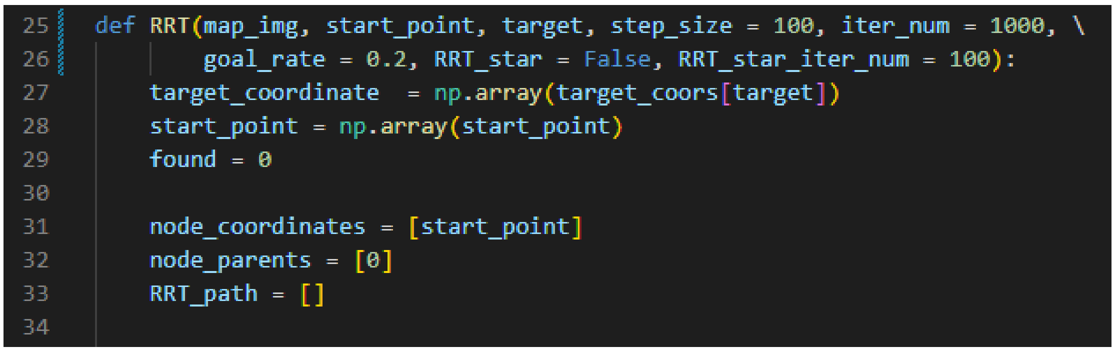
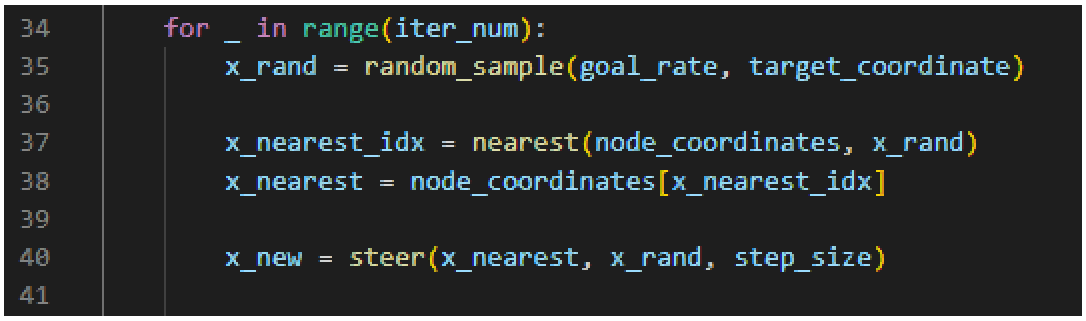
i. Code

a. how do you do RRT algorithm

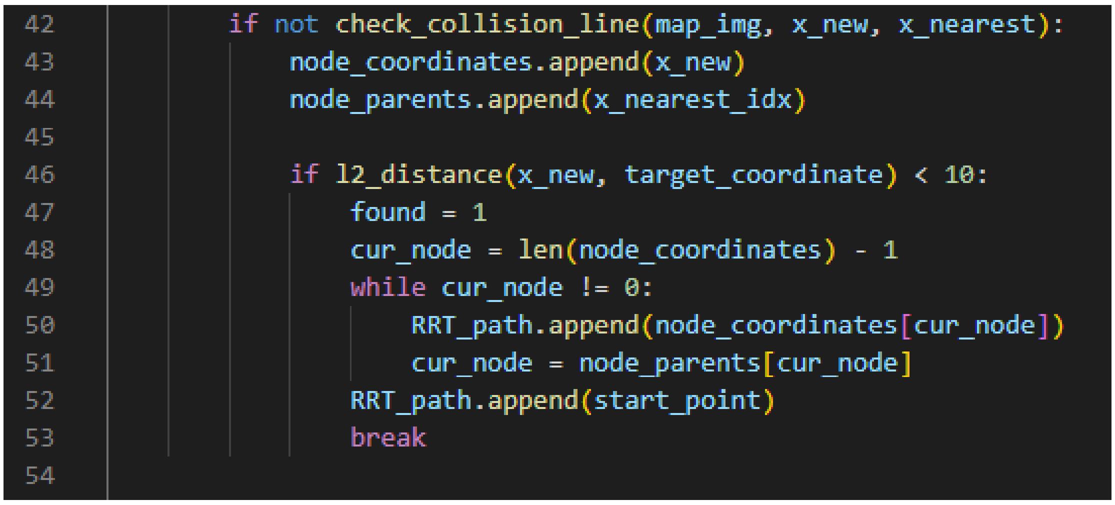

```python
def RRT(map_img, start_point, target, step_size = 100, iter_num = 1000, \
        goal_rate = 0.2, RRT_star = False, RRT_star_iter_num = 100):
    target_coordinate  = np.array(target_coors[target])
    start_point = np.array(start_point)
    found = 0

    node_coordinates = [start_point]
    node_parents = [0]
    RRT_path = []

```

In the RRT function, I will take several inputs. “map_img” is the scatter point cloud image of that scene. “start_point” is the coordinate of starting point. “target” is the name of target category. “step_size” is the step size that use in RRT. “iter_num” is the number of iteration in RRT. “goal_rate” is the probability that RRT choose the target point to be the x_rand. “RRT_star” to indicate whether to use RRT*. “RRT_star_iter_num” is the number of iteration in RRT*.

In the function, I first get the coordinate of target point. In the RRT tree structure, I only record each nodes’ coordinate and its parent since I can just backtrack on the target point to get the path on RRT tree.

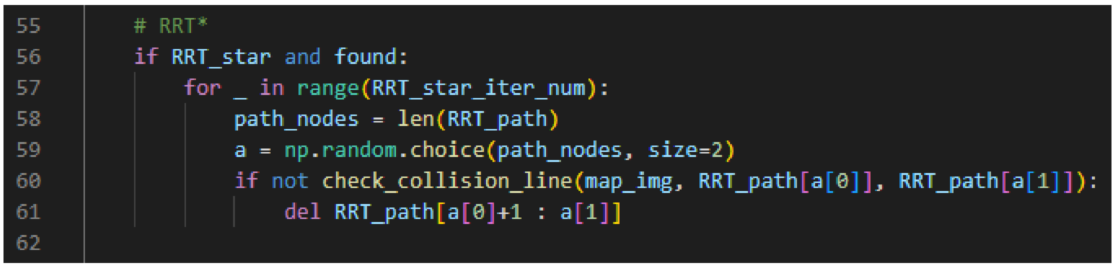

```python
    for _ in range(iter_num):
        x_rand = random_sample(goal_rate, target_coordinate)

        x_nearest_idx = nearest(node_coordinates, x_rand)
        x_nearest = node_coordinates[x_nearest_idx]

        x_new = steer(x_nearest, x_rand, step_size)

```

In the iteration of RRT, I first random sample a point on the image. Then, find the nearest point on the RRT tree. And use “steer” to get “x_new” from “x_nearest” by a step size.



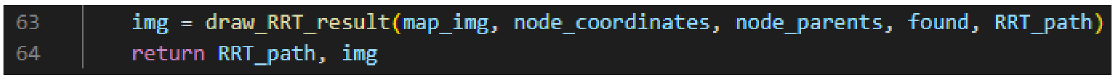
```python
        if not check_collision_line(map_img, x_new, x_nearest):
            node_coordinates.append(x_new)
            node_parents.append(x_nearest_idx)

            if l2_distance(x_new, target_coordinate) < 10:
                found = 1
                cur_node = len(node_coordinates) - 1
                while cur_node != 0:
                    RRT_path.append(node_coordinates[cur_node])
                    cur_node = node_parents[cur_node]
                RRT_path.append(start_point)
                break

```

Then, check if the line between x_new and x_nearest collide on obstacles. If yes, continue to next iteration. Else, I would add the “x_new” into the RRT tree. I would also check if the current node reaches the target. If it reaches the target I would backtrack the nodes on RRT tree and save them to the RRT path.



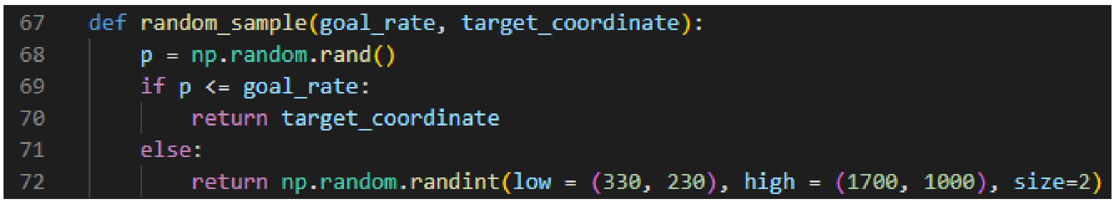
```python
    # RRT*
    if RRT_star and found:
        for _ in range(RRT_star_iter_num):
            path_nodes = len(RRT_path)
            a = np.random.choice(path_nodes, size=2)
            if not check_collision_line(map_img, RRT_path[a[0]], RRT_path[a[1]]):
                del RRT_path[a[0]+1 : a[1]]

```

If we use RRT* algorithm, I would repeatedly pick two nodes at random and check whether they can be connected by a straight line without collision. If yes, I would delete the node between the 2 points.



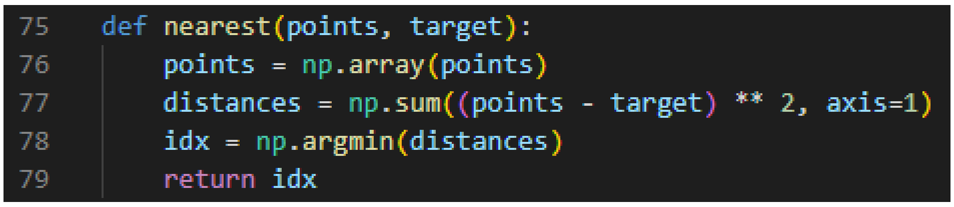
```python
    img = draw_RRT_result(map_img, node_coordinates, node_parents, found, RRT_path)
    return RRT_path, img
```

In the end, I will draw the RRT result on the map image. Then, return the RRT path and that image.



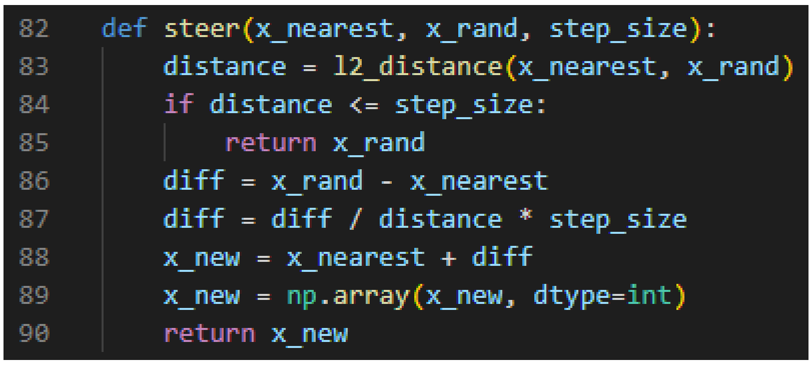
```python
def random_sample(goal_rate, target_coordinate):
    p = np.random.rand()
    if p <= goal_rate:
        return target_coordinate
    else:
        return np.random.randint(low = (330, 230), high = (1700, 1000), size=2)
```

In the “random_sample” function, it would return the target coordinate or the random point on the map by a probability.



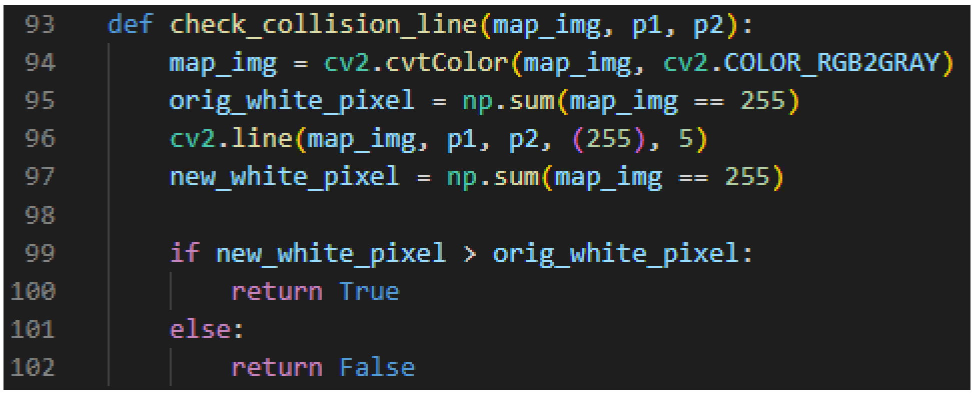
```python
def nearest(points, target):
    points = np.array(points)
    distances = np.sum((points - target) ** 2, axis=1)
    idx = np.argmin(distances)
    return idx
```

In the “nearest” function, I would calculate the L2 distance between a point set and a target point. Then, return the index that is closet to the target.



```python
def steer(x_nearest, x_rand, step_size):
    distance = l2_distance(x_nearest, x_rand)
    if distance <= step_size:
        return x_rand
    diff = x_rand - x_nearest
    diff = diff / distance * step_size
    x_new = x_nearest + diff
    x_new = np.array(x_new, dtype=int)
    return x_new
```

In the “steer” function, I will calculate the L2 distance between these 2 points. If the distance is less than the step size, the x_new would be directly the input x_rand. If not, the x_new would be the step size length from x_nearest to x_rand.



```python
def check_collision_line(map_img, p1, p2):
    map_img = cv2.cvtColor(map_img, cv2.COLOR_RGB2GRAY)
    orig_white_pixel = np.sum(map_img == 255)
    cv2.line(map_img, p1, p2, (255), 5)
    new_white_pixel = np.sum(map_img == 255)

    if new_white_pixel > orig_white_pixel:
        return True
    else:
        return False
```

In the “check_collision_line” function, since the number of obstacle points is too many, I use a tricky way to detect collision. First, I would turn the map image to gray scale. Then, I would draw a white line with a thickness between 2 points. And, I would count the number of white pixels before and after. Compare these 2 numbers, if the number of white pixels is greater than before, it means the white line covered some colored points (obstacles). That is, collision occurred.

b. how do you convert route to discrete actions

```
agent_state = agent.get_state()
sensor_state = agent_state.sensor_states['color_sensor']
cur_pos = np.array([sensor_state.position[0], sensor_state.position[2]])
w, x, y, z = sensor_state.rotation.w, sensor_state.rotation.x, sensor_state.rotation.y, sensor_state.rotation.z
```

First, I would get the agent's position and rotation.

```
face_angle = np.rad2deg(np.arccos(w)) * 2
```

```
next_target = path_on_habitat[cur_node]
diff = next_target - cur_pos
target_angle = np.rad2deg(np.arctan2(diff[0], diff[1]))
```

Then, get agent's yaw angle and the angle of vector from agent to target.

```
diff_angle = face_angle - target_angle
```

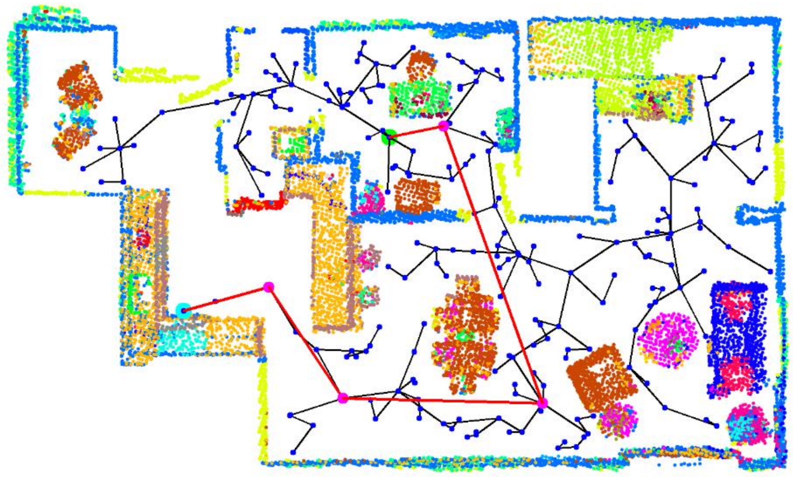
```
if diff_angle > 2:
    action = "turn_right"
    frame = navigateAndSee(action)
    #print("action: RIGHT")
    out.write(frame)
elif diff_angle < -2:
    action = "turn_left"
    frame = navigateAndSee(action)
    #print("action: LEFT")
    out.write(frame)
else:
    action = "move_forward"
    frame = navigateAndSee(action)
    #print("action: FORWARD")
    out.write(frame)
```

Calculate the angle difference. If the difference is larger than 2, the agent would turn left or turn right. If the difference is less than 2, the agent would move forward.

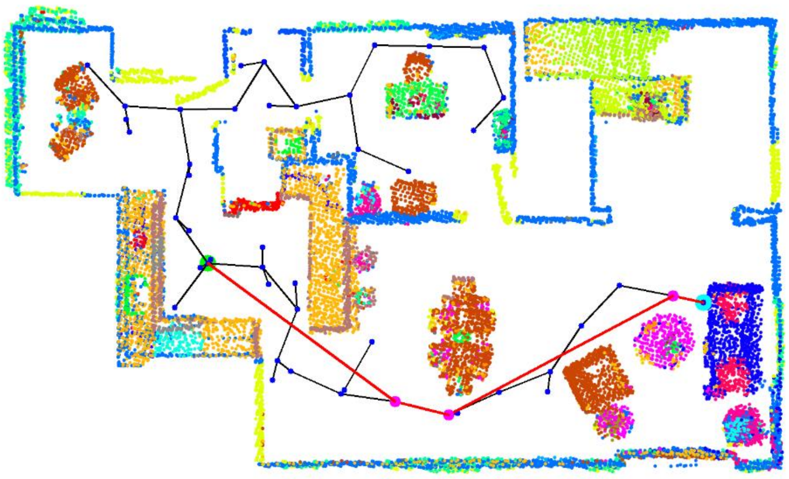
ii. Results and Discussion

1. Show and discuss the results from the RRT algorithm with different start points and targets.

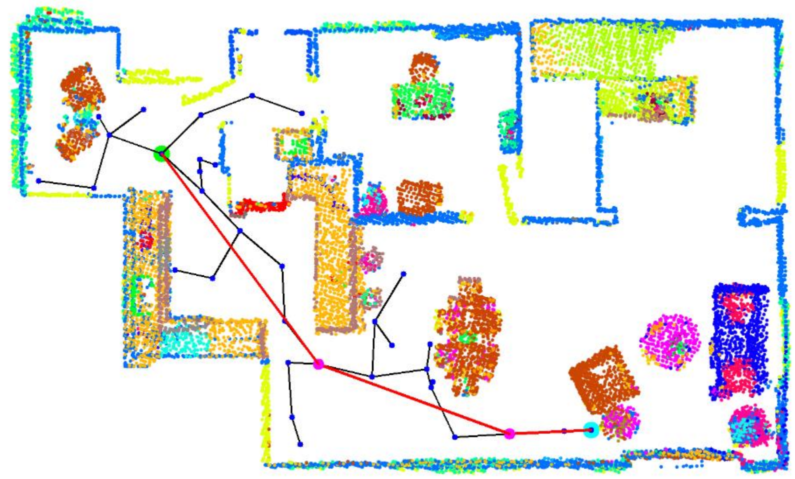
Cooktop:



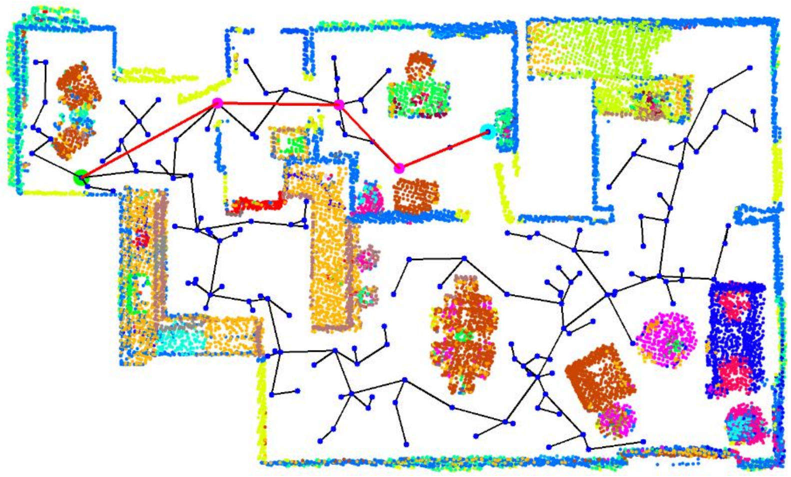
Cusion:



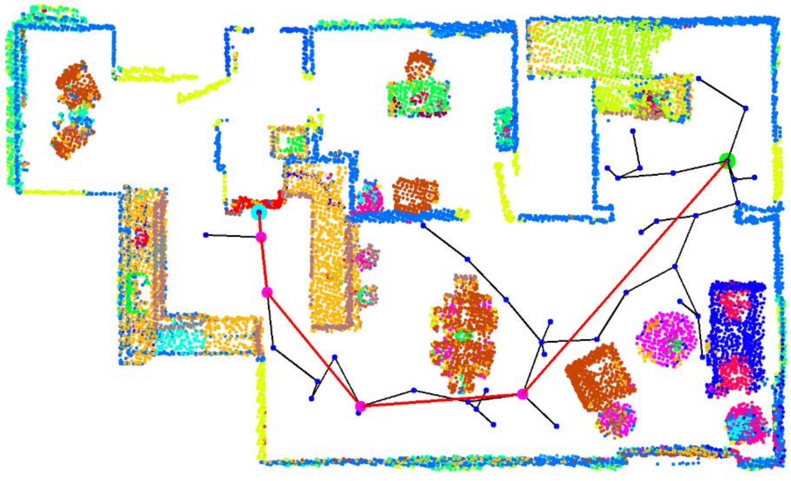
Lamp:



Rack:



Refrigerator:



From the results above, we can find that my implemented RRT algorithm can find a path from selected start point to the target.

I found that if the starting point is located at a narrow place (like the cooktop and rack ones), then the RRT needs more effort to find the target. These 2 examples spread more nodes on the map than the others.

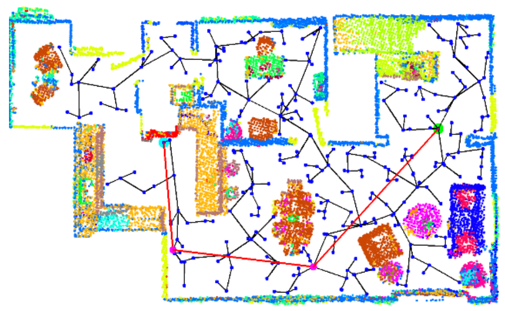
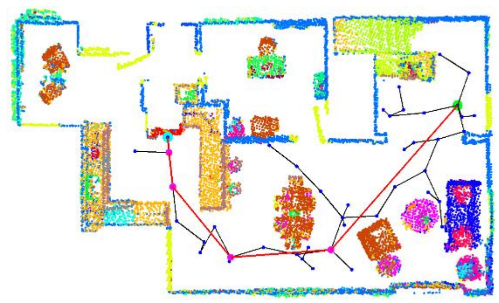
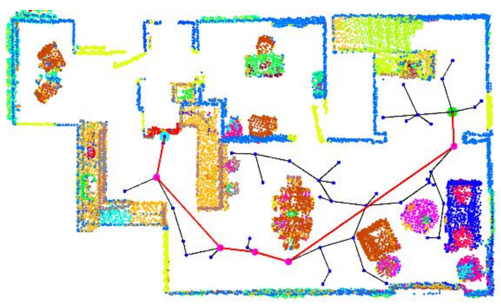
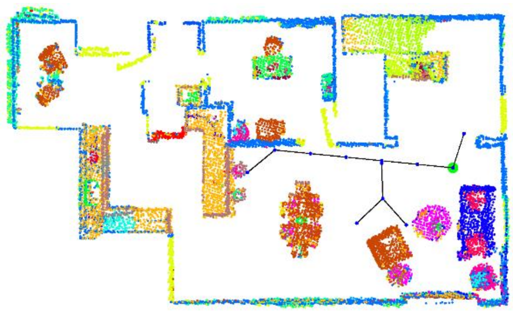
2. Discuss about the robot navigation results

In most situations, my implemented robot navigation can reach the target successfully, just like the result videos.

However, the result is not always perfect. For example, in “cooktop.mp4” 0:04, the agent is stuck on the wall for 1 second. That is because the volume of agent. This problem is easy to solve. Just modify the collision detection in RRT. In my implementation, I can just adjust the thickness of the line I draw. Making it thicker, then it would make less collision. But it would cause the number of iteration that RRT need to find the path to target.

3. Anything you want to discuss

I test the probability that “random_sample” function choose the target point.

Probability: 0.0 (986 iterations)	Probability: 0.2 (275 iterations)
	
Probability: 0.5 (358 iterations)	Probability:0.99 (1000 iterations more)
	

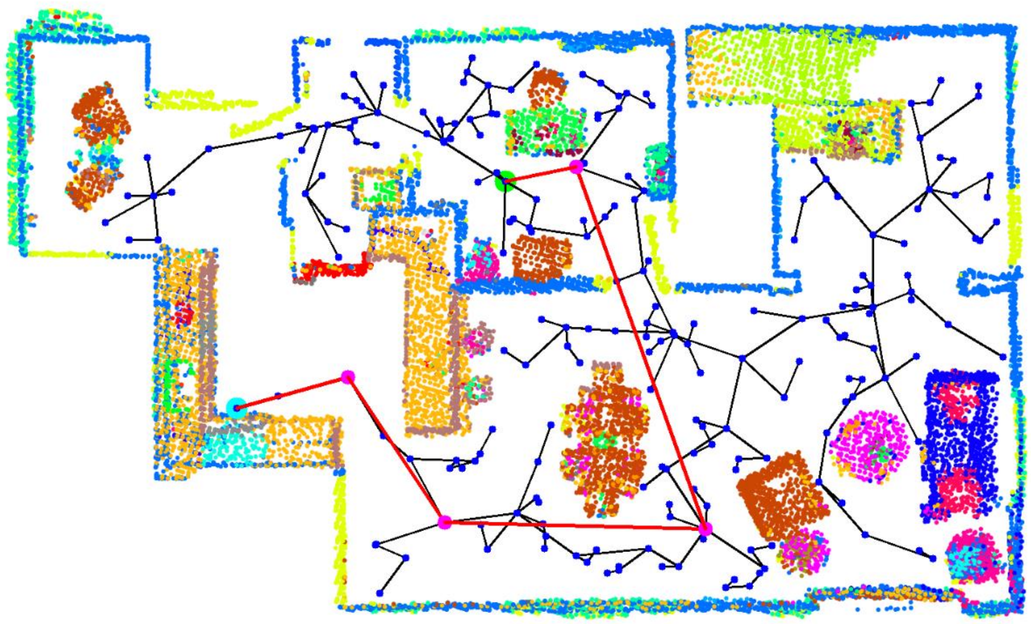
These result, I ran for 10 times for each condition. The number of iterations is the average number. When the probability is 0, we can see that the RRT need much more iteration to reach the target since it is totally random. With some probability to choose the target point, the RRT can reach the target point much faster. However, with the probability increase, the speed to reach target is becoming slower. When the probability is 0.99, it cannot reach the target.

Bonus

RRT*:

I use RRT* to improve the performance. After found a path to the target, I would repeatedly pick 2 nodes randomly to see if the can connect together without collision.

For example:



From the example we can see that there are only 6 nodes in the pruned path. However, there are more than 20 nodes in the original path. The RRT* reduce the distance on RRT path.

iii. Reference

Conversion between quaternions and Euler angles - Wikipedia
-[https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles]