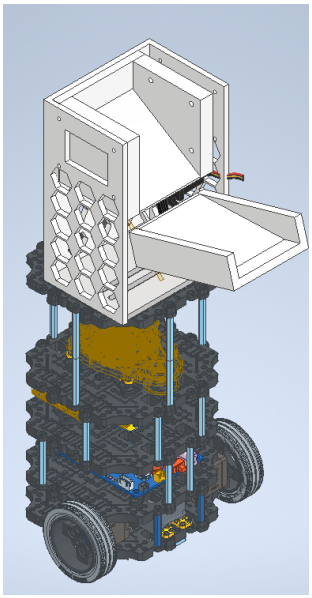
S/Ue Documentation

EG2310 Group 5



Team Members

Datta Anitej	A0284708U
Namit Deb	A0273659M
Sriram Anupama	A0276023L
Vishwanath Annanya	A0277110N
Wong Weng Hong	A0272156E

1.0 System Overview

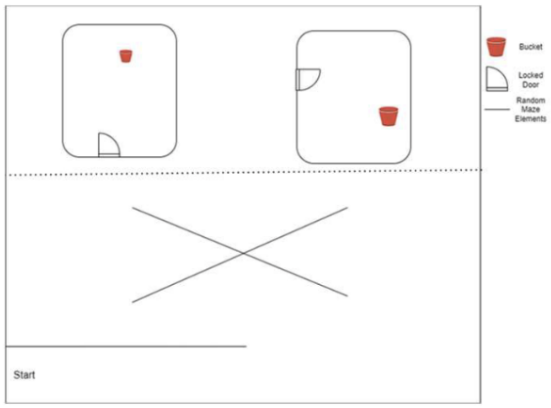
This report delves into the design, functionality, and troubleshooting methods relating to S/Ue, an autonomous robot that can navigate through and map a randomised maze using SLAM (Simultaneous Localisation and Mapping), communicate with and open a secured door, locate a bucket behind the door, and drop 5 ping-pong balls into said bucket.

2.0 Problem Definition

The entire mission duration is 20 minutes, including setup, documentation review, execution, and teardown.

During the setup phase, the team is to:

- Place temporary markers in the maze
- Place the robot at the start zone
- Set up the computer on the table near the start zone
- Start all the software
- Load ping-pong balls into the robot
- Any other activity, setup, calibration or tests required for completion of the mission



The mission execution will consist of 3 phases, a start zone, a randomized maze zone, and the lift lobby. The robot is to leave the start zone to enter the maze zone. In the maze zone, the robot is to map out the maze and find the exit of the maze. The robot will then enter the lift lobby.

At the lift lobby, there are 2 lift doors. The robot is to make an HTTP call to a web server. The server will then reply with a "Success" and the ID of the door that will open. The lift door will

remain open for 60 seconds. The server will not respond to HTTP calls during this 60 second. The robot can make another HTTP call after this 60 seconds to attempt to open a door. If a failure error is encountered, the robot can attempt to make another call until a "Success" is received.

The door is a simple hinge door without any automatic actuation. The robot must push the door to open the door.

Once the robot enters the lift, the robot is to find the target bucket and fire 5 ping pong balls into the bucket. The mission timing will end once the ping pong balls have been fired.

The teaching assistant (TA) may catch and power off the robot if necessary to prevent any damage or contact with the arena walls, the robot or any other elements of the maze. In this case, the current mission attempt will end and the team will receive a penalty.

The mission will end when one of two conditions is satisfied:

- The team agrees with the final score.
- The team disagrees with the final score or want to do a re-attempt to obtain a better score. If so, the team will have access to the arena to adjust the robot, markers and make modifications to the software.

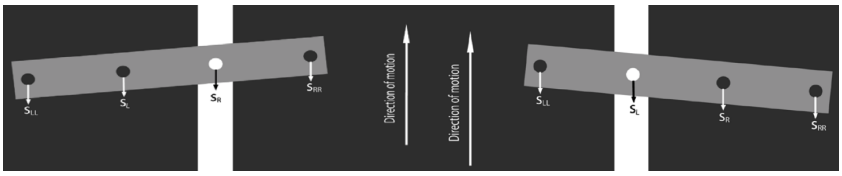
3.0 Literature Review

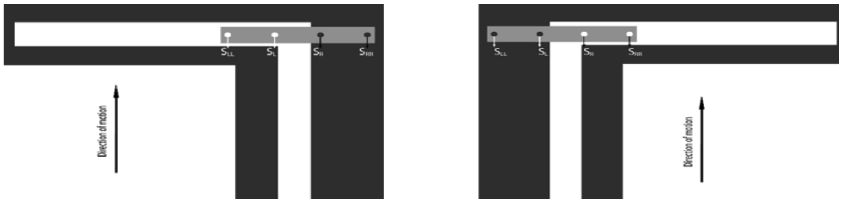
3.1 Navigation

To navigate the randomised maze the robot will need some maze solving algorithms. Options include:

Line Following

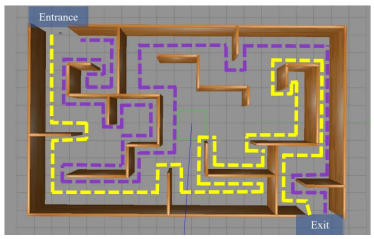
Team will use a version of the algorithm by Chowdhury, Khushi, and Rashid [1], and lay out a line for the robot to follow using coloured tape. With a system of 4 sensors, the robot will follow along the line correcting its path by adjusting the left/right motor speeds to steer. The outermost sensors are to help detect sharper 90º turns.





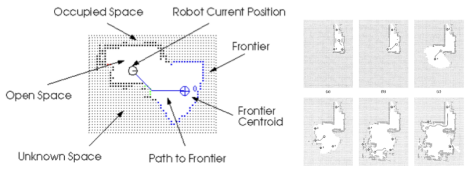
Wall Following

Robot chooses a direction (left/right) and follows the wall along that side. It will eventually completely explore a simply connected maze. This is a form of depth-first search in-order tree traversal [2].



Frontier Based Exploration

Here, the robot develops frontiers on the boundary between explored and unexplored space on the map. By moving to new frontiers, the robot can build the environment map until there are no more frontiers left [3].



3.2 Communication & Sensing

3.2.1 Python Requests Library

Python has a developed Requests library which the robot can use for interaction with the web server. This allows for sending of HTTP requests and receiving of JSON responses [4].

3.2.2 Laser Distance Sensor (LDS-02)

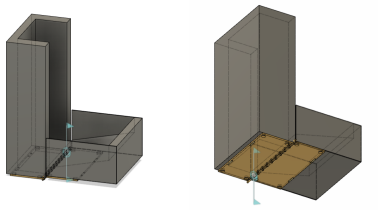
The robot uses a LDS-02 sensor to map its environment by identifying obstacles like walls. The LDS sensor has an angular range of 360° which is used for SLAM. This involves using a Python

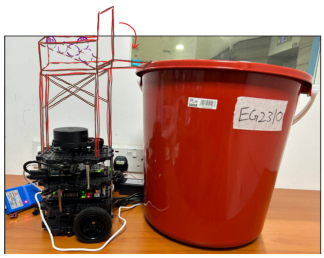
program to continually update the position of the robot on the current mapped area in its internal memory [5].

3.2.3 Door & Bucket Identification

NFC Tags NFC tags can be placed around the maze for the robot to receive information on specific locations relative to the map, for example its location relative to the elevator doors and instructions on how to proceed. The tags can contain data allowing the robot to differentiate doors [6].
Open Computer Vision (OpenCV) Library Applications of OpenCV such as Object Detection and Recognition can be used by using cameras for the algorithm to detect a marker or recognise the door and the bucket [7].
Lines (If Implementing Line Following) Lines made of tape can be placed in the map for the robot to directly follow, without needing any decision making [1]. Can be placed for specific tasks such as moving from door to bucket.

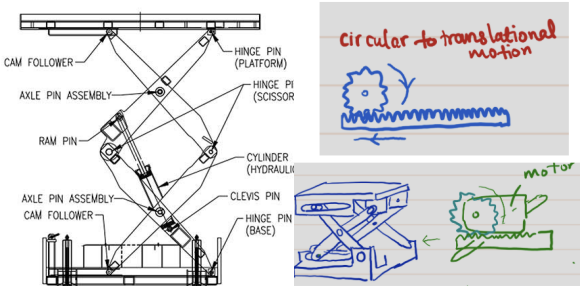
3.3 Payload

Projectile System TurtleBot navigates to a predefined distance from the bucket, and employs a high-speed rotating wheel in a chamber to launch ping pong balls.
Payload Mechanism The design features a box with a sloped floor, facilitating the rolling of the ping-pong balls into the bucket. To enhance functionality, one side of the box is designed as a convertible ramp. A butt hinge connects the ramp to the box, and a servo motor adds automation to the system, enabling the opening of the box up to a predefined angle. A thin gauze will be attached over the top of the box to prevent any balls from falling out of the box during the run. 
Tower System TurtleBot moves as close to the bucket as possible and rolls the ping pong balls into the bucket. The box containing the balls will be placed at a height, taller than the bucket, using stilts.



Scissor lift & Ramp System

An alternative to stilts in the tower system is employing a scissor lift mechanism, activated once the TurtleBot reaches the bucket [8]. This ensures a compact design for the TurtleBot throughout the run. Instead of a hydraulic actuator, as depicted in below, a heavy-duty servo motor will be utilised to initiate the scissor lift.



Positioned at the base's centre is a motor, paired with a spur gear and a rack, enabling the conversion of the motor's circular motion into translational motion. Extending from the sliding end of one scissor arm to the rack, a long cylinder is integrated. This configuration, activated by the motor, induces the movement of the sliding arms, raising the lift's height.

4.0 Concepts Design

The mission's three phases each have their own subsystem, of which the options are detailed below, along with their respective requirements.

4.1 The Maze

Proposed Subsystem	**Hardware**	**Software**
Line Following	Floor facing reflective sensors OpenCR	ROS2 OpenCR interfacing node PID controller for motor speed

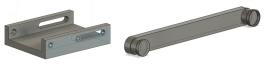
	Raspberry Pi 3 DynaMixel Motors (Wheels) Additional: Tape as temporary marker (line)	Line following algorithm RViz
Wall Following	LDS-02 OpenCR Raspberry Pi 3 DynaMixel Motors (Wheels)	ROS2 OpenCR & LDS-02 interfacing node PID controller for motor speed Wall following algorithm Data structure for backtracking & path memory RViz
Frontier Based Exploration	LDS-02 OpenCR Raspberry Pi 3 DynaMixel Motors (Wheels)	ROS2 OpenCR & LDS-02 interfacing node Frontier based navigation algorithm RViz

4.2 The Elevator Door & Bucket Identification

Proposed Subsystem	**Hardware**	**Software**
NFC Tags	NFC Tag NFC Sensor	Python nfcpy library A* Algorithm for pathfinding
OpenCR	Raspberry Pi Camera	Algorithm for recognizing and interpreting door features ROS2 camera interfacing node OpenCV library RViz
Line Following	See Section 4.1 for details	

4.3 The Bucket

Proposed Subsystem	**Hardware**	**Software**
Scissor Lift Mechanism	3D printed Base 3D printed scissor arms x 8 EzRobot Heavy Duty servo which can lift 15 kg/cm [9] Spur gear and rack	Program to initiate the servo motor and stop the motor once a the necessary height has been achieved

		
Tower Mechanism	3D printed stilts x 4 Triangular truss components	
Box & Ramp	3D printed box & ramp Butt Hinge Servo motor Thin gauze	Program to initiate the servo motor to 'open' the box until a predefined angle.

For 3D printing, all components will be crafted using Polylactic Acid (PLA) filaments, with an estimated tensile strength of 40MPa-50MPa [10]. To enhance the overall strength of the printed parts, we are considering the implementation of a honeycomb or triangle infill pattern.

5.0 Bunch of Guys Around Table (BOGAT)

The final solution is an integrated system which must balance mechanical, electrical, and software requirements. The options presented in Section 2.0 have been evaluated below, explaining the rationale behind our final decision.

5.1 Navigation

	Line Follower	**Wall Follower**	**Frontier Based**
Pros	Turtlebot does not need to make any decisions	Straightforward and guarantees full exploration	Have little setup as the algorithm needs little markers
Cons	Requires additional instructions to determine which door to move to Takes time to set up and remove the tape, might exceed time limit	Inefficient as every path is explored including those already mapped Relies on simply connected maze Needs separate algorithm to locate the	Robot may not be able to handle unforeseen scenarios or edge cases Needs a separate algorithm to locate the elevator doors

		elevator doors	
Decision	The team has decided to follow a Frontier Based navigation algorithm as it can efficiently map the entire maze portion without repeating previously explored segments, while also minimising setup time during the 20 minute mission execution window. A separate set of instructions are required once the robot exits the maze, however. This is the phase where the temporary markers will be crucial for assistance.		

5.2 Temporary Markers

	NFC Tags	**OpenCV**	**Lines (Line following)**
Pros	Cheap Can be embedded with data to help differentiate doors	Able to detect and recognise from afar	Cheap
Cons	Short range, requires accuracy of movement	Potentially unreliable and expensive	Takes time to setup
Decision	The team has decided to use line following to determine the robot's movement directly from the starting point towards the door, and again through the doors directly towards the bucket. This removes the need for error-prone decision making. Although it may take time to setup, this should be minimsed with practice and efficient planning.		

5.3 Payload

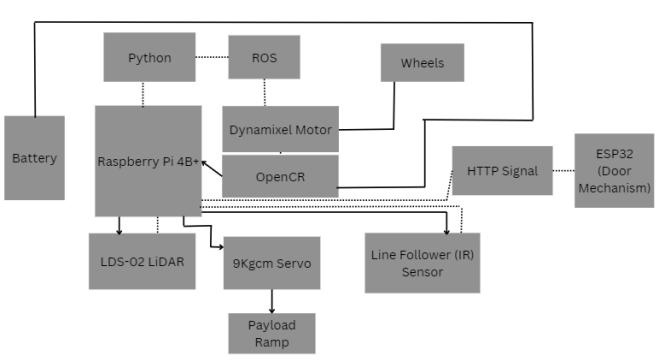
	Scissor Lift	**Tower**
Pros	More compact Will not disrupt the Lidar sensor	Simpler execution and construction
Cons	Complex design, requiring a separate platform Potential need for counterweights Decision-making between a motor or hydraulic actuator for automation.	Disrupt the Lidar sensor Unstable due to height Affects the balance of the turtlebot

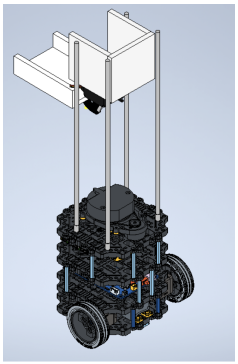
Decision	Since there is no height limit throughout the run, the team has decided to go forward with the tower design due to the ease of executing this design, and minimal disruption to the Lidar sensor.

6.0 Preliminary Design

6.1 Preliminary Design Overview

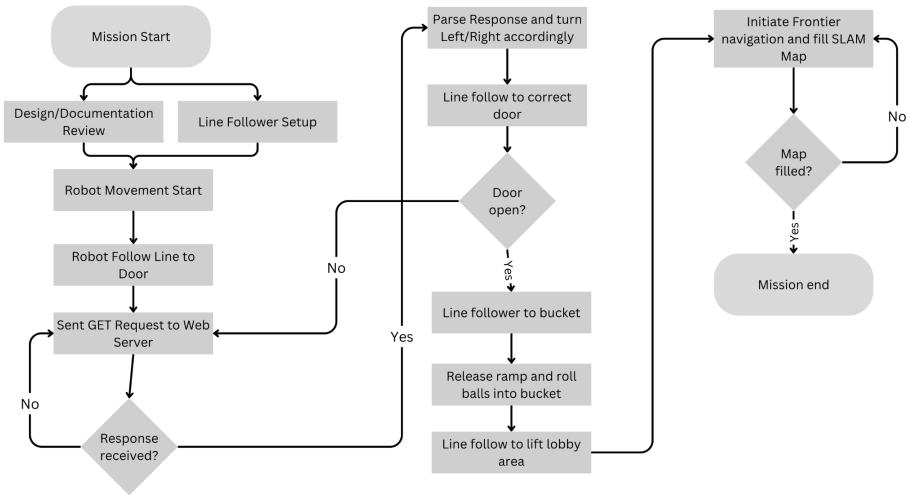
The proposed solution uses a TurtleBot3 Burger, integrated with ROS2 Foxy Fitzroy, OpenCR, and a Raspberry Pi 3. The robot is equipped with an LDS-02 for SLAM capabilities, visualised through RViz. For the maze, the team will use a Frontier Based maze solving algorithm. For the bucket section, the robot will be equipped with 4 stilts to elevate the balls and then roll them into the bucket. For the temporary markers, the team will use lines to guide the robot through the maze directly towards the doors and bucket.





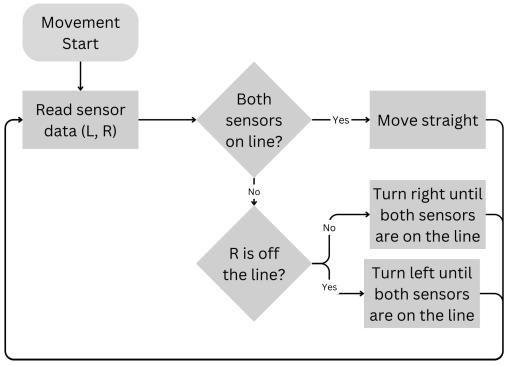
6.2 Mission Flow of Events

Based on the above design, the flow of the mission is detailed below:

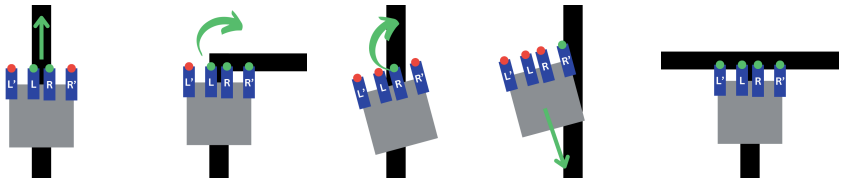


6.3 Line Follower Subsystem

Commented [1]: the final implementation and this different its ok right



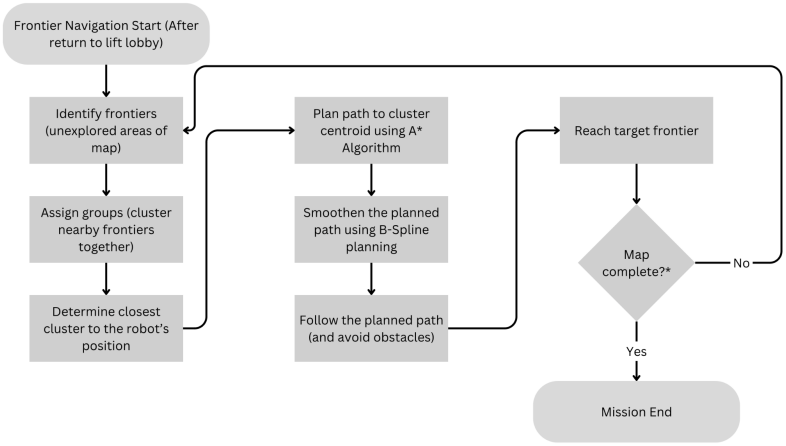
The robot will use an array of 4 IR sensors in order to control its movement. The above flowchart, highlights how the internal line sensors (denoted L and R) are utilised for general movement. The outer sensors (denoted L' and R') are used to handle edge cases, such as turning, stopping, and marking checkpoints. Highlighted below are how the sensors will work in conjunction to efficiently move S/Ue.



As seen above, different combinations of the IR sensors reading ‘true’ will publish different instructions to the RPi, causing S/Ue to move straight, make slight turns, or make full 90º turns. As seen in the fourth case, having either only L’ or R’ read true will indicate that the robot is fully off course, and thus must reverse until the normal conditions of only L and R being true are met.

Finally, having all 4 sensors read true simultaneously instructs S/Ue to halt completely. This case will be used for decision-making steps, such as when S/Ue needs to send the HTTP request and determine which door to choose (i.e., turn left or turn right), when S/Ue reaches the bucket and needs to actuate the payload, and when S/Ue transitions from using the line-following algorithm to using frontier-based navigation. A counter will be implemented to help indicate which specific ‘checkpoint’ S/Ue has reached for these cases.

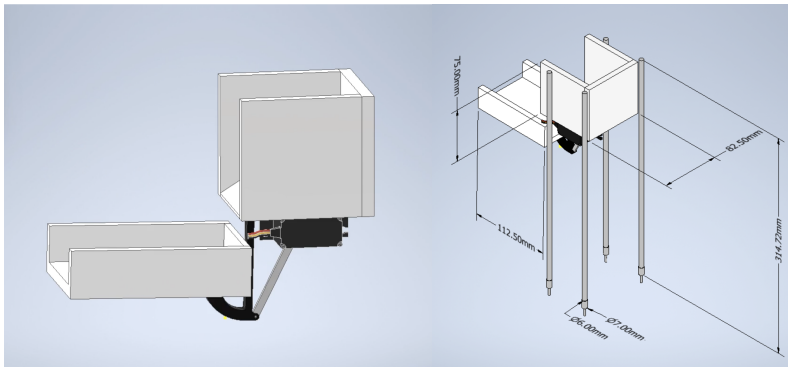
6.4 Autonomous Navigation

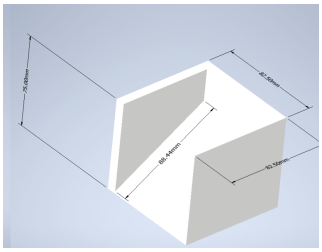
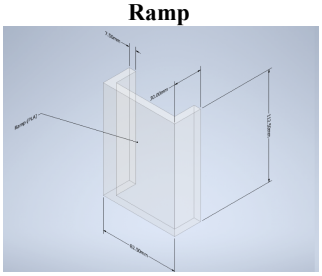


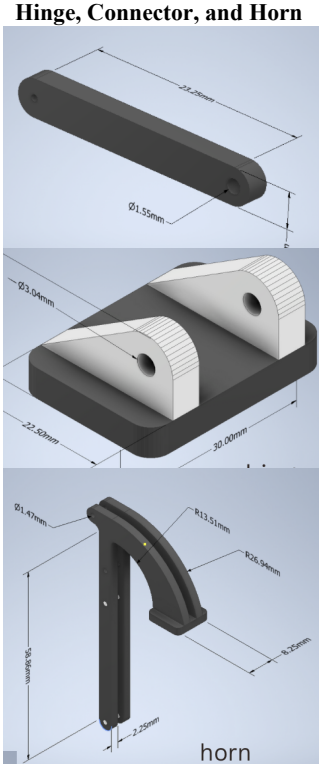
The above flow chart describes how S/Ue will use frontier-based navigation to map the maze. This phase will be executed after releasing the ping-pong balls into the bucket and returning to the lift-lobby.

6.5 Payload Design

The payload will be a box & ramp on stilts, placed above S/Ue. Upon reaching the line checkpoint (97mm away from the bucket) indicating to actuate the payload, the RPi will trigger a servo motor, causing the ramp to lower similar to a drawbridge. From here, the balls will roll down into the bucket, and then the ramp will close once again.



Component	Fabrication Method	Justification
Box 	Laser Cut & welded together with Tensol12 adhesive Made of acrylic Thickness < 5cm	Fast & cheap solution
Ramp 	3D printed with 20% infill PLA Triangular infill pattern	Simple assembly and light

Hinge, Connector, and Horn 	3D printed with 50% infill PLA Triangular infill pattern	Simple assembly Specialised design requires rapid prototyping/iteration in case of improvements required

6.5.1 Choice of Servo for Payload

Torque required to fip the ramp = F x r= Fr sin theta

since ramp is perpendicular to base, theta = 90 degrees; Torque = Fr

Weight of ramp = 0.36 kg

length of ramp = 5cm

Torque = 0.36kg x 5cm = 1.8 kgcm

Assuming safety factor of 4.5 (to calculate stall torque):

Torque(max) = 1.8 x 4.5 = 8.1 kgcm

Hence Servo chosen (MG995) has a stall torque of 9 kgcm

6.6 Electrical Components & Power Budgeting

Component	Power (W)	Time (s)	Energy (Wh)	Voltage (V)	Current (mA)
TurtleBot (Startup)	4.120	180	0.2060	11.1	371.0
TurtleBot (Standby)	6.512	420	0.7600	11.1	586.7
TurtleBot (Motion)	8.634	1500	3.5970	11.1	777.84
Payload Servo Motor	0.125	10	0.0003	4.8	27.8

6.6.1 Raspberry-Pi 4B+

The Raspberry-Pi will subscribe to the data published by the LIDAR, handle wireless communication with the ESP32 to open the locked doors, and also handle autonomous navigation to map the entire maze using Frontier based navigation.

6.6.2 OpenCR

The OpenCR board on the TurtleBot3 plays a crucial role in the robot's control and communication systems, enabling it to perform tasks such as navigation, obstacle avoidance, and mapping. It is the main microcontroller present on board the Turtlebot3.

6.6.3 LIDAR (LDS-02)

The role of LIDAR on the TurtleBot3 is to provide accurate distance measurements to help in mapping the surrounding environment and assist in obstacle detection and avoidance, enabling the robot to navigate through complex environments.

6.6.4 Li-Po (Lithium Polymer Battery)

The Li-Po battery that has a voltage of 11.1V is the main power source to all the electrical components i.e. Raspberry Pi, OpenCR, LIDAR, and the dynamixel motors on the Turtlebot3.

6.6.5 Dynamixel Motors

The Turtlebot3 consists of two dynamixel motors, one for each wheel, as the main actuators for its motion. The motors are used to control the speed and direction of the bot.

6.6.6 9kgcm Servo Motor (MG995)

The Servo Motor is part of the payload; it is attached to a hinge which connects to the ramp. When S/Ue reaches the bucket, the servo lowers the ramp, and the balls fall into the bucket.

6.6.7 Line Follower Sensors (based on Vishay TCRT5000)

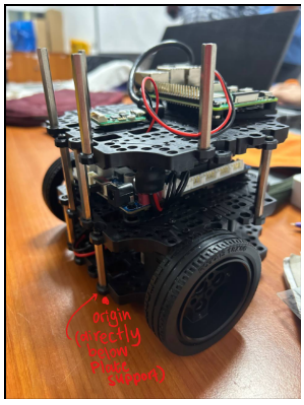
S/Ue consists of 4 of these line following sensors, which allow it to follow a predesignated route throughout the maze, and reach the bucket.

7.0 Prototyping & Testing

7.1 Payload

We ran into two main problems with the prototype:
1. The support for the payload wasn't strong enough to handle its high centre of gravity.
2. When the payload opened, there were two big gaps on the sides of the ramp where the balls could roll out.

A. Payload Support Design



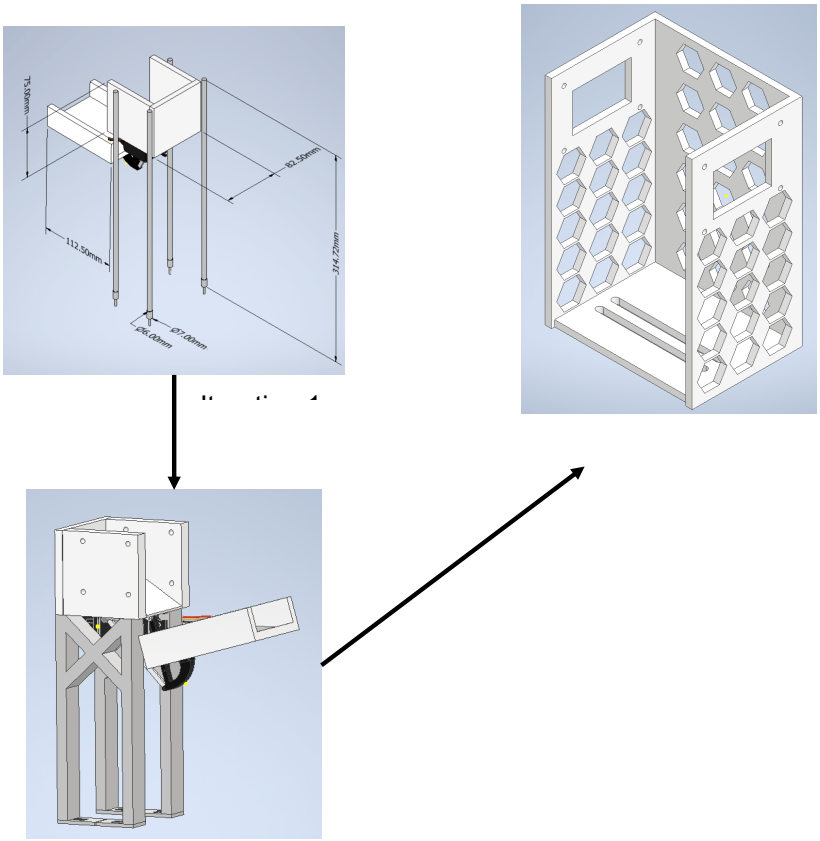
After some discussion, we recognized the need to change the payload support system due to the high centre of gravity of the overall Turtlebot system, leading to instability. By positioning the origin directly below the plate support, we determined the centre of gravity to be (-62.453mm,39.81mm, 172mm).

To address this issue, we evaluated these main factors:

- Weight distribution
- Structural integrity
- Integration with the Turtlebot

The initial design relied on four thin 'chopstick'-like supports to bear the weight of the payload. However, given the Turtlebot's height, this was quite impractical. We also considered a truss-like structure(iteration 2). While this seemed promising, the attachment of the support to the waffle plate would not have been sturdy enough to dampen vibrations. Realising that stability was the

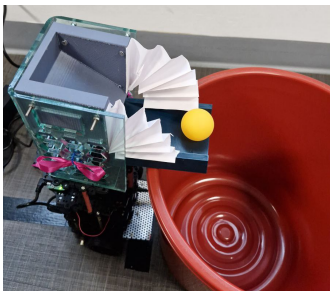
primary concern, we prioritised this over potential weight increases from the support structure. This led us to choose the idea of 'wrapping' the payload from three sides, effectively distributing the weight more evenly.



Iteration 2

B. Payload gaps

In 2 out of 10 attempts to drop the balls into the bucket, the balls would fall off from the sides. To reduce this, we decided to add two pieces of folded paper to the sides of the payload, as shown in the picture below. This paper will close and open as and when the ramp moves.



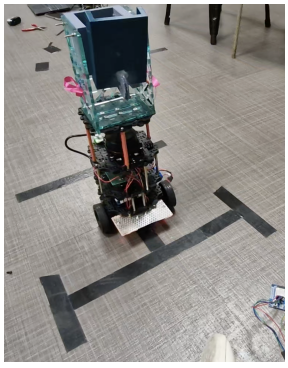
Parts for Payload			
Part	Quantity	Unit Cost ($)	Total Cost($)
sloped Box	1	Free Access to a 3D printer	
Ramp	1		
Hinge	1		
Horn	1		
Connector	1		
Servo Support	1		
U-shaped support	1	15	
Support base	2		
M4 x 20 screw	6	Provided by the iDP electronics Lab (Ms Annie is a legend)	
M4 Hex nut	6		
M4 Lock nut	6		
M3 tapping screw	2		

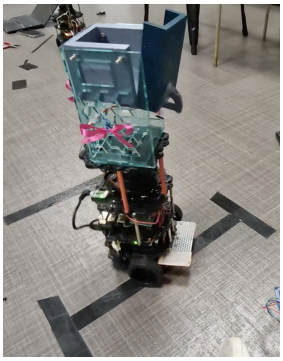
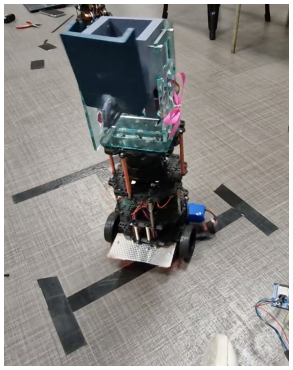
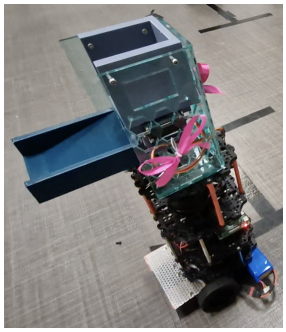
M2 x 25 screw	1	
M2 x 10 screw	1	
M2 Hex Nut	3	
M2 Lock Nut	1	
M2 Tapping screw	3	
Epoxy Glue	As needed	

7.2 Line Follower Subsystem

The prototype algorithm closely adhered to the processes outlined in Section 6.3, with the exception of handling left and right turns. Testing revealed a challenge: due to the non-alignment of all four line sensors and the difficulty in centering S/Ue on the tape, triggering all four line sensors simultaneously to activate a checkpoint was inconsistent. Typically, one of the outer line sensors would be triggered first, causing the bot to detect a turn instead of a checkpoint. Consequently, the group decided that it was not feasible to consistently align S/Ue to trigger all four sensors at once, and to scrap the idea of detecting turns with three of the four line sensors.

On a positive note, the other processes from the preliminary design worked quite successfully during our testing.

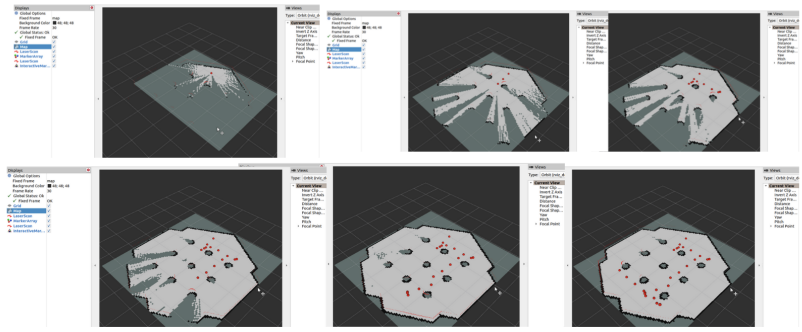
	Testing Checkpoint detection

	First checkpoint can be detected Received “door1” from ESP32, S/Ue turned left
	First checkpoint can be detected Received “door2” from ESP32, S/Ue turned right
	Second checkpoint can be detected Released Payload

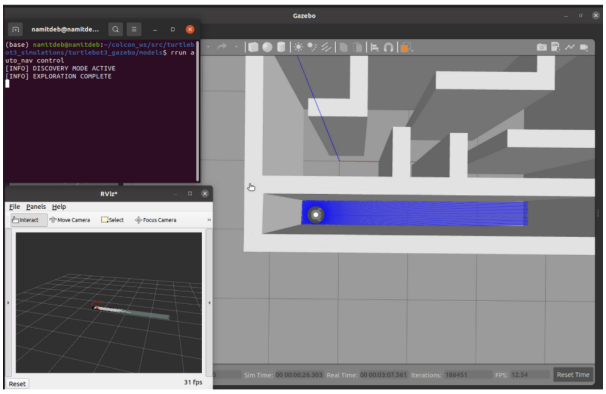
	Tested payload with bucket

7.3 Autonomous Navigation

The prototype algorithm implemented followed the processes defined in Section 6.4, with testing done in a simulated Gazebo environment. In relatively open environments, S/Ue was able to navigate freely and efficiently, with little bugs identified. The below image shows how S/Ue build the map of the simulated environment, with red markers indicating the path that it took, exploring frontier-by-frontier.



However, performance significantly dropped when placed in closed-off environments. This was due certain parameters in the original code not being fine-tuned for narrower hallway-like spaces such as a maze. One such parameter, `expansion_size`, created a ‘padding’ around obstacles causing the robot to avoid them. It was likely that this padding was too large for the narrower environments, causing bugs in the robot’s operations.



In order to better simulate the mission conditions, testing of the frontier-based algorithm was then brought into the real-world. A physical sample maze was created for S/Ue to test its algorithm in. After fine-tuning the parameters, S/Ue was able to autonomously navigate the area, however unlike in the Gazebo environment, in physical conditions it was not reliably avoiding walls, and would only sometimes stop and turn when facing an obstacle. Furthermore, the robot tended to get stuck on sharp corners, an edge case which the algorithm had not considered.

As seen above, the robot would fail to identify sharp corners and go directly into them. Then, when attempting to pivot, the inner face of the wheels would get stuck, preventing it from escaping this situation. To overcome this issue, tweaking parameters such as the published velocities to allow the robot to rotate at variable speeds proved to work occasionally, however this was not a reliable solution. The team was not able to implement a robust solution to this problem within the time given.

8.0 Critical Design

8.1 System Finances

Bill of Materials:

Component	Quantity	Cost/unit(S$)	Total Cost (unit cost and shipping charges)
IR Line Following Sensors TRCT5000	8 (Used 4)	1.99	22.26
Servo Motor (MG995)	1	0	0 (borrowed from Electronics Lab)
Laser Cut Payload Support	1	15	15
3D printed payload components: ● Sloped box ● Ramp ● Horn ● Hinge ● Connector	5	0	0 (Access to a 3D printer)

8.2 Line Follower Subsystem

Documentation for the final algorithm is detailed below, with the full script available in `lineFollower.py` on the robot's Github repository. The algorithm constantly checks the GPIO inputs of the IR sensor pins to determine where S/Ue is supposed to go.

`lineFollower.py`	
Constants	
Name	**Description**
`OPEN_ANGLE`	The angle of which the servo is opened to

CLOSE_ANGLE	The angle of which the servo is closed to
PAYLOAD_PIN	GPIO output pin for servo
LL_PIN	GPIO input pin for outer left line sensor
L_PIN	GPIO input pin for inner left line sensor
R_PIN	GPIO input pin for inner right line sensor
RR_PIN	GPIO input pin for outer right line sensor
rotatechange	Rotational speed of the robot
speedchange	Linear speed of the robot

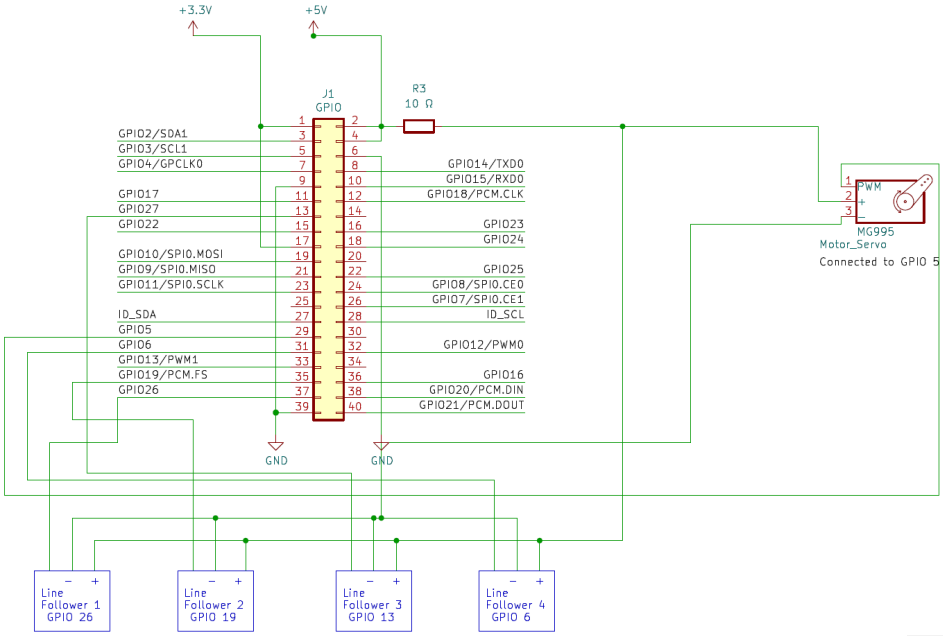
Functions

Name	Arguments	Description
payload	-	Open and closes payload based
door	-	Triggers ESP32 and returns “door1” or “door2”
GPIO_setup	-	GPIO setup for line sensors

Class: NavigationControl **Functions**

Name	Arguments	Description
init	self	Initializes the node, publishers, subscribers, and other necessary variables.
kill_line_callback	self, msg	Callback method to trigger autonomous navigation.
publish	self	Publishes twist to change velocity of robot
fullTurn	self	Method to make robot turn right 180°
turnRight	self	Method to make robot turn right 90°
turnLeft	self	Method to make robot turn left 90°
moveStraight	self	Method to make robot move straight
reverse	self	Method to make robot reverse
nudgeLeft	self	Method to make robot nudgeLeft
nudgeRight	self	Method to make robot nudgeRight
checkPoint	self	Method to handle checkpoints
stopbot	self	Method to stop bot from moving
mover	self	Main logic of program, determines which other method to call depending on GPIO input of line sensors

Electrical Schematic of the Line Follower and Payload Servo Motor subsystem



8.3 Autonomous Navigation

Documentation for the final algorithm is stated below, with the full script available in `control.py` on the robot's Github repository. The algorithm implements a BFS, frontier-based navigation as detailed during the Preliminary Design and Prototyping sections.

`control.py`	
Constants	
Name	**Description**

lookahead_distance	Degree to which robot obeys planned path
speed	Maximum velocity of the robot
expansion_size	'Padding' applied to obstacles for the robot to avoid being within
target_error	Margin of error for robot to reach each frontier
robot_security_radius	'Padding' around robot to improve reflexes and better avoid obstacles at close-range

Functions

Name	Arguments	Description
euler_from_quaternion	x, y, z, w	Convert quaternion to Euler angles.
heuristic	a, b	Calculate the heuristic distance between two points.
astar	array, start, goal	Implement the A* algorithm for path planning.
bspline_planning	array, sn	Generate a smooth path using B-spline interpolation.
pure_pursuit	current_x, current_y, current_heading, path, index	Implement the pure pursuit algorithm for path following.
frontier_B	matrix	Identify frontier cells in an occupancy grid.
assign_groups	matrix	Assign frontier cells to groups.
dfs	matrix, i, j, group, groups	Depth-first search algorithm for grouping frontier cells.
f_groups	groups	Filter and sort groups based on size.
calculate_centroid	x_coords, y_coords	Calculate the centroid of a group of points.
find_closest_group	matrix, groups, current, resolutions, originX, originY	Find the closest group of frontier cells to the current position.
path_length	path	Calculate the length of a path.
costmap	data, width, height, resolution	Generate a costmap from sensor data.
exploration	data, width, height, resolution, column, row, originX, originY	Implement the exploration strategy.
localControl	scan	Implement the local control strategy.

Class: NavigationControl **Functions**

Name	Arguments	Description
init	self	Initializes the node, publishers, subscribers, and other necessary variables.

`exp`	`self`	The main control loop, which is called periodically to implement the navigation strategy.
`target_callback`	`self`	Repeatedly calls the exploration function.
`checkpoint_callback`	`self, msg`	Callback function for line follower checkpoint counter
`scan_callback`	`self, msg`	Callback function for scan data.
`map_callback`	`self, msg`	Callback function for map data.
`odom_callback`	`self, msg`	Callback function for odometry data.

Commented [2]: method?

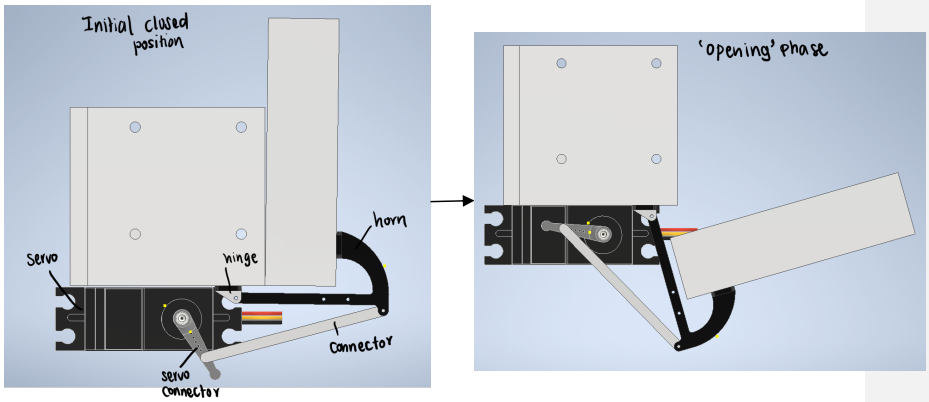
Note the `checkpoint_callback` function in the algorithm. This callback function subscribes to a boolean value, awaiting for the checkpoint to return 'True'. This essentially signals that the line follower script is complete, allowing S/Ue to transition into frontier-based navigation.

8.4 Payload Design

Our final payload design consists of two parts: the payload , payload support.

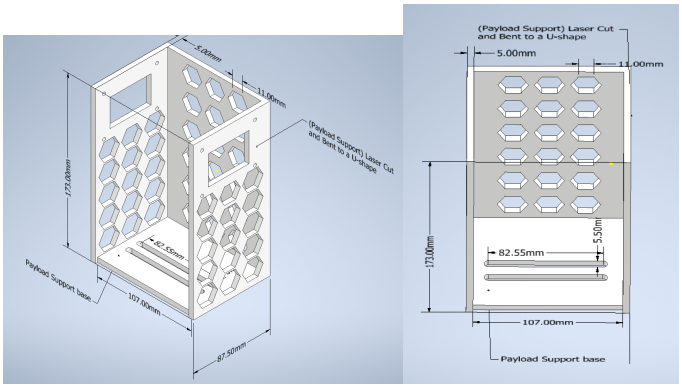
A. Payload

Opening and Closing mechanism of the payload:



This setup relies on the servo motor to release the ping pong balls from the sloped box. The servo motor's rotation moves a connector arm, which then shifts the horn. As the horn moves, it opens the ramp (which then moves 90 degrees clockwise) that keeps the balls inside the box. The slope of the box and gravity work together to ensure the balls roll out smoothly once the ramp is open. Personally, I found this mechanism to be really cool and so I had to try making it.

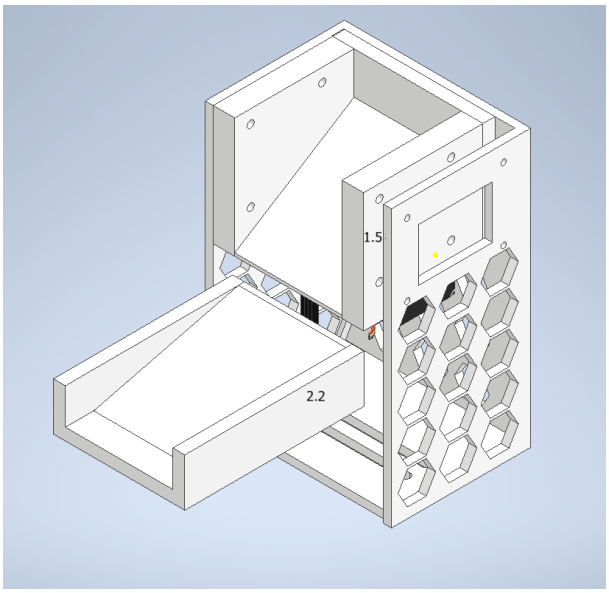
B. Payload Support



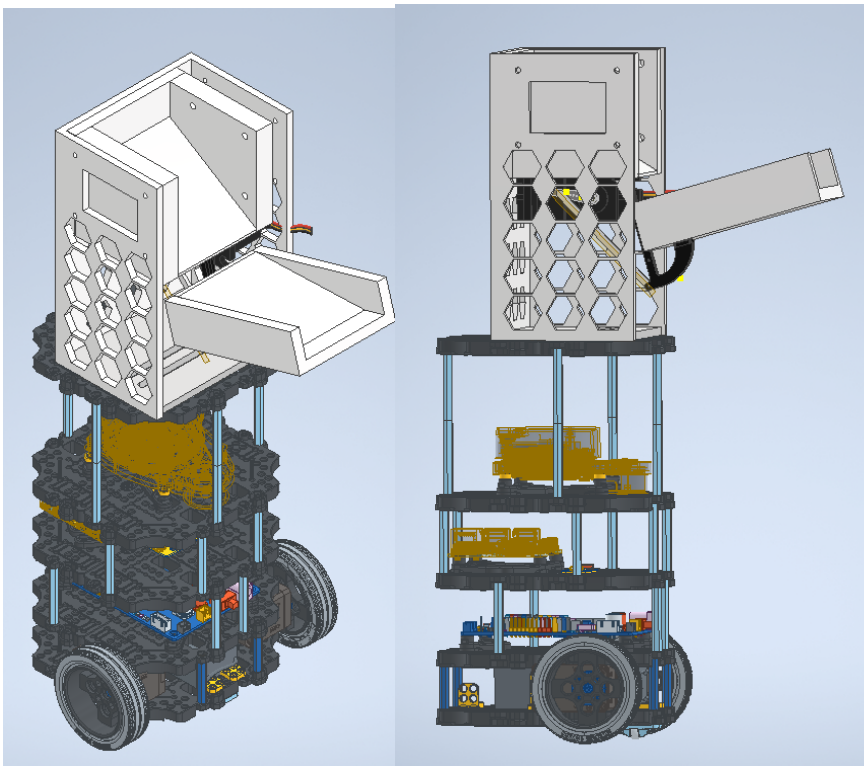
The final design we settled on prioritised sturdiness, which was crucial for the system's reliability. By surrounding the payload with support directly anchored to the waffle plate, we addressed concerns about vibration, particularly stemming from the high centre of gravity. This setup ensures that the weight of the payload is evenly distributed, reducing stress concentrations that could lead to structural issues.

- What is the reason for the hexagonal design?

Since I had access to a laser cutting machine, I thought why not and make it look cool so I just played around with some designs. My main intention was to reduce the weight of the support structure. This design also proved useful in wrapping the servo wire through it.



CAD of the Final Payload Design



Final CAD design of the turtlebot

8.5 Electrical Components & Power Budgeting

8.5.1 Power Consumption of Individual Components based on Mission Time

Component	Power (W)	Time (s)	Energy (Wh)	Voltage (V)	Current (mA)
TurtleBot (Startup)	4.120	180	0.2060	11.1	371.0
TurtleBot (Standby)	6.512	420	0.7600	11.1	586.7
TurtleBot (Motion)	8.634	1500	3.5970	11.1	777.84
Payload Servo Motor	0.133	10	0.0004	4.8	27.8
Linefollower Sensors (4x)	0.384	600	0.064	4.8	80 (20x4) mA

8.5.2 Maximum Safe Current of Battery

Capacity of battery= 1800mAh

C Rating of Battery= 5C

Maximum safe discharge current of battery= 1800 mAh % 1/5 C

= 9000 mA= 9A

Maximum total current: Maximum turtlebot current + servo current

= 777.84 mA + 27.8 mA + 80mA

= 885.64 mA < 9A

8.5.3 Energy Consumption

Total Energy = 0.2060 + 0.7600 + 3.5970 + 0.0004 + 0.064 Wh

= 4.6310 Wh

Energy provided by the Battery = 11.1 V x 1800 mAh

= 19.98 Wh

Assuming safety factor of 60%:

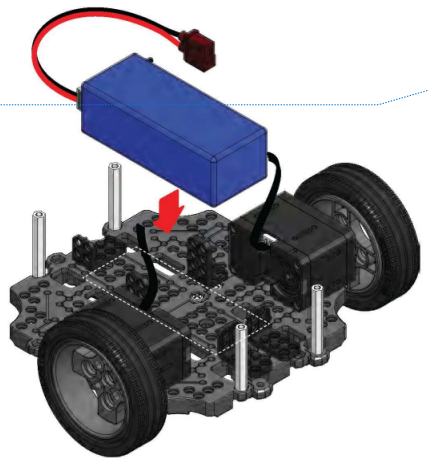
Total Energy consumed = 1.6 x 4.6310 Wh

=7.4096 Wh < 19.98 Wh

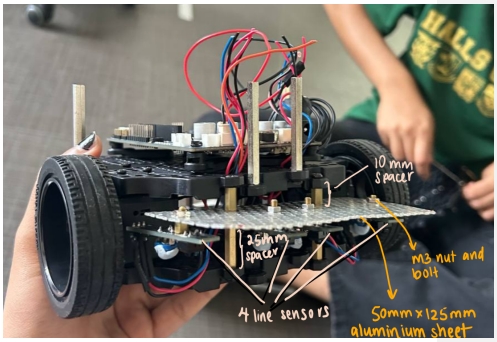
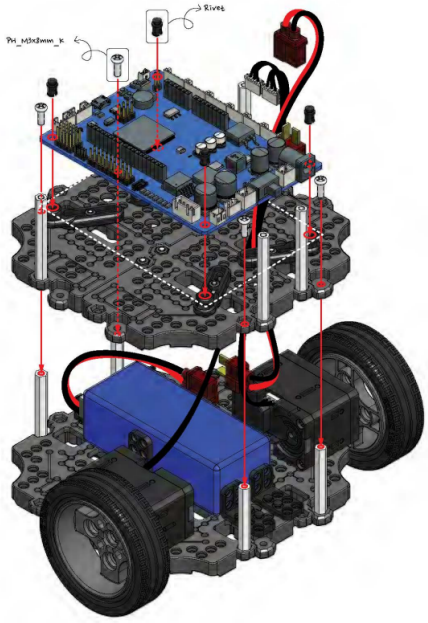
Total time the robot can run on a full charge: 19.98/7.4096 ~= 2.7 hours -> 2hr 40min

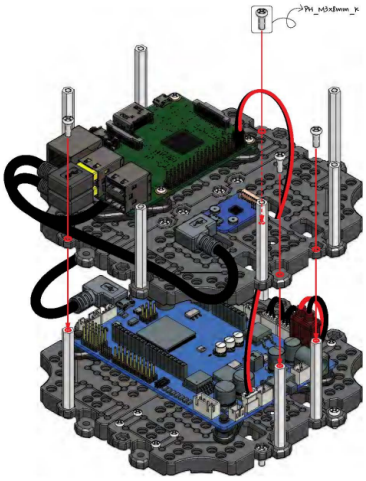
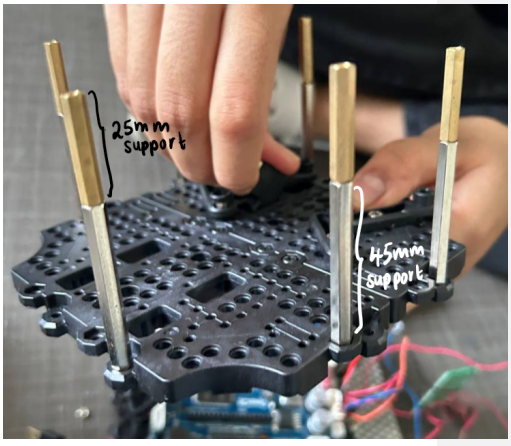
9.0 Assembly Instructions

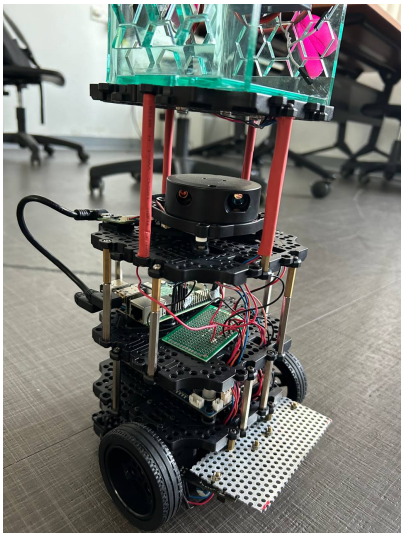
9.1 Mechanical Assembly of Turtlebot3 with Modifications and the Line follower setup

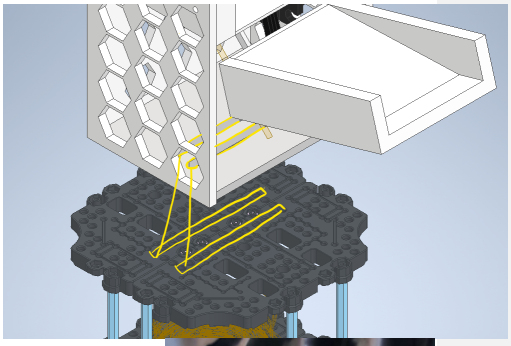
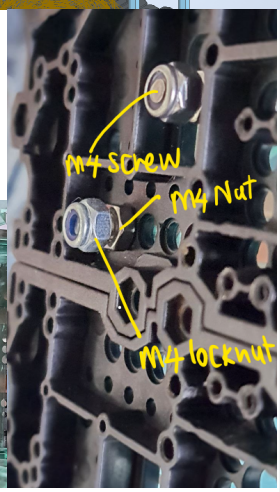
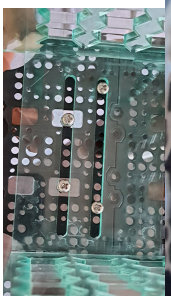
Steps	Description	Images
1	For assembly of base Turtlebot3 Burger, refer to the Turtlebot3 Burger Assembly Manual that can be found here.	

Commented [3]: should we embed images from the manual for this in the image column? or leave as it is

2	Attach 4 line sensors using M3 nuts and bolts, to an aluminium sheet 50 mm x 125 mm A 10mm spacer and a 25mm spacer is used to join the aluminium sheet between the first and second waffle plates	
3	For Assembly of the second and third waffle plate, refer to the Assembly manual found here	

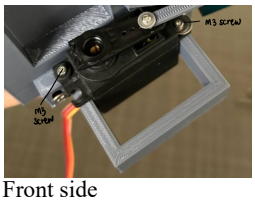
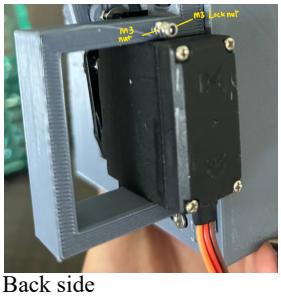
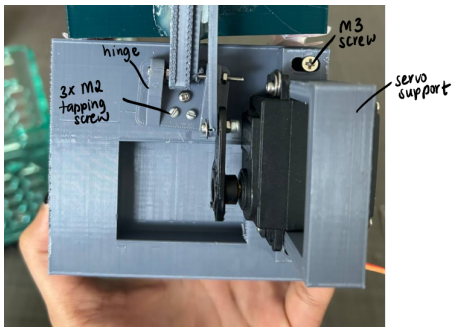
		
4	Between the 3rd and 4th assembly, we added an additional 25mm (male-female) spacers.	

5	Between the 4th and 5th waffle plates we added 3x30mm (male-female) spacers and shrink wrapped them. Instead of placing the lidar chip at the 3rd waffle plate, place it on the 4th waffle plate right beside the Lidar. This is because the wire is not long enough to reach the 3rd waffle plate below.	
6	We attached the fifth waffle plate to the 4 supports using 4 M3 nuts.	

7	Attach the base of the payload support to the the 5th waffle plate using 4 M4	  

9.2 Mechanical Assembly of the Payload

Step	**Description**	**Picture**
1	3D print the following components: ● Ramp ● Slope ● Servo connector ● Horn	

	• Hinge	
2	Laser cut the following components: • Payload Support • Support base Connect the base to the support using Arcylic glue.	
3	• Connect the servo to the servo support using M3 screws and nuts	 Front side  Back side
4	• Connect the hinge to the slope with 3 M2 tapping screws • Connect the servo support to the slope with 2 M3 tapping screws	

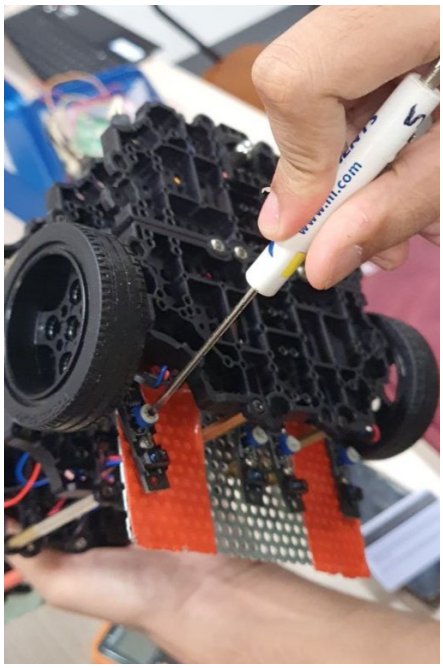
5	Next, is to attach the horn to the hinge. ● Use an M2.30 screw as an axis of rotation for the connector ● Secure the connector in place using some M2 nuts and a lock nut	
6	The connector is attached to the horn using an M2.25 screw and is secured in place using a hex nut.	
7	Finally, the ramp is attached to the hinge using epoxy glue. This has to be left in the same position to dry for 24hrs	
8	Once the glue has dried, drill a M2 sized hole into the servo arm and using: ● M2 screw ● M2 Washer ● M2 lock nut, connect the connector to the servo arm.	

9	Using 4 M3 x 20 screws, attach the payload to the support	

9.3 Line Follower Sensor Calibration

For the robot to work properly, the line sensors must be calibrated according to the surface being used on. This is because different surfaces have different color gradients and light absorption/reflection properties. It should be noted however that the line and the surface should be of contrasting colours such as black/white for the best performance.

The line following sensors can be calibrated by manually adjusting their potentiometers, using a screwdriver.



Once the red light emanating from the sensor turns off when the IR receiver is near the desired surface of line following, the line following sensor is ready to be used on that surface.

Next, adjust the other 3 line sensors accordingly. It is a good practice to follow up sensor calibration with actual testing by running the `line` program as mentioned in section 9.4.8 to ensure that the calibration is a success.

9.4 Software Setup

9.4.1 Software Dependencies

- Ubuntu 20.04

9.4.2 Installation of ROS2

Install ROS2 Foxy on Remote PC

```
wget https://raw.githubusercontent.com/ROBOTIS-
GIT/robotis_tools/master/install_ros2_foxy.sh
sudo chmod 755 ./install_ros2_foxy.sh
bash ./install_ros2_foxy.sh
```

9.4.4 Installation of Dependent ROS2 packages

Install Gazebo

```
sudo apt-get install ros-foxy-gazebo-*
```

Install Cartographer

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

Install Navigation2

```
sudo apt install ros-foxy-navigation2
sudo apt install ros-foxy-nav2-bringup
```

Install Slam Toolbox

```
sudo apt install ros-foxy-slam-toolbox
```

Install TurtleBot3 Packages

```
sudo apt install ros-foxy-turtlebot3
```

9.4.5 Configure Environment for ROS2 Development

```
wget https://raw.githubusercontent.com/ROBOTIS-
GIT/robotis_tools/master/install_ros2_foxy.sh
sudo chmod 755 ./install_ros2_foxy.sh
bash ./install_ros2_foxy.sh
sudo apt-get install ros-foxy-gazebo-*
sudo apt install ros-foxy-cartographer
sudo apt install ros-foxy-cartographer-ros
sudo apt install ros-foxy-slam-toolbox
source ~/.bashrc
sudo apt install ros-foxy-dynamixel-sdk
sudo apt install ros-foxy-turtlebot3-msgs
sudo apt install ros-foxy-turtlebot3
echo 'export ROS_DOMAIN_ID=30 #TURTLEBOT3' >> ~/.bashrc
echo 'export TURTLEBOT3_MODEL=burger' >> ~/.bashrc
source ~/.bashrc
```

9.4.6 Create Aliases

For ease of use, we have created the following aliases which we have added to the `~/.bashrc` of our Remote PC and RPi4. We have listed them below.

Note: For our mission we had setup an AWS server which allows us to fetch the RPi4's IP-address whenever `sshrp` was called.

Remote PC Alias	Command
sshrp	ssh ubuntu@<RPi4 IP-address>
toolbox	ros2 launch slam_toolbox online_async_launch.py
rvm	cd ~/.rviz2 && rviz --display-config mission.rviz
ctl	cd ~/colcon_ws && colcon build --packages-select auto_nav && cd ~/colcon_ws/src/auto_nav/auto_nav && ros2 run auto_nav control
exp	cd ~/colcon_ws && colcon build --packages-select auto_nav && cd ~/colcon_ws/src/auto_nav/auto_nav && ros2 run auto_nav explore
rteleop	ros2 run turtlebot3_teleop teleop_keyboard

RPi4 Alias	Command
rosbu	ros2 launch turtlebot3_bringup robot.launch.py
line	cd ~/turtlebot3_ws/src && colcon build --packages-select r2auto_nav && source install/setup.bash && ros2 run r2auto_nav l

9.4.7 Setting Up RViz Environment

Run Slam Toolbox

```
toolbox
```

Open RViz

```
rviz
```

Add the following Nodes:

- Map
- LaserScan
- Marker Array
- Interactive Markers

'File' > 'Save Config' and save to the default directory, `~/.rviz`, named as `mission.rviz` (So that this configuration launches upon calling `rvm`)

9.4.8 Executing program

Place S/Ue on the starting line.

Terminal 1: Bringup the TurtleBot

```
sshrp
rosbu
```

Terminal 2: Launch slam

```
toolbox
```

Terminal 3: Launch RViz

```
rvm
```

Terminal 4: Run `control.py` or `explore.py`,

```
ctl
```

OR

```
exp
```

Terminal 5: Run `lineFollower.py`

```
line
```

S/Ue should then proceed to execute the mission.

9.4 Safety Precautions

- Do not overcharge the robot to ensure battery health.
- Check that the robot is assembled properly. Improper assembly might lead to components' damage.
- Do not use the robot near fire or heat.
- Do not place Li-Po batteries in or near water.
- Do not use excessive force on nuts, bolts, or other parts of the robot.

9.5 Troubleshooting

Software
To test S/Ue's movement independently, use `rteleop` to manually control movement.
To test payload actuation, run `python3 payload.py`.
To test ESP32, run `python3 esp.py`
Hardware
Check battery level using the battery tester included in the kit. Additionally, beeping sounds from the OpenCR board indicated battery level is not adequate for motor operations.
To check the calibration of line follower sensor, follow the steps mentioned in section 9.3.
Ensure that all pins are connected as shown in schematic in case of failure of any component/subsystem. If the hardware subsystems do not work in spite of this, perform software debugging.

11.0 Future Scope of Expansion

11.1 Mechanical

One major change would be to reduce the centre of gravity of the payload. This could be done by using a scissor lift mechanism to lift up the payload. Additionally, we could also try out other mechanism ideas such as the box-slingshot mechanism or the flywheel, which does not result in a high centre of gravity.

Another would be altering the opening mechanism of the box. The ramp could be replaced by a simple flat wall which is attached to a servo. As the servo moves, the 'wall' moves sideways to

give way to the balls to fall off. Additionally an extendable ramp system could be attached to the base of the sloped box. This ramp slides and expands, leading the balls into the bucket.

For the mechanical aspect of the design, there are several possible ways one could design it. So if given another opportunity, I would want to try out the flywheel possibility and try to make the payload as small and efficient as possible.

11.2 Electrical

Better cable management of wires and labelling of components should be done to help reduce the time taken to fix electrical issues.This could be done by mapping individual wire to wire connections from the line sensors and payload servo to the Raspberry Pi on a printed circuit board, along with markings to indicate the appropriate layout.

Line following sensors with higher accuracy could be used to make sensor calibration easier.

11.3 Software

Many bugs still exist in the frontier-based navigation script preventing S/Ue from freely navigating and efficiently mapping the entire area. This includes the primary bug of S/Ue struggling to identify and thus colliding with corners. For further expansion, a better collision avoidance logic should be implemented. Furthermore, to improve the efficiency of the algorithm, perhaps a different data structure better suited for frontier-based navigation could be used. A specific form of frontier navigation which was explored in the literature review, known as wavefront frontier navigation, uses a priority queue to explore the maze. Implementation of such a data structure may allow S/Ue to more effectively navigate.