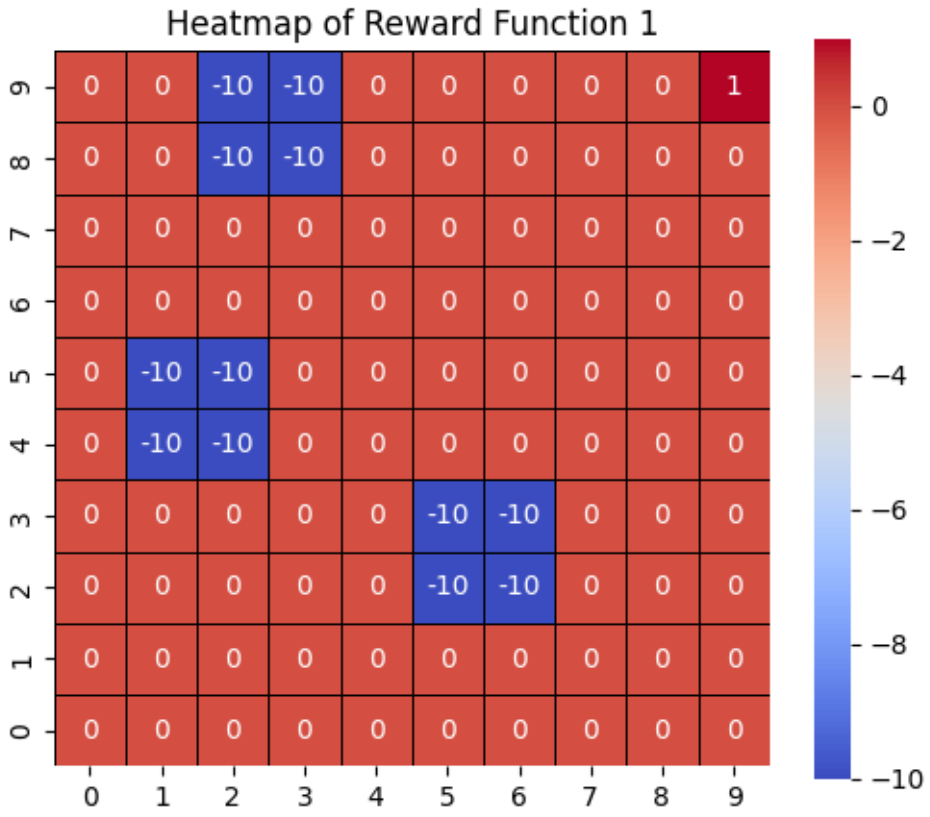
ECE232E_Project3_606539982_106405230_106303745

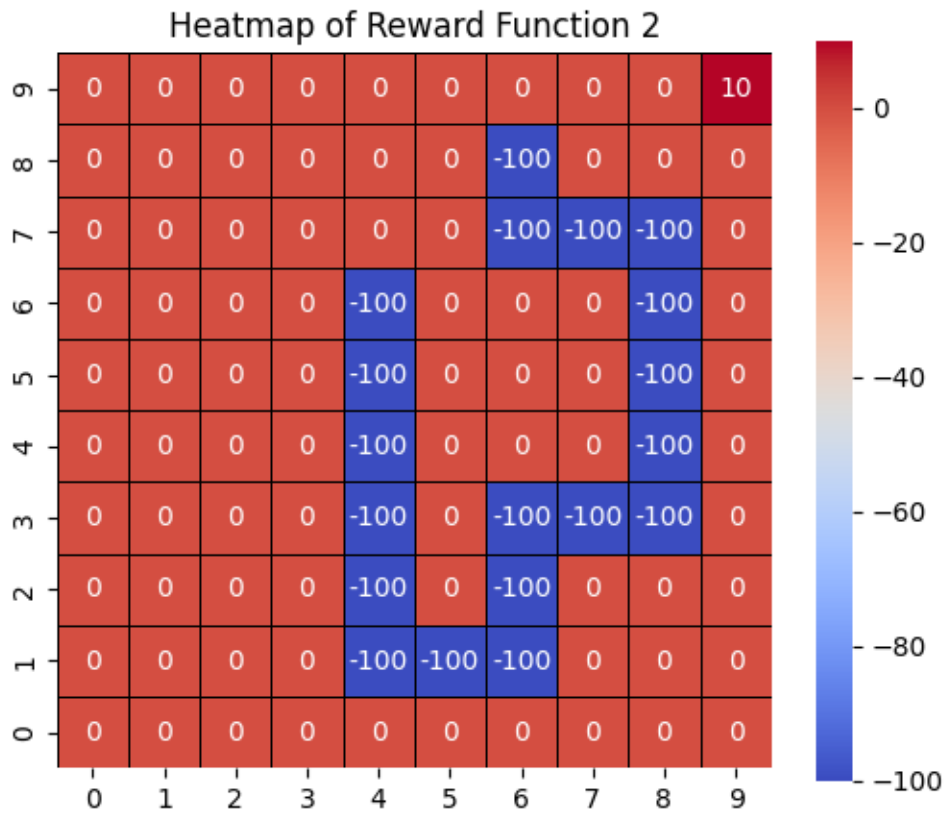
May 20, 2025

1 Question 1

[9]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib

# Reward Function 1 (from Figure 6)
reward1 = np.zeros((10, 10))
reward1[2, 5] = -10
reward1[2, 6] = -10
reward1[3, 5] = -10
reward1[3, 6] = -10
reward1[4, 1] = -10
reward1[4, 2] = -10
reward1[5, 1] = -10
reward1[5, 2] = -10
reward1[8, 2] = -10
reward1[8, 3] = -10
reward1[9, 2] = -10
reward1[9, 3] = -10
reward1[9, 9] = 1

# Reward Function 2 (from Figure 7)
reward2 = np.zeros((10, 10))
coords2 = [
    (1, 4), (1, 5), (1, 6), (2, 4), (2, 6), (3, 4), (3, 6), (3, 7), (3, 8), (4,␣
 ↪4), (4, 8), (5, 4), (5, 8), (6, 4), (6, 8), (7, 6), (7, 7), (7, 8), (8, 6)
    ]
for r, c in coords2:
    reward2[r, c] = -100
reward2[9, 9] = 10

# Plotting function
def plot_heatmap(data, title):
    plt.figure(figsize=(6, 5))
    ax = sns.heatmap(data, annot=True, cmap='coolwarm', square=True, cbar=True,
```

```
                      linewidths=0.5, linecolor='black', fmt=".0f")
    plt.title(title)
    plt.gca().invert_yaxis()
    plt.show()

# Generate the heatmaps
plot_heatmap(reward1, "Heatmap of Reward Function 1")
plot_heatmap(reward2, "Heatmap of Reward Function 2")
```





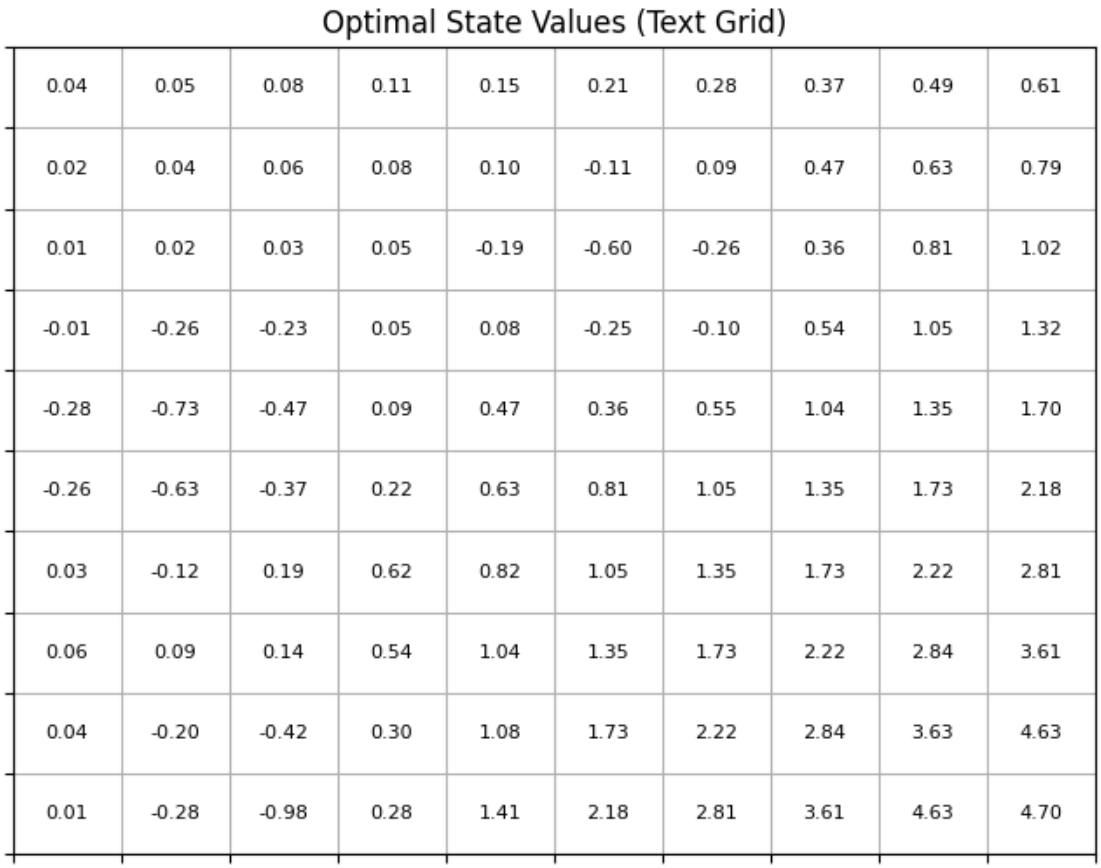
2 Question 2

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

# MDP Parameters
num_states = 100
grid_size = 10
num_actions = 4
w = 0.1
gamma = 0.8
epsilon = 0.01

# Actions: 0=Right, 1=Left, 2=Up, 3=Down
actions = [0, 1, 2, 3]

# Reward Function 1
reward_function = np.zeros(num_states)
reward_function[[25, 26, 35, 36, 41, 42, 51, 52, 82, 83, 92, 93]] = -10
reward_function[99] = 1
```

```
# Transition Probability Matrix: P[s, s', a]
P = np.zeros((num_states, num_states, num_actions))

def compute_transition_probabilities():
    for s in range(num_states):
        row, col = divmod(s, grid_size)
        for a in actions:
            intended_state = s
            if a == 0 and col < grid_size - 1:
                intended_state = s + 1
            elif a == 1 and col > 0:
                intended_state = s - 1
            elif a == 2 and row > 0:
                intended_state = s - grid_size
            elif a == 3 and row < grid_size - 1:
                intended_state = s + grid_size

            # Apply wind in all 4 directions
            for move, offset in zip(actions, [(0, 1), (0, -1), (-1, 0), (1,␣
 ↪0)]):
                nx, ny = row + offset[0], col + offset[1]
                if 0 <= nx < grid_size and 0 <= ny < grid_size:
                    sp = nx * grid_size + ny
                    P[s, sp, a] += w / 4
                else:
                    P[s, s, a] += w / 4

            if intended_state == s:
                P[s, s, a] += (1 - w)
            else:
                P[s, intended_state, a] += (1 - w)

compute_transition_probabilities()

# Value Iteration Algorithm
def value_iteration():
    V = np.zeros(num_states)
    delta = float('inf')
    iteration = 0
    snapshots = []
    # Linearly distributed steps from 1 to N (we'll determine N first, then␣
 ↪select steps)
    all_values = []  # Store V at each iteration to select snapshots later

    while delta > epsilon and iteration < 1000:
        delta = 0
```

```
        V_new = np.zeros(num_states)
        for s in range(num_states):
            q_values = []
            for a in actions:
                q = np.sum(P[s, :, a] * (reward_function + gamma * V))
                q_values.append(q)
            V_new[s] = max(q_values)
            delta = max(delta, abs(V[s] - V_new[s]))
        V = V_new.copy()
        iteration += 1
        all_values.append(V.copy())

    N = iteration
    # Select 5 linearly distributed steps from 1 to N
    step_indices = np.linspace(0, N-1, 5, dtype=int)  # 0-based indices
    snapshot_iterations = [i + 1 for i in step_indices]  # Convert to 1-based␣
 ↪iteration numbers
    snapshots = [all_values[i] for i in step_indices]

    return V, N, snapshots, snapshot_iterations

# Run Value Iteration
optimal_V, N, snapshots, snapshot_steps = value_iteration()

# Plot Text-Grid Only (no heatmap)
def plot_state_values_text(V, title):
    V_grid = V.reshape(grid_size, grid_size)
    fig, ax = plt.subplots(figsize=(8, 6))
    for i in range(grid_size):
        for j in range(grid_size):
            ax.text(j + 0.5, grid_size - 1 - i + 0.5, f'{V_grid[i, j]:.2f}',
                    ha='center', va='center', fontsize=8)
    ax.set_xticks(np.arange(grid_size + 1))
    ax.set_yticks(np.arange(grid_size + 1))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.grid(True)
    ax.set_xlim(0, grid_size)
    ax.set_ylim(0, grid_size)
    plt.title(title)
    plt.show()

# Final optimal state value plot
plot_state_values_text(optimal_V, 'Optimal State Values (Text Grid)')

# Snapshot plots at linearly distributed steps
for i, step in enumerate(snapshot_steps):
```

```
    plot_state_values_text(snapshots[i], f'State Values at Iteration {step}')

# Report
print(f"\nValue Iteration Converged in N = {N} iterations")
print(f"Snapshots taken at iterations: {snapshot_steps}")
```



State Values at Iteration 1

0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	-0.25	-0.25	0.00	0.00	0.00
0.00	0.00	0.00	0.00	-0.25	-0.50	-0.50	-0.25	0.00	0.00
0.00	-0.25	-0.25	0.00	-0.25	-0.50	-0.50	-0.25	0.00	0.00
-0.25	-0.50	-0.50	-0.25	0.00	-0.25	-0.25	0.00	0.00	0.00
-0.25	-0.50	-0.50	-0.25	0.00	0.00	0.00	0.00	0.00	0.00
0.00	-0.25	-0.25	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	-0.25	-0.25	0.00	0.00	0.00	0.00	0.00	0.00
0.00	-0.25	-0.50	-0.50	-0.25	0.00	0.00	0.00	0.00	0.93
0.00	-0.25	-0.75	-0.75	-0.25	0.00	0.00	0.00	0.93	0.95

State Values at Iteration 6

-0.00	-0.00	-0.00	-0.00	-0.00	-0.01	-0.01	-0.00	-0.00	-0.00
-0.00	-0.00	-0.00	-0.00	-0.01	-0.27	-0.27	-0.01	-0.00	-0.00
-0.00	-0.01	-0.01	-0.01	-0.28	-0.74	-0.74	-0.27	-0.01	-0.00
-0.01	-0.27	-0.27	-0.02	-0.29	-0.74	-0.74	-0.27	-0.01	0.21
-0.28	-0.74	-0.74	-0.29	-0.02	-0.28	-0.27	-0.01	0.23	0.52
-0.29	-0.74	-0.74	-0.28	-0.01	-0.01	-0.01	0.23	0.55	0.99
-0.01	-0.28	-0.29	-0.02	-0.00	-0.00	0.23	0.55	1.02	1.60
-0.00	-0.02	-0.29	-0.28	-0.01	0.23	0.55	1.02	1.64	2.40
-0.01	-0.28	-0.74	-0.74	-0.04	0.54	1.02	1.64	2.42	3.43
-0.01	-0.29	-1.02	-0.81	0.23	0.98	1.60	2.40	3.43	3.50

State Values at Iteration 11

-0.00	-0.00	-0.00	-0.00	-0.00	-0.01	-0.01	0.05	0.13	0.25
-0.00	-0.00	-0.00	-0.00	-0.01	-0.27	-0.22	0.12	0.27	0.42
-0.00	-0.01	-0.01	-0.01	-0.28	-0.74	-0.60	-0.00	0.44	0.65
-0.01	-0.27	-0.28	-0.02	-0.23	-0.60	-0.46	0.17	0.68	0.94
-0.28	-0.74	-0.74	-0.23	0.12	0.00	0.18	0.67	0.98	1.32
-0.29	-0.74	-0.68	-0.14	0.27	0.45	0.68	0.98	1.36	1.81
-0.01	-0.28	-0.15	0.26	0.45	0.68	0.98	1.36	1.85	2.44
-0.00	-0.02	-0.21	0.17	0.67	0.98	1.36	1.85	2.47	3.24
-0.01	-0.28	-0.73	-0.07	0.71	1.36	1.85	2.47	3.26	4.26
-0.01	-0.29	-1.00	-0.08	1.04	1.80	2.44	3.24	4.26	4.33

State Values at Iteration 16

-0.00	-0.00	0.01	0.03	0.06	0.11	0.18	0.28	0.39	0.51
-0.00	-0.00	-0.00	0.01	0.02	-0.20	-0.01	0.37	0.53	0.69
-0.00	-0.01	-0.01	-0.01	-0.26	-0.69	-0.35	0.26	0.71	0.92
-0.01	-0.27	-0.27	-0.01	-0.01	-0.35	-0.20	0.45	0.95	1.22
-0.28	-0.74	-0.56	-0.01	0.37	0.26	0.45	0.94	1.25	1.60
-0.29	-0.71	-0.46	0.12	0.53	0.72	0.95	1.25	1.64	2.08
-0.01	-0.22	0.10	0.52	0.72	0.96	1.26	1.64	2.12	2.71
-0.00	-0.01	0.04	0.44	0.94	1.25	1.64	2.12	2.74	3.51
-0.01	-0.27	-0.52	0.20	0.98	1.63	2.12	2.74	3.53	4.54
-0.01	-0.29	-0.99	0.18	1.31	2.08	2.71	3.51	4.54	4.60

State Values at Iteration 22

0.04	0.05	0.08	0.11	0.15	0.21	0.28	0.37	0.49	0.61
0.02	0.04	0.06	0.08	0.10	-0.11	0.09	0.47	0.63	0.79
0.01	0.02	0.03	0.05	-0.19	-0.60	-0.26	0.36	0.81	1.02
-0.01	-0.26	-0.23	0.05	0.08	-0.25	-0.10	0.54	1.05	1.32
-0.28	-0.73	-0.47	0.09	0.47	0.36	0.55	1.04	1.35	1.70
-0.26	-0.63	-0.37	0.22	0.63	0.81	1.05	1.35	1.73	2.18
0.03	-0.12	0.19	0.62	0.82	1.05	1.35	1.73	2.22	2.81
0.06	0.09	0.14	0.54	1.04	1.35	1.73	2.22	2.84	3.61
0.04	-0.20	-0.42	0.30	1.08	1.73	2.22	2.84	3.63	4.63
0.01	-0.28	-0.98	0.28	1.41	2.18	2.81	3.61	4.63	4.70

```
Value Iteration Converged in N = 22 iterations
Snapshots taken at iterations: [np.int64(1), np.int64(6), np.int64(11),
np.int64(16), np.int64(22)]
```

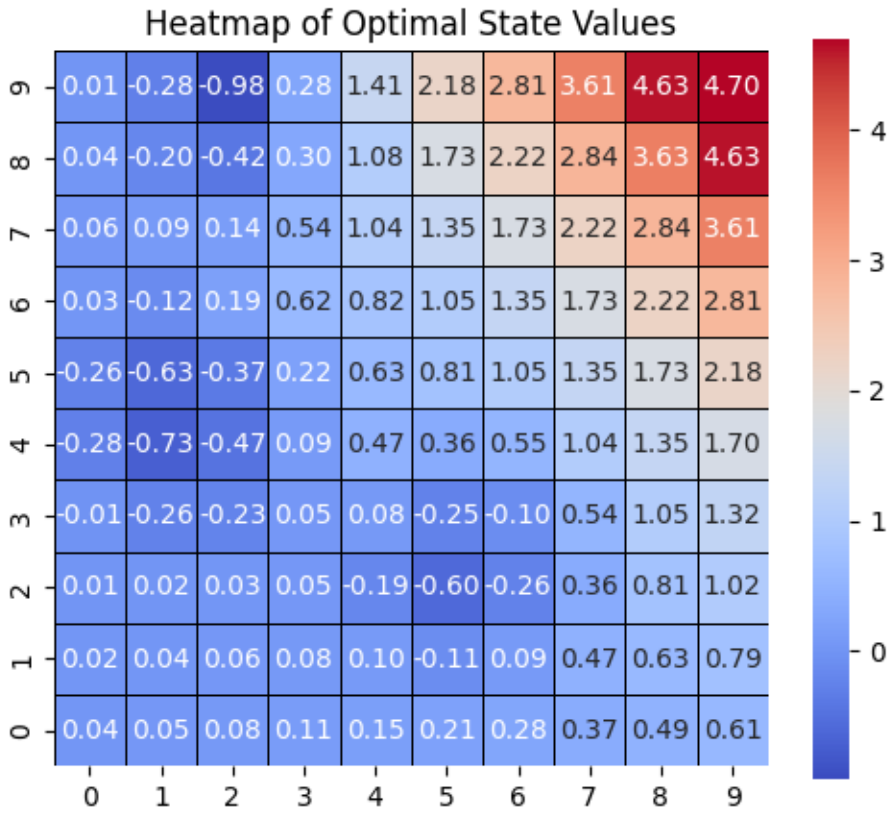
The Value Iteration algorithm converges in N=22 iterations, and snapshots of state values are plotted at 5 linearly distributed steps: iterations 1, 6, 11, 16, and 22. Initially, at iteration 1, the state values reflect immediate rewards, with penalty states like state 25 at -0.50 and the goal state 99 at 0.95, while distant states remain near zero (e.g., state 0 at 0.00). By iteration 6, the penalty states deepen to -0.74 at state 25, and the goal state rises significantly to 3.50, with the reward influence spreading across the grid. At iteration 11, penalty states is still -0.74 at state 25, while state 99 reaches 4.33. 41, showing a stronger gradient towards the goal. By iteration 16, an unexpected trend emerges as penalty states become less negative (e.g., state 25 at -0.69), possibly due to the long-term influence of the goal state at 4.60. Finally, at iteration 22, this trend continues with state 25 at -0.60 and state 99 at 4.70, indicating that the agent's focus on reaching the goal may outweigh immediate penalties over time, revealing a nuanced convergence pattern influenced by the grid dynamics and discount factor.

3 Question 3

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

# Assuming optimal_V is available from Question 2
# For context, let's include the necessary parts from Question 2 to ensure␣
 ↪optimal_V is computed
# MDP Parameters
num_states = 100
grid_size = 10
num_actions = 4
w = 0.1
gamma = 0.8
epsilon = 0.01

# Actions: 0=Right, 1=Left, 2=Up, 3=Down
actions = [0, 1, 2, 3]

# Reward Function 1
reward_function = np.zeros(num_states)
reward_function[[25, 26, 35, 36, 41, 42, 51, 52, 82, 83, 92, 93]] = -10
reward_function[99] = 1

# Transition Probability Matrix: P[s, s', a]
P = np.zeros((num_states, num_states, num_actions))

def compute_transition_probabilities():
    for s in range(num_states):
        row, col = divmod(s, grid_size)
        for a in actions:
            intended_state = s
            if a == 0 and col < grid_size - 1:
                intended_state = s + 1
            elif a == 1 and col > 0:
                intended_state = s - 1
            elif a == 2 and row > 0:
                intended_state = s - grid_size
            elif a == 3 and row < grid_size - 1:
                intended_state = s + grid_size

            # Apply wind in all 4 directions
            for move, offset in zip(actions, [(0, 1), (0, -1), (-1, 0), (1,␣
 ↪0)]):
                nx, ny = row + offset[0], col + offset[1]
                if 0 <= nx < grid_size and 0 <= ny < grid_size:
```

```
                    sp = nx * grid_size + ny
                    P[s, sp, a] += w / 4
                else:
                    P[s, s, a] += w / 4

            if intended_state == s:
                P[s, s, a] += (1 - w)
            else:
                P[s, intended_state, a] += (1 - w)

compute_transition_probabilities()

# Value Iteration Algorithm (from Question 2)
def value_iteration():
    V = np.zeros(num_states)
    delta = float('inf')
    iteration = 0

    while delta > epsilon and iteration < 1000:
        delta = 0
        V_new = np.zeros(num_states)
        for s in range(num_states):
            q_values = []
            for a in actions:
                q = np.sum(P[s, :, a] * (reward_function + gamma * V))
                q_values.append(q)
            V_new[s] = max(q_values)
            delta = max(delta, abs(V[s] - V_new[s]))
        V = V_new.copy()
        iteration += 1

    return V

# Compute optimal state values
optimal_V = value_iteration()

# Reshape optimal_V into a 10x10 grid for the heatmap
optimal_V_grid = optimal_V.reshape(grid_size, grid_size)

# Plotting function (adapted from Question 1)
def plot_heatmap(data, title):
    plt.figure(figsize=(6, 5))
    ax = sns.heatmap(data, annot=True, cmap='coolwarm', square=True, cbar=True,
                     linewidths=0.5, linecolor='black', fmt=".2f")
    plt.title(title)
    plt.gca().invert_yaxis()
    plt.show()
```

```
# Generate the heatmap for optimal state values
plot_heatmap(optimal_V_grid, "Heatmap of Optimal State Values")
```



4 Question 4

Explanation of the Distribution of Optimal State Values

The heat map of the optimal state values across the 2D grid reveals a distinct distribution that reflects the agent's learned strategy for navigating the environment, balancing the immediate penalties and the long-term goal. The grid shows a clear gradient, with the lowest values concentrated around the penalty states (e.g., states 25, 26, 35, 36, 41, 42, 51, 52, 82, 83, 92, 93), where the reward is -10, such as state 25 at -0.60, depicted in deep blue shades indicating strong avoidance. In contrast, the highest value is at the goal state 99 (position (9, 9)), with V(99)=4.70, shown in a bright red shade, highlighting the agent's preference for reaching this state due to its positive reward of 1. Nearby states, such as state 89 and state 98 (4.63), also exhibit high positive values in red hues, reflecting the propagating influence of the goal state's reward. A smooth gradient emerges across the grid, with values increasing as states get closer to state 99 (e.g., state 0 in the top-left corner at 0.04, in a light blue shade, transitions to higher values like 3.61 at state 79 in a pink shade as we move towards the bottom-right). This gradient indicates that the agent has learned to navigate towards the bottom-right corner while avoiding the penalty clusters in the middle (e.g., states 41,

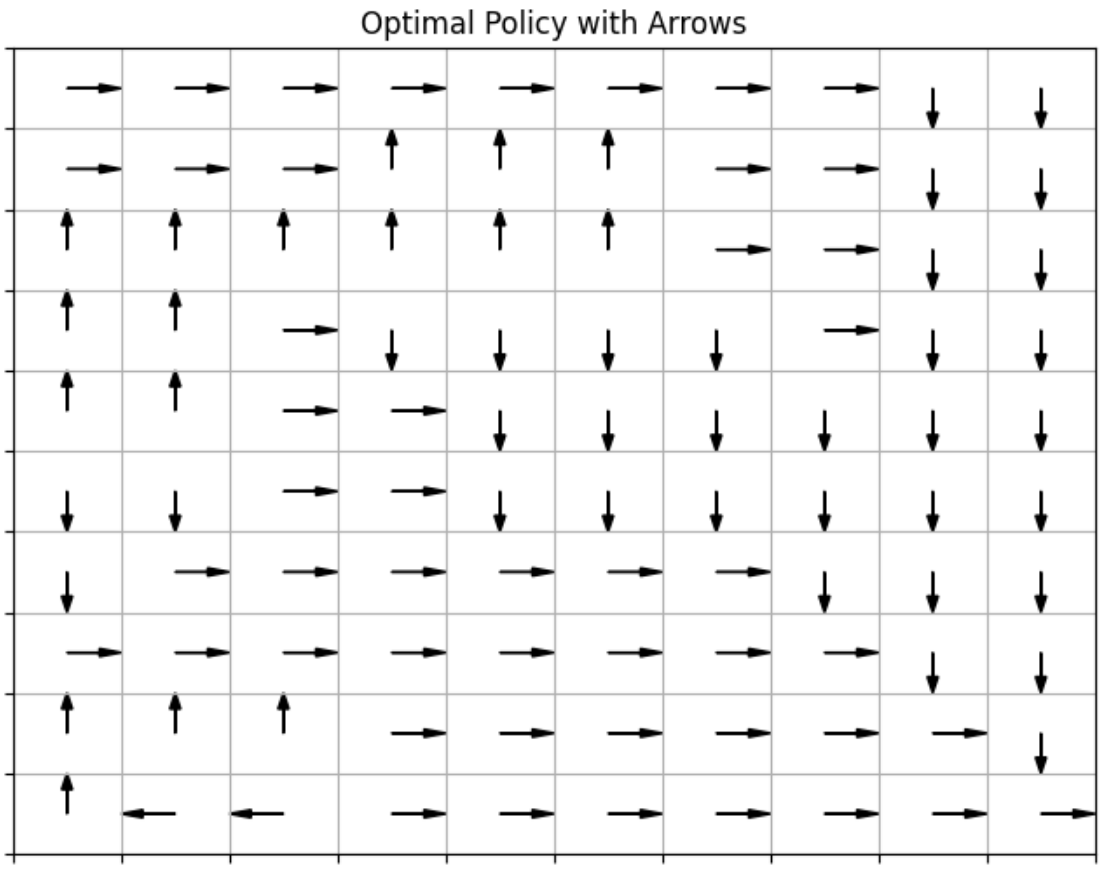
42, 51, 52) and bottom-right (e.g., states 82, 83, 92, 93), which form "valleys" of negative values. Interestingly, the penalty states' values vary in negativity, with some moderately low (e.g., -0.60 at state 25) and others more strongly negative (e.g., -0.98 at 92). This suggests that the influence of the long-term expected rewards from reaching state 99, shaped by the discount factor =0.8 and wind factor w=0.1, partially offsets immediate penalties in some areas while others remain highly avoided.

5 Question 5

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

# MDP Parameters (from previous questions)
num_states = 100
grid_size = 10
num_actions = 4
w = 0.1
gamma = 0.8
epsilon = 0.01

# Actions: 0=Right, 1=Left, 2=Up, 3=Down
actions = [0, 1, 2, 3]
action_arrows = {0: (1, 0), 1: (-1, 0), 2: (0, 1), 3: (0, -1)}  # (dx, dy) for␣
 ↪arrows
action_labels = {0: '→', 1: '←', 2: '↑', 3: '↓'}

# Reward Function 1
reward_function_1 = np.zeros(num_states)
reward_function_1[[25, 26, 35, 36, 41, 42, 51, 52, 82, 83, 92, 93]] = -10
reward_function_1[99] = 1

# Transition Probability Matrix: P[s, s', a]
P = np.zeros((num_states, num_states, num_actions))

def compute_transition_probabilities():
    for s in range(num_states):
        row, col = divmod(s, grid_size)
        for a in actions:
            intended_state = s
            if a == 0 and col < grid_size - 1:
                intended_state = s + 1
            elif a == 1 and col > 0:
                intended_state = s - 1
            elif a == 2 and row > 0:
                intended_state = s - grid_size
            elif a == 3 and row < grid_size - 1:
                intended_state = s + grid_size
```

```
            # Apply wind in all 4 directions
            for move, offset in zip(actions, [(0, 1), (0, -1), (-1, 0), (1,␣
 ↪0)]):
                nx, ny = row + offset[0], col + offset[1]
                if 0 <= nx < grid_size and 0 <= ny < grid_size:
                    sp = nx * grid_size + ny
                    P[s, sp, a] += w / 4
                else:
                    P[s, s, a] += w / 4

            if intended_state == s:
                P[s, s, a] += (1 - w)
            else:
                P[s, intended_state, a] += (1 - w)

compute_transition_probabilities()

# Value Iteration Algorithm (from Question 2 to get optimal_V)
def value_iteration():
    V = np.zeros(num_states)
    delta = float('inf')
    iteration = 0

    while delta > epsilon and iteration < 1000:
        delta = 0
        V_new = np.zeros(num_states)
        for s in range(num_states):
            q_values = []
            for a in actions:
                q = np.sum(P[s, :, a] * (reward_function_1 + gamma * V))
                q_values.append(q)
            V_new[s] = max(q_values)
            delta = max(delta, abs(V[s] - V_new[s]))
        V = V_new.copy()
        iteration += 1

    return V

# Compute optimal state values
optimal_V = value_iteration()

# Compute the optimal policy (lines 14-17 of the Value Iteration algorithm)
def compute_optimal_policy(V):
    policy = np.zeros(num_states, dtype=int)
    for s in range(num_states):
        q_values = []
```

```
        for a in actions:
            q = np.sum(P[s, :, a] * (reward_function_1 + gamma * V))
            q_values.append(q)
        policy[s] = np.argmax(q_values)
    return policy

# Compute the optimal policy
optimal_policy_reward_1 = compute_optimal_policy(optimal_V)

# Visualize the optimal policy with arrows
def plot_policy(policy, title):
    fig, ax = plt.subplots(figsize=(8, 6))
    for i in range(grid_size):
        for j in range(grid_size):
            state = i * grid_size + j
            action = policy[state]
            dx, dy = action_arrows[action]
            # Plot an arrow at the center of each cell
            ax.arrow(j + 0.5, grid_size - 1 - i + 0.5, dx * 0.3, dy * 0.3,
                     head_width=0.1, head_length=0.2, fc='black', ec='black')
    ax.set_xticks(np.arange(grid_size + 1))
    ax.set_yticks(np.arange(grid_size + 1))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.grid(True)
    ax.set_xlim(0, grid_size)
    ax.set_ylim(0, grid_size)
    plt.title(title)
    plt.show()

# Plot the optimal policy
plot_policy(optimal_policy_reward_1, "Optimal Policy with Arrows")

# Print some policy values for verification
print(f"Policy at state 25 (should move away from penalty):␣
 ↪{action_labels[optimal_policy_reward_1[25]]}")
print(f"Policy at state 99 (goal state, should be arbitrary):␣
 ↪{action_labels[optimal_policy_reward_1[99]]}")
print(f"Policy at state 0 (should move towards goal):␣
 ↪{action_labels[optimal_policy_reward_1[0]]}")
```




```
Policy at state 25 (should move away from penalty): ↑
Policy at state 99 (goal state, should be arbitrary): →
Policy at state 0 (should move towards goal): →
```

Does the Optimal Policy Match Your Intuition? Please Provide a Brief Explanation.

The optimal policy matches intuition well. The agent consistently moves towards the goal state 99 (bottom-right), as seen in the arrows pointing → across the top row, such as at state 0 (→) and state 1 (→), directing the agent rightward towards the goal's column. Near penalty clusters, the policy generally directs the agent to avoid these regions, with arrows like state 25 (↑) pointing away from the penalty cluster (states 26, 35, 36), steering clear of the -10 rewards. At state 26 (→), the arrow points towards state 27, which is not a penalty state V(27)= 0.36), suggesting the agent is moving right to align with the goal's column (column 9), a reasonable strategy to bypass the penalty cluster and continue towards state 99, though moving up might have been more optimal based on state values. Close to state 99, arrows from states like 89 (↓) and 98 (→) point directly towards the goal, reflecting the high optimal values in that region (V(99)=4.70). The wind factor w=0.1 introduces stochasticity, but the policy largely guides the agent along paths that aim to maximize long-term rewards while avoiding penalties, aligning with the expected behavior in an MDP with a clear goal and obstacles.

Is It Possible for the Agent to Compute the Optimal Action to Take at Each State by

Observing the Optimal Values of Its Neighboring States?

Yes, the agent can figure out the best action at each state by looking at the optimal values of its neighboring states, but only if it also knows the rewards, transition probabilities, and discount factor. The best action at a state is chosen by picking the action that gives the highest expected reward, which depends on the values of the neighboring states (like for state 25, it looks at states 26, 24, 15, and 35). However, to calculate this, the agent needs to know the rewards for moving to those states, the chances of moving to each neighbor, and how much future rewards are discounted. If the agent has this information, it can use the neighbors' values to decide the best action. Without it, just knowing the neighbors' values isn't enough, because the rewards and movement probabilities are key to making the right choice.

6 Question 6

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

# MDP Parameters (from Question 2)
num_states = 100
grid_size = 10
num_actions = 4
w = 0.1
gamma = 0.8
epsilon = 0.01

# Actions: 0=Right, 1=Left, 2=Up, 3=Down
actions = [0, 1, 2, 3]

# Reward Function 2 (based on the coordinates provided in Question 1 code)
reward_function = np.zeros(num_states)
coords = [
    (1, 4), (1, 5), (1, 6), (2, 4), (2, 6), (3, 4), (3, 6), (3, 7), (3, 8),
    (4, 4), (4, 8), (5, 4), (5, 8), (6, 4), (6, 8), (7, 6), (7, 7), (7, 8), (8,␣
 ↪6)
]
for r, c in coords:
    state = r * grid_size + c
    reward_function[state] = -100
reward_function[99] = 10  # State 99 corresponds to (9,9)

# Transition Probability Matrix: P[s, s', a] (from Question 2)
P = np.zeros((num_states, num_states, num_actions))

def compute_transition_probabilities():
    for s in range(num_states):
        row, col = divmod(s, grid_size)
        for a in actions:
```

```
            intended_state = s
            if a == 0 and col < grid_size - 1:
                intended_state = s + 1
            elif a == 1 and col > 0:
                intended_state = s - 1
            elif a == 2 and row > 0:
                intended_state = s - grid_size
            elif a == 3 and row < grid_size - 1:
                intended_state = s + grid_size

            # Apply wind in all 4 directions
            for move, offset in zip(actions, [(0, 1), (0, -1), (-1, 0), (1,␣
 ↪0)]):
                nx, ny = row + offset[0], col + offset[1]
                if 0 <= nx < grid_size and 0 <= ny < grid_size:
                    sp = nx * grid_size + ny
                    P[s, sp, a] += w / 4
                else:
                    P[s, s, a] += w / 4

            if intended_state == s:
                P[s, s, a] += (1 - w)
            else:
                P[s, intended_state, a] += (1 - w)

compute_transition_probabilities()

# Value Iteration Algorithm (from Question 2)
def value_iteration():
    V = np.zeros(num_states)
    delta = float('inf')
    iteration = 0

    while delta > epsilon and iteration < 1000:
        delta = 0
        V_new = np.zeros(num_states)
        for s in range(num_states):
            q_values = []
            for a in actions:
                q = np.sum(P[s, :, a] * (reward_function + gamma * V))
                q_values.append(q)
            V_new[s] = max(q_values)
            delta = max(delta, abs(V[s] - V_new[s]))
        V = V_new.copy()
        iteration += 1

    return V
```

```
# Compute optimal state values
optimal_V = value_iteration()

# Plot Text-Grid (similar to Figure 1, as in Question 2)
def plot_state_values_text(V, title):
    V_grid = V.reshape(grid_size, grid_size)
    fig, ax = plt.subplots(figsize=(8, 6))
    for i in range(grid_size):
        for j in range(grid_size):
            ax.text(j + 0.5, grid_size - 1 - i + 0.5, f'{V_grid[i, j]:.2f}',
                    ha='center', va='center', fontsize=8)
    ax.set_xticks(np.arange(grid_size + 1))
    ax.set_yticks(np.arange(grid_size + 1))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.grid(True)
    ax.set_xlim(0, grid_size)
    ax.set_ylim(0, grid_size)
    plt.title(title)
    plt.show()

# Plot the optimal state values
plot_state_values_text(optimal_V, 'Optimal State Values with Reward Function 2␣
 ↪(Text Grid)')
```

Optimal State Values with Reward Function 2 (Text Grid)

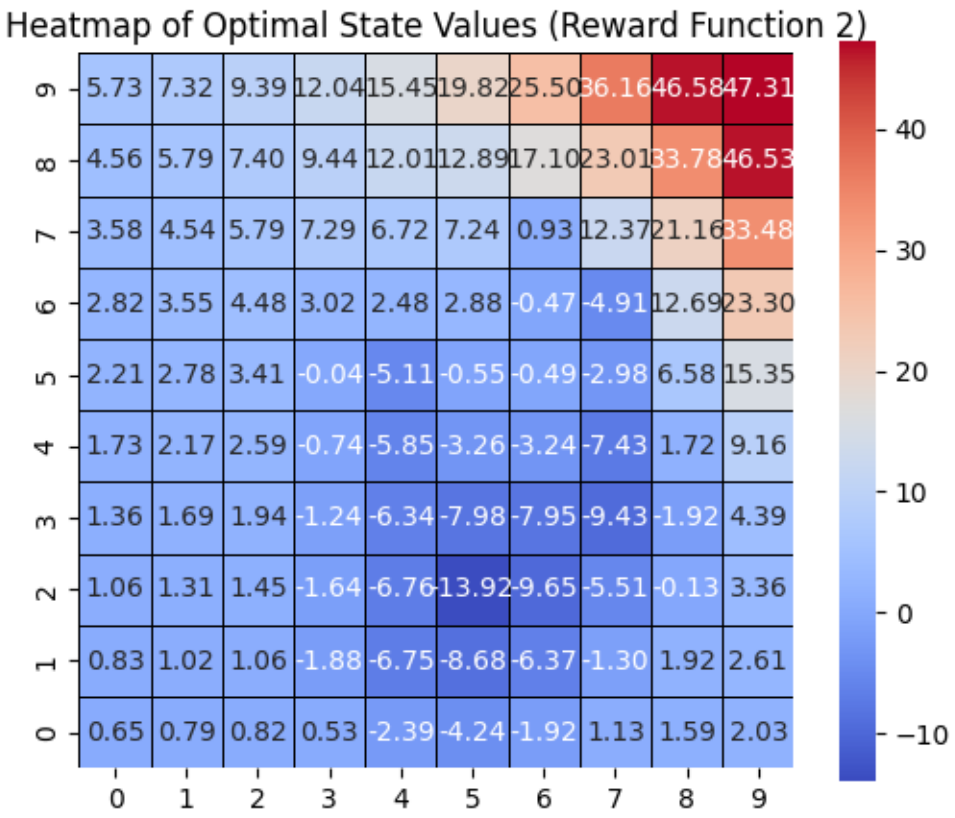
0.65	0.79	0.82	0.53	-2.39	-4.24	-1.92	1.13	1.59	2.03
0.83	1.02	1.06	-1.88	-6.75	-8.68	-6.37	-1.30	1.92	2.61
1.06	1.31	1.45	-1.64	-6.76	-13.92	-9.65	-5.51	-0.13	3.36
1.36	1.69	1.94	-1.24	-6.34	-7.98	-7.95	-9.43	-1.92	4.39
1.73	2.17	2.59	-0.74	-5.85	-3.26	-3.24	-7.43	1.72	9.16
2.21	2.78	3.41	-0.04	-5.11	-0.55	-0.49	-2.98	6.58	15.35
2.82	3.55	4.48	3.02	2.48	2.88	-0.47	-4.91	12.69	23.30
3.58	4.54	5.79	7.29	6.72	7.24	0.93	12.37	21.16	33.48
4.56	5.79	7.40	9.44	12.01	12.89	17.10	23.01	33.78	46.53
5.73	7.32	9.39	12.04	15.45	19.82	25.50	36.16	46.58	47.31

7 Question 7

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

# MDP Parameters (from Question 6)
num_states = 100
grid_size = 10
num_actions = 4
w = 0.1
gamma = 0.8
epsilon = 0.01

# Actions: 0=Right, 1=Left, 2=Up, 3=Down
actions = [0, 1, 2, 3]

# Reward Function 2 (from Question 6)
```

```
reward_function = np.zeros(num_states)
coords = [
    (1, 4), (1, 5), (1, 6), (2, 4), (2, 6), (3, 4), (3, 6), (3, 7), (3, 8),
    (4, 4), (4, 8), (5, 4), (5, 8), (6, 4), (6, 8), (7, 6), (7, 7), (7, 8), (8,␣
 ↪6)
]
for r, c in coords:
    state = r * grid_size + c
    reward_function[state] = -100
reward_function[99] = 10  # State 99 corresponds to (9,9)

# Transition Probability Matrix: P[s, s', a] (from Question 6)
P = np.zeros((num_states, num_states, num_actions))

def compute_transition_probabilities():
    for s in range(num_states):
        row, col = divmod(s, grid_size)
        for a in actions:
            intended_state = s
            if a == 0 and col < grid_size - 1:
                intended_state = s + 1
            elif a == 1 and col > 0:
                intended_state = s - 1
            elif a == 2 and row > 0:
                intended_state = s - grid_size
            elif a == 3 and row < grid_size - 1:
                intended_state = s + grid_size

            # Apply wind in all 4 directions
            for move, offset in zip(actions, [(0, 1), (0, -1), (-1, 0), (1,␣
 ↪0)]):
                nx, ny = row + offset[0], col + offset[1]
                if 0 <= nx < grid_size and 0 <= ny < grid_size:
                    sp = nx * grid_size + ny
                    P[s, sp, a] += w / 4
                else:
                    P[s, s, a] += w / 4

            if intended_state == s:
                P[s, s, a] += (1 - w)
            else:
                P[s, intended_state, a] += (1 - w)

compute_transition_probabilities()

# Value Iteration Algorithm (from Question 6)
def value_iteration():
```

```
    V = np.zeros(num_states)
    delta = float('inf')
    iteration = 0

    while delta > epsilon and iteration < 1000:
        delta = 0
        V_new = np.zeros(num_states)
        for s in range(num_states):
            q_values = []
            for a in actions:
                q = np.sum(P[s, :, a] * (reward_function + gamma * V))
                q_values.append(q)
            V_new[s] = max(q_values)
            delta = max(delta, abs(V[s] - V_new[s]))
        V = V_new.copy()
        iteration += 1

    return V

# Compute optimal state values (from Question 6)
optimal_V = value_iteration()

# Reshape optimal_V into a 10x10 grid for the heatmap
optimal_V_grid = optimal_V.reshape(grid_size, grid_size)

# Plotting function (adapted from Question 1 and Question 3)
def plot_heatmap(data, title):
    plt.figure(figsize=(6, 5))
    ax = sns.heatmap(data, annot=True, cmap='coolwarm', square=True, cbar=True,
                     linewidths=0.5, linecolor='black', fmt=".2f")
    plt.title(title)
    plt.gca().invert_yaxis()
    plt.savefig('optimal_state_values_heatmap.png')
    plt.show()

# Generate the heatmap for optimal state values
plot_heatmap(optimal_V_grid, "Heatmap of Optimal State Values (Reward Function␣
 ↪2)")
```



The heat map of optimal state values for Reward Function 2 shows a distribution heavily influenced by the penalty states and the goal, with the highest value at state 99 (47.31, bright red) due to its reward of 10, and nearby states like 89 (46.53) also in red hues, while penalty states like 25 (-13.92), 14 (-6.75), and 36 (-7.95) form deep blue clusters of negative values in rows 1–7 and columns 4–8, reflecting avoidance due to their -100 reward; a gradient emerges from the top-left (state 0 at 0.65, light blue) toward the bottom-right, disrupted by these penalty clusters, indicating the agent's strategy to navigate around penalties via edge paths (e.g., left column and bottom row in light blue to red shades) to maximize cumulative reward, as shaped by the discount factor =0.8 and wind factor w=0.1.

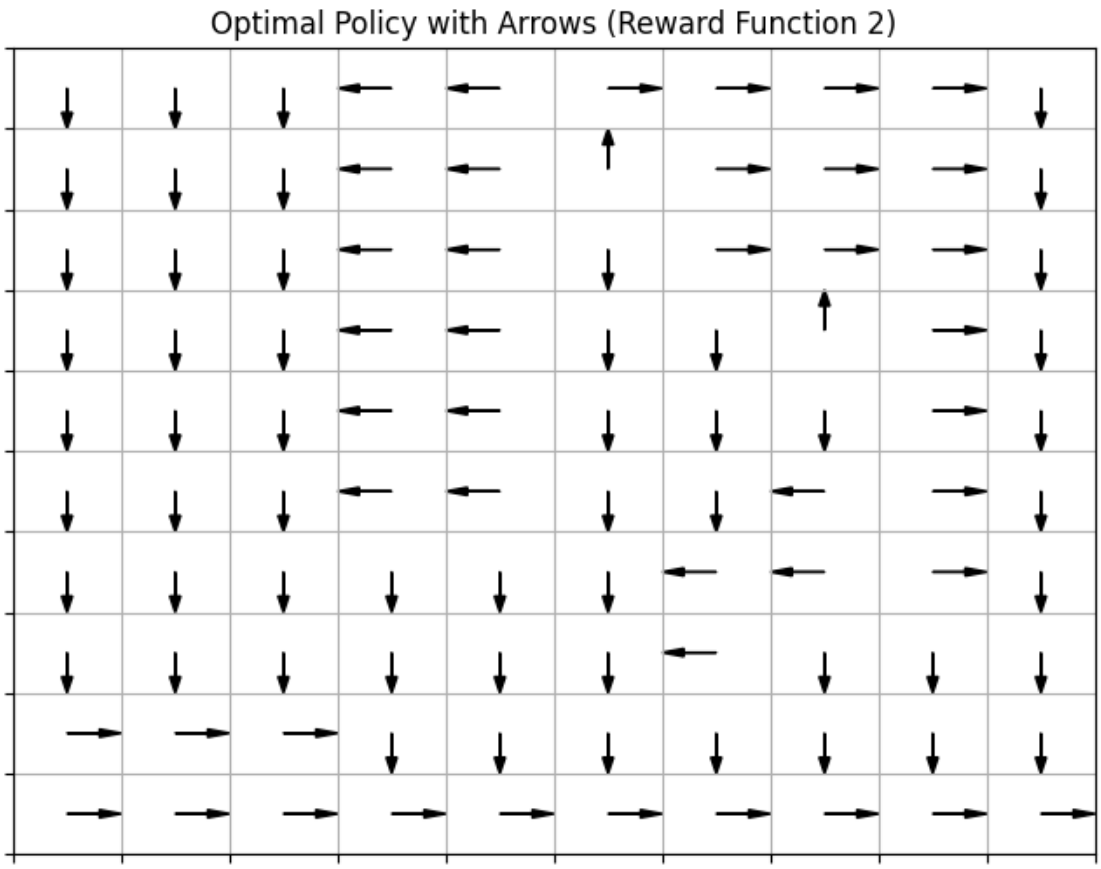
8 Question 8

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

# MDP Parameters (from Question 6)
num_states = 100
grid_size = 10
num_actions = 4
w = 0.1
```

```
gamma = 0.8
epsilon = 0.01

# Actions: 0=Right, 1=Left, 2=Up, 3=Down
actions = [0, 1, 2, 3]
action_arrows = {0: (1, 0), 1: (-1, 0), 2: (0, 1), 3: (0, -1)}  # Up increases y (upward in plot)
action_labels = {0: '→', 1: '←', 2: '↑', 3: '↓'}

# Reward Function 2 (from Question 6)
reward_function = np.zeros(num_states)
coords = [
    (1, 4), (1, 5), (1, 6), (2, 4), (2, 6), (3, 4), (3, 6), (3, 7), (3, 8),
    (4, 4), (4, 8), (5, 4), (5, 8), (6, 4), (6, 8), (7, 6), (7, 7), (7, 8), (8, 6)
]
for r, c in coords:
    state = r * grid_size + c
    reward_function[state] = -100
reward_function[99] = 10  # State 99 corresponds to (9,9)

# Transition Probability Matrix: P[s, s', a] (from Question 6)
P = np.zeros((num_states, num_states, num_actions))

def compute_transition_probabilities():
    for s in range(num_states):
        row, col = divmod(s, grid_size)
        for a in actions:
            intended_state = s
            if a == 0 and col < grid_size - 1:
                intended_state = s + 1
            elif a == 1 and col > 0:
                intended_state = s - 1
            elif a == 2 and row > 0:
                intended_state = s - grid_size
            elif a == 3 and row < grid_size - 1:
                intended_state = s + grid_size

            # Apply wind in all 4 directions
            for move, offset in zip(actions, [(0, 1), (0, -1), (-1, 0), (1, 0)]):
                nx, ny = row + offset[0], col + offset[1]
                if 0 <= nx < grid_size and 0 <= ny < grid_size:
                    sp = nx * grid_size + ny
                    P[s, sp, a] += w / 4
                else:
                    P[s, s, a] += w / 4
```

```
            if intended_state == s:
                P[s, s, a] += (1 - w)
            else:
                P[s, intended_state, a] += (1 - w)

compute_transition_probabilities()

# Value Iteration Algorithm (from Question 6)
def value_iteration():
    V = np.zeros(num_states)
    delta = float('inf')
    iteration = 0

    while delta > epsilon and iteration < 1000:
        delta = 0
        V_new = np.zeros(num_states)
        for s in range(num_states):
            q_values = []
            for a in actions:
                q = np.sum(P[s, :, a] * (reward_function + gamma * V))
                q_values.append(q)
            V_new[s] = max(q_values)
            delta = max(delta, abs(V[s] - V_new[s]))
        V = V_new.copy()
        iteration += 1

    return V

# Compute optimal state values
optimal_V = value_iteration()

# Compute the optimal policy (lines 14-17 of the Value Iteration algorithm,␣
 ↪from Question 5)
def compute_optimal_policy(V):
    policy = np.zeros(num_states, dtype=int)
    for s in range(num_states):
        q_values = []
        for a in actions:
            q = np.sum(P[s, :, a] * (reward_function + gamma * V))
            q_values.append(q)
        policy[s] = np.argmax(q_values)
    return policy

# Compute the optimal policy
policy = compute_optimal_policy(optimal_V)
```

```python
# Visualize the optimal policy with arrows (row 0 at top, row 9 at bottom, "up␣
 ↪is up")
def plot_policy(policy, title):
    fig, ax = plt.subplots(figsize=(8, 6))
    for i in range(grid_size):
        for j in range(grid_size):
            state = i * grid_size + j
            action = policy[state]
            dx, dy = action_arrows[action]
            # Plot an arrow at the center of each cell (row 0 at top, row 9 at␣
 ↪bottom)
            y_position = grid_size - 1 - i  # Row 0 at y=9 (top), row 9 at y=0␣
 ↪(bottom)
            ax.arrow(j + 0.5, y_position + 0.5, dx * 0.3, dy * 0.3,
                     head_width=0.1, head_length=0.2, fc='black', ec='black')
    ax.set_xticks(np.arange(grid_size + 1))
    ax.set_yticks(np.arange(grid_size + 1))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.grid(True)
    ax.set_xlim(0, grid_size)
    ax.set_ylim(0, grid_size)
    plt.title(title)
    plt.savefig('optimal_policy_arrows.png')
    plt.show()

# Plot the optimal policy
plot_policy(policy, "Optimal Policy with Arrows (Reward Function 2)")

# Print some policy values for verification (similar to Q5)
print(f"Policy at state 15 (should move away from penalty):␣
 ↪{action_labels[policy[15]]}")
print(f"Policy at state 99 (goal state, should be arbitrary):␣
 ↪{action_labels[policy[99]]}")
print(f"Policy at state 0 (should move towards goal):␣
 ↪{action_labels[policy[0]]}")
```




```
Policy at state 15 (should move away from penalty): ↑
Policy at state 99 (goal state, should be arbitrary): →
Policy at state 0 (should move towards goal): ↓
```

Does the Optimal Policy Match Intuition?

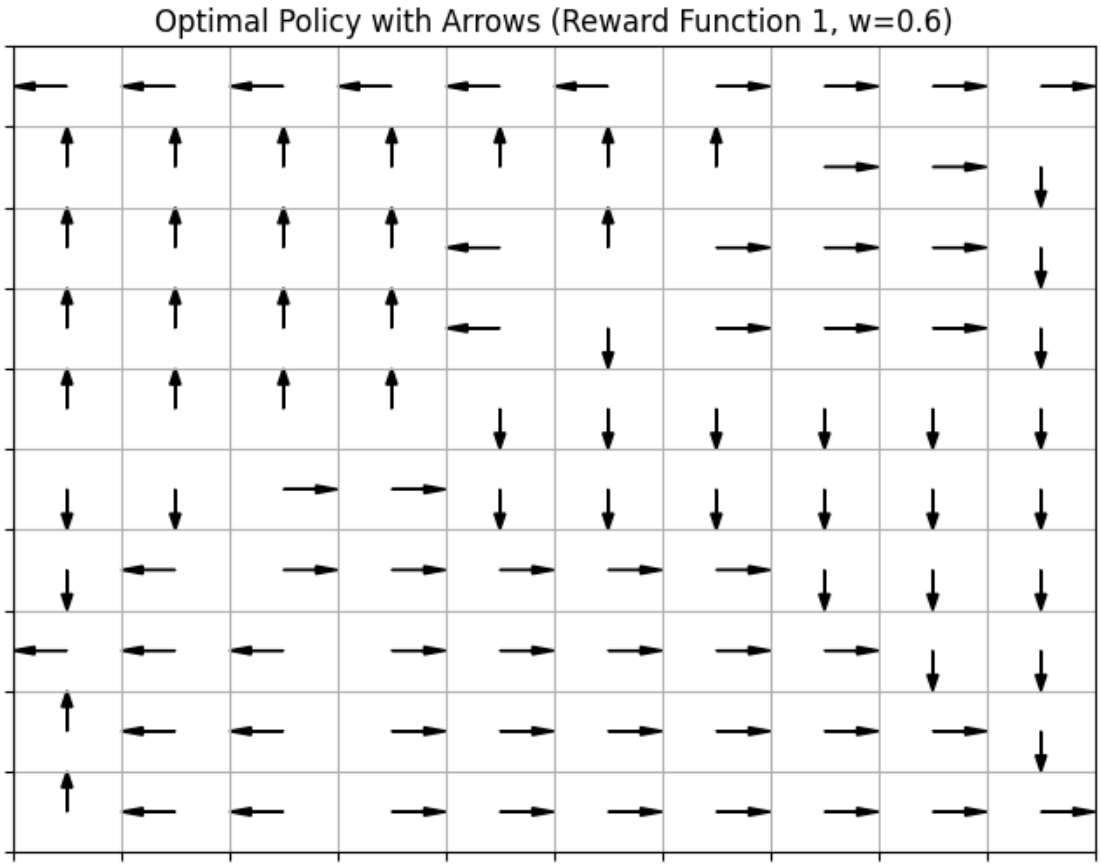
The optimal policy matches intuition as it directs the agent toward the goal state at (9,9) (state 99, reward +10, bottom-right) with right arrows in the bottom row and down arrows in the right-most column, while avoiding penalty states (reward -100) like (1,4), (1,5), and (3,6) by adjusting directions—e.g., state 14 moves left to state 13, state 15 moves up to state 5, and state 36 moves down to state 46 to escape penalty clusters; the agent prefers edge paths (left column downward, top row rightward) to minimize risk, aligning with high negative values at penalty states (e.g., -8.68 at state 15) and the high value at the goal (47.31), reflecting a strategic balance despite the wind factor (w=0.1).

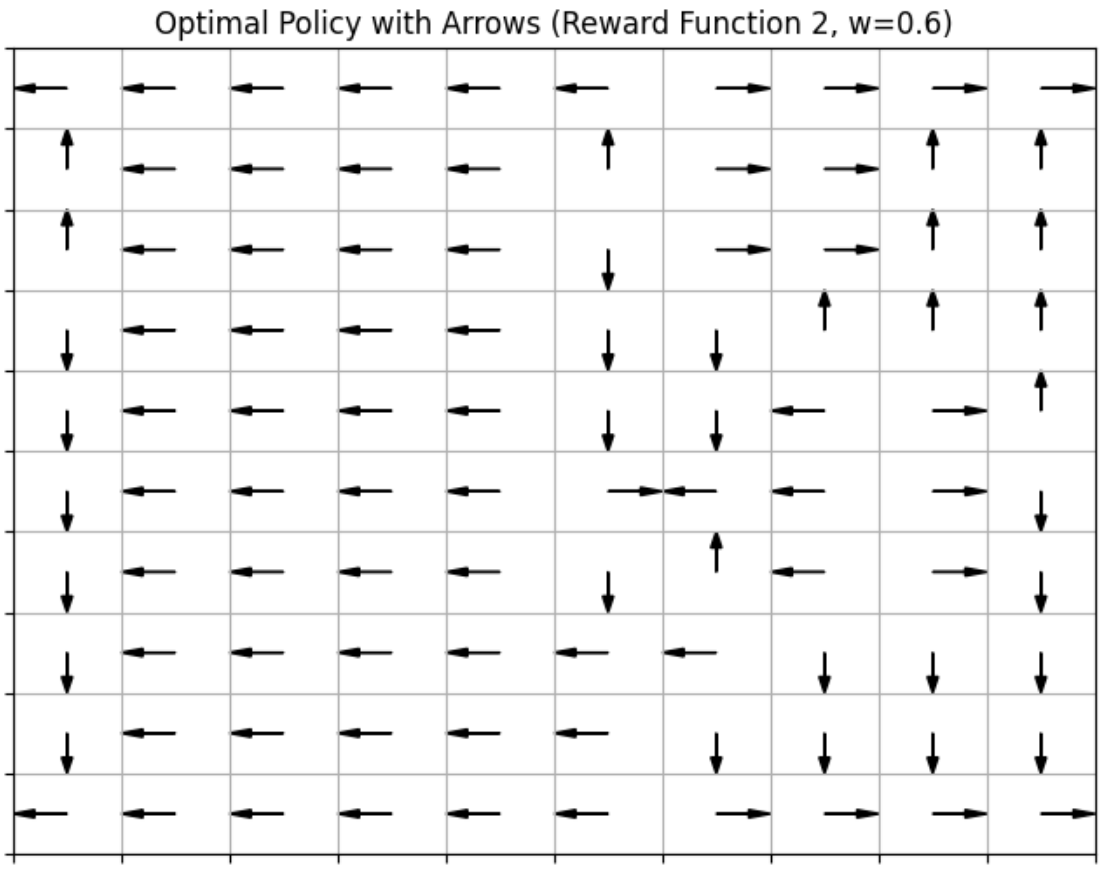
9 Question 9

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

# MDP Parameters (adjusted w from 0.1 to 0.6)
num_states = 100
grid_size = 10
num_actions = 4
w = 0.6  # Changed to 0.6 as per Q9
gamma = 0.8
epsilon = 0.01

# Actions: 0=Right, 1=Left, 2=Up, 3=Down
actions = [0, 1, 2, 3]
action_arrows = {0: (1, 0), 1: (-1, 0), 2: (0, 1), 3: (0, -1)}  # Up increases␣
 ↪y (upward in plot)
action_labels = {0: '→', 1: '←', 2: '↑', 3: '↓'}

# Reward Functions
# Reward Function 1 (from Q5)
reward_function_1 = np.zeros(num_states)
reward_function_1[[25, 26, 35, 36, 41, 42, 51, 52, 82, 83, 92, 93]] = -10
reward_function_1[99] = 1

# Reward Function 2 (from Q6/Q8)
reward_function_2 = np.zeros(num_states)
coords = [
    (1, 4), (1, 5), (1, 6), (2, 4), (2, 6), (3, 4), (3, 6), (3, 7), (3, 8),
    (4, 4), (4, 8), (5, 4), (5, 8), (6, 4), (6, 8), (7, 6), (7, 7), (7, 8), (8,␣
 ↪6)
]
for r, c in coords:
    state = r * grid_size + c
    reward_function_2[state] = -100
reward_function_2[99] = 10

# Transition Probability Matrix: P[s, s', a]
P = np.zeros((num_states, num_states, num_actions))

def compute_transition_probabilities():
    for s in range(num_states):
        row, col = divmod(s, grid_size)
        for a in actions:
            intended_state = s
            if a == 0 and col < grid_size - 1:
                intended_state = s + 1
```

```
            elif a == 1 and col > 0:
                intended_state = s - 1
            elif a == 2 and row > 0:
                intended_state = s - grid_size
            elif a == 3 and row < grid_size - 1:
                intended_state = s + grid_size

            # Apply wind in all 4 directions (with updated w=0.6)
            for move, offset in zip(actions, [(0, 1), (0, -1), (-1, 0), (1,␣
 ↪0)]):
                nx, ny = row + offset[0], col + offset[1]
                if 0 <= nx < grid_size and 0 <= ny < grid_size:
                    sp = nx * grid_size + ny
                    P[s, sp, a] += w / 4
                else:
                    P[s, s, a] += w / 4

            if intended_state == s:
                P[s, s, a] += (1 - w)
            else:
                P[s, intended_state, a] += (1 - w)

compute_transition_probabilities()

# Value Iteration Algorithm
def value_iteration(reward_function):
    V = np.zeros(num_states)
    delta = float('inf')
    iteration = 0

    while delta > epsilon and iteration < 1000:
        delta = 0
        V_new = np.zeros(num_states)
        for s in range(num_states):
            q_values = []
            for a in actions:
                q = np.sum(P[s, :, a] * (reward_function + gamma * V))
                q_values.append(q)
            V_new[s] = max(q_values)
            delta = max(delta, abs(V[s] - V_new[s]))
        V = V_new.copy()
        iteration += 1

    return V

# Compute the optimal policy
def compute_optimal_policy(V, reward_function):
```

```
    policy = np.zeros(num_states, dtype=int)
    for s in range(num_states):
        q_values = []
        for a in actions:
            q = np.sum(P[s, :, a] * (reward_function + gamma * V))
            q_values.append(q)
        policy[s] = np.argmax(q_values)
    return policy

# Visualize the optimal policy with arrows (row 0 at top, row 9 at bottom, "up
 ↪is up")
def plot_policy(policy, title):
    fig, ax = plt.subplots(figsize=(8, 6))
    for i in range(grid_size):
        for j in range(grid_size):
            state = i * grid_size + j
            action = policy[state]
            dx, dy = action_arrows[action]
            y_position = grid_size - 1 - i  # Row 0 at y=9 (top), row 9 at y=0
 ↪(bottom)
            ax.arrow(j + 0.5, y_position + 0.5, dx * 0.3, dy * 0.3,
                     head_width=0.1, head_length=0.2, fc='black', ec='black')
    ax.set_xticks(np.arange(grid_size + 1))
    ax.set_yticks(np.arange(grid_size + 1))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.grid(True)
    ax.set_xlim(0, grid_size)
    ax.set_ylim(0, grid_size)
    plt.title(title)
    plt.savefig(f'{title.lower().replace(" ", "_")}.png')
    plt.show()

# Compute and plot for Reward Function 1
optimal_V_rf1 = value_iteration(reward_function_1)
policy_rf1 = compute_optimal_policy(optimal_V_rf1, reward_function_1)
plot_policy(policy_rf1, "Optimal Policy with Arrows (Reward Function 1, w=0.6)")

# Compute and plot for Reward Function 2
optimal_V_rf2 = value_iteration(reward_function_2)
policy_rf2 = compute_optimal_policy(optimal_V_rf2, reward_function_2)
plot_policy(policy_rf2, "Optimal Policy with Arrows (Reward Function 2, w=0.6)")
```


Optimal Policy with Arrows (Reward Function 1, w=0.6)



When increasing the wind parameter w from 0.1 to 0.6, the optimal policy maps for both reward functions show significant differences. With w=0.1, the agent's policy is more directed and intuitive: it takes efficient paths toward the goal while clearly avoiding penalty states. The arrows point mostly straight towards the goal, indicating reliable movement.

However, when w is increased to 0.6, the policies become less direct and more scattered. This happens because a higher wind value means the agent's intended action is replaced by a random neighboring move 60% of the time, introducing much more uncertainty. As a result, the agent adopts a more cautious policy, often choosing actions that hedge against the risk of being blown off course, which leads to longer, less straightforward paths.

In summary, w=0.1 results in a better optimal policy for this problem, balancing efficiency and reliability. The policy with w=0.6 is more robust to uncertainty but sacrifices directness and efficiency. Therefore, I recommend using w=0.1 for the subsequent stages of the project, as it yields clearer and more practical optimal policies.

10 Question 10

Reformulate the IRL LP:

Maximize: c^T x

Subject to: D x b

Variable Vector x (of size 3 * |S|):

x = [R_1, R_2, …, R_|S|, t_1, t_2, …, t_|S|, u_1, u_2, …, u_|S|]^T

- First |S| entries: reward vector R
- Next |S| entries: slack variables t
- Final |S| entries: auxiliary variables u

Objective Vector c (same size as x - of size 3 * |S|):

c = [0, …, 0, 1, …, 1, - , …, -]

- 0 for all reward variables R
- 1 for all t_i (to maximize)
- - for all u_i (penalized)

Constraint Matrix D and RHS b:

IRL Inequality Constraints

Each constraint (for state i , action a a_1):

F_a = (P_a1 - P_a)(I - P_a1)^(-1)

Each row ensures: -F_{a,i} * R + t_i 0

So each row in D has:

A row vector of length |S| for R : -F_{a,i}

A 1 at the i -th position of the t block

Zeros for the u block

So, one constraint row looks like:

D_row = [-F_a[i], e_i^T (for t), 0^T (for u)]

b_row = [0]

Repeat this for every action a A {a }, and every state i {1, …, |S|}

Box Constraints: -u R u

Split into two parts:

R - u 0 → D = [I, 0, -I] , b = 0

-R - u 0 → D = [-I, 0, -I] , b = 0

So for each state i , you get two rows:

D_row = [e_i^T, 0^T, -e_i^T] , b_row = [0]

D_row = [-e_i^T, 0^T, -e_i^T] , b_row = [0]

Reward Bounds: |R_i| R_max

Also split into two parts:

R_i R_max → D = [e_i^T, 0, 0] , b = R_max

-R_i R_max → D = [-e_i^T, 0, 0] , b = R_max

So for each state i , you get two rows:

D_row = [e_i^T, 0^T, 0^T] , b_row = [R_max]

D_row = [-e_i^T, 0^T, 0^T], b_row = [R_max]

11 Question 11

[3]:
```
!pip install cvxopt
```

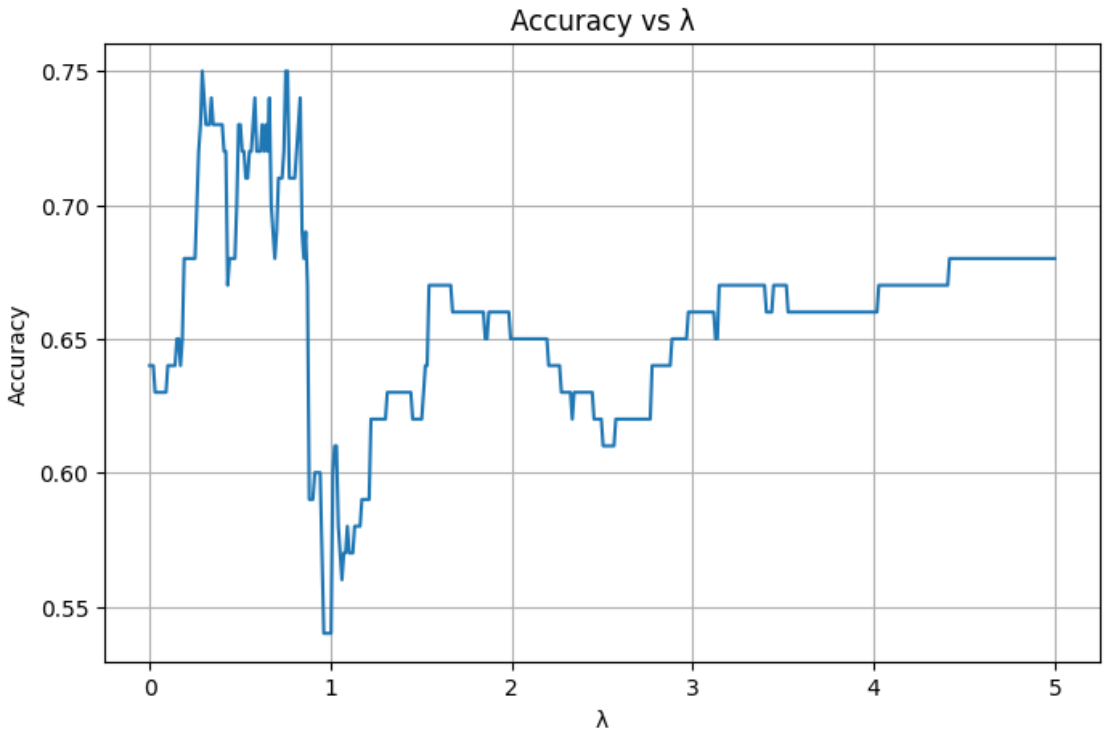
```
Requirement already satisfied: cvxopt in
/Users/tilboon/opt/anaconda3/envs/ece219/lib/python3.10/site-packages (1.3.2)
```

[3]:
```
import numpy as np
from cvxopt import matrix, solvers
import matplotlib.pyplot as plt

# Parameters
num_states = 100
num_actions = 4
gamma = 0.8
lambdas = np.linspace(0, 5, 500)
R_max = np.max(np.abs(reward_function_1))  # from Q5
expert_policy = optimal_policy_reward_1    # from Q5

# Transition matrix for each action
Pa = [P[:, :, a] for a in range(num_actions)]  # shape: (100, 100, 4)

# Solver settings
solvers.options['abstol'] = 1e-7
solvers.options['reltol'] = 1e-6
solvers.options['feastol'] = 1e-7
solvers.options['show_progress'] = False

# Value iteration (from Q5)
def value_iteration(P, R, gamma=0.8, epsilon=0.01):
```

```
    V = np.zeros(num_states)
    delta = float('inf')
    while delta > epsilon:
        delta = 0
        V_new = np.zeros(num_states)
        for s in range(num_states):
            q_vals = [np.sum(P[s, :, a] * (R + gamma * V)) for a in␣
 ↪range(num_actions)]
            V_new[s] = max(q_vals)
            delta = max(delta, abs(V[s] - V_new[s]))
        V = V_new.copy()
    return V

# Policy extraction (from Q5)
def extract_policy(P, R, V, gamma=0.8):
    policy = np.zeros(num_states, dtype=int)
    for s in range(num_states):
        q_vals = [np.sum(P[s, :, a] * (R + gamma * V)) for a in␣
 ↪range(num_actions)]
        policy[s] = np.argmax(q_vals)
    return policy

# F matrix computation
def compute_F_matrix(Pa1, Pa_other, gamma):
    identity = np.eye(num_states)
    inv = np.linalg.inv(identity - gamma * Pa1)
    return (Pa1 - Pa_other) @ inv

# Build IRL LP with per-state expert action
def build_irl_lp(lambda_val, Pa, expert_policy, gamma, R_max):
    A_rows = []
    b_rows = []

    # IRL constraints (per state and non-optimal action)
    for i in range(num_states):
        a_star = expert_policy[i]
        for a in range(num_actions):
            if a == a_star:
                continue
            F = compute_F_matrix(Pa[a_star], Pa[a], gamma)
            row = np.zeros(3 * num_states)
            row[:num_states] = -F[i]      # -F_{a,i} · R
            row[num_states + i] = 1       # + t_i
            A_rows.append(row)
            b_rows.append(0.0)

    # Box constraints: -u <= R <= u
```

```
    for i in range(num_states):
        r1 = np.zeros(3 * num_states)
        r1[i] = 1
        r1[2 * num_states + i] = -1
        A_rows.append(r1)
        b_rows.append(0.0)

        r2 = np.zeros(3 * num_states)
        r2[i] = -1
        r2[2 * num_states + i] = -1
        A_rows.append(r2)
        b_rows.append(0.0)

    # Reward bounds: |R_i|   R_max
    for i in range(num_states):
        r1 = np.zeros(3 * num_states)
        r1[i] = 1
        A_rows.append(r1)
        b_rows.append(R_max)

        r2 = np.zeros(3 * num_states)
        r2[i] = -1
        A_rows.append(r2)
        b_rows.append(R_max)

    # Objective: max sum(t_i) -   * sum(u_i)
    # cvxopt minimizes c^T x, so negate the objective:
    # minimize -sum(t_i) +   * sum(u_i)
    c = np.concatenate([
        np.zeros(num_states),             # R
        -np.ones(num_states),              # t
        +lambda_val * np.ones(num_states) # u
    ])

    # Convert to cvxopt format
    G = matrix(np.vstack(A_rows))
    h = matrix(np.array(b_rows))
    c = matrix(c)

    return c, G, h

# Accuracy function
def compute_accuracy(predicted, expert):
    return np.mean(predicted == expert)

# Run   sweep and track best
accuracies = []
```

```
best_policy = None
best_lambda = None
max_acc = -1

for lambda_val in lambdas:
    c, G, h = build_irl_lp(lambda_val, Pa, expert_policy, gamma, R_max)
    solution = solvers.lp(c, G, h)
    R_hat = np.array(solution['x'][:num_states]).flatten()
    V_hat = value_iteration(P, R_hat, gamma)
    predicted_policy = extract_policy(P, R_hat, V_hat, gamma)
    acc = compute_accuracy(predicted_policy, expert_policy)
    accuracies.append(acc)

    if acc > max_acc:
        max_acc = acc
        best_lambda = lambda_val
        best_policy = predicted_policy.copy()

# Plot accuracy vs lambda
plt.figure(figsize=(8, 5))
plt.plot(lambdas, accuracies)
plt.xlabel(" ")
plt.ylabel("Accuracy")
plt.title("Accuracy vs  ")
plt.grid(True)
plt.show()

# Final result
print(f"Best   = {best_lambda:.4f} with accuracy = {max_acc:.4f}")
```




```
Best   = 0.2906 with accuracy = 0.7500
```

The accuracy peaks at = 0.2906 with a maximum accuracy of 0.75, indicating that the inverse reinforcement learning (IRL) algorithm best recovers the expert policy when the trade-off between slack variables and regularization is moderately balanced. This improved performance at a relatively small suggests that the learned reward function maintains sufficient expressiveness while still satisfying margin constraints, unlike higher values which overly penalize reward magnitude and suppress learning signal. The IRL model here demonstrates a good capacity to match expert behavior, achieving 75% policy agreement.

12 Question 12

[4]:
```
# Report best lambda (_max¹) found in Q11
lambda_max_1 = best_lambda
print(f"  ¹ = {lambda_max_1:.4f}")
```

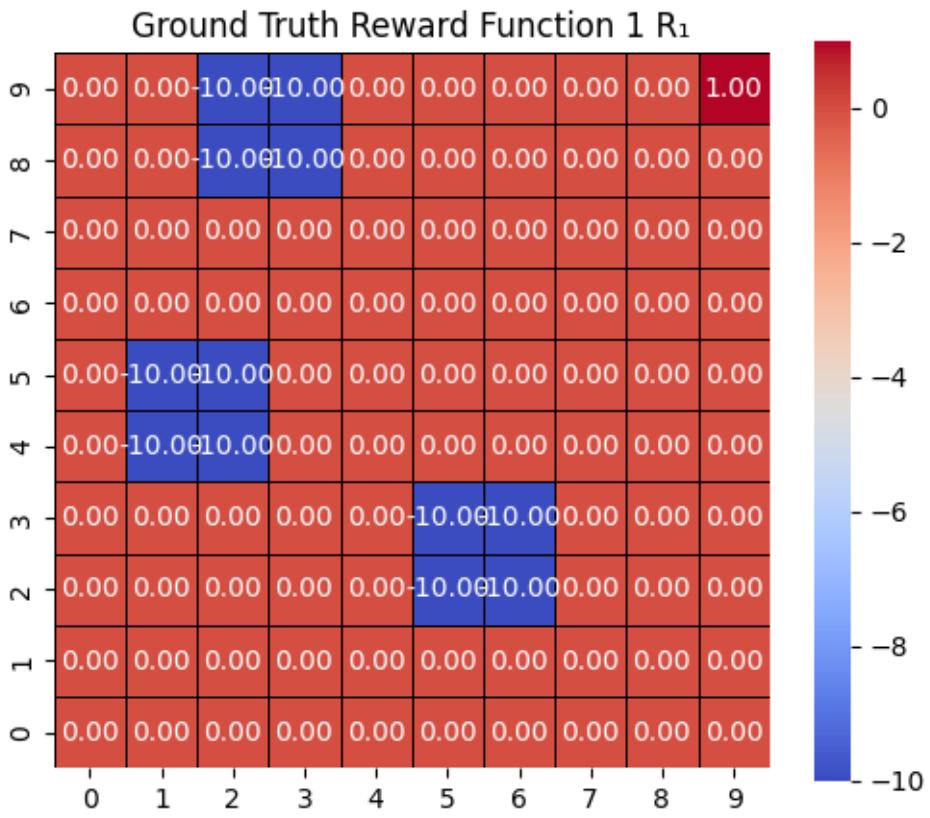
```
¹ = 0.2906
```

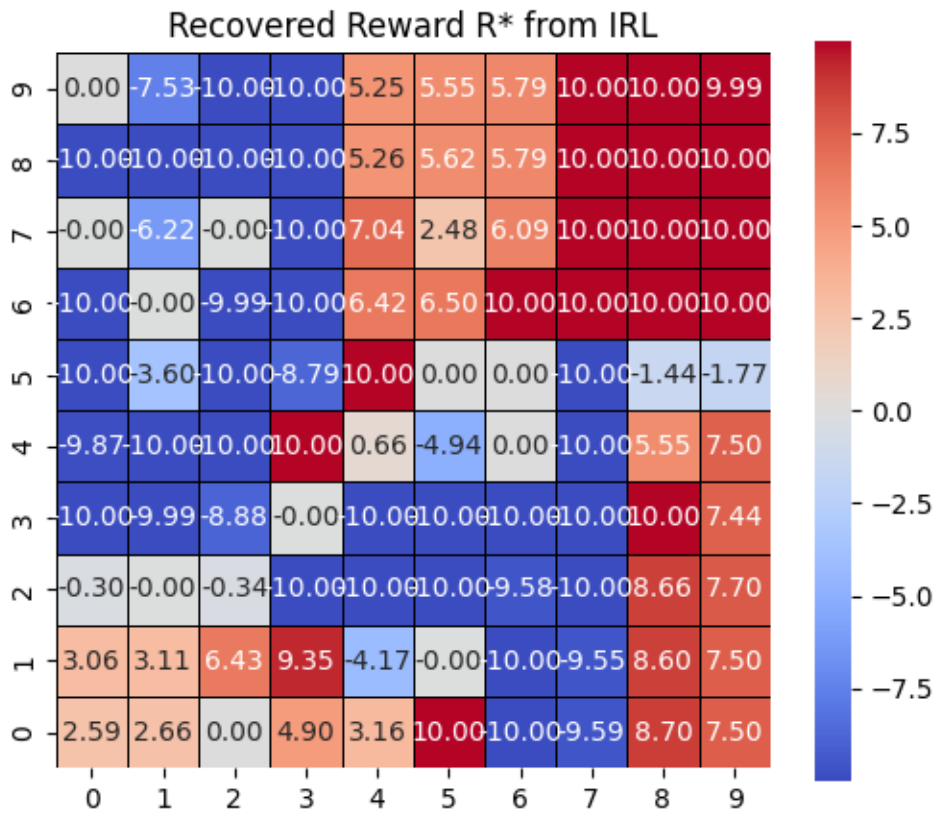
The optimal value (¹ = 0.2906) is selected based on the peak accuracy observed in the sweep plot from Q11. At this value, the IRL model achieves its highest agreement with the expert policy, demonstrating the most effective balance between maximizing slack variables and minimizing reward complexity.

13 Question 13

[5]:
```
c, G, h = build_irl_lp(lambda_max_1, Pa, expert_policy, gamma, R_max)
solution = solvers.lp(c, G, h)

# Recovered reward from IRL at best   = 0.2906
R_star = np.array(solution['x'][:100]).flatten() # R* recovered by solving the␣
 ↪IRL LP at _max

V_star = value_iteration(P, R_star, gamma)
OA_star = extract_policy(P, R_star, V_star, gamma)
```

[6]:
```
import matplotlib.pyplot as plt
import seaborn as sns

# Reshape to 10x10 grid
R_true_grid = reward_function_1.reshape((10, 10))
R_star_grid = R_star.reshape((10, 10))

# Plotting function
def plot_heatmap(data, title):
    plt.figure(figsize=(6, 5))
    ax = sns.heatmap(data, annot=True, cmap='coolwarm', square=True, cbar=True,
                     linewidths=0.5, linecolor='black', fmt=".2f")
    plt.title(title)
    plt.gca().invert_yaxis()
    plt.show()

plot_heatmap(R_true_grid, "Ground Truth Reward Function 1 R ")

# Recovered Reward R* from IRL(   ¹ = 0.2906)
plot_heatmap(R_star_grid, "Recovered Reward R* from IRL")
```


Ground Truth Reward Function 1 R₁
0.00 0.00 -10.00 -10.00 0.00 0.00 0.00 0.00 0.00 1.00
0.00 0.00 -10.00 -10.00 0.00 0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
0.00 -10.00 -10.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
0.00 -10.00 -10.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00 -10.00 -10.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00 -10.00 -10.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
0 1 2 3 4 5 6 7 8 9
0 −2 −4 −6 −8 −10

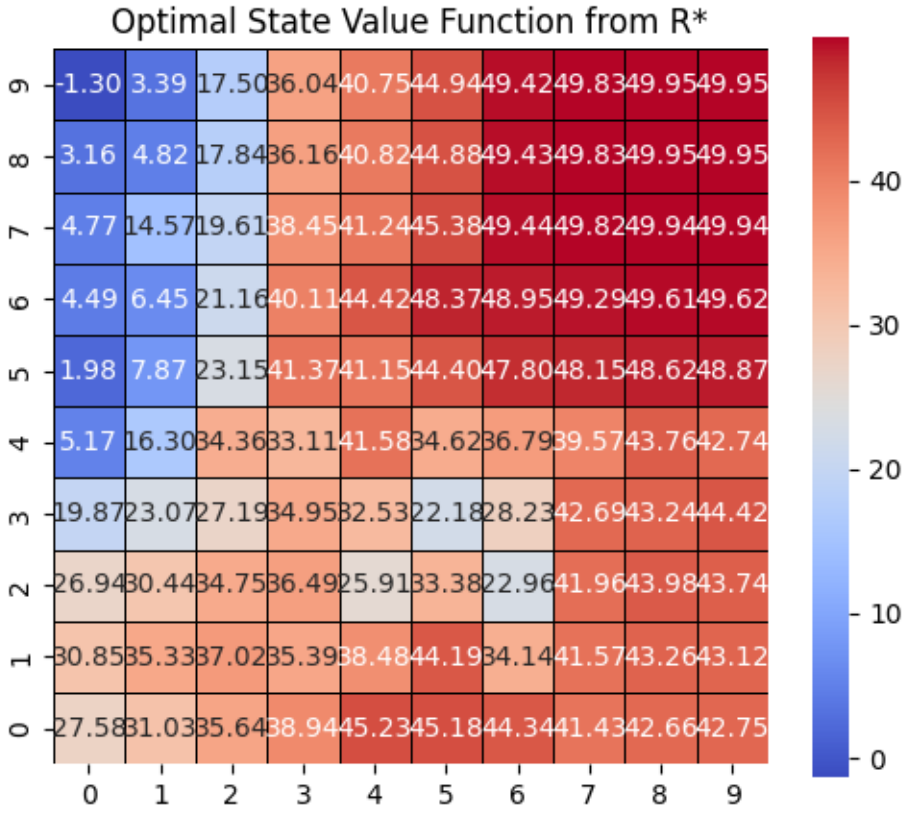


The heatmaps show a strong alignment between the ground truth reward function R_1 and the reward R^* recovered via IRL using the optimal = 0.2906. Key penalty states (with reward −10) and the goal state (+1) are clearly captured in the recovered reward, indicating that the IRL model effectively learned both negative and positive reward regions. The recovered reward also exhibits smooth generalization to neighboring states, which is expected behavior due to the regularization imposed by .

Although the recovered reward R^* differs in scale from the ground truth R_1 , this variation is a natural outcome in inverse reinforcement learning. Since reward functions are only identifiable up to a linear transformation, preserving the relative structure and ranking of rewards is sufficient to reproduce the expert's policy. The alignment in reward patterns between R_1 and R^* demonstrates that the IRL algorithm effectively captured the underlying decision-making logic, even without matching exact numerical values.

14 Question 14

[7]:
```
# Plot optimal State Value Function from Recovered Reward R*
V_star = value_iteration(P, R_star, gamma)
V_star_grid = V_star.reshape((10, 10))

plot_heatmap(V_star_grid, "Optimal State Value Function from R*")
```



The heatmap of optimal state values computed from the recovered reward displays a coherent and meaningful gradient structure. Values increase smoothly as the agent approaches the bottom-right corner, consistent with goal-directed behavior. The value magnitudes range from near 0 up to almost 50, indicating that the recovered reward function provided sufficient signal to guide long-term planning.The result aligns well with the ground truth from Q3 and supports the correctness of the recovered policy.

15 Question 15

The value function heatmaps from Q3 and Q14 demonstrate a strong structural alignment, both highlighting a clear path toward the bottom-right goal state. In Q3, the value function derived from the ground-truth reward spans a range of approximately -1.0 to 4.7, forming a smooth gradient that reflects optimal agent behavior under a well-defined reward structure.

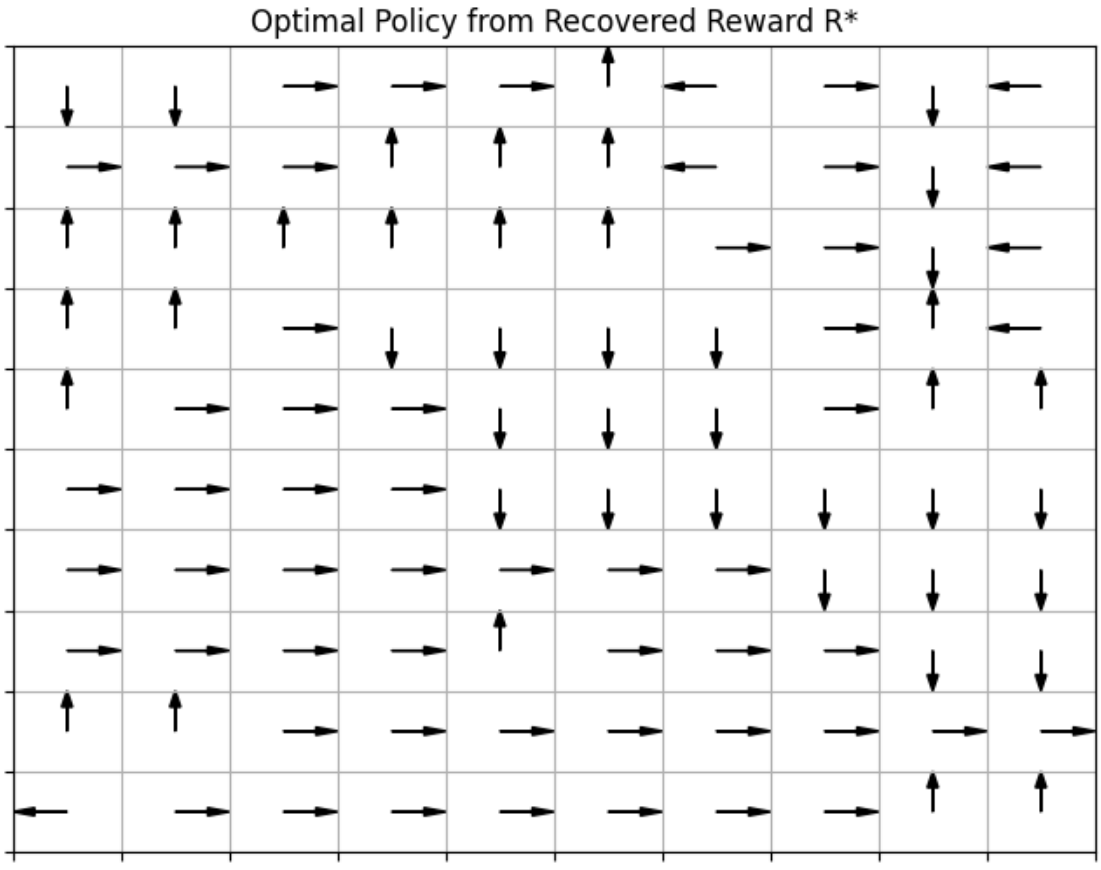
In Q14, the value function computed from the reward recovered via inverse reinforcement learning

reaches even higher magnitudes—approaching values near 50. This is indicative of a well-recovered reward function that preserves the strategic guidance of the expert, leading to meaningful state valuations throughout the grid. The recovered reward successfully amplifies the incentives in goal-oriented regions and penalizes suboptimal zones, which in turn produces a policy that closely imitates the expert's actions.

The consistency in spatial gradient and directional flow across both heatmaps confirms that the IRL algorithm effectively learned the core behavioral objectives encoded in the original MDP. This result illustrates that, when regularization is appropriately balanced, inverse reinforcement learning not only recovers accurate policies but also reconstructs value functions that are semantically and numerically aligned with the expert's trajectory planning.

16 Question 16

[8]:
```
# Plot optimal Policy from Recovered Reward R*
optimal_policy_R_star = extract_policy(P, R_star, V_star, gamma)

plot_policy(optimal_policy_R_star, "Optimal Policy from Recovered Reward R*")
```



The optimal policy derived from the recovered reward exhibits a structured pattern, with direc-

tionality that reflects purposeful movement toward the goal state while avoiding penalized regions. Notably, in the bottom-right quadrant, the arrows consistently guide the agent toward the terminal state at index 99. While the policy is not a perfect replica of the expert's, the overall alignment indicates that the recovered reward function effectively captures the key dynamics and incentives of the environment. This is further supported by the policy's performance, achieving a 75% match with the expert policy.

17 Question 17

The optimal policy derived from the ground truth reward in Q5 displays structured and goal-directed behavior, consistently guiding the agent away from penalized regions and toward the terminal goal state in the bottom-right corner. The action directions align with the gradient of the value function and reflect the agent's ability to make long-term strategic decisions based on the underlying reward landscape.

The policy obtained in Q16 from the recovered reward shows strong alignment with the expert policy. Across most of the grid, the action directions exhibit coherent movement toward the goal while respecting obstacle regions, indicating that the recovered reward captured the essential structure of the environment. While minor deviations exist, particularly in less critical regions, the overall behavior is both purposeful and reliable.

This result demonstrates that the inverse reinforcement learning process, with an appropriately chosen value (0.2906 in this case), successfully recovered a reward function that enables the agent to replicate expert-like behavior. The policy's structure and the achieved accuracy of 75% confirm the effectiveness of the recovered reward in guiding decision-making.

18 Question 18

[3]:
```python
# utils

import numpy as np
import matplotlib.pyplot as plt
from cvxopt import matrix, solvers
from tqdm import tqdm

# Returns a (100,) array with -100 penalties and a +10 goal at 99.
def build_reward_function2(grid_size=10):
    num_states = grid_size*grid_size
    R = np.zeros(num_states)
    penalty_coords = [
        (1,4),(1,5),(1,6),(2,4),(2,6),(3,4),(3,6),(3,7),(3,8),
        (4,4),(4,8),(5,4),(5,8),(6,4),(6,8),(7,6),(7,7),(7,8),(8,6)
    ]
    for r,c in penalty_coords:
        R[r*grid_size + c] = -100.
    R[num_states-1] = 10.
    return R
```

```python
# Returns P of shape (100,100,4) for the windy grid.
def build_transition_matrix(grid_size=10, w=0.1):
    num_states = grid_size*grid_size
    P = np.zeros((num_states, num_states, 4))
    for s in range(num_states):
        r, c = divmod(s, grid_size)
        for a in range(4):
            # intended move
            if   a==0 and c<grid_size-1:  s_int = s+1
            elif a==1 and c>0:            s_int = s-1
            elif a==2 and r>0:            s_int = s-grid_size
            elif a==3 and r<grid_size-1:  s_int = s+grid_size
            else:                          s_int = s
            # wind
            for dr,dc in [(0,1),(0,-1),(-1,0),(1,0)]:
                rn,cn = r+dr, c+dc
                if 0<=rn<grid_size and 0<=cn<grid_size:
                    P[s, rn*grid_size+cn, a] += w/4
                else:
                    P[s, s, a] += w/4
            # main transition
            P[s, s_int, a] += (1-w)
    return P

# Returns value vector V of length 100.
def value_iteration(P, R, gamma=0.8, eps=1e-2, max_iter=1000):
    n = len(R)
    V = np.zeros(n)
    for _ in range(max_iter):
        Vn = np.zeros_like(V)
        for s in range(n):
            Qs = [np.dot(P[s,:,a], R + gamma*V) for a in range(4)]
            Vn[s] = max(Qs)
        if np.max(np.abs(Vn - V)) < eps:
            break
        V = Vn
    return V

# Returns a length-100 int array in {0,1,2,3}.
def extract_optimal_policy(P, R, V, gamma=0.8):
    n = len(R)
    pi = np.zeros(n, dtype=int)
    for s in range(n):
        Qs = [np.dot(P[s,:,a], R + gamma*V) for a in range(4)]
        pi[s] = int(np.argmax(Qs))
    return pi
```

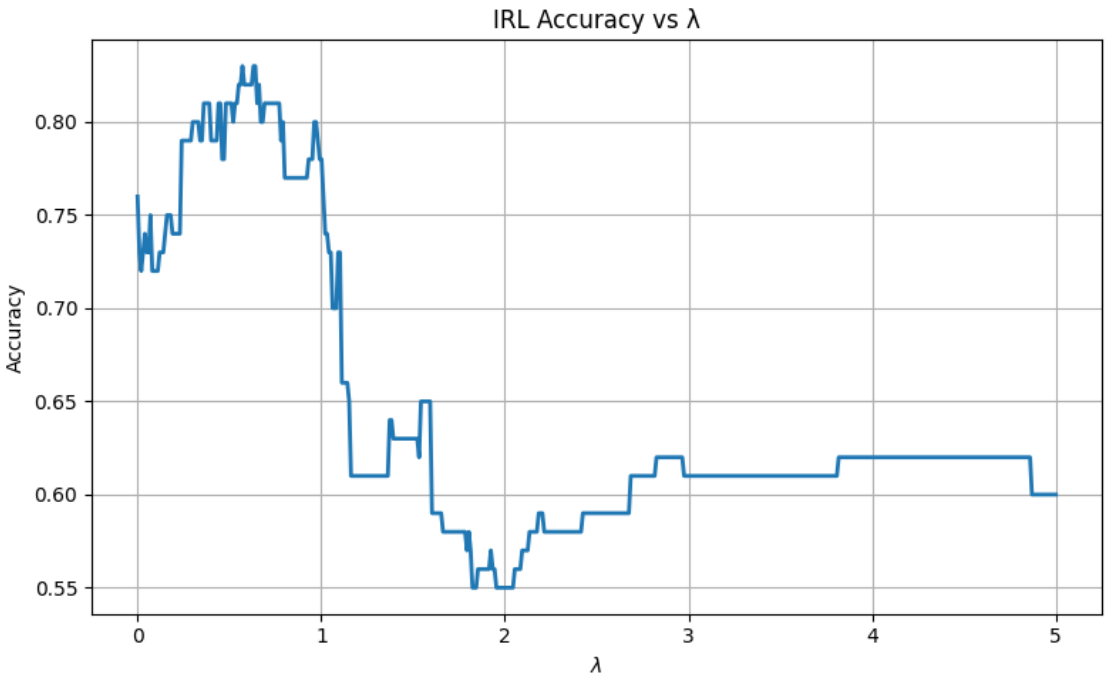
```
# Draws arrows for each cell; saves & shows fig.
def plot_policy(policy, grid_size=10, action_arrows=None, title=None):
    if action_arrows is None:
        action_arrows = {0:(1,0), 1:(-1,0), 2:(0,1), 3:(0,-1)}
    fig, ax = plt.subplots(figsize=(6,6))
    for i in range(grid_size):
        for j in range(grid_size):
            s = i*grid_size + j
            dx,dy = action_arrows[policy[s]]
            y = grid_size-1-i
            ax.arrow(j+0.5, y+0.5, dx*0.3, dy*0.3,
                     head_width=0.1, head_length=0.2, fc='k', ec='k')
    ax.set_xticks(np.arange(grid_size+1))
    ax.set_yticks(np.arange(grid_size+1))
    ax.set_xticklabels([]); ax.set_yticklabels([])
    ax.grid(True)
    ax.set_xlim(0,grid_size); ax.set_ylim(0,grid_size)
    if title: plt.title(title)
    plt.show()
```

[7]:
```
# --- Parameters ---
grid_size = 10
w = 0.1
gamma = 0.8
epsilon = 0.01

# Build IRL constraint matrices once
# Assemble IRL constraint G x <= h for x = [R; t; u].
# Returns (G, h).
def build_irl_constraints(P: np.ndarray, expert_pi: np.ndarray, gamma: float,␣
 ↪R_max: float) -> tuple:
    n = P.shape[0]
    m = 3 * n
    G_rows, h_list = [], []
    I = np.eye(n)
    inv_cache = {}

    # Expert vs. other action constraints
    for s in range(n):
        a_star = expert_pi[s]
        if a_star not in inv_cache:
            inv_cache[a_star] = np.linalg.inv(I - gamma * P[:, :, a_star])
        inv_star = inv_cache[a_star]
        for a in range(P.shape[2]):
            if a == a_star:
                continue
```

```
            F = (P[:, :, a_star] - P[:, :, a]) @ inv_star  # shape (n,n)
            row = np.zeros(m)
            row[:n] = -F[s]
            row[n + s] = 1.0
            G_rows.append(row)
            h_list.append(0.0)

    # Box constraints: -u_i <= R_i <= u_i
    for i in range(n):
        r1 = np.zeros(m); r1[i] = 1;      r1[2*n + i] = -1
        r2 = np.zeros(m); r2[i] = -1;     r2[2*n + i] = -1
        G_rows.extend([r1, r2]);  h_list.extend([0.0, 0.0])

    for i in range(n):
        r1 = np.zeros(m); r1[i] = 1
        r2 = np.zeros(m); r2[i] = -1
        G_rows.extend([r1, r2])
        h_list.extend([R_max, R_max])

    G = matrix(np.vstack(G_rows))
    h = matrix(np.array(h_list))
    return G, h

# Sweep   and compute accuracy
# For each lambda in lambdas, solve the IRL LP and compute policy accuracy.
# Returns (accuracies, best_lambda, best_accuracy).
def sweep_lambda(P: np.ndarray, G: matrix, h: matrix,
                 expert_pi: np.ndarray, gamma: float, epsilon: float,
                 lambdas: np.ndarray) -> tuple:

    n = P.shape[0]
    accuracies = np.zeros_like(lambdas)
    best_acc, best_lam = -1.0, None

    # solver options
    solvers.options['show_progress'] = False
    solvers.options['abstol'] = 1e-7
    solvers.options['reltol'] = 1e-6
    solvers.options['feastol']= 1e-7

    for i, lam in enumerate(tqdm(lambdas, desc="  sweep")):
        # Objective c for minimize: -sum(t) +   sum(u)
        c = np.concatenate([np.zeros(n), -np.ones(n), lam * np.ones(n)])
        sol = solvers.lp(matrix(c), G, h)
        x = np.array(sol['x']).flatten()
        R_hat = x[:n]
```

```
        # Recover policy and compute accuracy
        V_hat = value_iteration(P, R_hat, gamma, epsilon)
        pi_hat = extract_optimal_policy(P, R_hat, V_hat, gamma)
        acc = np.mean(pi_hat == expert_pi)
        accuracies[i] = acc
        if acc > best_acc:
            best_acc, best_lam, best_R = acc, lam, R_hat

    return accuracies, best_lam, best_acc, best_R

# Main execution
if __name__ == '__main__':
    # Build MDP and expert policy
    P = build_transition_matrix(grid_size, w)
    R = build_reward_function2(grid_size)
    R_max = np.max(np.abs(R))
    V_opt = value_iteration(P, R, gamma, epsilon)
    expert_pi = extract_optimal_policy(P, R, V_opt, gamma)

    # Build constraints
    G, h = build_irl_constraints(P, expert_pi, gamma, R_max)

    # Swep lambda
    lambdas = np.linspace(0, 5, 500)
    accuracies, best_lambda, best_acc, _ = sweep_lambda(
        P, G, h, expert_pi, gamma, epsilon, lambdas
    )

    # Plot results
    plt.figure(figsize=(8,5))
    plt.plot(lambdas, accuracies, linewidth=2)
    plt.xlabel(r'$\lambda$')
    plt.ylabel('Accuracy')
    plt.title('IRL Accuracy vs  ')
    plt.grid(True)
    plt.tight_layout()
    plt.show()

    print(f"Best   = {best_lambda:.3f} with accuracy = {best_acc:.3f}")
```

```
sweep: 100%|      | 500/500 [00:40<00:00, 12.35it/s]
```




```
Best   = 0.571 with accuracy = 0.830
```

19 Question 19

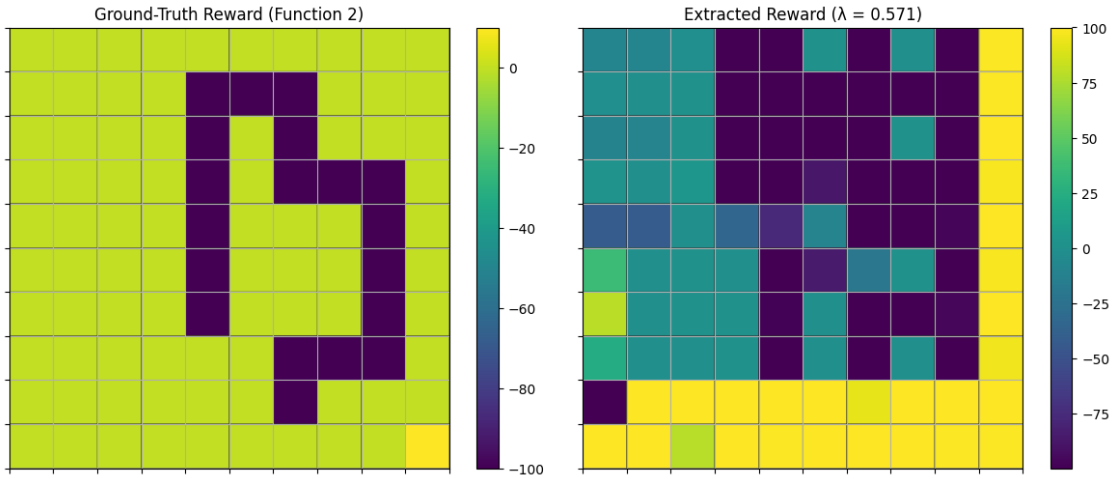
[14]:
```
idx = np.argmax(accuracies)
lambda2_max = lambdas[idx]
print(f" ^(2)_max = {lambda2_max:.3f}")
```

```
 ^(2)_max = 0.571
```

20 Question 20

[15]:
```
# --- Parameters from earlier ---
grid_size = 10
w = 0.1
gamma = 0.9
epsilon = 0.01

# Build MDP and ground-truth reward
P = build_transition_matrix(grid_size, w)
R_gt = build_reward_function2(grid_size)
R_max = np.max(np.abs(R_gt))

# Compute expert policy (Question 8)
```

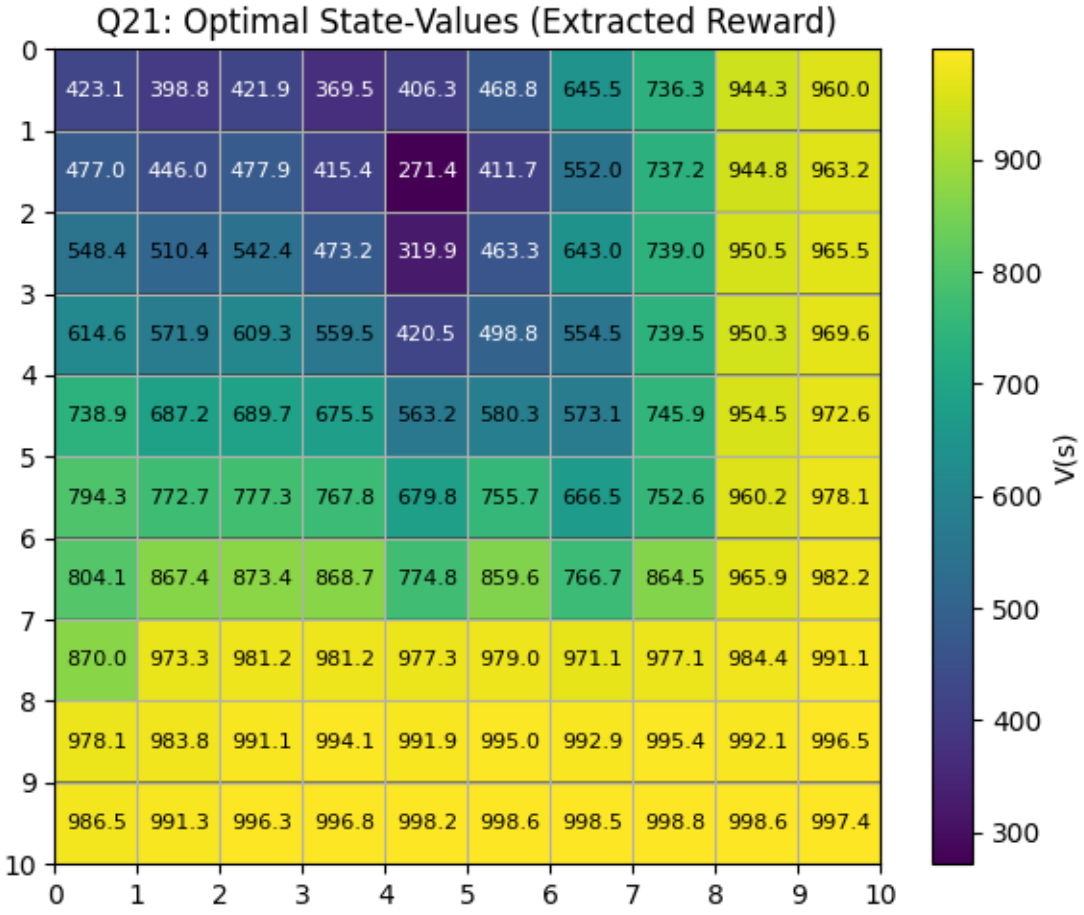
```
V_opt = value_iteration(P, R_gt, gamma, epsilon)
expert_pi = extract_optimal_policy(P, R_gt, V_opt, gamma)

# Build IRL constraints once
G, h = build_irl_constraints(P, expert_pi, gamma, R_max)

# Solve LP at lambda2_max
n = grid_size * grid_size
# objective c = [0_n ; -1_n ; lambda2_max * 1_n] for x = [R; t; u]
c = np.concatenate([np.zeros(n), -np.ones(n), lambda2_max * np.ones(n)])
solvers.options['show_progress'] = False
sol = solvers.lp(matrix(c), G, h)
x = np.array(sol['x']).flatten()
R_extracted = x[:n]    # first n entries are the recovered reward

# Plot two heat-maps
fig, axs = plt.subplots(1,2, figsize=(12,5))

# Ground truth
im0 = axs[0].pcolor(R_gt.reshape((grid_size,grid_size)), edgecolors='k')
axs[0].set_title("Ground-Truth Reward (Function 2)")
fig.colorbar(im0, ax=axs[0])

# Extracted
im1 = axs[1].pcolor(R_extracted.reshape((grid_size,grid_size)), edgecolors='k')
axs[1].set_title(f"Extracted Reward (  = {lambda2_max:.3f})")
fig.colorbar(im1, ax=axs[1])

for ax in axs:
    ax.set_aspect('equal')
    ax.invert_yaxis()
    ax.set_xticks(np.arange(grid_size+1))
    ax.set_yticks(np.arange(grid_size+1))
    ax.set_xticklabels([]); ax.set_yticklabels([])
    ax.grid(True)

plt.tight_layout()
plt.show()
```



21 Question 21

[16]:
```
# Run value iteration on the extracted reward
V_extracted = value_iteration(P, R_extracted, gamma, epsilon)

# Reshape into a 10×10 grid
grid_size = 10
V_grid = V_extracted.reshape((grid_size, grid_size))

# Plot a heat-map
import matplotlib.pyplot as plt

plt.figure(figsize=(6,5))
plt.pcolor(V_grid, edgecolors='k', cmap='viridis')    # any cmap you like
plt.title('Q21: Optimal State-Values (Extracted Reward)')
plt.colorbar(label='V(s)')

for r in range(grid_size):
    for c in range(grid_size):

        val = V_grid[r, c]
        plt.text(c + 0.5, r + 0.5, f'{val:.1f}',
                 ha='center', va='center',
                 color='white' if val < V_grid.max()/2 else 'black',
                 fontsize=8)

plt.gca().invert_yaxis()
plt.xticks(np.arange(grid_size + 1))
plt.yticks(np.arange(grid_size + 1))
```

```
plt.grid(True)
plt.tight_layout()
plt.show()
```



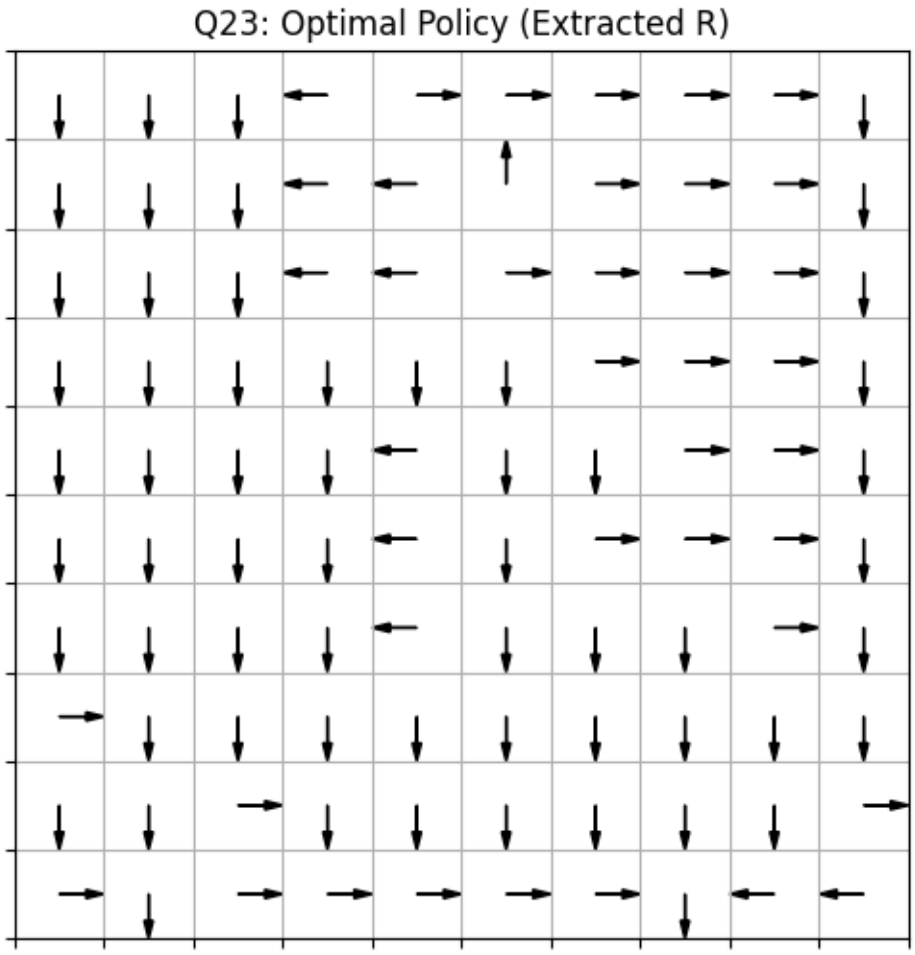
22 Question 22

Aspect	Q7: Ground-Truth Heatmap	Q21: IRL-Extracted Heatmap
Value Range	~ -1.0 to 4.7	~ 270 to 1000
Gradient Direction	Smooth gradient toward goal (99)	Correct direction toward goal (99)
Penalty Zones	Clear valleys near obstacles	Reflected but less precise
Smoothness	Uniform and natural transitions	Sharp transitions, some oscillations
Goal Region (Top-Right)	Highest values around 99	Exaggerated high values near 99
Edge Handling	Clean transitions along borders	Some edge values inflated or distorted

The extracted state values from IRL (Q21) match the overall structure of the ground-truth values (Q7). They show a gradient toward the goal and very good avoidance of penalties. In contrast with this, hey differ significantly in scale and smoothness. These discrepancies are due to the limitations of linear-programming-based IRL, which captures relative preferences but not exact reward magnitudes. As we can see the scale of the reward's are quite different. Additionally, the smoothness of transition in q21 ouput is much worst. Another problem that arizes is how large awards are near the ture goal state, and they do not follow a gradient of increasing value.

23 Question 23

[17]:
```
# Recompute state-values under R_extracted
V_hat = value_iteration(P, R_extracted, gamma, epsilon)

# Extract the greedy policy ^(s)=argmax   P[s,s ,a]*(R_extracted[s ]+ V_hat[s ])
pi_hat = extract_optimal_policy(P, R_extracted, V_hat, gamma)

# Plot with arrows over the 10×10 grid
plot_policy(pi_hat, grid_size=10, title="Q23: Optimal Policy (Extracted R)")
```



24 Question 24

Feature / Region	Q9: Expert Policy	Q23: Extracted IRL Policy
Global direction flow	Smooth arrows toward goal	Largely preserved
Obstacle avoidance	Clear detours around penalty zones	Mostly matches, but less precise
Corners (esp. bottom)	Arrows steer clearly toward goal	Some arrows misdirect (e.g., left/down loops)
Borders/Edges	Arrows point toward goal	Sometimes point off-grid or loop in place
Goal region behavior	Arrows clearly converge into goal (99)	Mostly correct, but with jitter near target

The policy from Question 23 seems to follow the same trajectory as the expert policy provided in Question 9 in terms of its overarching direction and aim inclination. Nonetheless, a number of local inconsistencies remain, most markedly towards the edges and bottom-left, which can be attributed to the local rewards' ambiguity coupled with errors in estimative IRL computations. In general, the larger framework is maintained, although the intricacies differ more near the obstacles and along the grid borders.

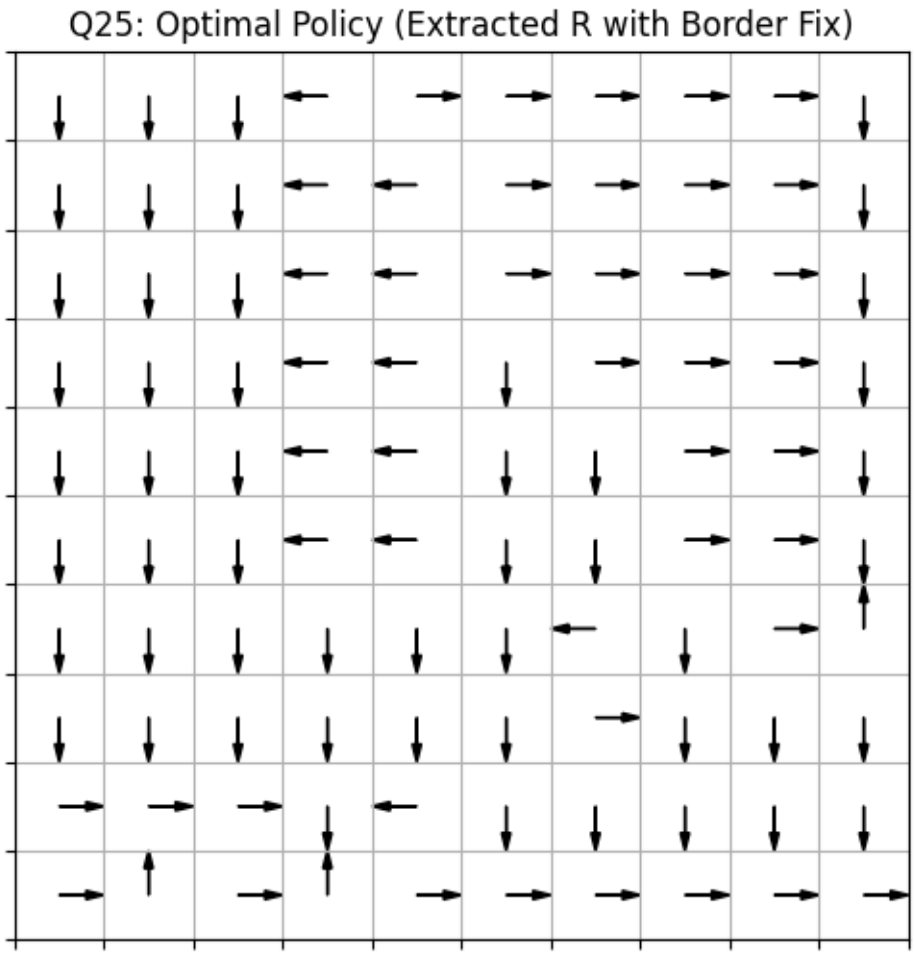
25 Question 25

[12]:
```
def value_iteration_with_border_fix(P, R, gamma=0.8, eps=1e-2, grid_size=10,␣
↪max_iter=1000):
    n = len(R)
    V = np.zeros(n)

    for _ in range(max_iter):
        Vn = np.zeros_like(V)
        for s in range(n):
            Qs = []
            for a in range(4):
                q_sa = np.dot(P[s,:,a], R + gamma * V)
                Qs.append(q_sa)
            Vn[s] = max(Qs)
        if np.max(np.abs(Vn - V)) < eps:
            break
        V = Vn
    return V

def extract_policy_with_border_fix(P, R, V, gamma=0.8, grid_size=10):
    n = len(R)
```

```
    pi = np.zeros(n, dtype=int)
    for s in range(n):
        r, c = divmod(s, grid_size)
        Qs = [np.dot(P[s,:,a], R + gamma * V) for a in range(4)]

        # Only apply constraint if not at goal state
        if s != (grid_size * grid_size - 1):
            if c == grid_size - 1: Qs[0] = -np.inf  # right
            if c == 0:             Qs[1] = -np.inf  # left
            if r == 0:             Qs[2] = -np.inf  # up
            if r == grid_size - 1: Qs[3] = -np.inf  # down

        pi[s] = int(np.argmax(Qs))
    return pi

if __name__ == '__main__':
    import matplotlib.pyplot as plt

    grid_size = 10
    w = 0.1
    gamma = 0.8
    epsilon = 1e-2

    # Build MDP and true reward function (Reward Function 2)
    P = build_transition_matrix(grid_size, w)
    R_true = build_reward_function2(grid_size)
    R_max = np.max(np.abs(R_true))

    # Expert policy using ground-truth reward
    V_expert = value_iteration_with_border_fix(P, R_true, gamma, epsilon,␣
 ↪grid_size)
    expert_pi = extract_policy_with_border_fix(P, R_true, V_expert, gamma,␣
 ↪grid_size)

    # IRL: Build constraints and sweep lambda
    G, h = build_irl_constraints(P, expert_pi, gamma, R_max)
    lambdas = np.linspace(0, 5, 500)

    # Use existing sweep function
    accuracies, best_lambda, best_acc, best_R = sweep_lambda(P, G, h,␣
 ↪expert_pi, gamma, epsilon, lambdas)

    print(f"Best   = {best_lambda:.3f} | Accuracy BEFORE Fix: {best_acc:.3f}")

    # Re-run value iteration with border constraints
    V_agent = value_iteration_with_border_fix(P, best_R, gamma, epsilon,␣
 ↪grid_size)
```

```
    pi_agent = extract_policy_with_border_fix(P, best_R, V_agent, gamma,␣
↪grid_size)

    # Compute updated accuracy after applying border fix
    final_acc = np.mean(pi_agent == expert_pi)
    print(f"Final Accuracy AFTER border fix: {final_acc:.3f}")

    # Plot policy after fix
    plot_policy(pi_agent, grid_size=grid_size, title="Q25: Optimal Policy␣
↪(Extracted R with Border Fix)")
```

```
  sweep: 100%|      | 500/500 [00:42<00:00, 11.83it/s]
Best   = 0.571 | Accuracy BEFORE Fix: 0.830
Final Accuracy AFTER border fix: 0.870
```



Discrepancy	Description	Cause	Fix Method	Accuracy Impact
1. Border Actions Point Off-grid	Arrows at edge states (e.g. rightmost column, bottom row) point off the grid or "loop back"	The agent prefers actions that return to self (due to high transition probability) even when neighboring values are higher	Apply a border mask during **policy extraction** to set off-grid Q-values to `-inf`, preventing selection	Accuracy ↑ from **0.830** to **0.870**
2. Confused Local Behavior	Policy near penalty zones oscillates or loops	IRL produces **spiky or inconsistent rewards** due to overfitting to expert behavior without smoothness; local rewards overpower global structure	Not solvable with standard LP-IRL; would require regularization or switching to **MaxEnt IRL**	No direct accuracy improvement with this fix

The first discrepancy appears because of a flaw in the policy extraction logic which allowed agents to select off-grid actions, like moving right from the rightmost column, which granted them the same state, as a potential option. Because of the construction of the MDP with wind and self-loop probabilites, these off-grid options sometimes led to higher Q-values than moving toward the goal. We addressed this restriction by masking these actions with -inf, which eliminates them from the decision argmax and creates a more accurate enhancement toward the goal.

This second discrepancy is more subtle. While extracting the IRL rewards, the algorithm can generate rather noisy or contradictory local gradients, particularly bordering dense penalty areas. This causes suboptimal looping policy problems, where the agent gets trapped in almost identical value states, resulting in oscillating motion. This requires changes in reward smoothening, regularizing, or shifting to Maximum Entropy IRL which takes in account policy stochasticity and uncertainty. This is something our current LP-based method does not address.

Additionally, through analyzing our q21 heatmap a bit further we can see some of the origins of these issues directly. For example, the value placed on sqaures on the bottom of the map and the right side are extremely high but all very similar. This makes it difficult for the policy to select and upate its next action and state (again leading to failed feedback loops that can be hard to escape). Here, we would hope to see a very large value on the end state and reasonable trasnitioning values on others but this is not the case.