ESE 650 HW 2

Zhanqian Wu
University of Pennsylvania
Philadelphia, PA 19104
{Zhanqian}@seas.upenn.edu
Coauthor Bowen Jiang bwjiang@seas.upenn.edu

1 Problem 1

```
import numpy as np
import matplotlib.pyplot as plt


def generate_transition_matrix(env_shape, obstacles_list, goal_idx):
    # The environment's shape is a square grid of size 'n'
    n = env_shape[0]

    # Initialize the transition matrix with zeros
    # The matrix is 5-dimensional: state (i, j), action, resulting state (new_i, new_j)
    # Actions are encoded as 0: left, 1: right, 2: up, 3: down
    T = np.zeros((n, n, 4, n, n))

    # Iterate over all cells in the grid to set transition probabilities
    for i in range(n):
        for j in range(n):
            # If the cell is the goal state, any action results in staying in the goal with
    probability 1
            if (i, j) == goal_idx:
                T[i, j, :, i, j] = 1
                continue  # Skip further processing for the goal state

            # If the cell is an obstacle, skip it as no action is applicable
            if (i, j) in obstacles_list:
                continue

            # Define transitions based on the control action taken by the robot
            # For each direction, there's a primary movement direction with p=0.7
            # and secondary movements (including staying in place) with p=0.1

            # For action 0 (left)
            T[i, j, 0, i, np.clip(j-1, 0, n-1)] += 0.7
            T[i, j, 0, np.clip(i-1, 0, n-1), j] += 0.1
            T[i, j, 0, np.clip(i+1, 0, n-1), j] += 0.1
            T[i, j, 0, i, j] += 0.1

            # For action 1 (right)
            T[i, j, 1, i, np.clip(j+1, 0, n-1)] += 0.7
            T[i, j, 1, np.clip(i-1, 0, n-1), j] += 0.1
            T[i, j, 1, np.clip(i+1, 0, n-1), j] += 0.1
            T[i, j, 1, i, j] += 0.1

```

```
            # For action 2 (up)
            T[i, j, 2, np.clip(i-1, 0, n-1), j] += 0.7
            T[i, j, 2, i, np.clip(j+1, 0, n-1)] += 0.1
            T[i, j, 2, i, np.clip(j-1, 0, n-1)] += 0.1
            T[i, j, 2, i, j] += 0.1

            # For action 3 (down)
            T[i, j, 3, np.clip(i+1, 0, n-1), j] += 0.7
            T[i, j, 3, i, np.clip(j+1, 0, n-1)] += 0.1
            T[i, j, 3, i, np.clip(j-1, 0, n-1)] += 0.1
            T[i, j, 3, i, j] += 0.1

    return T


def generate_state_map(env_shape):
    """
    Generates a map of the environment with obstacles and points of interest.

    Args:
        env_shape (tuple): The dimensions of the environment (height, width).

    Returns:
        np.ndarray: A 2D array representing the environment where
                    0 = free space,
                    1 = obstacle,
                    2 = goal,
                    3 = initial position,
                    4 = point of interest.
    """
    # Initialize the environment with free spaces
    state_map = np.zeros(env_shape)

    # Set the boundaries of the environment as obstacles
    state_map[:, 0] = state_map[:, -1] = state_map[0, :] = state_map[-1, :] = 1

    # Add specific obstacles within the environment
    state_map[2, 3:7] = 1
    state_map[4:8, 4] = 1
    state_map[7, 5] = 1
    state_map[4:6, 7] = 1

    # Flip the map if needed to match the desired orientation
    state_map = np.flip(state_map, axis=1)
    state_map = np.flip(state_map)

    # Mark specific points of interest
    state_map[8, 8] = 2  # Goal position
    state_map[6, 3] = 3  # Initial position
    state_map[8, 1] = 4  # Another point of interest

    return state_map

def generate_Q_map(states, env_shape, reward):
    """
    Generate the Q-value map for an environment.

    Args:
        states (np.ndarray): A 2D array where cells are marked with 0 for free space,
                             1 for obstacles, and 2 for the goal.
        env_shape (tuple): Shape of the environment as (height, width).
        reward (float): The reward for reaching the goal or hitting an obstacle.

    Returns:
        np.ndarray: A 3D array representing the Q-values for each action at each cell.
```

```
                    The shape of the array is (height, width, 4), corresponding to
                    the environment dimensions and four possible actions.
    """

    # Identify the coordinates of obstacles and the goal in the grid
    obstacles = np.where(states == 1)
    goal_idx = np.where(states == 2)

    # Initialize the Q-value map with -1 for all states and actions
    # This encourages exploration by giving a slightly negative value to unvisited states
    Q = -1 * np.ones((env_shape[1], env_shape[0], 4))  # Notice the corrected order of dimensions

    # Assign a negative reward to all actions leading to obstacle states
    # This penalizes hitting obstacles
    Q[obstacles[0], obstacles[1], :] = -reward

    # Assign a positive reward to all actions leading to the goal state
    # This incentivizes reaching the goal
    Q[goal_idx[0], goal_idx[1], :] = reward

    return Q



def policy_evaluation(T, Q, J_init, u):
    """
   Evaluates a policy to estimate the state-value function for each state.

   J[k, i, j]: This represents the estimated value (utility) of being in state (i, j) at iteration k under a specific policy u. The value function J estimates the total expected rewards from being in a particular state and following a certain policy thereafter.

    Q[i, j, u[i, j]]: This is the immediate reward (or the action-value) of taking action u[i, j] (the action recommended by the policy at state (i, j)) plus the expected future rewards. The Q matrix stores the action-value function, which gives the quality of each action at each state.

    gamma: This is the discount factor (denoted as ). It represents the difference in importance

    J[k-1].flatten().T: This term represents the value function from the previous iteration (k-1)

   Args:
       T (np.ndarray): The transition probabilities matrix of shape (n, n, 4, n, n).
       Q (np.ndarray): The action-value function matrix of shape (env_height, env_width, actions).
       J_init (np.ndarray): Initial state-value function of shape (env_height, env_width).
       u (np.ndarray): Policy matrix indicating the action for each state.

   Returns:
       np.ndarray: Estimated state-value function after policy evaluation.
   """
    iter = 300
    J = np.zeros((iter, env_shape[0], env_shape[1]))
    J[0] = J_init
    for k in range(1, iter):
        for i in range(10):
            for j in range(10):
                J[k,i,j] = Q[i,j,u[i,j]] + gamma*(T[i,j,u[i,j]].flatten() @ J[k-1].flatten().T)

    return J[-1]




def policy_improvement(T, Q, J_new, gamma, env_shape):
    """
```

```
    Performs policy improvement by finding an improved policy based on the updated state-value
    function.

    Args:
        T (np.ndarray): Transition probabilities matrix of shape (env_height, env_width, actions,
    env_height, env_width).
        Q (np.ndarray): Action-value function matrix of shape (env_height, env_width, actions).
        J_new (np.ndarray): Updated state-value function of shape (env_height, env_width) from
    policy evaluation.
        gamma (float): Discount factor for future rewards.
        env_shape (tuple): Shape of the environment (height, width).

    Returns:
        np.ndarray: Improved policy matrix indicating the best action for each state.
    """
    # Define a lambda function to find the action that maximizes the expected utility for each
    state
    get_best_action = lambda action_values: np.argmax(action_values)

    # Initialize the improved policy matrix with zeros
    u_k = np.zeros(env_shape)

    # Iterate over all states in the environment
    for i in range(env_shape[0]):
        for j in range(env_shape[1]):
            # Extract Q-values and transition probabilities for the current state
            Q_element = Q[i, j]
            T_element = T[i, j].reshape(4, -1)

            # Compute the expected utility for each action and select the best action
            u_k[i, j] = get_best_action(Q_element + (gamma * T_element @ J_new.reshape((-1, 1))).
    reshape(4))

    # Return the improved policy
    return u_k


def visualize(J_new, u, obstacles_list, goal_idx_list, iteration_number, special_cell=(3, 3)):
    """
    Visualizes the policy and value function of a grid.

    Args:
        J_new (np.ndarray): The value function to be visualized.
        u (np.ndarray): The policy matrix, with actions for each cell.
        obstacles_list (list of tuples): Coordinates of obstacles in the grid.
        goal_idx_list (list of tuples): Coordinates of goal(s) in the grid.
        iteration_number (int): Current iteration number.
        special_cell (tuple): Coordinates of the special cell to be highlighted.
    """
    # Set up the figure and axes for the visualization
    fig, ax = plt.subplots(figsize=(5, 5))
    ax.set_xticks(np.arange(0.5, J_new.shape[1], 1))
    ax.set_yticks(np.arange(0.5, J_new.shape[0], 1))

    # Use a heatmap to visualize the value function
    cmap = plt.get_cmap('bwr')
    im = ax.imshow(J_new, cmap=cmap)

    # Add text displaying the values of each cell with smaller font size and white color
    for i in range(J_new.shape[0]):
        for j in range(J_new.shape[1]):
            if special_cell is not None and (i, j) == special_cell:
                ax.add_patch(plt.Rectangle((j - 0.5, i - 0.5), 1, 1, fill=True, color='green',
    alpha=0.5))
```

```
            ax.text(j, i, round(J_new[i, j], 2), ha='center', va='center', color='white', fontsize=8)
    ax.add_patch(plt.Rectangle((1 - 0.5, 8 - 0.5), 1, 1, fill=True, color='orange', alpha=0.5))
    plt.grid(color='gray', linestyle='--', linewidth=0.5)

    # Draw arrows to represent the policy at each cell
    for i in range(J_new.shape[0]):
        for j in range(J_new.shape[1]):
            if (i, j) not in obstacles_list and (i, j) not in goal_idx_list:
                dx, dy = 0, 0
                if u[i, j] == 0:  # Left
                    dx = -0.25
                elif u[i, j] == 1:  # Right
                    dx = 0.25
                elif u[i, j] == 2:  # Up
                    dy = -0.25
                elif u[i, j] == 3:  # Down
                    dy = 0.25
                ax.arrow(j, i, dx, dy, head_width=0.1, head_length=0.1, fc='red', ec='red')  # Change arrow color here

    fig.colorbar(im, ax=ax)
    plt.title(f"Actions taken at iteration {iteration_number}")
    plt.show()

# Example usage:
# visualize(J_new, u, obstacles_list, goal_idx_list, iteration_number, special_cell=(2, 3))




# Initial setup
env_shape = (10, 10)
reward = 10
gamma = 0.9

# Generate environment states, transition matrix, and initial Q-values
states = generate_state_map(env_shape)
obstacles_list = list(zip(*np.where(states == 1)))
goal_idx_list = list(zip(*np.where(states == 2)))
T = generate_transition_matrix(env_shape, obstacles_list, goal_idx_list[0])
Q = generate_Q_map(states, env_shape, reward)

# Initialize policy evaluation and improvement
num_policy_iter = 5
J = np.zeros(env_shape)
u = np.ones((num_policy_iter+1, env_shape[0], env_shape[1]), dtype=int)

# Iterate over policy evaluation and improvement
for k in range(num_policy_iter):
    J_new = policy_evaluation(T, Q, J, u[k])
    J = J_new
    u[k+1] = policy_improvement(T, Q, J_new, gamma, env_shape)
    visualize(J_new, u[k+1],obstacles_list, goal_idx_list,k)
```

Code Listing 1: Policy Iteration Code

1.1 (b) Initialize policy iteration with a feedback control

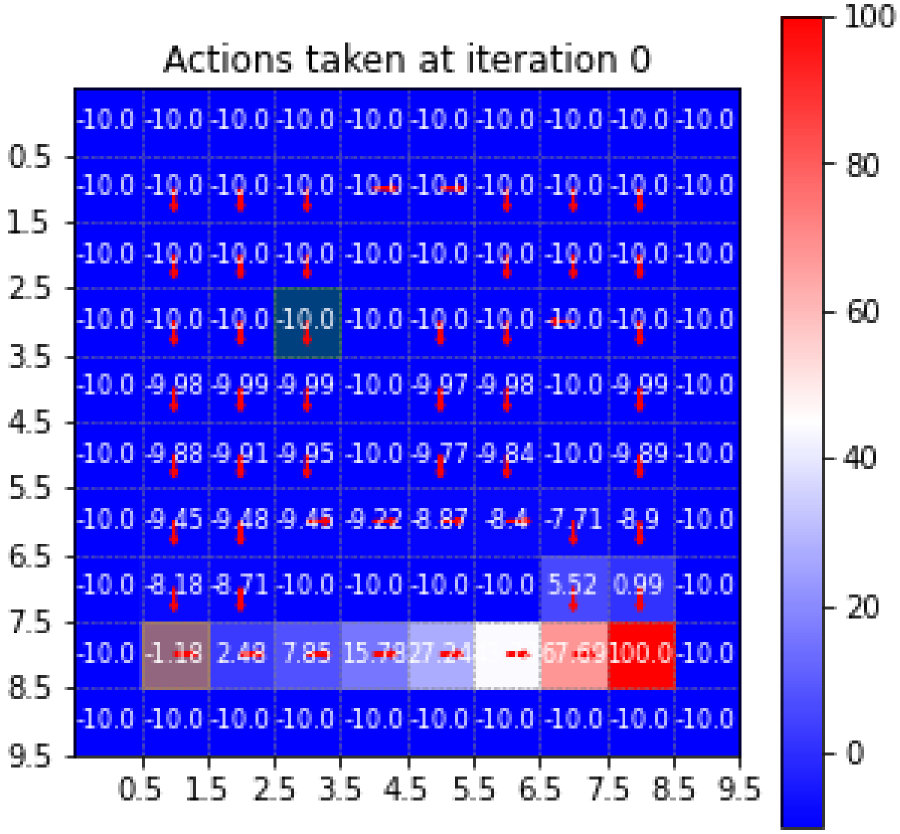

Figure 1: Value function J pi(0) (x) as a heatmap

1.2 (c) Execute the policy iteration algorithm

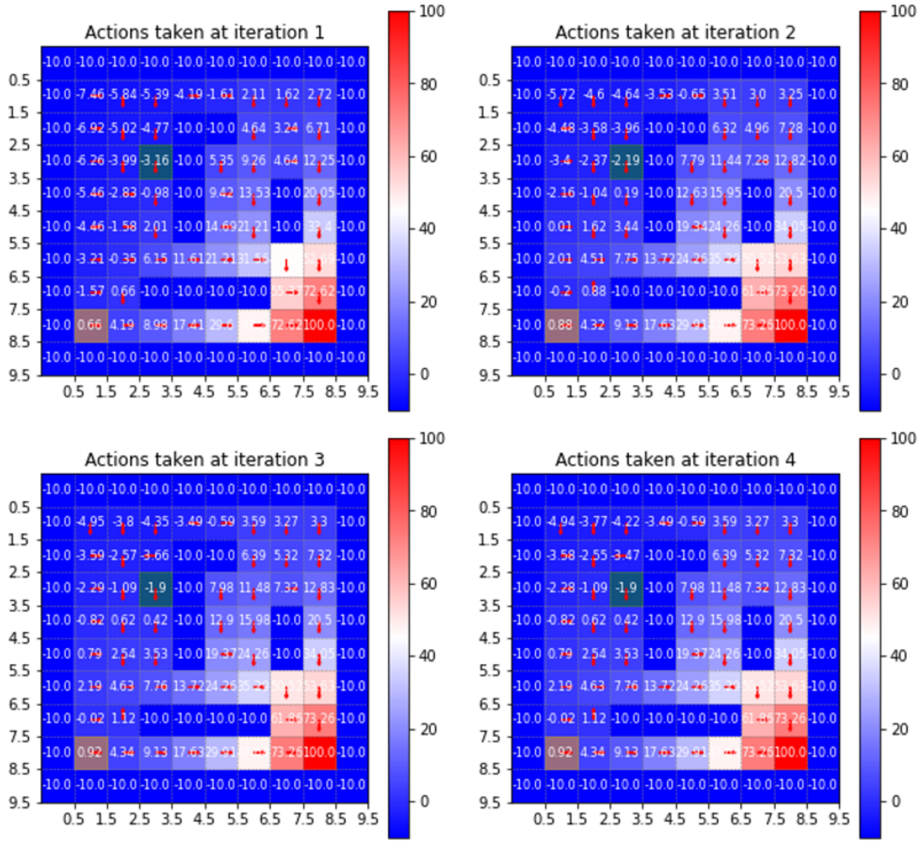

Figure 2: Plot the feedback control for first four iteration

2 Problem 2

```python
import os, sys, pickle, math
from copy import deepcopy

from scipy import io
import numpy as np
import matplotlib.pyplot as plt

from load_data import load_lidar_data, load_joint_data, joint_name_to_index
from utils import *

import logging

logger = logging.getLogger()
logger.setLevel(os.environ.get("LOGLEVEL", "INFO"))


class map_t:
    """
    This will maintain the occupancy grid and log_odds.
    You do not need to change anything in the initialization
    """

    def __init__(s, resolution=0.05):
        s.resolution = resolution
        s.xmin, s.xmax = -20, 20
        s.ymin, s.ymax = -20, 20
        s.szx = int(np.ceil((s.xmax - s.xmin) / s.resolution + 1))
        s.szy = int(np.ceil((s.ymax - s.ymin) / s.resolution + 1))

        # binarized map and log-odds
        s.cells = np.zeros((s.szx, s.szy), dtype=np.int8)  # initialize the map as empty
        s.log_odds = np.zeros(s.cells.shape, dtype=np.float64)

        s.log_odds_max = 5e6
        # number of observations received yet for each cell
        s.num_obs_per_cell = np.zeros(s.cells.shape, dtype=np.uint64)

        # we call a cell occupied if the probability of
        # occupancy P(m_i | ... ) is >= occupied_prob_thresh
        s.occupied_prob_thresh = 0.6
        s.log_odds_thresh = np.log(s.occupied_prob_thresh / (1 - s.occupied_prob_thresh))

    def grid_cell_from_xy(s, x, y):
        """
        x and y are 1-dimensional arrays, compute the cell indices in the map corresponding to these (x,y) locations.
        You should return an array of shape (2 x len(x)).
        Be careful to handle instances when x/y go outside the map bounds,
        you can use np.clip to handle these situations.
        """
        #### TODO: Checked
        x_indices = np.clip((x - s.xmin) / s.resolution, 0, s.szx - 1).astype(int)
        y_indices = np.clip((y - s.ymin) / s.resolution, 0, s.szy - 1).astype(int)

        # Stack the x_indices and y_indices vertically to create a 2D array where each column represents
        return np.vstack((x_indices, y_indices))


class slam_t:
    """
    s is the same as s. In Python it does not really matter
```

```python
    what we call s, s is shorter. As a general comment, (I believe)
    you will have fewer bugs while writing scientific code if you
    use the same/similar variable names as those in the mathematical equations.
    """

    def __init__(s, resolution=0.05, Q=1e-3 * np.eye(3), resampling_threshold=0.3):

        s.init_sensor_model()

        # dynamics noise for the state (x,y,yaw)
        s.Q = Q
        # s.Q = 1e-8 * np.eye(3)

        # we resample particles if the effective number of particles
        # falls below s.resampling_threshold*num_particles
        s.resampling_threshold = resampling_threshold

        # initialize the map
        s.map = map_t(resolution)

    def read_data(s, src_dir, idx=0, split='train'):
        """
        src_dir: location of the "data" directory
        """
        logging.info('> Reading data')
        s.idx = idx
        s.lidar = load_lidar_data(os.path.join(src_dir, 'data/%s/%s_lidar%d' % (split, split, idx)))
        s.joint = load_joint_data(os.path.join(src_dir, 'data/%s/%s_joint%d' % (split, split, idx)))

        # finds the closets idx in the joint timestamp array such that the timestamp
        # at that idx is t
        s.find_joint_t_idx_from_lidar = lambda t: np.argmin(np.abs(s.joint['t'] - t))

    def init_sensor_model(s):
        # lidar height from the ground in meters
        s.head_height = 0.93 + 0.33
        s.lidar_height = 0.15

        s.lidar_dmin = 1e-3
        s.lidar_dmax = 30
        s.lidar_angular_resolution = 0.25  # degrees
        s.lidar_angles = np.arange(-135, 135 + s.lidar_angular_resolution,
                                     s.lidar_angular_resolution) * np.pi / 180.0
        # Sensor model
        s.lidar_log_odds_occ = np.log(9)
        s.lidar_log_odds_free = np.log(1 / 9.)

    def init_particles(s, n=100, p=None, w=None, t0=0):
        """
        n: number of particles
        p: xy yaw locations of particles (3xn array)
        w: weights (array of length n)
        """
        s.n = n
        s.p = deepcopy(p) if p is not None else np.zeros((3, s.n), dtype=np.float64)
        s.w = deepcopy(w) if w is not None else np.ones(n) / float(s.n)  # 1/n

    @staticmethod
    def stratified_resampling(p, w):
        """
        resampling step of the particle filter, takes p = 3 x n array of
        particles with w = 1 x n array of weights and returns new particle
        locations (number of particles n remains the same) and their weights
```

```
        Parameters:
        - p: (3 x n) numpy array of particle states.
        - w: (1 x n) numpy array of particle weights.

        Returns:
        - Tuple of (resampled_particles, uniform_weights):
          - resampled_particles is a (3 x n) numpy array after resampling.
          - uniform_weights is a (1 x n) numpy array of equal weights for all particles.
        """
        #### TODO: Checked
        n = len(w)  # Total number of particles
        # Adjust weights and repeat particles based on adjusted weights
        adjusted_weights = (w * n * 10).astype(int)
        repeated_particles = np.repeat(p, adjusted_weights, axis=1)

        # Select a subset of particles randomly
        indexes = np.random.choice(repeated_particles.shape[1], n, replace=False)
        resampled_particles = repeated_particles[:, indexes]

        # Assign equal weight to all resampled particles
        uniform_weights = np.ones(n) / n

        return resampled_particles, uniform_weights


    @staticmethod
    def log_sum_exp(w):
        return w.max() + np.log(np.exp(w - w.max()).sum())

    def rays2world(s, p, d, head_angle=0, neck_angle=0, angles=None):
        """
        p: p is the pose of the particle (x,y,yaw), a 3x1 array describing the robot position and
    orientation
        d: an array that stores the distance along the ray of the lidar for each ray
           the length of d has to be equal to that of angles, this is s.lidar[t]['scan']
        head_angle: the angle of the head in the body frame, usually 0, need to be in radians
        neck_angle: the angle of the neck in the body frame, usually 0, need to be in radians
        angles: angle of each ray in the body frame in radians
                (usually be simply s.lidar_angles for the different lidar rays)

        Return an array (2 x num_rays) which are the (x,y) locations of the end point of each ray
    in world coordinates
        """
        # Filter valid LiDAR points based on distance constraints
        in_range = np.logical_and(d >= s.lidar_dmin, d <= s.lidar_dmax)
        d_filtered = d[in_range]
        angle_filtered = angles[in_range]

        # Transform distances to LiDAR frame points (2D to 3D)
        lidar_pts = np.vstack((d_filtered * np.cos(angle_filtered), d_filtered * np.sin(
    angle_filtered), np.zeros(d_filtered.size)))

        # Transformation from LiDAR to body frame
        lidar_to_body_tf = euler_to_se3(0, head_angle, neck_angle, np.array([0, 0, s.lidar_height])
    )
        body_pts_4d = lidar_to_body_tf @ make_homogeneous_coords_3d(lidar_pts)  # Transform to 4D
    for matrix multiplication

        # Transform from body frame to world frame
        body_to_world_tf = euler_to_se3(0, 0, p[2, 0], np.array([p[0, 0], p[1, 0], s.head_height]))
        world_pts_4d = body_to_world_tf @ body_pts_4d  # Apply transformation

        # Normalize and return 2D world frame points
        world_pts_2d = world_pts_4d[:3] / world_pts_4d[3]
```

```
        return world_pts_2d[:2]

    def get_control(s, t):
        """
        Use the pose at time t and t-1 to calculate what control the robot could have taken
        at time t-1 at state (x,y,th)_{t-1} to come to the current state (x,y,th)_t. We will
        assume that this is the same control that the robot will take in the function dynamics_step
        below at time t, to go to time t-1. need to use the smart_minus_2d function to get the difference of the two poses and we will simply set this to be the control (delta x, delta y, delta theta)
        """

        """

        Parameters:
        - t: The current time step (index) in the LIDAR data sequence.

        Returns:
        - control: The computed control signal as a difference in x, y coordinates,
                   and heading angle (theta),
                   indicating how to adjust the pose from time step t-1 to t.
        """

        if t == 0:
            return np.zeros(3)

        #### TODO: Checked

        # Compute the difference in pose between time t and t-1
        # Extract the previous and current poses from LIDAR data using the given time step t.
        previous_pose = s.lidar[t - 1]['xyth']
        current_pose = s.lidar[t]['xyth']

       # Compute the control signal as the difference between current and previous poses.
        return smart_minus_2d(current_pose, previous_pose)

    def dynamics_step(s, t):
        """"
        Compute the control using get_control and perform that control on each particle to get the updated locations of the particles in the particle filter, remember to add noise using the smart_plus_2d function to each particle
        """

        """
        Parameters:
            - control: The control signal to be applied, typically the difference in pose.
        """
        #### TODO: Checked
        control = s.get_control(t)

        # Generate noise for all particles at once
        noise = np.random.multivariate_normal(np.zeros(s.Q.shape[0]), s.Q, s.n)

        # Apply the noisy control to each particle
        for i in range(s.n):
            noisy_control = control + noise[i]
            s.p[:, i] = smart_plus_2d(s.p[:, i].copy(), noisy_control)

    @staticmethod
    def update_weights(w, obs_logp):
        """
        Given the observation log-probability and the weights of particles w, calculate the
          new weights as discussed in the writeup. Make sure that the new weights are normalized
        """
        # Parse the observation log-probability
```

```
        w = obs_logp + np.log(w)
        w -= slam_t.log_sum_exp(w)
        w = np.exp(w)
        return w

    def observation_step(s, t):
        """
        This function does the following things
            1. updates the particles using the LiDAR observations
            2. updates map.log_odds and map.cells using occupied cells as shown by the LiDAR data

        Some notes about how to implement this.
            1. As mentioned in the writeup, for each particle
                (a) First find the head, neck angle at t (this is the same for every particle)
                (b) Project lidar scan into the world frame (different for different particles)
                (c) Calculate which cells are obstacles according to this particle for this scan,
                calculate the observation log-probability
            2. Update the particle weights using observation log-probability
            3. Find the particle with the largest weight, and use its occupied cells to update the map.log_odds and map.cells.
        You should ensure that map.cells is recalculated at each iteration (it is simply the binarized version of log_odds). map.log_odds is of course maintained across iterations.
        """
        #### TODO: checked
        # Extract head and neck angles
        idx = s.find_joint_t_idx_from_lidar(s.lidar[t]['t'])
        angle_neck = s.joint['head_angles'][0, idx]
        angle_head = s.joint['head_angles'][1, idx]

        # Initialize observation probabilities
        log_prob_obs = np.zeros(s.n)

        for i in range(s.n):
            # Project lidar scan ---> world frame
            p = s.p[:, i].reshape((3, 1))
            world_frame_points = s.rays2world(p, s.lidar[t]['scan'], angle_head, angle_neck, s.lidar_angles)

            # grid cell indices of the occupied cells && observation log-probability
            occupied_cells = s.map.grid_cell_from_xy(world_frame_points[0], world_frame_points[1])
            log_prob_obs[i] = np.sum(s.map.log_odds[occupied_cells[0], occupied_cells[1]])

        # Update particle weights and estimate the pose from the best particle
        s.w = s.update_weights(s.w, log_prob_obs)
        best_idx = np.argmax(s.w)
        s.estimated_pose = s.p[:, best_idx]

        # Update the map based on the best particle's observation
        best_particle_world = s.rays2world(s.estimated_pose.reshape((3, 1)), s.lidar[t]['scan'], angle_head, angle_neck, s.lidar_angles)
        occupied_x, occupied_y = s.map.grid_cell_from_xy(best_particle_world[0], best_particle_world[1])

        # Compute and update free cells from best particle to observed obstacles
        limits_x, limits_y = s.calculate_free_space(s.estimated_pose, occupied_x, occupied_y)
        s.update_map(occupied_x, occupied_y, limits_x, limits_y)

        # Resample particles
        s.resample_particles()

    def calculate_free_space(s, pose, occupied_x, occupied_y):
        """
        Calculate free space coordinates based on the current pose and observed occupied cells.
        """
        # Compute limits based on lidar maximum distance and current pose
```

```
        limit_x = np.array([pose[0] - s.lidar_dmax / 2, pose[0] + s.lidar_dmax / 2, pose[0]])
        limit_y = np.array([pose[1] - s.lidar_dmax / 2, pose[1] + s.lidar_dmax / 2, pose[1]])
        limit_grid_x, limit_grid_y = s.map.grid_cell_from_xy(limit_x, limit_y)

        # Determine free cells
        free_x = np.linspace(limit_grid_x[2], occupied_x, endpoint=False).astype(int).flatten()
        free_y = np.linspace(limit_grid_y[2], occupied_y, endpoint=False).astype(int).flatten()

        return free_x, free_y


    def update_map(s, occupied_x, occupied_y, free_x, free_y):
        """
        Update SLAM map log-odds values based on observed and free cells, then binarize the map.
        """
        # Update log-odds for occupied and free cells
        s.map.log_odds[occupied_x, occupied_y] += s.lidar_log_odds_occ
        s.map.log_odds[free_x, free_y] += s.lidar_log_odds_free
        s.map.log_odds = np.clip(s.map.log_odds, -s.map.log_odds_max, s.map.log_odds_max)

        # Binarize the map based on log-odds threshold
        s.map.cells = (s.map.log_odds > s.map.log_odds_thresh).astype(int)

    def resample_particles(s):
        """
        Resampling is a (necessary) but problematic step which introduces a lot of variance in the particles.
        We should resample only if the effective number of particles falls below
          a certain threshold (resampling_threshold).
        A good heuristic to calculate the effective particles is 1/(sum_i w_i^2) where w_i are the weights
          of the particles, if this number of close to n, then all particles have about equal weights,
          and we do not need to resample
        """
        e = 1 / np.sum(s.w ** 2)
        logging.debug('> Effective number of particles: {}'.format(e))
        if e / s.n < s.resampling_threshold:
            s.p, s.w = s.stratified_resampling(s.p, s.w)
            logging.debug('> Resampling')
```

Code Listing 2: SLAM Code

```

# Pratik Chaudhari (pratikac@seas.upenn.edu)
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"

import click, tqdm, random
import numpy as np
from slam import *

def run_dynamics_step(src_dir, log_dir, idx, split, t0=0, draw_fig=False):
    """
    This function is for you to test your dynamics update step. It will create
    two figures after you run it. The first one is the robot location trajectory
    using odometry information obtained form the lidar. The second is the trajectory
    using the PF with a very small dynamics noise. The two figures should look similar.
    """
    slam = slam_t(Q=1e-8*np.eye(3))
    slam.read_data(src_dir, idx, split)

    # trajectory using odometry (xy and yaw) in the lidar data
    d = slam.lidar
    xyth = []
    for p in d:
        xyth.append([p['xyth'][0], p['xyth'][1],p['xyth'][2]])
    xyth = np.array(xyth)

    plt.figure(1); plt.clf();
    plt.title('Trajectory using onboard odometry')
    plt.plot(xyth[:,0], xyth[:,1])
    logging.info('> Saving odometry plot in '+os.path.join(log_dir, 'odometry_%s_%02d.jpg'%(split,
    idx)))
    plt.savefig(os.path.join(log_dir, 'odometry_%s_%02d.jpg'%(split, idx)))

    # dynamics propagation using particle filter
    # n: number of particles, w: weights, p: particles (3 dimensions, n particles)
    # S covariance of the xyth location
    # particles are initialized at the first xyth given by the lidar
    # for checking in this function
    n = 3
    w = np.ones(n)/float(n)
    p = np.zeros((3,n), dtype=np.float64)
    slam.init_particles(n,p,w)
    slam.p[:,0] = deepcopy(slam.lidar[0]['xyth'])

    print('> Running prediction')
    t0 = 0
    T = len(d)
    ps = deepcopy(slam.p)   # maintains all particles across all time steps
    plt.figure(2); plt.clf();
    ax = plt.subplot(111)

    for t in tqdm.tqdm(range(t0+1,T)):
        slam.dynamics_step(t)
        ps = np.hstack((ps, slam.p))

        if draw_fig:
            ax.clear()
            ax.plot(slam.p[0], slam.p[0], '*r')
            plt.title('Particles %03d'%t)
            plt.draw()
            plt.pause(0.01)

    plt.plot(ps[0], ps[1], '*c')
    plt.title('Trajectory using PF')
```

```python
    logging.info('> Saving plot in '+os.path.join(log_dir, 'dynamics_only_%s_%02d.jpg'%(split, idx)
    ))
    plt.savefig(os.path.join(log_dir, 'dynamics_only_%s_%02d.jpg'%(split, idx)))

def run_observation_step(src_dir, log_dir, idx, split, is_online=False):
    """
    This function is for you to debug your observation update step
    It will create three particles np.array([[0.2, 2, 3],[0.4, 2, 5],[0.1, 2.7, 4]])
    * Note that the particle array has the shape 3 x num_particles so
    the first particle is at [x=0.2, y=0.4, z=0.1]
    This function will build the first map and update the 3 particles for one time step.
    After running this function, you should get that the weight of the second particle is the
    largest since it is the closest to the origin [0, 0, 0]
    """
    slam = slam_t(resolution=0.05)
    slam.read_data(src_dir, idx, split)

    # t=0 sets up the map using the yaw of the lidar, do not use yaw for
    # other timestep
    # initialize the particles at the location of the lidar so that we have some
    # occupied cells in the map to calculate the observation update in the next step
    t0 = 0
    xyth = slam.lidar[t0]['xyth']
    xyth[2] = slam.lidar[t0]['rpy'][2]
    logging.debug('> Initializing 1 particle at: {}'.format(xyth))
    slam.init_particles(n=1,p=xyth.reshape((3,1)),w=np.array([1]))

    slam.observation_step(t=0)
    logging.info('> Particles\n: {}'.format(slam.p))
    logging.info('> Weights: {}'.format(slam.w))

    # reinitialize particles, this is the real test
    logging.info('\n')
    n = 3
    w = np.ones(n)/float(n)
    p = np.array([[2, 0.2, 3],[2, 0.4, 5],[2.7, 0.1, 4]])
    slam.init_particles(n, p, w)

    slam.observation_step(t=1)
    logging.info('> Particles\n: {}'.format(slam.p))
    logging.info('> Weights: {}'.format(slam.w))

def run_slam(src_dir, log_dir, idx, split):
    """
    This function runs slam. We will initialize the slam just like the observation_step
    before taking dynamics and observation updates one by one. You should initialize
    the slam with n=100 particles, you will also have to change the dynamics noise to
    be something larger than the very small value we picked in run_dynamics_step function
    above.
    """
    slam = slam_t(resolution=0.05, Q=np.diag([2e-4,2e-4,1e-4]))
    slam.read_data(src_dir, idx, split)
    T = len(slam.lidar)

    # again initialize the map to enable calculation of the observation logp in
    # future steps, this time we want to be more careful and initialize with the
    # correct lidar scan. First find the time t0 around which we have both LiDAR
    # data and joint data

    #### TODO: Checked

    # initialize the occupancy grid using one particle and calling the observation_step
    # function

    # Args:
```

```python
    # - src_dir: Source directory containing the dataset.
    # - log_dir: Directory to save the output map visualization.
    # - idx: Index of the dataset to process.
    # - split: Dataset split (e.g., 'train', 'test') to use.

    #### TODO: Checked

    # Initialize SLAM with specific noise parameters and resolution
    slam = slam_t(resolution=0.05, Q=np.diag([2e-4, 2e-4, 1e-4]))
    slam.read_data(src_dir, idx, split)
    total_steps = len(slam.lidar)

    # Initialize occupancy grid with one particle at the first lidar position
    init_pose = np.array(slam.lidar[0]['xyth'])
    init_pose[2] = slam.lidar[0]['rpy'][2]  # Use the yaw from 'rpy'
    slam.init_particles(n=1, p=init_pose.reshape((3, 1)), w=np.array([1.0]))
    slam.observation_step(t=0)

    # Log initialization info
    logging.info(f'> Initializing particles at first timestamp: {slam.lidar[0]["t"]}')

    # Re-initialize SLAM with 100 particles
    slam.init_particles(n=100)
    slam.dynamics_step(0)

    # Process dynamics and observations for each time step
    particle_positions = []
    for t in tqdm.tqdm(range(1, total_steps)):
        slam.dynamics_step(t)
        slam.observation_step(t)
        particle_positions.append(slam.estimated_pose)

    # Convert list of particle positions to a numpy array
    particle_positions = np.array(particle_positions)

    # Plot and save the map and particle trajectories
    fig, ax = plt.subplots(figsize=(10, 10))
    occupied_x, occupied_y = np.where(slam.map.cells == 1)                    # Plot occupied cells
    ax.plot(occupied_x, occupied_y, 'sk', markersize=1, label='Occupied')     # Plot particles
    particle_x, particle_y = slam.map.grid_cell_from_xy(particle_positions[:, 0], particle_positions[:, 1])
    ax.plot(particle_x, particle_y, '.r', markersize=5, label='Particles')
    ax.grid(True)
    ax.legend()
    ax.set(xlim=(0, slam.map.szx), ylim=(0, slam.map.szy), title=f'Map {idx}')
    ax.set_xlabel('X Coordinate')
    ax.set_ylabel('Y Coordinate')
    plt.show()
    fig.savefig(os.path.join(log_dir, f'Map{idx}_{split}.png'))
    plt.close(fig)


@click.command()
@click.option('--src_dir', default='./', help='data directory', type=str)
@click.option('--log_dir', default='logs', help='directory to save logs', type=str)
@click.option('--idx', default='0', help='dataset number', type=int)
@click.option('--split', default='train', help='train/test split', type=str)
@click.option('--mode', default='slam',
              help='choices: dynamics OR observation OR slam', type=str)
def main(src_dir, log_dir, idx, split, mode):
    # Run python main.py --help to see how to provide command line arguments

    if not mode in ['slam', 'dynamics', 'observation']:
        raise ValueError('Unknown argument --mode %s'%mode)
        sys.exit(1)
```

```python

    np.random.seed(42)
    random.seed(42)

    if mode == 'dynamics':
        run_dynamics_step(src_dir, log_dir, idx, split)
        sys.exit(0)
    elif mode == 'observation':
        run_observation_step(src_dir, log_dir, idx, split)
        sys.exit(0)
    else:
        p = run_slam(src_dir, log_dir, idx, split)
        return p

if __name__=='__main__':
    src_dir = './'
    log_dir = 'logs'
    idx = 3
    split = 'train'

    run_dynamics_step(src_dir, log_dir, idx, split)
    run_observation_step(src_dir, log_dir, idx, split)
    run_dynamics_step(src_dir, log_dir, idx, split)
    run_slam(src_dir, log_dir, idx, split)
    main()
```

Code Listing 3: Simultaneous Localization and Mapping (SLAM) with a particle filter main code

2.1 (c) dynamics step

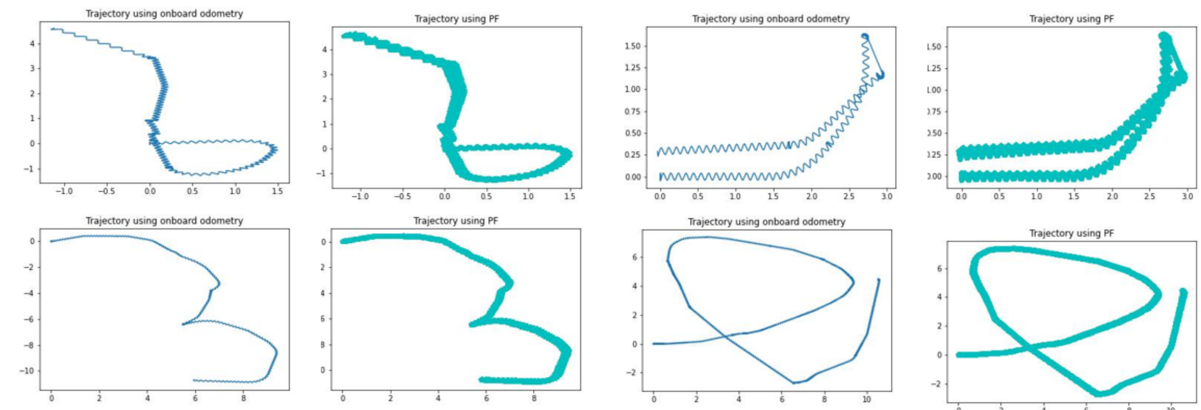

Figure 3: Odometry trajectory particle trajectories

2.2 (f) The full SLAM algorithm

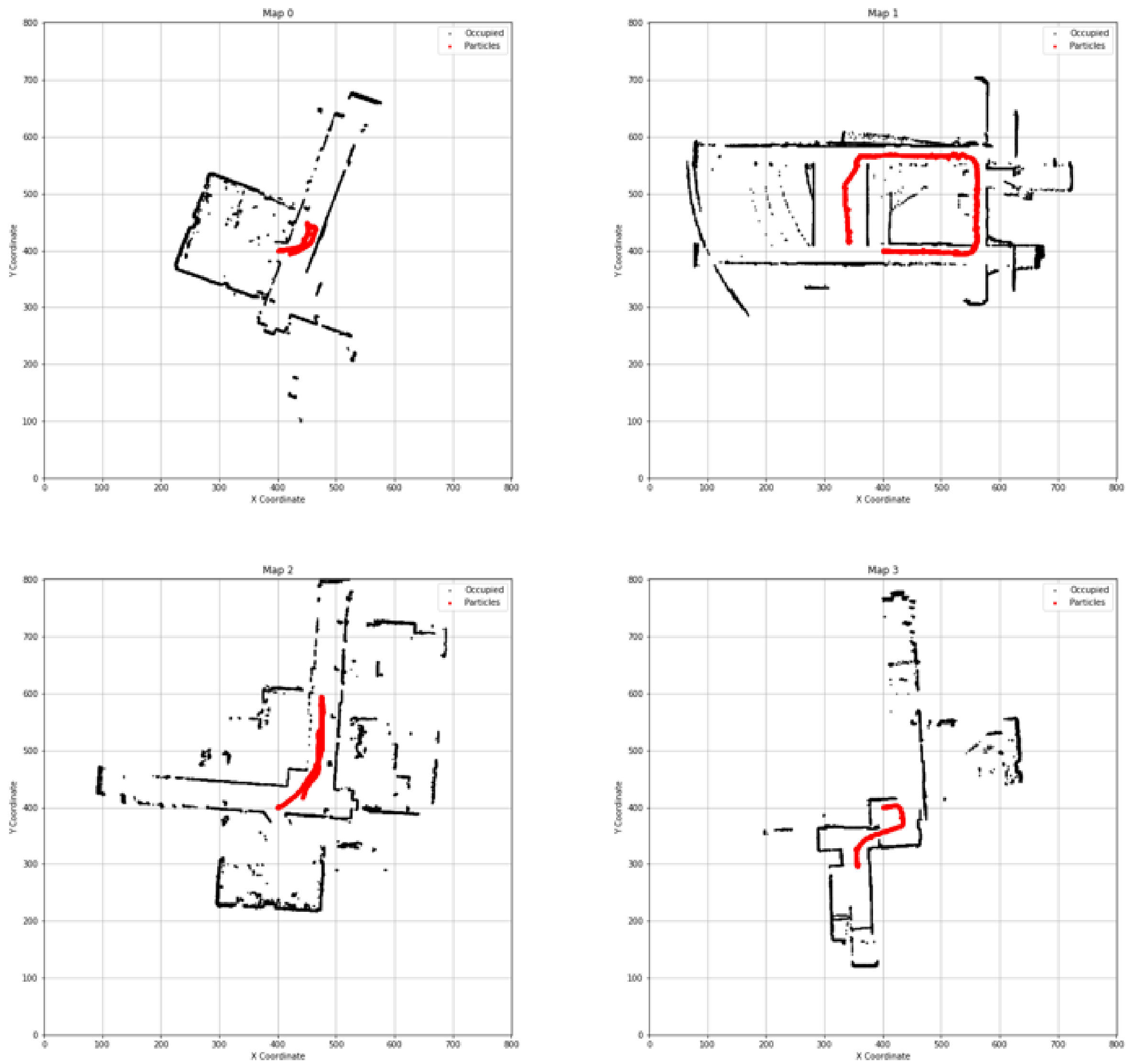

Figure 4: The final binarized version of the map

3 Problem 3

3.1 (a) Data Loading and COLMAP

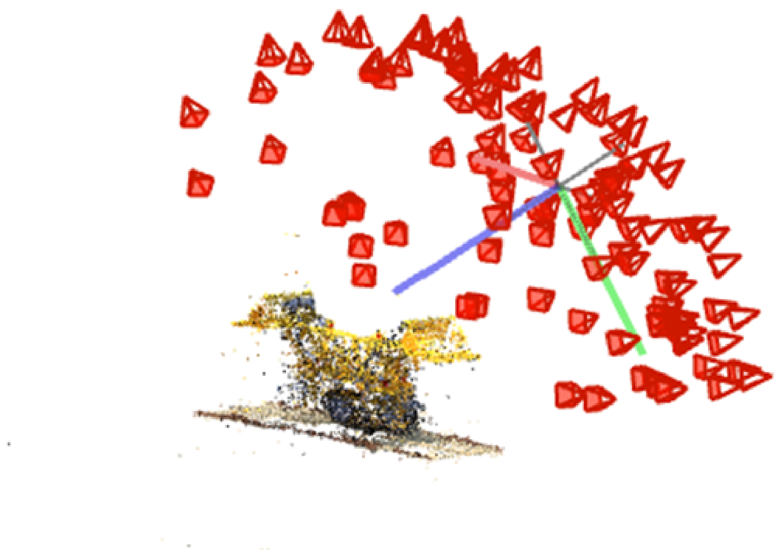

Figure 5: Data Loading and COLMAP

$$H = 100 \tag{1}$$

$$w = 100 \tag{2}$$

$$w = 95.8 \tag{3}$$

```
def load_colmap_data():
    r"""
    After using colmap2nerf.py to convert the colmap intrinsics and extrinsics,
    read in the transform_colmap.json file

    Expected Returns:
      An array of resized imgs, normalized to [0, 1]
      An array of poses, essentially the transform matrix
      Camera parameters: H, W, focal length

    NOTES:
      We recommend you resize the original images from 800x800 to lower resolution,
      i.e. 200x200 so it's easier for training. Change camera parameters accordingly
    """
    ################### YOUR CODE START ####################

    json_path='transforms_colmap.json'
    image_dir='./data/data/images'
    resize_dim=(100, 100)

```

```
    # Load JSON file
    with open(json_path, ’r’) as f:
        data = json.load(f)

    # Initialize lists to store processed data
    imgs = []
    poses = []

    # Process each frame in the JSON file
    for frame in data[’frames’]:
        # Load and resize image
        img_path = os.path.join(image_dir, frame[’file_path’][0])
        img = Image.open(img_path).resize(resize_dim)
        imgs.append(np.array(img) / 255.0)  # Normalize to [0, 1]

        # Process pose
        pose = np.array(frame[’transform_matrix’])
        #print (pose)
        poses.append(pose)

    # Assuming all images have the same size and focal length
    H, W = resize_dim
    focal = frame["camera_angle_x"] * W / (2 * np.tan(frame[’camera_angle_x’] / 2))

    # Convert lists to numpy arrays
    imgs = np.array(imgs)
    poses = np.array(poses)

    # Camera parameters: Height, Width, Focal length
    camera_params = (H, W, focal)

    return imgs, poses, camera_params
```

Code Listing 4: Calculate Camera Parameters

3.2 (b) Implementation of the NeRF

```python
# -*- coding: utf-8 -*-
"""Zhanqian Wu ESE 650 HW3 P3 Nerf

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/19y_9R6bB_r93PXfga6C_vNJZMRBfSzbU
"""

from typing import Optional, Tuple
import torch
from torch import nn
from torch.nn import functional as F
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
import os
import json
#import imageio
import cv2
import os
os.environ['PYTORCH_CUDA_ALLOC_CONF'] = 'expandable_segments:True'
from PIL import Image

from google.colab import drive
drive.mount('/content/drive')

def load_colmap_data():
    r"""
    After using colmap2nerf.py to convert the colmap intrinsics and extrinsics,
    read in the transform_colmap.json file

    Expected Returns:
      An array of resized imgs, normalized to [0, 1]
      An array of poses, essentially the transform matrix
      Camera parameters: H, W, focal length

    NOTES:
      We recommend you resize the original images from 800x800 to lower resolution,
      i.e. 200x200 so it's easier for training. Change camera parameters accordingly
    """
    ################### YOUR CODE START ###################

    json_path='transforms_colmap.json'
    image_dir='./data/data/images'

    json_path='/content/drive/MyDrive/ESE 650_HW3_P3/transforms_colmap.json'
    image_dir='/content/drive/MyDrive/ESE 650_HW3_P3/data/data/images'
    resize_dim=(200, 200)

    # Load JSON file
    with open(json_path, 'r') as f:
        data = json.load(f)

    # Initialize lists to store processed data
    imgs = []
    poses = []

    # Process each frame in the JSON file
    for frame in data['frames']:
        # Load and resize image
        img_path = os.path.join(image_dir, frame['file_path'][0])
```

```python
        img = Image.open(img_path).resize(resize_dim)
        imgs.append(np.array(img) / 255.0)  # Normalize to [0, 1]

        # Process pose
        pose = np.array(frame['transform_matrix'])
        #print (pose)
        poses.append(pose)

    # Assuming all images have the same size and focal length
    H, W = resize_dim
    focal = frame["camera_angle_x"] * W / (2 * np.tan(frame['camera_angle_x'] / 2))

    # Convert lists to numpy arrays
    imgs = np.array(imgs)
    poses = np.array(poses)

    # Camera parameters: Height, Width, Focal length
    camera_params = (H, W, focal)

    return imgs, poses, camera_params
    ################### YOUR CODE END ###################


def get_rays(H, W, focal, c2w,):
    """
    Compute rays passing through each pixel in the world frame.

    Parameters:
    - H: Height of the image.
    - W: Width of the image.
    - focal: Camera intrinsic matrix of shape (3, 3).
    - c2w: Camera-to-world transformation matrix of shape (4, 4).

    Returns:
    - ray_origins: Array of shape (H, W, 3) denoting the origins of each ray.
    - ray_directions: Array of shape (H, W, 3) denoting the direction of each ray.
    """
    # Generate mesh grid for pixel coordinates

    # Detect if a GPU is available and choose the device accordingly
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    #device = 'cpu' #Due to OOM
    #print(device)

    # Ensure c2w is a torch.Tensor and move it to the chosen device
    if not isinstance(c2w, torch.Tensor):
        c2w = torch.from_numpy(c2w).float()  # Convert from numpy to tensor if necessary
    c2w = c2w.to(device)

    # Generate a grid of (i, j) coordinates
    i, j = torch.meshgrid(torch.linspace(0, W-1, W, device=device), torch.linspace(0, H-1, H, device=device))
    i = i.t().flatten()
    j = j.t().flatten()

    # Normalize pixel coordinates (assuming the image center as origin)
    dirs = torch.stack([(i - W * 0.5) / focal, -(j - H * 0.5) / focal, -torch.ones_like(i, device=device)], -1)

    # Rotate ray directions from camera frame to the world frame
    rays_d = torch.sum(dirs[..., None, :] * c2w[:3, :3], axis=-1)

    # The origin of all rays is the camera position in the world frame
    rays_o = c2w[:3, -1].expand(rays_d.shape)

```

```python
    # Reshape rays_o and rays_d to [H, W, 3]
    rays_o = rays_o.view(H, W, 3)
    rays_d = rays_d.view(H, W, 3)

    return rays_o, rays_d

def sample_points_from_rays(ray_origins, ray_directions, snear, sfar, Nsample):
    """
    Sample 3D points along rays within the specified near and far bounds.

    Parameters:
    - ray_origins: Array of shape (H, W, 3) denoting the origins of each ray.
    - ray_directions: Array of shape (H, W, 3) denoting the direction of each ray.
    - snear: Scalar or array defining the near clipping distance for each ray.
    - sfar: Scalar or array defining the far clipping distance for each ray.
    - Nsample: Number of points to sample along each ray.

    Returns:
    - sampled_points: Array of shape (H, W, Nsample, 3) with sampled 3D points.
    - depth_values: Array of shape (H, W, Nsample) with depth values of sampled points.
    """

    # Make sure snear and sfar are tensors
    H, W, _ = ray_origins.shape
    device = ray_origins.device


    # Compute the depth values for each sample
    depth_values = torch.linspace(snear, sfar, Nsample, device=device)

    depth_values = depth_values.expand(H, W, Nsample)  # Make sure depth values have shape (H, W, Nsample)

    # Compute the 3D positions of each sample point along the rays
    sampled_points = ray_origins[..., None, :] + depth_values[..., :, None] * ray_directions[..., None, :]  # Correct shape: (H, W, Nsample, 3)

    return sampled_points, depth_values


def positional_encoding(x, max_freq_log2=10, include_input=True):

    """Apply positional encoding to the input. (Section 5.1 of original paper)
    We use positional encoding to map continuous input coordinates into a
    higher dimensional space to enable our MLP to more easily approximate a
    higher frequency function.

    Expected Returns:
      pos_out: positional encoding of the input tensor.
               (H*W*num_samples, (include_input + 2*freq) * 3)
    """
    frequencies = 2 ** torch.linspace(0, max_freq_log2, steps=max_freq_log2+1, device=x.device)
    # Create a list of frequencies, (sin(2^k * x), cos(2^k * x)) for k=0,...,max_freq_log2
    encodings = [torch.sin(x * freq) for freq in frequencies] + [torch.cos(x * freq) for freq in frequencies]
    # Stack all encodings along the last dimension
    encoded = torch.cat(encodings, dim=-1)

    if include_input:
        # Concatenate the original input with the encoded features
        pos_out = torch.cat([x, encoded], dim=-1)
    else:
        pos_out = encoded

    return pos_out
```

```python


def volume_rendering(
    radiance_field: torch.Tensor,
    ray_origins: torch.Tensor,
    depth_values: torch.Tensor
) -> Tuple[torch.Tensor]:
    """
    Differentiably renders a radiance field, given the origin of each ray in the bundle,
    and the sampled depth values along them.

    Args:
      radiance_field: Tensor containing RGB color and volume density at each query location,
                      shape (H, W, num_samples, 4).
      ray_origins: Origin of each ray, shape (H, W, 3).
      depth_values: Sampled depth values along each ray, shape (H, W, num_samples).

    Returns:
      rgb_map: Rendered RGB image, shape (H, W, 3).
    """
    # Extract sigma (density) and color from the radiance field
    sigma = torch.relu(radiance_field[..., 3])  # Extract volume density
    rgb = torch.sigmoid(radiance_field[..., :3])  # Extract RGB colors

    # Compute depth intervals
    dists = torch.cat([depth_values[..., 1:] - depth_values[..., :-1], torch.tensor([1e10], device=
    device).expand(depth_values[..., :1].shape)], dim=-1)
    alpha = 1.0 - torch.exp(-sigma * dists)
    weights = alpha * torch.cumprod(torch.cat([torch.ones_like(alpha[..., :1]), 1.0 - alpha + 1e
    -10], dim=-1), dim=-1)[..., :-1]

    rgb_map = torch.sum(weights[..., None] * rgb, dim=-2)

    return rgb_map

class TinyNeRF(torch.nn.Module):
    def __init__(self, pos_dim, fc_dim=128):
      r"""Initialize a tiny nerf network, which composed of linear layers and
      ReLU activation. More specifically: linear - relu - linear - relu - linear
      - relu -linear. The module is intentionally made small so that we could
      achieve reasonable training time

      Args:
        pos_dim: dimension of the positional encoding output
        fc_dim: dimension of the fully connected layer
      """
      super().__init__()

      self.nerf = nn.Sequential(
                    nn.Linear(pos_dim, fc_dim),
                    nn.ReLU(),
                    nn.Linear(fc_dim, fc_dim),
                    nn.ReLU(),
                    nn.Linear(fc_dim, fc_dim),
                    nn.ReLU(),
                    nn.Linear(fc_dim, 4)
                  )

    def forward(self, x):
      r"""Output volume density and RGB color (4 dimensions), given a set of
      positional encoded points sampled from the rays
      """
      x = self.nerf(x)
      return x

```

```python

def get_minibatches(inputs: torch.Tensor, chunksize: Optional[int] = 1024 * 8):
    r"""Takes a huge tensor (ray "bundle") and splits it into a list of minibatches.
    Each element of the list (except possibly the last) has dimension 0 of length
    chunksize.
    """
    return [inputs[i:i + chunksize] for i in range(0, inputs.shape[0], chunksize)]


def nerf_step_forward(height, width, focal_length, trans_matrix,
                            near_point, far_point, num_depth_samples_per_ray,
                            get_minibatches_function, model):
    r"""Perform one iteration of training, which take information of one of the
    training images, and try to predict its rgb values

    Args:
      height: height of the image
      width: width of the image
      focal_length: focal length of the camera
      trans_matrix: transformation matrix, which is also the camera pose
      near_point: threshhold of nearest point
      far_point: threshold of farthest point
      num_depth_samples_per_ray: number of sampled depth from each rays in the ray bundle
      get_minibatches_function: function to cut the ray bundles into several chunks
        to avoid out-of-memory issue

    Expected Returns:
      rgb_predicted: predicted rgb values of the training image
    """
    ################### YOUR CODE START ###################

    # Step 1: Generate rays
    # an implementation of get rays function that returns ray origins and directions
    ray_origins, ray_directions = get_rays(height, width, focal_length, trans_matrix)


    # Step 2: Sample points along each ray
    # an implementation of sample points from rays that returns sampled points and depth values
    sampled_points, depth_values = sample_points_from_rays(ray_origins, ray_directions, near_point, far_point, num_depth_samples_per_ray)


    # Step 3: Apply positional encoding
    # positional encoding expects a flattened list of points
    flattened_sampled_points = sampled_points.reshape(-1, 3)  # Flattening sampled points for positional encoding
    positional_encoded_points = positional_encoding(flattened_sampled_points)  # Apply positional encoding
    # print("positional_encoded_points",positional_encoded_points.shape)


    # Step 4: Run the model in batches
    # Splitting the points into manageable chunks to avoid OOM
    batches = get_minibatches_function(positional_encoded_points, chunksize=16384)
    predictions = []
    for batch in batches:
        #print(batch.shape)
        predictions.append(model(batch))

    radiance_field_flattened = torch.cat(predictions, dim=0)

    ################### YOUR CODE START ###################

    # Step 5: Volume rendering
    # Reshape the radiance field to its unflattened shape
```

```python
    radiance_field = radiance_field_flattened.view(height, width, num_depth_samples_per_ray, 4)
    #print(f"radiance_field is stored on: {radiance_field.device}")


    # volume rendering that takes the radiance field, ray origins, and depth values
    rgb_predicted = volume_rendering(radiance_field, ray_origins, depth_values)
    #print("rgb_predicted",rgb_predicted.shape)


    return rgb_predicted



def train(images, poses, hwf, near_point,
          far_point, num_depth_samples_per_ray,
          num_iters, model, History,DEVICE="cuda"):
    r"""Training a tiny nerf model

    Args:
      images: all the images extracted from dataset (including train, val, test)
      poses: poses of the camera, which are used as transformation matrix
      hwf: [height, width, focal_length]
      near_point: threshhold of nearest point
      far_point: threshold of farthest point
      num_depth_samples_per_ray: number of sampled depth from each rays in the ray bundle
      num_iters: number of training iterations
      model: predefined tiny NeRF model
    """


    H, W, focal_length = hwf
    H = int(H)
    W = int(W)
    n_train = images.shape[0]

    # Optimizer parameters
    lr = 5e-3
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    # Seed RNG, for repeatability
    seed = 9458
    torch.manual_seed(seed)
    np.random.seed(seed)


    for _ in tqdm(range(num_iters)):
      # Randomly pick a training image as the target, get rgb value and camera pose
      train_idx = np.random.randint(n_train)
      train_img_rgb = images[train_idx, ..., :3]
      train_pose = poses[train_idx]

      # Run one iteration of TinyNeRF and get the rendered RGB image.
      rgb_predicted = nerf_step_forward(H, W, focal_length,
                                        train_pose, near_point,
                                        far_point, num_depth_samples_per_ray,
                                        get_minibatches, model)


      train_img_rgb_tensor = torch.from_numpy(train_img_rgb)
      train_img_rgb= train_img_rgb_tensor.to(DEVICE).to(dtype=torch.float32)


      # Compute mean-squared error between the predicted and target images
      loss = torch.nn.functional.mse_loss(rgb_predicted, train_img_rgb)
      loss.backward()
```

```
      optimizer.step()
      optimizer.zero_grad()
      History.append(loss.cpu().detach().numpy())


      if _ % 100 == 0:
            with torch.no_grad():
                plt.figure(figsize=(10, 4))

                plt.subplot(121)
                plt.imshow(train_img_rgb_tensor.cpu().numpy())
                plt.title(f"Target Image at Iteration {_}")

                plt.subplot(122)
                plt.imshow(rgb_predicted.detach().cpu().numpy())
                plt.title(f"Predicted Image at Iteration {_}")

                plt.show()

                print('Loss at '+str(_)+" iterations: ",loss.cpu().detach().numpy())

    print('Finish training')

if __name__ == "__main__":
    images, poses, hwf=load_colmap_data()
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    #device = 'cpu' #Due to OOM
    near_point=2.
    far_point=6.
    num_depth_samples_per_ray = 64
    num_iters = 2000
    model = TinyNeRF(69)
    model.to(device)

    History=[]
    train(images, poses, hwf, near_point,
              far_point, num_depth_samples_per_ray,
              num_iters, model, History,DEVICE=device)

plt.plot(History)
plt.grid()
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("ESE 650 Zhanqian Nerf Training Process")
plt.show()
```

Code Listing 5: NERF Code

3.3 (c) Network Training

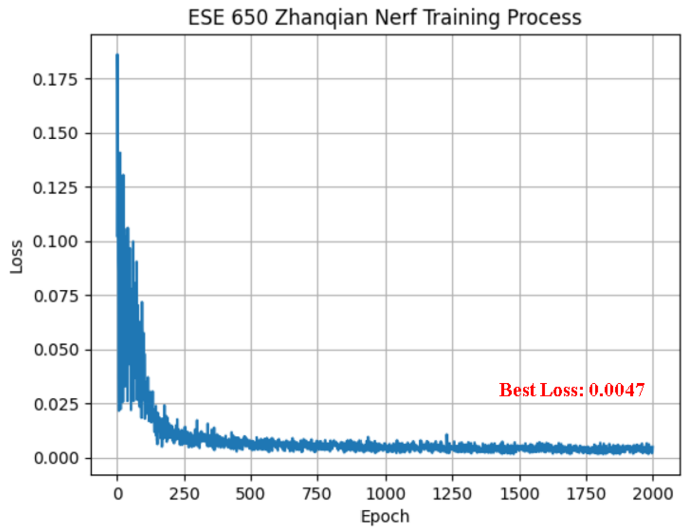

Figure 6: Nerf training loss graph

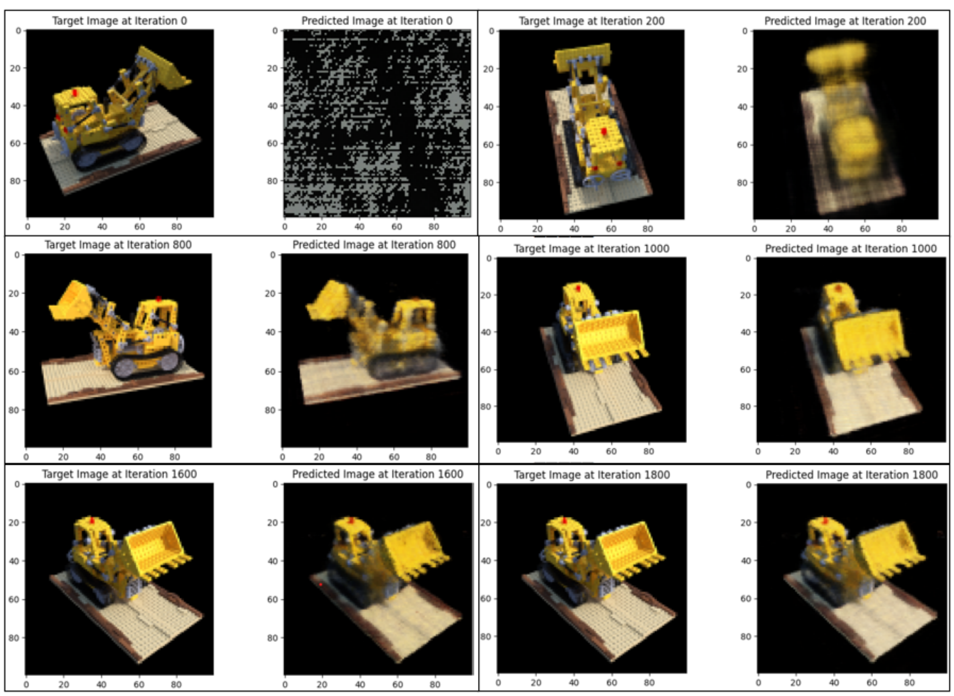

Figure 7: Comparison of the Results with the Number of Training Iterations

3.4 (d) Inference

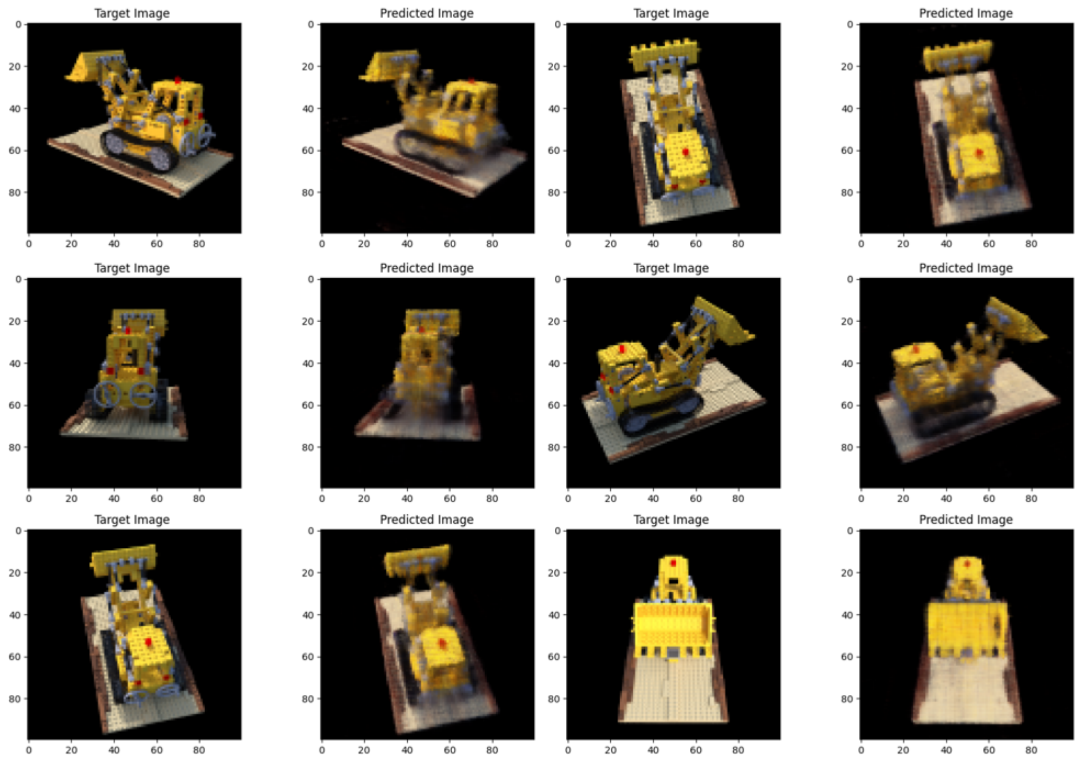

Figure 8: Five viewpoints and rendered RGB images

Where NeRF is Working Well:

- **Overall Shape Recognition:** In several predicted images, NeRF seems to capture the overall shape of the object quite well. The contours and the bulk of the object are recognizable.
- **Color Approximation:** The colors in the predicted images, while sometimes a bit off, are relatively close to the target images, indicating that NeRF is effectively capturing the color information.

Where NeRF is Struggling:

- **Fine Details:** NeRF appears to struggle with fine details. In the predicted images, areas with small or intricate details are blurred or misrepresented, such as the precise edges and small features of the object.
- **Texture Fidelity:** The textures in the predicted images are less crisp and detailed compared to the target images. This can be observed in the loss of texture clarity on surfaces.
- **Edges and Boundaries:** The edges of the objects in the predicted images are not as sharp as in the target images, showing some challenges in rendering sharp boundaries.
- **Consistent Lighting and Shadows:** Some of the predicted images appear to have slightly different lighting conditions and shadowing compared to the target images, suggesting difficulty in capturing the exact lighting of the scene.

In summary,Neural Radiance Fields often struggle with high-frequency details due to several factors inherent in their design and the training data used. High-frequency details refer to rapid changes in color or brightness in an image, which correspond to the fine details in a scene, like textures or edges.

References

[1] hmmlearn/hmmlearn. (2022, November 25). GitHub. `https://github.com/hmmlearn/hmmlearn`