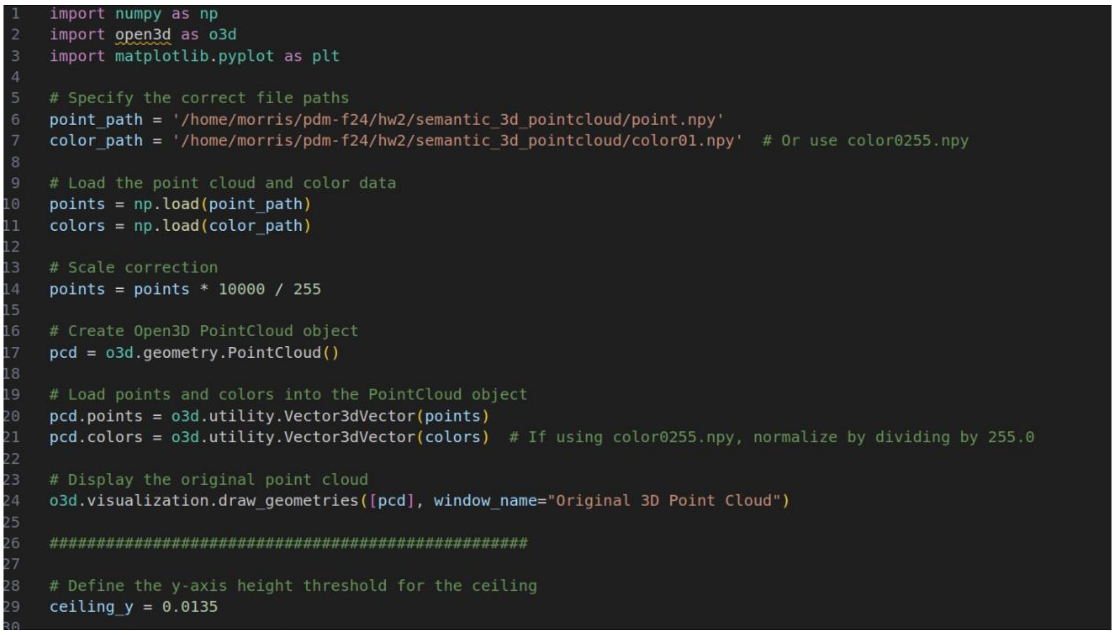
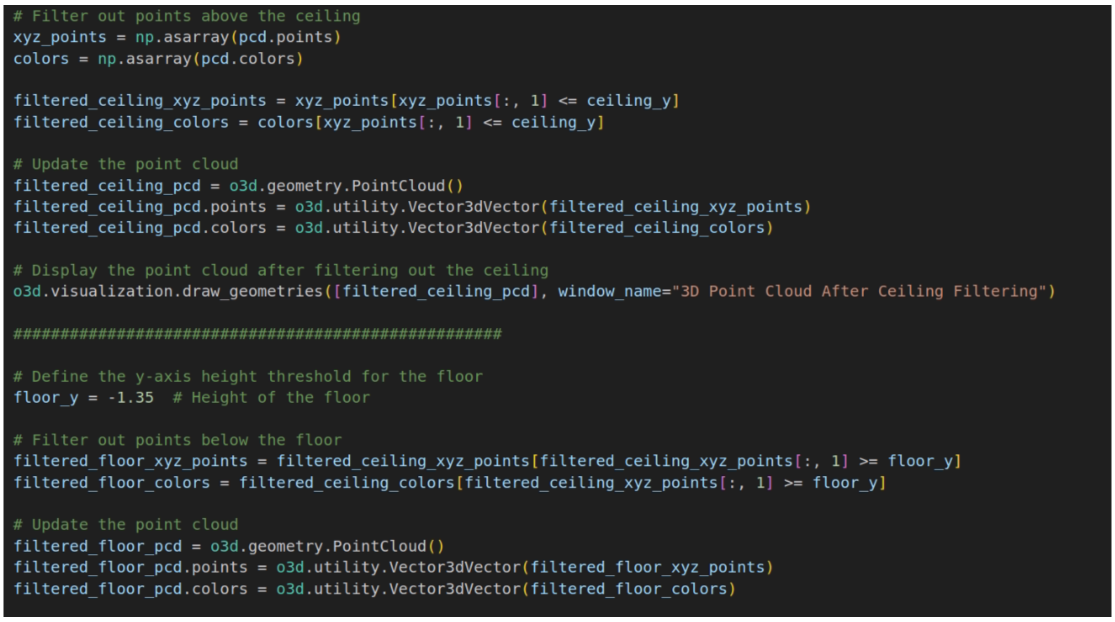
1. Implementation

a. Code

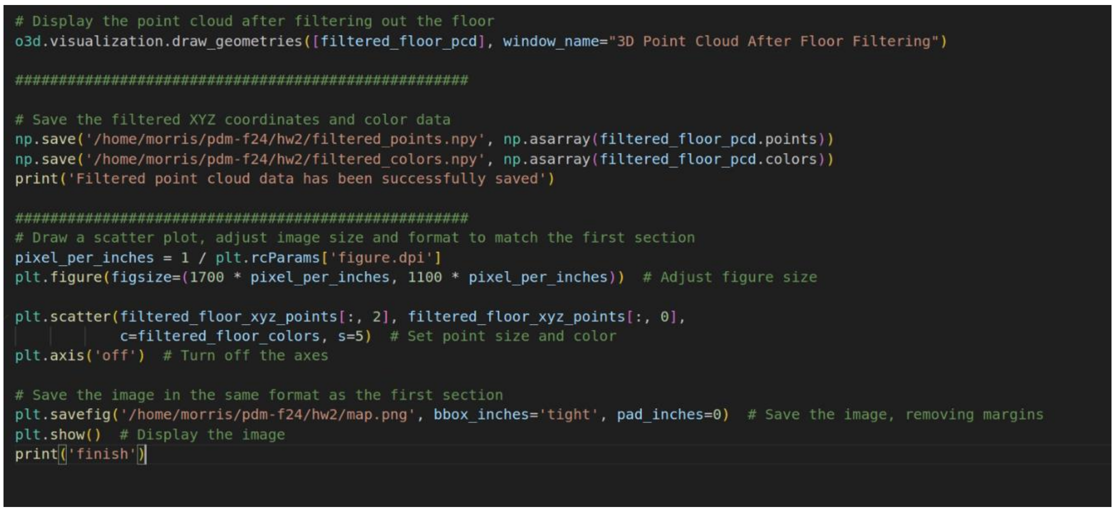
Part 1:2D semantic map construction



```python
import numpy as np
import open3d as o3d
import matplotlib.pyplot as plt

# Specify the correct file paths
point_path = '/home/morris/pdm-f24/hw2/semantic_3d_pointcloud/point.npy'
color_path = '/home/morris/pdm-f24/hw2/semantic_3d_pointcloud/color01.npy'  # Or use color0255.npy

# Load the point cloud and color data
points = np.load(point_path)
colors = np.load(color_path)

# Scale correction
points = points * 10000 / 255

# Create Open3D PointCloud object
pcd = o3d.geometry.PointCloud()

# Load points and colors into the PointCloud object
pcd.points = o3d.utility.Vector3dVector(points)
pcd.colors = o3d.utility.Vector3dVector(colors)  # If using color0255.npy, normalize by dividing by 255.0

# Display the original point cloud
o3d.visualization.draw_geometries([pcd], window_name="Original 3D Point Cloud")

#####################################################

# Define the y-axis height threshold for the ceiling
ceiling_y = 0.0135
```



```python
# Filter out points above the ceiling
xyz_points = np.asarray(pcd.points)
colors = np.asarray(pcd.colors)

filtered_ceiling_xyz_points = xyz_points[xyz_points[:, 1] <= ceiling_y]
filtered_ceiling_colors = colors[xyz_points[:, 1] <= ceiling_y]

# Update the point cloud
filtered_ceiling_pcd = o3d.geometry.PointCloud()
filtered_ceiling_pcd.points = o3d.utility.Vector3dVector(filtered_ceiling_xyz_points)
filtered_ceiling_pcd.colors = o3d.utility.Vector3dVector(filtered_ceiling_colors)

# Display the point cloud after filtering out the ceiling
o3d.visualization.draw_geometries([filtered_ceiling_pcd], window_name="3D Point Cloud After Ceiling Filtering")

#####################################################

# Define the y-axis height threshold for the floor
floor_y = -1.35  # Height of the floor

# Filter out points below the floor
filtered_floor_xyz_points = filtered_ceiling_xyz_points[filtered_ceiling_xyz_points[:, 1] >= floor_y]
filtered_floor_colors = filtered_ceiling_colors[filtered_ceiling_xyz_points[:, 1] >= floor_y]

# Update the point cloud
filtered_floor_pcd = o3d.geometry.PointCloud()
filtered_floor_pcd.points = o3d.utility.Vector3dVector(filtered_floor_xyz_points)
filtered_floor_pcd.colors = o3d.utility.Vector3dVector(filtered_floor_colors)
```

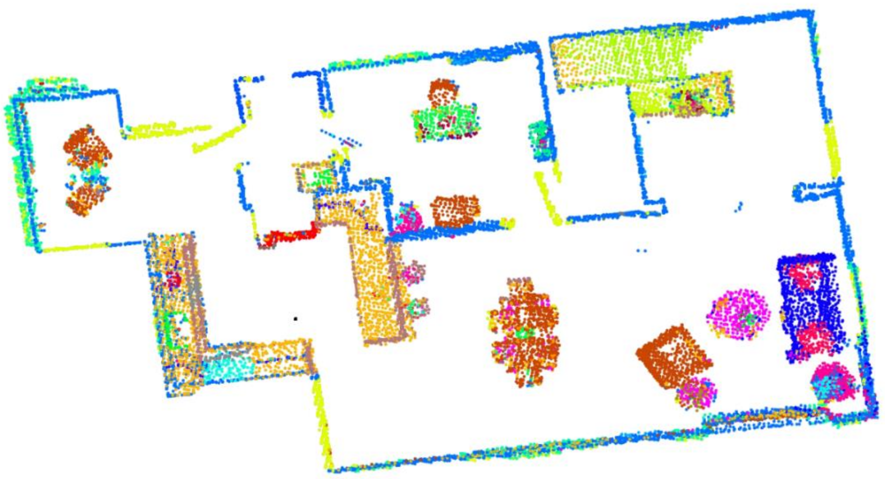


```python
# Display the point cloud after filtering out the floor
o3d.visualization.draw_geometries([filtered_floor_pcd], window_name="3D Point Cloud After Floor Filtering")

####################################################

# Save the filtered XYZ coordinates and color data
np.save('/home/morris/pdm-f24/hw2/filtered_points.npy', np.asarray(filtered_floor_pcd.points))
np.save('/home/morris/pdm-f24/hw2/filtered_colors.npy', np.asarray(filtered_floor_pcd.colors))
print('Filtered point cloud data has been successfully saved')

####################################################
# Draw a scatter plot, adjust image size and format to match the first section
pixel_per_inches = 1 / plt.rcParams['figure.dpi']
plt.figure(figsize=(1700 * pixel_per_inches, 1100 * pixel_per_inches))  # Adjust figure size

plt.scatter(filtered_floor_xyz_points[:, 2], filtered_floor_xyz_points[:, 0],
            c=filtered_floor_colors, s=5)  # Set point size and color
plt.axis('off')  # Turn off the axes

# Save the image in the same format as the first section
plt.savefig('/home/morris/pdm-f24/hw2/map.png', bbox_inches='tight', pad_inches=0)  # Save the image, removing margins
plt.show()  # Display the image
print('finish')
```

The main purpose of this program is to process a 3D semantic point cloud, filter out unnecessary parts (the ceiling and the floor), and generate a corresponding 2D map. The specific steps are as follows:

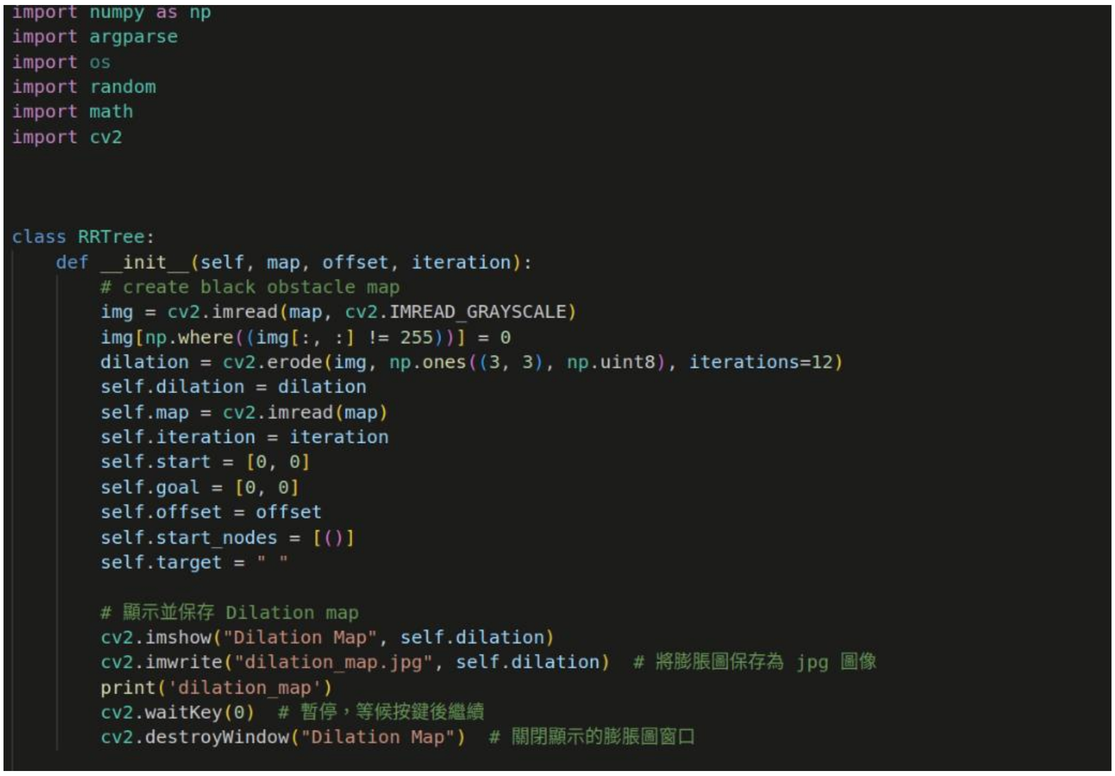
1. **Data Loading: Load the point cloud coordinates (point.npy) and color information (color01.npy) from the specified paths.**
2. **Scale Correction: Scale the point cloud coordinates to ensure correct display.**
3. **Display the Original Point Cloud: Use Open3D to visualize the loaded 3D point cloud.**
4. **Ceiling Filtering: Filter out points above the ceiling based on a height threshold, create a new point cloud, and display it.**
5. **Floor Filtering: Filter out points below the floor, generate a new filtered point cloud, and display it.**
6. **Save Filtered Data: Save the filtered point cloud coordinates and color information as .npy files for later use.**
7. **Generate 2D Map: Create a 2D scatter plot using the filtered point cloud data, save it as map.png, and use it to display the filtered scene.**

This process converts a complex 3D point cloud into a useful 2D view and saves the results.

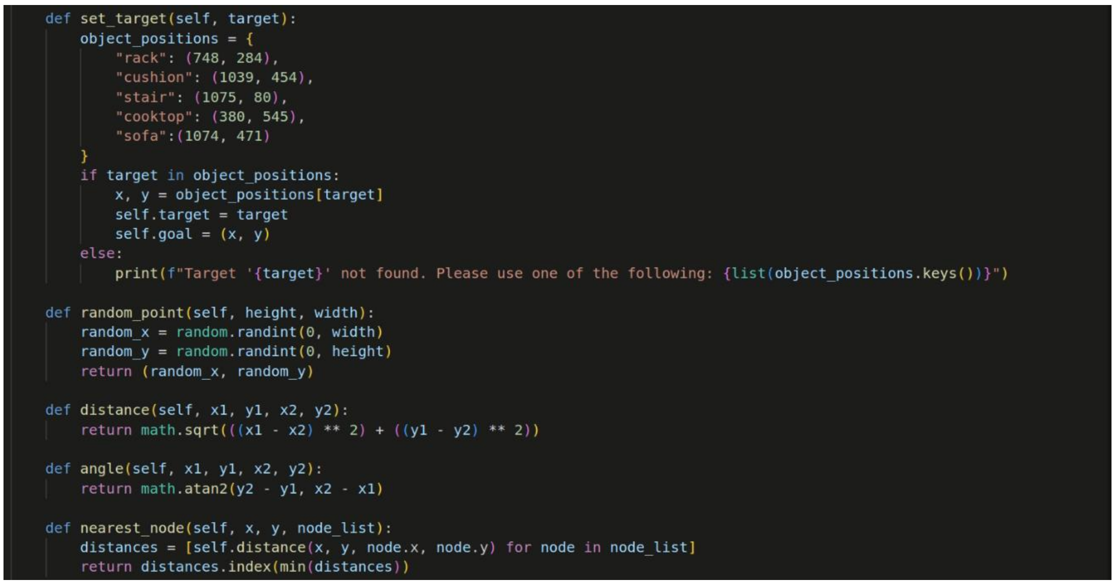


2D semantic map of the scene

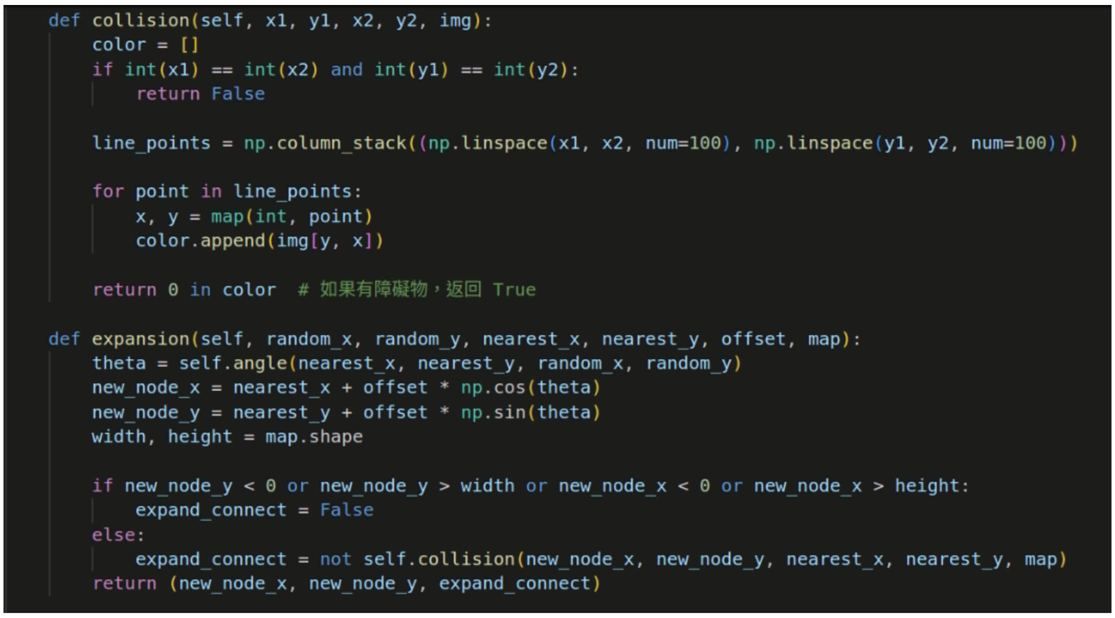
Part 2: RRT Algorithm



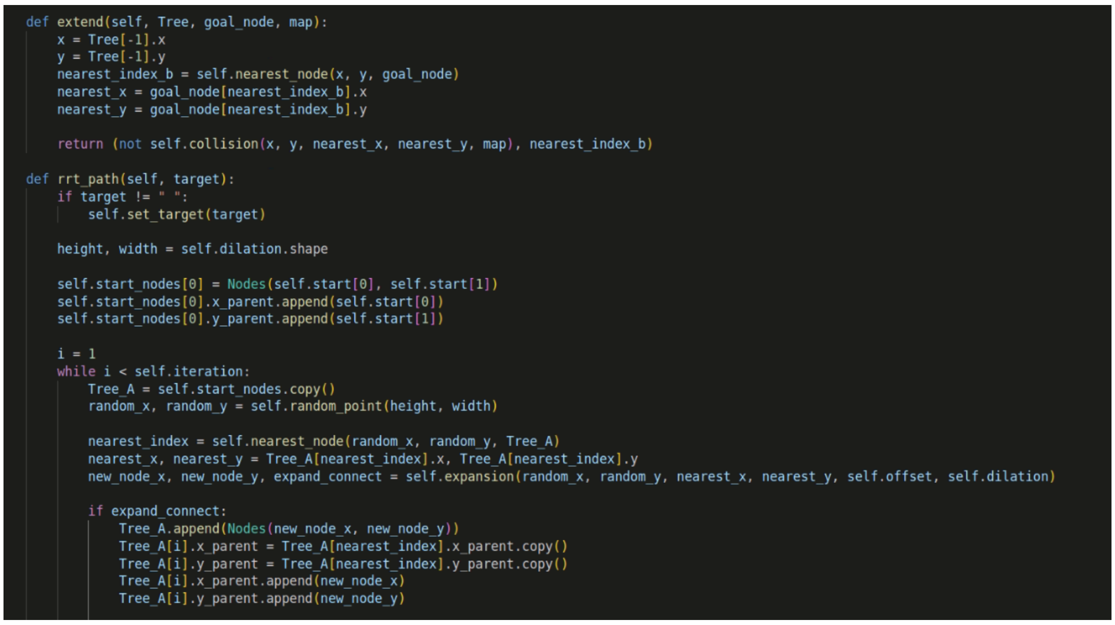
```python
import numpy as np
import argparse
import os
import random
import math
import cv2



class RRTree:
    def __init__(self, map, offset, iteration):
        # create black obstacle map
        img = cv2.imread(map, cv2.IMREAD_GRAYSCALE)
        img[np.where((img[:, :] != 255))] = 0
        dilation = cv2.erode(img, np.ones((3, 3), np.uint8), iterations=12)
        self.dilation = dilation
        self.map = cv2.imread(map)
        self.iteration = iteration
        self.start = [0, 0]
        self.goal = [0, 0]
        self.offset = offset
        self.start_nodes = [()]
        self.target = " "

        # 顯示並保存 Dilation map
        cv2.imshow("Dilation Map", self.dilation)
        cv2.imwrite("dilation_map.jpg", self.dilation)  # 將膨脹圖保存為 jpg 圖像
        print('dilation_map')
        cv2.waitKey(0)  # 暫停，等候按鍵後繼續
        cv2.destroyWindow("Dilation Map")  # 關閉顯示的膨脹圖窗口
```



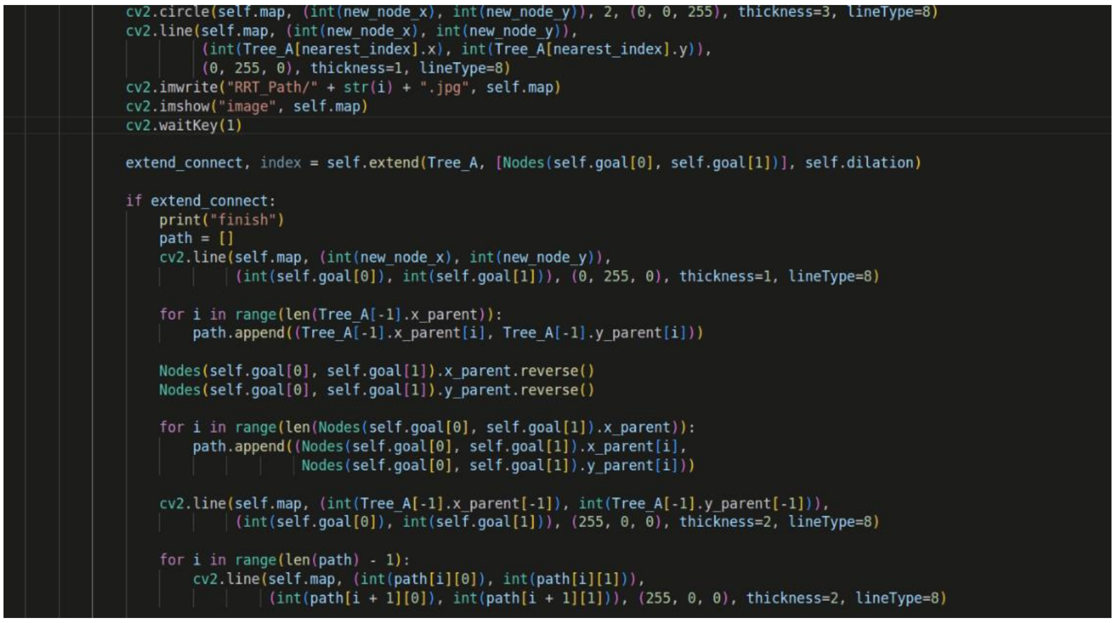
```python
    def set_target(self, target):
        object_positions = {
            "rack": (748, 284),
            "cushion": (1039, 454),
            "stair": (1075, 80),
            "cooktop": (380, 545),
            "sofa":(1074, 471)
        }
        if target in object_positions:
            x, y = object_positions[target]
            self.target = target
            self.goal = (x, y)
        else:
            print(f"Target '{target}' not found. Please use one of the following: {list(object_positions.keys())}")

    def random_point(self, height, width):
        random_x = random.randint(0, width)
        random_y = random.randint(0, height)
        return (random_x, random_y)

    def distance(self, x1, y1, x2, y2):
        return math.sqrt(((x1 - x2) ** 2) + ((y1 - y2) ** 2))

    def angle(self, x1, y1, x2, y2):
        return math.atan2(y2 - y1, x2 - x1)

    def nearest_node(self, x, y, node_list):
        distances = [self.distance(x, y, node.x, node.y) for node in node_list]
        return distances.index(min(distances))
```



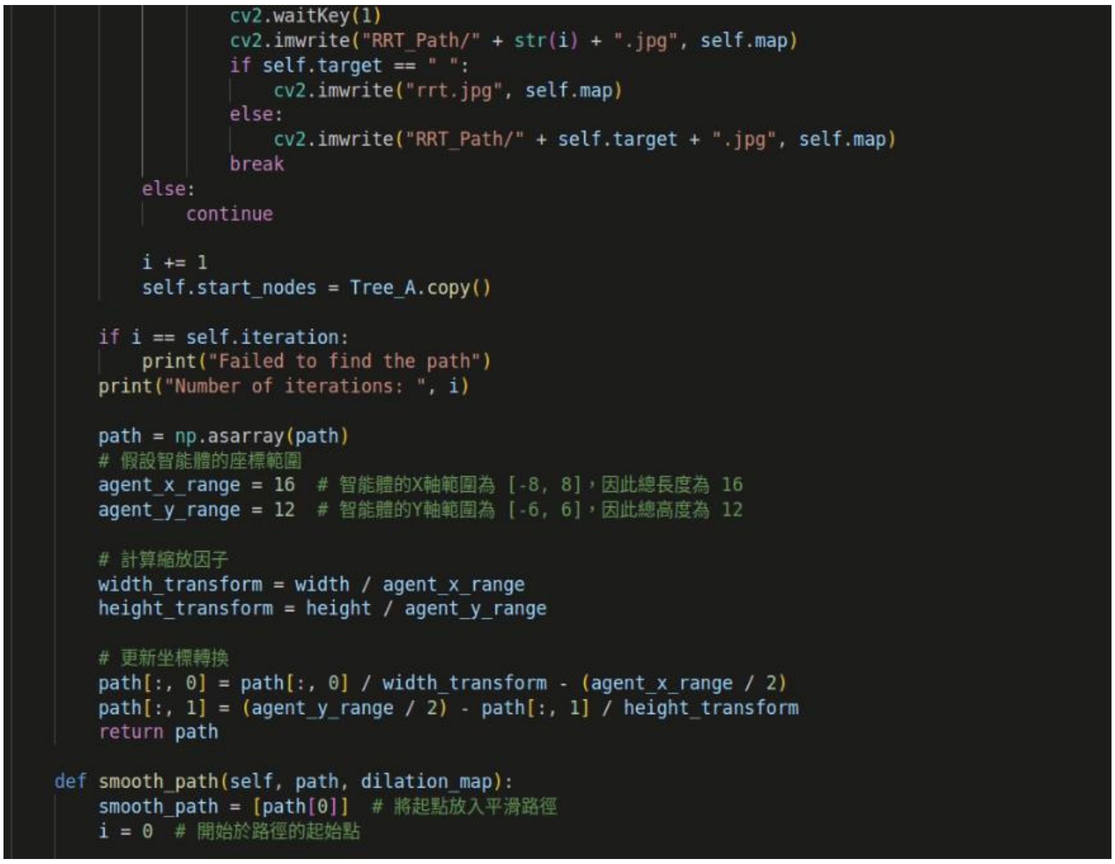
```python
    def collision(self, x1, y1, x2, y2, img):
        color = []
        if int(x1) == int(x2) and int(y1) == int(y2):
            return False

        line_points = np.column_stack((np.linspace(x1, x2, num=100), np.linspace(y1, y2, num=100)))

        for point in line_points:
            x, y = map(int, point)
            color.append(img[y, x])

        return 0 in color  # 如果有障礙物，返回 True

    def expansion(self, random_x, random_y, nearest_x, nearest_y, offset, map):
        theta = self.angle(nearest_x, nearest_y, random_x, random_y)
        new_node_x = nearest_x + offset * np.cos(theta)
        new_node_y = nearest_y + offset * np.sin(theta)
        width, height = map.shape

        if new_node_y < 0 or new_node_y > width or new_node_x < 0 or new_node_x > height:
            expand_connect = False
        else:
            expand_connect = not self.collision(new_node_x, new_node_y, nearest_x, nearest_y, map)
        return (new_node_x, new_node_y, expand_connect)
```



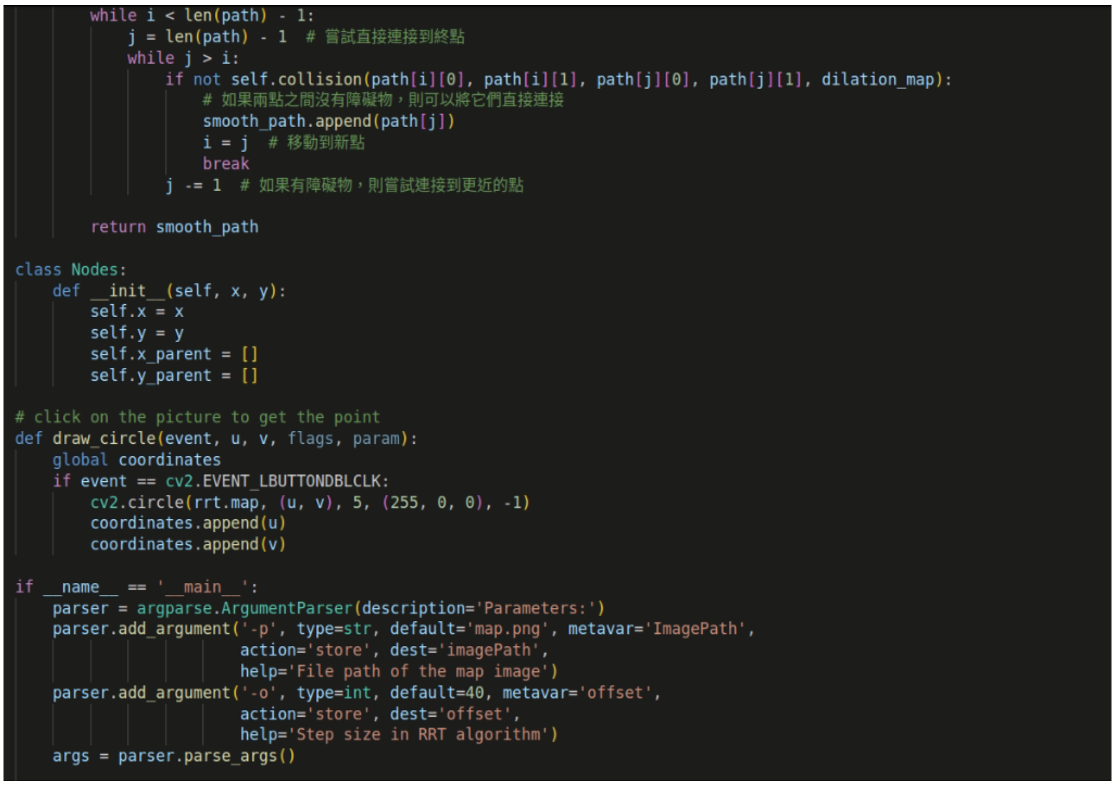
```python
    def extend(self, Tree, goal_node, map):
        x = Tree[-1].x
        y = Tree[-1].y
        nearest_index_b = self.nearest_node(x, y, goal_node)
        nearest_x = goal_node[nearest_index_b].x
        nearest_y = goal_node[nearest_index_b].y

        return (not self.collision(x, y, nearest_x, nearest_y, map), nearest_index_b)

    def rrt_path(self, target):
        if target != " ":
            self.set_target(target)

        height, width = self.dilation.shape

        self.start_nodes[0] = Nodes(self.start[0], self.start[1])
        self.start_nodes[0].x_parent.append(self.start[0])
        self.start_nodes[0].y_parent.append(self.start[1])

        i = 1
        while i < self.iteration:
            Tree_A = self.start_nodes.copy()
            random_x, random_y = self.random_point(height, width)

            nearest_index = self.nearest_node(random_x, random_y, Tree_A)
            nearest_x, nearest_y = Tree_A[nearest_index].x, Tree_A[nearest_index].y
            new_node_x, new_node_y, expand_connect = self.expansion(random_x, random_y, nearest_x, nearest_y, self.offset, self.dilation)

            if expand_connect:
                Tree_A.append(Nodes(new_node_x, new_node_y))
                Tree_A[i].x_parent = Tree_A[nearest_index].x_parent.copy()
                Tree_A[i].y_parent = Tree_A[nearest_index].y_parent.copy()
                Tree_A[i].x_parent.append(new_node_x)
                Tree_A[i].y_parent.append(new_node_y)
```



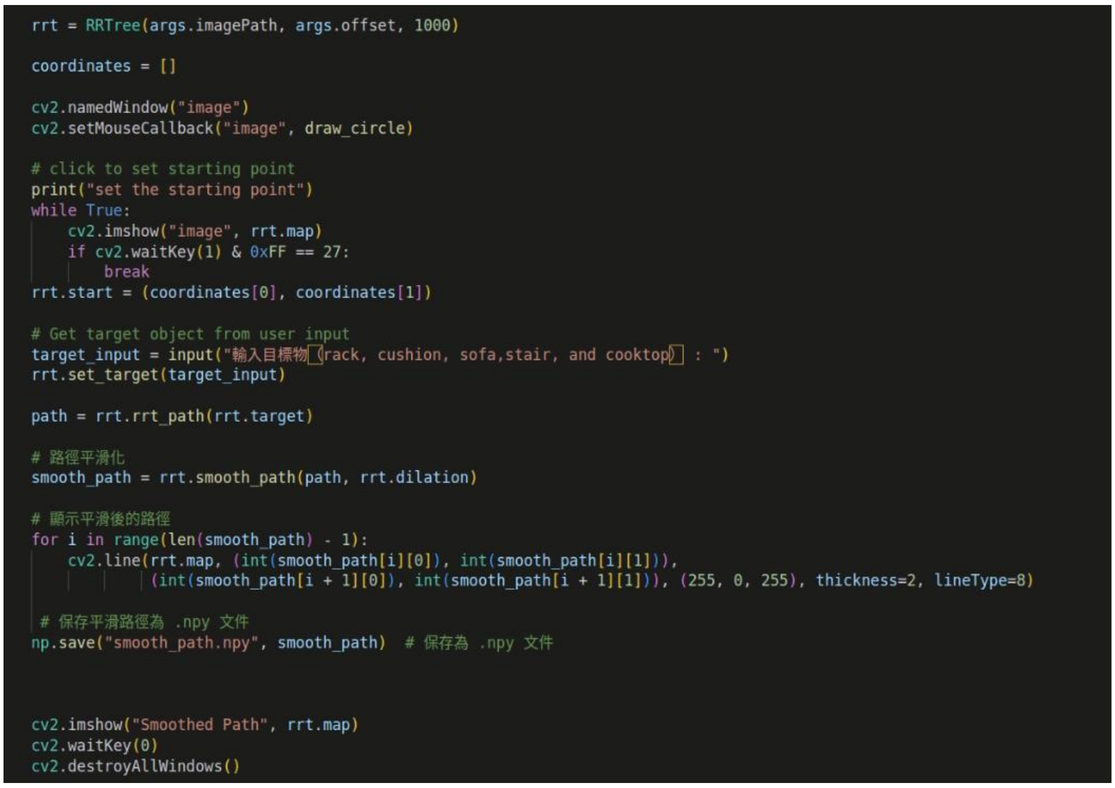
```python
                cv2.circle(self.map, (int(new_node_x), int(new_node_y)), 2, (0, 0, 255), thickness=3, lineType=8)
                cv2.line(self.map, (int(new_node_x), int(new_node_y)),
                         (int(Tree_A[nearest_index].x), int(Tree_A[nearest_index].y)),
                         (0, 255, 0), thickness=1, lineType=8)
                cv2.imwrite("RRT_Path/" + str(i) + ".jpg", self.map)
                cv2.imshow("image", self.map)
                cv2.waitKey(1)

                extend_connect, index = self.extend(Tree_A, [Nodes(self.goal[0], self.goal[1])], self.dilation)

                if extend_connect:
                    print("finish")
                    path = []
                    cv2.line(self.map, (int(new_node_x), int(new_node_y)),
                             (int(self.goal[0]), int(self.goal[1])), (0, 255, 0), thickness=1, lineType=8)

                    for i in range(len(Tree_A[-1].x_parent)):
                        path.append((Tree_A[-1].x_parent[i], Tree_A[-1].y_parent[i]))

                    Nodes(self.goal[0], self.goal[1]).x_parent.reverse()
                    Nodes(self.goal[0], self.goal[1]).y_parent.reverse()

                    for i in range(len(Nodes(self.goal[0], self.goal[1]).x_parent)):
                        path.append((Nodes(self.goal[0], self.goal[1]).x_parent[i],
                                     Nodes(self.goal[0], self.goal[1]).y_parent[i]))

                    cv2.line(self.map, (int(Tree_A[-1].x_parent[-1]), int(Tree_A[-1].y_parent[-1])),
                             (int(self.goal[0]), int(self.goal[1])), (255, 0, 0), thickness=2, lineType=8)

                    for i in range(len(path) - 1):
                        cv2.line(self.map, (int(path[i][0]), int(path[i][1])),
                                 (int(path[i + 1][0]), int(path[i + 1][1])), (255, 0, 0), thickness=2, lineType=8)
```



```python
                    cv2.waitKey(1)
                    cv2.imwrite("RRT_Path/" + str(i) + ".jpg", self.map)
                    if self.target == " ":
                        cv2.imwrite("rrt.jpg", self.map)
                    else:
                        cv2.imwrite("RRT_Path/" + self.target + ".jpg", self.map)
                    break
            else:
                continue

            i += 1
            self.start_nodes = Tree_A.copy()

        if i == self.iteration:
            print("Failed to find the path")
        print("Number of iterations: ", i)

        path = np.asarray(path)
        # 假設智能體的座標範圍
        agent_x_range = 16  # 智能體的X軸範圍為 [-8, 8]，因此總長度為 16
        agent_y_range = 12  # 智能體的Y軸範圍為 [-6, 6]，因此總高度為 12

        # 計算縮放因子
        width_transform = width / agent_x_range
        height_transform = height / agent_y_range

        # 更新坐標轉換
        path[:, 0] = path[:, 0] / width_transform - (agent_x_range / 2)
        path[:, 1] = (agent_y_range / 2) - path[:, 1] / height_transform
        return path

    def smooth_path(self, path, dilation_map):
        smooth_path = [path[0]]  # 將起點放入平滑路徑
        i = 0  # 開始於路徑的起始點
```



```python
        while i < len(path) - 1:
            j = len(path) - 1  # 嘗試直接連接到終點
            while j > i:
                if not self.collision(path[i][0], path[i][1], path[j][0], path[j][1], dilation_map):
                    # 如果兩點之間沒有障礙物，則可以將它們直接連接
                    smooth_path.append(path[j])
                    i = j  # 移動到新點
                    break
                j -= 1  # 如果有障礙物，則嘗試連接到更近的點

        return smooth_path

class Nodes:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.x_parent = []
        self.y_parent = []

# click on the picture to get the point
def draw_circle(event, u, v, flags, param):
    global coordinates
    if event == cv2.EVENT_LBUTTONDBLCLK:
        cv2.circle(rrt.map, (u, v), 5, (255, 0, 0), -1)
        coordinates.append(u)
        coordinates.append(v)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Parameters:')
    parser.add_argument('-p', type=str, default='map.png', metavar='ImagePath',
                        action='store', dest='imagePath',
                        help='File path of the map image')
    parser.add_argument('-o', type=int, default=40, metavar='offset',
                        action='store', dest='offset',
                        help='Step size in RRT algorithm')
    args = parser.parse_args()
```



```python
rrt = RRTree(args.imagePath, args.offset, 1000)

coordinates = []

cv2.namedWindow("image")
cv2.setMouseCallback("image", draw_circle)

# click to set starting point
print("set the starting point")
while True:
    cv2.imshow("image", rrt.map)
    if cv2.waitKey(1) & 0xFF == 27:
        break
rrt.start = (coordinates[0], coordinates[1])

# Get target object from user input
target_input = input("輸入目標物(rack, cushion, sofa,stair, and cooktop) : ")
rrt.set_target(target_input)

path = rrt.rrt_path(rrt.target)

# 路徑平滑化
smooth_path = rrt.smooth_path(path, rrt.dilation)

# 顯示平滑後的路徑
for i in range(len(smooth_path) - 1):
    cv2.line(rrt.map, (int(smooth_path[i][0]), int(smooth_path[i][1])),
             (int(smooth_path[i + 1][0]), int(smooth_path[i + 1][1])), (255, 0, 255), thickness=2, lineType=8)

 # 保存平滑路徑為 .npy 文件
np.save("smooth_path.npy", smooth_path)  # 保存為 .npy 文件



cv2.imshow("Smoothed Path", rrt.map)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The program implements an RRT-based path planning algorithm, which includes functionalities for image processing, path planning, and path smoothing. Here's a detailed breakdown:

1. Image Processing

At the beginning of the program, the map image is read, and non-white pixel values are set to black to distinguish obstacles. Then, a dilation operation is applied to the image to expand the boundaries of obstacles, creating a more conservative dilated map. This dilated map is used during path planning to detect collisions, ensuring that the path does not closely approach obstacles.

2. RRT Algorithm

- The program defines an RRTree class to represent the RRT tree. The path planning begins at the start point, and random points are generated to gradually extend the tree toward the goal. The node expansion process is controlled by a step size, where the angle between two nodes is calculated to determine the direction of the new node. The dilated map is used to check for collisions during the expansion.

- The algorithm generates random points using the random_point function and finds the nearest tree node to the random point using the nearest_node function. If the extension process avoids obstacles, the new node is added to the tree. This process continues until a path connecting the start and goal points is found.
- Finally, the rrt_path function draws the path once it is found and saves it as an image file.

3. Path Smoothing

Once an initial path is found, the program uses the smooth_path function to smooth the path. It attempts to directly connect two points that do not have obstacles between them, thereby reducing redundant points in the path. This is done by checking if the line segment between two points intersects with any obstacles. If no obstacles are present, the two points are directly connected to optimize the path.

4. User Interaction

The user interacts with the program by double-clicking on the map to select the start point. Then, the user inputs the target object name (e.g., "rack", "cushion", etc.) to set the goal point.

This structure provides a comprehensive approach to path planning using the RRT algorithm, ensuring efficient obstacle avoidance and user-friendly interaction.

1. **How do you implement the RRT algorithm?**

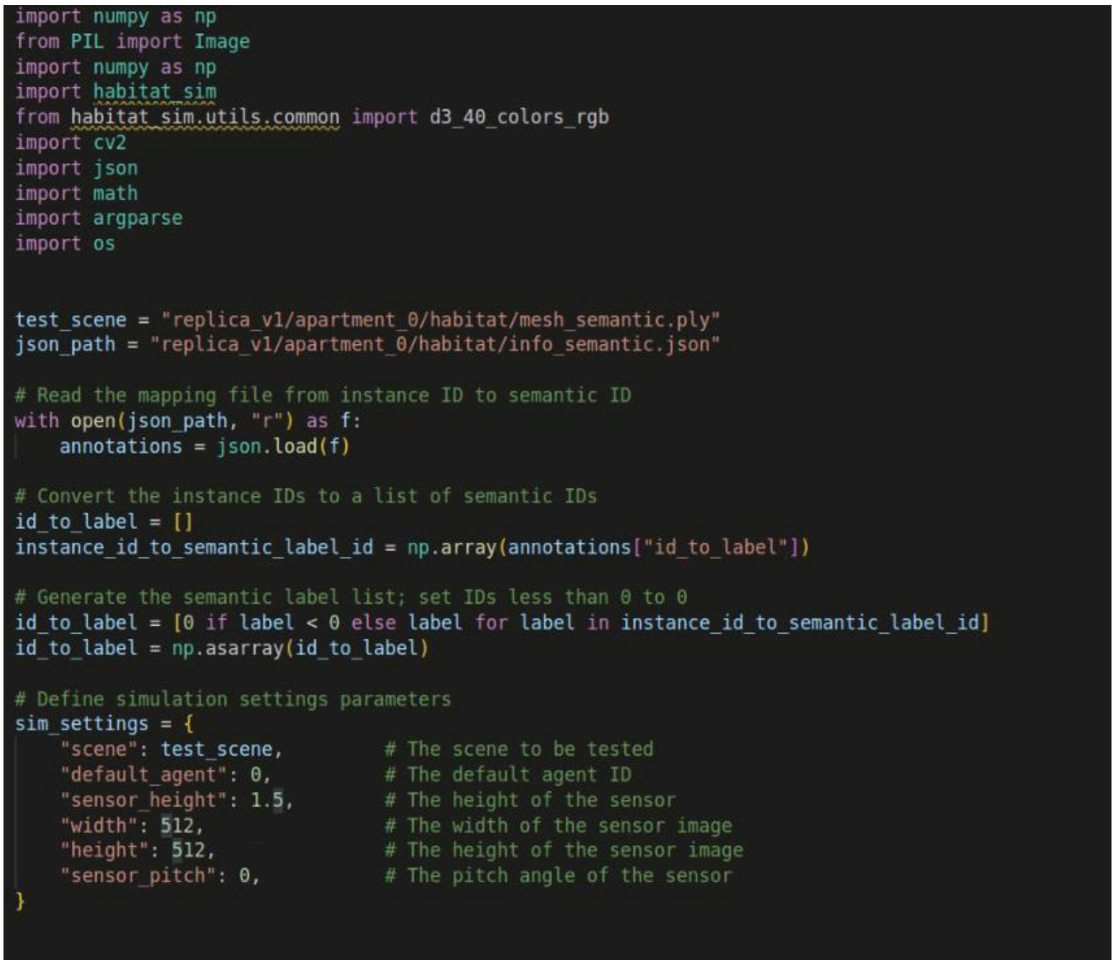
- **Map Initialization**: The program first loads the map image and generates a binary obstacle map through grayscale and dilation operations. This map is used to check for collisions during node expansion.
- **Setting the Start and Goal Points**: The start point is selected by double-clicking on the map, and the goal point is set based on the input target object name (e.g., "rack" or "sofa"). These objects have predefined coordinates.
- **Random Point Generation**: A random point is generated within the map boundaries as a potential expansion point, implemented by the random_point() function.

- **Finding the Nearest Node**: The nearest node in the existing tree to the random point is found using the nearest_node() function, which calculates the distance between the nodes in the tree and the random point.
- **Expanding a New Node**: Based on the angle between the nearest node and the random point, a new node is generated along this direction with a step size (offset), using the expansion() function. The program checks for obstacles between the nearest node and the new node. If no obstacles are detected, the expansion is successful, and the new node is added to the tree.
- **Repeating the Expansion**: The program repeatedly generates random points, expands the tree, and checks for collisions, gradually expanding the tree's coverage until a path to the goal is found.
- **Path Generation and Smoothing**: Once the tree reaches the goal, the algorithm records the path. The program then uses the smooth_path() function to smooth the path by removing unnecessary bends, making the route more efficient.

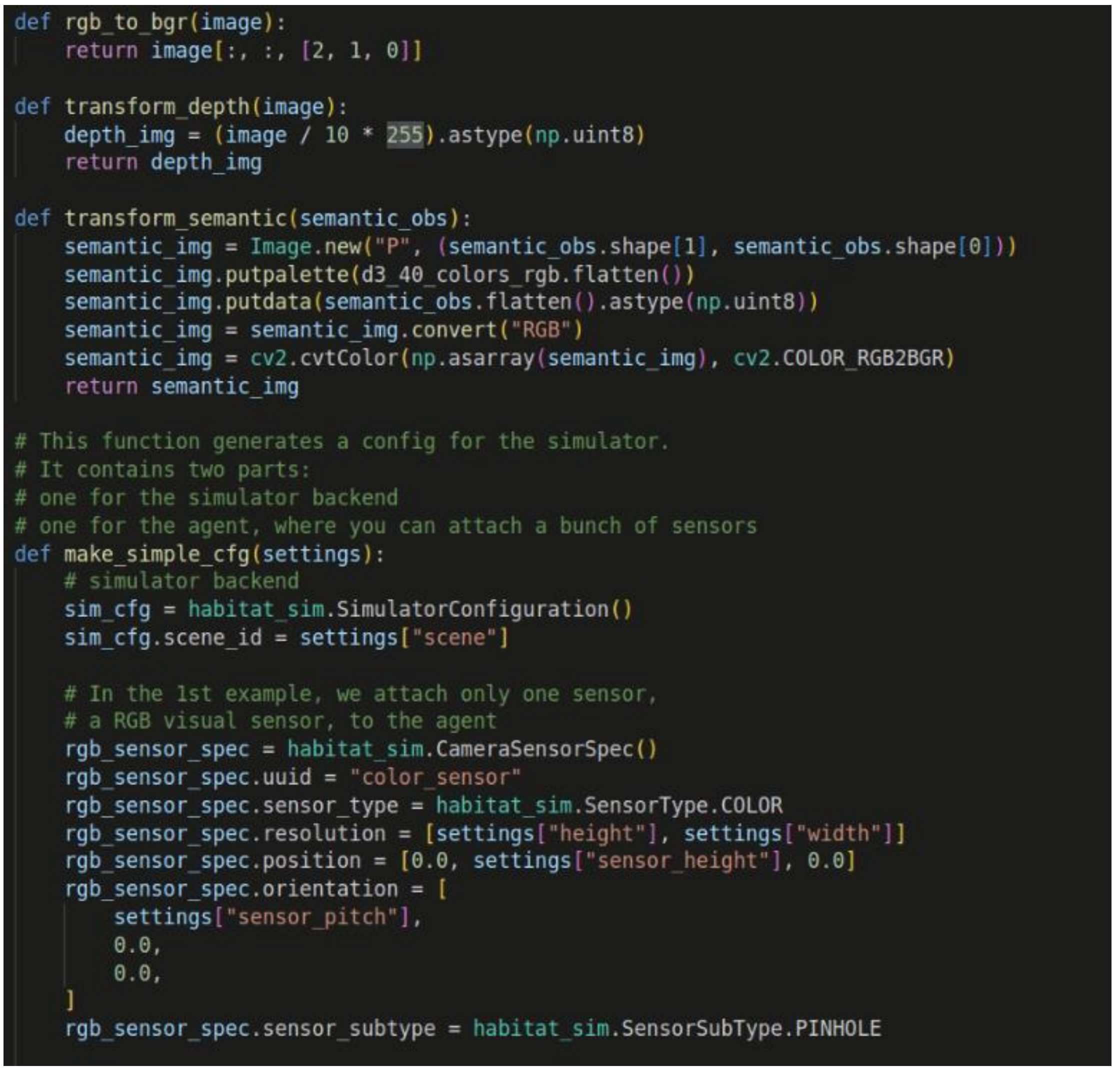
2. **How do you convert routes to discrete actions?** The path generated by RRT consists of a series of continuous points, each corresponding to a specific coordinate on the map. To convert these continuous path points into discrete actions, the following steps are performed:

- **Coordinate Transformation**: The path points are transformed from pixel coordinates into the agent's action space. This is done by scaling the map's pixel points into the actual movement range of the agent's X and Y axes, using a scaling factor defined in the program.
- **Converting to Discrete Actions**: Each pair of consecutive path points represents one action step for the agent. These steps can be interpreted as movements in a particular direction for a certain distance. For example:
  - If there is a significant change in the X-coordinate between two points, it indicates that the agent needs to move left or right.
  - If there is a significant change in the Y-coordinate, it means the agent needs to move forward or backward.
  - If both X and Y change, the agent needs to move along a certain angle.
- **Executing the Actions**: Finally, the agent performs discrete actions based on the changes between consecutive points. The program determines the step size and rotation angle based on the distance and direction between the path points, guiding the agent's navigation along the route.

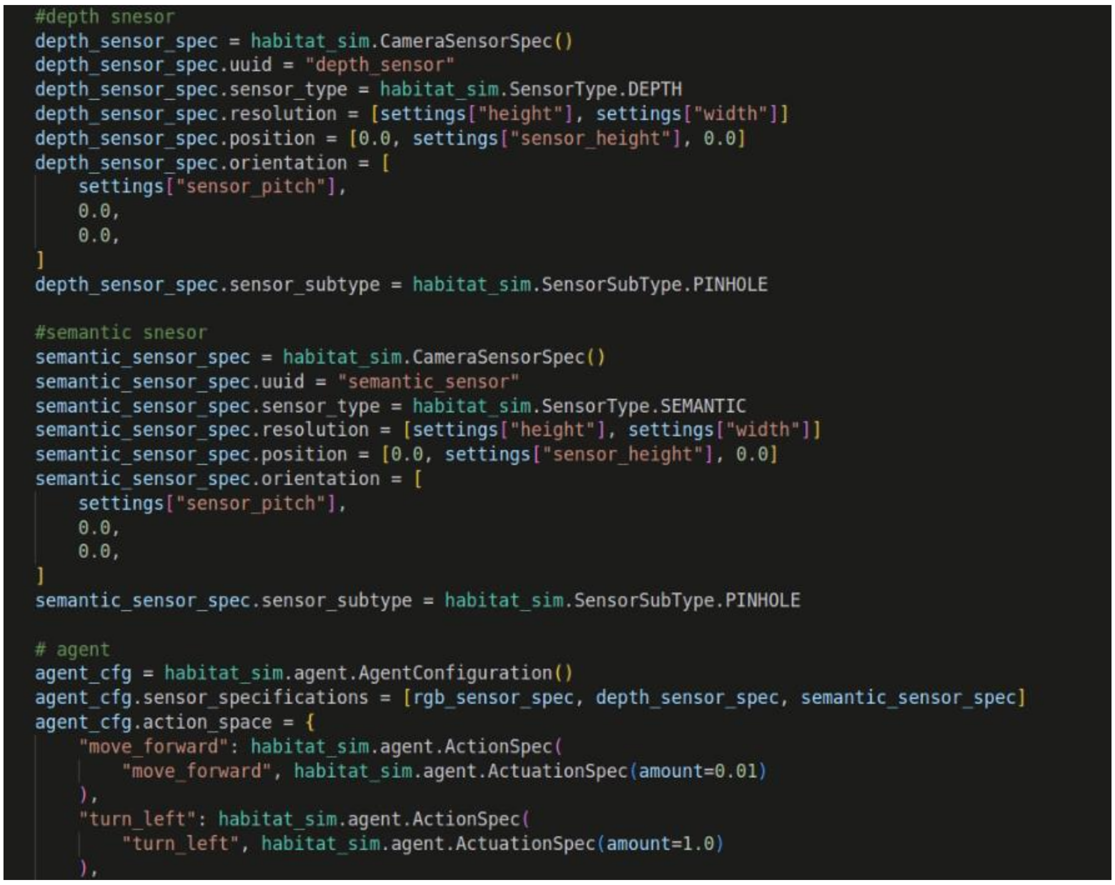
Part 3: Robot navigation



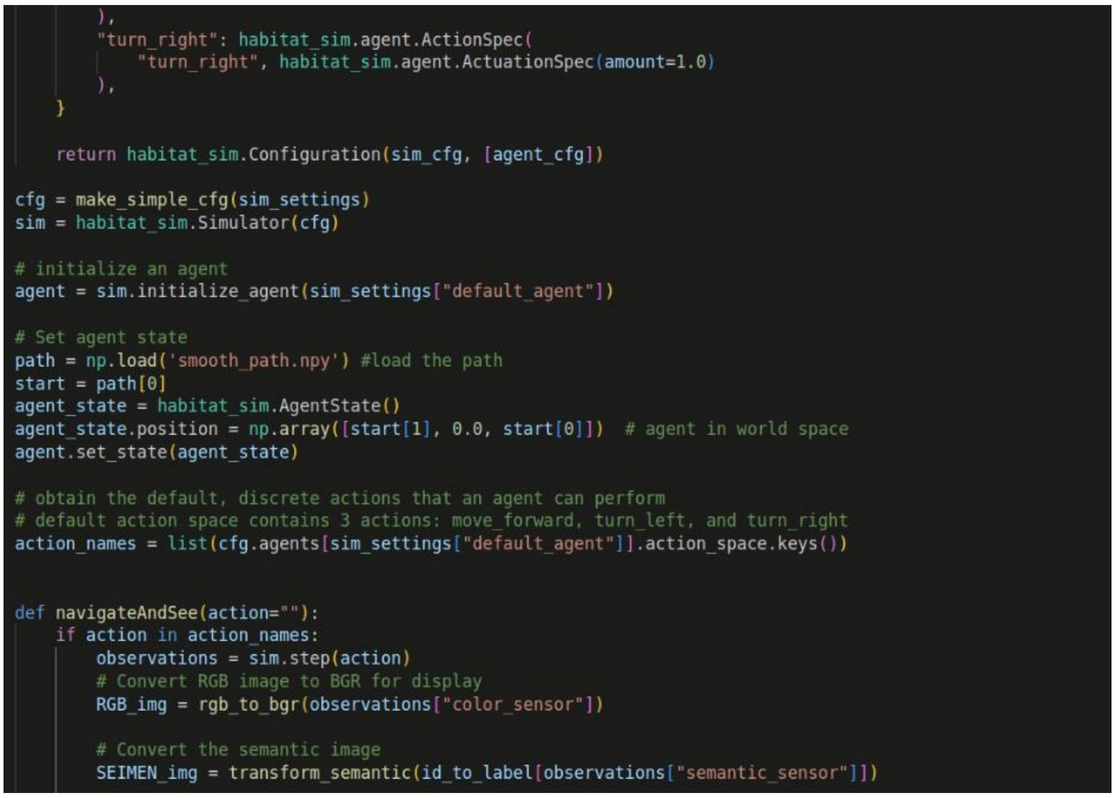
```python
import numpy as np
from PIL import Image
import numpy as np
import habitat_sim
from habitat_sim.utils.common import d3_40_colors_rgb
import cv2
import json
import math
import argparse
import os


test_scene = "replica_v1/apartment_0/habitat/mesh_semantic.ply"
json_path = "replica_v1/apartment_0/habitat/info_semantic.json"

# Read the mapping file from instance ID to semantic ID
with open(json_path, "r") as f:
    annotations = json.load(f)

# Convert the instance IDs to a list of semantic IDs
id_to_label = []
instance_id_to_semantic_label_id = np.array(annotations["id_to_label"])

# Generate the semantic label list; set IDs less than 0 to 0
id_to_label = [0 if label < 0 else label for label in instance_id_to_semantic_label_id]
id_to_label = np.asarray(id_to_label)

# Define simulation settings parameters
sim_settings = {
    "scene": test_scene,         # The scene to be tested
    "default_agent": 0,          # The default agent ID
    "sensor_height": 1.5,        # The height of the sensor
    "width": 512,                # The width of the sensor image
    "height": 512,               # The height of the sensor image
    "sensor_pitch": 0,           # The pitch angle of the sensor
}
```



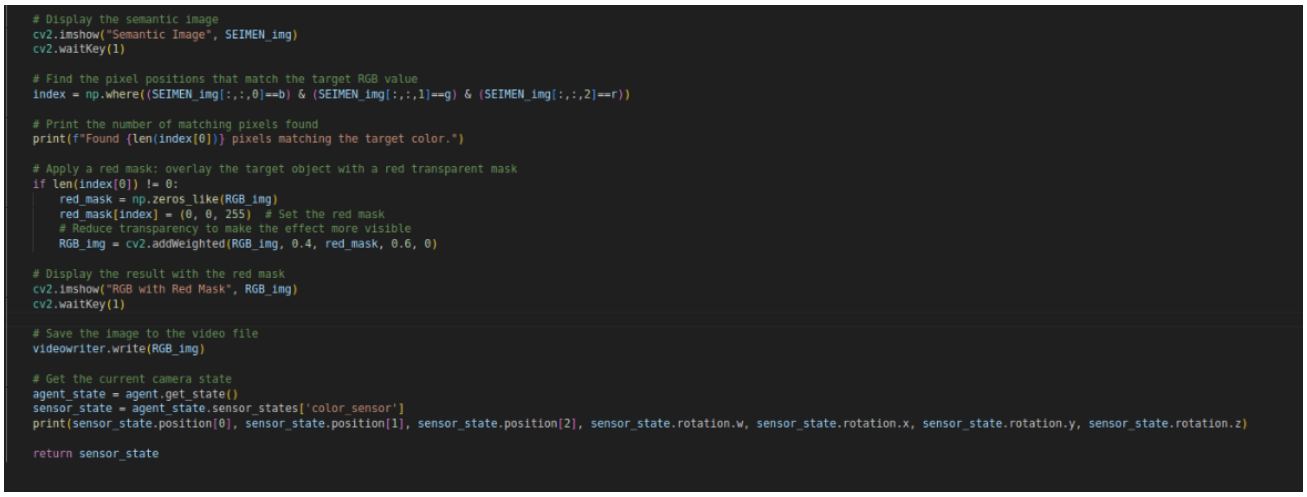
```python
def rgb_to_bgr(image):
    return image[:, :, [2, 1, 0]]

def transform_depth(image):
    depth_img = (image / 10 * 255).astype(np.uint8)
    return depth_img

def transform_semantic(semantic_obs):
    semantic_img = Image.new("P", (semantic_obs.shape[1], semantic_obs.shape[0]))
    semantic_img.putpalette(d3_40_colors_rgb.flatten())
    semantic_img.putdata(semantic_obs.flatten().astype(np.uint8))
    semantic_img = semantic_img.convert("RGB")
    semantic_img = cv2.cvtColor(np.asarray(semantic_img), cv2.COLOR_RGB2BGR)
    return semantic_img

# This function generates a config for the simulator.
# It contains two parts:
# one for the simulator backend
# one for the agent, where you can attach a bunch of sensors
def make_simple_cfg(settings):
    # simulator backend
    sim_cfg = habitat_sim.SimulatorConfiguration()
    sim_cfg.scene_id = settings["scene"]

    # In the 1st example, we attach only one sensor,
    # a RGB visual sensor, to the agent
    rgb_sensor_spec = habitat_sim.CameraSensorSpec()
    rgb_sensor_spec.uuid = "color_sensor"
    rgb_sensor_spec.sensor_type = habitat_sim.SensorType.COLOR
    rgb_sensor_spec.resolution = [settings["height"], settings["width"]]
    rgb_sensor_spec.position = [0.0, settings["sensor_height"], 0.0]
    rgb_sensor_spec.orientation = [
        settings["sensor_pitch"],
        0.0,
        0.0,
    ]
    rgb_sensor_spec.sensor_subtype = habitat_sim.SensorSubType.PINHOLE
```



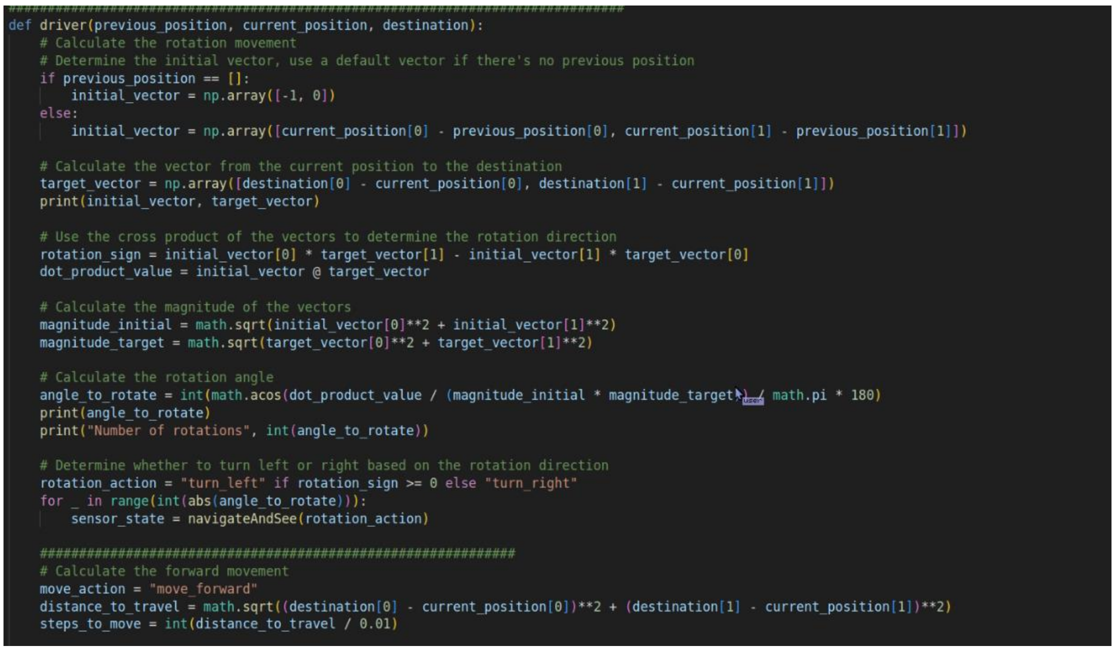
```python
    #depth snesor
    depth_sensor_spec = habitat_sim.CameraSensorSpec()
    depth_sensor_spec.uuid = "depth_sensor"
    depth_sensor_spec.sensor_type = habitat_sim.SensorType.DEPTH
    depth_sensor_spec.resolution = [settings["height"], settings["width"]]
    depth_sensor_spec.position = [0.0, settings["sensor_height"], 0.0]
    depth_sensor_spec.orientation = [
        settings["sensor_pitch"],
        0.0,
        0.0,
    ]
    depth_sensor_spec.sensor_subtype = habitat_sim.SensorSubType.PINHOLE

    #semantic snesor
    semantic_sensor_spec = habitat_sim.CameraSensorSpec()
    semantic_sensor_spec.uuid = "semantic_sensor"
    semantic_sensor_spec.sensor_type = habitat_sim.SensorType.SEMANTIC
    semantic_sensor_spec.resolution = [settings["height"], settings["width"]]
    semantic_sensor_spec.position = [0.0, settings["sensor_height"], 0.0]
    semantic_sensor_spec.orientation = [
        settings["sensor_pitch"],
        0.0,
        0.0,
    ]
    semantic_sensor_spec.sensor_subtype = habitat_sim.SensorSubType.PINHOLE

    # agent
    agent_cfg = habitat_sim.agent.AgentConfiguration()
    agent_cfg.sensor_specifications = [rgb_sensor_spec, depth_sensor_spec, semantic_sensor_spec]
    agent_cfg.action_space = {
        "move_forward": habitat_sim.agent.ActionSpec(
            "move_forward", habitat_sim.agent.ActuationSpec(amount=0.01)
        ),
        "turn_left": habitat_sim.agent.ActionSpec(
            "turn_left", habitat_sim.agent.ActuationSpec(amount=1.0)
        ),
```



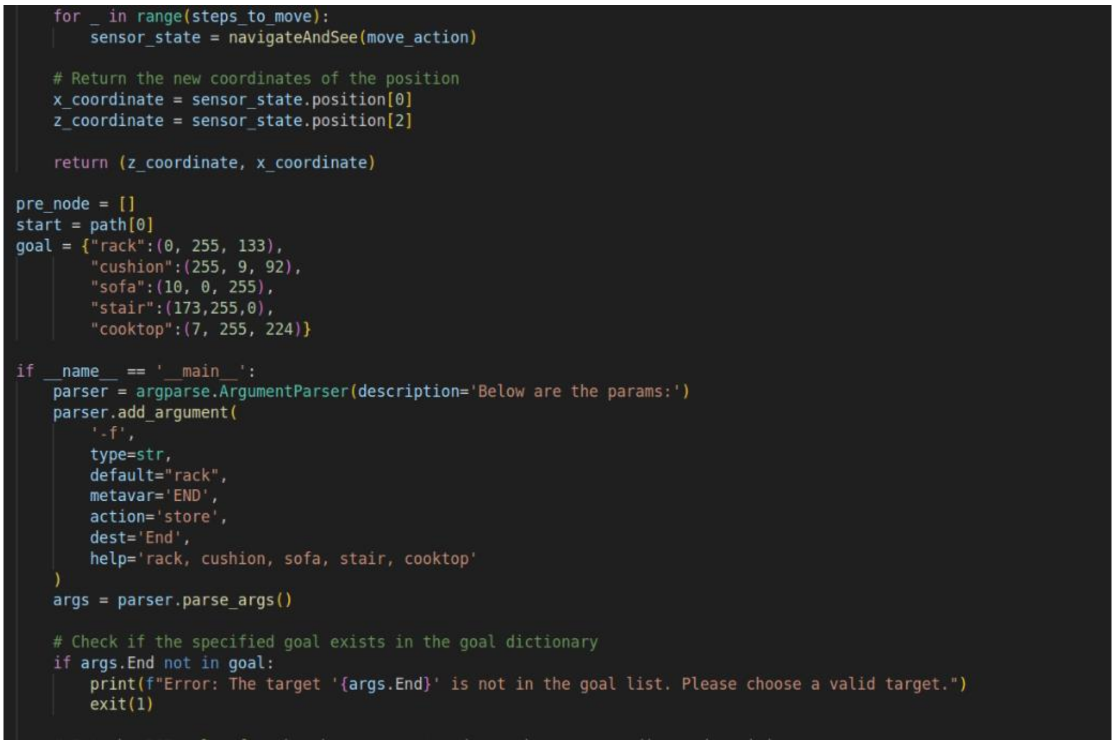
```python
        ),
        "turn_right": habitat_sim.agent.ActionSpec(
            "turn_right", habitat_sim.agent.ActuationSpec(amount=1.0)
        ),
    }

    return habitat_sim.Configuration(sim_cfg, [agent_cfg])

cfg = make_simple_cfg(sim_settings)
sim = habitat_sim.Simulator(cfg)

# initialize an agent
agent = sim.initialize_agent(sim_settings["default_agent"])

# Set agent state
path = np.load('smooth_path.npy') #load the path
start = path[0]
agent_state = habitat_sim.AgentState()
agent_state.position = np.array([start[1], 0.0, start[0]])  # agent 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())


def navigateAndSee(action=""):
    if action in action_names:
        observations = sim.step(action)
        # Convert RGB image to BGR for display
        RGB_img = rgb_to_bgr(observations["color_sensor"])

        # Convert the semantic image
        SEIMEN_img = transform_semantic(id_to_label[observations["semantic_sensor"]])
```



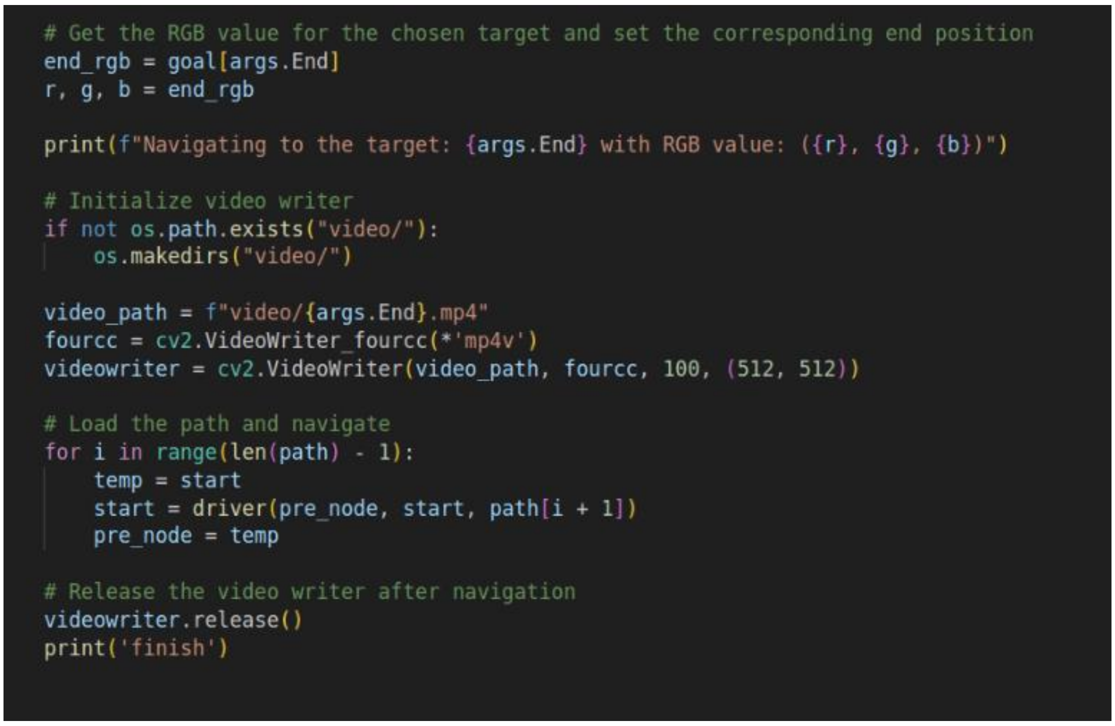
```python
    # Display the semantic image
    cv2.imshow("Semantic Image", SEIMEN_img)
    cv2.waitKey(1)

    # Find the pixel positions that match the target RGB value
    index = np.where((SEIMEN_img[:,:,0]==b) & (SEIMEN_img[:,:,1]==g) & (SEIMEN_img[:,:,2]==r))

    # Print the number of matching pixels found
    print(f"Found {len(index[0])} pixels matching the target color.")

    # Apply a red mask: overlay the target object with a red transparent mask
    if len(index[0]) != 0:
        red_mask = np.zeros_like(RGB_img)
        red_mask[index] = (0, 0, 255)  # Set the red mask
        # Reduce transparency to make the effect more visible
        RGB_img = cv2.addWeighted(RGB_img, 0.4, red_mask, 0.6, 0)

    # Display the result with the red mask
    cv2.imshow("RGB with Red Mask", RGB_img)
    cv2.waitKey(1)

    # Save the image to the video file
    videowriter.write(RGB_img)

    # Get the current camera state
    agent_state = agent.get_state()
    sensor_state = agent_state.sensor_states['color_sensor']
    print(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)

    return sensor_state
```



```python
##############################################################################
def driver(previous_position, current_position, destination):
    # Calculate the rotation movement
    # Determine the initial vector, use a default vector if there's no previous position
    if previous_position == []:
        initial_vector = np.array([-1, 0])
    else:
        initial_vector = np.array([current_position[0] - previous_position[0], current_position[1] - previous_position[1]])

    # Calculate the vector from the current position to the destination
    target_vector = np.array([destination[0] - current_position[0], destination[1] - current_position[1]])
    print(initial_vector, target_vector)

    # Use the cross product of the vectors to determine the rotation direction
    rotation_sign = initial_vector[0] * target_vector[1] - initial_vector[1] * target_vector[0]
    dot_product_value = initial_vector @ target_vector

    # Calculate the magnitude of the vectors
    magnitude_initial = math.sqrt(initial_vector[0]**2 + initial_vector[1]**2)
    magnitude_target = math.sqrt(target_vector[0]**2 + target_vector[1]**2)

    # Calculate the rotation angle
    angle_to_rotate = int(math.acos(dot_product_value / (magnitude_initial * magnitude_target)) / math.pi * 180)
    print(angle_to_rotate)
    print("Number of rotations", int(angle_to_rotate))

    # Determine whether to turn left or right based on the rotation direction
    rotation_action = "turn_left" if rotation_sign >= 0 else "turn_right"
    for _ in range(int(abs(angle_to_rotate))):
        sensor_state = navigateAndSee(rotation_action)

    ##############################################################
    # Calculate the forward movement
    move_action = "move_forward"
    distance_to_travel = math.sqrt((destination[0] - current_position[0])**2 + (destination[1] - current_position[1])**2)
    steps_to_move = int(distance_to_travel / 0.01)
```



```python
    for _ in range(steps_to_move):
        sensor_state = navigateAndSee(move_action)

    # Return the new coordinates of the position
    x_coordinate = sensor_state.position[0]
    z_coordinate = sensor_state.position[2]

    return (z_coordinate, x_coordinate)

pre_node = []
start = path[0]
goal = {"rack":(0, 255, 133),
        "cushion":(255, 9, 92),
        "sofa":(10, 0, 255),
        "stair":(173,255,0),
        "cooktop":(7, 255, 224)}

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Below are the params:')
    parser.add_argument(
        '-f',
        type=str,
        default="rack",
        metavar='END',
        action='store',
        dest='End',
        help='rack, cushion, sofa, stair, cooktop'
    )
    args = parser.parse_args()

    # Check if the specified goal exists in the goal dictionary
    if args.End not in goal:
        print(f"Error: The target '{args.End}' is not in the goal list. Please choose a valid target.")
        exit(1)
```



```python
    # Get the RGB value for the chosen target and set the corresponding end position
    end_rgb = goal[args.End]
    r, g, b = end_rgb

    print(f"Navigating to the target: {args.End} with RGB value: ({r}, {g}, {b})")

    # Initialize video writer
    if not os.path.exists("video/"):
        os.makedirs("video/")

    video_path = f"video/{args.End}.mp4"
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    videowriter = cv2.VideoWriter(video_path, fourcc, 100, (512, 512))

    # Load the path and navigate
    for i in range(len(path) - 1):
        temp = start
        start = driver(pre_node, start, path[i + 1])
        pre_node = temp

    # Release the video writer after navigation
    videowriter.release()
    print('finish')
```

The following content describes how the program navigates and moves a robot using the Habitat simulation platform. The main steps are as follows:

1. Scene Setup

The program loads a 3D scene and reads semantic labels of the objects within the environment through the info_semantic.json file, helping the robot identify different objects in the scene.

2. Sensor Configuration

The robot is equipped with three sensors: an RGB visual sensor, a depth sensor, and a semantic sensor. These sensors provide image data, depth information, and semantic classification of the environment.

3. Path Planning

The program loads a predefined navigation path (stored in the smooth_path.npy file), and the robot's movement is guided by the points in this path.

4. Robot Movement

The robot can execute three discrete actions: "move_forward", "turn_left", and "turn_right". During the path planning process, the robot calculates the required rotation angle and forward distance based on its current position relative to the target point, and gradually approaches the target through a series of turns and moves.

5. Visual Output

After each action, the robot captures the environment's image using its RGB sensor and displays the target object in red via the semantic sensor. This helps the user visualize the relative position of the target object.

6. Video Recording

As the robot navigates, its visual output from each step is saved into a video file, recording the entire navigation process.

The navigation and movement part of the program can be described in detail through the following key steps:

1. Loading the Path and Initializing the Starting Position

The program first loads the predefined navigation path by calling np.load('smooth_path.npy'). This path consists of a series of 2D coordinates that represent the robot's trajectory within the scene.

- path[0] serves as the initial position of the robot.
- The robot's state, including its position and rotation, is set using habitat_sim.AgentState().
- The starting position of the robot is set with agent_state.position = np.array([start[1], 0.0, start[0]]). Note that the path's (x, y) coordinates are converted to (y, 0, x) because Habitat uses a 3D coordinate system, where the second dimension represents height.

2. Navigation Function: navigateAndSee(action="")

This function is the core navigation logic of the robot, where the robot executes actions within the scene by calling sim.step(action) in Habitat.

- **Input Actions**: The action parameter represents the action that the robot is currently executing, primarily including:
  - "move_forward": The robot moves forward by 0.01 meters.
  - "turn_left": The robot turns left by 1 degree.
  - "turn_right": The robot turns right by 1 degree.
- **Perception Information**: After each action, the robot uses its three sensors (RGB, depth, and semantic) to capture the current state of the environment. The observations object contains the data from these sensors:
  - **RGB Image**: The robot obtains RGB images via observations["color_sensor"], and the image is converted to BGR format for display in OpenCV.
  - **Semantic Image**: The semantic information from observations["semantic_sensor"] provides the semantic segmentation of the scene. The program uses a predefined color palette to visualize the semantic image, which is combined with the RGB image.
- **Target Object Detection**: The program detects the target object based on the user-specified RGB color values (e.g., the rack has an RGB value of (0, 255, 133)). The program finds all pixels matching this color in the current image using the np.where() function and overlays a red mask on the target object in the image.

3. Movement Control: driver(previous_position, current_position, destination)

This function handles the robot's specific movement logic, which is divided into **rotation** and **forward movement** steps:

Rotation Part:

- **Calculating Rotation Direction and Angle**: The robot calculates the relative direction between its current position and the target point to determine whether to turn left or right.
    - initial_vector: This represents the robot's current direction vector. If no previous position is given, it defaults to [-1, 0], meaning the robot initially faces the negative x direction.
    - target_vector: This represents the direction vector from the robot's current position to the target position.
    - **Cross Product and Dot Product**: The program uses the cross product of initial_vector and target_vector to determine the direction of rotation. The sign of the cross product indicates whether the robot should turn left or right. The angle of rotation is calculated using the dot product between the two vectors.
    - **Executing Rotation**: Based on the calculated angle of rotation, the robot decides whether to turn left or right and performs multiple rotation steps until it is facing the target direction.

Forward Movement Part:

- **Calculating the Distance to Travel**: The program computes the straight-line distance between the robot's current position and the target point to determine how many steps the robot needs to move forward. Each step moves the robot 0.01 meters forward.

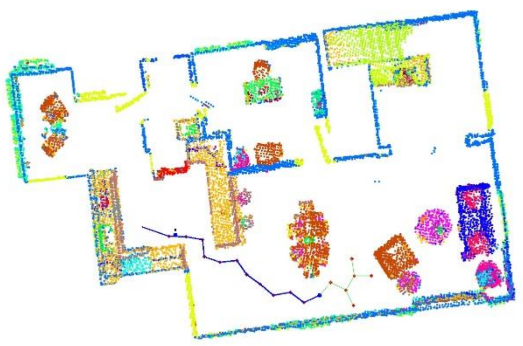
- **Executing Forward Movement**: The robot moves step by step towards the target point by calling navigateAndSee("move_forward") to update its position and display the current RGB image.

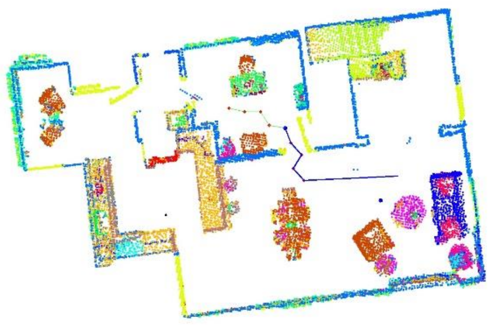
4. Overall Navigation Flow

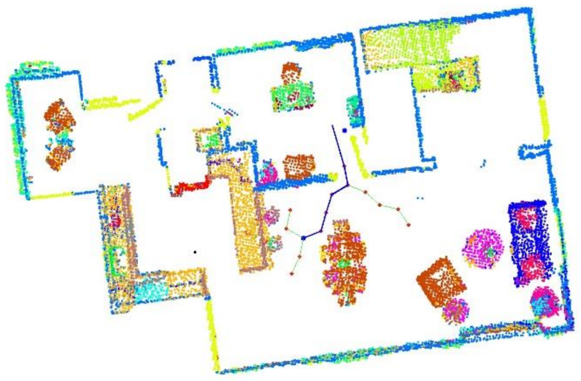
- The driver() function handles navigation between each point along the path, including both rotation and forward movement.
- During each movement, the robot uses navigateAndSee() to capture real-time visual information, displaying both the RGB view and the red overlay on the target object.
- By repeatedly calling the driver() function, the robot moves from the starting point (path[0]) through the intermediate points in the path until it finally reaches the target destination.

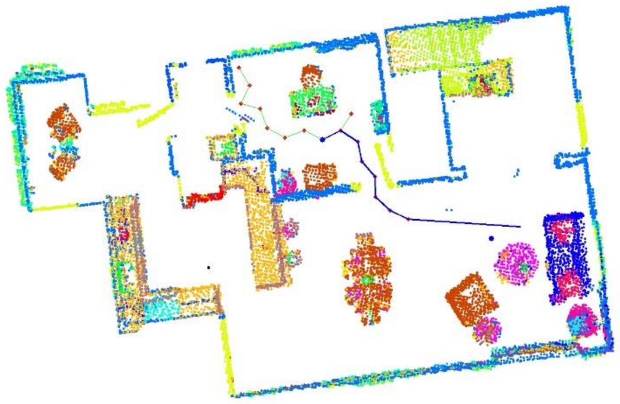
b. Result and Discussion

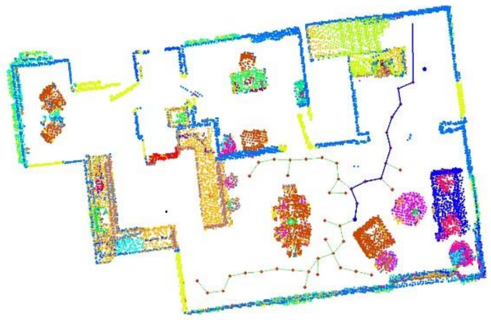
The following are the results of my RRT path planning


cooktop


cushion


rack


sofa


stair

I encountered the following issues in robot navigation:

Insufficient Path Smoothing:

- **Problem Description:** The paths generated by RRT are often discrete and contain many turns, which may lead to the robot frequently turning during its movement, resulting in low navigation efficiency. This can cause the robot to move unevenly, take longer to reach its destination, and even get stuck in narrow areas.
- **Solution:** Use path smoothing algorithms to further optimize the path, reducing unnecessary turns and making navigation more efficient and smooth.

Route Issues:

- **Problem Description:** The robot may get stuck in a locally optimal path, making it unable to find a globally optimal path, especially in complex environments with many obstacles. This can lead to the robot taking longer routes or being unable to reach the target correctly.
- **No solution has been devised yet.**

c. Reference

https://www.cs.cmu.edu/~motionplanning/lecture/lec20.pdf

2.Question

a. The Impact of Step Size and Bias on RRT Sampling Results (15%)

In the RRT (Rapidly-exploring Random Tree) algorithm, step size and bias are important parameters that influence sampling results. Here’s a breakdown of how each parameter affects the algorithm:

1. Step Size

Step size refers to the distance moved from the current node to a new node during each expansion of the tree. The setting of step size affects several aspects:

- **Path Smoothness**: A smaller step size allows the algorithm to adjust paths more precisely, resulting in smoother paths. This is particularly important in scenarios requiring fine control, such as navigating through narrow spaces.
- **Convergence Speed**: A larger step size enables the tree to cover larger areas more quickly, thus accelerating the convergence process. However, if the step size is too large, it may miss some narrow passages, preventing effective exploration of critical areas.
- **Exploration Capability**: A larger step size increases the efficiency of exploration in complex environments. However, in areas with many obstacles, a large step size may lead to frequent collisions, wasting computational resources.

2. Bias

Bias refers to the probability of preferentially sampling from the target area during the sampling process. Properly setting bias affects the balance between exploration and exploitation in the algorithm:

- **Balance between Exploration and Exploitation**: A high bias value means the algorithm will attempt to sample towards the target more frequently, which can speed up the process of finding the target. However, too high a bias might result in insufficient exploration of other areas, affecting the overall path.
- **Sampling Efficiency**: If the bias is set appropriately, the algorithm can quickly approach the target and find a suitable path. However, excessive bias may cause the algorithm to get stuck in local optima, preventing it from finding the best path.
- **Adaptability to the Environment**: In dynamic environments, an appropriate bias setting can help the algorithm effectively respond to changes, such as avoiding obstacles that appear suddenly.

In summary, both step size and bias are crucial in the RRT algorithm. Choosing the right step size can improve the smoothness and convergence speed of the path, while a suitable bias setting can enhance sampling efficiency and exploration capability. Therefore, it is essential to adjust these parameters based on specific scenarios in practical applications.

b. Potential Issues When Applying Indoor Navigation Pipelines in the Real World (5%)

When applying indoor navigation pipelines in real-world environments, several issues may arise:

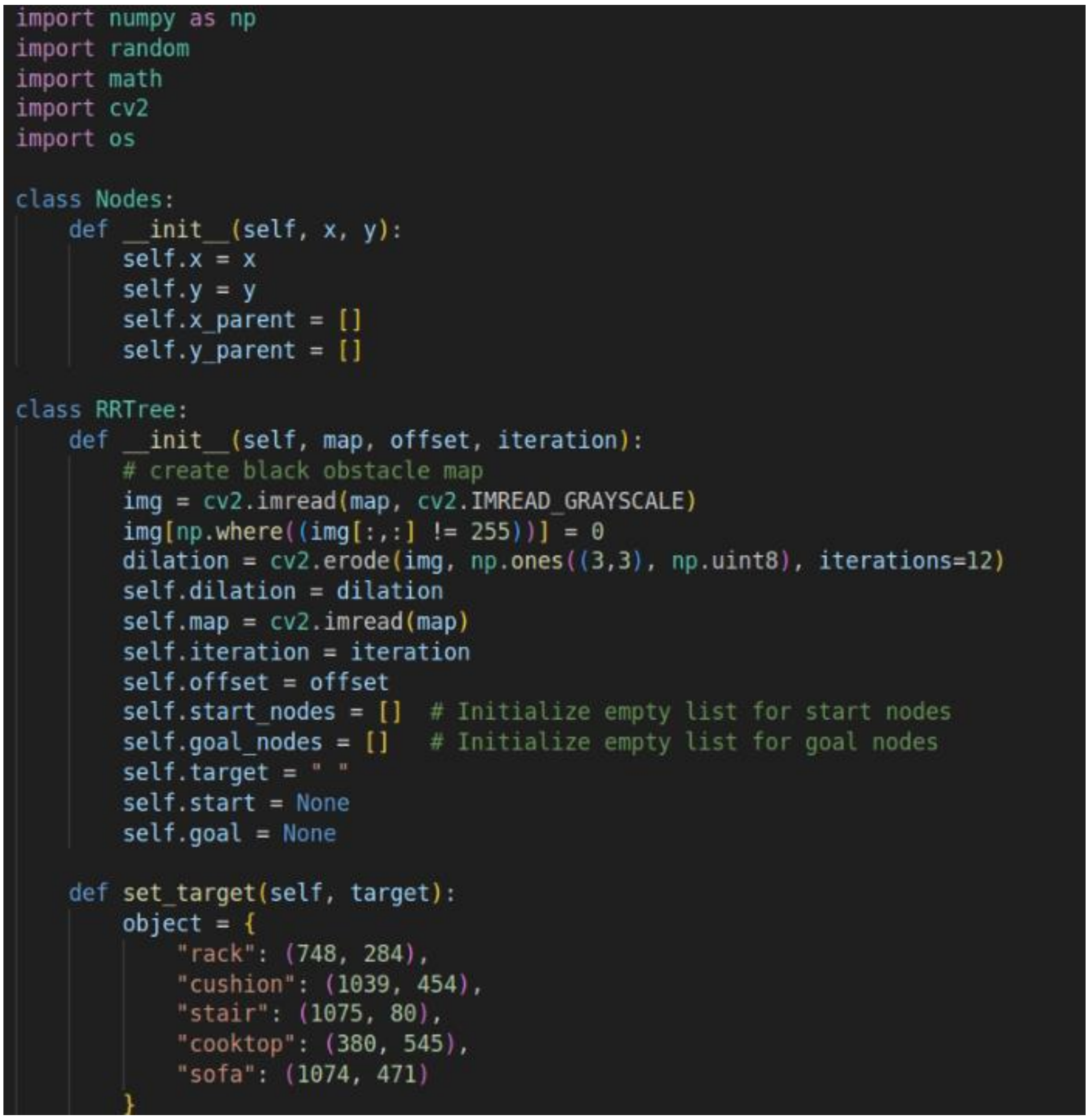
1. **Environmental Complexity**: Real-world environments often have complex and variable layouts, with various obstacles and dynamic changes (such as the movement of people or furniture). This requires the navigation algorithm to have good adaptability and real-time response capabilities.
2. **Sensor Noise**: Sensors (such as cameras or LIDAR) may be affected by noise when capturing environmental data, leading to inaccurate information. This noise can directly impact the effectiveness of path planning, causing navigation agents to incorrectly identify targets or avoid obstacles.
3. **Limited Computational Resources**: In practical applications, computational resources are often limited, especially on mobile devices. Complex navigation algorithms may require significant computation time and memory, which can affect real-time navigation performance.
4. **Dynamic Changes**: Indoor environments can change over time, such as the movement of people or rearrangement of objects. This necessitates the navigation system to update path planning in real time to respond to these changes, adding complexity to the system.
5. **Safety and Reliability**: Ensuring the safety and reliability of navigation is crucial. The algorithm must be capable of handling unexpected situations, such as sudden obstacles, to avoid collisions and other safety hazards.
6. **User Interaction**: In some applications, users need to interact with the navigation system, such as inputting destinations or adjusting settings.

Therefore, ensuring a positive user experience and timely system responses can be a challenge.

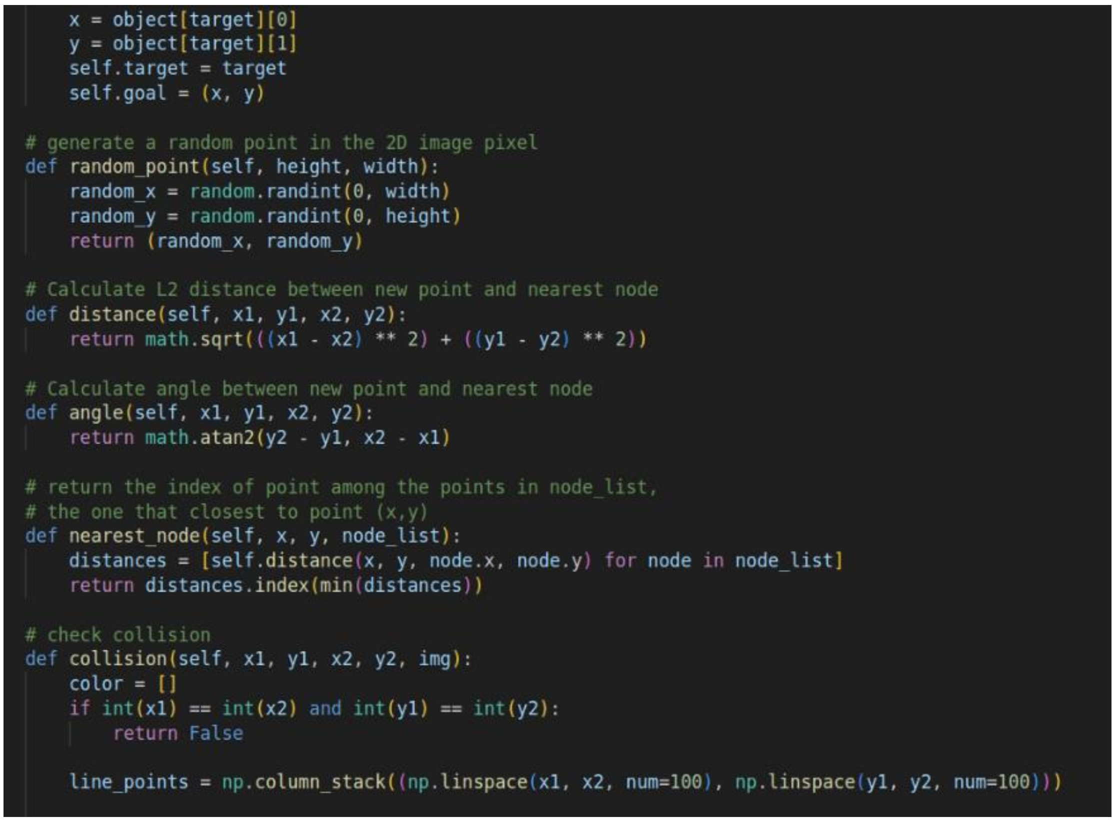
Overall, when applying indoor navigation pipelines to real-world scenarios, it’s important to consider these potential issues and make corresponding optimizations and adjustments in algorithm and system design.

Bonus 10%

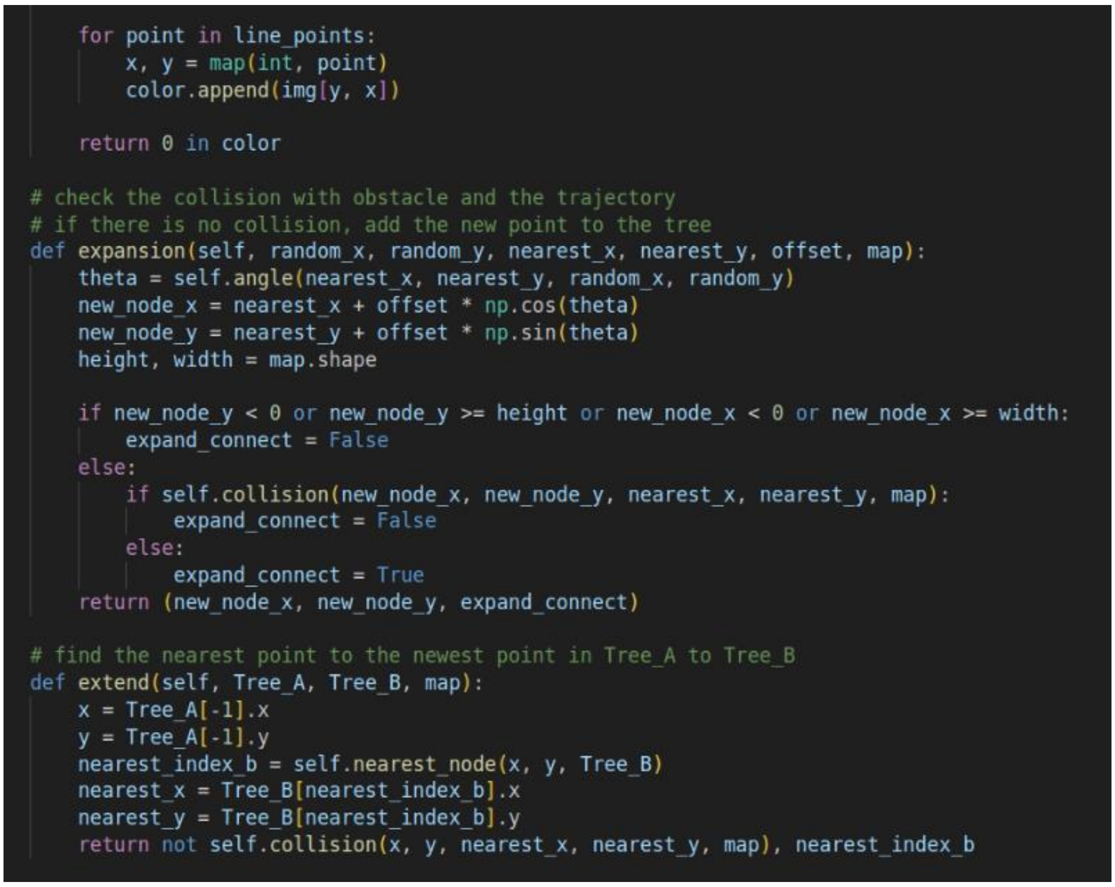
Try to improve the RRT algorithm by comparing and discussing it with the original one



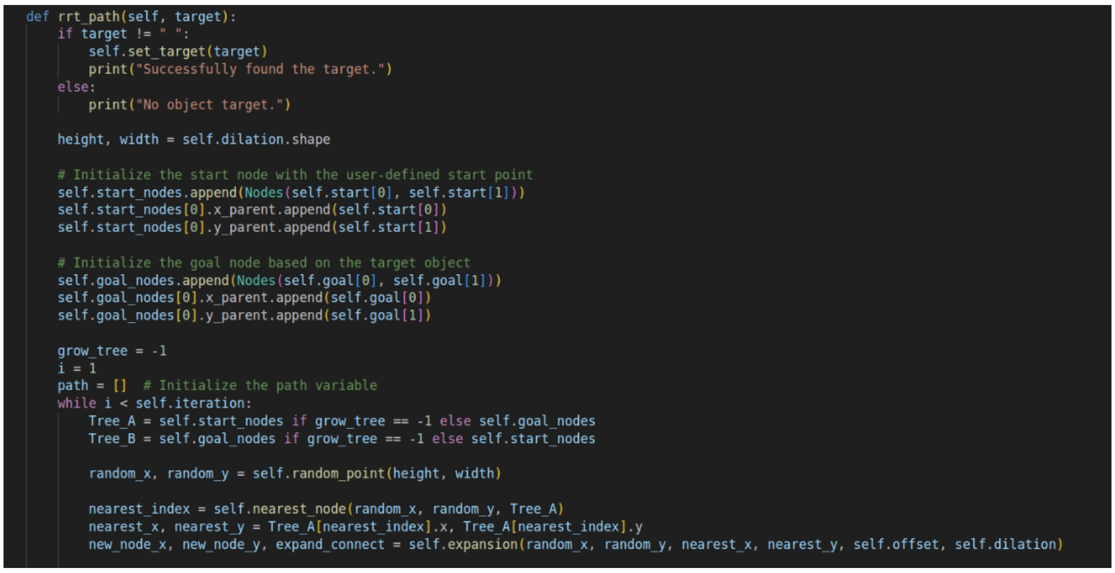
```python
import numpy as np
import random
import math
import cv2
import os

class Nodes:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.x_parent = []
        self.y_parent = []

class RRTree:
    def __init__(self, map, offset, iteration):
        # create black obstacle map
        img = cv2.imread(map, cv2.IMREAD_GRAYSCALE)
        img[np.where((img[:,:] != 255))] = 0
        dilation = cv2.erode(img, np.ones((3,3), np.uint8), iterations=12)
        self.dilation = dilation
        self.map = cv2.imread(map)
        self.iteration = iteration
        self.offset = offset
        self.start_nodes = []  # Initialize empty list for start nodes
        self.goal_nodes = []   # Initialize empty list for goal nodes
        self.target = " "
        self.start = None
        self.goal = None

    def set_target(self, target):
        object = {
            "rack": (748, 284),
            "cushion": (1039, 454),
            "stair": (1075, 80),
            "cooktop": (380, 545),
            "sofa": (1074, 471)
        }
```



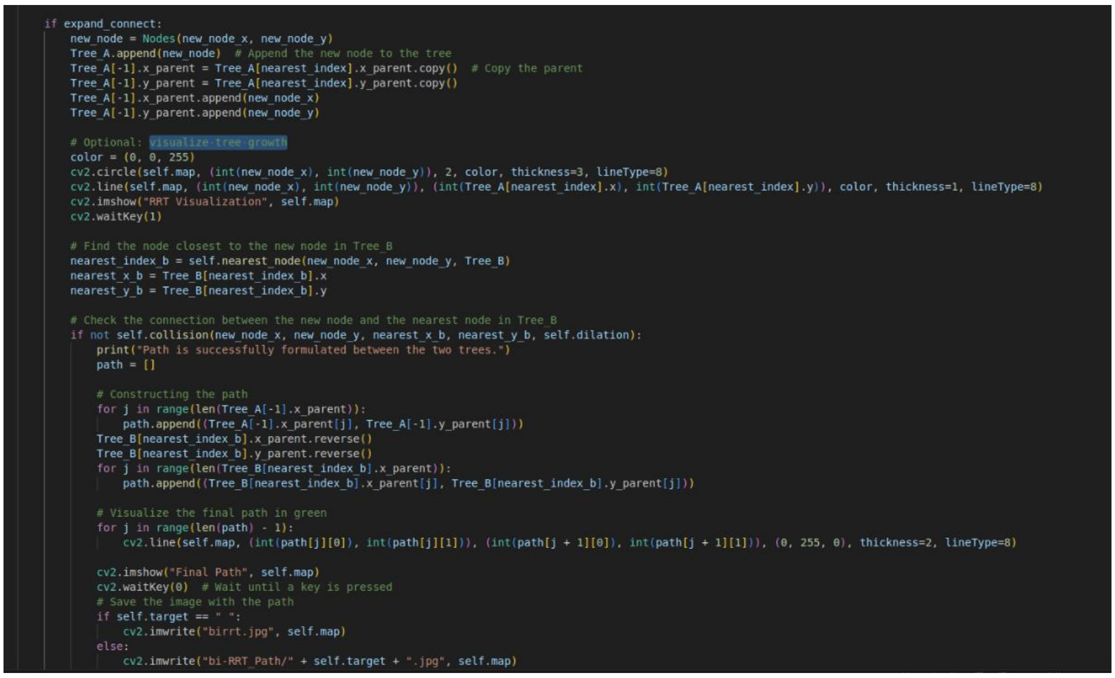
```python
        x = object[target][0]
        y = object[target][1]
        self.target = target
        self.goal = (x, y)

    # generate a random point in the 2D image pixel
    def random_point(self, height, width):
        random_x = random.randint(0, width)
        random_y = random.randint(0, height)
        return (random_x, random_y)

    # Calculate L2 distance between new point and nearest node
    def distance(self, x1, y1, x2, y2):
        return math.sqrt(((x1 - x2) ** 2) + ((y1 - y2) ** 2))

    # Calculate angle between new point and nearest node
    def angle(self, x1, y1, x2, y2):
        return math.atan2(y2 - y1, x2 - x1)

    # return the index of point among the points in node_list,
    # the one that closest to point (x,y)
    def nearest_node(self, x, y, node_list):
        distances = [self.distance(x, y, node.x, node.y) for node in node_list]
        return distances.index(min(distances))

    # check collision
    def collision(self, x1, y1, x2, y2, img):
        color = []
        if int(x1) == int(x2) and int(y1) == int(y2):
            return False

        line_points = np.column_stack((np.linspace(x1, x2, num=100), np.linspace(y1, y2, num=100)))
```



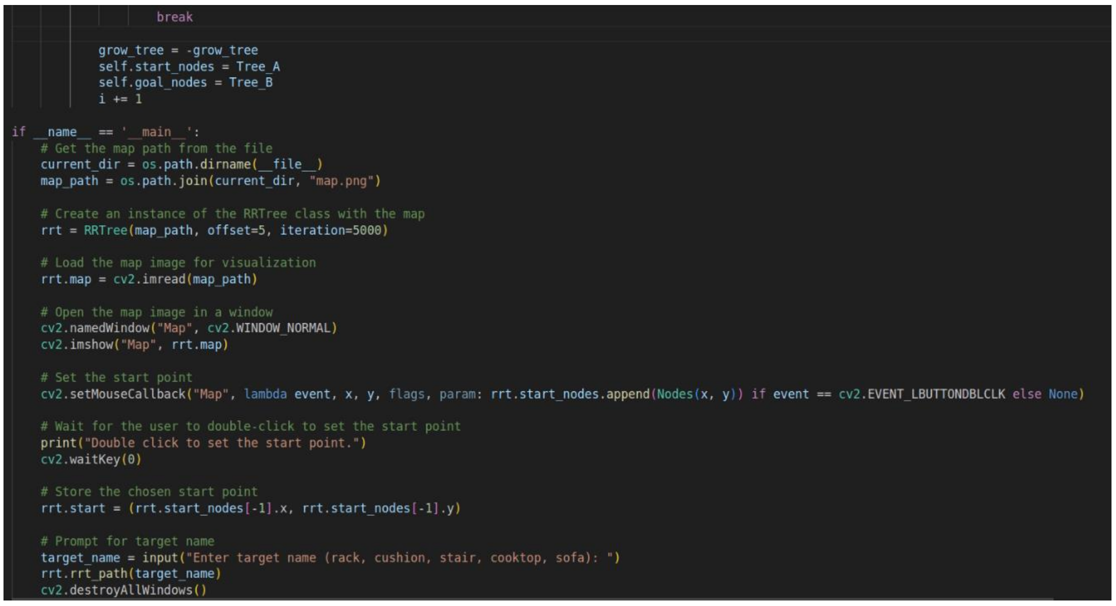
```python
        for point in line_points:
            x, y = map(int, point)
            color.append(img[y, x])

        return 0 in color

    # check the collision with obstacle and the trajectory
    # if there is no collision, add the new point to the tree
    def expansion(self, random_x, random_y, nearest_x, nearest_y, offset, map):
        theta = self.angle(nearest_x, nearest_y, random_x, random_y)
        new_node_x = nearest_x + offset * np.cos(theta)
        new_node_y = nearest_y + offset * np.sin(theta)
        height, width = map.shape

        if new_node_y < 0 or new_node_y >= height or new_node_x < 0 or new_node_x >= width:
            expand_connect = False
        else:
            if self.collision(new_node_x, new_node_y, nearest_x, nearest_y, map):
                expand_connect = False
            else:
                expand_connect = True
        return (new_node_x, new_node_y, expand_connect)

    # find the nearest point to the newest point in Tree_A to Tree_B
    def extend(self, Tree_A, Tree_B, map):
        x = Tree_A[-1].x
        y = Tree_A[-1].y
        nearest_index_b = self.nearest_node(x, y, Tree_B)
        nearest_x = Tree_B[nearest_index_b].x
        nearest_y = Tree_B[nearest_index_b].y
        return not self.collision(x, y, nearest_x, nearest_y, map), nearest_index_b
```



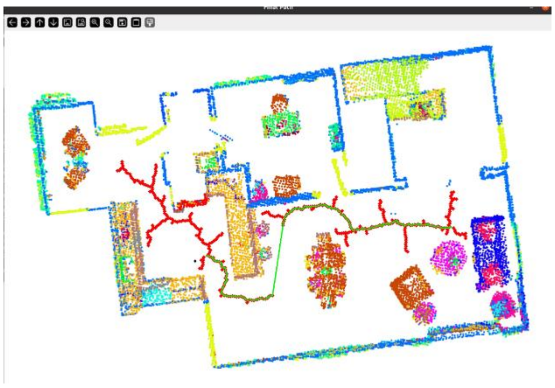
```python
def rrt_path(self, target):
    if target != " ":
        self.set_target(target)
        print("Successfully found the target.")
    else:
        print("No object target.")

    height, width = self.dilation.shape

    # Initialize the start node with the user-defined start point
    self.start_nodes.append(Nodes(self.start[0], self.start[1]))
    self.start_nodes[0].x_parent.append(self.start[0])
    self.start_nodes[0].y_parent.append(self.start[1])

    # Initialize the goal node based on the target object
    self.goal_nodes.append(Nodes(self.goal[0], self.goal[1]))
    self.goal_nodes[0].x_parent.append(self.goal[0])
    self.goal_nodes[0].y_parent.append(self.goal[1])

    grow_tree = -1
    i = 1
    path = []  # Initialize the path variable
    while i < self.iteration:
        Tree_A = self.start_nodes if grow_tree == -1 else self.goal_nodes
        Tree_B = self.goal_nodes if grow_tree == -1 else self.start_nodes

        random_x, random_y = self.random_point(height, width)

        nearest_index = self.nearest_node(random_x, random_y, Tree_A)
        nearest_x, nearest_y = Tree_A[nearest_index].x, Tree_A[nearest_index].y
        new_node_x, new_node_y, expand_connect = self.expansion(random_x, random_y, nearest_x, nearest_y, self.offset, self.dilation)
```



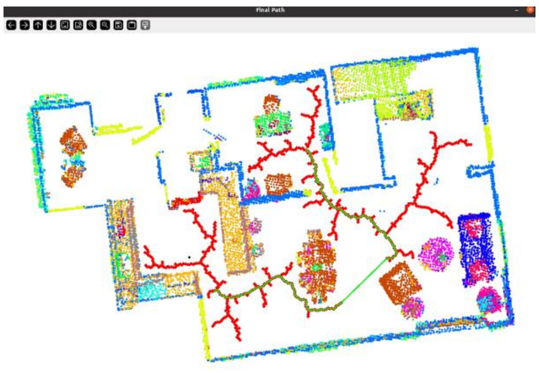
```python
        if expand_connect:
            new_node = Nodes(new_node_x, new_node_y)
            Tree_A.append(new_node)  # Append the new node to the tree
            Tree_A[-1].x_parent = Tree_A[nearest_index].x_parent.copy()  # Copy the parent
            Tree_A[-1].y_parent = Tree_A[nearest_index].y_parent.copy()
            Tree_A[-1].x_parent.append(new_node_x)
            Tree_A[-1].y_parent.append(new_node_y)

            # Optional: visualize tree growth
            color = (0, 0, 255)
            cv2.circle(self.map, (int(new_node_x), int(new_node_y)), 2, color, thickness=3, lineType=8)
            cv2.line(self.map, (int(new_node_x), int(new_node_y)), (int(Tree_A[nearest_index].x), int(Tree_A[nearest_index].y)), color, thickness=1, lineType=8)
            cv2.imshow("RRT Visualization", self.map)
            cv2.waitKey(1)

            # Find the node closest to the new node in Tree_B
            nearest_index_b = self.nearest_node(new_node_x, new_node_y, Tree_B)
            nearest_x_b = Tree_B[nearest_index_b].x
            nearest_y_b = Tree_B[nearest_index_b].y

            # Check the connection between the new node and the nearest node in Tree_B
            if not self.collision(new_node_x, new_node_y, nearest_x_b, nearest_y_b, self.dilation):
                print("Path is successfully formulated between the two trees.")
                path = []

                # Constructing the path
                for j in range(len(Tree_A[-1].x_parent)):
                    path.append((Tree_A[-1].x_parent[j], Tree_A[-1].y_parent[j]))
                Tree_B[nearest_index_b].x_parent.reverse()
                Tree_B[nearest_index_b].y_parent.reverse()
                for j in range(len(Tree_B[nearest_index_b].x_parent)):
                    path.append((Tree_B[nearest_index_b].x_parent[j], Tree_B[nearest_index_b].y_parent[j]))

                # Visualize the final path in green
                for j in range(len(path) - 1):
                    cv2.line(self.map, (int(path[j][0]), int(path[j][1])), (int(path[j + 1][0]), int(path[j + 1][1])), (0, 255, 0), thickness=2, lineType=8)

                cv2.imshow("Final Path", self.map)
                cv2.waitKey(0)  # Wait until a key is pressed
                # Save the image with the path
                if self.target == " ":
                    cv2.imwrite("birrt.jpg", self.map)
                else:
                    cv2.imwrite("bi-RRT_Path/" + self.target + ".jpg", self.map)
```



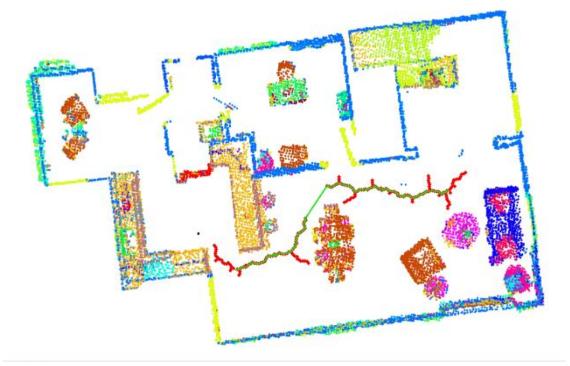
```python
                    break

            grow_tree = -grow_tree
            self.start_nodes = Tree_A
            self.goal_nodes = Tree_B
            i += 1

if __name__ == '__main__':
    # Get the map path from the file
    current_dir = os.path.dirname(__file__)
    map_path = os.path.join(current_dir, "map.png")

    # Create an instance of the RRTree class with the map
    rrt = RRTree(map_path, offset=5, iteration=5000)

    # Load the map image for visualization
    rrt.map = cv2.imread(map_path)

    # Open the map image in a window
    cv2.namedWindow("Map", cv2.WINDOW_NORMAL)
    cv2.imshow("Map", rrt.map)

    # Set the start point
    cv2.setMouseCallback("Map", lambda event, x, y, flags, param: rrt.start_nodes.append(Nodes(x, y)) if event == cv2.EVENT_LBUTTONDBLCLK else None)

    # Wait for the user to double-click to set the start point
    print("Double click to set the start point.")
    cv2.waitKey(0)

    # Store the chosen start point
    rrt.start = (rrt.start_nodes[-1].x, rrt.start_nodes[-1].y)

    # Prompt for target name
    target_name = input("Enter target name (rack, cushion, stair, cooktop, sofa): ")
    rrt.rrt_path(target_name)
    cv2.destroyAllWindows()
```



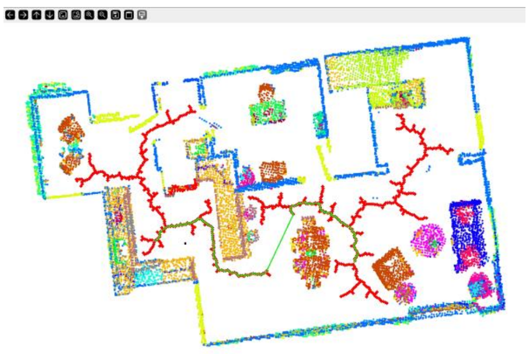
sofa



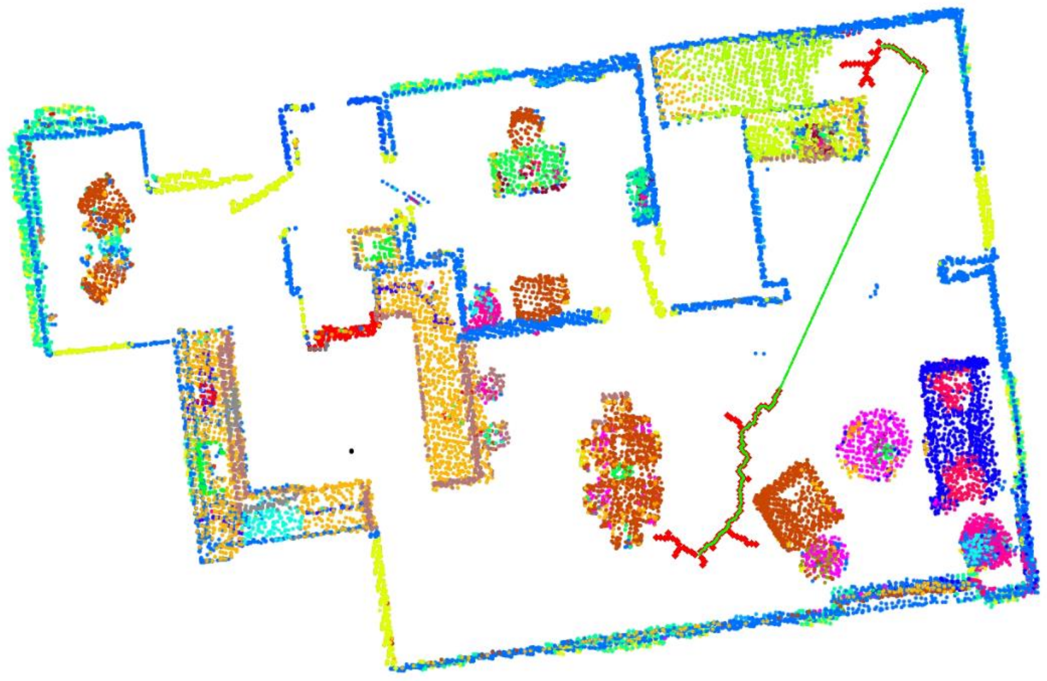
rack



cushion



cooktop



stair

In this program, the bi-RRT (Bidirectional Rapidly-exploring Random Tree) algorithm is used, which has several advantages over the standard RRT algorithm.

Compared to a single-tree RRT, bi-RRT grows two trees simultaneously: one from the starting point and another from the target point. The trees attempt to meet each other in the middle, which often leads to faster convergence, especially in environments with complex obstacles. By reducing the search space in half, bi-RRT improves the chances of finding a path efficiently. Additionally, the program also includes visualizations of tree growth and path formulation, making it easier to track progress during the algorithm's execution.

This bidirectional approach reduces the number of iterations required, enhances the ability to navigate around obstacles, and provides a more efficient solution to pathfinding problems, particularly in large and cluttered environments.