EE209AS (Fall 2018)
Computational Robotics

Prof. Ankur Mehta
mehtank@ucla.edu

Problem set 2
Due 2pm Tue. Oct. 16, 2018

Objectives

The goal of this lab is to explore Markov Decision Processes (MDPs) to control a simple discretized robot. You will develop and implement a model of the robot behavior and use it to accomplish a prescribed task.

Deliverables

This project will require you to write code. If you do not have one already, create an account on `http://github.com`, and create a project for this class. Make sure the **well commented** code for this lab is committed and pushed, and submit a link to the repository. For some possible resources on git, see below.

You may work individually or in pairs on this assignment. Each person needs to submit their own solutions, but the team can submit common code. Indicate on your solutions who you worked with, and for each person identify 1) the specific contributions made by each, and 2) an aggregate percentage of the total work done.

Upload your solutions to gradescope.

Preliminaries

0(a). What is the link to your (fully commented) github repo for this pset?

0(b). Who did you collaborate with?

0(c). What were the specific contributions of each team member?

0(d). What was the aggregate % contributions of each team member?

1 Setup

Consider a simple robot in a 2D grid world of length L and width W. That is, the robot can be located at any lattice point $(x, y) : 0 \leq x < L, 0 \leq y < W; x, y \in \mathbb{N}$. At each point, the robot can face any of the twelve headings identified by the hours on a clock $h \in \{0 \dots 11\}$, where $h = 0$ represents 12 o'clock is pointing up (i.e. positive y) and $h = 3$ is pointing to the right (i.e. positive x).

In each time step, the robot can chose from several actions. Each singular action will consist of a movement followed by a rotation.

- The robot can choose to take no motion at all, staying still and neither moving nor rotating.
- Otherwise the robot can choose to either move "forwards" or "backwards".
  - This may cause a pre-rotation error, see below.
  - This will cause the robot move one unit in the direction it is facing, rounded to the nearest cardinal direction.

  That is, if the robot is pointing towards either 2, 3, or 4 and opts to move "forwards", the robot will move one unit in the $+x$ direction. Similarly, if the robot is pointing towards either 11, 0, or 1 and opts to move "backwards", it will move one unit in the $-y$ direction.
- After the movement, the robot can choose to turn left, not turn, or turn right. A left (counter-clockwise) turn will decrease the heading by 1 (mod 12); right (clockwise) will increase the heading by 1 (mod 12). The robot can also keep the heading constant.
- Attempting to move off of the grid will result in no linear movement, but the rotation portion of the action will still happen.

Note that aside from the at edges of the grids, the robot can only rotate if it also moves forwards or backwards; it can move without rotating though.

The robot has an error probability p_e: if the robot choses to move, it will *first* rotate by +1 or -1 (mod 12) with probability p_e each, before it moves. It will not pre-rotate with probability $1 - 2p_e$. Choosing to stay still (take no motion) will not incur an error rotation.

Create code to simulate this system. You may need to create objects to represent states $s \in S$ and/or actions $a \in A$. Note that the state s will later be used as an index to a matrix/array when computing policies and values.

1(a). Create (in code) your state space $S = \{s\}$. What is the size of the state space N_S?

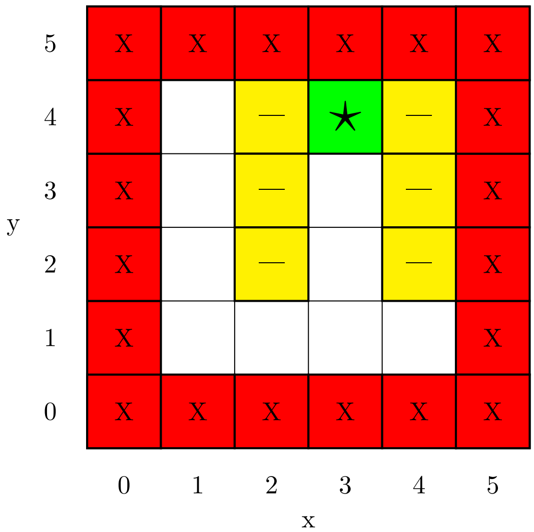
1(b). Create (in code) your action space $A = \{a\}$. What is the size of the action space N_A?

1(c). Write a function that returns the probability $p_{sa}(s')$ given inputs p_e, s, a, s'.

1(d). Write a function that uses the above to return a next state s' given error probability p_e, initial state s, and action a. Make sure the returned value s' follows the probability distribution specified by p_{sa}.

2 Problem

Consider the grid world shown below, with $L = W = 6$:



The rewards for each state are independent of heading angle (or action taken). The border states $\{x = 0, x = L, y = 0, y = W\}$ (red, marked X) have reward -100. The lane markers (yellow, marked —) have reward -1. The goal square (green, marked $\star$) has reward +1. Every other state has reward 0.

2(a). Write a function that returns the reward $R(s)$ given input s.

3 Policy iteration

Assume an initial policy π_0 of taking the action that gets you closest to the goal square. That is, if the goal is in front of you, move forward; if it is behind you, move backwards; then turn the amount that aligns your next direction of travel closer towards the goal (if necessary). If the goal is directly to your left or right, move forward then turn appropriately.

3(a). Create and populate a matrix/array that stores the action $a = \pi_0(s)$ prescribed by the initial policy π_0 when indexed by state s.

3(b). Write a function to generate and plot a trajectory of a robot given policy matrix/array π, initial state s_0, and error probability p_e.

3(c). Generate and plot a trajectory of a robot using policy π_0 starting in state $x = 1, y = 4, h = 6$ (i.e. top left corner, pointing down). Assume $p_e = 0$.

3(d). Write a function to compute the policy evaluation of a policy π. That is, this function should return a matrix/array of values $v = V^{\pi}(s)$ when indexed by state s. The inputs will be a matrix/array storing π as above, along with discount factor λ.

3(e). What is the value of the trajectory in 3(c)? Use $\lambda = 0.9$.

3(f). Write a function that returns a matrix/array π giving the optimal policy given a one-step lookahead on value V.

3(g). Combine your functions above in a new function that computes policy iteration on the system, returning optimal policy π^* with optimal value V^*.

3(h). Run this function to recompute and plot the trajectory and value of the robot described in 3(c) under the optimal policy π^*.

3(i). How much compute time did it take to generate your results from 3(h)? You may want to use your programming language's built-in runtime analysis tool.

4 Value iteration

4(a). Using an initial condition $V(s) = 0 \ \forall s \in S$, write a function (and any necessary subfunctions) to compute value iteration, again returning optimal policy π^* with optimal value V^*.

4(b). Run this function to recompute and plot the trajectory and value of the robot described in 3(c) under the optimal policy π^*. Compare these results with those you got from policy iteration in 3(h).

4(c). How much compute time did it take to generate your results from 4(b)? Use the same timing method as in 3(i).

5 Additional scenarios

5(a). Recompute the robot trajectory and value given initial conditions from 3(c) but with $p_e = 25\%$.

5(b). Assume the reward of +1 only applies when the robot is pointing down, e.g. $h \in \{5, 6, 7\}$ in the goal square; the reward is 0 otherwise. Recompute trajectories and values given initial conditions from 3(c) with $p_e \in \{0, 25\%\}$.

5(c). Qualitatively describe some conclusions from these scenarios.

Git Resources

- Getting started with GitHub: `https://guides.github.com/activities/hello-world/`
- Detailed documentation on how to use git: `https://git-scm.com/book/en/v2`
- Try git in the browser: `https://try.github.io/levels/1/challenges/1`

Name: Sahba Aghajani Pedram
UID: 504616252
Email: sahbaap@ucla.edu

Problem Set 2

Due 2pm Thursday Oct. 18, 2018

Preliminaries

0(a) `https://github.com/Sahbaap/ComputationalRoboticsUCLAEE209`

0(b) I did all by myself.

0(c) 100% myself.

0(d) 100% myself.

1 Setup

1(a) The state of the robot is composed of a tuple (x, y, h) where $0 \leq x < L$, $0 \leq y < W$, and $0 \leq h \leq 11$ and $x, y, h \in \mathbb{N}$. So the size of the state space is $N_s = L \times W \times 12$.
Note that I have defined the state class where I can instantiate the state of the robot with any (x,y,h). Moreover, I have define the environment class where I can instantiate the maze with given L and W.

State Space

```
% just initializtion of the state space to have zero values.
% as it can be seen, the size of the state space is 12*W*L.
% A more useful thing to define is the state class which can be found
% in this folder as well.
function sp = create_statespace(L,W)

    for k=1:1:12
        for i=1:1:L
            for j=1:1:W
                sp(i,j,k) = 0.0;
            end
        end
    end

end
```

State Class

```
% this state class generally defines the state of the robot
% which includes 'x' and 'y' position of the robot along with its heading 'h'
% The state space of this problem is created in the create_statespace
classdef state
    properties
        x = 0;   % 0,1,...,L-1 (L values)
        y = 0;   % 0,1,...,W-1 (W values)
        h = 0;   % 0,1,...,11  (12 values)
    end

    methods

        function obj = state(x,y,h)
            obj.x = x;
            obj.y = y;
            obj.h = h;
        end

        function r = eq(s1,s2,arg)
            if (nargin == 2)
                r = 0;
                if (s1.x == s2.x)
                    if (s1.y == s2.y)
                          r = 1;
                    end
                end
            elseif (strcmp(arg,'check_heading_too'))
                r = 0;
                if (s1.x == s2.x)
                    if (s1.y == s2.y)
                        if (s1.h == s2.h)
                            r = 1;
                        end
                    end
```

```
                    end
                end
            end

            function p = plot_state(obj)
                plot(obj.x-0.5,obj.y-0.5,'bo','MarkerSize',30)
                hold on
                line([obj.x-0.5 obj.x-0.5+0.5*cos(pi/2-obj.h*pi/6)],[obj.y-0.5 obj.y-0.5+0.5*sin(pi/2-obj.h*pi/6)])
            end

            function off = is_state_inside_environment(obj,L,W)

                off = 0;
                if (obj.x >= 0) && (obj.x <= L)
                    if (obj.y >= 0) && (obj.y <= W)
                        off = 1;
                    end
                end

            end

            function next_state = dynamics_deterministic(obj,a)
                next_state = state(obj.x,obj.y,obj.h);
                if (obj.h == 7 || obj.h == 6 || obj.h == 5)
                    next_state.y = next_state.y - a.t;

                elseif (obj.h == 8 || obj.h == 9 || obj.h == 10)
                    next_state.x = next_state.x - a.t;

                elseif (obj.h == 11 || obj.h == 0 || obj.h == 1)
                    next_state.y = next_state.y + a.t;

                elseif (obj.h == 2 || obj.h == 3 || obj.h == 4)
                    next_state.x = next_state.x + a.t;
                end
            end

        end
end
```

Environment Class

```
classdef environment
    properties
        L;
        W;
        Goal;
        fig_num;
    end
    methods
        function obj = environment(L,W,G,fig_num)
            obj.L = L;
            obj.W = W;
            obj.Goal.x = G.x;
            obj.Goal.y = G.y;
            obj.fig_num = fig_num;
        end

        function obj = sketch_environment(obj)
            figure(obj.fig_num)
            %filledCircle([obj.Goal.x-0.5,obj.Goal.y-0.5],0.3,1000,'g');
            hold on
            lane = [2.5 2.5 2.5 4.5 4.5 4.5; 2.5 3.5 4.5 2.5 3.5 4.5];
```

```
            border = [0.5 0.5 0.5 0.5 0.5 0.5 1.5 1.5 2.5 2.5 3.5 3.5 4.5 4.5 5.5 5.5 5.5 5.5 5.5 5.5; ...
                      0.5 1.5 2.5 3.5 4.5 5.5 0.5 5.5 0.5 5.5 0.5 5.5 0.5 5.5 0.5 1.5 2.5 3.5 4.5 5.5];

            for i=1:1:size(lane,2)
                %filledCircle([lane(1,i)-1,lane(2,i)-1],0.2,1000,'y');
                scatter(lane(1,i) - 1, lane(2,i) - 1, 2800, 'y', 'square', 'filled');
            end
            for i=1:1:size(border,2)
                %filledCircle([border(1,i)-1,border(2,i)-1],0.2,1000,'r');
                scatter(border(1,i) - 1, border(2,i) - 1, 2800, 'r', 'square', 'filled');
            end
            scatter(obj.Goal.x - 0.5, obj.Goal.y - 0.5, 2800, 'g', 'square', 'filled');
            axis square
            axis([-1 obj.L -1 obj.W])
            xticks([-0.5 0.5 1.5 2.5 3.5 4.5])
            xticklabels({'0','1','2','3','4','5'})
            yticks([-0.5 0.5 1.5 2.5 3.5 4.5])
            yticklabels({'0','1','2','3','4','5'})
            grid on
        end

    end


end
```

1(b) The size of the action space is $N_a = 7$. At each time step, the robot can (1) stay and not move, (2) move forward and turn right, (3) move forward and turn left, (4) move forward and not turn, (5) move backward and turn left, (6) move backward and turn right, and (7) move backward and not turn.

Action Class

```
% This is the action class which fully defines the robot actions.
% 't' represents the translation portion of the robot motion and
% 'r' represents the rotation portion of the robot motion. Note that
% out of all 9 mixtures, 2 of them (t=0,r=1) and (t=0,r=-1) is not
% acceptable. Hence the action space of the robot is composed of
% 7 actions:
% (1,0)     move forward, no turn
% (1,-1)    move forward, turn left
% (1,1)     move forward, turn right
% (-1,0)    move backward, no turn
% (-1,1)    move backward, turn right
% (-1,-1)   move backward, turn left
% (0,0)     stay still

classdef action
    properties
        t             % -1 (move backwards), 0 (not move), 1 (move forward
        r             % -1 (turn left),0 (not turn),1 (turn right)
    end
    methods
        function obj = action(t,r)
            obj.t = t;
            obj.r = r;
        end

        function obj = move_forward(obj)
            obj.t = 1;
        end

        function obj = move_backward(obj)
```

```
            obj.t = -1;
        end

        function obj = turn_right(obj)
            obj.r = 1;
        end

        function obj = turn_left(obj)
            obj.r = -1;
        end


    end
end
```

1(c)

Transition Probability

```
% This function takes as arguments the 'Pe' which is the probability of
% having pre-rotation error before motion, 's1' which the state that robot
% is in, and 'a' which is the action taken by robot, and returns the
% probability that the robot lands in state 's2'.
function pr = Transition_Probability(Pe,s1,a,s2)
    L = 5;
    W = 5;

    % no action taken by the robot
    if (a.t == 0) && (a.r == 0)

        if (s1.x == s2.x) && (s1.y == s2.y) && (s1.h == s2.h)
            pr = 1.0;
        else
            pr = 0.0;
        end

    % if one of the inputs are out of the grid
    elseif (is_state_inside_environment(s2,L,W) ~= 1 || is_state_inside_environment(s1,L,W) ~= 1)
        pr = 0;

    % the robot is in the middle of the maze
    else

        % if the action taken results in robot not falling off the grid
        s_vir = dynamics_deterministic(s1,a);

        if (is_state_inside_environment(s_vir,L,W) == 1)
            ss2 = state(s2.x,s2.y,s2.h);
            ss2.h = rem(s2.h - a.r + 12,12);

            ss1 = state(ss2.x,ss2.y,ss2.h);
            a.t = -a.t;
            ss1 = dynamics_deterministic(ss2,a);

            if eq(s1,ss1)
                if (s1.h == ss1.h)
                    pr = 1 - 2 * Pe;
                elseif (s1.h == rem(ss1.h - 1 + 12,12)) || (s1.h == rem(ss1.h + 1 + 12,12))
                    pr = Pe;
                else
                    pr = 0.0;
                end
            else
                pr = 0.0;
            end

        % if the action taken results in robot falling off the grid
        else
            ss1 = state(s2.x,s2.y,s2.h);
            ss1.h = rem(s2.h - a.r + 12,12);
            pr = 0;
            if eq(s1,ss1)
                if (s1.h == ss1.h)
                    pr = 1;
                end
            end
        end

    end
end
```

1(d)

Dynamics

```
% This function takes as arguments the 'Pe' which is the probability of
% having pre-rotation error before motion, 's1' which the state that robot
% is in, and 'a' which is the action taken by robot, and returns the state
% that robot will land in 'next_state'. The 'next_state' follows the
% distribution especified by P(s'|s,a).
% in summary what this function does is that it finds the most probable
% states: possible_state where (Transition_Probability(Pe,s1,a,possible_state)>0) and the
% associted probability with each. Then, we draw a random number from that
% distribution and select the next_state based on the radom number.
function next_state = f(Pe,s1,a)

next_state = state(0,0,0);
i = 0;

% all possible robot states after taking an action
for x=-1:1:1
    for y = -1:1:1
        for h = -2:1:2
            possible_state = state(s1.x + x, s1.y + y, rem(s1.h + h + 12,12));
            if (Transition_Probability(Pe,s1,a,possible_state) > 0)
                i = i + 1;
                most_possible_states(i) = possible_state;
                probability(i) = Transition_Probability(Pe,s1,a,possible_state);
            end
        end
    end
end

random_number = rand;

% the probability distribution changes when the robot is operating at the
% corner of the grid vs in the middle. This is because some states would
% become totally impossible to reach when the robot is at the border since
% it cannot leave the grid.
if size(probability,2) == 2
    t = probability(1)+probability(2);
    probability(1) = probability(1)/(t);
    probability(2) = probability(2)/(t);
end

if random_number < probability(1)
    next_state = most_possible_states(1);
elseif random_number < probability(1) + probability(2)
    next_state = most_possible_states(2);
elseif random_number < probability(1) + probability(2) + probability(3)
    next_state = most_possible_states(3);
end

end
```

2 Problem

2(a)

Reward

```
% This function takes as an argument the state that robot is in 's'
% and returns 'r' which is the reward associated with being in that state.
% Note that in this case, the reward is independent of the heading of the
% robot (except in problem 5) as well as independent of the action that robot takes.
function r = reward(s)
    % borders of the grid
    W = 6;
    L = 6;

    x = s.x;
    y = s.y;
    h = s.h;

    % robot is at the border states
    if (x==0) || (x==L-1) || (y==0) || (y==W-1)
        r = -100;
    % robot is on the lane markers
    elseif (x==2) || (x==4)
        if y ~= 1
            r = -10;
        else
            r = 0;
        end
    % robot is at the goal
    elseif (x==3) && (y==4)

  % only robot headings of 5,6,7 get reward 1
  %    if (h == 5 || h == 6 || h == 7)

  % heading does not matter for the +1 reward
      if (h == 1 || h == 2 || h == 3 || h == 4 || h == 5 || h == 6 || ...
      h == 7 || h == 8 || h == 9 || h == 10 || h == 11 || h == 0)
            r = 1;
        else
            r = 0;
        end
    else
        % otherwise
        r = 0;
    end

end
```

3 Policy iteration

3(a)

Generating initial policy π_0

```
% This function takes as an argument the state that robot is in 's'
% and returns initial single action called 'pi_initial' based on the
% the definition in Problem 3.
function pi_initial = generate_policy(s)
    % goal
    s_goal = state(3,4,0);

    % initiating action
    a_init = action(0,0);

    % this function especifies the relationship between the robot and goal
    rel_pos = state_to_goal_relative(s,s_goal);

    % the robot takes an action based on its relative position to the goal.
    % this action is defined based on the description given in the problem.

    % Target At Robot (TAR): robot does not do anything.
    if (strcmp(rel_pos,'TAR'))
        a = action(0,0);
        pi_initial = {s,a};

    % Target Left to Robot (TAR): robot moves forward and turns left
    elseif (strcmp(rel_pos,'TLR'))
        a_intermediate = move_forward(a_init);
        a = turn_left(a_intermediate);
        pi_initial = {s,a};

    % Target Right to Robot (TAR): robot moves forward and turns right
    elseif (strcmp(rel_pos,'TRR'))
        a_intermediate = move_forward(a_init);
        a = turn_right(a_intermediate);
        pi_initial = {s,a};

    % Target Front of Robot (TFR): robot moves forward with no turn.
    elseif (strcmp(rel_pos,'TFR'))
        a = move_forward(a_init);
        pi_initial = {s,a};

    % Target Back of Robot (TBR): robot moves backward with no turn.
    elseif (strcmp(rel_pos,'TBR'))
        a = move_backward(a_init);
        pi_initial = {s,a};

   % Target Front Right of Robot (TFRR) or Front Left of Robot (TFLR)
    elseif (strcmp(rel_pos,'TFRR') || strcmp(rel_pos,'TFLR'))

        % robot moves forward and depending on where it would lands,
        % decides where to go.
        a_intermediate = move_forward(a_init);
        s_next = f(0.0,s,a_intermediate);
        rel_pos_2 = state_to_goal_relative(s_next,s_goal);
        if (strcmp(rel_pos_2,'TFRR') || strcmp(rel_pos_2,'TBRR') || strcmp(rel_pos_2,'TRR'))
            a = turn_right(a_intermediate);
        elseif (strcmp(rel_pos_2,'TFLR') || strcmp(rel_pos_2,'TBLR') || strcmp(rel_pos_2,'TLR'))
            a = turn_left(a_intermediate);
        elseif (strcmp(rel_pos_2,'TFR') || strcmp(rel_pos_2,'TBR') || strcmp(rel_pos_2,'TAR'))
            a = action(a_intermediate.t,0); % no turn
        end
        pi_initial = {s,a};

```

```
     % Target Back Right of Robot (TBRR) or Back Left of Robot (TBLR)
    elseif (strcmp(rel_pos,'TBRR') || strcmp(rel_pos,'TBLR'))

        % robot moves backward and depending on where it would lands,
        % decides where to go.
        a_intermediate = move_backward(a_init);
        s_next = f(0.0,s,a_intermediate);
        rel_pos_2 = state_to_goal_relative(s_next,s_goal);
        if (strcmp(rel_pos_2,'TBRR') || strcmp(rel_pos_2,'TFRR') || strcmp(rel_pos_2,'TRR'))
            a = turn_right(a_intermediate);
        elseif (strcmp(rel_pos_2,'TBLR') || strcmp(rel_pos_2,'TFLR') || strcmp(rel_pos_2,'TLR'))
            a = turn_left(a_intermediate);
        elseif (strcmp(rel_pos_2,'TFR') || strcmp(rel_pos_2,'TBR') || strcmp(rel_pos_2,'TAR'))
            a = action(a_intermediate.t,0); % no turn
        end
        pi_initial = {s,a};
    end
end
```

```
% This function takes the state of the robot 's' and the state of the goal
% 's_goal' and returns in string the relationship between the robot and
% goal. For example 'TAR' means Target At Robot or in the other words the
% robot is at the target (goal) position.
function r = state_to_goal_relative(s,s_goal)

    if (eq(s,s_goal))
        r = 'TAR';  % target at the Robot position
    else
        eps = 0.001;
        angle_robot_to_target = atan2((s_goal.y-s.y),(s_goal.x-s.x));
        angle_robot_heading = (pi/2.0) - s.h*(pi/6.0);

        angle_heading_relative_to_target = atan2(sin(-angle_robot_heading+angle_robot_to_target),[cos(angle_robot_heading);sin(angle_robot_heading)]'*[cos(angle_robot_to_target);sin(angle_robot_to_target)]);

        if (angle_heading_relative_to_target > pi/2.0 - eps) && (angle_heading_relative_to_target < pi/2.0 + eps)
            r = 'TLR'; % target is in robot's left side;
        elseif (angle_heading_relative_to_target > -pi/2.0 - eps) && (angle_heading_relative_to_target < -pi/2.0 + eps)
            r = 'TRR'; % target is in robot's bottom side;
        elseif (angle_heading_relative_to_target > pi - eps) && (angle_heading_relative_to_target < pi + eps)
            r = 'TBR'; % target is in robot's back side;
        elseif (angle_heading_relative_to_target > -pi - eps) && (angle_heading_relative_to_target < -pi + eps)
            r = 'TBR'; % target is in robot's back side;
        elseif (angle_heading_relative_to_target > 0 - eps) && (angle_heading_relative_to_target < 0 + eps)
            r = 'TFR'; % target is in robot's front side;

        elseif (angle_heading_relative_to_target < pi/2.0 + eps) && (angle_heading_relative_to_target > 0.0 - eps)
            r = 'TFLR'; % target is in robot's front right side;
        elseif (angle_heading_relative_to_target < 0.0 + eps) && (angle_heading_relative_to_target > -pi/2.0 - eps)
            r = 'TFRR'; % target is in robot's front left side;
        elseif (angle_heading_relative_to_target > pi/2.0 - eps) && (angle_heading_relative_to_target < pi + eps)
            r = 'TBLR'; % target is in robot's front right side;
        elseif (angle_heading_relative_to_target < -pi/2.0 + eps) && (angle_heading_relative_to_target > -pi - eps)
            r = 'TBRR'; % target is in robot's front left side;
        end
    end

end
```

```
% This function takes as arguments takes no argument and returns the
% initial policy array that includes all the actions in initial_policy_generator
% array.

function initial_policy_generator = generate_initial_policy()
    L = 6;
    W = 6;
    initial_policy_generator = {};
    c = 0;
    for h=0:1:11
        for y=0:1:W-1
            for x=0:1:L-1
            s_now = state(x,y,h);
            state_index = get_index(s_now,L-1,W-1);
            v = generate_policy(s_now);
            initial_policy_generator{state_index} = v{2};
            end
        end
    end

end
```

3(b)

Trajectory plot

```
% This function takes as argument the policy 'policy_pi', the initial state of the
% robot 's', the pre-rotation probability 'Pe'. I personally also added
% fig_num so the plotted trajectory will be plotted in the figure with that
% number, and this does not in any sense changes the behaviour of the
% function. This function creates the trajectory of the robot based on the
% policy and given initial state and Pe and plots it in the figure with
% given fig_num.

function v = generate_plot_trajectory(policy_pi,s,Pe,fig_num) %policy_pi,

    s_now = state(s.x,s.y,s.h);
    s_goal = state(3,4,0);
    trajectory = {s_now};
    L = 6;
    W = 6;
    lambda = 0.9;
    v = 0;
    count = 0;
    % while the robot has not reached the target:
    while (eq(s_now,s_goal) ~= 1)
        state_index = get_index(s_now,L-1,W-1);
        a = policy_pi{state_index};
        v = v + lambda^(count)*reward(s_now)
        count = count + 1;
        s_now = f(Pe,s_now,a);
        trajectory{end+1} = s_now;
    end
    v = v + lambda^(count)*1;

    % creating the grid environment with given goal.
    goal = state(3,4,0);
    e = environment(5,5,goal,fig_num);
    sketch_environment(e)
    hold on

    % This plots the robot position with the line especifying the robot
    % orientation. The black line especifies the direction that robot is
    % moving.
    plot_trajectory(trajectory);
end
```

3(c)

Trajectory starting at (1,4,6)

The following sequence of actions were taken to move the robot from (1,4,6) tuple to the goal with (3,5,-) where only x and y positions of the goal matter. Fig. 1 shows the robot's trajectory under initial policy

1- Move Forward, Turn left	2- Move Forward, Turn left	3- Move Forward, Turn left
4- Move Forward, Turn left	5- Move Forward, Turn left	6- Move Forward, Turn left
7- Move Forward, Turn left	8- Move Forward, Turn left	9- Move Forward, Turn left
10- Move Forward, Turn left	11- Move Backward, Turn left	12- Move Forward, No Turn

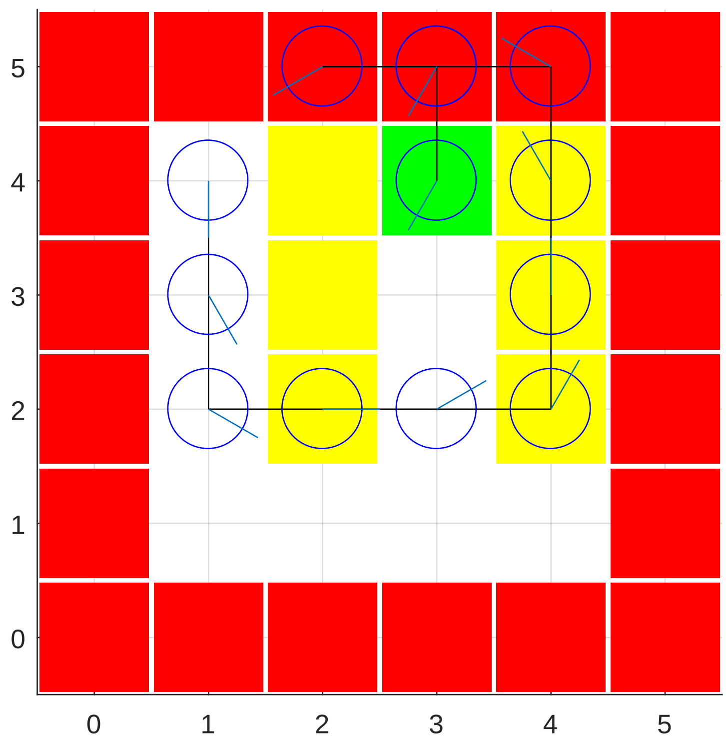


Figure 1: Robot's trajectory starting at state (1,4,6) under initial policy.

3(d)

Policy evaluation

```
% This function takes as input the policy 'pol_pi', the forgetting factor
% 'lambda'.
% The function outputs V values in the form of cell array. The input to the
% V values are states or state_index and the outputs of the V is double. We
% can access V data in the form of V{index}, e.g. V{23} = some_number.
% Of note, without max, evaluation of V is just a linear equation which is
% simple in matlab

function V = evaluate_policy(pol_pi,lambda) %(# of operations: num_iterate*S^2)
    L = 6;
    W = 6;
    Pe = 0.25;
    % (I-lambda*P)V = R  ====> V = inv(I-lambda*P)R

    % creating P matrix : P(s'|s,a): [(L)*(W)*12]*[(L)*(W)*12] matrix
    for idx_s1=1:1:(L)*(W)*12
        s1 = get_state(idx_s1,L-1,W-1);
        for idx_s2=1:1:(L)*(W)*12
            s2 = get_state(idx_s2,L-1,W-1);
            P(idx_s1,idx_s2) = lambda*Transition_Probability(Pe,s1,pol_pi{idx_s1},s2);
        end
        R(idx_s1) = reward(s1);
    end

    % A = (I-lambda*P)
    A = eye((L)*(W)*12) - P;

    % V = pinv(A)R
    V1 = A\R';

    % converting V from array to cell array
    for h=0:1:11
        for y=0:1:W-1
            for x=0:1:L-1
                s = state(x,y,h);
                state_index = get_index(s,L-1,W-1);
                V{state_index} = V1(state_index);
            end
        end
    end

end
```

3(e)

Evaluation of initial policy π_0

The value of the trajectory in 3(c) is -77.36. A plot of V values across the grid with robot's heading (h) be equal to 6, i.e. robot is pointing down, is shown in Fig. 2. Of note, it is expected that the final value of V at goal should be $\frac{1}{1-0.9} = 10$ at infinity time limit which it is.

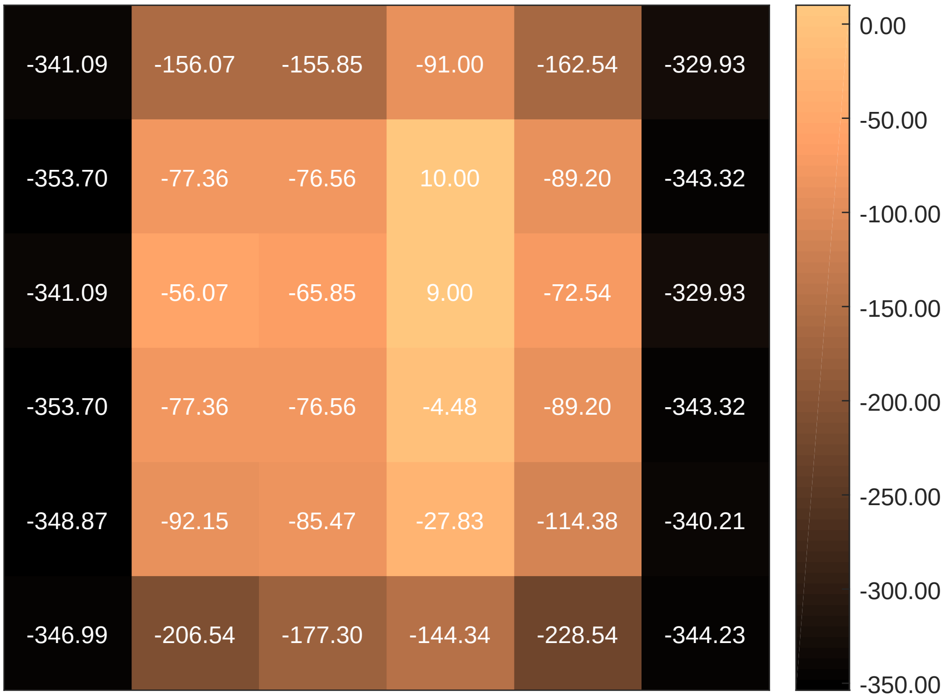


Figure 2: V values of different robot's states (heading = 6) under initial policy.

3(f)

Policy update

```
% This function takes as an argument the V values across the state space
% and returns the optimal policy based on one-step look ahead on value V.
function optimal_one_step_lookahead_policy = update_policy(V) %(AS^2 # of operations)

    optimal_one_step_lookahead_policy = {};
    L = 6;
    W = 6;
    %discount
    lambda = 0.9;

    Pe = 0.25;

    %iterating through states (S # of operations)
    for h=0:1:11
        for y=0:1:W-1
            for x=0:1:L-1
                s = state(x,y,h);

                % iterating through actions for given state (A number of operations)
                index_state_now = get_index(s,L-1,W-1);
                Q_max = -10000;

                for (r=-1:1:1)
                    for (t=-1:1:1)
                        if (t ~= 0)%~(t==0 && r ~= 0)

                             % limited heading
%                            if eq(s,state(3,4,5),'check_heading_too') || eq(s,state(3,4,6),'check_heading_too') || eq(s,state(3,4,7),'check_heading_too')
%                                t = 0;
%                                r = 0;
%                            end

                            % All heading
                            if eq(s,state(3,4,5))
                                t = 0;
                                r = 0;
                            end

                            a = action(t,r);
                            %(#S operations)
                            q_val = Q(V,a,s,Pe,lambda);
                            if (q_val > Q_max)  % finding Q_max to update policy
                                optimal_one_step_lookahead_policy{index_state_now} = a;
                                Q_max = q_val;
                            end

                        end
                    end
                end % actions

            end
        end
    end % states

end
```

3(g)

Policy iteration

```
% This function takes as argument 'lambda' which is the discount factor
% and return the optimal policy 'optimal_policy_PI' and optimal value
% 'optimal_value_PI'. This function also return the history of V values
% 'V_history_PI' and histoy of policies 'P_history_PI'.
function [optimal_policy_PI,optimal_value_PI] = policy_iteration(lambda)

    L = 6;
    W = 6;
    s_goal = state(3,4,0);

    eps = 0.01;
    V_diff = 1000;

    idx_policy_iteration = 0;

    policy_init = generate_initial_policy();

    % setting up the first column of P_history_PI to initial policy we
    % created.
    for h=0:1:11
        for y=0:1:W-1
            for x=0:1:L-1
                s_index = get_index(state(x,y,h),L-1,W-1);
                optimal_policy_PI{s_index} = policy_init{s_index};
            end
        end
    end


    % for each iteration, we calculate V given P and update P given V
    while (V_diff > eps)
        idx_policy_iteration = idx_policy_iteration + 1

        %calculate V
        optimal_value_PI = evaluate_policy(optimal_policy_PI,lambda);

        %calculate V diff
        if (idx_policy_iteration > 2)
            err(idx_policy_iteration - 1) = calculate_V_diff(optimal_value_PI,
    optimal_value_PI_Previous);    % between V{i} and V{i+1}
            V_diff = calculate_V_diff(optimal_value_PI,optimal_value_PI_Previous)
        end

        % update V previous
        if (idx_policy_iteration > 1)
            for h=0:1:11
                for y=0:1:W-1
                    for x=0:1:L-1
                    s_index = get_index(state(x,y,h),L-1,W-1);
                    optimal_value_PI_Previous{s_index} = optimal_value_PI{s_index};
                    end
                end
            end
        end

        %calculate P*
        optimal_policy_PI = update_policy(optimal_value_PI);

        %calculate V* from P*
        if (V_diff < eps)
            total_num_policy_iteration = idx_policy_iteration;
            optimal_value_PI = evaluate_policy(optimal_policy_PI,lambda);
```

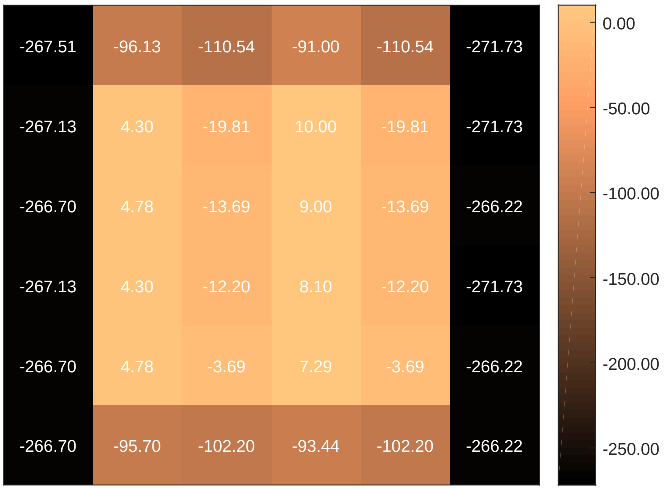
```
                end

        end

end
```

3(h)

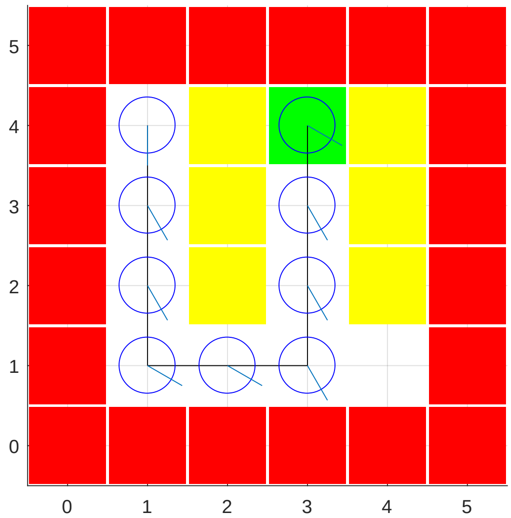
Optimal robot trajectory and value starting at (1,4,6) under P^* and V^*

Under this optimal policy, the V^* of the robot at state (1,4,6) is 3.48. Note that the heat of V is for robot states whose headings are 6. The π^* of the robot is:

1- Move Forward, Turn left 2- Move Forward, No Turn 3- Move Forward, Turn left
4- Move Forward, No Turn 5- Move Forward, Turn right 6- Move Backward, No Turn
7- Move Backward, No Turn 8- Move Backward, Turn left



(a) Optimal Policy (P^*) obtained from policy iteration algorithm.



(b) Optimal Value (V^*) obtained from policy iteration algorithm.

Figure 3: Question 3(h).

3(i) On my Lenovo ThinkPad laptop with 64-bit windows 10, 2.5 GHz CPU and 8 GB RAM, it took 272 (s) for policy iteration to run.

4 Value iteration

4(a)

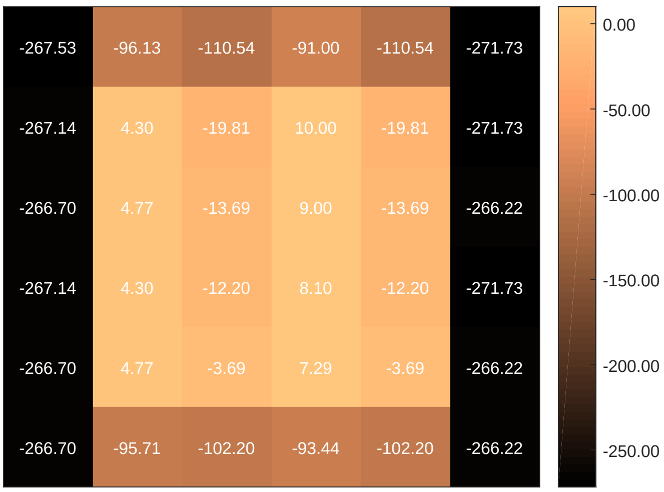
Value iteration

```
% This function takes as argument 'lambda' which is the discount factor
% and return the optimal policy 'optimal_policy_VI' and optimal value
% 'optimal_value_VI'. This function also return the history of V values
% 'V_history_VI' and histoy of policies 'P_history_VI'.
function [optimal_policy_VI,optimal_value_VI] = value_iteration(lambda)  %(#AS^2 operations)

    L = 6;
    W = 6;

    V_diff = 1000;
    idx_policy_iteration = 0;
    eps = 0.01;

    Pe = 0.25;

    % initializing V and P
    for idx_v=1:1:12*(L)*(W)
        optimal_value_VI{idx_v} = 0;
        optimal_policy_VI{idx_v} = action(0,0);  % do nothing
    end

    % iterating until convergance: V_diff is the difference between V{i+1}
    % and V{i}
    while (V_diff > eps)

        idx_policy_iteration = idx_policy_iteration + 1

        %iterating through states space
        for h=0:1:11
            for y=0:1:W-1
                for x=0:1:L-1

                    % select state
                    s = state(x,y,h);

                    index_state_now = get_index(s,L-1,W-1);
                    Q_max = -10000;

                    % iterating through actions space
                    % finding max across all actions
                    for (r=-1:1:1)
                        for (t=-1:1:1)
                            if (t ~= 0)%~(t==0 && r ~= 0)%

                                % limited heading
%                                 if eq(s,state(3,4,5),'check_heading_too') || eq(s,state(3,4,6),'check_heading_too') || eq(s,state(3,4,7),'check_heading_too')
%                                     t = 0;
%                                     r = 0;
%                                 end

                                % all heading
                                if eq(s,state(3,4,5))
                                    t = 0;
                                    r = 0;
                                end

                                % select action
                                a = action(t,r);

```

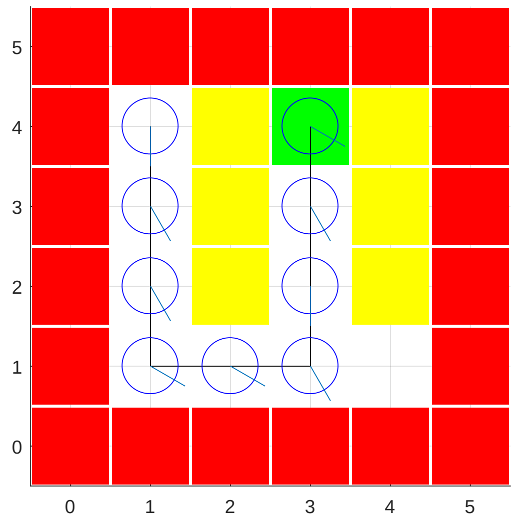
```
                                % get Q for given state and action
                                q_val = Q(optimal_value_VI,a,s,Pe,lambda);

                                % find Q max and assign to V
                                % update the action that caused Q max
                                if (q_val > Q_max)
                                    optimal_value_VI{index_state_now} = q_val;
                                    optimal_policy_VI{index_state_now} = a;
                                    Q_max = q_val;
                                end

                            end
                        end
                    end % actions

                end
            end
        end % states


        % calculate the difference between V{i} and V{i+1}
        if idx_policy_iteration > 2
            V_diff = calculate_V_diff(optimal_value_VI_Previous,optimal_value_VI)
        end


        % update optimal_value_VI_Previous
        if (idx_policy_iteration > 1)
            for h=0:1:11
                for y=0:1:W-1
                    for x=0:1:L-1
                        s_index = get_index(state(x,y,h),L-1,W-1);
                        optimal_value_VI_Previous{s_index} = optimal_value_VI{s_index};
                    end
                end
            end
        end


    end

end
```

4(b) The V^* for this policy with initial s_0 at (1,4,6) is 4.30 – see Fig. 4. As it can be seen, both value iteration and policy iteration algorithms yield to identical results for both V^* but with slightly different π^*. This is totally expected as both methods solved the same underlying Bellman equation for V^*. The optimal policy, as it happened here, is not unique however. This means that several optimal policies can result in the same optimal value function. Note that the computational complexity of both methods is also $N_A * N_S^2$. But since they have slightly different approaches, sometimes one converges faster than the other and vice versa. In fact the constant that multiplies to this complexity is different.

The optimal policy in this case is:
1- Move Forward, Turn left 2- Move Forward, No Turn 3- Move Forward, Turn left
4- Move Forward, No Turn 5- Move Forward, Turn right 6- Move Backward, Turn right.
7- Move Backward, Turn left 8- Move Backward, Turn left



(a) Optimal Policy (P^*) obtained from value iteration algorithm.



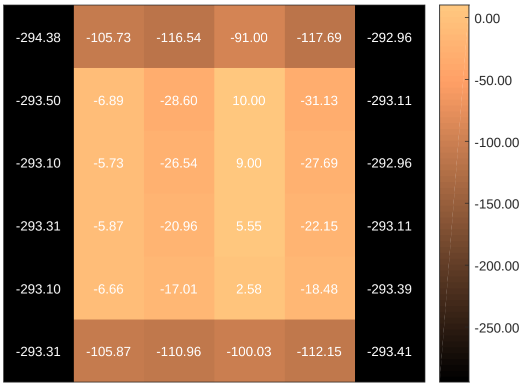
(b) Optimal Policy (P^*) obtained from value iteration algorithm.

Figure 4: Question 4(b).

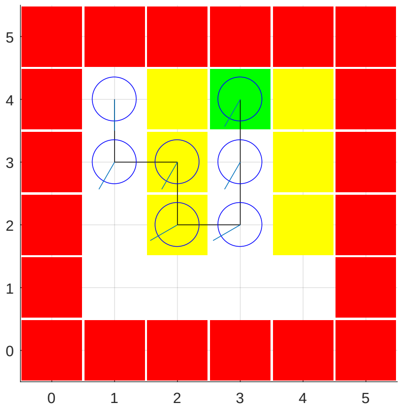
4(c) On my Lenovo ThinkPad laptop with 64-bit windows 10, 2.5 GHz CPU and 8 GB RAM, it took 571 (s) for this value iteration to run. As it can be seen the value iteration took more than two times to converge compared to the policy iteration.

5 Additional scenarios

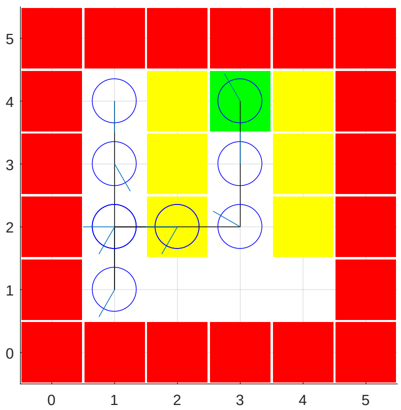
5(a) With $P_e = 0.25$, The optimal expected V (V^*) for the robot at state (1,4,6) os -6.86. Note that this means that if we run, for example, a Monte Carlo simulation and average the V values under this optimal policy starting at (1,4,6), this averaged V value will be -6.86. However, each time we run the simulation we are going to get V values that are different than -6.86 and their average will eventually converge to -6.86.
In terms of actions also, note that the robot's trajectory changes for every realization of the policy. This means that every time we run our matlab program it gives us different robot trajectory due to the pre-rotation error in our robot which is expressed in P_e.
I have included a plot of expected V values across robot states with heading equal to 6 (see Fig. 5). At the same time I have included different realizations of robot trajectory along with the value of that specific trajectory under the optimal policy.



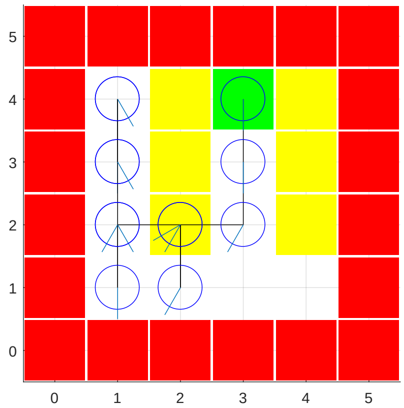
(a) Expected Optimal Value (V^*) under $P_e = 0.25$



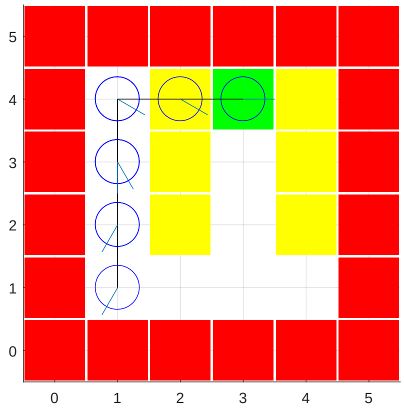
(b) V_1 with value of -14.86



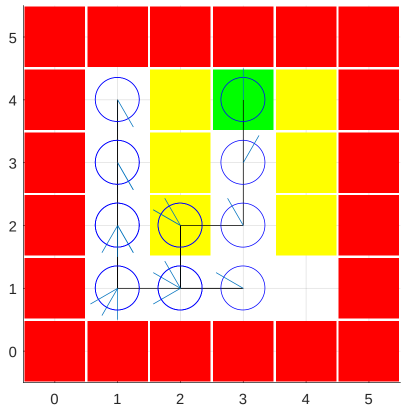
(c) V_2 with value of -10.34



(d) V_3 with value of -8.37



(e) V_4 with value of -3.52

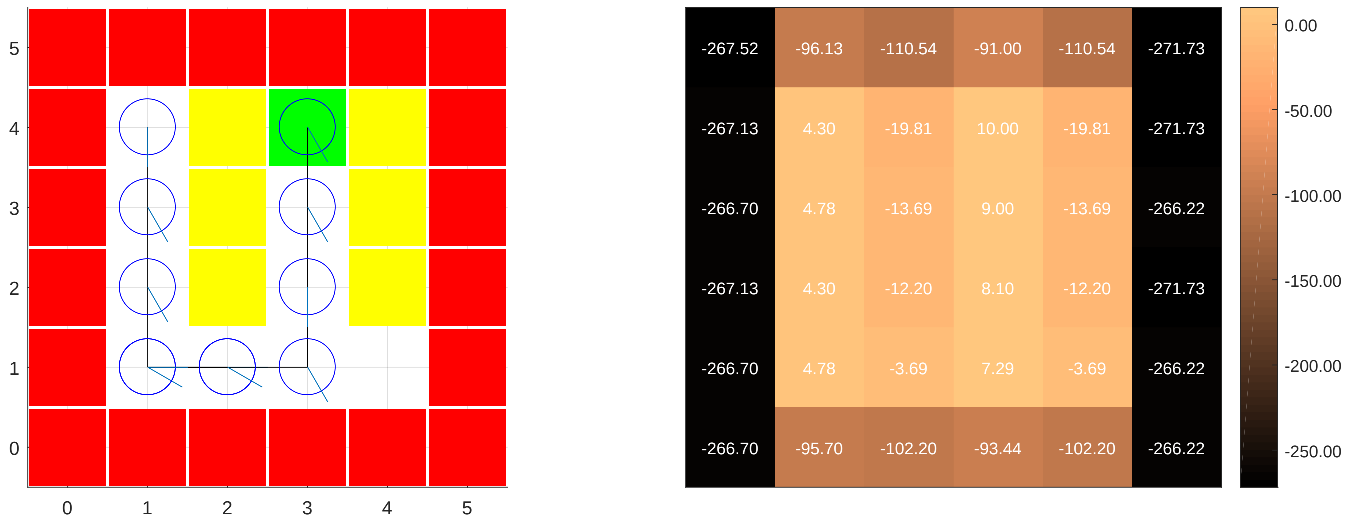


(f) V_5 with value of -4.97

Figure 5: Question 5(a).

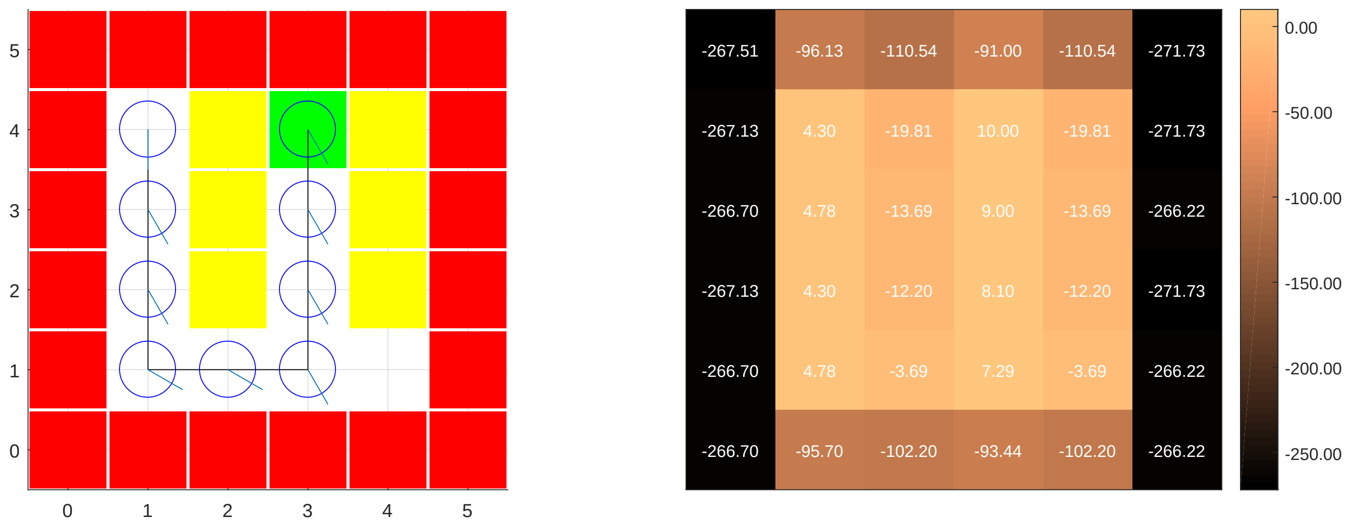
5(b) In this section, the robot only gets the final reward with specific headings (5,6,7) rather than all headings. Since the final robot state in previous sections under optimal policy was in the heading 5, when I ran the value iteration or policy iteration algorithms, the same optimal policy and value for the robot at (1,4,6) is obtained (as it can be seen in the Figures.). In this case in took 705 (s) for the value iteration algorithm to converge while it took 422 (s) for the policy iteration to converge. Again value iteration and policy suggests two slightly different trajectories for robot to follow starting at (1,4,6). Fig. 6 and Fig. 7 show these plots.

When $P_e = 0.25$, V^* and P^* change compared to the case where the heading of the robot did not matter at goal point of (3,4). Fig 8 shows the V map at robot heading of 6 and as it can be seen, the V^* of point (1,4,6) which is the expected value of sum of the discounted rewards is -8.30 as opposed to the other case (problem 5a) which was -6.89. In this case it took 798 (s) for the value iteration algorithm to converge and 400 (s) for the policy iteration algorithm.



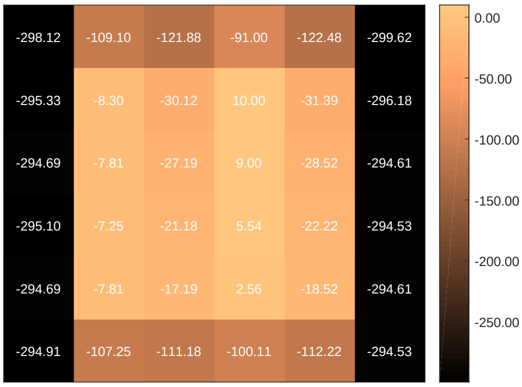
(a) Optimal Policy (P^*) (b) Optimal Value (V^*)

Figure 6: Question 5(b): V^* and P^* for limited heading +1 reward with $P_e = 0$ under value iteration.

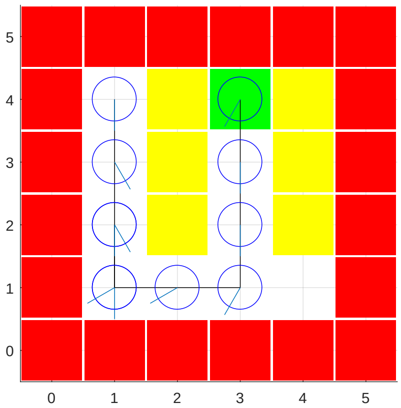


(a) Optimal Policy (P^*) (b) Optimal Value (V^*)

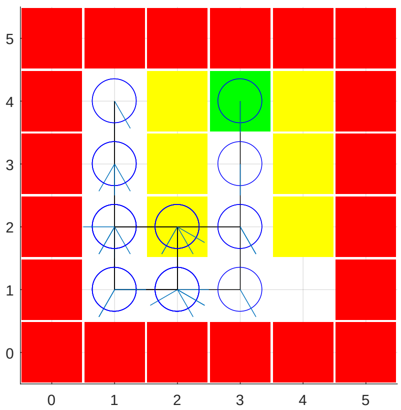
Figure 7: Question 5(b): V^* and P^* for limited heading +1 reward with $P_e = 0$ under policy iteration.



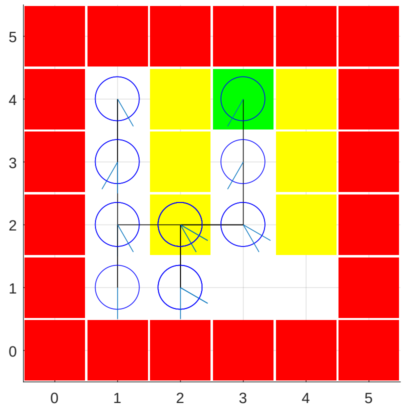
(a) Expected Optimal Value (V^*) under $P_e = 0.25$



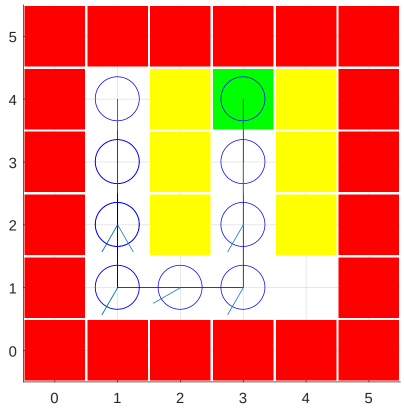
(b) V_1 with value of 0.35



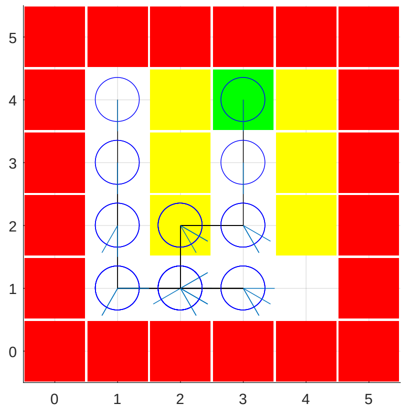
(c) V_2 with value of -9.50



(d) V_3 with value of -16.24



(e) V_4 with value of 0.23



(f) V_5 with value of -6.00

Figure 8: Question 5(b).

5(c) The overal time that it took either policy and value iteration algorithms under different heading rewards and p_e is shown in Table 1. It can be seen that the value iteration algorithm overall took longer compared to the policy iteration to converge. Also it can be seen that the limited heading reward took longer to converge than the one where heading does matter for the final reward. Also, it is witnessed that when $P_e = 0.25$ it took longer to converge for the algorithms compared to the $P_e = 0$. This is also expected since the uncertainty makes the robot to explore more on average which in turn makes the algorithm to converge slower.

	All head. (Pr = 0)	All head. (Pr = 0.25)	limited head. (Pr = 0)	limited head. (Pr = 0.25)
Policy iteration	272 (s)	373 (s)	422 (s)	477 (s)
Value iteration	571 (s)	597 (s)	705 (s)	798(s)

Table 1: time to converge for value and policy iteration algorithms.

An interesting point for when P_e was not zero is that for every realization of robot starting at (1,4,6), the trajectory of the robot will be different due to uncertainty from P_e yet the V^* and P^* remain the same, since V^* is in fact averages over all the possible realizations and not individual ones. This is different when P_e is zero since for every realization of the system, the trajectory of the robot and V value for that trajectory remains the same.

When one of the actions that robot can take is to stay still and do nothing, then for some robot trajectories where it needs to pass through those states the robot will stay and cannot hit the goal. Two options at least are available. One is to put some limits on the number of iterations for either value iteration or policy iteration so that if robot did not converge the program will be terminated. In my opinion this is kind of not nice/realistic given the fact that our robot does want to hit the goal and also has other actions to take. The second option is that whenever the optimum policy was to stay still, we do not pick that and find second best action that would maximize the Bellman equation. This is nice since at least staying still would not some how break the program and the robot will hit the goal.

Another interesting point here was that even though both the policy and value iteration algorithms converge to the same V^*, they did not converge to the same optimal policy. This should not be surprising as they solve the same Bellman equation for V^* but it was witnessed that actually different policies can acheive the same V^* values which was an interesting point to learn from this homework.

Another observation is that under some initial conditions and some policies for when $P_e = 0$, the robot would not converge to the goal, for example it could infinitely switch back and forth between two states. This brings up the point that in general this can happen with some specific MDPs. In fact the question is what happens when the agent bounces between two or more states where the actions some states would cancel out the actions in other states and make the agent not able to skip some inner manifolds within the state space. So in this case we would have that the robot would not converge. As a result what I think happens is that in fact having some sort of uncertainty into our dynamics (P_e here) improves the overal convergence behavior of the robot. This is somehow helps the robot to escape the local minimum or some lower dimention optimal manifold within the robot's state space. I believe very similar underlying idea is used for methods like stochastic gradient decent where uncertainty improves escaping unwanted local minimums.