SWARM
IIT KHARAGPUR

Task Round - Software & AI

Congratulations on making it to the final round of Swarm Software Team selections, that is the Task Round! This will be the most challenging round yet but also very rewarding and you will get to learn and implement a lot of new concepts.

Here we present you with five tasks and you must complete at least two. However we expect and encourage you to do more than two tasks as this will not only increase your chances of getting selected, but also help you explore new and exciting fields. Each task is provided with its specific instructions and deliverables.

Here are some general instructions and suggestions

1. There are no restrictions on using any programming language although we would prefer if you use Python. Also do try to make you programs Object Oriented.
2. You should not copy your code from any source as we will check for plagiarism, and heavily plagiarised submissions will be disqualified.
3. There will be a round of interviews after task submissions where you will be asked to present and explain your solutions.
4. The resources provided with each task are not enough! You are heavily encouraged to search the internet for relevant resources.
5. In case of any doubts feel free to ping any of us or post them on the Doubts and Queries channel of Swarm Tasks on MS Teams.

1 The RL Agent

1.1 Introduction

In the reading task you got yourself familiar with some basic concepts of Reinforcement Learning like the action value function Q, exploration, exploitation and the ϵ-greedy method. This task is aimed at taking you further into the world of RL and make you learn about some key concepts like Policies, Value Functions, and Policy Control.

1.2 Preliminaries (30% weightage)

First off you need get familiar with policies and value functions. A policy, denoted by π, is a mapping from states to probabilities of selecting each possible action. A state can be anything, for example in our grid world task below, a state is simply the block the agent is present in. Next you also have to read about the Bellman Equation which expresses a relationship between the value of a state and the values of its successor state.

One of the goals of reinforcement is to determine the optimal policy, because once you know that, you can just take the greedy action for each state to complete a task with most reward. Here we will focus on Sampling-based Learning Methods. Thus you are required to read about learning methods like Temporal-Difference Learning (or more specifically the TD(0) method), and control methods like SARSA and Q-Learning.

1.3 The Task (70% weightage)

Consider the 15 $\times$ 10 grid world shown in Figure 1. We want to move our agent, denoted by the light blue square, to the goal which is shown as the green square. However there are certain obstacles in the world, denoted by dark grey squares which our agent cannot cross. The agent can move in four directions, that is there are four actions possible in each state, `A = up, down, right, left`, which deterministically cause the corresponding state transitions, except that actions that would take the agent off the grid in fact leave the state unchanged. The agent also incurs a reward of $R = -1$ for each step it takes, except when the step leads to the goal, in which case it gets a reward $R = 0$. Also take the discount factor $\gamma = 1$, that is consider the whole reward of the future states while calculating action values.

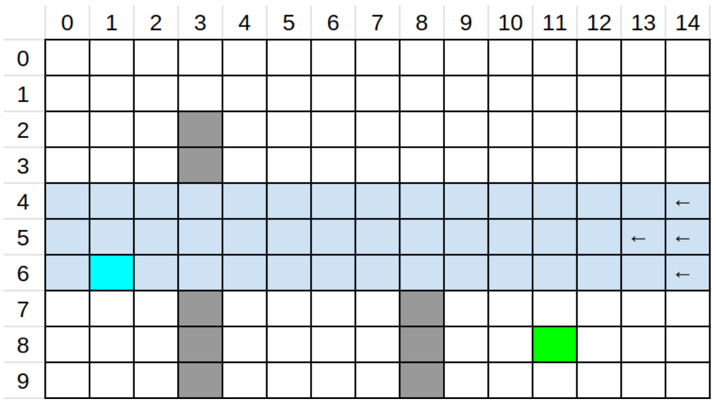

Figure 1: The Grid Environment

The agent has to face another adversary, that is a strong gale. The areas affected by the wind are shaded in blue and the direction is specified by the arrows. When the agent moves in the affected region the resultant next states are shifted rightwards by the "wind," the strength of which varies from row to row (denoted by the number of arrows). For example, if you try to move up in the wind, your resultant motion with diagonally up and right. Also note that the wind is not constant and only affects our agent 80% of the times.

Now your task is to come up with a policy that will take our agent to the goal, while maximizing the reward. You are free to use any method you like to achieve the above goals. **Bonus:** Try exploring planning methods like Dyna-Q, Dyna-Q+ or Prioritized Sweeping.

1.4 Submission

Clone this GitHub repository which contains the basic framework of the grid world and instructions on how to run your agent. You are not supposed to modify `gridworld.py` in any way.

1. You are expected to import `gridworld.py` and write an agent script to move the agent to the goal with highest reward possible.
2. You can visualise the optimal path that the agent takes using tools like OpenCV.
3. For the submission you are required to submit your agent script and a proper documentation of your solution.

2 Multiple Query PRM-APF Planner

2.1 Introduction

In the reading task, you learnt the basics of path planning, a sampling based global planner (PRM) and APF used for local planning. In this task, you will work on implementing them. You will be given a grid as the navigation environment. You need to first map the environment and then use a local planner to get to the goal node in the query.

2.2 The Task

You will need an area within which you would be working. Consider the map to be rectangular, with the bottom-left corner at $(0, 0)$ and the top right one at $(60, 40)$. You will also need some obstacles, which will be provided in a text file *obstacles.txt*. Assume all obstacles to be polygonal, and also assume the polygons to be convex.

The file contains n lines, each line describing an obstacle O_i. An obstacle O_i is described in a line with a list L of $2m_i$ numbers. Starting from the beginning of this list of numbers, if you merge two consecutive numbers $(L[a], L[a+1])$ such that $1 \leq a \leq 2m_i - 1$ and a is odd, you get the co-ordinates of one vertex V_j of O_i. Combining the m vertices (they are listed in an anticlockwise manner in the file), you get one obstacle. For example, [0, 0, 3, 1, 2, 5] denotes a triangular object with vertices $(0, 0)$, $(3, 1)$, $(2, 5)$.

There are n such obstacles.

Now you have a map populated with obstacles. Implement **PRM** on this map. The queries will be provided in another file *queries.txt* which will have n lines of four numbers. The first two numbers denote the start point and the second two numbers denote the goal point. **For each query Q_i DO NOT construct the roadmap over again**. The choice of the number of points to sample and the number of neighbours to consider is left to you.

Checkpoint 1 Construct the roadmap and show the graph on *matplotlib*. Show the sampled nodes in green, nodes that were rejected as red, obstacles as blue. Also connect the nodes that share a edge using straight lines.

Checkpoint 2 Augment the start and the goal point to the roadmap. Mark these points using a separate color. Show the graph on *matplotlib* again as described above. Then apply *Dijkstra* or A^* search to find a path from the start to the goal point. Mark the edges that constitute the path in some other color. Repeat for all queries in the file

If you have reached here, global planning is done. You have the waypoints, namely the nodes that were a part of the graph, and now you need to locally plan a path through them. For this you need to use **APF**.

We recommend you to use the **Brushfire** or the **Wavefront Planner** after breaking the map into grids.

Checkpoint 3 Divide the map into a grid (each square should be of 0.0625 sq units) and show it (use any method you want, we just want to look at it). Mark a grid square as blocked if it's centre lies inside an obstacle. Else mark it free. Color all blocked grid square as black, rest as white. (Do all this in a separate image/plot, don't make the previous plot cluttered).

Checkpoint 4 Apply **Brushfire** or the **Wavefront Planner** and find out the distances of each grid square from the obstacles. Use these distances for planning a path locally. Create an animation, or mark the path in the grid.

2.3 Submission

You need to submit all the codes that you have written. Apart from that, we also need you to show the results for each checkpoint. For checkpoint 1, submit the image of your *matplotlib* plot. Similarly for checkpoint 2, submit an image for each query, containing the expanded graph and the query. For checkpoint 3, we need an image of the black and white grid and for the final checkpoint, we need a path marked in the grid, or a *matplotlib* plot image, or an animation of the path.

Recommended If you are familiar with ipynb notebooks, instead of submitting images and all, you can just create a notebook that has everything in order. Make sure you comment well and add markdown texts to help us evaluate your submissions better. If you are using notebooks, just submit a notebook file.

NOTE: You will be provided *obstacles.txt* in a few days, but *queries.txt* only later.

3 Multilateration and Filters

3.1 Introduction

Multilateration is the process of determining the position of a vehicle/robot using time differences in sending and the arrival of signals from geostationary satellites or ground stations. Suggested reading: Multilateration Wikipedia. There is noise in the data, such as short time lags in receiving signals, which result in noise in the distance predicted. With data from redundant stations(3 required for 2D localization), you can apply filters to eliminate noise. However, the noise isn't Gaussian.

3.2 The Task

Consider the diagram shown in Figure 2. Our autonomous bot must localize itself, with the help of signals it receives from geostationary sources(stations or satellites). The signals have two components : *id of sender, timestamp*. Global positions of the signal sources are known to the bot. On receiving the signal, bot compares the timestamp with time of arrival of the signal. Based on the delay in arrival of signal, it estimates the distance from that particular source using wave velocity.

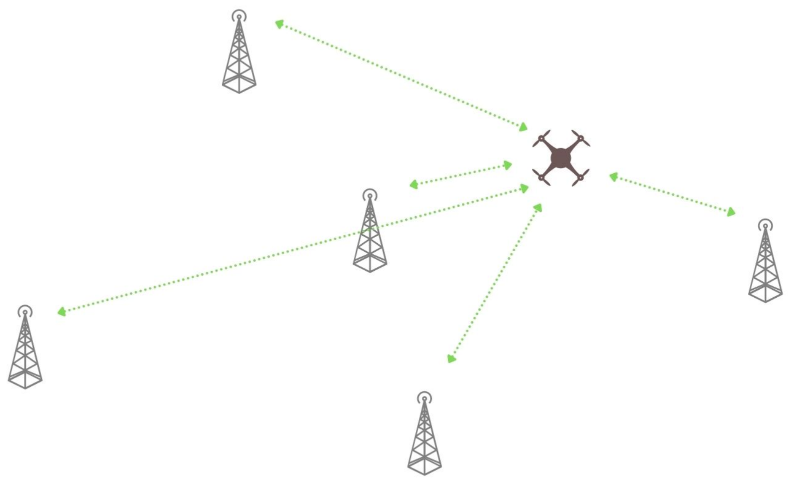

Figure 2: Multilateration

Given distance from three known points, one can perfectly localize an unknown point in 2D plane. But there are noises in the distances measured, due to various reasons such as frequency of senders, frequency of receivers, delays caused by computations itself, etc. Having more sources can help in eliminating the noise in data using filters.

Your task is to implement an appropriate filter that eliminates noise of unknown type, in order to localize the bot accurately.
Note : *Kalman filter assumes Gaussian noise. Therefore using Kalman filter might not give the best results and you may have to use Particle Filters.*

3.3 Submission

Your submissions will be judged based on accuracy, directly related to how well you model the noise.
Clone this repo: Multilateration Simulation, which contains a simplified GPS framework with visuals.
Your objective is to fill the following global function in `main.py`:

`predict(station_pos : list[coordinate], station_dis : list[float]) → coordinate`

Read in-code documentation and repo's README.md for more instructions.

1. You are expected to clone the repository as mentioned earlier and complete its **main.py**. Changing any other part of the code is neither allowed nor helpful as we will replace all of it during evaluation.

2. In `main.py`, you must implement the **predict** function. You may also declare a function for your filter. **You are not allowed to change main function and driver code**.

3. You are expected to document your approach as we will ask you to explain your approach during personal interview.

4 Lane Detection

4.1 Introduction

Lane detection is of utmost importance in autonomous robots. The vehicle must know where it is supposed to drive. Lane can be detected using basic Image Processing tricks on the visual feed of the robot. For suggested reading, refer to Reading Task resources on Image Segmentation. Lane detection isn't an easy task as the color of the lane and the lighting conditions may vary. Lane markers aren't always straight. So one must use Line detection algorithms with caution.

4.2 The Task (20 + 30 + 50%)

For this task, there are three levels, each having a video file. The objective is to process the video, detect lane and annotate the driving area, as shown in Figure 3.

Level 1 has a straight lane with white color.	20%
Level 2 has a straight lane with colors other than the white present.	30%
Level 3 has curved lanes with variable lighting conditions.	50%

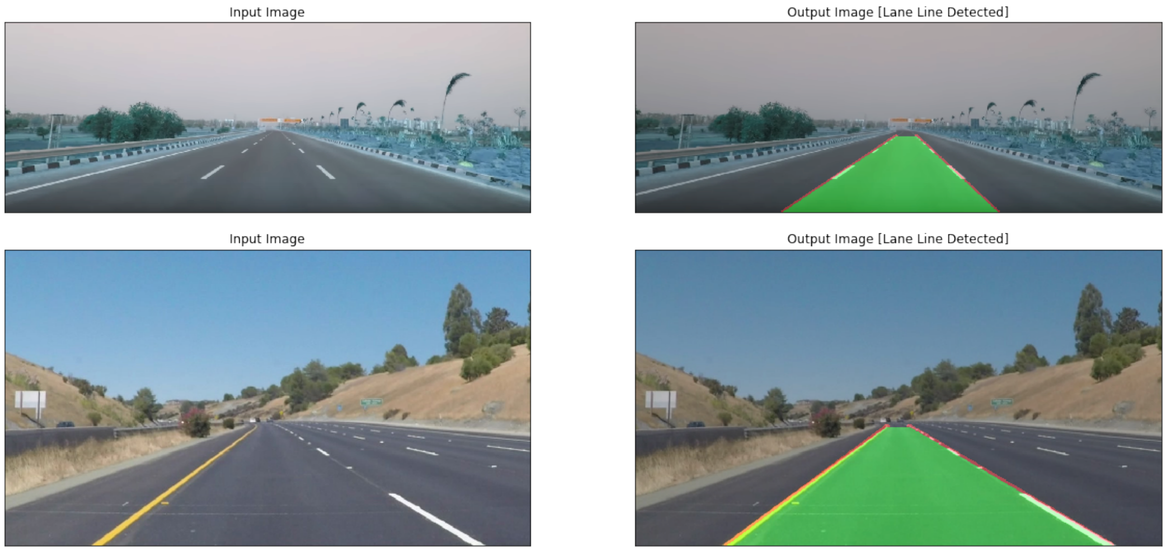
Input Image
Output Image [Lane Line Detected]
Input Image
Output Image [Lane Line Detected]

Figure 3: Lane Detection

4.3 Submission

Your submissions will be judged based on how accurately and reliably your code can detect lanes and driving area. Download the videos mentioned in the task section, and process and annotate them. Export the annotated videos using Open CV.

1. You are expected to write python or C++ code and use Open-CV for this task. Code must be documented and readable.
2. You must export the videos after processing them using Open-CV, keep them in a drive folder and send link for the same. Output videos must be named **Level1, Level2 and Level3 respectively**.
3. You are expected to document your approach as we will ask you to explain your approach during personal interview.

5 Grid Mapping

5.1 Introduction

You saw an example of *feature based* localization in the Kalman Filter task above, similarly there are mapping techniques too that map the environment based on certain target features that exhibit distinctive properties. However in this task we will learn about a *volumetric* mapping algorithm, that is the Occupancy Grid Mapping algorithm. Volumetric mapping algorithms do not depend on distinctive features in the environment for mapping.

For this task you will also have read about Binary Bayes filters, log odd notations and inverse sensor model. As a bonus you can also try looking up Scan Matching.

5.2 The Task

The purpose of this task is to implement the occupancy grid mapping algorithm. For this you will need to use GNU Octave, which is used for scientific computing and numerical computation. You can also use *Matlab* if you are already familiar as they have very similar syntax.

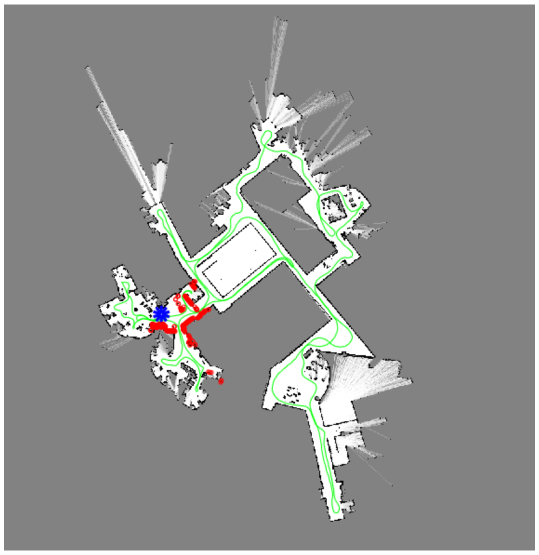

Figure 4: Grid Mapping in action

Next clone this repository containing a framework for running the grid mapping algorithm. The framework contains the following folders:

1. **data**: contains the recorded laser scans and known robot poses at each time step.
2. **octave**: contains the grid maps framework with stubs to complete.
3. **plots**: this folder is used to store images.

Now your task is to complete the functions mentioned below, which are located under the `octave` directory of the framework:

1. Implement the functions in `prob_to_log_odds.m` and `log_odds_to_prob.m` for converting between probability and log odds values.
2. Implement the function in `world_to_map_coordinates.m` for converting the (x, y) world frame coordinates of a point to its corresponding coordinates in the grid map. You might find the *Octave* functions `ceil` and `floor` useful.

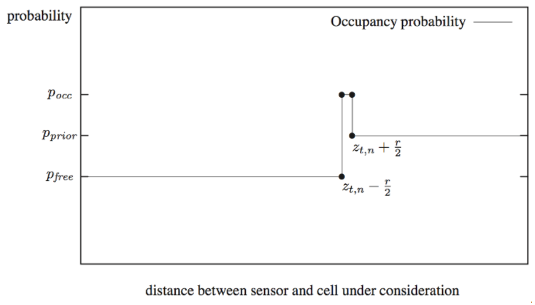

Figure 5: Inverse Sensor Model for Laser Range Finders

3. Implement the function in `inv_sensor_model.m` to compute the update to the log odds value of each cell in the map for a particular laser scan measurement.

4. Use an inverse sensor model corresponding to laser range finders (Figure 5). The corresponding p_{free} and p_{occ} values are specified in the `gridmap.m` script. Use p_{occ} to update the occupancy value of cells that laser beam endpoints hit and p_{free} for all other cells along the beam. Use the function `robotlaser_as_cartesian.m` to compute the Cartesian coordinates of the endpoints of a laser scan. The provided `bresenham.m` function can be used for computing the cells that lie along a laser beam in map coordinates. You can read about Bresenham's line algorithm.

After implementing the missing parts, you can run the occupancy grid mapping framework. To do that, launch *Octave* and change into the directory `octave` of the framework. Then either type `gridmap` in the *Command Window*, or run `gridmap.m` to start the main loop (this may take some time). The script will produce plots of the state of the resulting maps and save them in the `plots` directory.

5.3 Submission

For this task you need to submit the following:

- Completed functions as asked in the task.
- Output maps of your Grid Mapping algorithm. You can also use `ffmpeg` to generate a video from inside the plots directory as follows:

```
ffmpeg -r 10 -b 500000 -i gridmap_%03d.png gridmap.mp4
```

- A documentation of your solution.