EE209AS (Fall 2018) Problem Set 4

Jingwen Zhang

November 11, 2018

Preliminaries

0(a). Github repo:
`https://github.com/Jingwen-Zhang-Aaron/Computational-Robotics.git`
The fully commented script is under the folder "pset4-RRTplanner".

0(b). I did it individually. I used the book "Robotics, Vision and Control" from Peter Corke and the lecture notes of MIT class Robotics systems and science by Daniela Rus as my reference.

0(c). I coded all problems under IPython environment with Jupyter Notebook.

0(d). I contributed 100% of this work.

1 Robot Model

I consider the 2 wheeled robot as follows:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}$$

As our input is the angular vels of left and right wheels, our model can be expressed in the following way:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{-1}{w} & \frac{1}{w} \end{bmatrix} \begin{bmatrix} \omega_L \\ \omega_R \end{bmatrix}$$

where $w = 85mm$ is the distance between two wheels. And for planning, I consider the true shape of the robot to be a rectangle of length 80mm and width 85mm.

2 Trajectory Planning

2(a). Find the Closest Node

To find the closest node, just use a loop to compare every node in a list of all nodes with the target point so that it's easy to find the closest node. But I used a new class called $Node()$ to represent all nodes instead of just the state (x, y, θ) of the robot. Because the class $Node()$ includes both the

state and the parent info which is very useful to get the path when planning. The implementation is as follows:

```
class Node():
    """RRT Node using .parent to keep the info of edges"""
    def __init__(self, state):
        self.state = state
        self.parent = None

def find_closestnode(node_list, target_point):
    """ given a set of RRT nodes <node_list> in C-space and a target point
        <target_point>
    return the closest node to the target point inside the set and the index
        """
    Closestnode = np.zeros(3)
    Distmin = sys.maxsize
    min_index = sys.maxsize
    for i, node in enumerate(node_list):
        dist = np.linalg.norm(target_point - node.state, ord = 2)
        if dist < Distmin:
            Closestnode = node
            Distmin = dist
            min_index = i
    return Closestnode, min_index
```

2(b). Smooth Trajectory

Our goal is to drive the robot to a goal position and orientation but they are coupled. So I used the formulation from the book "Robotics, Vision and Control" which is shown in Figure 1

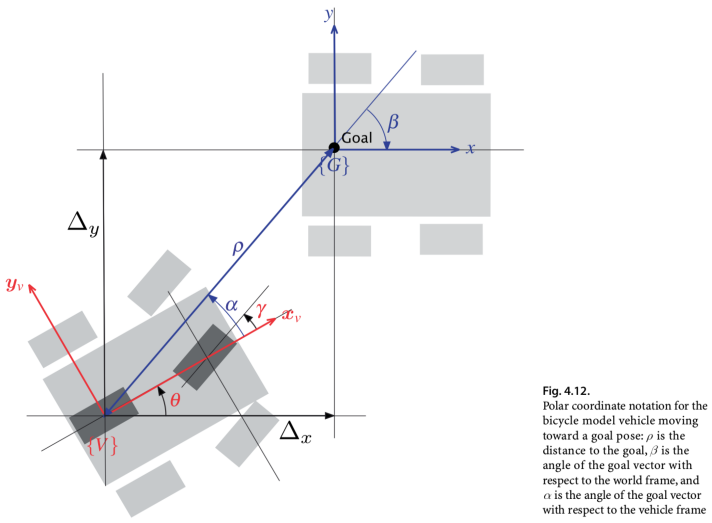

Figure 1: Polar Coordinate Notation for the Mobile Robot

Change the coordinates into polar coordinates:

$$\begin{cases} \rho = \sqrt{\Delta_x^2 + \Delta_y^2} \\ \alpha = \tan^{-1}\frac{\Delta_y}{\Delta_x} - \theta \\ \beta = -\theta - \alpha \end{cases}$$

which results in

$$\begin{pmatrix} \dot{\rho} \\ \dot{\alpha} \\ \dot{\beta} \end{pmatrix} = \begin{pmatrix} -\cos\alpha & 0 \\ \frac{\sin\alpha}{\rho} & -1 \\ -\frac{\sin\alpha}{\rho} & 0 \end{pmatrix} \begin{pmatrix} v \\ w \end{pmatrix}$$

And use a simple linear control law:

$$v = k_\rho \rho$$
$$\omega = k_\alpha \alpha + k_\beta \beta$$

It will drive the robot to $(\rho, \alpha, \beta) = (0, 0, 0)$ which means the target point. The implementation is:

```
# simulation parameters
Kp_rho = 0.3
Kp_alpha = 1.5
Kp_beta = -0.3
dt = 0.1

def trajectory_generation(initial_state, target_state, mode=0):
    """ given initial and target robot states in C-space
    return the smooth trajectory and corresponding control inputs"""
    """
    rho is the distance between the robot and the goal position
    alpha is the angle to the goal relative to the heading of the robot
    beta is the angle between the robot's position and the goal position plus
        the goal angle
    Kp_rho*rho and Kp_alpha*alpha drive the robot along a line towards the goal
    Kp_beta*beta rotates the line so that it is parallel to the goal angle
    """
    # Initilization
    x_start = initial_state[0]
    y_start = initial_state[1]
    theta_start = initial_state[2]
    x_goal = target_state[0]
    y_goal = target_state[1]
    theta_goal = target_state[2]

    x = x_start
    y = y_start
    theta = theta_start
    x_diff = x_goal - x
    y_diff = y_goal - y
    traj = []
    input_traj = []
    rho = np.sqrt(x_diff**2 + y_diff**2)
    t = 0
    while (t < 1.0) and (rho > 0.01):
        traj.append(np.array([x, y, theta]))
```

```
        x_diff = x_goal - x
        y_diff = y_goal - y
        """
        Restrict alpha and beta (angle differences) to the range
        [-pi, pi] to prevent unstable behavior e.g. difference going
        from 0 rad to 2*pi rad with slight turn
        """
        rho = np.sqrt(x_diff**2 + y_diff**2)
        alpha = (np.arctan2(y_diff, x_diff) -
                 theta + np.pi) % (2 * np.pi) - np.pi
        beta = (theta_goal - theta - alpha + np.pi) % (2 * np.pi) - np.pi

        v = Kp_rho * rho
        omega = Kp_alpha * alpha + Kp_beta * beta
        # Get the control inputs
        omega_right = (2*v + omega*w)/(2*r)
        omega_left = (2*v - omega*w)/(2*r)
        input_traj.append(np.array([omega_left, omega_right]))

        if alpha > np.pi / 2 or alpha < -np.pi / 2:
            v = -v

        theta = theta + omega * dt
        x = x + v * np.cos(theta) * dt
        y = y + v * np.sin(theta) * dt
        if mode is 0:
            t += dt
    return traj, input_traj
```

I used the parameter $mode$ to decide the robot should reach the goal or not. When $mode = 0$, the robot will go towards the target lasting 1 sec with a smooth achievable trajectory which is used to expand the tree of RRT. When $mode = 1$, the robot will reach the target point. In Figure 2 as an example, the solid line means that the robot will move towards the target lasting 1 sec with mode-0 while the dashed line means mode-1 indicated that the robot reached the goal.

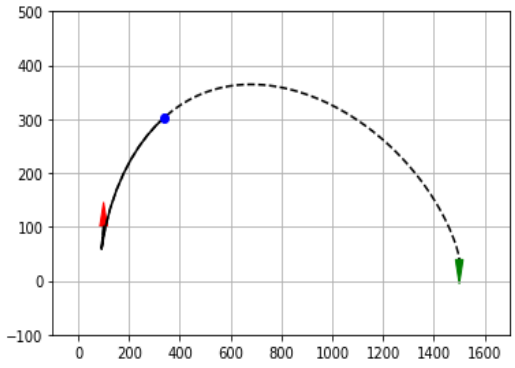

Figure 2: Smooth Achievable Trajectory of the Robot

And I also computed the control inputs when generating the smooth trajectory. The result $input\_traj$ is the control inputs for the trajectory which is computed based on the robot model.

2(c). Map Visualization

I defined an open parking environment in the operational space (O-space) as follows (red arrow indicates the initial state of the robot and green arrow indicates the goal state, black rectangles are 2D obstacles):

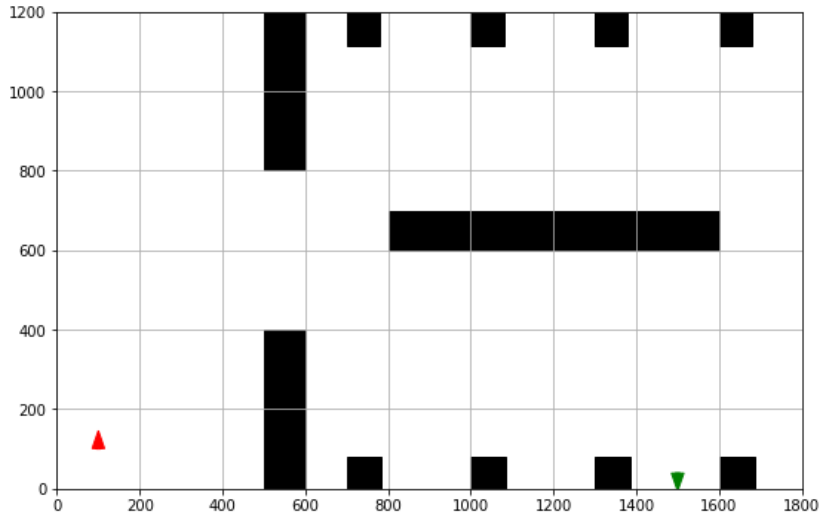

Figure 3: Planning Environment in Operational Space

To translate this map into configuration space (C-space), need to use the real shape of the robot to expand the obstacles. The general process is shown in Figure 3. After finding all possible vertices, using ConvexHull to form it.

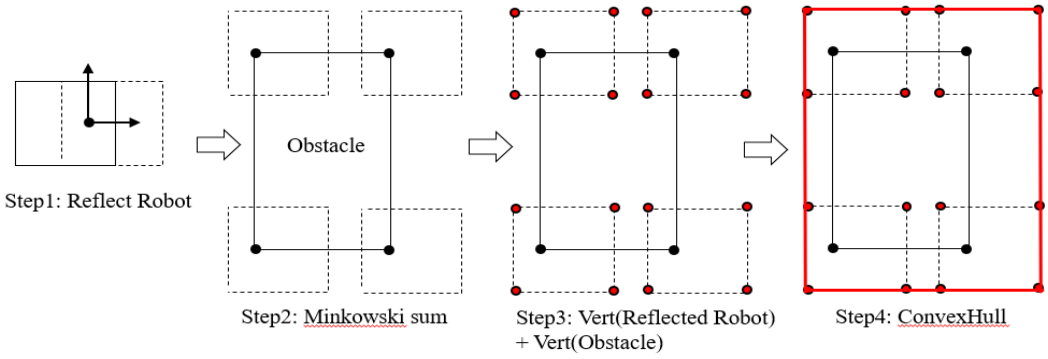

Figure 4: C-space Obstacles Algorithm

So for a given robot and its orientation, we can get the expanded C-space obstacle based on O-space obstacle. For example, the following figure showed the C-space obstacle with the orientation of the robot is $-60°$. The black rectangle is the O-space obstacle while the black lines indicate the boundaries of the C-space obstacle (the blue dots are vertices from the reflected robot and the original obstacle).

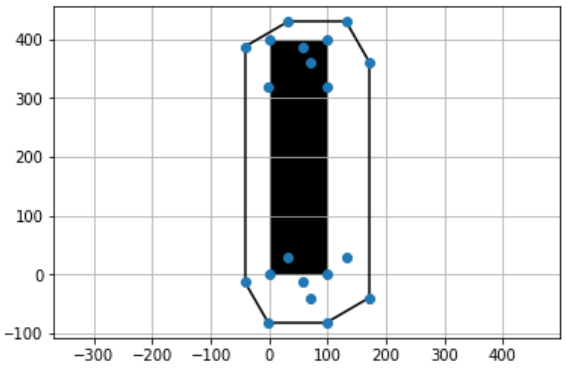

Figure 5: C-space Obstacle with -60degrees Robot

We knew the robot angle is $[-\pi, \pi]$, so compute the shape for every angle of the robot to form the real C-space obstacles in C-space. The implementation is:

```
def draw_Cspace(initial_state, goal_state, obstaclelist):
    """ translate the previous Ospace map into Cspace map using ConvexHull to
    form the C-space obstacles"""

    fig = plt.figure(figsize = (12, 8))
    #ax = fig.gca()
    ax = fig.gca(projection='3d')
    ax.set_xlim([0, 1800])
    ax.set_ylim([0, 1200])
    ax.view_init(elev=60)
    Cspace_obstacle = {}
    for ob in obstaclelist:
        theta_sample = np.linspace(-np.pi, np.pi, 100)
        points = []
        for theta in theta_sample:
            A = [[ob[0], ob[1], theta], [ob[0]+ob[2], ob[1], theta],
                 [ob[0], ob[1]+ob[3], theta], [ob[0]+ob[2], ob[1]+ob[3], theta]]
            for a in A:
                vertice1 = [a[0]+70*cos(theta)+42.5*cos(theta-np.pi/2),
                            a[1]+70*sin(theta)+42.5*sin(theta-np.pi/2), theta]
                vertice2 = [a[0]+70*cos(theta)+42.5*cos(theta+np.pi/2),
                            a[1]+70*sin(theta)+42.5*sin(theta+np.pi/2), theta]
                vertice3 = [a[0]+10*cos(theta+np.pi)+42.5*cos(theta-np.pi/2),
                            a[1]+10*sin(theta+np.pi)+42.5*sin(theta-np.pi/2), theta]
                vertice4 = [a[0]+10*cos(theta+np.pi)+42.5*cos(theta+np.pi/2),
                            a[1]+10*sin(theta+np.pi)+42.5*sin(theta+np.pi/2), theta]
                points.append(a)
                points.append(vertice1)
                points.append(vertice2)
                points.append(vertice3)
                points.append(vertice4)

        points = np.array(points)
        hull= ConvexHull(points)
        for i in hull.simplices:
            plt.plot(points[i,0], points[i,1], points[i,2], 'k-')
```

```
            Cspace_obstacle[ob] = hull
        ax.scatter(initial_state[0], initial_state[1], initial_state[2],
                   c = 'r', marker ='o', s=100)
        ax.scatter(goal_state[0], goal_state[1], goal_state[2],
                    c = 'g', marker ='o', s=100)
        plt.grid()
        plt.axis('equal')
        plt.show()
        return Cspace_obstacle
```

The returned environment with C-space obstacles shown in C-space (3 states) is shown as follows:

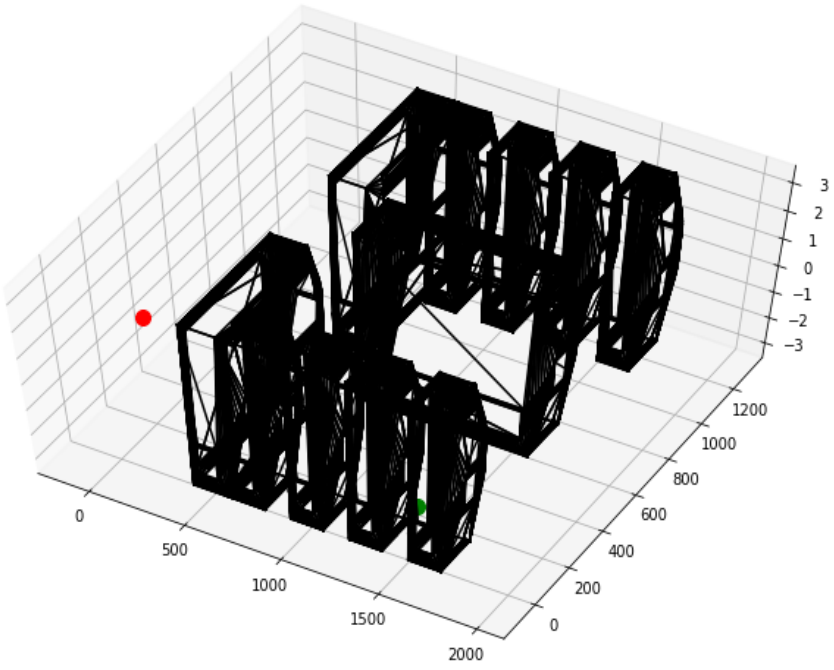

Figure 6: Planning Environment in Configuration Space

2(d). Check Collision Free

With the C-space obstacles generated using ConvexHull, it's easy to check collision-free condition. I just checked every point of a trajectory to see if it is inside the convex hull. If the point set of the previous hull and this new point will lead to a new convex hull, this point is definitely outside the hull which means collision-free. The implementation is:

```
def is_traj_collisionfree(traj, Cspace_obstacles):
    """ Given a trajectory in C-space and the C-space obstacles
    determine whether each sample point
    along this trajectory is outside any obstacle
    Note: Cspace_obstacles is a dir where <ob:ConvexHull>
    """
    collision_free = True
    # for each sample point
    for pnt in traj:
```

```
        # for each convex hull representing the C-space obstacle
        for ob in Cspace_obstacles.keys():
            hull = Cspace_obstacles[ob]
            # Form the new hull adding the point of the taj
            new_hull = ConvexHull(np.concatenate((hull.points, [pnt])))
            # If the hull not changed, indicate this point is inside this hull
            if np.array_equal(new_hull.vertices, hull.vertices):
                collision_free = False
    return collision_free
```

2(e). RRT Planner

Based on previous functions already developed, the implementation of RRT is easy. After initialization of $nodelist$ which is a list of all nodes in the tree, loop random point sampling and tree expanding until the robot can reach the goal. As I said in 2(a), I built the new class $Node$ to represent the node in the tree which kept the extra info about the parent node of one node so that we can use this info to find the final path. Additionally, I used max_step to avoid infinite loop whose default value is 2000.

```
    """ Args:
    initial_state - initial position and heading angle.
    goal_state - goal position and heading angle.
    sample_boundary - the random sample should be within this boundary.
    max_step - the max number of step to avoid infinite loop.
    this function will use RRT algorithm and return a traj
    as a list [[x0,y0,theta0],..., [xg,yg,thetag]]
    """
```

Inside the loop, the first step is that to check the distance between the final node of the tree and target point. If it is under some threshold, we can say that the robot already reach the goal.

```
# Whether can reach the goal
        close_to_goal = np.linalg.norm(goal_state - nodelist[-1].state.copy(),
            ord = 2)
        if close_to_goal < 100:
            reach_goal = True
            break
```

And if not, keep expanding the tree, so sample a random point

```
# Sample a random point
        pnt_random = np.zeros(3)
        for i in range(3):
            pnt_random[i] = np.random.uniform(sample_boundary[i][0],
                sample_boundary[i][1])
        step += 1
```

Then find the nearest node and use this node to expand tree with the function $trajectory\_generation$. If the trajectory is collision-free, add the final point of this trajectory into the tree and assign the parent of this new point to the nearest node.

```
# Find nearest node
      nearestNode, min_index = find_closestnode(nodelist, pnt_random)
      # expand tree
      traj_temp, _ = trajectory_generation(nearestNode.state.copy(),
          pnt_random, mode=0)
      if is_traj_collisionfree(traj_temp, Cspace_ob):
         newNode = Node(traj_temp[-1])
         newNode.parent = min_index
         nodelist.append(newNode)
```

And then just find the path starting from the goal, use its parent info to find last node and iterate until start node. Then trajectory can be generated with this path.

```
   # Find the path with all via points
   path = [goal_state]
   lastIndex = len(nodelist) - 1
   while nodelist[lastIndex].parent is not None:
      node = nodelist[lastIndex]
      path.append(node.state)
      lastIndex = node.parent
   path.append(initial_state)

   # Generate the traj
   traj = []
   i = len(path) - 1
   while i >= 1:
      traj_temp, _ = trajectory_generation(path[i], path[i-1], mode=1)
      traj = traj + traj_temp
      i = i - 1
   return traj, path
```

2(f). Improved RRT Planner

I didn't implement RRT* algorithm here. I just used a small trick to improve RRT planner. I changed the decision condition of reaching the goal. I used trajectory generation function with $mode = 1$ to find the trajectory between the final node of the tree and the goal. If this trajectory is collision free, there's no need to expand the tree at all, just use a long trajectory to reach the goal instead of using 1-sec trajectory every time. Because in fact some node in the tree can already "see" the goal in many cases.

```
# Whether can reach the goal
      traj_temp, _ = trajectory_generation(nodelist[-1].state.copy(),
          goal_state, mode=1)
      if is_traj_collisionfree(traj_temp, Cspace_ob):
         reach_goal = True
         break
```

To visualize the result, the final trajectory is plotted. Not only is the position of the robot shown, the rectangle outline of the robot is shown so that we can visualize how the heading angle changed along the trajectory. The implementation is as follows:

```
def draw_robot(traj):
    start = traj[0]
    end = traj[-1]
    ax = draw_Ospace(start, end, obstaclelist)
    for pnt in traj:
        theta = pnt[2]
        rec_corner = [pnt[0]+70*cos(theta+np.pi)+42.5*cos(theta-np.pi/2),
                      pnt[1]+70*sin(theta+np.pi)+42.5*sin(theta-np.pi/2)]
        robot = plt.Rectangle(rec_corner, 80, 85, np.degrees(theta),
            facecolor='w', edgecolor='b')
        ax.add_patch(robot)
    plt.show()
```

3. Evaluation

3(a). Parking Planning

The first demonstration is head-in parking. I defined the initial state at the position $(100, 100, 90°)$ which is outside the parking lot. And the goal state is $(1500, 40, -90°)$ which is inside the parking lot and belongs to the head-in parking area. Both sides of the goal position are other identical mobile robots.

```
initial_state = np.array([100, 100, np.pi/2])
goal_state = np.array([1500, 40, -np.pi/2])
sample_boundary = [(0, 1800), (0, 1200), (-np.pi, np.pi)]
traj, path = rrt_planner(initial_state, goal_state, sample_boundary,
    max_step=2000)
# head-in parking
draw_robot(traj)
```

The result is shown in Figure 7. The trajectory can avoid obstacle efficiently and can also reach the goal although the trajectory is not optimal. And the process is also computational efficient.

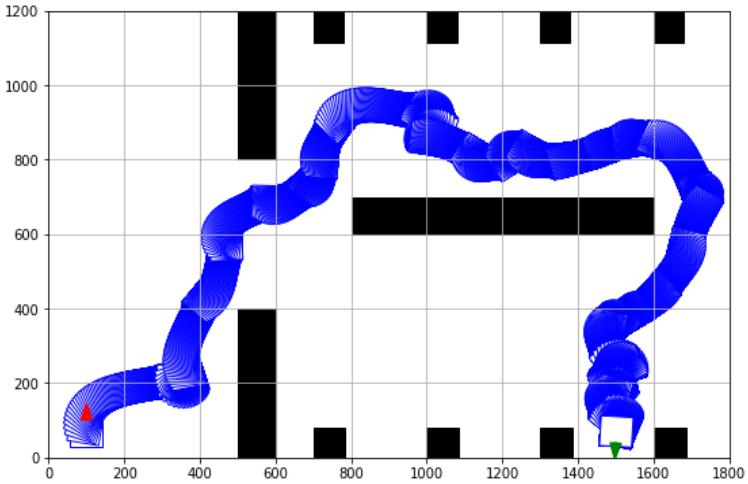

Figure 7: Head-in Parking Planning with RRT

I do the same simulation but with the improved RRT, the result is shown in Figure 8

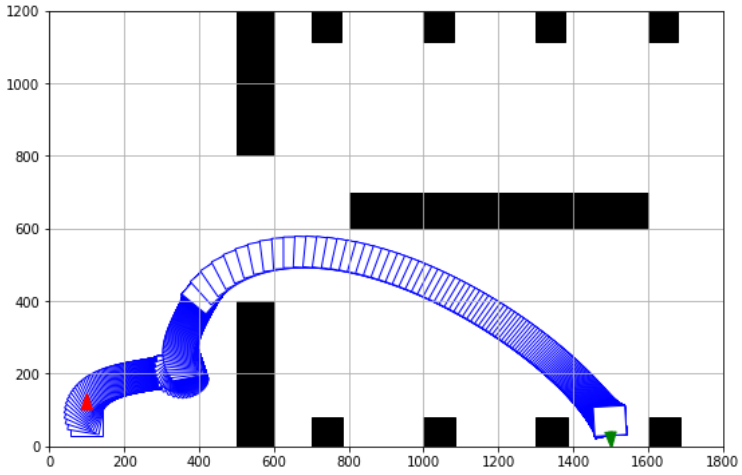

Figure 8: Head-in Parking Planning with improved RRT

The results show that although the trajectory planned by improved RRT is still not optimal (which RRT* guarantee), the trajectory is already very efficient compared to the traditional RRT. And in computational side, the tree of this improved RRT only included 5 nodes while the previous RRT need 183 nodes. And I also simulated the parallel parking where I changed the goal to $(1200, 1160, 0)$.

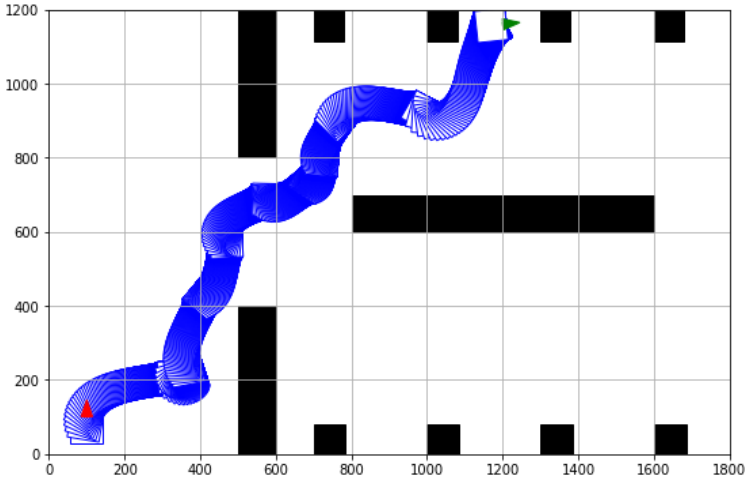

Figure 9: Parallel Parking Planning with improved RRT

3(b). Computational Cost at Various Operations of RRT

In the RRT planner, computational cost mainly came from the final break-out check and tree expanding while the random-point sampling is very efficient. For traditional RRT, the most computational cost is from the tree expanding (to find the final path, the tree expanded 183 nodes). For improved RRT, the most computational cost is from the final break-out check because the check

condition is that a long trajectory is collision-free. To make sure this long trajectory is collision-free, the sampling point along this trajectory cannot be too sparse so that I need to increase the sampling rate when generating this trajectory which will cause many more sample points. This will increase the computational cost during this operation. But for improved RRT, the tree only expanded 5 nodes which save computational cost during tree expanding.

3(c). Conclusions

(1) My planners work very well for both head-in and parallel parking. And the mobile robot can avoid obstacles efficiently.

(2) Although I didn't implement RRT* to get optimal trajectory, I used another way to improve the traditional RRT so that the trajectory is not that messy while keep a good computational efficiency. RRT* algorithm requires to explore almost the entire map and do the rewiring in every loop which cause much more computational cost. But in reality, we just need a comparatively efficient trajectory instead of an absolutely optimal trajectory. I improved the RRT considering this trade-off.

(3) To further improve the planner, we can do the shortcut at the final stage of the planner which is shown in Figure 10. RRT always caused a zig-zag trajectory while actually we can do the shortcut to make the path smoother and more efficient.

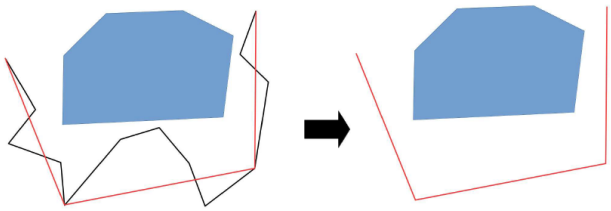

Figure 10: The Shortcut of Final Path