Innovation and Design Programme

EG2310

Charly Chandra	A0281413M
Edward Humianto	A0255862W
John Howard Wijaya	A0276087R
Karthik Sivasubramanian	A0273653Y
Seow Zhi Yong, Justin	A0272159Y

1. Problem Definition - The “Elevator” Problem

1.1 The Map

a. Robot must be able to move through the mazes
b. Temporary markers are allowed to identify each zone and points of interest

Objective	Requirements
a.	The robot must map out the maze elements and develop a full SLAM map of the area
b.	Temporary markers are allowed to identify each zone and points of interest

1.2 The “Lift” Door

The 2 door is secured by an electronic lock

a. Lock is wired to a magnetic system that unlocks via a web server
b. If the request is valid, the HTTP response will reply with the ID of the unlocked door

Objective	Requirements
a.	The robot will need to make a HTTP call to a server
b.	The server will reply with a standard HTTP response
c.	The robot needs to go to the unlocked door

1.3 The Bucket

a. The team is given 5 standard sized ping pong balls
b. The robot is to fire ping pong balls into an open bucket
c. The projectile must land into the bucket

Objective	Requirements
a.	The robot must create a system to fire the projectiles, ping pong balls
b.	The robot must have a system to launch the projectiles

2. Literature Review

2.1 Turtlebot Navigation Algorithm

1. Depth-First Search (DFS)
   Graph traversal algorithm that explores as far as possible along each branch before backtracking once the robot meets a dead end. It employs a stack to keep track of vertices to visit and systematically explores the deepest unexplored nodes first. DFS is often used for pathfinding but may not guarantee finding the shortest path.

2. Tremaux's Algorithm
   It is a maze-solving method in which a robot crosses through the maze while marking passages passed. Each passage is marked differently based on specific rules. If a passage is traversed for the first time, it is marked. However, should the same passage be crossed again, it would be marked differently. Upon reaching a junction, if all passages are marked, the junction is marked as "done". Hence, this ensures exploration of all paths while avoiding undesired repetition which may result in not finding the shortest and most efficient path.

3. Wall Follower
   The wall follower algorithm is a maze-solving technique where the robot follows a continuous wall (either left or right) as it navigates through the maze. The strategy involves keeping the robot's sensors in contact with the wall, determining the direction of the wall, and consistently turning in the same direction at junctions.

4. A* Algorithm
   This is a heuristic search algorithm used for pathfinding and graph traversal. It combines the data or cost to reach a node with the estimated cost to reach the goal from the same node through a heuristic function which gives off potential paths in a graph. Thus, this algorithm guarantees finding the shortest path efficiently.

5. Waypoint Algorithm
   The path of the robot to travel from any starting position to the destination should be predetermined. The robot will first determine its current position, and then use the set of programmed waypoints to determine and travel to the next location until it reaches the final destination. This guarantees the most efficient path as the waypoints are programmed manually.

6. Line Follower
   The robot follows a predetermined path marked by a black line on the ground. The robot uses sensors, typically infrared sensors, to detect the line and adjust its movements accordingly. By continuously monitoring the position of the line relative to the robot, adjustments are made to maintain the robot's alignment and stay on the desired path.

7. Frontier Navigation
   The robot searches for frontier points, points of boundary between the mapped and unmapped regions in order to fully map the area. The robot will continuously travel to all frontier until the area is fully mapped (no frontier left). This is achieved by accessing the occupancy map of the robot.

2.2 HTTP Call

HTTP protocol is used to transfer data from a client to a server. There are two main methods to consider:

1. GET method
   Carries request parameter in URL string. Only a limited amount of data can be transferred between client and server. Signal generated is idempotent, producing the same result despite several repetition of operation procedures.

2. POST method
   Carries the request parameter in a message body. Large amount of data can be transferred between client and server. Signal generated is non-idempotent.

2.3 Bucket Detector

1. Color Sensor TCS3200/230
   This sensor senses color utilising an 8 x 8 array of photodiodes. The photodiodes readings are converted into a square wave using a Current-to-Frequency Converter. Then, the Arduino Board can read the square wave output and get the results for the color. This sensor can be used to sense the red color, which is the color of the bucket.

2. OpenCV Object Detection
   Perform object detection by using Raspberry Pi Camera and OpenCV library. Coco library contains trained models of multiple objects and animals. The robot can actively search for the bucket by using this system.

3. Line Follower
   Using a predetermined path that is either hard coded or laid out using a black tape, the robot will follow the path towards its target until it's given a signal to stop either from its LiDAR detecting a collision with the bucket or the tape ending.

2.4 Ping Pong Ball Launcher

1. Sling-shot Mechanism
   The robot uses the potential of stretched elastic material to propel a projectile towards the target. The material has been stretched beforehand and on command, a mechanism that was previously maintaining the material stretched is released, allowing the projectile to launch towards the target.

2. Wheel Accelerated Mechanism
   The projectile is shot through a track after going through a series of motors. The motors are equipped with a form of traction (a flywheel) to grip onto the projectile and are activated to rotate at high speeds. The traction between the motors and the ball causes the projectile to accelerate and be shot in the direction of the target.

3. Piston / Flick Design
   Using an electric piston or solenoid the robot provides a quick and powerful impulse to the projectile. The impulse given allows the projectile to launch from the robot towards its target.

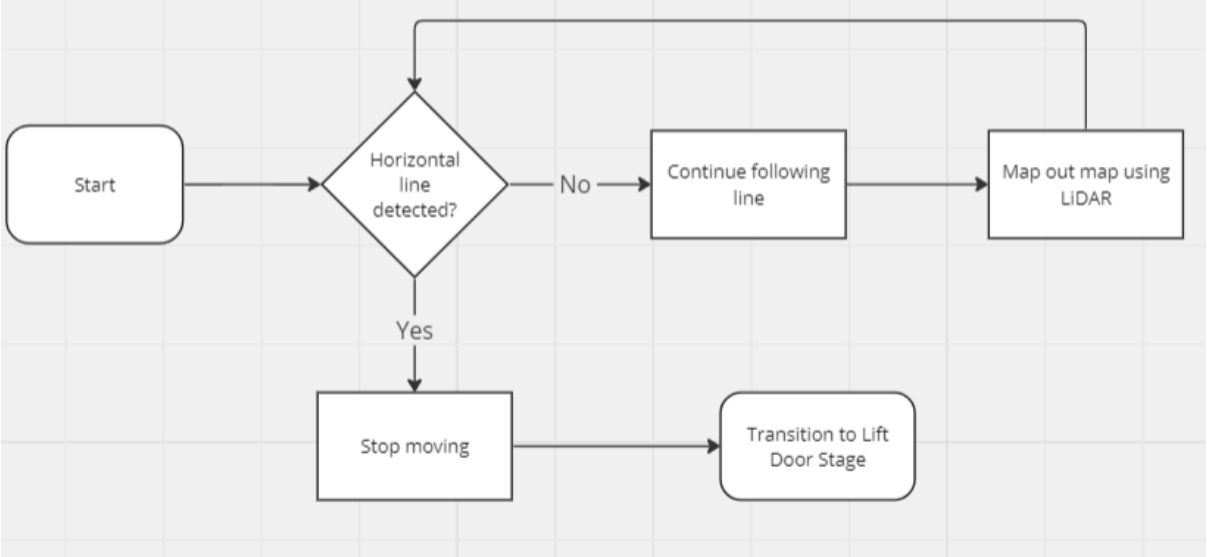
4. Ball dropper
   The robot will be equipped with a helical ping pong ball container mounted on its top. This container will have a servo-controlled hole on the bottom, which opens and closes to release of the balls. The robot will navigate towards the bucket and position itself such that the bottom hole aligns directly over the bucket, while ensuring the rest of the robot does not make contact with the bucket. Once in position, the servo will activate, flicking open the hole and dropping the balls into the bucket.

3. Concept Design

The main objective of this stage is to explore various possible solutions to solve the mission requirements and come up with a high-level concept design.

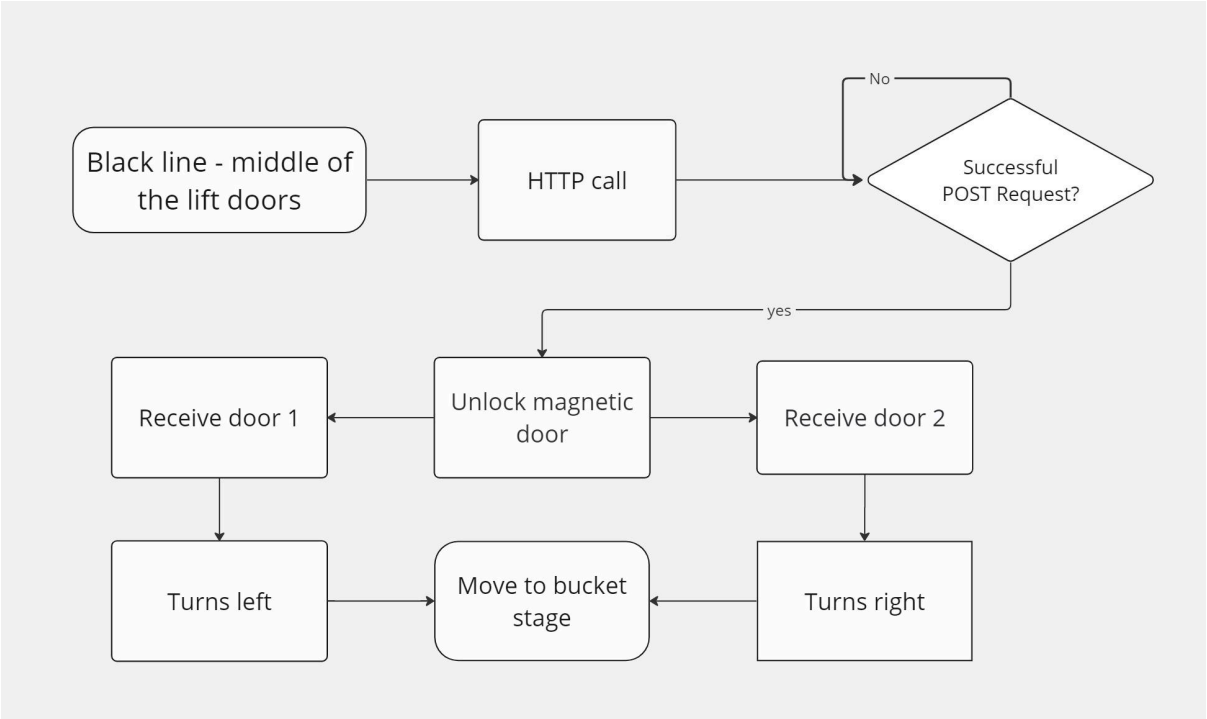
3.1 Line Following Navigation Stage

Before the mission begins, black tape will be placed strategically to navigate through a randomly arranged maze. The robot will be positioned on this tape and will execute a line-following algorithm to trace the path through the maze until it reaches the lobby. At this point it will detect a horizontal black line, prompting it to stop, and transition to the Lift Door Stage. While the robot is following the designated path, it will also map out the section of the map that it traverses through.



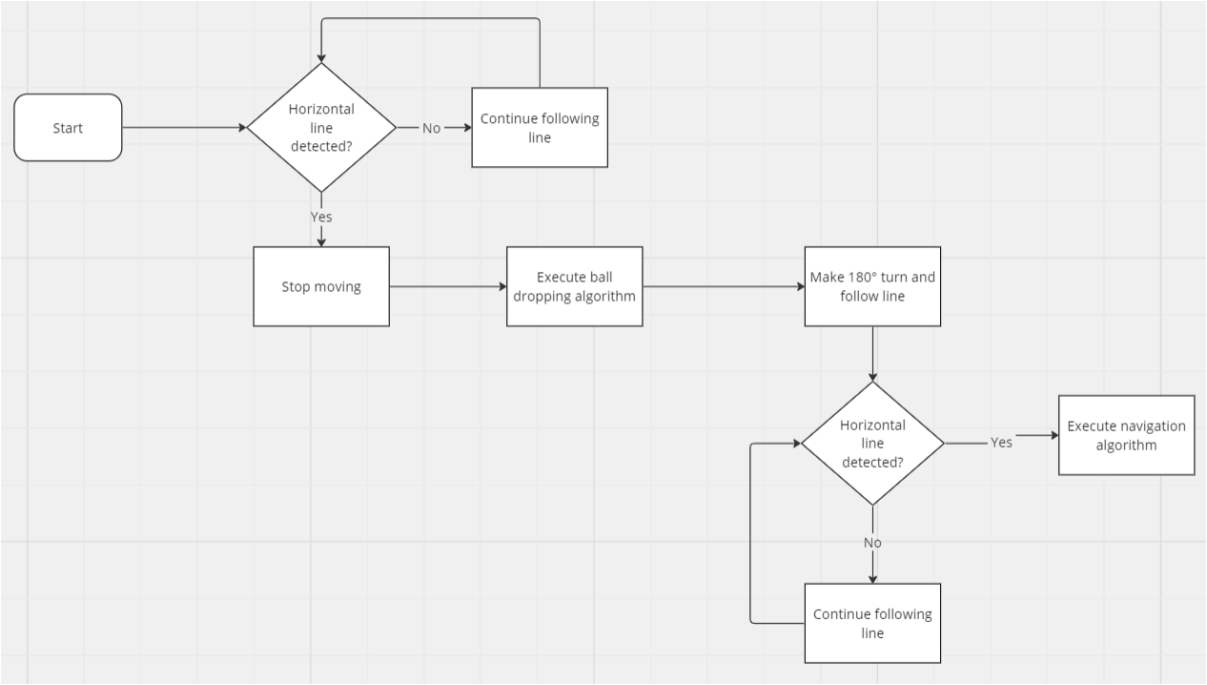
3.2 Lift Door Stage

Upon reaching the horizontal black line at the maze's exit, the robot will send a HTTP POST request to an operational ESP32 server. The server will process this request and respond with either "door1" or "door2". Based on the server's directive, the robot will either turn left for "door1" or right for "door2". Paths marked with black tape will lead the robot in either direction towards the designated doors. The robot will then proceed, using its line-following algorithm, to navigate towards and push open the corresponding door, entering the designated room, and advancing to the Bucket Stage.



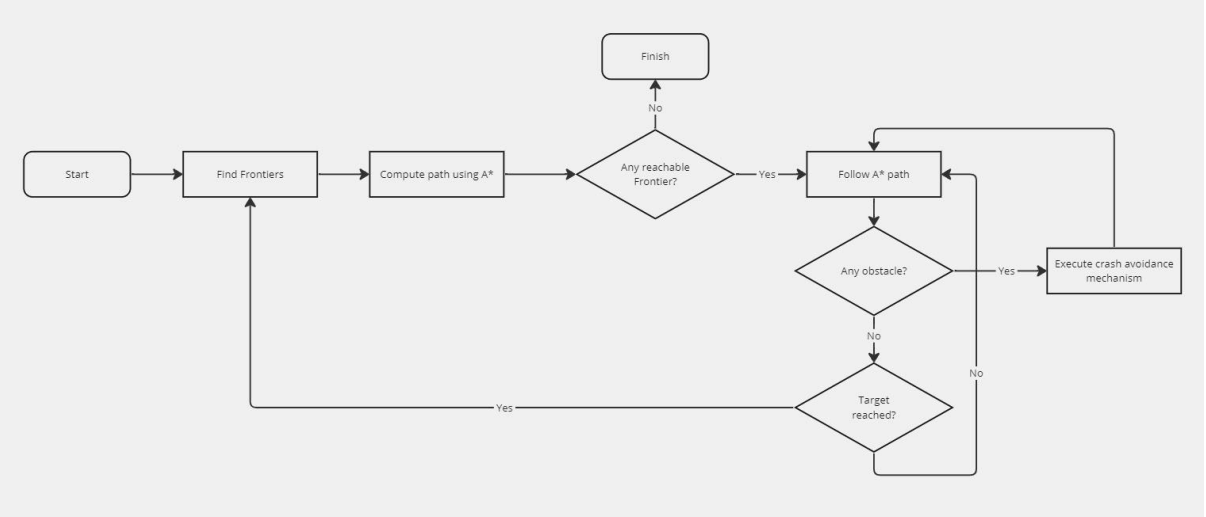
3.3 Bucket Stage

Once in the room, the robot will use a line follower to follow a line that leads directly to the bucket. Once the line follower detects a horizontal line, it will stop and execute the ball dropping algorithm which drops the ping-pong balls into the bucket. Afterwards, the robot will then make a 180° turn and follow the line back to the lobby. The robot will sense the first horizontal line, and then transition back to mapping in the Autonomous Navigation Stage.



3.4 Autonomous Navigation Stage

Upon entering this stage, the turtlebot will execute its Frontier and A* algorithm to explore and map the whole maze. First, it will identify which maze region has not been mapped by executing the Frontier algorithm. Then, the robot will utilise A* to compute the shortest path to the frontier point. The robot will then travel to the nearest frontier point, unlocking new frontiers, and the whole cycle repeats until no frontier is found. When no frontier is left, this marks the end of the navigation stage as the whole maze has been fully mapped.



4. BOGAT

4.1 Turtlebot Navigation Algorithm

- Random Maze: DFS is not suitable for our mission since it does not guarantee covering the whole map. DFS may have a chance to compute a direct path to the lobby, leaving the rest of the maze unmapped. Wall-follower is also not suitable since the algo will not work for complex mazes (mazes with unconnected walls). Frontier and A* is chosen to ensure the maze is fully mapped, despite taking longer computation time.

Autonomous Navigation Algorithm			
Algorithm	**DFS**	**Wall-follower**	**Frontier and A***
Pros	Simple to implement as it only needs the robot to backtrack once meets a dead end	Simple to implement as the robot will continuously detect any present wall on its left	Frontier guarantees a full map exploration while A* guarantees finding the shortest path to a given frontier point
Cons	May enter an infinite loop if not appropriately modified to handle cycles	May not find the exit for mazes with unconnected walls	Both algorithms require large computational power, causing longer delays

- Lift Door Navigation: A* and waypoint algorithms are inefficient as both need high computing power and do not guarantee to enter the door precisely. We chose to do line following as it is the fastest way to enter the lift door.

Lift Door Navigation Algorithm			
Algorithm	**A***	**Waypoint**	**Line Follower**
Pros	Guarantees finding the optimal path when used with appropriate heuristics.	Relatively simple to implement as the waypoint just needs to be pre-determined	The robot will be able to detect the lines accurately and go through the door well
Cons	Consume significant memory and processing resources	May cause problems if the robot does not start in a predetermined position	Time-consuming and tricky to paste the lines

4.2 HTTP Call

Method	**GET**	**POST**
Pros	Easy to use and idempotent which makes it safe for retrieval operations	Can send a larger amount of data, thus more suitable for complex interactions.
Cons	Has limitations on the amount of data that can be sent in the request.	Non-idempotent, thus multiple signals may result in a changed operation taking place

4.3 Bucket Detector

OpenCV consumes a lot of computational power which may not be necessary for this mission. The robot will first need to detect the bucket with OpenCV, align it perfectly, and move towards the bucket. This mechanism is prone to error as the robot may not be fully aligned with the bucket, hence, causing more risk to complete the ball dropper stage. Therefore, we chose to use line follower as it is more simple, accurate, and utilises significantly less computational power.

Hardware	**Line Follower**	**OpenCV Object Detection**
Pros	Able to detect a horizontal line which then tells the robot to stop just in front of the bucket	Able to distinguish and detect various objects accurately
Cons	Time consuming and tricky to place tape at the exact distance required to the bucket	Needs longer processing time and requires more computational power

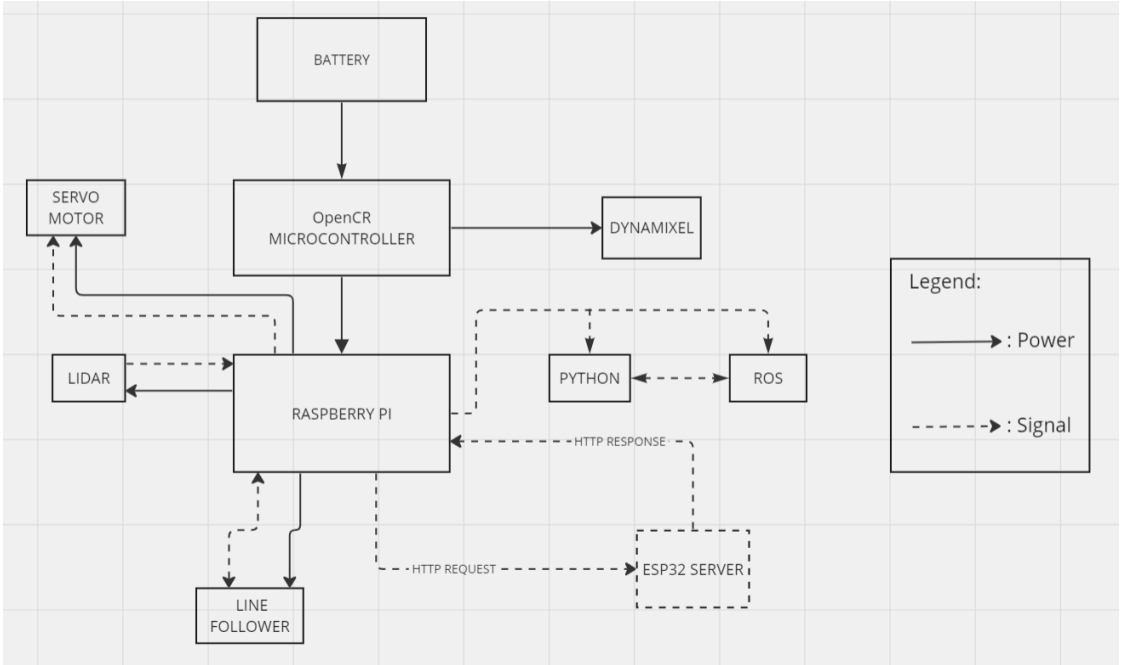
4.4 Ping Pong Ball Launcher

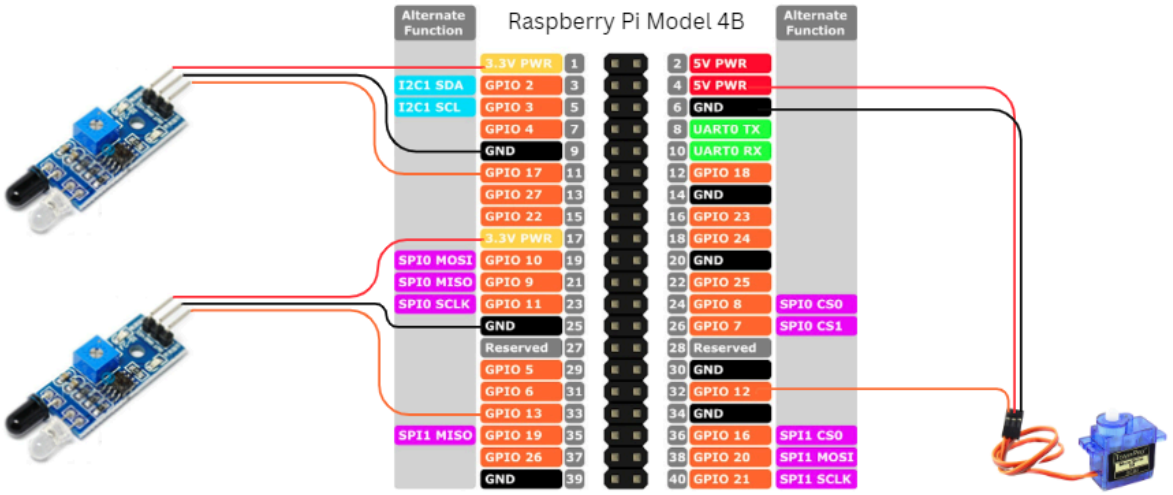
Mechanism	**Sling-shot**	**Wheel-accelerated**	**Piston/Flick design**	**Ball dropper**
Pros	It is a simple mechanism which depends on the elasticity of material	Simple to assemble and troubleshoot	It provides a quick impulse to propel the projectile by using an electric piston	It gives a reliable drop, without the inconsistent nature of projectile motions.
Cons	Difficulty in reloading/resetting and elasticity of material may degrade over time	Unable to reliably control the speed of the projectile and possibility of projectile escaping	There is a chance of ball bouncing off the bucket	Robot must stop precisely in front of the bucket, with the possibility of colliding if the robot does not stop in time.

5. Preliminary Design

The preliminary design phase, which follows the concept design stage, is dedicated to further elaborate and enhance the ideas and concepts previously identified. During this phase, the primary objective is to present a comprehensive overview of how energy and signals will be managed within the system. Additionally, it involves organising and classifying the subsystems according to their respective phases and processes.

5.1 Functional Block Diagram





A line sensor will be connected to Raspberry Pi's GPIO pins 13 and 17. The Turtlebot will start its mission by following a line through until the lift door stage. While following the line, the Turtlebot will map out the part of the maze that it passes through. It will eventually hit a horizontal line, at which point an interrupt will be sent to the RPi to stop the bot. The Turtlebot will then send a HTTP post request to the ESP32 server. Once the Turtlebot receives the door information from the server, it will either turn left or right depending on the door that opens. It will then continue following the line, pushing through the door, until it reaches a second horizontal line placed right in front of the bucket. The Turtlebot will stop, and then execute its ball dropping mechanism.

A helical ball holder will be mounted to the top of the Turtlebot onto one waffle plate. A servo motor will be attached to the bottom of this ball holder, to act like a door. Once the bot reaches the bucket, the RPi will send a signal to the servo, causing it to flick open, depositing the balls into the bucket. This servo will be connected to the 5V pin, GND pin and GPIO pin 12 of the RPi.

5.2 Phases

Phase	Line Following Navigation	HTTP Server Call	Lift Door Navigation	Ball Dropper	Autonomous Navigation
Systems	Line follower system, LiDAR system	HTTP Post Request System	Line follower system	Ball Dropper System	Turtlebot Navigation System
Objectives completed	Traversing through the maze while developing the SLAM map	Perform an HTTP call and receiving the door code from the ESP32 server	Moving to the correct unlocked lift door	Able to drop the ping-pong balls into the bucket	Complete mapping of the maze including any hidden rooms or unmapped areas
Hardware	IR sensors, LiDAR	Raspberry Pi	IR sensors, LiDAR	Servo motor	LiDAR
Software	Line follower algorithm, Rviz, SLAM	POST request system	Line follower algorithm	Ball dropper mechanism	SLAM, A* + Frontier algorithm

6. Prototyping & Testing

6.1 Autonomous Navigation Algorithm

6.1.1 Obtaining Frontier Clusters

The main idea for the navigation is Frontier exploration, where the robot scans the surrounding map and utilises its occupancy data to search for frontier points. In this case, a frontier is defined as a known open area which is adjacent to an unknown area. The robot operates via moving towards the closest frontier and scanning for new ones once it reaches the new location. This process repeats until there is no more frontier or an end goal is reached (no frontiers left implies that the maze has been fully mapped).

```
Require: queue_m // queue, used for detecting frontier points from
    a given map
Require: queue_f // queue, used for extracting a frontier from a
    given frontier cell
Require: pose // current global position of the robot

 1: queue_m ← ∅
 2: ENQUEUE(queue_m, pose)
 3: mark pose as "Map-Open-List"

 4: while queue_m is not empty do
 5:    p ← DEQUEUE(queue_m)

 6:    if p is marked as "Map-Close-List" then
 7:       continue
 8:    if p is a frontier point then
 9:       queue_f ← ∅
10:       NewFrontier ← ∅
11:       ENQUEUE(queue_f, p)
12:       mark p as "Frontier-Open-List"

13:       while queue_f is not empty do
14:          q ← DEQUEUE(queue_f)
15:          if q is marked as {"Map-Close-List","Frontier-Close-
             List"} then
16:             continue
17:          if q is a frontier point then
18:             add q to NewFrontier
19:             for all w ∈ adj(q) do
20:                if w not marked as {"Frontier-Open-
                   List","Frontier-Close-List", "Map-Close-List"}
                   then
21:                   ENQUEUE(queue_f,w)
22:                   mark w as "Frontier-Open-List"
23:          mark q as "Frontier-Close-List"
24:       save data of NewFrontier
25:       mark all points of NewFrontier as "Map-Close-List"
26:    for all v ∈ adj(p) do
27:       if v not marked as {"Map-Open-List","Map-Close-List"}
          and v has at least one "Map-Open-Space" neighbor then
28:          ENQUEUE(queue_m,v)
29:          mark v as "Map-Open-List"
30:    mark p as "Map-Close-List"
```

Frontier Detection Pseudocode

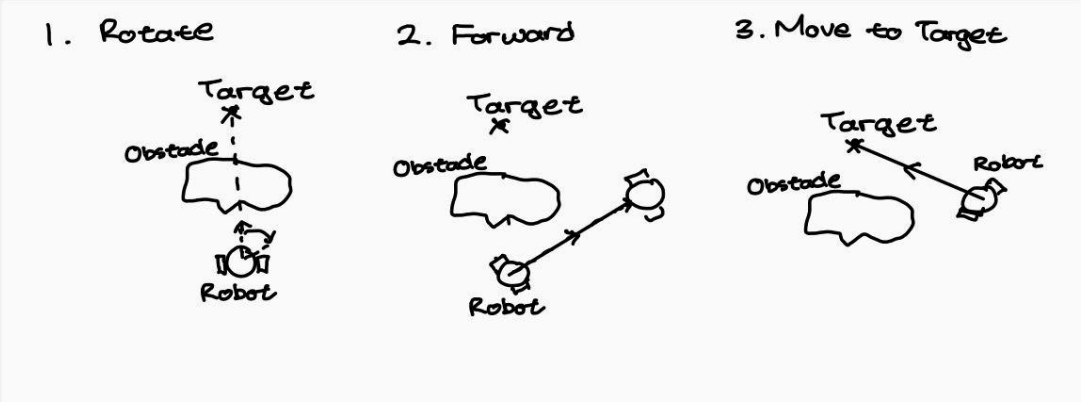
The first iteration of the algorithm utilizes the occupancy array obtained after processing the LiDAR data. Occupancy array contains the information of whether a certain grid is *unknown* (1), *empty* (2), or *occupied/wall* (3). Using the attached pseudocode, the turtlebot was able to use the data it subscribed from the *'map'* topic to calculate and group frontier points into specific clusters. Utilizing the *'tf2'* function, the algorithm connects the bases of *'base_link'* and *'map'* to obtain the relative position of the robot. The algorithm then finds the median point of the frontiers and calculates the closest one to the robot's position. The target position coordinate is then converted to meters relative to the map's origin position.

```python
# Function to find frontiers
def findfrontiers(odata,posi,map_res):
    frontiers = []
    markmap ={}

    # check whether a position is a frontier
    def isFrontier(pos):
        if odata[pos[0],pos[1]] == 2:
            if pos[0] ==0:
                temp = odata[:2,:]
            elif pos[0] == len(odata)-1:
                temp = odata[pos[0]-1:,:]
            else:
                temp = odata[pos[0]-1:pos[0]+2,:]

            if pos[1] == 0:
                a = temp[:,:2]
            elif pos[1] == len(odata[0])-1:
                a = temp[:,pos[1]-1:]
            else:
                a = temp[:,pos[1]-1:pos[1]+2]
            return np.any(a==1)
        return False

    # check if a cell has been marked
    def mark(p):
        return markmap.get(p,'Unmarked')

    # get all adjacent cells
    def adj(p):
        ans = []
        for i in range(-1,2):
            for j in range(-1,2):
                if (i==0 and j ==0) or p[0]+i < 0 or p[1] + j < 0 or \
                    p[1] + j > len(odata[0]) -1 or p[0]+i > len(odata) -1:
                    continue
                ans.append((p[0]+i,p[1]+j))
        return ans

    # create the map queue
    qm = []

    # add robot's current position to the map queue
    qm.append(posi)
    markmap[posi] = "Map-Open-List"

    while qm:
        # dequeue first element of map queue
        p = qm.pop(0)

        # skip the element if it has already been mapped
        if mark(p) == 'Map-Close-List':
            continue

        # check if the cell is a frontier
        if isFrontier(p):

            # queue for finding adjacent frontiers
            qf = []
            # list of frontier points
            nf = []
            qf.append(p)

            # mark the point as to be opened by the frontier
            markmap[p] = "Frontier-Open-List"

            while qf:
                q = qf.pop(0)

                # skip the cell if it's already been mapped or it has been searched
                if mark(q) in ["Map-Close-List","Frontier-Close-List"]:
                    continue
                # if the point is a frontier, add it to the nf arrray
                if isFrontier(q):
                    nf.append(q)
                    # check for cells adjacent to current frontier point
                    for w in adj(q):
                        # if there is a neighbor that has not been added to qf,
                        # and has not been mapped or checked add it to the queue
                        if mark(w) not in ["Frontier-Open-List",'Frontier-Close-List',
                                           'Map-Close-List']:
                            qf.append(w)
                            # mark the newly added cell as frontier open list
                            markmap[w] = "Frontier-Open-List"
                # mark the original point as being checked
                markmap[q] = 'Frontier-Close-List'
            # add the list of frontier points to the frontiers array
            if len(nf) > round(threshold/map_res):
                frontiers.append(nf)
            # mark all points in the nf array as being closed by the map
            for i in nf:
                markmap[i] = "Map-Close-List"
        # generate neighbors for the cells in the map queue
        for v in adj(p):
            # add any neighbors that have not been opened or checked
            if mark(v) not in ["Map-Open-List","Map-Close-List"]:
                if any([odata[x[0],x[1]] == 2 for x in adj(v)]):
                    qm.append(v)
                    markmap[v] = "Map-Open-List"
        # mark the point as checked in the map
        markmap[p] = "Map-Close-List"
    return frontiers
```

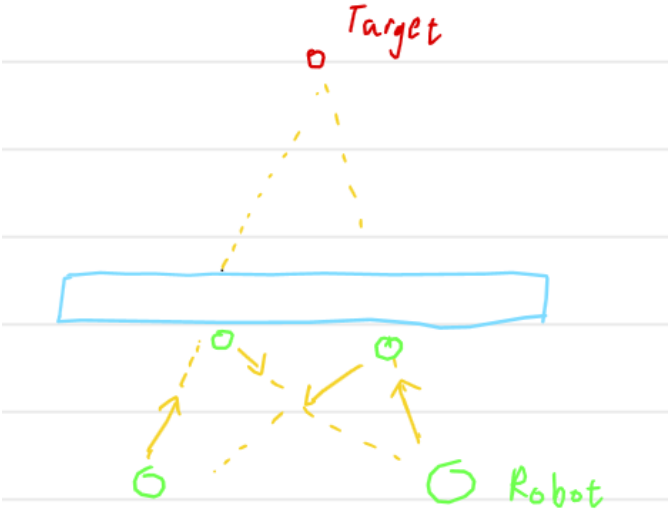
Frontier Implementation

6.1.2 Travel to Frontier and Crash Avoidance

The second iteration of the navigation integrates the robot movement to move to the frontier obtained from the previous iteration by publishing to the *'twist'* topic. It publishes an angular twist to orient the robot towards its target. Once the robot is aligned with the target, a linear twist is published to move the robot towards the target. A wall detection system was also implemented to avoid any nearby walls. This was done by subscribing to the *'scan'* topic of the lidar. We detected incoming collisions by scanning for the angles between -30 to 30 degrees in front of the robot. Once the LiDAR detects an object that is distanced lower than 20 cm in front of the robot's center, the robot will stop and begin to calculate the angle of the obstacle, presumably the angle with the nearest distance. The robot will then scan the angles to the left and right of that obstacle angle, detecting the angles where there is a 35 cm distance from any obstacle. It will then calculate the median of the two values, estimating the general position of the obstacle. Once the direction has been set, the robot will rotate away from that direction until it is not detecting an obstacle at its front angles. It will then move forward for a certain time before continuing navigation. This is effective at moving around small obstacles, but struggles with big obstacles that could cause an infinite loop of self correction.



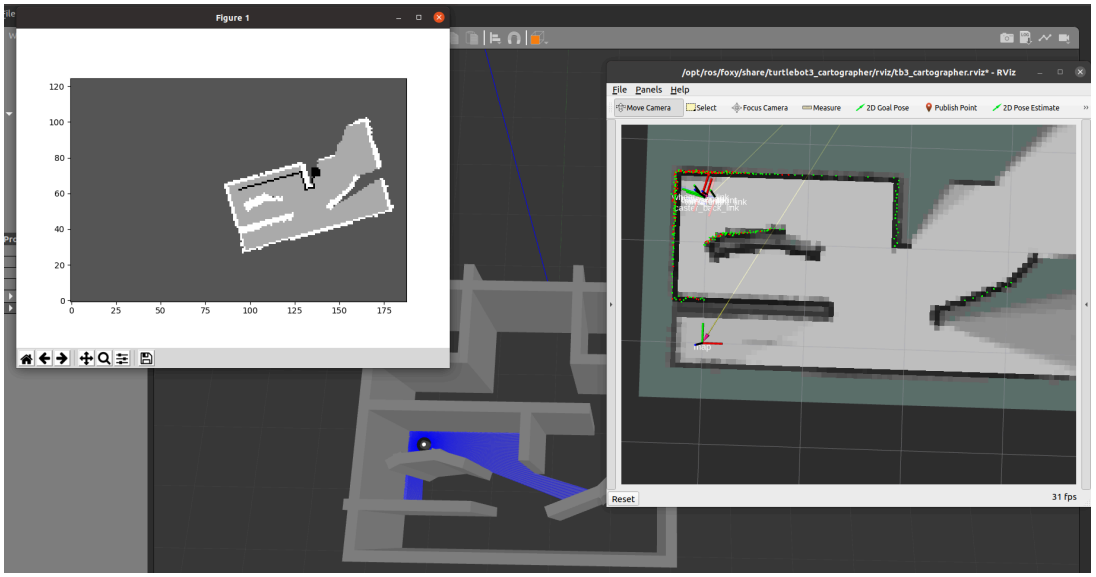
Sequence of Crash Avoidance Mechanism



"Infinite Loop" Problem for Large Obstacles

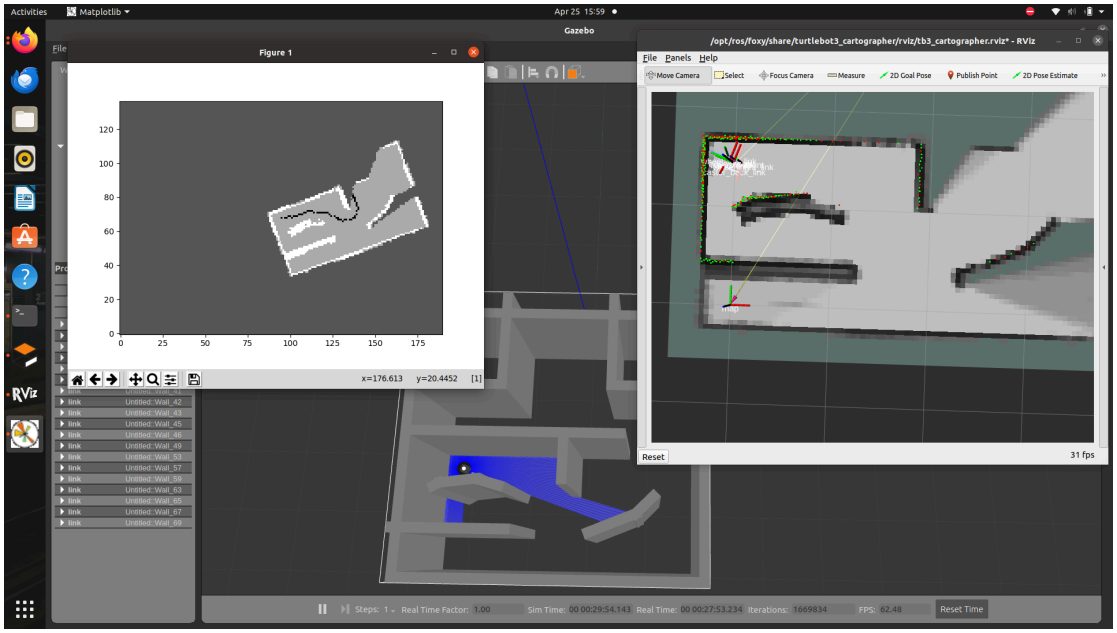
6.1.3 A* Algorithm

For the third iteration of the algorithm, a version of the A* algorithm was used to calculate a possible route for the robot. A* works by using heuristics to guide the robot's projected path towards a certain target. In this case the target was a frontier point in the maze. Our application of A* uses Euclidean distance as the heuristic 'price' of the path finding. This heuristic ensures the shortest path, but the path formed always sticks close to the walls of the maze as shown in the figure (target is denoted by a large black dot, path is denoted by black lines). This can lead to complications where the robot aims for a wall of the maze.



A Problem: Path (black lines) too close to the wall*

Hence, to solve this problem, we updated our heuristic cost to be inversely proportional to the distance to the wall. This implies that the closer a point to the wall, the higher the heuristic cost, and thus, A* will always compute a path that is not too close to the wall.



Modified A Path (black lines)*

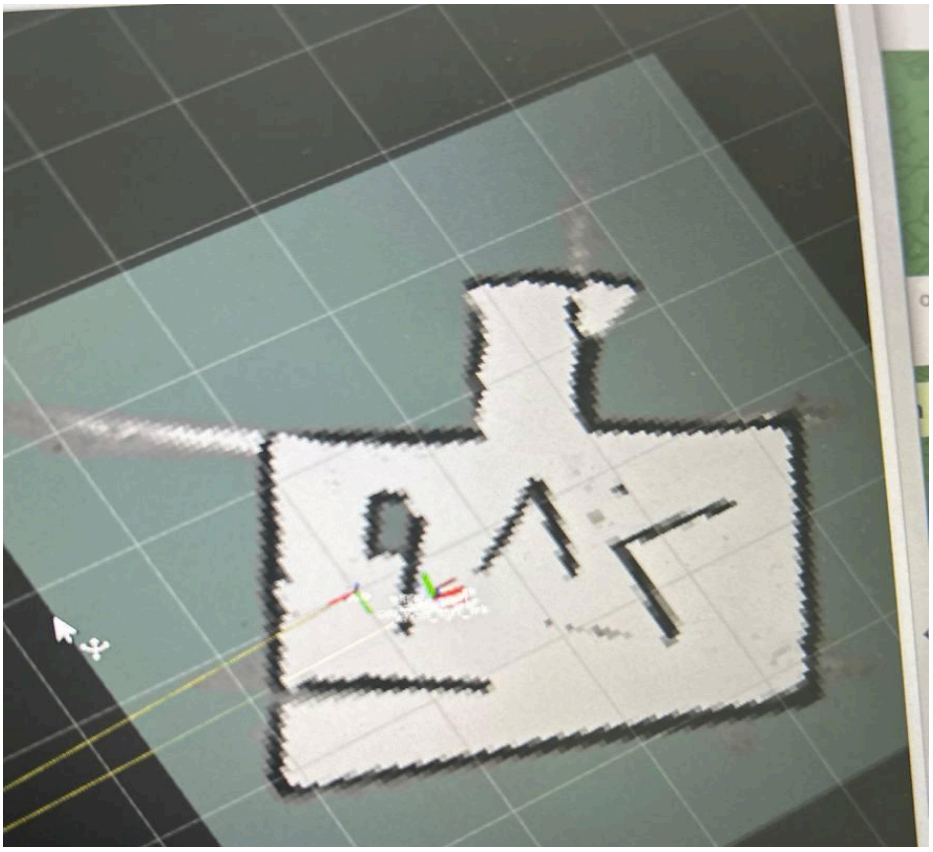
```python
def heuristic(node, target, occupancy_data, map_res):
    nrows = len(occupancy_data)
    ncols = len(occupancy_data[0])
    max_penalty_distance = round(0.25/map_res)
    # Euclidean distance between current node and target
    euclidean_distance = ((node[0] - target[0])**2 + (node[1] - target[1])**2) ** 0.5

    # Penalties for proximity to walls
    wall_penalty = 0 # You can adjust this distance as needed

    for dx in range(-max_penalty_distance, max_penalty_distance + 1):
        for dy in range(-max_penalty_distance, max_penalty_distance + 1):
            nx, ny = node[0] + dx, node[1] + dy
            if 0 <= nx < nrows and 0 <= ny < ncols and occupancy_data[nx, ny] == 3:  # Wall detected
                distance_to_wall = ((dx**2 + dy**2) ** 0.5)
                wall_penalty += max_penalty_distance - distance_to_wall

    # Combine Euclidean distance and wall penalty
    total_heuristic = euclidean_distance + wall_penalty
    return total_heuristic

# Generate neighbors for A*
def find_neighbors(node, occupancy_data, nrows, ncols):
    neighbors = []
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1),(1,1),(1,-1),(-1,1),(-1,-1)]
    for dir in directions:
        neighbor = (node[0] + dir[0], node[1] + dir[1])
        if 0 <= neighbor[0] < nrows and 0 <= neighbor[1] < ncols:
            if occupancy_data[neighbor[0], neighbor[1]] not in  (1,3):  # Check if it is not wall
                neighbors.append(neighbor)
    return neighbors

def a_star(start, target, occupancy_data, nrows, ncols, map_res):
    open_set = []
    closed_set = set()
    heapq.heappush(open_set, (0, start))
    came_from = {}

    g_score = {start: 0}
    f_score = {start: heuristic(start, target,occupancy_data, map_res)}

    while open_set:
        current = heapq.heappop(open_set)[1]

        if current == target:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            path.reverse()
            return path

        closed_set.add(current)

        for neighbor in find_neighbors(current, occupancy_data, nrows, ncols):
            if neighbor in closed_set:
                continue
            # Assuming cost of moving from one cell to another is 1
            tentative_g_score = g_score[current] + 1

            if neighbor not in [x[1] for x in open_set] or tentative_g_score < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = tentative_g_score + heuristic(neighbor, target, occupancy_data,map_res)
                heapq.heappush(open_set, (f_score[neighbor], neighbor))

    return None  # No path found
```

A Implementation*

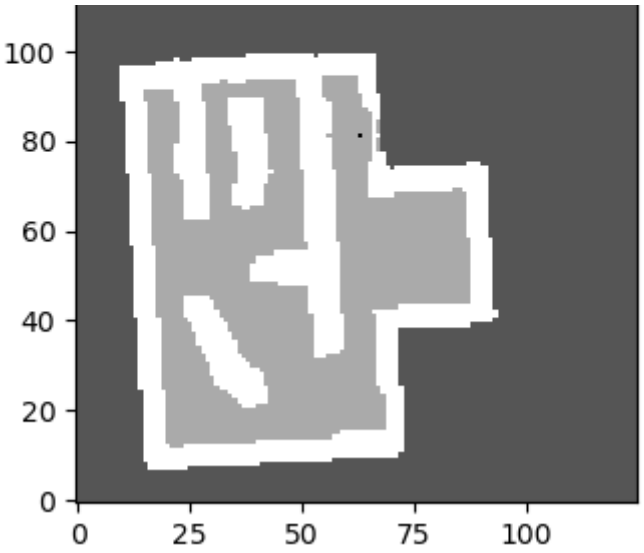
6.1.4 Wall Expansion

Another problem we faced while testing the turtlebot in the real maze is with ‘leakages’ in the RViz map. Leakages are errors in our RViz map that are either caused by unusual values returned by the LiDAR or by holes in the walls the robot is able to see through. This causes troubles as the robot often misinterprets them as frontiers and open spaces. The leakage often causes the robot to create a path that goes through the leak and out of the maze to reach outside frontiers. During our testing, we found out that the best way to prevent any leakages in the map is to virtually expand the walls observed by the robot. Since leaks are

often only caused by a handful of grid tiles being misinterpreted as an open space, expanding the known walls a certain size can fill in the empty spaces caused by the leaks in the occupancy map data. We do this by creating a 2D copy of the original array and expanding the walls by a fixed distance of 6 cm converted into grid size tiles.



Leakages within the map



Expanded walls of the occupancy map

6.1.5 Path Navigation

Once the A* algorithm returns a list of coordinates, we came up with two strategies to navigate the path. This is done to prevent the bot from continuously go through all the A* points one by one. The first idea was to splice the path into “turning points” where the robot needs to make turns or change direction. This was done by calculating the angle of each consecutive point. Despite providing correct points, the algorithm was unreliable as sometimes important points are skipped over and the robot would aim to move through a wall. The second idea was to ‘clear’ all the A* points with distance less than a certain threshold from the bot. By using this way, the bot will move to the next target in an increment of the threshold distance. After it navigates to the next point in the A* path, it will clear all points in a certain radius that are not separated by a wall, travel to the next point, and repeat until there are no more points in the path. This created a much smoother movement and thus, we chose to proceed with the second method.

```
# self.target = self.path.pop(0)
if a_star_list:
    path = []
    for i in range(len(a_star_list)):
        if not path:
            path.append(a_star_list[i])
            try:
                grad = (a_star_list[1][1]-a_star_list[0][1]) /(a_star_list[1][0]- a_star_list[0][0])
            except ZeroDivisionError:
                if a_star_list[1][1]-a_star_list[0][1] > 0:
                    grad = 10
                else:
                    grad = -10
            continue
        try:
            ngrad = (a_star_list[i][1]-a_star_list[i-1][1]) /(a_star_list[i][0]- a_star_list[i-1][0])
        except ZeroDivisionError:
            if (a_star_list[i][1]-a_star_list[i-1][1])  > 0:
                ngrad = 10
            else:
                ngrad = -10
        if i % 10== 0:
            path.append(a_star_list[i])
            continue
        if abs(np.arctan(grad) - np.arctan(ngrad)) < np.radians(5):
            continue
        # if abs(np.arctan(grad) - np.arctan(ngrad)) > np.radians(100):
        #     path.append(a_star_list[i-1][0] + 1, a_star_list[i-1][1] + grad)
        grad = ngrad
        path.append(a_star_list[i-1])
        path.append(a_star_list[i])
    path.append(a_star_list[-1])
```

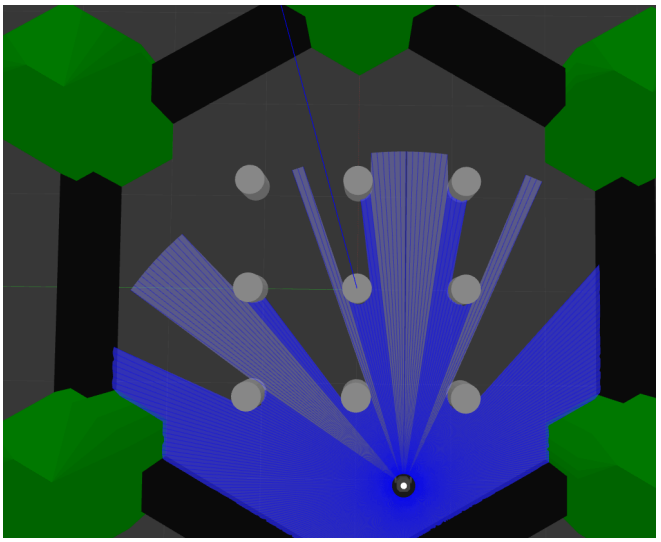
Code for the first method (splicing)

```
for i in self.path:
    if ((i[0] - self.y)**2 + (i[1] - self.x)**2)**0.5 < target_limit:
        if not is_wall_between(odata, (grid_y,grid_x),(round((i[0]-map_origin.y)/map_res),round((i[1]-map_origin.x)/map_res))):
            remove_index.append(i)
        elif odata[round((i[0]-map_origin.y)/map_res),round((i[1]-map_origin.x)/map_res)] == 3:
            remove_index.append(i)
    else:
        break
for i in remove_index:
    self.path.remove(i)
```

Code for the second method (clearing)

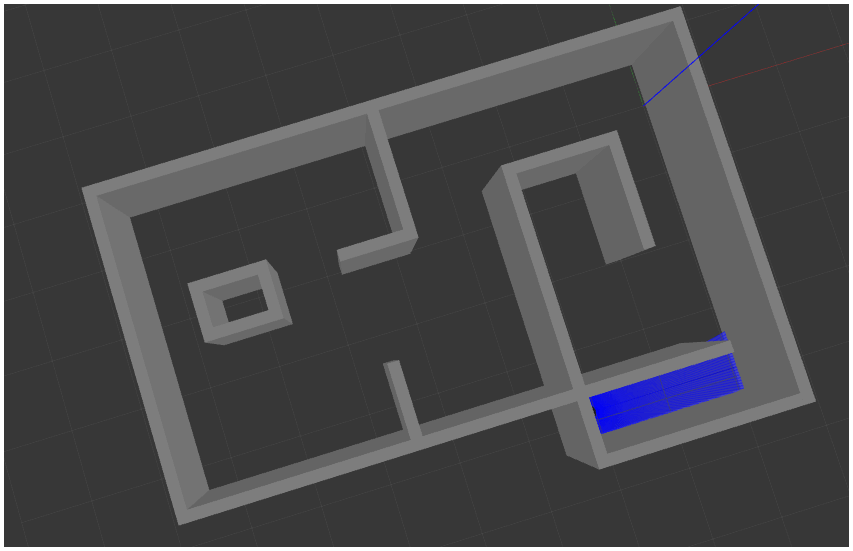
6.2 Simulations in Gazebo

In order to do testing on the turtlebot's navigation algorithm. Some gazebo environments were used to test the capabilities of the algorithm. The default *'test_world'* of gazebo was used to test the evasive maneuver abilities of early versions of the algorithm. This was possible as the columns in the test world were relatively small and could be avoided using the collision detection function. This world helped with testing the ranges and capabilities of crash avoidance and frontier detection.



Top Down View of the Original World

Once the algorithm was able to navigate through the original test world, another more challenging environment was created. This environment required the robot to do turns and double back through already-traveled regions to map the entire maze. This world helped with troubleshooting any problems with the A* algorithm whilst providing more calibration towards the collision detection of the algorithm.



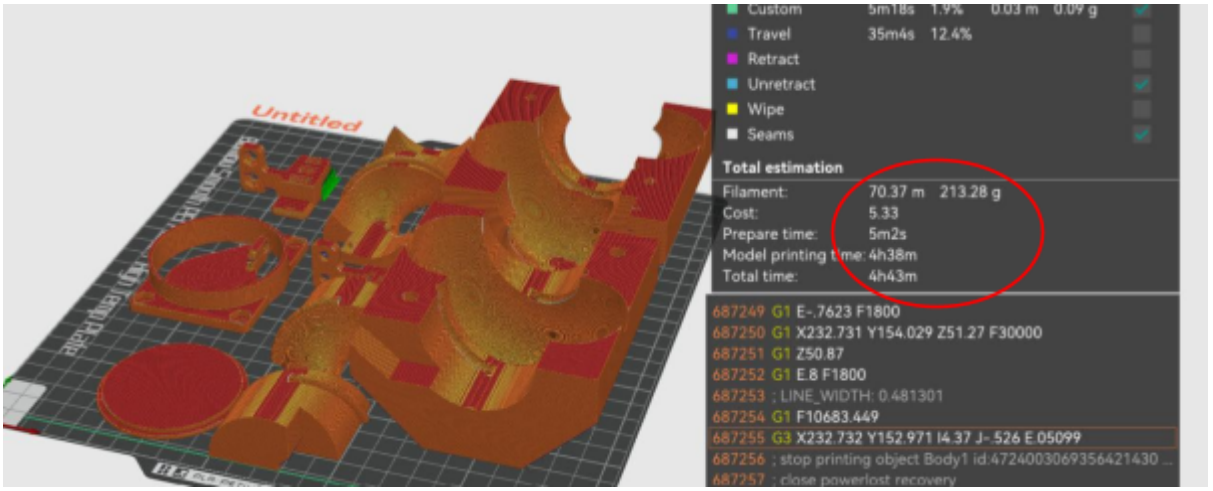
Top Down View of the Second Testing World

6.3 Payload

We started out prototyping while working with some imposed constraints:

1. The payload must block as little of the LiDAR's light path as possible
2. To use as little motors/ electronic components as possible to limit the complexity of the electronics.
3. As best as possible, to use off the shelf parts or any parts required should be easily 3D printable, without requiring much support
4. Minimize different screws and as much as possible, integrate captive nut designs to make assembly easier

In the end, all 3d printed parts could fit into a single print bed, with very minimal support (green parts in image) being used only on the IR sensor mounts. Total print time was around 5 hours with 0.24 layer height, 3 perimeters, 4 top and bottom layers, and 8% cubic subdivision infill.



6.3.1 Helical Ball Dropper

The ball dropping container was first prototyped with its height being the main constraint. A funnel was first tested but we were quickly met with the issue of balls being stuck at the top.

Instead of complicating the build to solve this problem, such as adding fins or ribs into the funnel to individually guide the balls, we decided to try an alternate method. Inspired by the slide in Changi Airport, we designed a ball slide, which was more compact in terms of height and lighter as it could have been printed with lesser infill, as compared to the funnel which was almost 100% plastic as it did not have much infill and comprised mainly of walls. However, the nature of the coil within the container made it difficult to 3D print, especially without support. Hence our mechanical engineer split the container into 4 pieces, using grooves to attach the inner parts of the "slide" with the outer container. This also served the purpose of "dummy-proofing" the assembly as each outer piece corresponded to the inner piece with the number of grooves it had. Furthermore, brass inserts were used to strengthen the joints between individual components as we wanted this ball dropper to be modular.

This serves as one of the unique features of our ball dropper – it can be used as a separate component to attach to other parts of a robot via the mounting holes prepared and would serve as a payload for future robots. Due to its light weight (130g) which was decided upon tweaking various Printer slicer settings in Orca Slicer, to find the balancing point of strength and weight, along with its high compactness, it can be used very conveniently in other ball dropping applications.

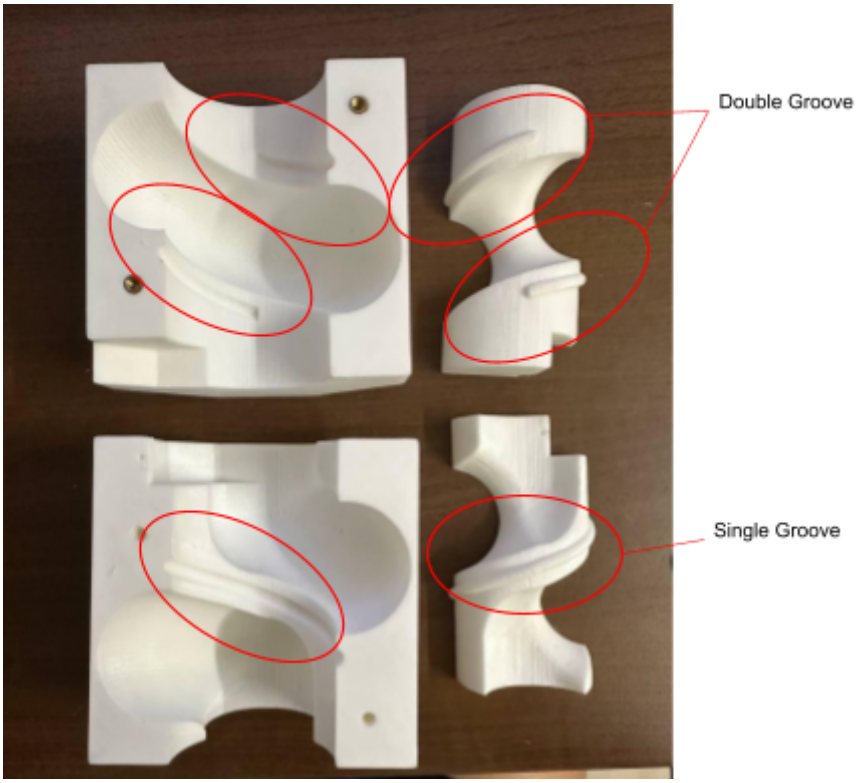


Figure showing how everything can be 3D printed without supports, along with the grooves

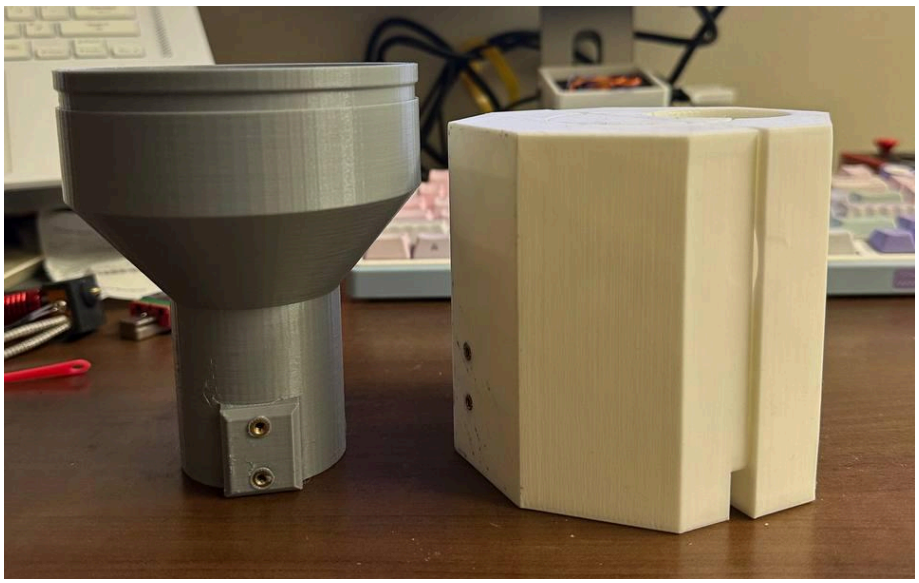
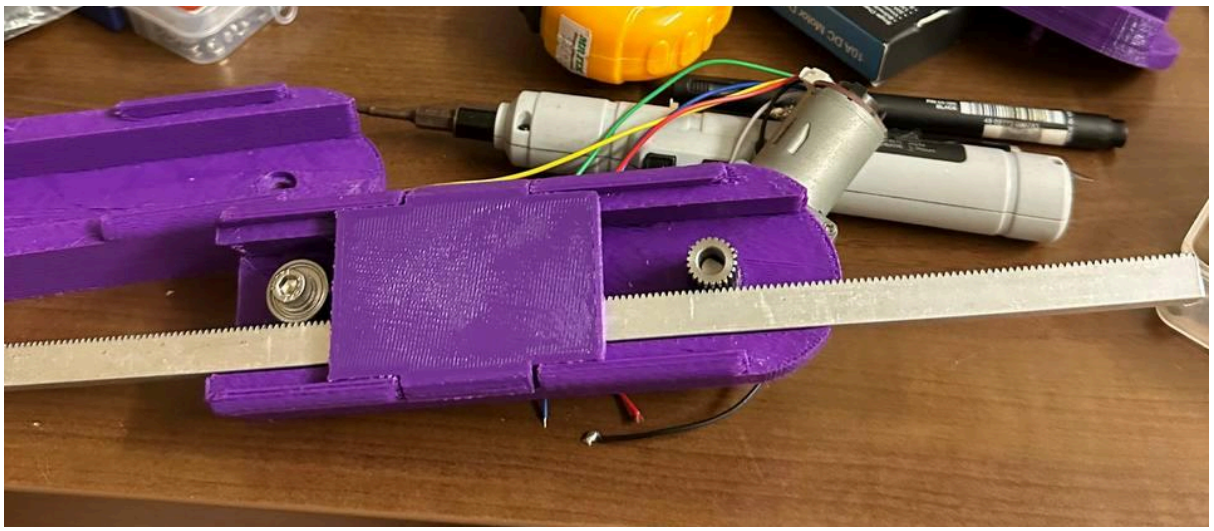


Figure showing height difference between old and new models

6.3.2 Rack and Pinion Arm Mechanism

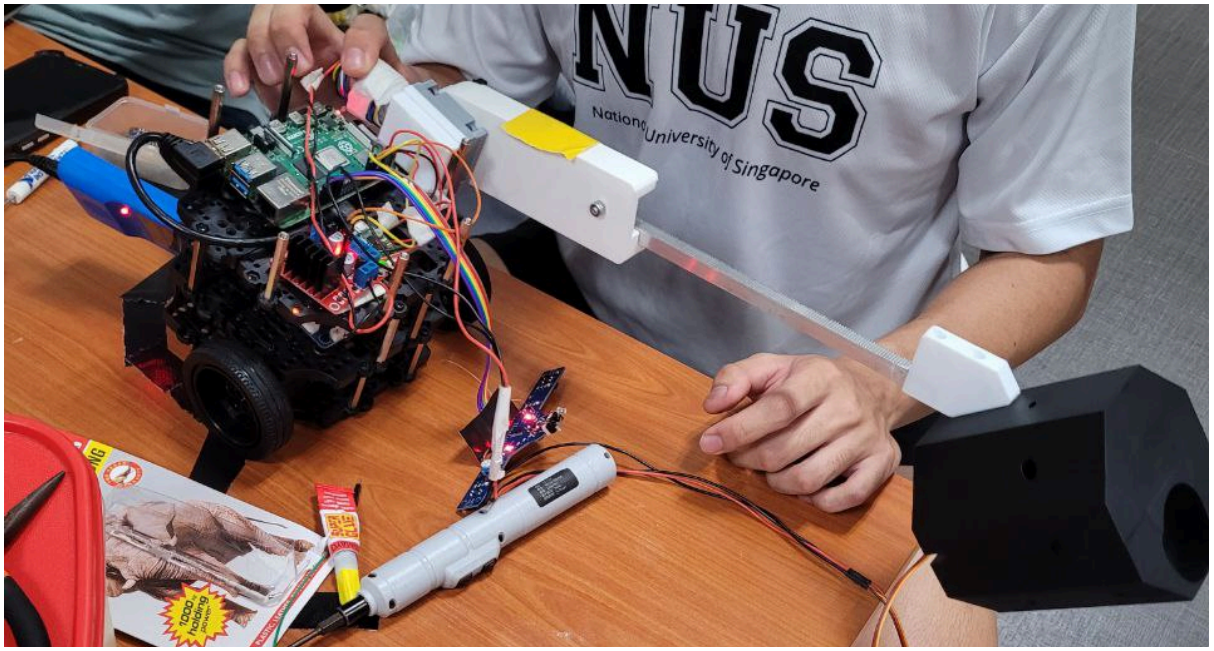
These constraints also led to our initial idea to allow the ball dropper to reach the tall bucket, which was to use a rack and pinion arm mechanism. It satisfies the second constraint by only using one DC motor and one servo motor. The DC motor, powered by the PWM from the H-bridge motor driver with voltage supplied from the RPi PWM pin, will slowly extend the rack arm. Once the end of the rack is caught with the pinion gear, it will cause rotational motion up to the desired height (height of the bucket).

Eventually, the ball dropper would be brought near to the bucket in a position high enough to drop the ping-pong balls by the actuation of a servo motor that holds the balls in the container until it is activated.



A prototype of our rack and pinion mechanism was created using a 3D printer. The rack is connected to the pinion which is connected to a JGY-370 geared motor. The motor will be powered through an H-bridge motor driver and the voltage is supplied through the PWM pins of the GPIO to allow control of the rotation of the motor and consequently, the pinion. This is due to the fact that the motor is most suited to be supplied using a 12 volts input. After testing a 3D printed rack, an aluminum rack was chosen in the end due to its relatively low density, yet high Young's Modulus, which was critical in ensuring it held the ball dropper up rigidly when fully extended and over the bucket.

We chose the L298N motor driver over the conventional L293D as we needed a driver capable of 12V outputs, and also being able to allow sufficient current for the motor, despite the lack of efficiency of the L298N. However, balancing the cost of alternate drivers, we decided the highest value option remained to be the L298N.



When testing, we realized that when the motor does not receive the full 12V from the L298N driver, it would stall and overheat, which also causes the L298N driver to heat up, further decreasing its efficiency. When supplying the full 12V to the motor, the rack would extend quickly and catch onto the end of the rack which is meant to give it the clockwise rotation we desired. However, due to how quickly the end of the rack caught onto the pinion gear, it caused the entire payload to jerk up in a clockwise direction very quickly. This induces a reaction anticlockwise moment about the pinion, which is connected to the robot. The high speed, along with the torque of the motor, causes a high impulse, which causes a high anticlockwise moment to act on the entire robot, causing it to flip itself forward.

We could attempt to do fine motor control, so as to smooth out the transition between the rack extension stage and spin stage. However, with the L298 driver and motor heating up, the PWM signals we sent started to go inconsistent. Furthermore, we had a large lack of time to prototype and test the payload, which was caused by our prototyping being delayed due to the JGY370 motor arriving later than expected. All these factors lead to a painful decision of scrapping this rather unique ball dropper design, and to the development of a new payload that reused the ball dropper component of our older design. Thankfully, the ball dropper was designed with modularity in mind.

Future development for rack and pinion mechanism:
This mechanism is very promising as it uses only 1 motor for 2 axes of motion. In order to combat Newton's third law of Motion, an endstop switch could be placed on the end of the rack. Once the pinion catches with that endstop, it would cause the motor to stall with the rack fully extended, and enter phase 2 of the payload, which is the rotation stage. It would be activated at a lower PWM so that the rotation speed would be slower and more controlled. This would decrease the impulse on the pinion gear, as the moment is over a longer period of time.

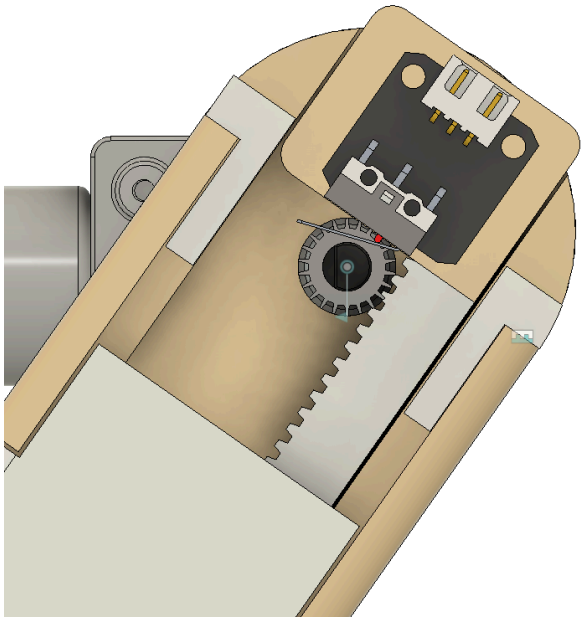


Figure demonstrating endstop at end of rack for fixing the issues above

We decided to stick to KISS principles (keep it simple,stupid), and raised the turtlebot to an appropriate height using waffle plates and standoffs. This elevated the ball dropper above the bucket, yet not bumping into the door. This was possible because the ball dropper was compact.

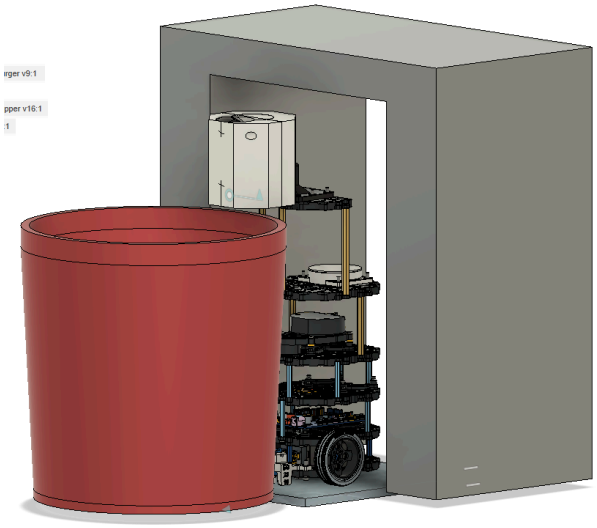


Figure of robot reaching bucket height yet not bumping into door

The high Centre of Gravity made the robot theoretically unstable. As a failsafe and insurance, we added a 200G counterweight on the robot to increase its stability.

In the end, all printed parts were lightweight and housed captive nut designs so that the user could assemble the robot without using a plier to hold the nut down. Furthermore, screws were generally standardized, such as only one type of M4 screw for assembly being used for the mounts, which are M4x12mm Flat head screws. M3 screws are used for mounting the ball dropper container to the ball dropper mount as the ball dropper container was modeled with modularity in mind, and M3 screws are used more widely in other payload mounting systems.

6.4 Line Follower, HTTP Calling and Servo integration

6.4.1 Line Follower to Lift Door Stage

Software Logic:
The line follower system has two infrared (IR) sensors and are connected to pins which are GPIO13 and GPIO17. If a pin detects black, it will be set to ‘1’ or HIGH, otherwise, it will be set to ‘0’ or LOW. Throughout the course of the run, the line follower will be on at all times. This is to ensure that it continuously checks if there are any lines to be detected. At the end of each stage, a horizontal “T shaped” black line will be placed on the floor, allowing the line follower to sense two black lines, thereby allowing the Rpi to run the scripts for the next stage.

From the starting point, a long black line guiding the bot to the lobby will be placed on the floor. The turtlebot will follow this path through the maze while simultaneously mapping its surroundings. At the lobby, the bot will encounter the horizontal “T shaped” black line, transitioning to the lift door stage.

Mechanical:

To decide on the hardware used for the line follower, we initially wanted to get off the shelf parts, especially parts specifically made for line following. This would give us the assurance that the performance will be reliable and good. We bought 3 different types of line follower PCBs that had varying numbers of IR sensors. We initially settled on a 6 sensor unit that also had IR sensors at the front as a form of bump detection.

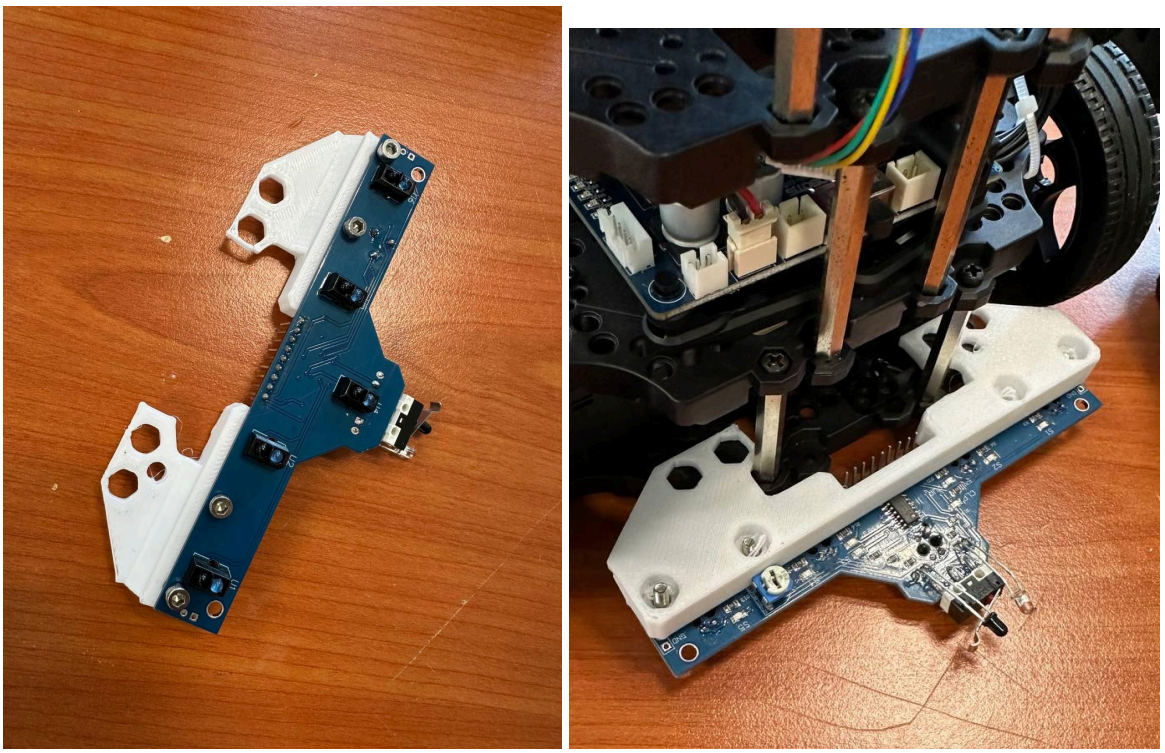


Figure showing original IR sensor being used along with its mount

However, just 3 hours before our trial run, we accidentally plugged the wires in wrongfully, causing the PCB to fry leading to only 3 out of 5 IR sensors working.

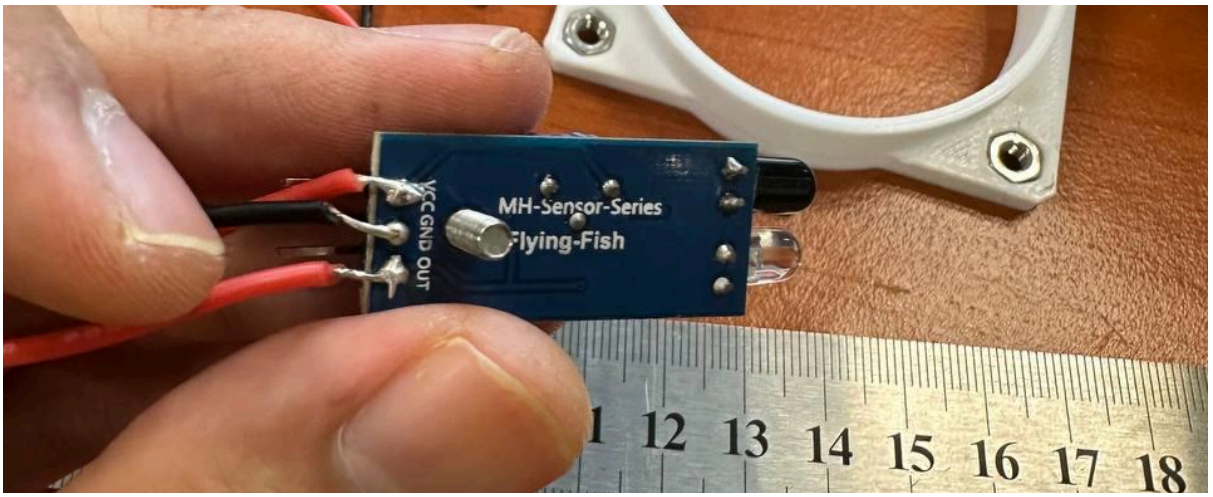
We immediately loaned 2 IR sensors from Ms Annie's lab and designed new mounts for them. for the spacing between the IR sensors, spacers were prepared. In the end, we used a m4 nut as a spacer as it was the perfect distance

New IR sensor mounts, with group numbers embossed on it

Electronics

Many times, we realised connectivity of the flimsy Dupont connectors was a big breaking point of our robot. Given the amount of movement and vibration, the connectors tended to come loose easily, Hence, we decided to solder down every single joint, except those going

into the Raspberry Pi 4 GPIO.



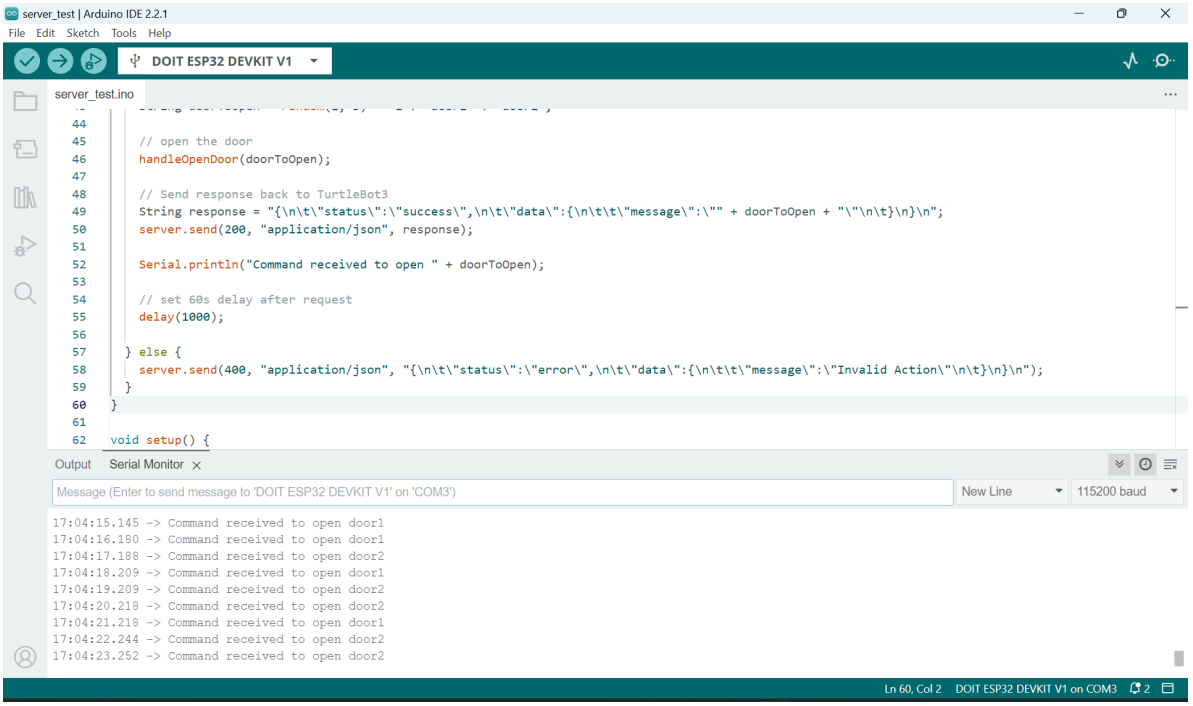
6.4.2 HTTP Calling and Line Follower integration

Once in the lift door stage, the turtlebot is coded such that it will make an HTTP POST request when it stops at the lobby, between the two locked doors when it has detected the black line. To check if the code works, an ESP32 relay and CH340 are used to create a ‘server’ which randomly unlocks either door 1 or door 2. The RPi will send a POST request to the server while sending the ‘esp_id’ and turtlebot id. Afterwards, upon a successful request, the RPi should receive the door number which is unlocked.

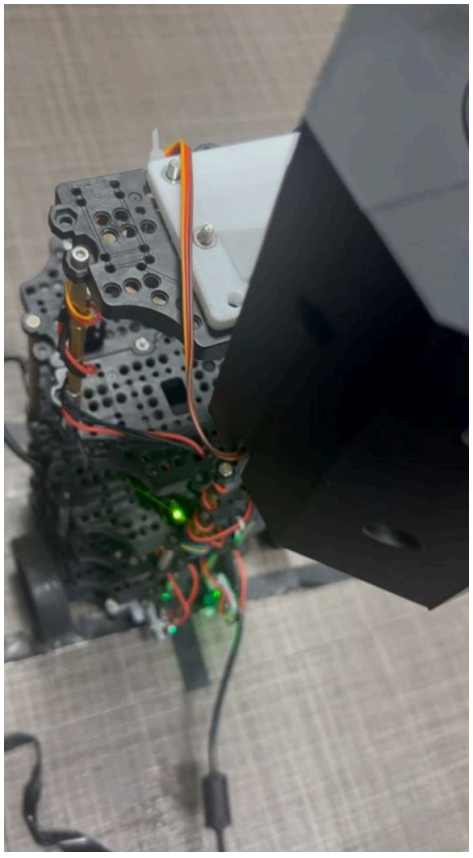
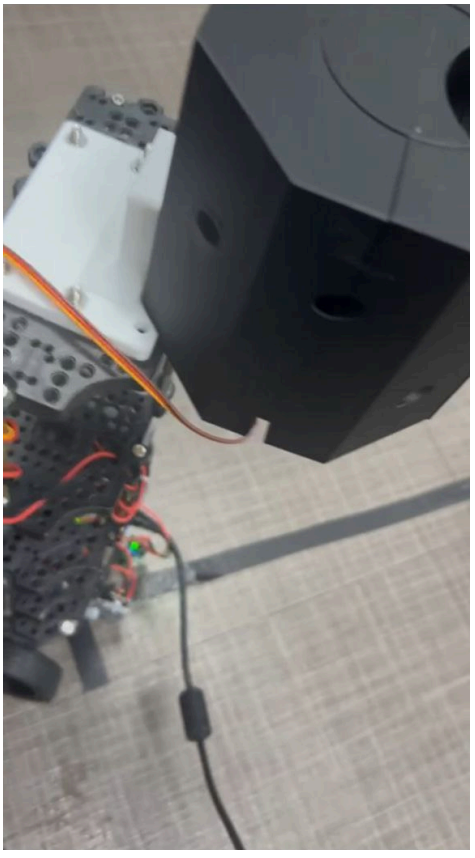
Line follower has been tested a lot of times and was proven to be successful as the turtlebot was able to follow the black line smoothly. The RPi was able to receive the correct door number which was unlocked and knew which direction to go. This proves the correct integration between the HTTP Calling and the line follower code. When ‘door1’ is received, it would be directed to move forward to the door and eventually, reaching the bucket. A similar event will happen when ‘door2’, but instead, it will go to the right. Two black lines will be placed from the horizontal line, leading to each bucket. Once the Turtlebot has turned in the appropriate direction, it continues following the line until it reaches another horizontal line right in front of the bucket. At this point, the Turtlebot transitions to the bucket stage.

We faced some issues with this integration. For instance, an issue was faced after HTTP calling. Despite the successful HTTP call and receiving the unlocked door number, the bot could continue to keep moving forward and not stop. Upon some debugging, it was found out that it was due to a callback issue which was not done fast enough for the turtlebot to recognize that it should have moved to the next stage. Hence, the delay was increased to ensure it has changed to the correct stage to prevent a misexecution of a code, especially the line follower code.

Another issue related to this was that the turtlebot moved too fast in some instances such that one of the IR sensors missed the black line. Hence, instead of performing a HTTP call, it turned left or right instead. To mitigate this issue, another horizontal black line was added to increase its width to reduce the chance of the IR sensors not detecting the black line.

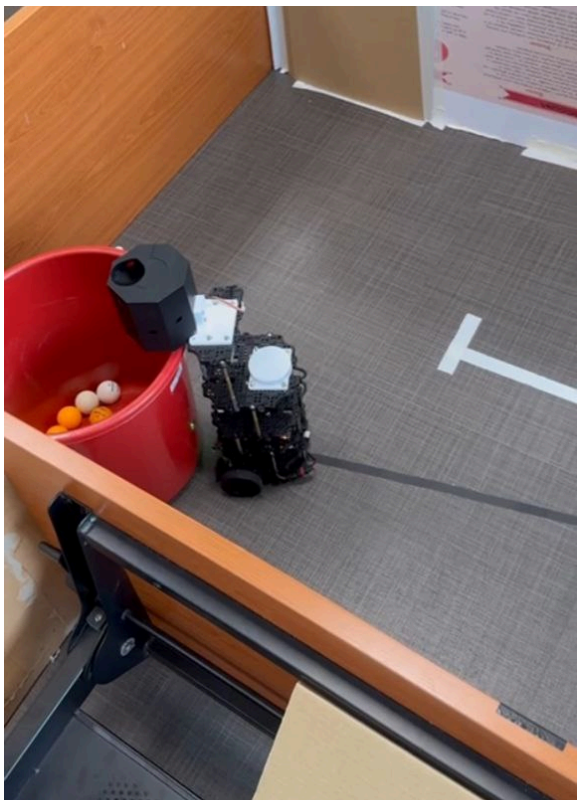


Sample of door number received from ESP server in Arduino IDE

Doing HTTP Calling	Going to the unlocked door
	
The bot stops as it performs HTTP calling	Since door 1 is unlocked, it is directed to turn left

6.4.3 Line Follower and Servo integration

In the Bucket Stage, a signal will be sent from the RPi to the servo, causing the servo to flick open and deposit the balls into the bucket. After dropping all the ping-pong balls to the bucket, the turtlebot needs to go back to the lobby to complete the mapping of the maze. To achieve this, the turtlebot will be coded to go backwards for a few seconds. The Turtlebot then makes a 180° turn in such a way that the line sensor is still perfectly aligned with the black line, but the robot is facing the opposite direction. It will then perform the line following code to reach the T-junction at the lobby. Once the Turtlebot reaches this junction, it transitions to the autonomous navigation stage, where neither the servo or Line Follower is used anymore.

Turtlebot going backwards	Turtlebot turning 180º
	

7. Critical Design

7.1 Key Specifications

List	Specifications	Remarks
Dimension	Turtlebot: 13.8 cm x 17.8cm x 19.2 cm With ball dropper: 14.6 cm x 17.8 cm x 43.7 cm	Length x Width x Height
Weight	1 kg	Only turtlebot (from datasheet)
	1.735 kg	With additional attachments and balls
Centre of Gravity	X: 6.16 cm (width) Y: 0.04 cm (length) Z: 6.76 cm (height)	Only turtlebot
	Shown in the 7.3 Turtlebot Subsystem	With additional attachments
Battery capacity	11.1 V, 1800 mAh, 19.98 Wh	
Robot Controller	Open CR 1.0	
Single board computer	Raspberry Pi Model 4B	
Wheel Actuators	2x Dynamixel XL430 - W250	
Servo Motor	1x SG90	
Communication Interface	GPIO pins	

7.2 System Finances

For this project, a complete set of turtlebot to create a fully functional system. With an additional budget of $100, there were parts bought and 3D-printed to equip the turtlebot to complete the tasks for this project. The list of items bought for prototyping and testing is shown in the table below.

No	Item	Unit Cost ($)	Quantity	Total Cost ($)	Remarks
1	IR Sensors	0	2	0	From iDP lab
2	3D-printing	5	5 (hours)	25	Purchased from Taobao
3	JGY370 right angle geared motor 12V	3.85	2	7.70	

4	Line Follower integrated module Variant A	0.92	2	1.84	
5	Line Follower integrated module variant B	1.16	1	1.16	
6	Line Follower integrated module variant C	0.91	1	0.91	
7	L298N Motor Driver	0.85	3	2.55	
8	1m x 0.5 Mod 8mm x 10mm aluminium rack	11.18	1	11.18	
9	0.5 Mod 22 teeth 6mm bore pinion	1.8	2	3.6	
10	625-2z bearing	0.56	5	2.8	
11	MR106 bearing	0.75	7	3.75	
12	Shipping	8	1	8	
13	Waffle-Plate		8		Came from the standard turtlebot set
14	OpenCR 1.0		1		
15	Raspberry Pi		1		
16	Wheel		2		
17	Tire		2		
18	360 Laser Distance Sensor LDS-02		1		
19	Li-Po Battery		1		
20	DYNAMIXEL, RL430		2		
21	USB Cable		2		
22	DYNAMIXEL to OpenCR Cable		2		
23	Raspberry Pi Power Cable		1		
24	Li-Po Battery		1		

	Extension Cable				
25	LDS-02 Cable		2		
26	Ball Caster		1		
27	USB2LDS		1		
29	Container & Lid		1		
31	SG-90 Servo		1		
32	Continuous Motor Servo		1		
			Total Cost	68.49	

7.3 Bill of Materials

This section provides the Bill of Materials for the components used in the final turtlebot.

No	Item	Unit Cost ($)	Quantity	Total Cost ($)	Remarks
1	IR Sensors		2		From iDP lab
2	3D-printing	5	5 (hours)	25	From Justin's room
3	M3x8 SH screws		11		From iDP lab
4	M3x16 SH screws		2		From iDP lab
5	M4x12 FH screws		12		From iDP lab
6	M3 nuts		5		From iDP lab
7	M4 nuts		12		From iDP lab
8	F-F Brass standoffs				From iDP lab, of varying lengths, to hit 104mm
9	M-F Brass standoffs				From iDP lab, of varying lengths, to hit 75mm

10	M2 self tapping screw		2		From iDP lab
11	M3 x 4.2 x 5 brass inserts		2		From iDP lab
12	M4 x 5.5 x 6 brass inserts		2		From iDP lab
13	SG90 servo motor		1		Standard EG2310 set
14	Waffle-Plate		8 + 3 (for additional layers)		Came from the standard turtlebot set
15	OpenCR 1.0		1		
16	Raspberry Pi		1		
17	Wheel		2		
18	Tire		2		
19	360 Laser Distance Sensor LDS-02		1		
20	Li-Po Battery		1		
21	DYNAMIXEL , RL430		2		
22	USB Cable		2		
23	DYNAMIXEL to OpenCR Cable		2		
24	Raspberry Pi Power Cable		1		
25	Li-Po Battery Extension Cable		1		
26	LDS-02 Cable		2		
27	Ball Caster		1		
28	USB2LDS		1		
			Total Cost	25	

7.4 Power Management

7.4.1 Power Consumption of turtlebot

We assume the voltage throughout the circuit is 11.1V
(i) Initial bootup = 0.85 A -> power = 9.435 W
(ii) Idle = 0.3 A -> power = 3.33 W
(iii) Operation = 0.8 A -> power = 8.88 W

7.4.2 Power Budgeting

Component	Voltage (V)	Current (A)	Qty	Power (W)	Remark
Servo motor	4.88	0.2	1	0.92	Operating current
LIDAR (LDS-02)	5	0.24	1	1.2	Idle state
OpenCR1.0	3.6	0.32	1	1.152	Assuming maximum standard operating voltage, and max current draw
Dynamixel(XL430) - Motor	11.1	0.3	2	26.64	For each motor On standby: 0.052 A Operating current: 1.2 A
Raspberry Pi 4 Model B	5	0.8	1	4.0	
Total Power Consumed				33.912	

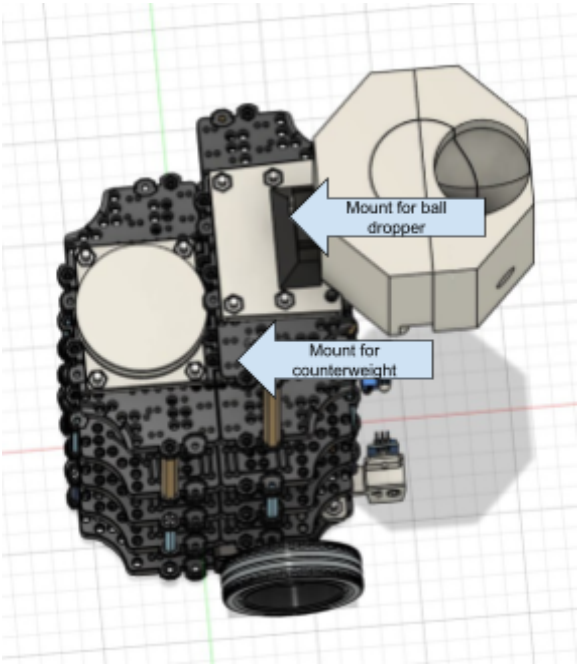
7.4.3 Estimation of System Operating Duration

Assuming only turtlebot is running:
Standard Li-Po battery provided in turtlebot = 11.1 V, 1800 mAh, 19.98 Wh
System operation duration = 11.1V x (1800 mAh / 1000) / 8.88 W = 2.25 h

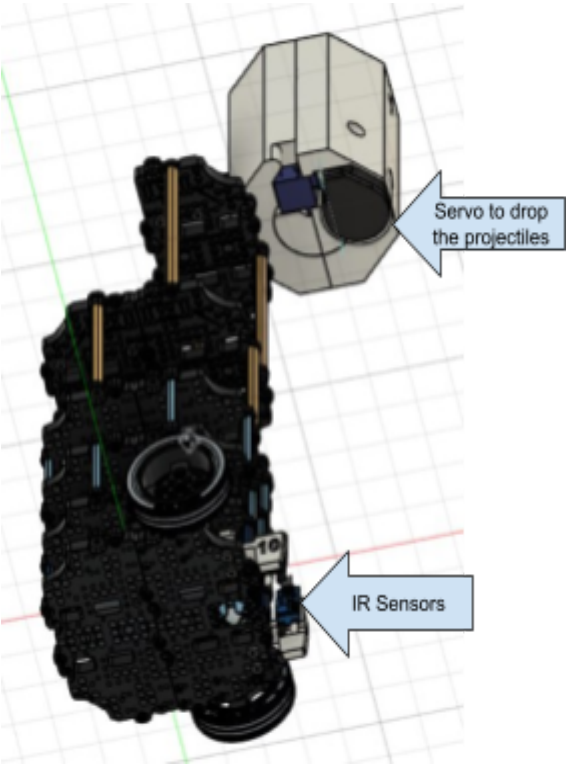
If all electronic parts are on at the same time:
System operation duration = 19.98 Wh / 33.912 W = 0.589 h = 35.35 min

7.5 Subsystem of Turtlebot

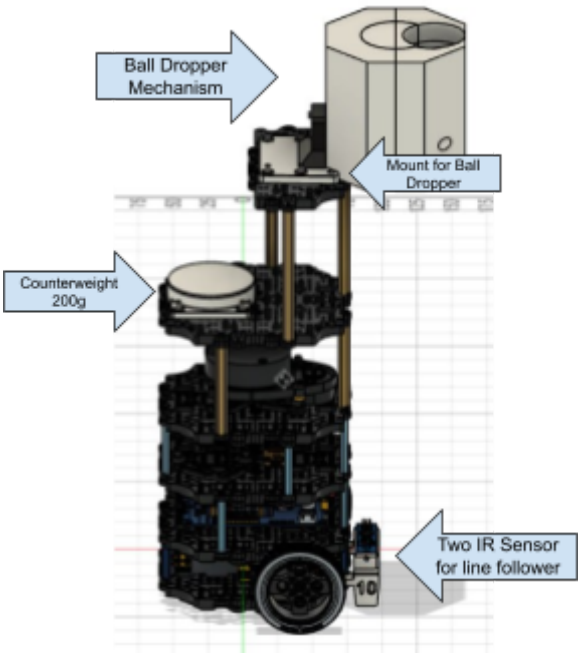
Top View



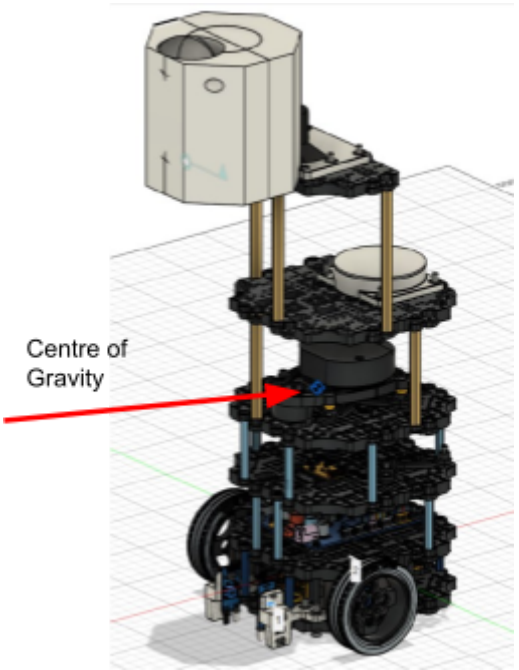
Bottom View



Side View



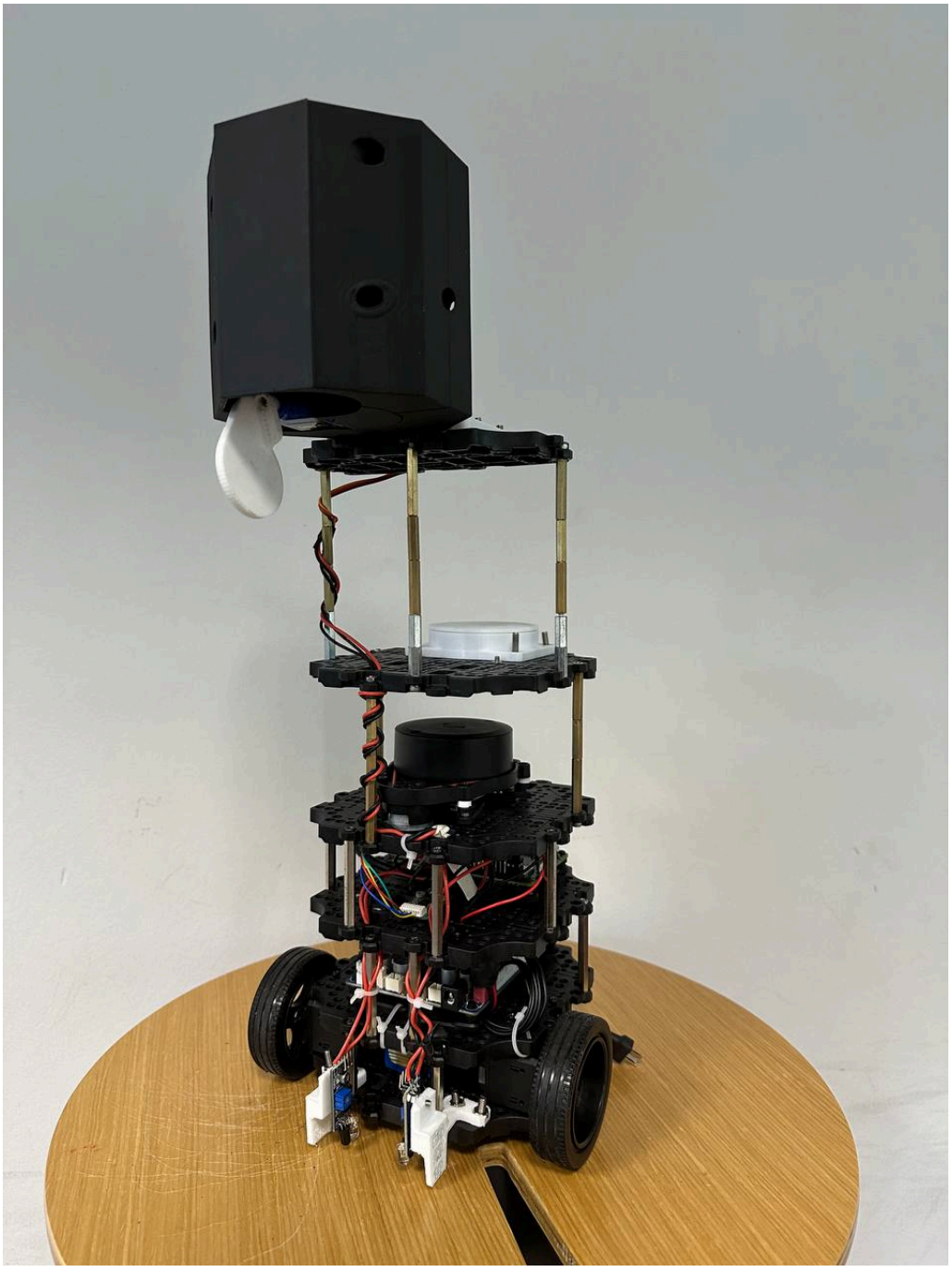
Centre of Gravity



8. Assembly Instruction

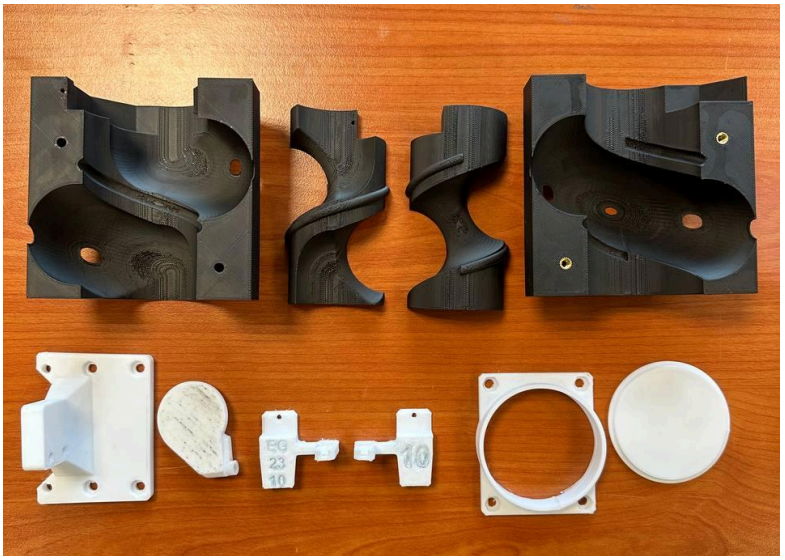
8.1 Mechanical Assembly

The turtlebot was assembled by following the manual guide given in the standard package. However, other hardwares have been assembled and attached to the turtlebot to complete the challenges of the mission which include the helical ball dropper, counterweight and two IR sensors. The final turtlebot product is as shown in the picture below.



Final Turtlebot product

8.1.1 Real-life Assembly

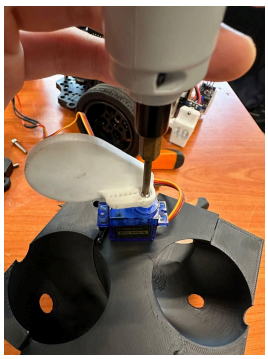


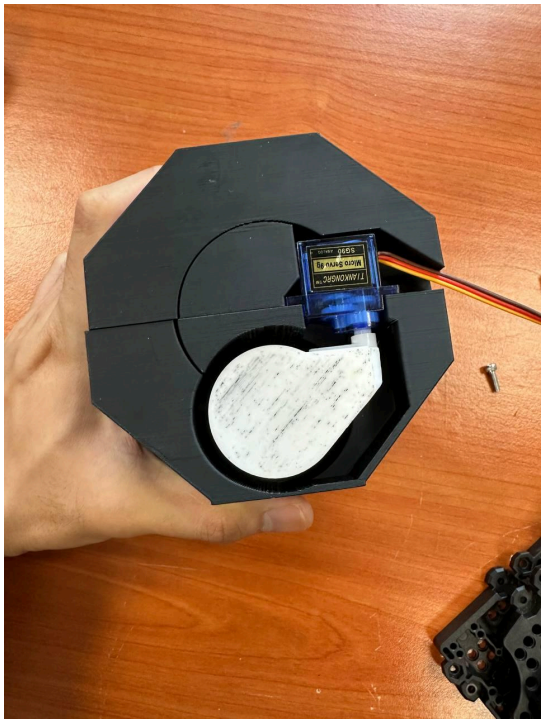
All 3D printed parts

(a) Ball Dropper Assembly

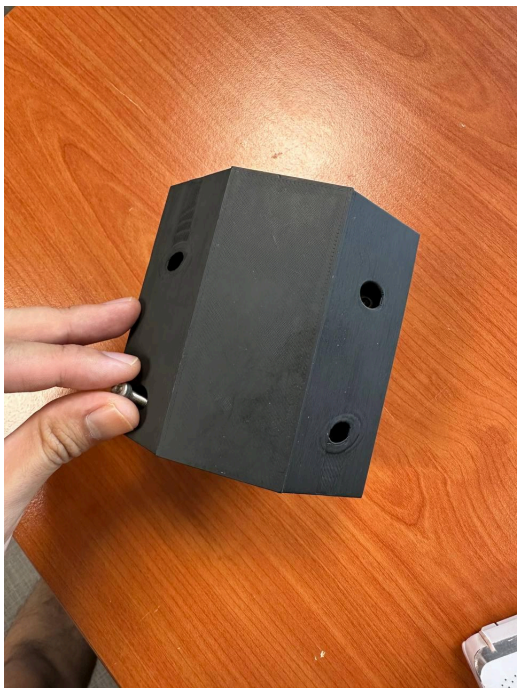
	1. Prepare 3D printed parts 2. Glue flap to servo horn using superglue by aligning the holes on the flap

 	Prepare and install heat set brass inserts (M3 inserts for mount (left picture) and M4 inserts for ball dropper (right picture)
   	Fit the inner loop onto the shells of the ball dropping container. Single ridge inner holders will only mate with single ridge outer holders, and the same for the double ridge holders.

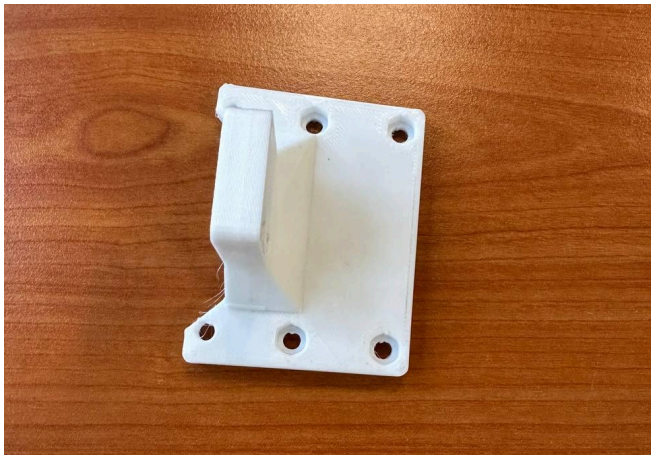
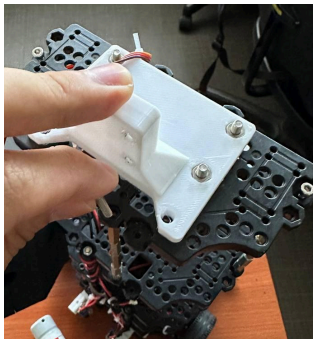
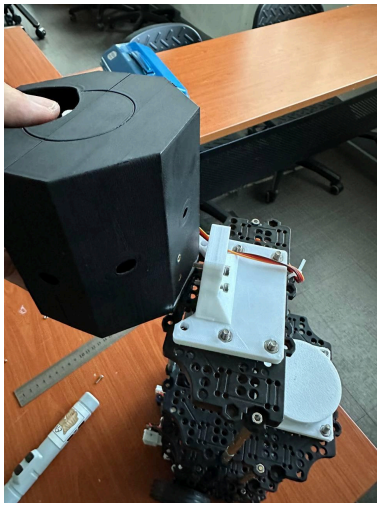




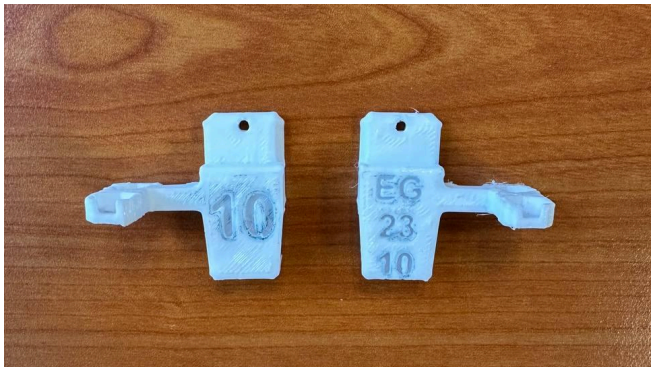
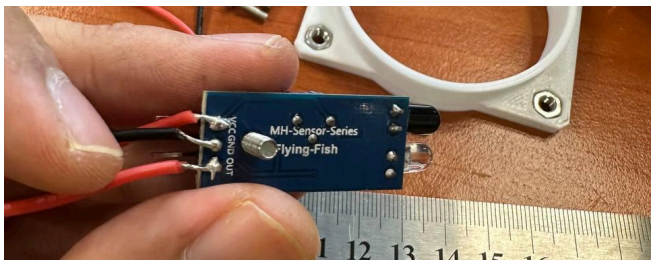
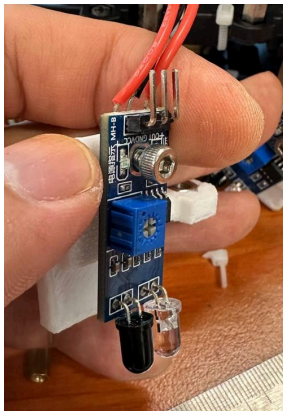
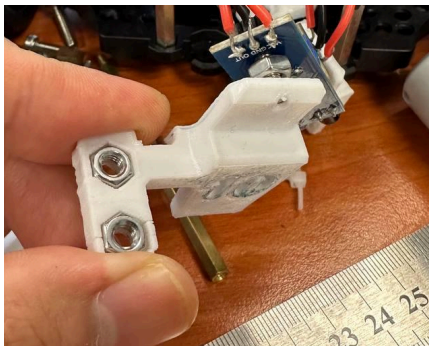
1. Screw the flap into the servo motor, screw the servo motor into the ball dropper
2. Mesh both sides of the ball dropper and check for tolerance issues, and that the bottom profile looks like the picture.

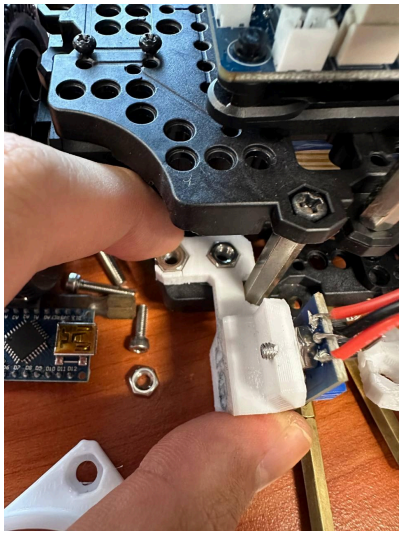
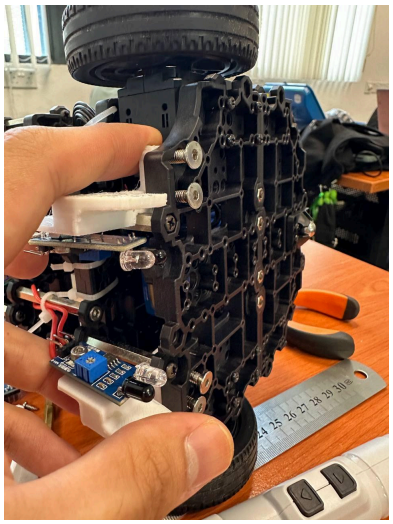


Using M4x12 FH screws, bolt both sides of the ball dropper together.

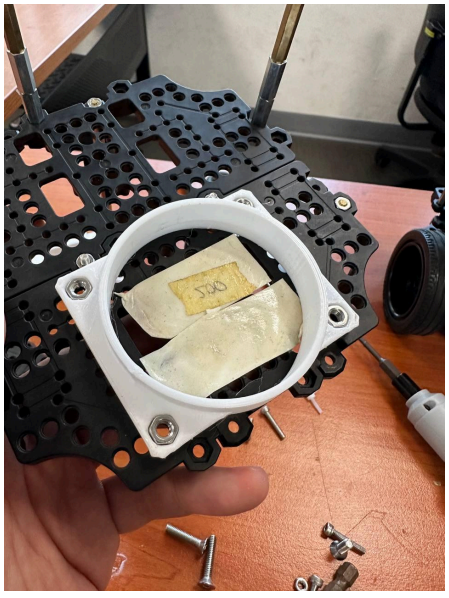
	Prepare ball dropper mounting plate, ensure through holes are clean, use a 4mm drill bit to drill through if holes are not clean
  	1. Place M4 nuts into the nut slot, and align holes with TurtleBot waffle plate holes as shown. 2. Attach mounting plate onto waffle plate using M4x12 FH screws
	Attach mounting plate to ball dropper container using M3x16 SH screws.

(b) IR Sensors

 	1. Prepare 3D printed parts 2. (optional) Solder wires to IR sensor for more stable joints
 	Using an M4 nut as a spacer between the part and the IR sensor, joint them using a M3x16mm SH screw
	Insert nuts into the nut holders

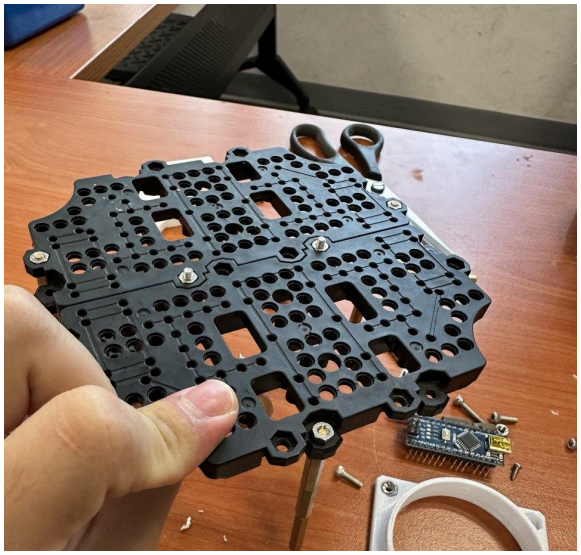
 	Align holes with Turtlebot waffle and attach using M4x12 FH screws.

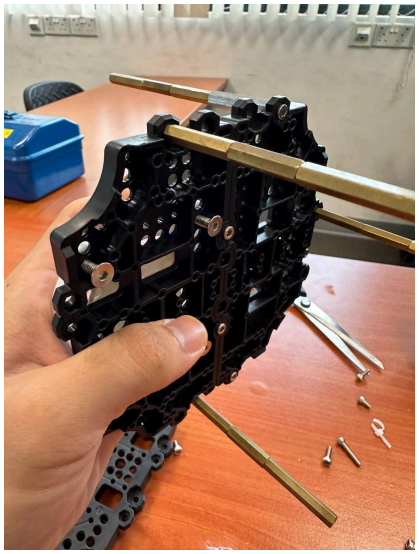
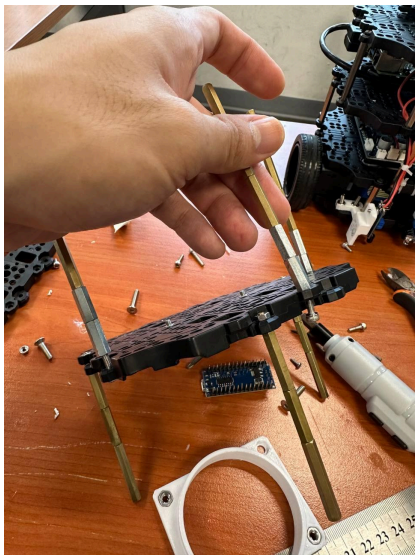
(c) Counter-weight

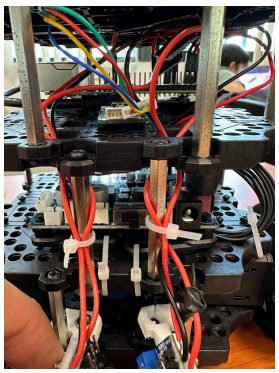
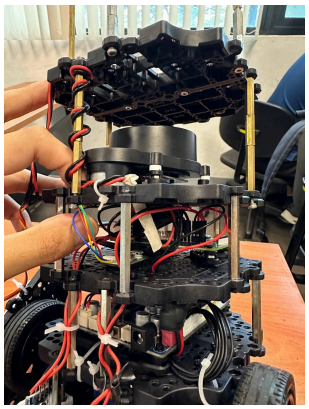
	Place m4 Nuts into nut holder, aligning holes with Turtlebot Waffle as shown

	Add weight into hold and snap on with the lid.

(d) Waffle plates assembly and cable management

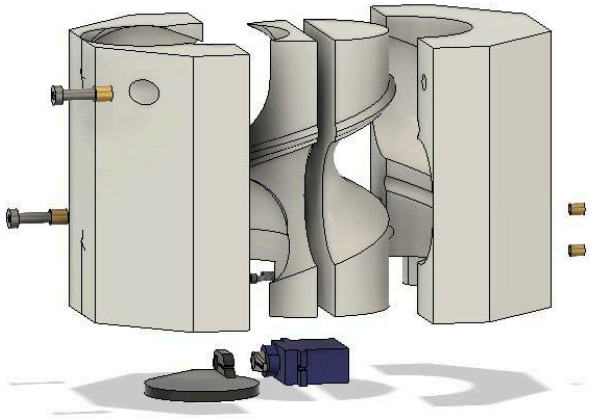
	Assemble waffle plates using M3x8 SH screws and nuts
	Prepare 3 x 104mm Female-Female standoffs and 3 x 75mm Male-Female Standoffs

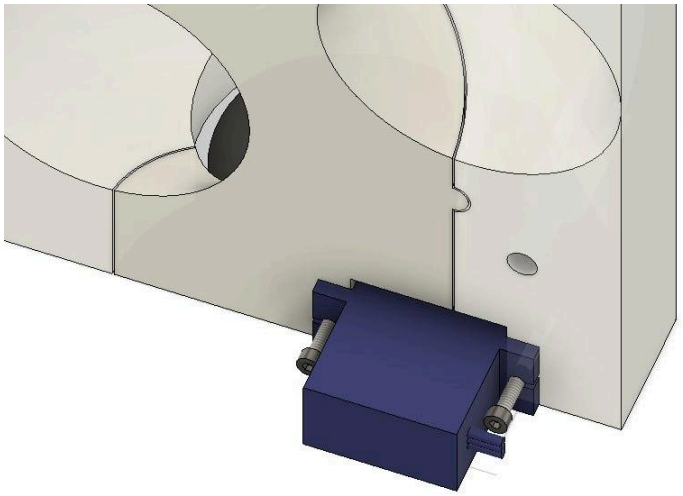
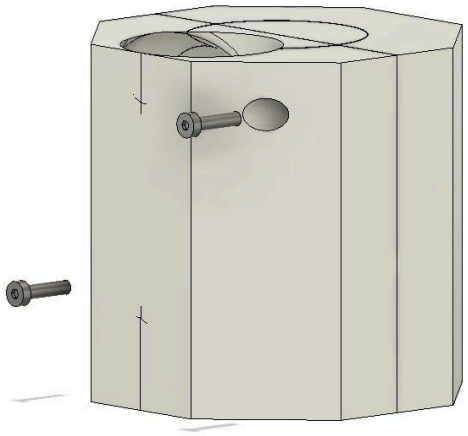
	
 	1. Attach 75mm standoffs to waffleplate using m3 nuts 2. Attach 104mm standoffs to waffleplate using M3x8 SH screws. 3. Attach top waffle plate to standoffs using M3x8 SH screws

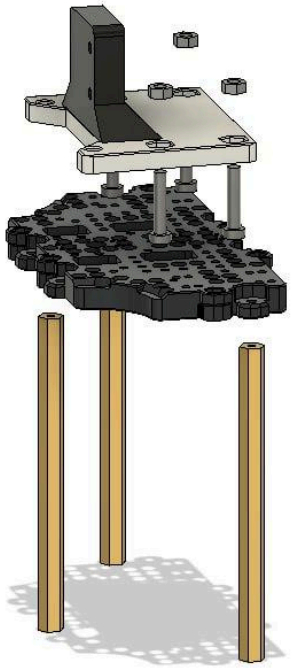
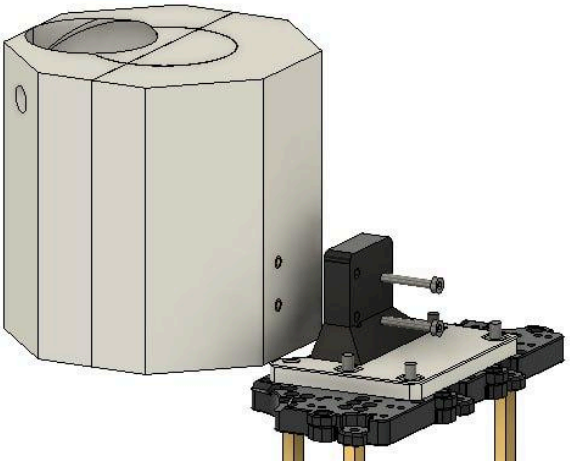
 	Cable is managed as shown

8.1.2 CAD Assembly

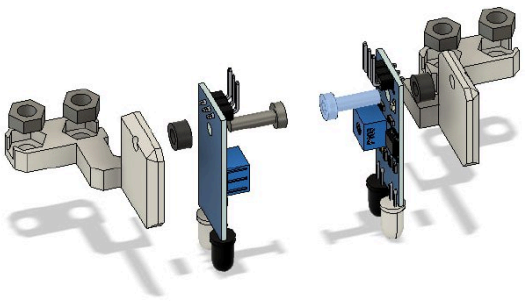
(a) Ball Dropper

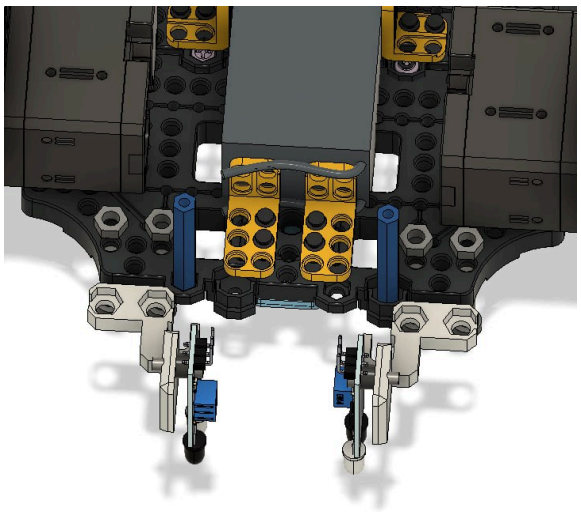
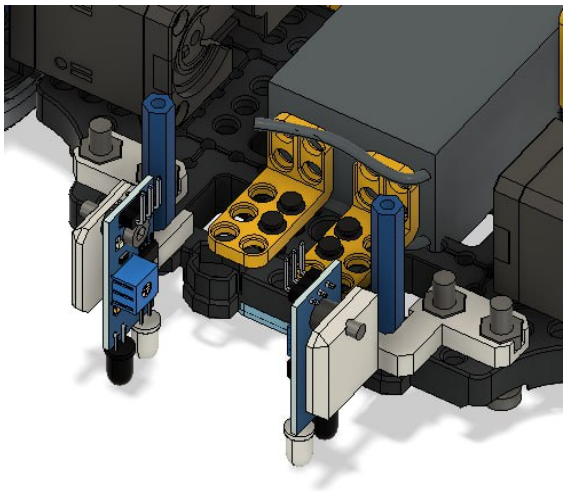
	The exploded view of the ball dropper container

	Attaching servo to container
	Combining the container into one

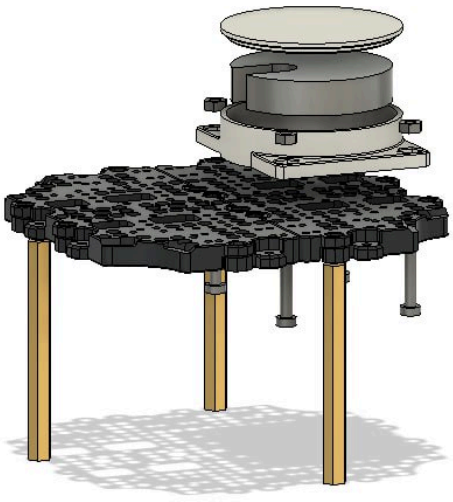
	Attaching the ball dropper mount to the waffle plate
	Mounting the ball dropper to the top layer

(b) IR Sensors

	Exploded view of IR sensors

	Attaching the IR sensors mount onto the waffle plate
	Screwing M4 into the nuts

(c) Counter-weight

	Attach the mount for the counter-weight onto the waffle plate

8.2 Software Assembly

8.2.1 Device Setup

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

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

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

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

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

6. Add the following lines to *.bahsrc* of the laptop.

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

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

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

8.2.2 Installing Software on Laptop

1. Create a ROS package on the laptop

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

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

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

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

```
git clone https://github.com/JohnE1129839/r2auto_nav.git .
```

4. Move the file __init__.py back.

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

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

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

8.2.3 Installing Software on RPi

1. Create a ROS 2 package on the RPi.

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

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

```
rm __init__.py
```

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

```
git clone https://github.com/JohnE1129839/r2auto_nav.git .
```

8.2.4 Calibration of Parameters

The beginning of autonav.py has some parameters that can be tuned to suit the desires of the mission.

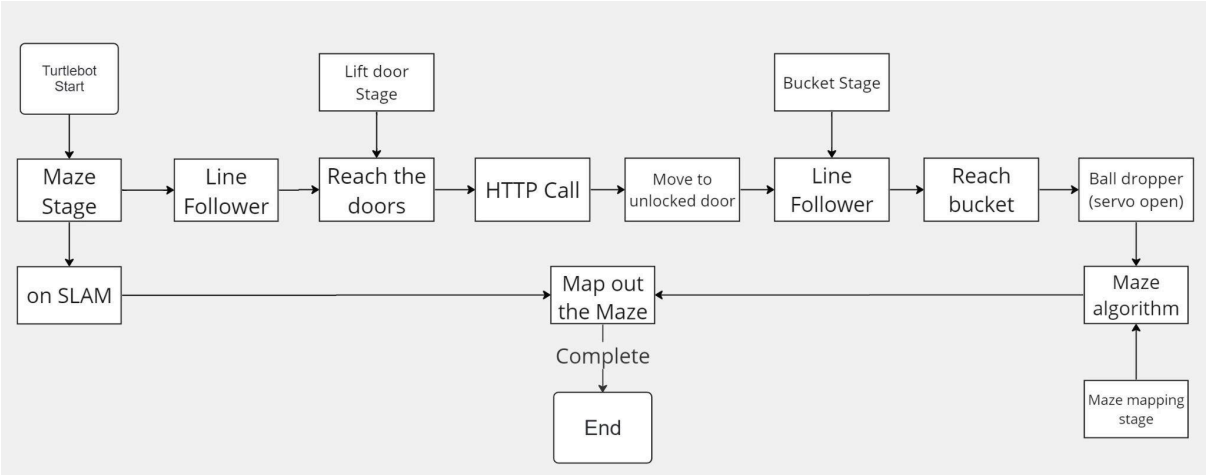
```
# constants
occ_bins = [-1, 0, 55, 100] #For bin separation
threshold = 0.3 #For frontier screening
proximity_limit = 0.25 #For detecting if target has been reached
target_limit = 0.7 #For target clearing
rotatechange = 0.25 #For rotating the robot
stop_distance = 0.2 #Maximal distance for crash avoidance
front_angle = 35 #Angles to scan for crash avoidance
testing = True #For troubleshooting
wall_size = 0.06 #For wall expansion
```

The breakdown of each parameter is shown in the table below

In the autonav.py script	
Parameter	**Function**
occ_bins	Histogram bins to group the occupancy values between open space and occupied area. Adjust the 3rd value allows for tuning

	the sensitivity of the occupancy map.
threshold	Minimum length for a frontier group to be considered a viable travellable frontier. (Given in meters)
proximity_limit	Minimum distance from the robot to a target to be considered to have reached said target. (Given in meters)
target_limit	Maximum distance of points cleared once a target has been reached (Given in meters)
rotatechange	Rotational speed of the robot
stop_distance	Maximum distance from the robot to an obstacle ahead to trigger obstacle avoidance
front_angle	Front angles the robot scans through to detect obstacles
testing	Set to *True* to view the path generated by the A*
wall_size	Parameter of which the walls are expanded (Given in meters)

8.3 Software Algorithm and Code Design



Flowchart of overall system

The turtlebot will map out the maze thoroughly using LiDAR while running the algorithm to traverse the maze and not hit any walls. A line follower will be used to check any lines which signify the end of a stage and the start of another stage. The functional description are stated below:

(a) Maze Navigation - Turtlebot is able to navigate and map a randomly generated environment
(b) HTTP calling - The Turtlebot is capable of initiating a HTTP call and receiving information from an ESP32 server
(c) Line Following - The Turtlebot can detect a line and follow its direction
(d) Load Deposit - System is able to deposit its load into a specified container

A github repository for the codes explained in the sections below is posted through the link shown in the appendix.

8.3.1 Line Following

(a) Detect black or white

Two IR sensors are used to guide the turtlebot to traverse through the maze. The sensors will try to detect the presence of a black line underneath it. The pins will recieve an input to the RPi GPIO pins 13 and 17 of '1' when a black line is detected and '0' otherwise. Afterwards, we set two new inputs s1 = 1 - GPIO.input(17) and s2 = 1 - GPIO.input(13).

(b) Movement

```
if self.stage != "bucket" and self.stage != "autonav":
    if s2 == 1 and s1 == 0:
        self.direction = "right"
        self.right()
     #left
    elif s1 == 1 and s2 == 0:
        self.direction = "left"
        self.left()
    elif s1 == 1 and s2 == 1:
        self.direction = "forward"
        self.forward()
    else:
        self.direction = "stop"
        self.stop()
```

When both the IR sensors detect a black line, which sets s1 and s2 to '0', it will stop. On the other hand, when there is no black line detected, and hence both s1 and s2 are '1', the turtlebot will continue to move forward. For turning movements, they are dependent on which IR sensor detects a black line. If the left IR sensor detects a black line, it would turn left. Similarly, the bot will turn right when the right IR sensor detects a black line. To ensure this function is not miscalled, two if conditions are used which ensure that it is not at neither the bucket nor autonav stage. This is done through the statement `if self.stage != "bucket" and self.stage != "autonav"`. This means that it will only follow the black line in stages other than the bucket and autonav stages.

(c) Speed of motion

While the line follower code is correct and the IR sensors are accurate, the speed of the turtlebot needs to be adjusted accordingly. This is to prevent the turtlebot from moving or turning too fast and thus, accidentally skipping a black line or going out of track. As such, the speed of the bot was adjusted to

- rotatechange = 1.0
- speedchange = 0.15

These values will then be used for different movements and turnings depending on which one is called. The functions defined are shown below.

```python
def forward(self):
    twist = Twist()
    twist.linear.x = speedchange
    twist.angular.z = 0.0
    # start movement
    self.publisher_.publish(twist)

def backward(self):
    twist = Twist()
    twist.linear.x = -1*speedchange
    twist.angular.x = 0.0
    self.publisher_.publish(twist)

def right(self):
    twist = Twist()
    twist.linear.x = 0.0
    twist.angular.z = -1*rotatechange
    # start movement
    self.publisher_.publish(twist)

def left(self):
    twist = Twist()
    twist.linear.x = 0.0
    twist.angular.z = rotatechange
    # start movement
    self.publisher_.publish(twist)
```

8.3.2 HTTP Call and Respective Movement

(a) POST Request

```python
class ESPServer(Node):
    def __init__(self):
        super().__init__('esp_server')
        self.publisher_ = self.create_publisher(String, 'door', 10)
        timer_period = 0.5
        self.timer = self.create_timer(timer_period, self.server_callback)
        self.subscription = self.create_subscription(
                String,
                'stage',
                self.listener_callback,
                10)
        self.subscription
        self.startCall = False

    def server_callback(self):
        if self.startCall:
            self.get_logger().info("STARTING HTTP CALL")
            door = "0"
            try:
                door = open_door("192.168.65.116", 39)
            except:
                print("Trying to connect")
            msg = String()
            msg.data = door
            self.publisher_.publish(msg)
            self.get_logger().info('Publishing Door: "%s"' % msg.data)

    def listener_callback(self, msg):
        self.get_logger().info('Received Stage: "%s":' % msg.data)
        if (msg.data == "server"):
            self.startCall = True
```

In this class, we will create two functions: `server_callback` and `listener_callback`. `server_callback` is used to call the function `open_door` which will do a POST request for the doors by sending the esp_id and turtlebot_id. For `listener_callback`, it is to receive

the stage we are in from linefollower.py. Once it receives stage as “server”, `self.startCall` will be set to ‘True’ which will allow the code within `server_callback` to be executed. Otherwise, the codes within `server_callback` will not be executed.

```
def open_door(esp_ip, robot_id):

    url = 'http://'+ esp_ip+ '/openDoor'
    payload = {
            "action": "openDoor",
            "parameters": {"robotId" : robot_id}
            }
    headers = {'Content-Type': 'application/json'}

    #while True:
    response = requests.post(url,json=payload,
headers=headers)
    if response.status_code == 200:
        response_data = response.json()
        return response_data['data']['message']

    elif (response.status_code == 400):
        response_data = response.json()
        print("Error:",response_data['data']['message'])
        return "0"
```

The function `open_door` will be called to do the POST request. The turtlebot, from the RPi, will send out some parameters to the ESP server which are the url with the esp_id, payload which has our robot_id and headers. Afterwards, a response will be received by the RPi. Based on the status_code from the response, if it receives code ‘200’, it shows a successful request and the bot will receive the number of the unlocked door. Otherwise a ‘400’ status_code is received signifying an error and no door will be unlocked.

(b) Left or Right Turn

`POST_request.py` publishes a topic “door” which contains the door number obtained from the call. On the other hand, `linefollower.py` subscribes to the topic “door” and it will receive the unlocked door number. The number will then be saved in `self.door` to be used in deciding whether the bot turns left or right through the code shown below. If ‘door1’ is saved in `self.door`, the bot will turn left. Otherwise, it will turn right.

The turtlebot will first move forward for a few milliseconds. Afterwards, in the case when ‘door1’ is received, the left IR sensor, which is s2, will be the point of reference. It will initially turn while the left IR sensor does not detect black. Once it detects a black line, it will go through another while loop to make it keep turning left until it does not detect a black line. Finally, it will go back to the normal line following until it reaches the unlocked door. A similar execution will happen for the case of turning right in which the right IR sensor, s1, is now the point of reference.

```
if self.stage == "server":
    #assuming door 1 is on the left
    if (self.door == "door1"):
        while s1==0 or s2 == 0:
            self.forward()
            s1 = 1-GPIO.input(ir1)
            s2 = 1-GPIO.input(ir2)
        while s2 == 1:
            self.left()
            s2 = 1-GPIO.input(ir2)
        while s2 ==0:
            self.left()
            s2 = 1-GPIO.input(ir2)
        self.doneServer = True
    #if rpi received that door 2 is unlocked
    elif (self.door == "door2"):
        while s1 == 0 or s2 == 0:
            self.forward()
            s1 = 1-GPIO.input(ir1)
            s2 = 1-GPIO.input(ir2)
        while s1 == 1:
            self.right()
            s1 = 1-GPIO.input(ir1)
        while s1 == 0:
            self.right()
            s1 = 1-GPIO.input(ir1)
        self.doneServer = True
```

8.3.3 Ball Dropper and Reversing

(a) Activating Servo

When the bot is at the bucket stage, the servo will be activated to ‘open the cap’ of the ball dropper. This will release all the balls into the bucket.

```
GPIO.setup(servo_pin, GPIO.OUT)
servo = GPIO.PWM(servo_pin,50)
servo.start(5.9)


def drop(args=None):
    servo.ChangeDutyCycle(2)
    time.sleep(5)
    servo.ChangeDutyCycle(5.5)
```

(b) Reversing

In `linefollower.py`, after dropping all the balls into the bucket, the bot will reverse back for 0.5 seconds and next, it will turn right for 2 seconds. This is to make the bot face the lift door again.

```
 if self.stage == "bucket":
      msg = String()
      msg.data = self.stage
      self.publisher2_.publish(msg)
      self.get_logger().info('Publishing Stage: "%s"' %
msg.data)
      time.sleep(5)
      #self.direction = "backward"
      self.backward()
      self.delay(0.5, "backward")
      #self.direction = "right"
      self.right()
      self.delay(2, "right")
      s1=  1-GPIO.input(ir1)
      s2 = 1-GPIO.input(ir2)
      while(s1 == 1 and s2 == 1):
          self.right()
          s1=  1-GPIO.input(ir1)
          s2 = 1-GPIO.input(ir2)
      self.stage = "reverse"
```

The turtlebot will keep turning right until one of the IR sensors detect a black line. Afterwards, it will go back to normal line following code until it reaches the junction in between the two lift doors. As soon as it stops, it will move to the next stage by updating the self.stage from “reverse” to “autonav”.

8.3.3 Maze Navigation Algorithm

a). Frontier Finding

The algorithm works by first finding the individual frontiers in the occupancy map. This is done by analysing the information from the occupancy map. The function works by using a couple of variables, there are 2 ‘queues’: one queue, ‘qm’ is used to note down the list of points to be opened by the map; whilst ‘qf’ is specifically used to find adjacent frontier points. There is the dictionary named ‘markmap’ which is used to mark individual grid points that have been explored. There are also several functions that work in the scope function which are *isFrontier,* which checks whether a point is a frontier, *mark* to view the mark of the frontier and *adj*, to generate adjacent points from a given point. Finally, there are some lists, nf is used to temporarily store the grouped coordinates of each frontier, and frontiers is used to store all the frontiers. In the beginning, the robot’s starting position is queued into ‘qm’. The starting point is then marked as ‘map-open’ meaning that it has already been opened by the map.

```
#Function to find frontiers
def findfronteirs(tmap,posi,map_res):
    frontiers = []
    markmap ={}

    #Function to check if a position is a fronteir or not
    def isFronteir(pos):
        if tmap[pos[0],pos[1]]   == 2:
            if pos[0] ==0:
                temp = tmap[:2,:]
            elif pos[0] == len(tmap)-1:
                temp = tmap[pos[0]-1:,:]
            else:
                temp = tmap[pos[0]-1:pos[0]+2,:]

            if pos[1] == 0:
                a = temp[:,:2]
            elif pos[1] == len(tmap[0])-1:
                a = temp[:,pos[1]-1:]
            else:
                a = temp[:,pos[1]-1:pos[1]+2]
            return np.any(a==1)
        return False

    #Function to check if a cell has been marked
    def mark(p):
        return markmap.get(p,'Unmarked')

    def adj(p):
        return Adj(p,tmap)

    #Creating the map queue
    qm = []

    #Adding robot's current position to the map queue
    qm.append(posi)
    markmap[posi] = "Map-Open-List"
```

Segment 1 of Frontier Identification code

A while loop is used to iterate through all elements in qm. After obtaining the first element, the algorithm will check if the loop has been checked by the map before by using the *mark* function. If the point is marked as “Map-Close-List”, it means that the point has already been checked by the map and the loop will continue to the next point. Otherwise, the function will check if the point is a frontier, which is defined as a point with a ‘known empty space’ signified by 2 on the occupancy map, that is bordering an ‘unknown space’ signified by a 1 on the occupancy map. If the point isn’t identified as a frontier, the code will generate new neighbours. Any neighbours that haven’t been marked, returning the ‘Unmarked’ label when run through the *mark* function, will be queued into the qm queue. On the other hand, if the point is indeed a frontier, the function will run another while loop iterating through qf. qf is initialised with the element being queued into qf. For each element in qf, the point will be marked as ‘Frontier-Open-List’ initially and dequeued from qf. if it is found to be a frontier, the coordinate will be added to the nf list and new neighbour coordinates will be generated, the coordinate will also now be marked ad ‘Frontier-Closed-List’ to avoid double counting of coordinates. The neighbour coordinates that have not been mapped are put into qf. This will repeat until all points belonging to a specific frontier are put into nf. If the list nf has enough

coordinates points to verify that the frontier is of adequate length, the points in nf will be put into the frontiers list. All the points within nf will then be marked as “Map-Close-List” meaning that they have already been checked. The resulting output is an array of all existing frontiers on the map. Each frontier is a list consisting of the coordinate points belonging to that frontier.

```
while qm:
    #Dequeueing first element of map queue
    p = qm.pop(0)

    #Skipping the element if it has already been mapped
    if mark(p) == 'Map-Close-List':
        continue

    #Checking if the cell is a frontier
    if isFronteir(p):

        #Creating a queue for finding adjacent frontiers qf
        #Creating a list of frontier points nf

        qf = []
        nf = []
        qf.append(p)

        #Marking the point as to be opened by the frontier
        markmap[p] = "Frontier-Open-List"

        while qf:
            #Dequeuing the first element of qf
            q = qf.pop(0)

            #Skipping the cell if it's already been mapped or if it has been searched through
            if mark(q) in ["Map-Close-List","Frontier-Close-List"]:
                continue
            #If the point is a frontier, add it to the nf arrray
            if isFronteir(q):
                nf.append(q)
                #Check for cells adjacent to current frontier point
                for w in adj(q):
                    #If there is a neighbor that has not been added to qf, and has not been mapped or checked add it to the queue
                    if mark(w) not in ["Frontier-Open-List",'Frontier-Close-List','Map-Close-List']:
                        qf.append(w)
                        #Mark the newly added cell as frontier open list
                        markmap[w] = "Frontier-Open-List"
            #Mark the original point as being checked
            markmap[q] = 'Frontier-Close-List'
        #Adds the list of frontier points to the frontiers array
        if len(nf) > round(threshold/map_res):
            frontiers.append(nf)
        #Marking all points in the nf array as being closed by the map
        for i in nf:
            markmap[i] = "Map-Close-List"
    #Generating neighbors for the cells in the map queue
    for v in adj(p):
        #Adding any neighbors that have not been opened or checked
        if mark(v) not in ["Map-Open-List","Map-Close-List"]:
            if any([tmap[x[0],x[1]] == 2 for x in adj(v)]):
                    qm.append(v)
                    markmap[v] = "Map-Open-List"
    #Marking the point as checked in the map
    markmap[p] = "Map-Close-List"
#returning the final list of frontiers
return frontiers
```

Segment 2 of Frontier Identification Code

b). Path Finding (A* algorithm)

Our pathfinding algorithm uses A* to find a path from the robot to the target frontier. The algorithm works by iterating through the elements in the open_set list, whilst adding the neighbours of each element to the open_set list itself. A* also uses heuristic costs to determine the priority of points to check. There is also the ‘camefrom’ dictionary to store a point’s ‘parent’ position that is used to return a path.

The heuristics used for our algorithm are euclidean distance as the main heuristics, with an additional wall penalty heuristic. We are using euclidean distance, since the robot is not limited by the orientation of the grid, hence the distance from the robot to the target is denoted by the euclidean distance from the robot to the target. Additionally, to prevent the algorithm from sticking to the wall, an additional heuristic named the wall penalty is added. This cost is proportional to the point's position to a wall within a 25 cm radius. This allows for the path points to become distanced from the wall without being troubled by smaller openings.

```python
def heuristic(node, target, occupancy_data, map_res):
    nrows = len(occupancy_data)
    ncols = len(occupancy_data[0])
    max_penalty_distance = round(0.25/map_res)
    # Euclidean distance between current node and target
    euclidean_distance = ((node[0] - target[0])**2 + (node[1] - target[1])**2) ** 0.5

    # Penalties for proximity to walls
    wall_penalty = 0 # You can adjust this distance as needed

    for dx in range(-max_penalty_distance, max_penalty_distance + 1):
        for dy in range(-max_penalty_distance, max_penalty_distance + 1):
            nx, ny = node[0] + dx, node[1] + dy
            if 0 <= nx < nrows and 0 <= ny < ncols and occupancy_data[nx, ny] == 3:  # Wall detected
                distance_to_wall = ((dx**2 + dy**2) ** 0.5)
                wall_penalty += max_penalty_distance - distance_to_wall
```

Heuristic function for A algorithm*

Initially, the g_score and f_score heuristics are set for the starting point which is the robot. The code then enters a while loop to iterate through the open_set. The heappop function is used to pop the element of the open_set with the smallest heuristic cost, the popped value is then stored into a variable named current. If the position in current is equal to the target position, the function would backtrack the path the current position came from to return a list of coordinates coming from the robot's location and leading towards its target. However, If the current position is not the target, it is added to the closed_set to indicate that it has been checked. New neighbors are generated from the current position.

```python
#Generating neighbors for A*
def find_neighbors(node, occupancy_data, nrows, ncols):
    neighbors = []
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1),(1,1),(1,-1),(-1,1),(-1,-1)]
    for dir in directions:
        neighbor = (node[0] + dir[0], node[1] + dir[1])
        if 0 <= neighbor[0] < nrows and 0 <= neighbor[1] < ncols:
            if occupancy_data[neighbor[0], neighbor[1]] not in  (1,3):  # Check if it is not wall
                neighbors.append(neighbor)
    return neighbors
```

find_neighbors function for A algorithm*

Neighbors are generated with the find_neighbors function. We use this function to generate all adjacent positions that are not obstacles within the map. Neighbor positions that are not in the closed_set are given a tentative g_score, which is the g_score of the current position plus one, indicating it takes one more step to reach the neighboring position than the current position. If the neighbor has not placed into the open_set, or if the new g_score is lower than the neighbor's current g_score in open_set, the current point replaces the neighbor's 'parent' in the camefrom dictionary and the g_score and the f_score are set to the tentative g and f scores. The neighbor is then put into the open_set to be iterated through, until the target is

reached. If all points have been exhausted, then the function would return None, meaning there is no path to the target.

```python
#Function to return a nearest path from one point to another
def a_star(start, target, occupancy_data, nrows, ncols, map_res):
    open_set = []
    closed_set = set()
    heapq.heappush(open_set, (0, start))
    came_from = {}

    g_score = {start: 0}
    f_score = {start: heuristic(start, target,occupancy_data, map_res)}

    while open_set:
        current = heapq.heappop(open_set)[1]

        if current == target:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            path.reverse()
            return path

        closed_set.add(current)

        for neighbor in find_neighbors(current, occupancy_data, nrows, ncols):
            if neighbor in closed_set:
                continue

            tentative_g_score = g_score[current] + 1  # Assuming cost of moving from one cell to another is 1

            if neighbor not in [x[1] for x in open_set] or tentative_g_score < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = tentative_g_score + heuristic(neighbor, target, occupancy_data,map_res)
                heapq.heappush(open_set, (f_score[neighbor], neighbor))

    return None  # No path found
```

Main function for the A algorithm*

c). Occupancy Map Acquisition & Treatment

To obtain the occupancy map, the algorithm uses a subscription to the OccupancyGrid from SLAM with the topic of 'map'. For each callback, the robot obtains a 2D numpy array in the form of the callback's message. In the numpy array, each coordinate (y,x) contains a value corresponding to the confidence level ranging from 0 to 100 that the SLAM thinks there is an obstacle, with -1 representing unknown space. To simplify the data, we are using the Scipy module to separate the values into bins ranging from 1 (ranging from -1 to 0), 2 (ranging from 0 to 55), 3 (ranging from 55 to 100). In the occupancy map, this means that 1 represents all space that has not been mapped, 2 represents mapped space that does not have an obstacle, and 3 represents mapped space that does have an obstacle. This data is then stored in a 2D numpy array named odata.

```python
# create subscription to track occupancy
self.subscription = self.create_subscription(
    OccupancyGrid,
    'map',
    self.listener_callback,
    qos_profile_sensor_data)
```

Subscription to the 'map' topic

```
def listener_callback(self, msg):
    if self.start_autonav:
        # create numpy array
        occdata = np.array(msg.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)
```

Creating a binned statistics object from the msg data

```
odata = np.uint8(binnum.reshape(msg.info.height,msg.info.width))
```

Saving the new statistics into the odata numpy array

The odata array provides a decent representation on the occupancy map of the maze, but it is not entirely reliable to be used raw. It is not rare that the LiDAR hallucinates when receiving a bad data reading and projects ‘leaks’ in the walls of the maze, leading to frontiers being detected outside of the map. Throughout testing, the most effective method to negate this is through virtually expanding the walls. We do this by creating a copy of the odata array named ndata. For each wall (point of value 3) in odata, the corresponding point in ndata along with a number of points neighboring it are changed to reflect an obstacle in that coordinate. This ndata is the data grid that will be used to calculate possible paths to reach the frontier.

```
#creating a modified occupancy map with expanded walls
ndata = np.copy(odata)
size = round(wall_size/map_res)
#size = 1
for i in range(len(odata)):
    for j in range(len(odata[0])):
        if odata[i,j] == 3:
            for k in range(-size,size):
                for l in range(-size,size):
                    if (0 < i+k < len(odata) and 0 < j+l < len(odata[0])) and abs(i+k - grid_y) + abs(j+l - grid_x) > 0.40/map_res :
                        ndata[i+k,j+l] = 3
```

Creation of ndata numpy array with ‘expanded walls’

d). Crash Detection

While A* provides a good path towards the frontier, it works on the assumption that the robot is a singular 1D object. This is in fact not true as the turtlebot has volume and takes up space. This means that if the robot follows the path provided by A* exactly, it could be prone to accidents like hitting a wall. To minimize the risk of crashing, a separate function is used to detect walls and potential crashes. The robot subscribes directly to the LiDAR data with the ‘scan’ topic.

```
# create subscription to track lidar
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([])
```

Creating a subscription to the ‘scan’ topic

Within the callback function, the robot will continuously scan for the front angles of the robot. In our implementation we chose the angles -30 to 30. The robot will continuously search for an obstacle with a distance smaller than a set ‘stop_distance’. If such an object is detected, the function shall turn the isCrashing attribute to True, the function will also stop the robot if crash avoidance is not in play. There is also an exception to the callback where the isCrashing attribute will not be triggered if the robot is currently rotating towards a target. It will only activate after the robot has finished rotating to a target.

```
def scan_callback(self, msg):
    if self.start_autonav:
        # self.get_logger().info('In scan_callback')
        # create numpy array
        self.laser_range = np.array(msg.ranges)
        # replace 0's with nan
        self.laser_range[self.laser_range==0] = np.nan

        #Crashing triggers when an object is at a certain distance in front of the robot
        #Crashing does not trigger during rotation towards a target
        #Robot should stop unless is doing crash avoidance
        for i in range(front_angle):
            if (self.laser_range[i] != np.nan and self.laser_range[i] < float(stop_distance)) and (not self.isSpinning) and self.target:
                self.crashAngle = i
                self.isCrashing = True
                print("CRASHHHH")
                if not self.avoidingCrash:
                    self.stopbot()
                break
            elif (self.laser_range[-1*i] != np.nan and self.laser_range[-1*i] < float(stop_distance)) and (not self.isSpinning) and self.target:
                self.crashAngle = 360 - i
                self.isCrashing = True
                print("CRASHHHH")
                if not self.avoidingCrash:
                    self.stopbot()
                break
            else:
                self.isCrashing = False
```

Callback function to detect potential crashes

e) Robot Movement

In order to move the robot, two main functions are used in navigation. Both of the functions work by publishing a twist vector to the ‘cmd vel’ topic. The Twist vector contains instructions on the linear and angular velocity for the robot. The two main attributes used are twist.linear.x which controls the forward velocity of the robot, and twist.angular.z which controls the angular speed in the yaw direction of the robot.

```
# create publisher for moving TurtleBot
self.publisher = self.create_publisher(Twist, 'cmd_vel', 10)
```

Creation of a publisher

The first function, stopbot, is used to stop the robot entirely, publishing both the linear and angular velocity of the robot to be zero. The second function, rotatebot, is used to rotate the robot a certain amount of degrees. It provides an angular speed to the robot’s twist until the robot has reached the desired angle. It does this by continuously checking if the yaw has reached the desired yaw.

```python
#Stops the robots
def stopbot(self):
    #self.get_logger().info('In stopbot')
    # publish to cmd_vel to move TurtleBot
    twist = Twist()
    twist.linear.x = 0.0
    twist.angular.z = 0.0
    # time.sleep(1)
    self.publisher.publish(twist)
```

Creation of the stopbot function

```python
#Rotates the bot a certain angle
def rotatebot(self, rot_angle):
    self.get_logger().info('In rotatebot')
    if abs(rot_angle) < 40:
        rotatechange = 0.2
    else:
        rotatechange = 0.4

    # create Twist object
    twist = Twist()

    # get current yaw angle
    current_yaw = self.yaw
    # log the info
    # 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))
    # 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)
    self.isSpinning = True
```

```python
# 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
    # if self.angularspeed == 0:
    #     break\
    self.isSpinning = True
    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))
    # self.get_logger().info('Target Yaw: %f' %math.degrees(target_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))

# set the rotation speed to 0
twist.angular.z = 0.0
# stop the rotation
self.publisher.publish(twist)
self.isSpinning = False
```

Rotatebot function

These functions are then used to form the function movetotarget. This function uses the arctan method of the numpy module to calculate the relative angle of the target from the robot's current position. It then uses the rotatebot function to turn the robot to face the target. Once the turtlebot has faced the target, there is a rclpy.spin_once function to run a callback function to check if a wall is in front of the bot.

```
#Main method to move towards a target
def movetotarget(self, target):
    if target:
        #Locate relative yaw of the target
        angle = np.arctan((target[0]-self.y)/(target[1]-self.x))
        if (target[1]-self.x) < 0:
            if (target[0]-self.y) > 0:
                angle += np.pi
            else:
                angle -= np.pi
        #print(np.degrees(self.yaw))
        #print(np.degrees(angle))
        angle = angle - self.yaw
        #print(np.degrees(angle))

        if angle > np.pi:
            angle = angle - 2 * np.pi
        if angle < -np.pi:
            angle = angle + 2 * np.pi

        #Rotate towards the target
        if abs(np.degrees(angle)) > 10 :
            self.stopbot()
            self.rotatebot(np.degrees(angle))

        #Callback to check for crashing
        rclpy.spin_once(self)
```

Initial rotation towards the target's direction

If an obstacle is detected in front of the bot, crash avoidance will trigger. The robot will scan the angles close to the obstacle's angle and identify the leftmost and rightmost edge angles of the object. The algorithm then calculates the mean of the object to estimate where the object is located. The robot is then given an angular velocity moving away from the obstacle. This will continue until the robot does not detect an object in front of it. Once the object is successfully avoided the robot is programmed to move forward a certain distance and then continue to move towards the target. If however, the robot did not detect an object in front, the robot would be given a linear velocity forward to move towards the target.

```
    if (self.isCrashing):
        # Handles crash avoidance
        print("Avoiding crash")
        self.stopbot()
        closestAngle = np.nanargmin(self.laser_range)
        degreeNum = len(self.laser_range)

        for i in range(len(self.laser_range)):
            print(i, self.laser_range[i])

        leftFound = False
        rightFound = False

        #Looks for open angles with 35cm clearance
        for i in range(0,round(degreeNum/2)):
            anglePos = (closestAngle +i)%degreeNum
            angleNeg = (closestAngle- i)%degreeNum
            try:
                if (not rightFound) and abs(self.laser_range[anglePos]) > 0.35:
                    posDisplace = i
                    rightFound = True
                    #print(anglePos)
                angleNeg = (closestAngle - i)%degreeNum
                if (not leftFound) and abs(self.laser_range[angleNeg]) > 0.35:
                    negDisplace = i
                    leftFound = True
                    #print(angleNeg)
                if leftFound and rightFound:
                    break
            except:
                continue
        if not leftFound:
            negDisplace = round(degreeNum/2)
        if not rightFound:
            posDisplace = round(degreeNum/2)
        #print(posDisplace,negDisplace)
        posAngle = (closestAngle + posDisplace)%degreeNum
        negAngle = (closestAngle - negDisplace)%degreeNum
        #print(posAngle,negAngle)
        if posAngle > degreeNum/2:
            posAngle = posAngle - degreeNum
        if negAngle > degreeNum/2:
            negAngle = negAngle - degreeNum

        if posAngle + negAngle < 0:
            print('LEFT\n\n\n\n')
        else:
            print('RIGHT\n\n\n\n')
```

```
        # print("Pos Angle = {}, negAngle = {}".format(posAngle,negAngle))
        # print("destination angle: ",destinationAngle)
        while self.isCrashing:
            rclpy.spin_once(self)
            #Moves towards the clear space
            self.avoidingCrash = True
            twast = Twist()
            twast.linear.x = 0.0
            if posAngle + negAngle > 0:
                angular = -0.2
            else:
                angular = 0.2
            twast.angular.z = angular
            self.publisher.publish(twast)
            time.sleep(0.1)
        self.stopbot()
        print('crash avoided')
        rclpy.spin_once(self)

        #Moves forward if robot is not detecting a crash
        if not self.isCrashing:
            twist = Twist()
            twist.linear.x = 0.2
            twist.angular.z = 0.0
            self.publisher.publish(twist)
            print('moving away')
            time.sleep(1)
        self.avoidingCrash = False
        self.stopbot()
    else:
        # Handles normal movement to target
        print('Start moving')
        twist = Twist()
        twist.linear.x = 0.2
        twist.angular.z = 0.0
        self.publisher.publish(twist)
rclpy.spin_once(self) #get the lidar data to prevent crash
```

Crash avoidance and Movement Forward Segment of the Script

f) Path Navigation and Integration

The path creation and navigation mainly happens within the *listener_callback* function. At the start of the callback, the algorithm checks if the robot's target is valid, which means it checks whether the target is an obstacle. If the target is viewed as invalid, the target and path are reset, targetReached is also set to True to trigger the creation of a new path. The algorithm will also check if the robot is close enough to the target and that there is no obstacle between the robot and the target. If these conditions are true, targetReached will be set to True.

```
#Checks if target has been reached
if self.target:
    mapTarget = (round((self.target[0] - map_origin.y)/map_res), round((self.target[1]-map_origin.x)/map_res))
    if odata[mapTarget[0],mapTarget[1]] == 3:
        self.target = []
        self.path = []
        self.targetReached = True
if not self.target or (((self.target[1]-self.x)**2 + (self.target[0]-self.y)**2)**0.5 < proximity_limit and\
                       not is_wall_between(odata, (grid_y,grid_x), mapTarget, map_res) and not self.isSpinning):
    self.targetReached = True
    self.isCrashing = False
    print('target Reached')
    self.stopbot()
```

Checking if the Target is Valid

If the targetReached attribute is True, the algorithm will begin to clear points near the bot. The points will be cleared within a certain radius if they are not blocked by a wall or obstacle. The algorithm shall also check if any of the points in the path are a wall, if so, the path and target will be reset. If none of the path reset conditions are triggered, the target will be taken out of the stack of the path.

```
if self.targetReached:
    self.stopbot()

    #Creating a modified occupancy map with expanded walls
    ndata = np.copy(odata)
    size = round(wall_size/map_res)
    #size = 1
    for i in range(len(odata)):
        for j in range(len(odata[0])):
            if odata[i,j] == 3:
                for k in range(-size,size):
                    for l in range(-size,size):
                        if (0 < i+k < len(odata) and 0 < j+l < len(odata[0])) and abs(i+k - grid_y) + abs(j+l - grid_x) > 0.40/map_res :
                            ndata[i+k,j+l] = 3

    self.targetReached = False
    if self.path:
        #Checks if current frontier is stil a frontier
        if self.currentFrontier:
            print("Checking Fronteir")
            if not isFronteir(odata, (round((self.currentFrontier[0]-map_origin.y)/map_res),round((self.currentFrontier[1]-map_origin.x)/map_res))):
                #If frontier has already been mapped clear path, target and frontier
                self.path = []
                self.currentFrontier = []
                self.target = []
                print("Frontier Removed")

        #Removing points close to the robot
        remove_index = []
        goesThroughWall = False
        for i in self.path:
            if ((i[0] - self.y)**2 + (i[1] - self.x)**2)**0.5 < target_limit:
                if not is_wall_between(ndata, (grid_y,grid_x),(round((i[0]-map_origin.y)/map_res),round((i[1]-map_origin.x)/map_res)), map_res):
                    remove_index.append(i)
                elif odata[round((i[0]-map_origin.y)/map_res),round((i[1]-map_origin.x)/map_res)] == 3:
                    goesThroughWall = True
                    remove_index.append(i)
            else:
                break
        for i in remove_index:
            self.path.remove(i)
        if goesThroughWall:
            self.path = []

        #If the entier path is removed, reset everything and set targetReached to be true
        if self.path:
            print('Target Set')
            self.target = self.path.pop(0)
        else:
            print('Frontier Filled')
            self.targetReached = True
            self.target = []
```

Clearing Points Near to the Bot

If the path is empty, either through clearing or by being emptied by the target being removed from the path stack, a new path would be created. First, the algorithm looks for frontier points in the map. For each frontier, the code stores the median in an array named midpoint_positions. For each point in the median array, the algorithm tries to run an A* algorithm to find a path towards the frontier. If a path is found, the grid coordinates of the path are converted into real-life conditions of meters. These coordinates are stored into the path attribute. Finally, the target is set as the first element of the path stack and targetReached is set to True to clear the points near to the robot.

```
#If path is empty, initialize path finding
if not self.path:
    self.stopbot()

    #Generates frontier positions
    frontier_positions = findfronteirs(odata,(grid_y,grid_x), map_res)

    #Adds the median of the frontiers to an array
    midpoint_positions = []
    for i in frontier_positions:
        midpoint_positions.append(median(i))


    if midpoint_positions:
        # WALL THICKENING
        odata = ndata
        a_star_list = []

        for i in range(len(midpoint_positions)):
            self.stopbot()
            print('astar running')
            test_target = midpoint_positions[i]

            # Searches through the array for travellable frontiers
            a_star_list = a_star((grid_y, grid_x), (test_target[0], test_target[1]), odata, iheight, iwidth, map_res)
            if a_star_list:
                print(a_star_list)
                break

        #Converts path fron A* algorithm into real coordinates in meters
        if a_star_list:
            path = a_star_list
            realpath = [] #path in real coordinates
            for point in path:
                realpath.append([point[0] * map_res + map_origin.y, point[1] * map_res + map_origin.x])
            self.path = realpath[1:]
            self.targetReached = True
            self.isCrashing = False
            self.currentFrontier = self.path[-1]
            #print('path' , self.path)
            self.target = self.path.pop(0)
        else:
            self.target = []
            print('no path found')
            self.stopbot()
```

Creation of New Path

These functions are used in the Move method in the node, the code runs by stopping the bot if the target has been reached and otherwise using the movetotarget function to move towards its next target.

```
    def Move(self):
        while (True):
            rclpy.spin_once(self)
            #Only activates if stage is in autonav
            if self.start_autonav:
                try:
                    if self.targetReached:
                        self.stopbot()
                    else:
                        self.movetotarget(self.target)
                except Exception as e:
                    print(e)
                    print('b')


def main(args=None):
    rclpy.init(args=args)

    occupy = Occupy()

    # create matplotlib figure

    occupy.Move()

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    occupy.destroy_node()
    rclpy.shutdown()
```

Move method that integrates the robot movement

8.4 Integration of Scripts

`linefollower.py` is the main script that decides the current stage of the mission. The stage is being published to a topic named “stage”, where other scripts such as `POST_request.py`, `dropball.py`, and `autonav.py` subscribe to this topic. All these scripts will only run when the “stage” sent by `linefollower.py` changes to their corresponding stage.

The stages naming and description is as follows

1. navigation : line follow from starting point to lobby
2. server : HTTP call to retrieve door number
3. door : travel through the open door to the bucket
4. bucket : drop all balls to the bucket
5. reverse : reverse and move back to lobby
6. autonav : autonomous navigation to cover unmapped area

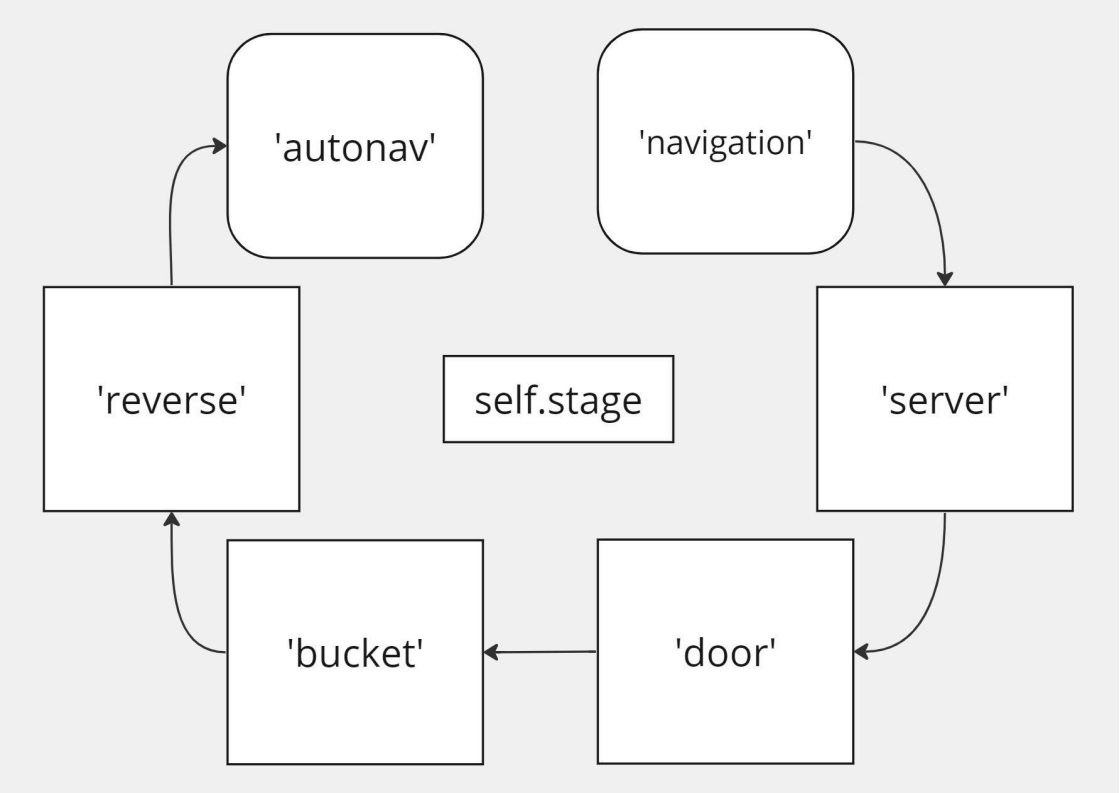


Figure of self.stage cycle

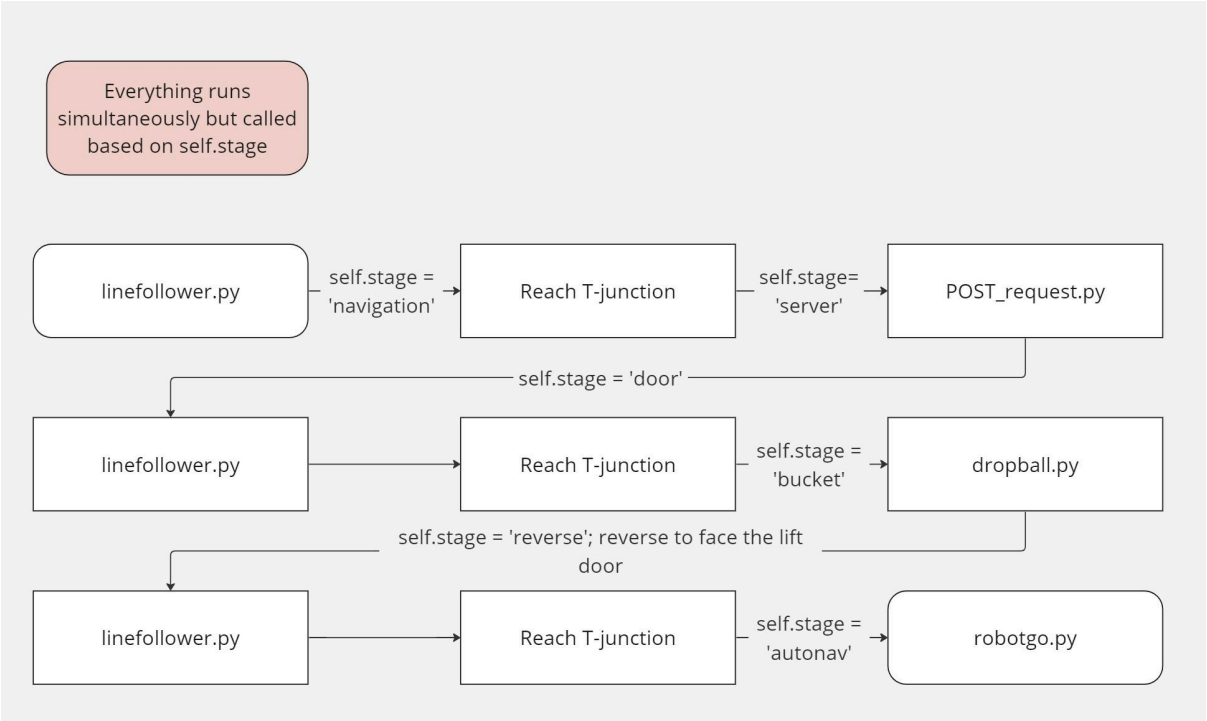
When starting the mission, the values will be initialised to values shown below:

```
#########################################################
# initialize stage to nav
self.stage = "navigation"

# Subscribe to esp server
self.subscription2 = self.create_subscription(
        String,
        'door',
        self.server_callback,
        10)
self.subscription2
################################################
self.door = "0"
self.doneServer = False
```

self.doneServer is a boolean variable to determine whether it has performed HTTP calling and goes through the unlocked door. Only after a successful HTTP call will the value of self.doneServer be set to 'True'. This ensures that HTTP Call will not be repeated once a successful request is received.

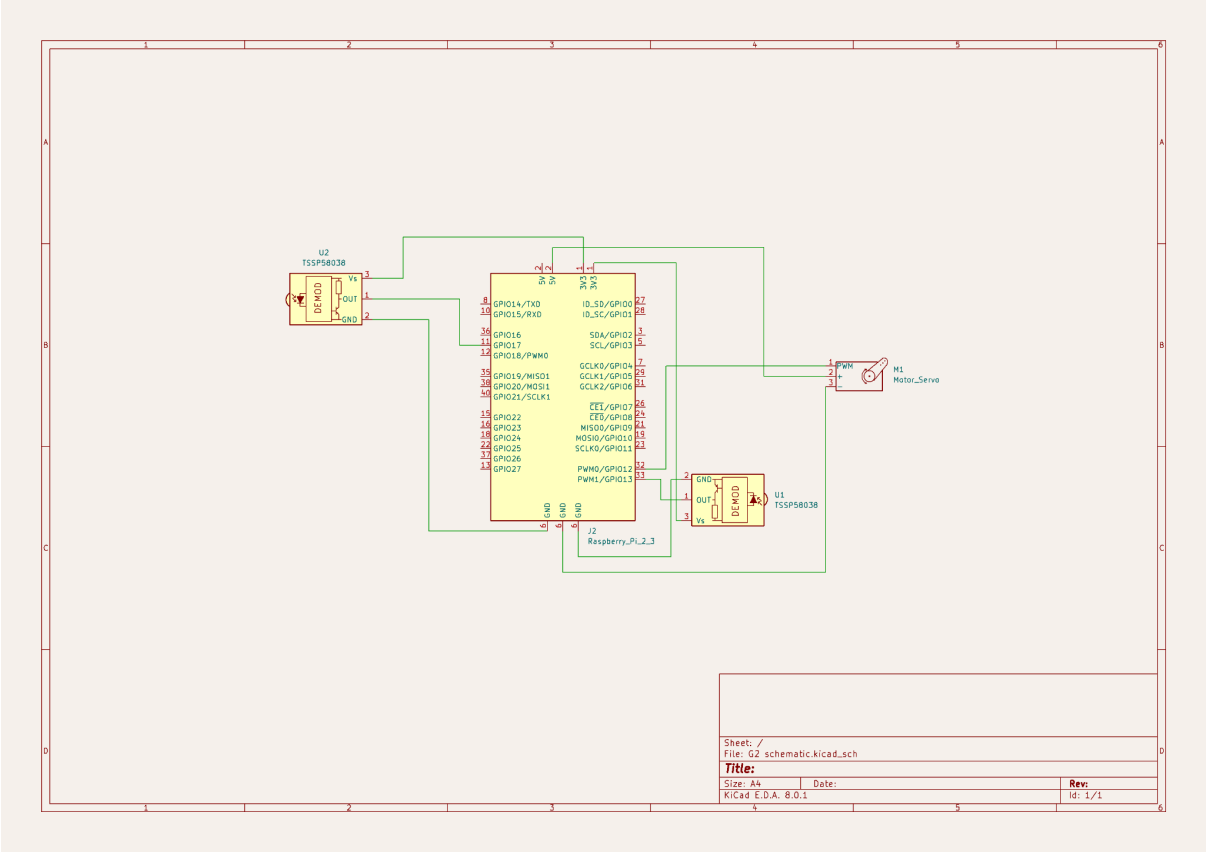
```
if ((s1 == 0) and (s2 == 0)  and (self.direction == 'stop' or self.direction == "forward")):
    if not self.stage == "server":
        self.direction = "stop"
        self.stop()

    if self.stage == "navigation":
        self.stage = "server"

    if self.stage == "door":
        self.stage = "bucket"

    if self.stage == "reverse":
        self.stage = "autonav"
```

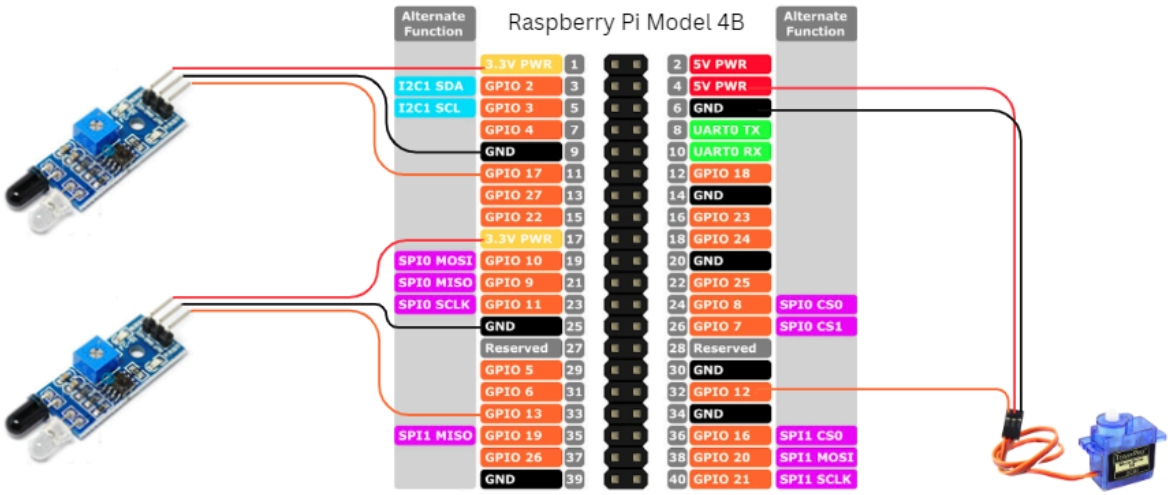
A flowchart of the overall software integration of the scripts is shown below:



8.5 Electrical Assembly

Below are two diagrams showcasing the final electrical connections between the servo, IR sensors and RPi.





9. Final Run Evaluation

This section is dedicated for the final tests and evaluation before the final run. The tests and checks are done before the robot to ensure that the turtlebot is fully functional to perform the mission and tasks while minimising issues. Additionally, this is to certify that the turtlebot has met the required specifications to be able to do the mission for the run.

9.1 Acceptable Defect Log

Defect	Acceptability Reason
LiDAR blocked by the standoff pillars	LiDAR will still be able to map out the maze
Servo motor may make an oscillating noise	Servo is still able to open and close as planned

9.2 Factory Acceptance Test

Part	Competence	Observation
Mechanical		
Turtlebot intact	No loose parts	Bot will move with ease and without shaky parts
Ball Dropper Mechanism	Intact, stable attachment	Able to store the ping pong balls without dropping them
LiDAR	Rotates when powered	Environment will be mapped on Rviz
Wheels (Dynamixel)	Able to move and rotate freely	Bot will traverse the maze with steady turnings and movement at any direction
Electrical		
Battery power	Sufficient power, no alerting sound	Robot is able to move through the maze without any alert sound
OpenCR	Able to be powered by the LiPo Battery	Green LED lights up when connected to power source and boot up tune played
Wiring	No loose wires	No wires sticking out and are pinned to their GPIO pins
	All connected to the correct pins	All functions can be executed well when respective code is called

Servo	Able to open and close accordingly when used	No shaky movements from the servo and opens up well
Software		
Raspberry Pi	Able to perform HTTP POST Request	A door number will be shown upon successful request

9.3 Maintenance and Part Replacement Log

Item	Description	Rectification
IR Sensor	IR Sensor not sensitive enough	Replaced with a new IR Sensor

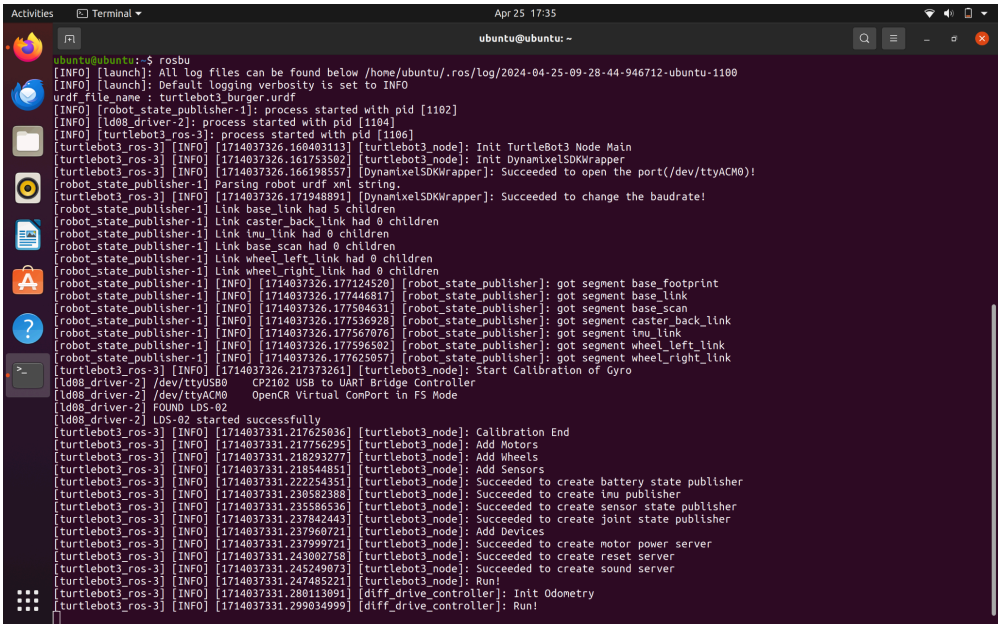
10. Systems Operation Manual

10.1 Charging and Checking Battery Level

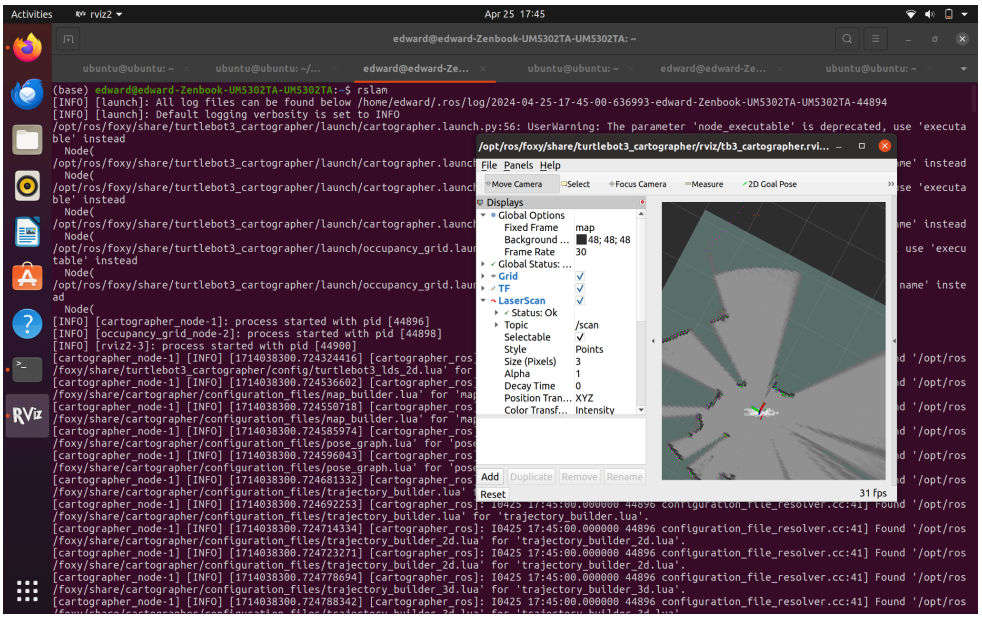
Before each run, it is important to have a fully charged battery as the low voltage could lead to a lower performance of the turtlebot and may cause unexpected behaviour from it. From experience, it is necessary for the battery level to be at least 11.5V to ensure an optimal performance from the turtlebot.

10.2 Software Boot-up Protocol

1. ssh into RPi on 4 separate terminals by inputting `sshrp` and enter `turtlebot` for the password
2. Run all commands below on separate terminals (RPi and Laptop)
3. Run `rosbu` to start up the turtlebot (on RPi)



4. Run `rslam` to get the RViz running (on Laptop)



5. In one terminal window, run the post request script (on RPi)

```
cd r2auto_nav/src/raspberry_pi
python3 POST_request.py
```

6. In another terminal window, run the ball dropper script (on RPi)

```
cd r2auto_nav/src/raspberry_pi
python3 dropball.py
```

7. In another terminal window, run the autonav script (on Laptop)

```
ros2 run auto_nav autonav
```

8. In another terminal window, run the line follower script (on RPi)

```
cd r2auto_nav/src/raspberry_pi
python3 linefollower.py
```

11. Conclusion & Areas of Improvement

Overall, EG2310 has been a very fulfilling journey for our group. It was filled with ups and downs, confusion and celebration during late and early nights. Throughout the process, we got to know one another better in terms of individual personality and skill sets. We handled the tasks well through efficient tasks delegation and finishing them well. Furthermore, our own initiative helped us to progress better as we entrust each other to do certain tasks while we focus on the tasks we had at that moment. This nurtured us to become more responsible and trustworthy engineers. In the process, we had fun while pushing through the intensity of the module and the rigorous lab tasks. We learnt to never overlook any small achievements as we believe that they still matter in the long run as they are the stepping stones to something greater, a fully functional turtlebot.

Despite the successful run thus far, we believe that there are still areas of improvement on our side. Firstly, we could have documented our journey in building the robot better. This would allow us to have a better understanding of the progress of the robot especially when debugging issues within the turtlebot. For instance, we should have documented pin mappings earlier or take note any changes in the code for easier debugging. Another improvement would be to plan better for our tasks. This is because there were instances when we did not know what to do since the items we bought had not arrived yet. Hence, we were stuck in doing nothing much other than writing up the report and creating codes but had no hardware to test them on. Nonetheless, as a team, we had persevered well and collaborated well with one another.

We would like to thank all the TAs and professors for guiding us from the start of the project, challenging our limits in the field of robotics and enlightening us in electrical, mechanical and software engineering aspects. With joy and gratitude, we are proud to say that we are EG2310 survivors!

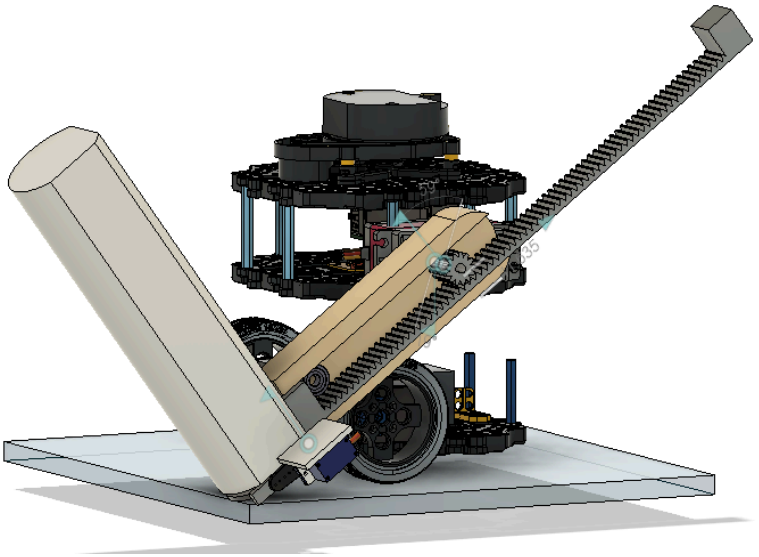
APPENDIX

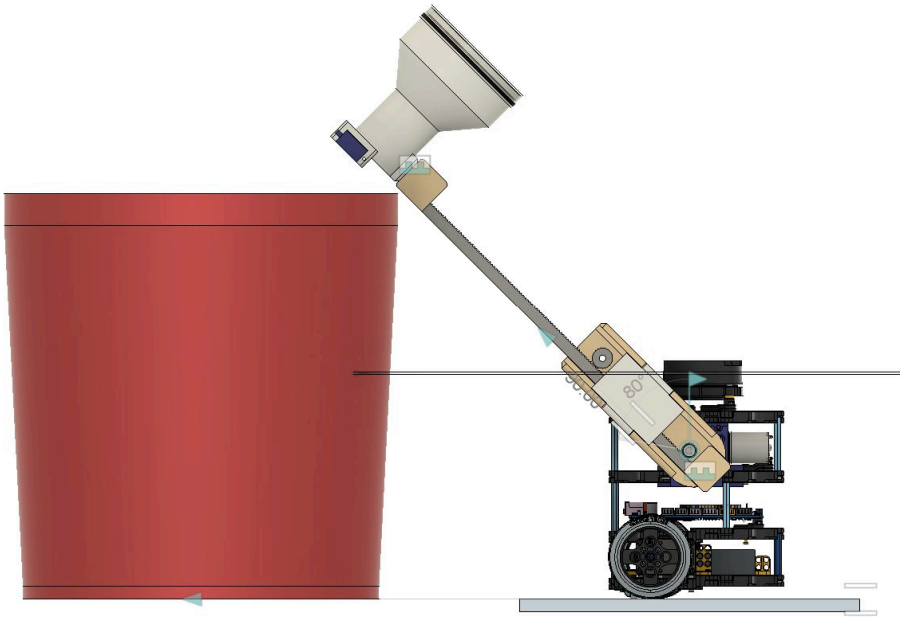
Weight Table

Object (quantity)	Weight of each object (gram)	Total Weight (gram)
Waffle-Plate (8x)	39	312
OpenCR 1.0 (1x)	64	64
Raspberry Pi (1x)	47	47
Wheel (2x)	13	26
Tire (2x)	14	28
360 Laser Distance Sensor LDS-02 (1x)	129	129
Li-Po Battery (1x)	141	141
DYNAMIXEL, RL430 (2x)	55	110
USB Cable (2x)	12	24
DYNAMIXEL to OpenCR Cable (2x)	3	6
Raspberry Pi Power Cable (x1)	2	2
Li-Po Battery Extension Cable (x1)	10	10
LDS-02 Cable (x2)	1	2
Ball Caster (x1)	5	5
USB2LDS (x1)	1	1
SG-90 Servo (1x)	9	9

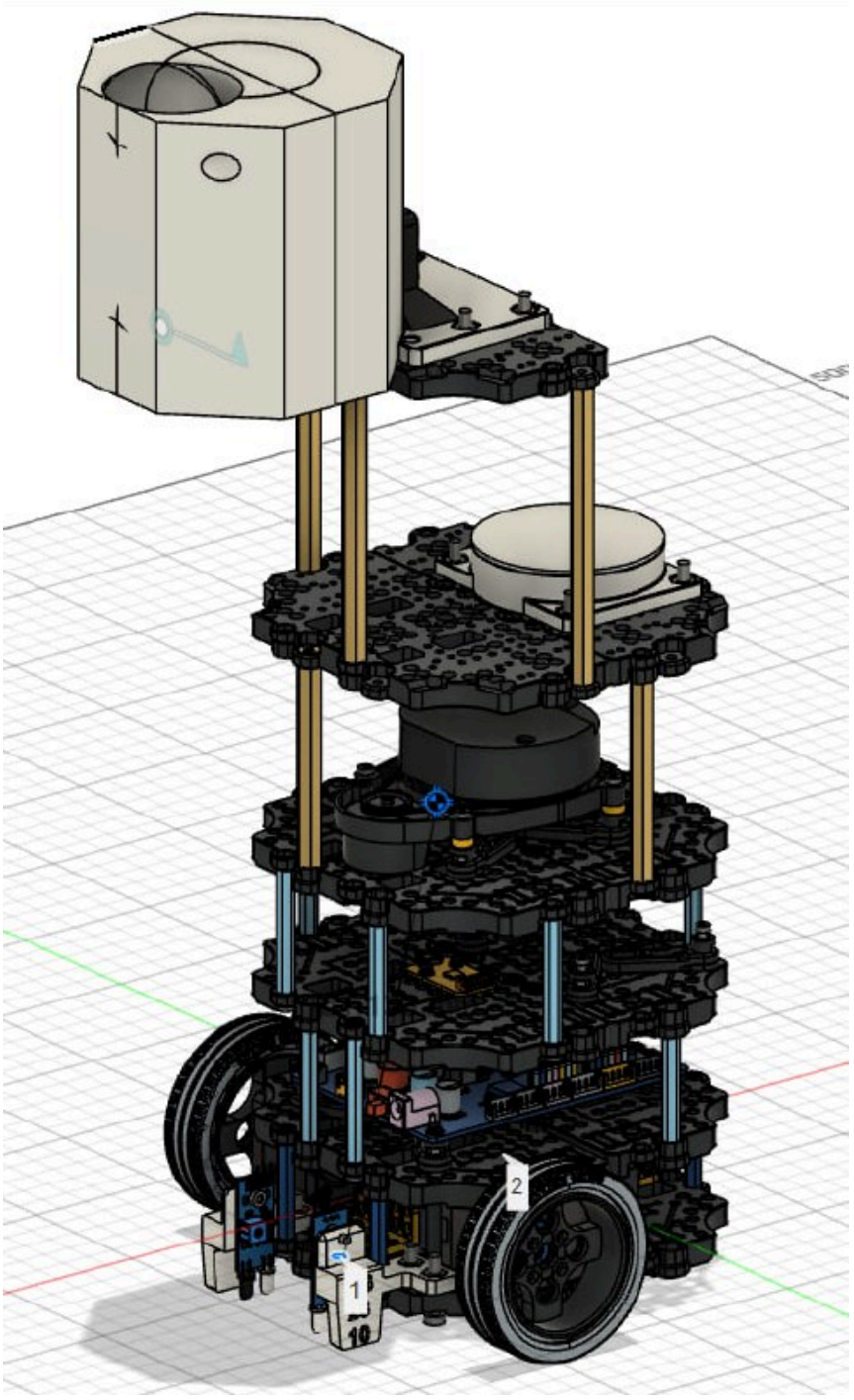
CAD of System

Old Design





Current Design



Github Repository

GitHub - JohnE1129839/r2auto_nav: ROS2 auto_nav code for EG2310