MSS-M-2: CASE STUDY AUTONOMOUS ROBOTICS

TurtleBot3 Path Finding and Automatic Parking

PROJECT REPORT

CASE STUDY AUTONOMOUS SYSTEMS

UNDER THE GUIDANCE OF PROF. ROMAN ZASHCHITIN

Faculty of Applied Natural Science and Industrial Engineering

Deggendorf Institute of Technology, Cham

Project Submitted by:

Team 3: Deathwing

JAYDIP BORAD	12203473	jaydip.borad@stud.th-deg.de
DHARMESH PATEL	12203573	dharmesh.patel@stud.th-deg.de
MILIND	12200002	milind.milind@stud.th-deg.de
SHRESTH SHARMA	12201944	shresth.sharma@stud.th-deg.de
AAYUSH SURANA	12200044	ayush.surana@stud.th-deg.de

Team 4: Vinus

MUHAMMAD SULEMAN	12203072	muhammad.suleman@stud.th-deg.de
MILAN BUHA	12200060	Milan.buha@stud.th-deg.de
HIRALBEN GONDALIYA	22111961	Hiralben.gondaliya@stud.th-deg.de
MUSTAFA IQBAL	12200204	mustafa.iqbal@stud.th-deg.de
PARTH BOGHANI	12100023	parth.boghani@stud.th-deg.de
MUHAMMAD AZHAR MEHBOOB	12202335	muhammad.mehboob@stud.th-deg.de

Table of Contents

Introduction

This project aims to provide a general overview of the project, its purpose, and objectives. The main goal of this project is to demonstrate the capabilities of a wall follower turtlebot3 in mapping and navigating through an environment. The turtlebot3 is equipped with sensors and algorithms that allow it to follow walls and avoid obstacles. The turtlebot3 was used to map the entire environment, record the intensities of the walls, and navigate to a specific location based on the intensity of a specific wall. Python is the primary programming language used to create ROS programs in this project.

- **SLAM**

SLAM (simultaneous localization and mapping) is an advanced mapping technique that enables robots and other autonomous vehicles to create a map and localize themselves on that map simultaneously. Every SLAM system has some sort of instrument or gadget that enables a robot or other vehicle to see and measure its surroundings. Sonar, LiDAR laser scanner technology, cameras, and other image sensors can all be used for this. A SLAM system can essentially be made up of any gadget that can be used to measure physical attributes like location, distance, or speed.

- **Navigation**

Navigation is the ability of the robot to ascertain its position inside its frame of reference and then to map out a path toward a specific point. The robot or any other mobility device needs representation, such as a map of the environment, and the ability to comprehend that representation to move in its environment. The primary purpose of navigation is to move the robot as safely and smoothly as possible in a congested environment, following a safe path, and creating the shortest possible path.

- **Docker File and Gazebo**

The commands needed to construct a Docker image are listed in a text file called a Docker file. Typically, these operations include editing an existing container image through a scripting interface.

Gazebo is a 3D Robotic simulator which can effectively and correctly model robot communities in challenging indoor and outdoor conditions. It has excellent graphics and offers robot programmers an easy way to develop integration and testing for robots in controlled dynamics, stable regions, kinematic modelling and characterization.

OBJECTIVE

The main goal of this project is to develop and implement an automatic parking robot using Ros Turtlebot3 at a desired location. The code also executes Trajectory generation and wall following as they are integral part of executing automatic parking.

The objectives of this project are:

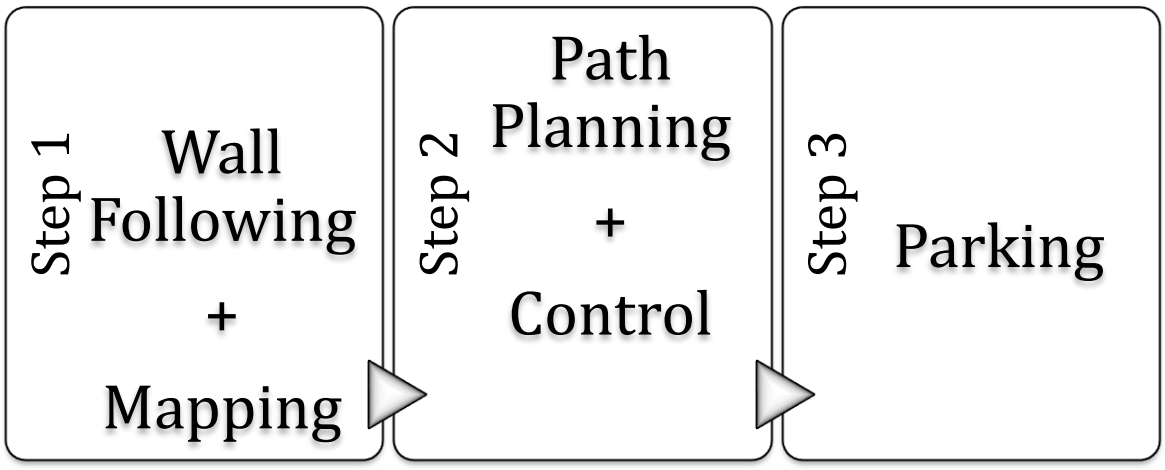
- To use a wall follower turtlebot3 to map the entire environment.
- To record the intensities of the walls
- To locate the position of the wall that has a specific intensity.
- To navigate to the specific location using two different techniques: the A* algorithm and wall following technique
- To compare and evaluate the efficiency of these techniques in allowing the turtlebot3 to reach the specific location while avoiding obstacles.

The project overview is as follows:

Initially, the turtlebot3 will be used to map the entire environment and record the intensities of the walls. After the complete area has been mapped, the turtlebot3 will return to its original position (0,0,0). Using the information gathered during the initial mapping, we will locate the position of the wall that has a specific intensity.

Methodology

The methodology for this project can be broken down into the following steps:



1. **Wall following:**
   i. In the first step, a LIDAR sensor is used to scan the environment and simultaneously get the distance and intensities of the wall with respect to Turtlebot3's current position.
   ii. Using the Turtlebot3's odometry data, save the starting position of the robot and find the nearest wall and move towards it.
   iii. When Turtlebot3 reaches the nearest wall, it stops for a moment, and saves the position then start following the left wall.
   iv. A control algorithm is used to adjust the robot's speed and direction based on the distance to the wall.
   v. Repeat steps 1 to 5 continuously to make the robot follow the wall till the robot reaches the second point.
   vi. When the Turtlebot3 reaches near to the second point, giving initial, and current position and the all possible location to park Turtlebot3.

2. **Mapping using LIDAR:**
    i. A wall-follower turtlebot3 was used to map the entire environment.
    ii. During lidar usage, the lidar sensor scans the environment and uses Turtlebot3 odometry data to generate a point cloud of the surrounding spaces.
    iii. Process the point cloud data and extract useful information, such as the location of walls, obstacles, and other features.
    iv. Use the extracted information to create a 2D occupancy grid map. This can be done by dividing the environment into a grid of cells and marking each cell as occupied or unoccupied based on whether there is an obstacle in that cell or not.
    v. Use the Matplotlib library to display the occupancy grid map and live changing map.
    vi. After that save all data in ranges.txt file to use later on for path planning.

3. **Returning to the original position:**
    i. Our main objective after completing the wall follow and mapping subtask is to reach the starting position. These were achieved by navigating towards the starting position using PI controller and A* algorithm.

4. **Locating the nearest parking spot:**
    i. Using the information gathered during mapping and all possible parking spots, locate the parking lot with a given intensity (the intensity of the black tape) which is closest to the initial position.

*

5. **Navigating to the specific location:**
    i. We employed two different techniques to navigate to a specific location. The first method used was the A* algorithm, a popular method for pathfinding and graph traversal.
    ii. The second method we used was the wall-following technique, which utilizes the information gathered during the initial mapping to follow the walls and reach the specified location.

    iii. After considering both approaches, we implement the A* Algorithm in our project. It takes the goal point from mapping and builds a path to the shortest point.

6. **Experimentation:**
    i. We performed some experiments with the Turtlebot3 using different parameters such as different intensities, different obstacle configurations, different start and end points, both in the physical environment and gazebo.
    ii. Result of both experiments are shown in the next section of report.

Code Explanations

1. Testspot.py module:

   This code appears to implement an autonomous robot navigation algorithm. The algorithm involves avoiding obstacles and following either the left or right wall. The robot has a state which changes based on conditions of its surroundings as determined by the readings from sensors (the values stored in the self.section dictionary). The states are:

   State 0: robot is resetting
   State 1: robot moves forward
   State 2: robot turns

State 3: robot starts following the right wall

State 4: robot is too close to the right wall

State 5: robot moves forward while following the right wall

State 6: robot corrects itself if it is too far from the right wall

State 7: robot stops because it is too close to an obstacle in front of it.

The robot continuously checks its surroundings using the sensor readings and updates its state accordingly. The Twist object, velocity, is used to control the

robot's movement based on its current state.**

```
def bug_action(self):
    b = 0.5  # maximum threshold distance
    a = 0.5
    c = 0.1 # minimum threshold distance
    velocity = Twist()  # Odometry call for velocity
    if self.state_ == 0 and self.follow_dir == -1:


        if self.section['front1'] < c and self.section['left1'] < c and self.section['right1'] > c:
            self.change_state(7)
            time.sleep(3)
        elif self.section['front1'] < c and self.section['left1'] > c and self.section['right1'] > c:
            self.change_state(7)
            time.sleep(3)
        if self.section['front1'] > b and self.section['left1'] > b and self.section['right1'] > b:  # Loop 1
            self.change_state(0)
            rospy.loginfo("Reset Follow_dir")
        elif self.section['front1'] < b:
            self.change_state(1)
            time.sleep(2)
            self.x = rospy.get_time()
        elif self.section['left'] < b and self.section['front'] < b and self.section['right'] > b:
            #self.change_state(7)
            print('.')
        elif self.section['front'] < b and self.section['left'] < b and self.section['right'] < b:
            #self.change_state(7)
            print('.')
    elif self.state_ == 1 and self.follow_dir == -1:
        self.change_state(1)
        if self.section['left1'] < b:
            self.change_state(2)
            y = rospy.get_time() - self.x
            #print(self.section['left1'])
    elif self.state_ == 2 and self.follow_dir == -1:
```

```
elif self.follow_dir == 1:  # Algorithm for right wall follower

    if self.state_ == 3:
        self.change_state(2)
    if self.section['right1'] < b:
        self.change_state(2)
    if self.section['right'] < a:
        self.change_state(4)
    if self.state_ == 2:
        if self.section['left'] < a:
            self.change_state(4)

    if self.section['front'] < a and self.section['left'] > b:
        self.change_state(1)
    elif self.section['right'] <= b and self.section['front1'] > b:
        self.change_state(2)
    elif self.section['right1'] <= b and self.section['front1'] <= b:
        self.change_state(1)
    elif self.section['right'] > b and self.section['front'] > b:
        self.change_state(5)
    elif self.section['right1'] < b and self.section['front1'] > b:
        self.change_state(3)
    else:
        rospy.loginfo("follow left wall is not running")
```

2. Astar.py module:

This module implements the A* algorithm to find a path between a start and goal in a 2D grid. The algorithm starts by initializing an open set, which stores the set of nodes to be evaluated. The start node is added to the open set. The algorithm then enters a loop where it selects the node in the open set with the lowest cost-plus heuristic cost, where the heuristic cost is a rough estimate of the distance from that node to the goal. The selected node is removed from the open set and added to the closed set, which stores nodes that have already been evaluated. The algorithm then expands the search to its neighbors, which are calculated using a motion model, and adds them to the open set if they are not in either the closed set or the open set. The algorithm continues until the goal node is reached or the open set is empty.

```
def planning(self, sx, sy, gx, gy):
    start_node = self.Node(self.calc_xy_index(sx, self.min_x),
                           self.calc_xy_index(sy, self.min_y), 0.0, -1)
    goal_node = self.Node-(self.calc_xy_index(gx, self.min_x),
                          self.calc_xy_index(gy, self.min_y), 0.0, -1)

    open_set, closed_set = dict(), dict()
    open_set[self.calc_grid_index(start_node)] = start_node

    while True:
        if len(open_set) == 0:
            print("Open set is empty..")
            break

        c_id = min(
            open_set,
            key=lambda o: open_set[o].cost + self.calc_heuristic(goal_node,
                                                                 open_set[
                                                                     o]))
        current = open_set[c_id]

        if current.x == goal_node.x and current.y == goal_node.y:
            print("Find goal")
            goal_node.parent_index = current.parent_index
            goal_node.cost = current.cost
            break

        # Remove the item from the open set
        del open_set[c_id]

        # Add it to the closed set
        closed_set[c_id] = current
```

A. The planning method in the code is an implementation of the A* (A-star) algorithm, a heuristic-based search algorithm used to find the shortest path between two points in a graph. The method starts by creating two sets, the open_set and the closed_set, which will store the nodes that have been considered and nodes that have been expanded, respectively. The algorithm starts by adding the start node to the open_set, and continuously performs the following steps until the goal node is found:
B. Find the node with the lowest total cost (cost so far + estimated cost to goal) in the open_set and assign it to current.
C. If the current node is the goal node, break from the loop and return the path.
D. Remove the current node from the open_set and add it to the closed_set.
E. Expand the current node by evaluating its neighbors. For each neighbor, calculate the cost so far and the estimated cost to the goal. If the neighbor has not been

expanded and is not in the open_set, add it to the open_set. If the neighbor is in the open_set, update its cost if it is lower than the previously calculated cost.

F. Repeat the process from step 1 until the goal node is found.

G. This method returns the path as a list of x and y positions of the nodes in the final path.

```
        # expand_grid search grid based on motion model
        for i, _ in enumerate(self.motion):
            node = self.Node(current.x + self.motion[i][0],
                             current.y + self.motion[i][1],
                             current.cost + self.motion[i][2], c_id)
            n_id = self.calc_grid_index(node)

            # If the node is not safe, do nothing
            if not self.verify_node(node):
                continue

            if n_id in closed_set:
                continue

            if n_id not in open_set:
                open_set[n_id] = node  # discovered a new node
            else:
                if open_set[n_id].cost > node.cost:
                    # This path is the best until now. record it
                    open_set[n_id] = node

    rx, ry = self.calc_final_path(goal_node, closed_set)
    D = 0
    t = len(rx)-1
    for i in range(len(rx)):
        if i < len(rx) - 1:
            x = rx[t - i] - rx[t - i - 1]
            y = ry[t - i] - ry[t - i - 1]
            d = sqrt((x**2) + (y**2))
            D = D + d


    return rx, ry, D
```

- "calc_final_path" is a function defined in the code snippet you provided. It calculates the final path by iterating through the parent dictionary and tracing the path from the goal node back to the start node. The function starts with the goal node and adds the parent node of the current node to the path until it reaches the start node. The final path is returned as a list of nodes in reverse order, starting from the goal node and ending at the start node.

```
def calc_final_path(self, goal_node, closed_set):
    # generate final course
    rx, ry = [self.calc_grid_position(goal_node.x, self.min_x)], [
        self.calc_grid_position(goal_node.y, self.min_y)]
    parent_index = goal_node.parent_index
    while parent_index != -1:
        n = closed_set[parent_index]
        rx.append(self.calc_grid_position(n.x, self.min_x))
        ry.append(self.calc_grid_position(n.y, self.min_y))
        parent_index = n.parent_index

    return rx, ry
```

- The calc_obstacle_map method generates a map that represents the presence or absence of obstacles in the environment. This map is then used to plan a path for the robot to reach its goal, while avoiding obstacles. It likely calculates the obstacle map based on input data such as range sensor readings, a map of the environment, or by processing images from cameras. The specific implementation details of the method would depend on the details of the robot's environment, sensors, and the desired behavior.

```
def main(sx, sy, gx, gy):
    print(__file__ + " start!!")
    file = open('ranges.txt','r')
    lines = file.readlines()
    collision = []
    for i in range(len(lines)):
        s12 = []
        s1 = lines[i]
        #s1 = lines[1]
        p = s1.split("\n")
        p = p[0].replace("[", "")
        p = p.replace("]", "")
        p = p.split(",")
        p = np.array(p)
        res = p.astype(np.float)
        c1 = res.tolist()
        collision.append(c1)

    ox, oy = [], []
    for i in range(len(collision)):
        p, q = collision[i][0], collision[i][1]
        ox.append(p)
        oy.append(q)

    grid_size = 0.05  # [m]
    robot_radius = 0.105  # [m]

    if show_animation:  # pragma: no cover
        plt.plot(ox, oy, ".k")
        plt.plot(sx, sy, "og")
        plt.plot(gx, gy, "xb")
        plt.grid(True)
        plt.axis("equal")

    a_star = AStarPlanner(ox, oy, grid_size, robot_radius)
```

- The main function in the code is the entry point of the program and coordinates the execution of all other functions. It initializes the variables and calls the calc_final_path and calc_obstacle_map functions to perform the necessary operations and produce the desired result.

Mapping module:

In our project, we use matplotlib.pyplot library to plot the x and y coordinates for obstacles, goals and walls.

- The main function is defined as ‘map’.
- Two rostopics are being used in the mapping function, one is ‘/cmd-vel’ and other one is ‘/scan’. ‘/cmd-vel’ is subscribed to get the velocity readings and ‘/scan’ is

subscribed to get the distance readings and intensity readings from the wall and obstacles.

- In below Fig, the x and y coordinates of a wall (i-e distance readings) are recorded. The laser output is in tuple and recorded in separate arrays.

```
def map(self):
    # creating empty list for cordinates
    x = []
    y = []


    # instance of class
    ld = LaserData()
    rospy.Subscriber('/scan', LaserScan, self.callback_laser)
    rospy.Subscriber("/cmd_vel", Twist, self.map1)


    plt.ion()
    fig, ax = plt.subplots()
    ax.set_xlim(-8, 8)
    ax.set_ylim(-8, 8)
    sc = ax.scatter(x, y)
    plt.draw()

    for i in range(100000):
        self.time = rospy.get_time()
        if self.time == 0.0:
            self.count += 1
        else:
            break

    self.dist = self.dist2goal
    x111 = []
    while not rospy.is_shutdown():
```

Obstacles coordinates are taken and intensity readings for the wall and goal point are recorded in below fig. Laser scan raedings are recorded as ‘msg.ranges’.

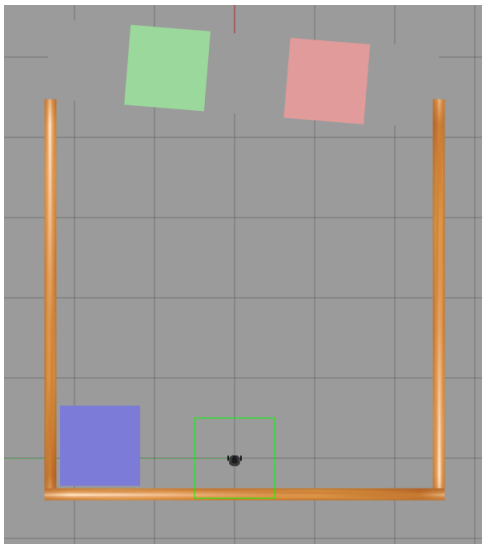
```
X = []
Y = []
x = []
y = []
msg = ld.get_laser_full()
a = msg.ranges
intensity = msg.intensities
intensities = np.asarray(intensity)
imax = intensities.max()

angles = np.arange(msg.angle_min, msg.angle_max, msg.angle_increment)
angles = np.append(angles, angles[0])

# converting tuple laser output to array
a_array = np.asarray(a)

for i in range(0, len(a_array)):

# removing inf (infinity) values from the arrays
    if a_array[i] == inf:

        pass

    else:

        r = a_array[i]

        a = angles[i] + self.cur_rot_z

        x1, y1 = r*math.cos(a), r*math.sin(a)# getting x and y cordinates of obstacle
        c = x1 + self.cur_position.x
```

In below fig, previously recorded coordinates for obstacles are obsoleted using complex numbers. This is one way of grouping the coordinate points.

- All the points are shown as ‘black stars’ in the map using the scatter and draw function which are imported from matplotlib.pyplot library. Robot position is shown as ‘red marker’. All the intensities are plotted in a separate graph.

```
    T = self.Twist
    S = T.angular.z
    if S == 0.0:
        for i in range(len(res1)):
            self.previous_list.append(res1[i])
            p.append(i)




        for i in range(len(res2)):
            self.previous_list.append(res2[i])

    coords2=[ x+1j*y for (x,y) in ruf_list] # using complex; a way for grouping
    uniques2,counts2=np.unique(coords2,return_counts=True)
    res3=[ [x.real,x.imag] for x in uniques2[counts2==1] ] # ungroup

    X = []
    Y = []
    for i in range(len(self.previous_list)):
        p, q = self.previous_list[i][0], self.previous_list[i][1]
        X.append(p)
        Y.append(q)
    x = np.append(x, X)
    y = np.append(y, Y)

    xl = x.max() + 2
    yl = y.max() + 2
    ax.cla()
    ax.set_xlim(-xl, xl)
    ax.set_ylim(-yl, yl)
```

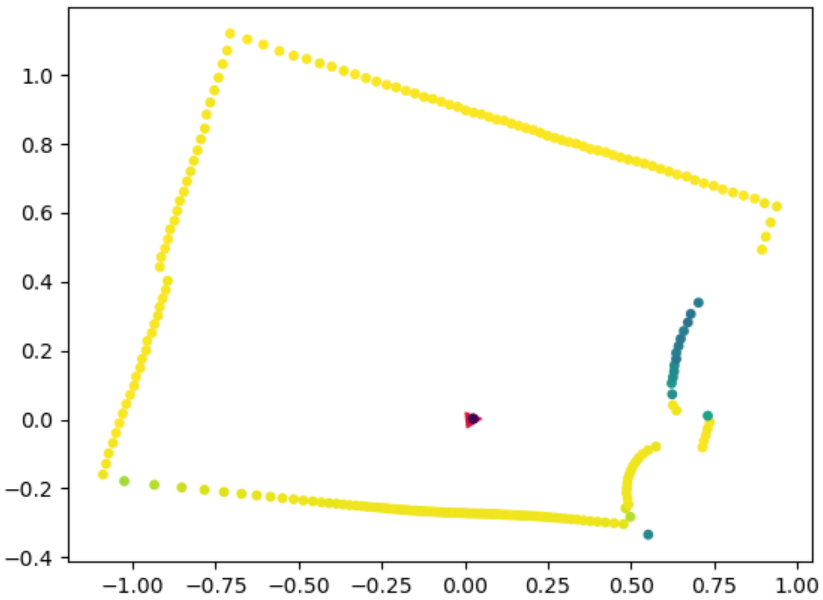

Test Environment SLAM



- Gazebo world with multiple parking arenas.

Real World Implementation

- **Laser Scan and Mapping:**



Here, Blue dots represents the location and intensity of the parking spot. Whereas, the yellow represents the wall of the environment.

Results and Analysis:

Collected Data:

i. Wall and obstacles data collected during wall-following.

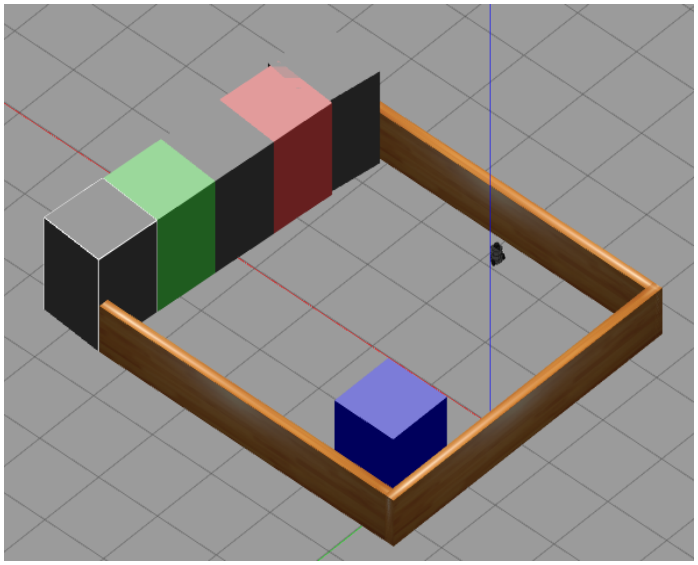
ii. Coordinate to the Initial and first closest position to the wall.

iii. All possible positions of parking spaces.

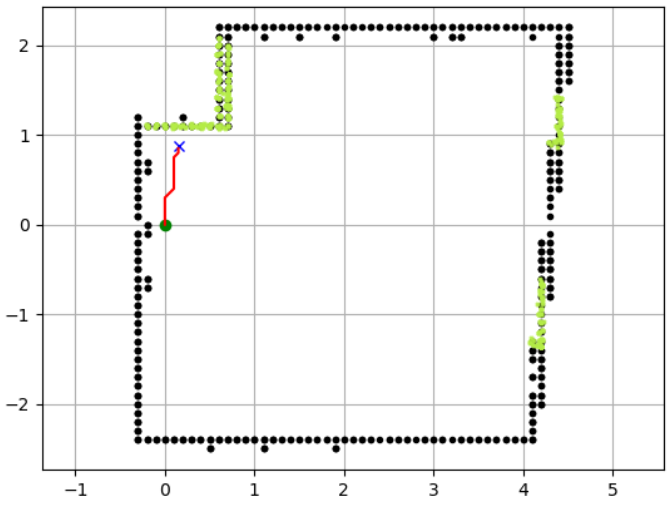
Results:

1. Results in Gazebo simulation:

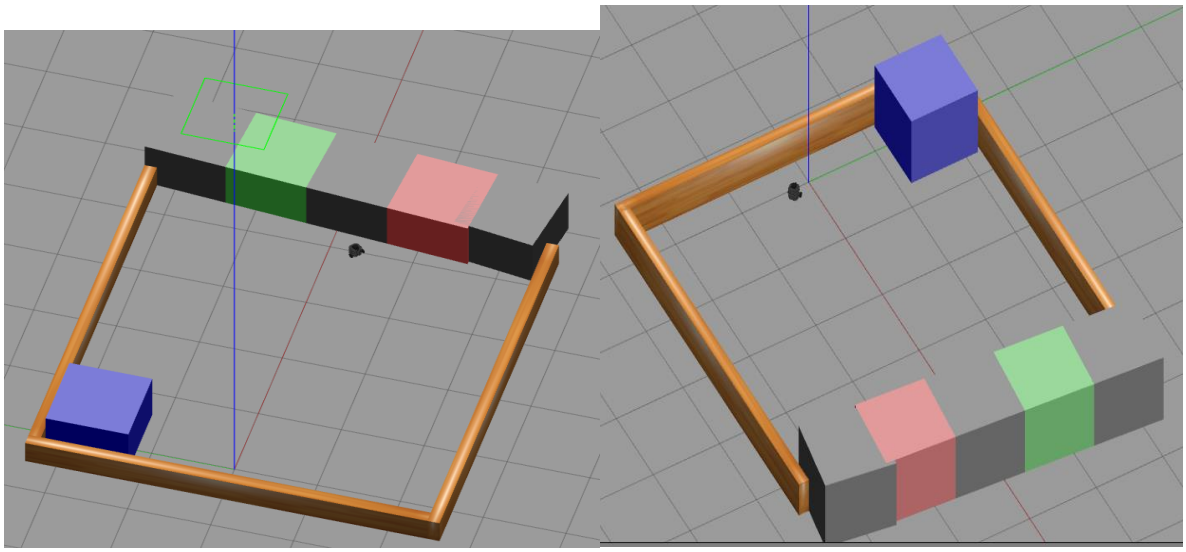
- Gazebo world with multiple parking arenas.



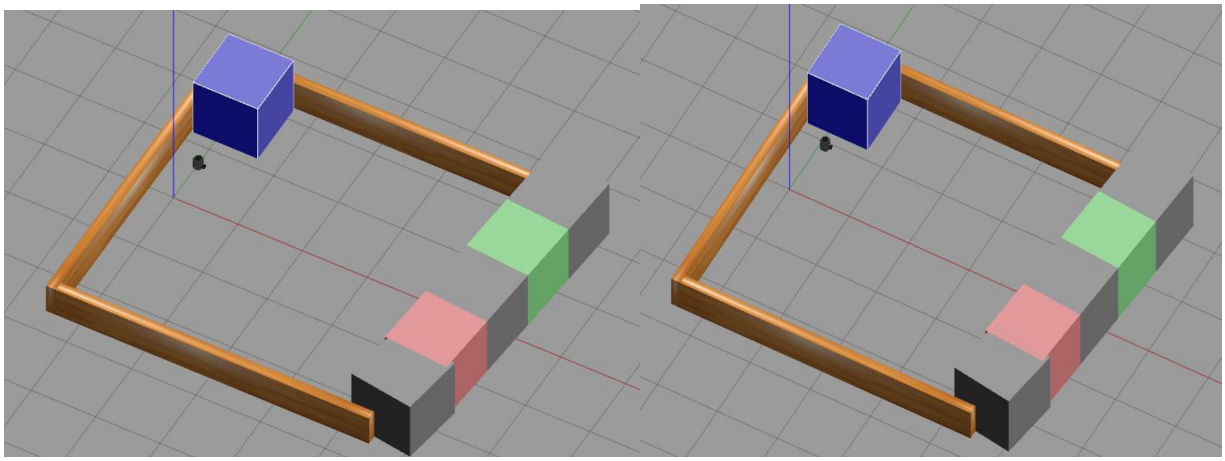
- **Map and find the nearest position and path to that position to park.**



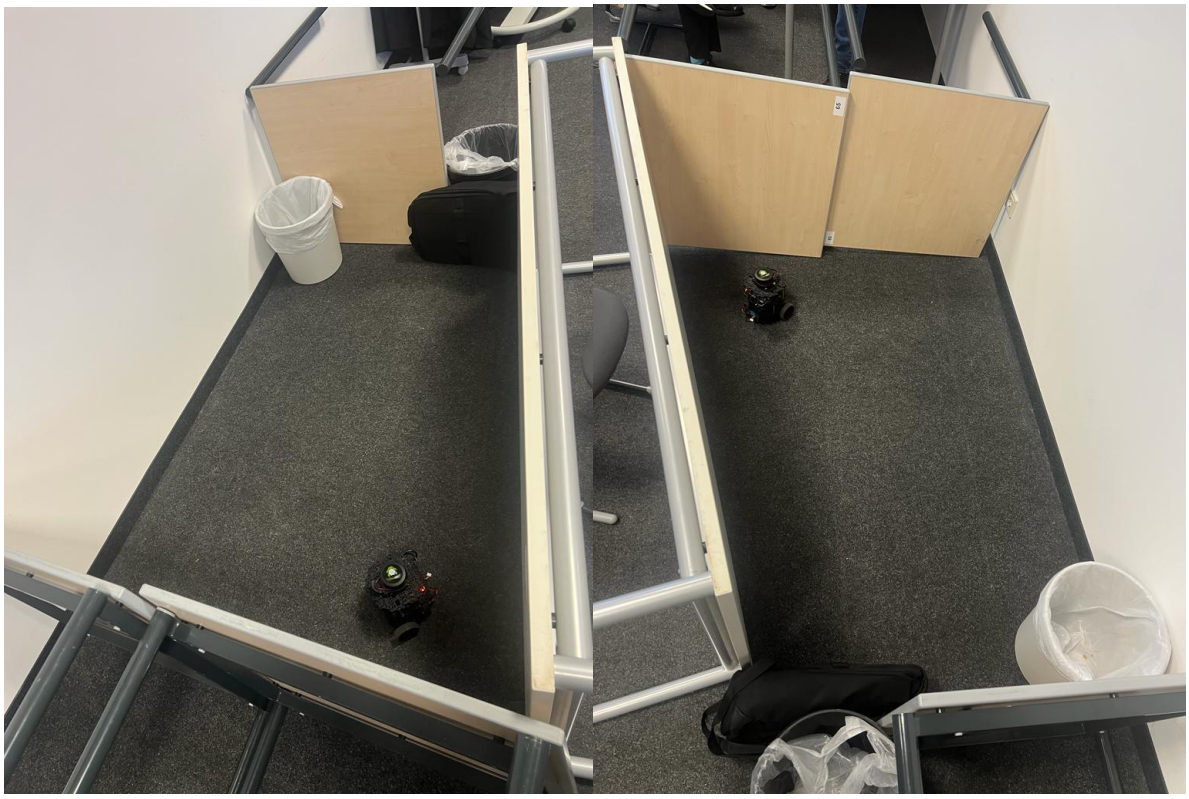
- **After completing the world, return to the starting position.**



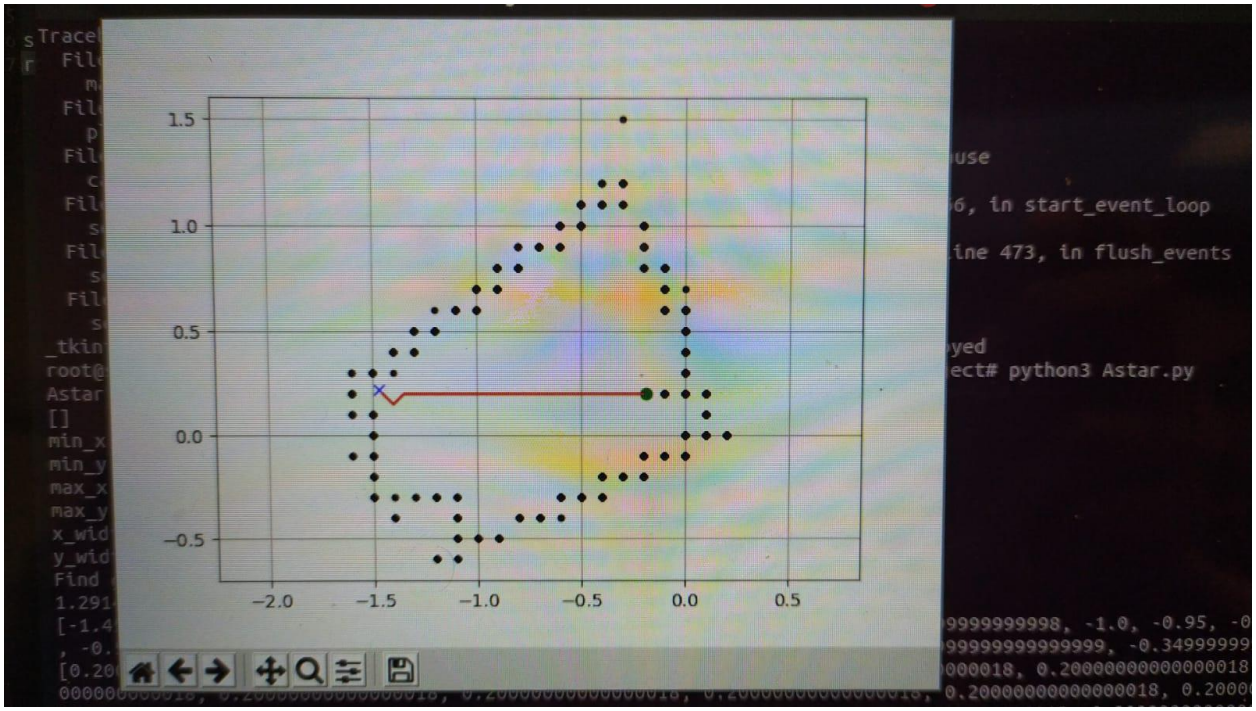
Move to the next parking spot and park the robot.



2. Results in real Environment:



Above images show the real world environment that was used for our project. Where we have implemented our code. As shown in figure black wall is used to be detected as parking spot.



Above image shows the real-world environment graph, where Turtlebot3 locate the parking position form origin position. The Red line is the path produced by the path planner code whereas the ’X’ represents the goal point.

- **Gazebo world with multiple parking arenas.**

- Map and find the nearest parking spot and direction to the spot.

- After completing the world, return to the starting position.

- Move to the next parking spot and park the robot.

Conclusion

In conclusion, this project aimed to demonstrate the capabilities of a wall follower turtlebot3 in mapping and navigating within an environment. A Turtlebot3 was used to map the entire environment, record the strength / intensities of walls, and navigate to a specific location based on the intensity of a particular wall. To achieve this, we employed two different techniques, the A* algorithm and wall-following technique.

Experiments have shown that both techniques successfully navigate the Turtlebot3 to the target location. However, the A* algorithm proved efficient in terms of time and energy consumption to reach the destination. The wall-following technique proved to be more robust in dealing with unexpected obstacles.

Our project uses multiple navigation techniques and emphasizes the importance of choosing the right technique according to the specific needs of the task. It also highlights the potential of wall follower turtlebot3 in various applications such as indoor navigation, search and rescue, and exploration.

References:

- https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/
- https://en.wikipedia.org/wiki/A*_search_algorithm
- http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html
- https://andrewyong7338.medium.com/maze-escape-with-wall-following-algorithm-170c35b88e00
- https://www.researchgate.net/figure/Hand-Wall-Follower-Algorithm_fig3_343187805