UNIVERSITY OF VIRGINIA

CS 6501-012: LEARNING IN ROBOTICS

HOMEWORK 3

DUE: 04/18/25 TUE BY 11.59 PM

Instructions

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

- You will submit solutions typeset in LaTeX on Gradescope. You can use hw_template.tex from the course website (under assignments).
- 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 UVA 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 or a figure feel free to draw it on paper clearly, click a picture and include it in your solution. But we cannot accept handwritten solutions. At the same time do not spend undue time just on typesetting solutions.
- On Gradescope you will see an entry of the form "HW 1 PDF" where you will upload the PDF of your solutions. You will also see entries like "HW 1 Problem X 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 the same filename as stated in the respective problem statement.
- **You should include all the relevant plots in the PDF if the problem requests them, without doing so you will not get full credit.** You can, for instance, export your Jupyter notebook as a PDF (you can also use text

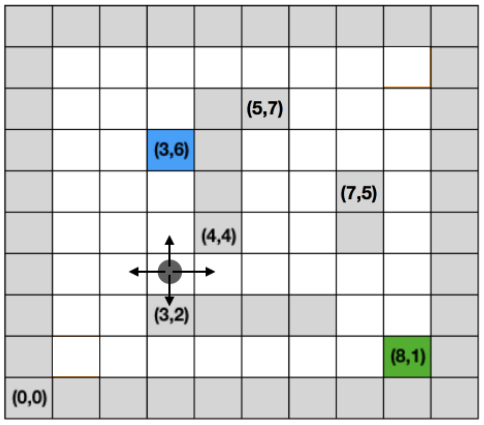
cells to write your solutions) and export the same notebook as a Python file to upload your code.

- **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, 40 points). Consider the following Markov Decision Process. The state-space is a 10×10 grid, cells that represent 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 control actions (North (Up), East (Right), West (Left) and South (Down)). The robot cannot move diagonally. The action 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 robot pays a cost of 1 (i.e., reward is -1) for each control action it takes, regardless of the outcome. If the robot enters a state marked as an obstacle, it gets a reward of -100 (so effectively a huge penalty) for each time-step that it remains inside the obstacle cell. This essentially ensures that the robot never tries to enter and obstacle cell and any action that directs it into an obstacle or a wall has a significant penalty. The robot is not allowed to take actions from within an obstacle cell — it effectively gets "stuck” there and continues to incur the heavy penalty of -100 per time-step. Your code should reflect this by setting the transition probabilities out of obstacle cells to zero, and the stay-in-place probability to 1. Finally, the robot is allowed to stay in the goal state when arrived indefinitely (i.e. take an action to not move).

You will implement the policy iteration algorithm to find the best trajectory for the robot (which maximizes expected accumulated reward) to go from the blue cell to the green cell.

(a) **(10 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 $\mathbb{R}^{100\times 100} \ni T_{x,x'}(u) = \mathrm{P}(x' \mid x, u)$, and the run-time cost $q(x, u)$. Since this is an infinite-horizon problem, your cost function should encourage the robot to reach the goal efficiently rather than wandering. Policy iteration is easier to implement if you represent all the above quantities as matrices and vectors. Before you begin, plot the environment you have coded to check if it conforms to the above picture. While defining the transition matrix and cost function, ensure that obstacle cells are treated as absorbing states with no outgoing transitions and high penalty.

What to submit:

- **(3)** Plot of your coded environment.
- **(7)** Explain in detail (in math/not in code) any and all cost functions that you decided to implement and why.

(b) **(15 points)** Initialize policy iteration with a feedback control $u^{(0)}(x)$ where the robot always goes east (Right), this results in a policy $\pi^{(0)} = (u^{(0)}(\cdot), u^{(0)}(\cdot), \ldots)$. Write the code for policy evaluation to obtain the cost-to-go from every cell in the above picture for this initial policy.

What to submit:

- **(10)** Plot the value function $J^{\pi^{(0)}}(x)$ as a heatmap corresponding to this initial policy. Please clearly label your heatmap. **(Submit this plot in your PDF)**
- **(5)** Interpret this policy evaluation heatmap. Explain whether grid cells immediately to the left of obstacles or walls exhibit higher or lower cost-to-go compared to other cells, and why. What about the cells in the vicinity of the goal state ?

(c) **(15 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 want you to include the plots for the first 4 iterations so we can inspect that the policy iteration is working correctly. This does not mean that the policy should converge in 4 iterations. Your policy $u(x)$ should be undefined or ignored at obstacle cells. These are not valid states for planning, and you should neither plot arrows nor assign actions to them.

What to submit:

- **(8)** The feedback control and value function heatmap for the first 4 iterations i.e. $u^{(1)}(x)$, $u^{(2)}(x)$, $u^{(3)}(x)$, and $u^{(4)}(x)$.

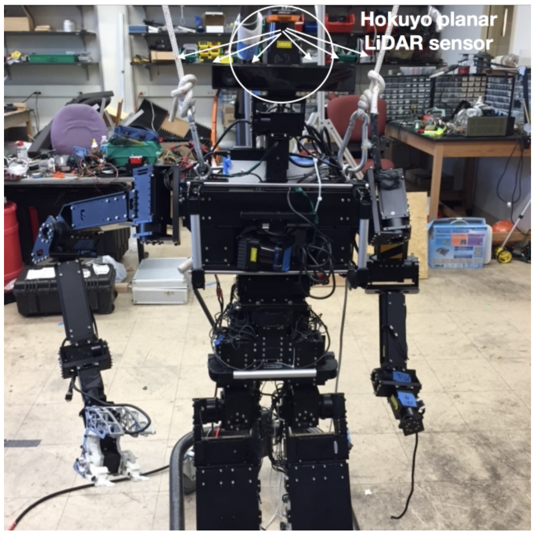
- **(7)** Report both the number of iterations it took to converge and the plot the final converged optimal policy as well as value function i.e. $u^*(x)$ and $J^*(x)$

There is no auto-grader setup for this problem. You still need to upload your entire code for this problem to get any grade.

You will not get any credit for the problem if there is no code submitted.

FAQ 1: How do we handle edges in transition probability matrix ? If the agent tries to leave the grid, should it just stay in the same place? Yes! the probability associated with the action that takes the robot out of bounds gets combined with the probability to stay in place.
FAQ 2: What is the purpose of the Initial State of the robot ? Yes. Policy iteration produces the optimal control for every state. The inital state is just a quick way to see if your optimal stationary policy is correct and is able to guide the robot around multiple obstacles to reach the goal. Other than that it has no special significance.
FAQ 3: Is there a difference between taking an action and choosing to “not move”? In this problem, we don’t explicitly define a "none” action. However, if the robot chooses an action that results in staying in the same cell (due to randomness or a wall/obstacle), it still pays the -1 cost. Staying in the goal state by *not moving* is the only case where the robot pays no penalty and gets to accumulate reward 10.

Problem 2 (Simultaneous Localization and Mapping (SLAM) with a particle filter, 80 points.). 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 real humanoid named THOR
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 2D 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 provided (x, y, θ) pose of the robot in the world coordinates (θ denotes heading/yaw). These poses were created presumably on the robot by running a filter on the IMU data (similar to what you did in problem 2 - UKF of hw2). 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 with 4 datasets corresponding to 4 different trajectories of the robot. 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. 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) **(20 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. Be sure you're applying the odometry difference (odom[t+1] - odom[t]) in the robot's local frame using smart_minus_2d.

In your PDF, explain in detail how you implemented the dynamics step and include these plots.

(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 filter localization part of SLAM.

(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. The weights should be updated in log-space, then normalized correctly. Avoid simply summing or multiplying raw probabilities, which leads to underflow and incorrect reweighting.

(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.

Let us now discuss the mapping part for SLAM. 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 free cells. You should add slam_t.log_odds_occ to all occupied cells and add slam_t.log_odds_free from free cells in the map. Do not subtract log_odds_free from all cells — only those free cells on rays between the robot and obstacle hits. Otherwise, you'll erase legitimate map content. It is also a good idea to clip the log_odds to lie 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.

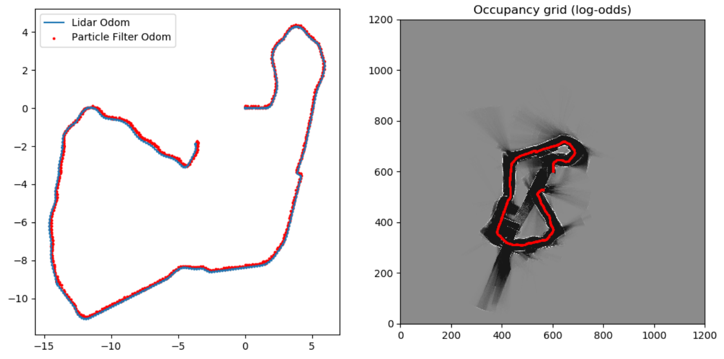
In your PDF, explain in detail how you implemented the localization and mapping steps

(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) **(40 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. Your SLAM trajectory may not perfectly match odometry — that's okay! But large drifts or jagged motions usually point to a bug in particle update or observation step.

What to submit For each dataset:

- **Final occupancy grid map:** The final binarized version of the map/occupancy grid, (x, y). These should be clear and high resolution.
- A plot showing the location of the particle in the particle filter with the largest weight (best particle) at each time-step; and ground truth Odometry trajectory (x, y) (in a different color); with a clearly marked legend.
- A discussion on tuning and any deviations from the provided template. Here is an example submission:



Clearly explain your implementation choices and note if you processed fewer timesteps for performance.

(g) **Code:** There is no auto-grader. You need to upload your entire code for this problem to get any grade. We will run main.py for your submitted code for the dataset and it should generate the plots/images you describe in the report.

Hints

- The THOR paper is very good for understanding the overall problem and you are encouraged skim through that. That paper does not map directly to the code base.
- We have prepared two helper documents: Instructor notes for the problem in *problem2_notes.pdf* and helper slides *guidance.pdf*. These documents will help you a lot with step by step suggestions and visuals about the problem.
- The two functions in main.py to check the dynamics and observation step are very important to find bugs.
- In the propagation step, add noise to the control obtained using smart minus. If you're seeing perfect overlap between particle trajectories and odometry, but your particles never diverge, you're probably forgetting to add noise.
- If your algorithm is too slow to run end-to-end, you may skip every 5–10 timesteps to speed things up. Just note this in your write-up.

FAQ 1: Do we update the map using all particles or just one? Just the highest-weight particle for this assignment.

FAQ 2: My map looks blank or totally black. What's wrong? Double-check if: Your LiDAR rays are transformed to the world correctly. You're filtering noisy scan points. Your log-odds values are being converted properly into a binary map.
FAQ 3: Can I rename functions or change structure? No, while there is no auto-grader; Please stick to the provided template. You can add helpers, but don't rename or restructure core logic — it may break grading.