Physical AI Homework 2:

A Robot Navigation Framework

111550132 張家睿

1. Implementation

a. Code

```
import numpy as np
from PIL import Image
import habitat_sim
from habitat_sim.utils.common import d3_40_colors_rgb
import cv2
import os
import shutil
import math
import matplotlib.pyplot as plt
import pandas as pd
import random
import math
from collections import namedtuple
import json

def part1():
    points = np.load('../semantic_3d_pointcloud/point.npy')
    colors = np.load('../semantic_3d_pointcloud/color0255.npy')

    # remove ceiling & floor
    y_min, y_max = np.percentile(points[:, 1], [25, 60])
    mask = (points[:, 1] > y_min) & (points[:, 1] < y_max)
    points = points[mask]
    colors = colors[mask]

    # plot x-z plane
    x = points[:, 0]
    z = points[:, 2]
    meta = {
        "x_min": x.min(),
        "x_max": x.max(),
        "z_min": z.min(),
        "z_max": z.max()
    }

    plt.figure(figsize=(6, 6))
    plt.scatter(x, z, c=colors / 255.0, s=0.1)
    plt.gca().invert_yaxis()
    plt.axis('equal')
    plt.axis('off')
    plt.tight_layout()
    plt.savefig('../results/map.png', dpi=600, bbox_inches='tight', pad_inches=0)
    plt.close()

    print("saved as results/map.png")
    return meta

def part2(meta):
    # RRT Hyperparameters
    MAX_ITER = 6000
    STEP_SIZE = 0.02
    GOAL_THRESHOLD = 0.03
    SAFETY_RADIUS = 0.002

    # load map
    map_img = cv2.imread("../results/map.png")
    if map_img is None:
        raise FileNotFoundError("../results/map.png not found")
    h, w, _ = map_img.shape
    gray = cv2.cvtColor(map_img, cv2.COLOR_BGR2GRAY)

    # occupancy map
    _, occ_map = cv2.threshold(gray, 250, 255, cv2.THRESH_BINARY)
    occ_map = 255 - occ_map
    occ_map = occ_map > 0
    # occupancy map inflation
    SAFETY_RADIUS_pixels = SAFETY_RADIUS * w / (meta['x_max'] - meta['x_min'])
    k = 2 * math.ceil(SAFETY_RADIUS_pixels) + 1
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (k, k))
    occ_map = cv2.dilate(occ_map.astype(np.uint8), kernel).astype(bool)

    non_white_mask = np.any(map_img != 255, axis=2)  # True where pixel is not white
    ys, xs = np.where(non_white_mask)
    v_min, v_max = int(ys.min()), int(ys.max())  # rows
    u_min, u_max = int(xs.min()), int(xs.max())  # cols
```

```python
# transfer coordinates between pixel & world
def pixel_to_world(u, v):
    if (u_max - u_min) == 0 or (v_max - v_min) == 0:
        raise ValueError("degenerate non-white bbox")
    x = meta['x_min'] + ( (u - u_min) / (u_max - u_min) ) * (meta['x_max'] - meta['x_min'])
    z = meta['z_min'] + ( (v - v_min) / (v_max - v_min) ) * (meta['z_max'] - meta['z_min'])
    return float(x), float(z)

def world_to_pixel(x, z):
    if (meta['x_max'] - meta['x_min']) == 0 or (meta['z_max'] - meta['z_min']) == 0:
        raise ValueError("degenerate world bbox")
    u = u_min + ( (x - meta['x_min']) / (meta['x_max'] - meta['x_min']) ) * (u_max - u_min)
    v = v_min + ( (z - meta['z_min']) / (meta['z_max'] - meta['z_min']) ) * (v_max - v_min)
    return int(round(u)), int(round(v))

# Parse color map from xlsx
df = pd.read_excel("../color_coding_semantic_segmentation_classes.xlsx", dtype=str)
color_dict = {}
for _, row in df.iterrows():
    name = str(row.get("Name", "")).strip().lower()
    rgb_str = row.get("Color_Code (R,G,B)", "")
    if not name or not rgb_str:
        continue
    rgb = list(int(c) for c in rgb_str.strip("()").split(","))
    bgr = [rgb[2], rgb[1], rgb[0]]
    color_dict[name] = bgr

# target color region
def find_target_region(img, target_name):
    if target_name not in color_dict:
        raise ValueError(f"Target '{target_name}' does not exist.")
    target_color = np.array(color_dict[target_name])
    mask = np.all(img == target_color, axis=-1)
    coords = np.column_stack(np.where(mask))  # rows, cols -> v,u
    target_id = int(float(df.loc[df["Name"].eq(target_name)].iloc[0, 0]))
    return target_id, coords

def distance(a, b):
    return math.hypot(a[0] - b[0], a[1] - b[1])

def nearest(nodes, point):
    dists = [distance((n.x, n.z), point) for n in nodes]
    return nodes[np.argmin(dists)]

def collision_free(a, b):
    ua, va = world_to_pixel(a[0], a[1])
    ub, vb = world_to_pixel(b[0], b[1])
    pix_dist = math.hypot(ub - ua, vb - va)
    xs_world = np.linspace(a[0], b[0], max(20, int(pix_dist)))
    zs_world = np.linspace(a[1], b[1], max(20, int(pix_dist)))
    for xw, zw in zip(xs_world, zs_world):
        u, v = world_to_pixel(xw, zw)
        if u < 0 or v < 0 or u >= w or v >= h:
            return False
        if occ_map[v, u]:
            return False
    return True

# RRT in world coordinates
def RRT(start, goal):
    Node = namedtuple("Node", ["x", "z", "parent"])
    nodes = [Node(start[0], start[1], -1)]
    edges = []
    for i in range(MAX_ITER):
        if random.random() < 0.1:
            sample = goal
        else:
            x_rand = random.uniform(meta['x_min'], meta['x_max'])
            z_rand = random.uniform(meta['z_min'], meta['z_max'])
            sample = (x_rand, z_rand)

        nearest_node = nearest(nodes, sample)
        theta = math.atan2(sample[1] - nearest_node.z, sample[0] - nearest_node.x)
        new_x = nearest_node.x + STEP_SIZE * math.cos(theta)
        new_z = nearest_node.z + STEP_SIZE * math.sin(theta)

        if not collision_free((nearest_node.x, nearest_node.z), (new_x, new_z)):
            continue

        new_node = Node(new_x, new_z, nodes.index(nearest_node))
        nodes.append(new_node)
        edges.append(((nearest_node.x, nearest_node.z), (new_x, new_z)))

        if distance((new_x, new_z), goal) < GOAL_THRESHOLD:
            path = [(new_x, new_z)]
            parent = new_node.parent
            while parent != -1:
                n = nodes[parent]
                path.append((n.x, n.z))
                parent = n.parent
            path.reverse()
            print(f"Goal reached in {i} iterations!")
            return path, edges
    print("Failed to reach goal.")
    return None, edges

def RRT_star(start, goal):
    Node = namedtuple("Node", ["x", "z", "parent", "cost"])
    nodes = [Node(float(start[0]), float(start[1]), -1, 0.0)]

    world_diam = math.hypot(meta['x_max'] - meta['x_min'], meta['z_max'] - meta['z_min'])
    GAMMA = 0.8 * world_diam

    def steer(from_xy, to_xy, step=STEP_SIZE):
        dx, dz = to_xy[0] - from_xy[0], to_xy[1] - from_xy[1]
        dist = math.hypot(dx, dz)
        if dist <= 1e-12:
            return from_xy
        scale = min(1.0, step / dist)
        return (from_xy[0] + dx * scale, from_xy[1] + dz * scale)

    def nearest_index(pt):
        best_i, best_d2 = 0, float("inf")
        for i, n in enumerate(nodes):
            d2 = (n.x - pt[0])**2 + (n.z - pt[1])**2
            if d2 < best_d2:
                best_i, best_d2 = i, d2
        return best_i
```

```python
    def neighbor_indices(center_xy, n_now):
        r = max(1.5 * STEP_SIZE, GAMMA * math.sqrt(max(1e-9, math.log(n_now + 1) / (n_now + 1))))
        r2 = r * r
        cand = []
        for i, n in enumerate(nodes):
            if (n.x - center_xy[0])**2 + (n.z - center_xy[1])**2 <= r2:
                cand.append(i)
        # sort by distance then cap
        cand.sort(key=lambda i: (nodes[i].x - center_xy[0])**2 + (nodes[i].z - center_xy[1])**2)
        return cand[:30]

    def backtrack(last_idx):
        path = []
        p = last_idx
        while p != -1:
            n = nodes[p]
            path.append((n.x, n.z))
            p = n.parent
        path.reverse()
        return path

    goal_idx = None
    edges = []
    for it in range(MAX_ITER):
        if random.random() < 0.1:
            sample = goal
        else:
            sample = (random.uniform(meta['x_min'], meta['x_max']), random.uniform(meta['z_min'], meta['z_max']))

        # Nearest + steer
        ni = nearest_index(sample)
        xn = (nodes[ni].x, nodes[ni].z)
        x_new = steer(xn, sample, STEP_SIZE)

        if not collision_free(xn, x_new):
            continue

        # parent among neighbors
        neigh = neighbor_indices(x_new, len(nodes))
        best_parent = ni
        best_cost = nodes[ni].cost + distance(xn, x_new)

        for j in neigh:
            pj = (nodes[j].x, nodes[j].z)
            if not collision_free(pj, x_new):
                continue
            cand_cost = nodes[j].cost + distance(pj, x_new)
            if cand_cost + 1e-12 < best_cost:
                best_parent = j
                best_cost = cand_cost

        # add node
        nodes.append(Node(x_new[0], x_new[1], best_parent, best_cost))
        new_idx = len(nodes) - 1

        # rewire neighbors if improves cost
        for j in neigh:
            if j == best_parent or j == new_idx:
                continue
            pj = (nodes[j].x, nodes[j].z)
            alt_cost = nodes[new_idx].cost + distance(x_new, pj)
            if alt_cost + 1e-12 < nodes[j].cost and collision_free(x_new, pj):
                nodes[j] = Node(nodes[j].x, nodes[j].z, new_idx, alt_cost)

        if distance(x_new, goal) < GOAL_THRESHOLD:
            goal_idx = new_idx
            print(f"Goal reached (RRT*) in {it} iterations!")
            path = backtrack(goal_idx)
            for i in range(1, len(nodes)):
                p = nodes[i].parent
                if p >= 0:
                    edges.append(((nodes[p].x, nodes[p].z), (nodes[i].x, nodes[i].z)))
            return path, edges

    print("Failed to reach goal (RRT*).")
    for i in range(1, len(nodes)):
        p = nodes[i].parent
        if p >= 0:
            edges.append(((nodes[p].x, nodes[p].z), (nodes[i].x, nodes[i].z)))
    return None, edges


def get_start_point(img, goal_pixel):
    fig, ax = plt.subplots()
    ax.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
    ax.plot(goal_pixel[0], goal_pixel[1], 'go', markersize=5)
    plt.title("Select start point (click anywhere on image)")
    pts = plt.ginput(1, timeout=0)
    plt.close()
    if not pts:
        raise ValueError("No point selected.")
    u, v = map(int, pts[0])
    return pixel_to_world(u, v)

def simplify_path(path):
    if not path or len(path) < 3:
        return path
    simplified = [path[0]]
    i = 0
    n = len(path)
    while i < n - 1:
        j = i + 1
        while j + 1 < n and collision_free(path[i], path[j + 1]):
            j += 1
        simplified.append(path[j])
        i = j
    return simplified
```

```python
    target = input("Enter target class (rack, cushion, sofa, stair, and cooktop): ").strip().lower()
    target_id, coords = find_target_region(map_img, target)
    if coords.size == 0:
        raise ValueError(f"No region found for {target}.")

    v_mean, u_mean = np.mean(coords, axis=0).astype(int)
    goal_world = pixel_to_world(u_mean, v_mean)
    start_world = get_start_point(map_img, (u_mean, v_mean))

    path, edges = RRT_star(start_world, goal_world)
    simplify = simplify_path(path)

    # draw edges & path on output image by converting world to pixel
    out = map_img.copy()
    for (a, b) in edges:
        p1 = world_to_pixel(a[0], a[1])
        p2 = world_to_pixel(b[0], b[1])
        cv2.line(out, p1, p2, (255, 0, 0), 1)

    if path:
        cv2.circle(out, world_to_pixel(*start_world), 7, (0, 0, 255), -1)
        cv2.circle(out, world_to_pixel(*goal_world), 7, (0, 255, 0), -1)
        for i in range(1, len(path)):
            cv2.line(out, world_to_pixel(*path[i-1]), world_to_pixel(*path[i]), (255, 0, 0), 5)
        for i in range(1, len(simplify)):
            cv2.line(out, world_to_pixel(*simplify[i-1]), world_to_pixel(*simplify[i]), (0, 0, 255), 3)
        cv2.imwrite(f"../results/path_{target}.png", out)
    display = cv2.resize(out, None, fx=0.2, fy=0.2)
    cv2.imshow("RRT", display)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

    return target_id, target, simplify

def part3(target_id, target, path):
    # This is the scene we are going to load.
    test_scene = "../../hw0/replica_v1/apartment_0/habitat/mesh_semantic.ply"
    test_scene_info_semantic = "../../hw0/replica_v1/apartment_0/habitat/info_semantic.json"

    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": 0,  # sensor pitch (x rotation in rads)
    }

    # 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 transform_rgb_bgr(image):
        return image[:, :, [2, 1, 0]]

    def semantic_label_to_id(semantic_sensor_label):
        with open(test_scene_info_semantic, "r") as f:
            annotations = json.load(f)
            id_to_label = np.where(np.array(annotations["id_to_label"]) < 0, 0, annotations["id_to_label"])
            id_mask = id_to_label[semantic_sensor_label]
            return id_mask

    def make_simple_cfg(settings):
        # simulator backend
        sim_cfg = habitat_sim.SimulatorConfiguration()
        sim_cfg.scene_id = settings["scene"]
        # agent
        agent_cfg = habitat_sim.agent.AgentConfiguration()

        # 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

        #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_cfg.sensor_specifications = [rgb_sensor_spec, semantic_sensor_spec]
        agent_cfg.action_space = {
            "move_forward": habitat_sim.agent.ActionSpec(
                "move_forward", habitat_sim.agent.ActuationSpec(amount=0.2)
            ),
            "turn_left": habitat_sim.agent.ActionSpec(
                "turn_left", habitat_sim.agent.ActuationSpec(amount=5)
            ),
            "turn_right": habitat_sim.agent.ActionSpec(
                "turn_right", habitat_sim.agent.ActuationSpec(amount=5)
            ),
        }
        return habitat_sim.Configuration(sim_cfg, [agent_cfg])

    def navigateAndSee(count, action="", data_root='data_collection/second_floor/'):
        observations = sim.step(action)

        rgb_img = transform_rgb_bgr(observations["color_sensor"])
        # add overlay for target object
        semantic_id_mask = semantic_label_to_id(observations["semantic_sensor"])
        target_id_region = semantic_id_mask == target_id
        overlay = rgb_img.copy()
        overlay[target_id_region] = (0, 0, 255)
        rgb_img = cv2.addWeighted(rgb_img, 0.5, overlay, 0.5, 0.0)

        cv2.imshow("RGB", rgb_img)
        agent_state = agent.get_state()
        sensor_state = agent_state.sensor_states['color_sensor']
        # print("Frame:", count)
        print("camera pose: x y z rw rx ry rz")
        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)

        cv2.imwrite(data_root + f"rgb/{count}.png", rgb_img)

    def frames_to_gif(frames_dir, out_gif_path, fps=10, hold_last_ms=500):
        files = glob.glob(os.path.join(frames_dir, "*.png"))
        if not files:
            raise FileNotFoundError(f"No frames found in: {frames_dir}")

        def sort_key(p):
            base = os.path.splitext(os.path.basename(p))[0]
            try:
                return int(base)
            except ValueError:
                return base

        files = sorted(files, key=sort_key)

        imgs = []
        first_size = None
        for f in files:
            im = Image.open(f).convert("RGB")
            if first_size is None:
                first_size = im.size
            elif im.size != first_size:
                im = im.resize(first_size, Image.BILINEAR)
            imgs.append(im)

        frame_duration = max(1, int(round(1000 / fps)))
        durations = [frame_duration] * len(imgs)
        if hold_last_ms and len(durations) > 0:
            durations[-1] += int(hold_last_ms)

        imgs[0].save(
            out_gif_path,
            save_all=True,
            append_images=imgs[1:],
            duration=durations,
            loop=0,
            disposal=2,
            optimize=True,
        )
        print(f"Saved GIF: {out_gif_path} ({len(imgs)} frames)")

    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
    agent_state = habitat_sim.AgentState()


    agent_state.position = np.array([path[0][0], 0.0, path[0][1]])  # 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())
    print("Discrete action space: ", action_names)

    data_root = "data_collection/first_floor/"

    if os.path.isdir(data_root):
        shutil.rmtree(data_root)  # WARNING: this line will delete whole directory with files

    for sub_dir in ['rgb/']:
        os.makedirs(data_root + sub_dir)

    count = 0
    action = "move_forward"

    navigateAndSee(count, action, data_root)
```

```python
    def quat_to_yaw(w, x, y, z):
        return math.atan2(2.0 * (w * y + x * z), 1 - 2.0 * (y**2 + z**2))

    for pos_x, pos_z in path[1:]:
        while True:
            agent_state = agent.get_state()
            sensor_state = agent_state.sensor_states['color_sensor']
            x = sensor_state.position[0]
            z = sensor_state.position[2]
            rw = sensor_state.rotation.w
            ry = sensor_state.rotation.y
            yaw = quat_to_yaw(rw, 0, ry, 0)

            target_yaw = -math.atan2(pos_x - x, -pos_z + z)
            angle_diff = np.degrees(np.arctan2(np.sin(target_yaw - yaw), np.cos(target_yaw - yaw))) # normalize to [-180, 180]
            # print(f"Angle diff: {angle_diff:.2f}")
            if (pos_x - x)**2 + (pos_z - z)**2 < 0.01:
                break
            elif abs(angle_diff) > 5:
                if angle_diff > 0:
                    action = "turn_left"
                    navigateAndSee(count, action, data_root)
                    print("action: LEFT")
                else:
                    action = "turn_right"
                    navigateAndSee(count, action, data_root)
                    print("action: RIGHT")
            else:
                action = "move_forward"
                navigateAndSee(count, action, data_root)
                print("action: FORWARD")
            count += 1
            delay = 15
            cv2.waitKey(delay)
    print("Navigation completed")
    frames_to_gif(data_root + "rgb/", f"../results/{target}.gif")

if __name__ == "__main__":
    meta = part1()
    target_id, target, path = part2(meta)
    path = [[x * 10000./255 for x in row] for row in path]
    part3(target_id, target, path)
```

part1(): This function loads a 3D point cloud and matching RGB colors from disk, removes the floor and ceiling by keeping only points whose height (y column) lies between 0.25 and 0.60 percent of the height, then projects the remaining points onto the x–z plane to make a top-down map. It extracts x = points[:,0] and z = points[:,2], computes their min & max to return as meta, and renders a dense scatter plot where each point is colored by its corresponding RGB value (normalized to 0~1). The plot uses an equal aspect ratio, inverts the y-axis, hides axes, tightens layout, saves a high-resolution image to ../results/map.png, closes the figure, prints a confirmation message, and finally returns the x/z bounds in meta for part2() use.

---

part2(): Sets RRT/RRT* parameters, loads the previously saved top-down map, builds a binary occupancy grid, then inflates obstacles by a safety radius via morphological dilation(to prevent collision). And computes the non-white pixel bounding box to anchor pixel to world conversions. It loads a color legend from Excel, lets you choose a target class by name, locates that class's pixel region, converts its centroid to world coordinates (goal), and asks you to click a start point. It runs RRT/RRT* to plan a collision-free path, simplifies the path by removing unnecessary waypoints, draws edges and paths onto the map, saves/opens the visualization, prints the result, and returns the target ID, target name and simplified path for part3().

pixel_to_world(): Converts pixel coordinates (u, v) within the non-white image bounds into world coordinates (x, z) using linear interpolation based on the meta x/z limits and the detected pixel bounding box, guarding against degenerate extents.

world_to_pixel(): Converts world coordinates (x, z) back into integer image coordinates (u, v) by linearly scaling with the meta ranges and the non-white pixel bounding box; it also checks for degenerate world extents.

find_target_region(): Looks up the BGR color for a semantic class name using the Excel color table, creates a boolean mask of pixels exactly matching that color, collects their (row, col) coordinates, and returns both the class ID and all pixel coordinates belonging to that class.

distance(): Computes standard Euclidean distance between two world-space points (x, z).

nearest(): Finds the node in the current tree whose (x, z) position is closest to the given world-space point by scanning all nodes and picking the one with the minimum Euclidean distance.

collision_free(): Samples points along the straight segment from world point a to b at a density tied to the pixel distance, converts each sample to pixels, rejects segments that leave the image or hit any occupied cell in the occupancy map, and returns True only if all sampled pixels are free.

RRT(): Creates Node(x, z, parent) and seeds the tree with the start node (parent = -1). An empty edges list will store line segments for visualization. Sampling with goal bias: up to MAX_ITER, each iteration draws a random point in the world bounds, but with 10% probability, sets the sample to the goal to speed convergence. Find the existing node closest to the sampled point using nearest(). Compute the heading theta through arctangent and take a single step of length STEP_SIZE along that direction to produce (new_x, new_z). Collision check: call collision_free(), which discretizes the segment at roughly one point per pixel and rejects it if any sample lands outside the image or on an occupied cell in occ_map. If blocked, skip this iteration. Append the new node with parent = index(nearest_node), and push the corresponding line segment into edges. If the new node is within GOAL_THRESHOLD of the goal, backtrack via parent pointers to reconstruct the path from start to goal, reverse it, print success, and return path, edges. If MAX_ITER is reached without satisfying the threshold, print failure and return None, edges.

RRT_star(): Creates Node(x, z, parent, cost) and seeds the tree with the start node (parent = -1, cost=0.0). Asymptotic-optimal neighbor radius. Compute world_diam and GAMMA = 0.8 * world_diam. Each iteration uses a dynamic radius to collect nearby nodes; the radius shrinks as the tree grows. Sampling with goal bias: up to MAX_ITER, each iteration draws a random point in the world bounds, but with 10% probability, sets the sample to the goal to speed convergence. Use steer(from_xy, to_xy, step) to get the point at most STEP_SIZE away toward the target. If collision_free() fails, skip the iteration. Choose the best parent: initialize best_parent = the nearest with baseline cost nearest index node + distance(). For each neighbor j inside the dynamic radius: if the straight line pj to x_new is collision-free, evaluate cand_cost = nodes[j].cost + distance(pj, x_new) and keep the parent that gives the lowest cost. Then append Node(x_new[0], x_new[1], best_parent, best_cost). For each neighbor j, if going through the new node reduces nodes[j].cost, then update that neighbor's parent to new_idx and set its lower cost. This step incrementally shortens paths in the tree. If the new node is within GOAL_THRESHOLD of the goal, backtrack via parent pointers to reconstruct the path from start to goal, reverse it, print success, and return path, edges. If MAX_ITER is reached without satisfying the threshold, print failure and return None, edges.

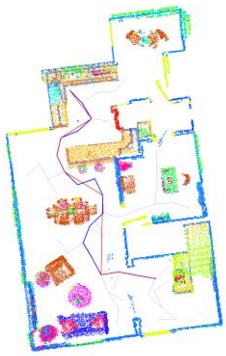
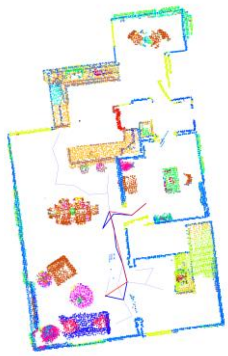
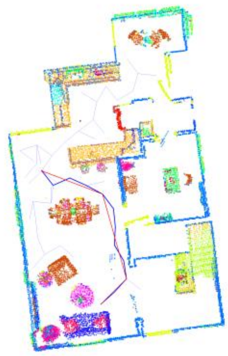
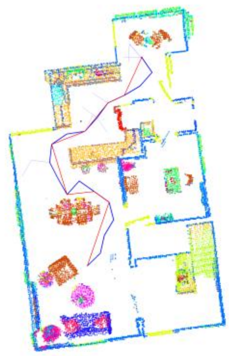
get_start_point(): Displays the map with the goal marked, prompts you to click anywhere to pick a start, captures that pixel via plt, and converts it from pixel to world coordinates for planning.

simplify_path(): Greedily compresses the waypoint list by skipping intermediate points whenever a direct straight line from the current kept point to a farther point remains collision-free, producing a shorter, cleaner path that preserves feasibility.

part3(): Defines the Replica scene paths and simulator settings, declares helper functions, builds a Habitat-Sim configuration, spawns a simulator and agent, sets the agent at the first (x,z) waypoint with y=0, prepares an output folder, then runs a waypoint-following loop that repeatedly chooses an action (turn_left, turn_right, or move_forward) to align the agent with the next waypoint and advance toward it while rendering, highlighting pixels that match target_id, saving frames, and printing camera pose until all waypoints are reached.
transform_rgb_bgr(): Rearranges the last axis of the image from RGB to BGR so that frames produced by Habitat can be consumed correctly by OpenCV's BGR-expecting functions.
semantic_label_to_id(): Loads info_semantic.json, fixes any negative entries in id_to_label to zero, and uses that array as a lookup table to map each per-pixel label from the semantic sensor into a global class-ID image that aligns with your dataset's canonical ID.
make_simple_cfg(): Constructs a habitat_sim.Configuration by setting the scene on a SimulatorConfiguration, creating an AgentConfiguration with color and semantic pinhole camera sensors, specifying 512×512 resolution, and defining a discrete action space: move_forward with 0.2 m step and turn_left/turn_right with 5 degrees increments.
navigateAndSee(): Executes one simulator step with the chosen action, converts the returned color frame to BGR, derives a per-pixel class-ID image from semantic_label_to_id(), builds a copy where pixels equal to target_id are tinted red, and blends it 50%/50% onto the original RGB for visualization. Shows the frame, prints the camera pose, and writes the image to data_root/rgb/{count}.png.
frames_to_gif(): Builds an animated GIF from a folder of PNG frames: it gathers all *.png files, sorts them by their numeric filename, loads each image as RGB, and resizes any mismatched frame to the first frame's size to keep the animation consistent. It computes a per-frame duration in milliseconds from fps, then adds extra time to the final frame so the GIF briefly pauses at the end. Finally, it writes the GIF using Pillow and prints a summary including the output path and frame count.
quat_to_yaw(): Extracts the yaw from a quaternion using arctangent $(2(wy + xz), 1 - 2(y^2 + z^2))$. It's called with (w, 0, ry, 0) because only the quaternion's w and y components are needed under the assumed convention to compute heading around Y.

b. Result and Discussion

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

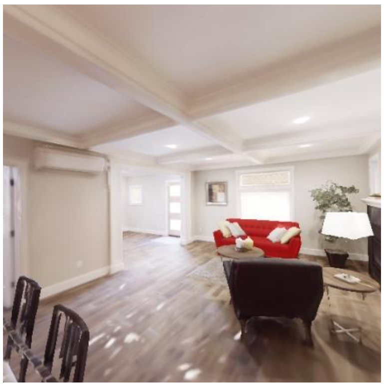
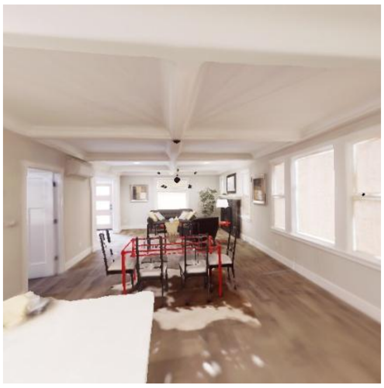





cooktop rack sofa table

The RRT blue path is a roundabout, so I use the greedy algorithm to reduce the redundant points and get the simplified red path. But it sometimes can't get a good result when passing through the narrow door.

ii. **Discuss about the robot navigation results**

When setting the amount of turn_left and turn_right to 10, the robot can get to the target, but it will turn its head too frequently to adjust to the right orientation. So, I adjusted the amount to 5 for smoother moving. Also, I adjusted the move_forward speed to 0.2 to prevent the robot from moving too fast compared to the length between different nodes in the path.

iii. **Anything you want to discuss**

Why sometimes my RRT path may let the robot bump into the wall? Because the robot radius isn't considered when path planning. Inflating the occupancy map with a safety radius can prevent the collision.

c. Any reference you take

i. https://docs.opencv.org/4.x/d5/dc4/tutorial_adding_images.html

ii. https://www.youtube.com/@NTHURNE-l9v (Robotic Navigation and Exploration course opened @NTHU last semester)

2. Questions

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

A small step size yields de nse coverage and better chances of threading narrow passages and smoother initial paths; however, it requires more iterations. A large step size explores the space quickly with fewer nodes, yet it often collides or skips feasible corridors, producing coarser, jaggier paths.

A high bias drives the tree quickly toward the target and can cut solution time in open spaces, but risks premature focus, missing alternative routes, or getting stuck near obstacles. A low bias improves global coverage and robustness in cluttered maps, at the cost of slower time to first solution.

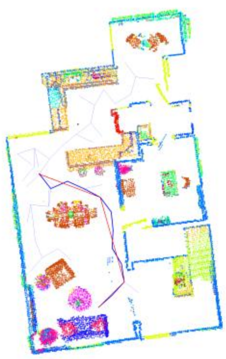
b. **If you want to apply the indoor navigation pipeline in the real world, what problems may you encounter?**

In the real world, we can't get the semantic images. This will make the localization of the target hard. Second, a real-world camera may have time sync issues with the wheel controller and main function. Third, the sensor may have move noise and drift in the real world.

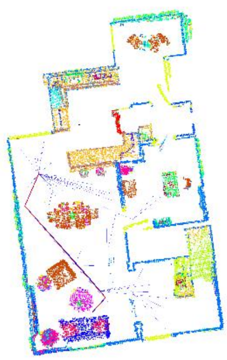
3. Bonus

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

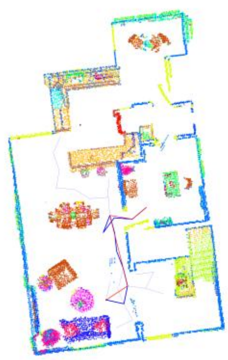
I implement the RRT* algorithm, the code and explain are already in the implementation part. Compared with RRT, RRT* improves the best path cost over iterations by rewiring nearby nodes, whereas RRT's best cost typically stalls near the initial solution. It's also more reliable in narrow passages: as the tree densifies, it can gradually thread through tight corridors and then use rewiring to straighten the path and lower the cost. However, time to first solution is usually slower. For the same time budget, RRT tends to find a feasible path sooner, while RRT* spends extra time on neighbor searches and rewiring.



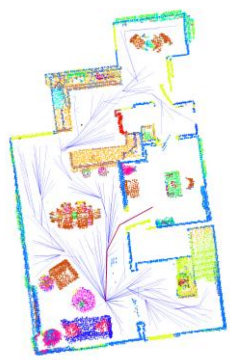
sofa(RRT)



sofa(RRT*)



rack(RRT)



rack(RRT*)