Project3_505851728_005627440_505297814

May 17, 2022

1 Project3: Reinforcement learning and Inverse Reinforcement learning

- 505851728 Yang-Shan Chen
- 005627440 Chih-En Lin
- 505297814 Rikako Hatoya

2 Q1

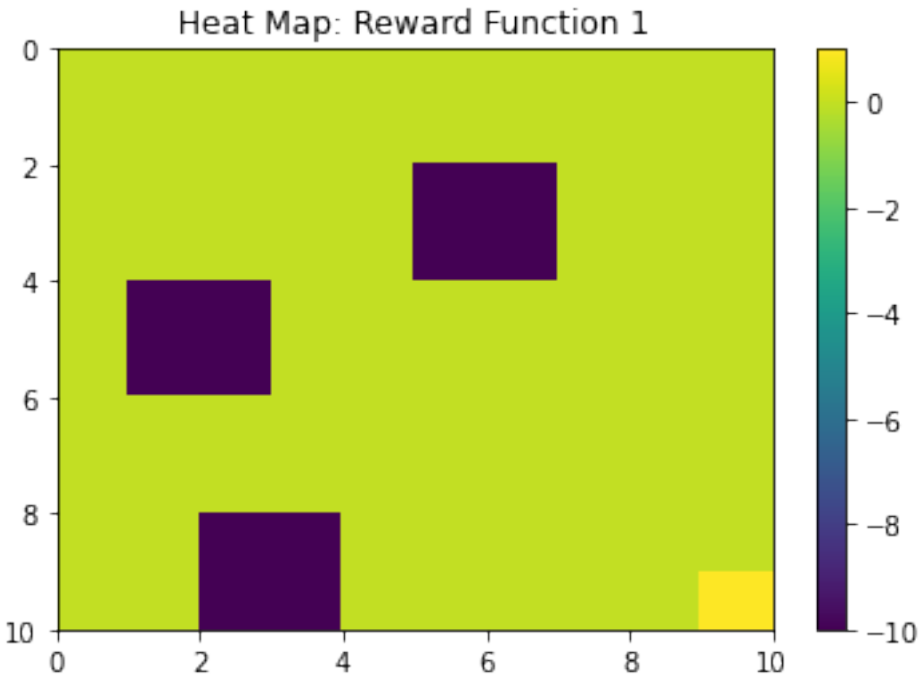
[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import numpy.random as rn
```

[ ]:
```
reward_function_1 = [[0]*10 for _ in range(10)]
reward_function_1[2][5] = -10
reward_function_1[2][6] = -10
reward_function_1[3][5] = -10
reward_function_1[3][6] = -10
reward_function_1[4][1] = -10
reward_function_1[4][2] = -10
reward_function_1[5][1] = -10
reward_function_1[5][2] = -10
reward_function_1[8][2] = -10
reward_function_1[8][3] = -10
reward_function_1[9][2] = -10
reward_function_1[9][3] = -10
reward_function_1[9][9] = 1
reward_function_1 = np.array(reward_function_1)
```

[ ]:
```
reward_function_2 = [[0]*10 for _ in range(10)]
reward_function_2[1][4] = -100
reward_function_2[1][5] = -100
reward_function_2[1][6] = -100
reward_function_2[2][4] = -100
reward_function_2[2][6] = -100
reward_function_2[3][4] = -100
reward_function_2[3][6] = -100
```

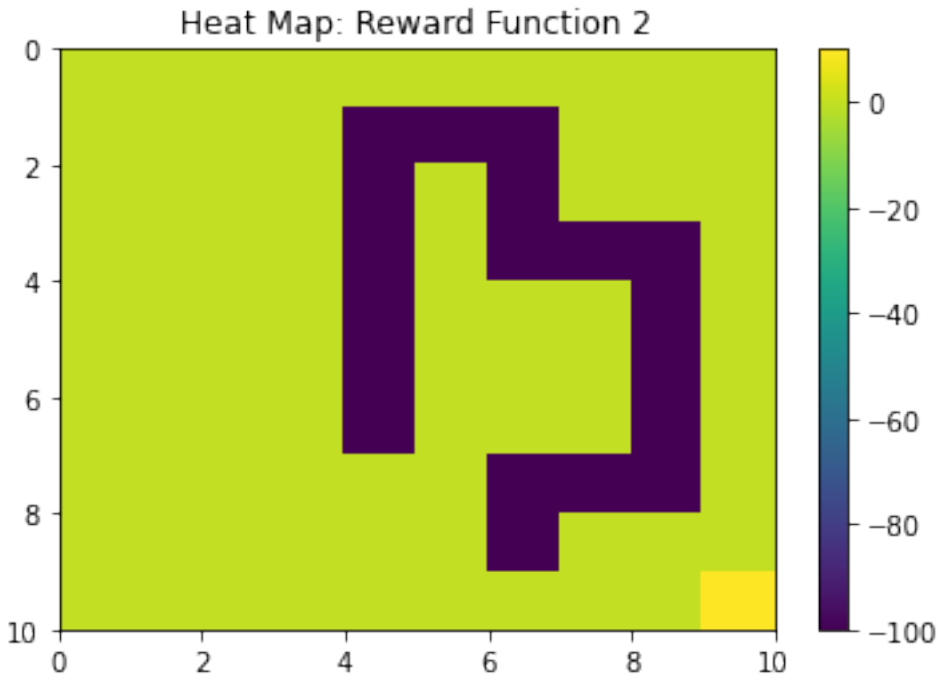
```
reward_function_2[3][7] = -100
reward_function_2[3][8] = -100
reward_function_2[4][4] = -100
reward_function_2[4][8] = -100
reward_function_2[5][4] = -100
reward_function_2[5][8] = -100
reward_function_2[6][4] = -100
reward_function_2[6][8] = -100
reward_function_2[7][6] = -100
reward_function_2[7][7] = -100
reward_function_2[7][8] = -100
reward_function_2[8][6] = -100
reward_function_2[9][9] = 10
reward_function_2 = np.array(reward_function_2)
```

[ ]:
```
plt.title("Heat Map: Reward Function 1")
plt.pcolor(reward_function_1)
plt.gca().invert_yaxis()
plt.colorbar()
plt.show()
```



[ ]:
```
plt.title("Heat Map: Reward Function 2")
plt.pcolor(reward_function_2)
plt.gca().invert_yaxis()
```

```
plt.colorbar()
plt.show()
```



3 Q2

[ ]:
```
class Gridworld(object):
    def __init__(self, grid_size, wind, discount, reward_function):
        self.actions = ((1, 0), (0, 1), (-1, 0), (0, -1)) #down, right, up, left
        self.n_actions = len(self.actions)
        self.n_states = grid_size**2
        self.grid_size = grid_size
        self.wind = wind
        self.discount = discount
        self.reward_function = reward_function

        self.transition_probability = np.array(
            [[[self._transition_probability(i, j, k)
               for k in range(self.n_states)]
              for i in range(self.n_states)]
             for j in range(self.n_actions)])

    def __str__(self):
        return "Gridworld({}, {}, {})".format(self.grid_size, self.wind,
```

```
                                                self.discount)

    def int_to_point(self, i):
        return (i % self.grid_size, i // self.grid_size)

    def point_to_int(self, p):
        return int(p[0] + p[1]*self.grid_size)

    def neighbouring(self, i, k):
        return abs(i[0] - k[0]) + abs(i[1] - k[1]) <= 1

    def _transition_probability(self, i, j, k):
        xi, yi = self.int_to_point(i)
        xj, yj = self.actions[j]
        xk, yk = self.int_to_point(k)


        if not self.neighbouring((xi, yi), (xk, yk)):
            return 0

        if (xi + xj, yi + yj) == (xk, yk):
            return 1-self.wind+self.wind/4

        if (xi, yi) != (xk, yk):
            return self.wind/4

        if (((xi == 0) and (yi == 0)) or ((xi == 0) and (yi == self.
↪grid_size-1)) or ((xi == self.grid_size-1) and (yi == 0)) or ((xi == self.
↪grid_size-1) and (yi == self.grid_size-1))):
            if ((xi+xj<0) or (xi+xj > self.grid_size-1) or (yi+yj <0) or (yi+yj␣
↪> self.grid_size-1)):
                return 1-self.wind+2*self.wind/4
            else:
                return 2*self.wind/4
        else:
            if ((xi != 0) and (xi != self.grid_size-1) and (yi != 0) and (yi !=␣
↪self.grid_size-1)):
                return 0

            else:
                if ((xi+xj < 0) or (xi+xj > self.grid_size-1) or (yi+yj < 0) or␣
↪(yi+yj > self.grid_size-1)):
                    return 1-self.wind+self.wind/4
                else:
                    return self.wind/4

    def reward(self, state_int):
```

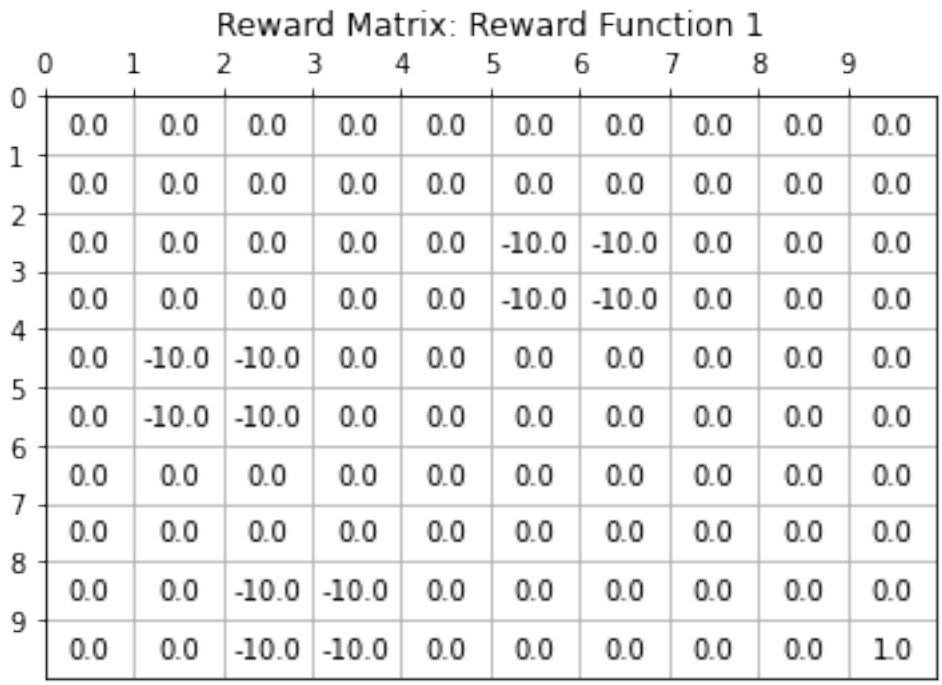
```
        xk, yk = self.int_to_point(state_int)
        return self.reward_function[xk][yk]
```

[ ]:
```
def plot_matrix(matrix, title):
    fig, ax = plt.subplots()
    num_rows = len(matrix)
    min_val, max_val = 0, num_rows

    for i in range(num_rows):
        for j in range(num_rows):
            c = matrix[i][j]
            ax.text(j + 0.5, i + 0.5, '{:.1f}'.format(c), va='center',␣
 ↪ha='center')

    ax.set_xlim(min_val, max_val)
    ax.set_ylim(max_val, min_val)
    ax.set_xticks(np.arange(max_val))
    ax.set_yticks(np.arange(max_val))
    ax.xaxis.tick_top()
    ax.grid()
    plt.title(title)
    plt.show()
    plt.close()
```

[ ]:
```
def reward_grid_plot(title):
    reward_matrix = np.zeros((grid_size, grid_size))
    for j in range(grid_size):
        for i in range(grid_size):
            reward_matrix[i][j] = gw.reward(gw.point_to_int((i,j)))
    plot_matrix(reward_matrix, title)
    return reward_matrix
```

[ ]:
```
grid_size = 10
wind = 0.1
discount = 0.8

gw = Gridworld(grid_size, wind, discount, reward_function_1)
```

[ ]:
```
reward_matrix = reward_grid_plot("Reward Matrix: Reward Function 1")
```



[ ]:
```
intermediate_steps = 5

def optimal_value(n_states, n_actions, transition_probabilities, reward,
                  discount, threshold=1e-2):
    v_array_intermediate=[]
    number_of_step = []
    N = 0
    v = np.zeros(n_states)
    delta = np.inf
    reward = (reward.T).ravel()
    while delta > threshold:
      N += 1
      v_small = v
      v = np.amax(np.matmul(transition_probabilities,reward+discount*v), axis=0)
      delta=np.amax(np.abs(v_small-v))
      if N%intermediate_steps==0:
            number_of_step.append(N)
            v_array_intermediate.append(np.transpose(v.reshape(10,10)))
    return N, v, v_array_intermediate, number_of_step
```

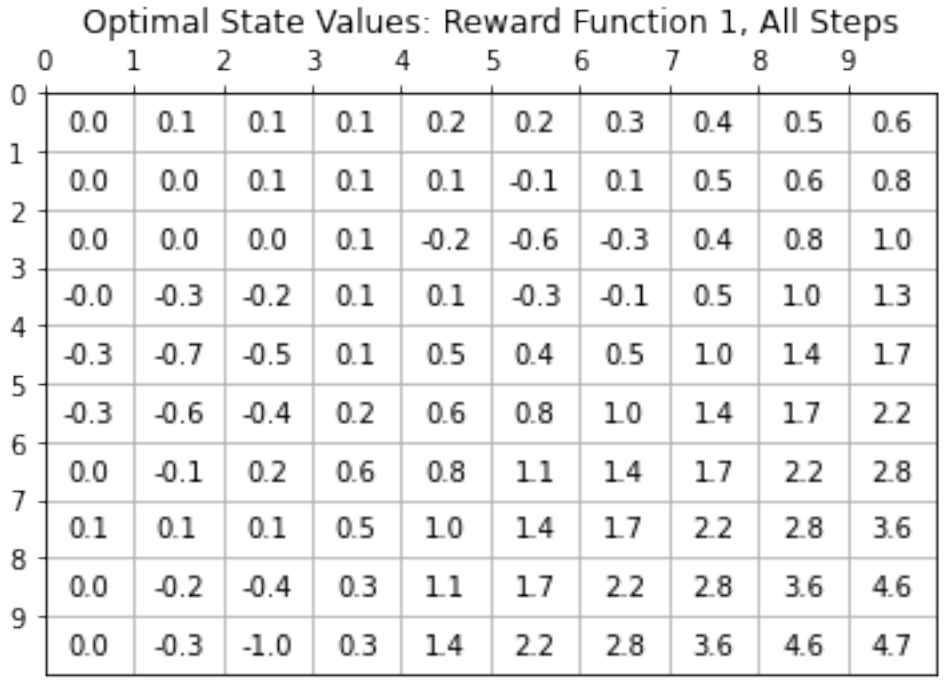
[ ]:
```
reward_states = np.zeros(gw.n_states)
for i in range(gw.n_states):
    reward_states[i] = gw.reward(i)
```

```
N, v, v_array_intermediate, number_of_step = optimal_value(gw.n_states, gw.
 ↪n_actions, gw.transition_probability, reward_states, gw.discount)

value_matrix = np.zeros((grid_size, grid_size))
for i in range(gw.n_states):
    value_matrix[int(i%grid_size)][int(i/grid_size)] = np.round(v[i], 1)
```

[ ]:
```
print("N: ", N)
print("Step Numbers: ", number_of_step)
```

```
N:  22
Step Numbers:  [5, 10, 15, 20]
```

[ ]:
```
plot_matrix(value_matrix, "Optimal State Values: Reward Function 1, All Steps")
```



[ ]:
```
for i in range(intermediate_steps-1):
  plot_matrix(v_array_intermediate[i], "Optimal State Values: Reward Function␣
 ↪1, Step "+str(number_of_step[i]))
```

Optimal State Values: Reward Function 1, Step 5

	0	1	2	3	4	5	6	7	8	9
0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0
1	-0.0	-0.0	-0.0	-0.0	-0.0	-0.3	-0.3	-0.0	-0.0	-0.0
2	-0.0	-0.0	-0.0	-0.0	-0.3	-0.7	-0.7	-0.3	-0.0	-0.0
3	-0.0	-0.3	-0.3	-0.0	-0.3	-0.7	-0.7	-0.3	-0.0	-0.0
4	-0.3	-0.7	-0.7	-0.3	-0.0	-0.3	-0.3	-0.0	-0.0	0.3
5	-0.3	-0.7	-0.7	-0.3	-0.0	-0.0	-0.0	-0.0	0.3	0.7
6	-0.0	-0.3	-0.3	-0.0	-0.0	-0.0	-0.0	0.3	0.7	1.3
7	-0.0	-0.0	-0.3	-0.3	-0.0	-0.0	0.3	0.7	1.3	2.1
8	-0.0	-0.3	-0.7	-0.7	-0.3	0.3	0.7	1.3	2.1	3.1
9	-0.0	-0.3	-1.0	-1.0	-0.0	0.7	1.3	2.1	3.1	3.2

Optimal State Values: Reward Function 1, Step 10

	0	1	2	3	4	5	6	7	8	9
0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	-0.0	0.1	0.2
1	-0.0	-0.0	-0.0	-0.0	-0.0	-0.3	-0.3	0.1	0.2	0.3
2	-0.0	-0.0	-0.0	-0.0	-0.3	-0.7	-0.7	-0.1	0.3	0.5
3	-0.0	-0.3	-0.3	-0.0	-0.3	-0.7	-0.6	0.1	0.6	0.8
4	-0.3	-0.7	-0.7	-0.3	0.1	-0.1	0.1	0.6	0.9	1.2
5	-0.3	-0.7	-0.7	-0.2	0.2	0.4	0.6	0.9	1.3	1.7
6	-0.0	-0.3	-0.2	0.2	0.4	0.6	0.9	1.3	1.7	2.3
7	-0.0	-0.0	-0.3	0.1	0.6	0.9	1.3	1.7	2.4	3.1
8	-0.0	-0.3	-0.7	-0.2	0.6	1.3	1.7	2.4	3.2	4.2
9	-0.0	-0.3	-1.0	-0.2	0.9	1.7	2.3	3.1	4.2	4.2

Optimal State Values: Reward Function 1, Step 15

	0	1	2	3	4	5	6	7	8	9
0	-0.0	-0.0	-0.0	0.0	0.0	0.1	0.2	0.2	0.4	0.5
1	-0.0	-0.0	-0.0	-0.0	0.0	-0.2	-0.0	0.3	0.5	0.7
2	-0.0	-0.0	-0.0	-0.0	-0.3	-0.7	-0.4	0.2	0.7	0.9
3	-0.0	-0.3	-0.3	-0.0	-0.0	-0.4	-0.2	0.4	0.9	1.2
4	-0.3	-0.7	-0.6	-0.0	0.3	0.2	0.4	0.9	1.2	1.6
5	-0.3	-0.7	-0.5	0.1	0.5	0.7	0.9	1.2	1.6	2.1
6	-0.0	-0.2	0.1	0.5	0.7	0.9	1.2	1.6	2.1	2.7
7	-0.0	-0.0	0.0	0.4	0.9	1.2	1.6	2.1	2.7	3.5
8	-0.0	-0.3	-0.5	0.2	0.9	1.6	2.1	2.7	3.5	4.5
9	-0.0	-0.3	-1.0	0.1	1.3	2.0	2.7	3.5	4.5	4.6

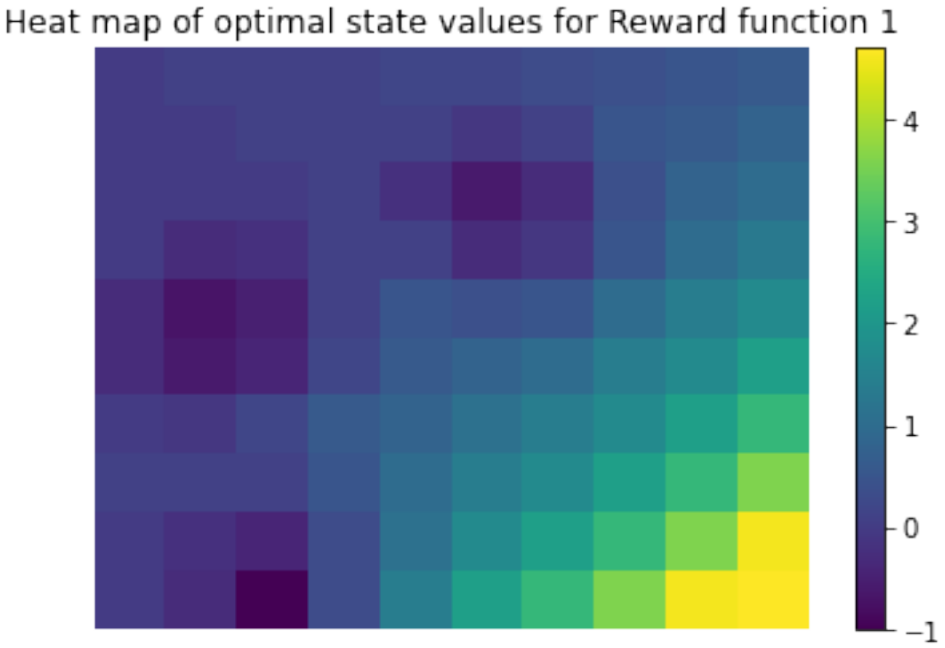
Optimal State Values: Reward Function 1, Step 20

	0	1	2	3	4	5	6	7	8	9
0	0.0	0.0	0.1	0.1	0.1	0.2	0.3	0.4	0.5	0.6
1	0.0	0.0	0.0	0.1	0.1	-0.1	0.1	0.5	0.6	0.8
2	0.0	0.0	0.0	0.0	-0.2	-0.6	-0.3	0.3	0.8	1.0
3	-0.0	-0.3	-0.2	0.0	0.1	-0.3	-0.1	0.5	1.0	1.3
4	-0.3	-0.7	-0.5	0.1	0.4	0.3	0.5	1.0	1.3	1.7
5	-0.3	-0.6	-0.4	0.2	0.6	0.8	1.0	1.3	1.7	2.2
6	0.0	-0.1	0.2	0.6	0.8	1.0	1.3	1.7	2.2	2.8
7	0.0	0.1	0.1	0.5	1.0	1.3	1.7	2.2	2.8	3.6
8	0.0	-0.2	-0.4	0.3	1.1	1.7	2.2	2.8	3.6	4.6
9	-0.0	-0.3	-1.0	0.3	1.4	2.2	2.8	3.6	4.6	4.7

Observations from plots: Optimal state values start from 0 and then increase to its maximum reward value as it progresses through further iterations.

4 Q3

```
[ ]: plt.pcolor(value_matrix)
     plt.gca().invert_yaxis()
     plt.colorbar()
     plt.axis('off')
     plt.title('Heat map of optimal state values for Reward function 1')
     plt.show()
```



5 Q4

Explain the distribution of the optimal state values across the 2-D grid. (Elaborate on Q3 plot) As observed from the heat map, optimal state values seem to get larger as the state progresses to the right bottom corner of the grid. Furthermore, 3 hotspots can be identified in the heat map, which indicates regions that have low optimal state values. This suggests that the optimal path would be to follow along the edges or through the center and reach the bottom right corner. Furthermore, the "hotspots" on the heat map reflects the regions with low scores in the reward function.

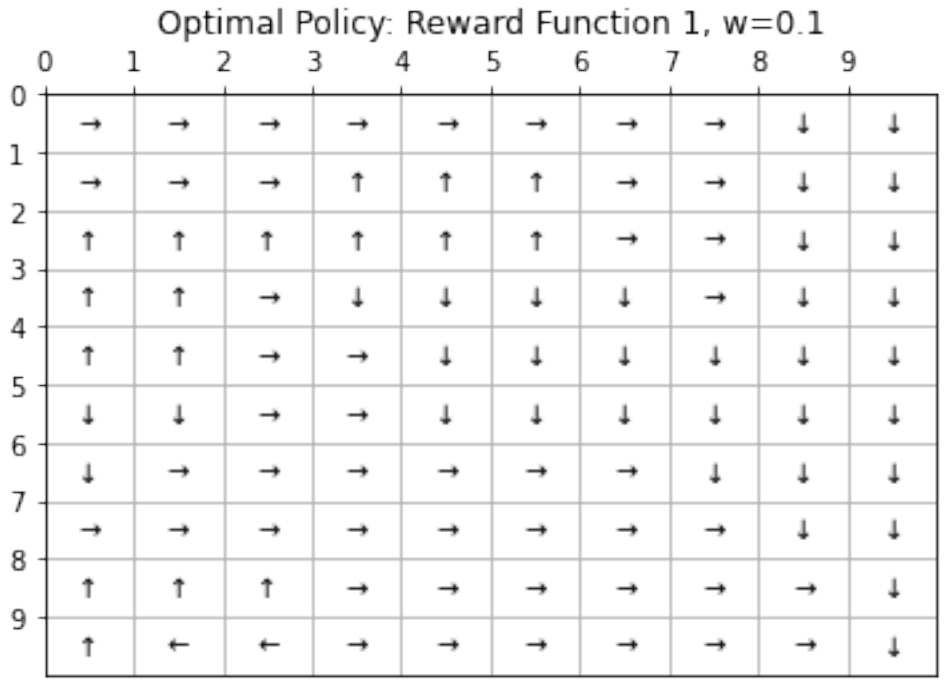
6 Q5

[ ]:
```
from matplotlib.image import PcolorImage

def find_policy(n_states, n_actions, transition_probabilities, reward, discount,
                threshold=1e-2, v=None, stochastic=False):

    if v is None:
        v = optimal_value(n_states, n_actions, transition_probabilities, reward,
                          discount, threshold)

    v = np.zeros(n_states)
    delta = np.inf
    reward = (reward_function_1.T).ravel()

    while delta > threshold:
      v_small = v
      v = np.amax(np.matmul(transition_probabilities,reward+discount*v), axis=0)
      delta=np.amax(np.abs(v_small-v))

    pi = np.argmax(np.matmul(transition_probabilities,reward+discount*v),␣
 ↪axis=0)
    return pi
```

[ ]:
```
def plot_arrow(action_matrix, title):

    fig, ax = plt.subplots()
    num_rows = len(action_matrix)
    min_val, max_val = 0, num_rows

    for i in range(num_rows):
        for j in range(num_rows):
            c = action_matrix[i][j]
            arrow = ''
            if(c == 0):
                arrow = u'↓'
            elif(c == 1):
                arrow = u'→'
            elif(c == 2):
                arrow = u'↑'
            else:
                arrow = u'←'

            ax.text(j + 0.5, i + 0.5, arrow, va='center', ha='center')

    ax.set_xlim(min_val, max_val)
    ax.set_ylim(max_val, min_val)
```

```
    ax.set_xticks(np.arange(max_val))
    ax.set_yticks(np.arange(max_val))
    ax.xaxis.tick_top()
    ax.grid()
    plt.title(title)
```

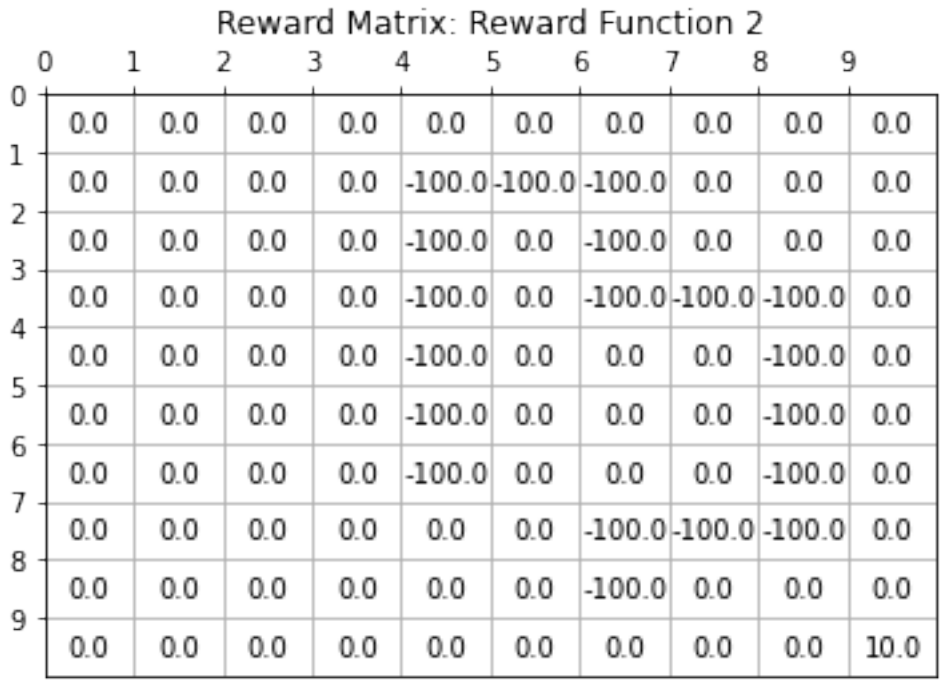
[ ]:
```
optimal_policy = find_policy(gw.n_states, gw.n_actions, gw.
 ↪transition_probability, reward_states, gw.discount, stochastic=False)

action_matrix = np.zeros((grid_size, grid_size))
for i in range(gw.n_states):
    action_matrix[int(i%grid_size)][int(i/grid_size)] = optimal_policy[i]

plot_arrow(action_matrix, "Optimal Policy: Reward Function 1, w=0.1")
```



Does the optimal policy of the agent match your intuition? Please provide a brief explanation. Is it possible for the agent to compute the optimal action to take at each state by observing the optimal values of it's neighboring states? Yes, it macthes our intuition of progressing down and to the right as much as possible to reach the maximum reward (state 99) as fast as possible, as we can observe with the diagonal line through the center in the optimal policy graph above. It is possible to some extent to compute the optimal action by observing its neighboring states as actions will be taken to progress to a neighbor with a high optimal state value and avoid a low optimal state value.

7 Q6

```
[ ]: gw = Gridworld(grid_size, wind, discount, reward_function_2)
reward_matrix = reward_grid_plot("Reward Matrix: Reward Function 2")
```



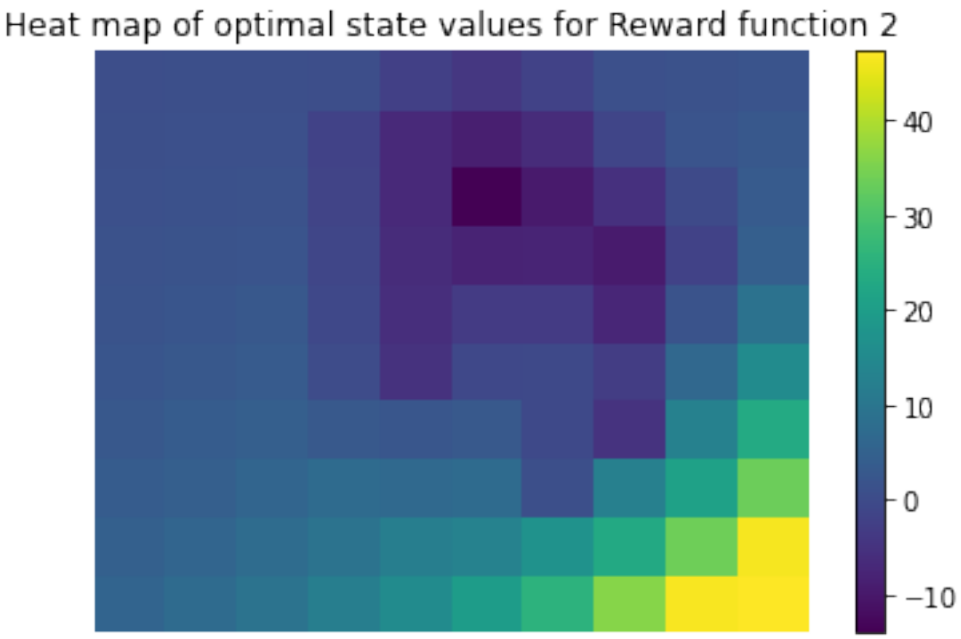

```
[ ]: reward_states = np.zeros(gw.n_states)
for i in range(gw.n_states):
    reward_states[i] = gw.reward(i)

N, v, v_array_intermediate, number_of_step = optimal_value(gw.n_states, gw.
 ↪n_actions, gw.transition_probability, reward_states, gw.discount)

value_matrix = np.zeros((grid_size, grid_size))
for i in range(gw.n_states):
    value_matrix[int(i%grid_size)][int(i/grid_size)] = np.round(v[i], 1)

plot_matrix(value_matrix, "Optimal State Values: Reward Function 2, All Steps")
```

Optimal State Values: Reward Function 2, All Steps

	0	1	2	3	4	5	6	7	8	9
0	0.6	0.8	0.8	0.5	-2.4	-4.2	-1.9	1.1	1.6	2.0
1	0.8	1.0	1.1	-1.9	-6.8	-8.7	-6.4	-1.3	1.9	2.6
2	1.1	1.3	1.4	-1.6	-6.8	-13.9	-9.7	-5.5	-0.1	3.4
3	1.4	1.7	1.9	-1.2	-6.3	-8.0	-7.9	-9.4	-1.9	4.4
4	1.7	2.2	2.6	-0.7	-5.8	-3.3	-3.2	-7.4	1.7	9.2
5	2.2	2.8	3.4	-0.0	-5.1	-0.6	-0.5	-3.0	6.6	15.4
6	2.8	3.6	4.5	3.0	2.5	2.9	-0.5	-4.9	12.7	23.3
7	3.6	4.5	5.8	7.3	6.7	7.2	0.9	12.4	21.2	33.5
8	4.6	5.8	7.4	9.4	12.0	12.9	17.1	23.0	33.8	46.5
9	5.7	7.3	9.4	12.0	15.5	19.8	25.5	36.2	46.6	47.3

8 Q7

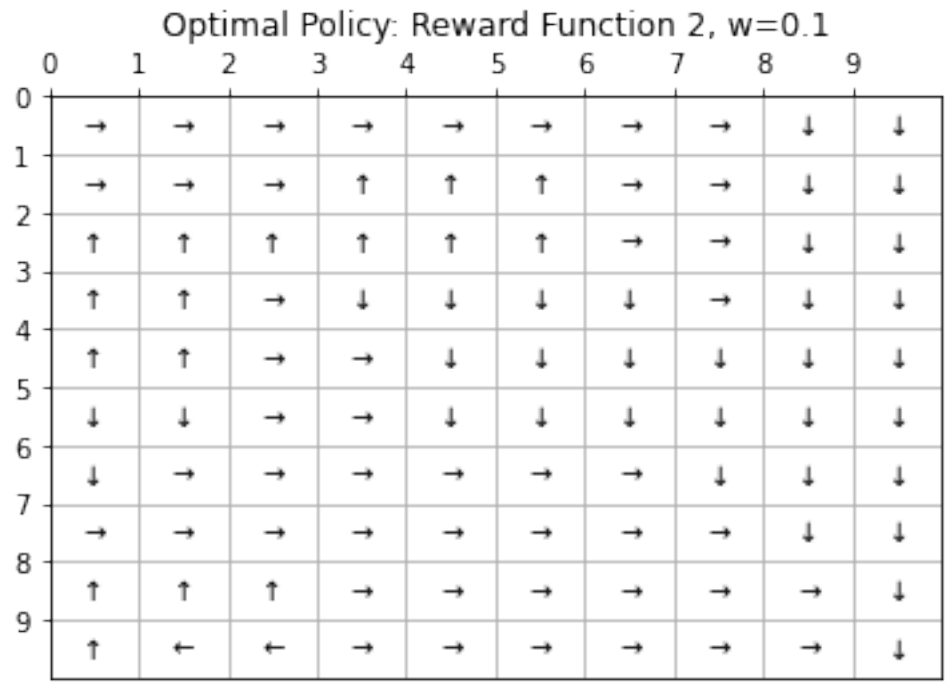
[ ]:
```
plt.pcolor(np.flipud(value_matrix))
plt.colorbar()
plt.axis('off')
plt.title('Heat map of optimal state values for Reward function 2')
plt.show()
```



Explain the distribution of the optimal state values across the 2-D grid. Similar to the optimal state values plot for Reward Function 1, the region with the highest optimal state values are the bottom right hand corner (state 99.) However, only one "hotspot" (region with low optimal state values) can be observed and it is centered and shifted to the upper right hand side, in correspondence to the Reward Function 2 grid.

9 Q8

[ ]:
```
optimal_policy = find_policy(gw.n_states, gw.n_actions, gw.
 ↪transition_probability, reward_states, gw.discount, stochastic=False)

action_matrix = np.zeros((grid_size, grid_size))
for i in range(gw.n_states):
    action_matrix[int(i%grid_size)][int(i/grid_size)] = optimal_policy[i]

plot_arrow(action_matrix, "Optimal Policy: Reward Function 2, w=0.1")
```

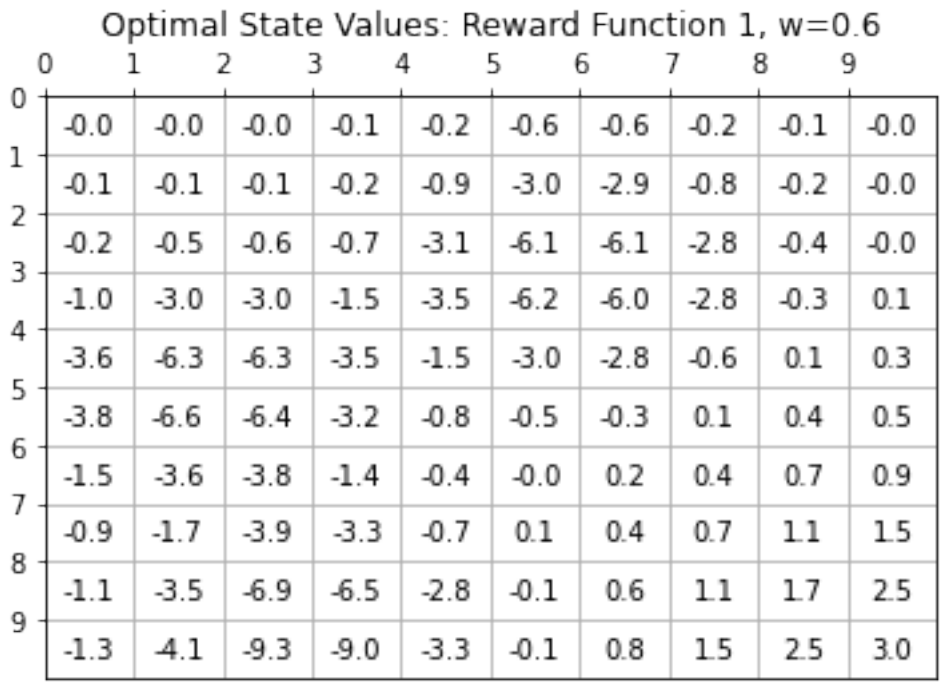


Does the optimal policy of the agent match your intuition? Please provide a brief explanation. Yes, it matches our intuition as regions with low reward values have arrows pointing out of it, which suggests that the agent will try to avoid that path and follow actions that will give a higher reward.

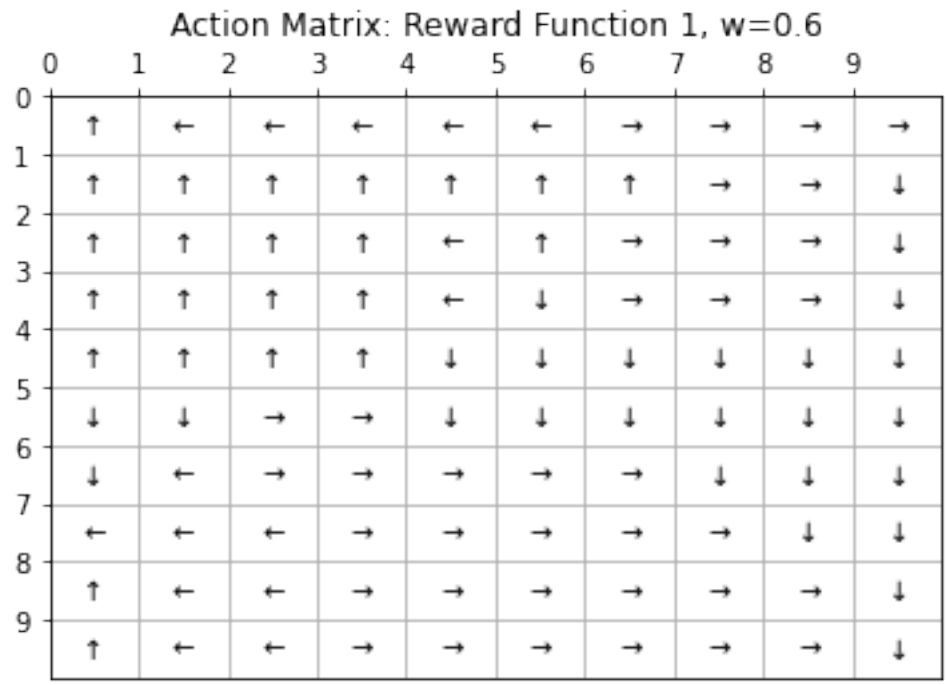
10 Q9

[ ]:
```
grid_size = 10
wind = 0.6
discount = 0.8
gw1 = Gridworld(grid_size, wind, discount, reward_function_1)
```

[ ]:
```
reward_states = np.zeros(gw1.n_states)
for i in range(gw1.n_states):
    reward_states[i] = gw1.reward(i)

N, v, v_array_intermediate, number_of_step = optimal_value(gw1.n_states, gw1.
 ↪n_actions, gw1.transition_probability, reward_states, gw1.discount)

value_matrix = np.zeros((grid_size, grid_size))
for i in range(gw1.n_states):
    value_matrix[int(i%grid_size)][int(i/grid_size)] = np.round(v[i], 1)

plot_matrix(value_matrix, "Optimal State Values: Reward Function 1, w=0.6")
```

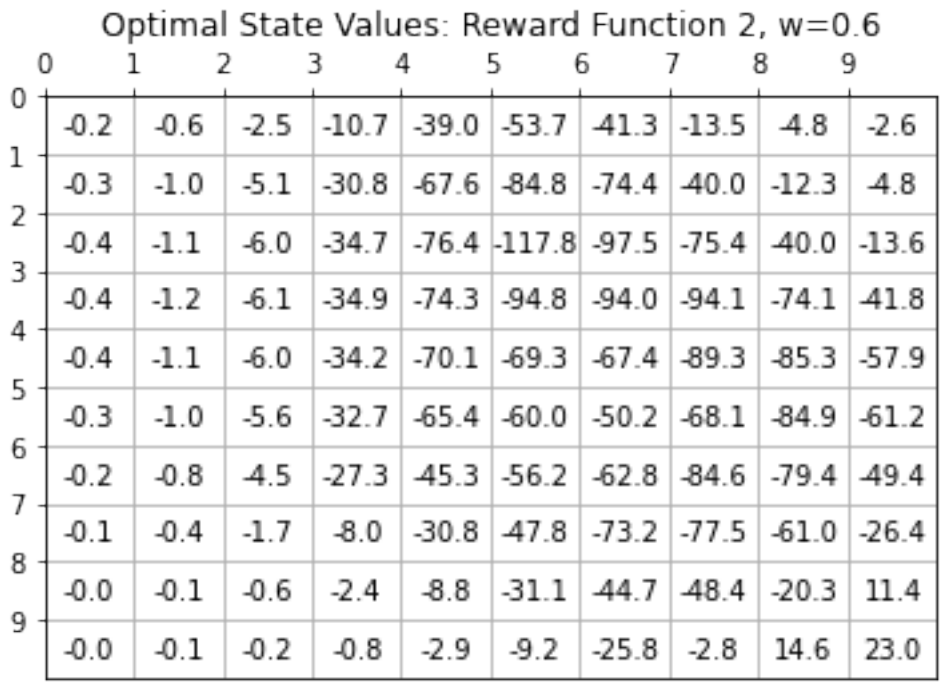


[ ]:
```
optimal_policy = find_policy(gw1.n_states, gw1.n_actions, gw1.
 ↪transition_probability, reward_states, gw1.discount, stochastic=False)

action_matrix = np.zeros((grid_size, grid_size))
for i in range(gw1.n_states):
    action_matrix[int(i%grid_size)][int(i/grid_size)] = optimal_policy[i]

plot_arrow(action_matrix, "Action Matrix: Reward Function 1, w=0.6")
```

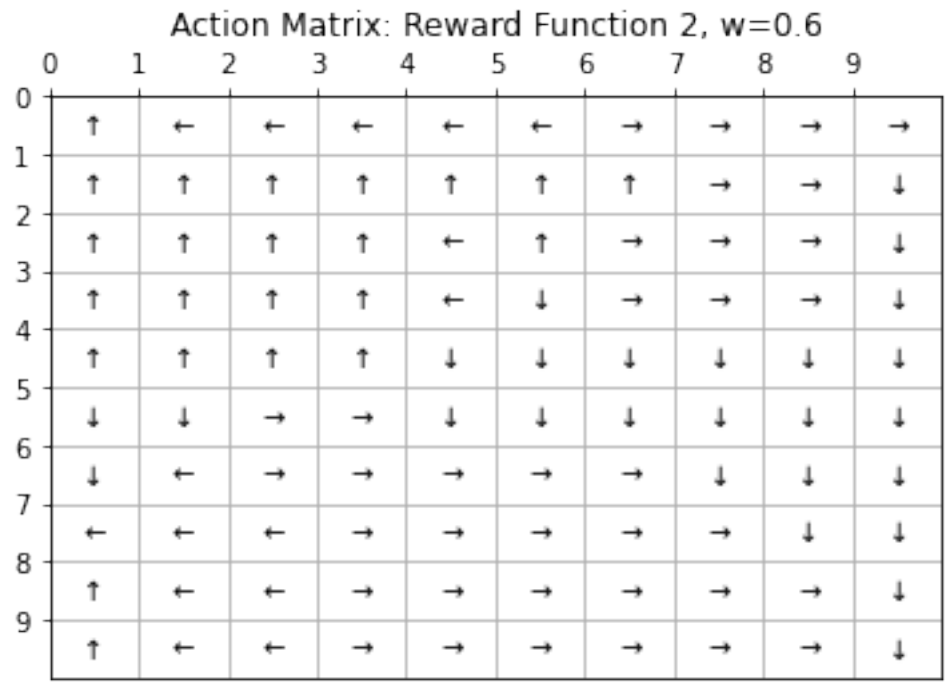


[ ]:
```
grid_size = 10
wind = 0.6
discount = 0.8
gw2 = Gridworld(grid_size, wind, discount, reward_function_2)
```

[ ]:
```
reward_states = np.zeros(gw2.n_states)
for i in range(gw2.n_states):
    reward_states[i] = gw2.reward(i)

N, v, v_array_intermediate, number_of_step = optimal_value(gw2.n_states, gw2.
 ↪n_actions, gw2.transition_probability, reward_states, gw2.discount)

value_matrix = np.zeros((grid_size, grid_size))
for i in range(gw2.n_states):
    value_matrix[int(i%grid_size)][int(i/grid_size)] = np.round(v[i], 1)

plot_matrix(value_matrix, "Optimal State Values: Reward Function 2, w=0.6")
```

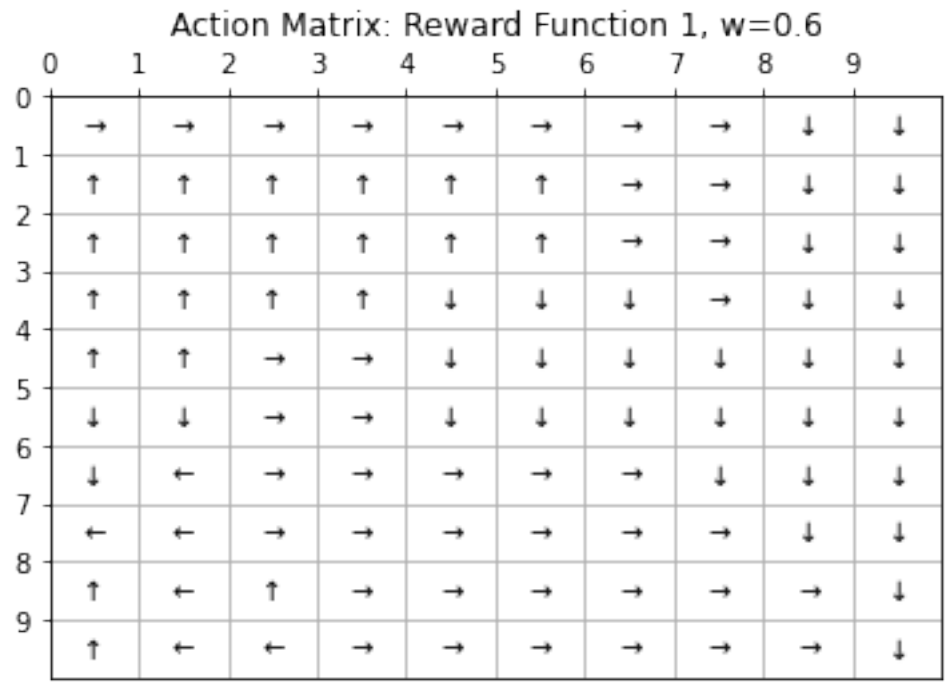


[ ]:
```
optimal_policy = find_policy(gw2.n_states, gw2.n_actions, gw2.
 ↪transition_probability, reward_states, gw2.discount, stochastic=False)

action_matrix = np.zeros((grid_size, grid_size))
for i in range(gw2.n_states):
    action_matrix[int(i%grid_size)][int(i/grid_size)] = optimal_policy[i]

plot_arrow(action_matrix, "Action Matrix: Reward Function 2, w=0.6")
```

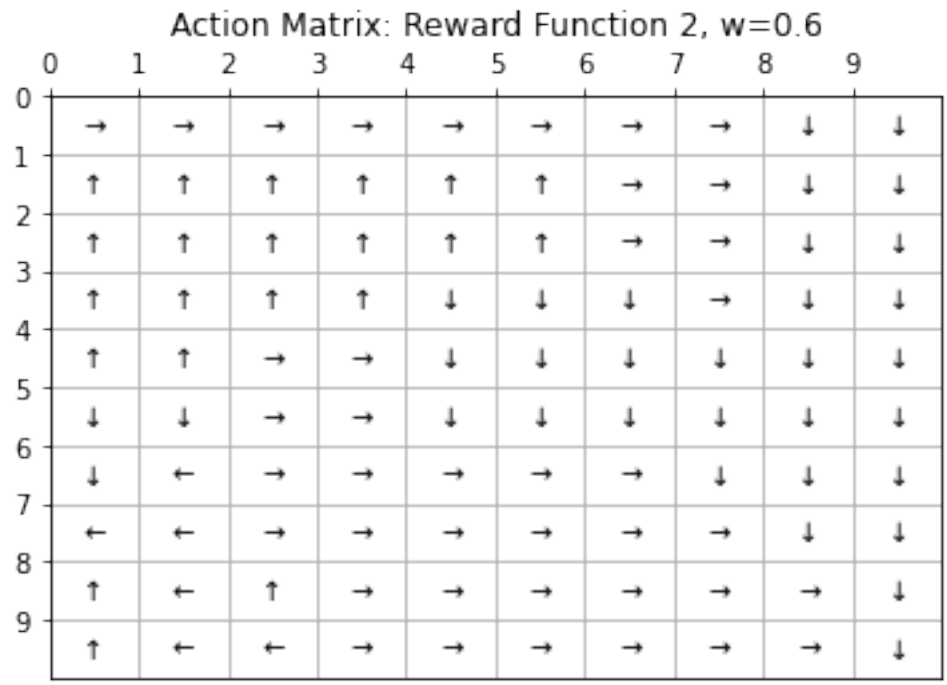


Explain the differences you observe. What do you think about value of new w compared to previous value? When w is modified from 0.1 to 0.6, both action matrices for Reward Function 1 & 2 shows arrows pointing upwards and to the left for the upper left hand region and arrows pointing to the right and downwards for the lower right hand quartile. This indicates that it is more difficult to reach the maximum reward compared to when w=0.1.

[ ]:
```
grid_size = 10
wind = 0.2
discount = 0.8
gw3 = Gridworld(grid_size, wind, discount, reward_function_1)

reward_states = np.zeros(gw3.n_states)
for i in range(gw3.n_states):
    reward_states[i] = gw3.reward(i)

optimal_policy = find_policy(gw3.n_states, gw3.n_actions, gw3.
 ↪transition_probability, reward_states, gw3.discount, stochastic=False)

action_matrix = np.zeros((grid_size, grid_size))
for i in range(gw3.n_states):
    action_matrix[int(i%grid_size)][int(i/grid_size)] = optimal_policy[i]

plot_arrow(action_matrix, "Action Matrix: Reward Function 1, w=0.6")
```



[ ]:
```
grid_size = 10
wind = 0.2
discount = 0.8
gw3 = Gridworld(grid_size, wind, discount, reward_function_2)

reward_states = np.zeros(gw3.n_states)
for i in range(gw3.n_states):
    reward_states[i] = gw3.reward(i)

optimal_policy = find_policy(gw3.n_states, gw3.n_actions, gw3.
 ↪transition_probability, reward_states, gw3.discount, stochastic=False)

action_matrix = np.zeros((grid_size, grid_size))
for i in range(gw3.n_states):
    action_matrix[int(i%grid_size)][int(i/grid_size)] = optimal_policy[i]

plot_arrow(action_matrix, "Action Matrix: Reward Function 2, w=0.6")
```



For both reward function 1&2, w=0.2 is observed to give the best optimal policy. As the value of w is increased, optimal policy arrows point towards up and to the left, indicating that the agent will stay at state 0 as where when w=0.2, a large portion of arrows point towards downward and right, allowing the agent to succesfully reach the maximum reward.

11 Q10~ Initialization

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

from tqdm import tqdm
from cvxopt import solvers, matrix
```

[ ]:
```
def tp_calc(w):
    tp_u = np.zeros((100, 100)) #Up
    tp_d = np.zeros((100, 100)) #Down
    tp_l = np.zeros((100, 100)) #Left
    tp_r = np.zeros((100, 100)) #Right

    #Up
    for i in range(0, 100):
        if (i%10 == 0):
            tp_u[i][i] += w/4
        else:
```

```
            tp_u[i][i-1] = 1-w+w/4

        if (i-10 < 0):
            tp_u[i][i] += w/4
        else:
            tp_u[i][i-10] = w/4

        if (i+10 > 99):
            tp_u[i][i] += w/4
        else:
            tp_u[i][i+10] = w/4

        if ((i+1)%10 == 0):
            tp_u[i][i] += w/4
        else:
            tp_u[i][i+1] = w/4

        if (i%10 == 0):
            tp_u[i][i] += 1-w

    #Down
    for i in range(0, 100):
        if (i%10 == 0):
            tp_d[i][i] += w/4
        else:
            tp_d[i][i-1] = w/4

        if (i-10 < 0):
            tp_d[i][i] += w/4
        else:
            tp_d[i][i-10] = w/4

        if (i+10 > 99):
            tp_d[i][i] += w/4
        else:
            tp_d[i][i+10] = w/4

        if ((i+1)%10 == 0):
            tp_d[i][i] += w/4
        else:
            tp_d[i][i+1] = 1-w+w/4

        if ((i+1)%10 == 0):
            tp_d[i][i] += 1-w

    #Left
    for i in range(0, 100):
```

```
        if (i%10 == 0):
            tp_l[i][i] += w/4
        else:
            tp_l[i][i-1] = w/4

        if (i-10 < 0):
            tp_l[i][i] += w/4
        else:
            tp_l[i][i-10] = 1-w+w/4

        if (i+10 > 99):
            tp_l[i][i] += w/4
        else:
            tp_l[i][i+10] = w/4

        if ((i+1)%10 == 0):
            tp_l[i][i] += w/4
        else:
            tp_l[i][i+1] = w/4

        if (i-10 < 0):
            tp_l[i][i] += 1-w

    #Right
    for i in range(0, 100):
        if (i%10 == 0):
            tp_r[i][i] += w/4
        else:
            tp_r[i][i-1] = w/4

        if (i-10 < 0):
            tp_r[i][i] += w/4
        else:
            tp_r[i][i-10] = w/4

        if (i+10 > 99):
            tp_r[i][i] += w/4
        else:
            tp_r[i][i+10] = 1-w+w/4

        if ((i+1)%10 == 0):
            tp_r[i][i] += w/4
        else:
            tp_r[i][i+1] = w/4

        if (i+10 > 99):
            tp_r[i][i] += 1-w
```

```
    return tp_u, tp_d, tp_l, tp_r
```

[ ]:
```
def value_iteration(state_num, w, gamma, rwd, epsilon, tp_u, tp_d, tp_l, tp_r):
    state_val=np.zeros(100)
    delta=np.inf
    r = (rwd.T).ravel()
    N = 0
    while (delta > epsilon):
        delta = 0
        old_state_val = np.copy(state_val)
        for s in range(0, 100):
            u_val = np.sum(tp_u[s]*(r+gamma*old_state_val))
            d_val = np.sum(tp_d[s]*(r+gamma*old_state_val))
            l_val = np.sum(tp_l[s]*(r+gamma*old_state_val))
            r_val = np.sum(tp_r[s]*(r+gamma*old_state_val))
            state_val[s] = max(u_val, d_val, r_val, l_val)
            delta = max(delta, abs(old_state_val[s]-state_val[s]))
        N = N+1
    state_val = np.transpose(state_val.reshape(10, 10))

    return state_val, N
```

[ ]:
```
def policy_iteration(state_num, w, gamma, rwd, epsilon, tp_u, tp_d, tp_l, tp_r):
    state_val = np.zeros(100)
    delta = np.inf
    policy = np.zeros(100)
    arrows = np.zeros(100)
    r = (rwd.T).ravel()
    while (delta > epsilon):
        delta = 0
        old_state_val = np.copy(state_val)
        for s in range(0, 100):
            u_val = np.sum(tp_u[s]*(r+gamma*old_state_val))
            d_val = np.sum(tp_d[s]*(r+gamma*old_state_val))
            l_val = np.sum(tp_l[s]*(r+gamma*old_state_val))
            r_val = np.sum(tp_r[s]*(r+gamma*old_state_val))
            state_val[s] = max(u_val, d_val, r_val, l_val)
            delta = max(delta, abs(old_state_val[s]-state_val[s]))
    for s in range(0, 100):
        u_val = np.sum(tp_u[s]*(r+gamma*state_val))
        d_val = np.sum(tp_d[s]*(r+gamma*state_val))
        l_val = np.sum(tp_l[s]*(r+gamma*state_val))
        r_val = np.sum(tp_r[s]*(r+gamma*state_val))
        arr = [u_val, d_val, l_val, r_val] #Up: 0, Down: 1, Left: 2, Right: 3
        policy[s] = np.amax(arr)
        arrows[s] = arr.index(np.amax(arr))
```

```
    arrows = np.transpose(arrows.reshape(10, 10))
    policy = np.transpose(policy.reshape(10, 10))
    pic_arrow = np.chararray((10, 10), unicode=True)
    for i in range(10):
        for j in range(10):
            if (arrows[j][i] == 0.):
                pic_arrow[j][i] = u'\u2191'
            elif (arrows[j][i] == 1.):
                pic_arrow[j][i] = u'\u2193'
            elif (arrows[j][i] == 2.):
                pic_arrow[j][i] = u'\u2190'
            elif (arrows[j][i] == 3.):
                pic_arrow[j][i] = u'\u2192'

    return policy, pic_arrow, arrows
```

12 Q10

$$\mathbf{c} = \begin{bmatrix} \mathbf{1}_{|S|\times 1} \\ -\lambda \cdot \mathbf{1}_{|S|\times 1} \\ \mathbf{0}_{|S|\times 1} \end{bmatrix} \backslash$$

$$\mathbf{x} = \begin{bmatrix} \mathbf{t} \\ \mathbf{u} \\ \mathbf{R} \end{bmatrix}$$

$$\mathbf{b} = \begin{bmatrix} \mathbf{0}_{(2\cdot(|A|-1)\cdot|S|)\times 1} \\ \mathbf{0}_{(2\cdot|S|)\times 1} \\ R_{max} \cdot \mathbf{1}_{|S|\times 1} \\ R_{max} \cdot \mathbf{1}_{|S|\times 1} \end{bmatrix} \backslash$$

$$\mathbf{D} = \begin{bmatrix} \mathbf{I}_{((|A|-1)\cdot|S|)\times|S|} & \mathbf{0}_{((|A|-1)\cdot|S|)\times|S|} & -\left[\left(\mathbf{P}_{a_1} - \mathbf{P}_{a_{2,\ldots,|A|}}\right)\left(\mathbf{I} - \gamma\mathbf{P}_{a_1}\right)^{-1}\mathbf{R}\right]_{((|A|-1)\cdot|S|)\times|S|} \\ \mathbf{0}_{((|A|-1)\cdot|S|)\times|S|} & \mathbf{0}_{((|A|-1)\cdot|S|)\times|S|} & -\left[\left(\mathbf{P}_{a_1} - \mathbf{P}_{a_{2,\ldots,|A|}}\right)\left(\mathbf{I} - \gamma\mathbf{P}_{a_1}\right)^{-1}\mathbf{R}\right]_{((|A|-1)\cdot|S|)\times|S|} \\ \mathbf{0}_{|S|\times|S|} & -\mathbf{I}_{|S|\times|S|} & -\mathbf{I}_{|S|\times|S|} \\ \mathbf{0}_{|S|\times|S|} & -\mathbf{I}_{|S|\times|S|} & \mathbf{I}_{|S|\times|S|} \\ \mathbf{0}_{|S|\times|S|} & \mathbf{0}_{|S|\times|S|} & \mathbf{I}_{|S|\times|S|} \\ \mathbf{0}_{|S|\times|S|} & \mathbf{0}_{|S|\times|S|} & -\mathbf{I}_{|S|\times|S|} \end{bmatrix} \backslash$$

In this case, $|A| = 4$, and $|S| = 100$

13 Q11

[ ]:
```
rwd_1 = np.array([[0,0,0,0,0,0,0,0,0,0],
                  [0,0,0,0,0,0,0,0,0,0],
                  [0,0,0,0,0,-10,-10,0,0,0],
                  [0,0,0,0,0,-10,-10,0,0,0],
                  [0,-10,-10,0,0,0,0,0,0,0],
                  [0,-10,-10,0,0,0,0,0,0,0],
                  [0,0,0,0,0,0,0,0,0,0],
                  [0,0,0,0,0,0,0,0,0,0],
                  [0,0,-10,-10,0,0,0,0,0,0],
                  [0,0,-10,-10,0,0,0,0,0,1]])

rwd_2 = np.array([[0,0,0,0,0,0,0,0,0,0],
                  [0,0,0,0,-100,-100,-100,0,0,0],
                  [0,0,0,0,-100,0,-100,0,0,0],
                  [0,0,0,0,-100,0,-100,-100,-100,0],
                  [0,0,0,0,-100,0,0,0,-100,0],
                  [0,0,0,0,-100,0,0,0,-100,0],
                  [0,0,0,0,-100,0,0,0,-100,0],
                  [0,0,0,0,0,0,-100,-100,-100,0],
                  [0,0,0,0,0,0,-100,0,0,0],
                  [0,0,0,0,0,0,0,0,0,10]])
```

[ ]:
```
state_num = np.zeros((10, 10))
k = 0
for i in range(0, 10):
    for j in range(0, 10):
        state_num[j][i] = k
        k += 1
```

[ ]:
```
def getcDbMatrices(arrows_expert, P_ss, ind,gamma, lambda_val, maximum):
    I = np.identity(100)
    mat1 = np.zeros((300, 100))
    iden_mat = np.zeros((300, 100))
    i = 0
    for s in range(100):
        opt = int(arrows_expert[ind][s])
        for action in range(len(P_ss)):
            if (opt == action):
                continue

            pa1 = P_ss[opt]
            pa = P_ss[action]
            mat1[i, :] = np.matmul((pa1[s]-pa[s]).reshape(1, 100), np.linalg.
 ↪inv(I-gamma*pa1))
            iden_mat[i, s] = 1
```

```
            i += 1
    mat1 *= -1
    R = np.vstack((mat1, mat1, -I, I, I, -I))
    t = np.vstack((iden_mat, np.zeros((700, 100))))
    u = np.vstack((np.zeros((600, 100)), -I, -I, np.zeros((200, 100))))
    D = np.hstack((R, t, u))
    ones = np.zeros((100, 1)) + 1
    c = np.vstack((np.zeros((100, 1)), ones, -lambda_val*ones))
    b = np.zeros((800, 1))
    Rmax = np.zeros((100, 1)) + maximum[ind]
    b = np.vstack((b, Rmax, Rmax))

    return c, D, b
```

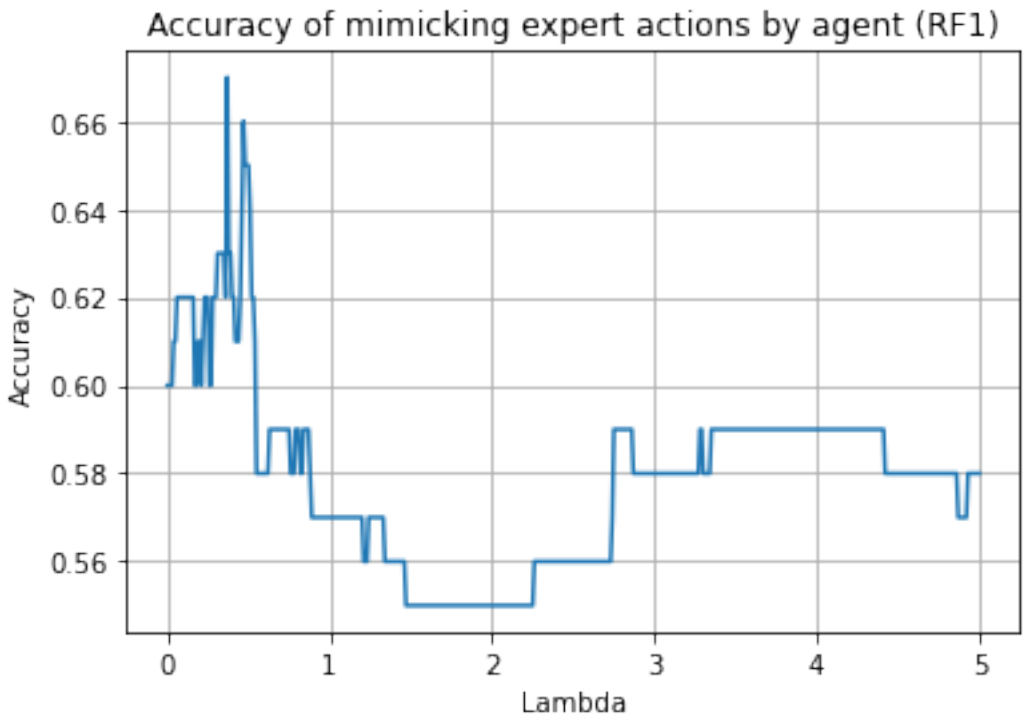
[ ]:
```
w = 0.1
gamma = 0.8
epsilon = 0.01
tp_u, tp_d, tp_l, tp_r = tp_calc(w)

P_ss = [tp_u, tp_d, tp_l, tp_r]

#Policies for Reward function 1 (expert)
V_opt_1, N_1 = value_iteration(state_num, w, gamma, rwd_1, epsilon, tp_r, tp_l,␣
 ↪tp_u, tp_d)
policy_1, pic_arrow_1, arrows_1 = policy_iteration(state_num, w, gamma, rwd_1,␣
 ↪epsilon, tp_u, tp_d, tp_l, tp_r)

#Policies for Reward function 2 (expert)
V_opt_2, N_2 = value_iteration(state_num, w, gamma, rwd_2, epsilon, tp_r, tp_l,␣
 ↪tp_u, tp_d)
policy_2, pic_arrow_2, arrows_2 = policy_iteration(state_num, w, gamma, rwd_2,␣
 ↪epsilon, tp_u, tp_d, tp_l, tp_r)

arrows_expert = [np.transpose(arrows_1).flatten(), np.transpose(arrows_2).
 ↪flatten()]
```

[ ]:
```
lambdas = np.arange(0, 5.01, 0.01)
maximum = [1, 10]
acc_list_list = []
I = np.identity(100)
ind = 0

while(ind < 2):
    acc_list = []
    for i, lambda_val in enumerate(tqdm(lambdas)):

        #Extract reward function
```

```
        c, D, b = getcDbMatrices(arrows_expert, P_ss, ind, gamma, lambda_val,␣
↪maximum)
        solvers.options['show_progress'] = False
        sol = solvers.lp(matrix(c), matrix(D), matrix(b))
        R = np.array(sol['x'][:100])

        #Extract agent's policy from extracted reward function
        policy_agent, pic_arrow_agent, arrows_agent =␣
↪policy_iteration(state_num, w, gamma, R, epsilon, tp_u, tp_d, tp_l, tp_r)

        #Performance measure
        acc = 0
        for j in range(len(np.transpose(arrows_agent).flatten())):
            if (np.transpose(arrows_agent).flatten()[j] ==␣
↪arrows_expert[ind][j]):
                acc += 1
        acc /= 100.0

        acc_list.append(acc)

    acc_list_list.append(acc_list)
    ind += 1
```

```
100%|      | 501/501 [02:12<00:00,  3.78it/s]
100%|      | 501/501 [02:09<00:00,  3.87it/s]
```

[ ]:
```
plt.plot(lambdas, acc_list_list[0])
plt.xlabel("Lambda")
plt.ylabel("Accuracy")
plt.title("Accuracy of mimicking expert actions by agent (RF1)")
plt.grid()
plt.show()
```



14 Q12

[ ]:
```
print("Max value of accuracy: {}".format(acc_list_list[0][np.
  ↪argmax(acc_list_list[0])]))
print("Corresponding value of lambda: {}".format(lambdas[np.
  ↪argmax(acc_list_list[0])]))
```

```
Max value of accuracy: 0.67
Corresponding value of lambda: 0.37
```

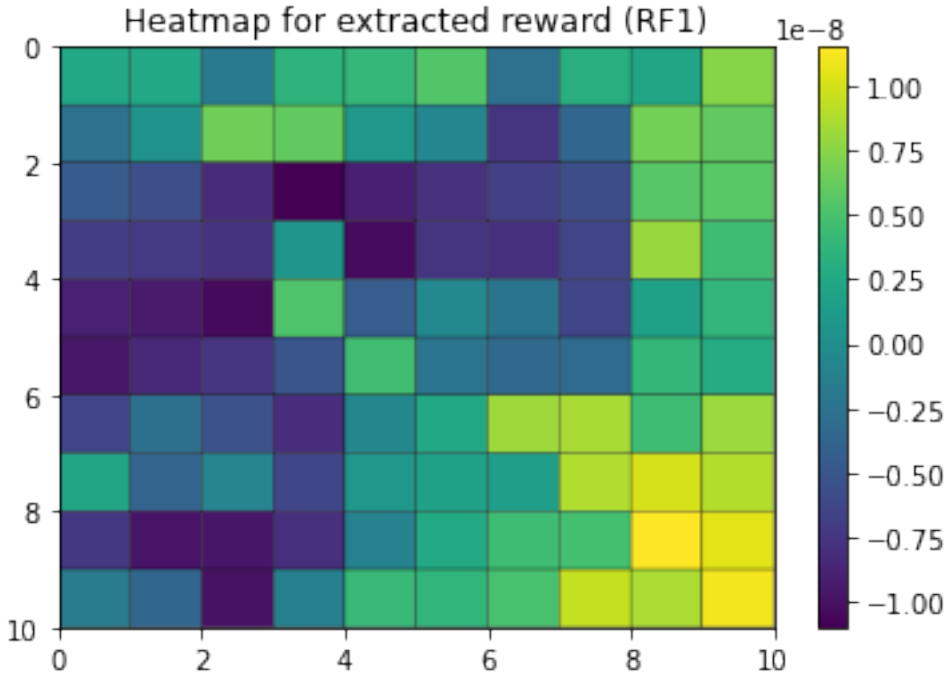
The maximum accuracy is 67%

The corresponding $\lambda_{max}^{(1)}$ is 0.37

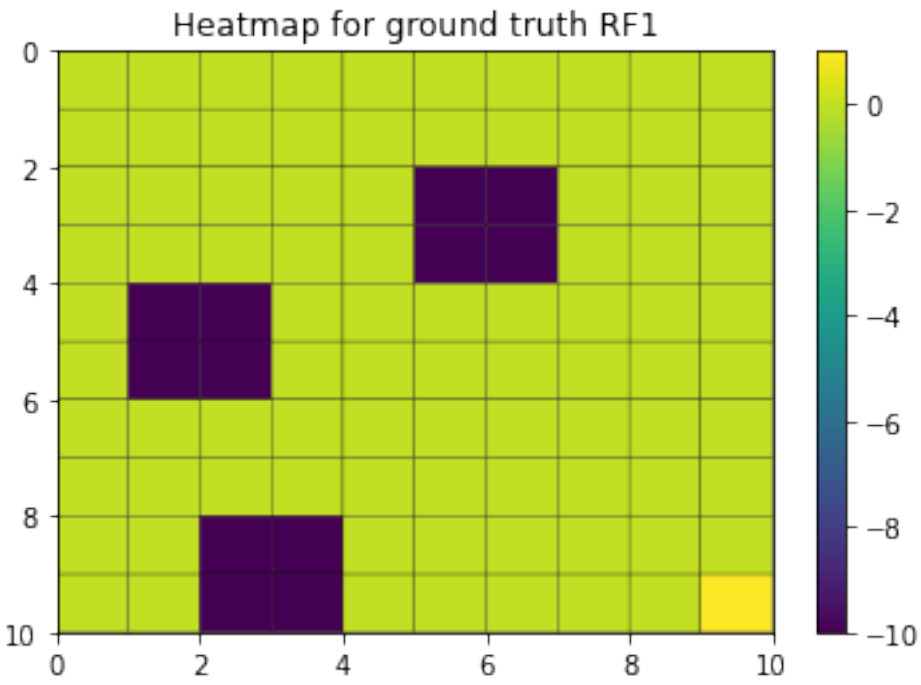
15 Q13

[ ]:
```
c, D, b = getcDbMatrices(arrows_expert, P_ss, 0, gamma, 0.37, maximum)
solvers.options['show_progress'] = False
sol = solvers.lp(matrix(c), matrix(D), matrix(b))
R = np.array(sol['x'][:100])
R = np.transpose(R.reshape(10, 10))
```

[ ]:
```
plt.pcolor(R, edgecolors='black')
plt.gca().invert_yaxis()
plt.colorbar()
plt.title("Heatmap for extracted reward (RF1)")
plt.show()
```



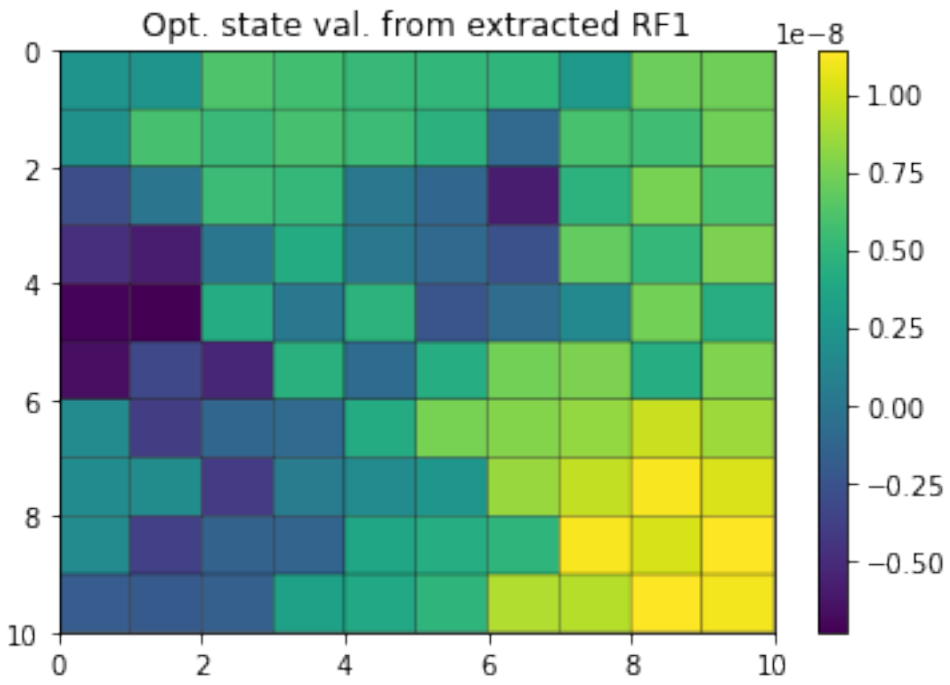
[ ]:
```
plt.pcolor(rwd_1, edgecolors='black')
plt.gca().invert_yaxis()
plt.colorbar()
plt.title("Heatmap for ground truth RF1")
plt.show()
```



16 Q14

[ ]:
```
V_extracted, N = value_iteration(state_num, 0.1, gamma, R, epsilon, tp_r, tp_l,␣
↪tp_u, tp_d)
```

[ ]:
```
plt.pcolor(V_extracted,edgecolors='black')
plt.gca().invert_yaxis()
plt.colorbar()
plt.title("Opt. state val. from extracted RF1")
plt.show()
```



17 Q15

Similarity:

The area near state 99 has the highest optimal values in both plots. Since state 99 provides the highest reward among all the states. When one moves away from state 99, the values would gradually decrease in both plots.

Futhermore, the low optimal values for those states where the reward was negative are approximately similar in the same position in the heatmaps. The patterns in the heatmap of reward function 1 are roughly visible in both heatmaps in Q3 and Q14. There are three blocks of negative rewards in Q3 for reward function 1, and the same regions are also roughly identifiable in Q14.

There is a visible and progressive decay of optimal values around the state with the highest reward in both plots. It is because of the discount factor in the Bellman equation, which discounts future rewards.

Difference:

The scaling is much smaller for the heatmap in Q14 over Q3. Since the heatmap in Q14 is evaluated from the extracted reward, which in turn is calculated from the expert policy and not expert reward. Thus, the scale of optimal state values would not be the same.

The agent is less penalized near a region with lower reward in the state-space in Q14 compared to the ground truth case in Q3. It will accumulate a lower expected reward in the long run for the state-space in Q14 over Q3. As a result, the agent is more likely to move off the grid for the state-space in Q14 over Q3.

The areas in the heatmap in Q3 are much more homogenous and well-defined than Q14.

18 Q16

[ ]:
```
policy_agent, pic_arrow_agent, arrows_agent = policy_iteration(state_num, w,␣
 ↪gamma, R, epsilon, tp_u, tp_d, tp_l, tp_r)

print(pic_arrow_agent)
```

```
[['→' '↓' '↓' '↓' '→' '↑' '←' '→' '→' '↑']
 ['→' '→' '→' '←' '←' '↑' '←' '→' '→' '↑']
 ['↑' '↑' '↑' '↑' '↑' '↑' '→' '→' '↓' '↑']
 ['↑' '↑' '→' '↓' '←' '↓' '→' '→' '↑' '←']
 ['↑' '→' '→' '↑' '←' '↓' '↓' '→' '↑' '↑']
 ['↓' '↓' '→' '↑' '↓' '↓' '↓' '↓' '↓' '↓']
 ['↓' '↓' '↓' '→' '→' '→' '→' '↓' '↓' '↓']
 ['←' '←' '←' '→' '→' '↑' '→' '→' '↓' '↓']
 ['↑' '↑' '↑' '→' '↓' '→' '→' '→' '→' '↓']
 ['↓' '←' '→' '→' '→' '→' '→' '→' '→' '↓']]
```

19 Q17

Similarity:

Most of the actions in both policy maps are moving down and right. It shows that the agent tries to reach the states with the highest rewards in both cases since the most optimal state, state 99 with highest reward, is located in the lower right corner in the state-space in both cases.

Difference:

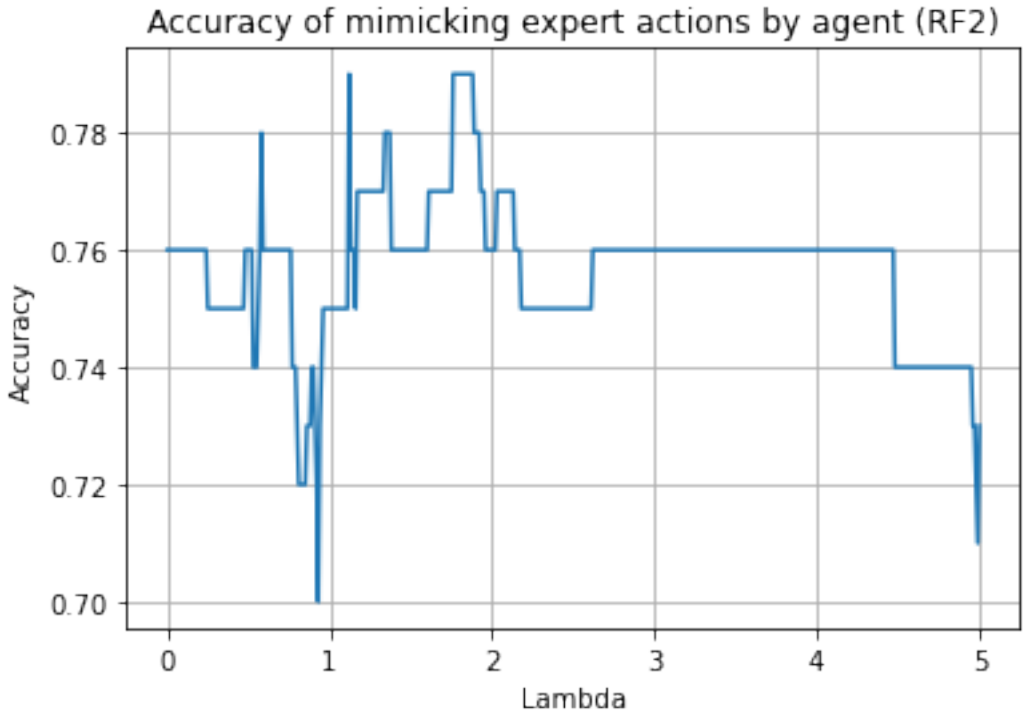
There is more variance and stochasticity in the actions taken in Q16 than Q5. In Q5, the actions are more ordered and homogenous towards state 99.

In Q5, the agent is guaranteed to reach state 99 from any state. While in Q16, there is no guarantee. The agent has a higher risk to get stuck in local optima and deadlock condition or be blown off the grid.

A few actions in Q16 point to each other. These are local optima in the policy plots, where the agent just oscillates between two states forever. This is a result of the gradually changing immediate rewards among neighboring states. If the state value between two neighboring states are very similar and non-zero, then the agent will be forced to go back and forth between the two states if exploration probability is low. On the other hand, deadlock condition is absent for the policy map in Q5, as most of the rewards in the state space are zero. It improves the foresight of the agent and passes on the influence of the high reward state to neighboring states much further than possible for the extracted values in Q16. As a result, the values in the state-space is much less noisy for Q5 over Q16, which allows the agent to filter out the correct direction from a further distance.

20 Q18

[ ]:
```
plt.plot(lambdas,acc_list_list[1])
plt.xlabel("Lambda")
plt.ylabel("Accuracy")
plt.title("Accuracy of mimicking expert actions by agent (RF2)")
plt.grid()
plt.show()
```



21 Q19

[ ]:
```
print("Max value of accuracy: {}".format(acc_list_list[1][np.
  ↪argmax(acc_list_list[1])]))
print("Corresponding value of lambda: {}".format(lambdas[np.
  ↪argmax(acc_list_list[1])]))
```

```
Max value of accuracy: 0.79
Corresponding value of lambda: 1.12
```

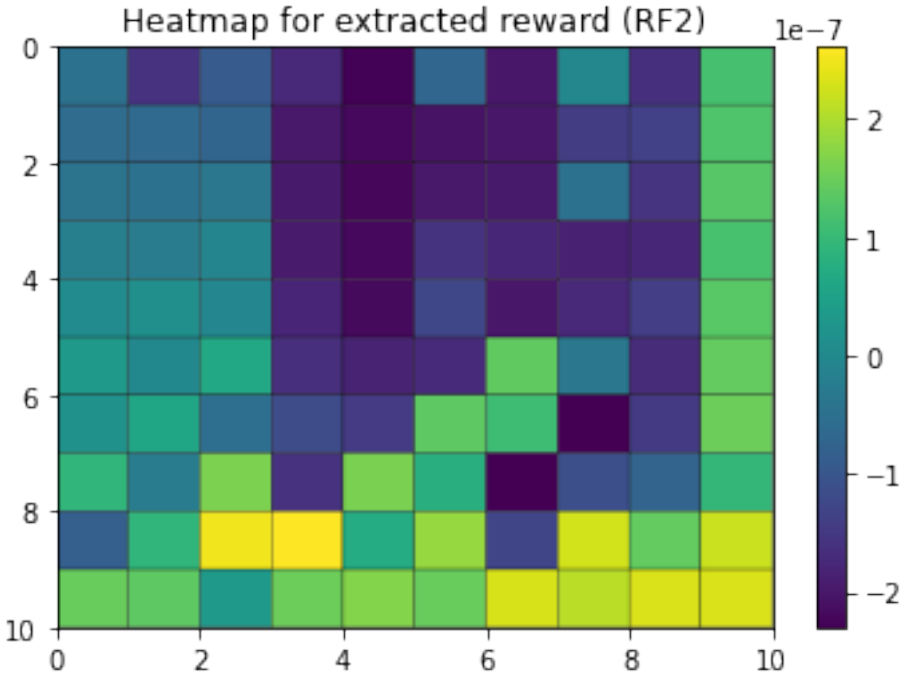
The maximum accuracy is 79%

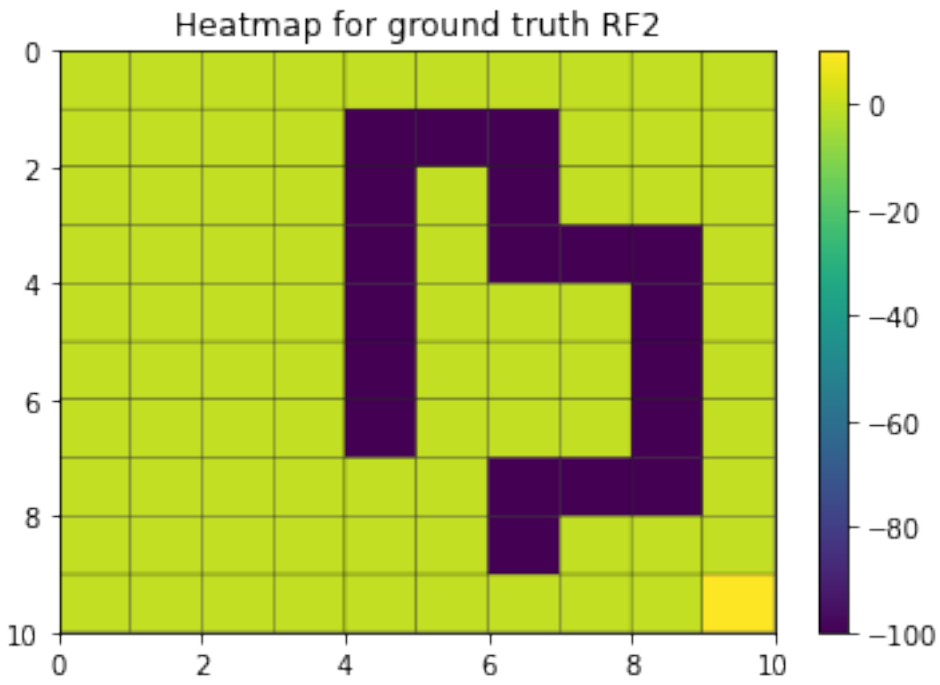
The corresponding $\lambda_{max}^{(2)}$ is 1.12

22 Q20

[ ]:
```
c, D, b = getcDbMatrices(arrows_expert, P_ss, 1, gamma, 1.12, maximum)
solvers.options['show_progress'] = False
sol = solvers.lp(matrix(c), matrix(D), matrix(b))
R = np.array(sol['x'][:100])
R = np.transpose(R.reshape(10, 10))
```

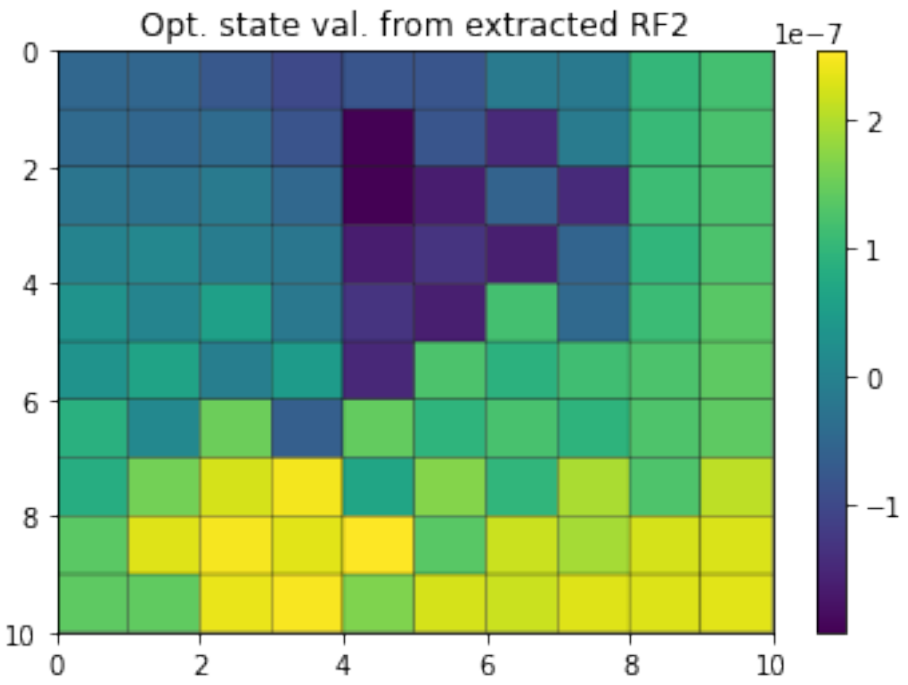
[ ]:
```
plt.pcolor(R, edgecolors='black')
plt.gca().invert_yaxis()
plt.colorbar()
plt.title("Heatmap for extracted reward (RF2)")
plt.show()

plt.pcolor(rwd_2, edgecolors='black')
plt.gca().invert_yaxis()
plt.colorbar()
plt.title("Heatmap for ground truth RF2")
plt.show()
```





23 Q21

[ ]:
```
V_extracted, N = value_iteration(state_num, 0.1, gamma, R, epsilon, tp_r, tp_l,␣
↪tp_u, tp_d)
plt.pcolor(V_extracted, edgecolors='black')
plt.gca().invert_yaxis()
plt.colorbar()
plt.title("Opt. state val. from extracted RF2")
plt.show()
```



24 Q22

Similarity:

The area with negative rewards are approximately similar in the same position in the heatmaps. Besides, the area near state 99 has high optimal values in both the plots. Since state 99 provides the highest reward among all the states. When one moves away from state 99, the values would gradually decrease in both plots.

Difference:

The scaling is much larger for the heatmap in Q7 over Q21. Since the heatmap in Q21 is evaluated from the extracted reward, which in turn is calculated from the expert policy and not expert reward. Thus, the scale of optimal state values would not be the same.

25 Q23

[ ]:
```
policy_agent, pic_arrow_agent, arrows_agent = policy_iteration(state_num, w,␣
↪gamma, R, epsilon, tp_u, tp_d, tp_l, tp_r)

print(pic_arrow_agent)
```

```
[['↑' '←' '↓' '←' '→' '↑' '→' '↑' '→' '↓']
 ['↓' '↓' '↓' '←' '←' '↑' '→' '↑' '→' '↓']
 ['↓' '↓' '↓' '←' '←' '→' '→' '→' '→' '→']
```

```
['↓' '↓' '↓' '←' '←' '↓' '↓' '→' '→' '↓']
['↓' '↓' '↓' '←' '←' '↓' '↓' '↓' '→' '↓']
['↓' '↓' '↓' '←' '↓' '↓' '↓' '←' '→' '↓']
['↓' '↓' '↓' '←' '↓' '↓' '←' '←' '→' '↓']
['←' '→' '↓' '↓' '↓' '↓' '←' '↓' '↓' '↓']
['→' '→' '→' '←' '←' '↓' '↓' '↓' '↓' '↓']
['↓' '↑' '↑' '↑' '←' '→' '↓' '→' '→' '↓']]
```

26 Q24

Similarity:

The agent tries to move away from regions of low-reward and move towards regions of high-reward in both cases.

Difference:

There is more variance and stochasticity in the actions taken in Q23 than Q8. In Q8, the actions are more ordered and homogenous.

In Q8, the agent is guaranteed to reach the optimal state from any state. While in Q16, there is no guarantee. The agent has a higher risk to get stuck in local optima and deadlock condition or be blown off the grid.

A few actions in Q23 point to each other. These are local optima in the policy plots, where the agent just oscillates between two states forever. This is a result of the gradually changing immediate rewards among neighboring states. If the state value between two neighboring states are very similar and non-zero, then the agent will be forced to go back and forth between the two states if exploration probability is low. On the other hand, deadlock condition is absent for the policy map in Q8, as most of the rewards in the state space are zero. It improves the foresight of the agent and passes on the influence of the high reward state to neighboring states much further than possible for the extracted values in Q23. As a result, the values in the state-space is much less noisy for Q8 over Q23, which allows the agent to filter out the correct direction from a further distance.

27 Q25

[ ]:
```
def policy_iteration_Q25(state_num, w, gamma, rwd, epsilon, tp_u, tp_d, tp_l,␣
 ↪tp_r):
    state_val=np.zeros(100)
    delta=np.inf
    policy=np.zeros(100)
    arrows=np.zeros(100)
    r = (rwd.T).ravel()
    while(delta>epsilon):
        delta=0
        old_state_val=np.copy(state_val)
        for s in range(0,100):
            u_val=np.sum(tp_u[s]*(r+gamma*old_state_val))
```

```
            d_val=np.sum(tp_d[s]*(r+gamma*old_state_val))
            l_val=np.sum(tp_l[s]*(r+gamma*old_state_val))
            r_val=np.sum(tp_r[s]*(r+gamma*old_state_val))
            state_val[s]=max(u_val,d_val,r_val,l_val)
            delta=max(delta,abs(old_state_val[s]-state_val[s]))
    for s in range(0,100):
        u_val=np.sum(tp_u[s]*(r+gamma*state_val))
        d_val=np.sum(tp_d[s]*(r+gamma*state_val))
        l_val=np.sum(tp_l[s]*(r+gamma*state_val))
        r_val=np.sum(tp_r[s]*(r+gamma*state_val))

        #Set state values of edge/corners states to -infinity
        if s!=99:
            if s%10==0:
                u_val=-np.inf
            if s%10==9:
                d_val = -np.inf
            if s<=9:
                l_val=-np.inf
            if s>=90:
                r_val=-np.inf

        arr=[u_val,d_val,l_val,r_val] #Up: 0, Down: 1, Left: 2, Right: 3
        policy[s]=np.amax(arr)
        arrows[s]=arr.index(np.amax(arr))
    arrows = np.transpose(arrows.reshape(10,10))
    policy = np.transpose(policy.reshape(10,10))
    pic_arrow=np.chararray((10,10),unicode=True)
    for i in range(10):
        for j in range(10):
            if(arrows[j][i]==0.):
                pic_arrow[j][i] = u'\u2191'
            elif(arrows[j][i]==1.):
                pic_arrow[j][i] = u'\u2193'
            elif(arrows[j][i]==2.):
                pic_arrow[j][i]=u'\u2190'
            elif(arrows[j][i]==3.):
                pic_arrow[j][i] = u'\u2192'
    return policy, pic_arrow, arrows
```

[ ]:
```
w = 0.1
gamma = 0.8
epsilon = pow(10,-10)
tp_u, tp_d, tp_l, tp_r = tp_calc(w)

P_ss = [tp_u, tp_d, tp_l, tp_r]
```

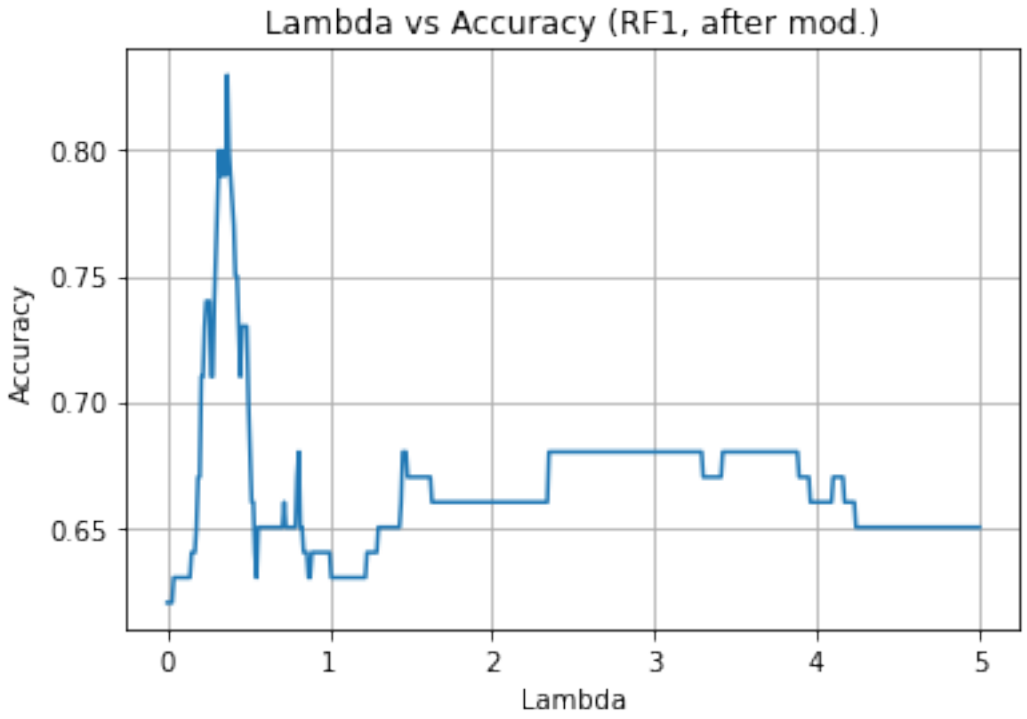
```
#Policies for Reward function 1 (expert)
V_opt_1, N_1 = value_iteration(state_num, w, gamma, rwd_1, epsilon, tp_r, tp_l,␣
 ↪tp_u, tp_d)
policy_1, pic_arrow_1, arrows_1 = policy_iteration(state_num, w, gamma, rwd_1,␣
 ↪epsilon, tp_u, tp_d, tp_l, tp_r)

#Policies for Reward function 2 (expert)
V_opt_2, N_2 = value_iteration(state_num, w, gamma, rwd_2, epsilon, tp_r, tp_l,␣
 ↪tp_u, tp_d)
policy_2, pic_arrow_2, arrows_2 = policy_iteration(state_num, w, gamma, rwd_2,␣
 ↪epsilon, tp_u, tp_d, tp_l, tp_r)

arrows_expert = [np.transpose(arrows_1).flatten(), np.transpose(arrows_2).
 ↪flatten()]
```

[ ]:
```
lambdas = np.arange(0, 5.01, 0.01)
maximum = [1, 10]
acc_list_list = []
I = np.identity(100)
ind = 0
while(ind<2):
    acc_list = []
    for i,lambda_val in enumerate(tqdm(lambdas)):

        #Extract reward function
        c,D,b = getcDbMatrices(arrows_expert, P_ss, ind, gamma, lambda_val,␣
 ↪maximum)
        solvers.options['show_progress']=False
        sol = solvers.lp(matrix(c),matrix(D),matrix(b))
        R = np.array(sol['x'][:100])

        #Extract agent's policy from extracted reward function
        policy_agent, pic_arrow_agent,arrows_agent =␣
 ↪policy_iteration_Q25(state_num, w, gamma, R, epsilon, tp_u, tp_d, tp_l, tp_r)

        #Performance measure
        acc=0
        for j in range(len(np.transpose(arrows_agent).flatten())):
            if(np.transpose(arrows_agent).flatten()[j]==arrows_expert[ind][j]):
                acc = acc+1
        acc = acc/100.0

        acc_list.append(acc)
    acc_list_list.append(acc_list)
    ind = ind+1
```

```
100%|      | 501/501 [02:23<00:00,  3.49it/s]
```

```
100%|      | 501/501 [02:51<00:00,  2.92it/s]
```

[ ]:
```
plt.plot(lambdas,acc_list_list[0])
plt.xlabel("Lambda")
plt.ylabel("Accuracy")
plt.title("Lambda vs Accuracy (RF1, after mod.)")
plt.grid()
plt.show()
```

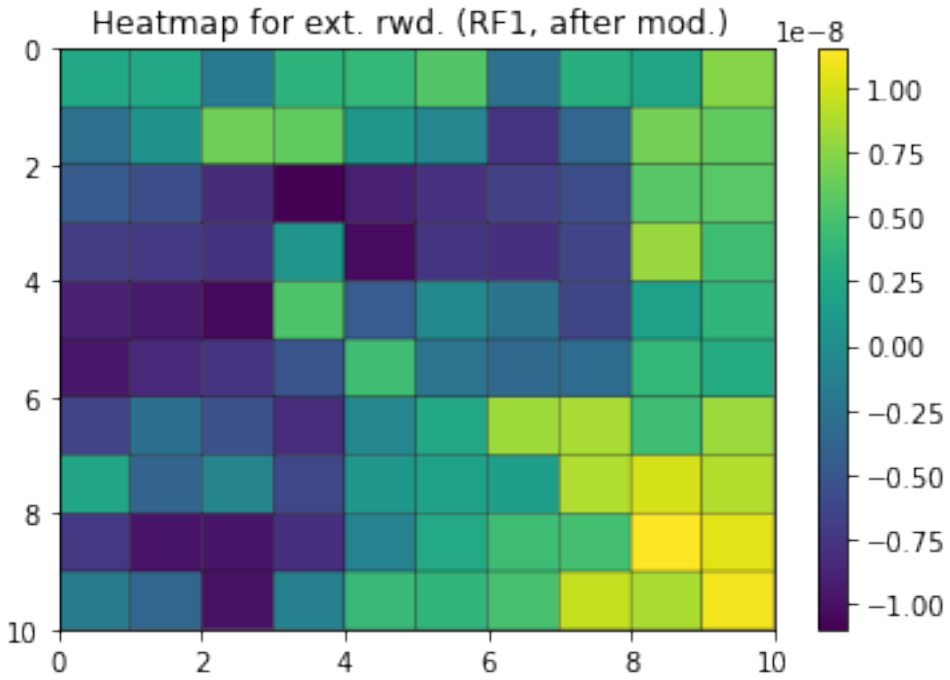


[ ]:
```
print("Max value of accuracy: {}".format(acc_list_list[0][np.
  ↪argmax(acc_list_list[0])]))
print("Corresponding value of lambda: {}".format(lambdas[np.
  ↪argmax(acc_list_list[0])]))
```

```
Max value of accuracy: 0.83
Corresponding value of lambda: 0.37
```

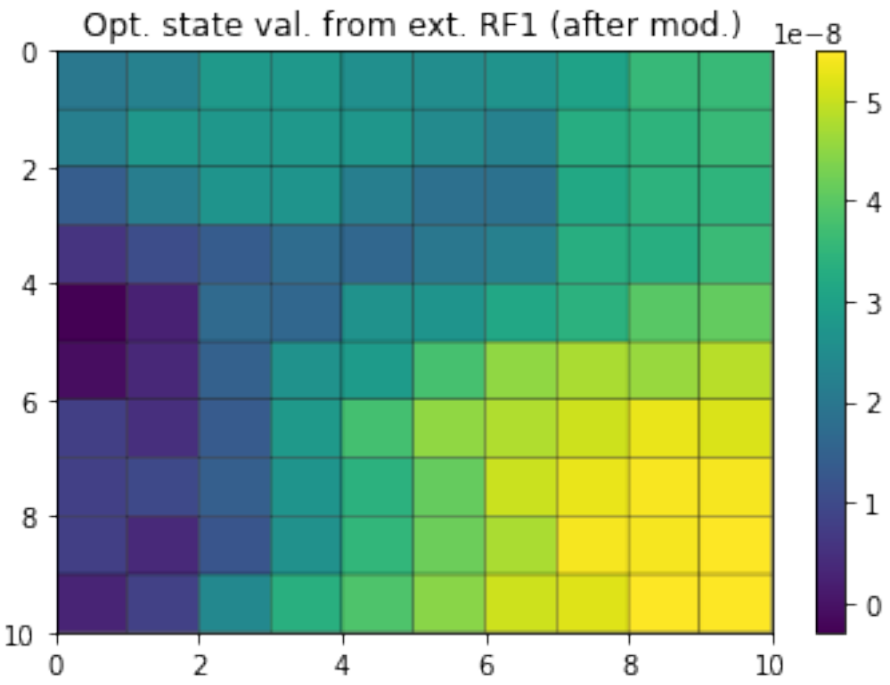
[ ]:
```
c, D, b = getcDbMatrices(arrows_expert, P_ss, 0, gamma, 0.37, maximum)
solvers.options['show_progress'] = False
sol = solvers.lp(matrix(c), matrix(D), matrix(b))
R = np.array(sol['x'][:100])
R = np.transpose(R.reshape(10, 10))
```

[ ]:
```
plt.pcolor(R, edgecolors='black')
plt.gca().invert_yaxis()
plt.colorbar()
plt.title("Heatmap for ext. rwd. (RF1, after mod.) ")
plt.show()
```



[ ]:
```
V_extracted, N = value_iteration(state_num, 0.1, gamma, R, epsilon, tp_r, tp_l,␣
↪tp_u, tp_d)
```

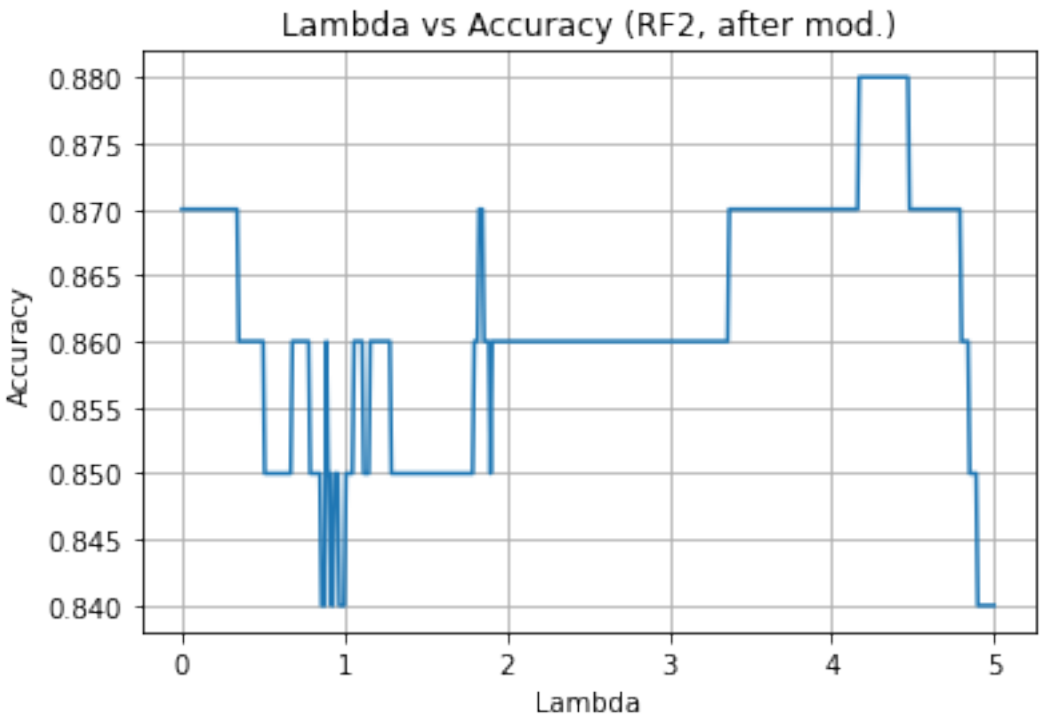
[ ]:
```
plt.pcolor(V_extracted, edgecolors='black')
plt.gca().invert_yaxis()
plt.colorbar()
plt.title("Opt. state val. from ext. RF1 (after mod.)")
plt.show()
```



[ ]:
```
policy_agent, pic_arrow_agent, arrows_agent = policy_iteration_Q25(state_num,␣
↪w, gamma, R, epsilon, tp_u, tp_d, tp_l, tp_r)

print(pic_arrow_agent)
```

```
[['→' '↓' '↓' '↓' '←' '←' '→' '→' '→' '↓']
 ['→' '→' '→' '←' '←' '↑' '→' '→' '→' '↑']
 ['↑' '↑' '↑' '↑' '↑' '↑' '→' '→' '↓' '↑']
 ['↑' '↑' '→' '↓' '↓' '↓' '↓' '→' '↓' '↓']
 ['↑' '→' '→' '→' '↓' '↓' '↓' '↓' '↓' '↓']
 ['↓' '→' '→' '→' '↓' '↓' '↓' '↓' '↓' '↓']
 ['↓' '→' '→' '→' '→' '→' '→' '↓' '↓' '↓']
 ['→' '→' '→' '→' '→' '→' '→' '→' '↓' '↓']
 ['↑' '↑' '→' '→' '→' '→' '→' '→' '→' '↓']
 ['→' '→' '→' '→' '→' '→' '→' '→' '→' '↓']]
```

[ ]:
```
plt.plot(lambdas,acc_list_list[1])
plt.xlabel("Lambda")
plt.ylabel("Accuracy")
plt.title("Lambda vs Accuracy (RF2, after mod.)")
plt.grid()
plt.show()
```

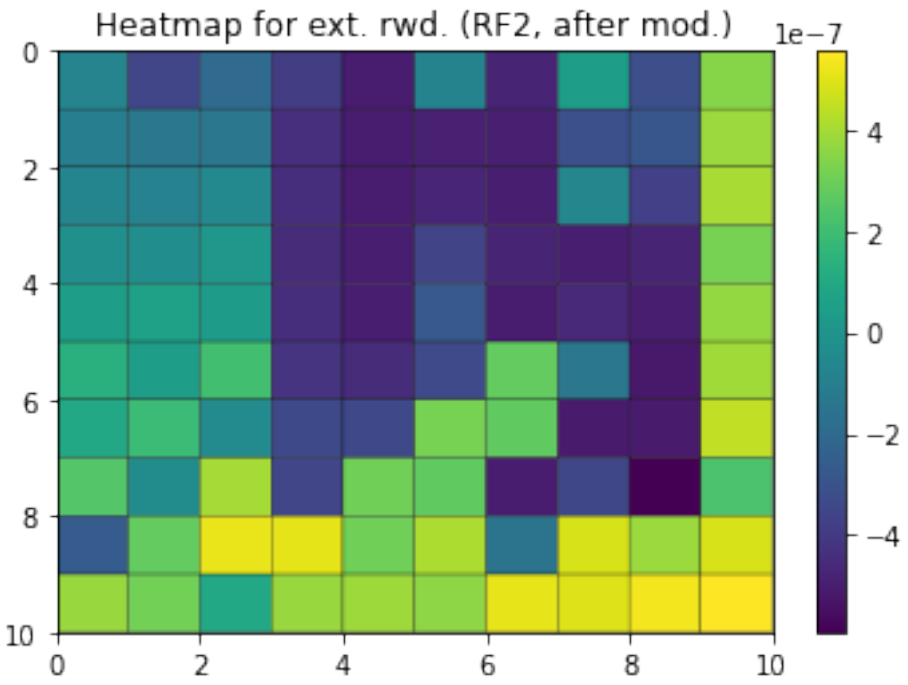


[ ]:
```
print("Max value of accuracy: {}".format(acc_list_list[1][np.
  ↪argmax(acc_list_list[1])]))
print("Corresponding value of lambda: {}".format(lambdas[np.
  ↪argmax(acc_list_list[1])]))
```

```
Max value of accuracy: 0.88
Corresponding value of lambda: 4.17
```

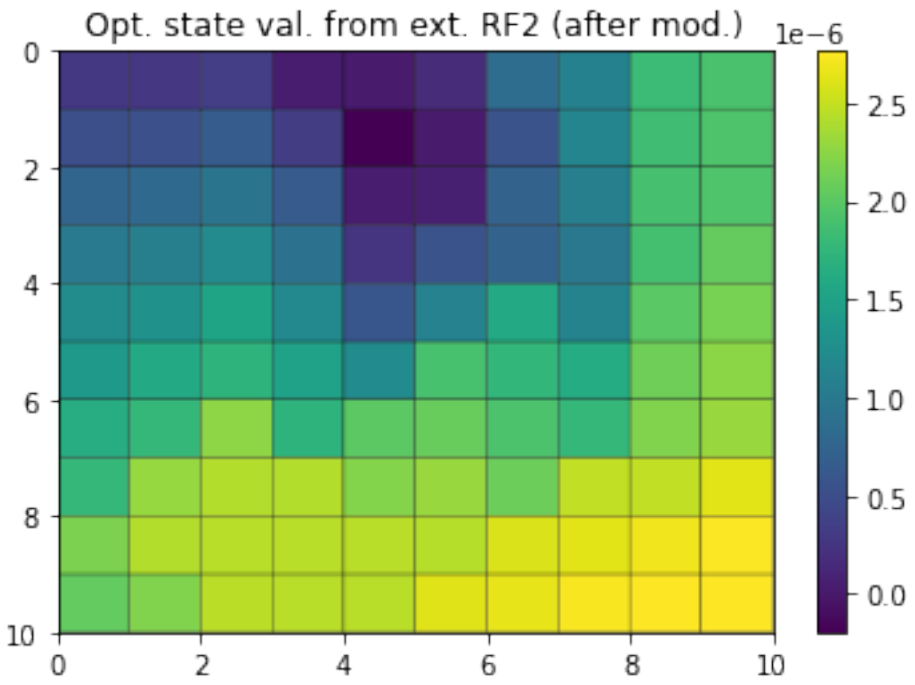
[ ]:
```
c, D, b = getcDbMatrices(arrows_expert, P_ss, 1, gamma, 4.17, maximum)
solvers.options['show_progress'] = False
sol = solvers.lp(matrix(c), matrix(D), matrix(b))
R = np.array(sol['x'][:100])
R = np.transpose(R.reshape(10, 10))
```

[ ]:
```
plt.pcolor(R, edgecolors='black')
plt.gca().invert_yaxis()
plt.colorbar()
plt.title("Heatmap for ext. rwd. (RF2, after mod.)")
plt.show()
```



[ ]:
```
V_extracted, N = value_iteration(state_num, 0.1, gamma, R, epsilon, tp_r, tp_l,␣
↪tp_u, tp_d)
```

[ ]:
```
plt.pcolor(V_extracted, edgecolors='black')
plt.gca().invert_yaxis()
plt.colorbar()
plt.title("Opt. state val. from ext. RF2 (after mod.)")
plt.show()
```



[ ]:
```
policy_agent, pic_arrow_agent, arrows_agent = policy_iteration_Q25(state_num,␣
 ↪w, gamma, R, epsilon, tp_u, tp_d, tp_l, tp_r)

print(pic_arrow_agent)
```

```
[['↓' '↓' '↓' '←' '→' '→' '→' '→' '→' '↓']
 ['↓' '↓' '↓' '←' '←' '↑' '→' '→' '→' '↓']
 ['↓' '↓' '↓' '←' '←' '↓' '→' '→' '→' '↓']
 ['↓' '↓' '↓' '←' '←' '↓' '↓' '→' '→' '↓']
 ['↓' '↓' '↓' '←' '→' '↓' '↓' '↓' '→' '↓']
 ['↓' '↓' '↓' '←' '↓' '↓' '↓' '←' '→' '↓']
 ['↓' '↓' '↓' '←' '↓' '↓' '←' '←' '→' '↓']
 ['→' '→' '↓' '↓' '↓' '↓' '←' '↓' '↓' '↓']
 ['→' '→' '→' '←' '←' '↓' '↓' '↓' '↓' '↓']
 ['→' '↑' '↑' '↑' '→' '→' '→' '→' '→' '↓']]
```

Two major discrepancies:

Moving off the grid: Some of the actions in Q23 will lead the agent to be blown off the grid. This is because there is a high probability for the agent to not move towards the optimal regions in case of the state-space in Q21, as immediate and local rewards can quickly accumulate in the wrong direction and overpower the maximum achievable reward. In other words, the agent will accumulate a lower expected reward in the long run for the state-space in Q21 over the agent in Q7 ultimately causing the agent to be more likely to move off the grid or take suboptimal actions in the long run.

Deadlock Condition: A few actions point to each other in Q23. These are local optimum in the policy plot, where the agent just oscillates between two states forever. This is a result of the gradually changing immediate rewards among neighboring states. If the state value between two neighboring states are very similar and non-zero, then the agent will be forced to go back and forth between the two states if exploration probability is low.

The maximum accuracy increases from 83% to 88% after the modification.