i. Code

1. RRT implementation

   Reference: https://iter01.com/42235.html

   First of all, I define some variables that will be needed in the following procedure. The variable "RRTtree" will be used to store all the nodes generated in RRT algorithm. It will be a 2d-array. Each row represents a node. The first two columns record the node coordinate, and the third column record the id (the row id of "RRTtree") of its parent node. There is no parent node the starting node, so id of its parent is "-1". The variable "fail_attemp" records how many tries the algorithm cannot add a new node into RRTtree. When the number of failed tries exceeds "max_fail_attemp", RRT algorithm ends.

```python
RRTtree = np.array([[start_y, start_x, -1]])    # y -> row, x -> column, index of parent (start point is the root, no parent, parent id = -1)
RRTtree = RRTtree.astype(int)
threshold = 20                                  # nodes closer than this threshold are taken as almost the same
fail_attemp = 0
max_fail_attemp = 1500
path_found = False
path = []
```

   In each iteration to find a new node which can be added to RRTtree, first, with probability of 0.7, I will generate a random node; with probability of 0.3, I select the target node so that RRTtree can grow toward the target more quickly. Then, I will find a node which is already in RRTtree and has the minimum distance with the (random/target) node I selected earlier (I will call it random node below).

```python
# while fail_attemp <= max_fail_attemp:
for i in range(max_fail_attemp):

    print(i)
    # with probability = 0.3 to pick the target
    if np.random.rand() < 0.7:
        rand_node =  np.multiply(np.random.rand(1, 2)[0], np.array([map.shape[0], map.shape[1]]))      # map.shape = (480, 640, 3)
        rand_node = rand_node.astype(int)
    else:
        rand_node = np.array([target_y, target_x])
        rand_node = rand_node.astype(int)

    # select the node in the RRT tree that is closest to the random node
    nearest_node, nearest_id, nearest_dis = find_nearest_node(rand_node)
```

   The way to find the nearest node in RRTtree is simply to calculate the least L2 distance.

```python
def find_nearest_node(node):

    global RRTtree

    min_dis = np.inf
    for i in range(RRTtree.shape[0]):
        dis = np.linalg.norm(RRTtree[i, 0:2]-node)
        if dis < min_dis:
            min_dis = dis
            nearest_id = i

    nearest_node = RRTtree[nearest_id, 0:2]
    return nearest_node, nearest_id, min_dis
```

   After finding the nearest node in RRTtree, I will find a new node which is 20 units far away from the nearest node and is located along the direction toward the random node. Use function "atan2", I can easily know the angle between the direction toward the random node and x (or z) axis according to your definition. Just use this calculated "theta" and functions like "sin" and "cos", we can compute the coordinate of the new node.

Then I will check whether there is no any obstacle between the nearest node and this new node. If any obstacle is detected, this iteration is a failed try to add a new node into RRTtree.

```
        step_size = 20
        theta = math.atan2(rand_node[0]-nearest_node[0], rand_node[1]-nearest_node[1])
        new_node = nearest_node + step_size * np.array([math.sin(theta), math.cos(theta)])
        new_node = new_node.astype(int)

        # check obstacle free
        if(not check_obstacle_free(nearest_node, new_node)):
            # print('fail')
            fail_attemp = fail_attemp + 1
            continue
```

To check whether there is any obstacle between the nearest node and the new node, I just simply check whether all pixels between the nearest node and the new node have white color. The way to calculate all the coordinate between the nearest node and the new node is as the way that I calculate the coordinate of the new node, which I have mentioned earlier. The only difference is that not only the coordinate of the 20-unit-far-away point is calculated, all the coordinates of points which are 0-to-20-unit (incremented by 0.5 unit) fay away from the nearest node are calculated.

```
def check_obstacle_free(n0, n1):
    # n0 -> nearest_node
    # n1 -> new_node
    dir = math.atan2(n1[0]-n0[0], n1[1]-n0[1])

    # print(np.arange(0, np.linalg.norm(n1-n0), 0.5))

    for r in np.arange(0, np.linalg.norm(n1-n0), 0.5):

        check_node = n0 + r * np.array([math.sin(dir), math.cos(dir)])

        free1 = (map[int(np.ceil(check_node[0])), int(np.ceil(check_node[1]))]==np.array([255, 255, 255])).all()
        free2 = (map[int(np.ceil(check_node[0])), int(np.floor(check_node[1]))]==np.array([255, 255, 255])).all()
        free3 = (map[int(np.floor(check_node[0])), int(np.ceil(check_node[1]))]==np.array([255, 255, 255])).all()
        free4 = (map[int(np.floor(check_node[0])), int(np.floor(check_node[1]))]==np.array([255, 255, 255])).all()

        if((not free1) or (not free2) or (not free3) or (not free4)):
            return False

    free1 = (map[int(np.ceil(n1[0])), int(np.ceil(n1[1]))]==np.array([255, 255, 255])).all()
    free2 = (map[int(np.ceil(n1[0])), int(np.floor(n1[1]))]==np.array([255, 255, 255])).all()
    free3 = (map[int(np.floor(n1[0])), int(np.ceil(n1[1]))]==np.array([255, 255, 255])).all()
    free4 = (map[int(np.floor(n1[0])), int(np.floor(n1[1]))]==np.array([255, 255, 255])).all()

    if((not free1) or (not free2) or (not free3) or (not free4)):
        return False

    return True
```

If there is no any obstacle between the nearest node and the new node, we can add the new node into RRTtree. The nearest node is assigned as the parent of the new node.

```
        RRTtree = np.append(RRTtree, np.array([[new_node[0], new_node[1], nearest_id]]), axis=0)

        # cv2.line(影像, 開始座標, 結束座標, 顏色, 線條寬度)
        cv2.line(map_show, (new_node[1], new_node[0]), (nearest_node[1], nearest_node[0]), (0, 0, 0), 1)
        # cv2.circle(影像, 圓心座標, 半徑, 顏色, 線條寬度)
        cv2.circle(map_show, (new_node[1], new_node[0]), 2, (255,200,0), 1)
        cv2.imshow('map', map_show)
```

Before ending this iteration and continuing to the next iteration, I check whether the target point can be reached with the current RRTtree. Reaching the target point is defined as distance between the target point and its nearest node in RRTtree is less than 20 units and there is no obstacle between them. If the target point can be reached, it means that we have found a path through which we can go from the starting point to the target point and there is no further need to continue another iteration.

```python
# check whether a path to goal is found
# select the node in the RRT tree that is closest to the random node
target_node = np.array([target_y, target_x])
nearest_node, nearest_id, nearest_dis = find_nearest_node(target_node)
if(nearest_dis < 20):

    if(check_obstacle_free(nearest_node, target_node)):
        print('find path!')
        path_found = True
        RRTtree = np.append(RRTtree, np.array([[target_node[0], target_node[1], nearest_id]]), axis=0)
        cv2.line(map_show, (target_node[1], target_node[0]), (nearest_node[1], nearest_node[0]), (0, 0, 0), 1)
        # cv2.circle(map_show, (target_node[1], target_node[0]), 2, (150,0,255), 1)
        cv2.imshow('map', map_show)

        break
```

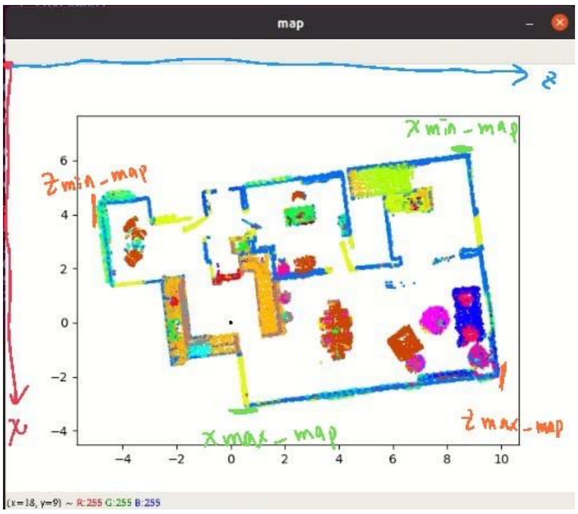
If a path is found, I will extract all the node in the path from the target point to the starting node by using the stored parent id information.

```python
if(path_found):
    node_id = RRTtree.shape[0] - 1      # target id
    cv2.circle(map_show, (target_node[1], target_node[0]), 3, (0,0,255), -1)
    path.append([target_node[0], target_node[1]])

    while True:
        parent_id = RRTtree[node_id][2]

        if(parent_id == -1):
            # start is found
            break

        cv2.line(map_show, (RRTtree[node_id][1], RRTtree[node_id][0]), (RRTtree[parent_id][1], RRTtree[parent_id][0]), (0, 0, 255), 1)
        cv2.circle(map_show, (RRTtree[parent_id][1], RRTtree[parent_id][0]), 2, (0,0,255), -1)
        cv2.imshow('map', map_show)
        path.append([RRTtree[parent_id][0], RRTtree[parent_id][1]])

        node_id = parent_id
```

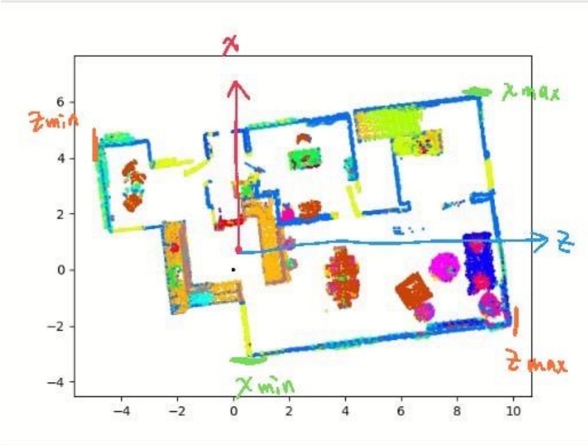
2. Converting route to discrete actions

   The following two pictures show the x axis, z axis and the origin of image coordinate system and point-cloud coordinate system.

   Image coordinate system:



Point-cloud coordinate system:



In order to change the node position from image coordinate system back to point-cloud coordinate system (because we need to navigate the agent using position represented in point-cloud coordinate system), first, I calculate the minimum and the maximum of x and z value in both of the points in the point cloud and the top-view image.

Calculate the minimum and maximum of x and z value for point cloud:

```python
def remove_ceiling_and_floor(pcd):

    global x_max, z_max, x_min, z_min
    pcd = pcd.select_by_index(np.where(np.asarray(pcd.points)[:, 1] < 0.13)[0])  # remove ceiling
    pcd = pcd.select_by_index(np.where(np.asarray(pcd.points)[:, 1] > -1.15)[0])   # remove floor

    x_max = np.max(np.asarray(pcd.points)[:,0])
    z_max = np.max(np.asarray(pcd.points)[:,2])
    x_min = np.min(np.asarray(pcd.points)[:,0])
    z_min = np.min(np.asarray(pcd.points)[:,2])
```

Calculate the minimum and maximum of x and z value for top-view image:

```python
def find_pixel_voxel_scale(map, color_set):
    global x_max_map, z_max_map, x_min_map, z_min_map, x_scale, z_scale

    x_max_map = -np.inf
    z_max_map = -np.inf
    x_min_map = np.inf
    z_min_map = np.inf

    for i in range(map.shape[0]):

        if(i<80):
            continue
        if(i>410):
            continue
        for j in range(map.shape[1]):
            if(j<90):
                continue
            if(j>5600):
                continue

            if(tuple(map[i, j]) in color_set):
                if(i > x_max_map):
                    x_max_map = i
                if(j > z_max_map):
                    z_max_map = j
                if(i < x_min_map):
                    x_min_map = i
                if(j < z_min_map):
                    z_min_map = j
```

By dividing the difference between the minimum and maximum of x (or z) value of the two coordinate system, we can know how much the coordinate should be scaled when the node is changed from the image to the point-cloud coordinate system.

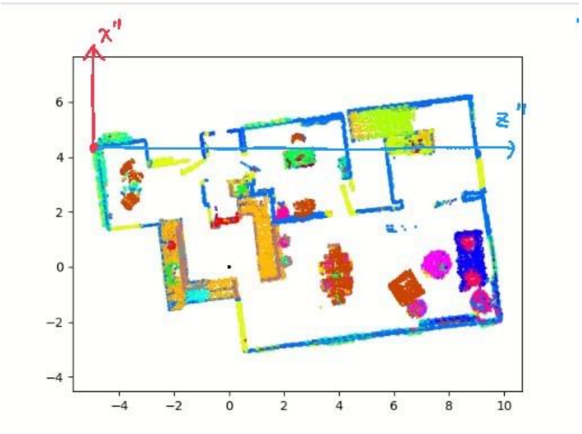
```
    z_scale = (z_max-z_min)/(z_max_map-z_min_map)
    x_scale = (x_max-x_min)/(x_max_map-x_min_map)
```

By using the following formula, we can change the coordinate back to the point-cloud coordinate system.

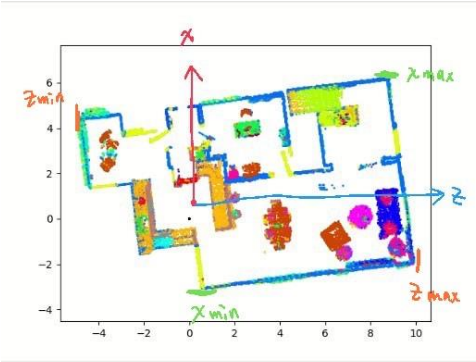
```
def pixel_to_voxel(pixel):
    voxel = np.array([0, 1.5, 0])
    voxel_x = (pixel[0]-x_min_map)*(-x_scale) + x_max
    voxel_z = (pixel[1]-z_min_map)*z_scale + z_min

    return voxel_x, voxel_z
```

After "pixel value – x (or) z min map" and multiply "x (or z) scale", x axis and z axis become x' axis and z' axis.

Because the positive x axis of the point-cloud coordinate system points to the bottom part of the image, we need to multiply "-1" when multiplying the x scale (that is, to multiply "-x scale").

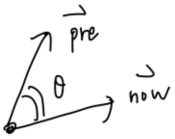


By adding x (or z) min value, we can change x'' axis and z'' axis into our target axis (axis of point-cloud coordinate system).



Having the point coordinate in point-cloud coordinate system, we can calculate how many degrees the agent should turn left or turn right, and how long the agent should move forward when it goes from the starting point to the target point.

By using "dot product" and "arccos", we can easily calculate how many degrees the agent should turn. By using "cross product", we can know whether the agent should turn left or turn right. As for how long the agent should move forward, I simply use the L2 distance between two adjacent nodes to calculate.



$$\vec{pre} \cdot \vec{now} = |\vec{pre}||\vec{now}| \cos\theta$$

$$\theta = a\cos\left(\frac{\vec{pro} \cdot \vec{now}}{|\vec{pre}||\vec{now}|}\right)$$

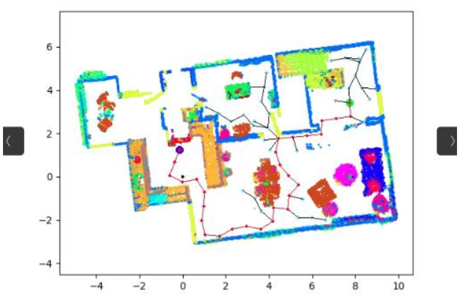
```python
            # dot -> to know how many degree to turn
            print(pre_forward,forward)
            theta = np.rad2deg(np.arccos(np.dot(pre_forward, forward)/length/pre_length))

            # cross -> to know turn left or turn right
            if(np.cross(pre_forward, forward) > 0):
                action = "turn_right"
            else:
                action = "turn_left"
```

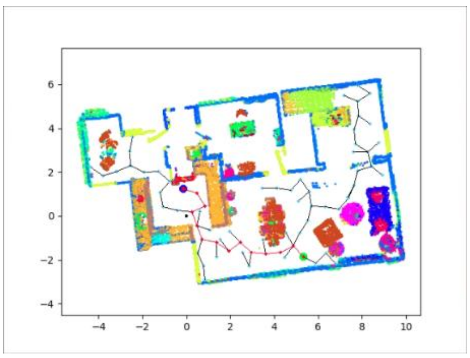
```python
        forward = path_voxel[i, :] - path_voxel[i+1, :]
        length = np.linalg.norm(forward)
```

ii. Result and Discussion

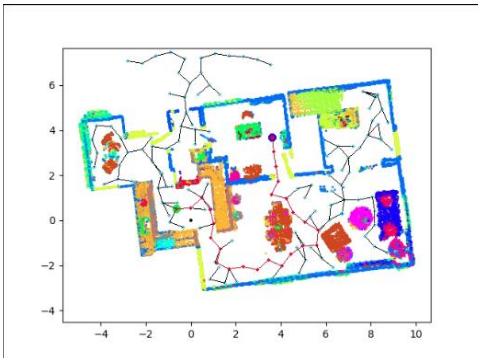
Starting point: green; Target point: blue

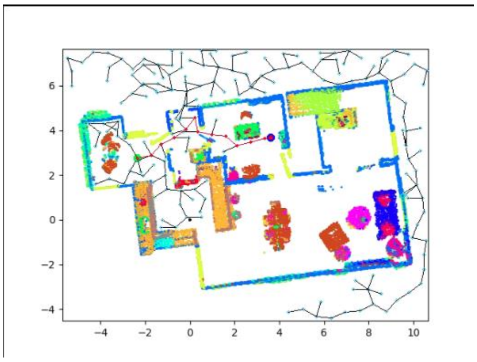
Refrigerator with different starting point:



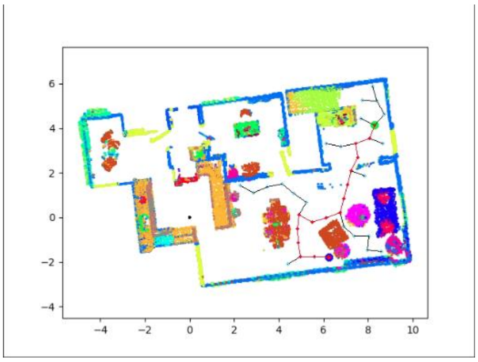


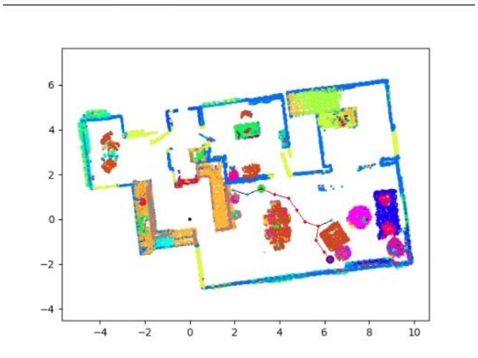
Rack with different starting point:



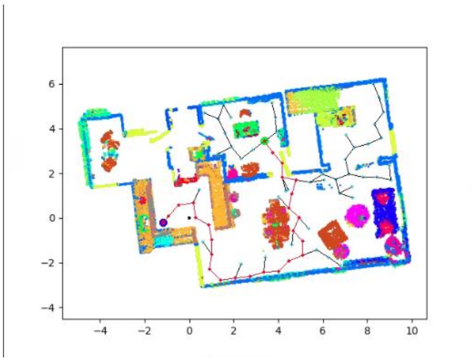


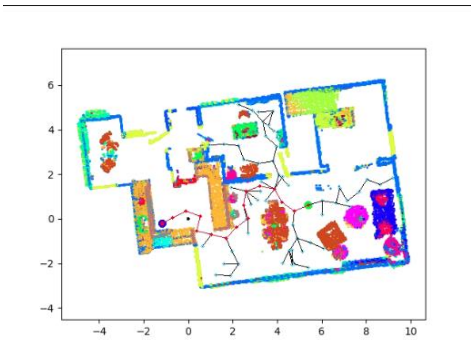
Lamp with different starting point:



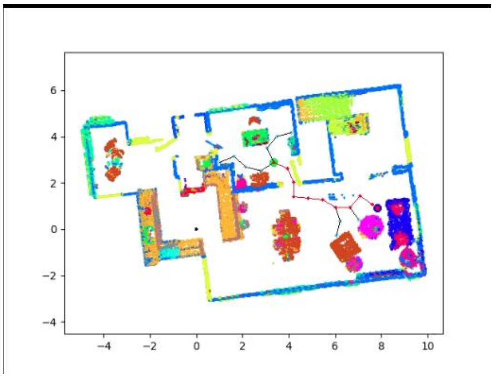


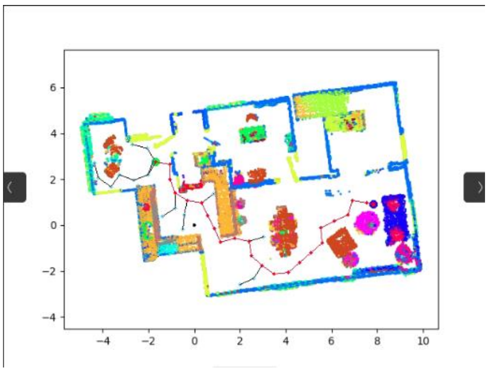
Cooktop with different starting point:



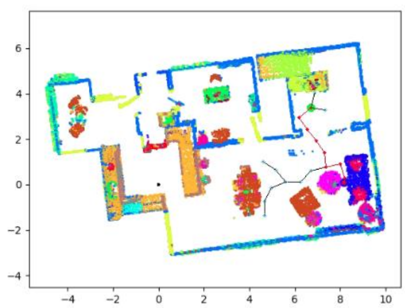


Cushion with different starting point:

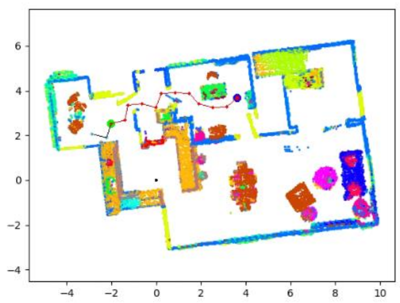


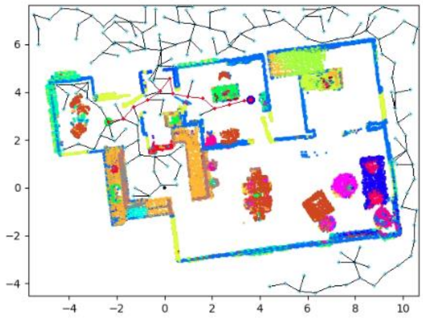


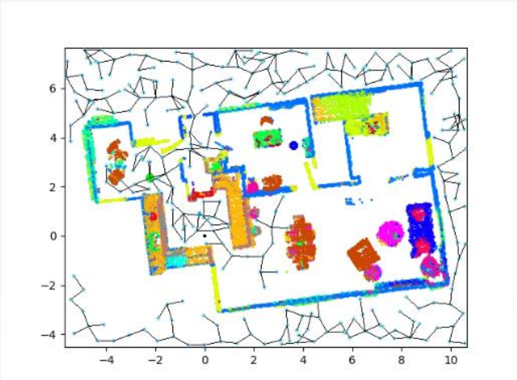
Actually, I have tried different target point for cusion. However, I found it was super hard to find a path successfully for some target point position. At the beginning, I chose the target point for cusion as the following picture shows. I tried to find a path with different starting point for this target point a lot of times, however, below was the only successful result. Although the target point is indeed at a “free” space where the agent can arrive, its position makes it hard to find a path by using RRT algorithm: this target point is just between the table and the sofa. Before RRT adds a point into RRT tree, it checks whether there is no obstacle between two points. When RRT trys to add a point, from which the agent can arrive the target point, into RRT tree, the point cannot be added into the tree as long as it deviates a little bit because the table and the sofa had made large part of grids the obstacles.



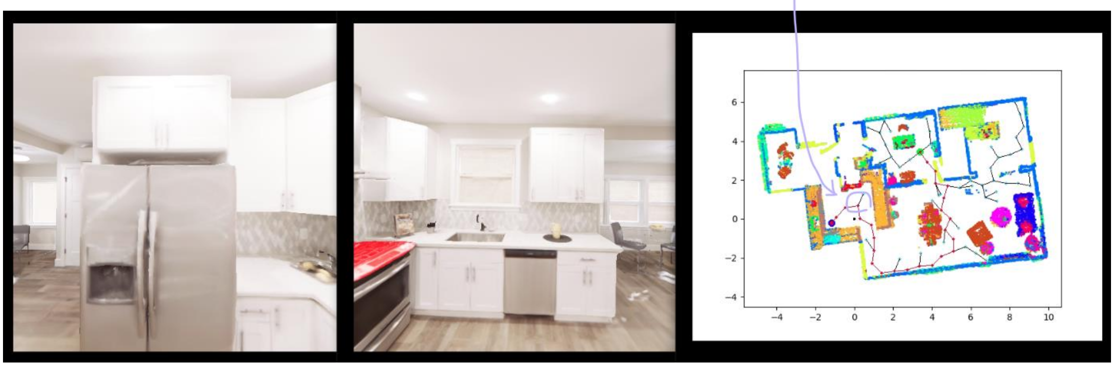
Because RRT algorithm bascally randomly choose a node in each iteration, sometimes it happens that I fixed the target point (e.g. rack) and chose a similar starting point, but the paths RRT found differed a lot. As the following three pictures, all of them have the same target point (rack) and a similar starting point. However, the path could be found with few iteration of search in the first picture, while the path was found with a lot of iteration in the second picture. Even, the path cannot be found in the third picture. If we wants to apply RRT method to some scenarios where a path found in each time should be stable, I think it would be the room that a standard RRT algorithm can be improved.







When I watched the video which recorded the robot navigation process, I found that when the orientation of the robot differed a lot between to actions, the video looked a little discontinuous, which is because I directly turn the robot to the orientation it would be at. Maybe we can limit the maximum degrees which the robot can turn at each action, so the robot will take more actions to turn to the target orientation when the difference between two orientations exceed the limit. I think this would make the result look more contiuous. Also, maybe a modified RRT algorithm can improve the quality of the found path (i.e. make it more smooth).
The following pictures show the consecutive two frames in which the robot trun about 90 degrees. This makes the result look dicontinuous.



Also, I found that sometimes when the robot arrives at the target point, actually we cannot see the object from the result image due to the orientation with which the robot arrives at the target point or the height of the camera.