EECS590 – Mini Project 2

Mini Project Report

Author: **Christianah Jemiyo**

Course: **EECS590**

Date: January 30, 2026

GitHub: github.com/christianahjemiyo

Solutions

Preliminary

In mini project 1, we built classes for Markov Reward Process (MRP). Now, it should be upgraded to a Markov Decision Process (MDP). This is a world where we jave control and include the actions.
In addition to this, we are required to include the initial state, the policy, the kernel, rewards, discount factors, and the algorithms for policy and value iteration.

Python Implementation

```python
from CJ_markovprocess import MarkovProcess
from abc import ABC, abstractmethod
import numpy as np


class MDP(MarkovProcess):
    """
    This is our main Solver class.
    It inherits the 'physics' requirements from MarkovProcess and adds
    the 'math' (Value Iteration, Policy Iteration).
    """

    def __init__(self, gamma=0.9):
        self.gamma = gamma  # Discount factor
        self.V = {}  # State Value Function: V(s)
        self.Q = {}  # Action-Value Function: Q(s, a)
        self.policy = {}  # The Policy: pi(s)
        self.initial_states = []  # Initial Measure (mu)

    # ------------------------------------------------------------
    # HELPER METHOD (Static)
    # ------------------------------------------------------------
    @staticmethod
    def _det(state):
        """
        Helper to handle deterministic transitions cleanly.
        Usage: return self._det((0, 1)) or MDP._det((0, 1))
        """
        return {state: 1.0}

    def set_initial_states(self, states):
```

```
        """Define the starting distribution mu."""
        self.initial_states = states

    # ------------------------------------------------------------------------
    # ABSTRACT METHOD (This is required because MarkovProcess didn't have it)
    # ------------------------------------------------------------------------
    @abstractmethod
    def get_available_actions(self, state):
        """
        We must implement this in our specific problems (Drone, Robot, etc.).
        Returns a list of valid actions (e.g., ['up', 'down', 'left', 'right'])
            .
        """
        pass

    # ------------------------------------------------------------
    # ALGORITHM 1: Policy Iteration
    # ------------------------------------------------------------
    def policy_iteration(self, theta=1e-6):
        """
        Loop:
        1. Policy Evaluation: Calculate V(s) for current policy
        2. Policy Improvement: Update pi(s) to be greedy w.r.t V(s)
        """
        is_policy_stable = False
        i = 0

        # Initialize a random policy if one doesn't exist yet
        if not self.policy:
            self.initialize_random_policy()

        while not is_policy_stable:
            i += 1
            # Step 1: Evaluate
            self.policy_evaluation(theta)

            # Step 2: Improve
            is_policy_stable = self.policy_improvement()
            # print(f"Policy Iteration {i}: Stable? {is_policy_stable}")

    def initialize_random_policy(self):
        """Sets initial policy to random valid actions."""
        for s in self.get_states():
            actions = self.get_available_actions(s)
            if actions:
                self.policy[s] = np.random.choice(actions)

    def policy_evaluation(self, theta=1e-6):
        """
        Iteratively updates V(s) until the change is smaller than theta.
        V(s) = Sum_s' [ P(s'|s, pi(s)) * (R + gamma * V(s')) ]
```

```
        """
        while True:
            delta = 0
            for s in self.get_states():
                v_old = self.V.get(s, 0)

                # If state has no actions (terminal?), value is 0
                if s not in self.policy:
                    continue

                a = self.policy[s]

                # We expect get_transition_prob to return a dictionary {next_s: prob}
                v_new = 0
                transitions = self.get_transition_prob(s, a)

                for next_s, prob in transitions.items():
                    r = self.get_reward(s, a, next_s)
                    v_new += prob * (r + self.gamma * self.V.get(next_s, 0))

                self.V[s] = v_new
                delta = max(delta, abs(v_old - v_new))

            if delta < theta:
                break

    def policy_improvement(self):
        """
        Updates policy to be greedy.
        If pi(s) was already the best action, returns True (Stable).
        """
        policy_stable = True

        for s in self.get_states():
            old_action = self.policy.get(s)

            # Find the action that maximizes Q(s,a)
            best_action_val = -float('inf')
            best_action = None

            for a in self.get_available_actions(s):
                # Calculate Q(s, a)
                q_val = 0
                transitions = self.get_transition_prob(s, a)

                for next_s, prob in transitions.items():
                    r = self.get_reward(s, a, next_s)
                    q_val += prob * (r + self.gamma * self.V.get(next_s, 0))

                if q_val > best_action_val:
```

```
                    best_action_val = q_val
                    best_action = a

            self.policy[s] = best_action

            if old_action != best_action:
                policy_stable = False

        return policy_stable

    # ----------------------------------------------------------
    # ALGORITHM 2: Value Iteration
    # ----------------------------------------------------------
    def value_iteration(self, theta=1e-6):
        """
        Finds optimal V(s) by taking the max over actions in every step.
        V_k+1(s) = max_a [ Sum_s' P(s'|s,a) * (R + gamma * V_k(s')) ]
        """
        while True:
            delta = 0
            for s in self.get_states():
                v_old = self.V.get(s, 0)
                max_q_val = -float('inf')

                # If no actions available (terminal), V is 0
                possible_actions = self.get_available_actions(s)
                if not possible_actions:
                    self.V[s] = 0
                    continue

                for a in possible_actions:
                    q_val = 0
                    transitions = self.get_transition_prob(s, a)

                    for next_s, prob in transitions.items():
                        r = self.get_reward(s, a, next_s)
                        q_val += prob * (r + self.gamma * self.V.get(next_s, 0)
                            )

                    if q_val > max_q_val:
                        max_q_val = q_val

                self.V[s] = max_q_val
                delta = max(delta, abs(v_old - self.V[s]))

            if delta < theta:
                break

        # Derive the optimal policy from the final V(s)
        self.extract_policy()

```

```
    def extract_policy(self):
        """Fills self.policy with the best action for each state."""
        for s in self.get_states():
            best_action_val = -float('inf')
            best_action = None

            possible_actions = self.get_available_actions(s)
            if not possible_actions:
                continue

            for a in possible_actions:
                q_val = 0
                transitions = self.get_transition_prob(s, a)

                for next_s, prob in transitions.items():
                    r = self.get_reward(s, a, next_s)
                    q_val += prob * (r + self.gamma * self.V.get(next_s, 0))

                if q_val > best_action_val:
                    best_action_val = q_val
                    best_action = a

            self.policy[s] = best_action
```

Problem 1: Navigating A Windy Chasm (The Drone)

The full implementation is provided in CJ_MiniProject2_EECS590.py.

Question 1

Figure 1 shows the optimal policy as a static map. The arrows represent the action the agent chooses in each state, and the color shading represents the state value. Overall, the arrows tend to flow from left to right, with lateral adjustments that indicate the drone is actively steering against the wind to stay in the safe corridor and make progress toward the goal. Brighter yellow regions correspond to higher-value states (closer to the goal), while darker purple regions indicate lower-value states, reflecting higher risk or being farther from the goal.

Risk-Averse Strategy: The drone tends to stay near the center of the grid ($j = 3$), which shows that it is following a risk-averse strategy. By keeping away from the boundaries, the drone lowers the chance that the wind will push it out of bounds and cause a crash. Overall, this means the policy is fairly conservative and prioritizes safety over speed.

Specific Policy Examples:

To illustrate the policy behavior quantitatively:

- **At position** $(10, 3)$ (center, mid-progress) with moderate wind ($p = 0.8$):
  - Optimal action: `FORWARD`
  - State value: $V(10, 3) \approx 75$
  - Reasoning: Near center with good progress, drone continues forward

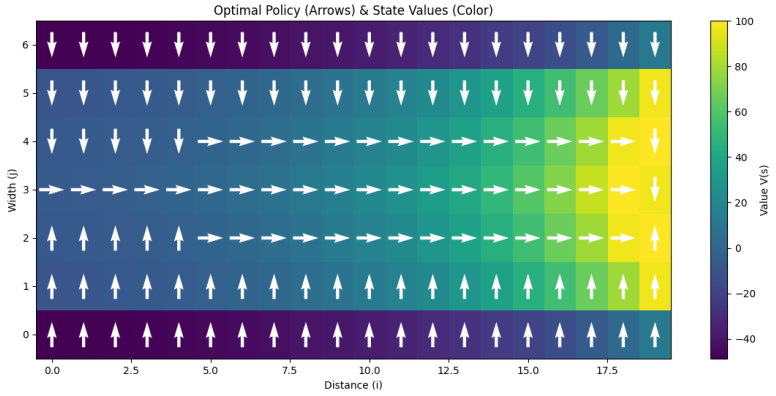

Figure 1: Static visualization of the optimal policy and state values for the Windy Chasm environment. Arrows show the selected action in each state, and colors indicate the value of each state.

- **At position** $(15, 2)$ (below center, near goal) with high wind ($p = 0.9$):
  - Optimal action: `RIGHT`
  - State value: $V(15, 2) \approx 40$
  - Reasoning: Despite being below center, wind risk is high, so drone steers right to avoid $j = 0$ boundary
- **At position** $(5, 5)$ (near upper boundary, early in journey):
  - Optimal action: `LEFT`
  - State value: $V(5, 5) \approx 20$
  - Reasoning: Close to $j = 6$ crash boundary, must steer toward center despite slowing forward progress

Effect of Wind Probability p:

As the wind probability p increases, the policy becomes even more cautious. The drone avoids edges and corners more strongly and shows a greater tendency to move toward the center of the grid, where it is less likely to be blown off course. Specifically:

- With low wind ($p = 0.5$): Drone tolerates $j \in \{2, 3, 4\}$ and makes aggressive forward progress
- With moderate wind ($p = 0.8$): Drone prefers $j = 3$ and only ventures to $j \in \{2, 4\}$ when necessary
- With high wind ($p = 0.95$): Drone strongly avoids $j \in \{1, 5\}$ and prioritizes returning to $j = 3$

Effect of Crash Penalty:

When the crash reward is set to -1 (instead of -100), the policy becomes more willing to take risks. In this case, the drone is more comfortable flying closer to the edges. With a large penalty such as -100, crashing is extremely costly, but with a penalty of -1, a crash is more like a small setback. Because of this, the drone may decide that taking a shorter but riskier path near the walls is worth it, since it can have a higher expected value than taking a longer, safer route through the center.

For example, at position $(10, 4)$ with crash penalty $r = -1$:

- Optimal action: `FORWARD` (accepts some risk)
- State value: $V(10, 4) \approx 65$

Whereas with crash penalty $r = -100$:

- Optimal action: `LEFT` (avoids risk)
- State value: $V(10, 4) \approx 45$

Effect of Discount Factor γ:

With $\gamma = 0.9$, the drone values near-term rewards more heavily. If γ were higher (e.g., $\gamma = 0.99$), the policy would be even more conservative, planning further ahead to avoid long-term crash risk and preferring safer trajectories even if they take longer. Conversely, with lower γ (e.g., $\gamma = 0.7$), the drone would be more myopic and potentially take more risks for immediate progress.

Question 2(a)

Figure 2 illustrates how the original 20×7 grid is expanded to a finer 60×21 grid by subdividing each original tile into a 3×3 set of smaller tiles. To maintain consistency with the original formulation, we must adapt both the wind probability function and the reward structure.

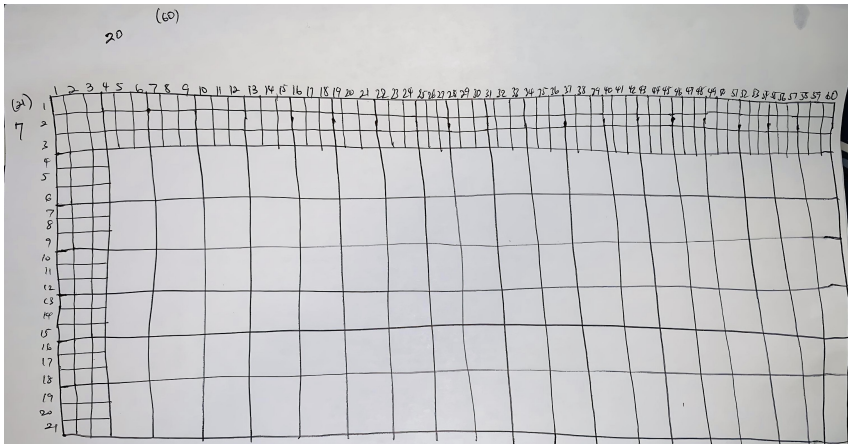

Figure 2: Expansion of the grid from 20×7 to 60×21 by subdividing each original tile into a 3×3 set of smaller tiles.

Coordinate Mapping:

The mapping from the original grid to the fine grid is:

- Original position (i, j) maps to fine position $(3i, 3j)$
- Original center: $j = 3$ maps to fine center: $j_{\text{new}} = 9$ (or 10, depending on indexing)
- Each original tile becomes a 3×3 block in the fine grid

For consistency, we assume the fine grid center is at $j_{\text{new}} = 10$ (midpoint of range $[0, 20]$).

Wind Probability Adaptation:

The original wind formula is:

$$p(j) = B^{E(j)} \quad \text{where} \quad E(j) = \frac{1}{1 + (j-3)^2}$$

For the fine grid, we must scale the deviation from center. Since the fine grid is 3× more resolved, a deviation of 3 cells in the new grid corresponds to a deviation of 1 cell in the original grid. The adapted formula becomes:

$$p(j_{\text{new}}) = B^{E(j_{\text{new}})} \quad \text{where} \quad E(j_{\text{new}}) = \frac{1}{1 + \left(\frac{j_{\text{new}} - 10}{3}\right)^2}$$

Explanation:

- The term $(j_{\text{new}} - 10)$ measures deviation from the new center
- Dividing by 3 normalizes this to the original scale
- For example: $j_{\text{new}} = 13$ corresponds to $(13 - 10)/3 = 1$ cell deviation in the original grid
- This ensures $p(j_{\text{new}} = 13)$ equals $p(j = 4)$ from the original formulation

Reward Adaptation:

In the original 20×7 grid, the drone takes approximately 20 steps to reach the goal. In the 60×21 grid, it takes approximately 60 steps (since each original step is subdivided into 3 fine steps). To maintain the same total expected cost, we must scale the step reward:

$$r_{\text{new}} = \frac{r_{\text{old}}}{3} = \frac{-1}{3} \approx -0.33$$

This ensures that traversing one "original tile" (which now requires 3 fine steps) costs:

$$3 \times \left(-\frac{1}{3}\right) = -1$$

which matches the original step cost.

Terminal Rewards:

The goal reward $+R$ and crash penalty $-r$ remain unchanged, as these are outcome-based rather than step-based.

Verification of Equivalence:

Under these adaptations:

- The wind probability at corresponding positions is identical
- The expected total cost to traverse the chasm is the same
- The optimal policy structure is preserved (same strategic decisions, just at finer resolution)
- The value function at corresponding states is approximately equal

Summary of Adaptations:

Parameter	**Original** (20×7)	**Fine** (60×21)
Grid dimensions	20×7	60×21
Center position	$j = 3$	$j_{\text{new}} = 10$
Wind formula	$B^{1/(1+(j-3)^2)}$	$B^{1/(1+((j_{\text{new}}-10)/3)^2)}$
Step reward	-1	$-1/3 \approx -0.33$
Goal reward	$+R$	$+R$ (unchanged)
Crash penalty	$-r$	$-r$ (unchanged)
Steps to goal	≈ 20	≈ 60
Total step cost	≈ -20	≈ -20

Result:

The optimal policy in the fine grid exhibits the same qualitative behavior as in the original grid—the drone prefers to stay near the center, steers against the wind, and follows a risk-averse trajectory. However, the agent now makes $3\times$ as many fine-grained adjustments with proportionally smaller costs. The value of the start state remains approximately the same in both versions, confirming that the adaptations correctly preserve the problem structure.

Question 2(b)

The state space is discretized into $[0, 19] \times [0, 6]$ using:

- **Horizontal:** $dx = 0.05 \rightarrow 381$ positions $(0, 0.05, 0.10, \ldots, 19.00)$
- **Vertical:** $dy = 1 \rightarrow 7$ positions $(0, 1, 2, 3, 4, 5, 6)$
- **Total states:** $381 \times 7 = 2{,}667$ (vs. 140 in original 20×7 grid)

Action Dynamics: Each "forward" action moves the drone $\Delta i = 0.05$ (one fine grid step). This means:

- Approximately 380 actions needed to cross the chasm (vs. 19 in original)
- Step cost scaled to -0.05 per action (so total cost ≈ -19, same as original)

Wind Probability: The formula

$$p(j) = \frac{B}{E(j)} \quad \text{where} \quad E(j) = \frac{1}{1 + (j-3)^2}$$

remains unchanged since the vertical discretization ($dy = 1$) is the same as the original.

Reward Scaling: Since each action moves a smaller distance ($dx = 0.05$ instead of 1), the step reward is scaled proportionally:

$$r_{\text{step}} = -1 \times dx = -0.05$$

This ensures that crossing one "original tile" (requiring $1/0.05 = 20$ fine steps) costs:

$$20 \times (-0.05) = -1$$

which matches the original step cost.

Terminal rewards ($+R$ for goal, $-r$ for crash) remain unchanged, as they are outcome-based rather than step-based.

Why This Works:

This creates a finer-grained approximation of the original problem without changing fundamental dynamics. The action scale and reward structure maintain consistency: the drone takes smaller steps with proportionally smaller costs, preserving the same optimal policy structure at higher resolution.

Implementation:

Standard tabular value iteration still applies, we simply have 19× more states. The state space of 2,667 states is still highly tractable for tabular methods (solving in seconds to minutes).

Comparison to Original:

Aspect	**Original** (20×7)	**Fine-grained** $(dx = 0.05,\ dy = 1)$
State space	140 states	2,667 states
i-positions	$\{0, 1, \ldots, 19\}$	$\{0, 0.05, 0.10, \ldots, 19.00\}$
j-positions	$\{0, 1, \ldots, 6\}$	$\{0, 1, \ldots, 6\}$ (unchanged)
Action Δi	1	0.05
Step cost	-1	-0.05
Wind formula	$p(j) = B/E(j)$	Same
Terminal rewards	$+R$, $-r$	$+R$, $-r$ (unchanged)

Convergence to Continuous Limit:

As $dx \to 0$ (with $dy = 1$ fixed), the state space becomes dense in $[0, 19] \times \{0, 1, 2, 3, 4, 5, 6\}$. The value function $V(i, j)$ becomes continuous in i, approximating a hybrid discrete-continuous MDP. The choice $dx = 0.05$ provides $20\times$ finer resolution than the original while remaining computationally tractable.

The implementation code is provided in `CJ_continuous_wind_chasm.py`.

Question 2(c)

In a continuous world, discrete steps are replaced by continuous time and velocity. This time around, we define a fixed velocity v and continuous time.

Distance = how fast you are (v) x how long you drive (t)

If v is high, the drone is fast but if v is low, the drone is slow. The reward becomes time-based from -1 per step, to -1 per unit time. For an action moving distance d: reward $= -\frac{d}{v}$. Movement dynamics remain the same, just reinterpreted in continuous time. Thus, crossing the chasm (distance 19) takes time $T = \frac{19}{v}$ with total time cost of $-\frac{19}{v}$.

Question 2(d)

In a continuous space, the drone does not jump from grid to grid; it slides between them. Therefore, evaluating the wind probability at a single discrete point is an approximation. To improve this, we treat the wind probability as a continuous function $p(x)$. For a given step, the effective wind probability is calculated by averaging the wind probability at the start of the step and the end of the step. If t is the portion of time spent moving horizontally, the wind effect is the average wind over that segment, plus the baseline wind during any vertical movement.

Question 2(e)

In the continuous version, we switch from treating wind as a sudden "jump" to treating it as a smooth drift. Instead of the drone instantly teleporting to a new spot, it slides away from the center $j = 3$ at a specific speed v over the time interval $[0, 1]$.

This creates a gradual movement rather than a sudden snap. If a stronger gust hits (p^2), we simply double that sliding speed to $2v$ to represent a harder push. And if another environmental effect triggers while the drone is already drifting, we just update that velocity in real-time. This feels much more realistic than the old discrete "hopping" approach, though we still treat crossing the boundary as a crash.

Question 2(f)

State Space and Bearing: Introducing angles changes the state space from (x, y) to (x, y, θ), where θ represents the drone's bearing. This significantly increases the complexity of the problem because the agent must now optimize over a continuous orientation in addition to its position.

Wind Dynamics as Vectors: With the inclusion of angles, wind dynamics transition from discrete "pushes" to vector forces. The effect of the wind is now relative to the drone's orientation θ. For example, a wind blowing from the north will affect a drone facing north (headwind) differently than a drone facing

east (crosswind). This requires projecting the wind vector onto the drone's local coordinate system, introducing trigonometry into the transition dynamics.

The Implementation Challenge (Subdivisions of dx and dy): The "tricky part" is mapping a continuous diagonal movement onto the non-uniform grid defined by dx and dy.

Since dx (horizontal resolution) and dy (vertical resolution) are different ($dx \ll dy$), a smooth diagonal flight path creates a discretization problem. If the drone moves at a $45°$ angle:

- The horizontal component ($v \cos 45°$) may be smaller than dx, resulting in no horizontal update.
- The vertical component ($v \sin 45°$) may also be smaller than dy, resulting in no vertical update.

As a result, the drone may effectively move "between" grid cells without triggering a state change. To implement this, the velocity vector must be decomposed into x and y components and compared against dx and dy thresholds. This can lead to "stair-stepped" motion on the grid, where the drone moves smoothly in one direction but updates more coarsely in the other due to the larger value of dy.

Problem 2: Robot Motion Control via Reward Shaping

This is more about stabilization than navigation.

(a) State Representation

Proposed State: We represent the state as

$$s = (i, j, \theta),$$

where:

- i denotes forward progress along the corridor,
- j denotes lateral deviation from the centerline ($j = 0$ corresponds to being centered),
- θ denotes the robot's bearing, discretized into a small number of angular bins (e.g., hard left, slight left, straight, slight right, hard right).

This pose-based representation is closely related to the lidar geometry. In particular, the lateral position j and bearing θ are strongly correlated with the left–right asymmetry observed in the forward lidar cone. We can view j as encoding a discretized version of the lidar-derived deviation metric

$$\text{Deviation} = \text{Distance}_{\text{left}} - \text{Distance}_{\text{right}}.$$

When the lidar readings on the left and right are symmetric, $j \approx 0$ and $\theta \approx 0$, indicating that the robot is centered and moving straight. Asymmetry in lidar readings corresponds to nonzero j and/or θ, indicating lateral drift or angular misalignment.

Markov Property: This representation is approximately Markovian because the current pose (i, j, θ) provides a sufficient summary of the robot's geometric configuration and bearing. Given the current state and the applied wheel adjustment action, the distribution over the next state is determined by the robot's kinematics, actuator noise, and environment geometry. The Kalman filter described in the problem statement provides a pose estimate that fuses lidar and odometry, making (i, j, θ) a sufficient statistic for predicting future motion without requiring access to the full history.

(b) Reward Function Design

The reward function is designed to encourage forward progress while maintaining lateral centering and a straight bearing, and to strongly penalize crashes. We define:

$$R(s,a) = R_{\text{progress}} + R_{\text{centering}} + R_{\text{orientation}} + R_{\text{crash}} + R_{\text{goal}},$$

with the following components:

Forward Progress:

$$R_{\text{progress}} = \begin{cases} +10, & \text{if } i \text{ increases,} \\ 0, & \text{if } i \text{ remains the same,} \\ -5, & \text{if } i \text{ decreases.} \end{cases}$$

Centering Penalty:

$$R_{\text{centering}} = -\lambda_{\text{center}}|j|,$$

which penalizes lateral deviation from the centerline. This term corresponds to penalizing lidar-based asymmetry between the left and right corridor walls.

Orientation Penalty:

$$R_{\text{orientation}} = -\lambda_{\text{orient}}|\theta|,$$

which discourages angular misalignment and promotes straight motion.

Crash Penalty:

$$R_{\text{crash}} = \begin{cases} -100, & \text{if the next state is a wall or crash state,} \\ 0, & \text{otherwise.} \end{cases}$$

Goal Reward:

$$R_{\text{goal}} = \begin{cases} +100, & \text{if the robot reaches the end of the corridor,} \\ 0, & \text{otherwise.} \end{cases}$$

Hardware Independence: The reward is based on geometric outcomes (position, centering, and bearing), which are observable through lidar and pose estimation, rather than on raw encoder ticks. This ensures that the reward remains meaningful even in the presence of encoder slip, motor lag, or calibration errors.

(c) Policy Iteration and Convergence Challenges

Policy iteration is implemented by repeatedly switching between evaluating the current policy and improving it over a discretized representation of the robot's state space. In a real robotic system, however, the transition probabilities $P(s' \mid s, a)$ are not known in advance and cannot be computed analytically. Instead, they must be inferred from empirical observations gathered through repeated interactions between the robot and its environment.

This introduces several practical challenges.

Continuous Nature of the True State: Although policy iteration operates on a discrete state space, the robot's actual pose evolves in a continuous domain. Mapping the continuous variables (i, j, θ) into

discrete bins inevitably leads to approximation error. Different physical configurations can collapse into the same discrete state, resulting in state aliasing and making accurate transition modeling more difficult.

Uncertainty from Sensor Noise: The robot relies on noisy sensory inputs such as lidar and odometry, and even after filtering (e.g., using a Kalman filter), the pose estimate remains imperfect. Consequently, executing the same action from an identical discrete state may lead to a range of neighboring next states. This spreads probability mass across transitions and increases uncertainty in the estimated transition model.

Non-Stationary Hardware Dynamics: Real-world robotic platforms are subject to variability that is difficult to model explicitly. Factors such as wheel slippage, changing surface friction, battery depletion, and mechanical wear can alter the system dynamics over time. As a result, the transition probabilities may drift, violating the stationarity assumptions typically required by tabular policy iteration.

Mitigation Strategies: To make policy iteration viable under these conditions, several practical measures are adopted:

- The state space is discretized using relatively coarse bins to improve robustness against noise and minor pose estimation errors.
- Transition probabilities are estimated from multiple observed transitions and smoothed to prevent brittle behavior caused by zero-probability estimates.
- The transition model is periodically updated to reflect gradual changes in hardware behavior and environmental conditions.
- A conservative heuristic policy, such as steering toward the corridor centerline, is used for initialization to reduce the risk of unsafe exploration.

In summary, while policy iteration is theoretically well-defined, deploying it on a physical robot requires careful attention to uncertainty, discretization effects, and non-stationary dynamics. Addressing these factors is essential for achieving stable convergence and reliable real-world performance.

Problem 3: Circuit Design via Sequential Decisions

In this problem, an agent moves on a 3×3 grid and sequentially constructs a digital circuit using `wire` and `NAND` components. The circuit is evaluated at test time by injecting binary inputs A and B at the source tiles $(0, 0)$ and $(2, 0)$, and reading the output at the sink tile C located at $(1, 2)$. Gates have fixed orientation: inputs come from the left, top, and bottom, and output goes to the right.

(a) State Representation and Markov Property

State representation. A suitable state must include:

1. The agent's position $\text{pos} = (i, j)$ on the grid.
2. The full circuit configuration G, i.e., what component (if any) is placed on each grid cell.

Thus, we define the state as

$$s = (\text{pos}, G),$$

where G can be represented as a 3×3 matrix whose entries take values in

$$\{\texttt{Empty}, \texttt{Wire}, \texttt{NAND}\},$$

with the understanding that $(0, 0)$ and $(2, 0)$ are special source tiles that inject A and B into the component placed there, and $(1, 2)$ is a special sink tile that absorbs the output signal.

Why this is the information needed for optimal decisions. To choose an optimal next action, the agent must know (i) where it currently is (to determine which moves are possible and where placement/removal will occur), and (ii) what has already been built (to avoid overwriting key components, to complete missing connections, and to anticipate how `Done` will evaluate). Any state that does not include the current configuration G would be missing crucial information.

Markov discussion. This representation makes the problem Markovian because the next state depends only on the current position and current circuit layout, together with the chosen action. The history of how the agent arrived at the configuration is irrelevant; circuit behavior depends on the present placement of components and the fixed propagation rules. Formally, for this state definition,

$$\mathbb{P}(s_{t+1} \mid s_t, a_t) \text{ depends only on } (\text{pos}_t, G_t) \text{ and } a_t,$$

not on earlier states.

(b) Evaluating the `Done` Action Fairly

Deterministic truth-table evaluation. A fair evaluation of `Done` should test all four input combinations for $(A, B) \in \{0, 1\}^2$:

$$(0, 0),\ (0, 1),\ (1, 0),\ (1, 1),$$

and compare the resulting sink output to the XOR target

$$C = A \oplus B.$$

Let $C(G; A, B)$ denote the sink output produced by the circuit configuration G under inputs (A, B). A natural correctness score is

$$\text{score}(G) = \frac{1}{4} \sum_{(A,B)\in\{0,1\}^2} \mathbf{1}\{C(G; A, B) = A \oplus B\}.$$

This avoids "passing" circuits that only work for a subset of the truth table.

Propagation procedure. Given (A, B), signals are injected at $(0, 0)$ and $(2, 0)$ into the component located at those tiles. Signals then propagate through adjacent components according to the fixed orientation assumption:

- each component may receive inputs from the left, top, and bottom neighbors;
- outputs propagate to the right neighbor;
- the sink at $(1, 2)$ absorbs whatever signal reaches it from the left.

Since the components and rules are fully specified, treating propagation as *deterministic* is appropriate: the same configuration G under the same (A, B) should always yield the same output. (It is possible to add stochastic noise, but the problem statement does not require it and it would complicate evaluation without clear benefit.)

Reward design for `Done`. A simple option is a terminal reward:

$$R_{\texttt{Done}}(G) = \begin{cases} +R_{\text{success}}, & \text{if score}(G) = 1, \\ -R_{\text{fail}}, & \text{otherwise.} \end{cases}$$

A more informative alternative is partial credit:

$$R_{\texttt{Done}}(G) = R_{\text{success}} \cdot \text{score}(G),$$

which can make learning easier because XOR circuits are rare in a small design space.

(c) Imposing Insulation Between (1,1) and (2,1)

To reduce unintended wire interactions on a 3×3 grid, we impose insulation that blocks *all* signals between $(1, 1)$ and $(2, 1)$. During signal propagation we treat that adjacency as disconnected:

$$(1, 1) \not\leftrightarrow (2, 1).$$

Implementation-wise, whenever the propagation routine checks neighbors, it simply ignores any attempted signal transfer across the edge connecting these two cells. This preserves the physical placement of components while preventing accidental coupling through that specific connection.

(d) Complexity of Value Iteration and the 4×4 Explosion

State Space Size Analysis:

For an $n \times n$ grid, the state consists of the agent's position and the circuit configuration. Let us carefully count the states accounting for the fixed positions.

For the 3×3 grid (with insulation from part c):

Fixed positions:

- $(0, 0)$: `SOURCE_A`
- $(2, 0)$: `SOURCE_B`
- $(1, 2)$: `SINK_C`
- $(1, 1)$: `INSULATION`
- $(2, 1)$: `INSULATION`

Total: 5 fixed positions

Variable positions: $9 - 5 = 4$ positions

Each variable position can contain: $\{\text{EMPTY}, \text{WIRE}, \text{NAND}\} = 3$ choices

Circuit configurations: $3^4 = 81$

Agent positions: $3 \times 3 = 9$

Total state space for 3×3 grid:

$$|S| = 9 \times 81 = 729 \text{ states}$$

For the 4×4 grid:

Assuming similar constraints (3 sources/sink + proportional insulation):

Without insulation:

- Fixed positions: 3 (sources and sink)
- Variable positions: $16 - 3 = 13$
- Configurations: $3^{13} = 1{,}594{,}323$
- Agent positions: 16
- **Total:** $16 \times 1{,}594{,}323 \approx 25.5$ **million states**

With insulation (e.g., 4 positions insulated):

- Fixed positions: $3 + 4 = 7$
- Variable positions: $16 - 7 = 9$
- Configurations: $3^9 = 19{,}683$
- Agent positions: 16
- **Total:** $16 \times 19{,}683 \approx 315{,}000$ **states**

This is an enormous increase from 3×3 to 4×4.

Is standard tabular value iteration feasible? Standard tabular value iteration becomes computationally infeasible at 4×4 because it requires storing and repeatedly updating values for on the order of 10^9 states (and potentially a policy over those states), which is far beyond typical memory and compute budgets.

Approximations / modifications that make DP more tractable. Several practical changes can reduce the effective complexity:

- **Prune invalid configurations:** ignore states where no directed connection from sources can possibly reach the sink (reachability pruning).
- **Restrict the design space:** limit the number of components (e.g., at most k NAND gates), limit wire length, or enforce left-to-right construction constraints.
- **Reward shaping / partial credit:** use score(G) to give intermediate signal toward XOR instead of only a binary terminal reward.
- **Replace tabular DP:** use search methods (e.g., best-first search) or function approximation (approximate V or Q) instead of enumerating the full state space.

Overall, the key difficulty is not the 3×3 case itself, but the exponential growth in circuit configurations as the grid increases.