UNIVERSITY OF PENNSYLVANIA

ESE 650: LEARNING IN ROBOTICS

[03/10] HOMEWORK 3

DUE: 04/01 MON 11.59 PM ET

Changelog: This space will be used to note down updates/errata to the homework problems.

Read the following instructions carefully before beginning to work on the homework.

- You will submit solutions typeset in LaTeX on Gradescope (strongly encouraged). You can use hw_template.tex on Canvas in the "Homeworks" folder to do so. If your handwriting is *unambiguously legible*, you can submit PDF scans/tablet-created PDFs.
- Please start a new problem on a fresh page and mark all the pages corresponding to each problem. Failure to do so may result in your work not graded completely.
- Clearly indicate the name and Penn email ID of all your collaborators on your submitted solutions.
- **For each problem in the homework, you should mention the total amount of time you spent on it. This helps us gauge the perceived difficulty of the problems.**
- You can be informal while typesetting the solutions, e.g., if you want to draw a picture feel free to draw it on paper clearly, click a picture and include it in your solution. Do not spend undue time on typesetting solutions.
- You will see an entry of the form "HW 3 PDF" where you will upload the PDF of your solutions. You will also see entries like "HW 3 Problem 1 Code" where you will upload your solution for the respective problems. **For each programming problem, you should create a fresh Python file**. This file should contain all the code to reproduce the results of the problem and you will upload the .py file to Gradescope. If we have installed Autograder for a particular problem, you will use the Autograder. Name your file to be

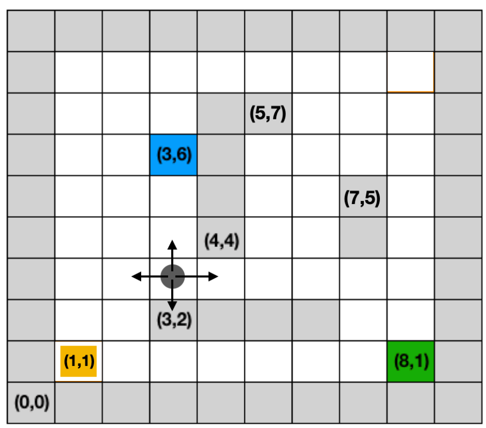
"pennkey_hw3_problem1.py", e.g., I will name my code for Problem 1 as "pratikac_hw3_problem1.py".

- **You should include all the relevant plots in the PDF, without doing so you will not get full credit. There is no auto-grader for this homework so this is particularly important.** You can, for instance, export your Jupyter notebook as a PDF (you can also use text cells to write your solutions) and export the same notebook as a Python file to upload your code.
- **Your PDF solutions should be completely self-contained. We will run the Python file to check if your solution reproduces the results in the PDF.**

Credit. The points for the problems add up to 120. You only need to solve for 100 points to get full credit, i.e., your final score will be min(your total points, 100).

Problem 1 (Policy Iteration, 20 points (No Autograder)). Consider the following Markov Decision Process. The state-space is a 10×10 grid, cells that are obstacles are marked in gray. The initial state of the robot is in blue and our desired terminal state is in green. The robot gets a *reward* of 10 if it reaches the desired terminal state with a discount factor of 0.9. At each non-obstacle cell, the robot can attempt to move to any of the immediate neighboring cells using one of the four controls (North, East, West and South). The robot cannot diagonally. The move succeeds with probability 0.7 and with remainder probability 0.3 the robot can end up at some other cell as follows:

$$
\begin{aligned}
\mathrm{P}(\text{moves north} \mid \text{control is north}) &= 0.7,\\
\mathrm{P}(\text{moves west} \mid \text{control is north}) &= 0.1,\\
\mathrm{P}(\text{moves east} \mid \text{control is north}) &= 0.1,\\
\mathrm{P}(\text{does not move} \mid \text{control is north}) &= 0.1.
\end{aligned}
$$



Similarly, if the robot desired to go east, it may end up in the cells to its north, south, or stay put at the original cell with total probability 0.3 and actually move to the cell east with probability 0.7. The cost pays a cost of 1 (i.e., reward is -1) for each control input it takes, regardless of the outcome. If the robot ends up at a state marked as an obstacle (all grey cells are obstacles, i.e., cell marked (0,0), (0,1), (3,2) etc. are obstacles), it gets a reward of -10 for each time-step that it remains inside the obstacle cell. The robot is allowed to stay in the goal state indefinitely (i.e., take a special action to "not move") and this action gets no reward/cost.

We would like to implement policy iteration to find the best trajectory for the robot to go from the blue cell to the green cell.

(a) **(0 points)** Carefully code up the above environment to run policy iteration. You will need to think about how to code up the probability transition matrix

$\mathbb{R}^{100\times 100} \ni T_{x,x'}(u) = \mathrm{P}(x' \mid x, u)$, the run-time cost $q(x,u)$, and the terminal cost $q_f(x)$. Policy iteration is easy to implement if you represent all the above quantities as matrices and vectors. Plot the environment to check if it confirms to the above picture.

(b) **(10 points)** Initialize policy iteration with a feedback control $u^{(0)}(x)$ where the robot always goes east, this results in a policy $\pi^{(0)} = (u^{(0)}(\cdot), u^{(0)}(\cdot), \dots)$. Write the code for policy evaluation to obtain the cost-to-go from every cell in the above picture for this initial policy. Plot the value function $J^{\pi^{(0)}}(x)$ as a heatmap in the above picture.

(c) **(10 points)** Execute the policy iteration algorithm, you will iteratively perform policy evaluation and policy improvement steps. For the first 4 iterations, plot the feedback control $u^{(k)}(x)$ (using arrows as shown in the lecture notes (https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.arrow.html, you can also write the control input in the cell). You should color the cell using the value function $J^{\pi^{(k)}}(x)$.

We have left the transition probabilities and the reward structure a bit vague to force you to think carefully of the nuances of this problem. But some clarification could be useful.

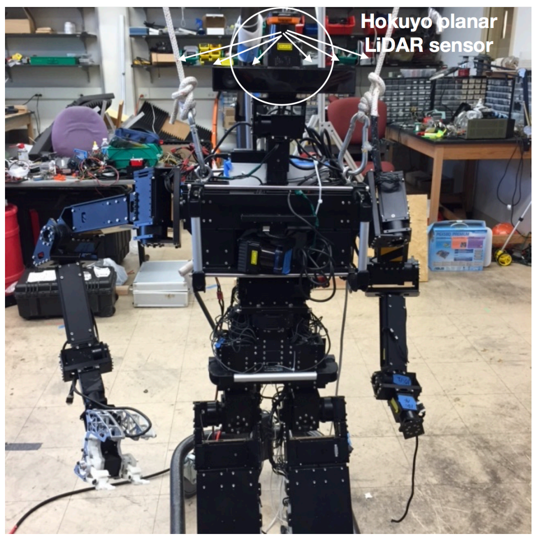
(1) You can code up what are called "sticky obstacles", i.e., if the robot enters an obstacle, then it stays there forever while incurring the obstacle cost at each time instant.

(2) It is easiest to think of the runtime cost in this problem as a function of three quantities $q(x, u, x')$ where x is the current state, u is the control and x' is the next state. The Bellman equation the becomes

$$J^*(x) = \min_{u \in U} \mathop{\mathrm{E}}_{x'} \left[q(x, u, x') + \gamma J^*(x') \right].$$

You will submit your own code for this problem; there is no auto-grader.

Problem 2 (Simultaneous Localization and Mapping (SLAM) with a particle filter, 60 points (No Autograder).). In this problem, we will implement mapping and localization in an indoor environment using information from an IMU and a LiDAR sensor. We have provided you data collected from a humanoid named THOR that was built at Penn and UCLA (https://www.youtube.com/watch?v=JhWYYuba1nE). You can read more about the hardware in this paper (https://ieeexplore.ieee.org/document/7057369.)



Hardware setup of Thor The humanoid has a Hokuyo LiDAR sensor (https://hokuyo-usa.com/products/lidar-obstacle-detection on its head (the final version of the robot had it in its chest but this is a different version); details of this are in the code (which will be explained shortly). This LiDAR is a planar LiDAR sensor and returns 1080 readings at each instant, each reading being the distance of some physical object along a ray that shoots off at an angle between (-135, 135) degrees with discretization of 0.25 degrees in an horizontal plane (shown as white rays in the picture). We will use the position and orientation of the head of the robot to calculate the orientation of the LiDAR in the body frame.

The second kind of observations we will use pertain to the location of the robot. However, in contrast to the previous homework were we used the raw accelerometer and gyroscope readings to get the orientation, we will directly use the (x, y, θ) pose of the robot in the world coordinates (θ denotes yaw). These poses were created presumably on the robot by running a filter on the IMU data (such estimates are called odometry estimates), and just as you saw some tracking errors in the previous homework, these poses will not be extremely accurate. However, we will treat them conceptually the same way as we treated Vicon in the previous homework, namely as a much more precise estimate of the pose of the robot that is used to check how well SLAM is working.

Coordinate frames The body frame is at the top of the head (X axis pointing forwards, Y axis pointing left and Z axis pointing upwards), the top of the head is at a height of 1.263m from the ground. The transformation from the body frame to the LiDAR frame depends upon the angle of the head (pitch) and the angle of the neck (yaw) and the height of the LiDAR above the head (which is 0.15m). The world coordinate frame where we want to build the map has its origin on the ground plane, i.e., the origin of the body frame is at a height of 1.263m with respect to the world frame at location (x, y, θ).

Data and code

(a) **(0 points)** We have provided you 4 datasets corresponding to 4 different trajectories of the robot in Towne Building at Penn. For example, dataset 0 consists of two files data/train/train_lidar0.mat and data/train/train_joint0.mat which contain the LiDAR readings and joint angles respectively. The functions load_lidar_data and load_joint_data inside load_data.py read the data. You can run the function show_lidar to see the LiDAR data. Each of the data reading functions returns a data-structure where t refers to the time-stamp (in seconds) of the data, xyth refers to (x, y, θ) *pose of the LiDAR* and rpy refers to Euler angles (roll, pitch, yaw). The joint data contains a number of fields, but we are only interested in the angle of the head and the neck at a particular time-stamp. The array slam_t.joint.head_angles contains the angles of neck and head respectively in the first two rows. This data is the same as the data inside the first two rows slam_t.joint.pos (that array contains the angles of all joints). You should read these functions carefully and check the values returned by them. The dicts joint_names and joint_names_to_index can be used to read off the data of a specific joint (we only need the head and the neck).

(b) **(0 points)** Next look at the slam.py file provided to you. Read the code for the class map_t and slam_t and the comments provided in the code very carefully. You are in charge of filling in the missing pieces marked as TODO: XXXXXX. A suggested order for studying this code is as follows: slam_t.read_data, slam_t.init_sensor_model, slam_t.init_particles, slam_t.rays2world, map_t.__init__, map_t.grid_cell_from_xy. Next, the file utils.py contains a few standard rigid-body transformations that you will need. You should pay attention to the functions smart_plus_2d and smart_minus_2d that will be used to code up the dynamics propagation step of the particle filter.

(c) **(10 points, dynamics step)** Next look at main.py which has two functions run_dynamics_step and run _observation_step which act as test functions to check if the particle filter and occupancy grid update has been updated correctly. The run_dynamics function plots the trajectory of the robot (as given by its IMU data in the LiDAR data-structure). It also initializes 3 particles and plots all particles at different time-steps while performing the dynamics step with a very small dynamics noise; this is a very neat way of checking if dynamics propagation in the particle filter is working correctly. This function will create two plots, one for the odometry trajectory and one more for the particle trajectories, both these trajectories should match after you code up the dynamics function slam_t.dynamics_step correctly.

(d) **(20 points, observation step)** The function run_observation_step is used to perform the observation step of the particle filter to get an estimate of the location of the robot and updates to the occupancy grid using observations from the LiDAR. First read the comments for the function slam_t.observation_step carefully.

We first discuss the particle filtering part.

(i) Compute the head and neck position for the time t. For each particle, assuming that that particle is indeed the true position of the robot, project the LiDAR scan slam_t.lidar[t]['scan'] into the world coordinates using the slam_t.ray2world function. The end points of each ray tell us which cells in the map are occupied, for each particle.

(ii) In order to compute the updated weights of the particle, we need to know the likelihood of LiDAR scans given the state (our current occupancy grid in the case of SLAM). We are going to use a simple model to do so

$$\log \mathrm{P}(\text{LiDAR scan as if the robot is at particle } p \mid m) = \sum_{ij \in O} m_{ij} \tag{1}$$

where O is the set of occupied cells as detected by the LiDAR scan assuming the robot is at particle p and m_{ij} is our current estimate of the binarized map (more on this below). In simple words, if the occupied cells as given by our LiDAR match the occupied cells in the binarized map created from the past observations, then we say the log-probability of particle p is large.

(iii) You will next implement the function slam_t.update_weights that takes the log-probability of each particle p, its previous weights, calculates the updated weights of the particles.

(iv) Typically, resampling step (slam_t.stratified_resampling) is performed only if the effective number of particles (as computed in slam_t.resample_particles) falls below a certain threshold (30% in the code). Implement resampling as we discussed in the lecture notes.

Mapping We have a number of particles $p^i = (x^i, y^i, \theta^i)$ that together give an estimate of the distribution of the location of the robot. For this homework, you will only use the particle with the largest weight to update the map although typically we update the map using all particles. Our goal is simple: we want to increase map_t.log_odds array at cells that are recorded as obstacles by the LiDAR and decrease the values in all other cells. You should add slam_t.log_odds_occ to all occupied cells and add slam_t.log_odds_free from all cells in the map. It is also a good idea to clip the log_odds to like between [-slam_t.map.log_odds_max, slam_t.map.log_odds_max] to prevent increasingly large values in the log_odds array. The array slam_t.map.cells is a binarized version of the map (which is used above to calculate the observation likelihood).

Check the run_observation_step function after you have implemented the observation step.

(e) Since the map is initialized to zero at the beginning of SLAM which results in all observation log-likelihoods to be zero in (1), we need to do something special for the first step. We will use the first entry in slam_t.lidar[0]['xyth'] to get an accurate pose for the robot and use its corresponding LiDAR readings to initialize the occupancy grid. You can do this easily by initializing the particle filter to have just one particle and simply calling the slam_t.observation_step as shown in main.py.

(f) **(30 points)** You will now run the full SLAM algorithm that performs one dynamics step and observation step at each iteration in the function run_slam in main.py. Make sure to start SLAM only after the time when you have both LiDAR scans and joint readings (the two arrays start at different times). For all 4 datasets, you will plot the final binarized version of the map, (x, y) location of the particle in the particle filter with the largest weight at each time-step and the odometry trajectory (x, y) (in a different color); this counts for 10 points each.

Some Notes This problem is much easier and shorter than it may seem. You should go through these steps carefully and in the suggested order. You should make sure that the results of the previous step are correct before proceeding. The two functions in main.py to check the dynamics and observation step are very important to find bugs. You do not need to implement more than 100 lines of code.

Problem 3 (Building a NeRF, 40 points (No Autograder)). NeRF is a technique for mapping complex scenes by optimizing an underlying continuous volumetric representation using a sparse set of input views. NeRFs represent the scene using a fully connected (non-convolutional) deep network. The input to the network is 5-dimensional $x \in \mathrm{SE}(3)$ (without the roll). This consists of the 3-dimensional location in Euclidean space, and two viewing directions. Using this input, the neural network inside the NeRF outputs volume density $\sigma(x)$ and a view-dependent color $c(x)$ at that spatial location. In this problem, we will implement a simplified NeRF, which only takes 3D Euclidean coordinates (as you can imagine, the pictures from such a NeRF do not change depending upon the viewpoint and therefore they will not look as natural). We will implement the simplest possible version of a NeRF without a lot of bells and whistles that are used in actual implementations, on downsized training images. This way, the model will be small enough to train locally on your laptop, or on Google Colab.

(a) Data Loading and COLMAP (5 points). We provide a dataset consisting of 100 LEGO images captured from various angles (you are also encouraged to capture your own dataset and show results on it). You will use COLMAP, a Structure-from-Motion (SfM), and Multi-View Stereo (MVS) pipeline to obtain camera extrinsic estimation. COLMAP is an open-source library that is compatible with Mac (install using Homebrew), Linux, and Windows. Install COLMAP first following instructions provided in the COLMAP documentation or the one that NeRF Studio provides.

Assuming the images are taken by the same camera (images have the same intrinsic parameters, i.e., the same camera calibration), you should use COLMAP to reconstruct a sparse model. The package also comes with a GUI (you can call it using "colmap gui") that provides a great interface and visualization. After getting the sparse model, you will have to understand the provided colmap2nerf script and use it to transform the sparse model file into a JSON file which contains information such as camera intrinsic and extrinsic corresponding to each image. We will need this information to begin training the NeRF.

You will then implement the **load_colmap_data** function, which reads in the generated JSON file as well as the raw images. We recommend you resize the raw images to a lower resolution, for example, from 800×800 to 200×200, so that it is feasible to train everything on your laptop. After resizing the images, remember to change the camera parameters (height, width, and focal length) accordingly. You should report these parameters in the PDF and how you calculated them.

(b) Implementation of the NeRF (20 points). You will now implement four key functions.

The get_rays function. Assuming a pinhole camera model, complete the **get_rays** function, which takes camera intrinsic parameters (camera calibration matrix) and extrinsic parameters (locations from where the images where collected) as input and returns a set of rays in the world frame. Each ray starts from the camera origin and passes through one of the pixels (see the figure in Section 4.5.1 in the lecture notes). We will use the homogeneous coordinates. Given a point $x_c = (i, j, k, 1)$ in the camera frame, the point can be transformed from the camera frame to the world frame with $x_w = T_w^c\, x_c$, where T_w^c denotes the 4 × 4 transformation matrix obtained from the previous question.

It is useful to emphasize the coordinate convention. We will adhere to the standard NeRF coordinate convention for camera coordinates: +X is right, +Y is up, and +Z points back and away from the camera, i.e., the -Z direction corresponds to the direction at which the looking at. It is important to note that other code-bases on the Internet may adopt the COLMAP/OpenCV convention, where the Y and Z axes are flipped compared to ours, but the +X axis remains the same. The world coordinate system is oriented such that the up vector is +Z. The XY plane is parallel to the ground plane.

The sample_points_from_rays function. Given a set of rays emanating from the camera center, we will discretize each ray into segments to approximate the integrals during volume rendering. Implement the sample_points_from_rays function, which returns an array of N_{sample} points along each ray in world coordinates.

(a) With a rough estimate of the distance from the object to the camera, we can determine the clipping thresholds s_{near} (the distance of the nearest point of interest) and s_{far} (the distance of the farthest point of interest). Each ray will only be evaluated within the range of s_{near} and s_{far}, which defines the volume of interest. Given a fixed number of points N_{sample}, a small $s_{\text{far}} - s_{\text{near}}$ means that sampled points along the ray are closer to each other; this leads to a better estimation for the integral.

(b) You can sample uniformly along the ray. For enhanced performance, consider incorporating some randomness into the sampling process while ensuring that there is at least one point every $(s_{\text{far}} - s_{\text{near}})/N_{\text{sample}}$.

The position_encoding function. Like we discussed in the lecture, an MLP with a finite width and a certain number of layers may not be able to represent functions of arbitrarily high bandwidths which are necessary to get high-frequency textures. This leads to blurry images from the NeRF. A neat solution to this issue is to use a different representation for the inputs $x \in \mathbb{R}^3$. Instead of using x we use

$$\varphi(x) = (\sin\left(2^k x_1\right), \sin\left(2^k x_2\right), \sin\left(2^k x_3\right), \dots)_{k=0,\cdots,10}$$

where k is the frequency and we choose, say 10 different frequencies. The input layer of the MLP would therefore be 30- instead of 3-dimensional.

The volume_rendering function. Here you will implement the volume rendering function in Equation 4.31 in the lecture notes. In summary, given a ray with points at distances

$$s_i = s_{\text{near}} + \frac{i}{N_{\text{sample}}}(s_{\text{far}} - s_{\text{near}})$$

we will calculate

$$\text{opacity: } \alpha_j = 1 - e^{-\sigma(s_j)(s_{j+1}-s_j)}$$

$$\text{transmittance: } p(s_i) = \prod_{j=1}^{i-1}(1-\alpha_j)$$

$$\text{color: } c = \sum_{i=1}^{N_{\text{sample}}} c(s_i)\alpha_i \prod_{j=1}^{i-1}(1-\alpha_j).$$

for each ray corresponding to each pixel. Implement the volume_rendering function, which renders an RGB image using the predicted radiance field.

(d) Network Training (10 points). We provide the neural network architecture and a simple training loop for you to start. Fill in the **nerf_step_forward** function with your implementations of the functions above. Your report should mention the parameters for the **train** function, including s_{near}, s_{far}, and N_{sample}. Start with N_{sample} of 32 and the hidden dimension of the MLP h_{dim} of 32. With this setting, you should be able to train the network on a laptop CPU in about 20 minutes.

(i) We highly recommend rendering and visualizing the network prediction every few iterations (doing so is similar to calculating the validation loss after few epochs while training a standard neural network-based classifier). This is an easy way to assess the network's performance. You can select one of the poses from the training set or randomly select your own pose as the test pose. Then, render an RGB image at the test pose using **nerf_step_forward** and check if the rendered image makes sense.

(ii) If you have access to a GPU, or decide to use Colab, consider increasing N_{sample} and h_{dim}. Doing so should lead to improved results.

You should report a plot of the training loss as a function of the number of weight updates. You should report the final training loss, and for about 5-6 randomly sampled images from the training dataset, you should show the original image and the one rendered from the NeRF from the same viewpoint (this is reporting predictions of the network on the training samples).

(e) Inference (5 points). Take the trained network, randomly pick 5 viewpoints from as different poses as you can and report the rendered RGB images from these viewpoints. Try to find viewpoints where the NeRF is working well as well as ones where it is not.