Perception and Decision Making in Intelligent Systems

Homework 2: A Robot Navigation Framework

109350008 張詠哲

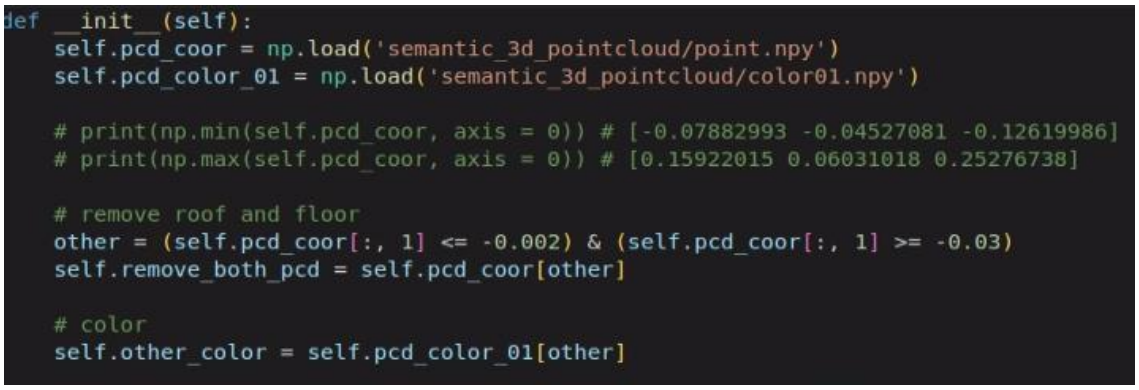
1. Implementation

a. code

Part1.

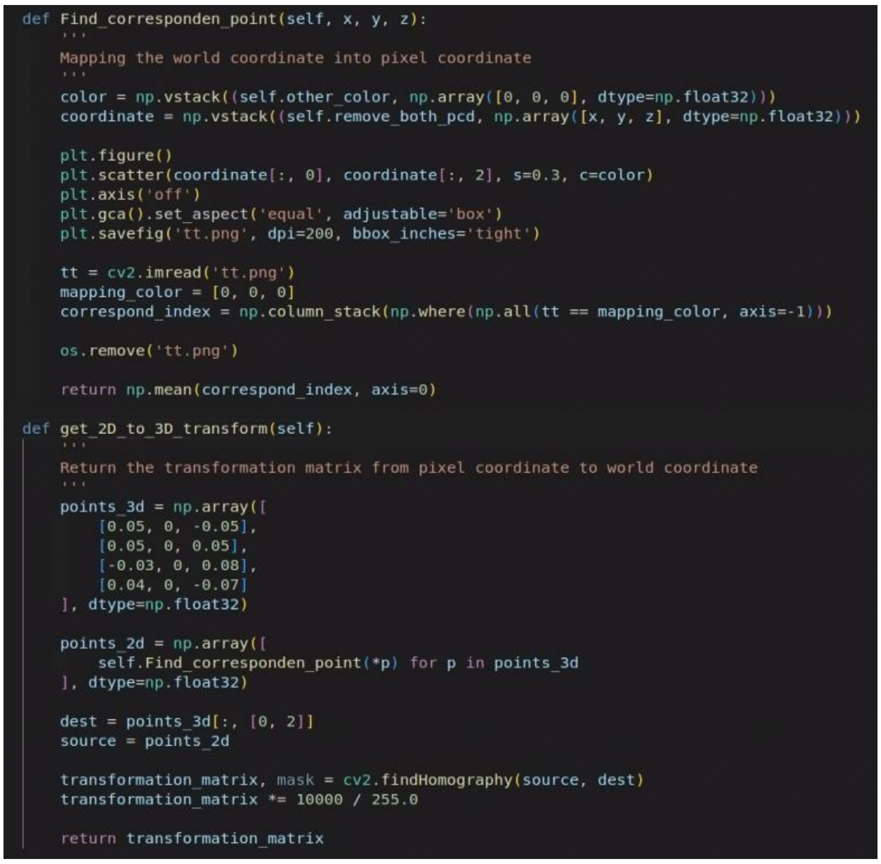
In this part we need to find the transformation matrix for pixel to Habitat coordinate and construct the semantic segmentation 2D map that remove roof and floor.

First, I filter the points that are locate between the -0.002 and -0.03 to remove the roof and floor. The reason of choosing this region is I found this region can remove the mose of noisy point also reserve the most of the indoor objects.



```
def __init__(self):
    self.pcd_coor = np.load('semantic_3d_pointcloud/point.npy')
    self.pcd_color_01 = np.load('semantic_3d_pointcloud/color01.npy')

    # print(np.min(self.pcd_coor, axis = 0)) # [-0.07882993 -0.04527081 -0.12619986]
    # print(np.max(self.pcd_coor, axis = 0)) # [0.15922015 0.06031018 0.25276738]

    # remove roof and floor
    other = (self.pcd_coor[:, 1] <= -0.002) & (self.pcd_coor[:, 1] >= -0.03)
    self.remove_both_pcd = self.pcd_coor[other]

    # color
    self.other_color = self.pcd_color_01[other]
```

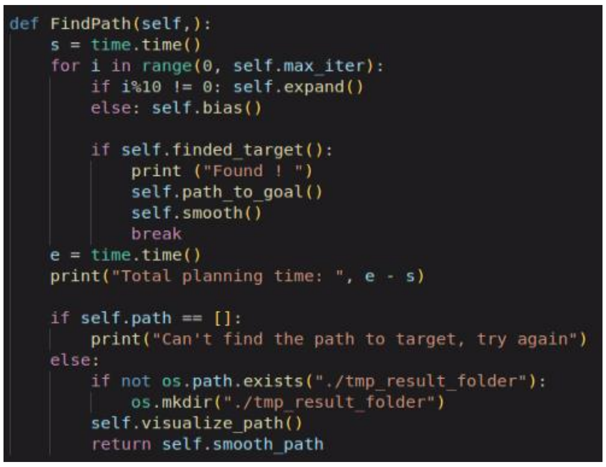
Second, we need to find the transformation matrix. I achieve this by mapping several points into 2D map and given them a color that doesn't show in color_coding_semantic_segmentation_classes. Then, we will have several 2D points that can be mapping to the Habitat coordinate, and we can use findHomography to find the transformation matrix. Here I choose black([0, 0, 0]) as my mapping color.



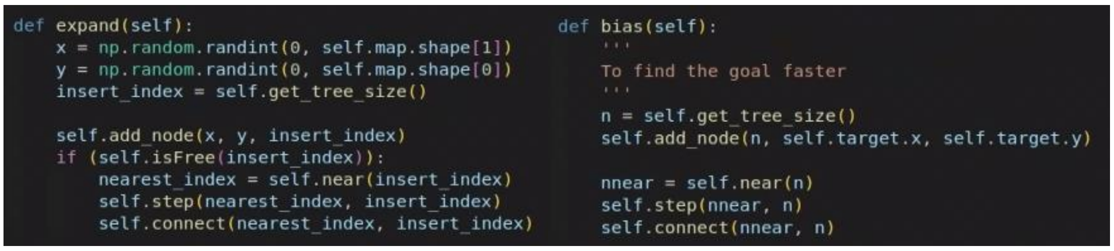
```python
def Find_corresponden_point(self, x, y, z):
    '''
    Mapping the world coordinate into pixel coordinate
    '''
    color = np.vstack((self.other_color, np.array([0, 0, 0], dtype=np.float32)))
    coordinate = np.vstack((self.remove_both_pcd, np.array([x, y, z], dtype=np.float32)))

    plt.figure()
    plt.scatter(coordinate[:, 0], coordinate[:, 2], s=0.3, c=color)
    plt.axis('off')
    plt.gca().set_aspect('equal', adjustable='box')
    plt.savefig('tt.png', dpi=200, bbox_inches='tight')

    tt = cv2.imread('tt.png')
    mapping_color = [0, 0, 0]
    correspond_index = np.column_stack(np.where(np.all(tt == mapping_color, axis=-1)))

    os.remove('tt.png')

    return np.mean(correspond_index, axis=0)

def get_2D_to_3D_transform(self):
    '''
    Return the transformation matrix from pixel coordinate to world coordinate
    '''
    points_3d = np.array([
        [0.05, 0, -0.05],
        [0.05, 0, 0.05],
        [-0.03, 0, 0.08],
        [0.04, 0, -0.07]
    ], dtype=np.float32)

    points_2d = np.array([
        self.Find_corresponden_point(*p) for p in points_3d
    ], dtype=np.float32)

    dest = points_3d[:, [0, 2]]
    source = points_2d

    transformation_matrix, mask = cv2.findHomography(source, dest)
    transformation_matrix *= 10000 / 255.0

    return transformation_matrix
```

Part2.

In part 2, we need to implement the RRT algorithm. The algorithm iteratively expands a tree by randomly sampling points in the map, then attempts to connect each new point to the nearest existing node in the tree. At each step, the algorithm checks for collisions with obstacles, for here I check if the selected random points in specific region are all white (it means it's a valid path for robot), and if a valid path is found, it adds the new node to the tree. The goal is to reach the target point identified by the color of target object. I've added a function "bias", it sometimes selects the target point as the new point to speed up the finding time. Once the target is found, the path to the goal is backtracked from the tree.

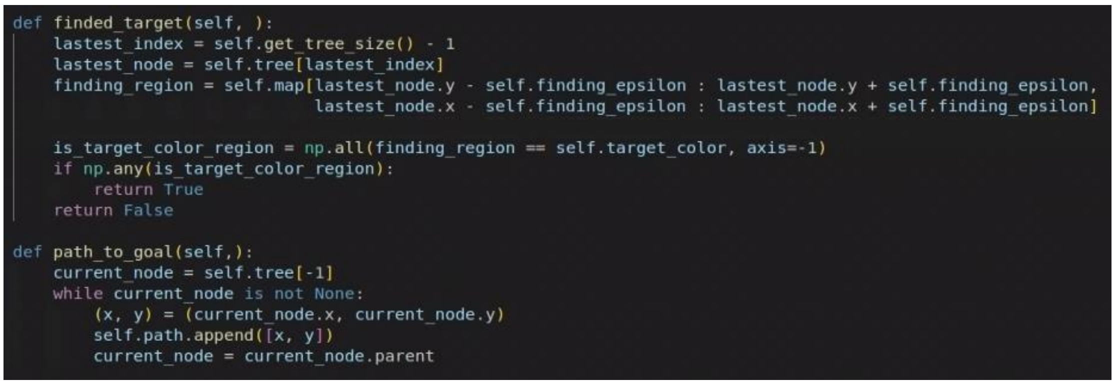


```python
def FindPath(self,):
    s = time.time()
    for i in range(0, self.max_iter):
        if i%10 != 0: self.expand()
        else: self.bias()

        if self.finded_target():
            print ("Found ! ")
            self.path_to_goal()
            self.smooth()
            break
    e = time.time()
    print("Total planning time: ", e - s)

    if self.path == []:
        print("Can't find the path to target, try again")
    else:
        if not os.path.exists("./tmp_result_folder"):
            os.mkdir("./tmp_result_folder")
        self.visualize_path()
        return self.smooth_path
```

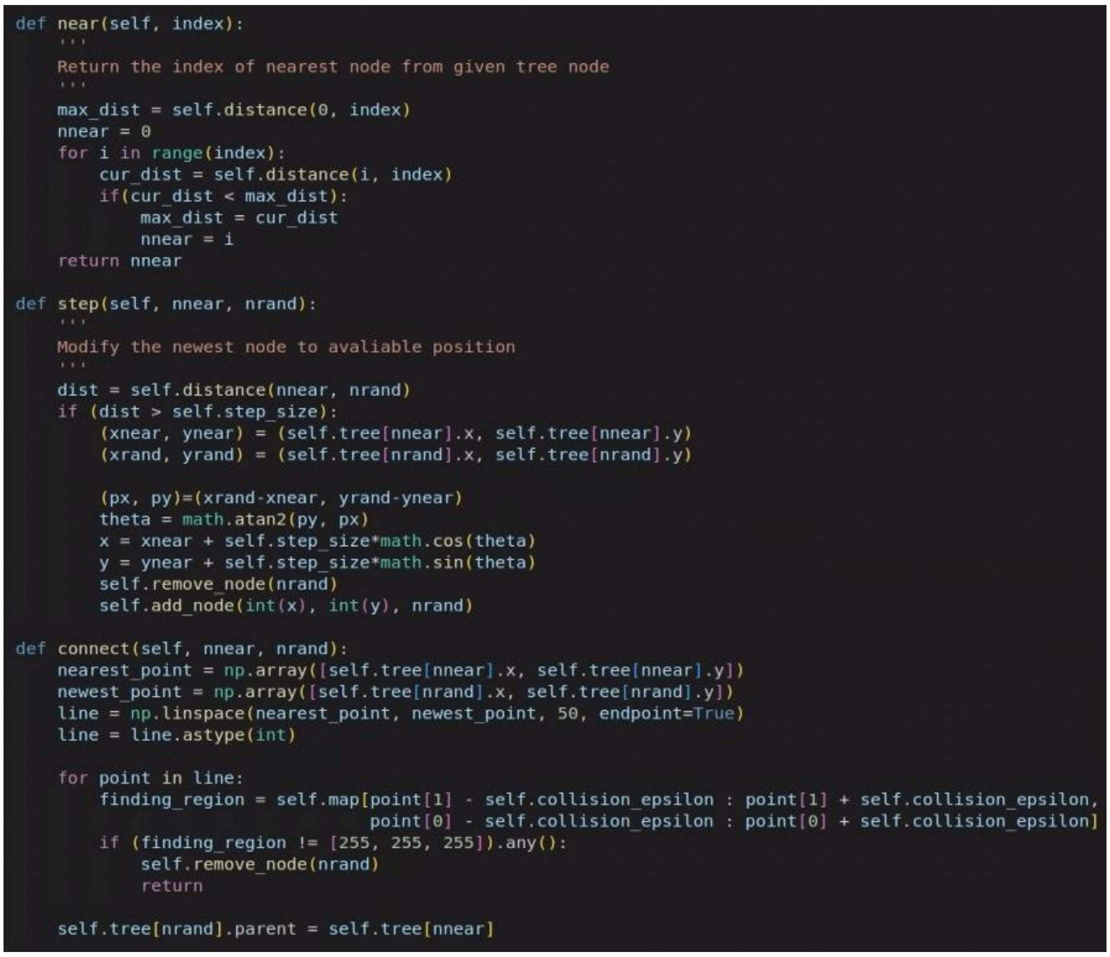


```python
def expand(self):
    x = np.random.randint(0, self.map.shape[1])
    y = np.random.randint(0, self.map.shape[0])
    insert_index = self.get_tree_size()

    self.add_node(x, y, insert_index)
    if (self.isFree(insert_index)):
        nearest_index = self.near(insert_index)
        self.step(nearest_index, insert_index)
        self.connect(nearest_index, insert_index)
```

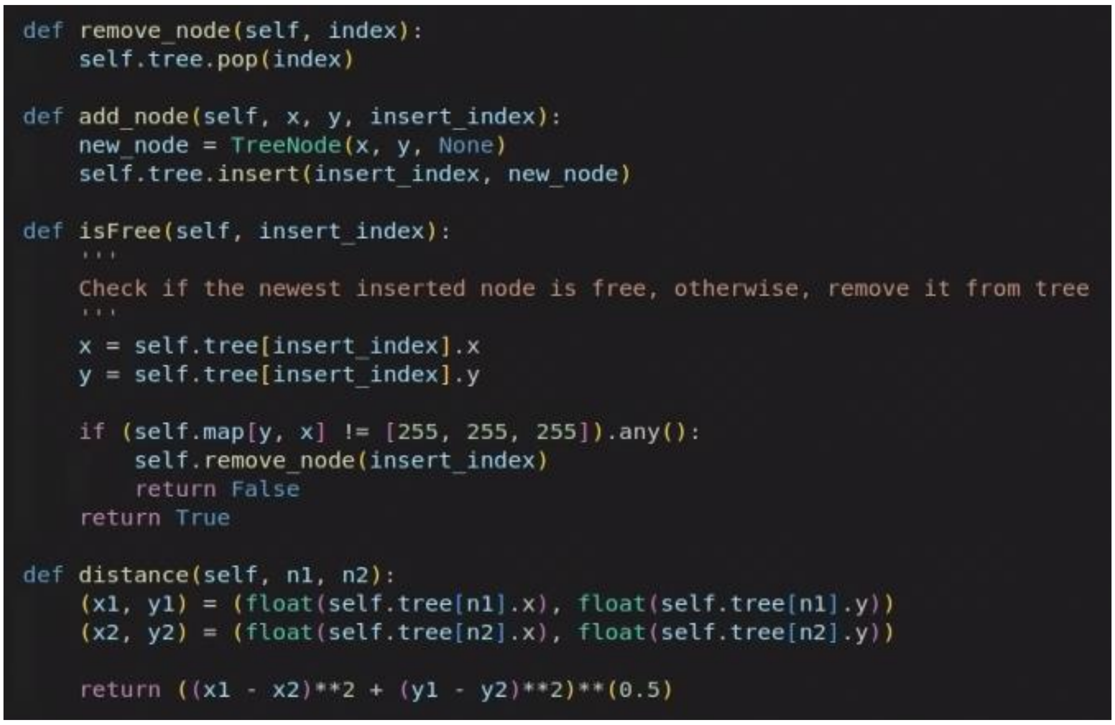
```python
def bias(self):
    '''
    To find the goal faster
    '''
    n = self.get_tree_size()
    self.add_node(n, self.target.x, self.target.y)

    nnear = self.near(n)
    self.step(nnear, n)
    self.connect(nnear, n)
```



```python
def finded_target(self, ):
    lastest_index = self.get_tree_size() - 1
    lastest_node = self.tree[lastest_index]
    finding_region = self.map[lastest_node.y - self.finding_epsilon : lastest_node.y + self.finding_epsilon,
                              lastest_node.x - self.finding_epsilon : lastest_node.x + self.finding_epsilon]

    is_target_color_region = np.all(finding_region == self.target_color, axis=-1)
    if np.any(is_target_color_region):
        return True
    return False

def path_to_goal(self,):
    current_node = self.tree[-1]
    while current_node is not None:
        (x, y) = (current_node.x, current_node.y)
        self.path.append([x, y])
        current_node = current_node.parent
```



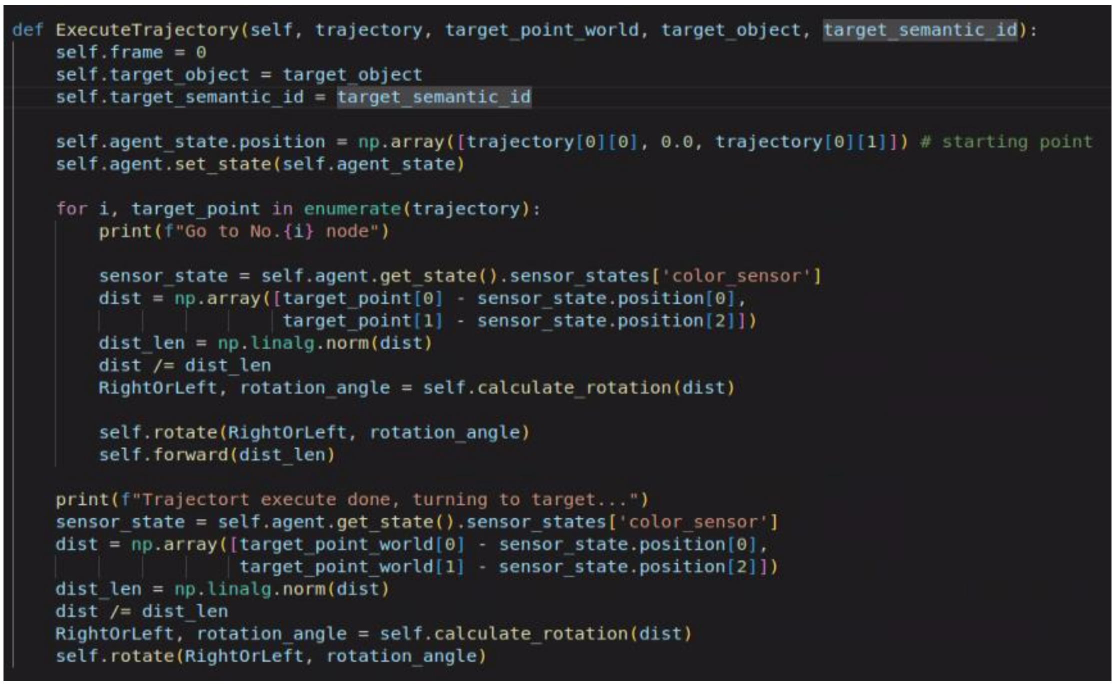
```python
def near(self, index):
    '''
    Return the index of nearest node from given tree node
    '''
    max_dist = self.distance(0, index)
    nnear = 0
    for i in range(index):
        cur_dist = self.distance(i, index)
        if(cur_dist < max_dist):
            max_dist = cur_dist
            nnear = i
    return nnear

def step(self, nnear, nrand):
    '''
    Modify the newest node to avaliable position
    '''
    dist = self.distance(nnear, nrand)
    if (dist > self.step_size):
        (xnear, ynear) = (self.tree[nnear].x, self.tree[nnear].y)
        (xrand, yrand) = (self.tree[nrand].x, self.tree[nrand].y)

        (px, py)=(xrand-xnear, yrand-ynear)
        theta = math.atan2(py, px)
        x = xnear + self.step_size*math.cos(theta)
        y = ynear + self.step_size*math.sin(theta)
        self.remove_node(nrand)
        self.add_node(int(x), int(y), nrand)

def connect(self, nnear, nrand):
    nearest_point = np.array([self.tree[nnear].x, self.tree[nnear].y])
    newest_point = np.array([self.tree[nrand].x, self.tree[nrand].y])
    line = np.linspace(nearest_point, newest_point, 50, endpoint=True)
    line = line.astype(int)

    for point in line:
        finding_region = self.map[point[1] - self.collision_epsilon : point[1] + self.collision_epsilon,
                                  point[0] - self.collision_epsilon : point[0] + self.collision_epsilon]
        if (finding_region != [255, 255, 255]).any():
            self.remove_node(nrand)
            return

    self.tree[nrand].parent = self.tree[nnear]
```



```python
def remove_node(self, index):
    self.tree.pop(index)

def add_node(self, x, y, insert_index):
    new_node = TreeNode(x, y, None)
    self.tree.insert(insert_index, new_node)

def isFree(self, insert_index):
    '''
    Check if the newest inserted node is free, otherwise, remove it from tree
    '''
    x = self.tree[insert_index].x
    y = self.tree[insert_index].y

    if (self.map[y, x] != [255, 255, 255]).any():
        self.remove_node(insert_index)
        return False
    return True

def distance(self, n1, n2):
    (x1, y1) = (float(self.tree[n1].x), float(self.tree[n1].y))
    (x2, y2) = (float(self.tree[n2].x), float(self.tree[n2].y))

    return ((x1 - x2)**2 + (y1 - y2)**2)**(0.5)
```

Part3.

This part I modify the load.py to navigate the robot with the path finding by RRT. After we find the path by RRT in 2D map, we mapping the path (in pixel) to Habitat coordinate by the transformation matrix I discussed in part1. For each path point, we calculate the vector that point to the current robot position and the next path point for further calculation.



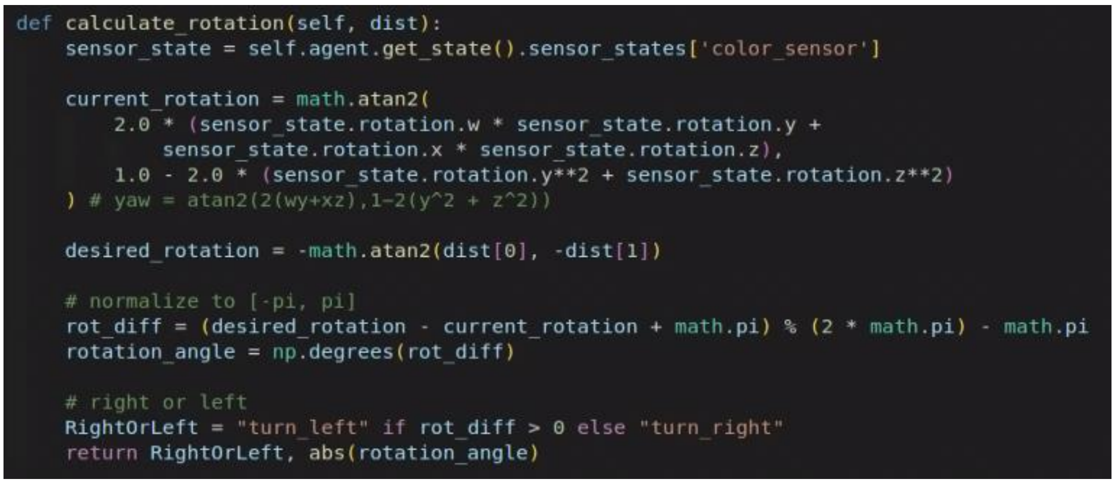
```python
def ExecuteTrajectory(self, trajectory, target_point_world, target_object, target_semantic_id):
    self.frame = 0
    self.target_object = target_object
    self.target_semantic_id = target_semantic_id

    self.agent_state.position = np.array([trajectory[0][0], 0.0, trajectory[0][1]]) # starting point
    self.agent.set_state(self.agent_state)

    for i, target_point in enumerate(trajectory):
        print(f"Go to No.{i} node")

        sensor_state = self.agent.get_state().sensor_states['color_sensor']
        dist = np.array([target_point[0] - sensor_state.position[0],
                         target_point[1] - sensor_state.position[2]])
        dist_len = np.linalg.norm(dist)
        dist /= dist_len
        RightOrLeft, rotation_angle = self.calculate_rotation(dist)

        self.rotate(RightOrLeft, rotation_angle)
        self.forward(dist_len)

    print(f"Trajectort execute done, turning to target...")
    sensor_state = self.agent.get_state().sensor_states['color_sensor']
    dist = np.array([target_point_world[0] - sensor_state.position[0],
                     target_point_world[1] - sensor_state.position[2]])
    dist_len = np.linalg.norm(dist)
    dist /= dist_len
    RightOrLeft, rotation_angle = self.calculate_rotation(dist)
    self.rotate(RightOrLeft, rotation_angle)
```

To calculate the desired rotation. I normalize the direction vector, then from the following equation, we can know the yaw rotation can be calculate by $\theta_{\mathrm{yaw}} = \arctan\left(2(\mathrm{wy}+\mathrm{xz}), 1-2(\mathrm{y}^2+\mathrm{z}^2)\right)$

$$\begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} \arctan \frac{2(q_0q_1+q_2q_3)}{1-2(q_1^2+q_2^2)} \\ \arcsin(2(q_0q_2 - q_1q_3)) \\ \arctan \frac{2(q_0q_3+q_1q_2)}{1-2(q_2^2+q_3^2)} \end{bmatrix}$$

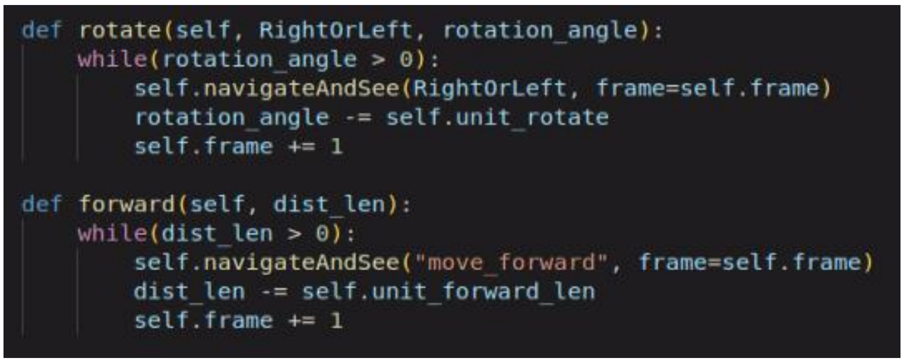
For $\mathrm{q} = (\mathrm{q}_0, \mathrm{q}_1, \mathrm{q}_2, \mathrm{q}_3)$, γ is yaw rotation

Then, we can know the rotation difference, and normalize it into $[-\pi, \pi]$



```python
def calculate_rotation(self, dist):
    sensor_state = self.agent.get_state().sensor_states['color_sensor']

    current_rotation = math.atan2(
        2.0 * (sensor_state.rotation.w * sensor_state.rotation.y +
            sensor_state.rotation.x * sensor_state.rotation.z),
        1.0 - 2.0 * (sensor_state.rotation.y**2 + sensor_state.rotation.z**2)
    ) # yaw = atan2(2(wy+xz),1-2(y^2 + z^2))

    desired_rotation = -math.atan2(dist[0], -dist[1])

    # normalize to [-pi, pi]
    rot_diff = (desired_rotation - current_rotation + math.pi) % (2 * math.pi) - math.pi
    rotation_angle = np.degrees(rot_diff)

    # right or left
    RightOrLeft = "turn_left" if rot_diff > 0 else "turn_right"
    return RightOrLeft, abs(rotation_angle)
```

Then, move the robot step by step by the step size and rotation angle we set.

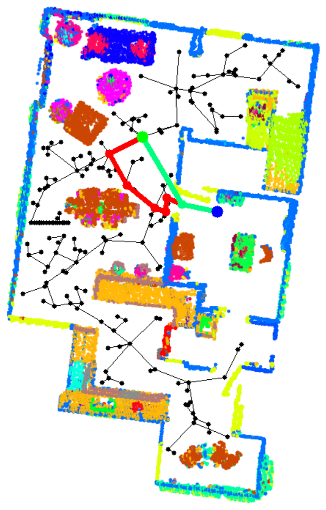


```python
def rotate(self, RightOrLeft, rotation_angle):
    while(rotation_angle > 0):
        self.navigateAndSee(RightOrLeft, frame=self.frame)
        rotation_angle -= self.unit_rotate
        self.frame += 1

def forward(self, dist_len):
    while(dist_len > 0):
        self.navigateAndSee("move_forward", frame=self.frame)
        dist_len -= self.unit_forward_len
        self.frame += 1
```

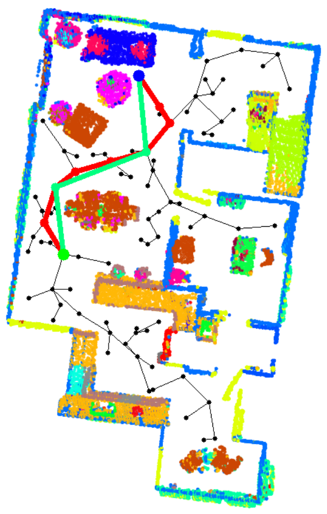
b. Result and Discussion

- **Show and discuss the results from the RRT algorithm with different start points and targets**

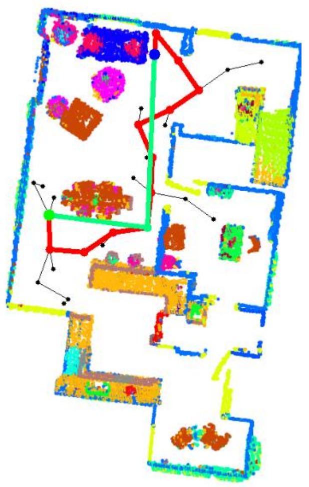
Rack



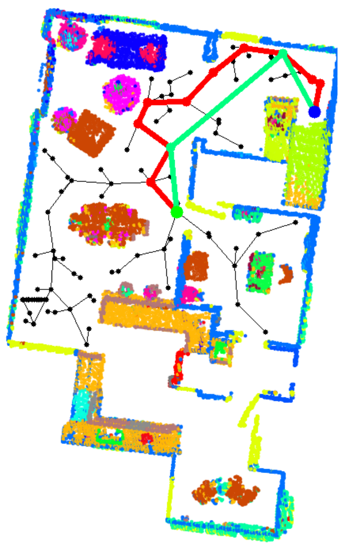
Cushion



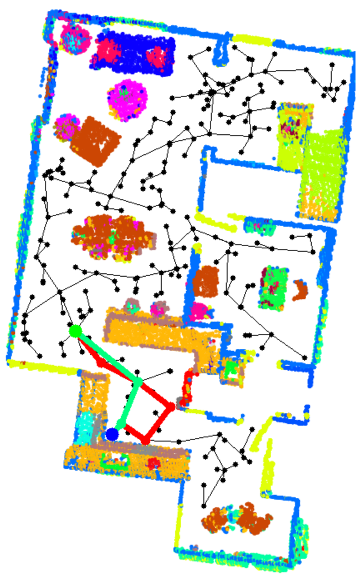
Sofa



Stair



Cooktop



The above table shows the different start point and target object. We can see that the larger object or the more open object, the fewer explored point (black circle), it also contributes to the fewer planning time. The small object or the more occupy object like cooktop and the rack need more explored point to reach the target. Need to mention that the cooktop is the most difficult planning object, cause the path is too narrow so that the random point is hard to find a collision-free trajectory to reach it.

- **Discuss about the robot navigation results**

Most of the path can do the collision-free navigation. But I found that even I've reserved some collision-free area in RRT implementation, some of cases still collide with the small object, such as chair or the corner of objects. Thought it can still reach the target even collision occurred, we can't say it is complete collision-free trajectory. Maybe increase the collision-free area is a solution, but the difficulty of planning will also increase.

c. Reference

https://github.com/mohamedbanhawi/RRT/blob/master/RRT-3D-class.py
https://www.cnblogs.com/21207-iHome/p/6894128.html
https://theclassytim.medium.com/robotic-path-planning-rrt-and-rrt-212319121378
https://vslam.net/2021/04/10/route_planning/%E8%B7%AF%E5%BE%84%E8%A7%84%E5%88%92%EF%BC%88%E4%B9%9D%EF%BC%89-RRT-Star%E7%AE%97%E6%B3%95/

2. Question

a. In the RRT algorithm, you can adjust the step size and bias (the number balance between exploration and exploitation). Please explain how the two numbers affect the RRT sampling result. (15%)

- **Step size**
  - A larger step size allows the tree to explore the environment more quickly by covering greater distances in fewer steps. But this easily leading to suboptimal paths or collisions.
  - A smaller step size provides more finer control over the tree. But it comes at the cost of increased computational time, as the tree must take more steps to explore the same area.

- **bias**
  - Increasing bias (exploitation more) makes RRT more likely to expand towards the goal, reducing the time to find a solution when the target is easily reachable. But this may lead to premature convergence and prevent the tree from fully exploring the environment, also easy occur collision cause if the obstacles are large in the scenario.
  - Increasing exploration increasing the chance of finding optimal paths in complex environments. But it also may result in longer search times, as the tree may not focus on moving towards the target as directly.

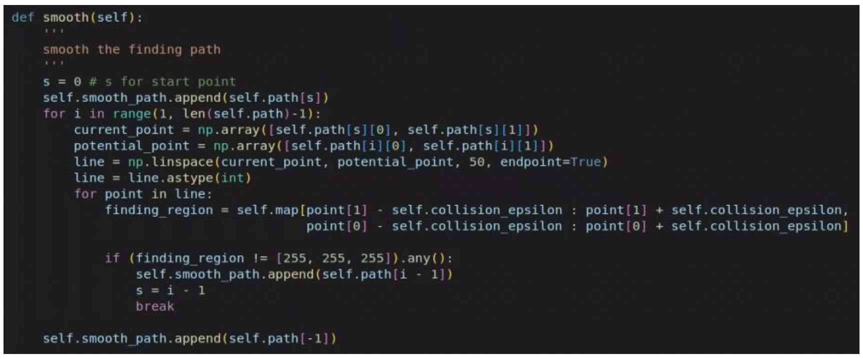
b. If you want to apply the indoor navigation pipeline in the real world, what

problems may you encounter? (5%)

I think the potential problems may be the dynamic obstacles and sensor noise. Cause indoor environments often contain moving objects (ex. people, furniture), which are not accounted for in a static map. The robot must be able to adapt to such changes in real-time, which is challenging with the static nature of the basic RRT algorithm.
Also, in real-world environments, sensors (such as camera) are prone to noise and errors. This uncertainty can affect the accuracy of obstacle detection and the robot's position, leading to incorrect path planning.

Bonus

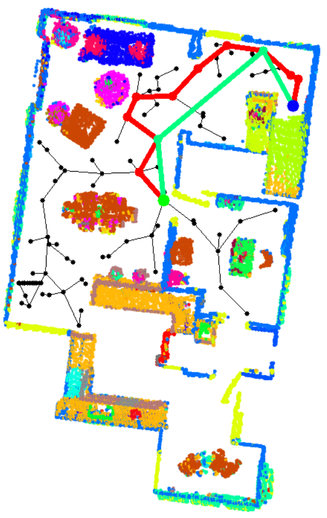
I've also smoothed the path that make robot moves not that stupid. It refines the path by connecting points with straight lines, removing unnecessary turns while avoiding obstacles.



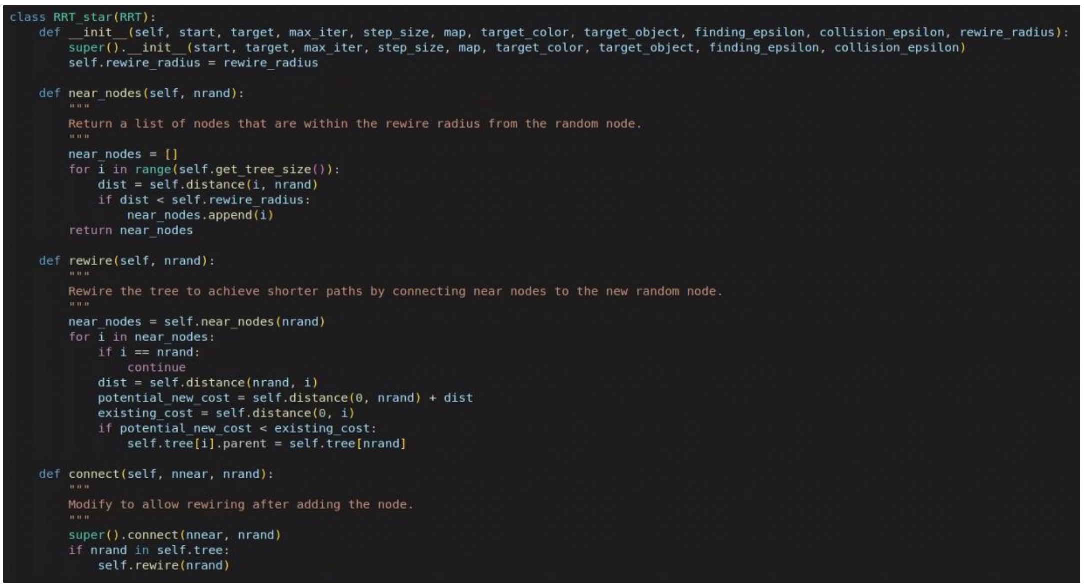
```python
def smooth(self):
    '''
    smooth the finding path
    '''
    s = 0 # s for start point
    self.smooth_path.append(self.path[s])
    for i in range(1, len(self.path)-1):
        current_point = np.array([self.path[s][0], self.path[s][1]])
        potential_point = np.array([self.path[i][0], self.path[i][1]])
        line = np.linspace(current_point, potential_point, 50, endpoint=True)
        line = line.astype(int)
        for point in line:
            finding_region = self.map[point[1] - self.collision_epsilon : point[1] + self.collision_epsilon,
                                      point[0] - self.collision_epsilon : point[0] + self.collision_epsilon]

            if (finding_region != [255, 255, 255]).any():
                self.smooth_path.append(self.path[i - 1])
                s = i - 1
                break

    self.smooth_path.append(self.path[-1])
```

Result:

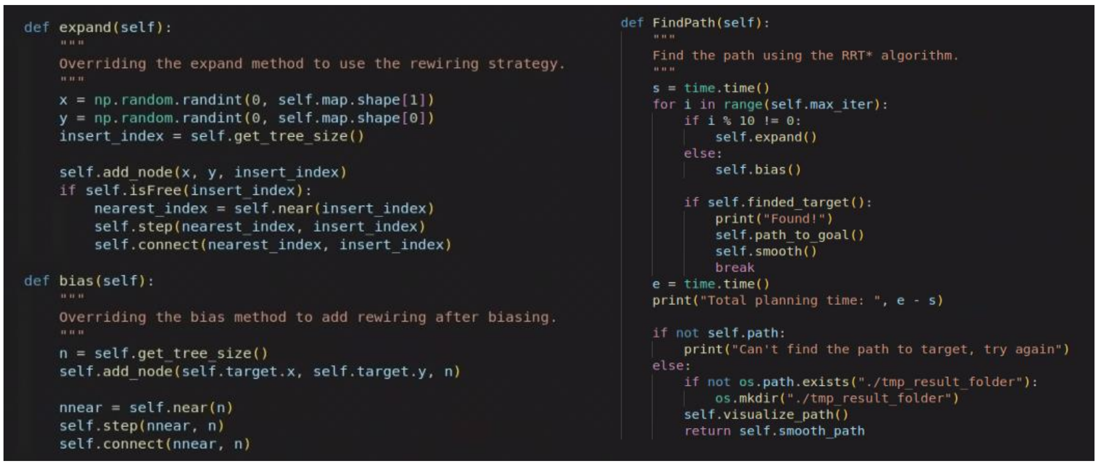
We can see the following figure, the smoothed path (green) looks more direct, efficient that reduces unnecessary turns compare to the original one (red one).



I also implement the RRT*, in order to compare to the original RRT.



```python
class RRT_star(RRT):
    def __init__(self, start, target, max_iter, step_size, map, target_color, target_object, finding_epsilon, collision_epsilon, rewire_radius):
        super().__init__(start, target, max_iter, step_size, map, target_color, target_object, finding_epsilon, collision_epsilon)
        self.rewire_radius = rewire_radius

    def near_nodes(self, nrand):
        """
        Return a list of nodes that are within the rewire radius from the random node.
        """
        near_nodes = []
        for i in range(self.get_tree_size()):
            dist = self.distance(i, nrand)
            if dist < self.rewire_radius:
                near_nodes.append(i)
        return near_nodes

    def rewire(self, nrand):
        """
        Rewire the tree to achieve shorter paths by connecting near nodes to the new random node.
        """
        near_nodes = self.near_nodes(nrand)
        for i in near_nodes:
            if i == nrand:
                continue
            dist = self.distance(nrand, i)
            potential_new_cost = self.distance(0, nrand) + dist
            existing_cost = self.distance(0, i)
            if potential_new_cost < existing_cost:
                self.tree[i].parent = self.tree[nrand]

    def connect(self, nnear, nrand):
        """
        Modify to allow rewiring after adding the node.
        """
        super().connect(nnear, nrand)
        if nrand in self.tree:
            self.rewire(nrand)
```

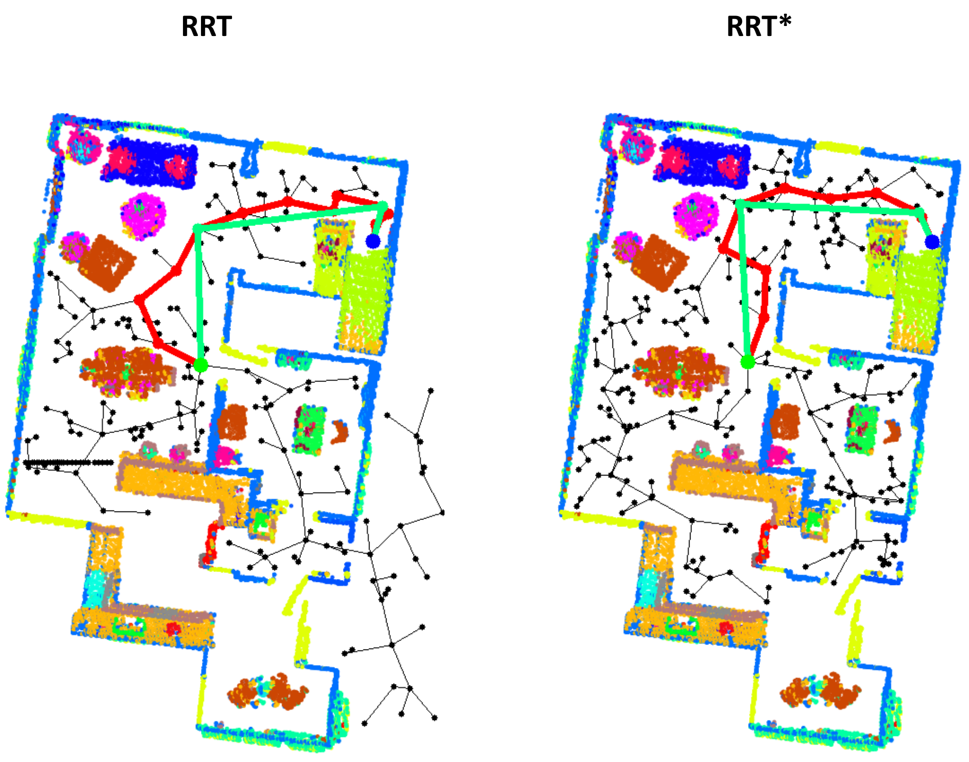


```python
def expand(self):
    """
    Overriding the expand method to use the rewiring strategy.
    """
    x = np.random.randint(0, self.map.shape[1])
    y = np.random.randint(0, self.map.shape[0])
    insert_index = self.get_tree_size()

    self.add_node(x, y, insert_index)
    if self.isFree(insert_index):
        nearest_index = self.near(insert_index)
        self.step(nearest_index, insert_index)
        self.connect(nearest_index, insert_index)

def bias(self):
    """
    Overriding the bias method to add rewiring after biasing.
    """
    n = self.get_tree_size()
    self.add_node(self.target.x, self.target.y, n)

    nnear = self.near(n)
    self.step(nnear, n)
    self.connect(nnear, n)
```

```python
def FindPath(self):
    """
    Find the path using the RRT* algorithm.
    """
    s = time.time()
    for i in range(self.max_iter):
        if i % 10 != 0:
            self.expand()
        else:
            self.bias()

        if self.finded_target():
            print("Found!")
            self.path_to_goal()
            self.smooth()
            break
    e = time.time()
    print("Total planning time: ", e - s)

    if not self.path:
        print("Can't find the path to target, try again")
    else:
        if not os.path.exists("./tmp_result_folder"):
            os.mkdir("./tmp_result_folder")
        self.visualize_path()
        return self.smooth_path
```

Result:

We can observe that in the left image (RRT), the red path is non-optimized and includes unnecessary detours. In contrast, the red path in the right image (RRT*) is more direct and efficient due to iterative optimization. But the improvements from RRT* are not very noticeable in this scenario. I think It’s because the space is too small, limiting the diversity of possible paths and making the optimization less apparent.


RRT
RRT*