ELECENG 3EY4: Electrical System Integration Project

Lab09_Autonomous Driving Using Virtual Separating Barriers

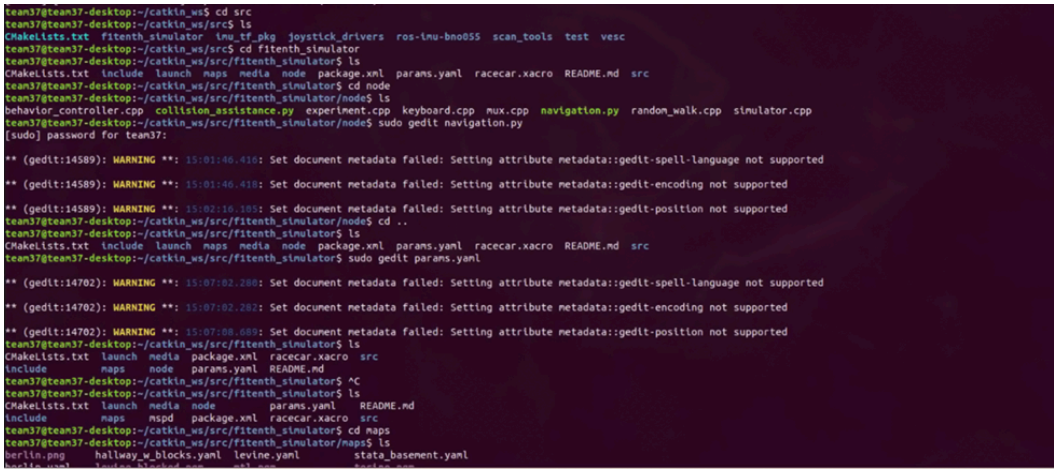
Junbo Wang – wangj430 – 400249823

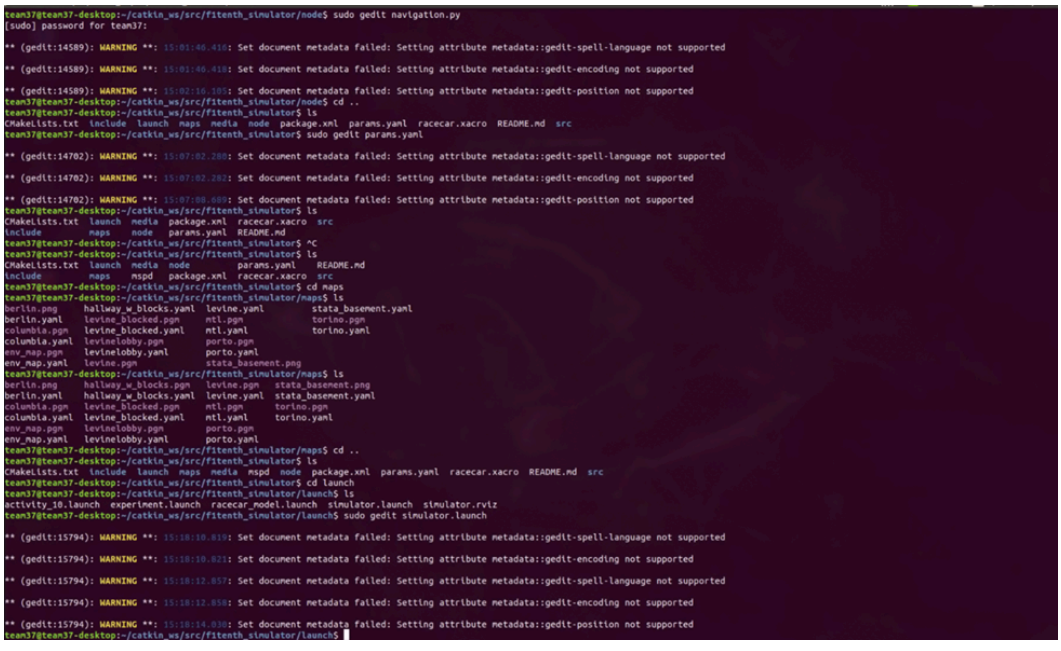
Preet Batra - batrap5 - 400341704

Yichen Lu - luy191 - 400247938

Activity 1: You have been provided with the incomplete file "navigation_lab9_inc.py". Copy the content of this file into your "navigation.py". Also, update your local copy of the file "param.yaml" using the one posted on Avenue. Make sure that you retain the calibrated VESEC parameters from Lab 6 and the value of "scan_beams" that corresponds to the LiDAR in BOOST mode on your AEV. Also, copy the files "hallway_w_blocks.yaml" and "hallway_w_blocks.pgm" from Avenue to the "maps" subdirectory of the f1tenth_simulator package. Modify the file "simulator.launch" accordingly to load this map for the simulations.

For this activity, we just followed the steps in the lab manual. The steps are shown below.





Activity 2: Study the files “navigation_lab9_inc.py” and “params.yaml” and identify parts of the code that accomplish this step. Step 1: Define a Field of View (FOV) in front of the vehicle. This is where we will look for potential obstacles.

After studying the files “navigation_lab9_inc.py” and “params.yaml”, we identify parts of the code that accomplish this step. The screenshots are shown below.

The relevant parameters from the "params.yaml" file

```
safe_distance: 0.5
right_beam_angle: rad(2*pi*4.0/16)
left_beam_angle: rad(2*pi*12.0/16)
```

```
# Lidar simulation parameters
scan_beams: 720
scan_field_of_view: 6.2831853 #4.71 # radians
scan_range: 12.0
```

In the “params.yaml” file, the field of view (FOV) is defined by some parameters that specify the angles and the number of points used to detect obstacles for both the right side and left side of the vehicle.

For example, “scan_beams” defines the number of data points in a full LiDAR scan. Additionally, “scan_field_of_view” defines the angular range of the LiDAR scan.. Morever, “right_beam_angle” and “left_beam_angle " define the angular range that detects the obstacles on the right and left sides of the vehicle.

The relevant codes in the "Navigation.py" file:

```
self.ls_ang_inc=2*math.pi/self.scan_beams

# Define field of view (FOV) to search for obstacles

self.ls_str=int(round(self.scan_beams*self.right_beam_angle/(2*math.pi)))
self.ls_end=int(round(self.scan_beams*self.left_beam_angle/(2*math.pi)))
self.ls_len_mod=self.ls_end-self.ls_str+1
self.ls_fov=self.ls_len_mod*self.ls_ang_inc
self.angle_cen=self.ls_fov/2
self.ls_len_mod2=0
self.ls_data=[]

self.drive_state="normal"
self.stopped_time =0.0
self.yaw0 =0.0
self.dtheta = 0.0
self.yaw = 0.0
t = rospy.Time.from_sec(time.time())
self.current_time = t.to_sec()
self.prev_time = self.current_time
self.time_ref = 0.0
self.wl0 = np.array([0.0, -1.0])
self.wr0 = np.array([0.0, 1.0])
```

In the “Navigation.py” file, the field of view (FOV) is defined by the above codes. Firstly, the code calculates the start (self.ls_str) and end (self.ls_end) indices for the LiDAR data points that fall within the FOV. In addition, it also calculates the angular increment (self.ls_ang_inc) between consecutive beams of the LiDAR scan, the modified length of the FOV (self.ls_len_mod), the total field of view (self.ls_fov), and the central angle of the FOV (self.angle_cen). More specifically, these values are used to detect the range of the LiDAR scan that contains obstacle points. In summary, this setup is extremely important for the navigation of the car since it determines the potential obstacle points that need to be avoided. The vehicle uses this FOV to process LiDAR data in order to identify safe paths and make driving decisions.

Activity 3: Explain the role of function “preprocess_lidar” and how it helps accomplish the objective of Step 2. Step 2: Set to zero the range values for all LiDAR returns that fall within a safe distance in FOV in front of the vehicle. The zero range values represent immediate obstacles within the safety zone and non-zero LiDAR returns are considered free space.

```python
# Pre-process LiDAR data
def preprocess_lidar(self, ranges):

    data=[]
    data2=[]

    for i in range(self.ls_len_mod):
        if ranges[self.ls_str+i]<=self.safe_distance:
           data.append ([0,i*self.ls_ang_inc-self.angle_cen])
        elif ranges[self.ls_str+i]<=self.max_lidar_range:
            data.append ([ranges[self.ls_str+i],i*self.ls_ang_inc-self.angle_cen])
        else:
            data.append ([self.max_lidar_range,i*self.ls_ang_inc-self.angle_cen])

    k1 = 100
    k2 = 40

    for i in range(k1):

        s_range = 0

        for j in range(k2):
            index1 = int(i*len(ranges)/k1+j)
            if index1 >= len(ranges):
                index1 = index1-len(ranges)

            index2 = int(i*len(ranges)/k1-j)
            if index2 < 0:
                index2= index2 + len(ranges)

            s_range = s_range + min(ranges[index1],self.max_lidar_range)+ min(ranges[index2],self.max_lidar_range)

        data2.append(s_range)


    return np.array(data), np.array(data2)
```

The function “preprocess_lidar” is to preprocess the raw LiDAR data. More specifically, it includes establishing a safe distance in front og the vehicle. Additionally, it also assigns a value of zeros to any LiDAR data that fall within this safety margin. All in all, this function helps the vehicle to navigate and avoid obstacles.

In terms of helping accomplish the objective of Step 2, the function firstly initializes two empty lists, data and data2. These two empty lists will hold the processed LiDAR data and the sums of ranges over a window respectively.

Afterwards, the function iterates over the LiDAR ranges within the vehicle's field of view (FOV), which is determined by the “self.ls_len_mod” parameter. As for different ranges in the FOV, there are many different cases.

For instance, if the distance of the obstacle is less than or equal to “self.safe_distance”, a value of [0, angle] will be appended to the data list. Specifically, this angle is the angle of the LiDAR beam with respect to the center of the FOC. This case will be seen as an immediate obstacle.

Additionally, if the distance of the obstacle is within the operational range of the LiDAR, the distance and angle will be included in the data list.

What is more, if there is no obstacle detected, then the maximum range and angle will be included in the list.

Next, two nested loops are used to calculate the sum of the minimum range values for each pair of indices in a window. Afterwards, the sums are appended to the data2 list, which can provide a broader overview of the space surrounding the vehicle within the LiDAR's FOV.

Finally, the function returns two NumPy arrays. One is for the processed LiDAR data with immediate obstacles set to zero, which is in the data list. Another is for the sum of ranges over the window, which is in the data2 list.

Activity 4: Complete the missing parts in the function “find_max_gap” to achieve the objective of Step 3. Step 3: Find the longest sequence of consecutive non-zero LiDAR returns among the free-space points; this represents the largest gap in front of the vehicle.

```
# Return the start and end indices of the maximum gap in free_space_ranges
def find_max_gap(self, proc_ranges):

    j=0
    str_indx=0;end_indx=0
    str_indx2=0;end_indx2=0

    range_sum = 0
    range_sum_new = 0

    for i in range (self.ls_len_mod):

        if proc_ranges[i,0]!=0:
            if j==0:
                str_indx=i
                range_sum_new = 0
                j=1
            range_sum_new = range_sum_new + proc_ranges[i,0]
            end_indx=i

        if  j==1 and (proc_ranges[i,0]==0 or i==self.ls_len_mod-1):

            j=0

            if  range_sum_new > range_sum:
                    end_indx2= end_indx
                    str_indx2= str_indx
                    range_sum = range_sum_new

    return str_indx2, end_indx2
```

Activity 5: Complete the code for the function “find_best_point” to compute the desired direction of movement according to the formulation in (2). Explain why this might be a better choice than the furthest point in the largest gap in (1).

```
# start_i & end_i are the start and end indices of max-gap range, respectively
# Returns index of best (furthest point) in ranges
def find_best_point(self, start_i, end_i, proc_ranges):

    range_sum = 0
    best_heading = 0

    for i in range (start_i, end_i+1):

        range_sum = range_sum + proc_ranges[i,0]

        best_heading = best_heading + proc_ranges[i,0]*proc_ranges[i,1]

    if range_sum!=0:
        best_heading=best_heading/range_sum

    return best_heading
```

The purpose of the "find_best_point" function is to detect the direction which represents the longest clear distance. With respect to the second method, it is easily used to detect the least obstructed path. Therefore, this algorithm works perfectly while trying to avoid some specific type of obstacle, such as walls. As for the first method, it focuses on identifying the widest gaps between barriers, which is helpful to determine the largest open area. As a result, when we tested our vehicle in the hallway, the second method was a better choice.

Activity 6: Derive the optimization formulation in (7).

Formulation of Inequality Constrained Quadratic Program

The goal here is to find a line with the *greatest* distance from the vehicle that would separate it from all LiDAR returns that fall between two pre-defined beam angles. Consider the equation of a line in the 2D $x - y$ space of the “base_link” frame in Fig. 1:

$$\bar{w}^T p + b = 0 \tag{3}$$

Here $p \in \mathbb{R}^2$ represents the coordinates of any point on the line with respect to the “base_link” frame, $\bar{w} \in \mathbb{R}^2, b \in \mathbb{R}$ are a constant vector and scalar that parameterize the line. The separating line we are trying to find would not pass through the origin of the “base_link” frame, hence, $b \neq 0$. We divide both sides of the equation by non-zero b to obtain:

$$w^T p + 1 = 0 \tag{4}$$

with $w \in \mathbb{R}^2$ is an unknown vector that parameterizes the line. We must find the unknown vector w corresponding to the line has the *largest* distance from the *origin*, while keeping all obstacle-related LiDAR returns on the other side of the line. The distance of some arbitrary point p_0 from the line is given by

$$d = \frac{|w^T p_0 + 1|}{\sqrt{w^T w}} \tag{5}$$

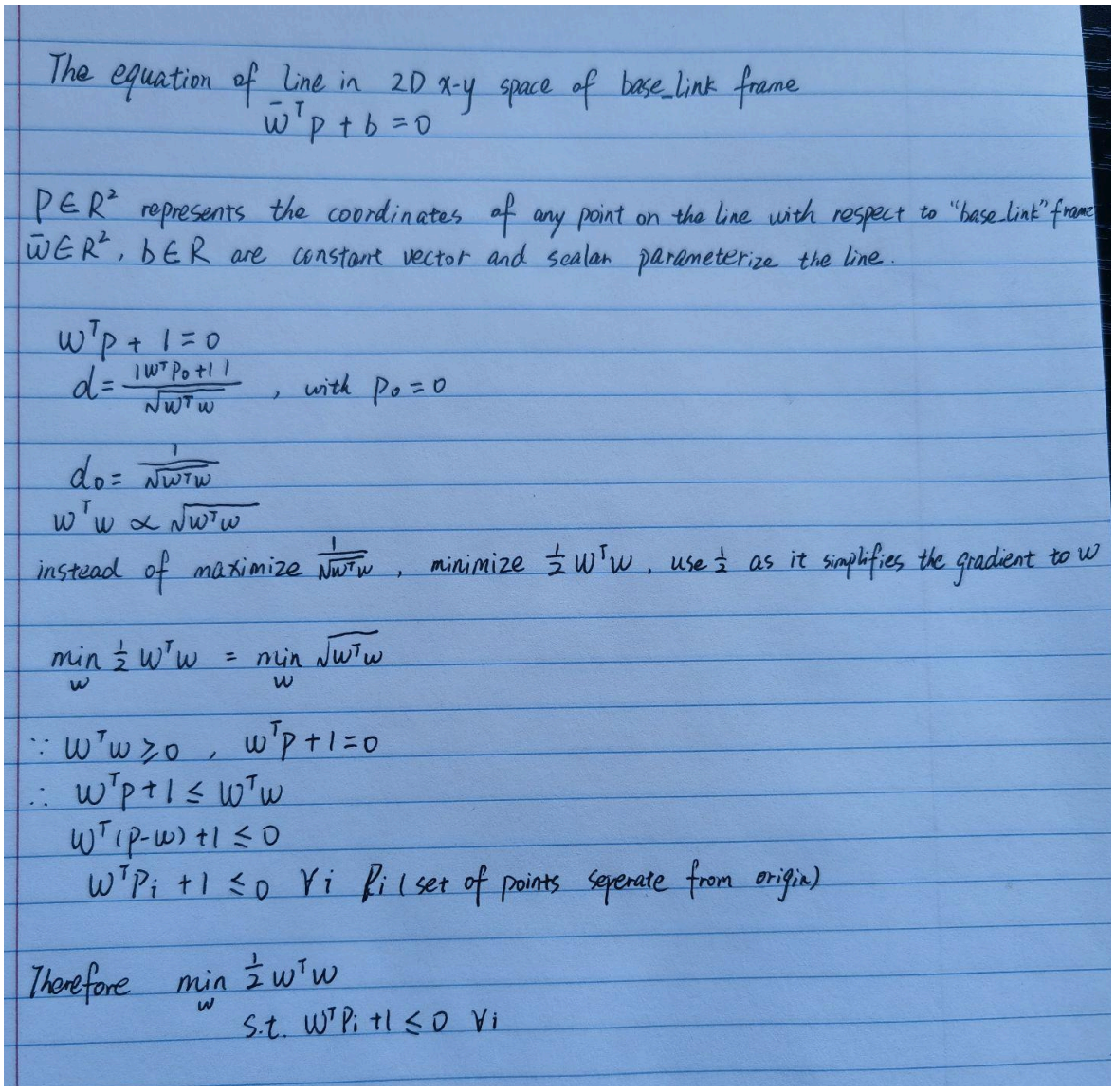
Therefore, the distance of the origin of the "base_link" frame to the line, with $p_0 = 0$ can be written as

$$d_0 = \frac{1}{\sqrt{w^T w}} \tag{6}$$

The solution to the following optimization problem characterizes a line with the largest distance to the origin that separates a set of points $\{p_i\}$ from the origin:

$$\begin{aligned} &\min_{w} \ \frac{1}{2} w^T w \\ &s.t. \\ &w^T p_i + 1 \leq 0 \quad \forall i \end{aligned} \tag{7}$$

Detrivation:



Activity 7: Explain how the value of 0≤α≤1 affects the solution to the optimization problem in (8). Consider what happens as α changes between the two extreme values of α=0 and α=1.

$$\min_{w_k} \quad \frac{1}{2}\alpha w_k^T w_k + \frac{1}{2}(1-\alpha)(w_k - w_{k-1})^T(w_k - w_{k-1}) \tag{8}$$
$$s.t.$$
$$w_k^T p_i + 1 \leq 0 \qquad \forall i$$

The parameter α controls the relative weight between w_k and $w\_k - w\_\{k-1\}$. w_k corresponds to maximizing the distance from the separation line to the origin, and $w\_k - w\_\{k-1\}$ represents the change in the separation line between two consecutive time steps.

When α=0, the optimization problem formula reduces to ½ * $(w\_k - w\_\{k-1\})$^T * $(w\_k - w\_\{k-1\})$. In this case, it represents that the optimization problem solution only focuses on minimizing the change in the separation line between consecutive time steps. It does not consider the distance from the divider to the origin and can be used in environments where the direction of obstacles remains consistent, such as when the vehicle travels on a smooth path without suddenly changing direction.

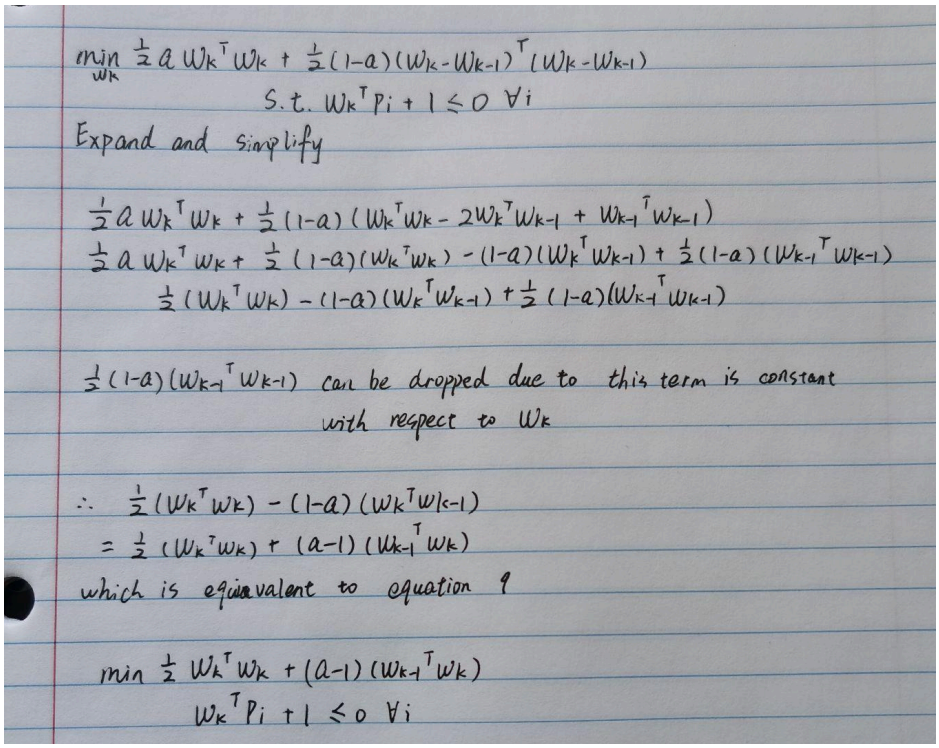
When α=1, the optimization problem formula simplifies to $1/2 * w\_k$^$T * w\_k$. In this case, the optimization problem aims to maximize the distance of the dividing line from the origin, no matter how much the dividing line changes between successive time steps. While the main concern is to maximize the margin between the vehicle and the obstacle, this puts more emphasis on maintaining a safe distance between the vehicle and the obstacle, even if the obstacle's direction of motion may suddenly change.

For 0<α<1: As α increases from 0 to 1, the optimization starts to focus more on the distance of the divider from the origin, while still taking into account the smoothness of the divider change. By adjusting α, a balance can be found between the two extremes, which can be adjusted to the specific requirements of the vehicle's operating environment.

Activity 8: Show that the following optimization problem has the same solution as the one in (8). We call these two optimization problems equivalent.

$$\min_{w_k} \quad \frac{1}{2}\alpha w_k^T w_k + \frac{1}{2}(1-\alpha)(w_k - w_{k-1})^T(w_k - w_{k-1}) \tag{8}$$
$$s.t.$$
$$w_k^T p_i + 1 \leq 0 \qquad \forall i$$

$$\min_{w_k} \quad \frac{1}{2} w_k^T w_k + (\alpha - 1) w_{k-1}^T w_k \tag{9}$$
$$s.t.$$
$$w_k^T p_i + 1 \leq 0 \qquad \forall i$$



$\min_{w_k} \frac{1}{2} a\, w_k^T w_k + \frac{1}{2}(1-a)(w_k - w_{k-1})^T (w_k - w_{k-1})$

$S.t.\ w_k^T p_i + 1 \leq 0 \quad \forall i$

Expand and simplify

$\frac{1}{2} a\, w_k^T w_k + \frac{1}{2}(1-a)(w_k^T w_k - 2 w_k^T w_{k-1} + w_{k-1}^T w_{k-1})$

$\frac{1}{2} a\, w_k^T w_k + \frac{1}{2}(1-a)(w_k^T w_k) - (1-a)(w_k^T w_{k-1}) + \frac{1}{2}(1-a)(w_{k-1}^T w_{k-1})$

$\frac{1}{2}(w_k^T w_k) - (1-a)(w_k^T w_{k-1}) + \frac{1}{2}(1-a)(w_{k-1}^T w_{k-1})$

$\frac{1}{2}(1-a)(w_{k-1}^T w_{k-1})$ can be dropped due to this term is constant with respect to w_k

$\therefore \frac{1}{2}(w_k^T w_k) - (1-a)(w_k^T w_{k-1})$

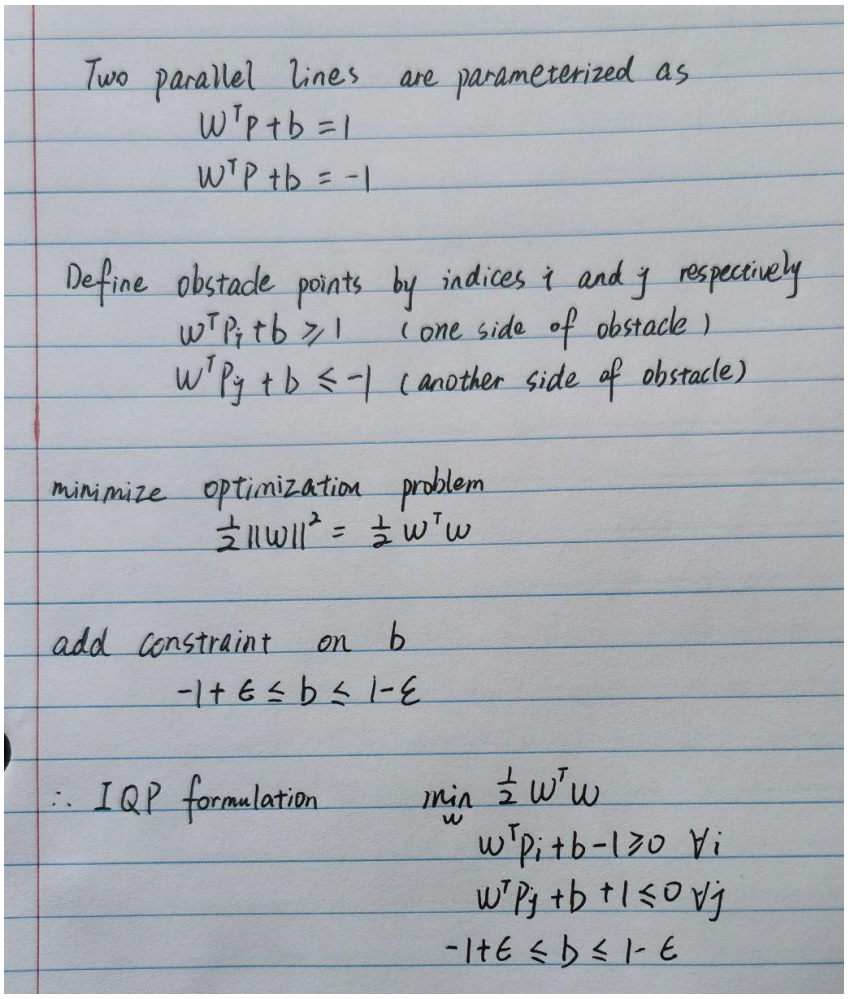
$= \frac{1}{2}(w_k^T w_k) + (a-1)(w_{k-1}^T w_k)$

which is equivalent to equation 9

$\min \frac{1}{2} w_k^T w_k + (a-1)(w_{k-1}^T w_k)$

$w_k^T p_i + 1 \leq 0 \quad \forall i$

Activity 9: Derive the IQP formulation in (20). Explain the rational for the additional constraints on the value of b.



$-1+\epsilon \leq b \leq 1-\epsilon$

An additional constraint on b prevents the parallel virtual line barrier from being too far from the origin. If b is not constrained, the optimization may produce large absolute values of b, which may move the parallel planes unnecessarily further from the origin, causing the actual data points to be farther from where they should be. While this does not affect margin size, it may affect generalization of parallel planes to new data points. Since the parameter ϵ is a small number, some flexibility is allowed in the location of the parallel planes, keeping them within reasonable limits.

Activity 10: Install the Pytrhon QP solver quadprog using the following command:

```
$ sudo pip install quadprog
```

We use the above command to install the Python QP solver quadprog to formulate the IQP equation.

Activity 11: Consider the optimization formulations for obtaining the virtual line barriers in (9) and (20), and the standard form for the quadprog given in (21). Identify the corresponding variables and complete the missing code in the function “getWalls” to find the solution for the problem in (9) using quadprog solver.

```
def getWalls(self, left_obstacles, right_obstacles, wl0, wr0, alpha):


    if self.optim_mode == 0:

        Pr = np.array([[1,0],[0,1]])
        Pl = np.array([[1,0],[0,1]])

        bl = np.full((self.n_pts_l), 1.0,dtype=np.float64)
        br = np.full((self.n_pts_r), 1.0,dtype=np.float64)

        Cl= -(left_obstacles.T)
        Cr= -(right_obstacles.T)

        al = (1-alpha)*wl0
        ar = (1-alpha)*wr0


        wl = solve_qp(Pl.astype(np.float), al.astype(np.float), Cl.astype(np.float),  bl.astype(np.float), 0)[0]
        wr = solve_qp(Pr.astype(np.float), ar.astype(np.float), Cr.astype(np.float),  br.astype(np.float), 0)[0]

    else:
```

Activity 12: Consider the optimization formulation for obtaining the parallel virtual line barriers in (21) and the standard form for the quadprog given in (15). Identify the corresponding variables and complete the missing code in the function “getWalls” to find the solution for the problem in (20) using quadprog solver.

```
else:

    P = np.array([[1.0,0,0],[0,1.0,0],[0,0,0.0001]])

    bl = np.full((self.n_pts_l), 1.0,dtype=np.float64)
    br = np.full((self.n_pts_r), 1.0,dtype=np.float64)
    b = np.concatenate((br,bl,np.array([-0.99, -0.99])))

    Cl= -(left_obstacles.T)
    Cr= -(right_obstacles.T)
    C1 = np.vstack((-Cr,br))
    C2 = np.vstack((Cl,- bl))
    C =np.hstack((C1,C2))
    C =np.hstack((C,np.array([[0,0],[0,0],[1.0,-1.0]])))

    a = np.zeros(3)

    ws  = solve_qp(P.astype(np.float), a.astype(np.float), C.astype(np.float),  b.astype(np.float), 0)[0]


    wr = np.array([ws[0]/(ws[2]-1),ws[1]/(ws[2]-1)])
    wl = np.array([ws[0]/(ws[2]+1),ws[1]/(ws[2]+1)])

return  wl, wr
```

Activity 13: Complete the missing parts in the above equations and the corresponding code in “navigation_lab9_inc.py”.

```
        alpha = 1-math.exp(-dt/self.tau)

        wl, wr = self.getWalls(obstacle_points_l, obstacle_points_r, self.wl0, self.wr0, alpha)

        self.wl0 = wl
        self.wr0 = wr

        dl = 1.0/math.sqrt(np.dot(self.wl0.T,self.wl0))
        dr = 1.0/math.sqrt(np.dot(self.wr0.T,self.wr0))


        wl_h = self.wl0*dl
        wr_h = self.wr0*dr
```

Activity 14: Plot the vehicle velocity vs as a function of $dmin$ and use it to explain what (28) is trying to accomplish. Identify the relevant sections in the code and the parameters in “params.yaml”. Explain how different parameters in (28) can impact the vehicle commanded velocity, vs.

$vs = vs0 * [1 - \exp(-\max(dmin - dstop, 0)/d\_dec)]$

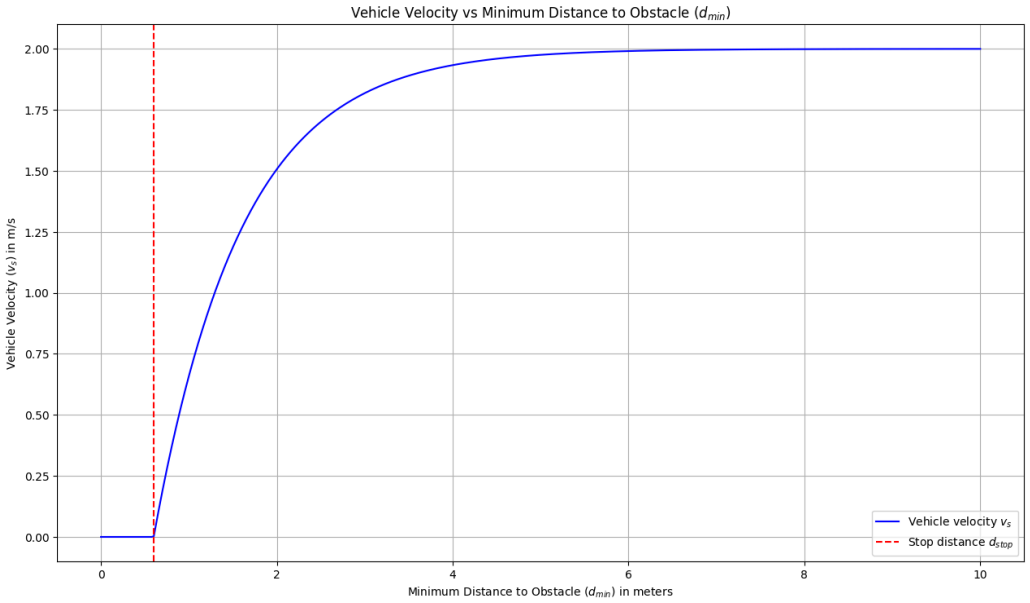
Based on the parameters from params.yaml, we know that

max_speed: vs0 = 2.0,

stop_distance: dstop = 0.6,

stop_distance_decay: d_dec = 1.0.

We can plot the vehicle velocity vs as a function of $dmin$.



The above function vs=vs0*[1−exp(−max($dmin$−$dstop$,0)/d_dec)] represents vehicle speed control, whose purpose is to dynamically adjust the speed of the vehicle based on the proximity of obstacles detected by the sensor. When the vehicle approaches an obstacle, we can safely control the vehicle's speed by slowing down to prevent a collision.

In the Navigation.py script, we define a function called lidar_callback. It calculates vs as an exponentially decaying function of $dmin$, referencing the max_speed, stop_distance, and stop_distance_decay parameters in the params.yaml file.

In the params.yaml file, there are some parameters impacting the vehicle's velocity.

max_speed (equivalent to vs0 in the function): This is the nominal speed of the vehicle, which will never be exceeded if there are no obstacles within the range.

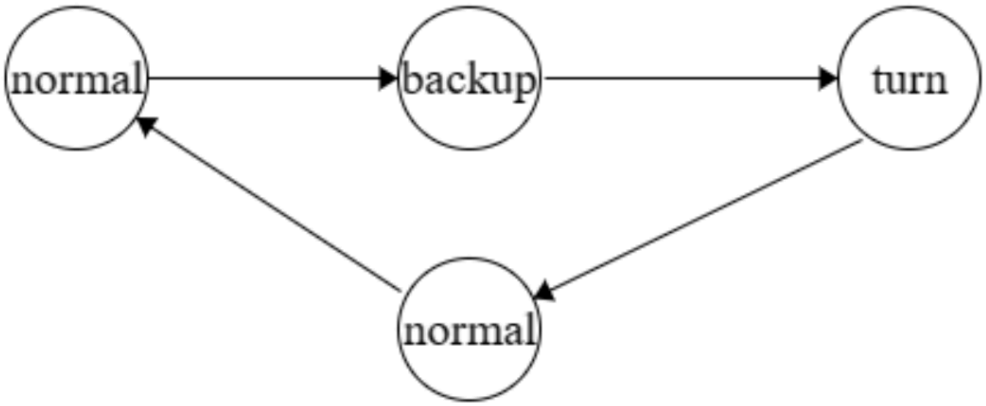
stop_distance (equivalent to $dstop$): Defines the required stopping distance from the obstacle. If $dmin$ is greater than $dstop$, the vehicle maintains its nominal speed; if $dmin$ is less than $dstop$, the vehicle slows down.

stop_distance_decay` (equivalent to d_dec): This determines how quickly the vehicle slows down as it approaches "stop_distance". Large values will cause the vehicle's speed to gradually decrease as it approaches an obstacle.

Vehicle speed can be calibrated for obstacles detected by adjusting parameters in the "params.yaml" file. This ensures that the vehicle can adapt to various conditions on the road, thereby ensuring the safety and efficiency of its operation.

Activity 15: The “back-up and turn” functionality is implemented using a finite-state machine with three states: “normal”, “backup” and “turn”. Carefully examine the Python code in “navigation_lab9_inc.py” for the implementation of this functionality. Draw the finite-state machine diagram with all necessary details to explain the vehicle operation. You are encouraged to suggest (and implement) improvements to this functionality as you see fit.

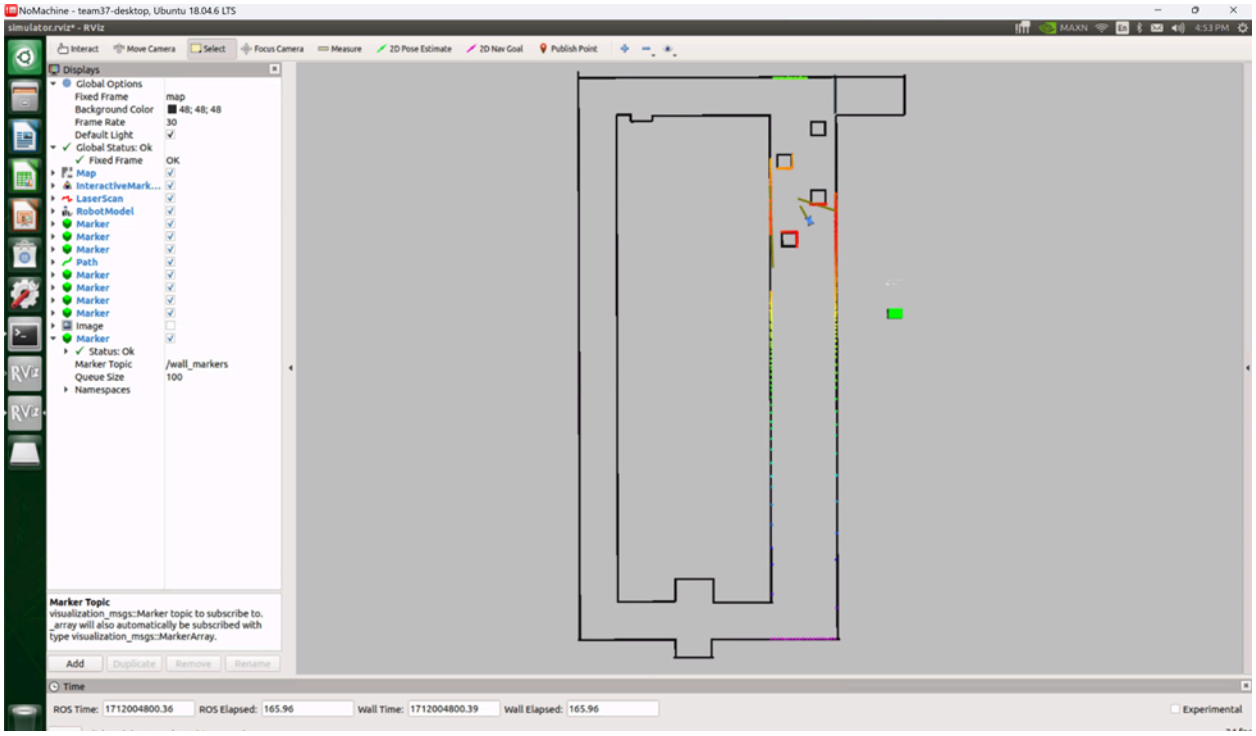
In the finite state machine of the vehicle navigation system, the vehicle remains in the "normal" state under general conditions and moves forward while avoiding obstacles detected by LIDAR. If an obstacle is detected within a predetermined safe distance, the vehicle switches to "backup" mode and reverses until there is enough room to turn. Subsequently, it enters the "turn" state and turns around the obstacle with a minimum radius. After the turn is completed, the vehicle's LIDAR evaluates the route ahead; if it is unobstructed, it returns to the "normal" state and continues forward.

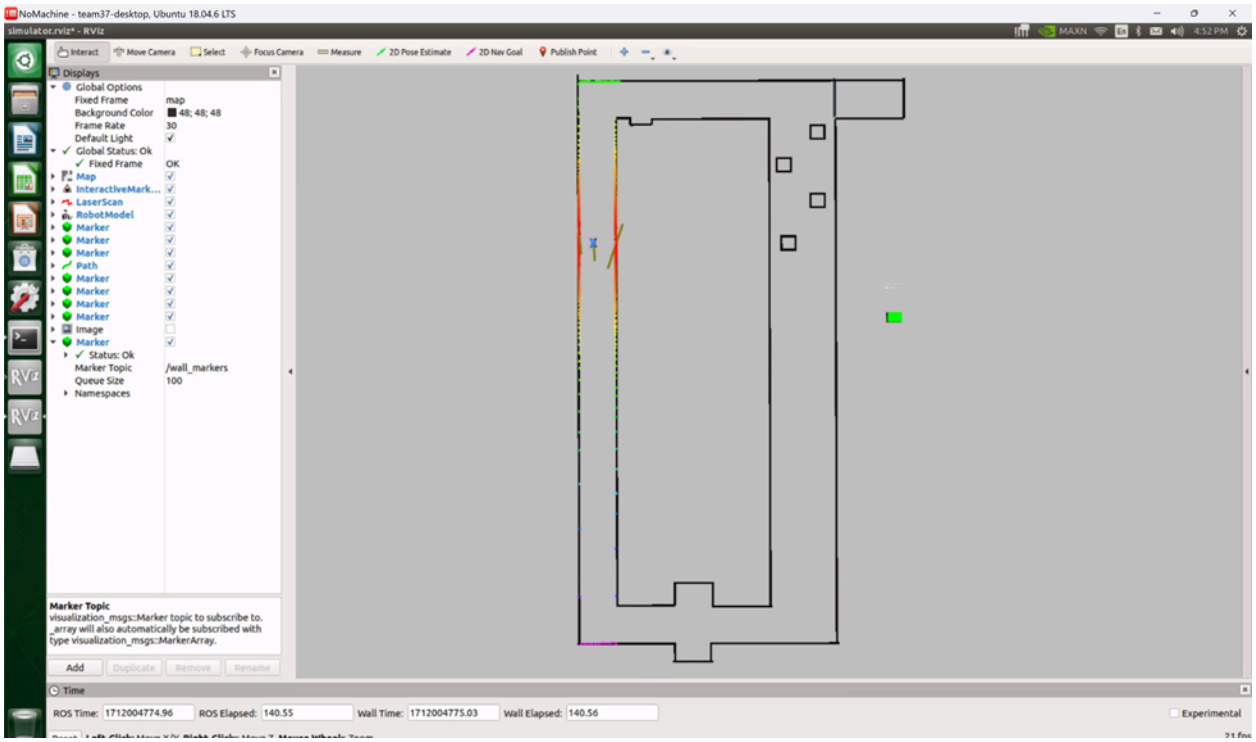


Activity 16: Launch the f1tenth_simulator “simulator.launch” file. Add a display of Marker type with the topic “wall_markers” to the “rviz” visualization environment. These markers represent the virtual separating line barriers and desired heading direction of the vehicle. Use the simulator environment to fine-tune the values of the control algorithm parameters until you achieve a satisfactory response in the

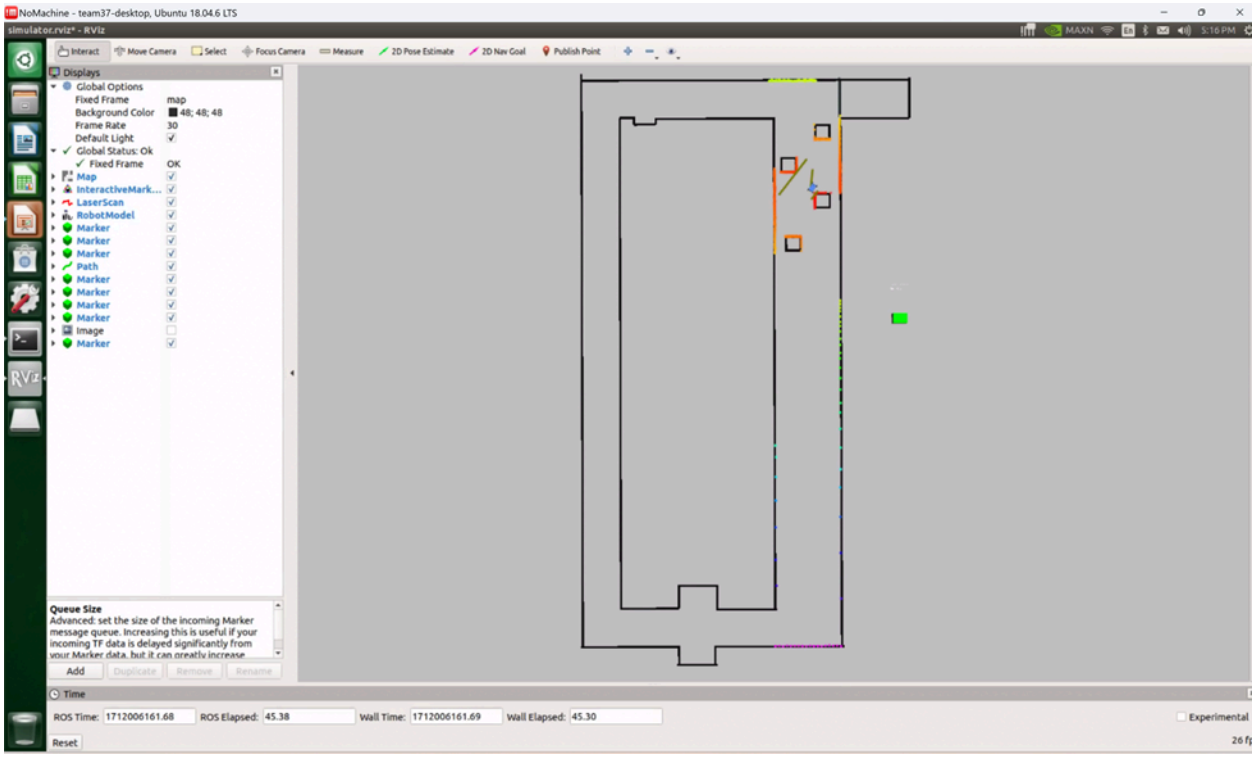
simulation environment. You must try both formulations for finding the virtual separating line barriers in (9) and (20). Note that the value of the parameter “optim_mode” determines which formulation is used. Pay close attention to the “wall_markers” as you drive the vehicle around in the simulation environment.

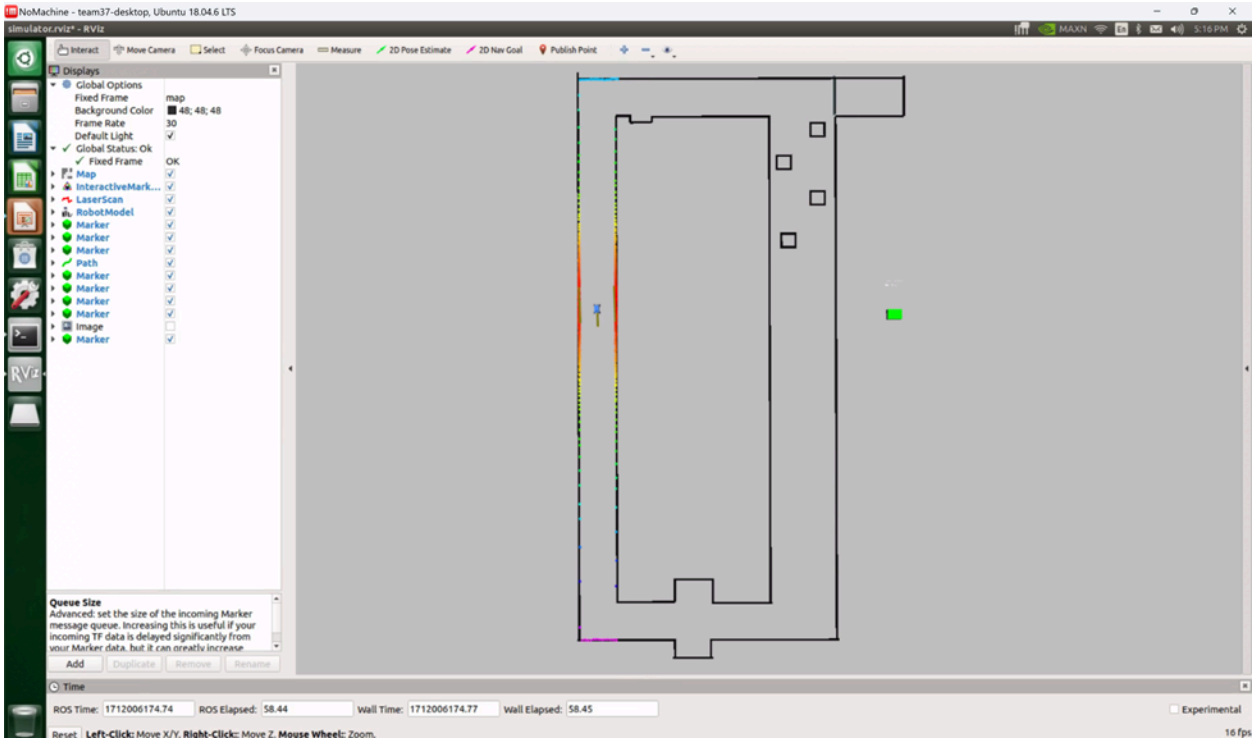
```
# Separating Barrier Line Optimization Mode: 0 (indepndent) 1 (parallel)
optim_mode: 0
```






```
# Separating Barrier Line Optimization Mode: 0 (indepndent) 1 (parallel)
optim_mode: 1
```





The "wall_markers" displays the markers used to create wall visualizations. To calculate markers, the algorithm determines left and right wall segments by searching for lidar point clusters from the surroundings. The visual walls shown by "wall_markers" are useful for whether the vehicle is correctly detecting walls in its environment.

Activity 17: Compare the performance of the self-driving algorithm using the two optimization formulations in (9) and (20). Briefly report your observations on the impact of the parameters “k_p”, “k_d” and “n_pts_l”, “n_pts_r”, and “safe_distnace” on the vehicle self-driving performance.

When optim_mode is set to 0, the vehicle adjusts its path based on the results of a simple linear program. It performs better in simpler environments, such as driving in straight lines or with slight curves, but may not be agile enough in complex obstacle environments.

When optim_mode is set to 1, the vehicle will adjust its path based on more complex optimization algorithms. As more variables and constraints are introduced into the model, the vehicle becomes more efficient and flexible in handling complex environments. For example, when a vehicle is driving on a road with frequent obstacles, it can successfully avoid collisions with obstacles and drive out.

In the control system, Kd and Kp are important parameters that affect the response characteristics. Kd represents the differential gain, which adjusts the differential response, while Kp represents the proportional gain, which controls the proportional response of the controller to error.

The parameters “n_pts_l” and “n_pts_r” control the number of LIDAR points used in the vehicle left wall following algorithm and right wall following algorithm respectively. Increasing the number of points increases the accuracy of the algorithm, but also increases the calculation time and reduces the speed.

The parameter "safe_distance" controls the minimum distance between the vehicle and the wall in the algorithm. When the safety distance is increased, the vehicle moves farther away from the wall, which helps reduce the risk of a collision but may also impair the vehicle's ability to maneuver in tight places. Reducing the safety distance improves efficiency but also increases the risk of collision.

Activity 18: After you are satisfied with the performance of the self-driving controller in the simulations, you must try it on the actual vehicle. Before starting, set the value of “optim_mode” to select the right optimization mode for your experiment. Make sure the environment around the vehicle is free of people and safe for vehicle operation and launch the file “experiment.launch”. As in Lab 8, the self-driving mode can be activated and de-activated at by toggling the “RB” on Joystick. Note that pressing the “LB” would activate the manual driving mode. You can use either of these options to stop self-driving in case of an emergency. Remember to always keep the vehicle in your reach so you would not lose the connection between the vehicle and the Joystick.

In the experiment part, when we start self-driving, the vehicle suddenly goes backward with turns. Even with TA’s assistance, our vehicle still doesn’t move forward. In this case, based on the simulation result, we predict that the vehicle can drive well in simple environments such as straight lines or slightly curved paths but may be less agile in complex environments with many obstacles when optim_mode is set to 0. When optim_mode is set to 1, the vehicle can smoothly pass through roads with frequent obstacles and avoid collisions with obstacles.

Activity 19: Adjust the controller parameters as you see necessary to achieve a satisfactory response. Briefly report on your observations from the experiments. Comment on any differences that you may

observe between the system responses under the two optimization modes.

Even though our vehicle didn’t complete the experimental part, we made an estimation of the vehicle's self-driving after adjusting the controller parameters. When increasing the value of k_p, since it controls the response of the controller, the response becomes faster and turns when the safe distance threshold is reached. For example, when a vehicle detects an obstacle, it can react quickly by steering and avoiding it.

Activity 20: Reflect on the operation of the self-driving control algorithm and suggest potential ways for improving it as you see fit.

Autonomous driving control algorithms combine wall tracking and gap tracking algorithms to guide the car along its route while avoiding collisions with walls and other obstacles. The wall following algorithm uses the data scanned by the lidar sensor to calculate the angle and distance between the vehicle and the wall, and then modifies the steering angle to keep the vehicle in the direction of avoiding obstacles, thereby keeping the vehicle at a safe distance away from walls and obstacles. While the gap following algorithm is used to find the gap in the wall and pass through it.

For improving autonomous driving, we believe that machine learning and computer vision technology should be integrated into autonomous driving control algorithms, using advanced sensor technologies such as stereo cameras or radars to provide more accurate environmental information, and combining them with advanced path planning and obstacle avoidance algorithms, which can significantly improve the efficiency and safety of autonomous driving. At the same time, incorporating machine learning to improve driving behavior and using 3D mapping technology to create more detailed environmental maps are potential ways to improve the performance of autonomous driving algorithms.