Qiong Hu (405065032)
Zhuyun Xiao (504592699)
Hailin Yu (305231131)

Lab 2 Writeup

EE209AS
Computational Robotics
November 4, 2019

1 Robot model

1(a). What is the dimensionality of the input (action) space, robot state (configuration) space, and operational space?

- Input (action) space: (w_1, w_2, t). w_1, w_2 represent angular velocity of the wheel 1 and 2, respectively (w_1, $w_2 \in$ [-1, 1] RPS). t represents the lasting time of the action. The dimensionality is 3.
- State space: (x, y, θ). (x, y) represents the location of the centerpoint of the wheels and θ refers to the direction of the robot. The dimensionality is 3.
- Operational space: every point (x, y) in a grid map of $m * n$, except obstacles. The dimensionality is 2.

Qiong Hu (405065032)
Zhuyun Xiao (504592699)
Hailin Yu (305231131)

Lab 2 Writeup

EE209AS
Computational Robotics
November 4, 2019

1(b). Define the (continuous space) system model. Assume a fully observable system, i.e. the state itself is directly accessible with no sensor model.

The environment is a rectangular of m*n with several obstacles in it. Each obstacle is also a rectangular.

start state: s0

target state: s1

Algorithm 1: Continuous space system model

```
# Define map
# an m * n rectangular environment
m = 5000   # unit: mm, length of the map
n = 5000   # unit: mm, width of the map

# obstacles: a list to store obstacle states
# obstacles= {obstacle}, obstacle = [x, y, w, h, angle]
# each obstacle [x, y, w, h, angle] is a rectangle, (x, y) is the
    coordinate of the lower-left corner of the obstacle, (w, h) is the
    width and height of the obstacle, angle is the orientation of the
    rectangle
# randomly generate 10 obstacle rectangles
obstacles = []
for i in range(10):
    x = random.uniform(0, m)
    y = random.uniform(0, n)
    w = random.uniform(0, m / 10)
    h = random.uniform(0, n / 10)
    angle = random.uniform(0, 180)
    obstacles.append([x, y, w, h, angle])

# start state: s0 = [x0, y0, theta_0]
s0 = [0, 0, 0]

# target state: st = [xt, yt, theta_t]
# randomly generate a target state
x = random.uniform(0, m)
y = random.uniform(0, n)
z = random.uniform(0, 2 * math.pi)
st = [x, y, z]
```

2 Trajectory planning

2(a). Given a set of points V in C-space and a single target point x_t, define and justify a metric that can be used to determine which of the points in V is closest to x_t.

Because line velocity and angle velocity are different, so we calculate the weighted distance according to the velocity between a point in set V to the target point x_t.

$$d^2 = ((x_i[0] - x_t[0])/v_{max})^2 + ((x_i[1] - x_t[1])/v_{max})^2 + ((x_i[2] - x_t[2])/w_{max})^2$$

Algorithm 2: Metric definition for closest point to target

```
# RRT node
class Node():
    def __init__(self, state):
        self.state = state
        self.path = []
        self.parent = None
        self.t = t

# determine which of the points in V is the closest to xt
def find_closestNode(V, xt):
    # V: a set of RRT points. V = {v}, v = (x, y, theta) in C-space
    # xt: a single target point. xt = (x_t, y_t, theta_t)
    # Return: the closest node to the target point xt inside the set V

    closest_node = V[0]

    # Calculate the weighted distance between a point in set V to the target
      point xt
    # For delta_x: distance between x and x_t divided by the maximum velocity
      of the robot vx_max
    # For delta_y: distance between y and y_t divided by the maximum velocity
      of the robot vy_max
    # For delta_theta: difference between theta and theta_t1 divided by the
      maximum angular velocity of the robot wmax_robot
    # By dividing by the maximum value of each motion, the weighted distance
      of three different dimensions that the robot can reach is normalized.
    weighted_dist = ((V[0].state[0] - xt[0]) / vx_max) ** 2 + ((V[0].state[1]
      - xt[1]) / vy_max) ** 2 + ((V[0].state[2] - xt[2]) / wmax_robot) ** 2

    for point in V:
        dist = ((point.state[0] - xt[0]) / vx_max) ** 2 + ((point.state[1] -
            xt[1]) / vy_max) ** 2 + ((point.state[2] - xt[2]) / wmax_robot) **
            2
        if dist < weighted_dist:
            closest_node = point
            weighted_dist = dist
    return closest_node
```

2(b). Given arbitrary initial robot state x_i and target robot state x_t (in C-space), generate a smooth achievable trajectory (that matches the metric you defined above) from the x_i towards x_t lasting 1 second. What are the control inputs for this trajectory?
We divide the trajectory into three steps, first is to rotate the robot facing the target point, second it to go straight to the target and last is to rotate the robot facing the target point direction. The whole process is shown in Fig. 1. Fig.2 shows how to rotate in step 1.

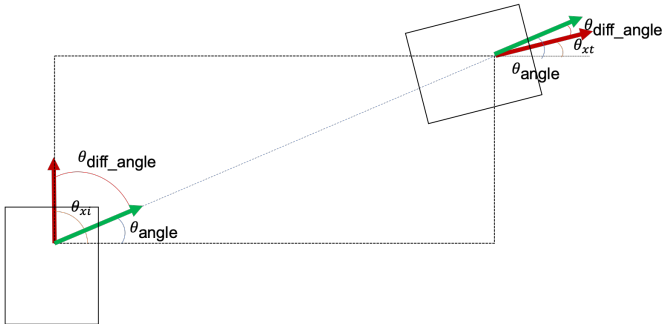

Figure 1: a smooth achievable trajectory of the robot

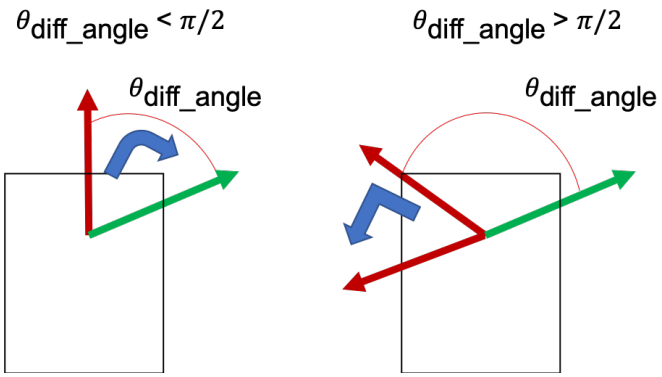

Figure 2: how the robot rotate in step 1

The control inputs for this trajectory are (w_1, w_2, t). For example, if xi = [0,0,0] and xt = [100,100,0], the control inputs are [(-1, 1, 0.225), (1, 1, 0.0900), (-1, 1, 0.225)]

Algorithm 3: Pseudocode of Achievable Trajectory Generation for 1 sec path

```
# three-step policy:
# 1, rotate from the initial orientation to the angle that points to (or is
    back to) the target position
# 2, go straight until arrive at the target position
# 3, rotate until the orientation is the same as the target state
# 4, if one-second time is not enough to accomplish all three steps, stop
    wherever when running out of time
def generate_trajectory(xi, xt):
```

```
# Given the initial and target robot state: xi, xt in C-space.
# Return: an achievable trajectory

# a list to store each point in the trajectory within one second
trajectory = []
trajectory.append(xi)

# control inputs of two wheels
inputs = []

# remaining time within one second, updated after each step
remain_time = 1

if remain_time>0:
    do step 1, remain_time -= time_need
    add end state into trajectory
    add control input into inputs
    if remain_time>0:
        do step 2, remain_time -= time_need
        add end state into trajectory
        add control input into inputs
        if remain_time>0:
            do step 3, remain_time -= time_need
            add end state into trajectory
            add control input into inputs
```

2(c). Given a smooth robot trajectory in C-space and obstacles defined in operational space, determine whether this trajectory is collision free.

Since the robot and obstacles are all rectangles. So we only need to check whether robot rectangle is collision with obstacle rectangle during every point in the trajectory.

For checking whether two rectangles are collision, we check if any two sides of them respectively are intersect.

Fig. 3 shows how to calculate vertexes of the robot.

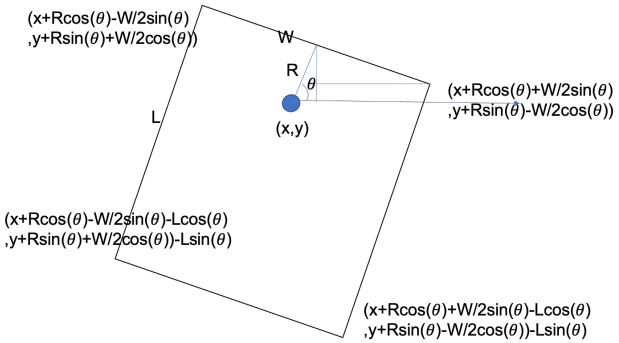

Figure 3: a smooth achievable trajectory of the robot

Algorithm 4: Pseudocode of Determine collision free trajectory

```
# determine whether trajectory is collision free.
def isCollisionTrajectory(trajectory, obstacles):
    for each state in trajectory:
        for each obstacle in obstacles:
            check whether robot is collision with obstacle
```

2(d). Combine the above to implement an RRT planner generating a trajectory from a specified initial state to a desired goal region. Visualize the evolution of the RRT.

The following are some results for our RRT algorithm: Fig.4, Fig.5, Fig.6, Fig.7, Fig.**??**, Fig.**??**, Fig.8, Fig.9, Fig.10, Fig.11.

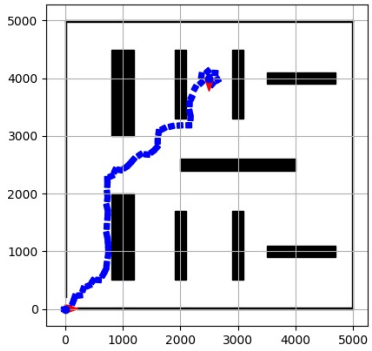

Figure 4: target state = (2500, 4000, -π/2)

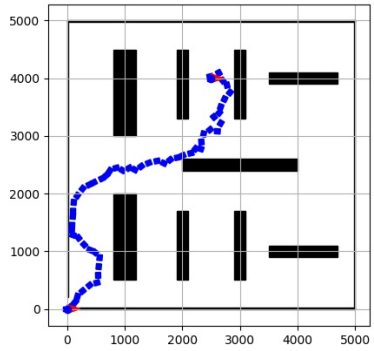

Figure 5: target state = (2500, 4000, 0)

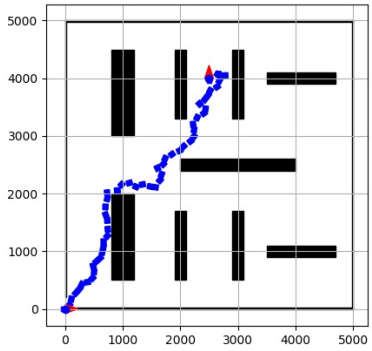

Figure 6: target state = (2500, 4000, π/2)

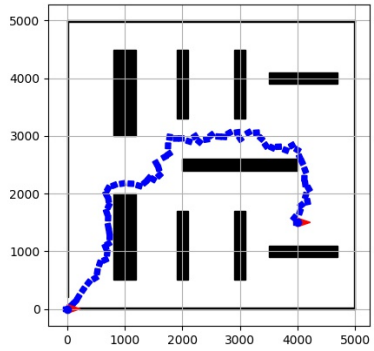

Figure 7: target state = (4000, 1500, 0)

Algorithm 5: Pseudocode of RRT planner for trajectory generation

```
def RRT(s0, s1, obstacles):
    # V used to store all possible states
    V = [s0]
    state = s0
    # interation until find a path
    while state!= s1:
        randomly choose a state st in operational space
        determine which state si in V is closest to st
        generate a trajectory from si to st, the end of trajectory is state
```

```
        if trajectory is collision free:
            add state into V
            state.parent = si
    #when find a path,use trajectory to restore it
    trajectory = []
    end = s1
    while end !=s0:
        add end into trajectory
        end = end.parent
```

Algorithm 6: Visualize the RRT evolution

```
    # For visualization: plot trajectory, obstacles, and points that represent
        the robot
def plotTrajectory(trajectory, ax):
    for point in trajectory:
        # plot robot
        x = point[0] + R*math.cos(point[2]) + W/2 * math.sin(point[2]) - L *
            math.cos(point[2])
        y = point[1] + R*math.sin(point[2]) - W/2 * math.cos(point[2]) - L *
            math.sin(point[2])
        rec = plt.Rectangle((x, y), L, W, angle = point[2]*180/math.pi, color
            = 'b')
        ax.add_patch(rec)
    return ax

def plotObstacles(obstacles, ax):
        for obstacle in obstacles:
                rec = plt.Rectangle((obstacle[0],obstacle[1]), obstacle[2],
                    obstacle[3], angle = obstacle[4], color = 'k')
                ax.add_patch(rec)
        return ax

def plotPoint(point,ax):
    # plot arrow
    plt.arrow(point[0], point[1], 0.5*np.cos(point[2]),0.5*np.sin(point[2]),
        color='r', width=m/100)
    # plot center point
    plt.plot(point[0], point[1],'bo')
    # plot robot
    x = point[0] + R * math.cos(point[2]) +W / 2*math.sin(point[2]) - L * math
        .cos(point[2])
    y = point[1] + R * math.sin(point[2]) - W / 2*math.cos(point[2]) - L *
        math.sin(point[2])
    rec = plt.Rectangle((x, y), L, W, angle = point[2]*180 / math.pi, color =
        'b')
    ax.add_patch(rec)
    return ax
```

3 Evaluation and Extensions

3(a). Run some examples that demonstrate the performance (in terms of computational efficiency, trajectory efficiency, and obstacle avoidance) of your planner as your robot tries to achieve various goals (such as head-in parking and parallel parking between other such parked vehicles). Clearly describe the experiments that were run, the data that was gathered, and the process by which you use that data to characterize the performance of your planner. Include figures; you may also refer to animations uploaded to your git repo.

We use three metrics to demonstrate the performance of RRT. They are the running time of RRT, the number of nodes needed of RRT and the robot running time of trajectory RRT has found.

For experiments set up, we fixed obstacles as following:

obstacles.append([800, 500, 400, 1500, 0])
obstacles.append([800, 3000, 400, 1500, 0])
obstacles.append([1900, 500, 200, 1200, 0])
obstacles.append([2900, 500, 200, 1200, 0])
obstacles.append([1900, 3300, 200, 1200, 0])
obstacles.append([2900, 3300, 200, 1200, 0])
obstacles.append([2000, 2600, 200, 2000, -90])
obstacles.append([3500, 4100, 200, 1200, -90])
obstacles.append([3500, 1100, 200, 1200, -90])
Also, we fixed start state (0,0,0)
We try 10 different target state:[[2500, 4000, π / 2], [2500, 4000, π / 2], [2500, 4000, 0], [2500, 1000, π / 2], [2500, 1000, π / 2], [2500, 1000, 0], [4000, 500, 0], [4000, 1500, 0], [4000, 3500, 0], [4000, 4500, 0]]

Some results are shown in the following figures:

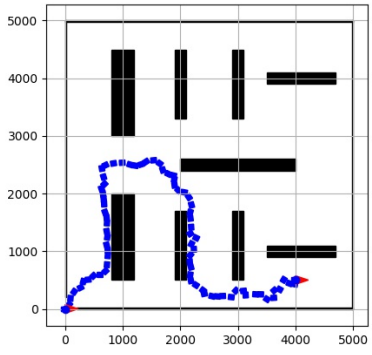

Figure 8: target state = (4000, 500, 0)

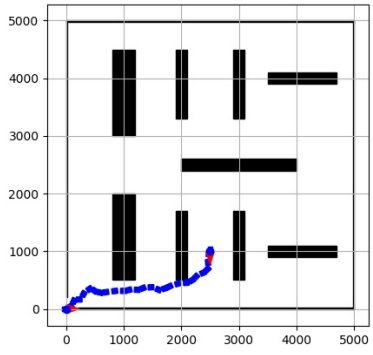

Figure 9: target state = (2500, 1000, -π/2)

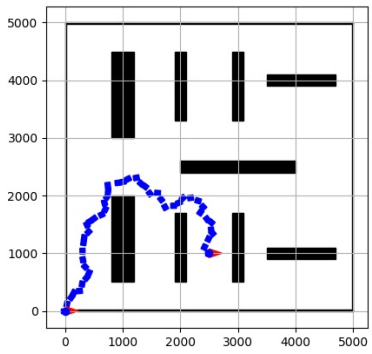

Figure 10: target state = (2500, 1000, 0)

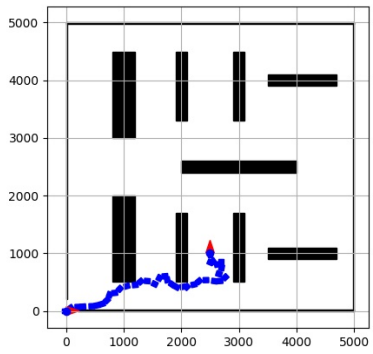

Figure 11: target state = (2500, 1000, $\pi/2$)

The results of performance are shown in the following table.

algorithm type	target state	run time	number of searched nodes	trajectory time
RRT	[2500, 4000, π]	4.31	326	37.3
RRT	[2500, 4000, -π]	4.88	334	38.1
RRT	[2500, 4000, 0]	4.67	315	43.6
RRT	[2500, 1000, π]	23.7	1498	23.7
RRT	[2500, 1000, -π]	0.8	56	19.2
RRT	[2500, 1000, 0]	14.3	985	34.3
RRT	[4000, 500, 0]	26.6	1650	54.6
RRT	[4000, 1500, 0]	3.0	220	51.2
RRT	[4000, 3500, 0]	21.3	1326	61.2
RRT	[4000, 4500, 0]	48.5	2701	49.56

Algorithm 7: RRT Performance Testing

```
#test performance of RRT
def RRT_test(s0, s1, obstacles):
    start = time.time()
    trajectory,V = RRT(s0,s1,obstacles)
    end = time.time()
    # Caculate time RRT need to find a trajectory
    print(end - start)
    # Caculate number of nodes need to find a trajectory
```

```
    space = len(V)
    print(space)
    #Caculate time robot need to go for a trajectory
    t = 0
    for i in range(len(trajectory) - 1):
        dx = trajectory[i + 1][0] - trajectory[i][0]
        dy = trajectory[i + 1][1] - trajectory[i][1]
        dz = trajectory[i + 1][2] - trajectory[i][2]
        t += math.sqrt(dx ** 2 + dy ** 2) / vx_max + dz / wmax_robot
    print (t)
    #plot figure
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    # plot start and goal
    plotpoint(s0, ax)
    plotpoint(s1, ax)
    # plot trajectory
    plottrajectory(trajectory, ax)
    # plot obstacles
    plotobstacles(obstacles, ax)
    # plot grid
    plt.xlim((-m / 10, m))
    plt.ylim((-n / 10, n))
    plt.grid()
    plt.show()
```

3(b). How much relative computational cost is associated with the various operations of the RRT planner?

1.Randomly choose a state st in operational space : $2 * 10^{-6}$ s
2.Determine which state si in V is closest to st: $1 * 10^{-6}*$ number of nodes in V
3.Generate a trajectory from si to st : $3 * 10^{-5}$ s
4.Determine whether trajectory is collision free: 0.01 s

3(c). Improve on your planner using the RRT* algorithm, and compare to your original RRT planner using the above metrics.

We use a node.t to track time robot needed to go from start state to the node state. So for every iteration, we check whether the new node can be a better parent or has a better parent with smaller node.t.

We have compared results of RRT and RRT* in following figures, in the figures, red path refer to RRT* and blue path refer to RRT:

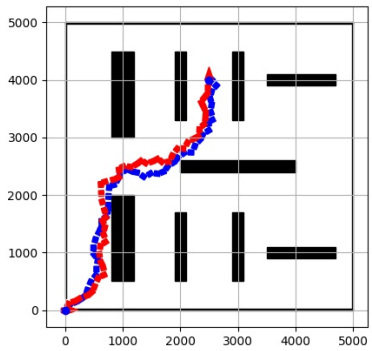

Figure 12: RRT vs RRT*, blue path is for RRT and red path is for RRT*

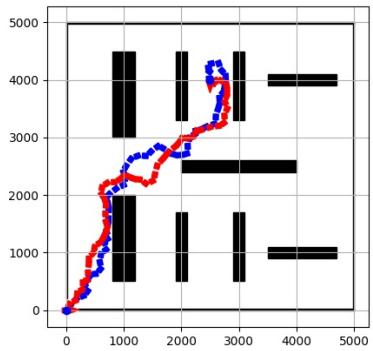

Figure 13: RRT vs RRT*, blue path is for RRT and red path is for RRT*

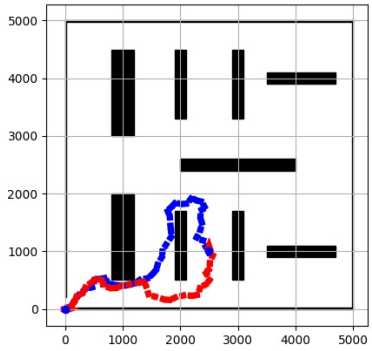

Figure 14: RRT vs RRT*, blue path is for RRT and red path is for RRT*

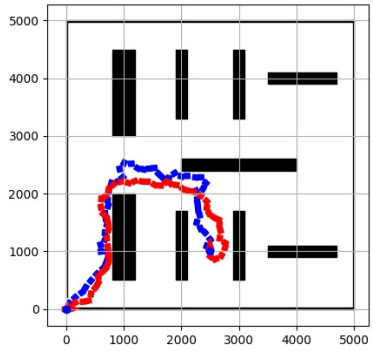

Figure 15: target state = RRT vs RRT*, blue path is for RRT and red path is for RRT*

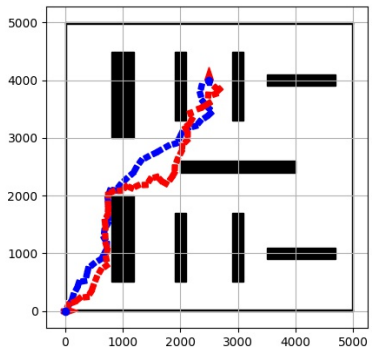

Figure 16: RRT vs RRT*, blue path is for RRT and red path is for RRT*

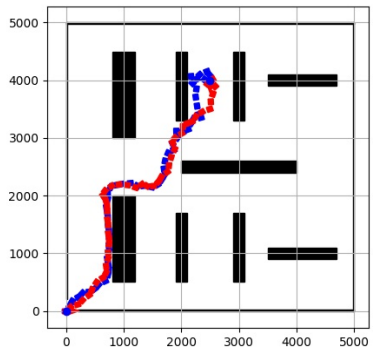

Figure 17: RRT vs RRT*, blue path is for RRT and red path is for RRT*

Algorithm 8: Pseudocode of RRT*

```
def RRT*(s0, s1, obstacles):
    # V used to store all possible states
    V = [s0]
    state = s0
    # interation until find a path
    while state!= s1:
        randomly choose a state st in operational space
        determine which state si in V is closest to st
        generate a trajectory from si to st, the end of trajectory is state
        if trajectory is collision free:
            add state into V
            state.parent = si
            state.t = si.t + time need for trajectory
            check if state has a better parent
            check if state can be a better parent

    #when find a path,use trajectory to restore it
    trajectory = []
    end = s1
    while end !=s0:
        add end into trajectory
        end = end.parent
```

3(d). Qualitatively describe some conclusions about the effectiveness of your planner for potential tasks your robot may encounter. For example, what happens to your planner in the presence of process noise, i.e. a stochastic system model? How might you modify your algorithm to better handle noise?

The results of performance of RRT vs RRT* are shown in the following table. Usually, RRT* takes more run time than RRT and has a smaller trajectory time, however, because of randomness, sometimes it might be worse than RRT. So, for future improvement, we will add a minium threshold steps for the RRT*, as a result, it will get a better path in most situations.

algorithm type	target state	run time	number of searched nodes	trajectory time
RRT	[2500, 4000, π]	2.45	125	37.1
RRT*	[2500, 4000, π]	1.36	128	39.2
RRT	[2500, 4000, -π]	11.1	790	39.2
RRT*	[2500, 4000, -π]	26.5	1329	37.6
RRT	[2500, 4000, 0]	4.67	315	43.6
RRT*	[2500, 4000, 0]	25.2	1562	38.8
RRT	[2500, 1000, π]	23.7	1498	23.7
RRT*	[2500, 1000, π]	16.7	1159	24.3
RRT	[2500, 1000, -π]	0.8	56	19.2
RRT*	[2500, 1000, -π]	13.4	936	18.2
RRT	[2500, 1000, 0]	14.3	985	34.3
RRT*	[2500, 1000, 0]	19.7	1202	34.6