this problem, the camera was mounted to a higher location, and extended outward, removing most of the

gripper from the camera's vision. With this change, the consistency of the software significantly improved, because the TurtleBot could now accurately track the package.

After some additional testing with this setup, it was noted that sometimes the gears of the gripper would still slip, causing arm misalignment, and the package to fall. To eliminate this problem, more modifications to the gripper were made. The gear mounts were reinforced, and elastics were bound between them to ensure they could mesh properly, without skipping. After this final addition, the major issues found within testing were resolved, and the TurtleBot was ready for the live demonstration.

During the first live run, the TurtleBot identified the package and moved towards it, but slightly missed the package, resulting in a failed pickup attempt. The robot then returned home to its original location. This error was believed to be caused by an odometer error, causing slight misinterpretation of the TurtleBots actual position relative to the package.

During the second attempt, the TurtleBot completed the run successfully, picking up the package, lifting it off the ground, and returning home without any issues.

Logbook

Topic	Date	Members Present
Tutorial Meeting #1 -ROS installation	Thursday, Jan 21 2021	Everyone
Tutorial Meeting #2 -ROS orientation	Thursday, Jan 28 2021	Everyone
Milestone 1 ROS session	Saturday, Jan 30 2021	Everyone
Tutorial Meeting #3 - Milestone 1 Prep and Questions	Thursday, Feb 4 2021	Everyone
Milestone 1 programming and testing session	Saturday, Feb 6 2021	Everyone
Tutorial Meeting #4 - Milestone 1 Update	Thursday, Feb 11 2021	Everyone
Milestone 1 Final Testing	Thursday, Feb 11 2021	Connor, Myles, Hudson
Demonstration Date	**Feb 12, 2021, 12:40 PM**	Everyone
Aruco Marker and Obstacle Work Session	Saturday, Feb 13 2021	Everyone
Tutorial Meeting #5 - Milestone #1 Questions	Thursday, Feb 18, 2021	Everyone
Milestone #1 Report Writing Session	Thursday, Feb 18, 2021	Everyone
Milestone Report #1 Due	**Monday, Feb 22, 2021, 5:00 PM**	Everyone
Tutorial Meeting #6 - Aruco marker refinement questions	Thursday, Feb 25, 2021	Everyone
Tutorial Meeting #7 - ROS mapping questions	Thursday, Mar 4, 2021	Everyone
Milestone 2 Final Testing	Thursday, Mar 4, 2021	Connor, Myles, Hudson
Demonstration Date	**Friday, March 5, 2021, 12:40 PM**	Everyone
Milestone #2 Writing Sessions	Saturday, March 6, March 11	Everyone

Tutorial Meeting #7 - Milestone #2 Questions	Thursday, Mar 11, 2021	Everyone
Milestone Report #2 Due	**Friday, March 12, 2021, 5:00 PM**	Everyone
Kenneth joins group	Friday, March 12, 2021	Kenneth
Milestone 3 Coding Session	Saturday, March 13, 2021	Connor, Myles, Hudson
Camera Calibration	Sunday, March 14, 2021	Connor, Kenneth
Tutorial Meeting #8 - Navigation questions	Thursday, Mar 18, 2021	Everyone
Milestone 3 Final Testing	Thursday, March 18, 2021	Connor, Myles, Hudson
Demonstration Date	**Friday March 19, 2021, 12:40 PM**	Everyone
Milestone #3 Writing Sessions	Saturday March 20, 2021	Everyone
Tutorial Meeting #9 - Milestone #3 Marking Questions	Thursday, Mar 25, 2021	Everyone
Milestone 3 Report Due	**March 26, 2021, 5:00 PM**	Everyone
Tutorial Meeting #10 - Gripper and Magnet Questions	Thursday, April 1, 2021	Everyone
Tutorial Meeting #10 - Robot - Package alignment questions	Thursday, April 8 2021	Everyone
Gripper Integration Work Session	Thursday, April 8th, 2021	Connor, Hudson, Evan
Final Report Writing Sessions	Saturday, April 10th, Sunday April 11th	Everyone
Demonstration Date	**April 12, 2021, 12:40 PM**	Everyone
Final Report Due	**April 19, 2021, 5:00 PM**	Everyone

Table 10: Project Logbook

CAD & Part Details

BOM

It should be noted that no prices are listed for the BOM because all parts were already owned by the group members, no expenses were made.

Part Name	Quantity
TurtleBot3 Slider Mount	2
Servo Motor	2
K’nex Green Axle	3
K’nex Blue Axle	4
K’nex Red Axle	2
K’nex Yellow Axle	2
K’nex Red Connecter	9
K’nex Grey End Piece	6
Cardboard	1
Hot Glue	3
Small TurtleBot Screws	1
Rubber Bands	3
K’nex Gears	3
3D Printed Mounting Arms	2
K’nex Grey Spacer	3
K’nex Blue Spacer	2
K’nex Black Connector	4
Circular Servo Arm Mounts	2

K’nex Straight Orange Connector	3
K’nex Angled Orange Connector	2
K’nex White Axle	1
K’nex Black Axle	2
Foam Grip	2

Table 11: Bill of Materials

Assembly/Sub-Assembly Drawings

Two assemblies were created in NX for the TurtleBot3, the gripper sub-assembly, and the main TurtleBot3 assembly with the gripper mounted. The gripper sub-assembly is featured in Figure 22 and includes the servo motors, gripper arm mounts, and the required K’nex pieces. In addition to the model, Figure 23 displays a sketch and BOM of the gripper-assembly.

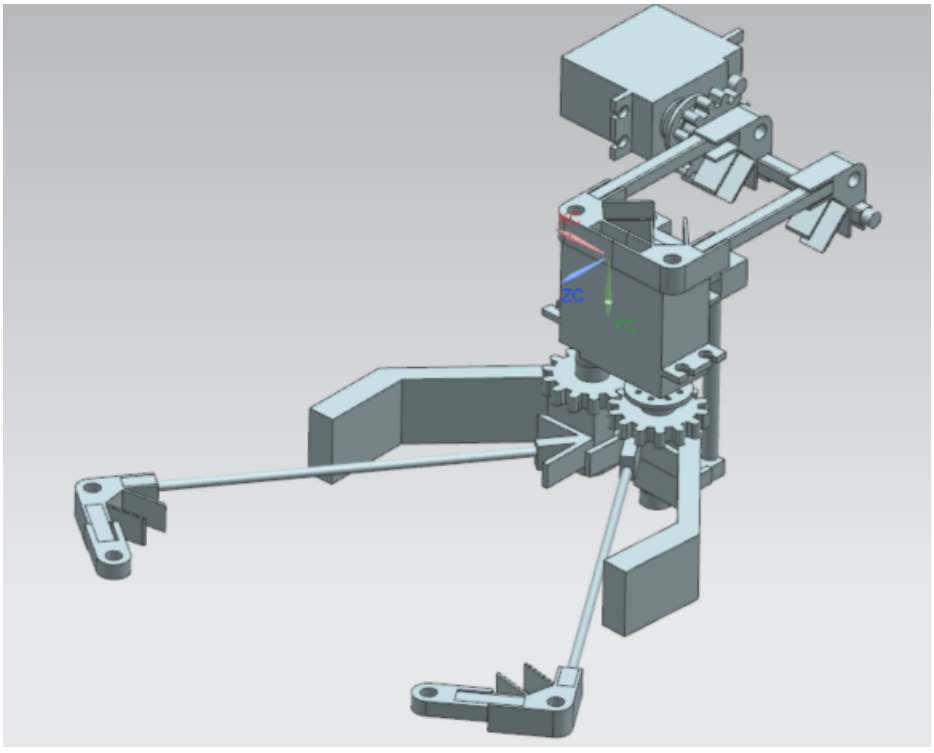
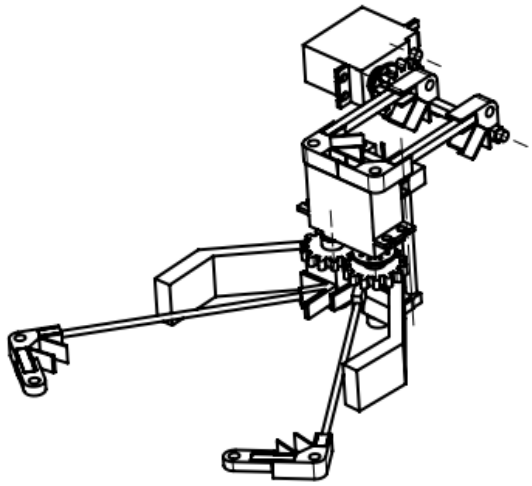


Figure 22: Gripper Sub-Assembly NX Model

PC NO	PART NAME	QTY
17	SMALL BLACK KNEX	2
16	RED KNEX GRIPPER	2
15	GRIPPER RED MOUNT KNEX	2
14	VERTICAL AXLE SERVO1	1
13	BLUE CONNECTOR	2
12	RED KINEX	6
11	RED MOUNT	1
10	ORANGE MOUNT	1
9	YELLOW AXLE	2
8	BLACK CONNECTOR	2
7	GRIPPER AXLE	1
6	GRIPPER ARM RIGHT	1
5	GRIPPER ARM LEFT	1
4	KINEX SILVER	5
3	GEAR GRIPPER	3
2	SERVO WHEEL	2
1	SERVO	2



SIEMENS

THIS DRAWING HAS BEEN PRODUCED USING AN EXAMPLE TEMPLATE PROVIDED BY SIEMENS PLM SOFTWARE

FIRST ISSUED

TITLE

Figure 23: Gripper Sub-Assembly Sketch and BOM

The main TurtleBot3 assembly has all standard and non-standard parts which can be seen in Figure 24. The displays the gripper sub-assembly mounted onto the TurtleBot3. The following Figure 25 also shows the sketch of the main assembly with the BOM. All CAD files and PDFs of the drafting are included in the submission folder.

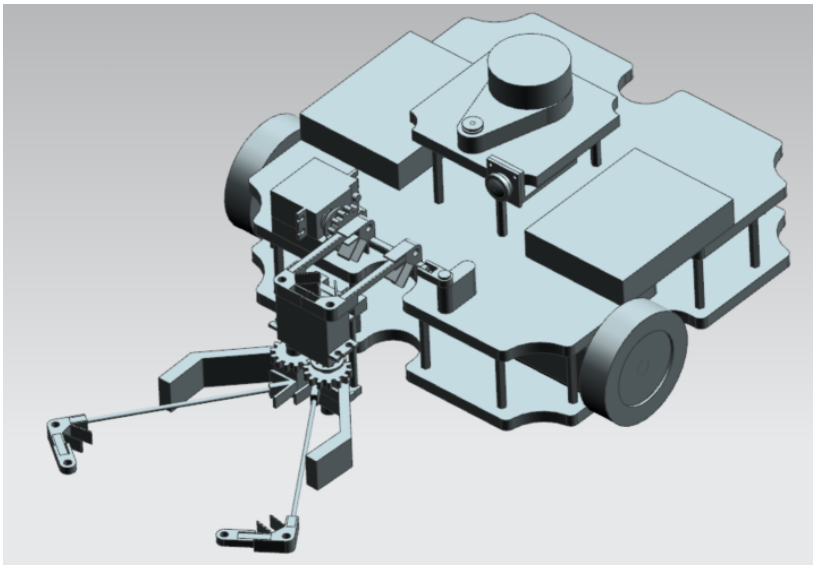


Figure 24: Main TurtleBot3 Assembly

24	LIDAR	1
23	LIDAR WHITE WHEEL	1
22	RASPIANDOPENCR BOARD	2
21	UPPER PLATE TURTLEBOT3	1
20	GRIPPER MOUNT SERVO	1
19	GRIPPER SERVO SUB	1
18	GRIPPER ARM LEFT	1
17	GRIPPER ARM RIGHT	1
16	GRIPPER AXLE	1
15	KINEX SILVER	6
14	BLACK CONNECTOR	2
13	YELLOW AXLE	2
12	ORANGE MOUNT	1
11	RED MOUNT	1
10	BLUE CONNECTOR	2
9	RED KINEX	6
8	GEAR GRIPPER	3
7	VERTICAL AXLE SERVO1	1
6	SERVO WHEEL	2
5	SERVO	2
4	TBOT WHEEL	2
3	TURTLEBOT3 MOTOR	2
2	HEXAGON SUPPORT	20
1	TURTLEBOT3 WAFFLE PI	2
PC NO	PART NAME	QTY

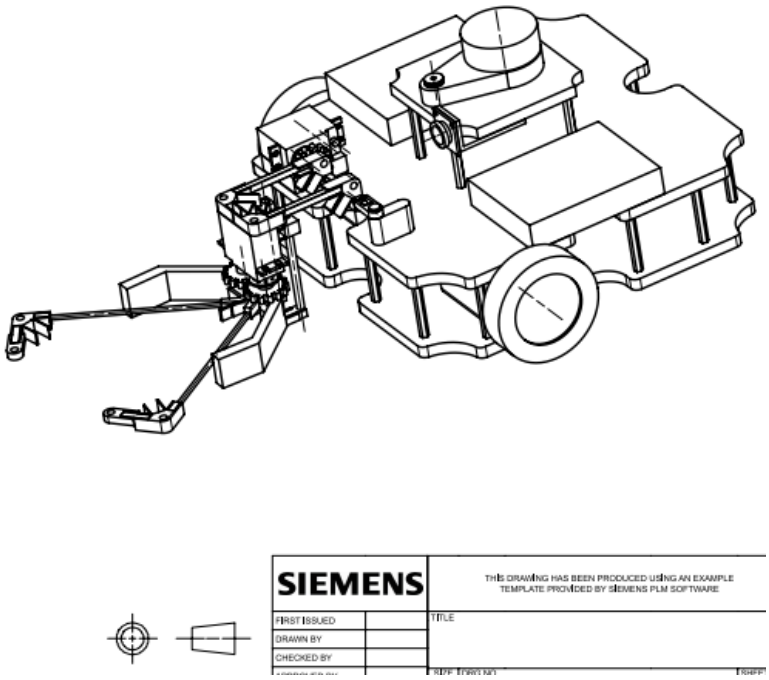


Figure 25: Main TurtleBot3 Assembly Sketch and BOM

Tolerances

The assembly of the TurtleBot3 had virtually no tolerancing required due to almost all parts being OEM and adhesive being utilized. The only tolerance required is for the holes through the 3D printed servo gripper mounts. A tolerance is needed since the fit of the axle in the hole needs to be snug enough that it does not allow the arm mounts to tilt while letting it rotate freely. A tolerance of +/- 0.10mm was given to this hole after construction since a drill was required to adjust the hole size slightly. The drill was utilized to slightly widen the hole's diameter to allow the arm mounts to rotate freely. No other tolerancing was required for the assembly of the gripper or TurtleBot3.

3D Render of Final Design

The pictures of the final renderings are included in the "3D Renders" folder in the submission. The following Figures 26, 27, and 28 also provide two views of the rendered main TurtleBot3 assembly. Figures # and # is an overview of the entire assembly, while Figure # is a close up on the rendered gripper.

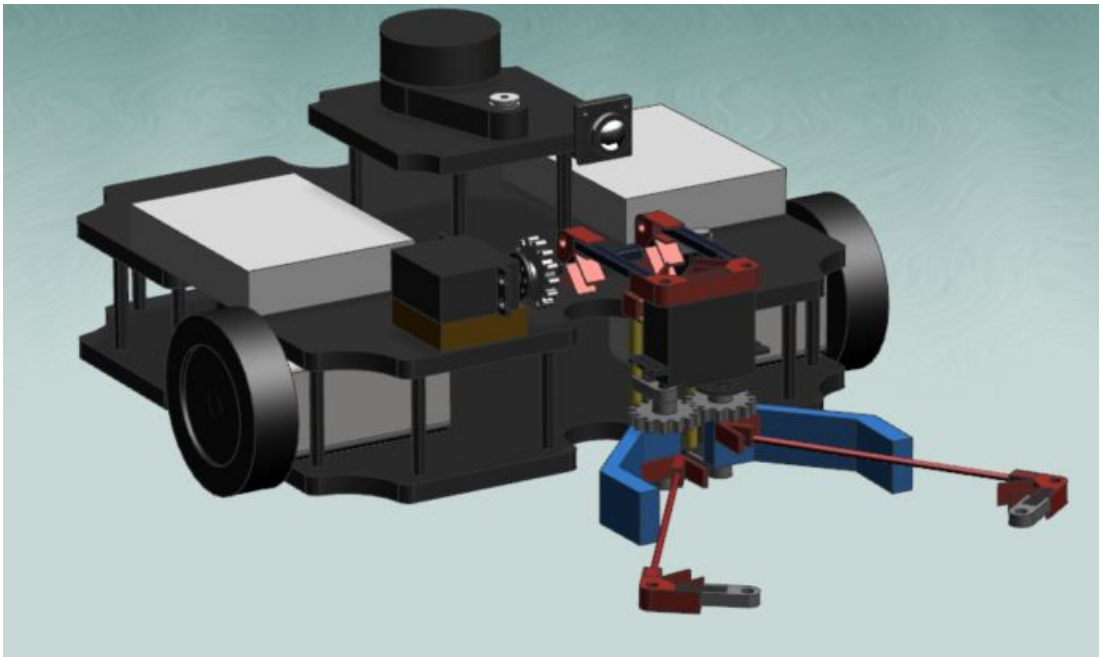


Figure 26: CAD Main TurtleBot3 Assembly Render

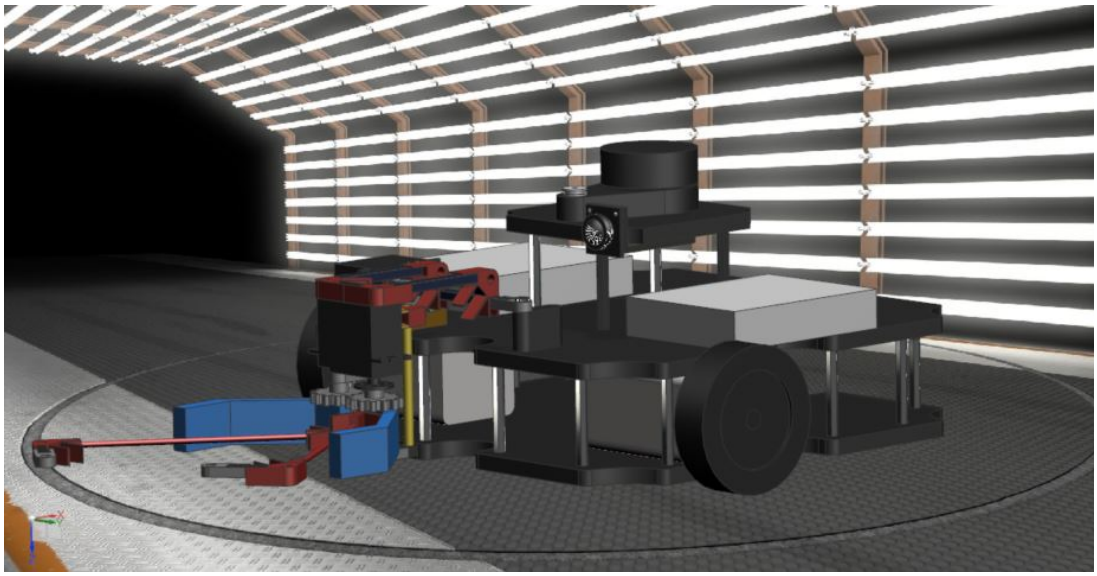


Figure 27: CAD Main TurtleBot3 Assembly Render

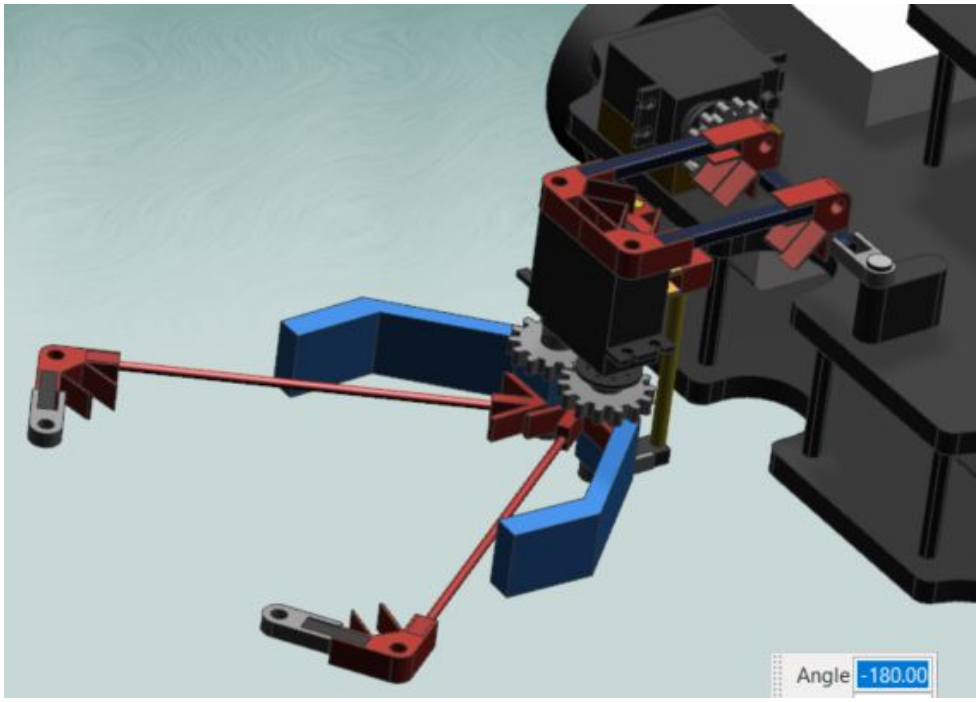


Figure 28: CAD Main TurtleBot3 Assembly Render Gripper Closeup

Detailed Drawings

Detailed Design Drawings (Non Standard Parts)

All parts of the gripper assembly were drafted and PDFs of these are included in the submission folder. All parts utilized in the gripper assembly and the TurtleBot3 are standard parts except for the 3D printed gripper arm mounts. The drafting of the right gripper arm mount can be seen in Figure 29. In addition to this the model of the right gripper arm mount is featured in Figure 30. The left gripper arm mount CAD model and draft PDF are also included in the submission folder. As mentioned in the tolerance section a +/- 0.10mm tolerance is applied on the hole. After printing of these parts a drill was required to adjust the hole size slightly.

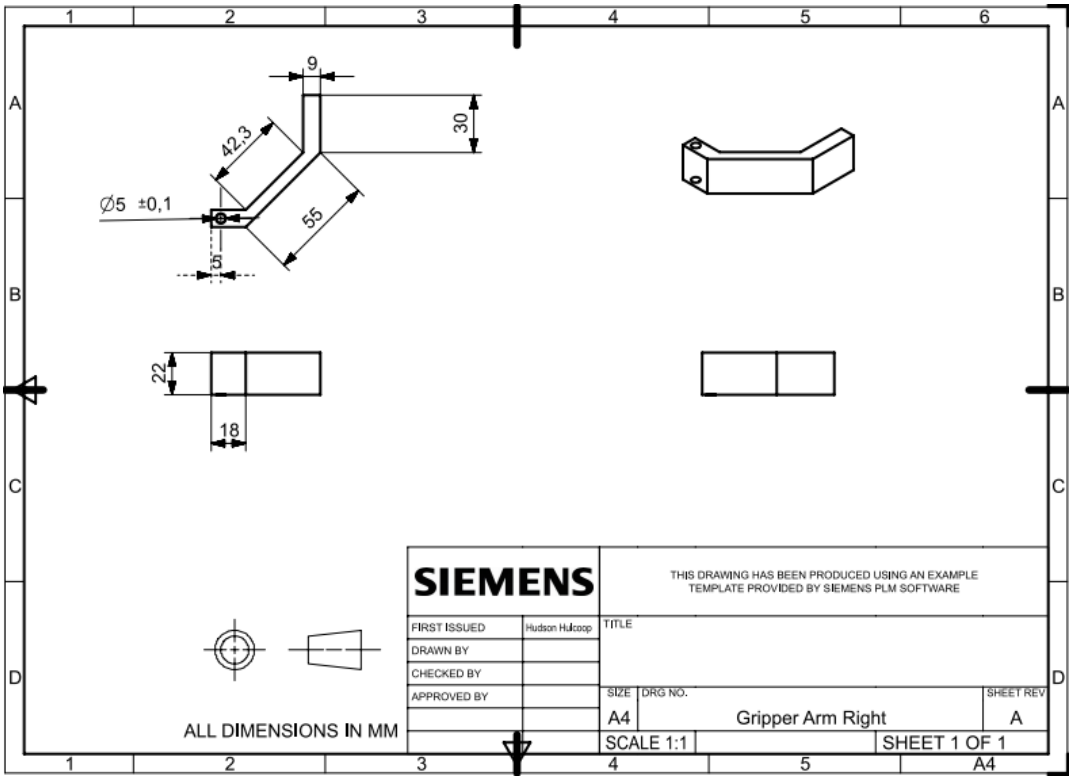


Figure 29: Gripper Arm Mount Right Drafting

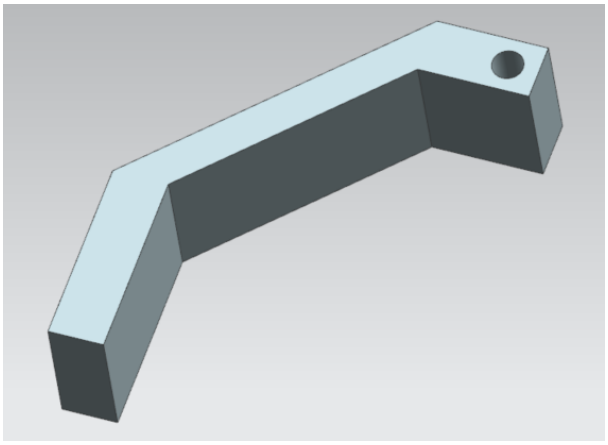


Figure 30: Gripper Arm Mount Right Drafting

Circuit Drawings

The following drawing was created for the newly implemented gripper circuit for Milestone 4. The servo motors were connected to the Raspberry Pi GPIO pins to control the PWM signals and power. A bread board was needed to split the 5V power as the Raspberry Pi only contains two 5V pins and one was taken for the TurtleBot lidar power. The circuit diagrams for TurtleBot3 components can be found in the user manual.

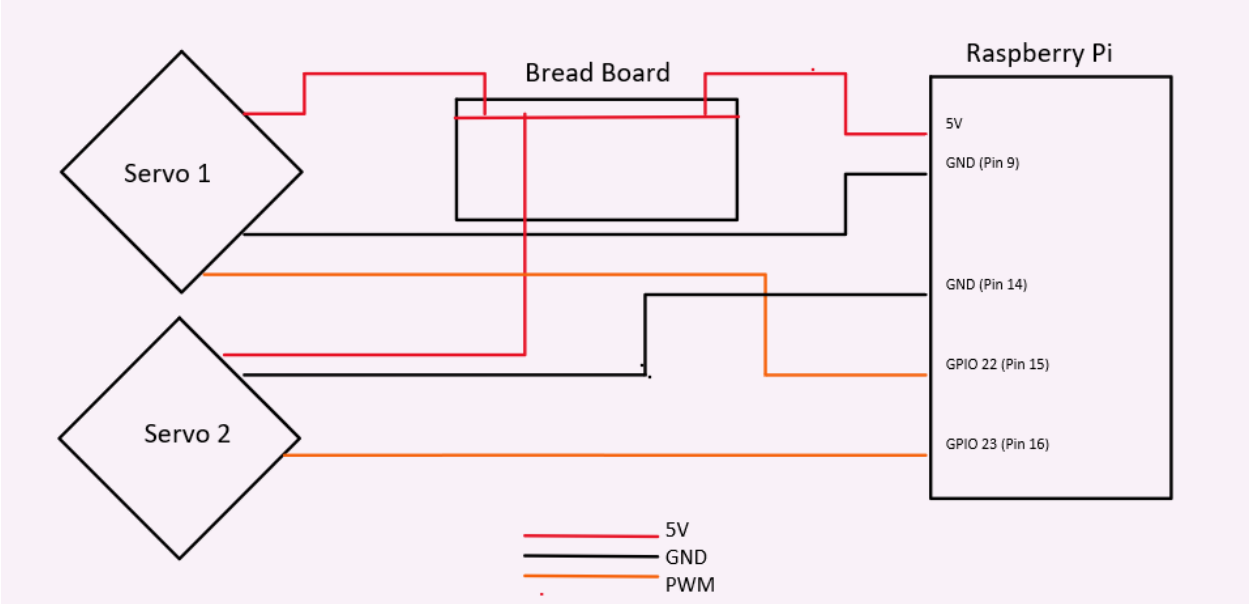


Figure 31: Electrical Circuit Diagram

Conclusion

In conclusion, a robust mobile robotics solution was successfully designed and tested to meet the challenges associated with autonomous package retrieval. The hypothetical customer and class provided project requirements which dictated the projects objectives and timeline. An emphasis on visual output, as well as a discrete pass/fail nature of each milestone allowed the team to produce a robot capable of mapping and package retrieval, utilizing concepts learned within mobile robotics, implemented by the industry standard of the robot operating system. The team submitted successful video demonstrations of each milestone, and performed all of them live. The robot served as a learning tool to implement many key concepts of mobile robotics, including utilizing multiple sensor inputs, processing information about the outside world, and managing mapping and navigation software. Overall, the team is content with the successful completion of this project.

References

[1] “ROBOTIS e-Manual.” [Online]. Available: https://emanual.robotis.com/docs/en/platform/TurtleBot3/overview/. [Accessed: 17-Jan-2021].

[2] ”SuperDroid Robots”, *Autonomous Programmable ROS SLAM Tracked Robot Package*, 2021. [Online]. Available: https://www.superdroidrobots.com/shop/item.aspx/autonomous-programmable-ros-slam-tracked-robot-package/2610/. [Accessed: 19-Feb-2021].

[3] “Ecovacs Deebot T8 Robot Vacuum & Mop Cleaner,” *Amazon.ca: Home & Kitchen*, 2021. [Online]. Available: https://www.amazon.ca/T8-Detection-Avoidance-Navigation-Multi-floor/dp/B08CSVDHTR/ref=sr_1_3?dchild=1&gclid=CjwKCAiAg8OBBhA8EiwAlKw3krAJoGYS0kcyYb4MYPD1-5X4VnnnhjwQel6WMbygCkP4zoDCJnndZhoCNdIQAvD_BwE&hvadid=208258339595&hvdev=c&hvlocphy=9000971&hvnetw=g&hvqmt=e&hvrand=10444769934496295952&hvtargid=kwd-296557630245&hydadcr=14057_9293336&keywords=deebot&qid=1613880010&sr=8-3&tag=googcana-20. [Accessed: 22-Feb-2021].

[4]“frontier_exploration - ROS Wiki,” *ros.org*. [Online]. Available: http://wiki.ros.org/frontier_exploration. [Accessed: 23-Jan-2021].

[5] “costmap_2d-ROS Wiki,” *ros.org*. [Online]. Available: http://wiki.ros.org/costmap_2d. [Accessed: 21-Jan-2021].

[6]“How to control a servo motor with the Raspberry Pi,” *Howto Raspberry Pi*, 27-Aug-2020. [Online]. Available: https://howtoraspberrypi.com/servo-raspberry-pi/. [Accessed: 19-Mar-2021].

[7] “opencv-python-tutorials” *OpenCV*. [Online]. Available: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/py_calibration.html. [Accessed: 3-Mar-2021]

[8]“Failure Mode and Effects Analysis (FMEA),” *ASQ*. [Online]. Available: https://asq.org/quality-resources/fmea. [Accessed: 18-Oct-2020].

[9] “dwa_local_planner - ROS Wiki,” *ros.org*. [Online]. Available: http://wiki.ros.org/dwa_local_planner. [Accessed: 21-Jan-2021].

[10] “aruco_detect,” ros.org. [Online]. Available: http://wiki.ros.org/aruco_detect. [Accessed: 10-Feb-2021].

[11] vinaykumarhs2020, “vinaykumarhs2020/lidar_publisher,” GitHub. [Online]. Available: https://github.com/vinaykumarhs2020/lidar_publisher/blob/master/src/lidar_publisher_node.cpp. [Accessed: 12-Feb-2021].

[12] “explore_lite - ROS Wiki,” *ros.org*. [Online]. Available: http://wiki.ros.org/explore_lite. [Accessed: 23-Jan-2021].

Appendix

```
#tb_explore.launch
#Starts all ROS packages required for Milestone One

<launch>

        <!-- Launches SLAM package with default TurtleBot3 Parameters -->
        <include file="$(find TurtleBot3_slam)/launch/TurtleBot3_slam.launch"/>

        <!-- Launches aruco_detect package with default 4x4 Parameters -->
        <include file="$(find aruco_detect)/launch/aruco_detect.launch"/>

        <!-- Launches marker node for placing package markers on SLAM map -->
        <node pkg="mobile_robotics" type="marker.py" name="package_marker"
        output="screen"/>

        <!-- Launches fake_laser_publisher node placing LaserScan points on packages -->
        <node pkg="mobile_robotics" type="fake_laser_publisher" name="fake_laser_publisher"
        output="screen"/>

        <!-- Launches move_base package with dwa_local_planner and parameters from
             specified .yaml files -->
        <node pkg="move_base" type="move_base" name="move_base" output="screen">
        <param name="base_local_planner" value="dwa_local_planner/DWAPlannerROS"/>
        <rosparam file="$(find mobile_robotics)/config/costmap_common_params.yaml"
command="load" ns="global_costmap" />
        <rosparam file="$(find mobile_robotics)/config/costmap_common_params.yaml"
command="load" ns="local_costmap" />
        <rosparam file="$(find mobile_robotics)/config/local_costmap_params.yaml"
command="load" />
        <rosparam file="$(find mobile_robotics)/config/global_costmap_params.yaml"
command="load" />
        <rosparam file="$(find mobile_robotics)/config/trajectory_planner.yaml"
command="load" />
        <rosparam file="$(find mobile_robotics)/config/move_base_params.yaml"
command="load" ns="global_costmap" />
```

```
        <rosparam file="$(find
mobile_robotics)/config/dwa_local_planner_params_waffle_pi.yaml" command="load"
ns="global_costmap" />
        </node>

        <!-- Launches return_home node for taking TurtleBot3 home after exploration -->
        <node pkg="mobile_robotics" type="return_home.py" name="return_home"
        output="screen">
        <rosparam file="$(find mobile_robotics)/config/exploration.yaml" command="load" />
        </node>

        <!-- Launches get_package node for setting package location as goal -->
        <node pkg="mobile_robotics" type="get_package.py" name="get_package"
        output="screen">
        </node>

        <!-- Launches surveying node -->
        <node pkg="mobile_robotics" type="survey1.py" name="survey_one" output="screen">
        </node>

        <node pkg="mobile_robotics" type=”package_search.py" name="package_search"
        output="screen">
        </node>

        <!-- Launches explore_lite package with params for frontier exploration -->
        <node pkg="explore_lite" type="explore" respawn="true" name="explore"
        output="screen">
        <rosparam file="$(find mobile_robotics)/config/exploration.yaml" command="load" />
        </node>

</launch>
```

Figure A1: tb_explore.launch File for Starting ROS Packages

```
#return_home.py node
#Makes TurtleBot3 to return to initial pose at home when finished exploring all possible frontiers

import rospy
```

```
import actionlib
from std_msgs.msg import String
from move_base_msgs.msg import MoveBaseActionGoal, MoveBaseAction, MoveBaseGoal
from nav_msgs.msg import Odometry

# Declaration of variables which will store position and orientation of initial pose
i = 0

homePos_x = 0
homePos_y = 0
homePos_z = 0

homeOri_x = 0
homeOri_y = 0
homeOri_z = 0
homeOri_w = 0

# Pipeline set-up
def listener():
    # Set-up subscribers
    data = rospy.Subscriber('/odom', Odometry, callback)
    start = rospy.Subscriber('/found_packages', String, movebase_client)

    # Initialize node
    rospy.init_node('return_home', anonymous=False)

    # Connect to move_base
    client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
    client.wait_for_server()

    # Prevents exit of node
    rospy.spin()

def movebase_client(start):
    # Calling global variables
    global homePos_x
    global homePos_y
    global homePos_z

    global homeOri_x
```

```
global homeOri_y
global homeOri_z
global homeOri_w

# Acts as a declaration
client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
client.wait_for_server()

# Node msg setup
# Header of msg
goal = MoveBaseGoal()
goal.target_pose.header.frame_id = "map"
goal.target_pose.header.stamp = rospy.Time.now()

# Set position
goal.target_pose.pose.position.x = homePos_x
goal.target_pose.pose.position.y = homePos_y
goal.target_pose.pose.position.z = homePos_z

# Set orientation
goal.target_pose.pose.orientation.x = homeOri_x
goal.target_pose.pose.orientation.y = homeOri_y
goal.target_pose.pose.orientation.z = homeOri_z
goal.target_pose.pose.orientation.w = homeOri_w

# Send msg
client.send_goal(goal)

# Waits for server to finish performing action
wait = client.wait_for_result

# If no result, assume server unavailable
if not wait:
   rospy.logerr("Action server not available!")
   rospy.signal_shutdown("Action sever not available!")
else:
   #Result
   return client.get_result()
   rospy.loginfo("Franklin is returning home!")
```

```
# Get initial position and orientation from /odom
def callback(data):
    # Calling global variable
    global i

    global homePos_x
    global homePos_y
    global homePos_z

    global homeOri_x
    global homeOri_y
    global homeOri_z
    global homeOri_w

    # Ensure only happens initially
    if i<1:
        # Save position
        homePos_x = data.pose.pose.position.x
        homePos_y = data.pose.pose.position.y
        homePos_z = data.pose.pose.position.z

        # Save orientation
        homeOri_x = data.pose.pose.orientation.x
        homeOri_y = data.pose.pose.orientation.y
        homeOri_z = data.pose.pose.orientation.z
        homeOri_w = data.pose.pose.orientation.w

        # Print position and orientation
        print("Home Position:")
        print(data.pose.pose.position)
        print("Home Orientation:")
        print(data.pose.pose.orientation)
        i+=1

if __name__ == '__main__':
    try:
        listener()
    except rospy.ROSInterruptException:
        pass
```

Figure A2: return_home.py Node Code

```
// Edited code within tb_explore.cpp node
// New topic /finished_exploring published when exploration stopped

void Explore::stop()
{
  move_base_client_.cancelAllGoals();
  exploring_timer_.stop();
  ROS_INFO("Exploration stopped.");
  ros::NodeHandle n;
  ros::Publisher return_pub = n.advertise<std_msgs::String>("/finished_exploring", 1000);
  std_msgs::String msg;
  std::stringstream ss;
  ss << "return";
  msg.data = ss.str();
  ROS_INFO("%s", msg.data.c_str());
  return_pub.publish(msg);
  std::cout << "FINISHED EXPLORING. EXPLORATION STOPPED";
}

}  // namespace explore

int main(int argc, char** argv)
{
  ros::init(argc, argv, "explore");
  ros::NodeHandle n;
  ros::Publisher return_pub = n.advertise<std_msgs::String>("/finished_exploring", 1000);
  if (ros::console::set_logger_level(ROSCONSOLE_DEFAULT_NAME,
                                     ros::console::levels::Debug)) {
    ros::console::notifyLoggerLevelsChanged();
  }
```

Figure A3: Edited tb_explore.cpp Node Code

```
#marker.py node for marking packages on map by using /fiducial_transforms topic
#!/usr/bin/env python
```

```
import rospy
import sys
import tf
import tf2_ros
import tf2_geometry_msgs
import actionlib

from visualization_msgs.msg import Marker
from move_base_msgs.msg import MoveBaseActionGoal, MoveBaseAction, MoveBaseGoal
from fiducial_msgs.msg import FiducialTransformArray
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
from geometry_msgs.msg import TransformStamped
from std_msgs.msg import String
from geometry_msgs.msg import PoseStamped


class marker:

    def __init__(self):
        self.markerPub = rospy.Publisher("/fiducials", Marker, queue_size = 10)
        self.countPub = rospy.Publisher("/package_count", PoseStamped, queue_size = 10)
        self.obstaclePub = rospy.Publisher("/obstacle_location", PoseStamped, queue_size = 10)
        self.speedPub = rospy.Publisher("/cmd_vel", Twist, queue_size = 10)
        self.pickupPub = rospy.Publisher("package_here", PoseStamped, queue_size=10)
        #self.pickupSub = rospy.Subscriber("/go_to_package", String, self.pickup_callback)
        self.imuSub = rospy.Subscriber("/odom", Odometry, self.callbackImu)
        #self.velSub = rospy.Subscriber("/cmd_vel", Twist, self.callbackVel)
        self.poseCollect = rospy.Subscriber("/fiducial_transforms", FiducialTransformArray,
self.callbackMarker)

        self.client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
        self.client.wait_for_server()

        #self.tfBuffer = tf2_ros.Buffer()
        #self.listener = tf2_ros.TransformListener(self.tfBuffer)
        #self.tf1_listener = tf.TransformListener()
        self.marker1 = 0
        self.marker2 = 0
```

```
    self.tfl = tf.TransformListener()
    self.markerX = 0
    self.markerY = 0
    self.markerZ = 0



#def callbackVel(self, data):
#    self.velX = data.linear.x

def callbackImu(self, data):
    self.posX = data.pose.pose.position.x
    self.posY = data.pose.pose.position.y
    self.posZ = data.pose.pose.position.z

def callbackMarker(self, data):

    self.see_package = 1
    for m in data.transforms:
        id = m.fiducial_id
        frame_id = 'camera_rgb_optical_frame'
        trans = m.transform.translation
        rot = m.transform.rotation
        marker = Marker()
        #package_to_camera = tf2_geometry_msgs.PoseStamped()
        package_pose = PoseStamped()
        count = PoseStamped()
        box_loc = PoseStamped()

        #package_to_camera.header.frame_id = frame_id
        #package_to_camera.pose.position.x = trans.x
        #package_to_camera.pose.position.y = trans.y
        #package_to_camera.pose.position.z = trans.z

        if id == 13:
            #marker.color.r = 0.5 #0
            #marker.color.g = 0.5 #1
            #marker.color.b = 0.9
            #marker.color.a = 1
```

```
            marker.mesh_resource =
'package://mobile_robotics/Charizard/Charizard_ColladaMax.DAE'
#'package://mobile_robotics/Snorlax/snorlax.dae'
            marker.mesh_use_embedded_materials = True
            marker.pose.orientation.x = rot.x -0.75
            marker.pose.orientation.y = rot.y
            marker.pose.orientation.z = rot.z
            marker.pose.orientation.w = rot.w

            marker.scale.x = 0.05
            marker.scale.y = 0.05
            marker.scale.z = 0.05
            self.marker1 = 1

        if id == 69:
            #marker.color.r = 1
            #marker.color.g = 0.5 #0
            #marker.color.b = 0.2
            #marker.color.a = 1
            marker.mesh_resource = 'package://mobile_robotics/coke.3ds'
            marker.mesh_use_embedded_materials = True
            marker.pose.orientation.x = rot.x
            marker.pose.orientation.y = rot.y - 1
            marker.pose.orientation.z = rot.z + 1.2
            marker.pose.orientation.w = rot.w

            marker.scale.x = .0015
            marker.scale.y = .0015
            marker.scale.z = .0015
            self.marker2 = 1


        package_pose.header.frame_id = frame_id
        package_pose.pose.position.x = trans.x
        package_pose.pose.position.y = trans.y
        package_pose.pose.position.z = trans.z
        package_pose.pose.orientation.x = rot.x
        package_pose.pose.orientation.y = rot.y
        package_pose.pose.orientation.z = rot.z
        package_pose.pose.orientation.w = rot.w
```

```
        if self.tfl.canTransform('/map', frame_id, rospy.Time(0)):

            self.package_transform = self.tfl.transformPose('/map',package_pose)

            package_pose.pose.position.x = self.package_transform.pose.position.x
            package_pose.pose.position.y = self.package_transform.pose.position.y
            package_pose.pose.position.z = id

            box_loc.pose.position.x = self.package_transform.pose.position.x
            box_loc.pose.position.y = self.package_transform.pose.position.y
            box_loc.pose.position.z = self.package_transform.pose.position.z

            self.obstaclePub.publish(package_pose)
            self.pickupPub.publish(box_loc)

        if trans.y < 1.25:
            marker.header.frame_id = frame_id
            marker.id = id
            marker.type = marker.MESH_RESOURCE
            marker.action = marker.ADD
            marker.pose.position.x = trans.x
            marker.pose.position.y = trans.y
            marker.pose.position.z = trans.z

            count.pose.position.x = self.marker1 + self.marker2
            self.countPub.publish(count)
            self.markerPub.publish(marker)
            #self.countPub.publish(count)

    def pickup_callback(self,data):
        self.see_package = 0
        goal = MoveBaseGoal()
        self.rangeXh = goal.target_pose.pose.position.x + 0.03
        self.rangeXl = goal.target_pose.pose.position.x - 0.03
        self.rangeYh = goal.target_pose.pose.position.y + 0.03
        self.rangeYl = goal.target_pose.pose.position.y - 0.03
        stop = Twist()
        print(self.posX)
        print(self.posY)
```

```
        print("range")
        print(self.rangeXh)
        while (self.posX > self.rangeXl and self.posX < self.rangeXh) and (self.posY > self.rangeYl
and self.posY<self.rangeYh): #or self.see_package == 0:

            goal.target_pose.header.frame_id = "map"
            goal.target_pose.header.stamp = rospy.Time.now()

            goal.target_pose.pose.position.x = self.markerX #self.package_transform.pose.position.x
            goal.target_pose.pose.position.y = self.markerY #self.package_transform.pose.position.y
            goal.target_pose.pose.position.z = self.markerZ #self.package_transform.pose.position.z

            goal.target_pose.pose.orientation.x = 0 #self.package_transform.pose.orientation.x
            goal.target_pose.pose.orientation.y = 0 #self.package_transform.pose.orientation.y
            goal.target_pose.pose.orientation.z = 0 #self.package_transform.pose.orientation.z
            goal.target_pose.pose.orientation.w = 1

            self.client.send_goal(goal)
            print("marker")

        stop.linear.x = 0
        stop.linear.y = 0
        stop.linear.z = 0

        stop.angular.x = 0
        stop.angular.y = 0
        stop.angular.z = 0

        self.speedPub.publish(stop)




def main(args):
    rospy.init_node('package_marker', anonymous=False)
    marker()
    try:
        rospy.spin()
```

```
    except rospy.ROSInterruptException:
        print("Shutting Down")
        pass

if __name__ == '__main__':
    main(sys.argv)
```

Figure A4: marker.py Node Code

```
//fake_laser_publisher node created using open source code on Github
//Publishes /fake_scan topic

#include <ros/ros.h>
#include <sensor_msgs/LaserScan.h>
#include "std_msgs/String.h"
#include <stdlib.h>
#include <cmath>
#include <tf/transform_datatypes.h>

#define PI 3.14159265359

  double tfX13; // fetch x coordinate of fiducial 13 relative to map here
  double tfY13; // fetch y coordinate of fiducial 13 relative to map
  double tfX69; // fetch x coordinate of fiducial 69 relative to map
  double tfY69; // fetch y coordinate of fiducial 69 relative to map
  double tfID;
//Retrieve the coordinates of one or both fiducials
void obstacleCallback(const geometry_msgs::PoseStamped& pose)
{
    tfID = pose.pose.position.z;
    if(tfID == 13){
        tfX13 = pose.pose.position.x;
        tfY13 = pose.pose.position.y;
    }
    if(tfID == 69){
        tfX69 = pose.pose.position.x;
        tfY69 = pose.pose.position.y;
    }
```

```
}

int main(int argc, char** argv){
  ros::init(argc, argv, "fake_laser_publisher");

  ros::NodeHandle n;
  ros::Publisher scan_pub = n.advertise<sensor_msgs::LaserScan>("fake_scan", 50);
  ros::Subscriber sub = n.subscribe("obstacle_location", 1000, obstacleCallback);
  unsigned int num_readings = 1440;
  double laser_frequency = 50;
  double ranges[num_readings];
  double intensities[num_readings];
  double scanAngle13;
  double scanAngle69;
  double tfPoseIncrement13;
  double tfPoseIncrement69;


  int count = 0;
  srand(time(0));
  ros::Rate r(1.0);
  while(ros::ok()){
//Test to see what Quadrant the fiducial position is in.
if(tfX13 > 0 && tfY13 > 0){ //Q1

    scanAngle13 = atan(tfY13/tfX13);

  }

  if(tfX13 < 0 && tfY13 > 0){ //Q2

    scanAngle13 = atan(tfY13/tfX13) + PI;

  }

  if(tfX13 < 0 && tfY13 < 0){ //Q3

    scanAngle13 = PI + atan(tfY13/tfX13);

  }
```

```
if(tfX13 > 0 && tfY13 < 0){ //Q4

  scanAngle13 = atan(tfY13/tfX13) + 2*PI;

}

if(tfX13 == 0 || tfY13 == 0){
        if(tfX13 == 0 && tfY13 != 0){
                if(tfY13 > 0){
                scanAngle13 = PI/2;
                }
                if(tfY13 < 0){
                scanAngle13 = PI + PI/2;
                }
        }
        if(tfX13 != 0 && tfY13 == 0){
                if(tfX13 > 0){
                scanAngle13 = 0;
                }
                if(tfX13 < 0){
                scanAngle13 = PI;
                }
        }

}

if(tfX69 > 0 && tfY69 > 0){ //Q1

  scanAngle69 = atan(tfY69/tfX69);

}

if(tfX69 < 0 && tfY69 > 0){ //Q2

  scanAngle69 = atan(tfY69/tfX69) + PI;

}

if(tfX69 < 0 && tfY69 < 0){ //Q3
```

```
  scanAngle69 = PI + atan(tfY69/tfX69);

}

if(tfX69 > 0 && tfY69 < 0){ //Q4

  scanAngle69 = atan(tfY69/tfX69) + PI*2;

}

if(tfX69 == 0 || tfY69 == 0){
        if(tfX69 == 0 && tfY69 != 0){
                if(tfY69 > 0){
                scanAngle69 = PI/2;
                }
                if(tfY69 < 0){
                scanAngle69 = PI + PI/2;
                }
        }
        if(tfX69 != 0 && tfY69 == 0){
                if(tfX69 > 0){
                scanAngle69 = 0;
                }
                if(tfX69 < 0){
                scanAngle69 = PI;
                }
        }

}

tfPoseIncrement13 = (scanAngle13)/(0.25*PI/180);
tfPoseIncrement69 = (scanAngle69)/(0.25*PI/180);

  //generate ranges for fake LaserScan
  for(unsigned int i = 0; i < num_readings; ++i){
    if(i > (tfPoseIncrement13 - 5) && i < (tfPoseIncrement13 + 5)){
       ranges[i] = sqrt(pow(tfX13,2) + pow(tfY13,2));
    }
    else if(i > (tfPoseIncrement69 - 10) && i < (tfPoseIncrement69 + 10)){
```

```
        ranges[i] = sqrt(pow(tfX69,2) + pow(tfY69,2));
      }
      else{
      ranges[i] = 0;
      }
      intensities[i] = 10;
    }
    ros::Time scan_time = ros::Time::now();

    //Set fake laser scan parameters
    //then populate the LaserScan message
    sensor_msgs::LaserScan scan;
    scan.header.stamp = scan_time;
    scan.header.frame_id = "map";
    scan.angle_min = 0; //angle correspond to FIRST beam in scan ( in rad)
    scan.angle_max = 360 * (PI/180); //angle correspond to LAST beam in scan ( in rad)
    scan.angle_increment = 0.25 * (PI/180); // Angular resolution i.e angle between 2 beams
    scan.time_increment = (1 / 50) / (1440);
    scan.range_min = 0.02;
    scan.range_max = 20.0;

    scan.ranges.resize(num_readings);
    scan.intensities.resize(num_readings);
    for(unsigned int i = 0; i < num_readings; ++i){
      scan.ranges[i] = ranges[i];
      scan.intensities[i] = intensities[i];
    }

    scan_pub.publish(scan);

    //r.sleep();
    ros::spinOnce();
  }
}
```

Figure A5: fake_laser_publisher.cpp Node Code

```
#package_search node for surveying for packages
```

```
#! /usr/bin/python

import rospy
import sys
import actionlib

from std_msgs.msg import String
from move_base_msgs.msg import MoveBaseActionGoal, MoveBaseAction, MoveBaseGoal
from nav_msgs.msg import Odometry
from geometry_msgs.msg import PoseStamped

from occupancy_grid_python import OccupancyGridManager

class packageSearch():

    def __init__(self):
        #self.searchPub =
        self.goHomePub = rospy.Publisher("/go_to_package", String, queue_size = 10)
        self.goSurveyPub = rospy.Publisher("/survey1", String, queue_size = 10)
        self.packageCountSub = rospy.Subscriber('/package_count', PoseStamped, self.count_Cb)
        self.startSub = rospy.Subscriber('/finished_exploring', String, self.callback)
        self.location = rospy.Subscriber('/odom', Odometry, self.location_Cb)
        self.ogm = OccupancyGridManager('/move_base/global_costmap/costmap',
subscribe_to_updates=True)
        self.count = 0
        self.client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
        self.client.wait_for_server()


    def location_Cb(self,data):
        self.x = data.pose.pose.position.x
        self.y = data.pose.pose.position.y
        self.ori_x = data.pose.pose.orientation.x
        self.ori_y = data.pose.pose.orientation.y
        self.ori_z = data.pose.pose.orientation.z
        self.ori_w = data.pose.pose.orientation.w

    def count_Cb(self,data):
        self.count = data.pose.position.x
```

```
#print("LOLOLOLOLOLOLOLOLOLOLOLOLOLOLOOLOLOLOLOLOLOLOLOLOLOLOLOOL"
)
        #print(self.count)

    def search_Cb(self):
        #print("HI HI H IH IH IH IH IH I HI HI HI HI HI HI H IH IH IH I")
        self.run = True
        self.cost = 0
        self.checkX = self.x
        self.checkY = self.y
        goal = MoveBaseGoal()
        goal.target_pose.header.frame_id = "map"
        goal.target_pose.header.stamp = rospy.Time.now()

        if self.count != 2:
            #print("jajajajajajajajajajajajajajajajajajajajajaja")
            self.int = 0
            self.cornerX = 0
            self.cornerY = 0
            self.check = 1000

            while  self.count != 2:
                #print("Go to wall!!!!!!!")
                if (self.cost>50 or self.cost<40) and self.int == 0:
                    self.checkX = self.checkX + 0.05
                    self.cost = self.ogm.get_cost_from_world_x_y(self.checkX, self.checkY)
                elif self.int == 0:
                    goal.target_pose.pose.position.x = self.checkX
                    goal.target_pose.pose.position.y = self.checkY
                    goal.target_pose.pose.position.z = 0

                    goal.target_pose.pose.orientation.x = self.ori_x
                    goal.target_pose.pose.orientation.y = self.ori_y
                    goal.target_pose.pose.orientation.z = self.ori_z
                    goal.target_pose.pose.orientation.w = self.ori_w

                    if self.check == 1000:
                        self.client.send_goal(goal)
                        self.check = 0
```

```
                print("ONE ONE ONE ONE ONE")

            #self.client.wait_for_result
            self.check = self.check + 1

            self.rangeXh = self.checkX + 0.25
            self.rangeXl = self.checkX - 0.25
            self.rangeYh = self.checkY + 0.25
            self.rangeYl =  self.checkY -0.25
            #print(self.x)
            #print(self.y)
            #print(self.checkX)
            #print(self.checkY)
            if (self.x > self.rangeXl and self.x<self.rangeXh) and (self.y > self.rangeYl and
self.y<self.rangeYh):
                self.int = 1
                self.cost = 0
                self.check = 100
        elif (self.cost>=50 or self.cost<=40) and self.int == 1:
            self.cornerX = self.cornerX + 0.05
            self.cost = self.ogm.get_cost_from_world_x_y(self.cornerX, self.cornerY)
        elif (self.cost<50 or self.cost>40) and self.int == 1:
            goal.target_pose.pose.position.x = self.cornerX
            goal.target_pose.pose.position.y = self.cornerY
            goal.target_pose.pose.position.z = 0

            goal.target_pose.pose.orientation.x = self.ori_x
            goal.target_pose.pose.orientation.y = self.ori_y
            goal.target_pose.pose.orientation.z = self.ori_z
            goal.target_pose.pose.orientation.w = self.ori_w

            if self.check == 1000:
                self.client.send_goal(goal)
                self.check = 0
                print("TWO TWO TWO TWO TWO  TWO TWO TWO")

            self.check = self.check + 1
            self.rangeXh = self.cornerX + 0.25
            self.rangeXl = self.cornerX - 0.25
            self.rangeYh = self.cornerY + 0.25
```

```
            self.rangeYl =  self.cornerY -0.25

            if (self.x > self.rangeXl and self.x<self.rangeXh) and (self.y > self.rangeYl and
self.y<self.rangeYh):
                self.int = 2
                self.cost = 0
                self.cornerX = 0
                self.check = 100
                #print("DONE TWO DONE TWO DONE TWO DONE TWO")
            elif (self.cost>=50 or self.cost<=40) and self.int == 2:

                self.cornerY = self.cornerY + 0.05
                self.cost = self.ogm.get_cost_from_world_x_y(self.cornerX, self.cornerY)
            elif (self.cost<50 or self.cost>40) and self.int == 2:
                if self.cornerY > 1:
                    goal.target_pose.pose.position.x = self.cornerX
                    goal.target_pose.pose.position.y = self.cornerY
                    goal.target_pose.pose.position.z = 0

                    goal.target_pose.pose.orientation.x = self.ori_x
                    goal.target_pose.pose.orientation.y = self.ori_y
                    goal.target_pose.pose.orientation.z = self.ori_z
                    goal.target_pose.pose.orientation.w = self.ori_w

                    if self.check == 1000:
                        self.client.send_goal(goal)
                        self.check = 0
                    print("THREE THREE THREE THREE THREE THREE")

                    self.check = self.check + 1
                    self.rangeXh = self.cornerX + 0.25
                    self.rangeXl = self.cornerX - 0.25
                    self.rangeYh = self.cornerY + 0.25
                    self.rangeYl =  self.cornerY -0.25
                    if (self.x > self.rangeXl and self.x<self.rangeXh) and (self.y > self.rangeYl and
self.y<self.rangeYh):
                        self.int = 3
                        self.cost = 0
                        self.cornerY = 0
                        self.check = 100
```

```
            else:
                #print(self.cornerY)
                #print(self.cornerY)
                #print(self.cornerY)
                #print(self.cornerY)
                #print(self.cornerY)
                self.int = 3
                self.cost = 0
                self.cornerY = 0
        elif (self.cost>=50 or self.cost<=40) and self.int == 3:

            self.cornerY = self.cornerY - 0.05
            self.cost = self.ogm.get_cost_from_world_x_y(self.cornerX, self.cornerY)
        elif (self.cost<50 or self.cost>40) and self.int == 3:
            if self.cornerY < -1:
                goal.target_pose.pose.position.x = self.cornerX
                goal.target_pose.pose.position.y = self.cornerY
                goal.target_pose.pose.position.z = 0

                goal.target_pose.pose.orientation.x = self.ori_x
                goal.target_pose.pose.orientation.y = self.ori_y
                goal.target_pose.pose.orientation.z = self.ori_z
                goal.target_pose.pose.orientation.w = self.ori_w

                if self.check == 1000:
                    self.client.send_goal(goal)
                    self.check = 0
                print("FOUR FOUR FOUR FOUR FOUR FOUR FOUR FOUR")

                self.check = self.check + 1
                self.rangeXh = self.cornerX + 0.25
                self.rangeXl = self.cornerX - 0.25
                self.rangeYh = self.cornerY + 0.25
                self.rangeYl =  self.cornerY -0.250
                if (self.x > self.rangeXl and self.x<self.rangeXh) and (self.y > self.rangeYl and
self.y<self.rangeYh):
                    #print(self.cornerY)
                    #print(self.cornerY)
                    #print(self.cornerY)
                    #print(self.cornerY)
```

```
                #print(self.cornerY)
                self.int = 0
                self.cost = 0
                self.cornerY = 0
                self.check = 1000
            else:
                self.int = 0
                self.cost = 0
                self.cornerY = 0


    print("BEFORE BEFORE BEFORE BEFORE BEFORE")
    self.goHomePub.publish("go home")
    print("AFTER AFTER AFTER AFTER AFTER AFTER")

def callback(self,data):
    client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
    client.wait_for_server()
    self.int = 0

    # Node msg setup
    # Header of msg
    while self.count == 0:
        if self.int == 0:
            goal = MoveBaseGoal()
            goal.target_pose.header.frame_id = "map"
            goal.target_pose.header.stamp = rospy.Time.now()

            # Set position
            goal.target_pose.pose.position.x = 0
            goal.target_pose.pose.position.y = 2.5
            goal.target_pose.pose.position.z = 0

            # Set orientation
            goal.target_pose.pose.orientation.x = 0
            goal.target_pose.pose.orientation.y = 0
            goal.target_pose.pose.orientation.z = 0
            goal.target_pose.pose.orientation.w = 1

            # Send msg
```

```
            client.send_goal(goal)

            self.rangeXh = goal.target_pose.pose.position.x + 0.2
            self.rangeXl = goal.target_pose.pose.position.x - 0.2
            self.rangeYh = goal.target_pose.pose.position.y + 0.2
            self.rangeYl = goal.target_pose.pose.position.y - 0.2
          if (self.x > self.rangeXl and self.x<self.rangeXh) and (self.y > self.rangeYl and
self.y<self.rangeYh):
            self.goSurveyPub.publish("next point")
            self.int = 1

       if self.int == 0:
          self.goHomePub.publish("go home")









def main(args):
   rospy.init_node('package_search', anonymous=False)
   cs = packageSearch()
   try:
      rospy.spin()
   except rospy.ROSInterruptException:
      print("Shutting Down")
      pass


if __name__ == '__main__':
   main(sys.argv)
```

Figure A6: package_search.py

```
#get_package node used to go to package location
```

```
#!/usr/bin/env python

import rospy
import sys
import actionlib

from move_base_msgs.msg import MoveBaseGoal, MoveBaseAction, MoveBaseActionGoal
from geometry_msgs.msg import Twist
from std_msgs.msg import String
from nav_msgs.msg import Odometry
from geometry_msgs.msg import PoseStamped

class grab_package:

    def __init__(self):
        self.pickupPub = rospy.Publisher("/grab_em", String, queue_size = 10)
        self.speedPub = rospy.Publisher("/cmd_vel", Twist, queue_size = 10)
        self.homePub = rospy.Publisher("/found_packages", String, queue_size = 10)
        self.packageSub = rospy.Subscriber("/package_here", PoseStamped, self.grab)
        self.pickupSub = rospy.Subscriber("/go_to_package", String, self.pickup_callback)
        self.imuSub = rospy.Subscriber("/odom", Odometry, self.callbackImu)
        self.int = 0

        self.client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
        self.client.wait_for_server()

    def callbackImu(self, data):
        self.posX = data.pose.pose.position.x
        self.posY = data.pose.pose.position.y
        self.posZ = data.pose.pose.position.z

    def grab(self, data):
        self.box_X = data.pose.position.x
        self.box_Y = data.pose.position.y
        self.box_Z = data.pose.position.z
        print(self.box_X)
        print(self.box_Y)
        print(self.box_Z)

    def pickup_callback(self, data):
```

```
client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
client.wait_for_server()
#self.see_package = 0
#self.int = 0

#x = self.box_X
#y = self.box_Y
#z = self.box_Z


#print(self.posX)
#print(self.posY)
#print("range")
#print(self.rangeXh)
#print(self.rangeXh)
#print(self.rangeYh)
#print(self.rangeXh)
while self.int == 0:


    goal = MoveBaseGoal()
    goal.target_pose.header.frame_id = "map"
    goal.target_pose.header.stamp = rospy.Time.now()

    goal.target_pose.pose.position.x = self.box_X #self.package_transform.pose.position.x
    goal.target_pose.pose.position.y = self.box_Y  #self.package_transform.pose.position.y
    goal.target_pose.pose.position.z = self.box_Z #self.package_transform.pose.position.z

    goal.target_pose.pose.orientation.x = 0 #self.package_transform.pose.orientation.x
    goal.target_pose.pose.orientation.y = 0 #self.package_transform.pose.orientation.y
    goal.target_pose.pose.orientation.z = 0 #self.package_transform.pose.orientation.z
    goal.target_pose.pose.orientation.w = 1

    client.send_goal(goal)
    #print("get_package")
    #print(x)
    #print(y)
    #print(self.posX)
    #print(self.posY)
```

```
            rangeXh = self.box_X + 0.1
            rangeXl = self.box_X - 0.1
            rangeYh = self.box_Y + 0.1
            rangeYl = self.box_Y - 0.1

            if (self.posX > rangeXl and self.posX < rangeXh) and (self.posY > rangeYl and
self.posY<rangeYh):
                self.int = 1
                self.pickupPub.publish("stop")
                #self.homePub.publish("stop")
                print("donedonedonedonedonedonedonedonedonedonde")
                client.cancel_all_goals()

        stop = Twist()

        stop.linear.x = 0
        stop.linear.y = 0
        stop.linear.z = 0

        stop.angular.x = 0
        stop.angular.y = 0
        stop.angular.z = 0

        self.speedPub.publish(stop)
        #print("stopstopstopstopstopstopstopstopstopstopstopstop")

def main(args):
    rospy.init_node('get_package', anonymous=False)
    grab_package()
    try:
        rospy.spin()
    except rospy.ROSInterruptException:
        print("Shutting Down")
        pass

if __name__ == '__main__':
    main(sys.argv)
```

Figure A7: get_package.py Node Code

```
#GripperROS.py node for picking up package when in position
#! /usr/bin/python

import rospy
import RPi.GPIO as GPIO
import time
import sys

from std_msgs.msg import String

class gripper():

    def __init__(self):
        self.gotPub = rospy.Publisher("/got_em", String, queue_size = 10)
        self.atPackageSub = rospy.Subscriber("", String, self.callbackPick)
        self.placePackageSub = rospy.Subscriber("", String, self.callbackPlace)
        self.pwm_gripper = 15
        self.pwm_height = 16
        self.frequence = 50
        GPIO.setup(self.pwm_gripper, GPIO.OUT)
        GPIO.setup(self.pwm_height, GPIO.OUT)

        self.pwm = GPIO.PWM(pwm_gripper, frequence)
        self.pwm1 = GPIO.PWM(pwm_height, frequence)
        GPIO.setmode(GPIO.BOARD) #Use Board numerotation mode
        GPIO.setwarnings(False) #Disable warnings



    #Set function to calculate percent from angle
    def angle_to_percent(angle) :
        if angle > 180 or angle < 0 :
            return False

        start = 4
        end = 12.5
        ratio = (end - start)/180 #Calcul ratio from angle to percent
```

```
        angle_as_percent = angle * ratio

        return start + angle_as_percent

    def start(self)
    self.pwm.start(angle_to_percent(0))
    self.pwm1.start(angle_to_percent(90))
    time.sleep(1)


    def pickup(self):
        self.pwm.ChangeDutyCycle(angle_to_percent(70)) #close Gripper
        time.sleep(1)

        self.pwm1.ChangeDutyCycle(angle_to_percent(45)) #Lift
        time.sleep(1)

    def place(self):

        self.pwm1.ChangeDutyCycle(angle_to_percent(90)) #Open Gripper
        time.sleep(1)

        self.pwm.ChangeDutyCycle(angle_to_percent(0)) #Let down
        time.sleep(1)

    def callbackPick(self,data):
        pickup()

    def callbackPlace(self,data):
        place()

def main(args):
    rospy.init_node('gripper', anonymous=False)
    g = gripper()
    g.start()
    try:
        rospy.spin()
    except rospy.ROSInterruptException:
        print("Shutting Down")
        pass
```

```
if __name__ == '__main__':
    main(sys.argv)

    #Close GPIO & cleanup
    #pwm.stop()
    #GPIO.cleanup()
```

Figure A8: GripperROS.py Code