# EE 209AS Lab1

Adrian Lam, Andrew Chen, Meet Taraviya, Yuanyuan Xiang

**Note:** The code also contains all these answers, in an interactive fashion.
https://github.com/aandrewzc/Computational-Robotics-Lab1

## 1. MDP System

We made a class for a general MDP. We pass (S, A, P, R, gamma) to this class to create its instance. This class contains all the algorithms, **which work without assuming any structure regarding the grid world example.**

a. The state-space S consists of all possible (x,y,h) where (x,y) are coordinates and h is a heading, from 0 to 11. We use 0 to represent the heading of 12 o'clock. Hence,

$$N_S = L * W * 12$$

In our grid world example with L = 8 and W = 8,

$$N_S = 8 * 8 * 12 = 768$$

b. The action space consists of no movement, moving forward with 3 possible rotations, and moving backward with 3 possible rotations.

$$N_A = 1 + 3 + 3 = 7$$

c. The function that returns the probability $p_{sa}(s')$ is called "*P_function(s, a, s_)*". This is located on line 24 of main.py.

d. The function that returns a next state *s'* following the probability distribution from $p_{sa}$ is called "*next_state_draw(s, a)*". This is located on line 137 of main.py.

## 2. Planning Problem

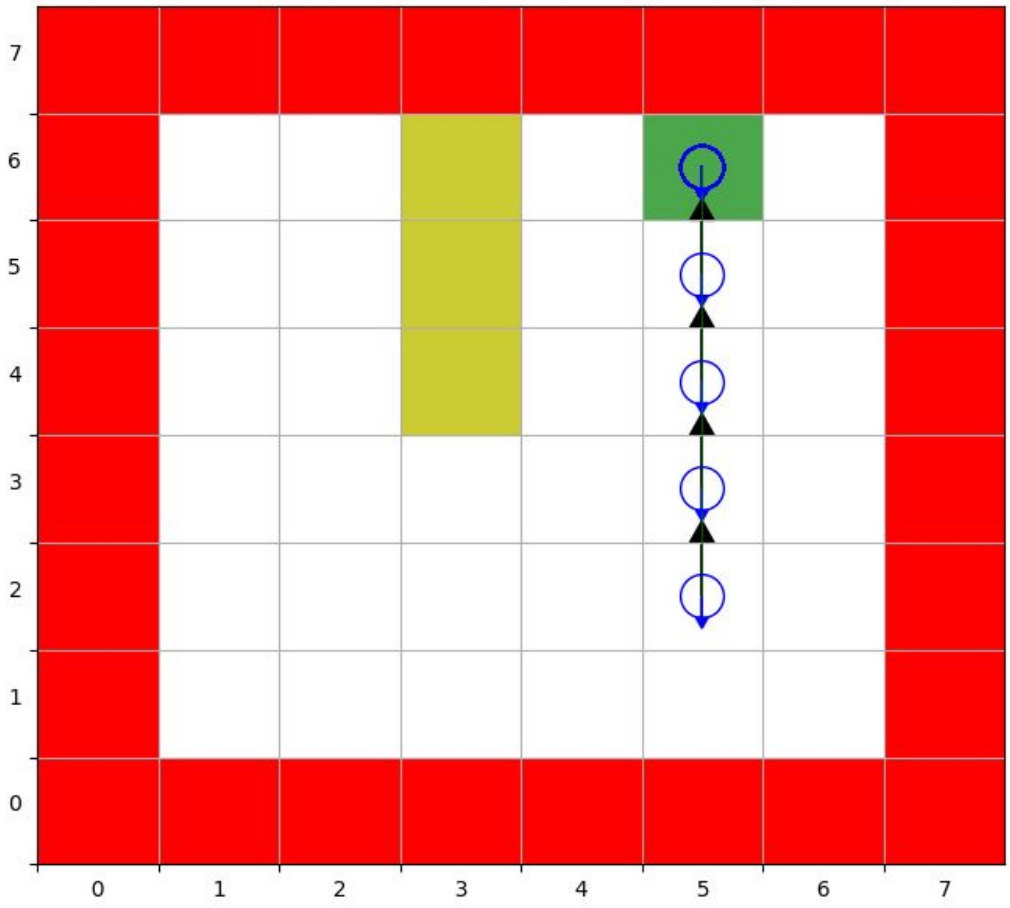
a. The function that returns the reward *R(s')* is called "*R_function(s, a, s_)*". This is located on line 59 of main.py. We assumed that **the reward is received when we enter the MDP state**. So the function only depends on *s_* for this particular MDP.

# 3. Policy Iteration

a. The function that generates the initial policy is called '*initial_policy()*'. This is located on line 244 of main.py. First, we tried to determine what move (forward/backward) should be taken, by selecting the one which takes us closer to the final state. Then, we used the turn direction taking closer to the target's direction when moving forward and farther from the target's direction when moving backward.

b. The function that generates and plots the robot trajectory is *'trajectory(s_0, policy)'.* This function also returns the cumulative discounted reward for the trajectory. This is located on line 161 of main.py. **We have not taken p_error as input** because we only use p_error in the initialization of the MDP. Once that is done, we take samples using the MDP's probability matrix, which will also contain the information regarding p_error. We limit our simulation to 200 steps.

   Example plot, starting from (5,2,6) and given initial policy:-

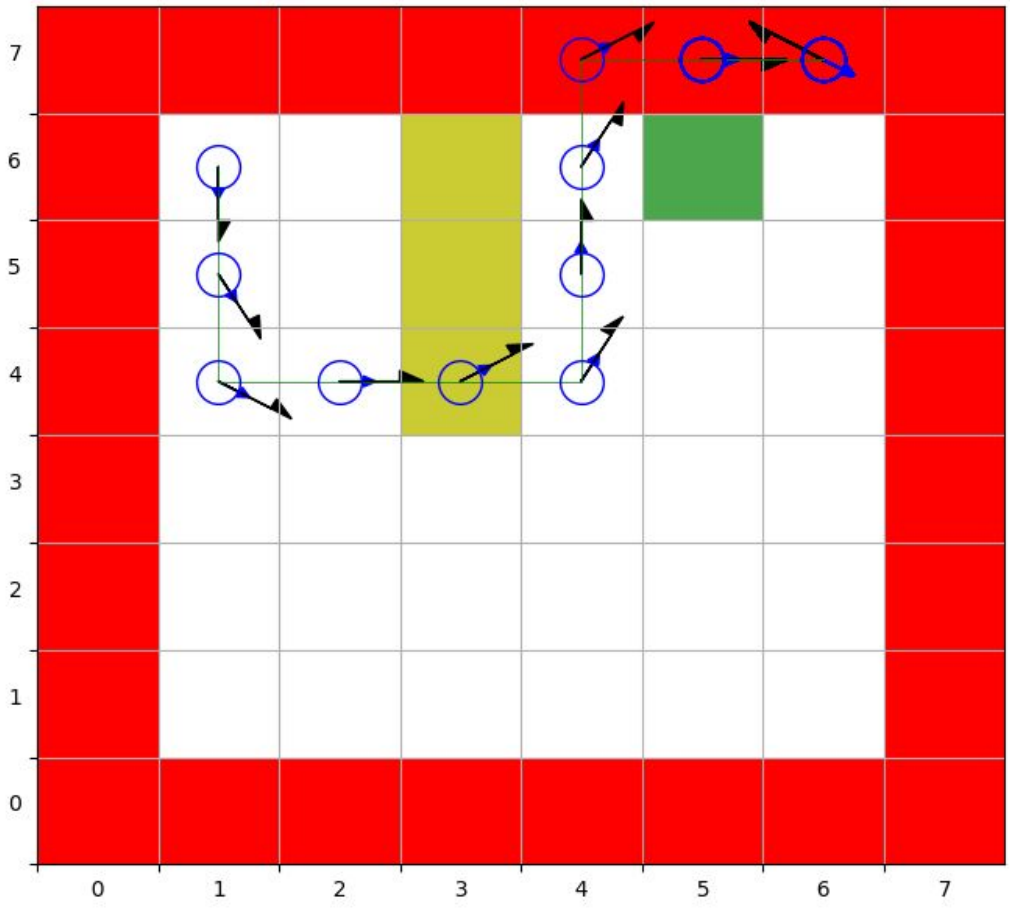



The blue circle denotes the agent. The smaller blue arrow denotes his heading. The black arrow encodes the action taken by the agent at the state. No arrow means that the action is (0,0): don't move. Otherwise, the direction of the arrow indicates the move direction (forward/backward) and the head shape indicates the turn direction. Left half arrow denotes a left turn, right half arrow for a right turn and full arrow for no turn. Green lines denote the locus of the agent.

Note that the agent is using backward steps to reach the goal state faster.

c. Below is a plot of the robot's trajectory for (1,6,6) as starting state. We can see that the initial policy as specified in the question does not get to the goal state but instead oscillates between (5,7) and (6,7). This is consistent with our dot product algorithm, but

the agent is kind of facing a “turning radius limitation”.




d. The function that computes policy evaluation is called ‘*evaluate_policy_accurate(self, pi)*’ It is located on line 52 of mdp.py. The function formulates Bellman’s equations as a set of linear equations and solves them using *np.linalg.solve*.

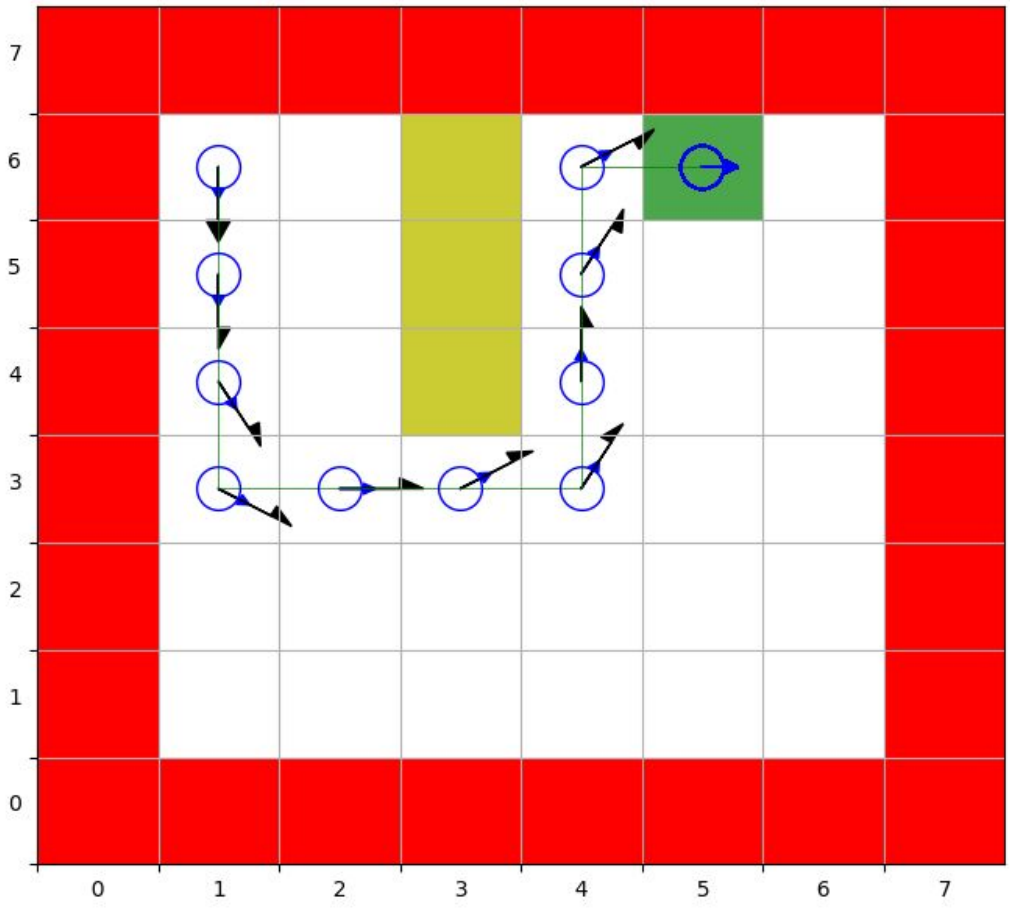
   Earlier, we didn’t cache the probability values and called a function every time we required an entry. Because of this, the program was slow, and we tried to gain speed by implementing a faster version of policy evaluation ‘*evaluate_policy_fast(self, pi)’* on line 99 of mdp.py which tried to find the value of the policy pi by iteratively applying a Bellman backup operator.

e. The cumulative reward of this trajectory is -485.58689929449224

f. The function that returns the optimal policy given a one-step lookahead on value *V* is called '*create_policy(self, V)*'. it is located on line 153 of mdp.py. It simply finds *a* such that *V(s) = Q(s, a)* and assigns it the state *s*: *pi(s)=a.*

g. The combined function that returns the optimal policy and the optimal value using policy iteration is called *'policy_iteration(self, pi_0=None)*'. It is located on line 191 of mdp.py. If an initial policy is not provided, we arbitrarily take the first action for each state as a starting policy. This function calls *evaluate_policy_accurate(self, pi).*

h. Below is a plot of the trajectory under the optimal policy.
Our *'policy_iteration(self, pi_0=None)*' function prints one dot to the console for each iteration. Thus, this policy iteration function takes 5 iterations to output the optimal policy. Observe that it avoids the yellow lane of negative reward, and starts turning at (4,5) to complete turning at the next state.




Cumulative reward: 3.874

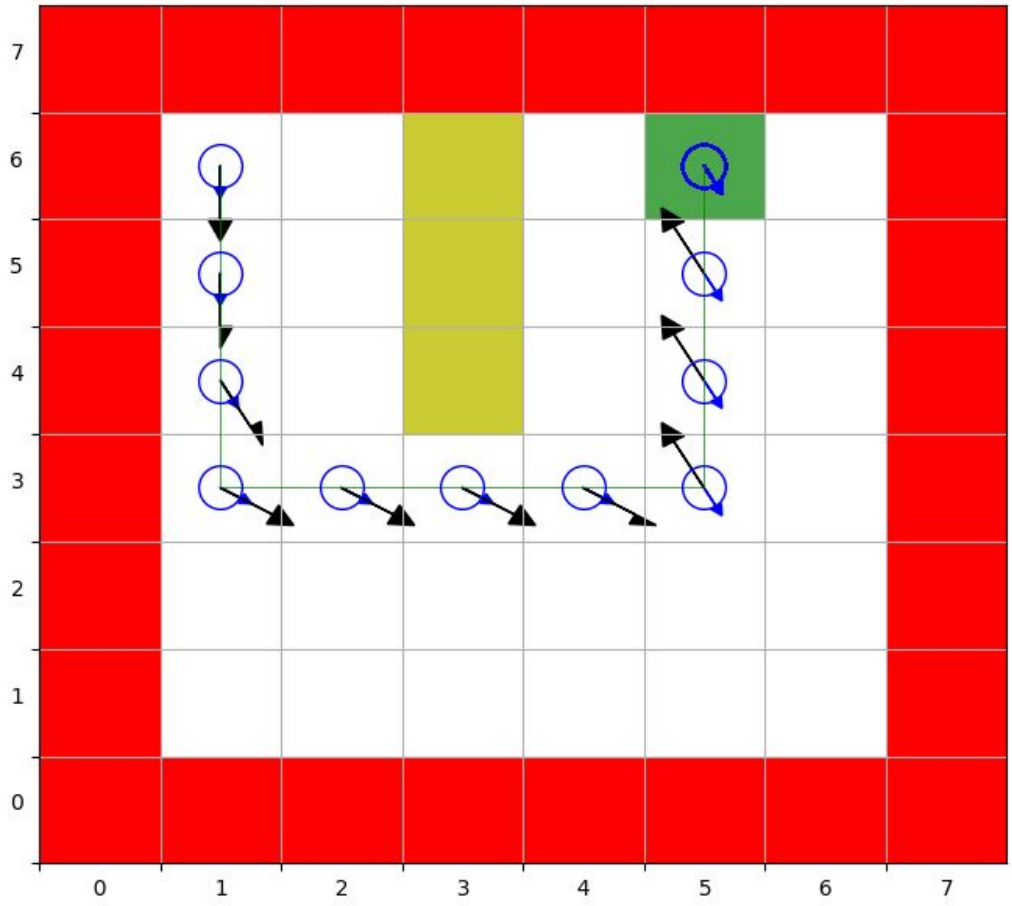
i. The measured compute time is 0.63 sec, measured using the inbuilt *timeit* module of python3.

# 4. Value Iteration

a. The function that computes value iteration is called '*value_iteration(self)*'. It is located on line 243 of mdp.py. It iteratively applies the Bellman optimality operator to get the optimal value function.

b. Below is a plot of the trajectory under the optimal policy

Our *'value_iteration(self)'* function prints one dot for each iteration. Thus, this value iteration function takes 134 iterations to output the optimal policy. Note that this policy is different from the one obtained from the policy iteration. However, we confirmed that both of them are equally good, in terms of their value functions being equal. This is also reflected in our answers for cumulative rewards. The policy learned from value iteration knows how to take advantage of "backward" moves.

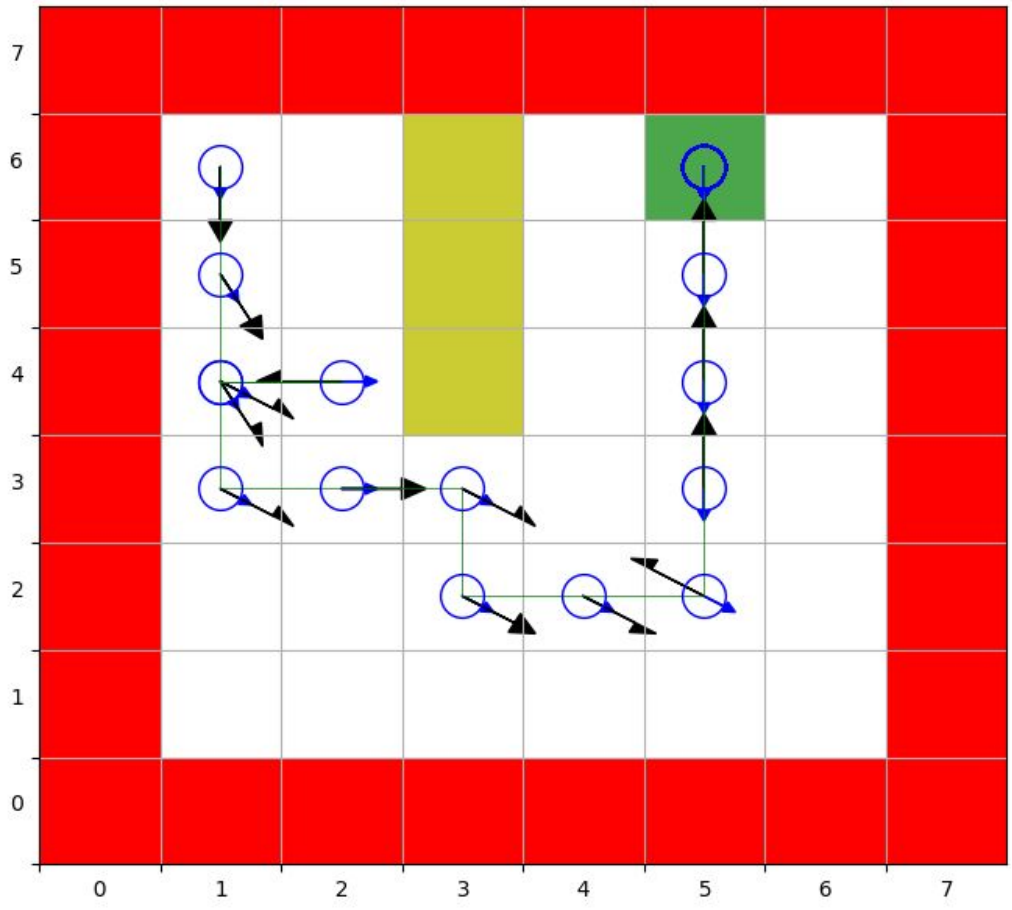



Cumulative reward: 3.874

c. The measured compute time is 5.32 sec. Value iteration is slower. From our observations, a single iteration of VI took less time than that of PI, but VI took much more iterations than PI.

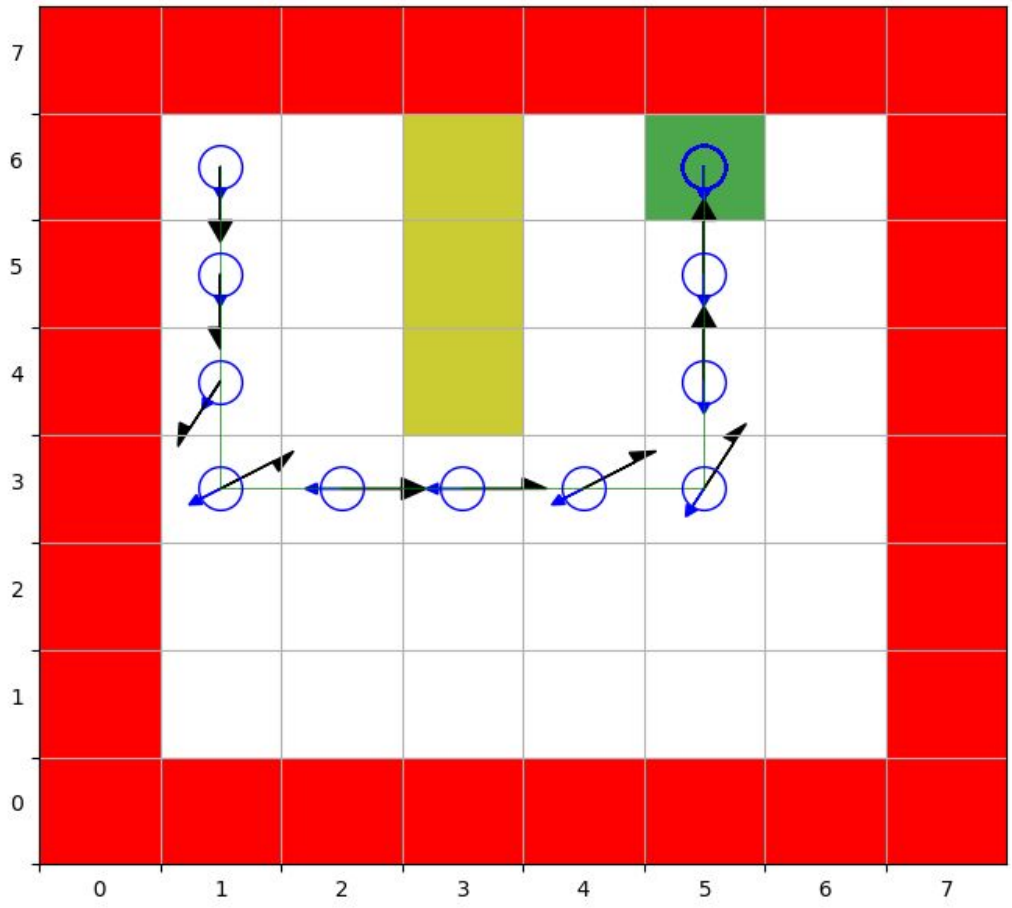
## 5. Additional Scenarios

a. Below is a plot of the recomputed trajectory
   The policy iteration function takes 9 iterations to output the optimal policy. Because of uncertainty, we start seeing inefficient behavior like visiting the same state twice. Hence, it is taking more steps to reach the goal state and the cumulative reward is less because of discounting. This agent also gets to use “backward” moves.




Cumulative reward: 2.542

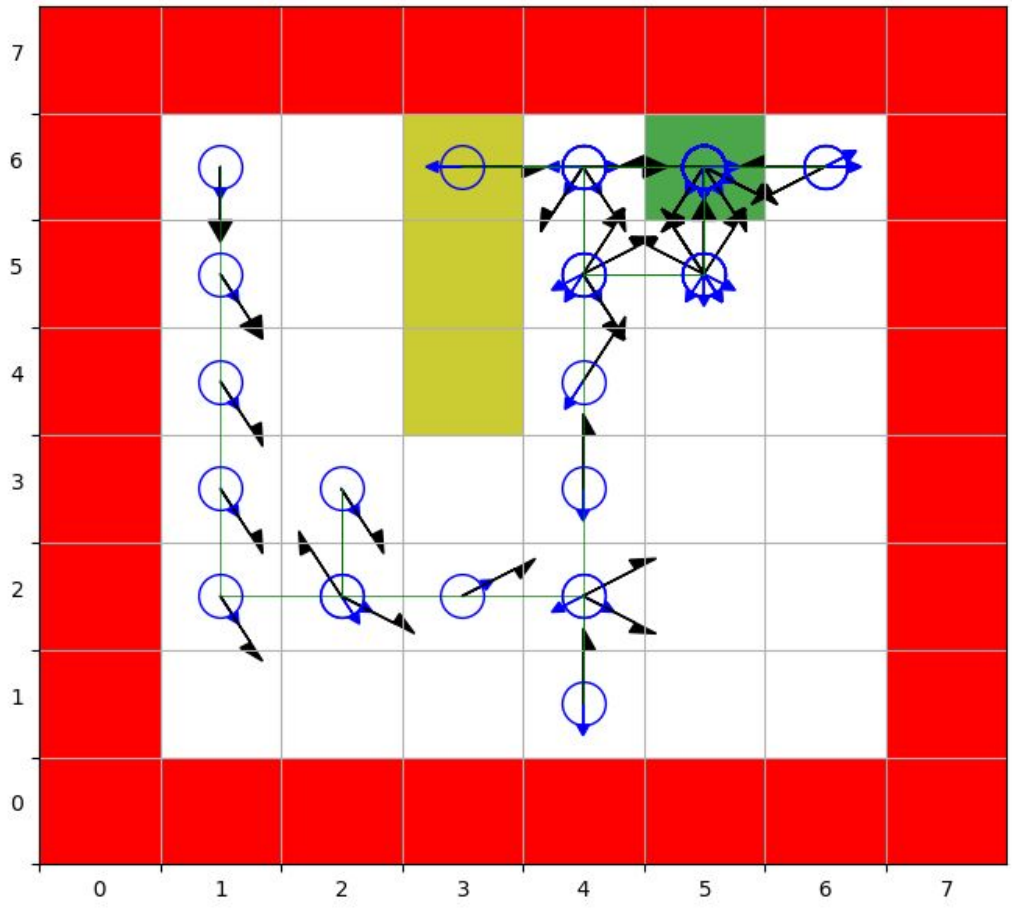
b. With the modified reward that the robot's heading must be down ($h = 6$) and no error probability, the recomputed trajectory is plotted below




$P_e$ = 0, Cumulative reward: 3.874
The policy iteration function takes 6 iterations to output the optimal policy.

Thus, the agent places itself facing 6 o'clock and then uses "backward" moves to get to the goal state, which now requires the heading to be 6. Again this is different from the earlier two policies in the deterministic case, but the value function is the same for the states visited by the trajectory.

With the modified reward that the robot's heading must be down (*h* = *6*) and error probability of 0.25, the recomputed trajectory is plotted below




$P_e$ = 0.25, Cumulative reward: -0.647

The policy iteration function takes 10 iterations to output the optimal policy.

Like earlier, the agent took more time in an uncertain environment than a deterministic one. This time, the agent also enters the yellow lane due to uncertainty. Therefore the cumulative reward is negative in this case. Moreover, since just getting to (5,6) is not enough, the agent goes out of and in this state again and again until the desired heading is not obtained. However, it does not make a move that may end him up in a red state, which is 10x worse than a yellow state.

c. Conclusions:

Some qualitative observations:

1) We don’t need to code a complex initial policy for faster convergence of policy iteration.
2) Policy iteration runs faster than value iteration in this environment.
3) It is sometimes an optimal action to not do anything in an uncertain environment even if we are not the goal state. For example, at (4,6,2), the agent does not move forward to avoid getting into red states and it does not move backward to avoid getting into yellow states. This is because the negative rewards are much larger than the positive ones.
4) There are multiple optimal policies in the deterministic case (they are equally good and the best).
5) See other comments for qualitative analysis of each optimal policy found.