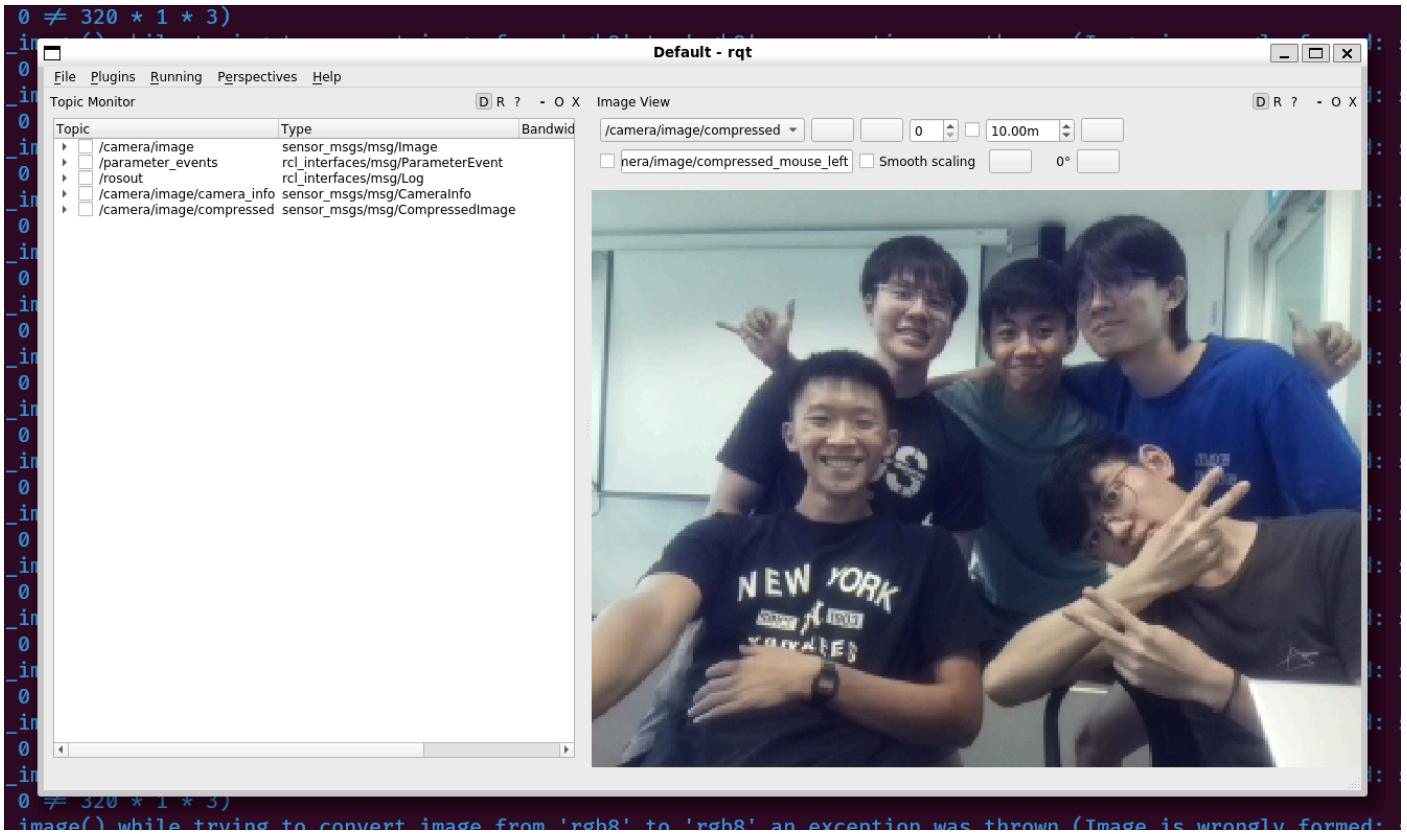


AY23/24 Semester 2

EG2310 Group 13 G2 Report

Name	Student ID
Dylan Liew Yan Hong	A0255312N
Jervin Tan Si Kai	A0286659E
Kashfy Ilxilim Bin Zulkarna'in	A0272245H
Lim Jing Heng	A0272417E
Toh Yan Ting	A0261533L

1. Con-Ops ***3***
a. Original Con-Ops *3*
b. Main Changes from original Con-Ops *3*
i. Camera *3*
ii. Mechanical Linkage *3*
2. High Level Design ***4***
a. Mechanical CAD *4*
b. Electrical Schematic *6*
c. Software Con-Ops and Flowchart *6*
3. Subsystem Design ***7***
a. Mechanical Design *7*
i. Servo Torque Calculations *9*
b. Software Design & Documentation *10*
i. Costmap *10*
ii. Origin Coordinates *10*
iii. A Goal Identification* *10*
iv. A Heuristics* *11*
v. Movement *11*
vi. Bucket Detection *11*
4. User Manual, Interface Control ***12***
a. Operating Instructions for User *12*
b. Interface Control *13*
5. Testing Documentation, Problems Faced and Future Works ***13***
a. Mechanical *13*
i. Weight reduction of arm and carrier subsystem *13*
ii. 3D-Print Friendly *13*
iii. Future Works *13*
b. Software *14*
i. Robot Movement and Path-Planning *14*
ii. Threshold Values for Occupied and Unoccupied *14*
iii. Bucket Identification *14*
iv. Lobby Localisation *14*
v. Future Works *15*
6. Annex ***16***
a. Camera Setup *16*
b. Different Views of our Mission Robot *17*
c. High Level Design – Electrical Block Diagram 20
d. High Level Design - Software Con-ops and Flowchart *21*
e. Interface Control *22*

1. Con-Ops

a. Original Con-Ops

Our original Con-Ops Document and Preliminary Design Review Presentation is hosted in our GitHub Repository.

b. Main Changes from original Con-Ops

i. Camera

We were intending to utilise the Raspberry Pi Camera V2 to orient our robot towards the bucket inside the 2 rooms, by the setup process shown on Camera Setup and the code can be found here. However, the camera was not used because the colour detection and thresholding was too dependent on the lighting and environment of the maze setup, and the calibration would have taken up too much time out of the 20 minutes of set-up time we had available to us.

Instead of a camera, the LiDAR data was used to identify the bucket by checking the distances of adjacent angles with the expected distances of a curved surface.

ii. Mechanical Linkage

A parallel 4-bar linkage is used instead of a timing belt as originally planned. This allows all parts of the robot to be coupled together securely with screws, rather than relying on friction to keep the timing belt attached and transmitting motion. This also reduces complexity and allows the mechanism to be rapidly prototyped out of 3D-printed parts, rather than needing to purchase and wait for the timing belt and gears to arrive.

2. ***High Level Design***

a. Mechanical CAD

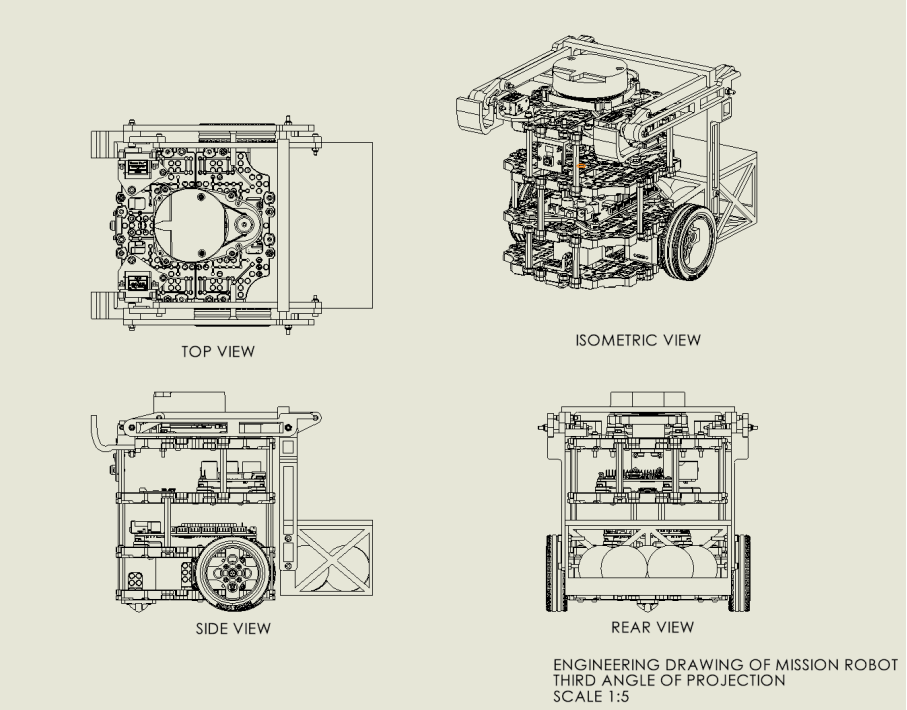


Figure 1

As shown in the above engineering drawing (*Figure 1*), our mission robot is an extension of the base Turtlebot3. Our main design consideration were:

- **Simplicity** - As much as possible, the base structure remained untouched, and we limited the usage of extra electrical components. We also minimised the number of moving parts to decrease the possibility of mechanical failure.
- **Low Center of Gravity** - Height of CG will cause *r* (perpendicular distance to pivot of robot) to increase. Since moment due inertia *m* is proportional to *r*, a larger *r* value will generate larger moment, increasing the possibility of the robot losing traction when it accelerates / brakes. (Refer to *Figure 2*)

$\boldsymbol{r}$ = Distance of CG from pivot point b (Base of wheel)

$$moment\ due\ to\ inertia\ when\ robot\ accelerates \propto r$$

More Centred/Lower CG, ⬇ Moment, ⬇ Tipping

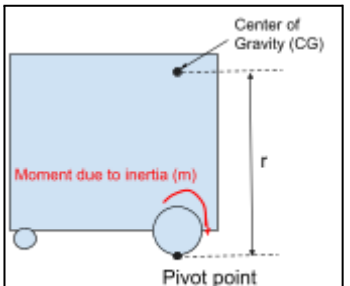


Figure 2

- **Centred Center of Gravity** - As much as possible, we ensured that the overall weight of the robot was centred at the drive wheels. This minimises the occurrence of wheel slippage when travelling about the playing field as the weight of the robot is concentrated on the wheels. (Refer to *Figure 3*)

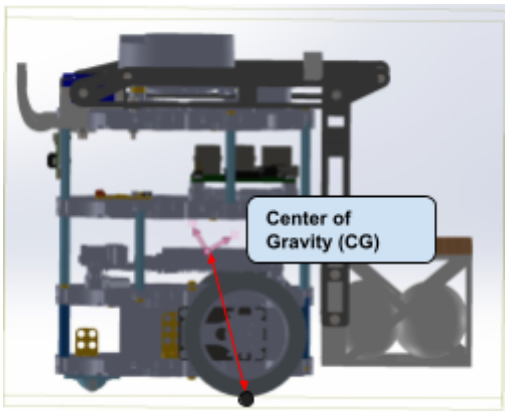


Figure 3

- **LiDAR should not be blocked** - The Mission Robot is capable of scanning 360° around itself. (Refer to *Figure 4*)

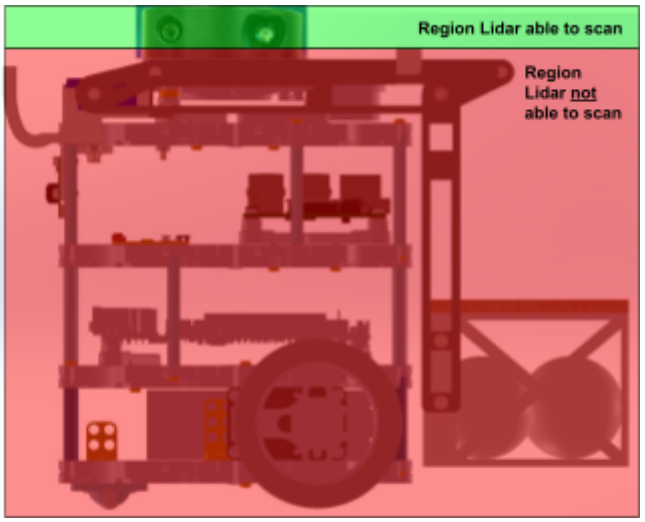
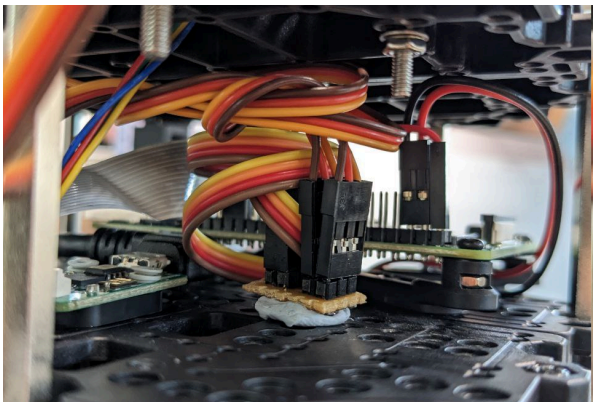


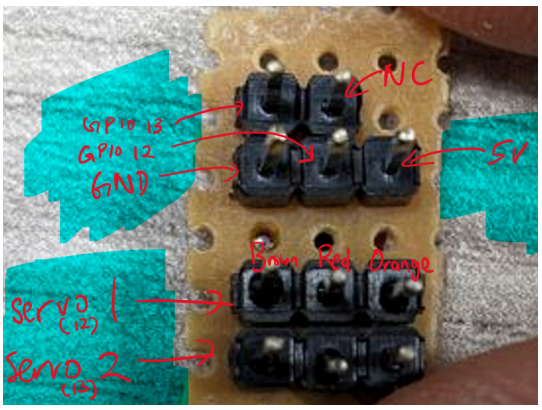
Figure 4

b. Electrical Schematic

The electrical block diagram is shown in the Annex.

Aligned with the mechanical design goal of simplicity, we wanted to decrease the number of extra electrical components beyond whatever was included with the base Turtlebot3. Hence, we only utilised 2 extra SG90 servos and 1 Camera connected directly to the Raspberry Pi. The 2 servos are connected to a simple proto-board which splits the 5V and GND wires into the servo header pins.





c. Software Con-Ops and Flowchart

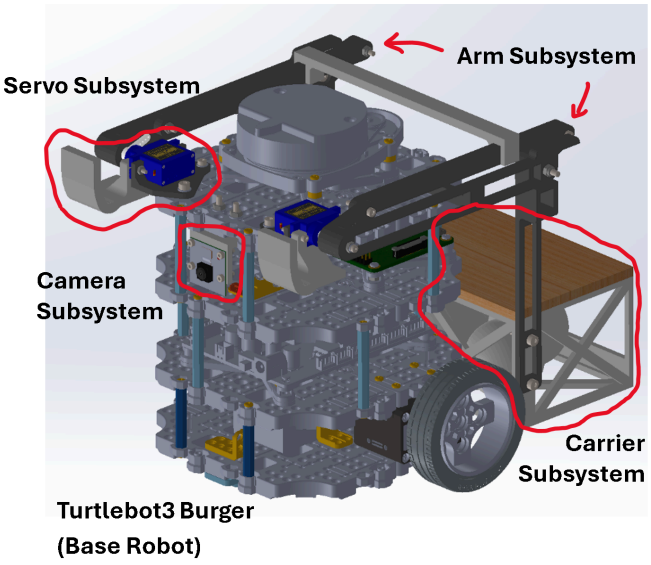
The overall software flowchart is shown in the Annex.

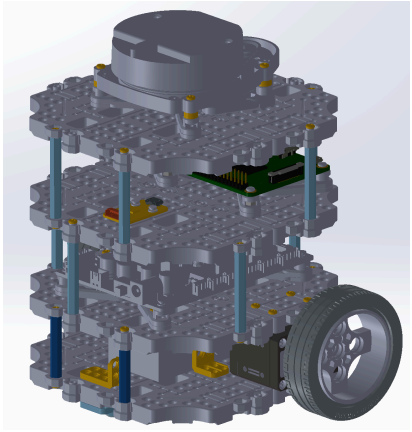
Due to the very short 20 minutes of runtime that we are given during the actual mission run, we did not want to waste time setting up and tearing down markers such as coloured codes, AprilTags, or lines for the robot to trace. We relied fully on the SLAM generated from the LiDAR and Odometry data to plan and follow a path through the maze.

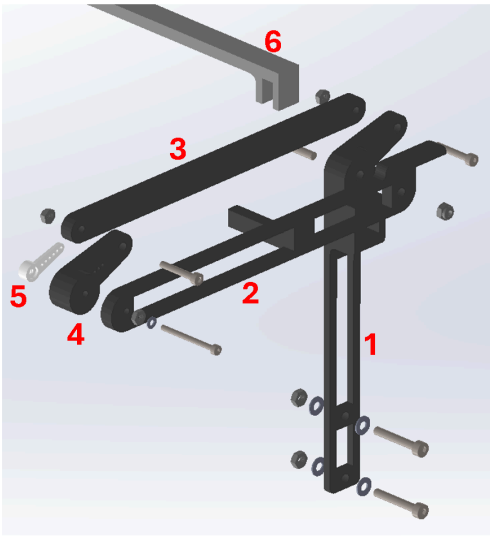
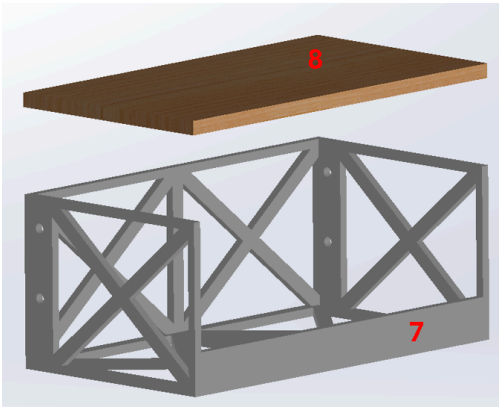
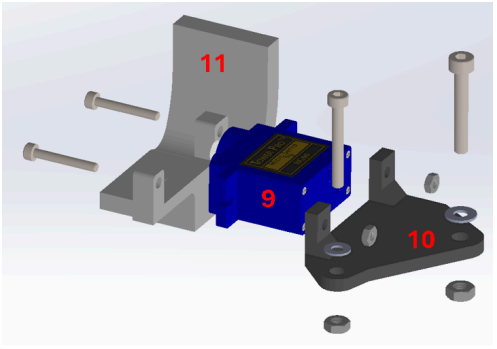
3. ***Subsystem Design***

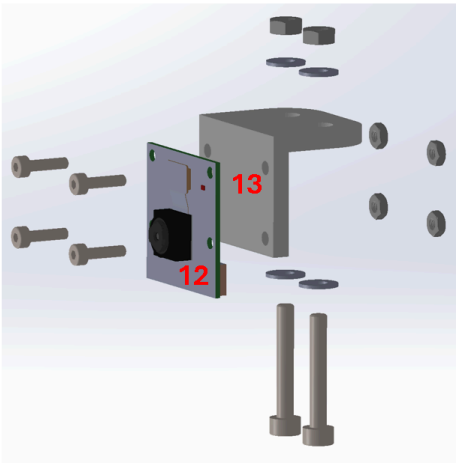
a. Mechanical Design

The mission robot can be broken down into different subsystems, namely Turtlebot3, Arm, Carrier, Servo, and Camera subsystems. The locations of the subsystems can be seen on the mission robot below.



Subsystem	No.	Part	Qty
Turtlebot3 Burger Subsystem 		Refer to the following link for the assembly instructions of the Turtlebot3 Burger robot: https://www.robotis.com/service/download.php?no=748	1

Subsystem	No.	Part	Qty
Arm Subsystem 	1	Upper Arm	2
	2	Lower Arm	2
	3	Connecting Arm	2
	4	Servo Horn Connector	2
	5	Servo Horn	2
	6	Linking Bridge	1
	Explanation: The four arm linkage was chosen over other methods of payload delivery as it was in line with our main design considerations. The design mechanism was consistent and reliable, manufacturing was simple, and the centre of gravity was kept low.		
Carrier Subsystem 	7	Ball Carrier	1
	8	Cardboard Lid	1
	Explanation: The carrier is meant to contain the payload and deliver them when the mechanism is deployed. As the carrier was positioned at the end of the delivery mechanism, weight had to be reduced as much as possible to ensure the servo motors were able to lift the carrier and payload.		
Servo Subsystem 	9	SG90 Servo Motor	2
	10	Servo Mount	2
	11	Arm Barrier	2
	Explanation: The servo subsystem is the powerhouse of the delivery mechanism. The servos lift the carrier and payload to deliver the payload to the intended location. The arm barrier serves as a physical barrier to stop the arms from rotating past its intended limit.		

Subsystem	No.	Part	Qty
Camera Subsystem 	12	Raspberry Pi Camera V2	1
	13	Camera Mount	1
	Explanation: Our initial concept design required the use of the RPi camera to recognise the colour and direct the mission robot to the red bucket. However, during testing, we realised that the LiDAR was more reliable and thus we switched to using the LiDAR. As the RPi camera was already installed during the decision making (and that the robot looks aesthetically better with the camera), we did not remove the camera from the robot.		

i. Servo Torque Calculations

Carrier Subsystem

- Weight with 5 balls = 123g
- Length from CG to Servo = 220mm

Arm Subsystem

- Weight = 63g
- Length from CG to Servo = 101mm

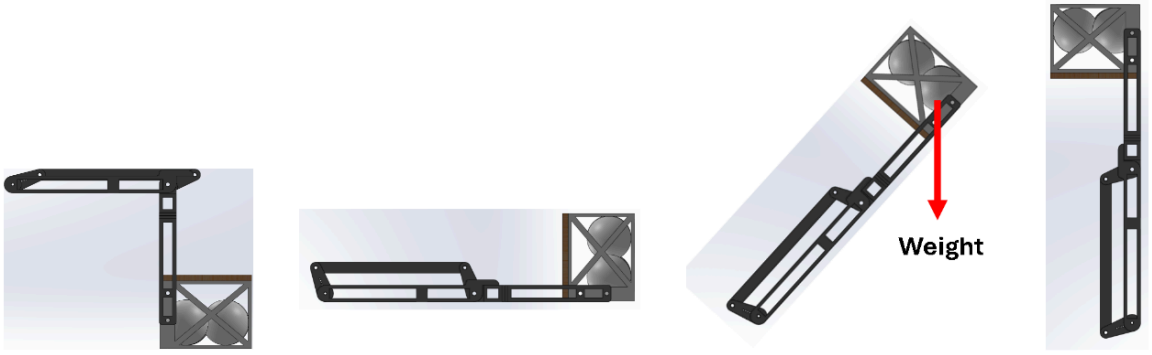
Required Torque = 0.123 * 22 + 0.063 * 10.1 = **3.34 kg cm**

SG90 Servo Motor Max Torque = 1.6 kg cm

Total Torque = 1.6 * 2 = **3.2 kg cm**

This calculation shows the maximum torque needed to lift the carrier and payload when the arms are horizontal (Stage 2). At first glance, the servos seem to have insufficient torque to lift the carrier and payload.

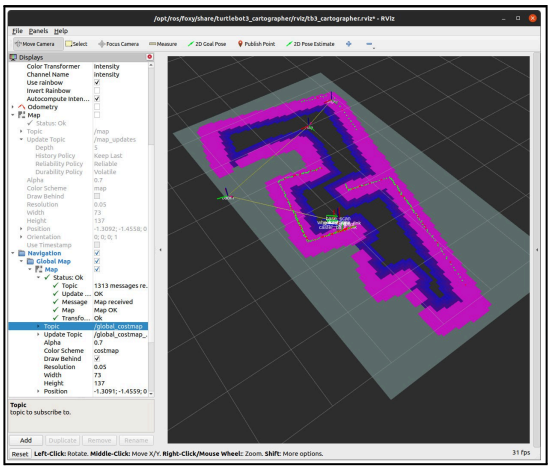
- However, the arms do not start at Stage 2 (Instead, it starts at Stage 1). At stage 1, the servo motors only need to rotate the upper arm and carrier around a smaller radius, as such, the actual torque needed will be much smaller than 3.2 kg cm.
- When the arms become horizontal (when it is momentarily at Stage 2), there exists upwards momentum of the arm. This, together with the servo's continued torque, is able to lift the carrier and payload from Stage 1 directly to Stage 3, skipping past Stage 2 where the servos would have had insufficient torque.
- Hence, the effective torque required reduces as the weight of the payload is not perpendicular to the arms, making the servos strong enough to lift the arm from Stage 3 to Stage 4.



Stage 1 Stage 2 Stage 3 Stage 4

b. Software Design & Documentation

Our source code can be found in our GitHub repository.

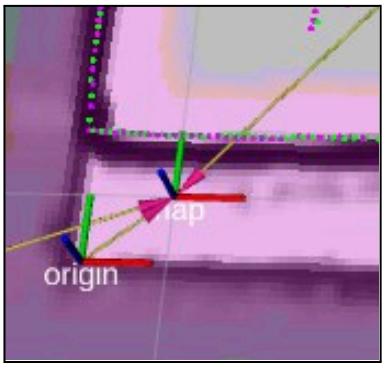


i. Costmap

The costmap is generated from the SLAM occupancy grid map, and publishes to the `/global_costmap/costmap` topic to be visualised in RViz and used as a heuristic for our A* search. The code for this node can be found here.

- The inflation function will first erode away stray unoccupied points (such as those created due to gaps between walls). This is done by a morphological closing of the unknown points (dilating then eroding).
- It will then dilate all walls (walls are defined as grids with occupied probability above the specified `threshold`)
- From the limit of the dilated walls, the adjacent grids (number of adjacent grids are determined by the specified `inflation_radius`) would then be inflated with a cost corresponding to the specified `inflation_step`, with the closest grids inflated the most.

ii. Origin Coordinates



Upon start-up, an occupancy map is produced based on LiDAR data. Based on this occupancy map, the `find_origin()` function searches for an unoccupied point with the lowest x and y coordinates in the occupancy map. With proper orientation of the robot in the maze prior, this unoccupied point corresponds to the bottom-left corner of the maze environment. This point is then defined as the origin and a transform is defined in RViz so that this origin can be used as a reference point for the rest of the mission.

iii. A Goal Identification*

The robot is programmed to exit the maze as quickly as possible upon finding the exit to complete the mission, before exploring the rest of the maze to map the maze completely. As such, from the start position, the A* a path-finding algorithm will attempt to locate an unknown point in the current occupancy map with the highest possible y-coordinate, with reference to the defined origin. This point will be defined as the goal for the current path to be found and serves as the heuristic for the current iteration of `a_star_search()`. By searching for unknown points with the highest y-coordinate, this allows the robot to move towards the maze exit as soon as possible to complete the mission in the shortest time possible.

Upon completion of the mission, the robot will instead locate an unknown point within the maze, rather than prioritising unknown points with the highest y-coordinate. This is to prevent the robot from continuing to explore areas outside the maze so as to complete mapping the maze in the shortest time possible.

iv. A Heuristics*

To find an optimum path between the robot and its identified goal, a costmap was implemented. The inflation cost is implemented into our `a_star_search()` pathfinding function so that the returned path will prioritise paths that are further from the wall. Hence, the risk of collisions between the robot and maze walls are significantly reduced.

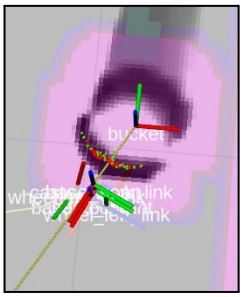
In addition, a turning cost was implemented into `a_star_search()`. By assigning an additional cost when a turn is made during path-planning, the A* algorithm prioritises a path with fewer turns. This causes the robot to make fewer turns when travelling along the path, thereby allowing the robot to traverse the maze more quickly.

v. Movement

After the path is generated, the robot will decompose the path into individual waypoints at the end of straight line segments. The robot will then continuously turn towards and move to the next waypoint. The robot does this through PID controllers targeting the output linear and angular velocities. The move functions can be found here.

- The robot first obtains its own transform, and the target waypoint's coordinates.
- From the delta x and delta y between the robot's and the waypoint's coordinates, the target angle can be determined from an arctangent calculation.
- The euclidean distance can also be obtained from that delta x and delta y, to control the linear velocity (the further the euclidean distance, the faster the robot should move).

vi. Bucket Detection

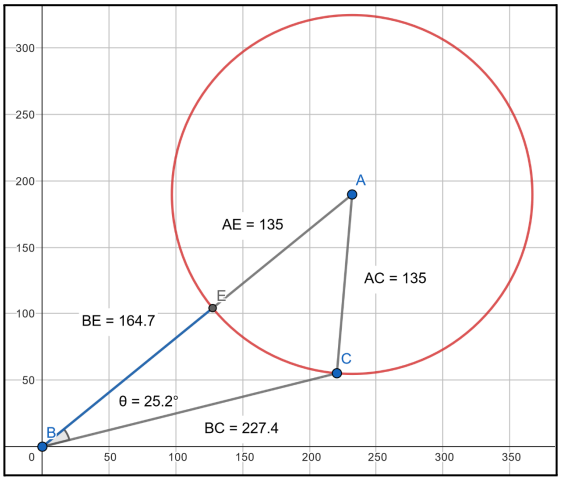


The robot utilises the LiDAR data to detect the curved surface of the bucket, based on the radius of the bucket. This function can be found here. Once a curved surface similar to that of the bucket is detected by the LiDAR, the robot locates the curved surface, publishes a transform, and moves towards it.

Specifically, in the function to check for the LiDAR curved surface, the robot will first sort the LiDAR data in ascending order, then check each data point with adjacent points to verify whether that data point actually corresponds to the bucket.

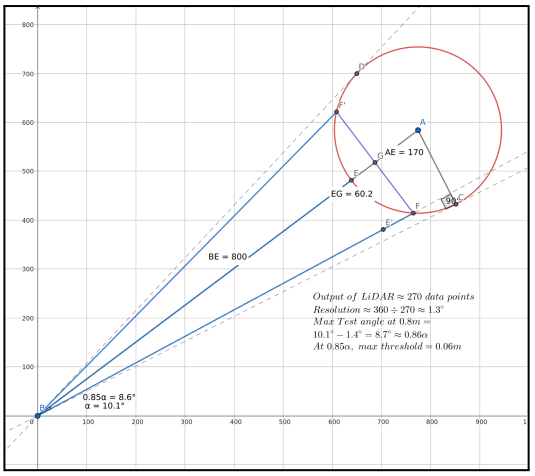
In this case:

- Distance AE = AC is the measured radius of the bucket
- Distance BE is the shortest distance from robot to bucket (as obtained from LiDAR)
- Distance BC is the distance to verify whether this particular angle and data being checked corresponds to a bucket or not.



In the triangle ABC, since we know AB and AC, as well as ∠ABC, we can use the cosine rule to find the expected distance of BC to verify against our LiDAR data.

- The cosine rule is: $c^2 = a^2 + b^2 - 2ab \cdot cos(\theta)$, where θ is the angle opposite side C.
- In this case, the distance BC corresponds to side b, which is the unknown. Rearranging the equation, $(1) \cdot b^2 + [-2a \cdot cos(\theta)] \cdot b + (a^2 - c^2) \cdot b^0 = 0$.
- This is a quadratic equation, with the 2 roots corresponding to the 2 possible points on the circle where this equation can be true. Since the LiDAR data would correspond to the closer out of the 2 points of the circle, by the quadratic formula, $b = \frac{-[-2a \cdot cos(\theta)] \pm \sqrt{[-2a \cdot cos(\theta)]^2 - 4 \cdot (1) \cdot (a^2 - c^2)}}{2 \cdot (1)}$.



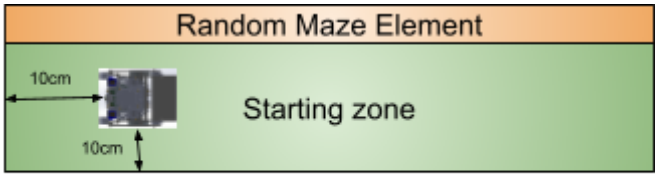
Adjacent points will be checked based on fractions of the maximum expected θ. As shown in this figure, assuming the robot detects the bucket from 0.8m away, with the resolution of our LiDAR ≈1.3°, we are only able to check values up to $0.85*\theta_{max}$.

4. *User Manual, Interface Control*

This section includes operating instructions for user and factory acceptance tests. For more details about how the system's components interact with each other, please refer to our full hardware design document.

a. *Operating Instructions for User*

i. Place the robot in the following orientation such that the turtblebot's RPi Camera will be **facing the wall boundary** of the playing field and **10cm** away. (Refer to figure below)



ii. Place the **five** ping pong balls into the ball carrier

iii. Connect both the robot and your laptop to a mobile network (hotspot) with the SSID *pixel* and password ********

iv. Switch on the robot with the switch on the OpenCR board

v. Append the underline portion of the following code with the obtained IP address. (Following code is found in line 41 of the `maze.py` file)

```
ipaddr = "<insert.ip.address.here>"
```

vi. On your laptop terminal, key in the following command: `ssh ubuntu@turtlepi1.local`. Key in the following password: **eg2310grp13**

vii. On the Raspberry-Pi, run the following command: `rosbu`

viii. On the Raspberry-Pi (in a separate terminal), run the following command: `rosservo`

ix. On your laptop, run the following command: `rslam`

x. On your laptop (in a separate terminal), run the following command: `costmap`

xi. On your laptop (in a separate terminal), run the following command: `ros2 run auto_nav maze`

b. Interface Control

A flowchart depicting the interface control is provided in the Annex. Only rosservo is run on the Raspberry-Pi, with all other programs being run on the user's laptop.

5. Testing Documentation, Problems Faced and Future Works

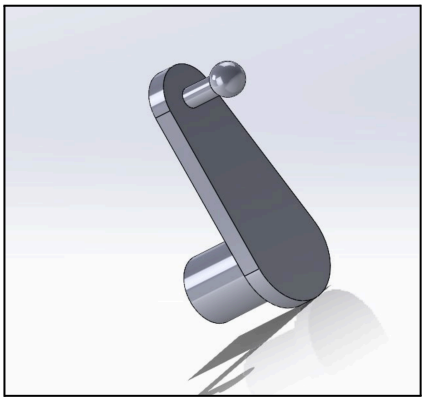
a. Mechanical

i. Weight reduction of arm and carrier subsystem

During testing, we noticed that the torque provided by 2 SG90 servo motors is not enough to actuate the payload. Thus we reduced the weight of the arm and carrier by cutting out unnecessary parts.

- The upper part of the carrier is replaced by a piece of cardboard to reduce weight.
- The carrier was designed to be "mesh-like", with holes slightly smaller than each ping pong ball's size to minimise weight.
- Each linkage arm was also made to be "hollow", with holes in the middle to cut material and weight.

ii. 3D-Print Friendly



After designing the necessary components for our payload, we realised that the majority of the components are not 3D-printing friendly. They had multiple overhanging parts which resulted in the need to print supports, which increased printing time significantly and wasted material. For example, for the part shown, there is no orientation that it can be printed from without requiring supports.

To tackle this issue, we redesigned our payload components such that there are minimal to no overhanging parts.

iii. Future Works

- In order to improve the reliability and performance of the arm and carrier mechanism, the weight of the arm and carrier could be further reduced by using lighter materials, such as balsa wood. These wooden pieces can be laser cut to our specifications.
- In addition, a stronger servo (capable of producing higher torque) could be used to replace the SG90 servos on the robot. These changes would allow for a smoother actuation of the arm.

b. Software

i. Robot Movement and Path-Planning

Initially, only the dilate function was used as a way to prevent collisions between the robot and the maze walls. However, after further testing, it was found that dilating the walls by a factor of half the width of the robot meant that the robot may still come into contact with the walls when turning near a wall due to protruding parts from the robot (such as our payload mechanism). However, dilating the walls by too large a factor would result in no path being found during path planning. This occurred in cases where the narrow passages in the maze were exactly two turtlebot widths wide. As such, the occupancy map would reflect an entirely occupied area that will not be explored by the robot. This led to the implementation of the costmap, which would cause the robot to find a path that is not too close to the walls of the maze, thereby reducing collisions. With the costmap, the walls of the maze need not be dilated excessively to prevent the robot from colliding with the walls.

ii. Threshold Values for Occupied and Unoccupied

Using the default occupancy threshold value of 50 to identify a point in the occupancy map as "occupied", several issues were experienced with the identification of occupied spaces in the maze. For example, the origin in `FirstOccupy()` was not defined accurately to the inner bottom left corner of the maze. With higher threshold values, more time was also required to accurately define a space in the maze as occupied. After further experimentation, it was found that an occupancy threshold value of 52 yielded the best results.

iii. Bucket Identification

Initially, the Raspberry-Pi Camera v2 was used as a means to identify the red bucket. This camera would be used to detect red objects in its field of view, thereby locating the red bucket. This method was found to be unreliable, due to various conditions, such as lighting, affecting colour detection. Hence, calibration of the camera's colour detection would be required before the run. As this calibration is time-consuming, the camera was not used so as to maximise the 20 minute time limit given for the run. It was also found that the boards being used as walls may contain red coloured graphics, thus leading to possible false detections by the camera. This led to the usage of LiDAR data to detect the bucket instead.

iv. Lobby Localisation

Initially, we relied on the origin transform and measured lobby coordinates to localise our robot and set the goal in the middle of both doors. However, if the SLAM was not started with the robot pointed straight, the coordinates would not be accurate since the x and y axes would not be aligned with the actual length and width of the maze. Furthermore, since we only open the doors when we reach the lobby, the SLAM detects the door as occupied, and we are unable to generate a path into the room's coordinates based on the SLAM map alone.

Hence, we had to resort to some "hard-coding" by measuring the LiDAR distance from the robot to the end wall, and positioning our robot in between the doors through this distance. The movement into the door (turning left/right then moving forward) would also have to be hard-coded as a result.

v. Future Works

- During our final robot run, the robot was not successful on the first attempt due to a bad network connection between the laptop's `cmd_vel` publisher and the robot.
    - A router could be used to solve the unstable mobile hotspot connection between the RPi and the laptop, to ensure that ROS topics published will be subscribed to without any delay.
    - The router can also be mounted directly on the robot, such that the robot can always maintain a stable network connection
    - The current mover functions could also be run locally on the RPi, and be exposed via a ROS action to the laptop. This allows most of the computation to be run on the laptop, while the lower-level PID movement control can run without latency on the robot itself.
- Our A* search is currently limited to 8 directions (45 degree directions), which will sometimes result in needless turns and movements.
    - Path smoothing algorithms such as a "string-pulling" algorithm can be implemented to decrease and optimise our robot's movements.
    - We could move to any-angle path planners such as Theta* such that the path generated would already be optimised.
    - We could also implement Voronoi Diagrams to ensure that our robot always moves in between walls, to decrease the chances where it would collide with the wall.
    - Rather than our current naive approach of turning and moving towards individual waypoints, we could implement a path-following controller such as Regulated Pure Pursuit so that the robot's movement would be smoothened dynamically.
- There is currently no active collision avoidance implemented, apart from simply dilating the walls and generating a path based on those dilated walls.
    - A path-following controller can be integrated with a collision monitor, to dynamically modify the robot's trajectory to avoid obstacles in the current path.
- Some helper functions could have been better implemented to save some time while we were troubleshooting our robot / during the run when we need to quickly tune some parameters.
    - Could have made use of RViz `/clicked_point` topic to publish the origin transform when the map is not being generated properly.
    - Could have published the A* generated paths and waypoints to RViz rather than matplotlib, so that it would be less buggy.

6. Annex

a. Camera Setup

1. Install OpenCV from the Ubuntu Repository by running the following command in a terminal instance.

```
$ sudo apt update
$ sudo apt install libopencv-dev python3-opencv
```

2. On Ubuntu: Install the VideoCore libraries. Since these conflict with packages *libgles2-mesa-dev* and *mesa-common-dev*, those ones need to remove first.

```
sudo apt autoremove --purge libgles2-mesa-dev mesa-common-dev
sudo add-apt-repository ppa:ubuntu-pi-flavour-makers/ppa
sudo apt install libraspberrypi-bin libraspberrypi-dev
```

3. Change to the workspace and build:

```
colcon build
```

4. Clone the GitHub repository to the ubuntu laptop.

```
git clone git@github.com:christianrauch/raspicam2_node.git
```

5. Activate the camera via raspi-config (sudo apt install raspi-config)
6. Make sure that your user is in the *video* group by running the following command.

```
groups|grep video
```

7. Start the node executable.

```
ros2 run auto_nav jervin_rpicam
```

8. Run the following command on laptop to display video from camera.

```
rqt
```

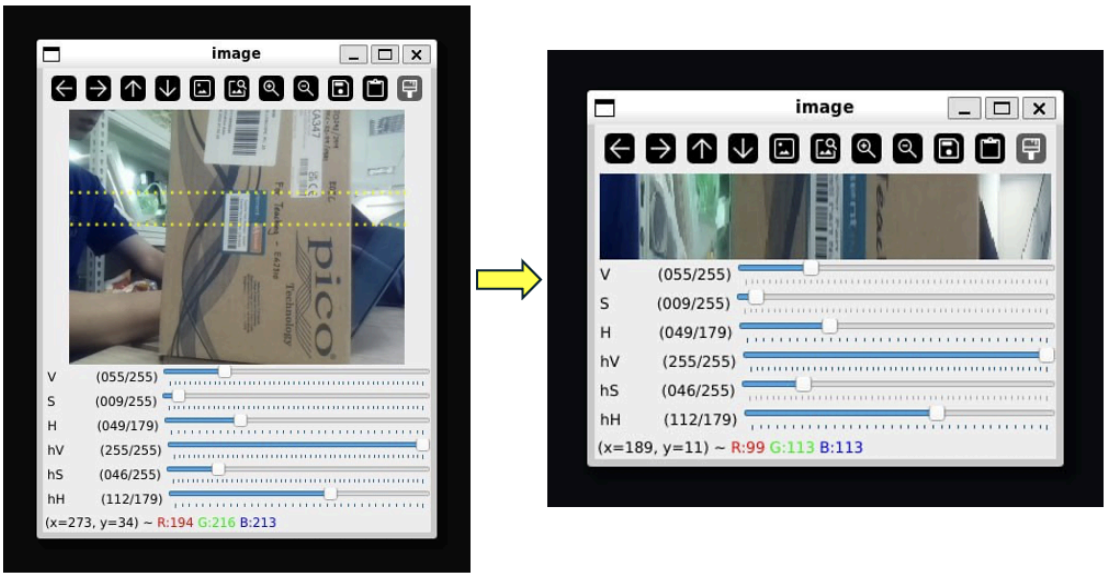
9. Run the python file for colour detection and the centre point of frame capture will publish.

```
ros2 run auto_nav jervin_rpicam
```

10. Change the default parameters in params.yaml, set the **height to 60**, **fps to 30** and **roi_h to 0.25** to reduce latency and only the region of interested will be detected.

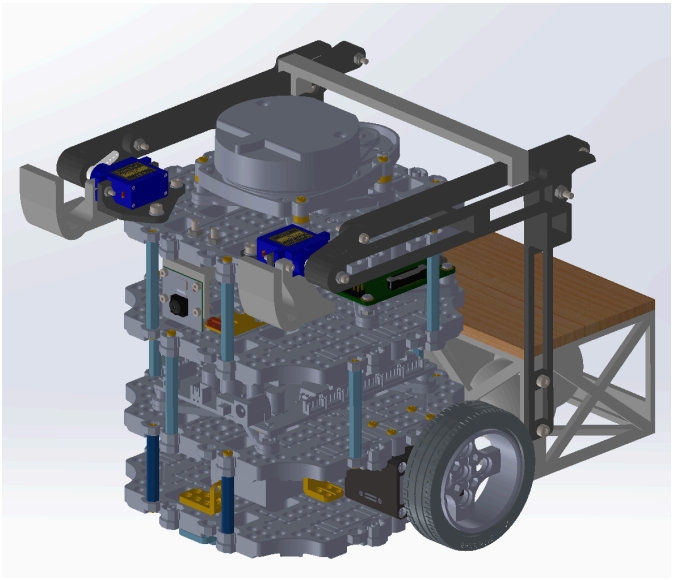
```
camera:
    raspicam2:
        ros__parameters:
            image_transport: compressed
            width: 320
            height: 240
            fps: 90
            quality: 80                # Set JPEG quality (0 to 100)
            roi_h: 1.0                 # Set sensor region of interest, h
```

11. The change of parameters will result the image shows as below:

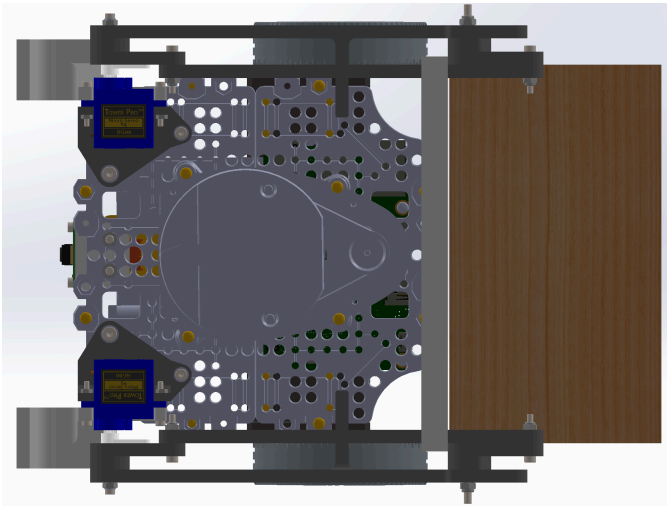


b. Different Views of our Mission Robot

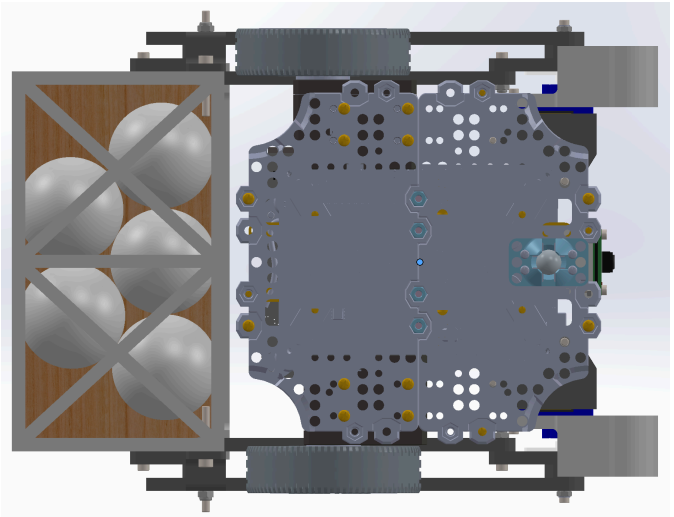
Isometric View

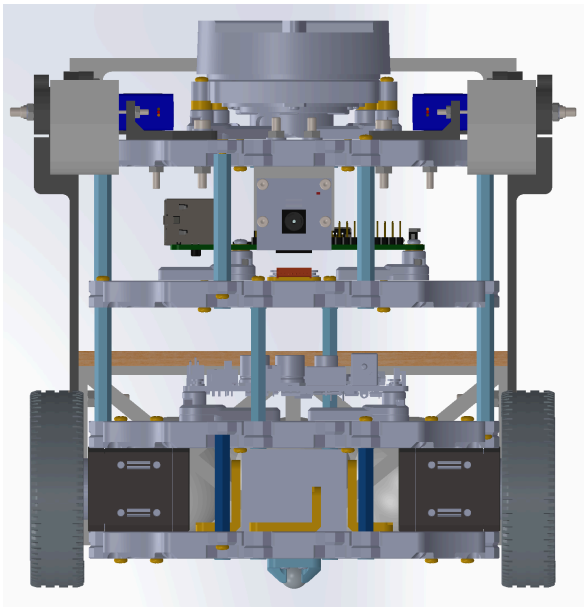
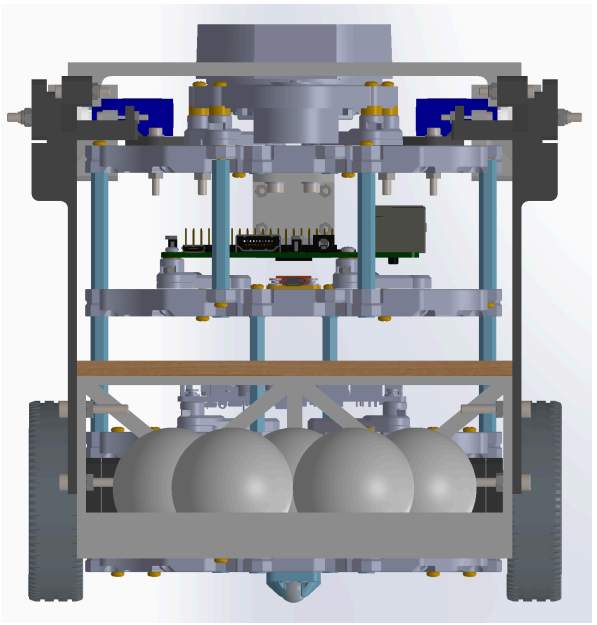
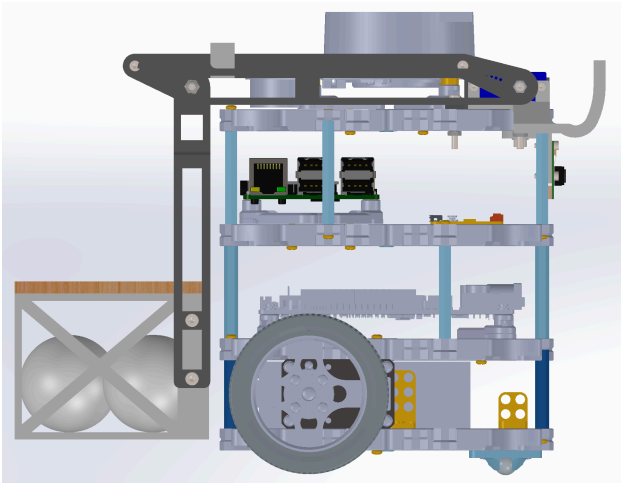
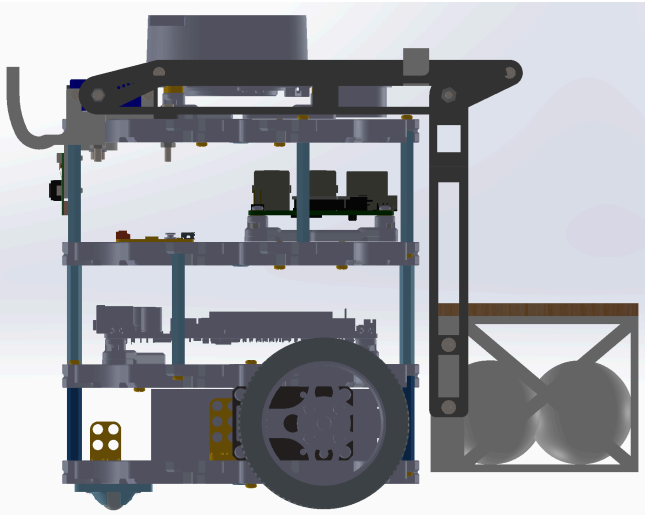


Top View

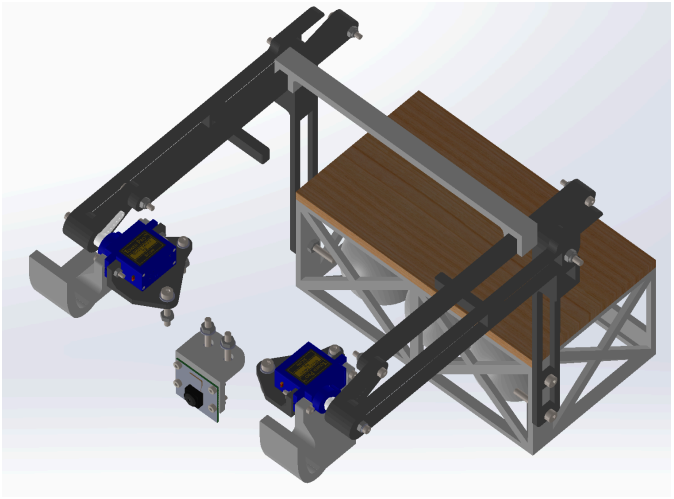


Bottom View

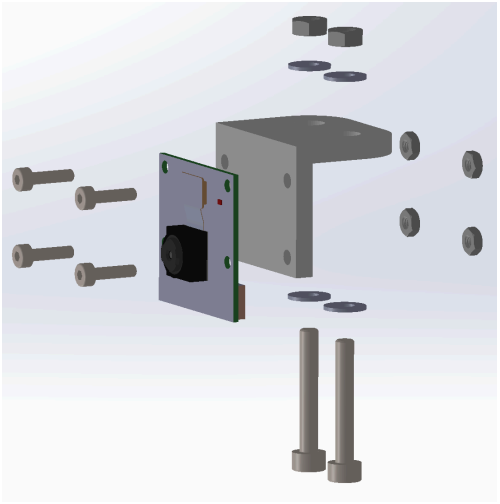


Front View	Back View
	
Right Side View	**Left Side View**
	

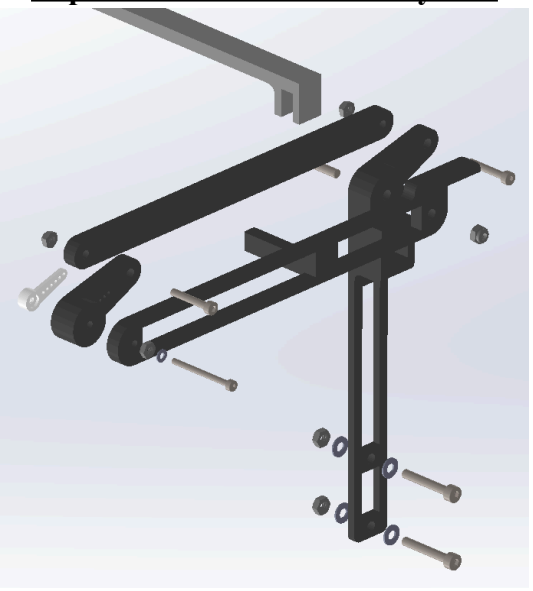
View of Payload System



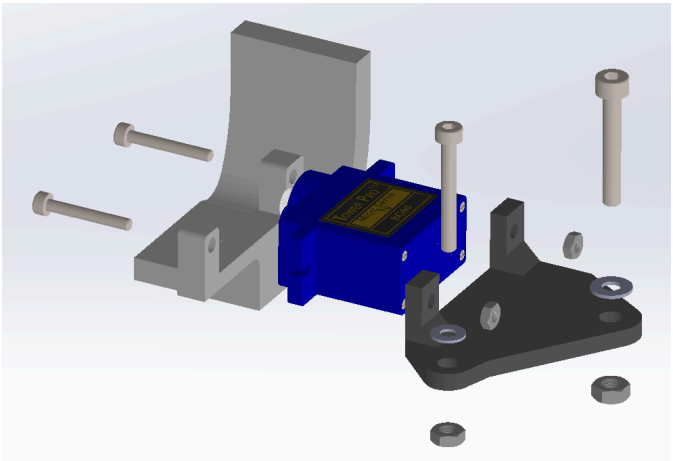
Exploded View of Camera Subsystem



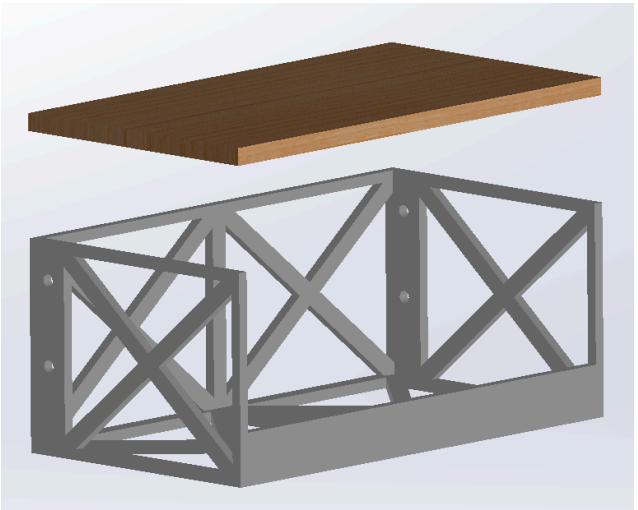
Exploded View of Arm Subsystem

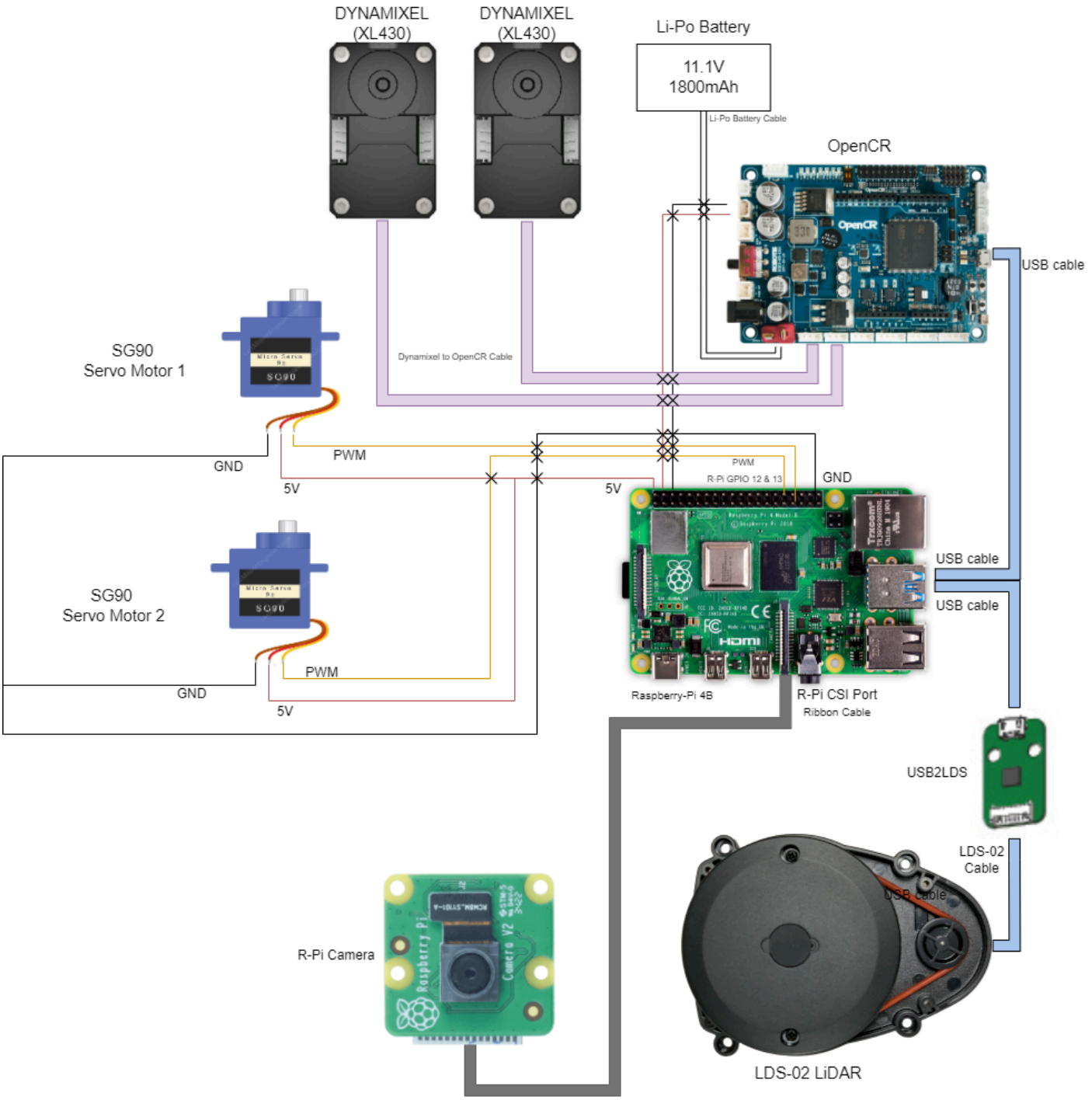


Exploded View of Servo Subsystem

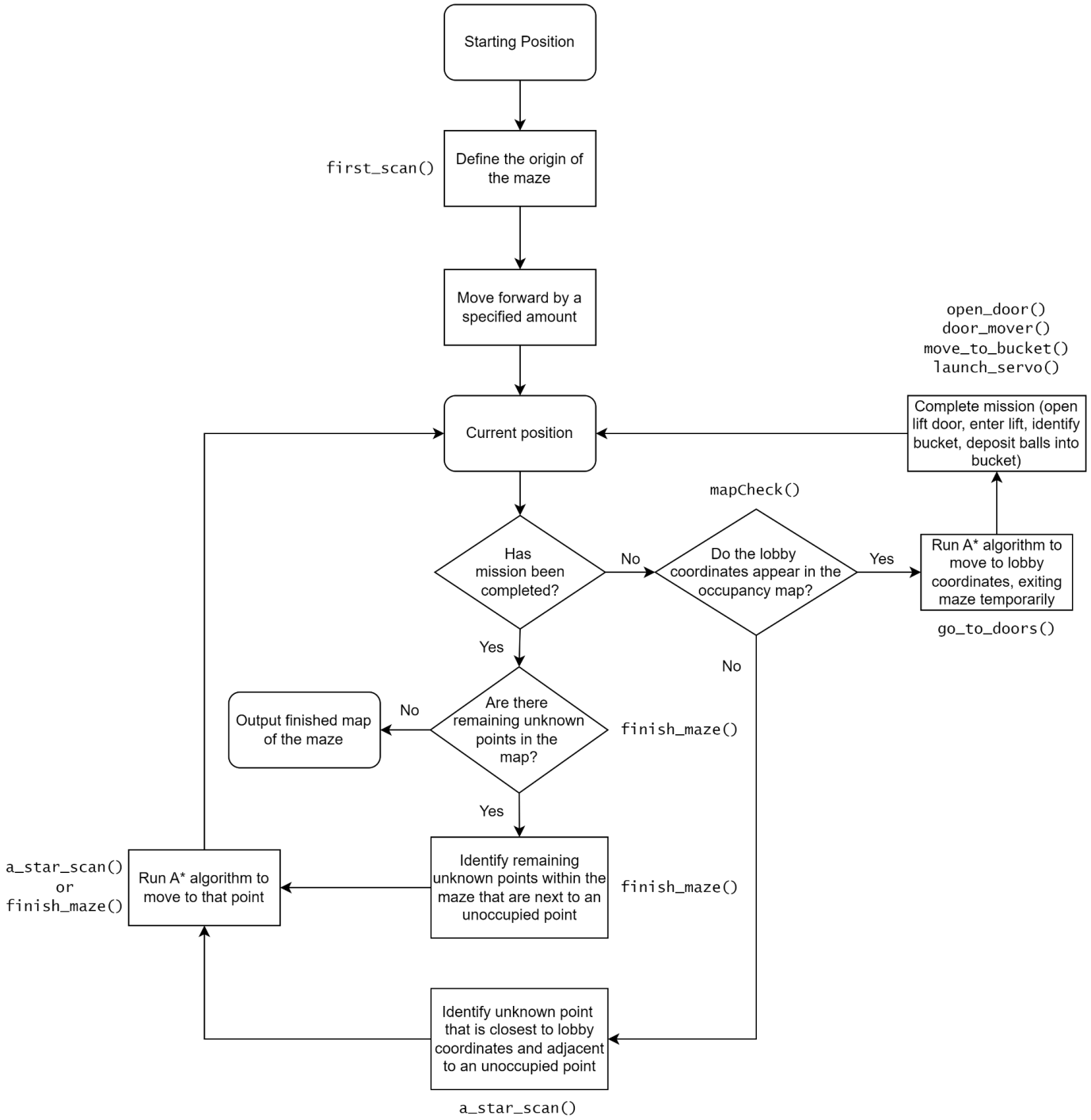


Exploded View of Carrier Subsystem

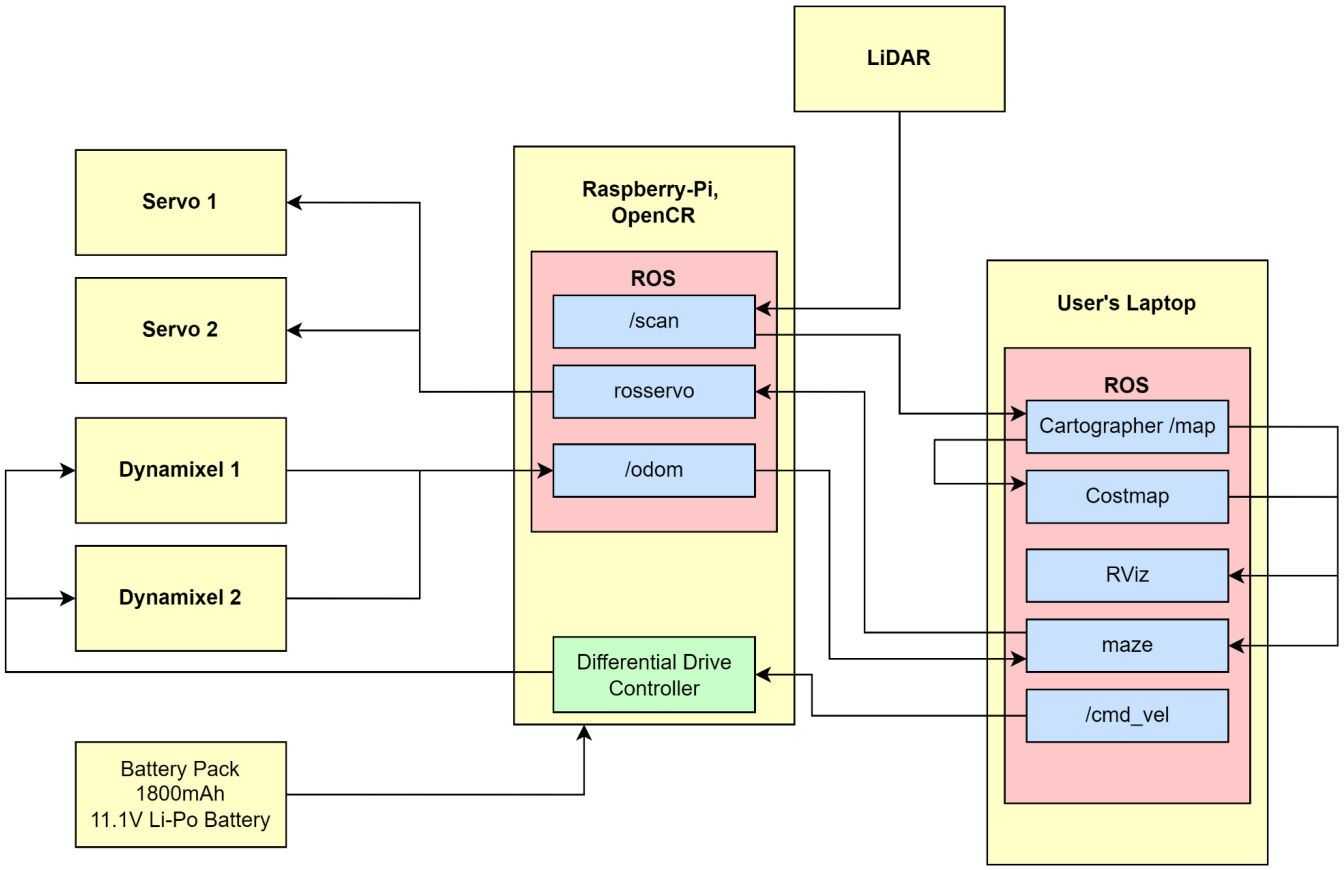




c. High Level Design – Electrical Block Diagram



d. High Level Design - Software Con-ops and Flowchart



e. Interface Control