TURTLEBOT3 AUTOMATIC PARKING

Real-world and ROS simulation

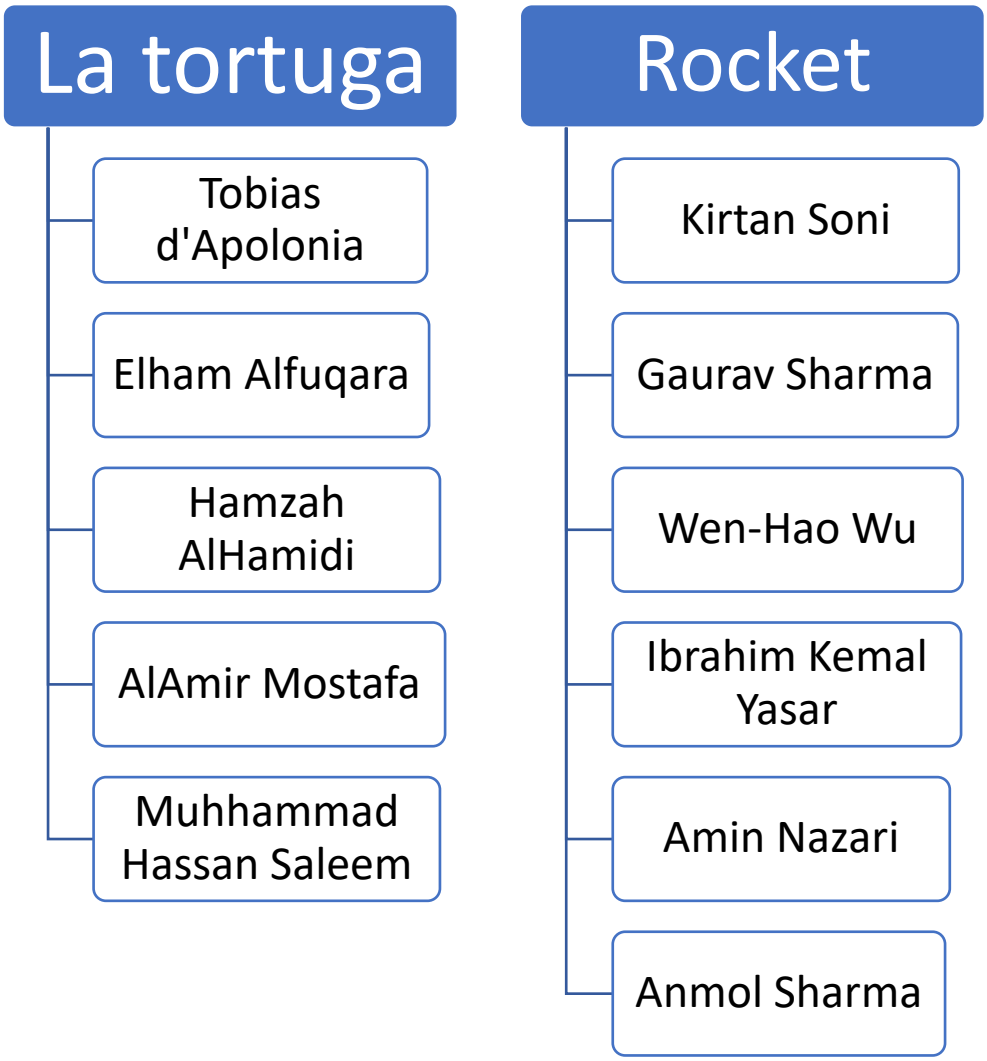
Groups Name:

- La tortuga
- Rocket

Technische Hochschule Deggendorf- Campus Cham

MSS-M-2: Case Study Autonomous Systems (WS22/23)

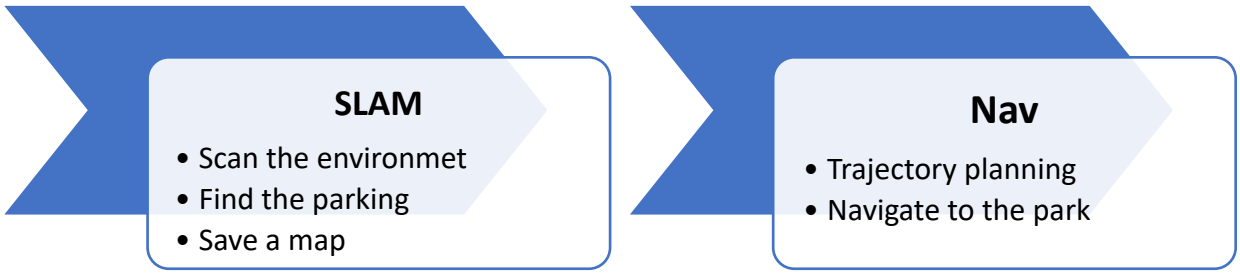
Teams Members



Contents

Project overview

The objective of this project is to implement the automatic parking for turtlebot3 using self-written python functions. The main steps needed to achieve this goal is illustrated as below:



SLAM

The SLAM (Simultaneous Localization and Mapping) is a technique used in robotics to create a map of an unknown environment while simultaneously keeping track of the robot's location within that environment. The goal of SLAM is to create a map of the environment with the goals which can be used by the Navigation to generate the trajectory.

The SLAM step achieved by using:

- LIDAR sensor: uses the laser beams to measure the distance to surrounding objects.
- Reflecting tape: used to define the goal since the reflective tape has a higher intensity which will help to differentiate the goal from the other object in the environment.
- Matplotlib: to draw the maps using the points detected by the LIDAR sensor

Navigation

The goal of the step is to find the optimal trajectory (the shortest trajectory) then go to the parking slot. For implement the trajectory planning, three different methods were studied:

- Move-base packages in ROS
- RRT and RRT*: Rapidly-exploring random tree
- I-RRT*: Informed Rapidly-exploring random tree

The table below summarize the pros and cons of each method:

The method	Pros	Cons
Move-base packages	Large scale environment Dynamin environment Global and local planning	Predefined packages by ROS- out of the project objectives Complex to set up, configure, and debug using python
RRT and RRT*	-Memory efficient -Flexible path planning -Controllable parameters	-Take more time to converge to optimal path. -Trajectory path is not optimized.

The method	Pros	Cons
I-RRT*	-Improvement over RRT* -Take less time to converge -Probabilistic node sampling. -Ellipsoidal heuristic planning -Rewire existing nodes. -Algorithm parameters are adjustable	-Sometime take more time to find first path due to random nodes.

The pictures below show the performance of RRT* and I-RRT* for a pre-generated map of Plaza environment in gazebo.

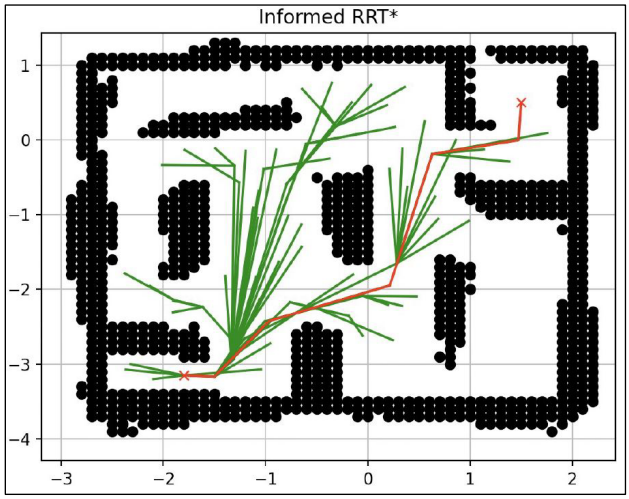


*Figure 2 Informed RRT**

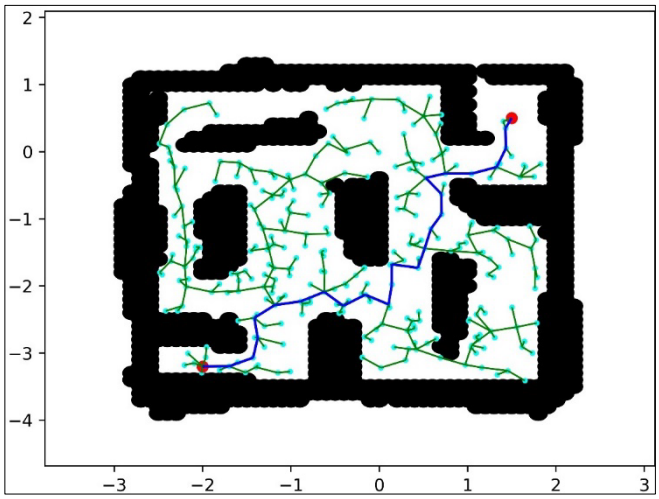


*Figure 1 RRT**

For finding the optimal trajectory, RRT* took approximately 2 minutes while I-RRT* took less than 1 minutes, subsequently I-RRT* is used for trajectory planning.

After that, the generated trajectory will be used to navigate to the goal.

Now, before moving to the project implementation, the robot specification should be reviewed.

Robot model: Turtlebot3 burger

Items	Values	Notes
Maximum translational velocity	0.22 m/s	For velocity publish
Maximum rotational velocity	2.84 rad/s (162.72 deg/s)	
Size (L x W x H)	138mm x 178mm x 192mm	
LiDAR range	360 degrees	The range is not always 360

Project Implementation

The project was divided to 5 main functions, as shown in the illustration below:

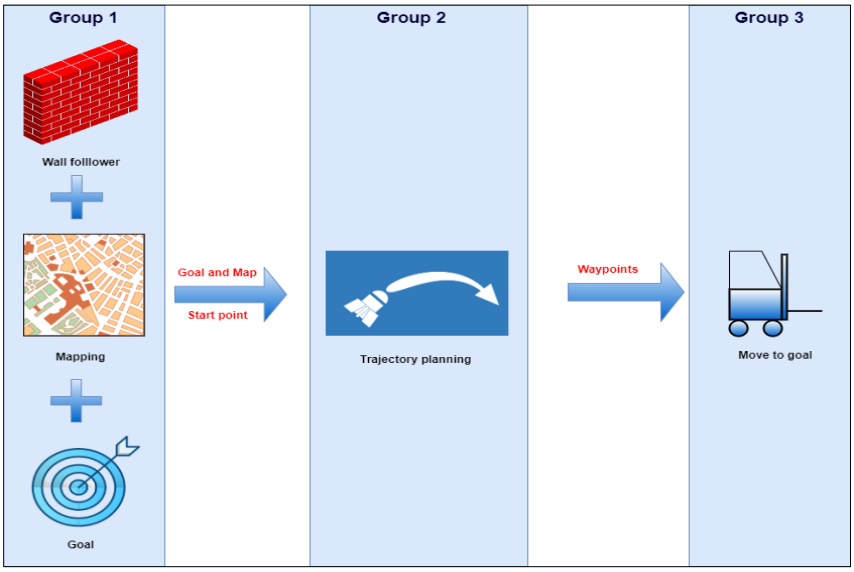


Figure 3 The project main functions

At the beginning the robot will start to follow the wall which is away to do the scanning, in parallel the LiDAR values are read and saved to create the map each point in the map has 3 attributes X,Y coordinates and intensity. Based on the intensity values the goals will be defined.

Starting point, goals and the map for the environment are the output of the first group which will be used to create the trajectory by the trajectory planning function.

Eventually, the waypoints generated by the trajectory planning will feed to the point-to-point function to move the robot towards the goal.

However, one of the biggest challenges for project implementation was reflective tape intensity. This is due to fact that both wall and reflective tape have the same intensity range. It was also difficult to implement the intensity testing in the simulation environment.

To tackle this issue, black box is used instead of the reflective tape. Now instead of finding the goals based on high intensity values, lower intensity values are considered as a goal.

The picture below shows an example for intensity plotting using black box and reflective tape.

Testing environments

As previously mentioned before, the project code was tested for debugging using gazebo simulation and real test environment.

The pictures below show these environments.

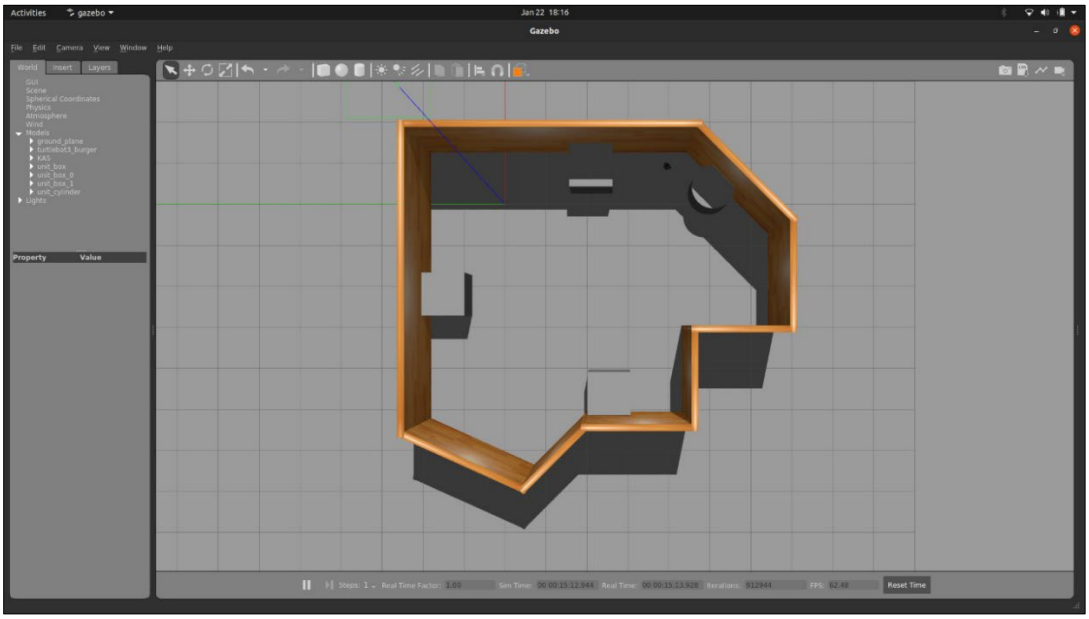


Figure 4 Gazebo environment

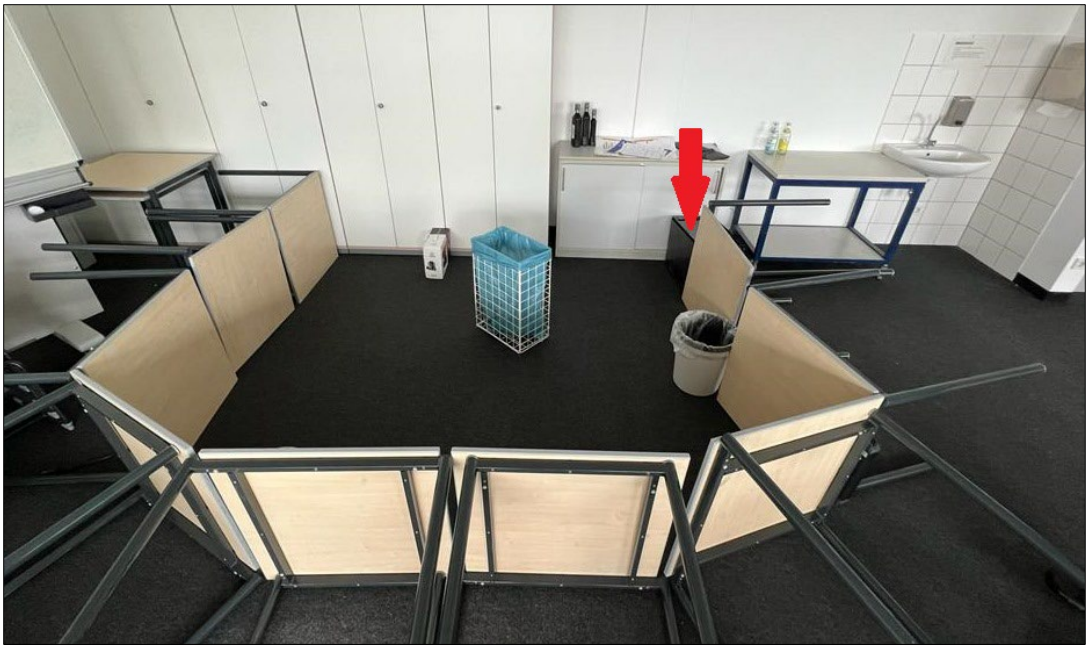


Figure 5 real testing environment- the goal is marked

The code structures

Before starting with the codes, the picture below shows the UML diagram for the main class (Parking). Please refer to appendix 1 for full scaled picture.

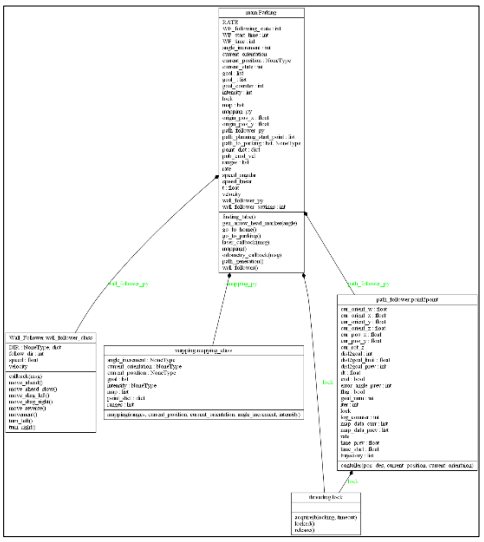


Figure 6 UML diagram for code classes

Through this report, the main functions will be highlighted while the complete code will be submitted via GitHub.

Starting first with the main states of robot journey, the variable self.current_state indicates to the current state.

- Mapping and wall_follower: self.current_state equals to 0
- Path planning: self.current_state equals to 1
- Go to goal: self.current_state equals to 2

Based on the self.current_state value the actual state will change.

```
if __name__ == "__main__":
    task= Parking()
    task.rate.sleep()
    start_time = rospy.get_time()
    task.t = 0
    while task.t < 2.0:
        task.t = rospy.get_time() - start_time
    while not rospy.is_shutdown():
        if (task.origin_pos_x == 0 or task.origin_pos_y == 0) and task.WF_following_state != 0:
            task.origin_pos_x = task.current_position.x
            task.origin_pos_y = task.current_position.y
        if task.current_state == 0:
            task.wall_follower()
            task.mapping()
            task.plot_map()
            print("Goals found: ", task.goal)
        elif task.current_state == 1:
            task.path_generation()
            # print(task.path_to_parking)
        elif task.current_state==2:
            task.go_to_parking()
            task.plot_map()
        task.rate.sleep()
```

Figure 7 changing between function statement

Wall follower

The wall follower class consists mainly these functions:

Init function: for initialize the main variables.

```
def __init__(self):
    self.speed = 0.08 #Set the speed of the robot
    self.DIR = None
    self.velocity = Twist()

    self.follow_dir = 0
    '''
    self.follow_dir = 0 -> wall finding loop
    self.follow_dir = 1 -> Follow right wall
    self.follow_dir = 2 -> Follow left wall
    '''
```

Figure 8 initialization

self.follow_dir variable represent the status of robot which will affect the robot movement.

Callback function: for adjusting the LiDAR ranges and sorting the ranges for Font_list, left_list and right_list then find the minimum range then assign to directory.

```
def callback(self,msg):
    try:
        front_min_list = []
        left_min_list = []
        right_min_list = []

        ranges = msg
        radar_len=len(ranges)

        #Sorting the ranges into direction lists
        front_list = ranges[int(345*radar_len/360):radar_len] + ranges[0:int(15*radar_len/360)]
        left_list = ranges[int(70*radar_len/360):int(90*radar_len/360)]
        right_list = ranges[int(270*radar_len/360):int(290*radar_len/360)]

        #Finding the minimum range from the direction lists
        for i in range(len(front_list)):
            if front_list[i] != 0:
                front_min_list.append(front_list[i])
            else:
                pass
        front_min = min(front_min_list)

        for i in range(len(left_list)):
            if left_list[i] != 0:
                left_min_list.append(left_list[i])
            else:
                pass
        left_min = min(left_min_list)

        for i in range(len(right_list)):
            if right_list[i] != 0:
                right_min_list.append(right_list[i])
            else:
                pass
        right_min = min(right_min_list)

        #Assigning minimum values to a dictionary
        self.DIR = {
        'left': left_min,
        'front': front_min,
        'right': right_min,
        }
```

Figure 9 callback function

The second function is movement: in this function the minimum and maximum thresholds are defined which will be used to move the robot ahead, turn right, turn left, diagonal left, and diagonal right based on certain conditions.

```
def movement(self):

    self.velocity = Twist()

    b = 0.5   # maximum threshold distance
    a = 0.4  # minimum threshold distance
```

Figure 10 setting the threshold

The conditions are shown as below:

```
if self.DIR['front'] > b and self.DIR['left'] > b and self.DIR['right'] > b:
    if self.follow_dir == 0:
        #Move forward till you find a wall
        self.move_ahead()

elif self.follow_dir == 0:
    #Setting the direction of wall to follow
    if self.DIR['left'] < b:
        #Follow left wall
        self.turn_right()
        self.follow_dir = 2
    elif self.DIR['right'] < b:
        #Follow right wall
        self.turn_left()
        self.follow_dir = 1
    else:
        if self.follow_dir == 0:
            #Go right on the very first time you find the wall
            self.turn_left()
        elif self.follow_dir == 1:
            #When you follow right wall and find right opening, go right
            self.turn_left()
        elif self.follow_dir == 2:
            #When you follow left wall and find left opening, go left
            self.move_diag_left()
```

Figure 11 wall follower conditions

```
if self.follow_dir == 2:
    #Algorithm for left wall follower
    if self.DIR['left'] >= b and self.DIR['front'] >= b and self.DIR['right'] >= b:
        self.move_diag_left()
    elif self.DIR['left'] >= b and self.DIR['front'] >= b and self.DIR['right'] <= b:
        self.move_diag_left()
    elif self.DIR['left'] >= b and self.DIR['front'] <= a and self.DIR['right'] >= b:
        self.turn_right()
    elif self.DIR['left'] >= b and self.DIR['front'] <= b and self.DIR['right'] <= b:
        self.turn_right()
    elif self.DIR['left'] <= b and self.DIR['front'] >= b and self.DIR['right'] >= b:
        self.move_ahead()
    elif self.DIR['left'] <= b and self.DIR['front'] >= b and self.DIR['right'] <= b:
        self.move_ahead()
    elif self.DIR['left'] <= b and self.DIR['front'] >= b and self.DIR['right'] <= a:
        self.move_diag_left()
    elif self.DIR['left'] <= a and self.DIR['front'] >= b and self.DIR['right'] <= b:
        self.move_diag_right()
    elif self.DIR['left'] <= b and self.DIR['front'] <= b and self.DIR['right'] >= b:
        self.turn_right()
    elif self.DIR['left'] <= b and self.DIR['front'] <= b and self.DIR['right'] <= b:
        self.turn_right()
    else:
        self.move_ahead_slow()
```

Figure 12 wall follower conditions

```
elif self.follow_dir == 1:
    #Algorithm for right wall follower
    if self.DIR['left'] >= b and self.DIR['front'] >= b and self.DIR['right'] >= b:
        self.move_diag_right()
    elif self.DIR['left'] >= b and self.DIR['front'] >= b and self.DIR['right'] <= b:
        self.move_ahead()
    elif self.DIR['left'] >= b and self.DIR['front'] <= a and self.DIR['right'] >= b:
        self.turn_left()
    elif self.DIR['left'] >= b and self.DIR['front'] <= b and self.DIR['right'] <= b:
        self.turn_left()
    elif self.DIR['left'] <= b and self.DIR['front'] >= b and self.DIR['right'] >= b:
        self.move_diag_right()
    elif self.DIR['left'] <= b and self.DIR['front'] >= b and self.DIR['right'] <= b:
        self.move_ahead()
    elif self.DIR['left'] <= a and self.DIR['front'] >= b and self.DIR['right'] <= b:
        self.move_diag_right()
    elif self.DIR['left'] <= b and self.DIR['front'] >= b and self.DIR['right'] <= a:
        self.move_diag_left()
    elif self.DIR['left'] <= b and self.DIR['front'] <= b and self.DIR['right'] >= b:
        self.turn_left()
    elif self.DIR['left'] <= b and self.DIR['front'] <= b and self.DIR['right'] <= b:
        self.turn_left()
    else:
        self.move_ahead_slow()
```

Figure 13 wall follower conditions

```
def move_diag_left(self):
    print("move d left")
    self.velocity.linear.x = self.speed
    self.velocity.angular.z = 3*self.speed

def move_diag_right(self):
    print("move d right")
    self.velocity.linear.x = self.speed
    self.velocity.angular.z = -3*self.speed

def move_ahead(self):
    print("move ahead")
    self.velocity.linear.x = 3*self.speed
    self.velocity.angular.z = 0

def turn_right(self):
    print("move right")
    self.velocity.linear.x = 0
    self.velocity.angular.z = -3*self.speed

def turn_left(self):
    print("move left")
    self.velocity.linear.x = 0
    self.velocity.angular.z = 3*self.speed

def move_ahead_slow(self):
    print("move ahead slow")
    self.velocity.linear.x = self.speed
    self.velocity.angular.z = 0

def move_reverse(self):
    print("move reverse")
    self.velocity.linear.x = -3*self.speed
    self.velocity.angular.z = 0
```

Figure 14 the robot movement functions

Based on the conditions which is basically a comparison between thresholds and Lidar reading, the robot will move and change the status. The movement done by publishing the velocities (linear and angular).

```python
def wall_follower(self):
    self.velocity, self.WF_following_state = self.wall_follower_py.callback(self.ranges)
    self.pub_cmd_vel.publish(self.velocity)
    distance2orign_x = self.origin_pos_x - self.current_position.x
    distance2orign_y = self.origin_pos_y - self.current_position.y
    distance2origin = sqrt(distance2orign_x**2 + distance2orign_y**2)
    if self.wall_follower_settings == 0:
        self.WF_start_time = rospy.get_time()
        self.wall_follower_settings = 1
    elif self.wall_follower_settings == 1:
        try:
            self.WF_time = rospy.get_time() - self.WF_start_time
        except:
            pass
        if self.WF_time > 10 and self.origin_pos_x != 0:
            if (distance2origin < 0.2 and len(self.goal) != 0 )or (self.WF_time > 120 and len(self.goal)):
                print("Robot is back to its origin")
                self.path_planning_start_point = [self.current_position.x, self.current_position.y]
                print("Stop point is: ", self.path_planning_start_point)
                self.wall_follower_settings = 2
                self.velocity.linear.x = 0
                self.velocity.angular.z = 0
                self.pub_cmd_vel.publish(self.velocity)
                self.current_state = 1
    else:
        pass
```

Figure 15 Wall follower function in the main code

The wall follower function will stop if the robot is close to the starting point and the robot find a goal or if the robot search for 120 second and find a goal.

The self.current_state will change to 1 which will lead to move to path planning stage.

Mapping and goal finding

For mapping the matplotlib.pyplot is used to plot x,y coordinated for obstacles, goal and wall and to implement this:

- Looping through the LiDar reading to convert them to x,y coordinates using the Lidar values and angle_increment
- Both x,y points were converted from body frame to global frame using the rotation matrix.
- To plot a clear map without overlapping (also for the trajectory planning), before appending the points from the previous step, the points were rounded to 1 digit then if the points are not in the list (new point), they will append.
- For detected the goal, the intensity values are used, the values should be less that 80 (the reflective tape issue was discussed before) also the goad should be detected 3 times (12 low intensity points in sequence) to be added to the goals list (to avoid errors in intensity values).
- The goal list with the map will be returned out from the mapping function to be used in trajectory planning.

```python
def mapping(self,ranges,current_position,current_orientation,angle_increment, intensity):
    self.ranges = ranges
    self.current_position = current_position
    self.current_orientation = current_orientation
    self.angle_increment = angle_increment
    self.intensity = intensity

    # initialize counter
    angle_counter = 0
    pattern_lenght_counter = 0
    pattern_start_index = 0

    # Loop through LIDAR Values, convert to X/Y Coordinates and save in List
    for count, value in enumerate(self.ranges):
        angle_counter += self.angle_increment

    # check if intensity is high, if yes increase leanght count, if no reset it.
        if len(self.intensity) == len(self.ranges):
            if self.intensity[count-1] < 80:
                pattern_lenght_counter += 1

                if pattern_start_index == 0:
                    pattern_start_index = count
            else:
                pattern_lenght_counter = 0
                pattern_start_index = 0

        if not math.isnan(value) and not math.isinf(value) and self.current_position != None:

            x = value * cos(angle_counter)
            y = value * sin(angle_counter)


            xr = (x * cos(self.current_orientation)) - (y * sin(self.current_orientation))
            yr = (x * sin(self.current_orientation)) + (y * cos(self.current_orientation))

            xr += self.current_position.x
            yr += self.current_position.y

            xr = round(xr, 1)
            yr = round(yr, 1)
```

Figure 16 Mapping function-intensity values pattern and points coordinates

```
        if pattern_lenght_counter >= 12:


            q = self.ranges[int(pattern_start_index + (pattern_lenght_counter / 2))]

            x = q * cos(angle_counter)
            y = q * sin(angle_counter)


            xr = (x * cos(self.current_orientation)) - (y * sin(self.current_orientation))
            yr = (x * sin(self.current_orientation)) + (y * cos(self.current_orientation))

            xr += self.current_position.x
            yr += self.current_position.y

            xr = round(xr, 1)
            yr = round(yr, 1)

            if self.goal_dict.get(f"{(xr,yr)}") == None:
                self.goal_dict[f"{(xr,yr)}"] = (1)
            else:
                self.goal_dict[f"{(xr,yr)}"] = (self.goal_dict.get(f"{(xr,yr)}") + 1)

            if self.goal_dict.get(f"{(xr,yr)}") > 3:

                not_aleady_detected_flag = True
                for point in self.goal:
                    if abs(xr-point[0]) < 1.0 and abs(yr-point[1]) < 1.0:
                        not_aleady_detected_flag = False
                if not_aleady_detected_flag:
                    if not math.isnan(xr) and not math.isinf(xr) and not math.isnan(yr) and not math.isinf(yr):
                        self.goal.append((xr,yr))

            pattern_lenght_counter = 0
            pattern_start_index = 0

    else:
        pass

return self.map, self.goal
```

Figure 17 goal list

Finally, it is worth mentioning that the map consists x, y and intensity values for each point.

Trajectory planning

Informed I-RRT* is used for trajectory planning, the algorithm will go through all the goal points inside the goal list and using the mapping to create the trajectory. For each goal, two paths are created to choose the optimal path (the shortest one).

```python
def path_generator(start=None,goal_points=None, maxIter=None, input_points=None,):
    print("Start informed rrt star planning")
    # create obstacles circle(x,y,radius)
    # obstacleList = [ (5, 5, 0.5), (9, 6, 1), (7, 5, 1), (1, 5, 1), (3, 6, 1), (7, 9, 1) ]
    points = np.array(input_points)
    obstacleList=[]
    for point in points:
        point=list(point)
        point.insert(2,0.2)
        obstacleList.append(point)
    #for node random value & plot area
    x_min=min([point[0] for point in points])
    y_min=min([point[1] for point in points])
    x_max=max([point[0] for point in points])
    y_max=max([point[1] for point in points])
    path_pnt_list=[]
    path_len_list=[]
    path=None
    # START PARAMETERS
    # start=[-2.3, -3.0]
    # goal_points=[[1.5, 0.8], [1.5, -1.5]]
    expandDis=0.1 #min lenght of step
    goalSampleRate=20 # rate of node generation (on goal/not on goal) (max is 100)
    maxIter=100 #numbers of iteration/nodes
    randArea=[(x_min,x_max), (y_min,y_max)] #random point generation range(min ,max)
```

Figure 18 path_generator function

Here the minimum length of steps can be adjected as well as the goal sample rate. These values were optimized for our case study. While the number of iterations is coded to be dynamic based on the algorithm needs.

```python
for goal in goal_points:
    rrt = InformedRRTStar(start=start, goal=goal, randArea=randArea, obstacleList= obstacleList, expandDis=expandDis, goalSampleRate=goalSampleRate, maxIter=maxIter)
    path ,path_len, goal_list = rrt.informed_rrt_star_search(animation=show_animation)

    # Plot and save data of path
    if path != None:
        print("Goal Path found Successfuly...!!")
        # pd.DataFrame(goal_list).to_csv("goal_list.csv", index=False) #save all possible paths in current loop in csv file
        #draw path in plot
        rrt.draw_graph()
        plt.plot([x for (x, y) in path], [y for (x, y) in path], '-r')
        plt.grid(True)
        plt.pause(2)
        plt.savefig('informed_rrt*_goal_path.png', dpi=500) #save figure
        plt.draw()
    else:
        print('Goal Path not found ..!!')
    # append path and path length in list
    path_pnt_list.append(path)
    path_len_list.append(path_len)

#return shortest path
index_min=path_len_list.index(min(path_len_list)) #index of shortest path
path_points=path_pnt_list[index_min] #shortest path

#save best path points
print('----------Best Goal length and path points:----------')
# print(path_points)
print("Path_points saved in current directory")
np.savetxt('path_points.txt', path_points, delimiter=',', fmt='%f', comments='')

return path_points
```

Figure 19 Looping through the goals, finding the shortest path for each goal and find the shortest goal waypoints

Path following and Park

The waypoints created by the trajectory planning used as input for the path_follower. PID controller is used with point-to-point method.

```python
def contoller(self, pos_des, current_position, current_orientaion):

    self.cur_pos_x = current_position.x
    self.cur_pos_y = current_position.y
    self.cur_rot_z = current_orientaion

    #while self.dist2goal > self.dist2goal_limit:
    t = rospy.get_time() - self.time_start
    self.dt = t - self.time_prev
    self.time_prev = t
    #Calcultion for distance to goal
    delta_x=pos_des[0]-self.cur_pos_x #delta x to goal
    delta_y=pos_des[1]-self.cur_pos_y #delta y to goal
    self.dist2goal=sqrt(delta_x**2+delta_y**2) #distance to goal
    angle2goal=arctan2(delta_y, delta_x) #angle to goal
    error_angle = self.cur_rot_z -angle2goal #arror angle wrt angle to goal

    #angle error correction for -pi, +pi, + and -
    if error_angle > pi:
        error_angle =(2*pi)-error_angle
    elif error_angle < -pi:
        error_angle =-(2*pi)-error_angle
    else:
        error_angle =-error_angle*2

    #calcuale delta errors of distance and angle
    diff_dist2goal=self.dist2goal-self.dist2goal_prev
    diff_error_angle=error_angle-self.error_angle_prev

    #calculate sum of distance and angle
    sum_dist2goal=self.dist2goal+self.dist2goal_prev
    sum_error_angle=error_angle+self.error_angle_prev
```

Figure 20 measuring the distance to goal and error

```python
#PID controller
# for distance
linear_kp=kp_distance*self.dist2goal #control distance for P
linear_ki=ki_distance*sum_dist2goal #control distance for I
linear_kd=kd_distance*diff_dist2goal/self.dt #contol distance for D
control_signal_distance=linear_kp+linear_ki+linear_kd #total control signal
#for angle
angular_kp=kp_angle*error_angle  #control angle for P
angular_ki=ki_angle*sum_error_angle #control angle for I
angular_kd=kd_angle*diff_error_angle/self.dt  #control angle for D
control_signal_angle=angular_kp+angular_ki+angular_kd

#publish linear and angular velocity
velocity=Twist() #calling velocity message to publish

#manual control
if error_angle < pi/25 and error_angle > -pi/25: #+-0.13
    velocity.linear.x = min(control_signal_distance, 0.45)
elif error_angle < pi/15 and error_angle > -pi/15: #+-0.21
    velocity.linear.x = min(control_signal_distance, 0.30)
elif error_angle < pi/10 and error_angle > -pi/10: #+-0.31
    velocity.linear.x = min(control_signal_distance, 0.20)
else: # > +- 0.31
    velocity.linear.x = min(control_signal_distance, 0.10)

#angular velocity limits
velocity.angular.z=control_signal_angle #net control gain for angular velocity
if velocity.angular.z > 0:
    velocity.angular.z = min(velocity.angular.z, 2.0)
else:
    velocity.angular.z = max(velocity.angular.z, -2.0)

# set angle to goal_angle and velocities to zero at goal
if self.dist2goal < self.dist2goal_limit:
    velocity.linear.x=0
    velocity.angular.z=0
else:
    pass

self.dist2goal_prev=self.dist2goal
self.error_angle_prev=error_angle

return velocity,self.dist2goal
```

Figure 21 Controller part

```python
def go_to_parking(self):
    threshold = 0.1
    self.velocity, dist2goal = task.path_follower_py.contoller(self.path_to_parking[self.goal_counter],self.current_position,self.current_orientation)
    if dist2goal < threshold:
        self.goal_counter += 1
    if len(self.path_to_parking) == self.goal_counter:
        self.velocity.linear.x = 0
        self.velocity.angular.z = 0
        self.pub_cmd_vel.publish(self.velocity)
        self.current_state +=1
        print("The robot is parked!")

    self.pub_cmd_vel.publish(self.velocity)
```

Figure 22 go_to_parking function

Code Integration

All the previous function are integrated in one main file. In this file the ROS call back functions were defined, the needed inputs are assigned for each stage and function, and the plot function is defined.

```python
def odometry_callback(self, msg):
    self.lock.acquire()
    self.current_position = msg.pose.pose.position
    cur_q= msg.pose.pose.orientation
    cur_rpy = tftr.euler_from_quaternion((cur_q.x, cur_q.y, cur_q.z, cur_q.w))  # roll pitch yaw
    self.current_orientation = cur_rpy[2]
    self.speed_linear=msg.twist.twist.linear.x
    self.speed_angular=msg.twist.twist.angular.z
    self.lock.release()

def laser_callback(self,msg):
    self.lock.acquire()
    self.ranges = msg.ranges
    self.intensity = msg.intensities
    self.angle_increment = msg.angle_increment
    self.lock.release()
```

Figure 23 call back functions

```python
def plot_map(self):
    plt.clf()
    if len(self.goal) > 0:
        for point in self.goal:
            plt.plot(point[0], point[1], "xr")
    try:
        # plt.quiver(self.current_position.x, self.current_position.y, 1, 0)
        marker, scale = self.gen_arrow_head_marker(self.current_orientation)
        markersize = 25
        plt.scatter(self.current_position.x, self.current_position.y, marker=marker, s=(markersize*scale)**2)
    except:
        pass

    plt.scatter(task.map[0],task.map[1], c =self.map[2],cmap = "Spectral", s=0.5)

    if len(self.path_to_parking) != 0:
        plt.plot([x for (x, y) in self.path_to_parking], [y for (x, y) in self.path_to_parking], '-r')
    plt.colorbar()
    plt.draw()
    plt.pause(0.01)
    plt.clf()
```

Figure 24 plotting function

Results

The attached video shows the final results of Automatic parking project. However, the pictures below show captions of the results.

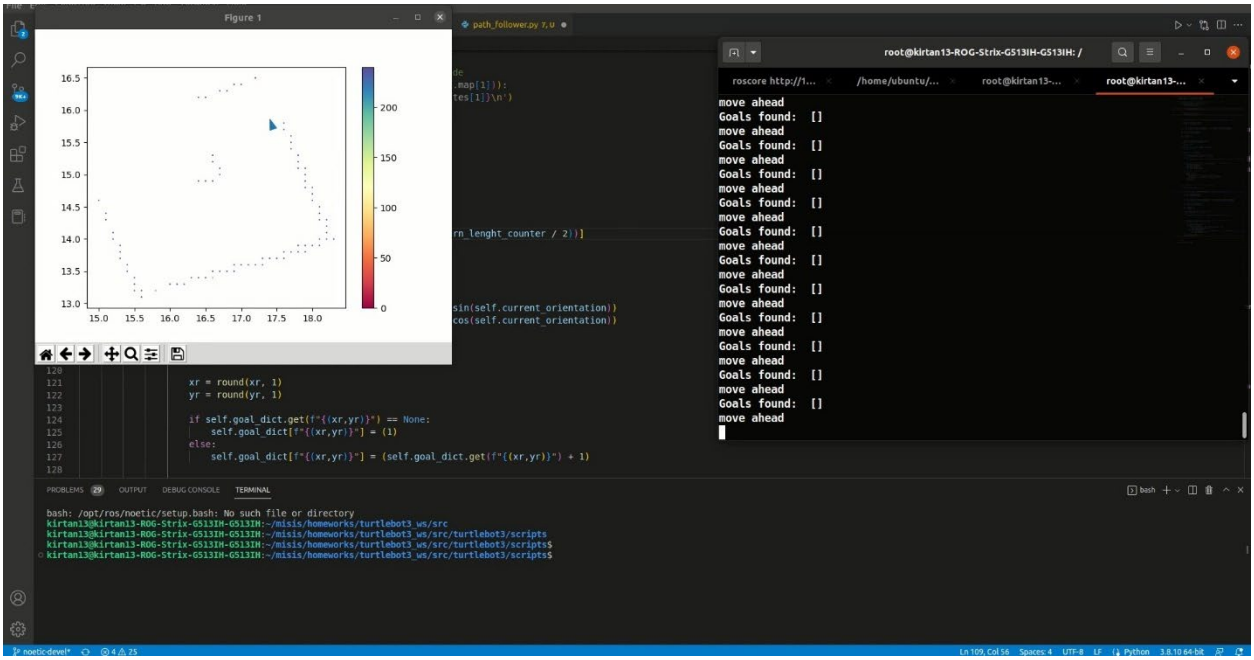


Figure 25 wall follower and mapping

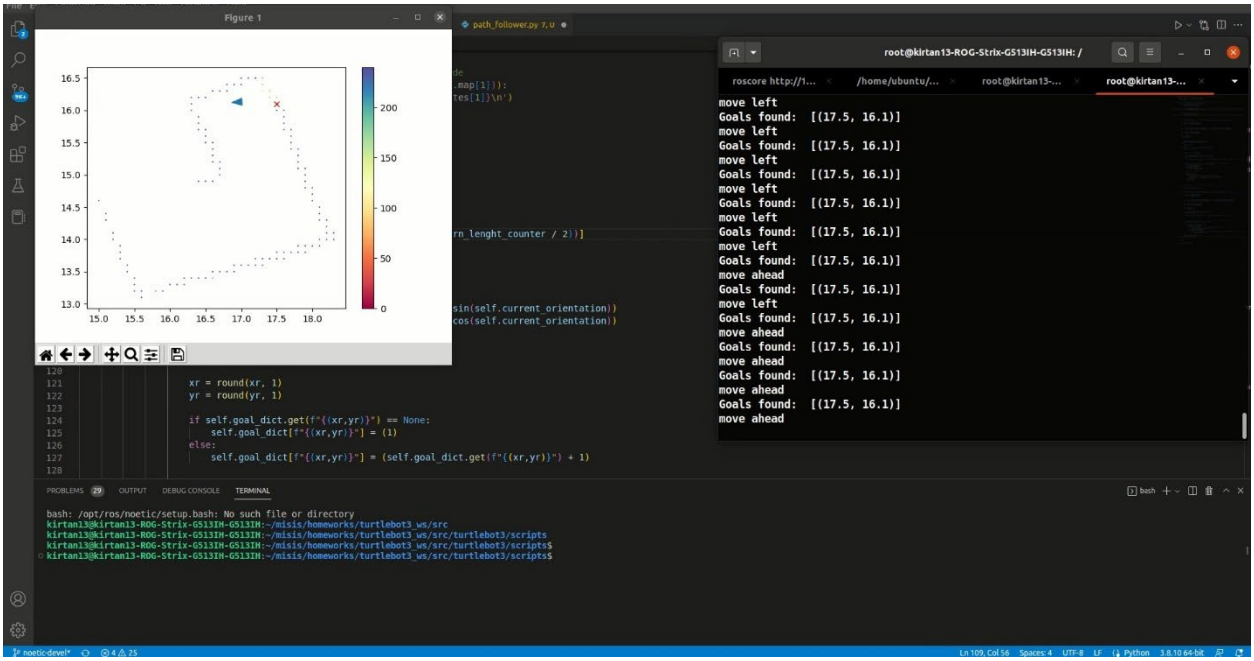


Figure 26 find a goal (x marked) the goal coordinates are printed in the terminal

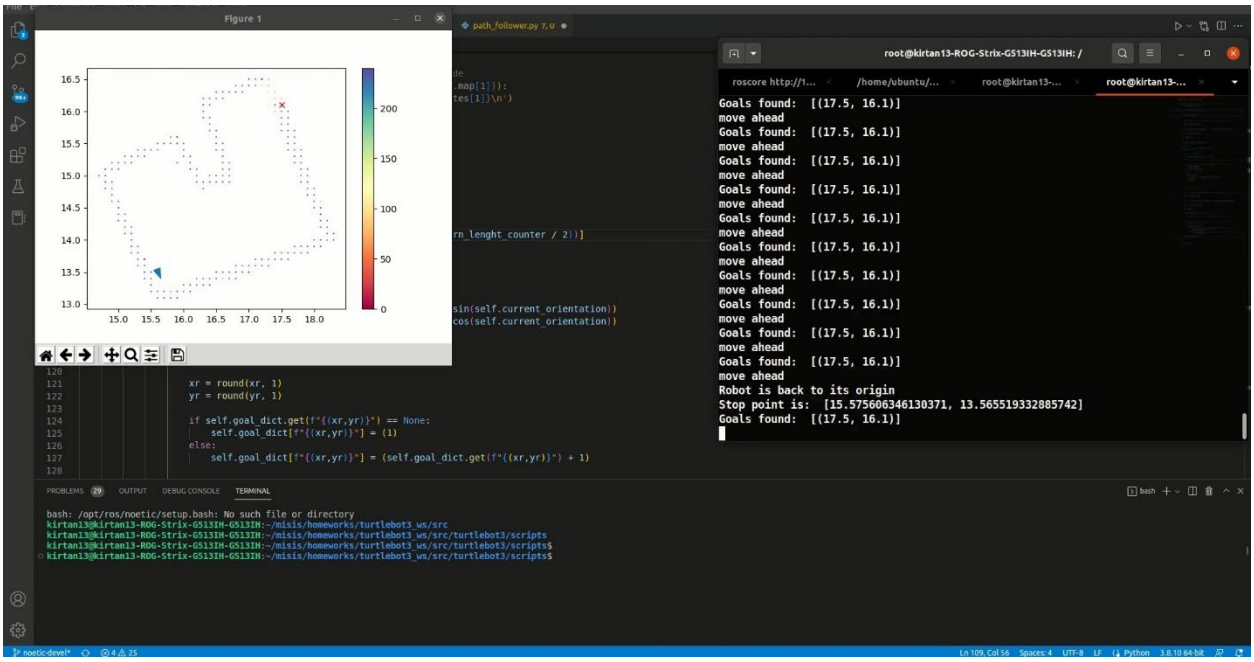


Figure 27 Back to origin (home)- see the terminal

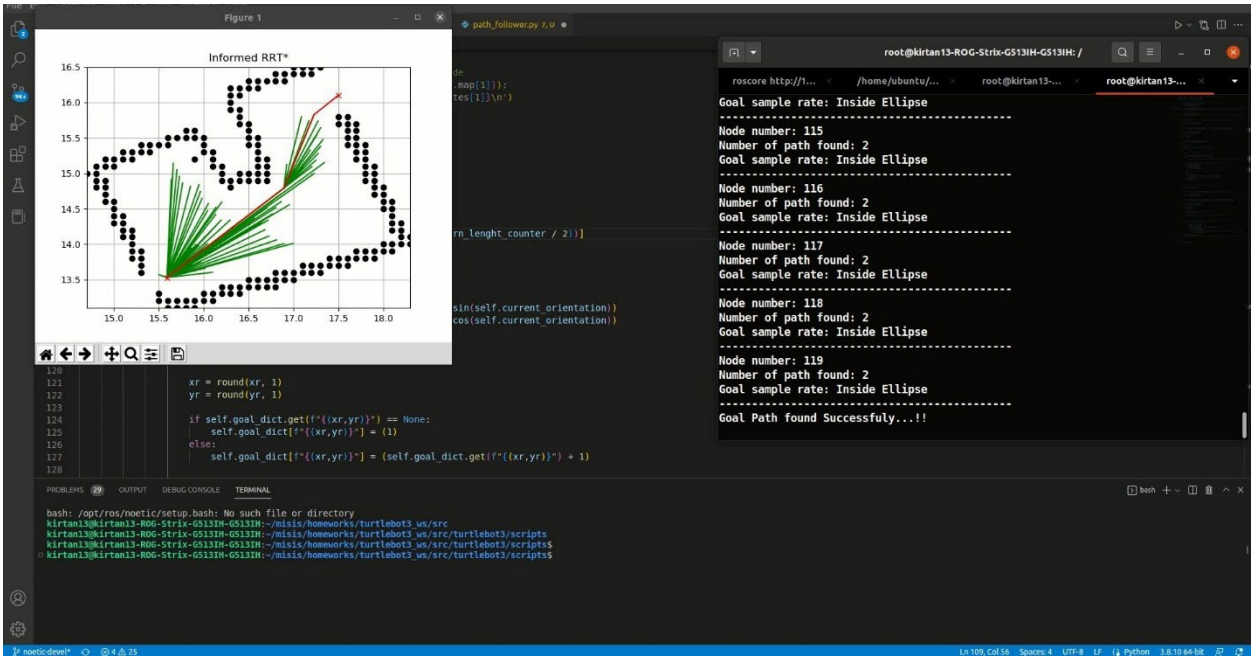


Figure 28 Trajectory planning results the shortest path found- see the terminal

The shortest path was found in 41 seconds

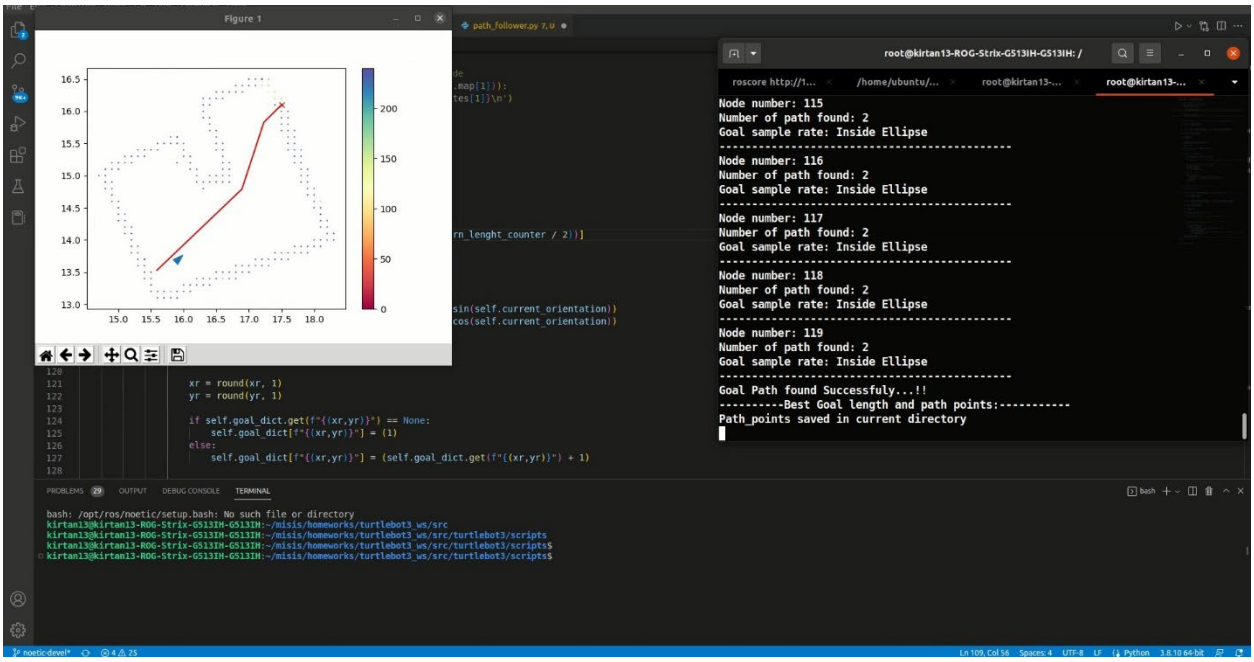


Figure 29 Following the trajectory

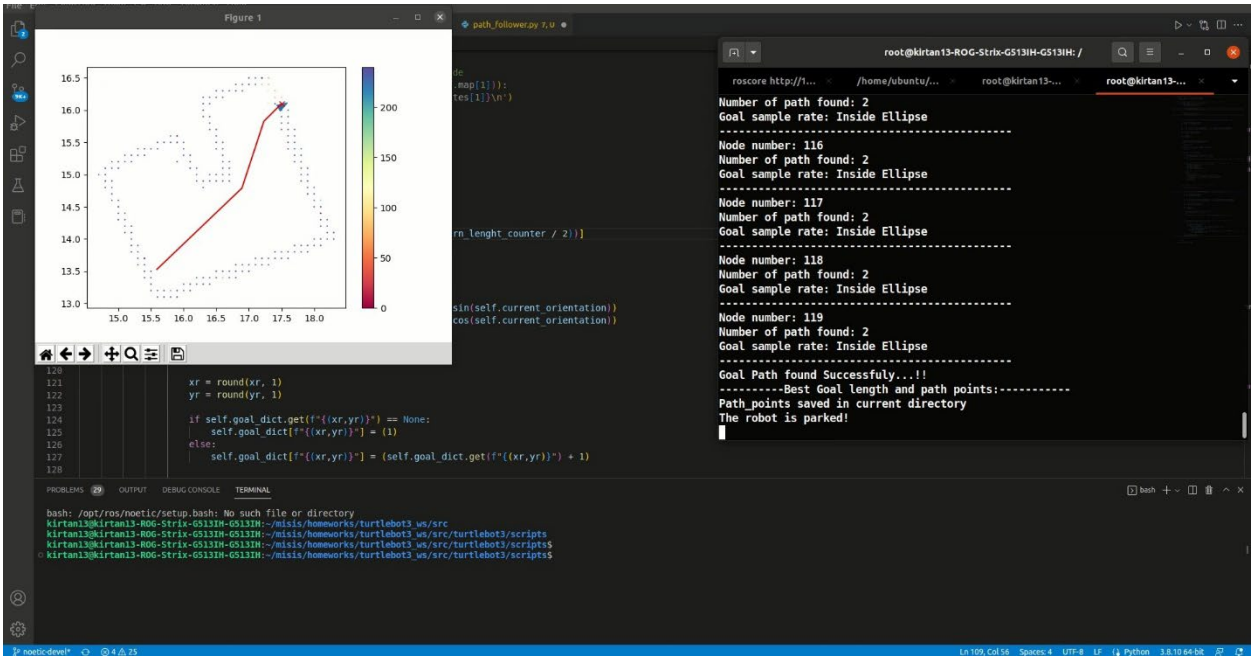


Figure 30 The robot parked!!