HW3 Report

311554021 張祐閤

i. Code

- RRT

  The RRT algorithm can be broken down into 4 steps: random sample, connect new node to graph, check collision, and generate path.

  1. Random Sample

```python
def sample(self):
    if random.randint(0, 100) > self.goal_sample_rate:
        rnd = [random.uniform(self.x_min_rand, self.x_max_rand),
               random.uniform(self.y_min_rand, self.y_max_rand)]
    else:
        rnd = [self.goal.x, self.goal.y]

    return rnd
```

  I randomly sample a new point from the map, and with a probability of *goal_sample_rate*, the newly sampled point would be the goal. This gives a clue to the agent where might be a better direction for it to take a step forward.

  2. Connect New Node to Graph

```python
@staticmethod
def get_nearest_node_index(node_list, rnd):
    dist = [(node.x-rnd[0])**2 + (node.y - rnd[1])**2 for node in node_list]
    min_index = dist.index(min(dist))
    return min_index
```

```python
def get_new_node(self, theta, nearest_index, nearest_node):
    new_node = copy.deepcopy(nearest_node)
    new_node.x += self.expand_dis * math.cos(theta)
    new_node.y += self.expand_dis * math.sin(theta)

    new_node.cost += self.expand_dis
    new_node.parent = nearest_index

    return new_node
```

  After sampling a new point, I find the nearest node in the current graph to the new point, then compute the orientation from the node to the sampled point, and along this direction, a new node is generated with a distance of *expand_dis* from the nearest node.

  3. Check collision

  I then check if the segment that connect the new node and its nearest node is collide with an obstacle with a radius of 0.1m

```python
def check_collision(self, x1, y1, x2, y2):
    for (oy, _, ox) in self.obstacle_list:
        dd = self.distance_square_point_to_segment(
            np.array([x1, y1]),
            np.array([x2, y2]),
            np.array([ox, oy])
        )

        if dd <= self.radius ** 2:
            return True

    return False
```

4. Generate Path

```
#  check if new node is close to goal
if self.is_near_goal(new_node):
    if not self.check_collision(new_node.x , new_node.y, self.goal.x, self.goal.y):
        last_index = len(self.node_list) - 1
        path = self.get_final_course(last_index)
```

For every new node, I record its nearest node as its parent when it was added into the graph. Thus, when a new node is close to the goal and there is no collision between obstacles and the new segment, the path is then generated by simply tracing back the predecessors from the latest node

After integrating the above 4 steps, the whole RRT process is shown as below:

```
def rrt_planning(self, start, goal, animation=True):
    self.start = Node(start[0], start[1])
    self.goal = Node(goal[0], goal[1])
    self.node_list = [self.start]
    path = None

    for i in range(self.max_iter):
        # print(f"==================\nstep: {i + 1}\nNodes in RRT Tree: {len(self.node_list)}\n==================\n")
        # random sample
        rnd = self.sample()

        # find the nearest node to the sample point in node tree
        nearest_index = self.get_nearest_node_index(self.node_list, rnd)
        nearest_node = self.node_list[nearest_index]

        # get next node
        theta = math.atan2(rnd[1] - nearest_node.y, rnd[0] - nearest_node.x)
        new_node = self.get_new_node(theta, nearest_index, nearest_node)

        # check collision
        collision = self.check_collision(new_node.x, new_node.y, nearest_node.x, nearest_node.y)
        if not collision:
            self.node_list.append(new_node)

            if animation:
                time.sleep(0.1)
                self.draw_graph(new_node, path)

            #  check if new node is close to goal
            if self.is_near_goal(new_node):
                if not self.check_collision(new_node.x , new_node.y, self.goal.x, self.goal.y):
                    last_index = len(self.node_list) - 1
                    path = self.get_final_course(last_index)


                    if animation:
                        self.draw_graph(new_node, path)
                    return path
```

- Generate actions

```
def onclick(event):
    global coords
    print(event.xdata, event.ydata)
    coords.append((event.xdata, event.ydata))
    fig.canvas.mpl_disconnect(cid)

cid = fig.canvas.mpl_connect('button_press_event', onclick)
plt.show()
```

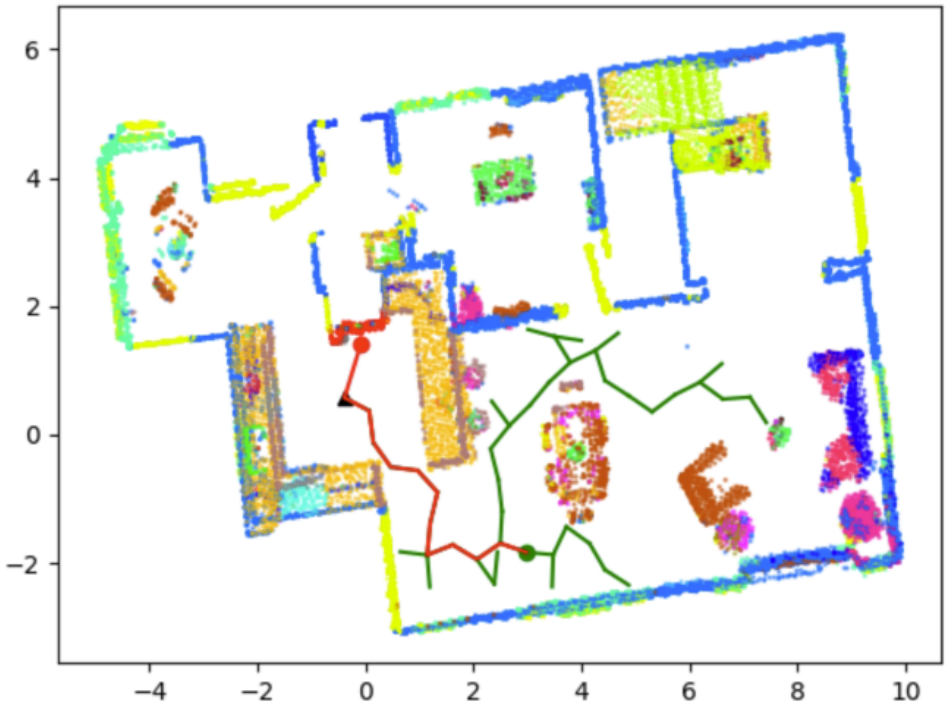
Since the starting point selection is done with matplotlib interactive plot, the coordinate system I use in RRT planning is the same in habitat, so there is no need to calculate pixel-to-point transformation when generating actions from the path.

There are two steps for a habitat agent to move from a starting point to a destination, which are rotating itself then moving forward. From this point of view, I compute the distance to step forward by calculating the length of a path vector, and the orientation is obtained by subtracting the direction of a path vector to the current orientation of the habitat agent, since I know that the initial orientation of the agent is 180°. The whole action list is then defined during the configuration of the habitat agent.

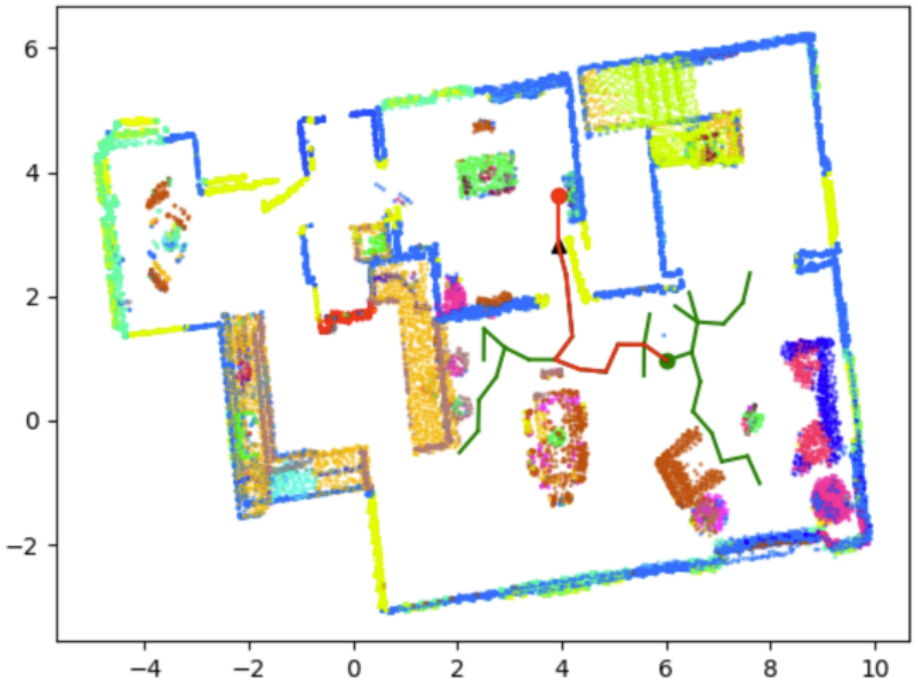
```
action_list = []
for i in range(1, len(navigate_path)):
    theta = math.atan2(navigate_path[i][1] - navigate_path[i - 1][1], navigate_path[i][0] - navigate_path[i - 1][0])
    if theta < 0:
        theta += 2*math.pi
    print(180*theta/math.pi)
    temp = theta
    theta -= cam_ori
    cam_ori = temp
    if theta != 0:
        action_list.append(habitat_sim.agent.ActionSpec(
                "turn_left", habitat_sim.agent.ActuationSpec(amount=180 * theta/math.pi) # 1.0 means 1 degree
            ))
    action_list.append(habitat_sim.agent.ActionSpec(
            "move_forward", habitat_sim.agent.ActuationSpec(amount=0.5) # 0.01 means 0.01 m
        ))
```

ii. Results & Discussion

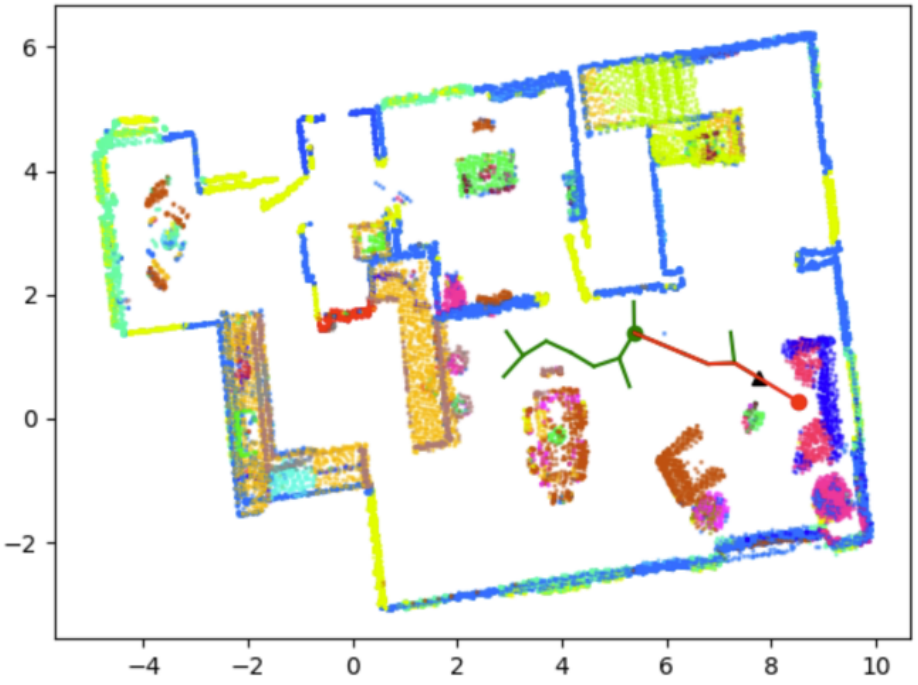
- RRT Planning Results
  - Refrigerator



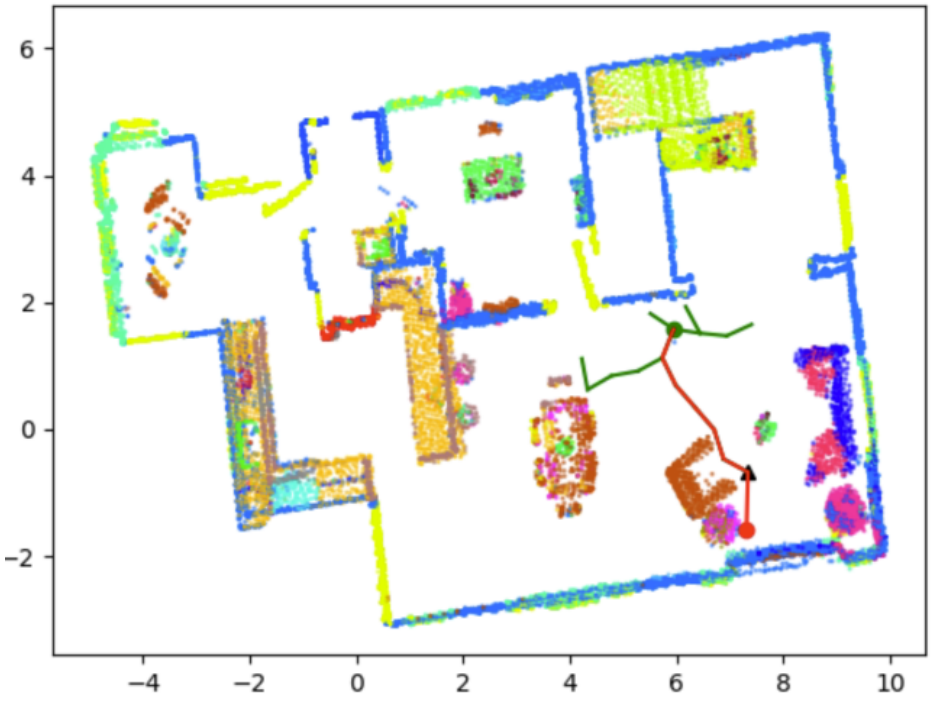
- Rack



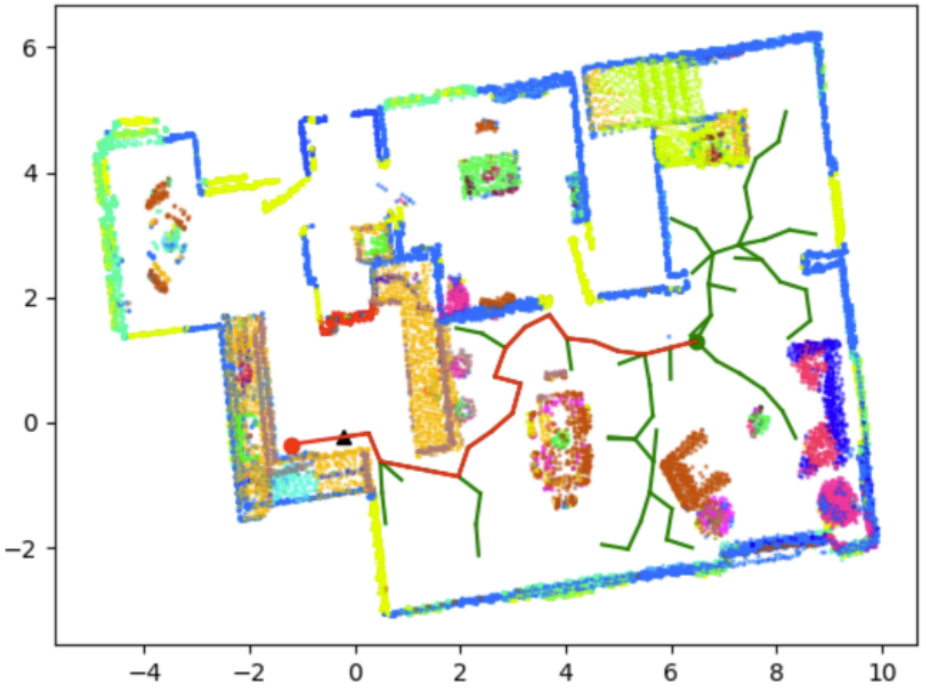
- Cushion



- Lamp



- Cooktop



- Discussion

  I found that *goal_sample_rate* is an important parameter when the path to the target will bypass a narrow tunnel, like rack, refrigerator or cooktop. The reason is that randomly sampling new points can give the agent better clues to dodge an obstacle but lacking the prior of where the goal is will make the agent have difficulty getting out of one room since the doorway is always surrounded by obstacles.