Perception and Decision Making in Intelligent Systems
Homework 3

310551003 陳嘉慶

i. Code

Detailed explanation of your implementation

a. how do you do RRT algorithm

1. Initialize parameter:

All parameter are initialized in the beginning of code

```
if __name__ == '__main__':
    pcd_path = './semantic_3d_pointcloud/'
    points_path = 'point.npy'
    colors_path = 'color01.npy'
    fig_path = './figure/'
    rrt_path = './rrt'
    image_name = './rrt/rrt_start.png'
    test_scene =
"../dataset/apartment_0/apartment_0/habitat/mesh_semantic.ply"
    os.makedirs(fig_path, exist_ok=True)
    os.makedirs(rrt_path, exist_ok=True)

    # initial variable, parameter
    ceil_th = -1e-3  # remove ceiling: -1e-3,
    gd_th = -3.5e-2  # remove ground: -3.5e-2
    scale_num = 2  # scale between RRT and scene
    scale = 10000 / 255 * scale_num
    rot = 0  # rotation for point cloud
    count = 0  # image sequence name
    start_point_list = []  # list for store click points
    args = parser_arg()  # for input target by command

    list_num = get_target(args)  # 0: refrigerator, 1: rack, 2: cushion, 3:
lamp, 4: cooktop

    goal_colors = [[255, 0, 0], [0, 255, 133], [255, 9, 92], [160, 150, 20],
[7, 255, 224]]

    load_colors = loadmat('data/color101.mat')['colors']
    scene = "apartment_0"
    habitat_rotation = 1.0  # rotation step
    habitat_forward = 0.01  # forward step
    name = ""  # RGB images folder suffix
    current_degree = 0  # initial direction
```

2. Load numpy file, check on point could and save as 2D map for RRT:

The points and colors are given by .npy, which can be used np.load to get it. To prevent any problems on reading, points and colors are assigned to point clouds for checking and removing ceiling and ground. Then, convert to a 2D map for RRT.

```
points = np.load('%s%s' % (pcd_path, points_path))  # 10000 / 255
colors = np.load('%s%s' % (pcd_path, colors_path))  # / 255.

points, colors = show_pcd(points, colors)

x_lim, y_lim = save_main_png(points, colors, fig_path, '1st_floor')
```

Perception and Decision Making in Intelligent Systems
Homework 3

310551003 陳嘉慶

Remove ceiling and grounding points for point cloud:
Initialize point cloud for assign numpy array including points and colors, remove ceiling points and grounding points.

```
def show_pcd(points, colors):
    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(points)
    pcd.colors = o3d.utility.Vector3dVector(colors)

    vis = o3d.visualization.Visualizer()
    points, colors = rmv_ceiling_ground(pcd, vis, ceil_th, gd_th)  # -1e-3,
3.5e-2
    return points, colors
```

Remove ceiling and grounding points according to threshold, filter out upper bound and lower bound points, show point cloud to check remove correctly if necessary.

```
def rmv_ceiling_ground(pcd, vis, ceil_th, gd_th):
    pcd_tmp = copy.deepcopy(pcd)
    points = np.array(pcd_tmp.points)
    colors = np.array(pcd_tmp.colors)

    mask = points[:, 1] < ceil_th  # remove points at ceiling
    points = points[mask]
    colors = colors[mask]
    mask = points[:, 1] > gd_th  # remove points at ground
    points = points[mask]
    colors = colors[mask]
    return points, colors
```

Save 2D map for RRT:
Remove unnecessary axis(height) for 2D map, plot it accordingly and get the minimum and maximum value of points at both x and y axis.

```
def save_main_png(points, colors, fig_path, name):
    points_2d = np.delete(points, 1, axis=1)

    plt.figure()
    plt.scatter(points_2d[:, 1], points_2d[:, 0], c=colors, s=1)
    x_lim = plt.gca().get_xlim()  # get max and min value at x axis
    y_lim = plt.gca().get_ylim()  # get max and min value at y axis

    plt.savefig('{}{}'.format(fig_path, name))
    plt.close()
    x_lim = [x_lim[0] * scale, x_lim[1] * scale]  # scale and format as list
    y_lim = [y_lim[0] * scale, y_lim[1] * scale]
    return x_lim, y_lim
```

310551003 陳嘉慶

3. <u>Separate goal and obstacle points, collect start point and implement RRT:</u>

Separate goal and obstacle points utilize label and color code, show the saved 2D map for selecting start point (which reference from homework1) and implement RRT to generate a path from start point to goal point.

```
goal_points, goal_colors_list, obs_points, obs_colors = \
    seperate_target_obstacle(points, colors, goal_colors, fig_path, x_lim,
y_lim, listnum=list_num)

save_start_map(list(goal_points), x_lim, y_lim, list(obs_points),
obs_colors)   # show map for click start point
img = cv2.imread(image_name, 1)   # top_rgb, front_rgb

# click start point
cv2.imshow('image', img)
cv2.setMouseCallback('image', click_event)
cv2.waitKey(0)
cv2.imwrite('./save/color_img.jpg', img)
cv2.destroyAllWindows()
print("pixel: ", start_point_list)

start_point = get_start_point()
print("START POINT: ", start_point)
RRT_implementation(start_point, goal_points, x_lim, y_lim,
list(obs_points), obs_colors)
```

Separate target and obstacle points:

1. Copy a point cloud and delete the redundancy axis.
2. Find goal points by color code: Select the target which we input at beginning (or it default as 0: refrigerator), de-normalize colors from array and filter the points which equal the target color code. Then, get target/goal points from the above mask/idx.
3. Get obstacle points: use an inverse mask to get obstacle points, switch axis to [z, x] and insert size of every obstacle point.
4. Filter outlier goal points: calculate mean value goal points and set a threshold value to filter out noise points.
5. Plot png to check result

```
def seperate_target_obstacle(points, colors, target, fig_path, x_lim,
y_lim, listnum=0):
    # Copy a point cloud
    colors_tmp = copy.deepcopy(colors)
    points_tmp = copy.deepcopy(points)
    points_tmp = np.delete(points_tmp, 1, axis=1)    # del addition dimension

    # Find goal points by color code & Get target point
    target_points = target[listnum]
    arr = np.array(np.around(colors_tmp * 255.), dtype=int)
    mask = np.all(arr == target_points, axis=1)   # colors, arr
    idx = np.where(mask)
    points_target = points_tmp[idx]
    colors_target = colors_tmp[idx]   # / 255.
```

Perception and Decision Making in Intelligent Systems
Homework 3

310551003 陳嘉慶

```
    # Get obstacle points
    o_idx = np.where(~mask)
    points_obstacle = points_tmp[o_idx] * scale
    points_obstacle.T[[1, 0]] = points_obstacle.T[[0, 1]]  # x,z to z, x
    points_obstacle = np.insert(points_obstacle, 2, 0.5, axis=1) # insert
markersize for later use
    colors_obstacle = colors_tmp[o_idx]

    # Filter outlier points
    mask = np.all(abs(np.mean(points_target, axis=0) - points_target) <
2e-2, axis=1)  # filter
    idx = np.where(mask)
    points_target = points_target[idx] * scale
    points_target.T[[1, 0]] = points_target.T[[0, 1]]  # x, z to z, x
    colors_target = colors_target[idx]

    # plot png to check result
    save_png(points_target, colors_target, fig_path,
'target_{}'.format(target_points), x_lim, y_lim)
    save_png(points_obstacle, colors_obstacle, fig_path,
'obstacle_{}'.format(target_points), x_lim, y_lim)
    return points_target, colors_target, points_obstacle, colors_obstacle
```

Plot and save 2D map:
Save 2D map for obstacle and target points for checking.

```
def save_png(points, colors, fig_path, name, x_lim, y_lim):
    plt.figure()
    plt.scatter(points[:, 1], points[:, 0], c=colors, s=1)
    plt.xlim(x_lim)
    plt.ylim(y_lim)
    plt.savefig('{}{}'.format(fig_path, name))
    plt.close()
```

Save 2D map with obstacle and goal points:
Set figure/pixel size for 2D map, plot it as fixed min and max axis value to provide the same dimension figure every time, limit the figure as close as the set pixel size.

```
def save_start_map(goal_list, x_lim, y_lim, obstacle_list, obs_colors):
    plt.figure(figsize=(5.12, 5.12))  # pixel size
    plt.clf()
    # plot map and goal points
    if len(goal_list) > 1:
        for i in range(len(goal_list)):
            plt.plot(goal_list[i][0], goal_list[i][1], "ok", ms=1)
    else:
        plt.plot(goal_list[0], goal_list[1], "ok")

    plt.scatter(np.array(obstacle_list)[:, 0], np.array(obstacle_list)[:,
1], c=obs_colors, s=1)

    # standardize figure
    plt.axis('equal')
    plt.xlim((np.around(x_lim[0]), np.around(x_lim[1])))
    plt.ylim((np.around(y_lim[0]), np.around(y_lim[1])))
```

```
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(image_name, bbox_inches='tight', pad_inches=0)
    plt.close()
```

Get start point from click event:

1. Click a point from 2D map as start point, convert it from pixel coordinate to 2D map coordinate where all parameters are checked and confirmed by save 2D map, the original point for 2D map is at pixel (190, 270) which is considered as (dx, dy) translation from pixel to map coordinates.
2. Multiple pixels with inverse intrinsic matrix to get the 2D map coordinate, then, scale it to simplify RRT implementation, return the last pixel point to prevent click mistake.

```
def get_start_point():
    width = 480  # pixel = 480x480
    focal = width / (np.tan(np.pi / 4) * 2)
    in_matrix = np.array([[focal, 0, 190], [0, focal, 270], [0, 0, 1]])  #
dx, dy = center point at RRT map
    in_matrix_inv = np.linalg.inv(in_matrix)

    sensor_height_bev = 1
    for i in start_point_list:
        i.append(1)
        uv = np.array(i) * sensor_height_bev
        XY = np.dot(in_matrix_inv, np.reshape(uv, (3, 1))) * 10 * scale_num
# 9
    XY = np.array([XY[0][0], -XY[1][0]])
    return XY
```

4. RRT implementation:

Initialize RRT algorithm, use RRT planning to return a path and provide state every step if show animation is True

```
def RRT_implementation(start_point, goal_points, x_rand_area, y_rand_area,
obstacle_list, obs_colors):
    print('Start RRT planning!')
    show_animation = True

    rrt = RRT(goal_points, x_rand_area=x_rand_area, y_rand_area=y_rand_area,
obstacle_list=obstacle_list, expand_dis=1,
             max_iter=2000, epoch=4)
    path = rrt.rrt_planning(obs_colors, start=start_point,
goal=np.mean(goal_points, axis=0), animation=show_animation)
    print('RRT finished!')
    if show_animation and path:
        plt.savefig('./rrt/rrt.png')
        plt.show()
        plt.close()
```

1. Create node structure and initialize RRT parameter

Perception and Decision Making in Intelligent Systems
Homework 3

310551003 陳嘉慶

```
class Node:
   def __init__(self, x, y):
       self.x = x
       self.y = y
       self.cost = 0.0
       self.parent = None
```

```
class RRT:
   # initialize
   def __init__(self,
                goal_list,
                obstacle_list,  # obstacle points list
                x_rand_area,  # x sampling range
                y_rand_area,  # y sampling range
                expand_dis=2,  # step for build tree: 2.0
                goal_sample_rate=30,  # sample percentage toward goal
points
                max_iter=200, epoch=4):  # iteration for build RRT

       self.start = None
       self.goal = None
       self.goal_list = list(goal_list)
       # print("goal_mean: ", np.mean(goal_list, axis=0))
       # print("goal_list: ", self.goal_list)
       self.x_min_rand = x_rand_area[0]
       self.x_max_rand = x_rand_area[1]
       self.y_min_rand = y_rand_area[0]
       self.y_max_rand = y_rand_area[1]
       self.expand_dis = expand_dis
       self.goal_sample_rate = goal_sample_rate
       self.max_iter = max_iter
       self.obstacle_list = obstacle_list
       self.node_list = None
       self.progress = tqdm(total=self.max_iter / epoch)
       self.epoch = epoch
       self.distance = scale
       print("x random range: ", (np.around(self.x_min_rand),
np.around(self.x_max_rand)))
       print("y random range: ", (np.around(self.y_min_rand),
np.around(self.y_max_rand)))
```

2. Start RRT planning: Initial start and goal as node class structure, fix plot figure size.
3. Sample a point, find the nearest point to sample point from RRT.
4. Get new node by provided step and angle from RRT to sample point
5. Check collision for new nodes before adding to RRT, add to RRT path if no collision.
6. Check whether a new node is close enough to goal points.

```
def rrt_planning(self, obs_colors, start, goal, animation=True):  #
start=[x,y]
   print("GOAL CENTER: ", goal)
   self.start = Node(start[0], start[1])  # set start = struct with x, y,
cost, parent
   self.goal = Node(goal[0], goal[1])
   self.node_list = [self.start]
```

310551003 陳嘉慶

```
    path = None
    loop = self.max_iter / self.epoch
    plt.figure(figsize=(5.12, 5.12))  # pixel size
    for i in range(self.max_iter):
        # 1. Sample a point
        rnd = self.sample()

        # 2. Find the nearest built tree point connected to rnd point
        n_ind = self.get_nearest_list_index(self.node_list, rnd)
        nearest_node = self.node_list[n_ind]

        # 3. Get new node by provide step and calculated angle
        theta = math.atan2(rnd[1] - nearest_node.y, rnd[0] - nearest_node.x)
        new_node = self.get_new_node(theta, n_ind, nearest_node)

        # 4. Check collision: see any collision for a new point
        no_collision = self.check_segment_collision(new_node.x, new_node.y,
nearest_node.x, nearest_node.y)
        if no_collision:
            self.node_list.append(new_node)

            # plot each step
            if animation:
                time.sleep(1)
                self.draw_graph(obs_colors, new_node, path, i)

            # Check every new node near to goal point
            if self.is_near_goal(new_node):  # if close enough to goal point
consider no collision
                last_index = len(self.node_list) - 1
                path = self.get_final_course(last_index)   # Get the overall
RRT path
                path_length = self.get_path_len(path)  # Calculate path
length
                print("current path length : {}, # of nodes:
{}".format(path_length, len(path)))

                path_arr = np.array(path)  # L[::-1]: reverse read arr
                path_pf = pd.DataFrame(path_arr)
                path_pf.to_csv('./rrt/path.txt', header=False, index=False)

                if animation:
                    self.draw_graph(obs_colors, new_node, path, i)
                return path
        self.progress.update(1)
        if i % loop + 1 >= loop:  # Restart RRT if RRT failed
            self.progress.refresh()
            self.progress.reset()
            self.start = Node(start[0], start[1])
            self.goal = Node(goal[0], goal[1])
            self.node_list = [self.start]
            path = None
```

Sample point: uniform sample a point within a range from x,y minimum and maximum, set a rate to increase probability toward to goal points, default is 30% will sample on goal point.

Perception and Decision Making in Intelligent Systems
Homework 3

310551003 陳嘉慶

```
def sample(self):
    """ Uniform sample point and provide certain probability to sample
towrad goal """
    if random.randint(0, 100) > self.goal_sample_rate:
        rnd = [random.uniform(self.x_min_rand, self.x_max_rand),   # random
x,
               random.uniform(self.y_min_rand, self.y_max_rand)]   # random y
    else:
        rnd = [self.goal.x, self.goal.y]
    return rnd
```

Get the nearest node to sample point: compare the distance for all nodes and return the minimum distance index

```
def get_nearest_list_index(nodes, rnd):
    """ Find the nearest node from build tree to new node"""
    d_list = [(node.x - rnd[0]) ** 2 + (node.y - rnd[1]) ** 2
              for node in nodes]
    min_index = d_list.index(min(d_list))  # get the index of tree
    return min_index
```

Get new node: according to configured step, toward sample point a step.

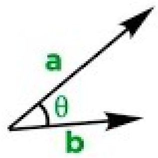
```
def get_new_node(self, theta, n_ind, nearest_node):
    """ Calculate new node """
    new_node = copy.deepcopy(nearest_node)

    new_node.x += self.expand_dis * math.cos(theta)  # calculate x and y
    new_node.y += self.expand_dis * math.sin(theta)

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

    return new_node
```

Check collision: check whether connecting a new node will have collision to obstacle points, no collision if the obstacle point projection on the new path is smaller than the markersize/radius we set.

```
def check_segment_collision(self, x1, y1, x2, y2):
    """ Check collision """
    for (ox, oy, radius) in self.obstacle_list:
        dd = self.distance_squared_point_to_segment(
            np.array([x1, y1]),  # new node
            np.array([x2, y2]),  # the nearest node from tree
            np.array([ox, oy])  # the obstacle point
        )
        # print("dd: ", dd)
        if dd <= radius ** 2:  # if projection < threshold: collision exist,
threshold=obstacle point size
            return False
    return True  # no collision
```

Calculate obstacle projection: calculate vector dot product as below formula and return projection of obstacle to a new path as $|a| \cdot cos\theta$

The Vector Dot Product



$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos\theta$$

$$\cos\theta = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|}$$

```
def distance_squared_point_to_segment(v, w, p):
    """ Calculate the shortest distance from a new line (tree node
connected a new node) and obstacle point p """
    if np.array_equal(v, w):  # if v and w are the same point
        return (p - v).dot(p - v)

    l2 = (w - v).dot(w - v)  # vector wv^2
    t = max(0, min(1, (p - v).dot(w - v) / l2))  # with t*(w-v): (pv dot
wv)/|wv||wv| = |pv||wv|cos(theta)
    projection = v + t * (w - v)  # obstacle projection vector on build
tree: e.g.: pv dot cos(theta)
    return (p - projection).dot(p - projection)
```

Draw graph: clear previous figure, draw a new node on it, check all the nodes and plot it. Also plot obstacle points and goal points. Fix the size of figure to show and save.

```
def draw_graph(self, obs_colors, rnd=None, path=None, step=0):

    plt.clf()  # clear figure for new plot

    # Draw a new node on fig
    if rnd is not None:
        plt.plot(rnd.x, rnd.y, '^k')

    # Plot RRT path
    for node in self.node_list:
        if node.parent is not None:
            if node.x or node.y is not None:
                plt.plot([node.x, self.node_list[node.parent].x],
                         [node.y, self.node_list[node.parent].y],
                         '-g')

    # Plot start point and goal point
    plt.plot(self.start.x, self.start.y, "og")
    # plt.plot(self.goal.x, self.goal.y, "or")
    if len(self.goal_list) > 1:
        for i in range(len(self.goal_list)):
            plt.plot(self.goal_list[i][0], self.goal_list[i][1], "ok", ms=1)
```

```
    else:
        plt.plot(self.goal.x, self.goal.y, "ob")

    # Plot obstacle points
    plt.scatter(np.array(self.obstacle_list)[:, 0],
np.array(self.obstacle_list)[:, 1], c=obs_colors, s=1)

    # Plot path
    if path is not None:
        plt.plot([x for (x, y) in path], [y for (x, y) in path], '-r')

    # Plot setup
    plt.axis('equal')
    plt.xlim((np.around(self.x_min_rand), np.around(self.x_max_rand)))
    plt.ylim((np.around(self.y_min_rand), np.around(self.y_max_rand)))
    plt.grid(True)
    plt.tight_layout()
    plt.savefig('./rrt/rrt{}.png'.format(step), bbox_inches='tight',
pad_inches=0)
    plt.pause(0.01)
```

Check the new point is near enough to goal points: check the new point distance to goal points and return True to end RRT planning if it is near enough, 1.5 is set as not all goals are exposed to walkable locations.

```
def is_near_goal(self, node):
    for goal in self.goal_list:
        d = self.line_cost(node, goal)
        # print("distance: ", d)
        if d < self.expand_dis * 1.5:
            return True
    return False

def line_cost(self, node1, goal):
    distance = math.sqrt((node1.x - goal[0]) ** 2 + (node1.y - goal[1]) **
2)
    if distance < self.distance: self.distance = distance
    self.progress.set_description("dis: {:.2f}".format(distance))
    return distance
```

Get the final path: check the parent from the goal point and get the RRT path.

```
def get_final_course(self, last_index):
    """ Get the path from goal point, check parent"""
    min_path = 10
    min_path_idx = 0
    for idx, goal in enumerate(self.goal_list):
        d = self.line_cost(self.node_list[last_index], goal)
        if d < min_path:
            min_path = d
            min_path_idx = idx

    path = [[self.goal_list[min_path_idx][0],
self.goal_list[min_path_idx][1]]]  # [self.goal.x, self.goal.y]
    while self.node_list[last_index].parent is not None:
```

310551003 陳嘉慶

```
        node = self.node_list[last_index]
        path.append([node.x, node.y])
        last_index = node.parent
    path.append([self.start.x, self.start.y])
    return path
```

Calculate the length of RRT path:

```
def get_path_len(path):
    """ Calculate path length"""
    path_length = 0
    for i in range(1, len(path)):
        node1_x = path[i][0]
        node1_y = path[i][1]
        node2_x = path[i - 1][0]
        node2_y = path[i - 1][1]
        path_length += math.sqrt((node1_x - node2_x) ** 2 + (node1_y -
node2_y) ** 2)
    return path_length
```

b. how do you convert route to discrete actions

1. Read the RRT path and set the start point accordingly.
2. Initialize simulator and agent

```
# Read RRT path and initial start point
rrt_arr = read_rrt(rrt_path=rrt_path, list_num=list_num, mode='Nrecord')
start_x = rrt_arr[0][0]  # agent in world space,  2.6, 2.5
start_z = rrt_arr[0][1]  # 7.8, -2.5

# Initialize simulator and agent
sim_settings = {
    "scene": test_scene,  # Scene path
    "default_agent": 0,  # Index of the default agent
    "sensor_height": 1.5,  # Height of sensors in meters, relative to the
agent
    "width": 512,  # Spatial resolution of the observations
    "height": 512,
    "sensor_pitch": -np.pi / 2,  # sensor pitch (x rotation in rads)
}
cfg = make_simple_cfg(sim_settings)
sim = habitat_sim.Simulator(cfg)
agent = sim.initialize_agent(sim_settings["default_agent"])

# Set agent state
agent_state = habitat_sim.AgentState()
agent_state.position = np.array([start_x, 0.0, start_z])  # agent
coordinate in world space
agent.set_state(agent_state)

# obtain the default, discrete actions that an agent can perform
# default action space contains 3 actions: move_forward, turn_left, and
turn_right
action_names =
list(cfg.agents[sim_settings["default_agent"]].action_space.keys())
init_file(name, list_num=list_num)
```

Perception and Decision Making in Intelligent Systems
Homework 3

310551003 陳嘉慶

Read RRT path: according to the mode, select to read the record path or the new generated path from RRT. As the path returns from goal to start, it needs to read from the end to start, swap also (z,x) to (x,z).

```
def read_rrt(scale_num=2, rrt_path='./rrt/backup/all_pc_scale-2/5_-5',
list_num=0, mode='record'):
    # rrt_path = './rrt/backup/all_pc_scale-2/5_-5'
    if mode =='record:':
        rrt_arr = np.array(pd.read_csv('{}/{}/path.txt'.format(rrt_path,
list_num), header=None)[::-1])
    else:
        rrt_arr = np.array(pd.read_csv('{}/path.txt'.format(rrt_path),
header=None)[::-1])
    rrt_arr.T[[1, 0]] = rrt_arr.T[[0, 1]]  # (z,x) -> (x,z)
    rrt_arr = rrt_arr / scale_num
    return rrt_arr
```

```
def init_file(name, list_num=0):
    os.makedirs('./save/', exist_ok=True)
    fp = open('./save/record%s.txt' % name, 'w')
    fp.close()
    os.makedirs('./save/RGB%s/' % name, exist_ok=True)
    os.makedirs('./save/depth%s/' % name, exist_ok=True)
    os.makedirs('./save/semantic/', exist_ok=True)
    os.makedirs('./save/RGB_bev/', exist_ok=True)
    os.makedirs('./save/depth_bev/', exist_ok=True)
    os.makedirs('./save/semantic_bev/', exist_ok=True)
    navigateAndSee("move_forward", list_num=list_num)
```

3. <u>Start Navigation:</u>
   (optional) utilize the simulator sensor information including location and degree to verify result. As we have the RRT path, the rotation degree and forward step can be estimated. Then, the control simulator follows the estimated value, moving forward after rotating to the correct direction. Finally, according to the goal, fine tune the rotation toward the object if necessary.

```
print("\n ==================================Start
Navigation================================== \n")
print("Discrete action space: ", action_names)

sensor = record_path()  # get habitat sensor [x,z,rw,deg]
print(sensor[0], rrt_arr[0][0], rrt_arr[-1][0], sensor[1], rrt_arr[0][1],
rrt_arr[-1][1])
print("sensor start point: ", sensor[0], sensor[1])
print("rrt start point: {}, next: {}, total shape: {}".format(rrt_arr[0],
rrt_arr[1], rrt_arr.shape))

progress = tqdm(total=rrt_arr.shape[0] - 1)

# Start Navigation
for idx in range(1, rrt_arr.shape[0]):
    rotate, action, current_degree = estimate_turn(rrt_arr[idx], rrt_arr[idx
- 1], current_degree)
    count = take_turn(rotate, action, count, list_num=list_num)  # rotate
```

310551003 陳嘉慶

```
    move = np.array([rrt_arr[idx][0] - rrt_arr[idx - 1][0], rrt_arr[idx][1]
- rrt_arr[idx - 1][1]])
    move = np.sqrt(move[0] ** 2 + move[1] ** 2)
    count, move = take_forward(move, "move_forward", count,
list_num=list_num)  # move forward
    sensor = record_path()  # get current sensor info
    progress.update(1)
    progress.set_description("rotate {}: {:.2f}, forward:
{:.2f}".format(action, rotate, move))
    # prev_rotate = rotate

rotate, action, current_degree = final_turn(current_degree,
list_num=list_num)
count = take_turn(rotate, action, count, list_num=list_num)
progress.update(1)
print("Navigation Finished!")
```

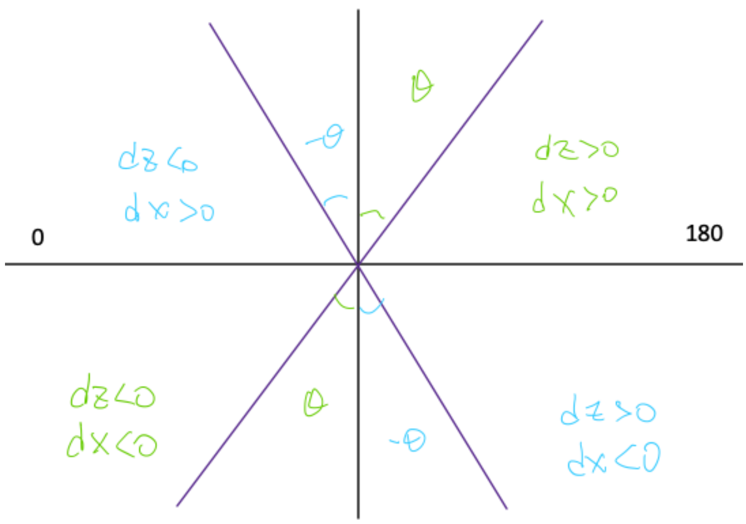
Record and print simulator information for verification

```
def record_path():
    agent_state = agent.get_state()
    sensor_state = agent_state.sensor_states['color_sensor']
    x, y, z, rw, rx, ry, rz = sensor_state.position[0],
sensor_state.position[1], sensor_state.position[2], \
           sensor_state.rotation.w, sensor_state.rotation.x,
sensor_state.rotation.y, \
           sensor_state.rotation.z
    deg = np.around(np.arccos(rw) * 360 / np.pi)
    # print("camera pose: x y z rw rx ry rz deg")
    # print(x, y, z, rw, rx, ry, rz, deg)  # convert quaternion to degree
    fp = open('./save/record%s.txt' % name, 'a')
    print(x, y, z, rw, rx, ry, rz, deg, file=fp)
    fp.close()
    return np.array([x, z, rw, deg])
```

Estimate rotation degree: according to the RRT path, estimate the rotation degree, calculate (x,z) different between 2 nodes. If target x > current x, target degree will be in range [180,360] .
For arctan θ will not be larger than 90 degree, in different casa of dz, arctan θ will be in range of (-90, 90). As arctan θ < 0, target degree will be in (270,360] or arctan θ > 0, target degree will be in (180,270].
Below figure will show you more clearly, note that the 0 degree in the simulator starts from the left hand side and the target degree may be larger than 180, if so, the target degree will be switched to anti-direction, thus 360 - target degree.

Perception and Decision Making in Intelligent Systems
Homework 3

310551003 陳嘉慶




```
def estimate_turn(rrt_arr_target, rrt_arr_current, current_degree):
    diff_x = rrt_arr_target[0] - rrt_arr_current[0]
    diff_z = rrt_arr_target[1] - rrt_arr_current[1]
    target_rotate = np.around(np.degrees(np.arctan(diff_z / diff_x)))
    if diff_x > 0:
        target_degree = 270 - target_rotate
    else:
        target_degree = 90 - target_rotate

    rotate = current_degree - target_degree  # current degree - target
degree
    if rotate > 0:
        action = "turn_right"
        if rotate > 180:
            action = "turn_left"
            rotate = 360 - rotate
    else:
        action = "turn_left"
        rotate = abs(rotate)
        if rotate > 180:
            action = "turn_right"
            rotate = 360 - rotate
    return rotate, action, target_degree
```

Rotate simulator sensor: following the calculated target degree, rotate the sensor until the current degree smaller than target. Rotation step is 1 degree.

```
def take_turn(rotate, action, count, list_num=0):
    rotate_temp = copy.deepcopy(rotate)
    while rotate_temp > habitat_rotation / 2:
        count = navigateAndSee(action, count, name, list_num=list_num)
        rotate_temp = abs(rotate_temp - habitat_rotation)
    return count
```

Perception and Decision Making in Intelligent Systems
Homework 3

310551003 陳嘉慶

Move forward: Utilize two nodes distance to determine move forward step, move sensor until it reaches target point.

```
def take_forward(forward, action, count, list_num=0):
    forward_temp = copy.deepcopy(forward)
    while forward_temp > habitat_forward / 2:
        count = navigateAndSee(action, count, name, list_num=list_num)
        forward_temp = abs(forward_temp - habitat_forward)
        # print(forward_temp)
    # print("action: ", action)
    return count, forward_temp
```

Fine tune rotation degree toward target if necessary(optional).

```
def final_turn(current_degree, list_num=0):
    if list_num == '1' or list_num == 1:  # rack
        target_degree = 200
    elif list_num == '2' or list_num == 2:  # cushion
        target_degree = current_degree
    elif list_num == '3' or list_num == 3:  # lamp
        target_degree = current_degree
    elif list_num == '4' or list_num == 4:  # cooktop
        target_degree = 90
    else:  # refrigerator
        target_degree = current_degree
    rotate = current_degree - target_degree  # current degree - target
degree
    print("target_degree: {}, current_degree: {}, rotate:
{}".format(target_degree, current_degree, rotate))
    if rotate > 0:
        action = "turn_right"
        if rotate > 180:
            action = "turn_left"
            rotate = 360 - rotate
    else:
        action = "turn_left"
        rotate = abs(rotate)
        if rotate > 180:
            action = "turn_right"
            rotate = 360 - rotate
    return rotate, action, target_degree
```

Sensor navigate function: the sensor will move according to action and a mask will be cover to a target object at RGB sensor.

```
def navigateAndSee(action="", num=0, name="", list_num=0):
    if action in action_names:
        observations = sim.step(action)
        img_rgb = transform_rgb_bgr(observations["color_sensor"])
        save_color_observation(observations["color_sensor"], num,
out_folder='./save/generate/images/')
        img_sem101 =
save_semantic_observation(observations["semantic_sensor"], num,

scene_dict=load_scene_semantic_dict(),
```

310551003 陳嘉慶

```
out_folder='./save/generate/annotations/')
        save_depth_observation(observations["depth_sensor"], num, 
out_folder='./save/generate/depth/')
        idx, goal_colors = get_mask(img_sem101, list_num=list_num)
        img_rgb_new = mask_RGB(img_rgb, idx, goal_colors)
        cv2.imshow("RGB", img_rgb_new)
        cv2.imwrite("./save/RGB{}/{}.png".format(name, num), img_rgb)
        cv2.waitKey(1)
        agent_state = agent.get_state()
        sensor_state = agent_state.sensor_states['color_sensor']
        fp = open('./save/record%s.txt' % name, 'a')
        fp.close()
        num += 1
        return num
```

Mask: get a mask according to target semantic, check semantic color return pixel index, then cover a mask on RGB sensor, tune the mask as transparent red.

```
def get_mask(image, list_num=0):
    '''# 0: refrigerator, 1: rack, 2: cushion, 3: lamp, 4: cooktop
    # 0: [0, 163, 255], 1: [0, 0, 255], 2: [255, 9, 112], 3: [255, 163, 
0], 4: [6, 184, 255]'''
    image_tmp = copy.deepcopy(image).reshape(-1, 3)
    goal_colors = [np.array([0, 163, 255]), np.array([0, 0, 255]), 
np.array([255, 9, 112]),np.array([255, 163, 0]), np.array([6, 184, 255])]
    goal = goal_colors[list_num]
    mask = np.all(image_tmp == goal, axis=1)
    idx = np.where(mask)
    return idx, list_num

def mask_RGB(image, idx, list_num):
    if len(idx[0]) > 1:
        arr_idx = np.array([])
        image_tmp = copy.deepcopy(image).reshape(-1, 3)

        if list_num == 1:
            for i, arr in enumerate(image_tmp[idx[0]]):
                arr = np.array([arr[0], arr[1], arr[2] + 128])
                arr_idx = np.concatenate((arr_idx, arr))
        elif list_num == 3:
            for i, arr in enumerate(image_tmp[idx[0]]):
                mean = np.mean(arr)
                if mean > 128 and idx[0][i] > 133120:  # y>260, mean > 128
                    arr = np.array([arr[0] - 128, arr[1] - 128, 255])
                arr_idx = np.concatenate((arr_idx, arr))
        else:
            for i in image_tmp[idx[0]]:
                i = np.array([i[0], i[1], 255])
                arr_idx = np.concatenate((arr_idx, i))
        arr_idx = arr_idx.reshape(-1, 3)
        image_tmp[idx[0]] = arr_idx
        image_tmp = image_tmp.reshape(512, 512, 3)
        return image_tmp
    else:
        return image
```

ii. Results and Discussion

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

I chose 2 start points for targets, the results shown as below Figures.

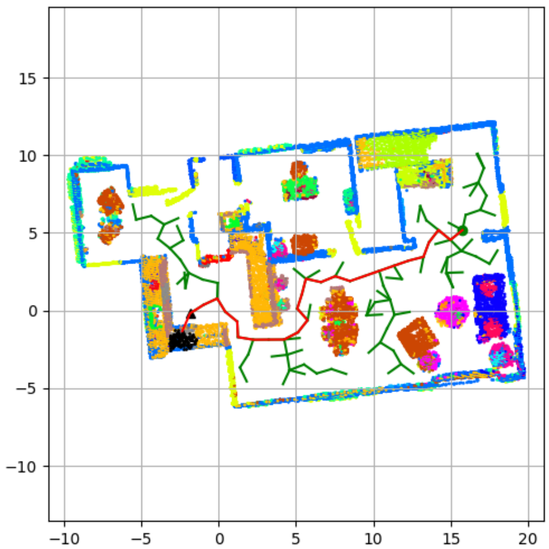


Figure 2.1 cooktop

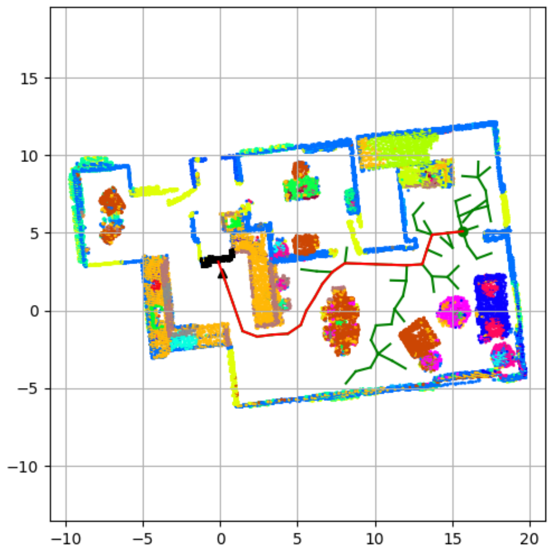


Figure 2.2 refrigerator

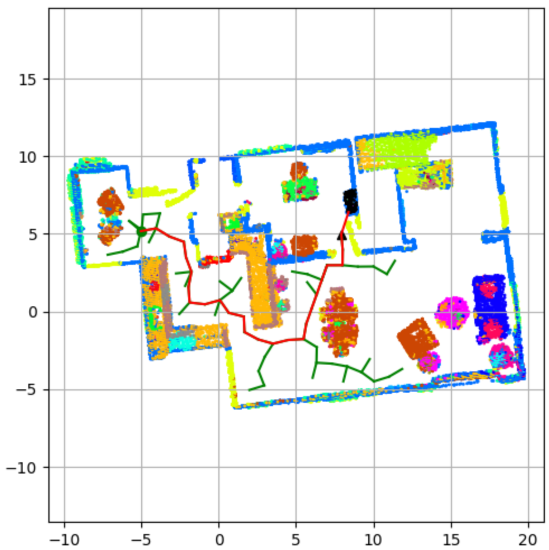


Figure 2.3 rack

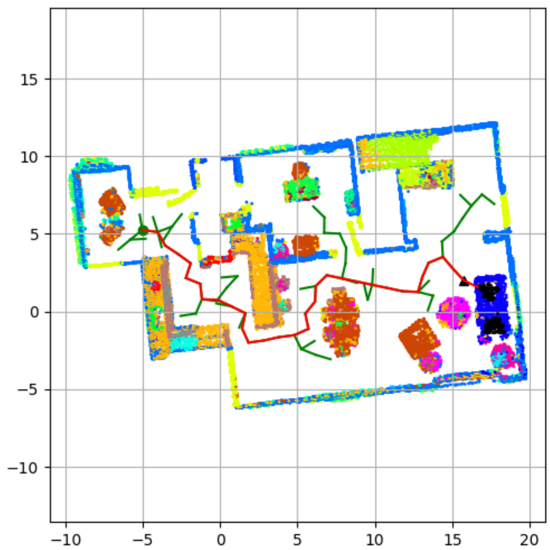


Figure 2.4 cushion

RRT succeeds in finding a path to targets, but sometimes it may find worse nodes as a result it cannot find target nodes easily. If the start point is set inside the room, it will need more time for getting out of the room and even cannot get out of the room in certain iterations as it is Rapidly-exploring RANDOM tree.

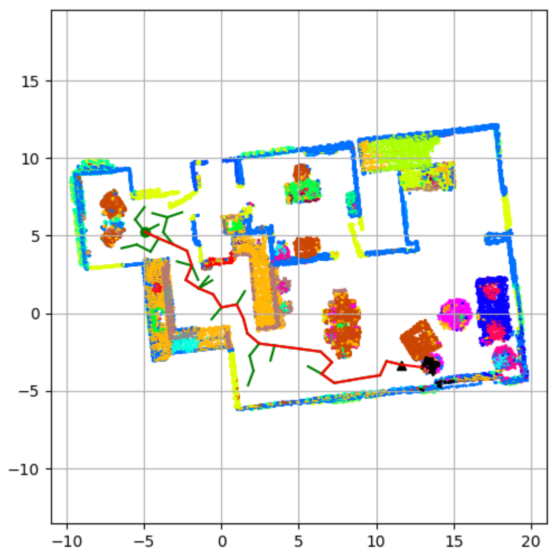


Figure 2.5 lamp

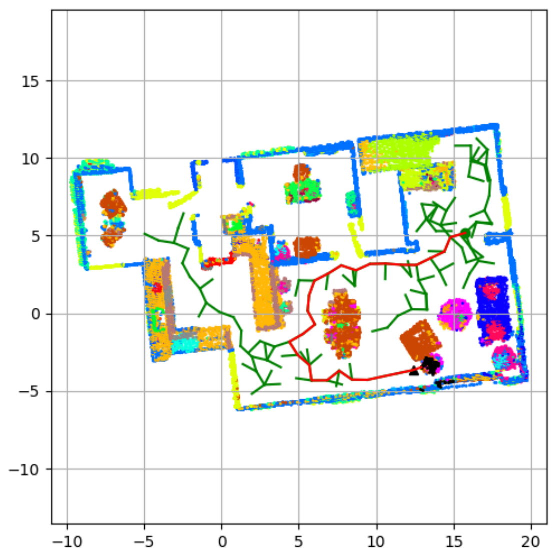


Figure 2.6 lamp

In Figure 2.6, we can easily obtain that the RRT path is not optimized, because a random point is not good enough for determining the best path, but it still can reach the target object.

2. Discuss about the robot navigation results

As the steps for rotation and move are 1 and 0.1, it is quite accurate to follow the path. However, if the step is large, it may not be good enough to follow the path, that is, the accumulated errors may cause the wrong path and finally cannot reach the target object, especially will make collisions to the environment.

3. Anything you want to discuss

I try to increase the number of iterations (to 2000) in the beginning for finding the target, but it is not a good approach to reach the target obviously, thus, I add a loop to restart RRT implementation in case a certain iteration(500) cannot reach the target object. In this approach, I find that 2000 iterations still cannot find the target object, but if I divide it to 4 rounds of 500 iterations. The success rate is much better than 2000 iterations.

In the other hand, we can use RRT* to improve the path for navigation, or control the random direction when collision since random sometime is not smart enough.

iii. Reference

https://blog.csdn.net/feriman/article/details/113828617
https://docs.python.org/release/2.3.5/whatsnew/section-slices.html
https://stackoverflow.com/questions/46761745/replace-values-in-an-nd-numpy-array-at-given-axis-index
https://stackoverflow.com/questions/50608021/replace-elements-of-particular-axis-index-in-n-dimensional-numpy-array

Perception and Decision Making in Intelligent Systems
Homework 3

310551003 陳嘉慶

https://clideo.com/editor/video-maker
https://indianaiproduction.com/image-to-video-opencv/
https://blog.csdn.net/weixin_43869605/article/details/119826406