Lab 1

Computational Robotics

Yuki Shirai

UID: 605228207

Preliminaries

0(a): https://github.com/YukiShirai/ComputationalRobotics/tree/master/lab1

0(b): I did it for myself.

0(c): 100 %

1. MDP system

   1(a) and (b): I created my state space S and action space A in the following code. The size of the state system is 768, and the that of the action space is 7.

```
# EE Computational Robotics: lab 1
#
# Problem 1

# Problem 1(a), (b)

import numpy as np
import matplotlib.pyplot as plt
import math
import time

# Define constants

L = 8
W = 8

# State space: 8 * 8 * 12
s = np.empty([8, 8, 12])
S = []
for i in range(8):
    for j in range(8):
        for k in range(12):
            temp = np.array([i, j, k])
            S.append([temp])
# S = [i,j,k] describes the state space. i represents x position, j
represents y position, and k represents heading angle.
S = np.array(S)
# A = [i,j] represents action of a robot, where i == +1/-1 represents
forward/backward linear motion.
# j == +1/-1 represents clockwise/counter-clockwise rotational motion.
A = np.array([[1, 1, 1, -1, -1, -1, 0], [-1, 0, 1, -1, 0, 1, 0]])
nA = len(A[0])
print('The size of the state space is', s.size)
print('The size of the action space is', len(A[0]))
```

1(c): My function that returns the transition probability is as follows:

```
# Problem 1(c)
def calc_prob(pe, s, a, s_prime):
    """
    Calculate the transition probability
    :param pe:error probability
    :param s:state
    :param a:action
    :param s_prime:transit state
    :return:transition probability

    # Idea: create array that deal with all possible state transitions after
action takes.
    # Then, calculate the possibility based on the current & future state and
action.

    # P_trans [i] = [(current intended action), (error action left), (error
action right)]
    # i: heading angle
    # current intended action = [(intended linear movement), (heading
direction), possibility of action]
    linear movement: +x [1,0], -x [-1,0], +y [0,1], -y [0,-1]
    heading direction: 0:1:11


    """

    P_trans = {}
    Prob_tmp = {}
    P_trans[0] = [([0, 1], 0, 1 - 2 * pe), ([0, 1], 11, pe), ([0, 1], 1, pe)]
    P_trans[1] = [([0, 1], 1, 1 - 2 * pe), ([0, 1], 0, pe), ([1, 0], 2, pe)]
    P_trans[2] = [([1, 0], 2, 1 - 2 * pe), ([0, 1], 1, pe), ([1, 0], 3, pe)]
    P_trans[3] = [([1, 0], 3, 1 - 2 * pe), ([1, 0], 2, pe), ([1, 0], 4, pe)]
    P_trans[4] = [([1, 0], 4, 1 - 2 * pe), ([1, 0], 3, pe), ([0, -1], 5, pe)]
    P_trans[5] = [([0, -1], 5, 1 - 2 * pe), ([1, 0], 4, pe), ([0, -1], 6,
pe)]
    P_trans[6] = [([0, -1], 6, 1 - 2 * pe), ([0, -1], 5, pe), ([0, -1], 7,
pe)]
    P_trans[7] = [([0, -1], 7, 1 - 2 * pe), ([0, -1], 6, pe), ([-1, 0], 8,
pe)]
    P_trans[8] = [([-1, 0], 8, 1 - 2 * pe), ([0, -1], 7, pe), ([-1, 0], 9,
pe)]
    P_trans[9] = [([-1, 0], 9, 1 - 2 * pe), ([-1, 0], 8, pe), ([-1, 0], 10,
pe)]
    P_trans[10] = [([-1, 0], 10, 1 - 2 * pe), ([-1, 0], 9, pe), ([0, 1], 11,
pe)]
    P_trans[11] = [([0, 1], 11, 1 - 2 * pe), ([-1, 0], 10, pe), ([0, 1], 0,
pe)]

    if a[0] == 1:  # Forward motion
        for P_tmp in P_trans[s[2]]:
            x_goal = s[0] + P_tmp[0][0]
            y_goal = s[1] + P_tmp[0][1]
            h_goal = (a[1] + P_tmp[1]) % 12
            # At edges of the grids, the robot can only rotate:
            if x_goal < 0 or x_goal > L - 1:
                x_goal = s[0]
            if y_goal < 0 or y_goal > W - 1:
                y_goal = s[1]

            Prob_tmp[x_goal, y_goal, h_goal] = P_tmp[2]
```

```
    if a[0] == -1:  # Backward motion
        for P_tmp in P_trans[s[2]]:
            x_goal = s[0] - P_tmp[0][0]
            y_goal = s[1] - P_tmp[0][1]
            h_goal = (a[1] + P_tmp[1]) % 12
            # At edges of the grids, the robot can only rotate:
            if x_goal < 0 or x_goal > L - 1:
                x_goal = s[0]
            if y_goal < 0 or y_goal > W - 1:
                y_goal = s[1]

            Prob_tmp[x_goal, y_goal, h_goal] = P_tmp[2]

    if a[0] == 0:  # No linear motion, i.e. no error
        x_goal = s[0]
        y_goal = s[1]
        h_goal = (a[1] + s[2]) % 12
        Prob_tmp[x_goal, y_goal, h_goal] = 1

    if s_prime in Prob_tmp.keys():
        return Prob_tmp[s_prime]
    else:
        return 0
```

1(d): My function is as follows:

```
def calc_next_state(pe, s, a):
    """
    Calculate a next state
    :param pe:error probability
    :param s:state
    :param a:action
    :return:next state that follows the pdf

    # Idea: using random values, I calculate the next state

    """
    # s = list(a)
    P_trans = {}
    Prob_tmp = {}
    next_state = {}
    P_trans[0] = [([0, 1], 0, 1 - 2 * pe), ([0, 1], 11, pe), ([0, 1], 1, pe)]
    P_trans[1] = [([0, 1], 1, 1 - 2 * pe), ([0, 1], 0, pe), ([1, 0], 2, pe)]
    P_trans[2] = [([1, 0], 2, 1 - 2 * pe), ([0, 1], 1, pe), ([1, 0], 3, pe)]
    P_trans[3] = [([1, 0], 3, 1 - 2 * pe), ([1, 0], 2, pe), ([1, 0], 4, pe)]
    P_trans[4] = [([1, 0], 4, 1 - 2 * pe), ([1, 0], 3, pe), ([0, -1], 5, pe)]
    P_trans[5] = [([0, -1], 5, 1 - 2 * pe), ([1, 0], 4, pe), ([0, -1], 6,
pe)]
    P_trans[6] = [([0, -1], 6, 1 - 2 * pe), ([0, -1], 5, pe), ([0, -1], 7,
pe)]
    P_trans[7] = [([0, -1], 7, 1 - 2 * pe), ([0, -1], 6, pe), ([-1, 0], 8,
pe)]
    P_trans[8] = [([-1, 0], 8, 1 - 2 * pe), ([0, -1], 7, pe), ([-1, 0], 9,
pe)]
    P_trans[9] = [([-1, 0], 9, 1 - 2 * pe), ([-1, 0], 8, pe), ([-1, 0], 10,
pe)]
    P_trans[10] = [([-1, 0], 10, 1 - 2 * pe), ([-1, 0], 9, pe), ([0, 1], 11,
pe)]
    P_trans[11] = [([0, 1], 11, 1 - 2 * pe), ([-1, 0], 10, pe), ([0, 1], 0,
pe)]

    # Create random values that follows uniform pdf
```

```
    delta = np.random.uniform(0, 1)
    # No error case
    if delta <= 1 - 2 * pe:
        P_tmp = P_trans[s[2]][0]
    elif 1 - 2 * pe < delta and delta < 1 - pe:  # Error rotation to left
        P_tmp = P_trans[s[2]][1]
    else:  # Error rotation to right
        P_tmp = P_trans[s[2]][2]

    if a[0] == 1:  # Forward motion

        x_goal = s[0] + P_tmp[0][0]
        y_goal = s[1] + P_tmp[0][1]
        h_goal = (a[1] + P_tmp[1]) % 12
        # At edges of the grids, the robot can only rotate:
        if x_goal < 0 or x_goal > L - 1:
            x_goal = s[0]
        if y_goal < 0 or y_goal > W - 1:
            y_goal = s[1]

        next_state = [x_goal, y_goal, h_goal]

    if a[0] == -1:  # Backward motion

        x_goal = s[0] - P_tmp[0][0]
        y_goal = s[1] - P_tmp[0][1]
        h_goal = (a[1] + P_tmp[1]) % 12
        # At edges of the grids, the robot can only rotate:
        if x_goal < 0 or x_goal > L - 1:
            x_goal = s[0]
        if y_goal < 0 or y_goal > W - 1:
            y_goal = s[1]

        next_state = [x_goal, y_goal, h_goal]

    if a[0] == 0:  # No linear motion, i.e. no error
        x_goal = s[0]
        y_goal = s[1]
        h_goal = (a[1] + s[2]) % 12
        next_state = [x_goal, y_goal, h_goal]

    return next_state
```

2. Planning problem

2(a): My reward function is as follows:

```
def reward(s):
    """
    Calculate reward based on the given state
    :param s:given state
    :return: reward given state
    """
    reward = 0 # Initialization
    if (s[0] in [0, 7]) or (s[1] in [0, 7]): # The border states
        reward = -100
    if s[0] == 3 and s[1] in [4, 5, 6]: # The lane markers
        reward = -10
    if s[0] == 5 and s[1] == 6: # The goal square
        reward = 1

    return reward
```

3. Policy iteration

3(a): I created and populated a matrix that stores the action prescribed by the initial policy.

```
for s in S:
    # For all state, let's calculate the length between the current location
and the goal location. In addition,
    # let's calculate the orientation difference, too.
    tmp_action = [0, 0]
    diff = goal_xy_state - s[0][0:2] #Linear difference
    diff_h = goal_h_state - s[0][2] #Rotational difference

    # If a robot is at goal location
    if diff[0] == [0] and diff[1] == [0]:
        tmp_action[0] = 0
    # If a robot is heading north:
    if s[0][2] in [11, 0, 1]:
        if diff[0] >= 0 and diff[1] >= 0: # If a location of a robot is
south-west from a goal
            tmp_action[0] = 1 #Forward
        if diff[0] >= 0 and diff[1] < 0: # If a location of a robot is north-
west from a goal
            tmp_action[0] = -1 #Backward
        if diff[0] < 0 and diff[1] >= 0: # If a location of a robot is south-
east from a goal
            tmp_action[0] = 1 #Forward
        if diff[0] < 0 and diff[1] < 0: # If a location of a robot is north-
east from a goal
            tmp_action[0] = -1 #Backward
    if s[0][2] in [2, 3, 4]:
        if diff[0] >= 0 and diff[1] >= 0: # If a location of a robot is
south-west from a goal
            tmp_action[0] = 1 #Forward
        if diff[0] >= 0 and diff[1] < 0: # If a location of a robot is north-
west from a goal
            tmp_action[0] = 1 #Forward
        if diff[0] < 0 and diff[1] >= 0: # If a location of a robot is south-
east from a goal
            tmp_action[0] = -1
        if diff[0] < 0 and diff[1] < 0: # If a location of a robot is north-
east from a goal
            tmp_action[0] = -1
    if s[0][2] in [5, 6, 7]:
        if diff[0] >= 0 and diff[1] >= 0: # If a location of a robot is
south-west from a goal
            tmp_action[0] = -1
        if diff[0] >= 0 and diff[1] < 0: # If a location of a robot is north-
west from a goal
            tmp_action[0] = 1 #Forward
        if diff[0] < 0 and diff[1] >= 0: # If a location of a robot is south-
east from a goal
            tmp_action[0] = -1
        if diff[0] < 0 and diff[1] < 0: # If a location of a robot is north-
east from a goal
            tmp_action[0] = 1 #Forward
    if s[0][2] in [8, 9, 10]:
        if diff[0] >= 0 and diff[1] >= 0: # If a location of a robot is
south-west from a goal
            tmp_action[0] = -1
        if diff[0] >= 0 and diff[1] < 0: # If a location of a robot is north-
west from a goal
            tmp_action[0] = -1
        if diff[0] < 0 and diff[1] >= 0: # If a location of a robot is south-
```

```
east from a goal
            tmp_action[0] = 1 #Forward
        if diff[0] < 0 and diff[1] < 0: # If a location of a robot is north-
east from a goal
            tmp_action[0] = 1 #Forward
    # If a robot is at goal:
    if diff[0] == [0] and diff[1] == [0]:
        tmp_action[0] = 0

    # About heading state:
    temp_h = 0
    # Angle difference from the current state and the goal state:
    temp_h = math.atan2(diff[1], diff[0])
    current_h = s[0][2]
    # calculate corresponding angle
    current_rad = (90 - 30 * current_h) * math.pi / 180
    # Move the robot's posture so that the difference between the current
posture and the goal posture becomes smaller.
    if (temp_h - current_rad) < math.pi and (temp_h - current_rad) > 0:
        tmp_action[1] = -1
    elif (temp_h - current_rad) == math.pi and (temp_h - current_rad) == 0:
        tmp_action[1] = 0
    else:
        tmp_action[1] = 1

    policy[tuple(s[0])] = tmp_action
```

3(b): My function is as follows:

```
# Problem 3(b)
def generate_plot_trajectory(policy, s0, pe):
    """
    Generate and plot a trajectory of a robot
    :param policy: given policy
    :param s0: initial state
    :param pe:error probability
    :return:trajectory of a robot & figure of the trajectory
    """

    trajectory = []
    s_current = s0
    count = 1
    while True:
        trajectory.append([(s_current), policy[tuple(s_current)]])
        # If a robot is at the goal
        if s_current[0] == 5 and s_current[1] == 6:
            break
        s_current = calc_next_state(pe, s_current, policy[tuple(s_current)])
        count += 1

    # Plot part
    fig = plt.figure(figsize=(L, W))
    ax = fig.add_subplot(1, 1, 1)
    plt.xlim((0, L))
    plt.ylim((0, W))
    plt.grid(True, color='k')

    # Plot boarder area
    edge1 = plt.Rectangle((0, 0), 1, W, color='r')
    edge2 = plt.Rectangle((0, 7), W, 1, color='r')
    edge3 = plt.Rectangle((7, 0), 1, W, color='r')
    edge4 = plt.Rectangle((0, 0), W, 1, color='r')
    ax.add_patch(edge1)
    ax.add_patch(edge2)
```

```
    ax.add_patch(edge3)
    ax.add_patch(edge4)
    # Plot yellow area
    yello1 = plt.Rectangle((3, 4), 1, 3, color='gold')
    ax.add_patch(yello1)
    # Plot green goal
    greengoal = plt.Rectangle((5, 6), 1, 1, color='lime')
    ax.add_patch(greengoal)

    plt.plot(s0[0] + 0.5, s0[1] + 0.5, 'o', markersize='10.5', color='k')
    ax.arrow(s0[0] + 0.5, s0[1] + 0.5, 0.4 * np.sin(30 * s0[2] * np.pi /
180), 0.4 * np.cos(30 * s0[2] * np.pi / 180), \
             head_width=0.1, head_length=0.2, fc='k', ec='k')

    for i in range(1, len(trajectory)):
        xplot = trajectory[i - 1][0][0]
        yplot = trajectory[i - 1][0][1]
        nextxplot = trajectory[i][0][0]
        nextyplot = trajectory[i][0][1]
        nexthplot = trajectory[i][0][2]
        plt.plot([xplot + 0.5, nextxplot + 0.5], [yplot + 0.5, nextyplot +
0.5], 'k--')
        plt.plot(nextxplot + 0.5, nextyplot + 0.5, 'o', markersize='10.5',
color='k')
        ax.arrow(nextxplot + 0.5, nextyplot + 0.5, 0.4 * np.sin(30 *
nexthplot * np.pi / 180),
                 0.4 * np.cos(30 * nexthplot * np.pi / 180), \
                 head_width=0.1, head_length=0.2, fc='k', ec='k')

    plt.show()

    return trajectory
```

3(c): Fig. 1 shows a trajectory of a robot using my initial policy.

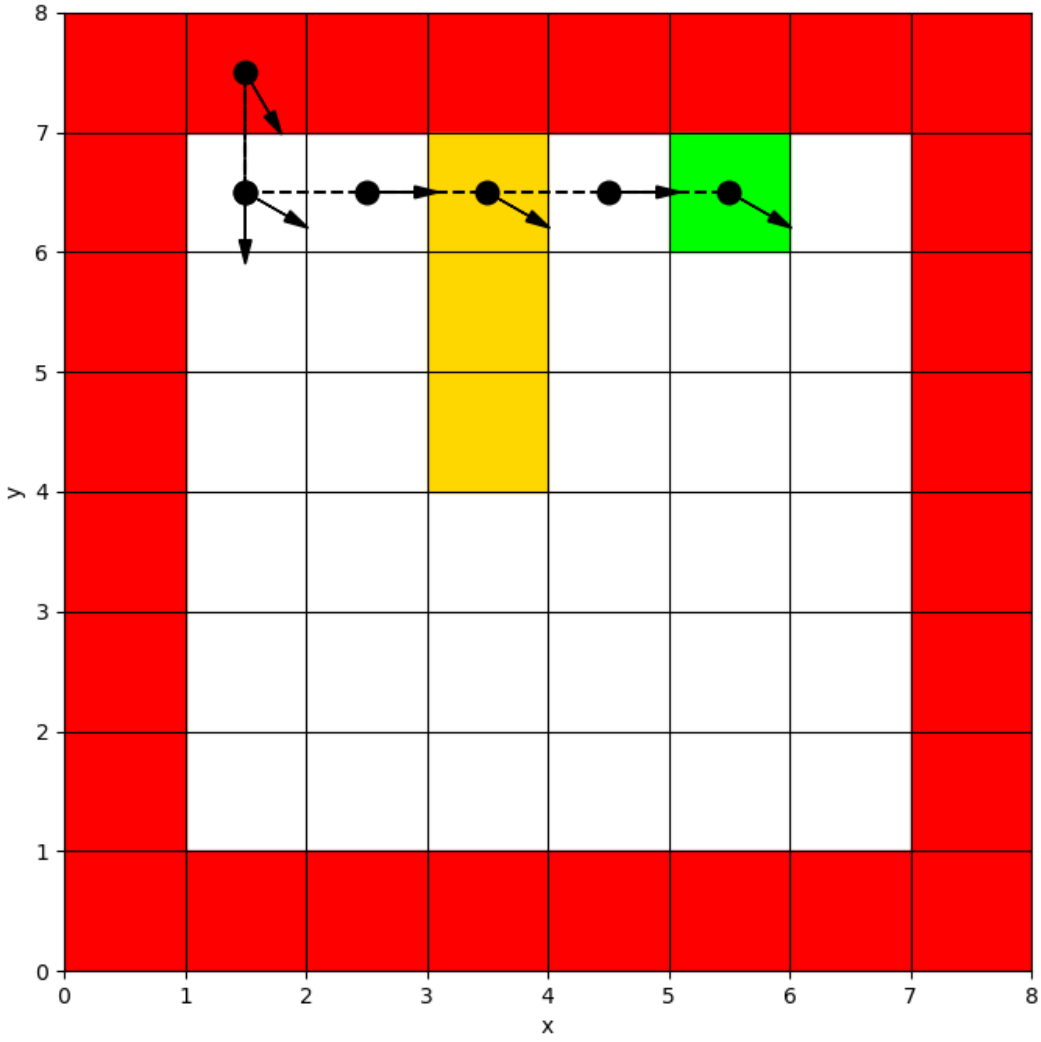

Fig. 1: Trajectory of a robot using policyπ_0 starting in state $x = 1, y = 6, h = 6$

This is the snapshot of the trajectory when I run my code:

```
Initial policy is as follows:  [[[1, 6, 6], [-1, -1]], [[1, 7, 5], [1, -1]], [[1, 6, 4], [1, -1]], [[2, 6, 3], [1, 1]], [[3, 6, 4], [1, -1]], [[4, 6, 3], [1, 1]], [[5, 6, 4], [0, -1]]]
```

where each element consists of two parts. The first element represents pose of a robot, and the second element represents the policy at the pose. For example, the 1st element [[1, 6, 6], [-1, -1]] represents that the robot is at [1, 6, 6], and the policy is [-1, -1] which means the robot will go backward and rotate to counter-clockwise direction.

3(d): My function is as follows:

```
def policy_evaluation(policy, discount_factor, pe, threshold = 0.0001):
    """
```

```
    Evaluate policy
    :param policy: given policy
    :param discount_factor: discount_factor
    :param pe: error possibility
    :param threshold: this is used to stop the iteration
    :return:trajectory of a robot & figure of the trajectory
    """
    V = {} # dictionary for value function
    listS= list(S)
    s_current = []

    # Initialize
    for s in listS:
        V[tuple(s[0])] = 0
    while True:
        delta_test = 0
        for s in listS:
            v = 0
            tmps = s.tolist()
            s_current = tmps[0]


            a = policy[tuple(s_current)]
            s = s_current

            P_trans = {}
            next_state = {}
            P_trans[0] = [([0, 1], 0, 1 - 2 * pe), ([0, 1], 11, pe), ([0, 1],
1, pe)]
            P_trans[1] = [([0, 1], 1, 1 - 2 * pe), ([0, 1], 0, pe), ([1, 0],
2, pe)]
            P_trans[2] = [([1, 0], 2, 1 - 2 * pe), ([0, 1], 1, pe), ([1, 0],
3, pe)]
            P_trans[3] = [([1, 0], 3, 1 - 2 * pe), ([1, 0], 2, pe), ([1, 0],
4, pe)]
            P_trans[4] = [([1, 0], 4, 1 - 2 * pe), ([1, 0], 3, pe), ([0, -1],
5, pe)]
            P_trans[5] = [([0, -1], 5, 1 - 2 * pe), ([1, 0], 4, pe), ([0, -
1], 6, pe)]
            P_trans[6] = [([0, -1], 6, 1 - 2 * pe), ([0, -1], 5, pe), ([0, -
1], 7, pe)]
            P_trans[7] = [([0, -1], 7, 1 - 2 * pe), ([0, -1], 6, pe), ([-1,
0], 8, pe)]
            P_trans[8] = [([-1, 0], 8, 1 - 2 * pe), ([0, -1], 7, pe), ([-1,
0], 9, pe)]
            P_trans[9] = [([-1, 0], 9, 1 - 2 * pe), ([-1, 0], 8, pe), ([-1,
0], 10, pe)]
            P_trans[10] = [([-1, 0], 10, 1 - 2 * pe), ([-1, 0], 9, pe), ([0,
1], 11, pe)]
            P_trans[11] = [([0, 1], 11, 1 - 2 * pe), ([-1, 0], 10, pe), ([0,
1], 0, pe)]


            if a[0] == 1:  # Forward motion

                for result in P_trans[s[2]]:
                    x_goal = s[0] + result[0][0]
                    y_goal = s[1] + result[0][1]
                    h_goal = (a[1] + result[1]) % 12
                # At edges of the grids, the robot can only rotate:
                    if x_goal < 0 or x_goal > L - 1:
                        x_goal = s[0]
                    if y_goal < 0 or y_goal > W - 1:
```

```
                        y_goal = s[1]

                    if result[2] != 0.0:
                        next_state[result[2]] = (x_goal, y_goal, h_goal)

            if a[0] == -1:  # Backward motion

                for result in P_trans[s[2]]:
                    x_goal = s[0] - result[0][0]
                    y_goal = s[1] - result[0][1]
                    h_goal = (a[1] + result[1]) % 12
                    # At edges of the grids, the robot can only rotate:
                    if x_goal < 0 or x_goal > L - 1:
                        x_goal = s[0]
                    if y_goal < 0 or y_goal > W - 1:
                        y_goal = s[1]

                    if result[2] != 0.0:
                        next_state[result[2]] = (x_goal, y_goal, h_goal)

            if a[0] == 0:  # No linear motion, i.e. no error
                x_goal = s[0]
                y_goal = s[1]
                h_goal = (a[1] + s[2]) % 12
                next_state[1.0] = (x_goal, y_goal, h_goal)



            # Search all potential next step. Then, caclulate value.
            for probability in next_state.keys():
                nextS = next_state[probability]
                v = v + probability *
(reward(tuple(nextS))+discount_factor*V[tuple(nextS)])

            delta_test = max(delta_test, np.abs(v-V[tuple(s)]))
            V[tuple(s)] = v
            # delta_test = np.abs(v-V[tuple(s)])
        # Stop iteration
        if delta_test < threshold:
            break
    return V
```

3(e): I run the code with the following command:

```
# Problem 3(e)
V = policy_evaluation(policy,discount_factor=0.9,pe=0.0)
for tmp in range(len(tra)):
    print("Value along trajectory",tra[tmp][0],V[tuple(tra[tmp][0])])
```

The result is as follows:

```
Value along trajectory [1, 6, 6] -101.38510257585469
Value along trajectory [1, 7, 5] -1.5390023182692152
Value along trajectory [1, 6, 4] -1.7100025758546835
Value along trajectory [2, 6, 3] -1.9000023182692143
Value along trajectory [3, 6, 4] 8.999997913557708
Value along trajectory [4, 6, 3] 9.999998122201937
Value along trajectory [5, 6, 4] 9.999998309981743
```

For example, the first line mentions that value is -101.38510257585469 at pose [1,6,6].

3(f): My function is as follows:

```
# Problem 3(f)
def one_step_lookahead(V,pe=0.0,discount_factor = 0.9):
    """
    Compute an optimal policy given one-step lookhead on V
    :param V: value function for current policy
    :param pe: error possibility
    :param discount_factor: discount factor for future reward
    :return: optimal policy
    """
    policy = {}
    tmp_action = {}


    for tmps in S:
        action = np.zeros(nA)
        for i in range(len(A[0])):
            tmp_action[i] = [A[0][i],A[1][i]]



            a = tuple(tmp_action[i])
            s = tuple(tmps[0])

            P_trans = {}
            next_state = {}
            P_trans[0] = [([0, 1], 0, 1 - 2 * pe), ([0, 1], 11, pe), ([0, 1],
1, pe)]
            P_trans[1] = [([0, 1], 1, 1 - 2 * pe), ([0, 1], 0, pe), ([1, 0],
2, pe)]
            P_trans[2] = [([1, 0], 2, 1 - 2 * pe), ([0, 1], 1, pe), ([1, 0],
3, pe)]
            P_trans[3] = [([1, 0], 3, 1 - 2 * pe), ([1, 0], 2, pe), ([1, 0],
4, pe)]
            P_trans[4] = [([1, 0], 4, 1 - 2 * pe), ([1, 0], 3, pe), ([0, -1],
5, pe)]
            P_trans[5] = [([0, -1], 5, 1 - 2 * pe), ([1, 0], 4, pe), ([0, -
1], 6, pe)]
            P_trans[6] = [([0, -1], 6, 1 - 2 * pe), ([0, -1], 5, pe), ([0, -
1], 7, pe)]
            P_trans[7] = [([0, -1], 7, 1 - 2 * pe), ([0, -1], 6, pe), ([-1,
0], 8, pe)]
            P_trans[8] = [([-1, 0], 8, 1 - 2 * pe), ([0, -1], 7, pe), ([-1,
0], 9, pe)]
            P_trans[9] = [([-1, 0], 9, 1 - 2 * pe), ([-1, 0], 8, pe), ([-1,
0], 10, pe)]
            P_trans[10] = [([-1, 0], 10, 1 - 2 * pe), ([-1, 0], 9, pe), ([0,
1], 11, pe)]
            P_trans[11] = [([0, 1], 11, 1 - 2 * pe), ([-1, 0], 10, pe), ([0,
1], 0, pe)]


            if a[0] == 1:  # Forward motion
                # For every possible action:
                for result in P_trans[s[2]]:
                    x_goal = s[0] + result[0][0]
                    y_goal = s[1] + result[0][1]
                    h_goal = (a[1] + result[1]) % 12
                # At edges of the grids, the robot can only rotate:
                    if x_goal < 0 or x_goal > L - 1:
                        x_goal = s[0]
                    if y_goal < 0 or y_goal > W - 1:
                        y_goal = s[1]
                    # Add the possible next state if the possibility of
```

```
achiving there is not 0 %
                    if result[2] != 0.0:
                        next_state[result[2]] = (x_goal, y_goal, h_goal)

            if a[0] == -1:  # Backward motion

                for result in P_trans[s[2]]:
                    x_goal = s[0] - result[0][0]
                    y_goal = s[1] - result[0][1]
                    h_goal = (a[1] + result[1]) % 12
                    # At edges of the grids, the robot can only rotate:
                    if x_goal < 0 or x_goal > L - 1:
                        x_goal = s[0]
                    if y_goal < 0 or y_goal > W - 1:
                        y_goal = s[1]

                    if result[2] != 0.0:
                        next_state[result[2]] = (x_goal, y_goal, h_goal)

            if a[0] == 0:  # No linear motion, i.e. no error
                x_goal = s[0]
                y_goal = s[1]
                h_goal = (a[1] + s[2]) % 12
                next_state[1.0] = (x_goal, y_goal, h_goal)#Possibility is
always 1 (100 %)

            # Search all next state
            for probability in next_state.keys():
                nextS = next_state[probability]
                # Calculate Q value
                action[i] +=
probability*(reward(tuple(nextS))+discount_factor*V[tuple(nextS)])
        # Find the optimal Q.
        optimal_num = np.argmax(action)
        policy[tuple(s)] = [A[0][optimal_num],A[1][optimal_num]]

    return policy
```

3(g): My policy iteration code is as follows:

```
# Problem 3(g)
def policy_iteration(policy, p_e=0.0, discount_factor=0.9):
    """
    Policy Iteration


    policy: initial policy used to iterate
    p_e: the error probability
    discount_factor: discount factor.

    Returns:
        optimal policy and optimal value
    """
    while True:
        V = policy_evaluation(policy,discount_factor, p_e)

        # To judge whether the policy converges or not:
        frag = True

        new_policy = one_step_lookahead(V, p_e)

        if policy != new_policy:
```

```
            frag = False
        policy = new_policy

        if frag:
            return [policy, V]
```

3(h): I used the following code to plot trajectory of a robot using optimal policy.

```
start_time = time.time()
optimal_p, optimal_v = policy_iteration(policy)
end_time = time.time()
print("Time of computation using policy iteration is: %s s" %str(end_time -
start_time) )
s0 = (1,6,6)
traj = generate_plot_trajectory(optimal_p,s0,pe=0.0)
print('Optimal policy is as follows: ',traj)
for tmp in range(len(tra)):
    print("Value along
trajectory",traj[tmp][0],optimal_v[tuple(traj[tmp][0])])
```

Then, the result is as follows:

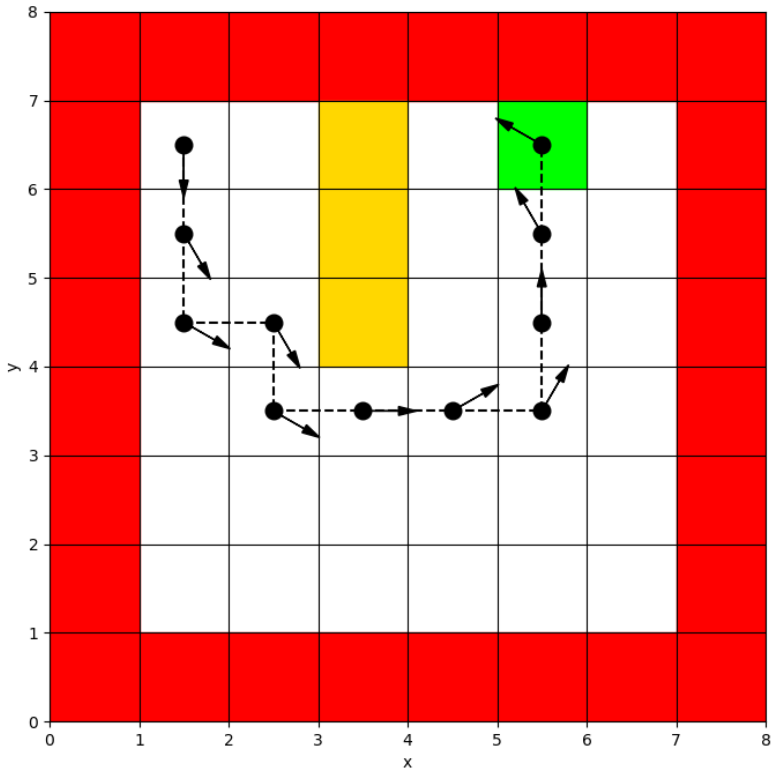

Fig. 2: Trajectory of a robot using optimal policy π^* starting in state $x = 1, y = 6, h = 6$

To make it clear, I also put the optimal policy and optimal value as follows:

Optimal policy is as follows: [[(1, 6, 6), [1, -1]], [[1, 5, 5], [1, -1]], [[1, 4, 4], [1, 1]], [[2, 4, 5], [1, -1]], [[2, 3, 4], [1, -1]], [[3, 3, 3], [1, -1]], [[4, 3, 2], [1, -1]], [[5, 3, 1], [1, -1]], [[5, 4, 0], [1, -1]], [[5, 5, 11], [1, -1]], [[5, 6, 10], [0, 0]]]

```
Value along trajectory (1, 6, 6) 3.8735878534808297
Value along trajectory [1, 5, 5] 4.303986503867589
Value along trajectory [1, 4, 4] 4.782207226519543
Value along trajectory [2, 4, 5] 5.313648226519542
Value along trajectory [2, 3, 4] 5.904053585021714
Value along trajectory [3, 3, 3] 6.560153585021716
Value along trajectory [4, 3, 2] 7.289153585021714
Value along trajectory [5, 3, 1] 8.099153585021716
Value along trajectory [5, 4, 0] 8.999153585021714
Value along trajectory [5, 5, 11] 9.999153585021714
Value along trajectory [5, 6, 10] 9.999153585021714
```

3(i): I used the following code to estimate running time of the policy iteration:

```
start_time = time.time()
optimal_p, optimal_v = policy_iteration(policy)
end_time = time.time()
print("Time of computation using policy iteration is: %s s" %str(end_time -
start_time) )
```

It takes **13.889 s**

4. Value iteration

4(a): My value iteration function is as follows:

```
def value_iteration(policy, pe=0.0, discount_factor=0.9, threshold = 0.0001):
    """
    Value Iteration


    policy: initial policy
    pe: the error probability
    discount_factor: discount factor
    threshold: threshold for iteration

    Returns:
        optimal policy and optimal value
    """
    # Create dictionary for value functions
    V = {}
    tmp_action = {}
    # Initialize
    for s in S:
        V[tuple(s[0])] = 0
    # Iteration
    while True:
        delta_test = 0
        for tmps in S:
            action = np.zeros(nA)
```

```
            for i in range(len(A[0])):
                tmp_action[i] = [A[0][i], A[1][i]]

                a = tuple(tmp_action[i])
                s = tuple(tmps[0])

                P_trans = {}
                next_state = {}
                P_trans[0] = [([0, 1], 0, 1 - 2 * pe), ([0, 1], 11, pe), ([0,
1], 1, pe)]
                P_trans[1] = [([0, 1], 1, 1 - 2 * pe), ([0, 1], 0, pe), ([1,
0], 2, pe)]
                P_trans[2] = [([1, 0], 2, 1 - 2 * pe), ([0, 1], 1, pe), ([1,
0], 3, pe)]
                P_trans[3] = [([1, 0], 3, 1 - 2 * pe), ([1, 0], 2, pe), ([1,
0], 4, pe)]
                P_trans[4] = [([1, 0], 4, 1 - 2 * pe), ([1, 0], 3, pe), ([0,
-1], 5, pe)]
                P_trans[5] = [([0, -1], 5, 1 - 2 * pe), ([1, 0], 4, pe), ([0,
-1], 6, pe)]
                P_trans[6] = [([0, -1], 6, 1 - 2 * pe), ([0, -1], 5, pe),
([0, -1], 7, pe)]
                P_trans[7] = [([0, -1], 7, 1 - 2 * pe), ([0, -1], 6, pe), ([-
1, 0], 8, pe)]
                P_trans[8] = [([-1, 0], 8, 1 - 2 * pe), ([0, -1], 7, pe), ([-
1, 0], 9, pe)]
                P_trans[9] = [([-1, 0], 9, 1 - 2 * pe), ([-1, 0], 8, pe), ([-
1, 0], 10, pe)]
                P_trans[10] = [([-1, 0], 10, 1 - 2 * pe), ([-1, 0], 9, pe),
([0, 1], 11, pe)]
                P_trans[11] = [([0, 1], 11, 1 - 2 * pe), ([-1, 0], 10, pe),
([0, 1], 0, pe)]

                if a[0] == 1:  # Forward motion

                    for result in P_trans[s[2]]:
                        x_goal = s[0] + result[0][0]
                        y_goal = s[1] + result[0][1]
                        h_goal = (a[1] + result[1]) % 12
                        # At edges of the grids, the robot can only rotate:
                        if x_goal < 0 or x_goal > L - 1:
                            x_goal = s[0]
                        if y_goal < 0 or y_goal > W - 1:
                            y_goal = s[1]

                        if result[2] != 0.0:
                            next_state[result[2]] = (x_goal, y_goal, h_goal)

                if a[0] == -1:  # Backward motion

                    for result in P_trans[s[2]]:
                        x_goal = s[0] - result[0][0]
                        y_goal = s[1] - result[0][1]
                        h_goal = (a[1] + result[1]) % 12
                        # At edges of the grids, the robot can only rotate:
                        if x_goal < 0 or x_goal > L - 1:
                            x_goal = s[0]
                        if y_goal < 0 or y_goal > W - 1:
                            y_goal = s[1]

                        if result[2] != 0.0:
                            next_state[result[2]] = (x_goal, y_goal, h_goal)
```

```
                if a[0] == 0:  # No linear motion, i.e. no error
                    x_goal = s[0]
                    y_goal = s[1]
                    h_goal = (a[1] + s[2]) % 12
                    next_state[1.0] = (x_goal, y_goal, h_goal)

                for probability in next_state.keys():
                    nextS = next_state[probability]
                    action[i] += probability * (reward(tuple(nextS)) +
discount_factor * V[tuple(nextS)])


            optimal_action = np.max(action)
            optimal_index = np.argmax(action)
            policy[tuple(s)] = [A[0][optimal_index], A[1][optimal_index]]
            # To judge whether iteration ends or not
            delta_test = max(delta_test, np.abs(optimal_action -
V[tuple(s)]))

            V[tuple(s)] = optimal_action
        # If difference between current values and one step before value is
smaller than threshold, stop iteration
        if delta_test < threshold:
            break
    return policy, V
```

4(b) and (c):I used the following code:

```
# Problem 4(b) and (c)
start_time = time.time()
optimal_p, optimal_v = value_iteration(policy, pe =0.0, discount_factor=0.9)
end_time = time.time()
print("Time of computation using value iteration is: %s s" %str(end_time -
start_time) )

s0 = (1, 6, 6)
traj = generate_plot_trajectory(optimal_p,s0,pe=0.0)
print('Optimal policy is as follows: ',traj)
for tmp in range(len(tra)):
    print("Value along
trajectory",traj[tmp][0],optimal_v[tuple(traj[tmp][0])])
```

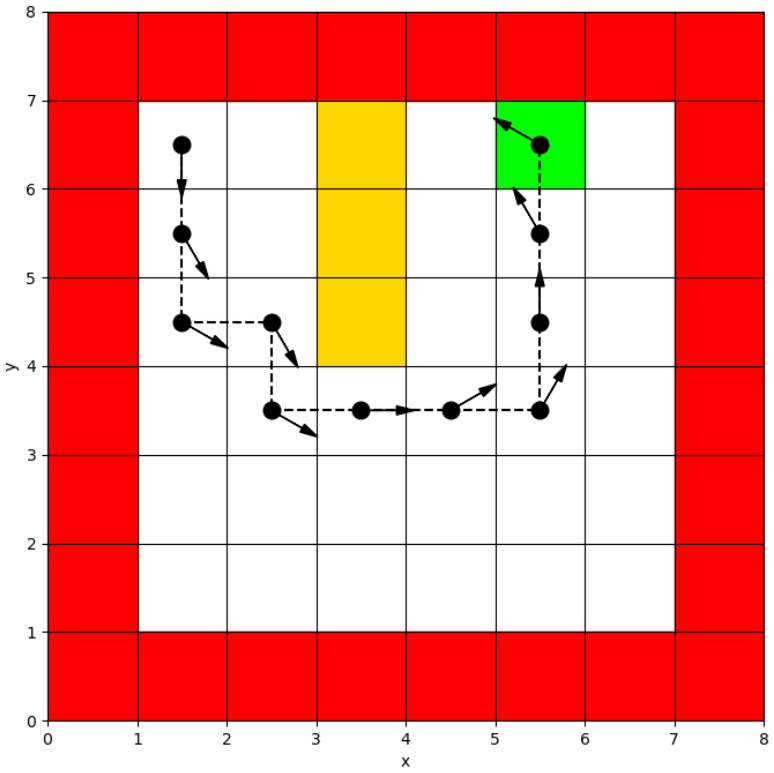

Fig. 3: Trajectory of a robot using optimal policy π^* from value iteration starting in state $x = 1, y = 6, h = 6$

To make it clear, I also put the optimal policy and optimal value as follows:

Optimal policy is as follows: [[(1, 6, 6), [1, -1]], [[1, 5, 5], [1, -1]], [[1, 4, 4], [1, 1]], [[2, 4, 5], [1, -1]], [[2, 3, 4], [1, -1]], [[3, 3, 3], [1, -1]], [[4, 3, 2], [1, -1]], [[5, 3, 1], [1, -1]], [[5, 4, 0], [1, -1]], [[5, 5, 11], [1, -1]], [[5, 6, 10], [0, 0]]]
Value along trajectory (1, 6, 6) 3.8735878534808297 Value along trajectory [1, 5, 5] 4.303986503867589 Value along trajectory [1, 4, 4] 4.782207226519543 Value along trajectory [2, 4, 5] 5.313648226519542 Value along trajectory [2, 3, 4] 5.904053585021714 Value along trajectory [3, 3, 3] 6.560153585021716 Value along trajectory [4, 3, 2] 7.289153585021714 Value along trajectory [5, 3, 1] 8.099153585021716 Value along trajectory [5, 4, 0] 8.999153585021714 Value along trajectory [5, 5, 11] 9.999153585021714 Value along trajectory [5, 6, 10] 9.999153585021714

The result is shown in Fig.3. As you see, we get the same result when we use the policy iteration as shown in Fig.2. This is reasonable because our possibility of error is 0. Of course, if we have different iteration threshold, we may get the different result. However, as long as the iteration converges, we should have the same result. Hence, we can claim that my policy iteration and value iteration codes work well.

Time of computation using value iteration is **20.169 s.** It means that the computation time is larger than that of policy iteration. It's reasonable since our discount factor is 0.9, which is a little bit large value. If we use a much lower discount factor, the value iteration has the possibility to decrease the computation time.

5. Additional scenarios

5(a): I used the following code.

```
# Problem 5(a)
optimal_p, optimal_v = policy_iteration(policy, p_e=0.25)
s0 = (1,6,6)
traj = generate_plot_trajectory(optimal_p,s0,pe=0.25)
print('Optimal policy is as follows: ',traj)
for tmp in range(len(tra)):
    print("Value along
trajectory",traj[tmp][0],optimal_v[tuple(traj[tmp][0])])
optimal_p, optimal_v = value_iteration(policy, pe=0.25)
s0 = (1,6,6)
traj = generate_plot_trajectory(optimal_p,s0,pe=0.25)
print('Optimal policy is as follows: ',traj)
for tmp in range(len(tra)):
    print("Value along
trajectory",traj[tmp][0],optimal_v[tuple(traj[tmp][0])])
```

First, I show the result when I use the policy iteration.

```
Optimal policy is as follows:  [[(1, 6, 6), [1, -1]], [[1, 5, 5], [1, 0]], [[1, 4, 6], [1, 1]], [[1, 3, 8], [-1, 0]], [[2, 3, 9], [-1, -1]], [[3, 3, 8], [-1, 1]], [[4, 3, 9], [-1, 1]], [[5, 3, 10], [1, 0]], [[4, 3, 10], [-1, 1]], [[5, 3, 11], [1, 0]], [[4, 3, 10], [-1, 1]], [[5, 3, 10], [1, 0]], [[4, 3, 10], [-1, 1]], [[5, 3, 11], [1, 0]], [[5, 4, 0], [1, -1]], [[5, 5, 11], [1, -1]], [[5, 6, 10], [0, 0]]]
```

```
Value along trajectory (1, 6, 6) 0.1051538095364189
Value along trajectory [1, 5, 5] 0.1557834215354354
Value along trajectory [1, 4, 6] 0.24019182434464786
Value along trajectory [1, 3, 8] 0.5310608128809813
Value along trajectory [2, 3, 9] 0.7867695963589488
Value along trajectory [3, 3, 8] 1.1457425619395922
Value along trajectory [4, 3, 9] 1.6458887464964382
Value along trajectory [5, 3, 10] 1.9491590641812664
Value along trajectory [4, 3, 10] 1.800479091750195
Value along trajectory [5, 3, 11] 3.416830418681036
Value along trajectory [4, 3, 10] 1.800479091750195
Value along trajectory [5, 3, 10] 1.9491590641812664
Value along trajectory [4, 3, 10] 1.800479091750195
Value along trajectory [5, 3, 11] 3.416830418681036
Value along trajectory [5, 4, 0] 5.062023891574714
Value along trajectory [5, 5, 11] 7.499365188766285
Value along trajectory [5, 6, 10] 9.999153585021714
```

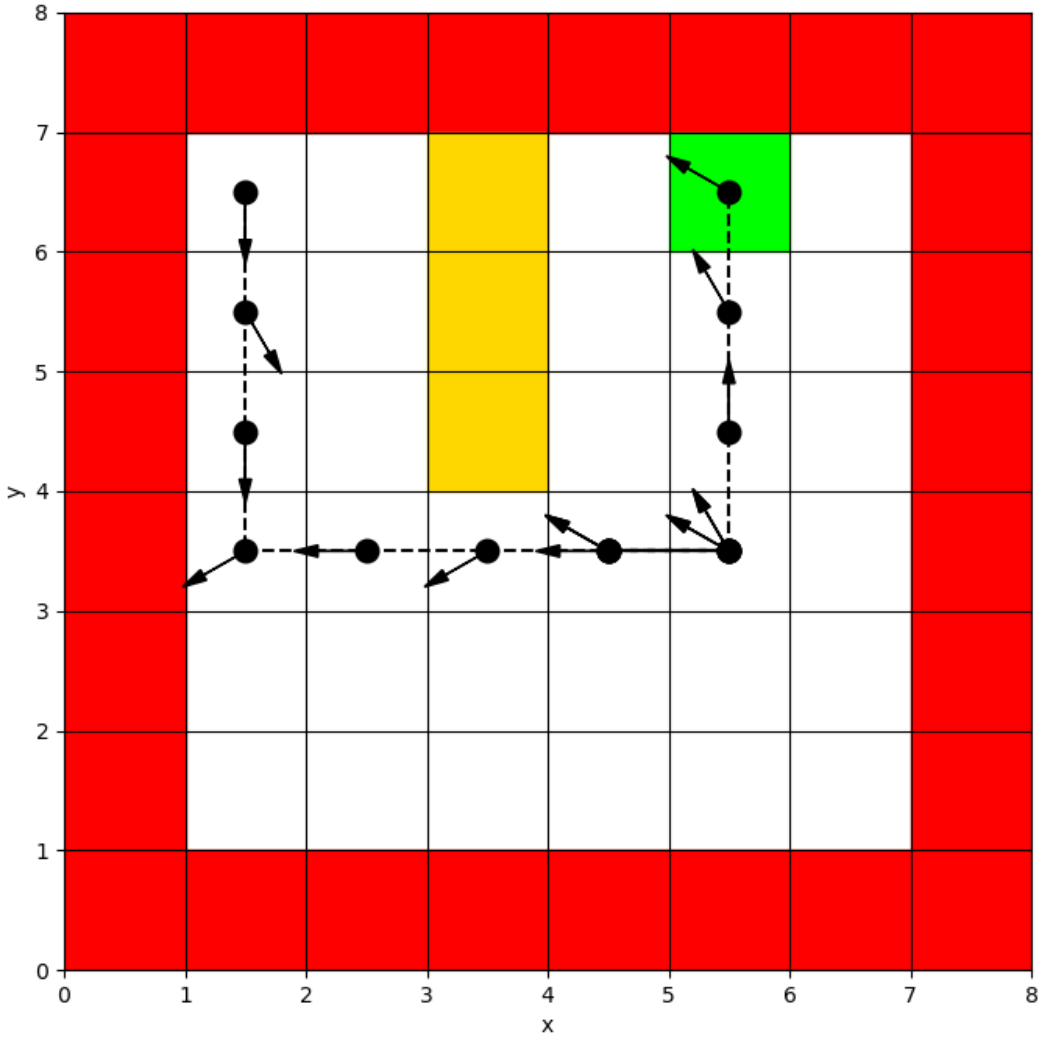

Fig. 4: Trajectory of a robot using optimal policy π^* from policy iteration starting in state $x = 1, y = 6, h = 6$

Compared with Fig.2, in Fig.4, it takes longer path to achieve the goal since error possibility is not 0 anymore. It makes sense because the robot needs to avoid obstacles with high confidence and to avoid the obstacles, the robot needs to use the trajectory that is far from the obstacles. In addition, the results were different every time because the pose of the robot follows the discrete probability density function. In other words, the dynamics of the robot is stochastic.

I got the similar result but not the same result when I used the value function as follows:

Optimal policy is as follows: [[(1, 6, 6), [1, -1]], [[1, 5, 5], [1, 0]], [[2, 5, 4], [-1, 0]], [[1, 5, 3], [1, 1]], [[2, 5, 3], [-1, 1]], [[1, 5, 4], [1, 1]], [[1, 4, 6], [1, 1]], [[1, 3, 7], [-1, 1]], [[1, 4, 7], [-1, -1]], [[1, 5, 5], [1, 0]], [[1, 4, 5], [1, -1]], [[1, 3, 5], [1, 1]], [[1, 2, 6], [-1, 1]], [[1, 3, 8], [-1, 0]], [[1, 4, 7], [-1, -1]], [[1, 5, 5], [1, 0]], [[1, 4, 6], [1, 1]], [[1, 3, 7], [-1, 1]], [[2, 3, 9], [-1, -1]], [[3,

3, 8], [-1, 1]], [[4, 3, 9], [-1, 1]], [[5, 3, 9], [-1, 1]], [[6, 3, 10], [1, 1]], [[5, 3, 11], [1, 0]], [[4, 3, 10], [-1, 1]], [[4, 2, 0], [1, -1]], [[4, 3, 0], [1, -1]], [[4, 4, 0], [1, 1]], [[4, 5, 0], [1, 1]], [[4, 6, 0], [-1, 1]], [[4, 5, 1], [1, 1]], [[5, 5, 3], [-1, 0]], [[4, 5, 4], [1, 1]], [[5, 5, 4], [-1, 0]], [[4, 5, 4], [1, 1]], [[5, 5, 5], [-1, -1]], [[5, 6, 4], [0, 0]]]

Value along trajectory (1, 6, 6) 0.1051538095364189
Value along trajectory [1, 5, 5] 0.1557834215354354
Value along trajectory [2, 5, 4] 0.11450157481289654
Value along trajectory [1, 5, 3] 0.10878326056856474
Value along trajectory [2, 5, 3] 0.11090335068779768
Value along trajectory [1, 5, 4] 0.16856017964961045
Value along trajectory [1, 4, 6] 0.24019182434464786
Value along trajectory [1, 3, 7] 0.2682292032142824
Value along trajectory [1, 4, 7] 0.1573770431092562
Value along trajectory [1, 5, 5] 0.1557834215354354
Value along trajectory [1, 4, 5] 0.22608946901753246
Value along trajectory [1, 3, 5] 0.18883479387680213
Value along trajectory [1, 2, 6] 0.24018735476874684
Value along trajectory [1, 3, 8] 0.5310608128809813
Value along trajectory [1, 4, 7] 0.1573770431092562
Value along trajectory [1, 5, 5] 0.1557834215354354
Value along trajectory [1, 4, 6] 0.24019182434464786
Value along trajectory [1, 3, 7] 0.2682292032142824
Value along trajectory [2, 3, 9] 0.7867695963589488
Value along trajectory [3, 3, 8] 1.1457425619395922
Value along trajectory [4, 3, 9] 1.6458887464964382
Value along trajectory [5, 3, 9] 1.3236309272036082
Value along trajectory [6, 3, 10] 2.0084835694975185
Value along trajectory [5, 3, 11] 3.416830418681036
Value along trajectory [4, 3, 10] 1.800479091750195
Value along trajectory [4, 2, 0] 1.1685478107096752
Value along trajectory [4, 3, 0] 1.6814342617036853
Value along trajectory [4, 4, 0] 2.4880941188013233
Value along trajectory [4, 5, 0] 1.8725592072229564
Value along trajectory [4, 6, 0] 2.4881208026015686
Value along trajectory [4, 5, 1] 3.9392684002573066
Value along trajectory [5, 5, 3] 2.5092971559147204
Value along trajectory [4, 5, 4] 3.934503771113441
Value along trajectory [5, 5, 4] 4.270315093256477
Value along trajectory [4, 5, 4] 3.934503771113441
Value along trajectory [5, 5, 5] 7.499365188766285
Value along trajectory [5, 6, 4] 9.999153585021714

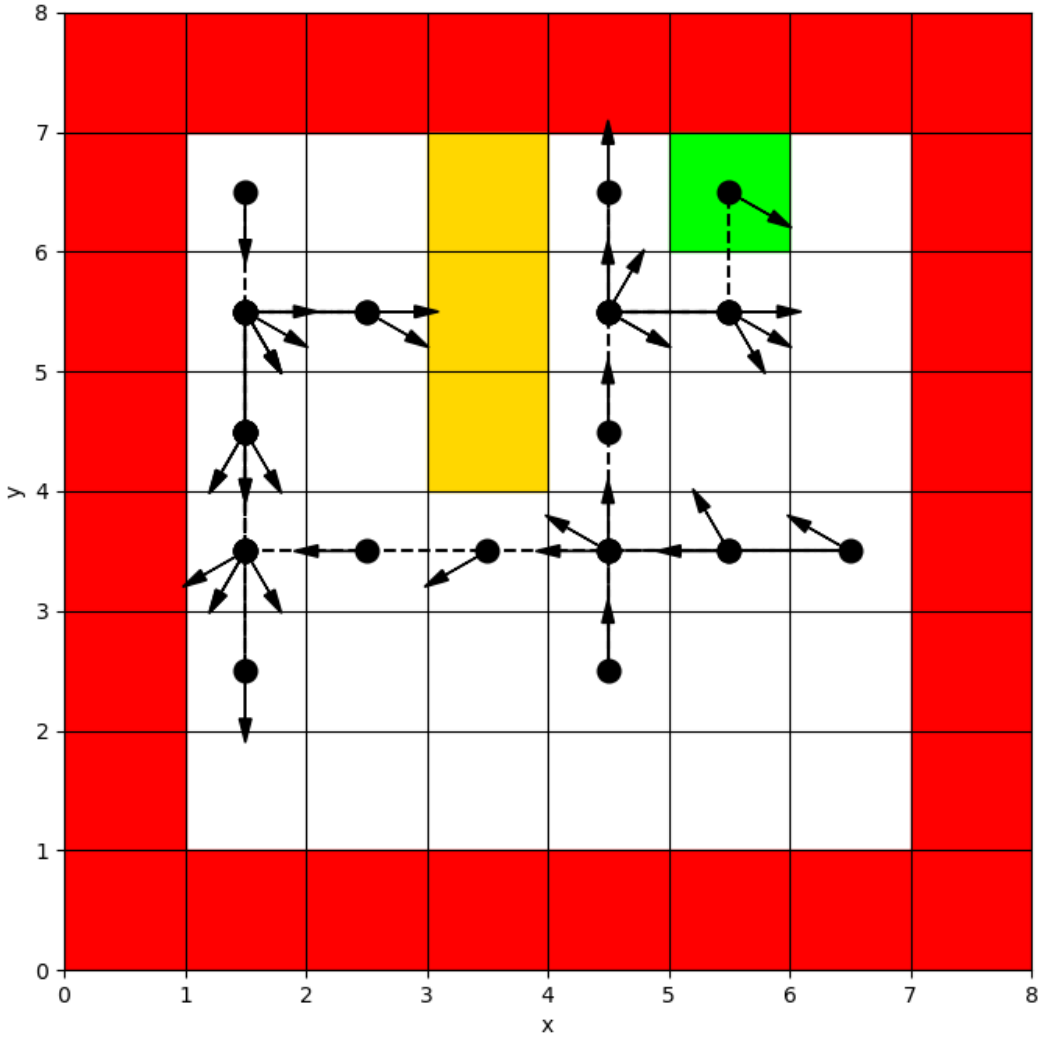

Fig. 5: Trajectory of a robot using optimal policy π^* from value iteration starting in state $x = 1, y = 6, h = 6$

Compared with Fig.4, the result is less reasonable than that of policy iteration because it as I mentioned earlier, our discount factor is a little bit large which means that the value iteration function needs more iteration to have more reasonable result. In Fig.3, we get the appropriate result even though the number of iterations is the same as the number of iterations in Fig. 5 because the error possibility is 0.

5(b): I created the new reward function as follows:

```
def reward(s):
    """
    Calculate a next state
    :param pe:error probability
    :param s:state
    :param a:action
    :return:next state that follows the pdf

    # Idea: using random values, I calculate the next state

    """
    reward = 0
    if (s[0] in [0, 7]) or (s[1] in [0, 7]):
        reward = -100
    if s[0] == 3 and s[1] in [4, 5, 6]:
        reward = -10
    if s[0] == 5 and s[1] == 6 and s[2] in [6]:
        reward = 1

    return reward
```

First, I show the result using policy and value iteration with 0 error possibility.

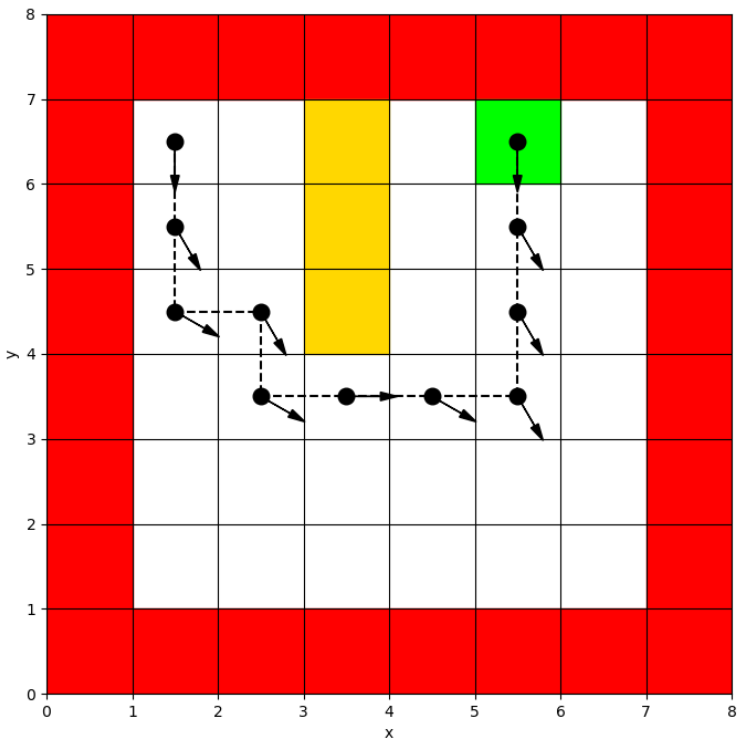

Fig. 6: Trajectory of a robot using optimal policy π^* from policy iteration starting in state $x = 1, y = 6, h = 6$ with $p_e = 0$

```
Optimal policy is as follows:  [[(1, 6, 6), [1, -1]], [[1, 5, 5], [1, -1]], [[1, 4, 4], [1, 1]],
[[2, 4, 5], [1, -1]], [[2, 3, 4], [1, -1]], [[3, 3, 3], [1, 1]], [[4, 3, 4], [1, 1]], [[5, 3, 5], [-1,
0]], [[5, 4, 5], [-1, 0]], [[5, 5, 5], [-1, 1]], [[5, 6, 6], [0, 0]]]
Value along trajectory (1, 6, 6) 3.8735878534808297
Value along trajectory [1, 5, 5] 4.303986503867589
Value along trajectory [1, 4, 4] 4.782207226519543
Value along trajectory [2, 4, 5] 5.313648226519542
Value along trajectory [2, 3, 4] 5.904053585021714
Value along trajectory [3, 3, 3] 6.560153585021716
Value along trajectory [4, 3, 4] 7.289153585021714
Value along trajectory [5, 3, 5] 8.099153585021716
Value along trajectory [5, 4, 5] 8.999153585021714
Value along trajectory [5, 5, 5] 9.999153585021714
Value along trajectory [5, 6, 6] 9.999153585021714
```

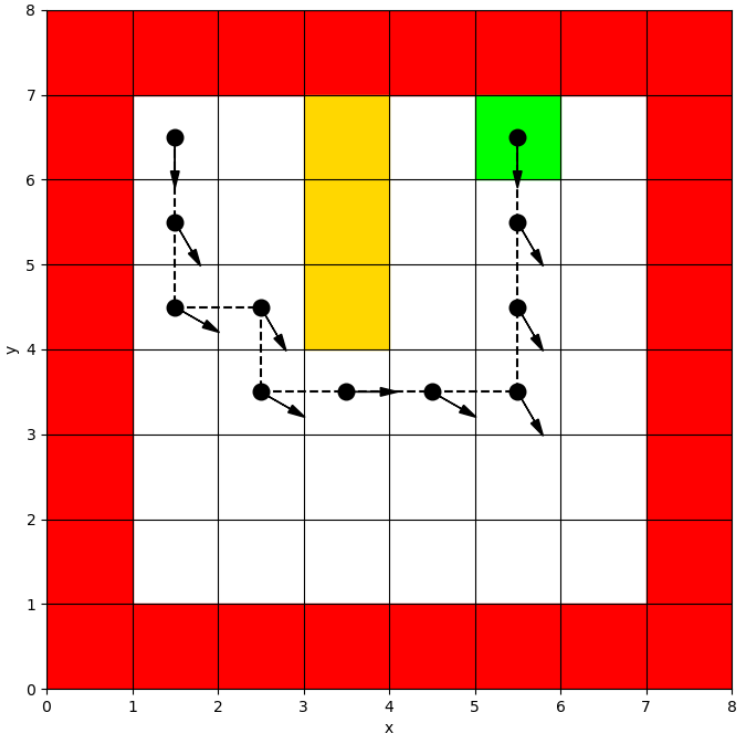

Fig. 7: Trajectory of a robot using optimal policy π^* from value iteration starting in state $x = 1, y = 6, h = 6$ with $p_e = 0$


```
Optimal policy is as follows:  [[(1, 6, 6), [1, -1]], [[1, 5, 5], [1, -1]], [[1, 4, 4], [1, 1]],
[[2, 4, 5], [1, -1]], [[2, 3, 4], [1, -1]], [[3, 3, 3], [1, 1]], [[4, 3, 4], [1, 1]], [[5, 3, 5], [-1,
0]], [[5, 4, 5], [-1, 0]], [[5, 5, 5], [-1, 1]], [[5, 6, 6], [0, 0]]]
Value along trajectory (1, 6, 6) 3.8735878534808297
Value along trajectory [1, 5, 5] 4.303986503867589
```

Value along trajectory [1, 4, 4] 4.782207226519543
Value along trajectory [2, 4, 5] 5.313648226519542
Value along trajectory [2, 3, 4] 5.904053585021714
Value along trajectory [3, 3, 3] 6.560153585021716
Value along trajectory [4, 3, 4] 7.289153585021714
Value along trajectory [5, 3, 5] 8.099153585021716
Value along trajectory [5, 4, 5] 8.999153585021714
Value along trajectory [5, 5, 5] 9.999153585021714
Value along trajectory [5, 6, 6] 9.999153585021714

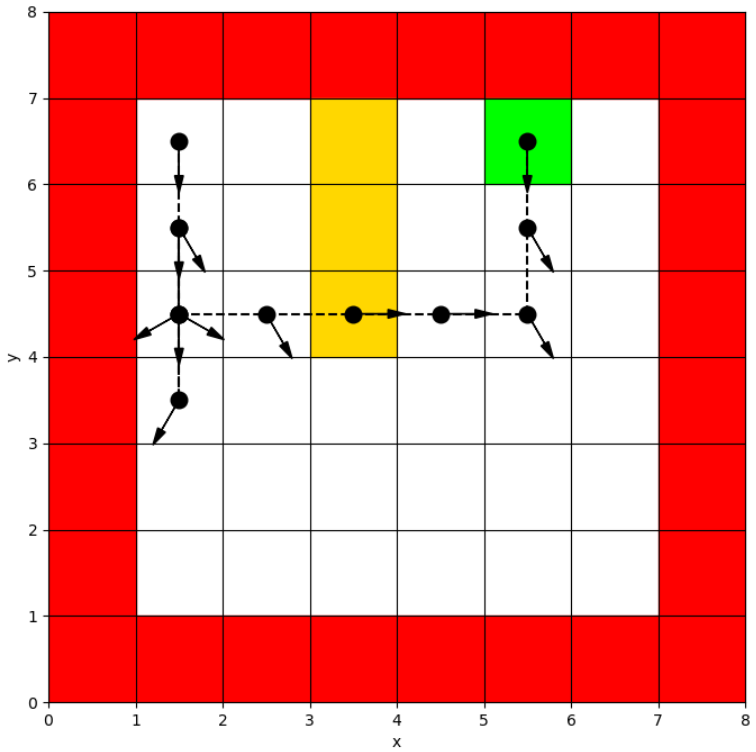

Fig. 8: Trajectory of a robot using optimal policy π^* from policy iteration starting in state $x = 1, y = 6, h = 6$ with $p_e = 0.25$

Optimal policy is as follows: [[(1, 6, 6), [1, -1]], [[1, 5, 5], [1, 0]], [[1, 4, 6], [1, 1]], [[1, 3, 7], [-1, 1]], [[1, 4, 8], [-1, -1]], [[1, 5, 6], [1, -1]], [[1, 4, 4], [1, 1]], [[2, 4, 5], [1, -1]], [[3, 4, 3], [1, 0]], [[4, 4, 3], [1, 1]], [[5, 4, 5], [-1, 0]], [[5, 5, 5], [-1, 1]], [[5, 6, 6], [0, 0]]]
Value along trajectory (1, 6, 6) 0.06986579797760366
Value along trajectory [1, 5, 5] 0.10350488589274616
Value along trajectory [1, 4, 6] 0.1364545786110382
Value along trajectory [1, 3, 7] 0.15255007185419775
Value along trajectory [1, 4, 8] 0.1157693455240448

```
Value along trajectory [1, 5, 6] 0.10350488589274616
Value along trajectory [1, 4, 4] 0.1442329239716576
Value along trajectory [2, 4, 5] 0.25115811490863815
Value along trajectory [3, 4, 3] 1.2539124164435536
Value along trajectory [4, 4, 3] 1.816211453702235
Value along trajectory [5, 4, 5] 3.795365618707918
Value along trajectory [5, 5, 5] 5.622817632932442
Value along trajectory [5, 6, 6] 9.999153585021714
```

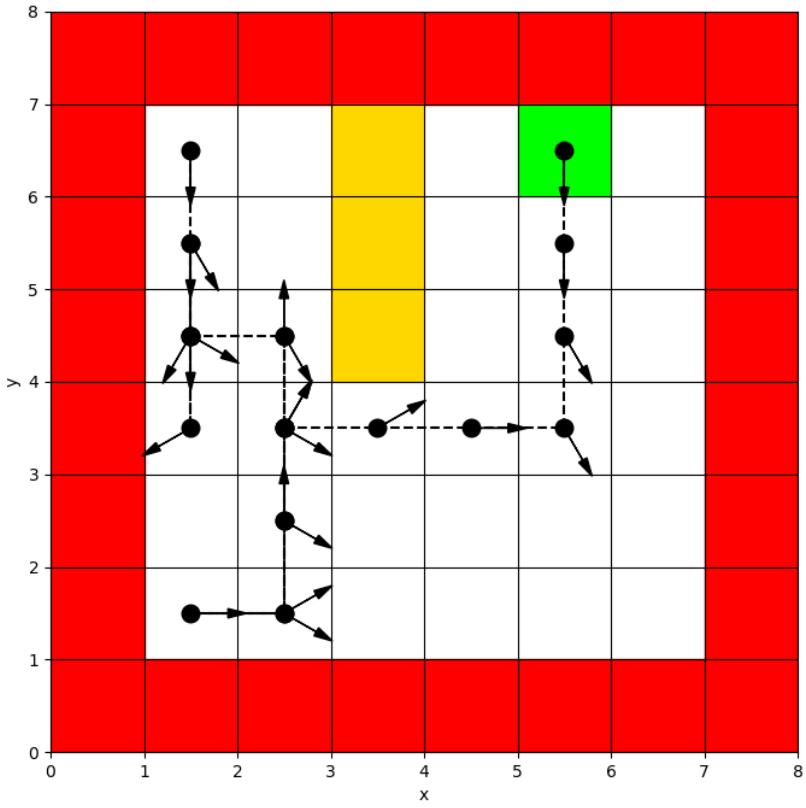

Fig. 9: Trajectory of a robot using optimal policy π^* from value iteration starting in state $x = 1, y = 6, h = 6$ with $p_e = 0.25$

```
Optimal policy is as follows:  [[(1, 6, 6), [1, -1]], [[1, 5, 5], [1, 0]], [[1, 4, 6], [1, 1]], [[1, 3, 8], [-1, 0]], [[1, 4, 7], [-1, -1]], [[1, 5, 6], [1, -1]], [[1, 4, 4], [1, 1]], [[2, 4, 5], [1, -1]], [[2, 3, 4], [1, -1]], [[2, 2, 4], [1, -1]], [[2, 1, 4], [-1, -1]], [[1, 1, 3], [1, -1]], [[2, 1, 2], [1, -1]], [[2, 2, 0], [1, 1]], [[2, 3, 1], [1, 0]], [[2, 4, 0], [-1, 1]], [[2, 3, 1], [1, 0]], [[3, 3, 2], [1, 1]], [[4, 3, 3], [1, 1]], [[5, 3, 5], [-1, 0]], [[5, 4, 5], [-1, 0]], [[5, 5, 6], [-1, 0]], [[5, 6, 6], [0, 0]]]
Value along trajectory (1, 6, 6) 0.06986579797760366
```

```
Value along trajectory [1, 5, 5] 0.10350488589274616
Value along trajectory [1, 4, 6] 0.1364545786110382
Value along trajectory [1, 3, 8] 0.30136465011844105
Value along trajectory [1, 4, 7] 0.09632718540126761
Value along trajectory [1, 5, 6] 0.10350488589274616
Value along trajectory [1, 4, 4] 0.1442329239716576
Value along trajectory [2, 4, 5] 0.25115811490863815
Value along trajectory [2, 3, 4] 0.4615253155040947
Value along trajectory [2, 2, 4] 0.30377345155894486
Value along trajectory [2, 1, 4] 0.14129884428997974
Value along trajectory [1, 1, 3] 0.16211354919536106
Value along trajectory [2, 1, 2] 0.24838839863311443
Value along trajectory [2, 2, 0] 0.27453771708872743
Value along trajectory [2, 3, 1] 0.3199938985315158
Value along trajectory [2, 4, 0] 0.27454239512308914
Value along trajectory [2, 3, 1] 0.3199938985315158
Value along trajectory [3, 3, 2] 0.8524849183254222
Value along trajectory [4, 3, 3] 1.229838808735011
Value along trajectory [5, 3, 5] 2.561844579986735
Value along trajectory [5, 4, 5] 3.795365618707918
Value along trajectory [5, 5, 6] 5.622817632932442
Value along trajectory [5, 6, 6] 9.999153585021714
```

5(c): I already mentioned the result of 5(a). So, here I describe the result from 5(b).
Compared with the result in 5(a), the length of the trajectory is longer with $p_e = 0.25$, which is reasonable because now my reward function awards +1 only when the robot is pointing straight down in the goal point. Hence, the robot needs to take more actions to achieve the pose. Furthermore, the interesting result is shown in Fig.8, where the robot is moving on the yellow markers by having -10 reward. This is because the algorithm (in this case, my policy iteration algorithm) guess that it's better to traverse yellow areas than to traverse the trajectory around the yellow area such as Fig.7. In Fig.8, the error possibility is 25 %, and the dynamics of the robot is stochastic, so the planner guess that randomness increases as the length of the trajectory increases (i.e. the distribution of the robot evolves), so the algorithm decided that it would be better to select a trajectory with a shorter distance, even if the robot get a negative reward.

Reference

1. Sebastian Thrun , Wolfram Burgard , Dieter Fox, Probabilistic Robotics (Intelligent Robotics and Autonomous Agents), The MIT Press, 2005
2. Pieter Abbeel, CS 287: Advanced Robotics, Fall 2019, UC Berkeley, https://people.eecs.berkeley.edu/~pabbeel/cs287-fa19/ (2019/10/12 Access)