EG2310
Group 10
Final Report

Student Team Members:

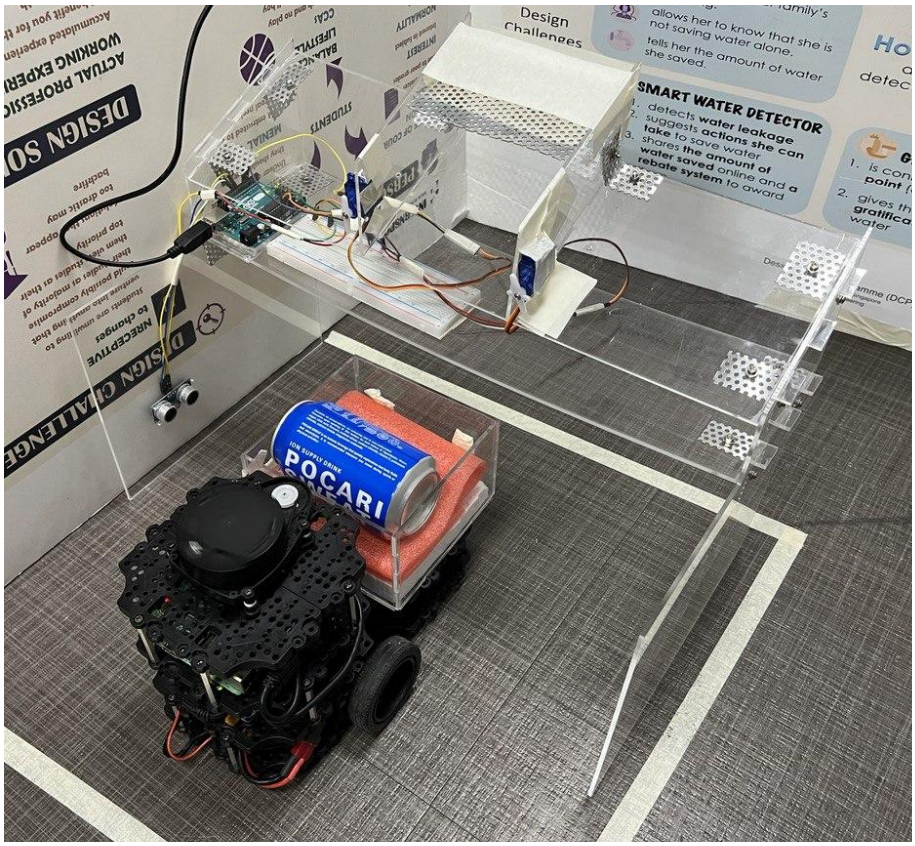
Nicholas Tan Yun Yu	A0254482B
Loh Ji Yong Aloysius	A0190205R
Hiew T G	A0252718B
Patrick Joy Surbakti	A0266288M

Table of Contents

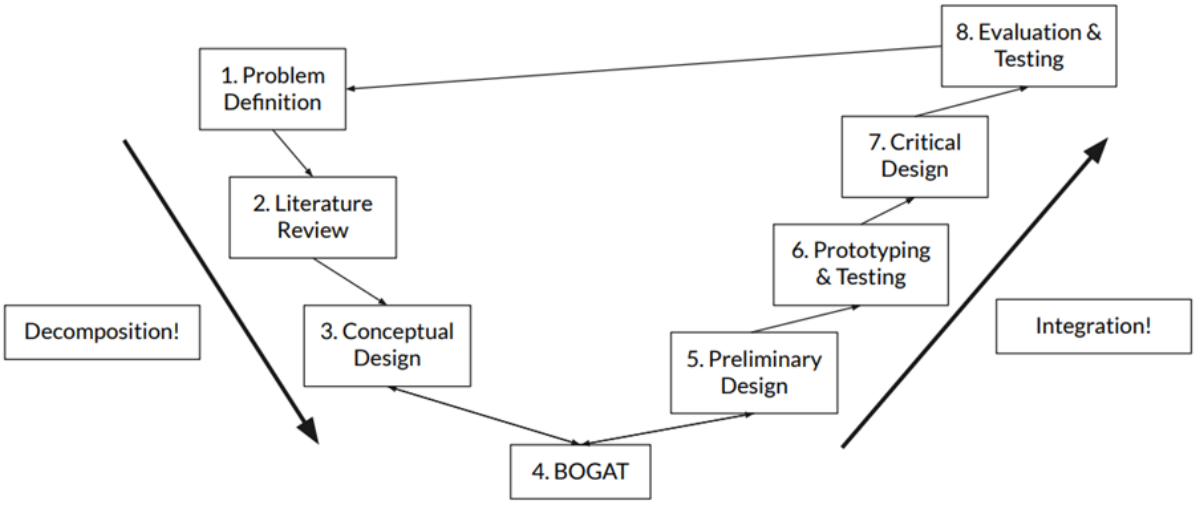
1. Introduction

This document describes the system design process of our EG2310 robotic system. The system was designed to achieve the following purpose:

> *To deliver a canned drink from a fixed dispenser to a designated table by navigating a restaurant-like map autonomously. Once the canned drink has been delivered and successfully received, it will return to the dispenser, ready for the whole process to be repeated again for the next designated table.*



The system design process utilised follows the modified EG2310 systems engineering V-Model as shown below:



NUS EG2310 System Design V-Model

The following Sections detail each part of the system design process:

1. V-Model parts 1 - 3, 5 and 6 are described in their respective Sections 2, 3, 4, 6 and 7
2. Critical Design is described in detail in Section 8
3. Evaluation & Testing is presented in Sections 11 and 12 with a list of possible future works

2. Problem Definition

The mission is for the Turtlebot to deliver a can from a fixed dispenser to a specific table autonomously in a simulated restaurant environment. To accomplish this robotic system, the following four subsystems can be devised: Dispenser Storage, Dispatching, Turtlebot Load-sensing Holder and Turtlebot Navigation.

2.1 Dispenser's Objectives:

D1 - The dispenser receives a can of soda manually loaded by a Teaching Assistant (TA).
D2 - The TA sets the destination table, among at least 6 tables, in the dispenser interface.
D3 - The dispenser places the can on the TurtleBot.
D4 - Then it waits for the next delivery.

Objective	**Requirements**
D1	The dispenser has a storage location to hold the can before it is being dispensed.
D2	There should be a means for sending instructions of the next delivery table to the Turtlebot.
D3	The dispenser should activate when the Turtlebot is in position to receive the can.
	The dispenser needs some mechanism to place the can onto the Turtlebot precisely.
D4	No more action by the dispenser till the Turtlebot comes back.

2.2 Turtlebot's Objectives:

T1 - The TurtleBot waits to receive the can
T2 - The robot autonomously navigates through the restaurant to the designated table.
T3 - The robot waits at the table until the TA picks up the can of soda.
T4 - The TurtleBot returns to the dispenser and waits to receive a new order.

Objective	**Requirements**
T1	The Turtlebot needs to sense when it has received the can from the dispenser.
T2	The Turtlebot should be familiar with the layout of the tables as well as its own location in the restaurant.
	The Turtlebot needs to plan an appropriate path from its current position to the customer.
T3	The Turtlebot should stop near enough to the customer's table.

	The Turtlebot needs to sense when the customer has taken the can.
T4	The Turtlebot should then move back to the dispenser automatically to prepare for the next delivery.

3. Literature Review

3.1 PuduBot

The PuduBot is an autonomous delivery robot that is designed to navigate and deliver items of interest to the instructed destination autonomously, with in-built technology to detect obstacles and prevent contact with them. The PuduBot has already been widely implemented in restaurants worldwide, boasting an excellent reputation of being an intelligent robot that is able to deliver not just food and beverages from the restaurants' menu to the customers' tables, but also tangible results to the businesses.

Hence, we believe that it will be wise to consider PuduBot as a good learning model for us to advance towards accomplishing our mission.

Subsystems of PuduBot

The 2 main subsystems of the PuduBot is as shown:

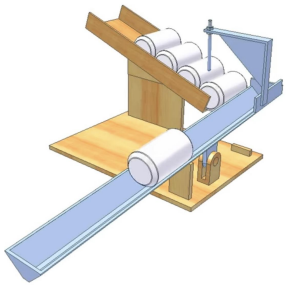
Navigation & Dispatching Subsystems of PuduBot

In our case, the Dispatching and Navigation subsystems of PuduBot are insufficient to carry us to the end goal of our mission. Therefore, we have added the Dispenser Storage subsystem and Turtlebot Load-Sensing Holder subsystem to cater to the need for a dispenser.

3.2 Dispenser Storage Subsystem

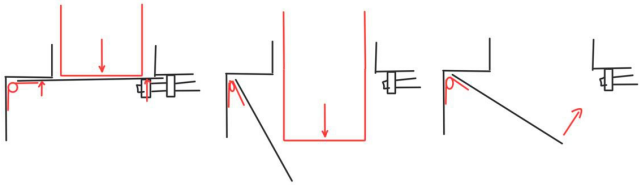
The Roll/Drop-down Mechanism

A roll-down/drop-down can dispenser mechanism is feasible for a canned drink to be loaded and dispensed. The diagram on the right illustrates that when the longer rail is tilted in the clockwise direction about the pivot, a can will be loaded and tilting it back to its original position in the anticlockwise direction about the pivot causes the can to automatically slide down the rail due to gravity together with an appropriate angle.



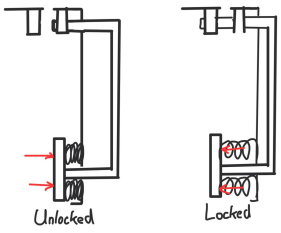
The Trapdoor

Taking inspiration from the mechanics of a regular trash can, we plan to incorporate a trapdoor mechanism with torsion spring. When the lock is released, the force of the spring itself will not be enough to hold the weight in place, which will then release the can. With the absence of the weight of the can, the spring will now have enough strength to push back the trapdoor up to its initial position. The lock should also be reset for the trapdoor to be able to hold the next can.



The Spring Lock Mechanism

The pedal will be attached on the inner wall of the dispenser and there will be springs installed in between the pedal and the wall. The pedal is connected to the lock and will act as the mechanism to lock and unlock it. When the pedal is pushed, the springs are compressed and the door will be opened. When the pedal is released, the potential force from the spring will push the pedal back to its position as well as re-locking the trapdoor.



HC-SR04 Ultrasonic Distance Sensor for Obstacle Distance Detection

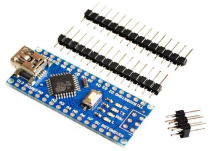
The sensor should be able to detect the incoming cart of the Turtlebot to send the signal to the Arduino board which then passes it to the servo motor to open the trapdoor.

Servo Motor

Servo Motor is a motor that can rotate with high angular precision through PWM (Pulse Width Modulation). It can be programmed to close the trapdoor (replacement for the torsion spring mentioned earlier) and hold in the can drink, and then release it when the ultrasonic sensor detects the Turtlebot.

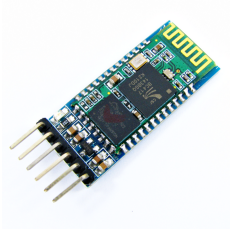
ATmega328P Microcontroller Board (Arduino Uno)

A programmable microcontroller to receive the signal from the ultrasonic sensor and control the servo motor to open and close the trapdoor.



3.3 Dispatching Subsystem

HC-05 Bluetooth Module



It is a low-cost, versatile bluetooth module based on the Bluetooth 2.0+EDR (Enhanced Data Rate) specification. Its built-in antenna is suitable for short-range communication for about 10 meters. It can be connected to the OpenCR board in the Turtlebot for wireless communication with the mobile application which will be used to dispatch the Turtlebot to collect the can from the dispenser before going to the specified table.

Physical Buttons to select delivery table number

By attaching 6 B3F-1020 push buttons to an Arduino Uno, we can use the buttons as a means of selecting any one of the table numbers for delivery.

MIT App Inventor Mobile Application

MIT App Inventor is a Graphic User Interface (GUI) used to create a mobile application that can allow the user to choose the table to be delivered to. It is versatile and able to integrate with the HC-05 Bluetooth Module as well as the OpenCR board on the Turtlebot for data transmission and communication purposes.

3.4 Turtlebot Load-sensing Holder Subsystem

The Cart

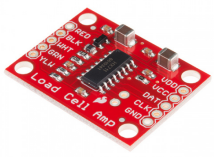
An open rectangular box will be attached to the back of the Turtlebot to serve as a can holder. Nevertheless, the cart needs to have a mechanism installed to detect whether the can drink is inside.

Load Cell

Load cell is a transducer. It can convert force to measurable electrical output. There are several types of load cells depending on what they are used for. There are pneumatic load cells, hydraulic load cells, and strain gauge load cells. The most widely used are the strain-gauge load cells. The gauges are connected to a beam or other types of structural member that deforms when weight is applied, which will also change the electrical resistance of the gauges proportionally to the load.

Load Cell Amplifier HX711

A load cell amplifier amplifies the strength of signals that are delivered from the load cells, which can sometimes be too weak to work with (usually in mV). The whole process of amplifying the signal is not causing any inaccuracies in the measured



value. The load cell together with the load cell amplifier works together to detect the weight of the can drinks when it is loaded.

Ball Caster

The can holder will be mounted with a ball caster which will provide a multi-angular movement rotational move to keep the rotation convenient.

Button

Another idea of the can drink sensing mechanism is to install a B3F-1020 push button at the bottom of the cart which is connected to the RPi. When the drink is loaded into the cart, the button is pushed to send the signal.

Light-Dependent Resistor (Photoresistor)

LDR is a resistor specifically designed to be sensitive to light which will change its resistivity. The idea is to install it at the bottom of the cart, facing upward. When the can is loaded, it will block the photoresistor, removing the presence of light and hence detecting the presence of the can.

3.5 Turtlebot Navigation Subsystem

LiDAR for Simultaneous Localization And Mapping (SLAM)

LiDAR (Light Detection and Ranging) is a remote sensing technology that measures distances of objects relative to itself. It emits laser beams in all directions, and then measures the time it takes for the beam to bounce back after hitting a surface or object. The sensor can then calculate the distance to the object and create a resultant 2D point cloud based on the surrounding environment.

LiDAR for Monte Carlo Localization (MCL)

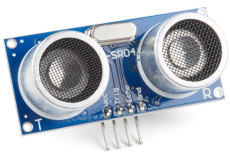
MCL uses the particle filtering technique to estimate the location of the Turtlebot within the environment. It involves initialising a set of particles to represent all possible locations of the Turtlebot, then resampling the particles based on their probabilities. The end result is a small group of particles on a 2D plane which represents the estimated location of the Turtlebot.

Raspberry Pi Camera Module 2 for Obstacle Detection

The Raspberry Pi Camera Module V2 is a small camera module that connects to the Raspberry Pi's Camera Serial Interface (CSI) bus. The module sends the image data over the CSI bus to the Raspberry Pi's processor. The Raspberry Pi can then use the image data to perform tasks such as object detection, facial recognition, or video streaming.

HC-SR04 Ultrasonic Distance Sensor for Obstacle Distance Detection

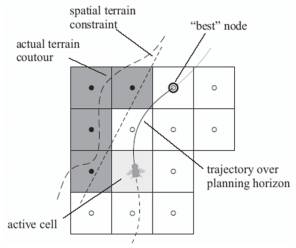
HC-SR04 uses ultrasound to measure the distance to an object. The sensor works by emitting an ultrasonic sound wave at a frequency of 40kHz, and then measuring the time it takes for the sound wave to bounce back after hitting an



object. The sensor can calculate the distance to the object with knowledge of the speed of sound. This module acts as a complementary feature to LiDAR.

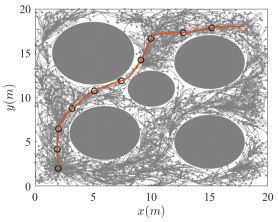
RH-A*

Receding Horizon Dynamic A* (RH-A*) is a variant of the A* algorithm that uses a receding horizon approach to continuously replan the path based on the updated information of the environment and the robot's state. It replans the path from the current location of the robot to the goal location using the A* algorithm based on the position of dynamic obstacles and repeats the process until the goal is reached.



Kino-RRT*

Kinodynamic RRT* (Kino-RRT*) is a variant of the Rapidly-exploring Random Trees (RRT*) algorithm. It uses a sampling-based approach to find the optimal path by generating feasible trajectories for the robot and selecting the one that reaches the goal location with the lowest cost. It is designed to handle dynamic environments and can find paths to avoid moving obstacles.



Simplified A*

This approach is a simplified version of the A* algorithm. By splitting the 2D map area into grids and finding the shortest path from the initial position to the goal, we are able to find a reasonably accurate path which performs adequate obstacle avoidance.

Waypoint Algorithm

The waypoint algorithm is one that allows the robot to move from its current location to its destination by following a set of waypoints. The robot first identifies its current pose, then identifies the position of the next waypoint. The robot then rotates in its current position to face the next waypoint and starts travelling towards it in a straight line until it reaches the desired waypoint.

4. Concept(s) Design

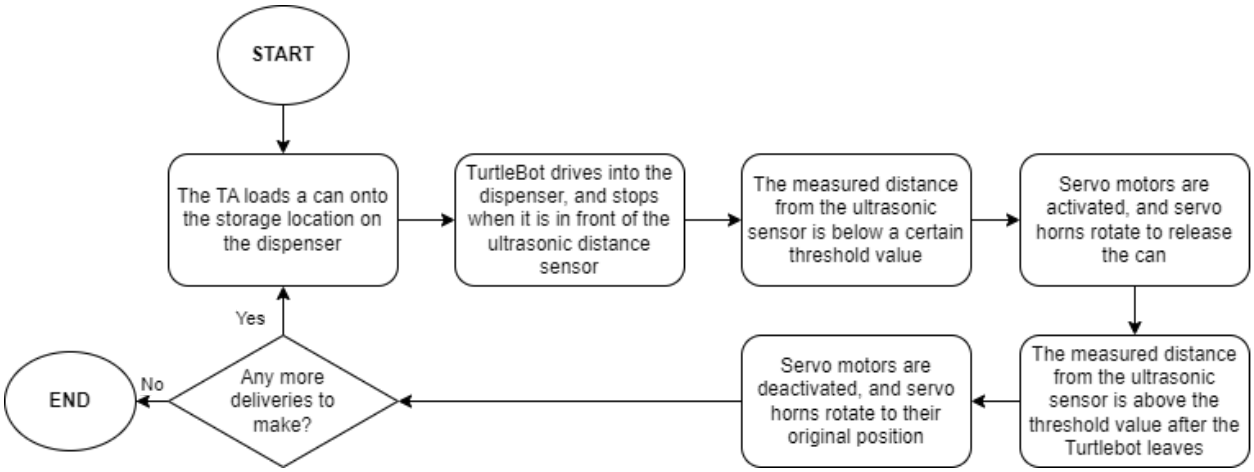
The main objective of the concept design stage is to identify and explore potential solutions to meet the customer's requirements. During this stage, we will identify the concepts considered and share our motivations for embarking on the chosen design. The output of this stage is a set of high-level requirements and a rough concept design.

4.1 Dispenser Storage Subsystem

The dispenser is built to simulate a drive-through. It is open on both sides (front and back) and will be placed sideways, with the left side of the dispenser aligned to the wall of the maze. By constructing the dispenser in this manner, we have greater flexibility in the approach to deploy the can, and with the pathways the Turtlebot can take to navigate the map.

The dispenser is designed to hold the load of a single can. The can is held in a fixed position on an inclined slope by 2 servo motors which are secured horizontally along the edge of the dispenser. The servo motors are connected to an Arduino Uno, which is the main microcontroller unit setting the logic to the electrical components used. The Arduino Uno is connected to an ultrasonic distance sensor mounted to the right side of the dispenser. This sensor faces inwards, and is responsible for detecting the presence of the incoming Turtlebot in the dispenser.

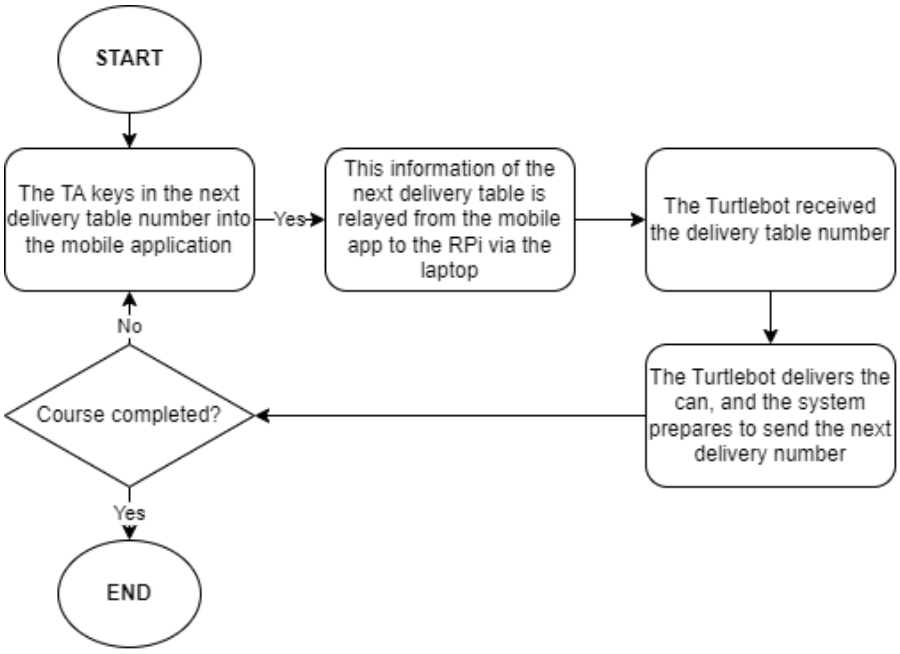
When the ultrasonic distance sensor measures a distance below a certain threshold value, it sends a signal to actuate the servo motors, causing the can to be released from its rest position. The can rolls off the inclined plane due to gravity and drops into the Turtlebot Load-sensing Holder Subsystem. Once the Turtlebot moves out of the dispenser, the ultrasonic distance sensor detects the change in distance measured and sends another signal to the servo motors to deactivate them, moving them back to their original position. Another can is then loaded into the loading area, to prepare for the next delivery.



4.2 Dispatching Subsystem

To simulate the dispatching interface commonly seen in food delivery robots such as the PudoBot, we created a mobile application to facilitate the process of dispatching orders to their respective tables. Using MIT App Inventor, a simple and effective GUI tool for developing mobile applications, we created a mobile application consisting of 2 buttons to connect and disconnect to the MQTT broker, 6 buttons for the respective 6 table numbers, and 1 button to send the order.

The user links the mobile application with the Turtlebot via an MQTT connection. When the Turtlebot has received the can in the Turtlebot Load-sensing Holder Subsystem, the user keys in the designated table number in the mobile application and sends the data over to the Turtlebot. The Turtlebot receives this information and moves to the corresponding table, in accordance with the Turtlebot Navigation Subsystem.



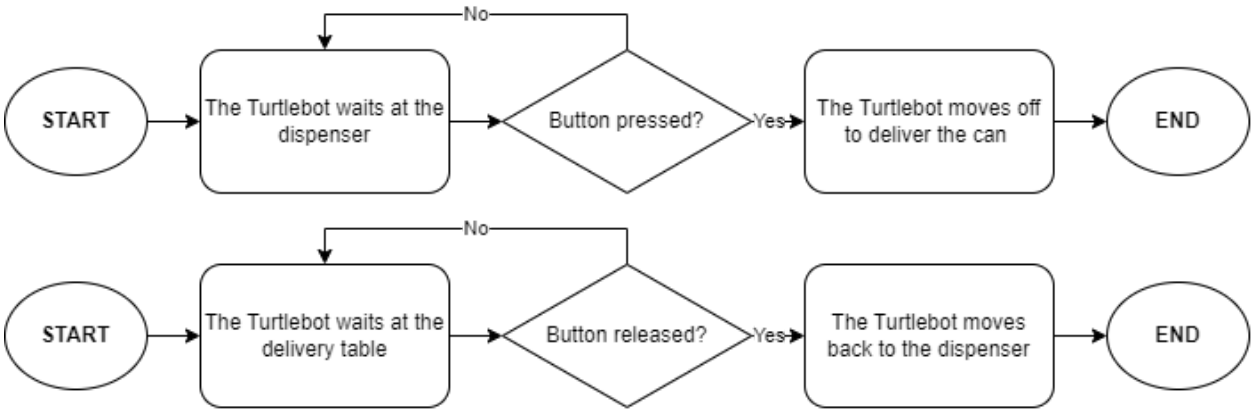
4.3 Turtlebot Load-sensing Holder Subsystem

The can holder on the Turtlebot has a basic design, and it complements the size and shape of the Turtlebot as well as the can. Being no wider than the Turtlebot and just slightly wider than the length of the can, the can holder is designed to allow for slight variations in the position of the Turtlebot relative to the dispenser, in order for the can to fall accurately into the can holder.

To detect the presence of the can in the can holder, a button is placed at the bottom of the can holder, which will be pressed due to the force exerted by the can's weight on the button as the can falls from the Dispenser. Attached to the button is a light, flat metal plate which has a larger surface area than the button, making it easier to be pressed. To cushion the impact of the falling

can onto the can holder and the button, the interior of the can holder is layed with soft plastic foam.

Once the Turtlebot is parked in the dispenser, it waits for a can to be received. In the process, the button is constantly ready to be pressed. When the button is not pressed, this means that the can has not been received by the Turtlebot. The Turtlebot simply waits until there is a change in the button state. Upon receiving the can, the button is pressed, which sends a signal over to RPi to trigger the autonomous navigation in the Turtlebot Navigation Subsystem.

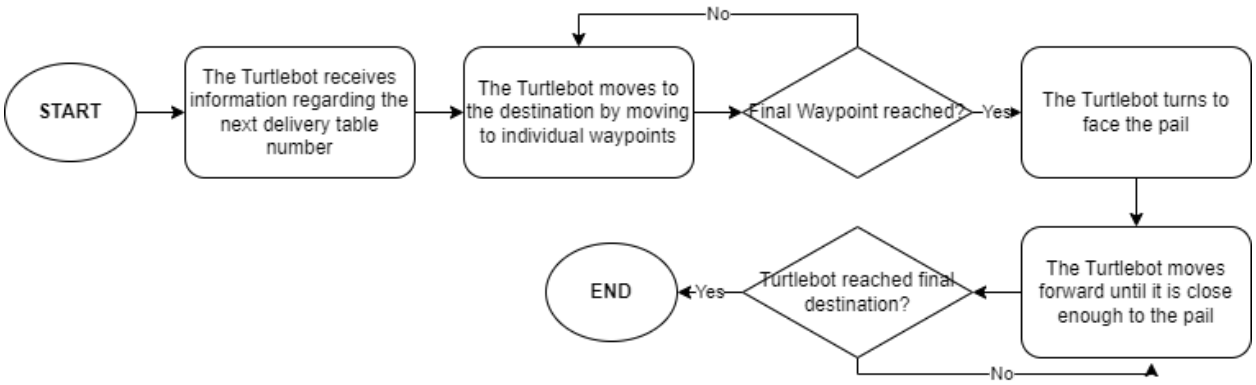


4.4 Turtlebot Navigation Subsystem

Given the dynamic position of Table 6 and the potential for sensor noise, we considered implementing a waypoint following algorithm to help the Turtlebot navigate around the arena.

Prior to actual run, the Turtlebot is able to register pre-set waypoints. In this process, the Turtlebot is tele-operated around the entire map, and the user is able to mark the coordinates of interest. These settings are stored into a file, which can be accessed by the Turtlebot in future programs.

During navigation, the Turtlebot is set to move from one waypoint to another, starting from the dispensing area all the way to each delivery table. The job of the developer is then made simple: to get the Turtlebot from one location to another, simply modify the program to include the relevant waypoint number and in the correct order for the Turtlebot to travel to. The navigation algorithm is automatically started when the Turtlebot receives the can as well as information regarding the table number. After the can has been delivered, the Turtlebot automatically navigates back to the dispenser using the same algorithm.


START
The Turtlebot receives information regarding the next delivery table number
The Turtlebot moves to the destination by moving to individual waypoints
No
Final Waypoint reached?
Yes
The Turtlebot turns to face the pail
The Turtlebot moves forward until it is close enough to the pail
Turtlebot reached final destination?
No
Yes
END

5. BOGAT

5.1 Choice of Dispenser Storage Subsystem

	Dispenser Storage Subsystem	
Mechanism	Ultrasonic Sensor and Servo Motors	Reverse Mechanism of Pedal Trash Can
Pros	A more familiar concept	Fully Mechanical
Cons	More electrical components to be installed	Turtlebot need to be programmed to push the pedal

We chose to implement the Dispenser Storage Subsystem using an Arduino Uno together with an ultrasonic sensor and 2 servo motors.

5.2 Choice of Dispatching Subsystem

	Dispatching Subsystem		
Mechanism	Bluetooth module	Physical buttons	Mobile application
Pros	Low cost Detection range is up to 10 meters which is enough for our application	Low cost, tactile	All-in-one easy-to-use interface
Cons	Bluetooth connection may be lost during navigation	Still requires a means to track that the input table number has been received by OpenCR	-

We chose to implement the Dispatching Subsystem using a mobile application which is able to send the orders to the Turtlebot.

5.3 Choice of Turtlebot Load-Sensing Subsystem

	Turtlebot Load-sensing Subsystem		
Mechanism	Load Sensors	LDR Sensors	Button
Pros	High accuracy Simple Mechanism	Simple Component Cheap	Low cost
Cons	May requires an additional amplifier	Hard to ensure that the environment is free from other light sources	May not have a high accuracy depending on orientation of can

We chose to implement the Turtlebot Load-Sensing Subsystem using a button which is easily implementable and can connect directly to the RPi. We can mitigate the limitation of the inaccuracies in the button press by adding the light plate above it to increase the pressable surface area.

5.4 Choice of Turtlebot Navigation Subsystem

	Turtlebot Navigation Subsystem			
Hardware	IR Sensor	Ultrasonic Sensor	Camera Module	LiDAR
Pros	Can detect soft objects	Can detect objects from more than 1m away	Low cost Easy to set-up	Can perform SLAM algorithms accurately, can use to detect distances
Cons	High interference	May not be able to detect soft objects easily	Lack of IR illumination	-

We chose to implement the Turtlebot Navigation Subsystem using the LiDAR which can perform basic SLAM algorithms and is still effective overall in measuring distances and other relevant information.

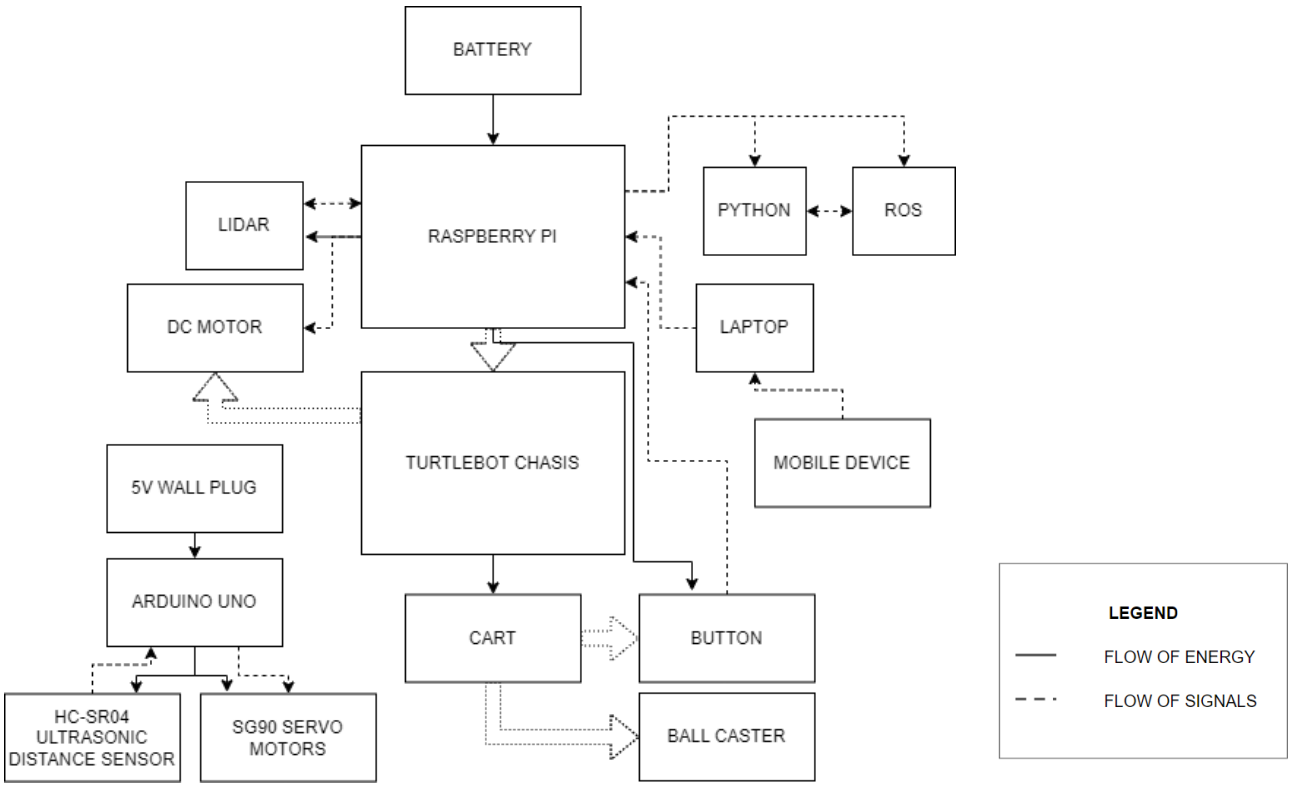
	Turtlebot Navigation Subsystem					
Software	SLAM	MCL	RH-A*	Kino-RRT*	Simplified A*	Waypoint Algorithm
Pros	Works well in dynamic environments	Less computationally expensive	Less computationally expensive for small map	Generally creates smoother paths	Much easier to implement	Implementation using simple trigonometry
Cons	Computationally expensive	Struggles with highly dynamic environments	Result in unsafe or non-optimal paths (for low update frequency of LiDAR)	Very computationally expensive	May give less precise paths	Limited accuracy

We chose to implement the Turtlebot Navigation Subsystem using the Waypoint Algorithm which is the easiest to implement among all the other algorithms. We will be using SLAM to carry out localisation for the Turtlebot and the mapping of the terrain.

6. Preliminary Design

Preliminary design is the second stage of the design process, where the focus is on developing and refining the concepts identified in the concept design stage. In this stage, we provide an overall view of the flow of energy and signals and categorise the subsystems into their respective phases and processes.

6.1 Functional Block Diagram



In this setup, the RPi and the laptop are the main drivers of all subsystems and processes. The functional block diagram for the overall system is as shown above.

6.2 Phases

Phase	Selection	Loading	Delivery	Collection	Return
Systems	Dispatching System, Turtlebot Navigation System	Dispenser Storage System, Turtlebot Load Sensing Holder System	Turtlebot Navigation System	Turtlebot Load Sensing Holder System, Turtlebot Navigation System	Turtlebot Navigation System
Objectives satisfied	D2	D1, D3, D4	T2,T3	T1, T3	T2, T4
Hardware	Android mobile device (works with MIT App Inventor)	Servo motors	LiDAR	Can holder, Button	LiDAR
Software	Mobile App which sends data over MQTT	-	SLAM, waypoint algorithm	Button script on RPi	SLAM, waypoint algorithm

6.2.1 Phase 1 (Selection Phase)

1. Objective of this phase

To allow the user to select the delivery table using a GUI on a phone/tablet.

2. Description of Operation

The device (phone/tablet) will have an interactive interface and the data will be transmitted through MQTT connection to the laptop, which will then be relayed to the RPi.

6.2.2 Phase 2 (Loading Phase)

1. Objective of this phase

The Turtlebot is to standby at a fixed position to received the can after it enters the dispenser.

2. Description of Operation

The Turtlebot waits in the dispenser for the can to be received. The phase ends when the Turtlebot detects that the can has been received.

6.2.3 Phase 3 (Delivery Phase)

1. Objective of this phase

To move towards the delivery table autonomously. The Turtlebot will stop once it reaches the correct coordinates.

2. Description of Operation
The Turtlebot relies on mapping and path planning algorithms to travel to the designated table.

6.2.4 Phase 4 (Collection Phase)
1. Objective of this phase
The objective of this phase is for the customer to collect the delivered can drink once the robot has arrived at the correct table, and after the can drink is collected, the robot will begin its navigation back to the Dispenser.

2. Description of Operation
Upon reaching and stopping at the correct table, the Turtlebot will wait until the can drink has been collected from the can holder. Once the can drink is collected, a Python script of the button on the can holder will send a signal to the RPi that the can has been removed to resume navigation back home.

6.2.5 Phase 5 (Return Phase)
1. Objective of this phase
Similar to Phase 3, the Turtlebot will travel autonomously back to the Dispenser and stop once the destination has been reached.

2. Description of Operation
The Turtlebot uses path planning algorithms to travel to the designated table and avoid obstacles.

7. Prototyping & Testing

7.1 Navigation Algorithm

When testing out the wall-following code on the Turtlebot, the navigation worked and the robot was able to map out and navigate to every part of the entire maze. However, it took a considerable amount of time, which given the new grading criteria that the timing of the whole navigation run was a huge deterministic factor, our team decided to opt for a simpler yet more effective algorithm which is to use the waypoints algorithm. Using waypoints algorithm reduced the timing required to navigate in the maze significantly and also allowed us to set the waypoint coordinates of each designated table beforehand, which potentially removed the need for us to even map out the map of the maze before navigation.

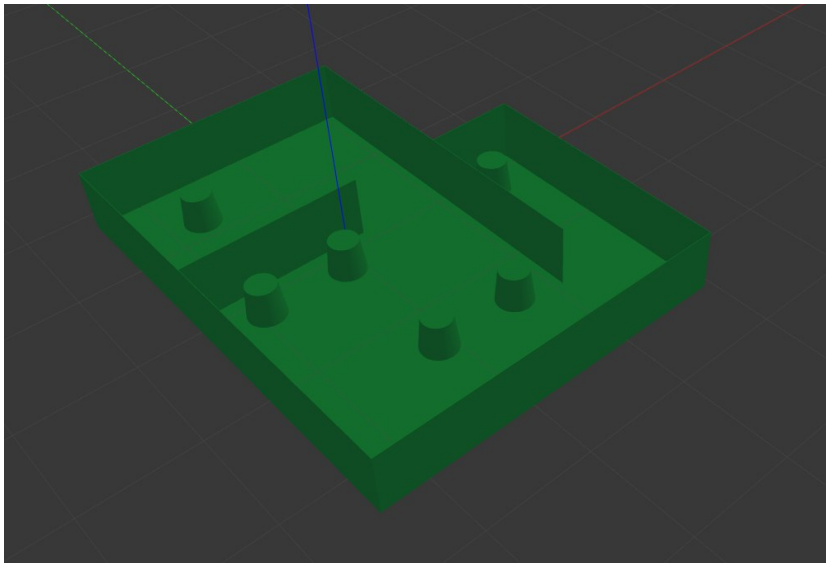
For our first iteration of the waypoints algorithm, we made use of the Odometry data of the Turtlebot to navigate around the maze. First, we obtain the odometry data from the Turtlebot's *"nav_msgs/Odometry"* message type and subscribe to the *"/odom"* topic, which provides the robot's odometry information. Then we define a set of waypoints that the Turtlebot needs to navigate to as a list of coordinates in the map frame. We also define a threshold value to determine when the Turtlebot has reached a waypoint. Under the 'main loop', the navigation node will publish the linear velocity data using the *"geometry_msgs/Twist"* message type to the *"/cmd_vel"* topic, which is subscribed by the robot's motion controller. However, we realised when we used the *'Odometry'* data, it was not very accurate due to the drifting of the Turtlebot in the SLAM environment in Rviz. As a result, we had to do a lot of calibration and error adjustment and this introduced new bugs to the docking aspect of the Turtlebot.

Hence, we reiterated and came up with the second iteration of our waypoints algorithm, which listens for transform information between two coordinate frames, *"/map"* and *"/base_link"*. It uses the *"tf2_ros"* library to get the current transform between these frames and publishes it to a topic *"/map2base"* at a fixed frequency. The transform information is packaged in a *"Pose"* message containing the position and orientation of the *"/base_link"* frame relative to the *"/map"* frame. The transform data is obtained through a TransformListener object which listens to a TransformBroadcaster object that is publishing the transform between the frames. The node sets up a timer that triggers the callback function at a fixed rate, and within the callback function, the current transform data is obtained using the *"tf2_ros"* buffer. If the transform is available, it is packaged into a *"Pose"* message and published to the *"/map2base"* topic. This way allows us to track the position and orientation of a robot within a known map frame, and does not lead to any form of drifting inside Rviz and hence there is also less error adjustment in the pose of the Turtlebot needed to be made.

Another point of consideration was the drive-through element of our Dispenser. We decided to not follow through with a drive-through as we found that it was actually more cumbersome to integrate with the robot's navigation algorithm and it did not help us in reducing the time taken to collect the can from the Dispenser. Therefore, we opted to modify our Turtlebot to dock inside the Dispenser inside and we had to consider the proper docking pose of the Turtlebot inside the

Dispenser as we had to ensure that it is able to dock in the correct position properly between the walls of the Dispenser. Additionally, we also had to ensure that there was sufficient space for the Turtlebot to turn inside the Dispenser without colliding into the walls of the Dispenser which would either cause it to be stuck, or would mess up the map. As a result, the waypoints coordinates set to dock the Turtlebot inside the dispenser had to be carefully calibrated and adjusted through trial and error to ensure the Turtlebot is in proper orientation before entering the Dispenser and the Turtlebot will not hit the walls of the Dispenser when leaving.

7.2 Simulations in Gazebo



Simulations of the maze setup were created in Gazebo. This setup was mainly used to program, test and debug the different navigation algorithms before putting them to test in the actual setup. In this manner, we could save the time and resources required to test and run new algorithms.

The simulation environment includes a cadded model of the maze with the dimensions as per given. A model of the Turtlebot without the can holder was placed in the maze early on, as the simulation environment was purely for testing the navigation algorithm and not other subsystems.

7.3 Can Holder

Initially, the can holder was designed to be a funnel-like container which allowed a larger surface area for the can to be dropped on, reducing the possibility of the can dropping outside the can holder. However, the preliminary design showed that 3D-printing such a container would have incurred a huge amount of materials and time which we did not have the luxury of and hence we switched over to using acrylic to build our can holder. The rectangular opening of the can holder also meant that we needed to be more precise with the dropping the can in order to ensure that the success rate of the can falling into the can holder is maximised as much as possible. We made use of cheap materials such as bubble wraps and plastic foams as a layer of cushion to

reduce the impact of the fall in order to reduce the damage done to both the can holder and the can drink.

Another point of consideration was the data transmission from the can holder to inform the Turtlebot whether the can is in the can holder or not. Our first idea was to use a load sensor, however, we thought of a much more simpler way which was to connect a mechanical button at the base of the can holder directly to the RaspberryPi of the Turtlebot. After multiple iterations together with the padding of the foam around it, we managed to adjust the correct layers and type of foam needed such that when the can is dropped into the can holder, the button will be pressed and the *Boolean* type message will be published to the *'button_pressed'* topic which is subscribed by the RPi for navigation purposes in the main code. Likewise, the button will be released when the can is removed from the can holder resulting in a change in the *Boolean* message.

7.4 Mobile Application

Our mobile application is created using the MIT App Inventor. We chose this framework over the other standard frameworks for mobile development, such as React Native and Flutter, as the MIT App Inventor offered a more user-friendly approach to coding the mobile application. With block-based coding and seamless integration with MQTT connections, we saved time and effort in the creation of a mobile application which was readily usable and deployable.

Our original idea was to establish communication between the mobile application and the RPi using Bluetooth. After setting up *bluetoothctl*, the command-line tool for Bluetooth connections in Linux, and setting up the RPi as the "master" and the mobile device as the *"slave"*, we realised that we were unable to establish a stable connection between both devices. This issue also occurred when the RPi became the *"slave"* and the mobile device the *"master"*. We quickly realised that this was a limitation of the Bluetooth service that the MIT App Inventor provides, as seen from several complaints from users facing similar issues with the Bluetooth connection.

We needed to find another alternative for the mobile application to send a single integer of data over to the Turtlebot. We found that the simplest way of doing so was to send this information directly from the mobile application to the RPi via an MQTT broker. However after much trial and error, we discovered that this approach limited our debugging capabilities for the system. We were not able to identify if the correct number was sent to the RPi, not to mention whether any integer was sent to the RPi in the first place. As a result, we decided that a more robust approach to communication would be to establish an MQTT connection between the mobile device and the laptop. Then, the laptop will send this integer over ROS2, by creating a node which publishes this data. The RPi then has a node which subscribes to this data, thereby being able to process the information regarding the table number.

7.5 Dispenser

When the Turtlebot enters the entrance of the dispenser, the HC-SR04 ultrasonic sensor detects its presence and the servo motor holding the can will be activated such that the servo horns will

shift out of the way for the can to roll and drop down from the dispenser due to gravity. To rapidly prototype and test this out, Arduino Uno, breadboard, two SG90 servo motors, jumper wires and the HC-SR04 ultrasonic sensor are used. After wiring the circuit accurately, the distance required for the ultrasonic sensor to actuate the servo motors by a specified value of degree is then iteratively adjusted in the code in the Arduino IDE. We then used masking tape to secure the servo motors, breadboard the arduino temporarily onto the assembled acrylic dispenser structure and adjusted their positions due to the plug and play nature that the masking tape accords.

We conducted tests to check the appropriate width for the dispenser. To do this, we tele-operated the Turtlebot near walls to determine the rough minimum distance to which the LiDAR stopped mapping an accurate map in RViz (via the command *'rslam'*). We confirmed the distance to use to be in relation to the minimum scanning range of the LiDAR (~12cm) and were able to validate our prior calculations for the eventual width of the dispenser to use.

7.6 Testing of all components

In order to test that all the components are working, we wrote a script to automate the whole checking process. The script can be run by first ssh into RPi using the command *'sshrp'* and, followed by starting up all hardware using the command *'rosbu'* and *'button_pub'* in the terminal on the RPi. After which, we enter *'factory_test'* on the terminal on our laptop to start the Factory Acceptance Test.

Follow the given instructions on the console

Verify dynamixels are working	On typing ***'Y'*** followed by the return key, the turtlebot should move approx 10cm forward, 10cm back, turn 90 deg left, then turn 90 deg right.
Verify LiDAR is working	On typing ***'Y'*** followed by the return key. The user should observe ***'laserscan data is VALID, lidar is functional'*** printed in the console if the lidar is working properly.
Verify button on can holder is working	On seeing ***'press the button to test…'*** in the console, the user should press the key cap on the turtlebot, he has 20 seconds to complete this. The user should then observe ***'button is functional…'*** printed in the console if the button is functioning properly.
Verify ultrasonic sensor is working	On seeing ***'testing ultrasonic sensor…'*** in the console, the user should flash his hand across the ultrasonic sensor, he has 20 seconds to complete this. The user should then observe ***'ultrasonic sensor is functional…'*** printed in the console if the ultrasonic sensor is functioning properly.
Verify servo motors are working	On typing ***'Y'*** followed by the return key, the 2 servo motors should actuate 90 degrees forward, followed by 5 seconds delay, and 90 degrees backwards.

8. Critical Design

Critical design is the last design stage whereby the design is finalised, and all aspects of the system are thoroughly detailed. During this stage, we present detailed design specifications and manufacturing instructions, select and test materials, and develop a comprehensive test plan. The output of this stage is typically a final design review, which is a comprehensive assessment of the design and its readiness for production.

8.1 Key Specifications

List	Specifications	Note
Size (mm)	280 x 175 x 195 (L x W x H)	Full assembly (Turtlebot + Can holder)
Weight (g)	1041.05	
Drive Actuator	2x XL430-W250 DYNAMIXEL	
Maximum Translational Speed	0.22 m/s	
Maximum Rotational Speed	2.84 rad/s (162.72 deg/s)	
Sensors	LiDAR 1x 360° LDS-01 Odometry 1x ICM-20648 (On OpenCR 1.0)	
Buttons & Switches	1x Push button	In addition to Turtlebot pre-existing buttons
Battery Capacity	LiPo Battery 11.1V 1,800mAh	
Expected Operating Time	0.663 hours	
Communication interface	MQTT, GPIO	MQTT MIT App Inventor GPIO Push Button

8.2 System Finances

A Turtlebot set was provided to create the system, as well as miscellaneous fasteners and electronic components. A budget of up to $100 was provided for supplementary parts. The breakdown of parts and their costs is as indicated below in the bill of materials.

Bill of Materials

No.	Part	QTY	Unit Cost	Total Cost
Turtlebot (Mechanical)				
1	Waffle Plate	8	Provided	Provided
2	Wheel	2		
3	Tire	2		
4	M2.5 Nuts (0.45P)	20		
5	M3 Nuts	16		
6	Spacer	4		
7	Rivet (Short)	14		
8	Rivet (Long)	2		
9	Plate Support M3x35	4		
10	Plate Support M3x45	10		
11	Adaptor Plate	1		
12	Adaptor Bracket	5		
13	PCB Support	12		
14	PH_M2x4mm K	8		
15	PH_M2x6mm K	4		
16	PH_M2.5x8mm K	16		
17	PH_T 2.6x12mm K	16		
18	PH_M 2.5x16mm K	4		
19	PH_M 3x8mm K	44		
20	Rear Ball Caster (w/ Ball)	2		
Turtlebot (Electrical)				
21	Li-Po battery	1	Provided	Provided
22	Li-Po battery charger	1		
23	USB Cable	2		
24	Dynamixel to OpenCR Cable	2		
25	Raspberry Pi 3 Power Cable	1		

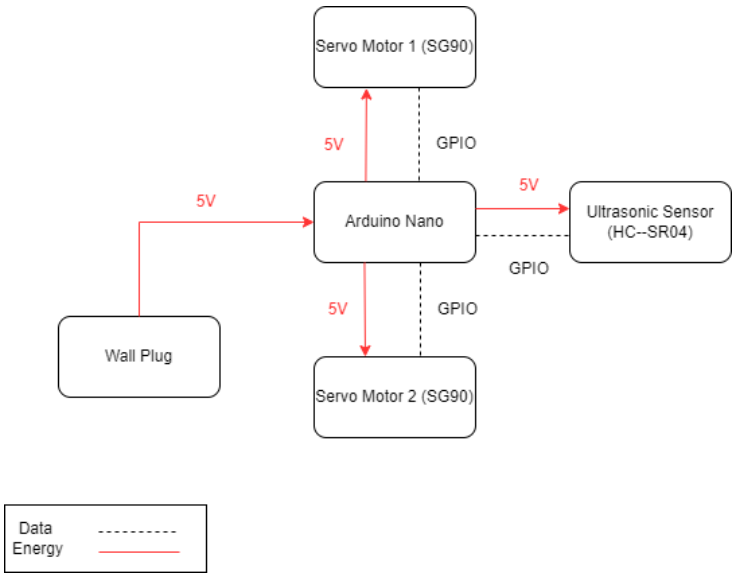
26	Li-Po Battery Extension Cable	1		
27	SMPS	1		
28	AC-Cord	1		
29	Dynamixel XL 430	2		
Turtlebot (for Software)				
30	OpenCR 1.0	1	Provided	Provided
31	Raspberry Pi 3B	1		
32	360 Laser Distance Sensor LDS-01	1		
33	USB2LDS	1		
34	Push Button	1		
Can Holder (Mechanical)				
	Laser Cut Acrylic Plate			
1	Bottom Plate	1	$6 service fee	
2	Side Plate	4		
	Miscellaneous			
3	Breadboard	1	Provided	
4	Plastic Foam	1		
5	Push Button	1		
	Turtlebot items/Bolts/Nuts/Spacers			
6	Waffle	4	Provided	
7	Ball Caster	1		
8	M3x35mm Spacers	4		
9	M3x8 Bolt	8		
10	M2.5x10 Bolt	2		
11	M2.5 Nut	2		
Dispenser (Mechanical)				
	Laser Cut Acrylic Plate			

1	Side Plate	2	$6 (double-counted)
2	1st Layer	1	
3	2nd Layer	1	
4	Walls	2	
	Bolts/Nuts/Brackets		
5	Aluminium Bracket	10	Provided
6	M3x10 Bolt	10	
7	M3 Nut	10	
Dispenser (Software)			
1	SG90 Servo Motor	2	Provided
2	Arduino Uno	1	
3	Ultrasonic Sensor	1	
4	Android Tablet	1	
Grand Total			$6.0

8.3 Power Budget

8.3.1 Dispenser

The following electronic architecture diagram shows the flow of data and energy within the electrical components in the dispenser.



The following power budget table shows how the total power consumption is derived based on the calculated power consumption from the individual electrical components.

Component	Voltage (V)	Current (A)	Quantity	Power (W)	Remarks
Ultrasonic Sensor (HC-SR04)	5	0.015	1	0.075	max current: 15mA
Servo motor (SG90)	5	0.22	2	1.1	stall current: 650mA
Arduino Uno	5	2	1	10	
			Total	11.175	

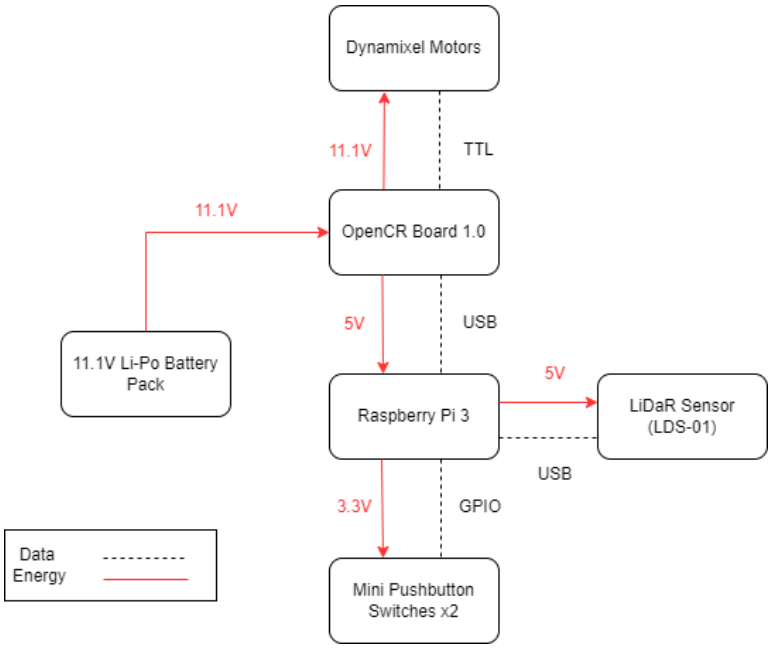
Based on the power budgeted above and the capacity of the typical 5V battery pack, we are able to provide an estimated duration in which the dispenser is able to operate, in the event that we use a 5V battery pack.

	Voltage (V)	**Capacity (Ah)**	**Power**	**Duration the battery pack can last (h)**
5V battery pack (5V battery has 750mAh)	5	0.75	11.175	0.663

In reality, we are connecting the Arduino Uno on the dispenser to the wall plug. In that case, there is no restriction to the duration that the dispenser is able to operate under those conditions.

8.3.2 Turtlebot

The following electronic architecture diagram shows the flow of data and energy within the electrical components in the Turtlebot.



The following power budgeting table shows how the total power consumption is derived based on the calculated power consumption from the individual electrical components.

Component	Voltage (V)	Current (A)	Quantity	Power (W)	Remarks
Mini Pushbutton Switch (Omron B3F-1000)	3.3	0.05	2	0.33	
Turtlebot3 Burger*	11.1	0.688	1	7.64	*Operating Power calculated in E2 = 7.64W
Turtlebot3 Can Holder (additional Power to drive this)	-	-	-	2.45	P = F x v where F = 11.11N (calculated earlier) and v = 0.22 m/s (max linear velocity of Turtlebot3 Burger)
			Total	10.42	

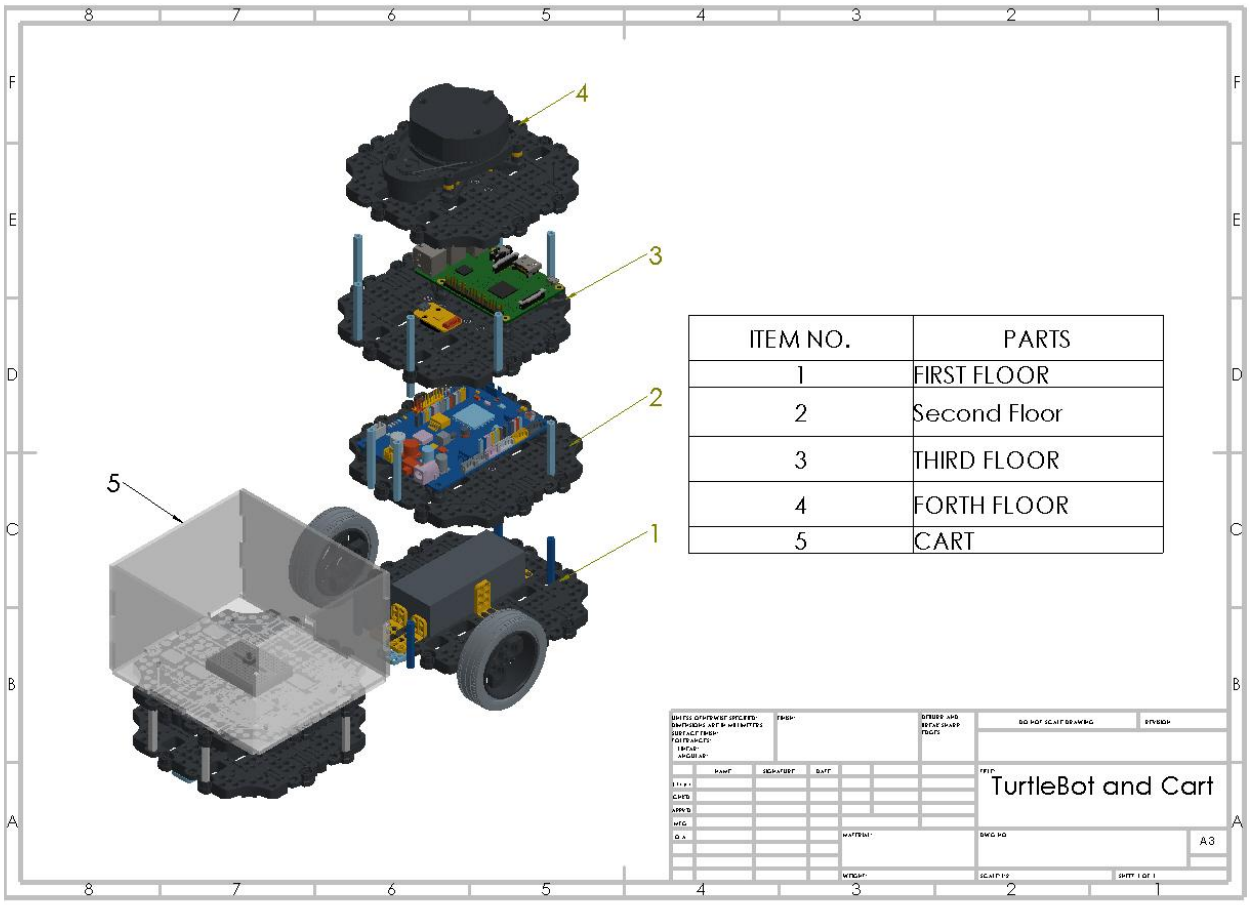
Based on the power budgeted above and the capacity of the 11.1V LiPo battery, we are able to provide an estimated duration in which the Turtlebot is able to operate, which is around 1.688 hours.

	Voltage (V)	Capacity (Ah)	Power (W)	Duration the battery pack can last (h)
Turtlebot Battery Pack	11.1	1.8	11.84	1.688

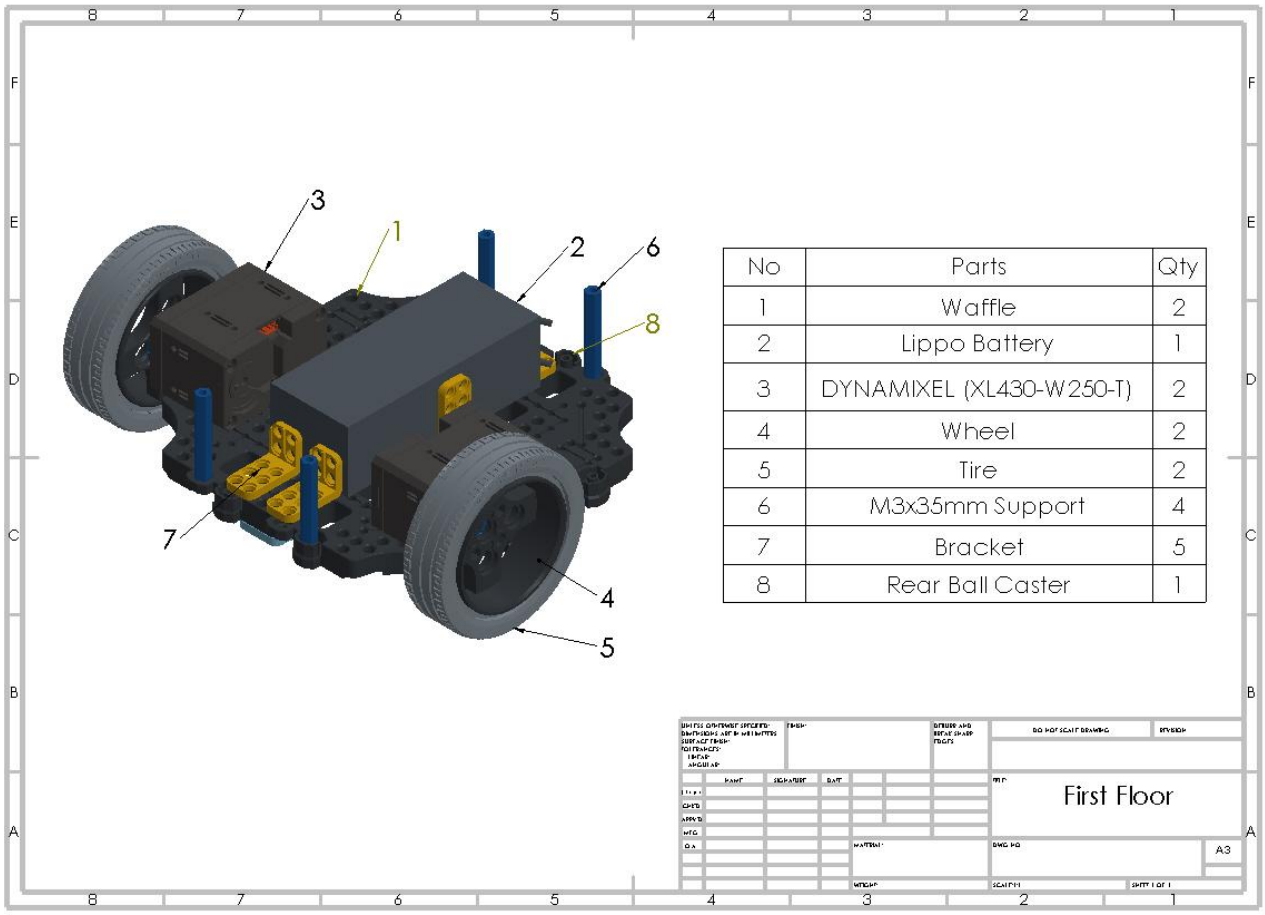
9. Assembly Instructions

9.1 Mechanical Assembly

9.1.1 Overview of the Assembly



9.1.2 Turtlebot 3 First Layer

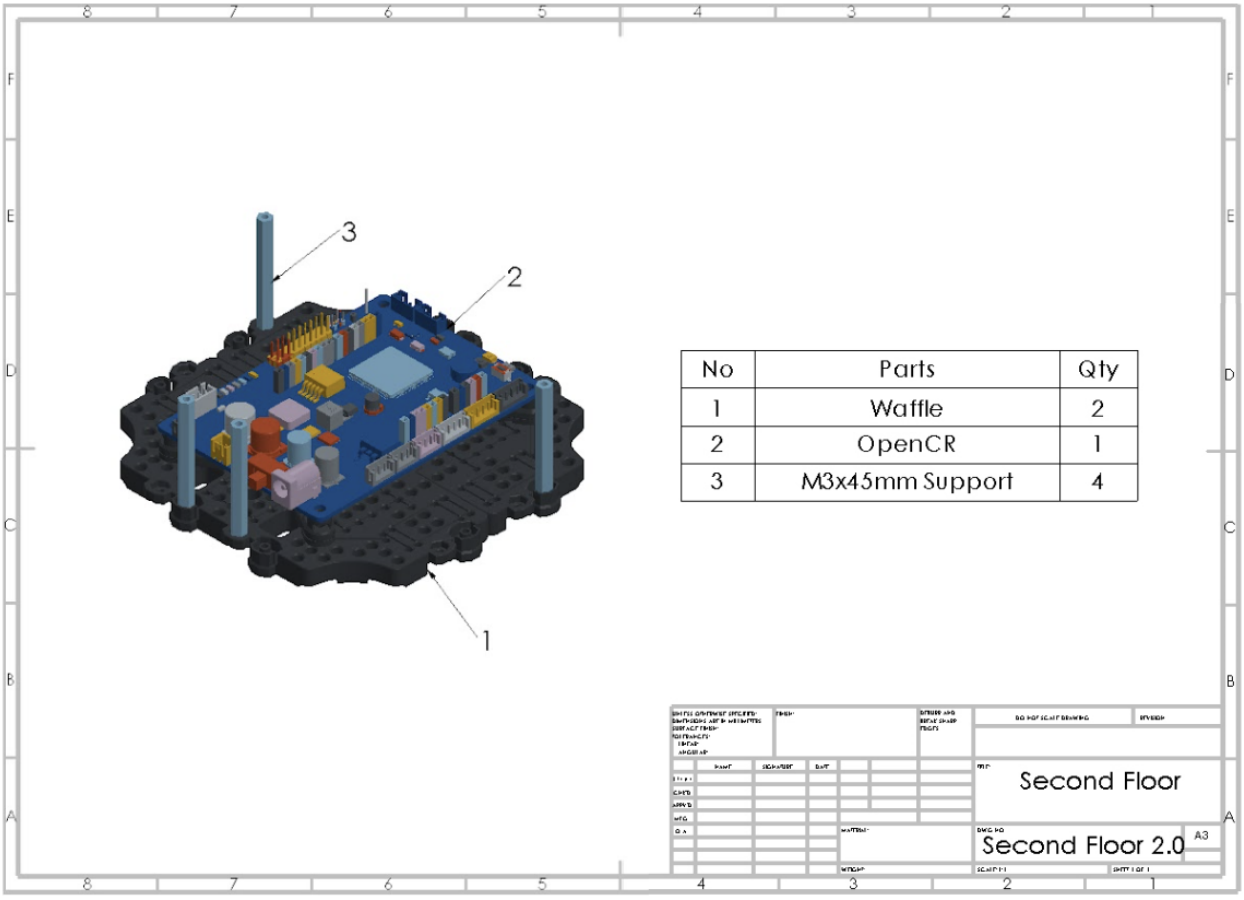


No	Parts	Qty
1	Waffle	2
2	Lippo Battery	1
3	DYNAMIXEL (XL430-W250-T)	2
4	Wheel	2
5	Tire	2
6	M3x35mm Support	4
7	Bracket	5
8	Rear Ball Caster	1

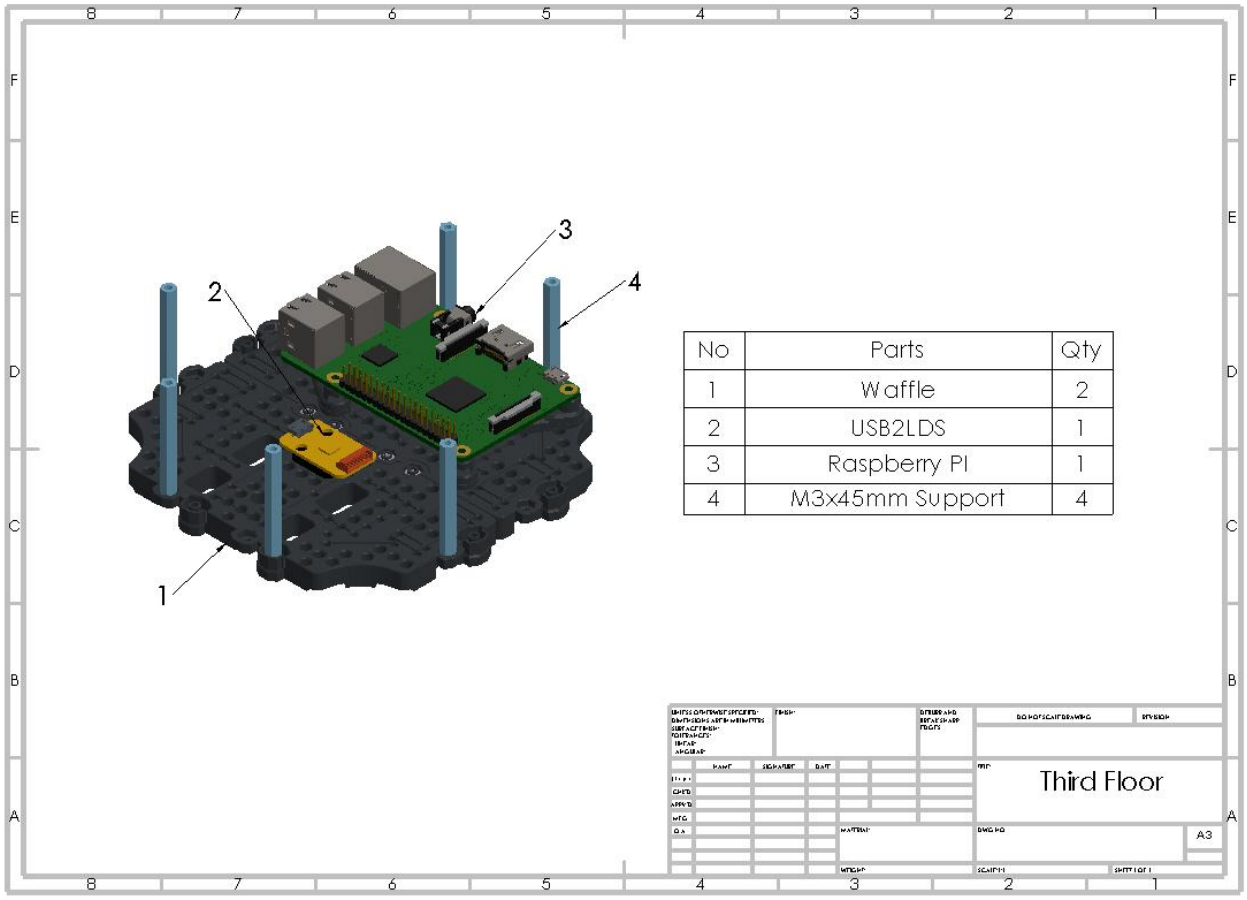
First Floor

A3

9.1.3 Turtlebot 3 Second Layer



9.1.4 Turtlebot 3 Third Layer

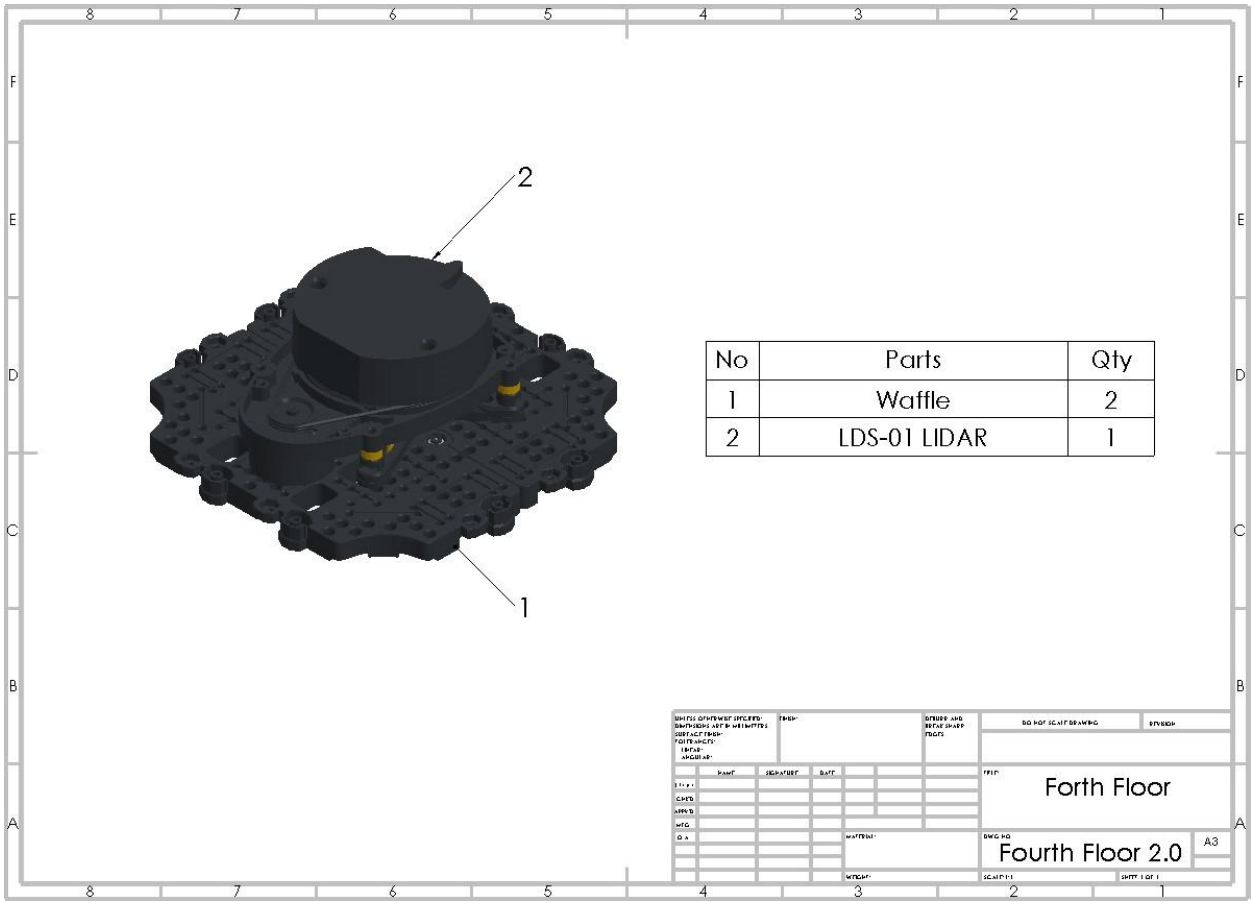


No	Parts	Qty
1	Waffle	2
2	USB2LDS	1
3	Raspberry PI	1
4	M3x45mm Support	4

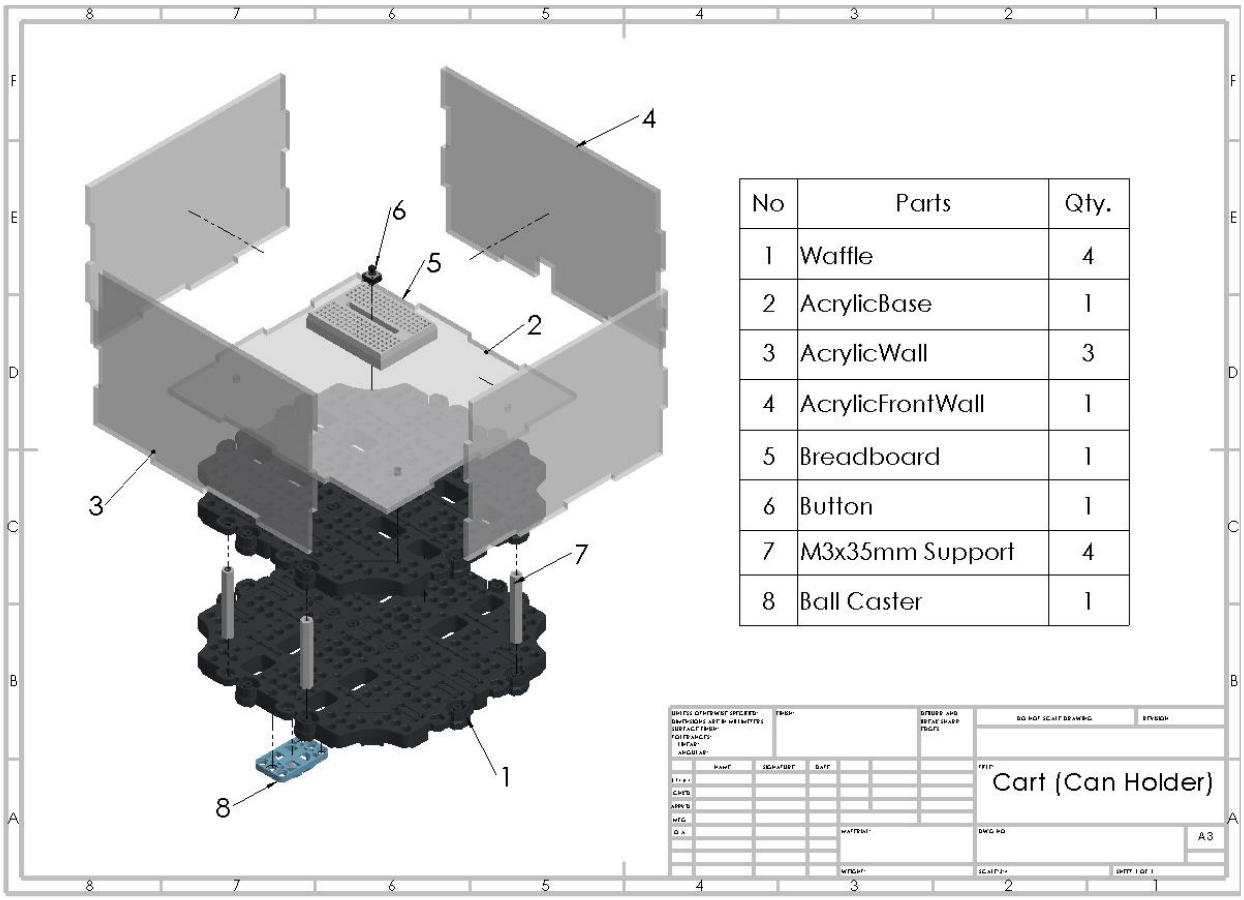
Third Floor

A3

9.1.5 Turtlebot 3 Fourth Layer



9.1.6 Cart (Can Holder)

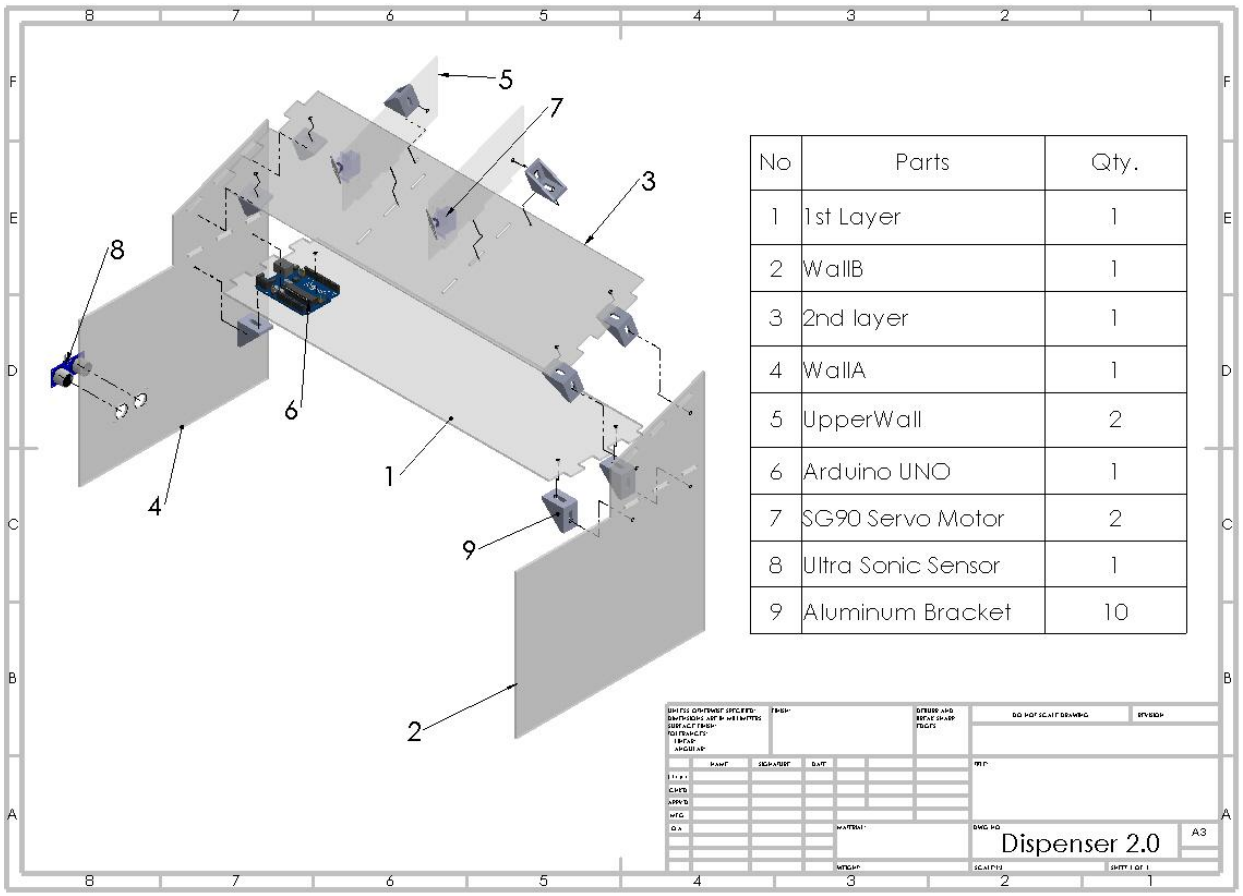


No	Parts	Qty.
1	Waffle	4
2	AcrylicBase	1
3	AcrylicWall	3
4	AcrylicFrontWall	1
5	Breadboard	1
6	Button	1
7	M3x35mm Support	4
8	Ball Caster	1

Cart (Can Holder)

A3

9.1.7 Dispenser



No	Parts	Qty.
1	1st Layer	1
2	WallB	1
3	2nd layer	1
4	WallA	1
5	UpperWall	2
6	Arduino UNO	1
7	SG90 Servo Motor	2
8	Ultra Sonic Sensor	1
9	Aluminum Bracket	10

9.2 Software Assembly

9.2.1 Setting up the devices

1. On the laptop, ensure that you have Ubuntu 20.04 and ROS 2 Foxy installed. Refer here on how to install the required software. Ensure that you are following the instructions under the "Foxy" tab.

2. Test that the ROS development environment is working by ensuring that a simple working publisher and subscriber can be created using the instructions here.

3. Using Ubuntu, follow the instructions here, burn the ROS 2 Foxy Image to the SD card onto the RPi on the Turtlebot3. Follow through the "Quick Start Guide" to get a working Turtlebot3.

4. Test that the ROS development environment is working by ensuring that a simple working publisher and subscriber can be created using the instructions here.

5. Once the ROS development environment is working on both the remote laptop and the RPi, run the publisher from the RPi and the subscriber on the laptop, and ensure that the subscriber on the laptop replicates what is being produced by the publisher. Swap the device that the publisher and subscriber are publishing from to ensure that two-way communication between the RPi and the remote laptop can be established.

6. Add the following lines to *.bashrc* of the remote laptop.

```
export TURTLEBOT3_MODEL=burger
alias rteleop='ros2 run turtlebot3_teleop teleop_keyboard'
alias rslam='ros2 launch turtlebot3_cartographer'
alias sshrp='ssh ubuntu@<IP_ADDRESS>’
```

7. Add the following lines to *.bashrc* of the RPi.

```
 export TURTLEBOT3_MODEL=burger
 alias rosbu='ros2 launch turtlebot3_bringup robot.launch.py
```

8. Install the Mosquitto MQTT broker on the Linux laptop by following the instructions here.
9. Install the Arduino IDE from the link here.

9.2.2 Installing the program on the remote laptop

1. Create a ROS 2 package on the remote laptop.

```
cd ~/colcon_ws/src
ros2 pkg create --build-type ament_python auto_nav
cd auto_nav/auto_nav
```

2. Move the file in the directory temporarily to the parent directory.

```
mv __init__.py ..
```

3. Clone the GitHub repository to the remote laptop. Make sure the period at the end is included.

```
git clone git@github.com:NicholasTanYY/r2auto_nav.git .
```

4. Move the file __init__.py back.

```
mv ../__init__.py .
```

5. Build the ‘auto_nav’ package on the laptop.

```
cd ~/colcon_ws && colcon build
```

6. Add the following line in the .bashrc file on the laptop.

```
alias factory_test='ros2 run auto_nav factory_test’
```

9.2.3 Installing the program on the RPi

1. Create a ROS 2 package on the RPi.

```
cd ~/turtlebot3_ws/src
ros2 pkg create --build-type ament_python auto_nav
cd auto_nav/auto_nav
```

2. Remove the file in the directory temporarily to the parent directory.

```
rm __init__.py
```

3. Clone the GitHub repository to the remote laptop. Make sure the period at the end is included.

```
git clone git@github.com:NicholasTanYY/r2auto_nav.git .
```

4. Build the ‘auto_nav’ package on the laptop.

```
cd ~/turtlebot3_ws && colcon build
```

5. Add the following line to the .bashrc file on the RPi.

```
alias button_pub='python3 turtlebot_ws/src/auto_nav/auto_nav/button.py’
```

9.2.4 Calibration of Parameters

The start of *'actual_navi.py'* on the laptop consists of constants that can be changed to fit the needs of the mission.

```
# calibration parameters
 rotate_change = 0.35 # for rotating the bot quickly
 slow_rotate_change = 0.10 # for rotating the bot slowly
 speed_change = 0.10 # forward speed of the bot
 box_thres = 0.15 # for the size of each waypoint coordinate box
 time_threshold = 1.28 * (box_thres/speed_change) # additional time taken
to travel within the box to make it to the centre of the box.
 dist_threshold = 0.18 # stopping distance of the bot to the table
```

The breakdown of the parameters are shown in the table below.

On the laptop	
Parameter	**Description**
`rotate_change`	Speed of the bot while it is rotating quickly. Positive value indicates rotation to the left, negative value indicates rotation to the right
`slow_rotate_change`	Speed of the bot while it is rotating slowly. Positive value indicates rotation to the left, negative value indicates rotation to the right
`speed_change`	Forward speed of the bot
`box_thres`	Dimensions of the waypoint tolerance box. e.g if the value is set as 0.15, the box will have a dimension of 0.30m x 0.30m. If the value is set as 0.25, the box will have a dimension of 0.50m x 0.50m. Generally, the larger the value of box_thres, the more tolerance there is for the turtlebot to stop away from each designated waypoint.
`time_threshold`	The additional time taken to travel within the box in order to make it to the centre of the box (as close as the actual waypoint coordinates as possible). This requires careful calibration depending on the value of `box_thres`
`dist_threshold`	Minimum allowable distance between the LDS-01 sensor (turtlebot) and the table/object

To change the parameters on the laptop, the following steps are as follows:

`slow_rotate_change` / `rotate_change` - If the rotation speed of the Turtlebot3 is too fast, decrease the values till the desired rotating speed is achieved.

`speed_change` - If the Turtlebot3 is moving too fast, decrease the value till the desired forward speed is reached. Take note that the default value of 0.22 is the maximum speed of the Turtlebot3 and increasing this value any further slows the Turtlebot3 down.

`distance_threshold` - The value can be increased or decreased to allow the Turtlebot3 to follow the wall from a larger or shorter distance respectively.

`box_threshold` - The value of the box tolerance for each waypoint coordinate can be adjusted. If box_threshold is too high, it may result in the Turtlebot stopping prematurely even before reaching the waypoint coordinates and this may require further calibration of `time_threshold`. Do also be aware that if the box_threshold is too low, there is a possibility that the Turtlebot may continue in its motion and not stop at the designated waypoint coordinates at all, resulting in a bad navigation run.

`time_threshold` - The constant value can be calibrated by increasing or decreasing depending on the Turtlebot3's final position with respect to the actual designated waypoint coordinates. If the Turtlebot stops before the actual coordinates, the value can be increased and if the Turtlebot stops after the actual coordinates, the value can be decreased. This value has to be calibrated every time the value of `box_threshold` is changed.

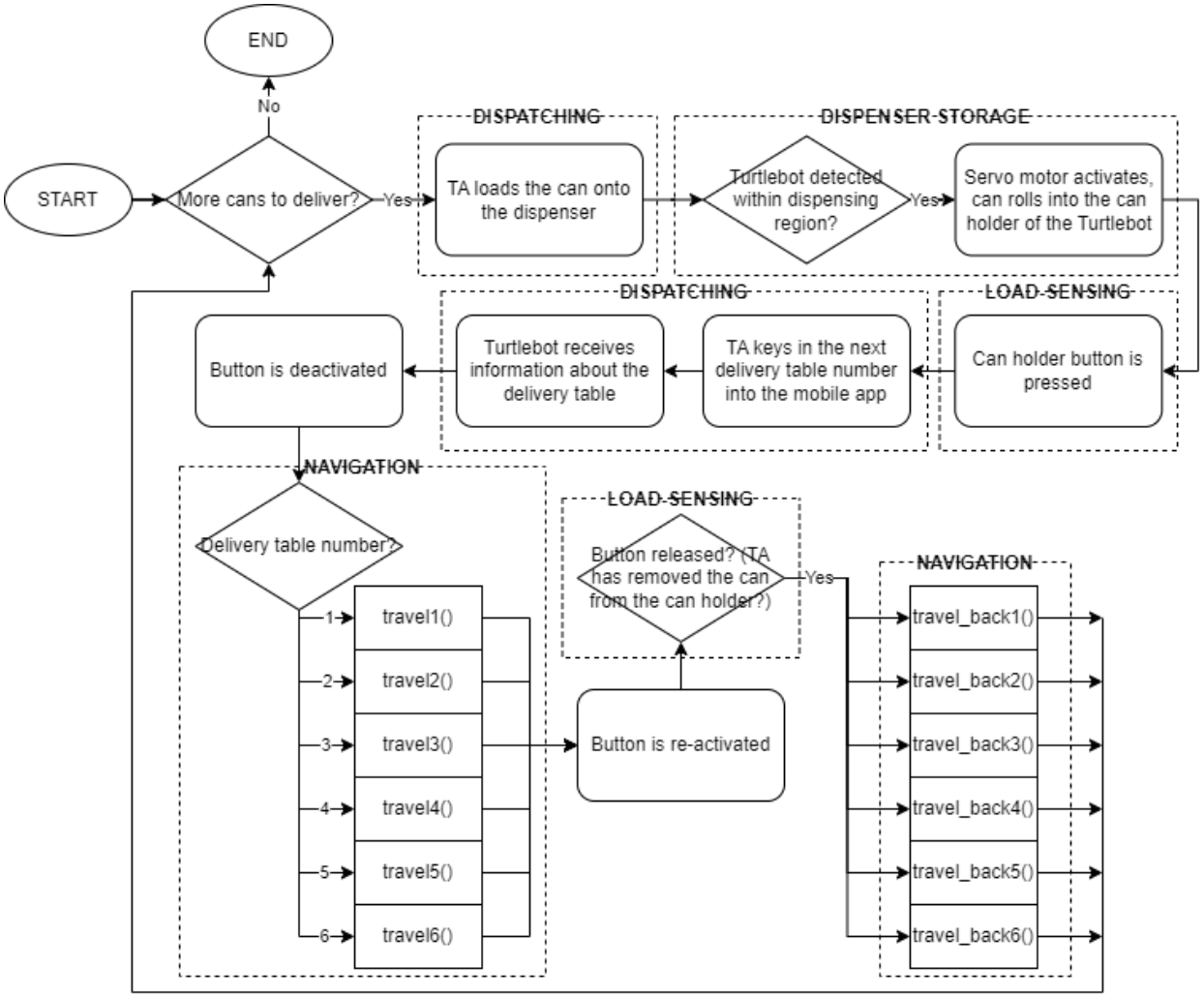
After changing the parameters, it is important to rebuild the package and source again before running the program, using the following command.

```
cd ~/colcon_ws && colcon build && source ~/.bashrc
source install/setup.bash
```

9.2.5 Setting up Mobile Application

1. On the website for MIT App Inventor, click "Create Apps!" to begin creating a mobile application.
2. Start a new project and import the .aia file from this webpage by clicking on Projects >> Import project from my computer.

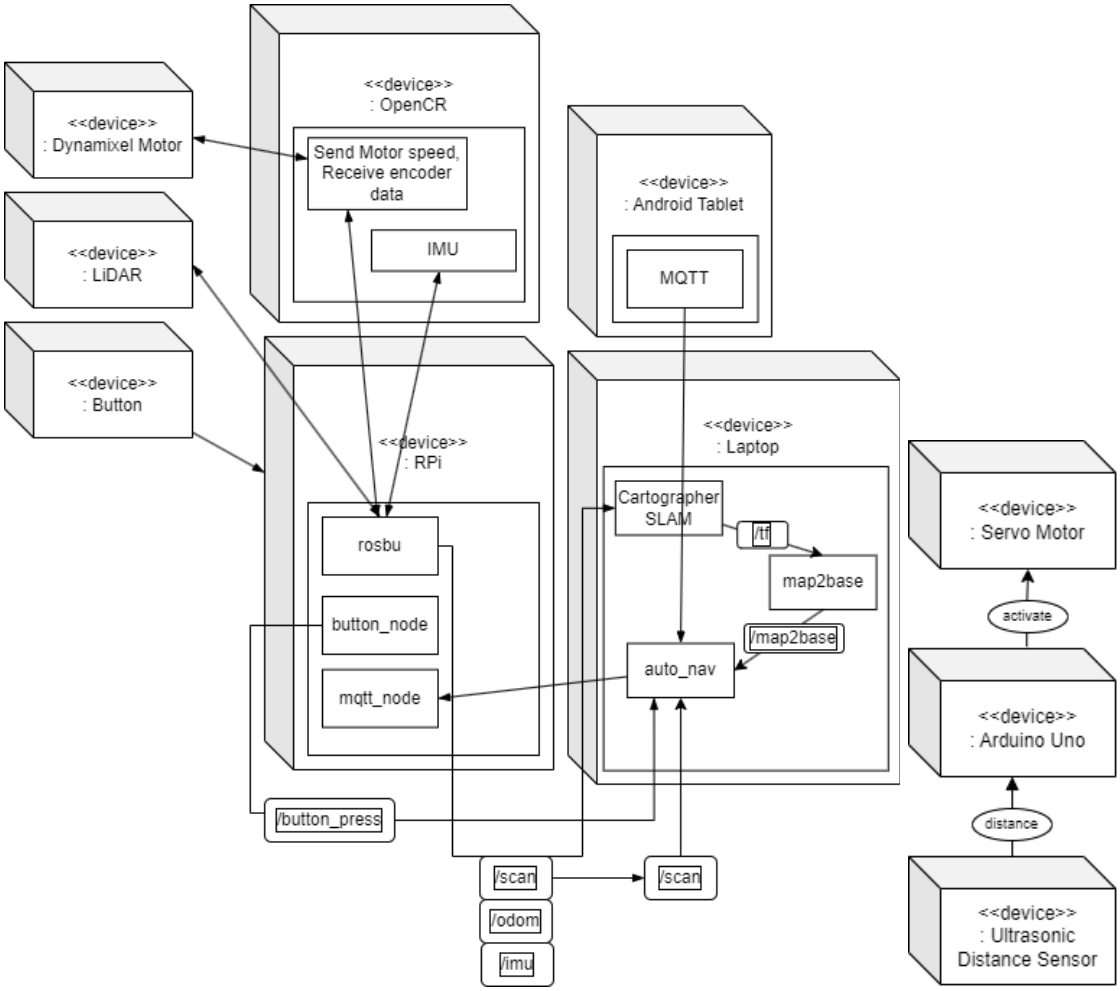
9.3 Overview of Algorithm



The algorithm is split into their subsystems for easy reference. It also follows the order of the phases for each iteration.

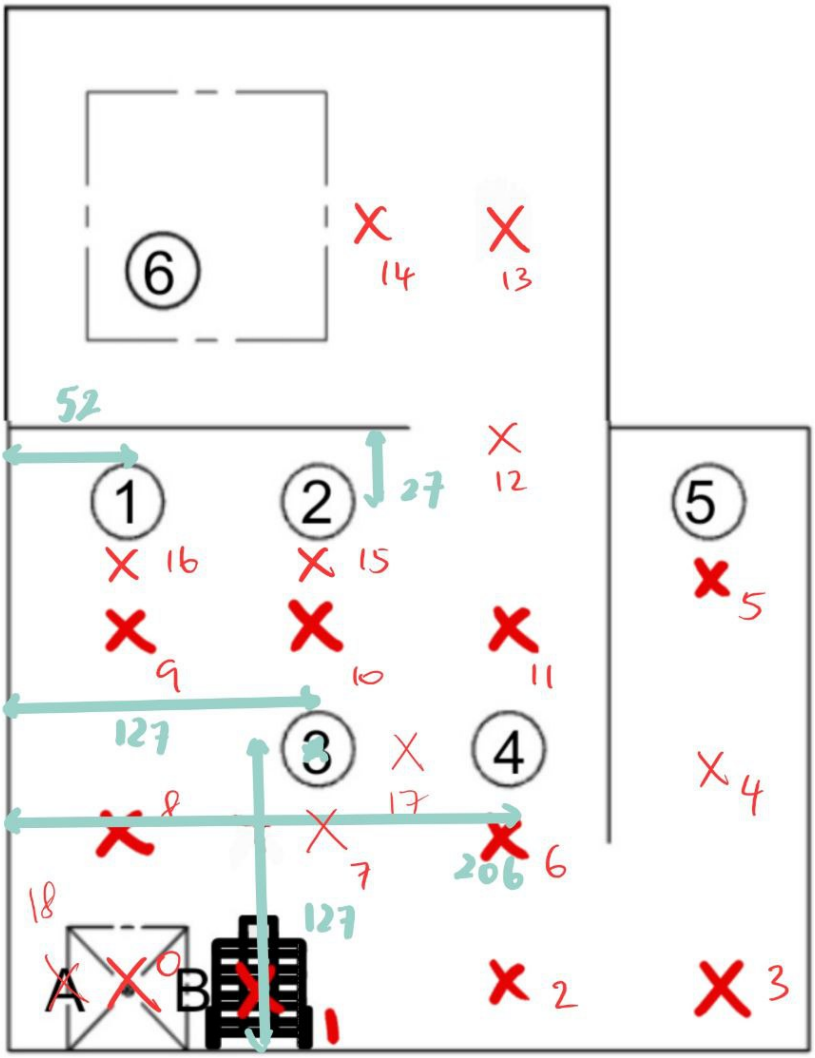
The algorithm begins when the Turtlebot is placed inside the dispenser, with the power to the Arduino Uno of the dispenser switched off. There are cans to deliver, so the TA loads the can into the dispenser, and switches on the power to the dispenser. When the can has successfully dropped into the can holder of the Turtlebot, the TA then keys in the delivery table number into the mobile application. This information is sent over to the Turtlebot, as input to the navigation algorithm on the Turtlebot. Based on the delivery table number, the Turtlebot follows a different set of waypoints to reach the destination. Once the Turtlebot reaches the destination table, it waits until the button is released, before following another set of waypoints back to the dispenser, based on the delivery table number. The whole process repeats if there are more cans to be delivered.

9.3.1 Communication between the devices



The above diagrams show the communication protocols between the different nodes in the system for both the Turtlebot as well as the dispenser.

9.3.2 Detailed breakdown of the Navigation Algorithm



The Navigation Algorithm uses waypoints for the Turtlebot to navigate to. These waypoints are marked out in the crosses with the corresponding label numbers for each of them.

9.3.2.1 Detailed breakdown of the Initialisation function of the Navigation Node

```
class Navigation(Node):

    def __init__(self):

        super().__init__('hardcoded_navi')
        self.publisher_ = self.create_publisher(
                Twist,
                'cmd_vel',
                10)

        self.map2base_subscriber = self.create_subscription(
            Pose,
            'map2base',
            self.map2base_callback,
            1)

        self.scan_subscriber = self.create_subscription(
            LaserScan,
            'scan',
            self.laser_callback,
            qos_profile=qos_profile_sensor_data)
```

The constructor function of the `Navigation` node initialises its attributes as well as its methods. As the node is called from the main code earlier, the constructor is automatically called which firstly creates a ROS publisher object using the *'create_publisher()'* method of the ROS node class. The publisher will publish messages of type *'Twist'* to the topic *'cmd_vel'*. A total of 4 subscriber objects are also created. Firstly, a ROS subscriber object is created using the *'create_subscription()'* method of the ROS node class which receives messages of type 'Pose' from the topic *'map2base'* and call the *'map2base_callback()'* callback function whenever a message is received. Secondly, another ROS subscriber object is created which receives messages of the type *'LaserScan'* from the topic *'scan'* and calls the *'laser_callback()'* callback function whenever a new message is received.

```
self.mqtt_subscription = self.create_subscription(
    String,
    'mqtt_data',
    self.mqtt_callback,
    1)
```

Next, the third ROS subscriber is created which will receive messages of type *'String'* from the topic *'mqtt_data'* and call the *'mqtt_callback()'* callback function when a new message is received.

```
self.buttonpressed = False

self.node = rclpy.create_node('button_subscriber')
self.node.create_subscription(Bool, 'button_pressed', self.button_callback, 10)
```

The final ROS subscriber is created to receive messages of type *'Bool'* from the topic *'button_pressed'* and it calls the *'button_callback()'* callback function. The buttonpressed attribute is set to be False initially as the can holder is empty.

```
self.cmd = Twist()

self.roll = 0.0
self.pitch = 0.0
self.yaw = 0.0
self.laser_range = np.zeros((2*front_angle+1,))
self.laser_range_6 = np.zeros((2*front_angle_6+1,))
self.laser_forward = 0.0
self.mapbase = Pose().position
self.x_coordinate = 0
self.y_coordinate = 0
self.storeX, self.storeY = 0, 0
self.XposNoAdjust = 0
self.YposNoAdjust = 0
self.mazelayout = []
self.visitedarray = np.zeros((300,300),int)
self.previousaction = []
self.resolution = 0.05
self.Xadjust = 0
self.Yadjust = 0
self.waypoint_arr = np.genfromtxt(f_path)
self.mqtt_val = 0
self.x_coordinate = 0.0
self.y_coordinate = 0.0
self.table6_turn_angle = 0
self.laser_angle_increment = 0
```

A number of attributes for the class are also initialised to store values. For example, *'self.cmd'* is an instance of the *'Twist'* message type used to store the linear and angular velocity of the turtlebot. *'self.laser_forward'* is used to store the double value of the distance measured by the forward facing LiDaR sensor while *'self.mapbase'* is a variable used to store the robot's pose at the current time in 3D space. *'self_mqtt_val'* is initialised to be 0 which is used to store the value received from the MQTT broker. The other maze-related attributes are initialised to be used for the mapping of the maze initially, but was subsequently not adopted as we changed our navigation algorithm from wall-following to waypoints which did not require us to map out the whole map to do our navigation.

9.3.2.2 Detailed breakdown of the Math functions used by Navigation Node

```
def calculate_yaw_and_distance(x1, y1, x2, y2, current_yaw):
    """
    Calculates the yaw the robot needs to turn to and the distance it needs to travel to move from point (x1, y1)
    to point (x2, y2) on a 2D plane, given its current yaw coordinate.
    Returns a tuple containing the new yaw and the distance as float values.
    """
    # Calculate the angle between the two points using the arctan2 function
    delta_x = x2 - x1
    delta_y = y2 - y1
    target_yaw = math.atan2(delta_y, delta_x)

    # Calculate the distance between the two points using the Pythagorean theorem
    distance = math.sqrt(delta_x ** 2 + delta_y ** 2)

    # Calculate the difference between the current yaw and the target yaw

    # print("target_yaw = ", target_yaw)
    # print("current_yaw = ", current_yaw)
    yaw_difference = target_yaw - current_yaw

    # Normalize the yaw difference to between -pi and pi radians
    if yaw_difference > math.pi:
        yaw_difference -= 2 * math.pi
    elif yaw_difference < -math.pi:
        yaw_difference += 2 * math.pi

    return (round(yaw_difference, 3), round(distance, 3))
```

The first function *'calculate_yaw_and_distance'* takes in the coordinates of two points on a 2D plane, the current yaw coordinate of a robot, and calculates the yaw the robot needs to turn to and the distance it needs to travel to move from the first point to the second point. The function returns a tuple containing the new yaw and the distance as float values.

```python
def euler_from_quaternion(quaternion):
    """
    Converts quaternion (w in last place) to euler roll, pitch, yaw
    quaternion = [x, y, z, w]
    Below should be replaced when porting for ROS2 Python tf_conversions is done.
    """
    x = quaternion[0]
    y = quaternion[1]
    z = quaternion[2]
    w = quaternion[3]

    sinr_cosp = 2 * (w * x + y * z)
    cosr_cosp = 1 - 2 * (x * x + y * y)
    roll = np.arctan2(sinr_cosp, cosr_cosp)

    sinp = 2 * (w * y - z * x)
    pitch = np.arcsin(sinp)

    siny_cosp = 2 * (w * z + x * y)
    cosy_cosp = 1 - 2 * (y * y + z * z)
    yaw = np.arctan2(siny_cosp, cosy_cosp)
```

The second function *'euler_from_quaternion'* takes in a quaternion, which is a mathematical notation used to represent orientations and rotations in 3D space in the format [x, y, z, w] and calculates the corresponding Euler angles (roll, pitch, and yaw) in radians. The function returns these Euler angles as float values.

```python
box_thres = 0.15
rotate_change = 0.35
slow_rotate_change = 0.10
speed_change= 0.10

time_threshold = 1.28 * box_thres / speed_change
dispenser_dist_threshold = 0.60
dist_threshold = 0.18        # Distance threshold for the robot to stop in front of the pail
initial_yaw = 0.0
front_angle = 3
front_angle_6 = 65
front_angle_range = range(-front_angle,front_angle+1,1)
stop_distance = 0.25
occ_bins = [-1, 0, 50, 101]
# f_path = '/home/nicholas/colcon_ws/src/auto_nav/auto_nav/waypoint_logging/waypoint_log.txt'
f_path = '/home/nicholas/colcon_ws/src/auto_nav/auto_nav/waypoint_logging/actual_waypoints.txt'
# f_path = '/home/nicholas/colcon_ws/src/auto_nav/auto_nav/waypoint_logging/test_waypoints.txt'
```

At the start of the program, a number of important variables are defined such as the box threshold, time threshold, distance threshold, rotation speed, linear velocity speed of the Turtlebot as well as the front angle range of the LiDAR sensor.

The box threshold represents the maximum tolerance allowed for the turtlebot to stop within the waypoint coordinates defined. If the turtlebot passes through any point in this box with the waypoint coordinates defined at the centre, the Turtlebot will stop. The time threshold represents the time it takes for the turtlebot to reach the centre of the box using the formula of time = distance/speed. It is multiplied by the arbitrary constant of 1.28 which is derived after numerous calibration attempts. The distance threshold represents the maximum distance of 0.18m that we allowed for the turtlebot to detect from its front-facing LiDAR scanner in order for it to stop in front of the tables.

9.3.2.3 Detailed breakdown of the Methods of the Navigation Node

```
def Clocking(self):
    time_started = self.get_clock().now().to_msg().sec
    time_elapsed = 0
    while time_elapsed <= time_threshold:
        rclpy.spin_once(self)
        time_diff = self.get_clock().now().to_msg().sec - time_started
        time_elapsed = self.get_clock().now().to_msg().nanosec/float(1000000000) + time_diff
        # print(self.cmd.linear.x)
```

The first method *'Clocking()'* is a timer function that we defined in order to allow the callback functions to be called simultaneously while the timer is counting. Essentially, we want the timer to count within the time defined in the time_threshold variable and at the same time, we are still able to receive the subscribed messages from the various topics without needing to use multiple threads in our program to achieve the same purpose.

```
def map2base_callback(self, msg):

    orientation_quat = msg.orientation
    quaternion = [orientation_quat.x, orientation_quat.y, orientation_quat.z, orientation_quat.w]
    (self.roll, self.pitch, self.yaw) = euler_from_quaternion(quaternion)
    self.mapbase = msg.position
    self.x_coordinate = msg.position.x
    self.y_coordinate = msg.position.y

def laser_callback(self, msg):

    # rclpy.spin_once(self)
    self.laser_range = np.array(msg.ranges[0:front_angle] + msg.ranges[-front_angle:])
    self.laser_range_6 = np.array(msg.ranges[0:front_angle_6] + msg.ranges[-front_angle_6:])
    self.laser_range[self.laser_range==0] = np.nan
    self.laser_range_6[self.laser_range_6==0] = np.nan
    self.laser_angle_increment = msg.angle_increment
    self.laser_forward = np.nanmean(self.laser_range) / 2
    self.laser_forward_6 = np.nanmean(self.laser_range_6) / 2
```

The callback functions from our first 2 subscriber objects are *'map2base_callback()'* and *'laser_callback()'* for their respective topics.

The *'map2base_callback()'* function is called whenever a message is received on the *'map2base'* topic. It extracts the roll, pitch, and yaw values from the orientation quaternion in the message and saves them as attributes of the Navigation node. It also extracts the *x* and *y* coordinates from the position in the message and saves them as attributes of the class.

The *'laser_callback()'* function is called whenever a message is received on the 'scan' topic. It extracts the laser range data from the message and stores it as an array in the class attribute laser_range. It also calculates the average distance of the laser data in front of the robot and stores it in the laser_forward attribute. Additionally, it extracts the laser angle increment from the message and stores it in the *'laser_angle_increment'* attribute.

```
def mqtt_callback(self, msg):
    data = msg.data
    print("Received MQTT data:", data)
    if data == "NIL":
        self.mqtt_val = 0
        print("MQTT data is NIL")
    else:
        self.mqtt_val = int(data)

def button_callback(self, msg):
    if msg.data:
        # print("Button pressed!")
        # self.get_logger().info('In button_callback')
        self.buttonpressed = True
    else:
        # print("Button released")
        self.buttonpressed = False
```

The next 2 callback functions from our next 2 subscriber objects are *'mqtt_callback()'* and *'button_callback()'* for their respective topics.

The *'mqtt_callback()'* callback function is called when new messages are received on the MQTT topic 'TableNum' and the *'button_callback()'* callback function is called when new messages are received on the topic *'button_pressed'* which indicates whether the button is pressed or released. In the *'mqtt_callback()'* function, the received data will be printed and if no data is received, the variable is set to be 0. Otherwise, the variable will be set to the integer value of the data. For the *'button_callback()'* function, it should be quite intuitive as the variable is set to be *'True'* when the button is pressed and *'False'* when the button is released.

```
def stopbot(self, delay):
    self.cmd.linear.x = 0.0
    self.cmd.angular.z = 0.0
    self.publisher_.publish(self.cmd)
    time.sleep(delay)


def rotatebot(self, rot_angle):
    self.get_logger().info('In rotatebot')

    rotation_speed = 0
    if abs(rot_angle) < 30:
        rotation_speed = slow_rotate_change
    else:
        rotation_speed = rotate_change
```

The stopbot method sets the linear and angular velocity of the Turtlebot to 0, publish it to the *'cmd_vel'* topic and introduce a sleep delay. Likewise, the rotatebot function prints a logged message to notify the user that the Turtlebot is rotating, takes in an argument of the angle to be rotated in degrees and implements the rotation behaviour of the turtlebot based on the input argument by publishing the angular velocity to *'cmd_vel'* topic.

```
def MoveForward(self, x_coord, y_coord):

    self.get_logger().info('Moving Forward...')
    while not ((self.mapbase.y < y_coord + box_thres and self.mapbase.y > y_coord - box_thres)
               and (self.mapbase.x < x_coord + box_thres and self.mapbase.x > x_coord - box_thres)):
        rclpy.spin_once(self)
        # self.get_logger().info('I receive "%s"' % str(self.mapbase.y))
        self.cmd.linear.x = speed_change
        self.cmd.angular.z = 0.0
        self.publisher_.publish(self.cmd)

    self.Clocking()
    self.cmd.linear.x = 0.0
    self.publisher_.publish(self.cmd)

def Reverse(self, x_coord, y_coord):

    self.get_logger().info('Reversing...')
    while not ((self.mapbase.y < y_coord + box_thres and self.mapbase.y > y_coord - box_thres)
               and (self.mapbase.x < x_coord + box_thres and self.mapbase.x > x_coord - box_thres)):
        rclpy.spin_once(self)
        # self.get_logger().info('I receive "%s"' % str(self.mapbase.y))
        self.cmd.linear.x = -speed_change
        self.cmd.angular.z = 0.0
        self.publisher_.publish(self.cmd)

    self.cmd.linear.x = 0.0
    self.publisher_.publish(self.cmd)
```

The *'MoveForward'* function accepts two parameters, *'x_coord'* and *'y_coord',* which specify where the robot should move to. It enters a loop where it checks if the robot has reached the designated location by comparing its current x and y coordinates with the specified coordinates using a specific threshold value. If the robot hasn't arrived at the target coordinates, the function sets the robot's *'linear.x'* velocity to *'speed_change'* and the *'angular.z'* velocity to zero, sends the command to the robot, and waits for the next iteration of the loop.

The *'Reverse'* method is almost identical to the *'MoveForward'* method, except that it sets the *'linear.x'* velocity to a negative value to make the robot move in reverse. The rest of the logic is the same, with the method checking whether the robot has reached the designated coordinates and publishing the appropriate commands to move the robot towards those coordinates. Once the robot reaches the designated coordinates, the method stops the robot by setting the *'linear.x'* velocity to 0.0 and publishing the command to stop the robot.

```
def move_to_waypoint(self, WP_num):
    x1, y1, x2, y2, current_yaw = self.x_coordinate, self.y_coordinate,
    self.waypoint_arr[WP_num][0], self.waypoint_arr[WP_num][1], self.yaw
    yaw_difference, distance = calculate_yaw_and_distance(x1, y1, x2, y2, current_yaw)
    yaw_difference = yaw_difference / math.pi * 180

    self.get_logger().info('Moving to waypoint %d' % WP_num)

    self.rotatebot(yaw_difference)
    self.stopbot(0.1)

    self.MoveForward(x2, y2)
    self.stopbot(0.1)

    self.get_logger().info('Waypoint reached!')

def reverse_to_waypoint1(self):

    x2, y2 = self.waypoint_arr[1][0], self.waypoint_arr[1][1]
    self.get_logger().info('Moving to waypoint 1')
    self.Reverse(x2, y2)
    self.stopbot(0.1)
    self.get_logger().info('Waypoint 1 reached!')
```

The ‘move_to_waypoint’ method takes in an argument *‘WP_num’* which is the waypoint number on the map. It calculates the difference between the current location of the robot and the designated waypoint location in terms of both the yaw angle and distance apart. Next, it then rotates the Turtlebot to face the designated direction before moving forward to the designated waypoint location and stopping there. All this is done by publishing the *‘Twist’* message consisting of its linear and angular velocity to the *‘cmd_vel’* topic. Finally, it will log the message that the waypoint has been reached.

The *‘reverse_to_waypoint1’* method moves the Turtlebot back to the waypoint just outside the dispenser, so that it can get into its docking mode once it reaches this point. It will log the message that it is moving to Waypoint 1, move the Turtlebot to the point, then stop the Turtlebot and log that Waypoint 1 has been reached.

```
def move_close(self):
    # scan the front of the robot to check the distance to the pail
    print("Moving closer to the pail...")
    print(self.laser_forward)
    while self.laser_forward > dist_threshold:
        rclpy.spin_once(self)
        # print("Moving ...")
        self.cmd.linear.x = speed_change
        self.cmd.angular.z = 0.0
        self.publisher_.publish(self.cmd)

    self.cmd.linear.x = 0.0
    self.publisher_.publish(self.cmd)
```

The '*move_close()'* method first logs the message that the Turtlebot is moving closer to the pail and the laser_forward value for the user to view. Then it checks, using a while loop, if the measured LiDAR distance is more than the distance threshold defined earlier. If True, the linear velocity will publish to the *'cmd_vel'* topic and the Turtlebot will continue moving forward. Once the distance becomes less than or equal to the threshold, the linear velocity is set to 0 and the Turtlebot stops. This is to ensure that the Turtlebot will not collide into the tables.

```
def wait_for_button_press(self):
    print("Waiting for button press...")
    while not self.buttonpressed:
        rclpy.spin_once(self.node)

def wait_for_button_release(self):
    print("Waiting for button release")
    while self.buttonpressed:
        rclpy.spin_once(self.node)
    print("Button released.")
```

The *'wait_for_button_press'* method is called at the initial stage when the Turtlebot has not received the can in the can holder. In the process, the button state is set as not buttonpressed, causing the function to be trapped in a while loop, until the button has been pressed. When the button is finally pressed, the program exits the loop and exits the function.

The *'wait_for_button_release'* method is called when the Turtlebot has reached the pail. In the process, the button state is set as buttonpressed, causing the function to be trapped in a while loop, until the button has been released. When the button is finally released, the program exits the loop and exits the function.

```python
def move_to_table6(self):
    # turn to that angle
    self.rotatebot(math.degrees(-self.yaw))
    rclpy.spin_once(self)

    # print("Laser range = ", self.laser_range)
    # print("Laser range 6 = ", self.laser_range_6)

    # Calculate angle of minimum range value
    min_range_index = np.nanargmin(self.laser_range_6)
    min_range = self.laser_range_6[min_range_index]
    min_range_angle = (min_range_index - front_angle_6) * self.laser_angle_increment

    # Convert angle to degrees and print result
    if min_range_angle < 0:
        min_range_angle_degrees = -(front_angle_6 + math.degrees(min_range_angle))
    else:
        min_range_angle_degrees = front_angle_6 - math.degrees(min_range_angle)

    # print("Angle to turn: ", min_range_angle_degrees)
    self.table6_turn_angle = min_range_angle_degrees

    print("Turning to table 6...")
    print("Angle to turn: ", self.table6_turn_angle)
    self.rotatebot(-self.table6_turn_angle)

    self.move_close()
```

After reaching the waypoint 14, the program calls the ‘rotatebot’ method to turn the Turtlebot to face the West. The LiDAR then scans the area ahead of it, finding the minimum distance of any object ahead of it, using the range of the laser, *‘laser_range_6’*. This is representative of the distance of the pail from the Turtlebot. The program obtains the information about the angle of the pail with respect to the Turtlebot facing West. Then, the program calculates the minimum angle the Turtlebot has to turn to face that angle from its current pose. The program calls the *‘rotatebot’* method once again, taking in the minimum turning angle as input. The Turtlebot turns to face the pail, then the *‘move_close’* method is called for the Turtlebot to move closer to the pail until it is within the acceptable distance.

```python
def dock(self):
    self.move_to_waypoint(1)
    self.move_to_waypoint(18)
    self.move_to_waypoint(0)
    self.rotatebot(-self.yaw)
```

The set of waypoints listed allow the Turtlebot to transition smoothly from returning to the dispenser to docking in the dispenser. The ‘*rotatebot()’* function is called to reorientate the Turtlebot to face the dispenser for better alignment to collect the can.

```
def face_front(self):
    rotation_angle = ((-(math.pi / 2) ) - self.yaw)
    self.rotatebot(math.degrees(rotation_angle))
```

In the waypoint algorithm, the Turtlebot always approaches the pail from the South of the arena. This function ensures that the Turtlelbot always faces North after reaching its final waypoint near the pail. The angle for the Turtlebot to rotate is calculated then set as input into the *'rotatebot()'* function in terms of degrees.

9.3.2.4 Detailed breakdown of the Flow of the Waypoint Algorithm

```
def main(args=None):

    rclpy.init(args=args)
    hardcoded_navi_node = Navigation()
    hardcoded_navi_node.motion()
    hardcoded_navi_node.destroy_node()
    rclpy.shutdown()
```

When the *'auto_nav'* program is started, a `Navigation` node is created and it is initialised with its constructor as well as its following methods.

```
def motion(self):

    try:
        while rclpy.ok():

            print("current yaw = ", self.yaw)
            print("current Xpos = ", self.x_coordinate)
            print("current Ypos = ", self.y_coordinate)

            self.wait_for_button_press()

            print("Waiting for mqtt input...")
            while self.mqtt_val == 0:
                rclpy.spin_once(self)

            table_num = self.mqtt_val
            print("table_num received = ", table_num)
```

The `motion()` function is called, and the program enters a while loop which loops indefinitely until the user presses *"Ctrl+C"* on the keyboard. The program prints the coordinates and the yaw of the Turtlebot to the console, then waits for the button to be pressed. The program only moves on after the button has been pressed (aka the can has fallen into the can holder and pressed onto the button in the can holder). During which, the program proceeds to wait for MQTT input

from the user. Once the user has keyed in a delivery table number into the mobile application and sent it over successfully, the program obtains the delivery table number and prints it to the console.

```
self.reverse_to_waypoint1()

if (table_num == 1):
    # moving to the table
    self.move_to_waypoint(9)
    self.move_to_waypoint(16)
    self.face_front()
    self.move_close()

    self.wait_for_button_release()

    # moving back to the dispenser
    self.move_to_waypoint(9)
    self.dock()
```

The Turtlebot starts off from within the dispenser. It is first made to move out of the dispenser by reversing to the first waypoint. Depending on the table number that was pressed, the Turtlebot would follow the same set of algorithms but for different waypoints. In each case, the Turtlebot is made to move to the waypoint closest to the table, then re-orientate its yaw to face the pail, then inch forward until it is within 15 cm from the pail. Once the Turtlebot has reached a position close enough to the pail, it stops moving and waits for the button to be released (aka waits for the can to be removed from the can holder). Once that action is completed, the Turtlebot proceeds to move back to the dispenser via the same set of waypoints but in reverse order. Finally, it performs a docking algorithm to be in the right position to receive the next can from the dispenser.

```
else: # table 6
    # moving to the table
    self.move_to_waypoint(17)
    self.move_to_waypoint(12)
    self.move_to_waypoint(13)
    self.move_to_waypoint(14)
    self.move_to_table6()

    self.wait_for_button_release()

    # moving back to the dispenser
    self.move_to_waypoint(14)
    self.move_to_waypoint(13)
    self.move_to_waypoint(12)
    self.move_to_waypoint(17)
    self.dock()
```

The algorithm for table 6 follows the same algorithm, except that it uses an additional special function which locates the position of the table.

```
        self.buttonpressed = False  # reset the buttonpressed
        self.mqtt_val = 0       # reset the mqtt_val to 0
```

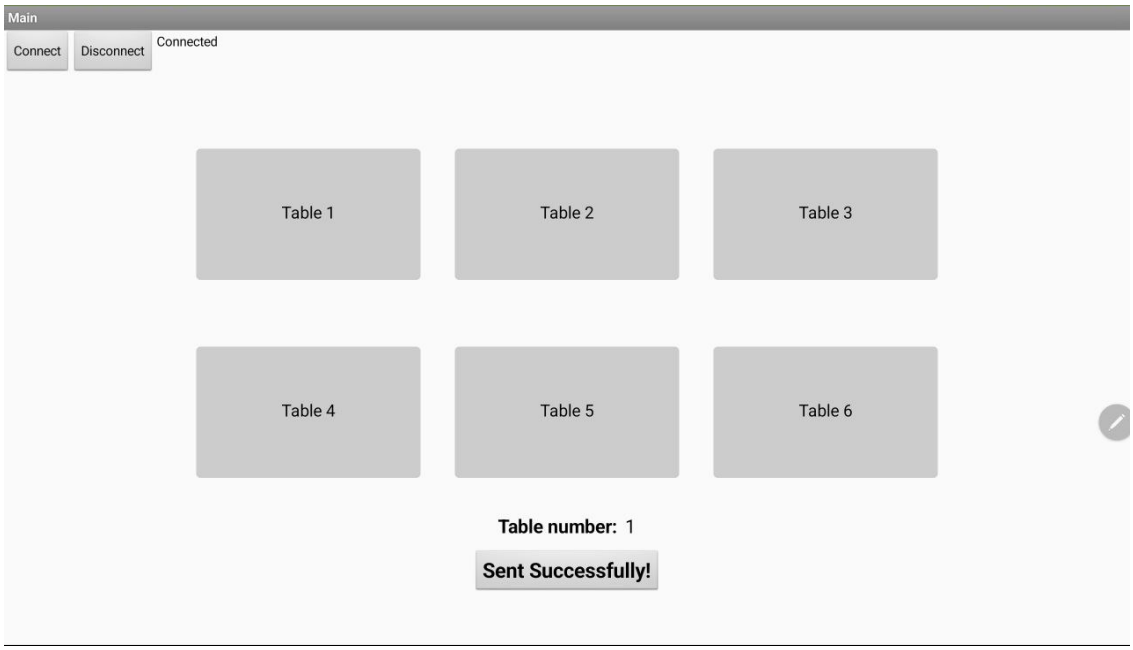
After the Turtlebot has docked in the dispenser, the Boolean for button pressed and the integer value for MQTT is reset before the next iteration of the program again.

```
except Exception as e:
    print(e)

# Ctrl-c detected
finally:
        # stop moving
    self.cmd.linear.x = 0.0
    self.cmd.angular.z = 0.0
    self.publisher_.publish(self.cmd)
```

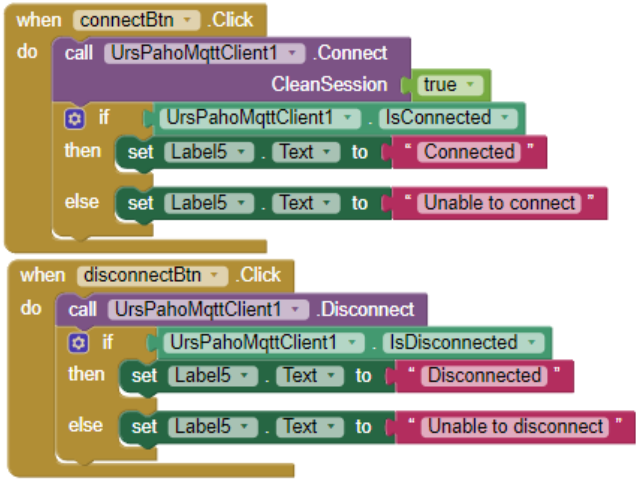
If an error has occurred anywhere in the program, the error is printed out on to console, then the program crashes. When the program receives the *"Ctrl+C"* user input, it is an indication to terminate the program. The Turtlebot is made to stop moving by publishing a zero linear and angular velocity to the topic *'cmd_vel'*.

9.3.3 Detailed breakdown of the Mobile Application



In MIT App Inventor, a Connect and Disconnect button allows the user to seamlessly establish and break a connection between the mobile application and the laptop. 6 buttons are created corresponding to each of the 6 delivery table numbers. There is a label which changes its state based on the table number pressed in the mobile application, to indicate the table number that has been registered. Finally, there is a *"Send"* button which changes to read *"Sent Successfully!"* when the user sends the information of the table number over to the laptop.

By ensuring that the fields of the MQTT broker, username and password are consistent in the MIT App Inventor and the laptop code, we can ensure that the laptop and the mobile application can communicate over the broker.

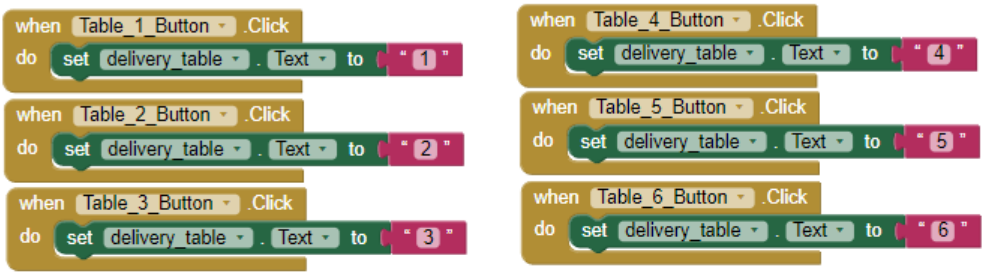


In the MIT App Inventor, we set logic for the *"Connect"* and *"Disconnect"* buttons. The *"UrsPahoMqttClient"* is the in-built MQTT widget for establishing MQTT connections with external devices.

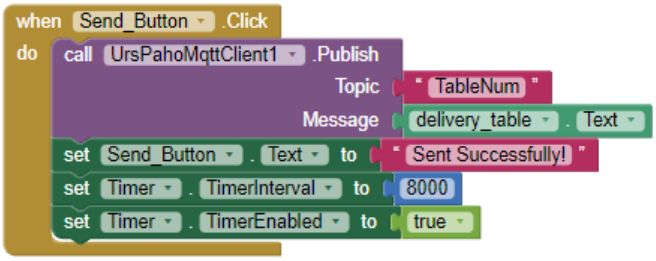
When the *"Connect"* button is pressed, we attempt to establish a new MQTT connection between the mobile application and the laptop over the MQTT broker. If the connection has been actualised, then show the status to be *"Connected"*. However, if the connection was not actualised, then an error must have occurred. Show the status as *"Unable to connect"*.

When the *"Disconnect"* button is pressed, we attempt to break the MQTT connection between the mobile application and the laptop over the MQTT broker. If the connection has been

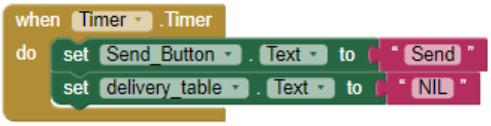
discontinued successfully, then show the status to be *"Disconnect"*. However, if the connection was not disabled, then an error must have occurred. Show the status as *"Unable to disconnect"*.



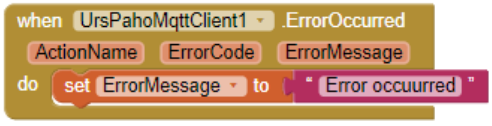
Then, we set logic for the buttons containing the table numbers. When the button to a table is pressed, we set the state of the delivery table number to that of the button pressed.



We set logic for the *"Send"* button. When the button is pressed, we call the MQTT Client to publish the *"TableNum"* topic with the message containing the details of the delivery table number that was entered. This is sent in the Text format, and will need to be converted into an integer later on. The *"Send"* caption changes to *"Sent Successfully!"* for a total of 8 seconds, before reverting to the original *"Send"* button again.



We set a *"Timer"* which resets the caption back to *"Send"* and the caption of the delivery table number back to *"NIL"* once the 8 second timer is up.



This code block is responsible for informing the user if there has been an error in establishing a stable MQTT connection at any point of the process.

```
broker_address = "[redacted]"
topic = "TableNum"

rclpy.init()
node = rclpy.create_node("mqtt_mobile")
pub = node.create_publisher(String, "mqtt_data", 10)
client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.username_pw_set("nicholas", "EG2310")
client.on_log = on_log
client.enable_logger()
client.connect(broker_address)
client.loop_start()  # Start networking daemon

try:
    rclpy.spin(node)
finally:
    client.loop_stop()
    client.disconnect()
    node.destroy_node()
    rclpy.shutdown()
```

In '*mqtt/receiver.py*', we execute some setup code. We initialise a node *"mqtt_mobile"* which publishes the topic *"mqtt_data"*. We start an MQTT client, which is set to perform a particular set of actions when it has successfully established an MQTT connection, when a message has been sent over the connection and when we want to log information onto the console. A username and password for the MQTT broker is set, and this same set of details need to be input into the *"UrsPahoMqttClient"* function. We also set the topic name to be *"TableNum"* which will be the topic which the node sends the information of the delivery table number over.

```
def on_connect(client, userdata, flags, rc):
    print("Connected with result code "+str(rc))
    client.subscribe(topic)
```

When the mobile application has successfully established a MQTT connection with the laptop, the program prints out a confirmation into the console. Then the client subscribes to the topic which was earlier mentioned to be *"TableNum"*, in order to receive messages sent over that topic.

```
def on_message(client, userdata, msg):
    message = str(msg.payload.decode("utf-8"))
    print("Message received on topic "+msg.topic+" with payload "+message)

    msg = String()
    msg.data = message
    pub.publish(msg)
```

When a message is sent over from the mobile application, it is in the form of an encoded string. The string first needs to be decoded, then stored into the appropriate *“String”* datatype before it is published to the ROS topic, *“mqtt_data”*.

```
def on_log(client, userdata, level, buf):
    print("Log: ", buf)
```

This function logs events that occurred, including the establishment of MQTT connection and the messages that have been received.

9.3.4 Detailed breakdown of the button program on the RPi

```
class ButtonInputNode(Node):

    def __init__(self):
        super().__init__('button_input_node')

        self.button_pin = 18

        GPIO.setmode(GPIO.BCM)
        GPIO.setup(self.button_pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)

        self.publisher_ = self.create_publisher(Bool, 'button_pressed', 10)

        self.timer_ = self.create_timer(0.1, self.check_button_press)
        self.get_logger().info('Ready for button press!')

    def check_button_press(self):
        if GPIO.input(self.button_pin) == GPIO.LOW:
            self.get_logger().info('Button pressed')
            msg = Bool()
            msg.data = True
            self.publisher_.publish(msg)
            time.sleep(0.2)
        else:
            self.get_logger().info('Button released')
            msg = Bool()
            msg.data = False
            self.publisher_.publish(msg)
            time.sleep(0.2)
```

The class ButtonInputNode is defined and its constructor initializes the node with the name *'button_input_node'*. It also sets the GPIO mode to BCM mode and sets up the defined button pin as an input with a pull-up resistor. A publisher is also created with message type '*Bool'* and topic name *'button_pressed'*, with a queue size of 10. A timer is created to call the callback function every 0.1 second. The callback function is defined to check whether the button is pressed or not. If it is pressed, a log message will be printed and the value of type *'Bool'* message will become *True* and will be subsequently published to the topic *'button_pressed'*. Similarly, if the button is released, the value of type *Bool* message will become *False* and will be published to the topic *'button_pressed'.* After publishing the message, there is a 0.2 second sleep to debounce the button.

```python
def main(args=None):
    print('Preparing setup... please wait')
    rclpy.init(args=args)

    node = ButtonInputNode()

    rclpy.spin(node)

    node.destroy_node()
    rclpy.shutdown()
```

Once the button program on the RPi is started, a `ButtonInputNode` node is created and the node is set to spin indefinitely to publish the changes in the pressed status of the button.

```
KeyboardInterrupt

ubuntu@ubuntu:~/turtlebot3_ws/src/auto_nav/auto_nav$ button_pub
Preparing setup... please wait
[INFO] [1682003411.188985548] [button_input_node]: Ready for button press!
[INFO] [1682003411.198107127] [button_input_node]: Button released
[INFO] [1682003411.407549326] [button_input_node]: Button released
[INFO] [1682003411.617193149] [button_input_node]: Button released
[INFO] [1682003411.826079175] [button_input_node]: Button released
[INFO] [1682003412.035451993] [button_input_node]: Button released
[INFO] [1682003412.245188825] [button_input_node]: Button released
[INFO] [1682003412.454553243] [button_input_node]: Button released
[INFO] [1682003412.664091941] [button_input_node]: Button released
[INFO] [1682003412.873624388] [button_input_node]: Button released
[INFO] [1682003413.083136308] [button_input_node]: Button released
[INFO] [1682003413.292736503] [button_input_node]: Button released
[INFO] [1682003413.502256486] [button_input_node]: Button released
[INFO] [1682003413.711778552] [button_input_node]: Button released
[INFO] [1682003413.921295931] [button_input_node]: Button released
[INFO] [1682003414.130635739] [button_input_node]: Button released
[INFO] [1682003414.339482289] [button_input_node]: Button released
[INFO] [1682003414.549014757] [button_input_node]: Button released
```

The log message of the button's pressed status will be printed every 0.1 second as shown above. When there is nothing inside the can holder, the `ButtonInputNode` will publish the message with '*Bool*' value of *False* to the topic *'button_pressed'* for the main navigation code *'auto_nav'* to subscribe to.

9.3.5 Detailed breakdown of the program on the Arduino Uno

```
int trigPin = 9;    // Ultrasonic sensor trigger pin
int echoPin = 10;   // Ultrasonic sensor echo pin
int threshold = 22.5; // Distance threshold in centimeters
Servo servo1;       // Servo motor 1
Servo servo2;       // Servo motor 2
```

Variables are initialised to be used later in the program. These include the trig and echo pins on the ultrasonic sensor, the distance threshold value of the ultrasonic sensor as well as instantiating instances for the 2 servo motors.

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600); // Initialize serial communication
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
  servo1.attach(5);   // Attach servo motor 1 to pin 5
  servo2.attach(6);   // Attach servo motor 2 to pin 6
  servo1.write(85);
  servo2.write(200);
}
```

In the '*setup()'* function, we initialise the serial communication to allow for messages to be printed out on the serial monitor for debugging. We set the trig pin on the ultrasonic sensor as output, and the echo pin as input. Then, we initialise the 2 servos with their corresponding PWM pins that they are attached to. Finally, we rotate the servo horns such that they are in the initial state to be able to hold the can in place.

```
void loop() {
  // put your main code here, to run repeatedly:
  long duration, distance;
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);
  duration = pulseIn(echoPin, HIGH);
  distance = duration / 58;
```

At the start of the '*loop()'* function, we set the trigPin to LOW state to ensure that the ultrasonic sensor is not emitting any pulse. After 2 microseconds, we set the trigPin to *'HIGH'* state to emit an ultrasonic pulse for 10 microseconds. Then, we set the trigPin to *'LOW'* state to stop the emission of the ultrasonic pulse. We wait for the ultrasonic pulse to bounce back from an object and be detected by the sensor. The pulseIn function returns the duration of the pulse in

microseconds. Finally, calculate the distance of the object in centimeters by dividing the duration by 58, which is a constant that represents the speed of sound in air in microseconds per centimeter. The resulting value is stored in the distance variable.

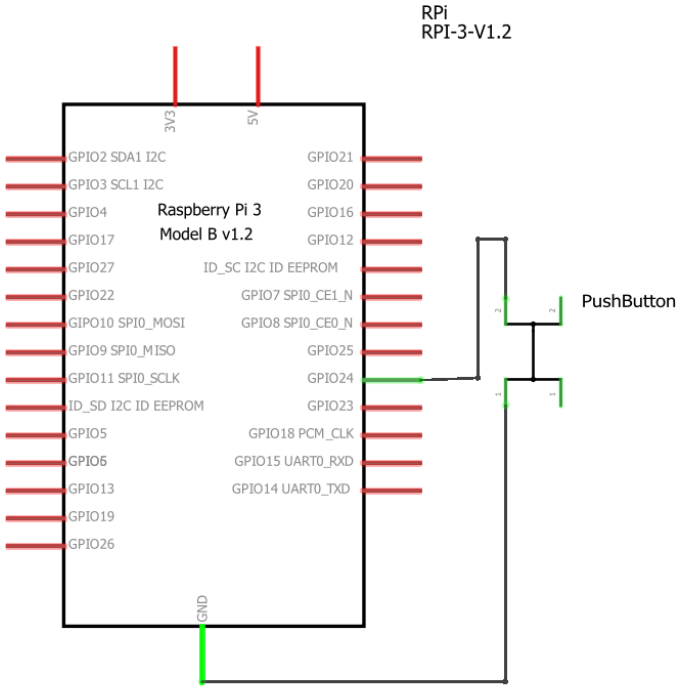
```
  if (distance <= threshold) { // If an object is detected
    Serial.println("Object detected!");
    delay(2000);       // Wait for 2 seconds
    servo1.write(175); // Rotate servo motor 1 to activated position
    servo2.write(110); // Rotate servo motor 2 to activated position
    delay(5000);       // Wait for 5 seconds
  }
  else {
    servo1.write(85);  // Rotate servo motor 1 to deactivated position
    servo2.write(200); // Rotate servo motor 2 to deactivated position
    delay(1000);
  }
}
```

If the ultrasonic sensor measures a distance less than or equal to the threshold distance set earlier, this means that the Turtlebot is detected in the dispenser. Then we write to the servo motors to actuate them. This drops the can. When the measured distance becomes greater than the threshold value, this means that the Turtlebot is no longer in the dispenser. We write to the servo motors to deactivate them, bringing the servo horns back to the original position.

9.4 Electrical Assembly

9.4.1 Schematic for RPi

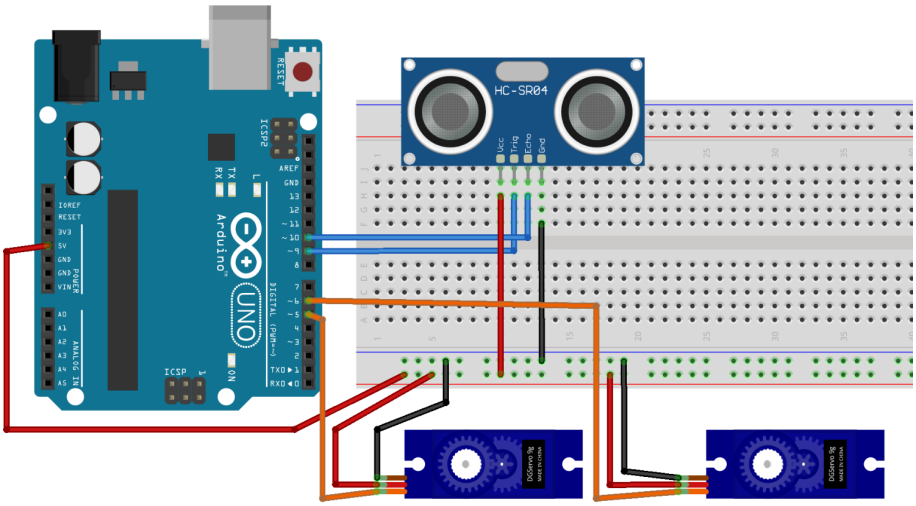
The diagram below shows the schematic of the connection between the RPi and the button on the can holder.



The push button is set as an input and is powered by 3.3V from the GPIO24 pin of the RPi. The other end of the push button is connected to ground as shown in the schematic. The RPi constantly checks whether the button has been pressed or released through the GPIO24 pin.

9.4.2 Schematic for Arduino Uno

The diagram below shows the circuit connections of the connection between the Arduino Uno and the ultrasonic distance sensor as well as the servo motors.

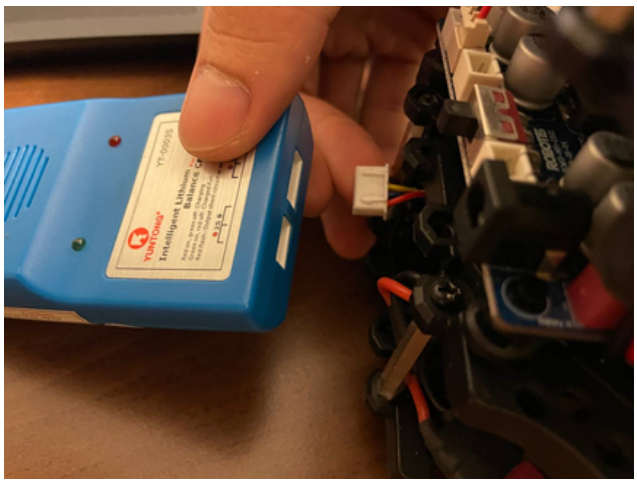


The ultrasonic sensor's Trig pin and Echo pins are connected to pins 9 and 10 respectively on the Arduino Uno. The pins of the servo motors are connected to the PWM pins 5 and 6 respectively.

10. System Operation Manual

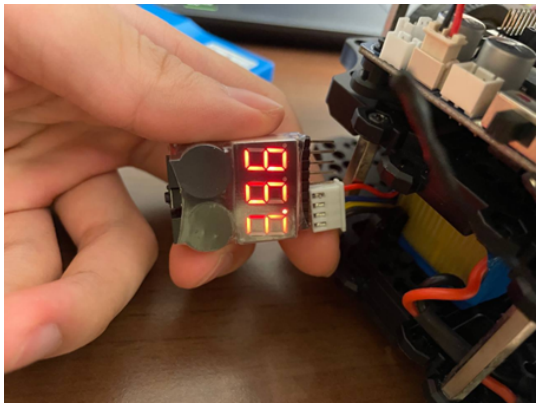
10.1 Charging Battery

1. Connect the Li-Po battery's charging head into the charger's port as in the picture below.



2. Connect the charger to a wall outlet using the included black cable as in the picture below. If the red LED light turns on, the Li-Po battery is being charged. When the battery is fully charged, the green LED light will turn on.

10.2 Checking Battery Level



The battery voltage level can be checked using the Li-Po Battery Voltage Tester. Connect the battery to the tester starting from the negative pin as shown on the right. The tester will indicate the voltage level of the battery.

10.3 Software Boot-up Protocol

1. ssh into the RPi on 2 separate terminals by inputting

```
sshrp
```

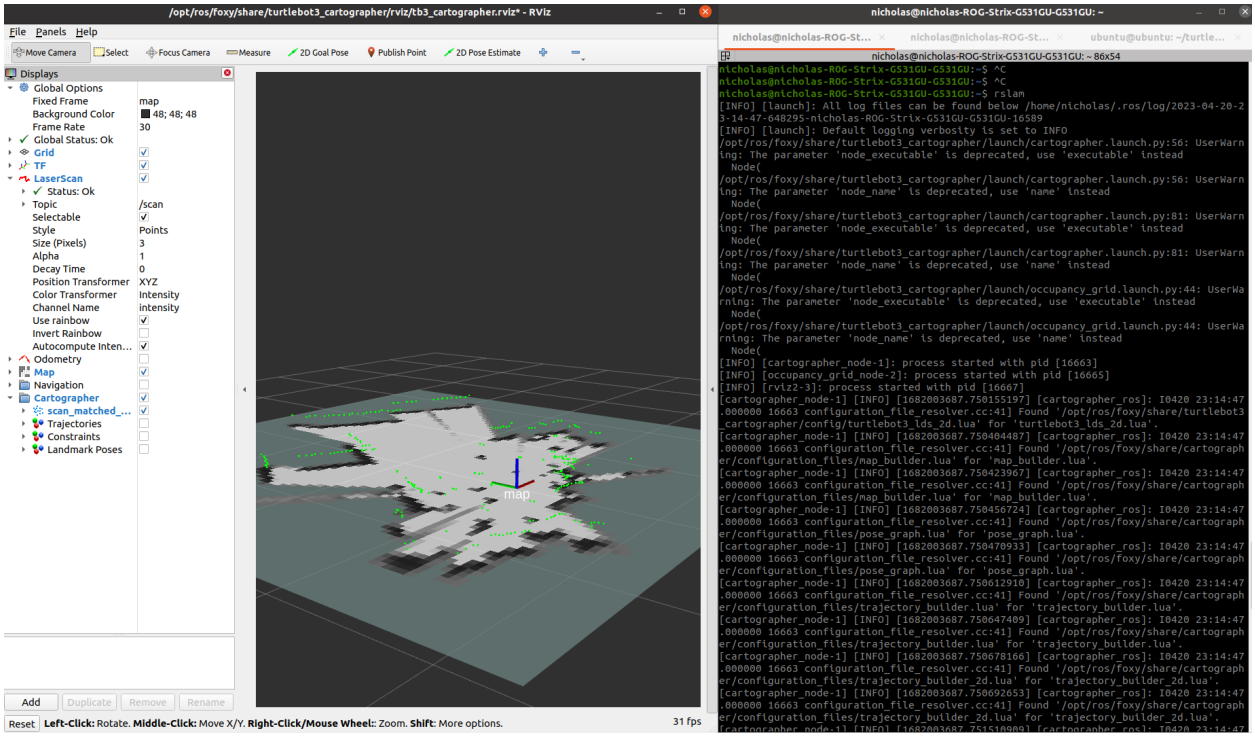
2. Once logged into the RPi, run *'rosbu'* and *'button_pub'* on the separate terminals to start up the Turtlebot and the button respectively

```
ubuntu@ubuntu:~/turtlebot3_ws/src/auto_nav/auto_nav$ rosbu
[INFO] [launch]: All log files can be found below /home/ubuntu/.ros/log/2023-04-20-15-13-01-524221-ubuntu-1893
[INFO] [launch]: Default logging verbosity is set to INFO
urdf_file_name : turtlebot3_burger.urdf
/opt/ros/foxy/share/hls_lfcd_lds_driver/launch/hlds_laser.launch.py:46: UserWarning: The parameter 'node_executable' is deprecated, use 'executable' instead
  Node(
/opt/ros/foxy/share/hls_lfcd_lds_driver/launch/hlds_laser.launch.py:46: UserWarning: The parameter 'node_name' is deprecated, use 'name' instead
  Node(
[INFO] [robot_state_publisher-1]: process started with pid [1895]
[INFO] [hlds_laser_publisher-2]: process started with pid [1897]
[INFO] [turtlebot3_ros-3]: process started with pid [1899]
[hlds_laser_publisher-2] [INFO] [1682003582.684128066] [hlds_laser_publisher]: Init hlds_laser_publisher Node Main
[hlds_laser_publisher-2] [INFO] [1682003582.684627147] [hlds_laser_publisher]: port : /dev/ttyUSB0 frame_id : base_scan
[robot_state_publisher-1] Parsing robot urdf xml string.
[robot_state_publisher-1] Link base_link had 5 children
[robot_state_publisher-1] Link caster_back_link had 0 children
[robot_state_publisher-1] Link imu_link had 0 children
[robot_state_publisher-1] Link base_scan had 0 children
[robot_state_publisher-1] Link wheel_left_link had 0 children
[robot_state_publisher-1] Link wheel_right_link had 0 children
[robot_state_publisher-1] [INFO] [1682003582.839723985] [robot_state_publisher]: got segment base_footprint
[robot_state_publisher-1] [INFO] [1682003582.841382954] [robot_state_publisher]: got segment base_link
[robot_state_publisher-1] [INFO] [1682003582.841552544] [robot_state_publisher]: got segment base_scan
[robot_state_publisher-1] [INFO] [1682003582.841654839] [robot_state_publisher]: got segment caster_back_link
[robot_state_publisher-1] [INFO] [1682003582.841702810] [robot_state_publisher]: got segment imu_link
[robot_state_publisher-1] [INFO] [1682003582.841751562] [robot_state_publisher]: got segment wheel_left_link
[robot_state_publisher-1] [INFO] [1682003582.841794220] [robot_state_publisher]: got segment wheel_right_link
[turtlebot3_ros-3] [INFO] [1682003582.878524992] [turtlebot3_node]: Init TurtleBot3 Node Main
[turtlebot3_ros-3] [INFO] [1682003582.881167905] [turtlebot3_node]: Init DynamixelSDKWrapper
[turtlebot3_ros-3] [INFO] [1682003582.886707335] [DynamixelSDKWrapper]: Succeeded to open the port(/dev/ttyACM0)!
[turtlebot3_ros-3] [INFO] [1682003582.895045620] [DynamixelSDKWrapper]: Succeeded to change the baudrate!
[turtlebot3_ros-3] [INFO] [1682003582.936229583] [turtlebot3_node]: Start Calibration of Gyro
[turtlebot3_ros-3] [INFO] [1682003587.936651580] [turtlebot3_node]: Calibration End
[turtlebot3_ros-3] [INFO] [1682003587.936914819] [turtlebot3_node]: Add Motors
[turtlebot3_ros-3] [INFO] [1682003587.938101999] [turtlebot3_node]: Add Wheels
[turtlebot3_ros-3] [INFO] [1682003587.939741124] [turtlebot3_node]: Add Sensors
[turtlebot3_ros-3] [INFO] [1682003587.949728116] [turtlebot3_node]: Succeeded to create battery state publisher
[turtlebot3_ros-3] [INFO] [1682003587.960556078] [turtlebot3_node]: Succeeded to create imu publisher
[turtlebot3_ros-3] [INFO] [1682003587.968845507] [turtlebot3_node]: Succeeded to create sensor state publisher
[turtlebot3_ros-3] [INFO] [1682003587.971647696] [turtlebot3_node]: Succeeded to create joint state publisher
[turtlebot3_ros-3] [INFO] [1682003587.971930832] [turtlebot3_node]: Add Devices
[turtlebot3_ros-3] [INFO] [1682003587.972042503] [turtlebot3_node]: Succeeded to create motor power server
[turtlebot3_ros-3] [INFO] [1682003587.978850833] [turtlebot3_node]: Succeeded to create reset server
[turtlebot3_ros-3] [INFO] [1682003587.982322631] [turtlebot3_node]: Succeeded to create sound server
[turtlebot3_ros-3] [INFO] [1682003587.986003135] [turtlebot3_node]: Run!
[turtlebot3_ros-3] [INFO] [1682003588.031831379] [diff_drive_controller]: Init Odometry
[turtlebot3_ros-3] [INFO] [1682003588.061137487] [diff_drive_controller]: Run!
```

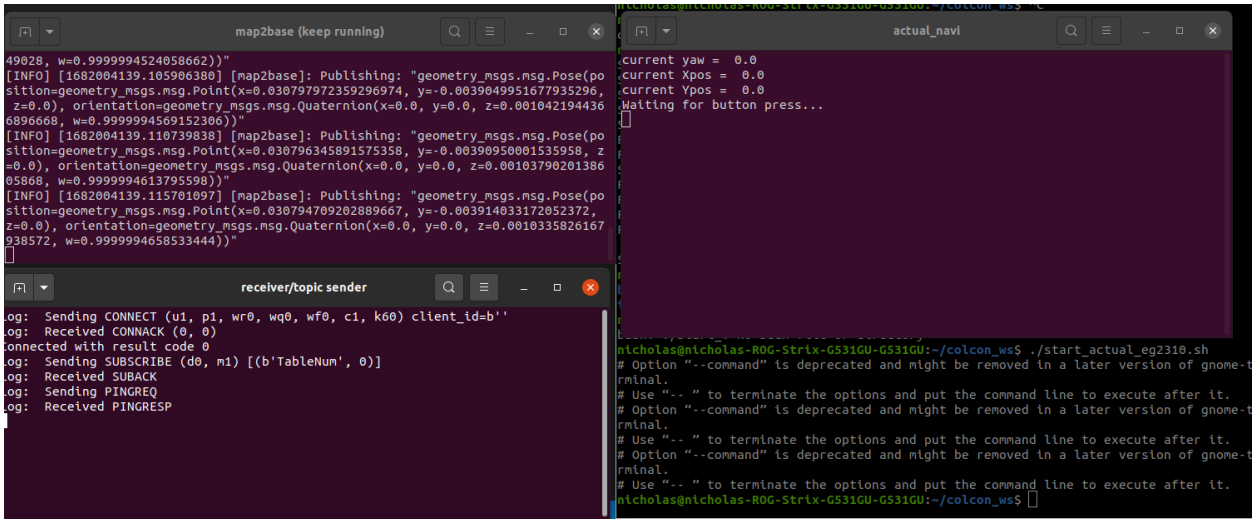
```
KeyboardInterrupt

ubuntu@ubuntu:~/turtlebot3_ws/src/auto_nav/auto_nav$ button_pub
Preparing setup... please wait
[INFO] [1682003411.188985548] [button_input_node]: Ready for button press!
[INFO] [1682003411.198107127] [button_input_node]: Button released
[INFO] [1682003411.407549326] [button_input_node]: Button released
[INFO] [1682003411.617193149] [button_input_node]: Button released
[INFO] [1682003411.826079175] [button_input_node]: Button released
[INFO] [1682003412.035451993] [button_input_node]: Button released
[INFO] [1682003412.245188825] [button_input_node]: Button released
[INFO] [1682003412.454553243] [button_input_node]: Button released
[INFO] [1682003412.664091941] [button_input_node]: Button released
[INFO] [1682003412.873624388] [button_input_node]: Button released
[INFO] [1682003413.083136308] [button_input_node]: Button released
[INFO] [1682003413.292736503] [button_input_node]: Button released
[INFO] [1682003413.502256486] [button_input_node]: Button released
[INFO] [1682003413.711778552] [button_input_node]: Button released
[INFO] [1682003413.921295931] [button_input_node]: Button released
[INFO] [1682003414.130635739] [button_input_node]: Button released
[INFO] [1682003414.339482289] [button_input_node]: Button released
[INFO] [1682003414.549014757] [button_input_node]: Button released
```

3. Open 2 tabs of Linux Bash terminal on your computer.
4. On one terminal, run *'rslam'* to start up Rviz once *'rosbu'* has finished setting up. *'rosbu'* indicates that it completes the setup when it returns the statement *'Run'*.

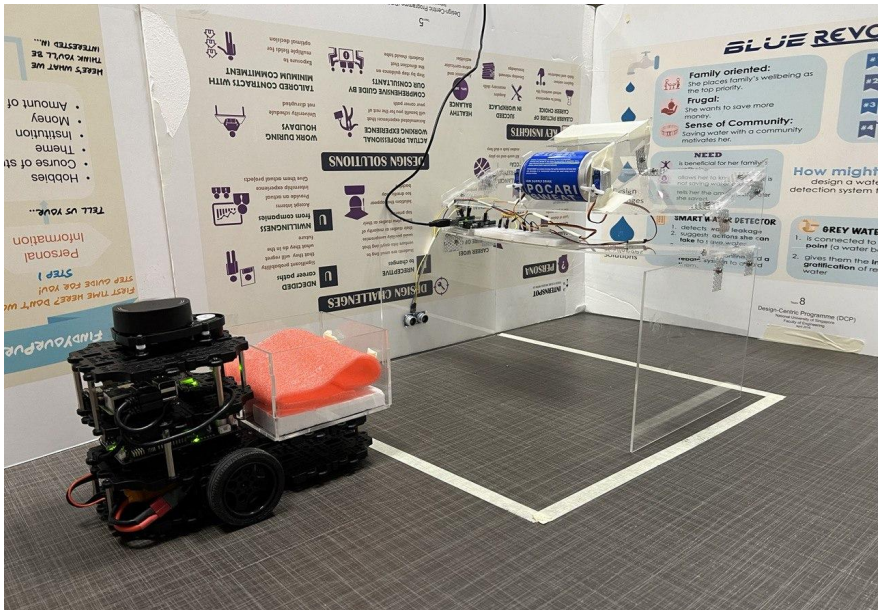


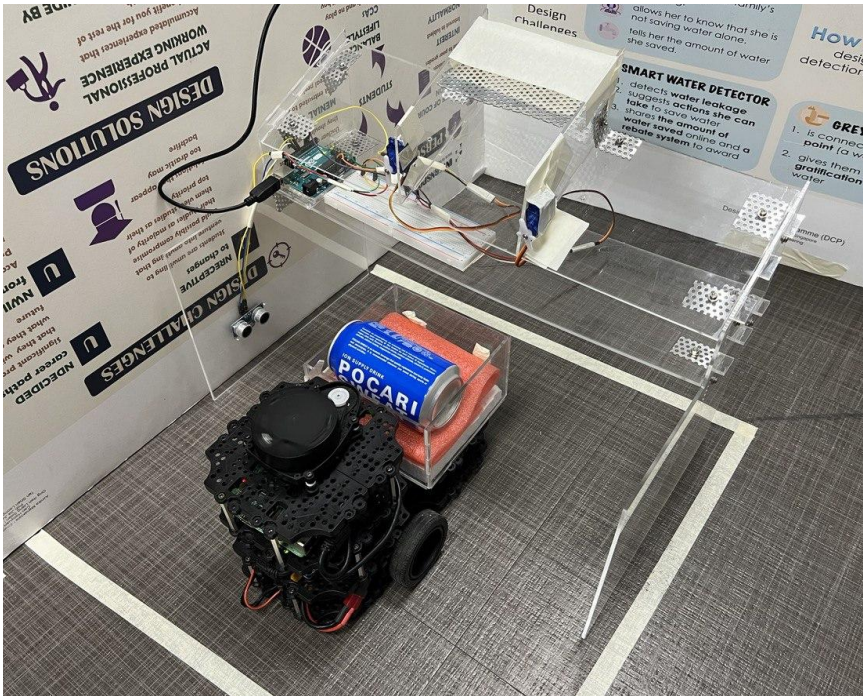
5. In the other new terminal on your computer, the user will enter the command *'./start_actual_eg2310.sh'* to run the script which will open 3 new terminals running the *'map2base'* node, MQTT receiver log and the navigation node.



10.4 Loading the Drink Can

For safety purposes, the drink can is only available for loading whenever the Turtlebot is at least 1 metre away from the dispenser. This is because the Dispenser will activate once it detects the Turtlebot is inside of it. Once the Turtlebot has stopped inside of the Dispenser, the servo motor will actuate 90 degrees and the drink can will roll off the Dispenser into the can holder as seen below.




DESIGN SOLUTIONS
ACTUAL PROFESSIONAL
WORKING EXPERIENCE
SMART WATER DETECTOR
POCARI

11. Troubleshooting

11.1 Troubleshooting - Software

1. Laptop unable to connect to the Turtlebot
    a. Ensure that the laptop and the RPi are connected to the same network.
    b. Ensure that the `ROS_DOMAIN_ID` in the `~/.bashrc` file of the laptop and the RPi is the same.
    c. Change the `ROS_DOMAIN_ID` on both the laptop and the RPi to prevent conflict of communication with those on the same network.

2. RPi unable to connect the WiFi
    a. Ensure that the correct SSID and password is in the netplan of the RPi

    ```
    cd /media/$USER/writable/etc/netplan
    $ sudo nano 50-cloud-init.yaml
    ```

    b. Replace `WIFI_SSID` and `WIFI_PASSWORD` with your wifi SSID and password

3. ImportError, no module named "***".
    a. Simply install the module using the following command:
    `pip install <module_name>`

4. *'colcon build'* returns error/warning
    a. Ensure that you are in the correct directory `~/colcon_ws` or `~/turtlebot3_ws`
    b. Run *'colcon build'* again

5. No change observed in program even after changes in code
    a. Ensure that *'colcon build'* was run in the correct directory
    b. Ensure `source install/setup.bash` was run
    c. Ensure that the correct program is being run
    d. Rebuild the package, source it, and then run the program again

6. Unable to run the program on either the laptop or RPi
    a. Rebuild the package, source it, and run the program again

7. Unable to run *'button_pub'* or command not found
    a. Ensure that the alias for *'button_pub'* is correct in .bashrc
    b. Check the package *'button.py'* was included in
    `~/colcon_ws/src/auto_nav/auto_nav/setup.py`
    c. Rebuild the package, source it, and then run the program again

8. Unable to establish MQTT connection between laptop and mobile device
    a. Ensure that both laptop and mobile device are connected to the same network used to connect to the RPi

b. Check that the address of the broker as listed in the `broker_address` variable in `~/colcon_ws/src/auto_nav/auto_nav/mqtt/receiver.py` is the same as the address stated in the *"Broker"* field of the `UrsPahoMqttClient1` in MIT App Inventor. This address should tally with the IP address of the connected device

11.2 Troubleshooting - Hardware

1. Servo Motor not moving
    a. Ensure that the wires have not come loose
    b. Ensure that the Ultrasonic Sensor light is flashed green
    c. Swap out with a different servo and run the program again

2. Ultrasonic Sensor's light is not flashed green
    a. Ensure that the wires behind have not come loose
    b. Swap out with a different ultrasonic sensor and run the program again

3. Button not activating
    a. Ensure that the wires have not come loose

4. Arduino's light is not flashed green
    a. Ensure that it is properly connected
    b. Ensure that the input voltage is at least 5V
    c. Replace the Arduino with a new one if necessary

5. Turtlebot not moving smoothly
    a. Check rotation of front ball caster is smooth
        i. If not smooth, either oil ball caster or replace it
    b. Check rotation of rear ball caster is smooth
        i. If not smooth, either oil ball caster or replace it
    c. Check that DYNAMIXEL motors are spinning consistently via `rteleop`

6. Turtlebot not moving
    a. Ensure that no connection wires are disconnected or loose
    b. Ensure that the Raspberry Pi's light is flashed green and red
        i. If the Raspberry Pi's light is red only, there might be an issue with the booting. Check the SD card and replace it if necessary
        ii. If it still doesn't work, it might be an issue with the Raspberry Pi. Replace it with another one
    c. Ensure that the LDS-01 LiDAR is spinning
        i. If it is not, try running *'rosbu'* in the terminal and check if the LiDAR spins after that. If it doesn't, replace the LiDAR

12. Future Work

The Turtlebot was able to complete the mission, despite a few flaws in the design and integration that limit the ability of the system to operate efficiently. This section lists out the flaws in terms of mechanical, electrical and software, and the possible ways to improve the system in future works.

12.1 Mechanical

Since we are only required to do simple tasks with the Cart (Can Holder) and the Dispenser, we came up with very straightforward designs that mostly use acrylic as its main material. Since we were able to obtain almost all the necessary material and components, our cost was down to almost zero despite the $6 Laser Cutting Machine fee. Furthermore, despite the flimsy and fragile appearance of the Cart (Can Holder) and the Dispenser, they both were able to carry out the mission without any design failure. However, during the process of integrating the mechanical, electrical and software, we made some changes in the system, such that the design could be improved to better match the system operation workflow.

Structural Integrity

Since both the Can Holder and Dispenser use acrylic sheet as their main material, they appear to be rather flimsy and fragile. Although our system still worked well enough to finish the mission, in the future, we might need to improve the structural integrity of our system to improve its strength and durability. There are many ways and materials to use to improve our system to have better structural integrity and the most common ways are to use aluminium profile bars as the main support structure or to make use of plastic injection mould to mass produce the can holder.

Height of the Dispenser

The initial system operation was to make the turtlebot drive through the dispenser, stop, let the dispenser drop the can, and continue to move out of the dispenser in a forward motion. However, after we manufactured both the Cart and the Dispenser, we decided that we would like to do docking instead of drive through, eliminating the need of raising the mechanism above the turtlebot level. This will also remove the need of raising the Cart by 1 layer to reduce fall damage on the can. In the future we might want to redesign the Dispenser to be more suitable for working with the Dispenser's docking mechanism. The main change will be to reduce the height of the dispenser so that we need not to worry about the fall damage either on the can or the can holder when the can unloads from the Dispenser.

Tolerance between the 2 Acrylic UpperWalls

Another improvement that can be made is to increase the space between the dispenser UpperWalls which the can resides in. We did not consider the tolerance between the can and the acrylic which resulted in the scenarios where the can could be stuck between the walls. We eventually had to put a small piece of aluminium perforated sheet between the 2 walls to make some space which allowed the can to drop without any difficulty. In the future, we can either expand the width of the space between the walls by shifting both walls more to each side or we

can improve the design of the Dispenser by accounting for a greater amount of tolerance at this crucial area of the system.

We also increased the height of the cart to minimise the falling impact; we needed to make sure that the whole cart is high enough such that the can's falling impact would not shake the whole Turtlebot, but also not too high such that it will obstruct the LiDAR sensor.

Height of the Cart (Can Holder)
To increase the height of the cart, we lifted the whole cart by one waffle layer together with several M3x35mm spacers. Hence, the first layer is connected to the first layer of the Turtlebot and the second layer is connected to the Turtlebot second layer. However, the need to do these height adjustments can be avoided if we simply reduce the height of the dispenser since the falling impact would not cause too many problems.

Can Position inside the Can Holder
The design of our Dispenser and Can Holder are both designed to work with a horizontally positioned can. There were not any problems with the dispenser since the can was already positioned horizontally between the walls. However, for the can holder, we needed to make sure it was aligned properly in order to drop it precisely such that the can stays in a horizontal position inside the cart. During our run, there were several instances when the can stood vertically inside the cart, obstructing the forward view of the LiDAR sensor.

Hence, the solution revolves around making sure that the can drops and stays vertically inside the cart or to make it such that however the dropping position of the can is, the forward view of the LiDAR sensor would not be obstructed. Removing the whole first floor of the Cart could already solve this problem, since we would like to also decrease the dispenser height. Other than that, we might want to make the Can Holder walls to be more of a slope rather than straight, vertical walls to ensure that the can always drops and stays horizontally inside the Can Holder.

12.2 Electrical

Wiring and Cable Management for the Electronics in the Dispenser
Despite a successful closed circuit connection with the use of Arduino Uno, jumper wires and breadboard to connect the ultrasonic sensor and servo motors together, we could have improved on grouping the wires leading to each electronic component by binding them together with a labelled tape. This can help to reduce confusion and make it easier to troubleshoot the system or any electronic components.

Overusing of core components
In the preparation for the final run, we ran the Turtlebot for multiple hours consecutively, swapping out batteries whenever one battery was out. After prolonged use of the Turtlebot, we were not able to ssh into the Turtlebot for an extended period of time. We soon came to realise that this was due to the overuse of Turtlebot, which made the RPi unable to start up properly

and read the SD card inserted into it. As a result, we were unable to run many tests and were not able to calibrate the waypoints accurately. From this experience, we learnt the importance of preparing backup hardware in the event of a worst case scenario like this.

12.3 Software

Dispenser Sensitivity

The Dispenser utilises an ultrasonic sensor to detect the incoming Turtlebot, which will then send a signal to the Arduino board to actuate the servo motors and drop the can after a few seconds delay. This system has an obvious weakness that it will detect anything that appears in front of the ultrasonic sensor and think that it's the Turtlebot. In other words, the can will drop if any random object were to appear in front of the ultrasonic sensor. This means that the Dispenser needs to be isolated from any foreign object, which is not very likely in a busy environment such as a restaurant. The suggested changes would be to use NFC tag or even a camera with computer vision to allow the can to drop only when the real Turtlebot is identified, instead of just any random object in close proximity.

Dispenser's Servo Motor Setup Position

The use of the ultrasonic sensor also provides us with another problem. Our robot uses a waypoint algorithm to navigate through the map and we set a '*Point of Return'* or the initial starting point. We set the starting point of the Turtlebot exactly under the dispenser, so that in the following runs the can may be automatically loaded into the Cart when the Turtlebot comes back to that initial point. However, during the initial setup when we were setting up the first waypoint, the Dispenser needed to be turned off so that it would not prematurely drop the can before the run started. This might not be that bothersome if the successive runs were all successful runs. However, whenever we decided to abort a turn, we needed to do the setup all over again, turning the Dispenser off and on. This might be more of a software issue where we might need to change the initial position of the waypoint.

Waypoint Algorithm for Docking

During the practice and final runs, the waypoint algorithm proved itself to be effective in helping the Turtlebot travel from start to destination by following the set of waypoints given. However, the waypoint algorithm lacked precision, which led to an inaccurate docking position. The Turtlebot always stopped near, but not exactly at the exact docking position in order for the can to drop into the can holder smoothly. We attempted to mitigate this by creating an additional set of waypoints within the dispenser to enable the Turtlebot to slowly adjust itself into the appropriate position. While this did help resolve some inaccuracies, the results were inconsistent and was not an effective solution.

Given more time, an improvement we could have made would have been to create a docking algorithm using the distances measured from the LiDAR sensor. The precision of LiDAR measurements of ~1 cm uncertainty would have been a more accurate solution as compared to the ~10 cm uncertainties we were getting. By ensuring that the Turtlebot is kept within a certain distance from the left wall whenever it is moving into a docking position, and ensuring that the

Turtlebot stops within a specified distance from the front wall, we can ensure that the Turtlebot is almost in the exact position required.

13. Conclusion

In conclusion, the robotic delivery system we designed and built was successful in achieving its intended purpose of autonomously delivering canned drinks from a fixed dispenser to designated tables on a restaurant-like map and this report details the entire system design process and sub-systems integration for our EG2310 robotic delivery system.

We followed the System Design V-Model framework, and integrated four different subsystems: Dispenser Storage, Dispatching, Turtlebot Load-sensing Holder, and Turtlebot Navigation, together with the navigation abilities of the waypoints algorithm.

During the actual navigation run to the six tables, the robot successfully delivered the canned drink to each table, despite encountering several hiccups due to bad calibration in an unfamiliar terrain. These hiccups were caused by a lack of testing and calibration on the actual map, as there were unexpected mechanical issues with the Turtlebot 12 hours before our run that we only managed to fix half an hour before our time slot. This experience has taught us the importance of testing and calibration in a real-world environment. Our lack of testing on the actual map led to issues during the navigation run that could have been avoided with more preparation time. We also learned the importance of being flexible and adaptable, as we had to quickly fix our mechanical issues with the limited time before our time slot. Despite everything, we still deemed our navigation run to be a successful one as we were able to complete all the missions at hand in delivering the cans to all six tables.

Moving forward, there are also several improvements that we could make to the whole robotic delivery system in terms of its mechanical, electrical and software components as mentioned in the report in order to enhance its efficiency and robustness. Finally, the completion of this report entails the end of our journey as a team and we are thankful for the memories made and friendships forged throughout this endearing journey. We would also like to thank our TA Jared Oong for his unwavering support and patience as well as the whole teaching team of EG2310 including Ms Annie for their guidance and inspiration which we will bring with us to our future endeavours.