Environment and Obstacle Generation

Environment and Obstacles are created using the same method used in the previous assignments. The environment is of 800*800 matrix with open space represented as '0' and obstacles are represented as '1'. Starting and ending points are represented as '5' and '6' respectively.

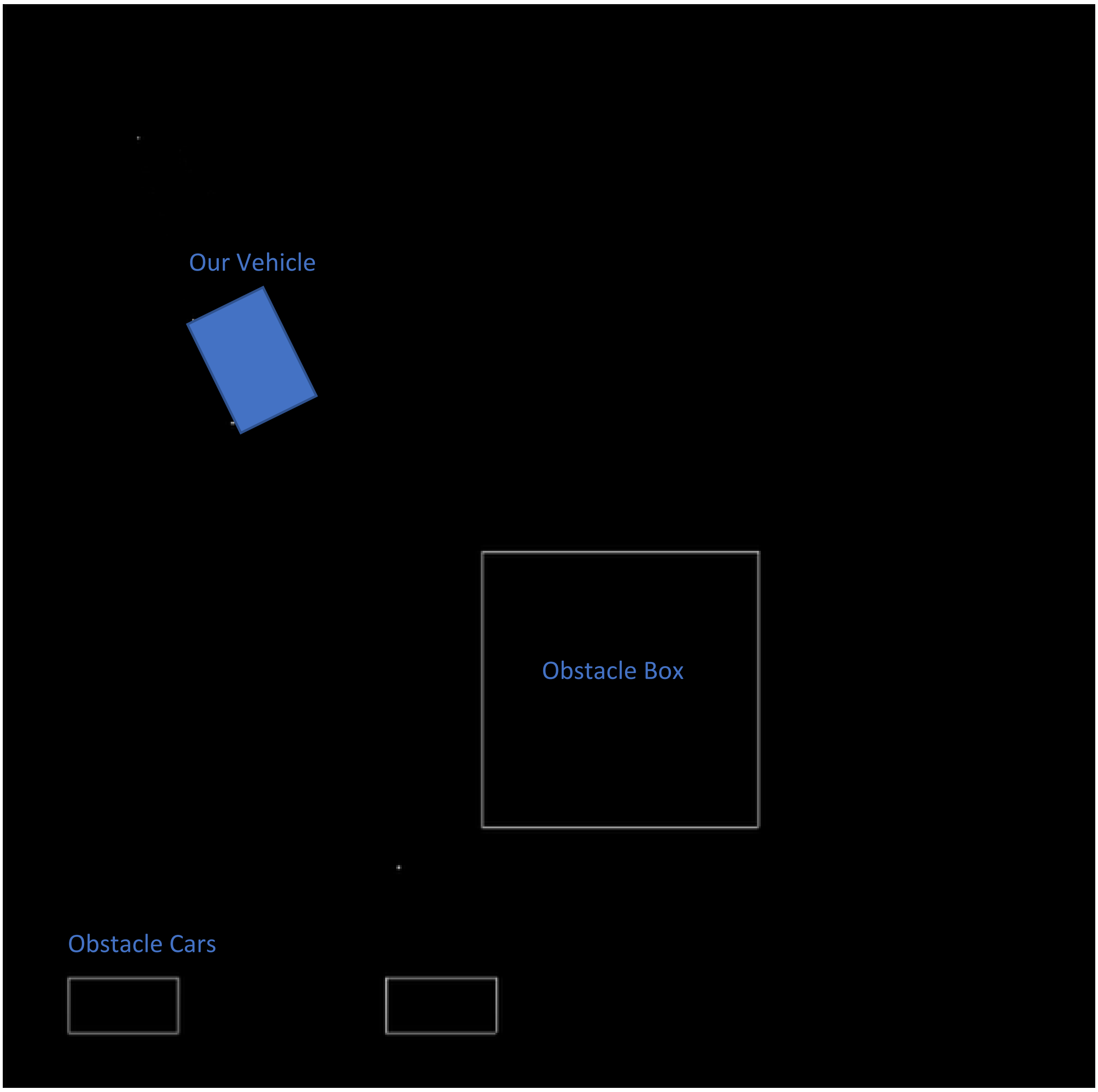


Figure 1: Environment with Obstacles

Collision Check

Collision check is performed by making sure that the points lie on the boundary of our vehicle won't collide with other obstacles I.e. make sure that the value of the car structure in the environment matrix is not equal to '1'. As we know that we have represented our obstacles with '1' in the matrix.

The pseudocodes for all three cases are as mentioned below:

(Here I haven't included the environment and obstacles generation code because it is already included in the previous assignments.)

Pseudocode – Planner Algorithm for Di-wheel Robot

1) r <- R (Wheel radius)
2) l <- L (wheel span)
3) dt <- 2
4) U_r, U_l <- [-1,0,1]
5) end_new = empty list
6) prev = empty list
7) grd_trc = 800*800 array of -1 to track the path
8) dist_from_start = cost calculation -> inf matrix of 800*800
9) start_angle <- 0
10) seen <- list that tracks the visited points
11) heap = empty list # here we are using heap instead of queue to optimize the time complexity of the program
12) x,y <- starting point of the vehicle
13) dist_from_start[x][y] <- 0
14) add current node to seen list
15) Found, Resign <- False
16) heap <- dist_from_start[x][y], (x, y), start_angle
17) While Found and Resign are True:
    a. x,y,start_angle <- pop from heap
    b. append x,y to previously visited nodes
    c. if (Found==True):
        i. exit the loop
    d. else
        i. for ii,item in enumerate(combination of U_r and U_l):
            1. theta_new = start_angle + ((r/l)*(item[0] - item[1]))*dt
            2. x1 = int(x + (r/2)*(item[0] + item[1])*math.cos(theta_new)*dt)
            3. y1 = int(y + (r/2)*(item[0] + item[1])*math.sin(theta_new)*dt)
            4. if x1 and y1 are withing the boundary and not colliding with anything:
                a. calculate the cost as we did in the Dijkstra
                b. update x and y to x1 and y1
                c. add x1,y1 and theta_new to 'grd_trc[ii]'

Pseudocode – Planner Algorithm for Ackermann

1) l <- L (wheel span)
2) dt <- 2 (Time stamp)
3) u_phi <- list of steering angles (list of angles)
4) u_s = [-1,-0.5,-0.1,0.1,0.5,1] (list of possible speed)
5) end_new = empty list
6) prev = empty list
7) grd_trc = 800*800 array of -1 to track the path
8) dist_from_start = cost calculation -> inf matrix of 800*800
9) start_angle <- 0
10) seen <- list that tracks the visited points
11) heap = empty list   # here we are using heap instead of queue to optimize the time complexity of the program
12) x,y <- starting point of the vehicle
13) dist_from_start[x][y] <- 0
14) add current node to seen list
15) Found, Resign <- False
16) heap <- dist_from_start[x][y], (x, y), start_angle
17) While Found and Resign are True:
    a. x,y,start_angle <- pop from heap
    b. append x,y to previously visited nodes
    c. if (Found==True):
        i. exit the loop
    d. else
        i. for U_phi in u_phi:
            1. for U_s in u_s
                a. theta_new = start_angle + ((math.tan(U_phi)*U_s)/l)*dt
                b. x1 = x + U_s*math.cos(theta_new)*dt
                c. y1 = y - U_s*math.sin(theta_new)*dt
                d. if x1 and y1 are withing the boundary and not colliding with anything
                    i. calculate the cost as we did in the dijkstra's
                    ii. update x and y to x1 and y1
                    iii. add x1,y1 and theta_new to 'grd_trc[ii]'

Pseudocode – Planner Algorithm for Truck and Trailer

1) l <- L (wheel span)
2) dt <- 2 (Time stamp)
3) r <- R (Wheel radius)
4) U_r, U_l <- [-1,0,1]
5) u_phi <- list of steering angles (list of angles)
6) u_s = [-1,-0.5,-0.1,0.1,0.5,1] (list of possible speed)
7) end_new_car = empty list
8) end_new_trailer = empty list
9) prev = empty list
10) grd_trc = 800*800 array of -1 to track the path
11) dist_from_start = cost calculation -> inf matrix of 800*800
12) start_angle_car <- 0
13) start_angle_trailer <- 0
14) seen <- list that tracks the visited points
15) heap = empty list # here we are using heap instead of queue to optimize the time complexity of the program
16) x_car,y_car <- starting point of the vehicle
17) x_trailer,y_trailer <- starting point of the Trailer
18) dist_from_start[x_car][y_car] <- 0
19) add current node to seen list
20) Found, Resign <- False
21) heap <- dist_from_start[x][y], (x_car, y_car), start_angle_car, (x_trailer, y_trailer), start_angle_trailer
22) While Found and Resign are True:
    a. xc,yc,xt,yt,start_angle <- pop from heap
    b. append xc,yc,xt,yt to previously visited nodes
    c. if (Found==True):
        i. exit the loop
    d. else
        i. **repeate part part 17.d to get new xt and yt position of the trailer**
        ii. **for U_phi in u_phi:**
            1. **for U_s in u_s**
                a. **theta_new_car = start_angle_car+ ((math.tan(U_phi)*U_s)/l)*dt**
                b. **x1c = xc + U_s*math.cos(theta_new_car)*dt**
                c. **y1c = yc - U_s*math.sin(theta_new_car)*dt**
                d. **if x1c and y1c are withing the boundary and not colliding with anything**
                    i. **calculate the cost as we did in the dijkstra's**
                    ii. **update xc and yc to x1c and y1c**
                    iii. **add x1c,y1c and theta_new_car to 'grd_trc[ii]'**

Results

Note: I am unable to do the parking problem of the trailer so I haven't included that here.

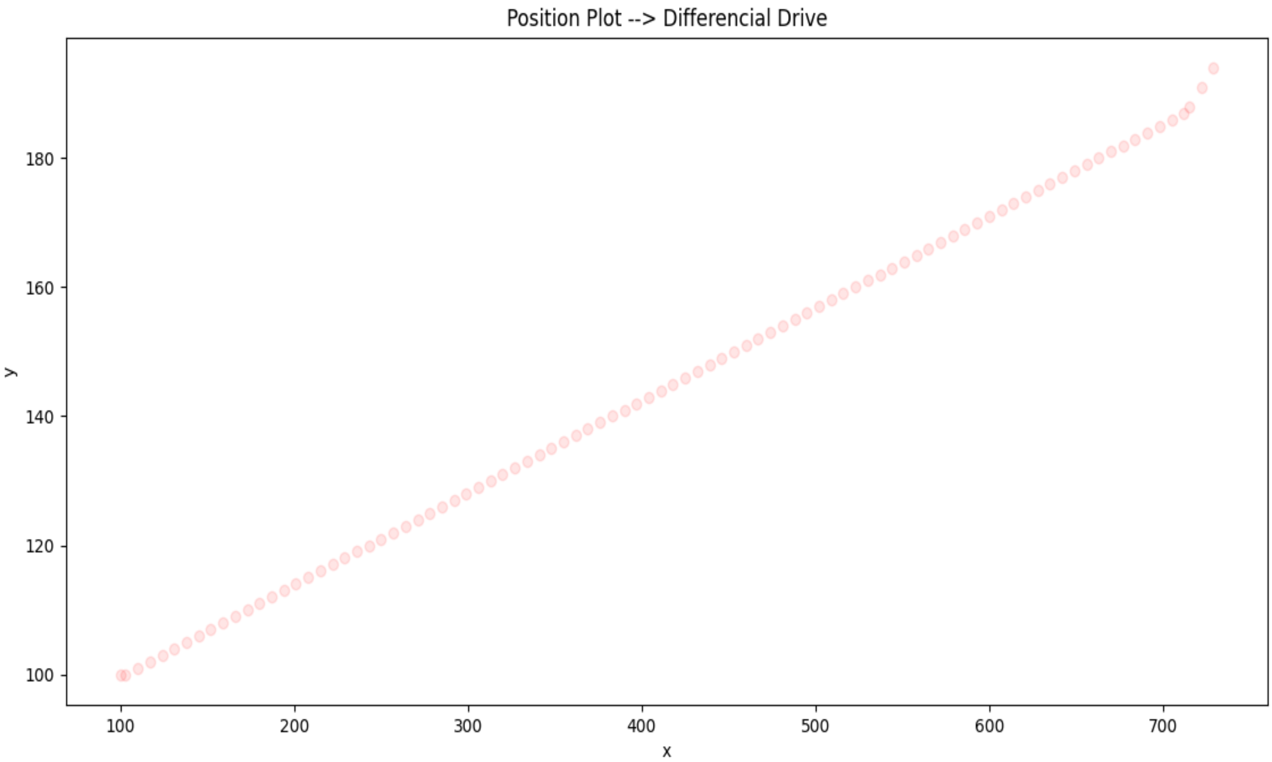


Figure 2: Di-wheel Robot path generation (starting point = (100,100))

As shown in Figure 2, the Di-wheel Robot moves from its starting point to the goal location. Before it reached the goal location, it is trying to change its angle to match the angle of the next parked car in the line. As we know that the di-wheel robot is capable of rotating without translation so there is no need to follow the standard parallel parking rules.

x,y, and 'theta' are generated using the equations[1] shown below:

$$\dot{x} = \frac{r}{2}(u_l + u_r)\cos\theta$$
$$\dot{y} = \frac{r}{2}(u_l + u_r)\sin\theta$$
$$\dot{\theta} = \frac{r}{L}(u_r - u_l).$$

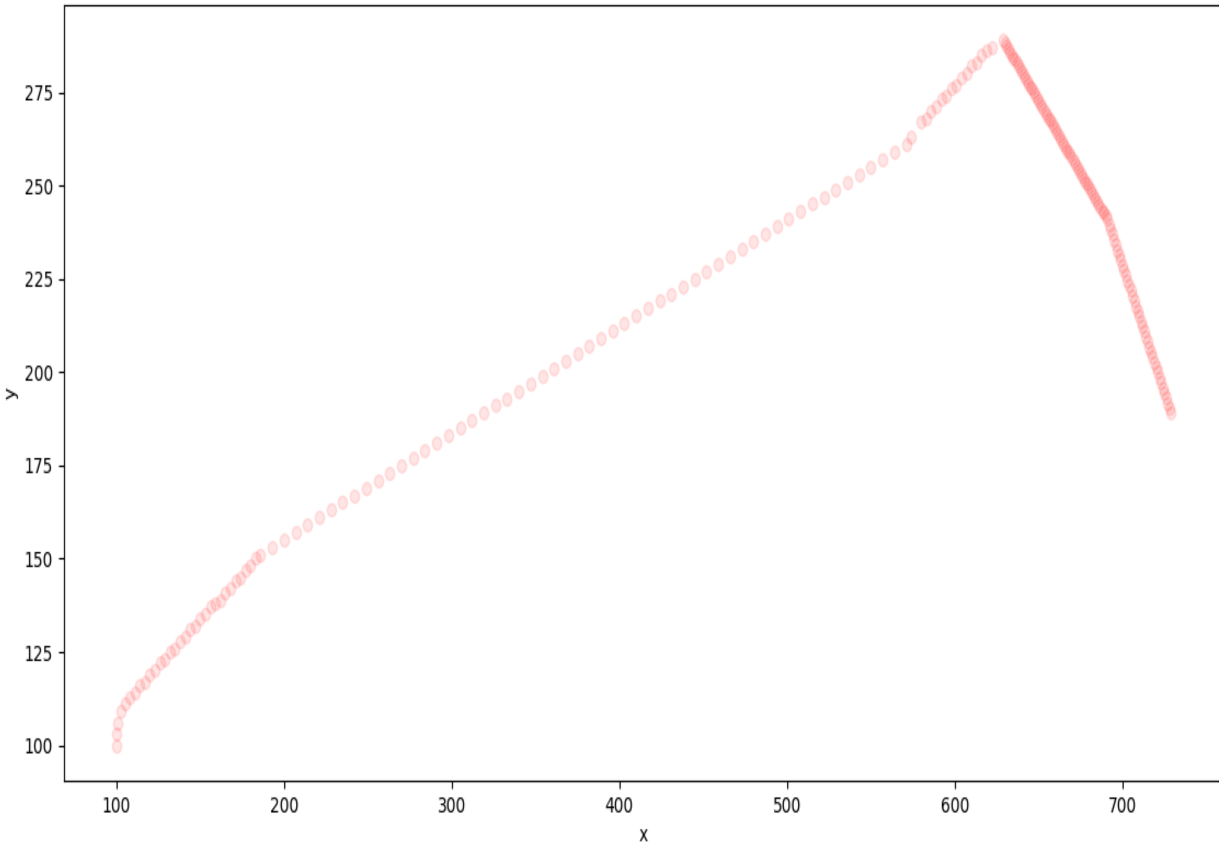
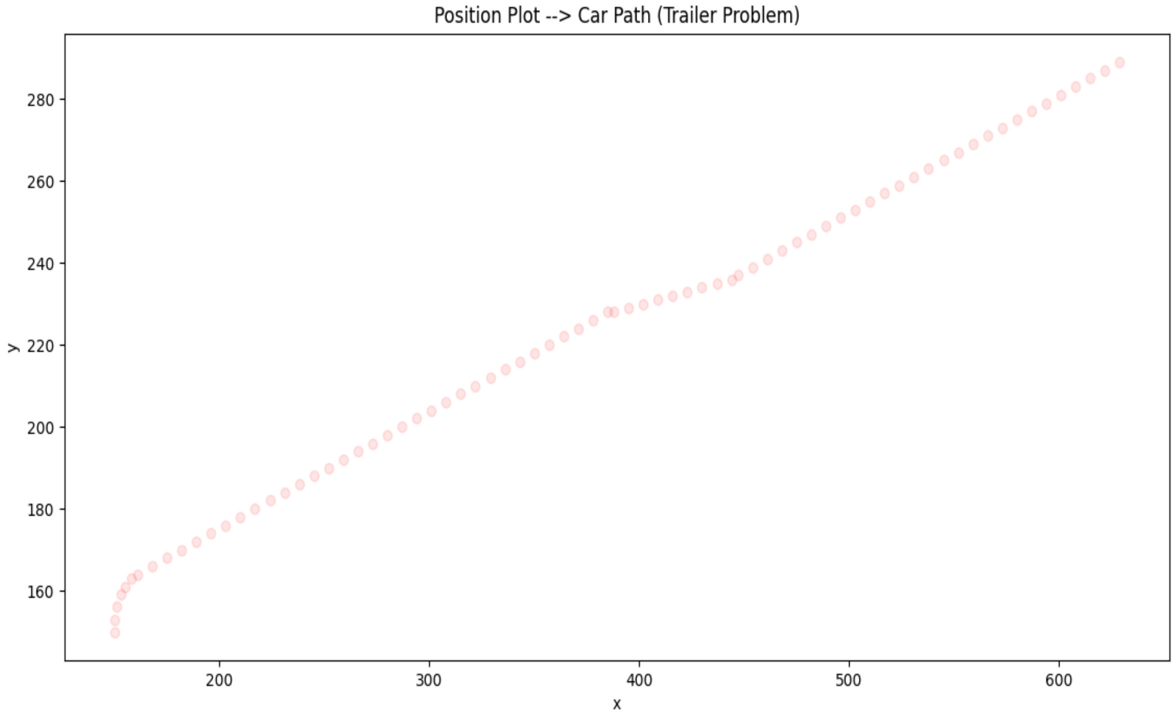


Figure 3: Ackermann Car path generation (starting point = (100,100))

As you can see in Figure 3, our car moves from the starting point to the location which is parallel to the car parked next to the open parking spot. Afterward, it follows the standard parallel parking rules to park itself.

x,y, and 'theta' are generated using the equation[1] shown below:

$$\dot{x} = u_s \cos\theta$$
$$\dot{y} = u_s \sin\theta$$
$$\dot{\theta} = \frac{u_s}{L} \tan u_\phi .$$



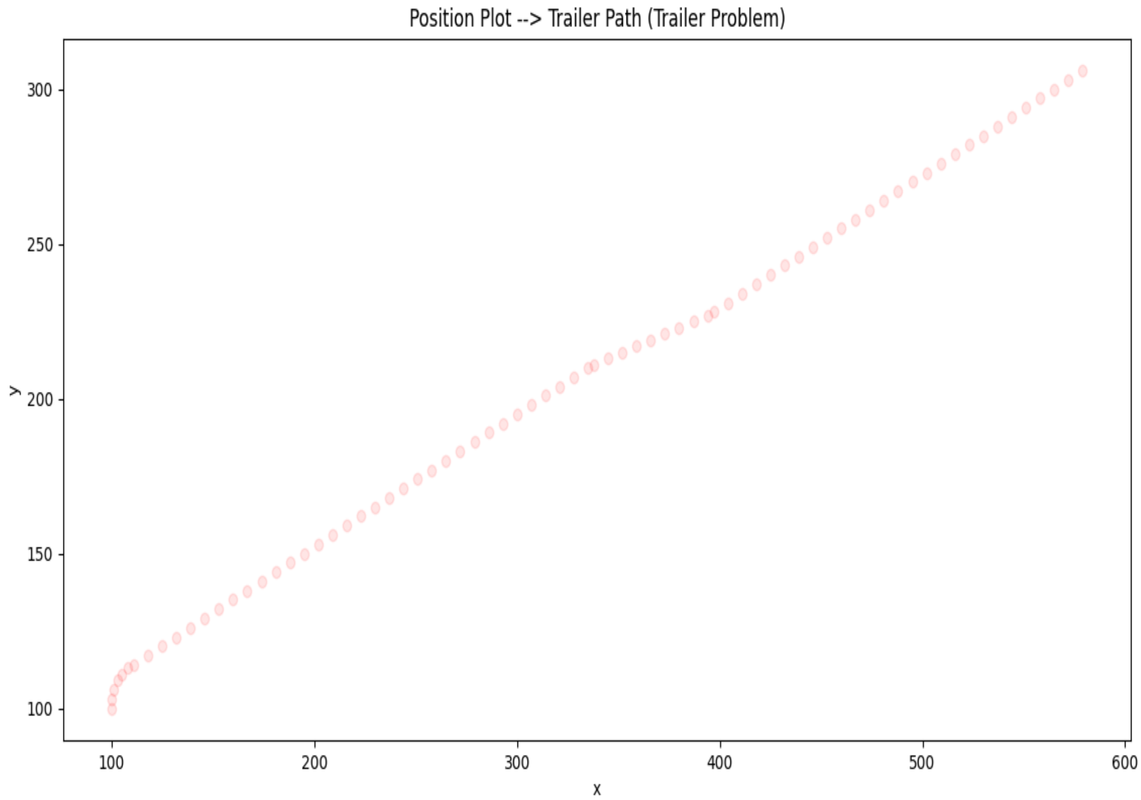


Figure 4: Trailer and truck path generation

A truck with a trailer path can be generated using the combination of the Ackermann and Di-wheel path generation method. Please see the code for more information.

x,y, and 'theta' are generated using the equation[1] shown below:

$$
\begin{aligned}
\dot{x} &= s\cos\theta_0 \\
\dot{y} &= s\sin\theta_0 \\
\dot{\theta}_0 &= \frac{s}{L}\tan\phi \\
\dot{\theta}_1 &= \frac{s}{d_1}\sin(\theta_0 - \theta_1) \\
&\vdots \\
\dot{\theta}_i &= \frac{s}{d_i}\left(\prod_{j=1}^{i-1}\cos(\theta_{j-1} - \theta_j)\right)\sin(\theta_{i-1} - \theta_i) \\
&\vdots \\
\dot{\theta}_k &= \frac{s}{d_k}\left(\prod_{j=1}^{k-1}\cos(\theta_{j-1} - \theta_j)\right)\sin(\theta_{k-1} - \theta_k).
\end{aligned}
$$

References

[1] Steven M. LaValle. Planning Algorithms. Cambridge University Press, May 2006. ISBN 9780521862059. URL http://lavalle.pl/planning/.