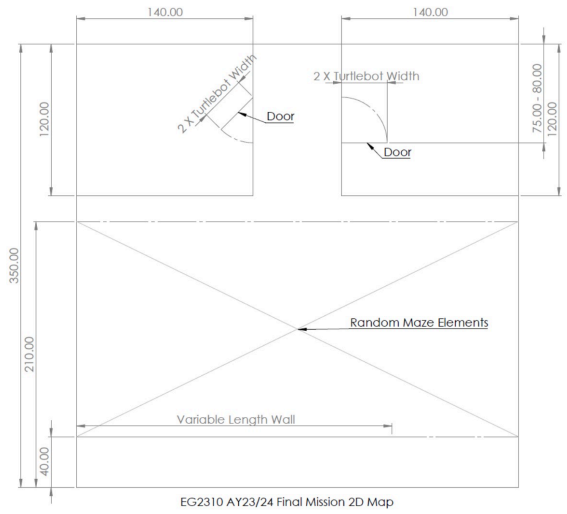
EG2310 Group 9 G2 Report

Neha Rajan	A0284755N
Felix Then Kai Fan	A0276165X
Hoo Zu Yang	A0255798E
John Yeap Teik Jien	A0288226W

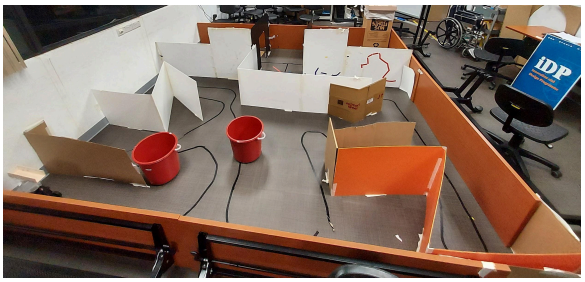
Table of Contents

1. Introduction

1.1 Problem Statement



EG2310 AY23/24 Final Mission 2D Map



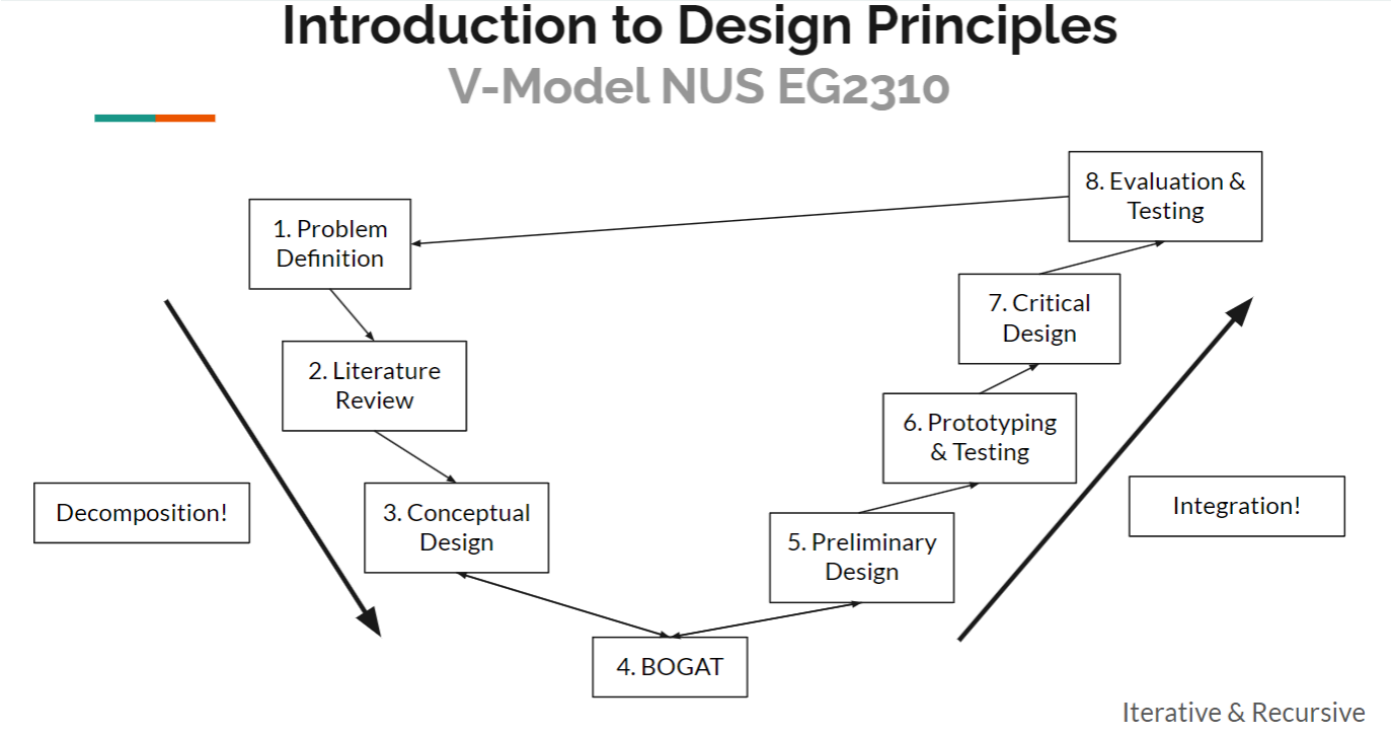
The mission is to modify a Turtlebot3 Burger robot to successfully navigate a 3-part obstacle course:

1. The Maze
   Commencing at the start zone, the robot must navigate a random maze and develop a full SLAM map displaying all elements of the given maze.

2. The Lift Door
   The robot will then need to make an HTTP call to a server and obtain the ID of the unlocked door. It must then proceed to open the door and travel through it.

3. The Bucket
   Finally, the robot must launch 5 ping-pong balls (stored on its body) into a bucket placed anywhere in the room it has entered.

The system design process utilises the following modified EG2310 V-model as shown below:



Figures 1.3 Systems engineering V diagram

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

1. V-Model parts 1-7 are described in Section1-7 respectively
2. Critical Design is described in detail in over Section 7-11
3. Evaluation and Testing is shown in Section 8
4. A list of possible future works is shown in Section 12

1.2 Requirement Specifications:

Requirements:

1. Robot must move autonomously
2. Robot must navigate an unknown maze
3. Robot must generate a full SLAM map of the maze
4. Robot must make a HTTP call to a server
5. Robot must receive the ID of the unlocked door from the response and navigate to the correct door

6. Robot must locate a bucket inside the room
7. Robot must launch 5 ping pong balls into the bucket

Autonomous Exploration for Mapping:

The robot must be able to navigate throughout the entire mission, consisting of its 3 main parts, autonomously. Once it commences its journey, no human intervention should take place. Using LiDAR we must procedurally create a Simultaneous Localisation and Mapping (SLAM) map as the robot. We need a navigation algorithm that will allow the robot to avoid crashing into walls or other maze elements. The algorithm should systematically take the robot closer to the lift doors while preventing backtracking and exploring unmapped areas.

HTTP- Based Communication

After mapping the maze elements, the robot must navigate to the doors. The system's computer must then be able to send a HTTP call to a server. It should also be able to receive a response and comprehend it to make further decisions on how to advance in the mission. Arriving at the doors the HTTP call must be made to the server. Subsequently, the server's response must be parsed and used to initiate movement towards and through the unlocked door.

Launching the Balls

Once inside the room, the robot must locate the bucket and move towards it, without touching or moving the bucket. Upon reaching, the balls will be deposited in the bucket and the mission will be completed.

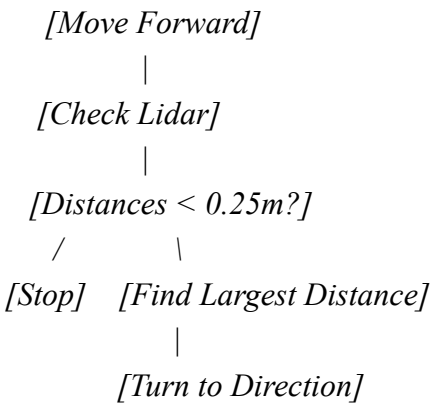
2. Literature Review

This section entails preliminary research conducted using various sources. The purpose of this research is to identify various possible methods to address the design challenges posed by each phase of the mission.

2.1 Exploration Algorithms

1. **Basic Obstacle-Avoiding Algorithm (*r2auto_nav*)**

   This algorithm relies on the LiDAR to allow us to use occupancy grid data to encode the SLAM map into an array format. 2D arrays can then be used to calculate the coordinate of a goal in the map. The LiDAR will also allow us to avoid obstacles along the way. Here is a basic control loop of the algorithm:



2. **Marker-Follower Algorithm**

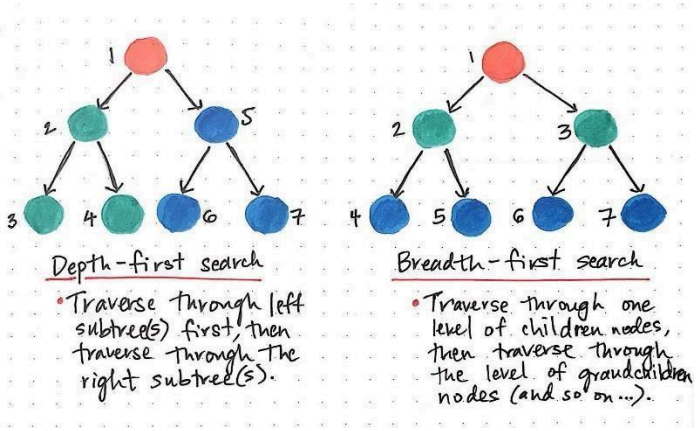
   Markers are coded with information about how to move. They may either repel or attract the robot away from/towards points of interest. Here is the basic pseudocode for the algorithm, which may be used with a variety of markers such as colour markers, NFC tags and ArUco markers:

   [Detect marker?] -> [Update position] -> [handle Event] -> [Scan for next Marker]

   When a marker is detected, for example, an event will be executed, such as turning right, moving straight, making a HTTP call and so on.

3. **Simple Graph Search Algorithm**

   This equips the robot with logic to navigate appropriately and do a complete mapping. There are two ways of searching: depth-first or breadth-first.



4. **Complex Pathfinding Algorithms**

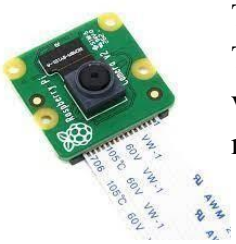
Our team considered using A* and Djikstra's path planning algorithms as well. These are graph search algorithms which find the optimal path to a goal coordinate on the occupancy grid. Dijkstra's algorithm works on the basis that the map is unknown, resulting in an uninformed search where no information about the environment is known. Meanwhile, A* is a heuristic search algorithm, which means that it factors in other costs such as distances to the goal in order to optimise the time complexity and search for paths in the direction of the goal rather than randomly.

2.2 Sensors and markers

1. **Light Detection and Ranging (LiDAR)**

The Turtlebot3 comes with a 360-degree Laser Distance Sensor LDS-02 (a LiDAR module), which emits lasers to measure distances of objects to itself, and allows us to create a 2D point map of the environment. It has a detection distance of 16 cm to 8 m. This map can be visualised in the RViz application.

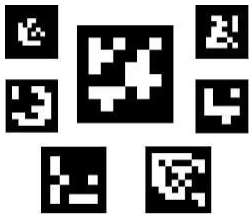
2. **Raspberry Pi Camera Module**



The camera module is used to capture video or image data and transfer it to the Raspberry Pi. This camera can be integrated with OpenCV and AruCo markers. OpenCV is a computer vision library which provides the framework for visual object recognition, and colour or marker recognition capabilities.

3. **ArUco Markers**

   These have unique IDs that can be used for identification by computer vision software and cameras. These markers can each encode information about the next actions the robot will take after detecting them.



4. **Near-Field Communication (NFC)**

   NFC is a short-range wireless RFID technology that transmits an NFC signal when an NFC reader is used to trigger an NFC tag which is placed on an object nearby. While the tag is active, the reader can detect an active tag, and the tag will become inactive again when the reader moves a distance away from it (the tags are passive components).

5. **Line Sensors (infrared transmitter and receiver)**

   The path the robot needs to follow is drawn as a line, and a line sensor is added to the base of the robot to follow the path. The sensor will transmit IR signals. Depending on the level of IR that is received after it has been reflected from a surface, the sensor will determine if the robot is following a black line. This is because of the difference in the amount of light absorbed by white and black.

2.3 Launcher subsystem

2.3.1 Launcher Mechanism

1. **Flywheel Launcher**

A flywheel is a launching mechanism that provides quick firing rate and great adjustability. Its range can be adjusted by altering the turning rate of the flywheel. By attaching a servo to the launcher, the range and height of the projectile can be adjusted to suit different conditions.

2. **Solenoid Launcher**

   The electromagnetic forces the armature away from the stationary core. This compresses the spring. When the current stops flowing, the armature is released and the ball will be launched.

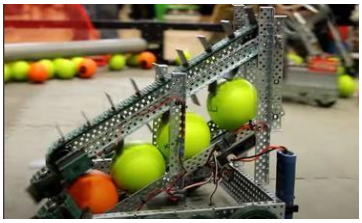
3. **Catapult**

   A catapult relies on elastic energy. The ball is first loaded onto a container at the tip of the catapult arm. Then, the catapult arm is released and the ball gains kinetic energy.

4. **Servo Motor**

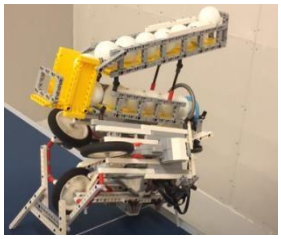
   A servo motor is a rotary or linear actuator that allows for precise control of angular or linear position, velocity, and acceleration in a mechanical system. The SG90 Servo motor is tiny and lightweight with high output power.

2.3.2 Ball Reload Mechanisms



1. **Conveyor Belt**

   The balls are loaded onto the belt and lined up next to the launcher. The belt moves the ball onto the launcher after the ball is launched.



2. **Incline Plane**

   Balls are held in place on an incline plane by a stopper. Once a ball is launched, the stopper allows the next ball to roll into the launcher.

3. **Turning Disk**

   This mechanism consists of two plates and a servo. The upper plate holds the 5 balls with one empty slot. The servo turns the upper plate such that a ball drops into the shooting mechanism.

3. Concepts of Design and Operations

We divided the operation into 3 major phases:

1. **Maze exploration** phase: The robot needs to navigate through the map without user input
2. **Door locating** phase: The robot must navigate to the door after the maze is sufficiently mapped and make a HTTP request to open the doors
3. **Ball delivery** phase: The robot must enter the unlocked rooms and locate the bucket, then move to it and deposit the ping-pong balls into the bucket

3.1 Phase 1 (Maze exploration)

Objective:

The Turtlebot is to navigate through the maze and generate a SLAM map of the environment

Description:

The Turtlebot will search for the nearest unexplored area. Once the coordinate of the frontier is obtained the robot will move to the goal while avoiding obstacles. The aim is not to get stuck in loops.

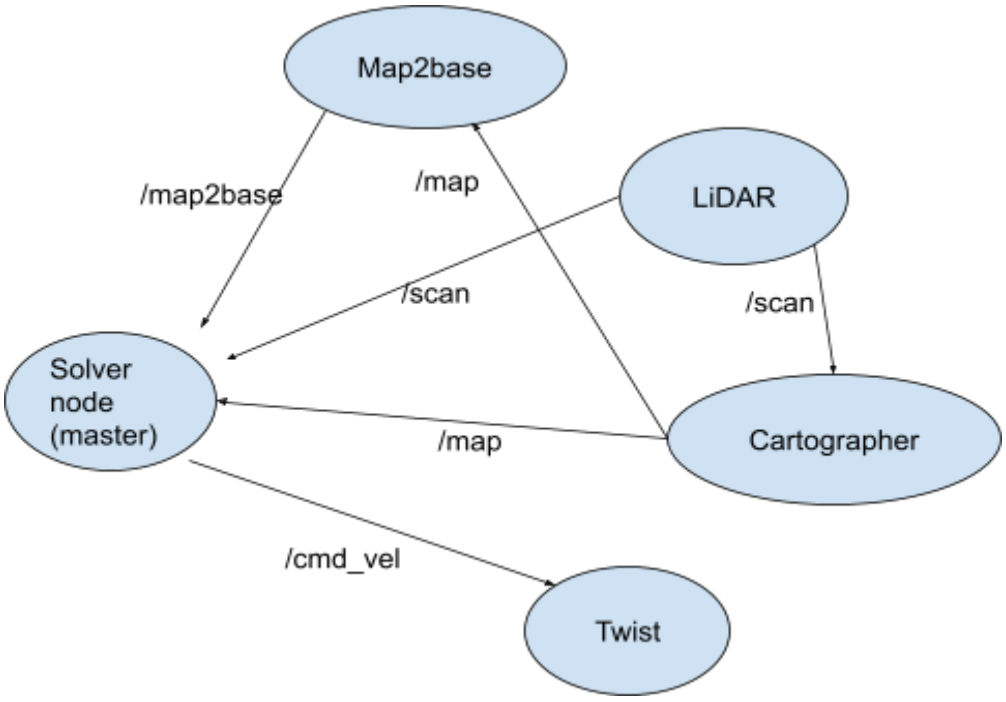


Figure 3.1 ROS2 pubsub structure

3.2 Phase 2 (Locating the door)

Objective:

The Turtlebot needs to make a **HTTP call** to the server to **obtain the door number**. The bot navigates to the door specified and pushes the door to enter the room

Description:

Before the start of the run, we will indicate the **IP** of the **ESP32** in our code, which then the raspberry pi will run the code and make a **HTTP call** to the server on ESP32 to obtain the door number. Next the robot will rotate to the position of the stated door which we have indicated in our code before the run.

3.3 Phase 3 (Delivering the balls)

Objective:

The Turtlebot needs to move at the direction of the bucket and stop in front of the bucket without hitting it.

Description:

When the Turtlebot enters the room, it will locate the obstacle with the **shortest distance** from it which is the bucket. The bot needs to run a code to **actuate** the **trapdoor** attached to a servo, which **allows the ball in the tunnel to fall inside the bucket.**

4. Our BOGAT (Bunch of Guys Around Table)

This is an evaluation of options and the weighing of the pros and cons of each to make a final design decision based on the requirements of our mission.

4.1 Navigation

The robot will search for the nearest frontier point with breadth first search implemented on the occupancy grid map. Once a frontier is selected, to navigate there, we admit a simple obstacle avoidance algorithm would seem too simple. *However, for the relatively simple maze requirements with large spaces between walls, it was deemed sufficient*. In contrast, the graph search algorithms for pathfinding were deemed too excessive for this maze task. Factoring in the short time span of 6 weeks for developing a fully functional navigational algorithm based on A*, for example, would require implementation of the search algorithms itself, cost maps, and a pursuit algorithm to publish commands to the robot to follow the path generated. We believed that a simple straight line movement was more than sufficient for a simple maze.

In addition, a lot of teams, including us, planned to do marker based navigation. However, marker based navigation is quite unreliable. For example, using NFC tags to program the robot's next movement every step would be inaccurate because of the small errors in turning and moving straight which may cause the robot to overshoot the next marker it needs to find, or not be able to detect it as errors compound throughout the maze. We seriously considered using an infrared emitter and sensor to implement line following around the maze. However, we felt that it was *not in the spirit of the mission* and chose to abandon that idea in favour of more autonomous solutions.

In the end, we chose to utilise **only the LiDAR module for autonomous exploration** with our algorithm, which would **travel in a straight line to goal and avoid obstacles along the way.** After avoiding, it would then reorient itself to the goal and attempt to make its way there again, while repeating obstacle avoidance.

4.2 Sensors for navigation and bucket detection

We require no auxiliary sensors for autonomous exploration of the maze, but in the early stages, because of a lack of visual knowledge of how the maze would look exactly, we planned to use line following algorithms to get from the exit of the maze to the door. **However, we also reasoned that our navigation algorithm should be able to go to the door, by setting the known location of the middle of the corridor in terms of coordinates in our navigation algorithm selected above.** So, we planned not to locate and navigate to the door with markers or extra sensors.

In order to detect the bucket, we planned to use integration of **OpenCV with the Raspberry Pi camera module** to detect the bucket's red colour. This was deemed to be the most reliable way of navigating to the

bucket, as it could be anywhere in the room, barring line following methods. This would be comparatively much easier than using ArUco markers with the extra integration required to set up and encode the markers into the software.

Components	**Description**	**Decision**
LiDAR	Sensitive at mid- to high- distances. Allows Turtlebot to 'see' all obstacles in a circular radius at one time	Yes
Camera module	Detects ArUco markers from relatively far away. Integration with ArUco markers is more complex than NFC readers.	Yes
ArUco markers	Can be detected by computer vision using a camera module, can be placed in the maze to indicate important locations or signify certain meanings.	No
Line sensor	Using a PID control loop to guide the robot to the correct door with a line detector is easy to implement. However, this can be troublesome as well, since there are time limits to lay the lines/tape. An algorithm would be able to perform tasks better with precise calculation.	No
NFC reader and tags	Enables communication between two electronic devices over short distances. It is simple to use, but requires the Turtlebot to be near the tag.	No

4.3 Launcher

There is a high amount of complexity in a flywheel launcher. Also even though we will be able to launch the ball with a great amount of force, it is too excessive for this mission. A catapult mechanism is relatively simpler but comes with drawbacks. The large compartment required to hold 5 balls means that we would have to place the catapult container behind the robot, and this will obstruct the LiDAR. Furthermore, the catapult arm would have to be lengthened, increasing space taken. Because of these reasons, we decided to go with a **solenoid plunger launcher**. It is easier to vary the robot's position to be further or nearer to the bucket in order to compensate for the lack of launching force, rather than to tune the launcher itself. The solenoid launcher would take up less horizontal space, and allow for the robot to turn better in tight spaces without hitting the walls. We made this decision based on *how compact the ball delivery mechanism should be.*

Type of launcher	**Analysis**	**Decision**
Flywheel	Able to launch the ball with great amount of force, however the system has high complexity	No
Catapult		

Solenoid plunger	The catapult arm throws the ball to a greater height through a lengthened arm, however it might require a very powerful servo	No
Rubber Band Launcher	Able to launch the ball with good amount of force and the component is readily available, but might draw a lot of power	No
Elevated Ball Dropper Tunnel	Able to launch with great amount of force, but requires precisely manufactured moving parts	Yes
	The simplest and easiest way to deliver the balls into the bucket, but might make the robot and centre of gravity too high causing it to be unstable	No

Types of reloading mechanism

Conveyor Belt	It is a good system to transport the balls, but requires a lot of moving parts that makes it complicated to implement it within a tiny Turtlebot	No
Incline Plane/tunnel	It does not require any electricity to run and relies only on the gravitational force, however it will take up more horizontal space	Yes
Turning disk	It can hold the balls within place and fits nicely inside a Turtlebot due to it being in a circle shape	No

Problems	**Solution A**	**Solution B**	**Decision**
Position of the launcher	**At the top of the robot:** Easier to get the ball into the pail. However, the lidar camera might be blocked.	**Beside the robot:** May cause instability since the robot will be heavier on one side.	Solution B
Materials	**Plastic:** Lighter and cheaper cost	**Steel:** More durable and sturdy but also heavier and costly	Plastic

5. Preliminary Design

5.1 Autonomous Navigation

For autonomous navigation, the robot will utilise algorithms such as wall following and obstacle avoidance to navigate through the maze. The combination of these both algorithms allow the robot to navigate through the maze and map the maze simultaneously.

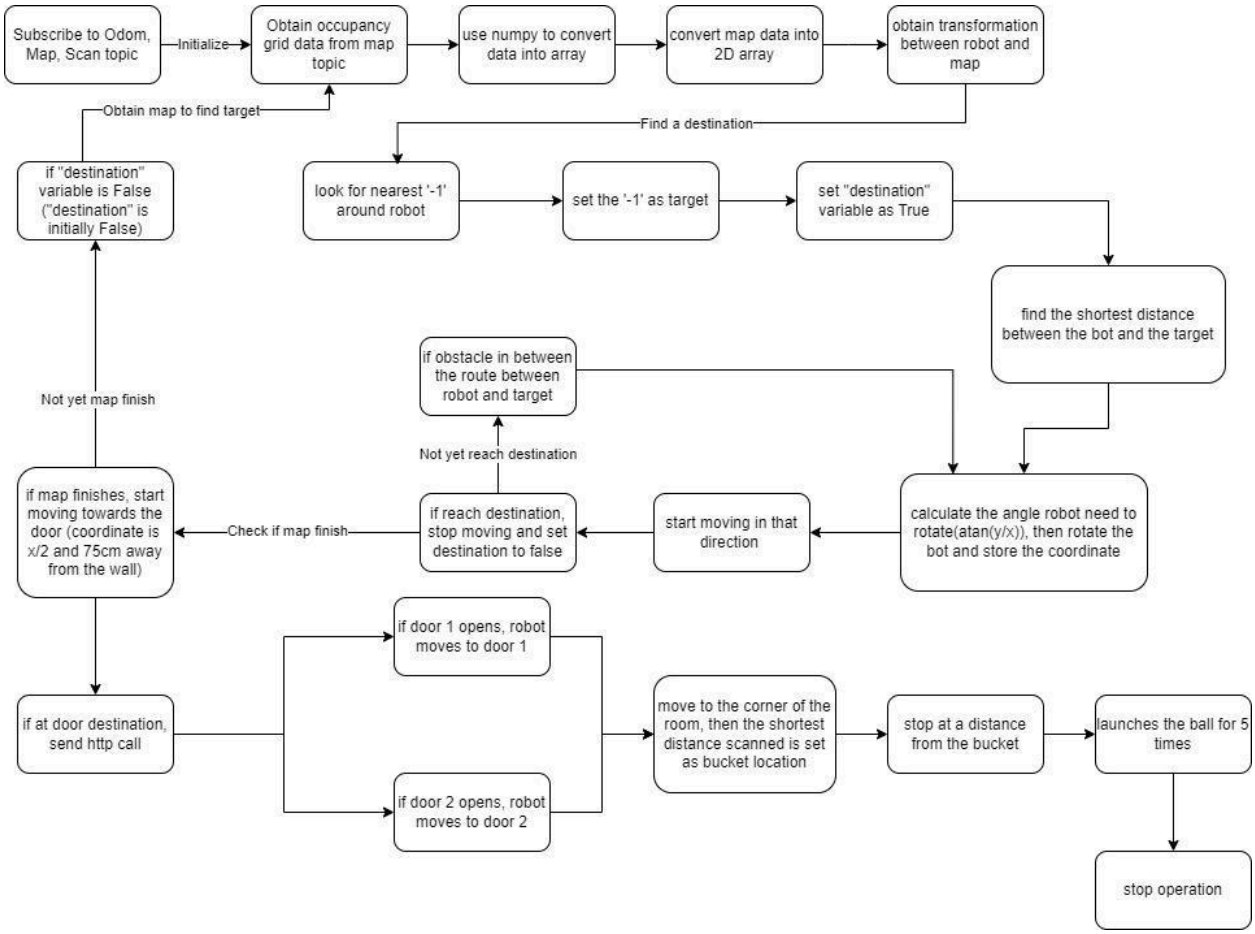
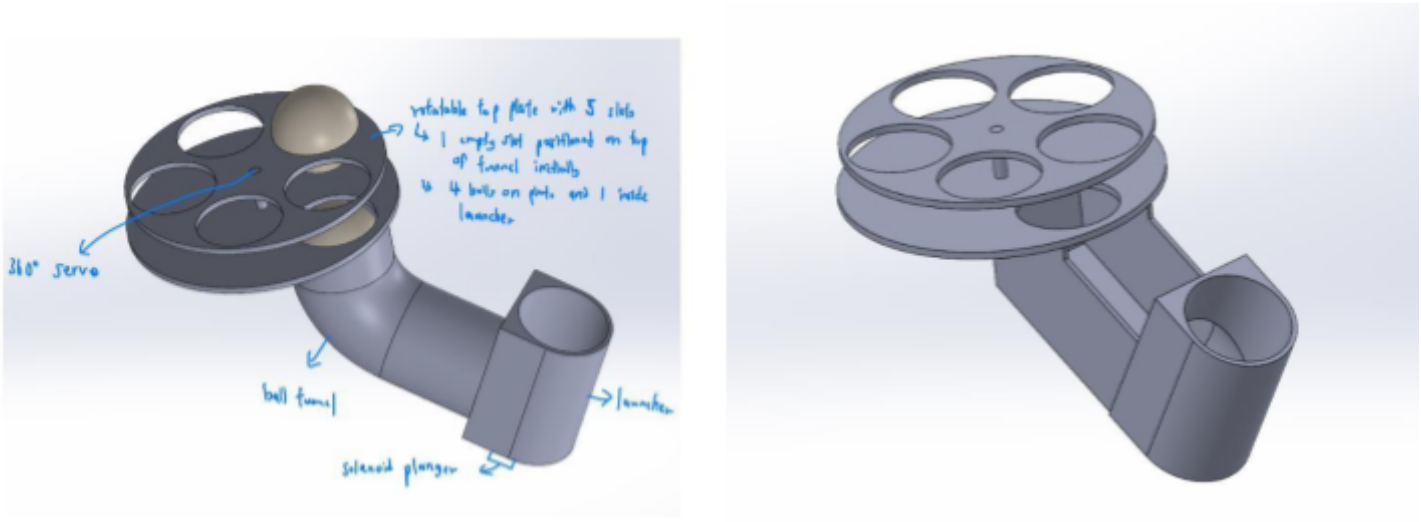


Figure 5.1 PDR flowchart

5.2 Payload Mechanism

We proposed to use a solenoid actuator to launch the ping pong balls in the bucket. We chose it as it is powerful and consistent while also being able to be controlled electronically. The 5 balls will be stored in a rotating plate and will be released in the launcher system one by one.



Version 1 Version 2 with half-open tunnel

Figure 5.2: CAD of reloading mechanism and launcher

5.3 Electrical Subsystem

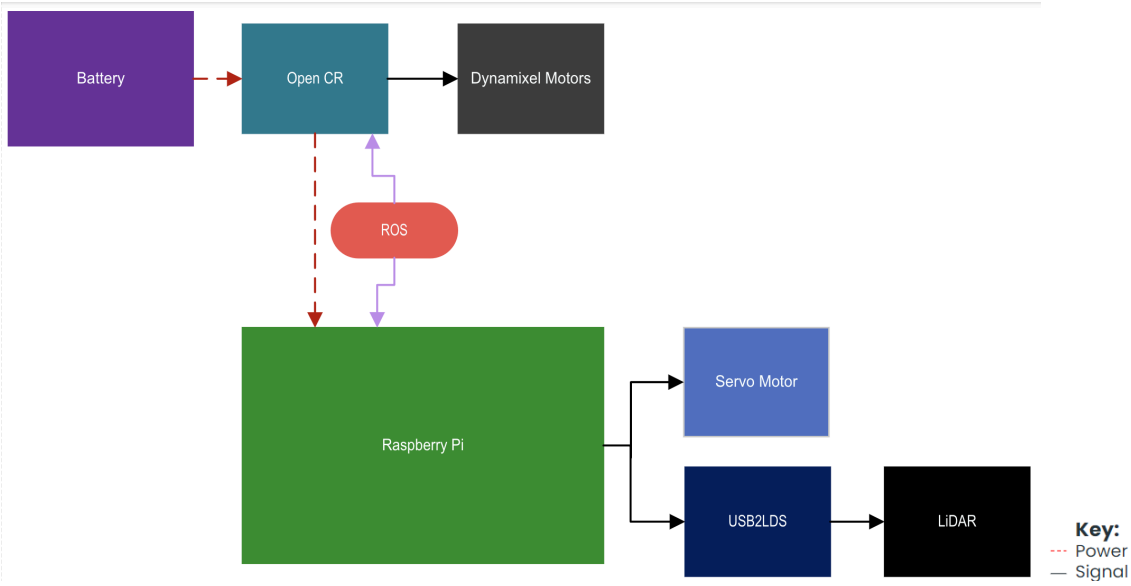


Figure 5.3 Functional Block Diagram

6. Prototyping and testing

6.1 Navigation Algorithm

To develop the navigation algorithm we first broke down the navigation requirements into different smaller scripts which we could test individually. This includes scripts for obtaining the occupancy grid, simplifying the data for our algorithm, searching for unmapped frontier points in the grid and displaying them on a Matplotlib graphic, moving the robot to a coordinate in the map frame and basic obstacle avoidance.

We faced a few challenges when developing our algorithm:

1. Due to the **Odom frame drifting** overtime in the map, we could not use the frame to get the position of the robot in the map. Instead we had to rely on a separate node to publish the transform of the base link of the robot to the global map frame, so we can measure the position of the robot with a fixed coordinate frame.
2. In RViz we could see some faint 'ghost' marks of LiDAR scans which were penetrating the maze wall. This phenomenon was especially prevalent when the **LiDAR 'peeks' through cracks in the maze walls** due to imperfections. This meant that the LiDAR was setting a low probability of occupancy for some space that it should not be able to see. Over time, if the robot remains in the same position, the confidence that the area outside the maze is unoccupied increases, until the Cartographer node assigns these ghost scans as not being occupied. This means that, overtime, our frontier search algorithm might select these frontiers which may be totally inaccessible as the next goal. We countered this by implementing two things: first, when we obtain the occupancy grid data, we use thresholds to bump the values of lowly confident … Another method was to store grid coordinates that the robot had already tried to move to, and when searching for new frontiers, it was not allowed to select from this set of coordinates.
3. There was a delicate balance between fine tuning the algorithm to make sure that the robot could clear walls in the obstacle avoidance phase. In the early stages, the robot would We also faced a lot of problems with the robot stopping and starting when trying to navigate through smaller gaps than usual. It took a while before we realised that there is also a balance when setting the angles which the LiDAR checks for obstacles while the robot is moving. **If the angle range to be searched is too wide, it would easily be stopped if there were walls on the sides of it**. **However, if it were too small, it is too blind to see walls hit the robot's edge.** Initially, we erred on the side of making the angles quite large, but in the end, the Turtlebot can avoid obstacles completely by just checking the 10 angles in front of it to the left and right every moment while moving. This was not intuitive for us, since we wanted to make sure the robot did not crash. However, this accomplished the goal of making sure the robot would not stop just because there were obstacles near it, but only if they directly impeded its path.
4. Due to the inaccuracies in the LiDAR, it was quite hard for us to implement a wall following algorithm. This is mainly because of the fact that the LiDAR **does not actually output 360 values** of distance for every angle. The amount of PID tuning for this part of the program alone

would be disproportionately high for the benefit we would receive, and so we did not include this functionality in our final navigation algorithm.

5. One problem that was very hard for us to solve was the fact that due to the LiDAR not being very good, **it could not detect walls when it was coming straight on (it is too thin apparently**). In fact, when the wall is parallel to an angle of our LiDAR scan, we can see it visibly disappear second by second on RViz since the robot cannot actually see it.

Some key algorithm design ideas we implemented are explained here:

1. The robot would also often get stuck in loops. It was only later that we would figure out that in a maze with right angled walls, there are three cases that we have to account for when avoiding obstacles and getting out of enclosures: coming at a wall at an angle, right-angled corners and deadends (walls on three sides).
2. We made one big change in the logic of how the robot moved. In the beginning, we programmed the robot to move a preprogrammed time, then stop and avoid obstacles for a preprogrammed time, then search for a new frontier goal, then repeat. This seemed to work in theory; the robot would move step by step, still be able to make its way to the goal in a systematic way. However, this resulted in erratic start-stop motion especially in places where there were walls on both sides of the robot. We kept it simple by cutting out the intermediate motion, and instead only initiated the next action whenever the robot actually arrives at the goal position, or detects an obstacle or wall. This caused it to have a motion similar to that of an arcade hockey puck which bounces around the map. The algorithm works very well for such a simple one when there are pretty sparse walls which are far apart, however, we noticed that it suffers in tightly spaced maps or those with non right angled walls.
3. When searching for frontier points, we made sure to check that **a)** points are more than 2 grid cells away from any occupied grid cells so that the points are more accessible and not touching the wall, and **b)** that there are at least 2 other unexplored grid cells in its immediate surroundings of 8 cells. This is to ensure that the robot is heading to a place that has a higher likelihood of having a cluster of unexplored cells, and not just a glitch in the map.

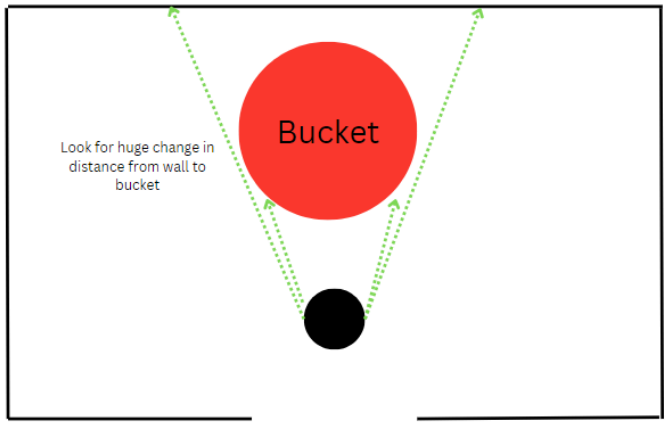
Some things we tried that did not make the final design were:

Breadth-First Search Algorithm

```
205         def finder(mapData,pos):#find nearby -1
206             pot_Target = []#store the neighbouring explored cells to check later
207             visited = {}#store the coordinates that is already visited
208             print(pos)
209             pot_Target.append(list(pos))
210             while pot_Target:#while there is still neighbouring cells to visit
211                 print(pot_Target)
212                 now = pot_Target[0]
213 
214                 if mapData[now[0]][now[1]] == -1:#if the current cell is unexplored
215                     return now
216 
217                 if now[0]+1<iwidth:
218                     if mapData[now[0]+1][now[1]] == 0 and (now[0]+1,now[1]) not in visited:
219                         visited[(now[0]+1,now[1])]=1
220                         pot_Target.append([now[0]+1,now[1]])#adding the neigbouring cells to the list to check later
221 
222                 if now[0]-1>=0:
223                     if mapData[now[0]-1][now[1]] == 0 and (now[0]-1,now[1]) not in visited:
224                         visited[(now[0]-1,now[1])]=1
225                         pot_Target.append([now[0]+1,now[1]])
226 
227                 if now[1]+1<iheight:
228                     if mapData[now[0]][now[1]+1] == 0 and (now[0],now[1]+1) not in visited:
229                         visited[(now[0],now[1]+1)]=1
230                         pot_Target.append([now[0],now[1]+1])
231 
232                 if now[1]-1>=0:
233                     if mapData[now[0]][now[1]-1] == 0 and (now[0],now[1]-1) not in visited:
234                         visited[(now[0],now[1]-1)]=1
235                         pot_Target.append([now[0],now[1]-1])
236 
237                 pot_Target.pop(0)#remove cells from the neghbouring cells list since it is already visited
238 
239             return None
240 
241         currentPos = pos
242         target = finder(mapData,pos)
243         moving = True
244         print(target)
```

We tried to implement a breadth-first search algorithm for searching the nearest “-1” or the unexplored cell around the robot. This function will start looking through the occupancy grid data array from the robot’s position (the variable “pos”) and check the surrounding 4 cells. If the cell is explored (shown as value “0” on the array), it will be added to the “pot_Target” list and considered as a potential target to be check afterwards. This while loop will continue to run until it finds an unexplored cell (“-1”) or there are no more potential targets that it can visit.

Finding Bucket



Turtlebot searching for bucket

```python
def find_bucket(self):
    rclpy.spin_once(self)
    time.sleep(1)
    print(self.laser_range)
    if self.laser_range.size != 0:

        left_ang = 0
        right_ang = 0
        big1 = 0
        big2 = 0
        for i in range(-89,91):
            ang = int(i/360*len(self.laser_range))
            first = self.laser_range[ang-1]
            sec = self.laser_range[ang]
            if not(math.isnan(sec)):
                if math.isnan(first):
                    if math.isnan(self.laser_range[ang-2]) or i-2<-90:
                        continue
                    else:
                        diff = abs(sec - self.laser_range[ang-2])
                else:
                    diff = abs(sec - first)
            else:
                continue
            if diff > big1 and diff > big2:
                left_ang = i
                big1 = diff#largest diff
            elif diff > big2:
                big2 = diff#2nd largest
                right_ang = i
            if left_ang>right_ang:
                left_ang,right_ang = right_ang,left_ang
        print(left_ang,right_ang)
        #lr2i = 0.5*(left_ang+right_ang)/len(self.laser_range)*360
        lr2i = 0.5*(left_ang+right_ang)
        if lr2i>180:
            lr2i -= 360
        elif lr2i<180 and lr2i >0:
            lr2i += 10 #10 degree offset

```

Initially, we were thinking of finding the bucket by searching for the biggest change in distance as we thought that the distance readings from the lidar would change drastically when it goes from the wall in the background to the bucket that is close to the robot. In essence, it looks for the tangent lines from the LiDAR to the bucket.

This code will calculate the difference in distance between 2 subsequent angles. It will first check if the readings from the lidar is NAN, then it does calculations when the readings are numerical values. Additionally, if the subsequent angle is “NAN”, it will check one angle next to it because we discovered that the lidar would often register a single “NAN” value within a group of numerical readings, so

we thought that checking one more angle next to it may help to overcome this issue. Lastly, the 2 biggest changes in distance are stored as “big1” and “big2” while the 2 angles associated with it are stored as “left_ang” and “right_ang”. Then, it would calculate the middle of the 2 angles which is interpreted as the centre of the bucket and it will be the direction the robot is rotating towards.

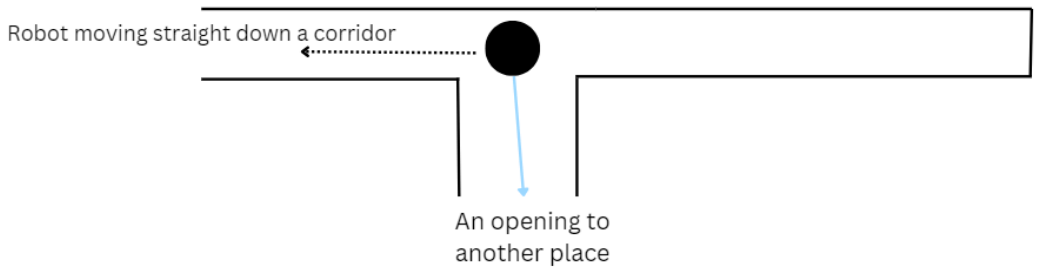
```
        rclpy.spin_once(self)
        print(lr2i)
        print(self.yaw)
        self.rotatebot(float(lr2i)-self.yaw)

        # start moving
        self.get_logger().info('Start moving')
        twist = Twist()
        twist.linear.x = 0.05
        twist.angular.z = 0.0
        # not sure if this is really necessary, but things seem to work more
        # reliably with this
        time.sleep(1)
        self.publisher_.publish(twist)

        rclpy.spin_once(self)
        while self.laser_range[0] > 0.16:
            rclpy.spin_once(self)

        #self.stopbot()
        twist = Twist()
        twist.linear.x = 0.0
        twist.angular.z = 0.0
        # not sure if this is really necessary, but things seem to work more
        # reliably with this

        self.publisher_.publish(twist)

        launch_ball()
```

Lastly, it will move towards the bucket and it will stop when the lidar readings in front of the robot is 0.16m. The robot will stop 0.16m in front of the bucket which is close enough to the bucket for the robot to drop the balls into it. **We needed to use a “while” loop after the movement of the robot was published because we wanted the robot to keep moving and the while loop would continuously check the distance in front of the robot until the loop breaks. Then, it will move on to the next part of the code which will tell the robot to stop.**

There were problems with using this method to find the bucket. Firstly, the lidar readings had too many “NAN” values which affected the calculations because we saw that there can be more than 2 consecutive ‘NAN” values and even as many as 5 consecutive “NAN” values. Therefore, many of the angles cannot be checked in the code because the code will ignore the angles that have “NAN” and the subsequent angles around the “NAN” as it is unable to calculate the change in distance with the “NAN” values.

Secondly, the “NAN” values always occur around the region where the lidar readings change from the wall in the background to the bucket. Therefore, the robot would often end up rotating in the wrong direction because it cannot check the region where the change from the wall to the bucket is. We concluded that the results from this method are highly inaccurate and decided to instead look for the smallest distance from -70 degrees to 70 degrees in front of the robot to locate the bucket. Our ability to search for the bucket when placed in the corners

Searching for "Openings"




```python
def listener_callback(self, msg):
    # create numpy array
    laser_range = np.array(msg.ranges)
    # replace 0's with nan
    laser_range[laser_range==0] = np.nan
    # find index with minimum value
    #lr2i = np.nanargmin(laser_range)

    self.laser_range = laser_range
    print(len(laser_range))
    print(laser_range)

    ang = int(45/360*len(laser_range))
    print(ang)
    if laser_range[ang]<0.7:
        print('obstacles')
    else:
        print('clear')
    '''
    '''
    for i in range(ang-10,ang+10):
        if not math.isnan(self.laser_range[ang]):
            if not math.isnan(self.laser_range[ang+1]):
                diff = abs(self.laser_range[ang] - self.laser_range[ang+1])
                if diff>0.4:
                    print(ang)
                    print('opening')

    time.sleep(2)
```

We have also tried to look for "openings" because we think that when a robot is moving down a corridor, there is a possibility that there will be an opening along the wall of the corridor which the robot can go into and explore other unmapped places.

In this code, **we experimented with checking the 45th degree angle of the robot for openings**, so the robot will check the region around that targeted angle (-10 from that angle until +10 from that angle) and calculate the change in distance. If the distance changes drastically (the change > 0.4 in our case), it will declare that an opening is found.

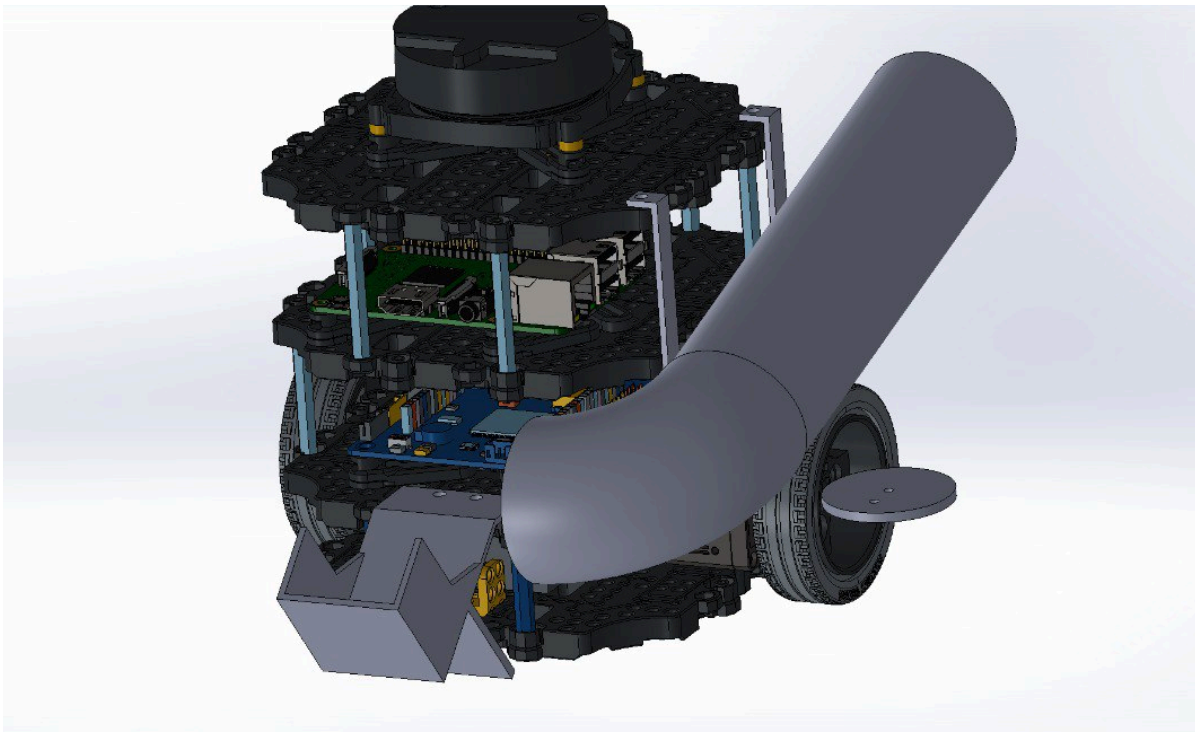
We discovered that this method did not work well because the lidar values are updated very slowly compared to the robot's movement, so it is not in sync with the robot moving. Thus, we often saw that after the robot move past the opening, it would still declare that an opening is found. Thus, we think it is too risky to implement this in our final design and decided to abandon this idea.

6.2 Payload mechanism

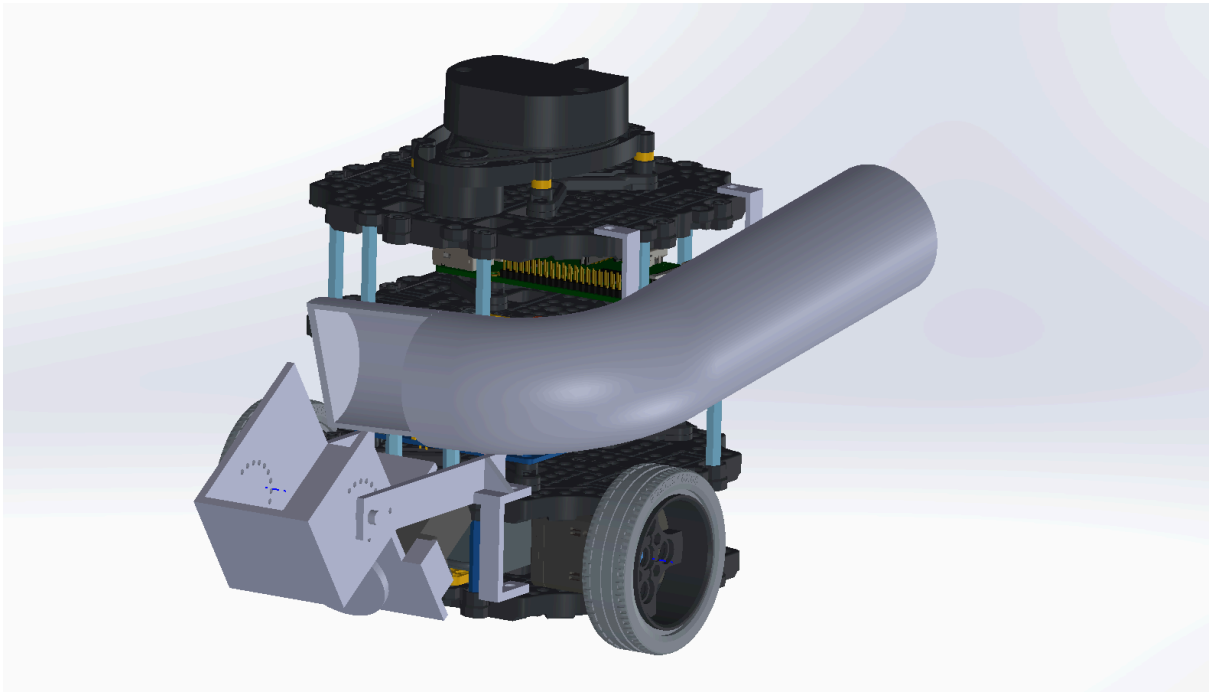
Based on our concept design of a **solenoid actuator ping pong ball launcher**, we conducted some testing with the solenoid launcher which was provided. The solenoid actuators are placed inside tunnels angled

at 45°. After conducting some testing with the provided solenoid actuator (5N) and a more powerful solenoid actuator sourced from the lab, we found out that it was **not powerful enough to launch the ping pong ball** to reach the height of the bucket.

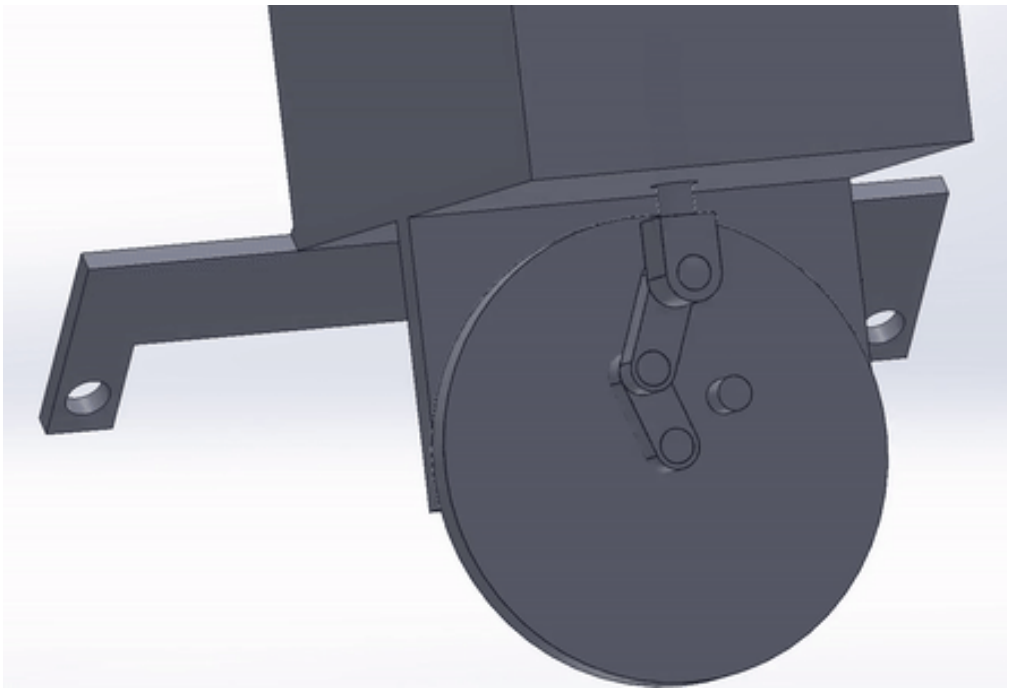
Hence, we decided to change our design of launcher to a **rubber band launcher mechanism** since it is **more powerful** to launch ping pong balls **without adding substantial weight and electricity consumption** to the robot. The initial design of the launcher is shown below. After some iterations and modifications, we added a **bracket** to mount the launcher main body as it helps to further extend the body from the Turtlebot and provide **angle adjustability** to the launcher.



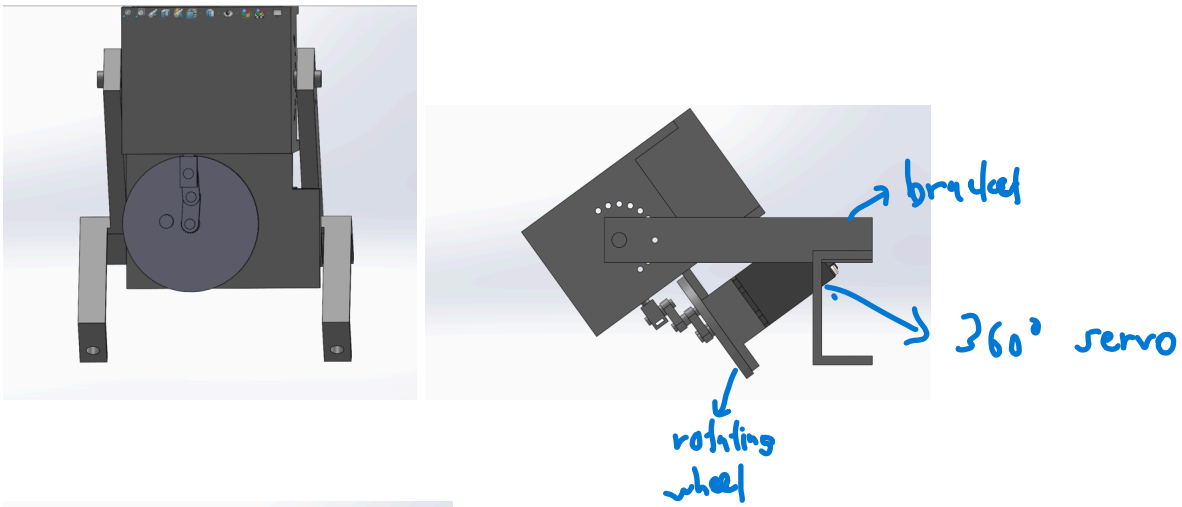
Initial design of rubber-band motorised launcher



Improved design - to ensure the balls land inside the launcher and provide adjustability



Picture showing the working mechanism of launcher



Pictures of rubber band launcher main body from multiple angles

The rubber band launcher consists of a **side tunnel** containing the 5 ping pong balls and a **launcher main body.** The launcher's main body consists of a **bracket**, **a rotating wheel** that serves to stretch the rubber band, a **continuous rotation servo (360° servo)** that attaches to the rotating wheel, joints and a rubber band. The joints are secured together using a 3mm stainless steel rod and retaining rings.

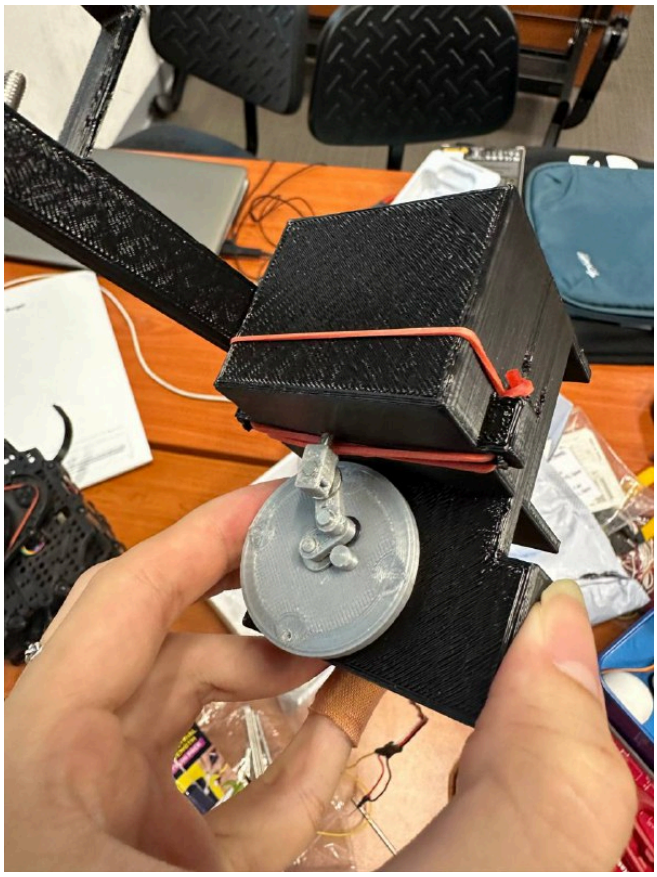
The video showing the working mechanism of the launcher is linked here (calculations for the payload can be found in the annex).

SM-S4303R specifications	
Measurement	Value
Operation Voltage	4.8-6.6 V
Torque	0.47 Nm (4.8 kgcm at 6 V)
Angular Velocity (Max.)	70 rpm
Gear Type	Metal
Rotational Degree	360°
Length	42 mm
Width	20.5 mm
Height	39.5 mm

Assembly List:

1. PVC Pipes [1 1/2 inch pvc pipe] x 1
2. 360 degree continuous rotation servo (screws included) x 1
3. 3D printed parts (bracket, launcher body, connectors, rotating plate)
4. Rubber band x 1
5. Washers x 2
6. Retaining rings (M3) x 5
7. Rods (M3) x 2
8. Nuts (M4, M2) x4, x2
9. Bolts (M4, M2) x4, x2

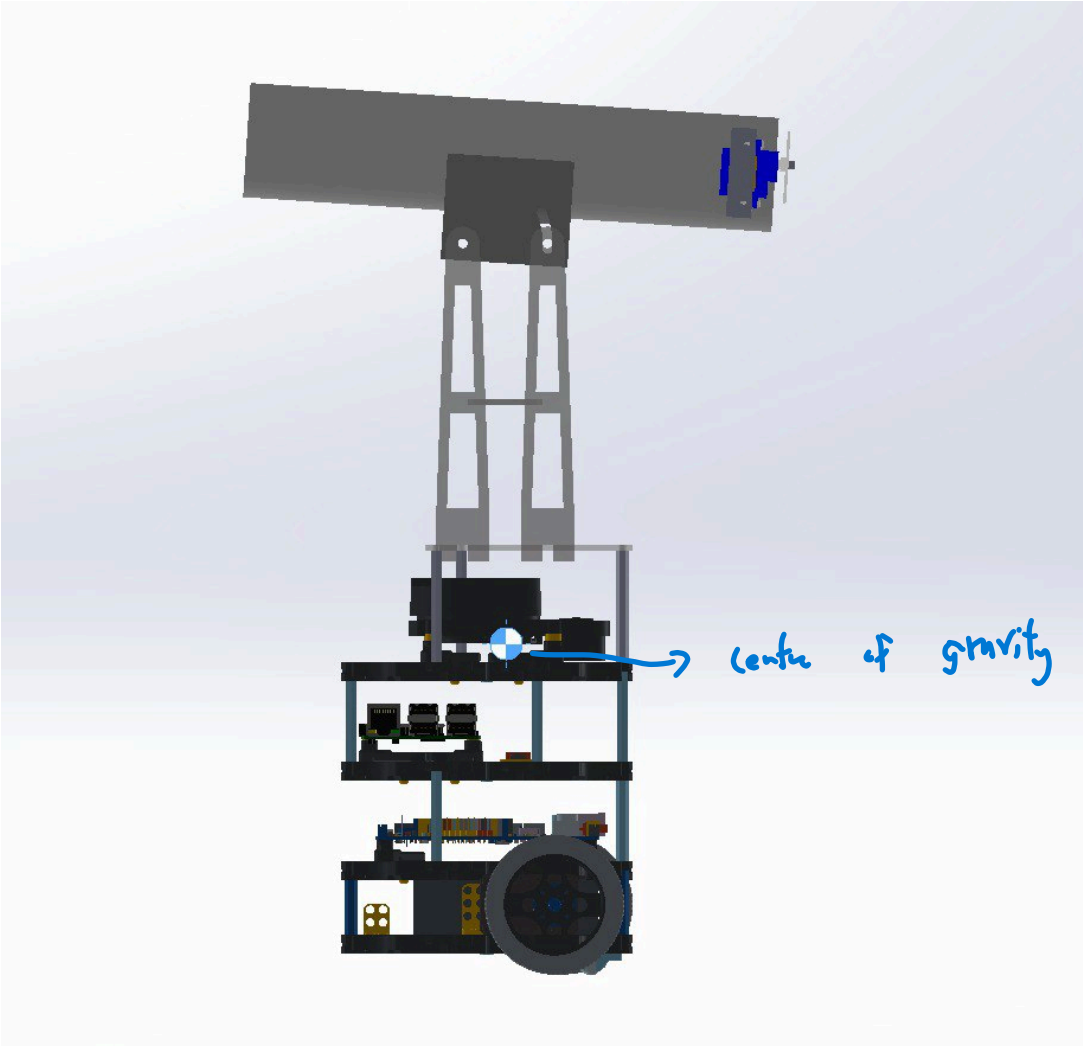
However, we faced some **problems** manufacturing the rubber band as it requires precisely manufactured and fitted parts to function properly. Due to **lack of design considerations**, the parts were dimensioned too small, making it **very difficult to 3D print precisely.** Moreover, we didn't account for tolerances for holes, making the whole assembly very time-consuming and difficult.



Video of the motorised launcher in action

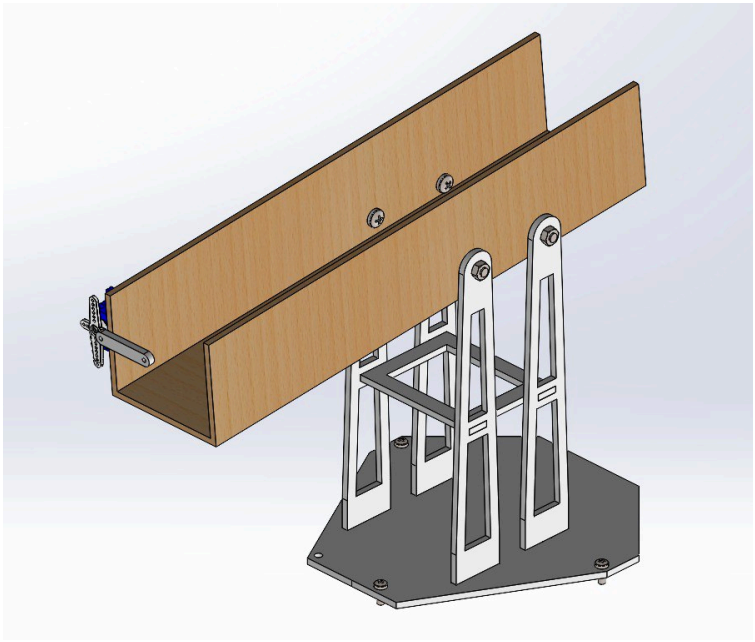
After **failing** to manufacture the rubber band launcher, **due to time and budget constraints**, we decided to go with a design that is more straightforward to implement, which is a **ping pong ball dropper.**

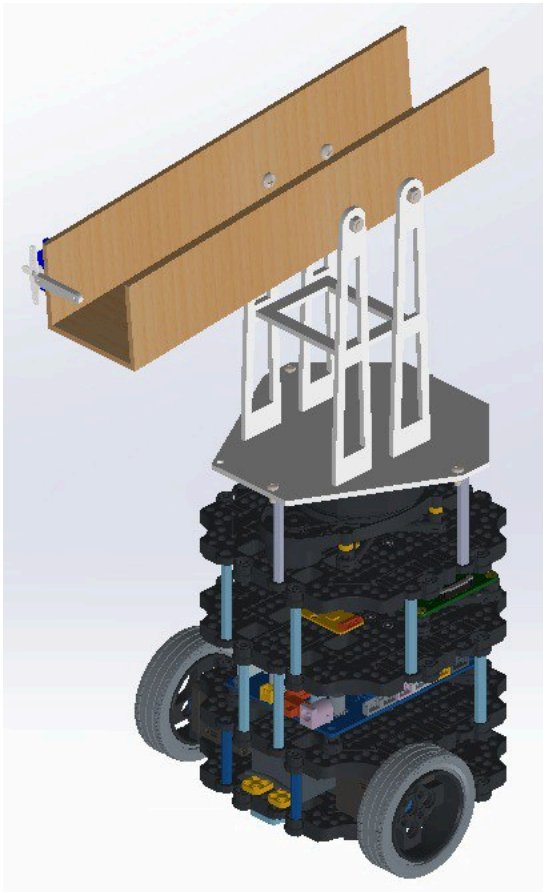
The ping pong ball dropper is an elevated inclined tunnel with a trapdoor that can contain 5 ping pong balls. The 5 balls will be released when the servo at the end is activated.

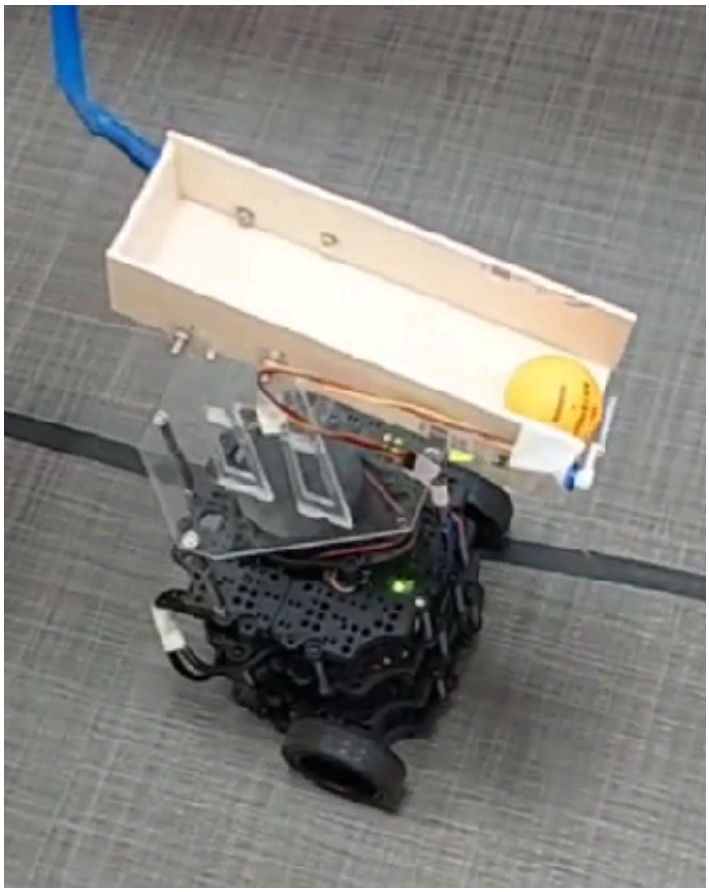


Centre of gravity of the Turtlebot3

When we were **manufacturing** the payload for Turtlebot, we noticed that our intended material for the circular tunnel, which was a **PVC pipe**, was too heavy and raised the **centre of gravity** of the Turtlebot significantly making it more **prone to topple over**. Hence, we went with a lighter material which is **balsa wood** that we found readily in the studio and it reduced our weight of payload substantially. The CAD of the balsa wood ball dropper is shown below.

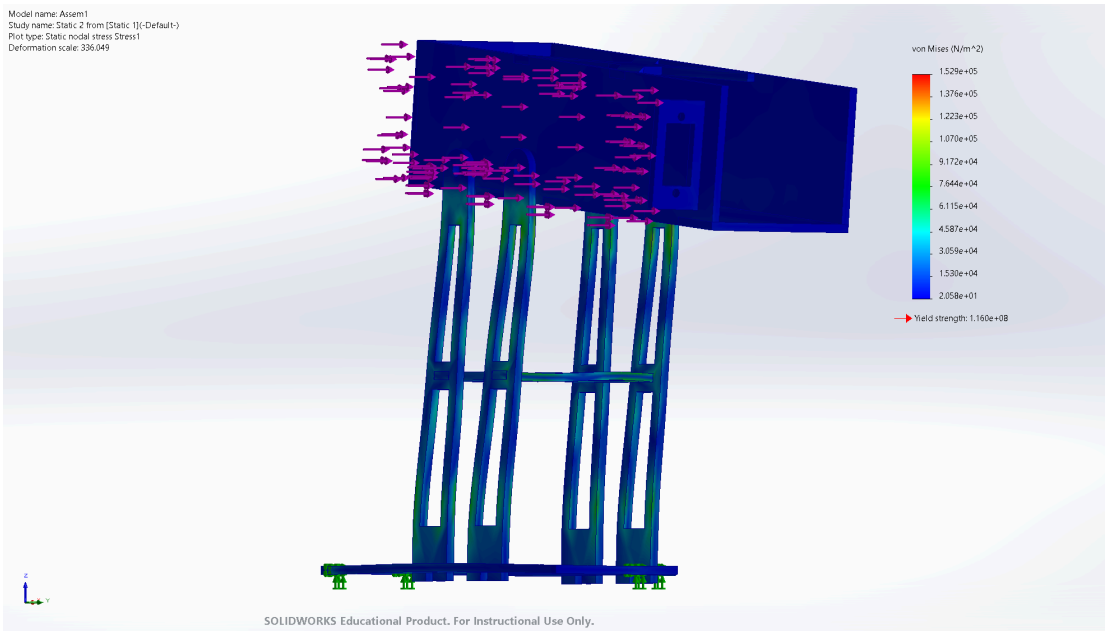




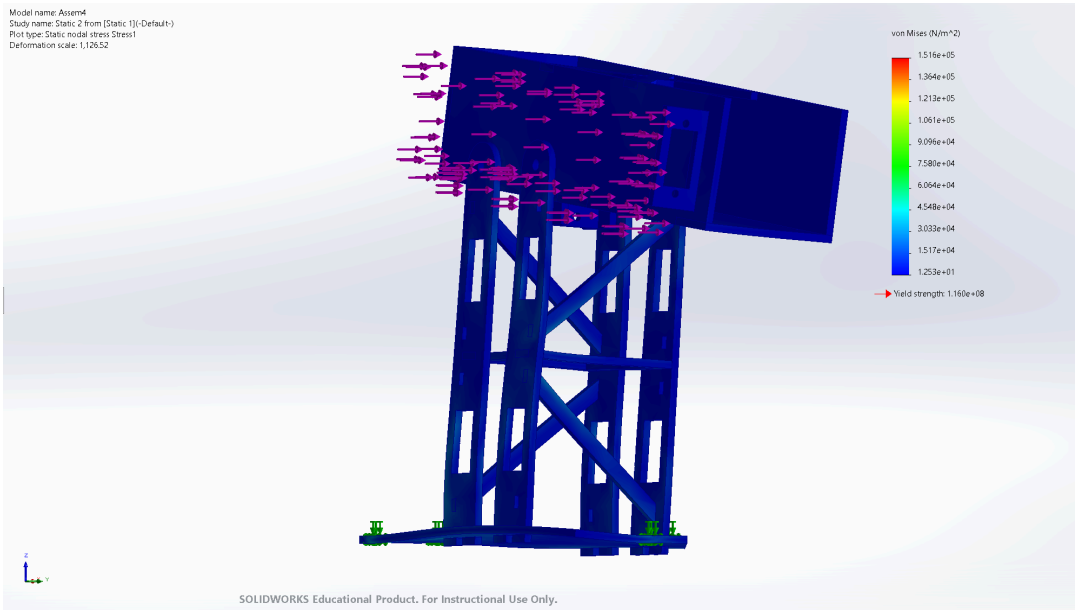


Turtlebot3 with balsa dropper

However, a **lighter** tunnel and **flimsy** supports that lack **rigidity** introduced more problems for us. The robot **wobbles significantly** when it moves as shown in this video. The wobble tunnel had potentially caused the robot to not move in a straight line as the wobbling caused the wheels to be lifted and lose contact with the ground making both wheels unable to move at the same speed. Hence, we came up with a solution which is to make the entire tunnel and support **sturdier** by adding **braces**, **diagonal braces** and changing the balsa tunnel to one made of **acrylic**. To tackle the wobbliness of the robot, we found out that adding a **tuned mass damper** helps dampen the vibration significantly and hence we decided to implement it in our final design.



FEA of ping pong ball dropper under sideways load



FEA of improved ping pong ball dropper under same sideways load

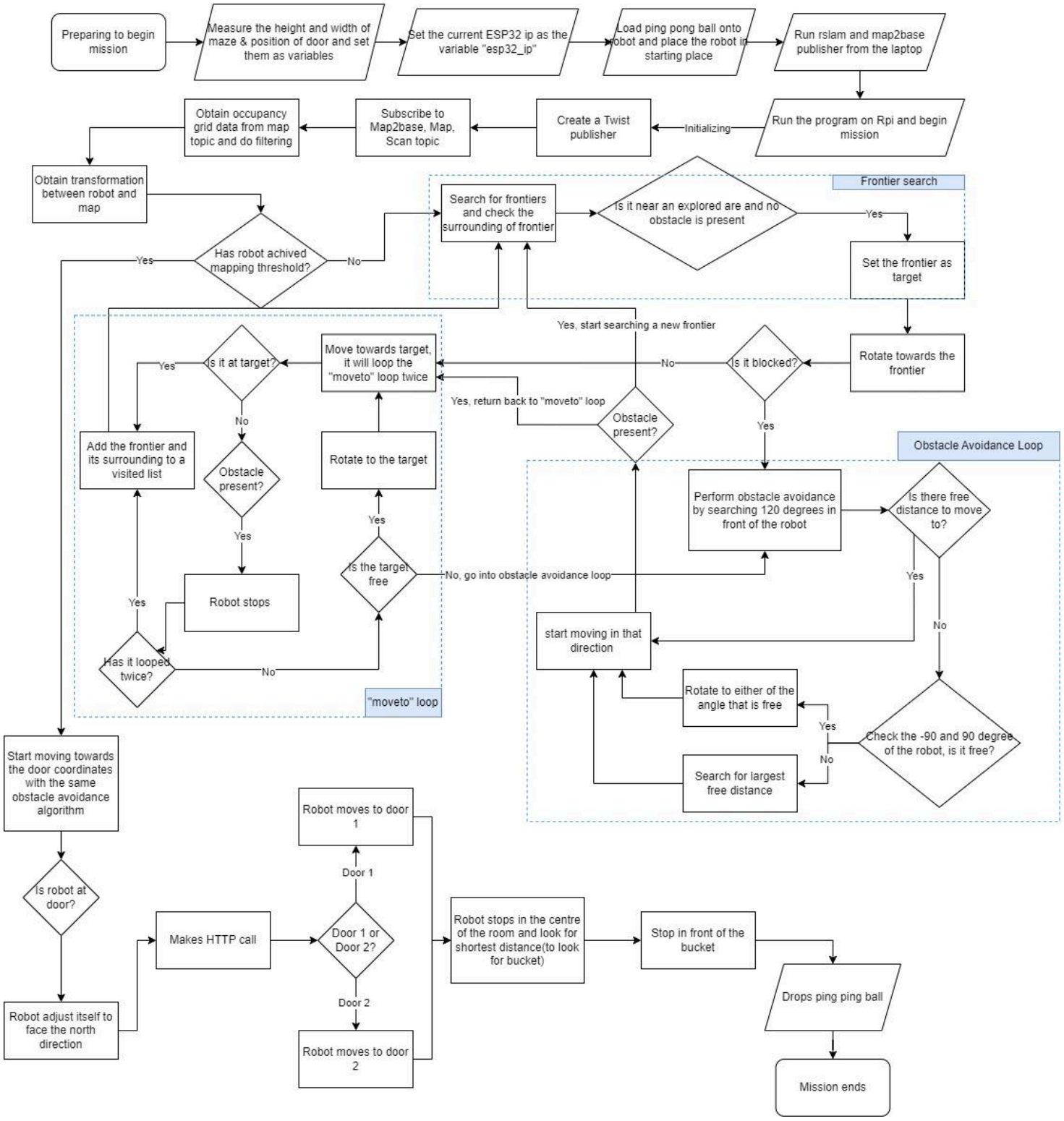
We conducted some simulation of the two versions of ping pong ball dropper, one with diagonal braces as well as widened supports and one without. The parameters of both simulations are identical and the results show that the improved ping pong ball dropper design experienced lesser deformation and lesser maximum stress, meaning it is able to withstand larger amounts of forces and be more rigid.

7. Final Design

The following details the design of the final system. This design can successfully complete the mission as well as overcome all the challenges posed by the previous versions.

7.1 Navigation Subsystem

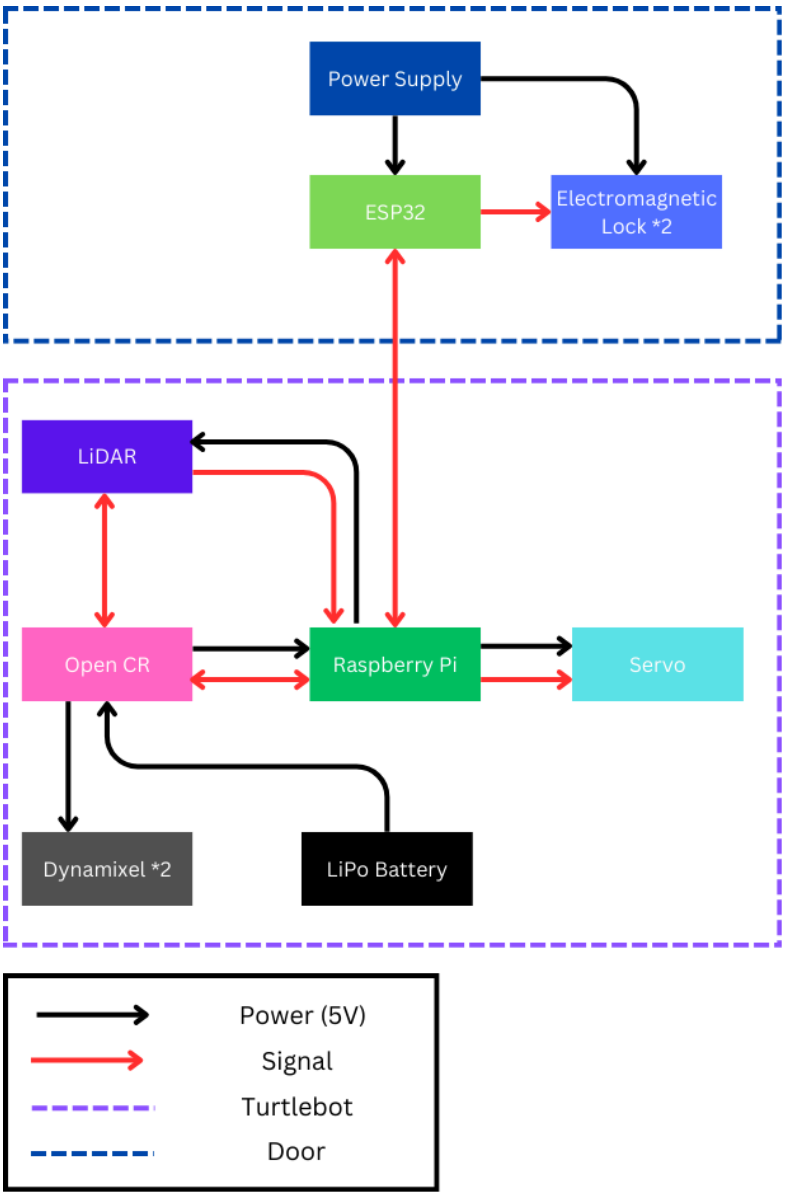
Our final navigation algorithm is a relatively simple go-to-goal algorithm. The robot uses a radial search to search for the **nearest frontier points** from its current position in the occupancy grid. The robot then simply **moves straight toward the goal**, checking if there are obstacles in its direct path and implementing obstacle avoidance to keep moving in approximately the same direction and prevent it from crashing into walls. After avoiding obstacles and walls, the robot will then reorient itself to the frontier goal and repeat the control loop. After a sufficient amount of the map has been mapped by us, the robot will then set the coordinates of the position between the doors as its goal. Using the same logic it **moves to the door**, and stops when it reaches the position. We will then reorient the robot to a yaw of 90 degrees with respect to the original position, and make **the HTTP call**, turning left or right depending on the result of the parsed response. It is able to take into account robot turning radius when avoiding obstacles and reverse out of dead ends and tight situations, given enough time.



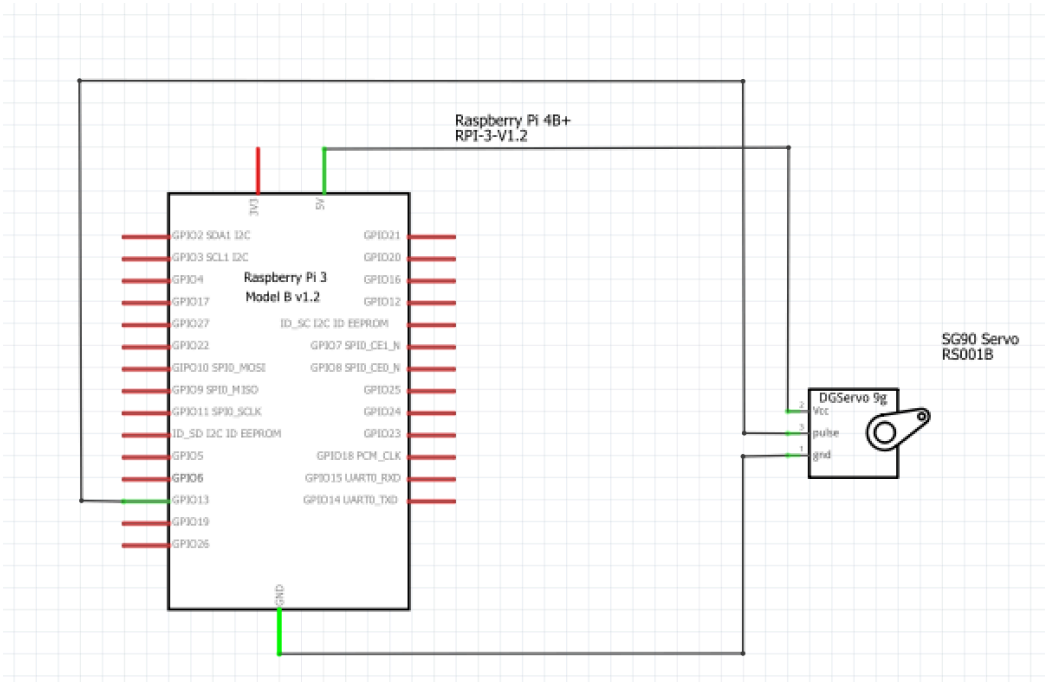
Navigation algorithm flowchart

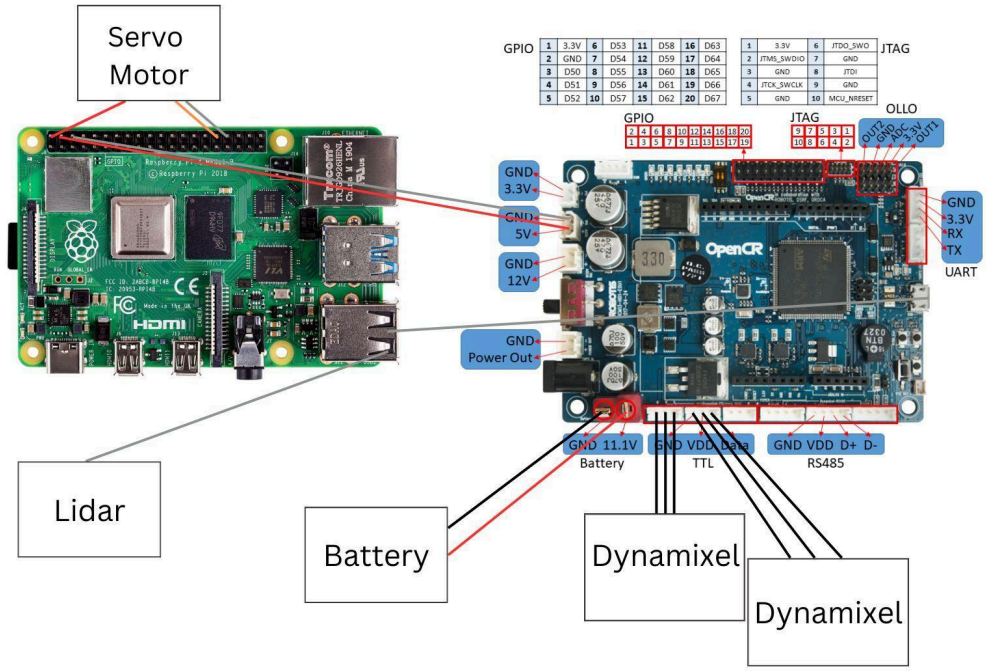
7.2 Electrical Subsystem

7.2.1 Electrical Architecture



7.2.2 Schematic Diagram:





7.2.3 Power Budgeting

Component	Quantity	Voltage (V)	Current (A)	Power (W)
Turtlebot (Booting)	1	11.1	0.65	7.215
Turtlebot (Stand-by)	1	11.1	0.5	5.55
Turtlebot (Performing)	1	11.1	0.85	9.435
Servo Motor (idle)	1	5	0.01	0.05
Servo Motor (during movement)	1	5	0.25	1.25

Power Consumption Calculation:
Assuming battery has 80% efficiency,
Battery energy = 11.1V * 1.800 mAh = 19.98Wh
Max runtime = 0.8*19.98/(9.435+1.25) = 1.50h

Even in the max power consumption scenario, where the Turtlebot is performing (moving) and the servo motor is turned on the entire time of 20 minutes in the final run, the power consumed by the system is less than the power supplied by the battery.

7.3 Communication Protocol

Hypertext Transfer Protocol (HTTP) and ESP32:

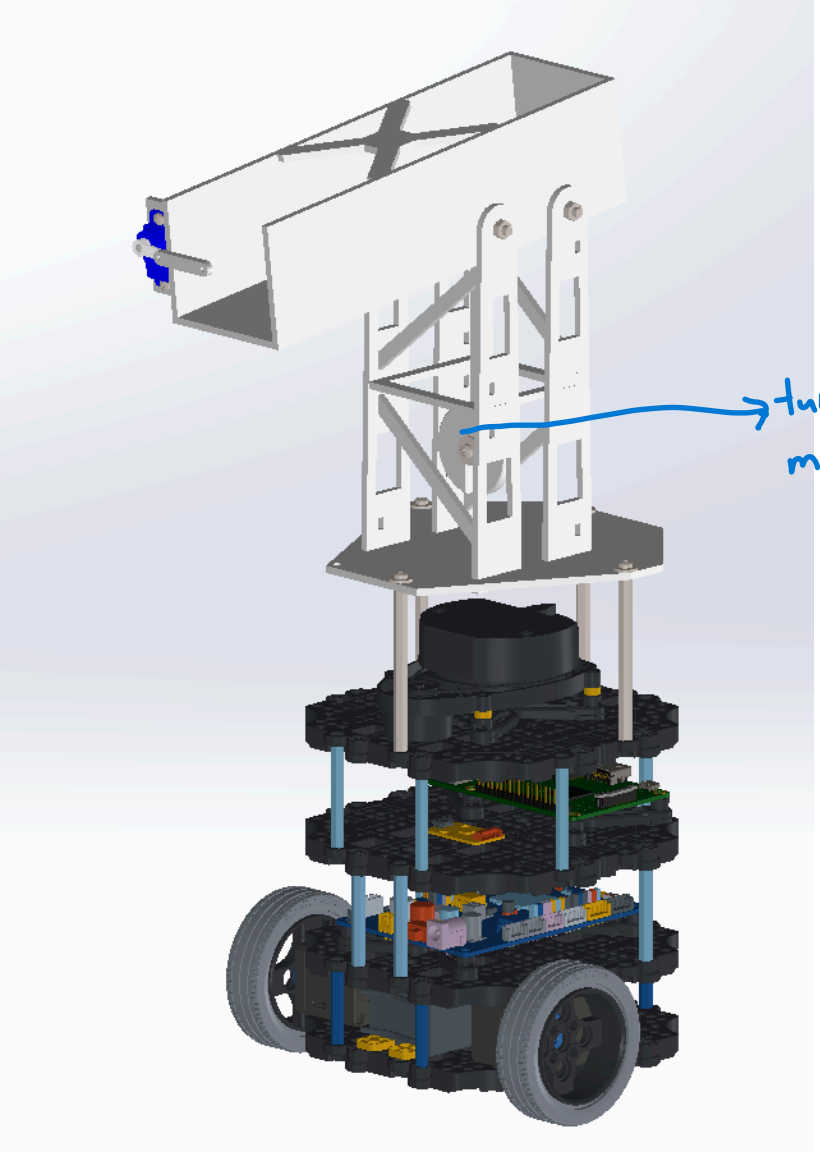
The second phase of the mission requires the robot to request information from an external server—i.e., the ESP32. Using the hypertext transfer protocol (HTTP), the Raspberry Pi makes a call to the server. If the request is valid, it receives information as to which of the doors is unlocked, and therefore proceeds to travel through it.

We coded the HTTP call in the python language using the “requests" library. We used “post” method to communicate with ESP32 to get a response back from the local server in ESP 32 which will tell the robot whether door 1 or 2 is opened. Then, the json() function would be used on the response to store the response in a dictionary object which makes it easier to access the information on which door is opened.

To perform testing using the ESP32 provided, we copied the code available in EG2310’s github (code attached in annex) and flashed it into the ESP32 using Arduino IDE by installing the dedicated library (Arduino Uno Wifi Dev Ed Library, ArduinoJson) and selecting the correct board type (FireBeetle-ESP32 or ESP32 Wrover Kit (all versions))

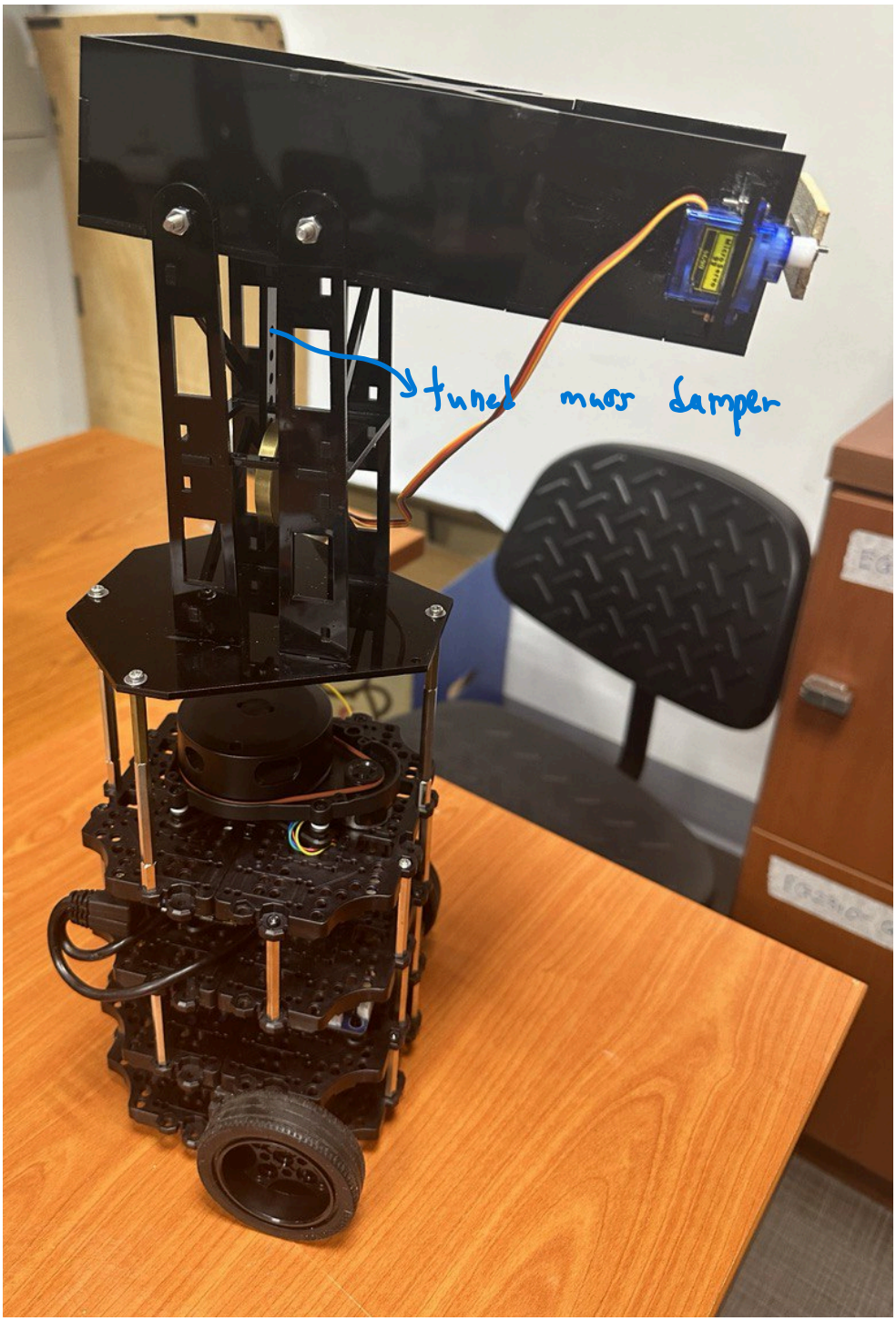
7.4 Payload Mechanism

Our payload evolved from our original idea, a solenoid launcher to a rubber band launcher, and finally came to our final solution – an elevated **ping pong ball dropper**



Picture showing full CAD of Turtlebot

Compared to our design of a **solenoid launcher** and **rubber band launcher** which **requires rigorous force calculations, tuning and testing,** our current design is **easier** to implement as we just need to make sure the opening of the tunnel is **taller than the bucket** so that it can deposit the balls into it. Moreover, we can **simplify two systems** which are launcher and loader into **one** in our final solution. To mitigate the **instability** issues, we implemented a **tuned mass damper** located between the supports; it significantly dampens the vibration experienced by the robot when it **accelerates**, **decelerates** and **turns**. The actual picture of the ball dropper is shown below. A video showing the improvement of stability of the payload is attached here.



Final sleek looking Turtlebot3 with modifications

7.5 Bill of Materials

No.	Part	QTY	Unit Cost	Total Cost
Turtlebot (Burger)				
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)	1		
21	Li-Po battery	1		
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	Prototyping Board	1		
30	Dynamixel XL 430	2		

31	OpenCR 1.0	1		
32	Raspberry Pi 3	1		
33	360 Laser Distance Sensor LDS-01	1		
34	USB2LDS	1		
35	Push Button	1		
36	M2x6 Bolt	2		
37	M3x8 Bolt	22		
38	M3x16 Bolt	4		
39	M3 Nut	24		
40	M2x10 Spacer	1		
41	M3x10 Spacer	4		
42	M3x40 Spacer	2		
Payload Parts (Ping Pong Ball Dropper)				
43	Acrylic Parts (Base, Supports*4, Support Brace, Support Diagonal Brace*4, Tunnel, Tunnel Brace, Servo Horn Extender, Servo Bracket, Damper Bracket, Damper Bar) [3mm acrylic sheets sourced from fabrication lab (FOC)]	1	5/hr	5
44	M4*10 Pan Head Cross Screws	4	Sourced From Lab	
45	M4 Washers	4		
46	M4 Nuts	4		
47	M3*25 Pan Head Cross Screws	2		
48	M3 Washers	4		
49	M3 Nuts	2		
50	M2*10 Pan Head Cross Screws	3		
51	M2 Nuts	3		
52	50g weights	1		
53	SG90 Servo	1	Provided	Provided
Grand Total				5

8. Evaluation and testing

Our final design required a lot of **parameter tuning**. In order to get the robot ready for the final mission run, we had to ensure that the parameters that we set for the robot were appropriate for the environment. This was done based on a visual of the maze, and on practice runs with other mazes of similar size and shape. For example, our code takes in parameters like the *search distance*, which is how we check the angle of LiDAR scan of the robot which has a value of more than said parameter, measured from the direction the robot is facing. The robot will turn to that angle and move there to avoid obstacles after stopping.

On the final run day, we were given 20 minutes to set up our computer, go through the user manual and do as many runs as required. Our robot looked like it was doing very well in the first few minutes. However, we did not notice that in the RViz display, the map being generated somehow was being compressed. I regret to say that we did not face this problem at all during testing, and we were completely caught off guard. Because our code relied on the fact that the door's coordinates are known and fixed, our code used the same constant value as the position of the door. Since the map was being compressed, as shown in the RViz grids, the map and the door coordinates started to drift away from our true real world measurements, by as much as 0.5 m in the x direction.

There are two possible reasons we came up with after analysing the video footage. The first, more unlikely one, is that due to screen recording, the computer that was used to run the scripts faced latency issues. However, we think that this is not a really grounded reason, as we tested the screen recording software the day before and it had no issues. The second, more plausible reason is that we decided to increase the robot's speed from 1.1 to 1.5 in order to clear the maze quicker. We ran some quick tests before the run to ensure that the robot could handle the speed, and that the robot could generate an accurate SLAM map with this speed for a short time. However, we did not realise that maybe after the errors compound after a longer period of time, the effect would be more detrimental. Testing after the final run yielded no visible evidence that the SLAM map was affected by the speed increase though.

We are quite satisfied with the robot, as **it did not crash into walls** — nor did we expect it to, since we did not have a crash despite numerous tests after tuning the obstacle avoidance algorithm. Another is that the last minute change for our payload really helped to make the robot more stable. Furthermore, the robot was able to get out of the initial obstacles in the lower half of the maze, and make its way to the doors. Barring the occupancy grid failure, the rest of the robot's movements were smooth and quite satisfactory. Unfortunately, the time constraints did not allow us to complete a second run, and we were forced to raise the white flag. This meant that we would accept less maximum points in exchange for starting at the door position immediately. This part went smoothly with no hiccups, as we already prepared a program to run in case this was necessary (*find_bucket.py*).

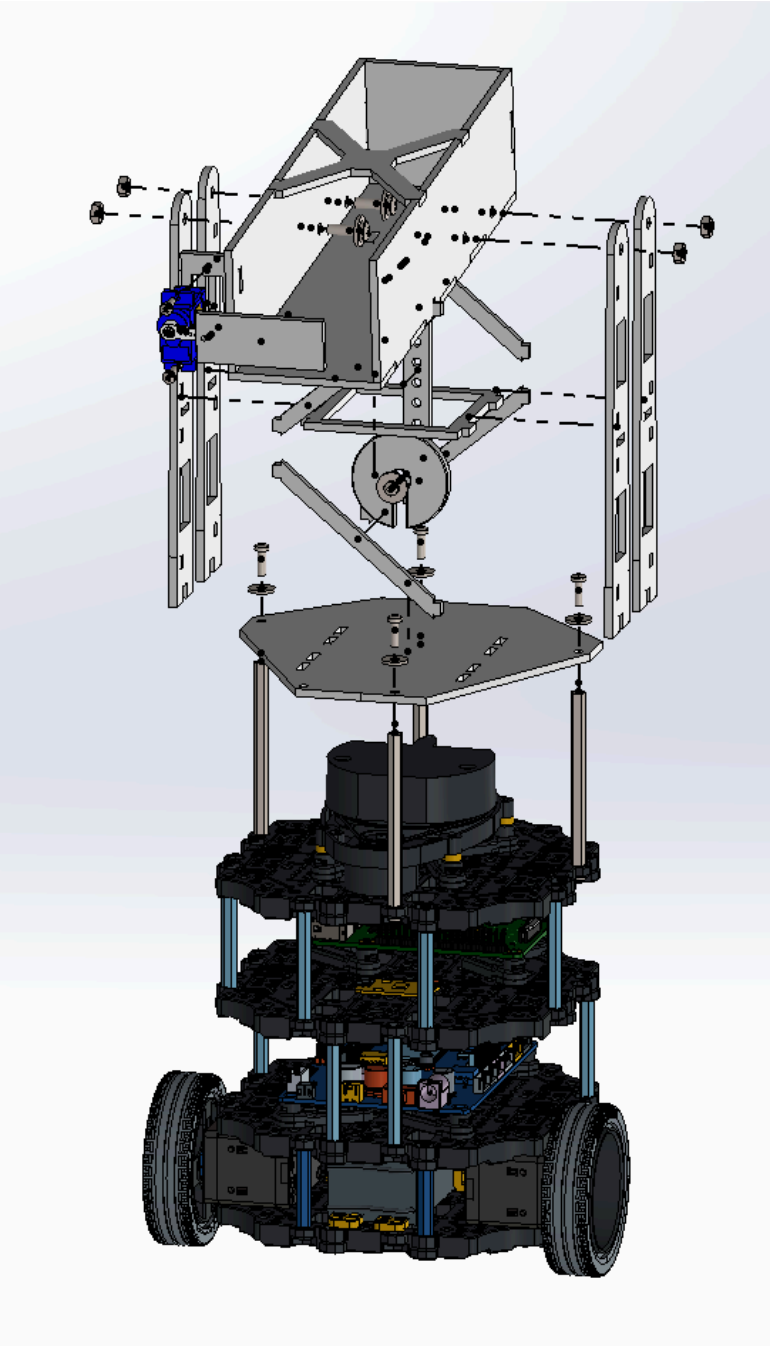
We spent a lot of time designing and improving our design of payload as we are keen to implement interesting ideas rather than normal ideas. We are very confident that we are able to implement the rubber band

launcher at first but we ended up failing to manufacture it. It is interesting how our design evolved from needing both a launch and reload system to just having a dropper. From the design process of the payload to deposit the ping pong balls, we learned that **a design is only a good design when it can be successfully manufactured and implemented**. Manufacturability is an important factor that we overlooked as not everything that looks good in CAD software can be realised due to time constraints, lack of skills and dedicated machining equipment. Another lesson learnt is that a **simple and consistent working design is often better than an inconsistent and complex design** as we only have limited time in our mission and we can't afford any mistakes. We still attempted to construct the initial launcher design afterwards: video of the motorised rubber band launcher.

Overall, we think that throughout this module, we were good at sticking to the KISS (Keep It Simple Stupid) philosophy, and trying to make our algorithms and payload mechanisms as simple as possible especially towards the final weeks. This is especially important when debugging and trying to make modifications, which is simpler to do with a simpler system. We believe a lesson we took away from this module is that in systems engineering, there will always be compromises, and it is our job to find the balance.

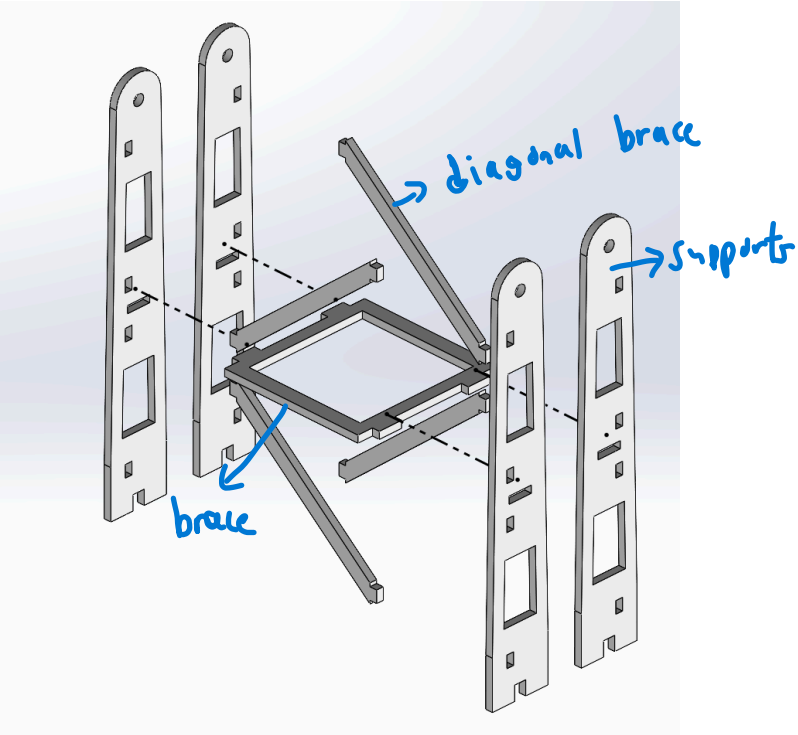
9. Assembly Instructions

9.1 Mechanical assembly of Turtlebot3 (with modifications)

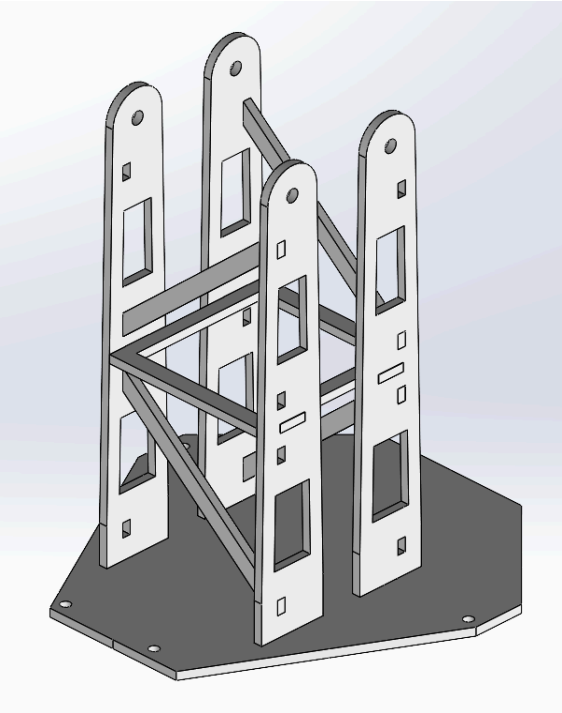


Exploded View of Ping Pong Ball Dropper

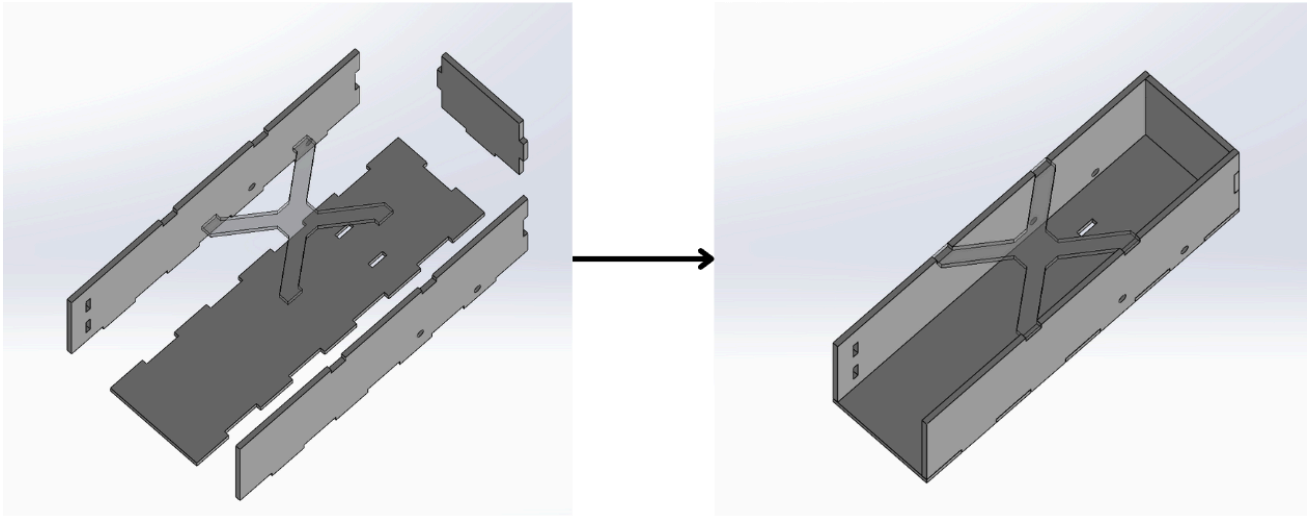
1. Glue 4 supports and braces together as shown below (Using acrylic glue sourced from lab)



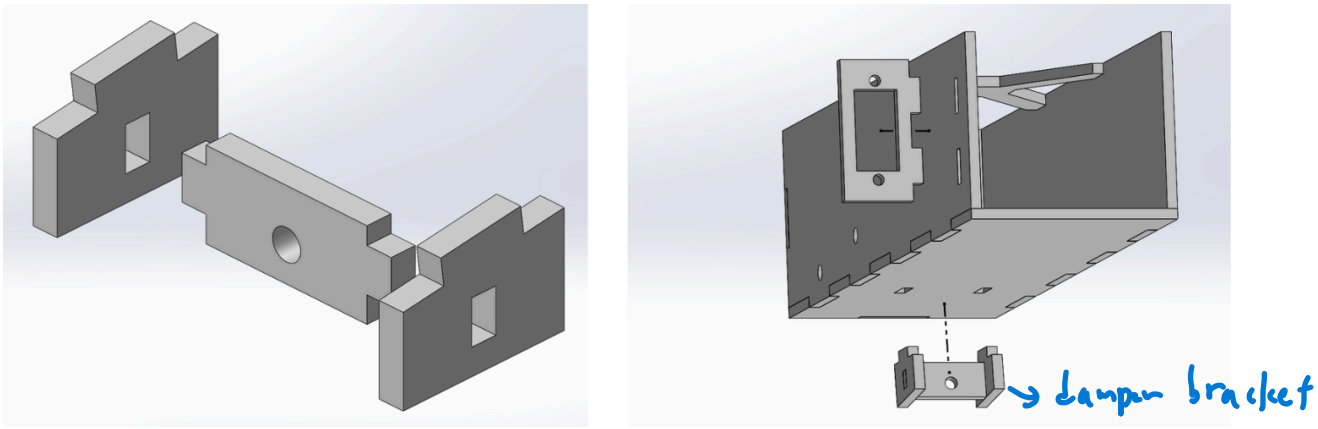
2. Glue the supports to the base



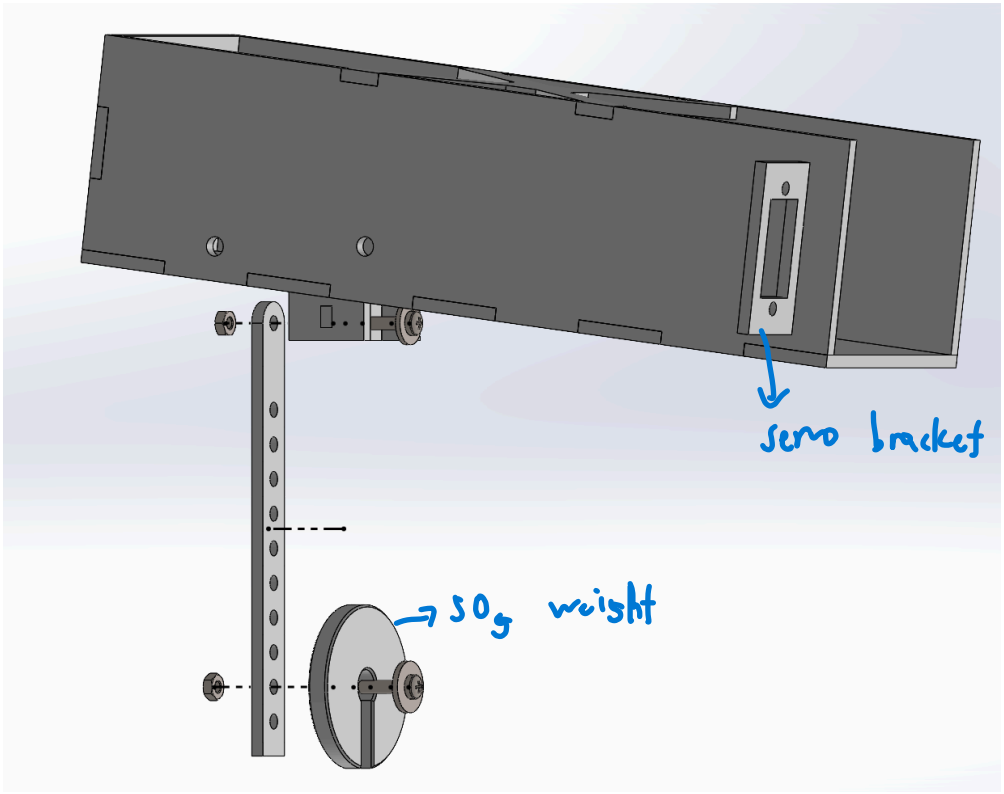
3. Glue the tunnel and tunnel brace together



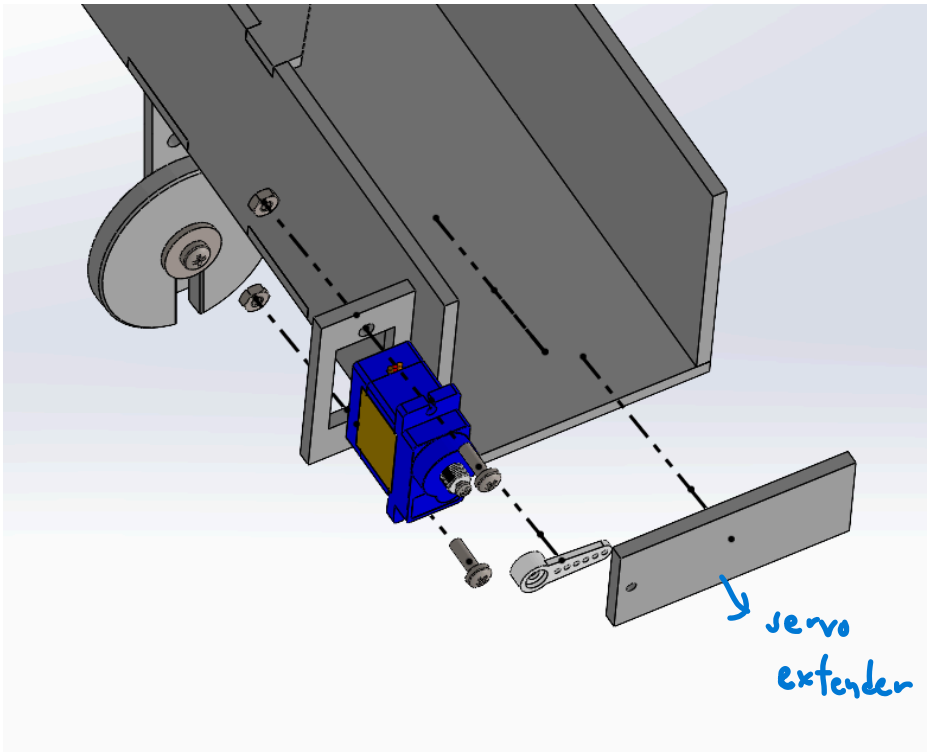
4. Glue the damper bracket together, glue it and servo bracket on the tunnel



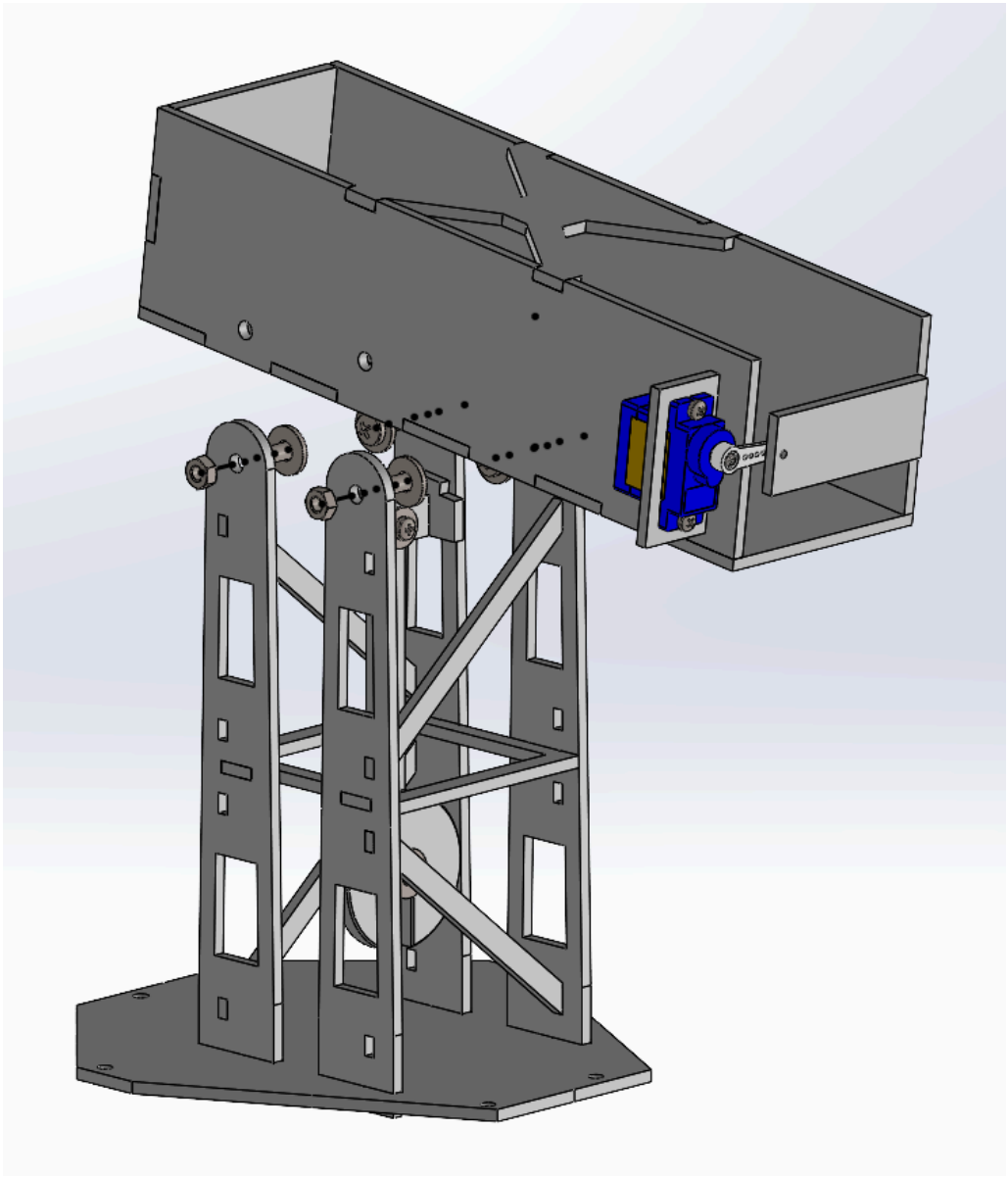
5. Secure the weight onto the damper bar, and then to the damper bracket using M3 screws, nuts and washer



6. Mount the servo extender onto the servo horn, then on the bracket using M2 screws.



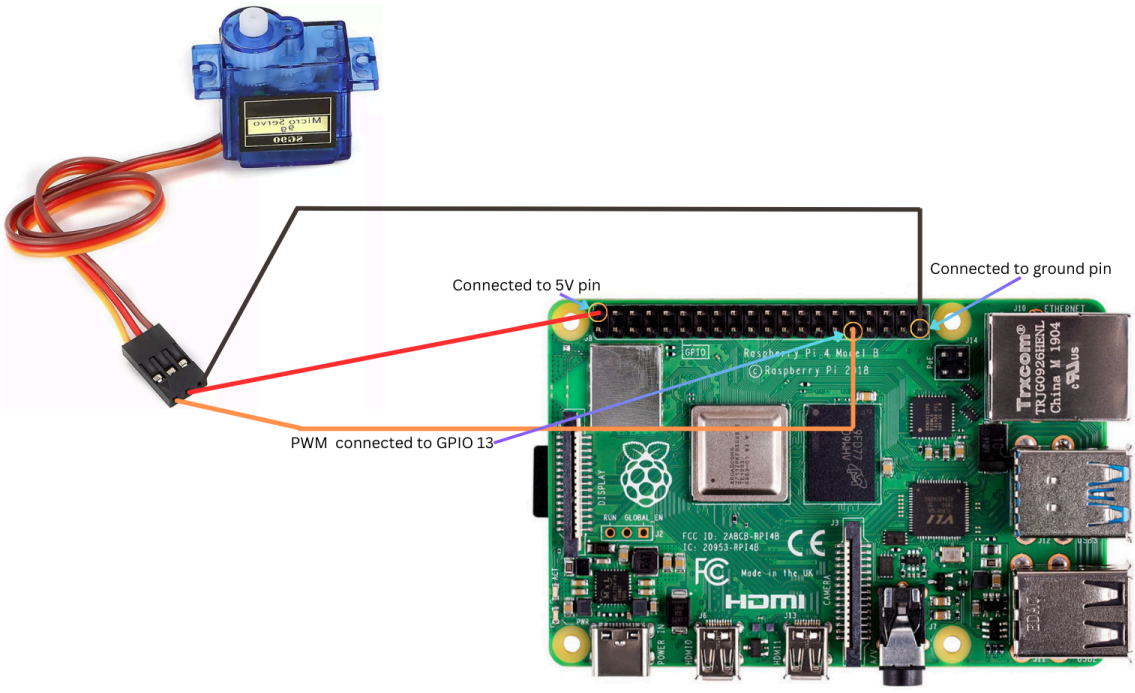
7. Secure the tunnel to the supports using M4 screws, nuts and washers



8. Secure 70mm (combination of 45mm+15mm+10mm) hex standoffs (3mm diameter) onto Turtlebot using M3 screws

9. Secure the ball dropper onto Turtlebot using M3 screws and washers

9.2 Electrical components setup



9.3 Software setup

9.3.1 Basic Installation

1. Ensure that Ubuntu 20.04 and ROS 2 Foxy is installed on your laptop (or on an external hard drive). Refer here for how to install the required software. You may have to dual boot or install a virtual machine on your laptop to make this work. Ensure that you are following the instructions under the "Foxy" tab. If you are interested in using an external SSD to boot Ubuntu for Mac users, check out this, this and this.
2. Check this tutorial here to make sure you can make a publisher/subscriber with ROS and test that everything is installed properly.
3. Follow the instructions here to use Ubuntu to burn the ROS2 Foxy image to the SD card on the 2 R-Pis.
4. After the ROS development environment is set up on both the remote laptop and the RPi, run the publisher from the RPi and the subscriber on the laptop. Make sure that the subscriber is able to replicate what is being produced by the publisher. Swap the publisher and subscriber (run publisher on the laptop and the subscriber on the RPi) to ensure that 2 way communication can take place.
5. Install Cartographer by running the following commands in a terminal instance.

```
sudo apt install ros-foxy-cartographer
```

```
sudo apt install ros-foxy-cartographer-ros
```

6. Add these lines to the .bashrc file in the root directory of RPi.

```
export Turtlebot3_MODEL=burger
alias rosbu='ros2 launch Turtlebot3_bringup robot.launch.py'
```

7. Use the instructions here to create an AWS ec2 instance, which will make it easier to boot into the Rpi by letting the Rpi update its IP address when it starts up.
8. Make a ROS2 package in the Raspberry Pi, if you do not have one, following this tutorial. Make sure to modify the package.xml and setup.py files in the auto_nav package or wherever you are installing auto_nav into to make sure the dependencies and node names match the one in the repository. After that build the colcon workspace and source your installation.

9.3.2 Installing the Programs on a Remote Laptop

1. Create a ROS 2 package on the remote laptop and move the file in the directory temporarily to the parent directory.

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

mv __init__.py ..
```

2. Clone the GitHub repository to the remote laptop from this repository. Make sure the period at the end is included.

```
git clone
https://github.com/john-yeap01/maze-solver-eg2310.git .
```

3. Remove unnecessary folders and files since you have already created your own workspace

```
rm -rf build install log .git .gitattributes README.md
```

4. Build the package

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

5. Add the following lines in ~/.bashrc file in the root directory of the laptop

```
alias sshrp='ssh ubuntu@`ssh aws cat rpi.txt`'
export Turtlebot3_MODEL=burger
alias rteleop='ros2 run Turtlebot3_teleop teleop_keyboard'
alias rslam='ros2 launch Turtlebot3_cartographer
cartographer.launch.py'
```

6. Edit the cartographer config.yaml file and change line 29 to the highlighted line.

```
cd /opt/ros/foxy/share/turtlebot3_cartographer/config/
sudo nano turtlebot3_lds_2d.lua
```

Edit

```
provide_odom_frame = false,
publish_frame_projected_to_2d = true,
use_odometry = false,
use_nav_sat = false,
use_landmarks = false,
```

7. Refer to the previous section — most steps are the same as for installing on the Rpi. You should be able to get the Turtlebot3 maze solving by running the ROS2 bringup on the Rpi (*rosbu*), then *rslam* and *ros2 run auto_nav map2base* on your laptop. Finally open a 4th and last terminal in the Rpi and *ros2 run auto_nav solver2* to start the mission.

9.4 Code explanation - *solver2.py*

Note: instead of getting odometry pose data from /*odom*, we will use the script *map2base.py.* Map2Base subscribes to cartographer for the TurtleBot transform data in the form of */tf* topic. To obtain the /*tf* data, the Cartographer subscribes to both /scan from the and /imu, data from the Inertial Measurement Unit published by node.

```python
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
from geometry_msgs.msg import Pose
from sensor_msgs.msg import LaserScan
from nav_msgs.msg import OccupancyGrid
import tf2_ros
from tf2_ros import LookupException, ConnectivityException, ExtrapolationException
from rclpy.qos import qos_profile_sensor_data
import numpy as np
import math
import cmath
import time
from rclpy.qos import ReliabilityPolicy, QoSProfile
import scipy.stats
import requests
import RPi.GPIO as GPIO

```

Importing libraries that will be used.

```python
startMove = 4
rotatechange = 0.35
speedchange = 0.15
stop_distance = 0.3
search_distance = 0.6

#angles will be scaled according to the length of the laser array not 360 (200+) -- adjust accordingly
front_angle = 12
front_angles = range(-front_angle,front_angle+1,1)
search_angle = 60
search_angles = range(-search_angle,search_angle+1,3)
box_thres = 0.1 #so that it reach frontier behind wall?

door_x = 1.6
door_y = 2.70
width_map = 3.55
height_map = 2.6   #1.6+1.08 #measured the table length, 1.6m
map_percent = 0.7

# !!!!!RMB TO CHANGE this to the IP of your ESP32!!!!!
esp32_ip = ''
#esp32_ip = '192.168.227.169'
#esp32_ip = '192.168.38.169'
TurtleBot3_ID = 'turtlebot'
door_time = 5   #time to go through the door before stopping
dist_threshold = 0.16 # Distance threshold for the robot to stop in front of the pail

free_dis_thres = 0.2
free_dis = 0.4
long_dis = 1.5

occ_bins = [-1, 0, 50, 100]
visited = set()

```

Constants declared.

startMove	the time for robot to move straight at the beginning to update slam map
rotatechange	speed of robot rotating
speedchange	speed of robot moving straight
stop_distance	distance for robot to stop in front of obstacles
search_distance	distance for robot to search to look for directions are that are free
front_angles	angles that robot will check to determine if there are obstacles in front
search_angles	angles that robot will check to look for free distance
door_x, door_y	position of door from robot's starting point

width_map, height_map	used for calculating the percentage of area that is mapped
ESP32_ip	variable for storing ESP32 IP
Turtlebot3_ID	used by Turtlebot to make HTTP call
door_time	time for robot to pass through the door and move to the centre of the room
dist_threshold	distance that robot will stop at in front of the bucket, it cannot be lower than 0.16 because it will cause lidar to go “blind” or not able to detect distance in front of it
free_dis_threshold	parameter to determine if that direction is free
free_dis	
long_dis	parameter to determine if a target is faraway from the robot
occ_bins	variable to be used in map
visited	create a list for storing visited place

```
#SERVO
GPIO.setmode(GPIO.BCM)##rpi , lidar, bucket
servo_pin = 13
GPIO.setup(servo_pin, GPIO.OUT)
pwm = GPIO.PWM(servo_pin, 50)
pwm.start(7.5) #90
time.sleep(0.5)

def launch_ball():
    print("Launching! Big Balls")
    duty_cycle = 2.5 + float(180) / 18#rotate 90
    pwm.ChangeDutyCycle(duty_cycle)
    time.sleep(0.5)  # Give time for servo to move
```

This code sets up the servo by setting the pin servo is connected to as output and sets the PWM frequency at 50Hz. The servo will be initialised at 90 degrees. The function `launch_ball` is to make the servo rotate to 180 degrees so that the ping pong balls will be released into the bucket.

```python
def euler_from_quaternion(x, y, z, w):
    """
    Convert a quaternion into euler angles (roll, pitch, yaw)
    roll is rotation around x in radians (counterclockwise)
    pitch is rotation around y in radians (counterclockwise)
    yaw is rotation around z in radians (counterclockwise)
    """
    t0 = +2.0 * (w * x + y * z)
    t1 = +1.0 - 2.0 * (x * x + y * y)
    roll_x = math.atan2(t0, t1)

    t2 = +2.0 * (w * y - z * x)
    t2 = +1.0 if t2 > +1.0 else t2
    t2 = -1.0 if t2 < -1.0 else t2
    pitch_y = math.asin(t2)

    t3 = +2.0 * (w * z + x * y)
    t4 = +1.0 - 2.0 * (y * y + z * z)
    yaw_z = math.atan2(t3, t4)

    return roll_x, pitch_y, yaw_z # in radians

def calculate_yaw_and_distance(x1, y1, x2, y2, current_yaw):
    # 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))
```

`euler_from_quaternion` is used for calculating yaw, pitch and roll.

`calculate_yaw_and_distance` is used for calculating the angle between 2 points and the distance between 2 points.

```python
class Solver(Node):
    def __init__(self):
        super().__init__('solver')
        # create publisher for moving TurtleBot
        self.publisher_ = self.create_publisher(Twist,'cmd_vel',10)

        # odometry
        self.roll = 0.0
        self.pitch = 0.0
        self.yaw = 0.0
        self.pos_x = 0.0
        self.pos_y = 0.0

        # map grid
        self.mapbase = Pose().position
        self.cmd = Twist()
        self.map_origin = Pose().position
        self.map_res = 0.0
        self.odata = np.array([])

        # occupancy grid and positions
        self.occ_subscription = self.create_subscription(
            OccupancyGrid,
            'map',
            self.occ_callback,
            qos_profile_sensor_data)
        self.occ_subscription  # prevent unused variable warning
        self.occdata = np.array([])
        self.tfBuffer = tf2_ros.Buffer()
        self.tfListener = tf2_ros.TransformListener(self.tfBuffer, self)

        # lidar scan
        self.scan_subscription = self.create_subscription(
            LaserScan,
            'scan',
            self.scan_callback,
            qos_profile_sensor_data)
        self.scan_subscription  # prevent unused variable warning
        self.laser_range = np.array([])

        self.tf_subscriber = self.create_subscription(
            Pose,
            'map2base',
            self.tf_callback,
            QoSProfile(depth=10, reliability=ReliabilityPolicy.RELIABLE))
        self.tf_subscriber
```

The attributes of the “Solver” node are declared.

- The code creates a publisher to publish the “Twist” topic which will be used for moving the robot.
- It subscribes to the map topic to obtain the occupancy grid data.
- It subscribes to the scan topic to obtain the readings of the lidar.
- It subscribes to the map2base topic to obtain the robot’s odometry with respect to the map.

`self.row, self.pitch, self.yaw, self.pos_x, self.pos_y`	Stores the robot’s odometry
`self.mapbase`	Stores the position of robot with respect to map frame
`self.cmd`	Twist object used for publishing movement
`self.map_origin`	Stores the map origin of the slam map with respect to map frame
`self.map_res`	Stores the map resolution
`self.odata, self.occdata`	Stores the occupancy grid data
`self.tfBuffer, self.tfListener`	Obtain transformation between the robot and map frame

`self.laser_range`	Stores the lidar readings

```python
    def tf_callback(self, msg):
        rclpy.spin_once(self)
        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(orientation_quat.x, orientation_quat.y, orientation_quat.z, orientation_quat.w)
        self.mapbase = msg.position
        self.pos_x = msg.position.x
        self.pos_y = msg.position.y

    def scan_callback(self, msg):
        rclpy.spin_once(self)
        # self.get_logger().info('In scan_callback')
        # create numpy array
        self.laser_range = np.array(msg.ranges)
        # print to file
        # np.savetxt(scanfile, self.laser_range)
        # replace 0's with nan
        self.laser_range[self.laser_range==0] = np.nan
```

The function `tf_callback` uses data from the `map2base` topic and stores it as the odometry of the robot.

The function `scan_callback` uses data from the `/scan` topic and stores the readings of the lidar as a numpy array.

```python
    def occ_callback(self, msg):
        # create numpy array occdata, WHICH IS 1 DIMENSIONAL --
        # the following code makes it into 2D for coordinate systems to make sense
        occdata = np.array(msg.data)

        # New threshold values for the desired conditions
        threshold_value_low = 60
        threshold_value_high = 95  # Set to the same value for exclusive handling

        # Apply thresholding with the updated conditions
        thresholded_data = np.where(occdata == -1,
                                    occdata,  # Leave -1 values unchanged
                                    np.where(occdata == 0, 0,  # Values equal to 0 remain 0
                                             np.where((occdata > 0) & (occdata < threshold_value_low), -1,  # Values between 0 and 60 become -1
                                                      np.where(occdata > threshold_value_high, 100, occdata))))  # Values above 60 become 100, others remain unch

        occdata = thresholded_data

        # compute histogram to identify bins with -1, values between 0 and below 50,
        # and values between 50 and 100. The binned_statistic function will also
        # return the bin numbers so we can use that easily to create the image
        occ_counts, edges, binnum = scipy.stats.binned_statistic(occdata, np.nan, statistic='count', bins=occ_bins)
        self.map_visit = occ_counts[1]
        # get width and height of map
        iwidth = msg.info.width
        iheight = msg.info.height
        self.map_width = iwidth
        self.map_height = iheight

        # calculate total number of bins
        total_bins = iwidth * iheight
        # log the info
        # self.get_logger().info('Unmapped: %i Unoccupied: %i Occupied: %i Total: %i' % (occ_counts[0], occ_counts[1], occ_counts[2], total_bins))
        try:
            trans = self.tfBuffer.lookup_transform('map', 'base_link', rclpy.time.Time())

        except (LookupException, ConnectivityException, ExtrapolationException) as e:
            self.get_logger().info('No transformation found')
            return

        cur_pos = trans.transform.translation
        cur_rot = trans.transform.rotation
        self.pos_x = cur_pos.x
        self.pos_y = cur_pos.y
        # self.get_logger().info('Trans: %f, %f' % (self.pos_x, self.pos_y))
```

The function `occ_callback` uses data from the `/map` and turns the occupancy grid data into a numpy array. Then, we implemented a custom filter such that the values where -1 remains as -1, 0 remains as 0 while values between 0 to 60 is changed to -1 while values between 95 to 100 remains as it is.

We changed the values between 0 to 60 to -1 because we discovered that this could help us remove the explored areas when the robot's LiDAR "peeks" through the gaps, so it improves our frontier search.

```
223         # convert quaternion to Euler angles
224         roll, pitch, yaw = euler_from_quaternion(cur_rot.x, cur_rot.y, cur_rot.z, cur_rot.w)
225         # self.get_logger().info('Rot-Yaw: R: %f D: %f' % (yaw, np.degrees(yaw)))
226 
227         # get map resolution
228         map_res = msg.info.resolution
229         self.map_res = map_res
230         # get map origin struct has fields of x, y, and z
231         map_origin = msg.info.origin.position
232         self.map_origin = map_origin
233         # get map grid positions for x, y position
234         grid_x = round((cur_pos.x - map_origin.x) / map_res)
235         grid_y = round(((cur_pos.y - map_origin.y) / map_res))
236         self.grid_x = grid_x
237         self.grid_y = grid_y
238         # self.get_logger().info('Grid Y: %i Grid X: %i' % (grid_y, grid_x))
239         # binnum go from 1 to 3 so we can use uint8
240         # convert into 2D array using column order
241         odata = np.uint8(binnum.reshape(msg.info.height,msg.info.width))
242         self.odata = odata
```

`grid_x` and `grid_y` calculates the index of the robot's position in the occupancy grid data, `self.grid_x` and `self.grid_y` stores the robot's index

```
244     def stopbot(self):
245         self.get_logger().info('stopping in stopbot')
246         self.cmd.linear.x = 0.0
247         self.cmd.angular.z = 0.0
248         self.publisher_.publish(self.cmd)
249 
250     def movebot(self):
251         self.get_logger().info('moving in movebot')
252         self.cmd.linear.x = speedchange
253         self.cmd.angular.z = 0.0
254         self.publisher_.publish(self.cmd)
```

`stopbot` is to publish to the Twist object to stop the robot.

`movebot` is to publish to the Twist object to give a linear velocity

```
    def rotatebot(self, rot_angle):
        # self.get_logger().info('In rotatebot')
        # create Twist object
        twist = Twist()
        rclpy.spin_once(self)
        # get current yaw angle
        current_yaw = self.yaw
        # log the info
        self.get_logger().info('Current: %f' % math.degrees(current_yaw))
        # we are going to use complex numbers to avoid problems when the angles go from
        # 360 to 0, or from -180 to 180
        c_yaw = complex(math.cos(current_yaw),math.sin(current_yaw))
        # calculate desired yaw
        target_yaw = current_yaw + math.radians(rot_angle)
        # convert to complex notation
        c_target_yaw = complex(math.cos(target_yaw),math.sin(target_yaw))
        self.get_logger().info('Desired: %f' % math.degrees(cmath.phase(c_target_yaw)))
        # divide the two complex numbers to get the change in direction
        c_change = c_target_yaw / c_yaw
        # get the sign of the imaginary component to figure out which way we have to turn
        c_change_dir = np.sign(c_change.imag)
        # set linear speed to zero so the TurtleBot rotates on the spot
        twist.linear.x = 0.0
        # set the direction to rotate
        twist.angular.z = c_change_dir * rotatechange
        # start rotation
        self.publisher_.publish(twist)

        # we will use the c_dir_diff variable to see if we can stop rotating
        c_dir_diff = c_change_dir
        # self.get_logger().info('c_change_dir: %f c_dir_diff: %f' % (c_change_dir, c_dir_diff))
        # if the rotation direction was 1.0, then we will want to stop when the c_dir_diff
        # becomes -1.0, and vice versa
        while(c_change_dir * c_dir_diff > 0):
            # allow the callback functions to run
            rclpy.spin_once(self)
            current_yaw = self.yaw
            # convert the current yaw to complex form
            c_yaw = complex(math.cos(current_yaw),math.sin(current_yaw))
            # self.get_logger().info('Current Yaw: %f' % math.degrees(current_yaw))
            # get difference in angle between current and target
            c_change = c_target_yaw / c_yaw
            # get the sign to see if we can stop
            c_dir_diff = np.sign(c_change.imag)
            # self.get_logger().info('c_change_dir: %f c_dir_diff: %f' % (c_change_dir, c_dir_diff))
```

```

        self.get_logger().info('End Yaw: %f' % math.degrees(current_yaw))
        # set the rotation speed to 0
        twist.angular.z = 0.0
        # stop the rotation
        self.publisher_.publish(twist)
```

`rotatebot` is obtained from *r2auto_nav* script and it is used for rotating the robot by an `angle` in degrees

```python
    def check_surroundings(self, array, x, y, radius=1):
        try:
            """
            Check the cells around a given center in a 2D grid.

            Parameters:
            - grid: 2D array representing the grid
            - x, y: Coordinates of the center
            - radius: Radius around the center to check (default is 1)

            Returns:
            - List of values in the surrounding cells
            """
            surroundings = []
            rows = len(self.odata)
            cols = len(self.odata[0])

            for i in range(x - radius, x + radius + 1):
                for j in range(y - radius, y + radius + 1):
                    # Check if the current coordinates are within the grid boundaries
                    if 0 <= i < rows and 0 <= j < cols:
                        surroundings.append(array[i,j]) #i,j flipped?
                    else:
                        # If outside the grid, append a default value (e.g., -1)
                        # surroundings.append(-1)  # or any default value you prefer
                        continue
            return surroundings

        except Exception as e:
            print('in checking surroundings')
            print(e)
```

The function `check_surroundings` adds the occupancy grid data bins within an inputted radius into a list. The list will then be used in frontier search to check the neighbours of that bin.

```python
    def get_grid(self, x, y):
        rclpy.spin_once(self)
        print('getting grid')
        grid_x = round((x - self.map_origin.x) / self.map_res)
        grid_y = round((y - self.map_origin.y) / self.map_res)
        print(f'gridx, gridy of goal: {grid_x}, {grid_y}')
        return int(grid_x),int(grid_y)   # may have to change back to `return (grid_x, grid_y)`

    def get_pose(self, grid_x, grid_y):
        rclpy.spin_once(self)
        print('getting pose')
        pose_x = grid_x*self.map_res + self.map_origin.x
        pose_y = grid_y*self.map_res + self.map_origin.y
        return pose_x, pose_y
```

The function “get_grid” real life positions into the index of the occupancy grid data array.

The function “get_pose” converts the index of the occupancy grid data array into real world positions.

```python
    def search_radius(self, array, x, y, max_radius):
        try:
            # rclpy.spin_once(self)
            frontier = True
            for radius in range(1, max_radius + 1):

                # Define the boundaries of the search area for the current radius
                min_x = max(0, x - radius)
                max_x = min(array.shape[1] - 1, x + radius)#shape 1 get columns
                min_y = max(0, y - radius)
                max_y = min(array.shape[0] - 1, y + radius)#shape 0 get rows

                # Iterate over the search area and check for the number 2
                # since odata is column ordered, in the for loop
                # we need to search the columns first...
                for i in range(min_y, max_y + 1):
                    for j in range(min_x, max_x + 1):
                        if array[i, j] == 1:
                            surrounding_cells = self.check_surroundings(array, i, j, 1)
                            # print(surrounding_cells)
                            # check for explored cells which is 2 in odata:

                            if (j,i) in visited:#j is x, i is y
                                #print('visited',(j,i))
                                continue

                            if 2 in surrounding_cells and 3 not in surrounding_cells and surrounding_cells.count(1)>2:
                                return True, (j,i)  # If found, return True and the tuple of coordinates
            return False, (max_radius,'no frontier') # If not found within max_radius, return False and max_radius

        except Exception as e:
            print('in search radius')
            print(e)
```

The function “search_radius” searches frontier around the robot and checks if it has not been added into the “visited” list, it will then pick the frontier if is around an explored area and no walls are present around it by using “check_surroundings” function.

```python
    def spin_test(self):
        time.sleep(0.6)
        rclpy.spin_once(self)
        time.sleep(0.6)
        rclpy.spin_once(self)

        timeout = time.time() + 5
        while True:
            rclpy.spin_once(self)
            if time.time()>timeout:
                break

    def search_frontiers(self):
        rclpy.spin_once(self)
        try:
            print('searching for frontiers...')
            rclpy.spin_once(self)
            time.sleep(0.2)
            grid = self.odata
            # print(grid)
            #change robot coordinates to the grid
            #robot_grid = self.get_grid(self.pos_x, self.pos_y)
            #print(f'robot\'s grid coordinates: {robot_grid}')

            # look for frontiers around the robot grid
            #start search at robot grid position
            found, coor = self.search_radius(grid, self.grid_x, self.grid_y, 150 )
            print(found, coor)
            goal_x, goal_y = self.get_pose(coor[0], coor[1])
            print(f'goal x and y in RW Pose: {goal_x}, {goal_y}')
            return goal_x, goal_y, found

        except Exception as e:
            print('tried searching frontiers but got Exception:')
            print(e)
```

The function “spin_test” is used for initialising values and because we find that the robot does not update values fast enough at the beginning of running the program.

```python
    def pick_direction(self):
        rclpy.spin_once(self)
        self.get_logger().info('In pick_direction')
        deadend = False
        angles = []
        angle = 0
        print(self.laser_range.size)
        if self.laser_range.size != 0:
            length_laser_array = len(self.laser_range) #which will be less than 360 ~250+
            # print(f'laser range arraY: {self.laser_range}')
            # print(f'length of array: {length_laser_array}')

            search_distances = []

            for i in search_angles:
                #do some transformation to get estimated angles and index in the laser_range
                index = int( i*length_laser_array/360)
                search_distances.append(self.laser_range[index])

            pairs = dict(zip(search_angles, search_distances))
            # print(f'pairs of angle/distance: {pairs}')

            for a in pairs:
                if pairs[a]>search_distance and a not in angles:
                    angles.append(abs(a))

            print(f'angles available: {angles}')
            rclpy.spin_once(self)

            #if there are angles within the
            if len(angles)!=0:
                angle = min(angles, key=abs)   #to avoid error
                for a in sorted(angles):
                    if pairs[a] == pairs[-a]:
                        continue
                    elif pairs[-a] > pairs[a]:
                        angle = -a -30
                        # break out of the loop if you want to search from the centre angle
                        # then remember to add offset
                        break
                    elif pairs[a] > pairs[-a]:
                        angle = a +30#+ some offset
                        break

                #angle= min(abs(angle) for angle in angles if angle!=0)
                print(f'angle: {angle}')
                time.sleep(0.3)
```

The function “pick_direction” is our obstacle avoidance algorithm. It will first look at the search_angles (which is from the -60 degree to 60 degree of lidar readings to search distance bigger than search_distance (a parameter we declared to determine if the direction is free). It will pick the first angle that has distance bigger than search distance.

```
            else:
                rclpy.spin_once(self)

                #check to the left of the turtlebot, and see if the distance is more  to the left than to the right
                rightAngle = int(90*length_laser_array/360)
                total = 0
                counter = 0
                for i in range(rightAngle-2,rightAngle+3):
                    if np.isnan(self.laser_range[i]):
                        continue
                    else:
                        total += self.laser_range[i]
                        counter += 1
                if counter != 0:
                    left_index = total/counter
                else:
                    left_index = 0 #check for zero error

                total = 0
                counter = 0
                for i in range(-rightAngle-2,-rightAngle+3):
                    if np.isnan(self.laser_range[i]):
                        continue
                    else:
                        total += self.laser_range[i]
                        counter += 1
                if counter != 0:
                    right_index = total/counter
                else:
                    right_index = 0 #check for zero error

                print(f'left {left_index}, right {right_index}')
                if left_index > search_distance:
                    if left_index > right_index:
                        angle = 90
                        print('LEFT!!!')
                        print(f'distance to the left: {left_index}')
                        print(f'distance to the right: {right_index}')
                        rclpy.spin_once(self)
                elif right_index > search_distance:
                    angle = -90
                    print('RIGHT@@!!')
                    print(f'distance to the left: {left_index}')
                    print(f'distance to the right: {right_index}')
                    rclpy.spin_once(self)
```

If all the search_angles are not “free”, it will then check the -90th degree and 90th degree of the robot. We look at 5 values around the angle and take the average because we found that the angle sometimes will have readings registered as “NAN” which affects the algorithm. If the 90th degree (left side) of the robot is free and has a distance bigger than the -90th degree(right side) of the robot, it will pick that angle, else it will be vice versa.

```
                else:
                    print('reverse')
                    lidar = np.nanargmax(self.laser_range) #reverse to original path
                    angle = lidar/len(self.laser_range)*360
                    if angle>0:
                        angle += 10
                    elif angle>180:
                        angle -= 10
                    deadend = True
                    print(angle)

            else:
                #reverse
                angle = 180
                self.get_logger().info('No data!')

            rclpy.spin_once(self)

            # rotate to that direction
            self.rotatebot(float(angle))
            time.sleep(0.3)
            rclpy.spin_once(self)

            # start moving
            self.get_logger().info('avoid')

            #CHECK THIS
            rclpy.spin_once(self)
            lri = (self.laser_range[front_angles]<float(stop_distance)).nonzero()
            if len(lri[0])==0:
                print('direction clear!')
                self.movebot()
            else:
                self.pick_direction()

            while len(lri[0])==0:
                print(f'avoiding pos,yaw:{self.pos_x},{self.pos_y},{self.yaw}')
                lri = (self.laser_range[front_angles]<float(stop_distance)).nonzero()
                rclpy.spin_once(self)

                if ((self.pos_y < door_y + box_thres and self.pos_y > door_y - box_thres)
                   and (self.pos_x < door_x + box_thres and self.pos_x > door_x - box_thres)):
                    print('robot at door')
                    self.stopbot()
                    break
```

Lastly, if all the previous angles checked are not free, it will pick the angle with the biggest free distance from the lidar readings and declare a variable named “deadend” as true. Alternatively, the robot will pick an angle of 180 if no lidar readings are available.

The robot will then rotate to the selected angle and move if the front of the robot is cleared or else it will perform obstacle avoidance one more time. We decided to code it to perform one more time because we saw that sometimes the robot rotated not enough so the obstacles were still in the way.

In the while loop, the robot will continue moving and only stops if there are distances of the front angles that are smaller than stop_distance or if the robot’s coordinates are at the door position.

```

            if (len(lri[0])>0):
                self.stopbot()
                print('stopped avoiding in pick direction')
                if deadend:
                    print('getting out deadend')
                    self.pick_direction()
                    deadend = False
                break

```

Additionally, we have a variable called “deadend” that will be set to true if it rotated towards the direction with the biggest free distance, this is to make the robot trigger the algorithm once more for it to get out of a stuck situation.

```python
    def rotateTo(self, x_goal, y_goal):
        try:
            rclpy.spin_once(self)
            x1, y1, x2, y2, current_yaw = self.pos_x, self.pos_y, x_goal, y_goal, self.yaw
            print(f'rotating to goal: {x_goal}, {y_goal}')
            print(f'current robot coordinates and yaw: {x1}, {y1}, {self.yaw}')
            yaw_difference, distance = calculate_yaw_and_distance(x1, y1, x2, y2, current_yaw)
            yaw_difference = yaw_difference / math.pi * 180
            self.rotatebot(yaw_difference)
            rclpy.spin_once(self)

        except Exception as e:
            print('in rotateTo')
            print(e)
```

The function “rotateto” is to make the robot rotate towards the position that is inputted in.

```python
    def moveTo(self,x_goal,y_goal):
        try:
            rclpy.spin_once(self)
            x1, y1, x2, y2 = self.pos_x, self.pos_y, x_goal, y_goal
            print(f'current robot coordinates and yaw: {x1}, {y1}, {self.yaw}')
            counter = 0
            x_goal_grid,y_goal_grid = self.get_grid(x_goal,y_goal)
            goalx,goaly = x_goal,y_goal
            while not ((self.pos_y < y2 + box_thres and self.pos_y > y2 - box_thres)
               and (self.pos_x < x2 + box_thres and self.pos_x > x2 - box_thres)):
                rclpy.spin_once(self)

                rad,dis = calculate_yaw_and_distance(self.pos_x, self.pos_y, x_goal, y_goal, self.yaw)
                angle  = int(math.degrees(rad))
                target_angle = int((angle-self.yaw)/360*len(self.laser_range))
                print(f'distance to target, lidar distance to target {dis},{self.laser_range[target_angle]}')

                #remove this CHUNK
                lri = (self.laser_range[range(target_angle-5,target_angle+6)]<float(dis+free_dis_thres)).nonzero()
                if not np.isnan(self.laser_range[target_angle]):
                    if(len(lri[0])==0):
                        print('target is free')
                        self.rotatebot(angle-self.yaw)
                elif dis>1.5:
                    print('chcecking target faraway')
                    total = 0
                    num = 0
                    for i in range(target_angle-5,target_angle+6):
                        if np.isnan(self.laser_range[i]):
                            continue
                        else:
                            total += self.laser_range[i]
                            num += 1
                    if num != 0:
                        target_index = total/num
                    else:
                        target_index = 0
                    if target_index>1:
                        print('free distance to target faraway')
                        self.rotatebot(angle-self.yaw)

                rclpy.spin_once(self)

                print(f'trying to move to goal... {x2}, {y2}')
```

The “moveTo” function is our goal navigation algorithm. This function consists of a while loop where the loop will break when the robot is at the target position.

Inside the loop, it will first check if the direction towards the target is free by comparing the distance from lidar readings with the distance between the robot and the target + free_dis_threshold (to have some room for error). If the direction towards the target is “free”, the robot will rotate towards the target. Else, if the target is faraway from the robot (>1.5m), it checks if the distance from the lidar readings in the target direction is > 1m to rotate towards the target.

```
637                 # before moving if there is stuff in front, avoid it
638                 lri = (self.laser_range[front_angles]<float(stop_distance)).nonzero()
639                 if len(lri[0])!=0:
640                     print('but target is blocked')
641                     self.pick_direction()
642                 else:
643                     self.movebot()
644 
645                 lri = (self.laser_range[front_angles]<float(stop_distance)).nonzero()
646                 while len(lri[0])==0:
647                     print(f'current robot coordinates and yaw: {self.pos_x}, {self.pos_y}, {self.yaw}')
648                     if goalx == door_x and goaly == door_y:
649                         if ((self.pos_y < door_y + box_thres and self.pos_y > door_y - box_thres)
650                         and (self.pos_x < door_x + box_thres and self.pos_x > door_x - box_thres)):
651                             print('robot at door')
652                             self.stopbot()
653                             break
654                     rclpy.spin_once(self)
655                     lri = (self.laser_range[front_angles]<float(stop_distance)).nonzero()
656                     if(len(lri[0])>0):
657                         self.stopbot()
658                         break
659 
660                 # if the list is not empty
661                 counter += 1
662                 rclpy.spin_once(self)
663 
664                 #COMMENT THIS OUT I THINK UNNECESSARY
665                 self.rotateTo(goalx, goaly)
666 
667                  #CHANGE THIS TO DETERMINE HOW MANY TIMES TO AVOID
668                 if counter > 1:
669                     break
670 
671         except Exception as e:
672             print('exception in moveTo function: ')
673             print(e)
674         finally:
675             # stop moving
676             self.stopbot()
677 
```

If the front of the robot has obstacles, it will perform obstacle avoidance or else it will move straight.

Then, it will enter a nested while loop. The while loop will break if there are obstacles in front of robot. It will also break when the robot is at the door position if the `goalx` and `goaly` inputted is the door position. We included the additional checking of the `goalx` and `goaly` to be the door position because we don't want the loop to break everytime it reaches the door position if the target is not the door which will cause the robot to be unable to move.

Lastly, the counter will be increased by 1. The counter is used for forcefully breaking the loop when counter > 1 because we do not want the robot to be stuck in going to the same target for many loops.

Additionally, we added the “rotateTo” function at the last moment because we saw that it was necessary to have the robot to keep rotating towards the target to help the robot move towards the target position.

```python
    def exploration(self):
        try:
            map_bins = height_map/self.map_res * width_map/self.map_res
            while rclpy.ok():
                rclpy.spin_once(self)
                print('visit %', self.map_visit/map_bins)
                if self.map_visit/map_bins < map_percent:
                    goalx, goaly, found = self.search_frontiers()
                    if found == False:
                        goalx, goaly = door_x, door_y

                #going to door
                else:
                    print('moving to door')
                    goalx, goaly = door_x, door_y

                    # door reached...
                    if ((self.pos_y < door_y + box_thres and self.pos_y > door_y - box_thres)
                       and (self.pos_x < door_x + box_thres and self.pos_x > door_x - box_thres)):

                        '''calibration, check x and y and make sure it is accurate'''

                        rclpy.spin_once(self)
                        print(f'current yaw: {math.degrees(self.yaw)}')
                        print('ADJUSTING ANGLE AT DOOR')
                        self.rotatebot(90-math.degrees(self.yaw))
                        time.sleep(1)

                        # URL to which the request will be sent
                        url = (f"http://{esp32_ip}/openDoor")

                        # Data to send in the request (if any)
                        data = {"action": "openDoor", "parameters": {"robotId": "{TurtleBot3_ID}"}}

                        # Send HTTP POST request
                        while True:
                            response = requests.post(url, json=data)
                            time.sleep(1)
                            # Print the response received
                            print("Response from ESP32:", response.text)
                            door_ans = response.json()["data"]["message"]
                            if door_ans == "door1":
                                self.rotatebot(90)
                                break
```

(variable) response: Response

The “exploration” is the main function that we are running in the program. It starts off by first calculating the mapped percentage of the map by taking the number of explored bins divided by the maze size. It then checks if the self.map_visit/map_bins which is the current map percentage is smaller than the map_percent which is a threshold that we set so that it performs frontier search, else it will set the goal as door position (“door_x, door_y”)

Then, there is an “if” condition that checks whether the robot is at the door so that it will perform a HTTP call. After getting an answer of “door1” or “door2”, the robot will rotate towards the chosen door.

```
            elif door_ans == "door2":
                self.rotatebot(-90)
                break
            else:
                time.sleep(60)
                response = requests.post(url, json=data)

        time.sleep(4)
        twist = Twist()
        twist.linear.x = speedchange
        twist.angular.z = 0.0
        self.publisher_.publish(twist)

        time.sleep(3)


        timeout = time.time()+door_time
        while time.time()<timeout:
            rclpy.spin_once(self)
            lri = (self.laser_range[range(-10,10)]<float(stop_distance)).nonzero()
            if(len(lri[0])!=0):
                print('bucket in front')
                self.stopbot()

        self.stopbot()

        #find bucket
        rclpy.spin_once(self)
        time.sleep(1)
        print(self.laser_range)
        if self.laser_range.size != 0:
            dis = 100
            for i in range(-70,70):
                ang = int(i/360*len(self.laser_range))
                if dis>self.laser_range[ang]:
                    dis = self.laser_range[ang]
                    lr2i = i
        else:
            lr2i = 0
            self.get_logger().info('No data!')

        # rotate to that direction
        print(lr2i)
        self.rotatebot(float(lr2i)-self.yaw)
```

The robot will then start moving through the door. The “timeout” is the time that we had determined for the robot to move towards the centre of the robot. Inside the while loop, it will check if there is an obstacle(or the bucket) in front of it and stop.

After it stops, it will check the range of lidar readings from -70 degree to 70 degree to find the shortest distance which we assumed will be where the bucket is at and rotate towards that direction.

```
                # start moving
                self.get_logger().info('Start moving')
                twist = Twist()
                twist.linear.x = 0.05
                twist.angular.z = 0.0
                # not sure if this is really necessary, but things seem to work more
                # reliably with this

                self.publisher_.publish(twist)
                rclpy.spin_once(self)

                while self.laser_range[0] > dist_threshold:
                    rclpy.spin_once(self)

                self.stopbot()

                #launching!
                launch_ball()
                break

        print(f'robot coordinates, yaw: {self.pos_x}, {self.pos_y}, {self.yaw}')
        print(f'goalx, goaly: {goalx}, {goaly}')

        self.rotateTo(goalx, goaly)
        rclpy.spin_once(self)

        lri = (self.laser_range[front_angles]<float(stop_distance)).nonzero()
        # if the list is not empty
        if(len(lri[0])>0):
            print('obstacles detected')
            rclpy.spin_once(self)
            self.pick_direction()

        else:
            self.moveTo(goalx, goaly)

        x,y = self.get_grid(goalx, goaly)

        for i in range(x-2,x+3):
            for j in range(y-2,y+3):
                visited.add((i,j))
        print('added',x,y)
        # print('vnumber of visited points',len(visited))
        rclpy.spin_once(self)
```

The robot will then move in that direction until it is right in front of the bucket (when the 0th degree of lidar reading is <=`dist_threshold` which is 0.16). Lastly, the robot will drop the ping pong balls and the program will then finish.

Continuing from the part after the goal is being selected, the robot will then rotate towards the target. If the front of the robot is blocked, it will perform obstacle avoidance. Else, the robot will run the goal navigation algorithm, the “moveTo” function.

Once the movement algorithm ends, it will add the target previously selected into the “visited” list so that it will not select the same target again.

The exploration ends when it finishes depositing the ping pong balls.

```
        except Exception as e:
            print('in mover')
            print(e)
        # Ctrl-c detected
        finally:
            # stop moving
            self.stopbot()
            self.get_logger().info('end robot coordinates -- x,y: %f, %f' % (self.mapbase.x, self.mapbase.y))
            print('mission finished! SU!')

def main(args=None):
    rclpy.init(args=args)
    solver = Solver()
    solver.spin_test()
    solver.movebot()
    time.sleep(startMove)
    solver.stopbot()

    solver.exploration()
    GPIO.cleanup()
    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    solver.destroy_node()

    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

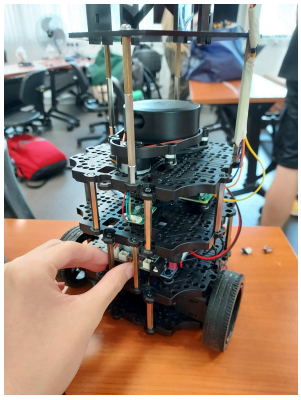
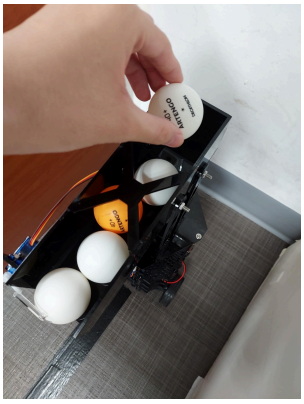
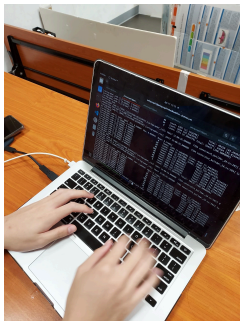
In the main function, the program will first run “spin_test” function to ensure the data is updated, then it will move straight for a fixed time to ensure slam map is updated, lastly it will then run the “exploration” function, which is the main function to carry out the entire mission.

Reflection:

It was tough coding all these out. We had to familiarise ourselves with ROS 2 which we knew nothing about and we had to spend countless days and nights to code out each function one by one, then test the algorithm to find bugs and problems and debug it until it can work properly. We managed to produce this within the span of 6 weeks, we worked on it until the very last moment to produce this finalised version of our code which we are really proud of. It was a really great journey in EG2310 and an unforgettable one for us who put in all the blood, sweat and tears to see the Turtlebot, “our child” carrying out the mission challenges.

10. System Operation Manual

Instructions: (from End User Documentation)

1) Flash ESP32 with your wifi and password, turn on Turtlebot 	2) Load ping pong ball, modify the ESP32 IP and map parameters in the program file 	3) Run rosbu, rslam, ros2 run auto_nav map2base, then run the program file in the Ubuntu terminal: ros2 run lidar solver2 
4) Let the robot explore the maze. When the threshold is reached, it will transition to 'going to door' mode. 	5) Go to door, go into room and launch ping pong 	6) Stop the robot, remove it and clear the maze. Mission complete 

10.1 Running the program

1. Make sure laptop and Pi both connected to same network (hotspot)
2. Open 1st Terminal on laptop and start ROS on the Rpi

```
sshrp # ssh into pi
rosbu # run this command after successful ssh into pi
```

3. Open 2nd Terminal on your computer to take in LiDAR data and create a map in RViz.

```
rslam
```

4. Open 3rd Terminal: to run a node that publishes the coordinates of the robot in the map. The map's origin is where the robot is initialised when rslam is called.

```
ros2 run auto_nav map2base
```

10.2 Mission

1. Place your bot in the starting position inside the maze and load the ping pong balls onto the ping pong ball dropper.
2. Open 4th Terminal: run the file in Rpi to start the mission

```
ros2 run auto_nav solver2
```

11. Troubleshooting

11.1 Hardware and electronics

1. Payload is loose or shaky - fasten all screws tightly
2. Servo does not rotate as intended
    a. Ensure wires are fully connected to Raspberry Pi - female heads fully fitted onto GPIO pins
    b. Ensure wires are connected to the correct GPIO pins
3. Raspberry Pi is not booting up / not accessing SD card - check that the green light is blinking
4. Ensure OpenCR switch is on and that the Raspberry Pi is connected to the OpenCR via the GPIO pins for 5V power

11.2 Software

1. The laptop unable to connect to Raspberry Pi
    a. Make sure that laptop and raspberry pi are on the same network. Make sure not to use networks that run on VPN like NUS's wifi as it causes some connectivity issues.
2. No map received - restart rslam and rosbu, try rosbu first then rslam.
3. Connection refused - check whether all devices are on the same hotspot, restart hotspot
4. RViz not starting up properly - restart rosbu
5. RViz map is creating many overlays; robot position and frames glitching

```
cd /opt/ros/foxy/share/Turtlebot3_cartographer/config
vim Turtlebot3_lds_2d.lua
# set use_odom to False
```

6. RPi is unable to connect to the WiFI - Ensure that the WiFi is 2.4 GHz compatible as the RPi used, RPi 4B+ cannot connect to 5GHz WiFi.
7. ROS2 topic list on laptop is not synchronised with RPi - Ensure that `ROS_DOMAIN_ID` in `~/.bashrc` of both devices are the same

12. Future Scope

General

Our methods to accomplish this mission were overly reliant on our software navigation algorithm. Exploring the maze, getting to the door and finding the bucket were all dependent on our algorithm, which could have been avoided by using some markers to lighten the burden.

Software

1. One hard challenge that we could not solve (not that we necessarily had to) was that even if the space was enough for the robot to pass through between two walls, it could not do so. This was just something to accept for the time being, because we needed to scan the front few angles to ensure that the robot stayed within about 0.2 meters of the walls in front of it. However, we could technically implement an algorithm that looks for openings and could accurately maneuver through them by searching for the center of the opening. However, this is quite difficult to do, due to the **imperfections with the LiDAR**, and with the turning and wheel rotation accuracy.
2. Implement a **path planning algorithm**. Our algorithm is relatively dumb, and in more complex walls without right-angled walls it would be harder to work well. This is because we need to take into account more cases that could make the robot move in a loop. Using a path planning algorithm like Djikstra's would allow us to attain true navigation capabilities since the robot can actually follow actual waypoints to a goal, rather than simply moving in the 'general direction'. Using a cost map to inflate walls when implementing the pathfinding algorithm would replace the need for our convoluted obstacle avoidance algorithm, and reduce the risk of our robot moving away from the goal in the obstacle avoidance loop.
3. Our method of finding the frontiers is too inconsistent and imperfect. We think we can improve it by **searching for areas with the largest frontier** instead so that it doesn't pick any random single unexplored cell as a frontier to go to. The frontier that is being selected influences our robot movement and navigation, therefore selecting the correct frontier would help the robot to navigate through the maze smoothly.

Mechanical and hardware

1. We are still interested in **implementing the rubber band launcher** as it is our initial design and we spent a lot of time designing it, hence we might try to improve our manufacturing skills and use the suitable equipment to manufacture the rubber band launcher in future. Besides, none of the groups this year used mechanisms that shoot the ping pong balls into the bucket and hence we feel that it will be more interesting and impressive if we managed to implement it in real life.

2. One problem with our current design of ping pong ball dropper is that it raises the centre of gravity of Turtlebot and it makes it unstable and has the risk of toppling over. A way to mitigate that issue is changing our design to **a catapult-like structure with a container for the ping pong balls** at the end of the arm. The arm rotates up to reach the height of the bucket when it reaches the bucket. This mitigates the problem of the robot being unstable due to high centre of gravity when the robot is moving as the container is kept at the bottom of the bot when it's moving.

References

The Bug2 algorithm for robot motion planning – automatic Addison. (n.d.). Automaticaddison.com. Retrieved April 25, 2024, from https://automaticaddison.com/the-bug2-algorithm-for-robot-motion-planning/

Chinenov, T. (2019, February 26). Robotic Path Planning: RRT and RRT*. Medium. https://theclassytim.medium.com/robotic-path-planning-rrt-and-rrt-212319121378

Robotic Path Planning. Path Planning. (n.d.). https://fab.cba.mit.edu/classes/865.21/topics/path_planning/robotic.html

Shieldsquare Captcha. (n.d.). https://iopscience.iop.org/article/10.1088/1755-1315/804/2/022024/pdf

(N.d.-b). Datacamp.com. Retrieved April 25, 2024, from https://www.datacamp.com/tutorial/making-http-requests-in-python

(N.d.-a). Cmu.edu. Retrieved April 25, 2024, from https://www.cs.cmu.edu/~motionplanning/papers/sbp_papers/integrated1/yamauchi_frontiers.pdf

Pykes, K. (2023, February 10). Python HTTP request tutorial: GET & post HTTP & JSON requests. DataCamp. https://www.datacamp.com/tutorial/making-http-requests-in-python

Vaidehi Joshi (2017, April 10). Breaking Down Breadth-First Search. Medium. https://medium.com/basecs/breaking-down-breadth-first-search-cebe696709d9

Ros 2 documentation. ROS 2 Documentation - ROS 2 Documentation: Foxy documentation. (n.d.). https://docs.ros.org/en/foxy/index.html

Annex

Annex A (Calculations for rubber band payload mechanism)

Calculations

Initial velocity required to achieve height of the bucket:

Assuming 30% kinetic energy lost to air resistance

H=0.325m

$m=2.7*10^{-3}$ kg

$g=9.81$ Nm^2kg^{-2}

$0.7(\frac{1}{2}mu_y^2) = mgh$

$u_y=3.018$ ms^{-1}

$u=3.018/\sin 55$

$=3.684$ ms^{-1}

Force exerted by the rod:

Assuming contact time of rod and pingpong to be 0.01s

By $F\Delta t = m(v-u)$,

Assuming time of contact is 0.01s,

$F(0.01) = 2.7*10^{-3}*(3.684-0)$

$F = 0.995$ N

Spring constant of rubber band required:

Measured maximum elongation 24mm

(include free body diagram)

$F = kx - mg$

$0.995 = k(24*10^{-3}) - (3*10^{-3})(9.81)$

$k = 42.684$ N/m

Calculations

Maximum torque required:

$T = F * r$

$= (42.684)(14.1*10^{-3})(6*10^{-3})$

$= 3.611*10^{-3}$ Nm

Hence motor selected is **SM-S4303R**

Specification:

Operation Voltage: 4.8 - 6.6VDC

Torque: 4.8kg.cm at 6V = **0.47Nm**

Max 70 rpm

Air Resistance

Velocity of ping pong, v = 3.684m/s, Ping pong radius = 20mm, Ping pong mass = 2.7g

Drag force: $F = 0.5*C_D\rho Apv^2$

Assuming contact area is sphere when ball is moving through the air,

Area, $A = 4*pi*(0.02)^2 = 0.005027m^2$

Drag coefficient, $C_D=0.47$ (for rough sphere)

Air density, $\rho = 1.2041kg/m^3$(at 20°C)

$F = 0.5*0.47*0.005027*1.204*3.684^2 = 0.0193N$(Very small)

Volume of ping pong, $V = 4/3*pi*(0.02)^3=0.0000335m^3$

Mass of air displaced by ping pong, $M_{air} = 1.204*V = 0.00004035kg$

$M_{air}/M_{ping\ pong}$ = 0.01494(very small fraction of ping pong, insignificant)

We may assume air resistance is small and negligible for the launch of ping pong

Trajectory

$a_y=-9.81m/s^2$, initial $v = 3.684m/s$

Launching angle: 55°

Initial position(x,y) = 0m, 0.04m

Initial velocity(v_x,v_y) = 5.269cos(55°)m/s , 5.269sin(55°)m/s

Ignoring air resistance, trajectory equations:

$X = 5.269\cos(55°)t$

$Y = 5.269\sin(55°)t - 0.5*9.81t^2$

Max height:

$Y_{max} = (3.684\sin(55°))^2 / (2*9.81) = 0.4642m$(greater than the height of bucket)

$X_{max} = 5.269\cos(55)*0.8799 = 2.6593m$

Annex B (Modified Turtlebot3 Specifications)

Robot System Name: Cheesy Boiii
System Purpose: To explore an unknown maze with unknown elements, navigate to 2 doors that are electronically locked, make a HTTP call to open them, and deposit balls into a bucket inside one room

---

Section 1: User Instructions

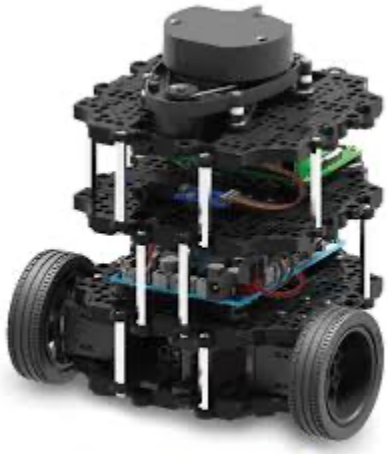
Hardware Specifications

	List	Specifications	Notes
Turtlebot	Dimensions (mm) (LxWxH)	226mm x 176mm x 410mm	Full assembly (Turtlebot3 + Ping pong dropper)
	Weight (kg)	Turtlebot: 0.9kg Ping pong dropper: 0.222kg Total weight: 1.122kg	
	Wheel Base (mm)	65mm	From front of DYNAMIXEL to the back of ball caster
Ping pong dropper	Dimensions (mm) (LxWxH)	234mm x 138mm x 196mm	Including Damper
	Weight (kg)	0.222kg	Including Damper
	SG90 Servo Motor	0.009kg	
	Damper	0.056kg	Damper Bracket + Damper Bar + 50g weight
ESP 32 relay (external)	Dimensions (mm) (LxWxH)	48 x 26 x 11.5 mm	Connected to laptop
	Weight (kg)	0.01kg	
System	Battery Capacity	LiPo Battery 11.1V 1,800mAh	
	Expected Average Operating Time	2 hours 30 minutes	
	Communication Interfaces	I2C, HTTP, GPIO, ROS	

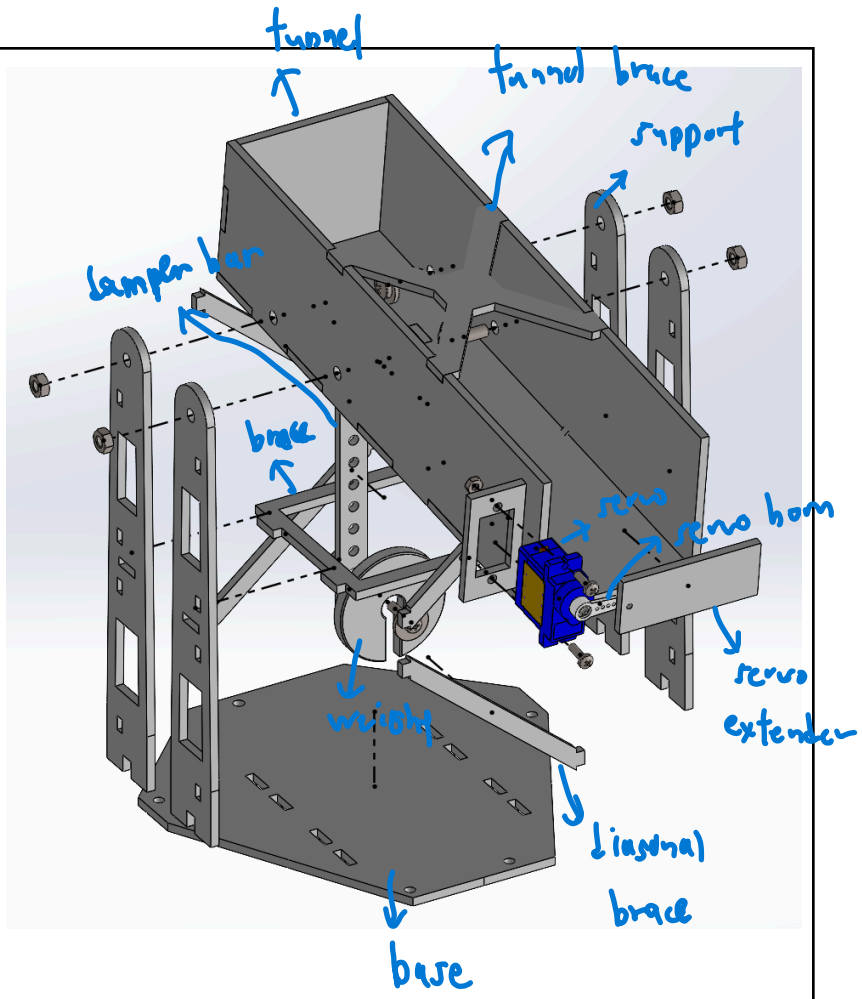
Section 2: Assembly Instructions

The ROBOTIS official manual provided in the official Turtlebot 3 Burger Kit was used to construct the primary body and the different waffle plate layers.

Item No.	Qty	Part Name
1	1	LIDAR
2	1	RPi
3	1	USB2LDS
4	1	OpenCR
5	2	Tire
6	2	DYNAMIXEL Motors
7	1	Lipo Battery
8	1	Ball Castor



Item No.	Qty	Part Name
1	1	Base (Acrylic)
2	4	Supports (Acrylic)
3	1	Support Brace (Acrylic)
4	4	Support Diagonal Brace (Acrylic)
5	1	Tunnel (Acrylic) (L x W x H: 200mm x 60mm x 50mm)
6	1	Tunnel Brace (Acrylic)
7	1	SG90 Servo
8	1	Servo Horn with Extender (Acrylic + Wood)
9	1	Damper Bracket (Acrylic)
10	1	Damper Bar (Acrylic)
11	1	Weights (50g)



Ping Pong Ball Dropper is an additional component that we have added to the Turtlebot.

Section 3: Acceptance Defect Log

Defect	Defect Classification		
	Critical	Major	Minor
Ball dropper platform is slightly shaky when the lidar is spinning			x
Servo motor may make a vibrating noise when turned on			x

Glue for acrylic supports coming off		x	
LiDAR registers NAN values for certain angles			x
LiDAR does not have values for 360 degrees			x

Section 4: Factory Acceptance Test

Component	To be checked	Observation
OpenCR	Able to be powered by the LiPo Battery	Green LED lights up when connected to a power source, boot up tune being played
RPi	Able to turn on the RPi when connected to OpenCR	Red light turns on while green light flashes
	Can be connected to from the remote laptop	Terminal returns “Welcome to Ubuntu…” when “sshrp” is run on laptop
Lidar	Able to spin and collect data consistently	Environment will be mapped on Rviz
SG90	Able to rotate the stopper to allow ping pong ball to drop	Ball drops down vertically when the stopper attached to the servo rotates 90 degrees counterclockwise
Wheels	Able to move the bot in all directions freely	Bot can be controlled properly when running ‘rteleop’
Ball caster	Able to roll in all directions freely	Turtlebot is able to move around in all direction smoothly with a ball caster attached
ESP 32	Able to connect to our network	Will make a “click” sound when we make http call from Rpi or laptop
Damper	Securely attached to the ping pong dropper and able to swing freely	Damper swings when the tunnel is shaking
Structural stability	Structural platforms and components installed correctly	Shake Turtlebot to verify all components are mounted securely
	Verify all fasteners installed and tightened	Make sure all fasteners are tightened and secure
Battery	Battery fully charged	Check battery level using battery tester

Section 5: Maintenance and Part Replacement Log

Log No.	Defect Date	Qty	Defect Component	Problem Description	Rectification	Close Date
1	30-3-24	1	Open-CR	Odom Frame is drifting in the Rvizz and it is not aligned with base link frame	Replaced Open-CR	31-3-24
2	18-4-24	1	Servo SG90	Servo motor was hot and short circuited	Replaced servo	18-4-24

Annex C (Monetary Budget)

Bill of Materials (Rubber Band Launcher)

Components	Unit Price($)	Quantity	Total Price($)
Continuous Rotation Servo (MG995)	10.36	1	10.36
3D-Print	5/hr	6 hr	30
PVC pipes	6.5/m	0.3m	1.95
Nuts (M4, M2)	0.04, 0.15	6, 2	0.54
Bolts (M4, M2)	0.04, 0.15	6, 2	0.54
Washers (M3)	0.03	6	0.18
Retaining Rings (M3)	0.87	6	5.22
Total Cost			48.79

Bill of Materials (Ping Pong Ball Dropper)

Components	Unit Price($)	Quantity	Total Price($)
SG90 Servo	2.90	1	2.9
Acrylic Parts (Base, Supports*4, Support Brace, Support Diagonal Brace*4, Tunnel, Tunnel Brace, Servo Horn Extender, Servo Bracket, Damper Bracket, Damper Bar) [3mm acrylic sheets sourced from fabrication lab (FOC)]	5	1	5
M4*10 Pan Head Cross Screws	0.04	4	0.16
M4 Washers	0.06	4	0.24
M4 Nuts	0.04	4	0.16
M3*25 Pan Head Cross Screws	0.09	2	0.18
M3 Washers	0.04	4	0.08
M3 Nuts	0.06	2	0.12
M2*10 Pan Head Cross Screws	0.23	3	0.69
M2 Nuts	0.01	3	0.03
Total Cost			9.56

Annex D Code for ESP32

```
#include <WiFi.h>
#include <WebServer.h>
#include "ArduinoJson.h"

// Replace with your network credentials
const char* ssid = "";
const char* password = "";

// Create a web server on port 80
WebServer server(80);

// Assign output variables to GPIO pins
const int pinDoor1 = 16;
const int pinDoor2 = 17;

void handleOpenDoor(String doorToOpen) {
  // Normally Closed configuration, send HIGH signal to let current flow
  if (doorToOpen == "door1") {
    digitalWrite(pinDoor1, HIGH);
```

```
    digitalWrite(pinDoor2, LOW);
  } else {
    digitalWrite(pinDoor1, LOW);
    digitalWrite(pinDoor2, HIGH);
  }
}

void handleCloseDoor() {
  // Normally Closed configuration, send LOW signal to stop current flow
  digitalWrite(pinDoor1, LOW);
  digitalWrite(pinDoor2, LOW);
}

void handleRequest() {
  if (server.hasArg("plain") == false) {
    // If the request does not have a body
    server.send(400, "application/json", "{\n\t\"status\":\"error\",\n\t\"message\":\"Bad Request - No Data Received\"\n}\n");
    return;
  }

  // Parse JSON object from request
  DynamicJsonDocument doc(1024);
  deserializeJson(doc, server.arg("plain"));
  String action = doc["action"];
  String robotId = doc["parameters"]["robotId"];

  // Check the action value
  if (action == "openDoor") {

    // Randomly decide which door to open
    String doorToOpen = random(1, 3) == 1 ? "door1" : "door2";

    // open the door
    handleOpenDoor(doorToOpen);

    // Send response back to TurtleBot3
    String response = "{\n\t\"status\":\"success\",\n\t\"data\":{\n\t\t\"message\":\"" + doorToOpen + "\"\n\t}\n}\n";
    server.send(200, "application/json", response);

    Serial.println("Command received to open " + doorToOpen);

    // set 60s delay after request
    delay(60000);

    // close the door
    handleCloseDoor();

  } else {
    server.send(400, "application/json", "{\n\t\"status\":\"error\",\n\t\"data\":{\n\t\t\"message\":\"Invalid Action\"\n\t}\n}\n");
  }
}

void setup() {
  Serial.begin(115200);

  // Initialize the output variables as outputs
  pinMode(pinDoor1, OUTPUT);
  pinMode(pinDoor2, OUTPUT);

  // Normally closed (NC): The circuit is complete when the switch is not operated.
  // Set outputs to LOW
  digitalWrite(pinDoor1, LOW);
  digitalWrite(pinDoor2, LOW);

  Serial.println(ssid);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  // Print local IP address and start web server
  Serial.println("");
  Serial.println("WiFi connected.");
  Serial.println("IP address: ");
```

```
  Serial.println(WiFi.localIP());
  server.begin();

  Serial.println("Connected to WiFi");

  // Define endpoint and corresponding handler function
  server.on("/openDoor", HTTP_POST, handleRequest);

  // Start the server
  server.begin();
  Serial.println("Server started");
}

void loop() {
  // Handle client requests
  server.handleClient();
}
```