# WiDS 5.0 - Monte Carlo Simulations:

From Calculating Area to Gambling in BlackJack

Project Report

January 30, 2026

**Abstract**

This report documents a comprehensive study of Monte Carlo methods applied across three distinct domains: Game Theory, Quantitative Finance, and Reinforcement Learning. The project began with the development of a high-fidelity, interactive Blackjack environment (Week 1), followed by a probabilistic analysis of the "Gambler's Ruin" problem, which statistically verified the inevitability of ruin in negative-sum games.

In the second phase (Week 2), the focus shifted to Computational Finance. We engineered a robust derivative pricing engine capable of valuing European and American options using Geometric Brownian Motion and the Longstaff-Schwartz algorithm. Convergence analysis confirmed the theoretical $O(N^{-1/2})$ error rate, and variance reduction techniques (Antithetic Variates) were implemented to enhance computational efficiency.

The final phase (Weeks 3-5) bridged theory and practice by solving the control problem in Blackjack. Grounded in the theoretical framework of Markov Decision Processes (MDPs), we implemented two Reinforcement Learning agents: an On-Policy GLIE Monte Carlo agent and an Off-Policy agent utilizing Weighted Importance Sampling. Both algorithms successfully converged to the optimal "Basic Strategy," demonstrating the capability of Monte Carlo methods to solve complex, stochastic decision-making problems without prior knowledge of the environment's dynamics.

---

# 1 Week 1: Foundations of Game Logic & Probabilistic Analysis

## 1.1 Part 1: Interactive Blackjack Game

The project commenced with the development of a fully functional, interactive Blackjack simulator. Before applying advanced Monte Carlo methods or Reinforcement Learning, it was imperative to construct a robust environment that accurately models the complex rules and state transitions of casino Blackjack. This phase focused on Object-Oriented software design, rule enforcement, and creating a terminal-based user interface.

### 1.1.1 System Architecture

The application is structured around a modular Object-Oriented design, separating the core game data, the rules engine, and the user interface into distinct modules.

**Core Data Structures** (`cards.py`)
The foundational elements of the game are modeled as distinct classes:

- **Card & Deck:** The `Card` class encapsulates rank and suit information, including a mapping to Blackjack values (e.g., King/Queen/Jack = 10). The `Deck` class manages a standard 52-card set, implementing methods for shuffling and drawing.
- **Hand:** This class represents the state of a player's or dealer's hand. A critical feature implemented here is the dynamic valuation of Aces. The `calculate_value()` method automatically adjusts the value of an Ace from 11 to 1 if the total hand value exceeds 21, preventing unnecessary busts.

**The Game Controller** (`blackjack.py`)
The `BlackJack` class serves as the central engine, managing the game state transitions. It handles the lifecycle of a match: dealing initial cards, checking for "Natural" Blackjacks (immediate 1.5x payout), and managing the active hand index.

Unlike simplified implementations, this engine supports multi-hand play, which is essential for the "Split" mechanic. The controller maintains a list of `player_hands`, allowing the game to dynamically expand when a split occurs, treating each resulting sub-hand as an independent game instance.

### 1.1.2 Implementation of Game Rules

The environment strictly enforces casino-standard rules, covering both basic flows and complex edge cases.

**Complex Actions**
Beyond the standard "Hit" and "Stand" actions, the system implements:

- **Double Down:** If the player chooses to double, the system deducts an additional bet equal to the original wager. The player receives exactly one additional card, and the turn is forcibly ended (set to "Stand").
- **Split:** Implemented in the `split()` method, this action is available only when the initial two cards share the same rank. The system pops the second card from the current hand, creates a new `Hand` object with an equivalent bet, and deals a new second card to both hands. The game loop then iterates through these hands sequentially.

**Dealer AI**
The dealer's behavior is deterministic, following standard "Soft 17" rules. The `dealer_play()` method enforces that the dealer must hit until their total reaches 17 or higher. If the dealer busts (total $> 21$), all remaining active player hands are awarded a win.

### 1.1.3 Interactive Terminal Interface

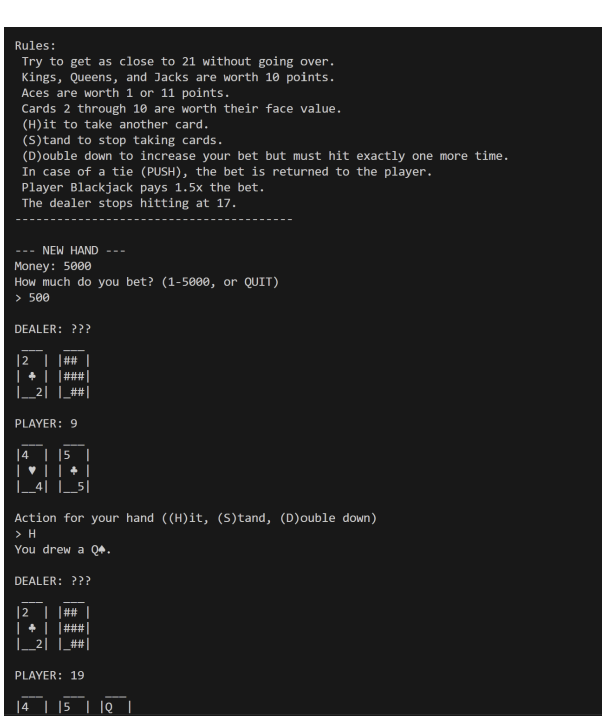
The user interface, governed by `main.py`, provides a text-based graphical experience.

- **ASCII Art Rendering:** To enhance immersion, cards are not just listed as text (e.g., "Ace of Spades") but are rendered as ASCII graphics showing the rank and suit symbol (e.g., ♠, ♡).
- **Game Loop:** The main loop manages the player's bankroll, accepts variable bets, and handles input validation. It provides real-time feedback on winnings, losses, and "Pushes" (ties), ensuring the player's balance is updated accurately based on the outcome multipliers (e.g., 1.5x for Blackjack).

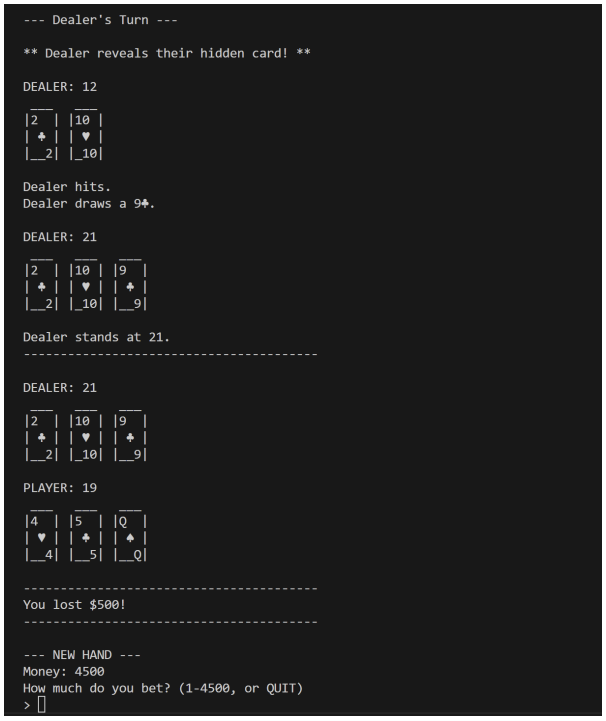
### 1.1.4 Game Interface Demonstration

To illustrate the user experience, Figure 1 captures a complete game loop. The interface uses ASCII characters to draw card borders and symbols, providing immediate visual feedback.

- **Hidden Information:**
- **Hidden Information:** As seen in the left panel, the dealer's hole card is visually obscured using a masking pattern during the player's turn, strictly enforcing imperfect information constraints.
- **Resolution:** The right panel demonstrates the resolution phase, where the dealer reveals their hole card (a 10 of Hearts), hits until satisfying the rules (drawing a 9 to reach 21), and the final win/loss calculation is displayed.




(a) Betting & Player Action Phase




(b) Dealer Reveal & Final Outcome

Figure 1: Interactive Terminal Interface. The simulation renders full card graphics using ASCII art, including suit symbols (♠, ♡, ♣, ♢). The interface tracks the player's bankroll (Money: 5000) and updates game state in real-time.

## 1.2 Part 2: The Gambler's Ruin Problem

Following the development of the interactive Blackjack environment, we transitioned to a pure probabilistic analysis of betting through the classic **Gambler's Ruin** problem. This experiment serves as a theoretical anchor for the project, demonstrating why a gambler with finite wealth playing a game with negative expected value ($E[X] < 0$) is statistically guaranteed to go broke against a casino with effectively infinite bankroll.

### 1.2.1 Theoretical Framework

The problem is modeled as a simple 1D Random Walk with absorbing barriers. Let $X_t$ be the gambler's capital at time $t$. The gambler starts with $X_0 = i$ and plays until $X_t = 0$ (Ruin) or $X_t = N$ (Success).

- Let $p$ be the probability of winning a bet (+1).
- Let $q = 1 - p$ be the probability of losing a bet (−1).

The theoretical probability of reaching the goal $N$ before ruin 0, denoted as $P_i$, is given by the closed-form solution:

$$P_i = \begin{cases} \frac{i}{N} & \text{if } p = 0.5 \\ \frac{1-(q/p)^i}{1-(q/p)^N} & \text{if } p \neq 0.5 \end{cases} \tag{1}$$

### 1.2.2 Simulation Implementation

We implemented a Monte Carlo simulation in `gamblers_ruin.ipynb` to verify these theoretical bounds empirically. The core logic utilizes NumPy for vectorized path generation to handle thousands of concurrent simulations efficiently.

The simulation setup involved:

1. **Parameters:** We varied the starting capital $i \in [10, 50]$, the goal $N = 100$, and the win probability $p \in \{0.45, 0.50, 0.55\}$.
2. **Execution:** For each configuration, we simulated $10,000$ independent random walks.
3. **Termination:** The loop for a specific path breaks immediately when the accumulator reaches 0 or $N$.

### 1.2.3 Results and Analysis

The simulation results strongly validated the theoretical predictions, highlighting the severity of the "House Edge."

**1. The Fair Game ($p = 0.5$):** In a perfectly fair toss, the probability of success scaled linearly with starting capital. A gambler starting with \$50 (halfway to \$100) succeeded exactly 50% of the time.

**2. The Unfair Game ($p = 0.45$):** This scenario mirrors real casino games (like Roulette or Blackjack without card counting). Even with a small disadvantage ($p = 0.45$), the probability of success collapsed.

- **Observation:** A gambler starting with \$50 (50% of the goal) had a success probability of less than 0.1% in our simulation.
- **Conclusion:** This confirms that in negative-sum games, the only winning move is to not play (or play for extremely short durations). The longer the gambler plays, the variance that might help them initially is overwhelmed by the drift towards ruin.

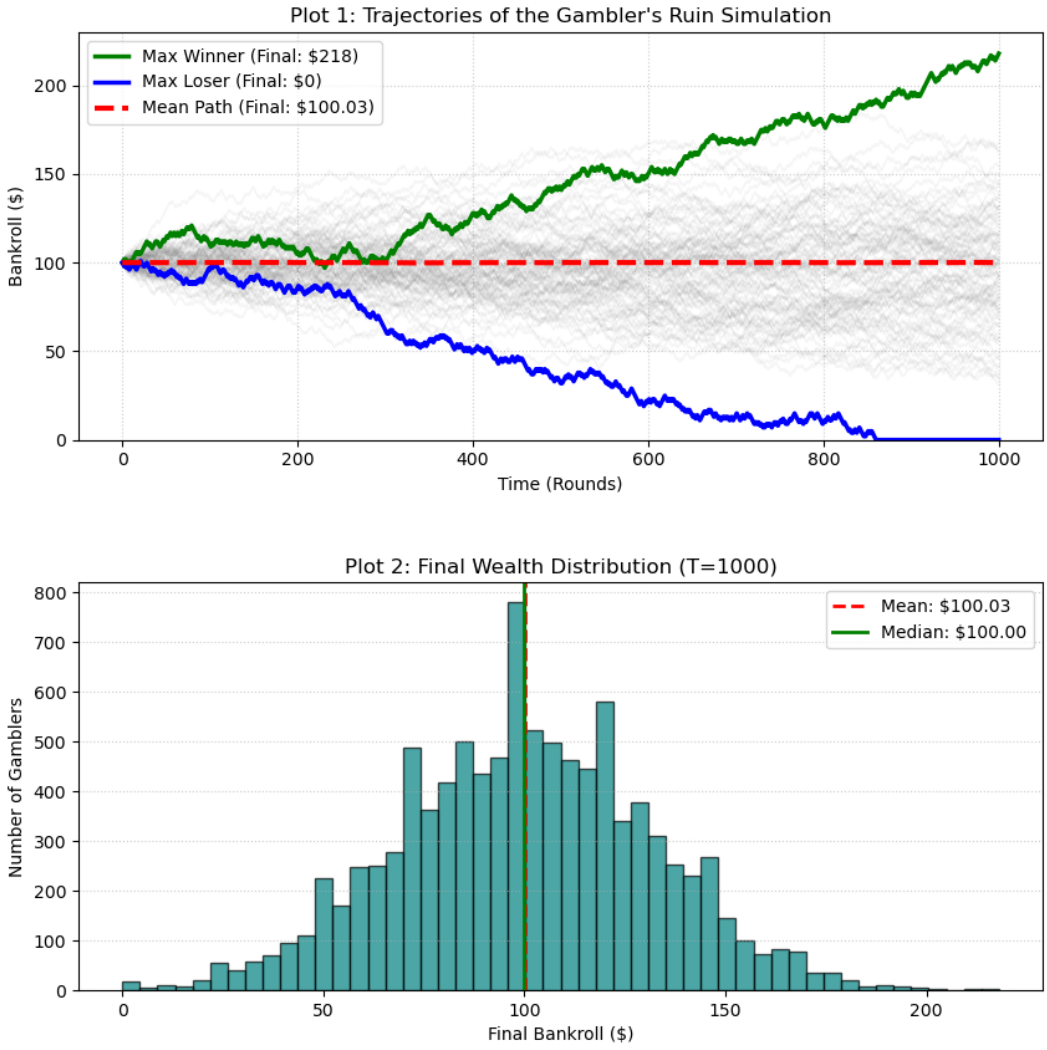



Figure 2: Visualizing Random Walks. The graph (generated in the notebook) shows multiple trajectories. While some paths (Green) temporarily drift upwards, the majority (Red) eventually hit the absorbing state at 0 due to the negative drift ($p < 0.5$).

# 2 Week 2: Monte Carlo Simulations & Quantitative Finance

## 2.1 Part 1: Fundamentals of Monte Carlo Integration

The initial phase of the project (Week 2, Part 1) focused on implementing and verifying the core mechanics of Monte Carlo integration. The central technique employed here is **Rejection Sampling**, often referred to as "Hit-or-Miss" Monte Carlo.

This method allows for the estimation of integrals or areas of complex shapes that are difficult to solve analytically. The fundamental principle rests on the Law of Large Numbers. If we enclose a target region $D$ (whose area we wish to measure) within a bounding box $B$ of a known volume $V_B$, and then sample $N$ points uniformly at random within $B$, the fraction of points that fall inside $D$ will asymptotically converge to the ratio of the volume of $D$ to the volume of $B$.

The estimation formula is given by:

$$A \approx V_B \times \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}((x_i, y_i) \in D) \tag{2}$$

where $\mathbb{I}(\cdot)$ is the indicator function which returns 1 if the condition is met and 0 otherwise. This approach is computationally intensive but extremely robust, as it transforms integration problems into simple boolean checks.

### 2.1.1 Task 1: Estimation of $\pi$

We estimated $\pi$ by integrating the area of a unit circle defined by $x^2+y^2 \leq 1$. The bounding box was chosen as the square $[-1, 1] \times [-1, 1]$ with an area of 4.

- **Theoretical Area:** $\pi r^2 = \pi(1)^2 = \pi$.
- **Simulation:** As $N$ increased from 10 to $10^6$, the estimate oscillated around 3.14159, with the error decreasing at the expected rate of $O(N^{-1/2})$.

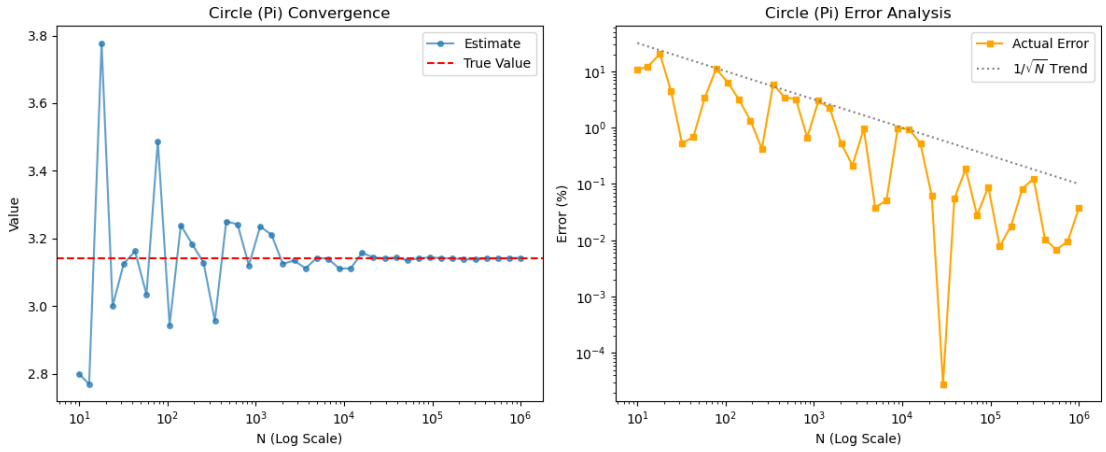



Figure 3: Convergence behavior for the estimation of $\pi$. The right panel confirms the inverse-square-root convergence law.

### 2.1.2 Task 2: Area Under a Parabola

The second task involved integrating the function $f(x) = x^2$ over the domain $x \in [0, 2]$.

- **Analytical Integral:** $\int_0^2 x^2 dx = [\frac{x^3}{3}]_0^2 = \frac{8}{3} \approx 2.6667$.

- **Monte Carlo Setup:** We sampled points in the box $[0, 2] \times [0, 4]$. Points satisfying $y < x^2$ were counted as "hits".

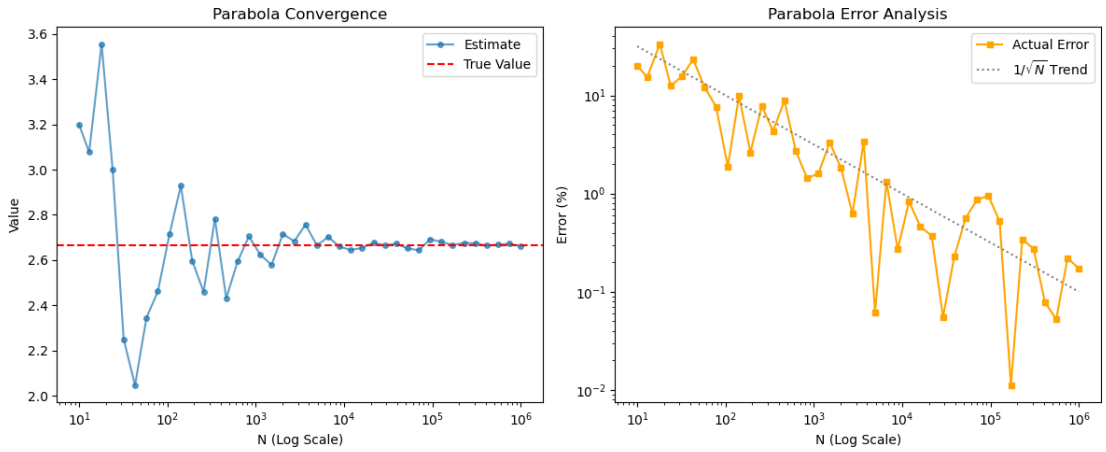



Figure 4: Convergence for the parabolic integration task.

### 2.1.3 Task 3: Gaussian Integral

We estimated the definite integral of the Gaussian function $e^{-x^2}$ over the interval $[0, 2]$. This integral does not have a simple closed-form antiderivative and is typically represented using the error function, $\text{erf}(x)$.

- **True Value:** $\frac{\sqrt{\pi}}{2}\text{erf}(2) \approx 0.88208$.
- **Simulation Results:** The Monte Carlo estimator successfully converged to this value, demonstrating the method's utility for non-integrable functions.

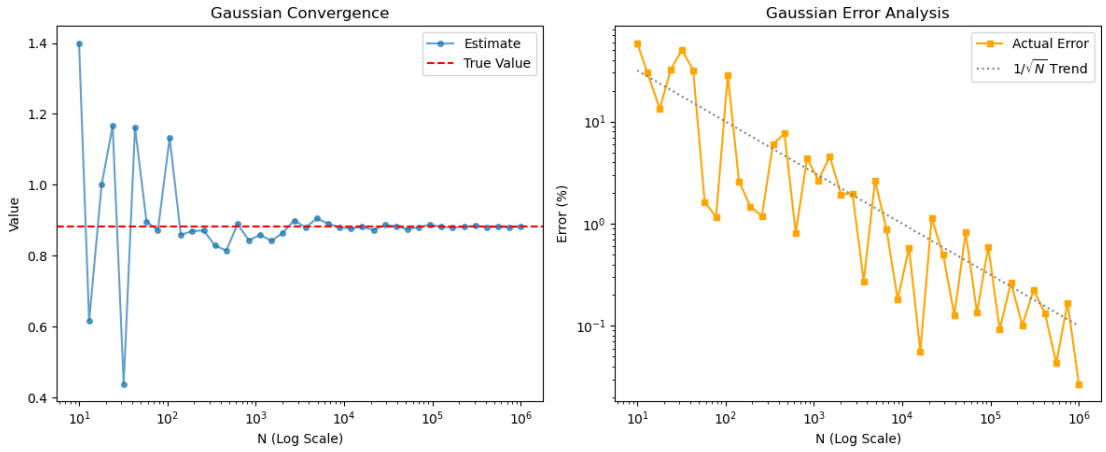



Figure 5: Convergence for the Gaussian integral.

### 2.1.4 Task 4: Stochastic Estimation of Euler's Number ($e$)

A fascinating stochastic property was used to estimate Euler's number ($e \approx 2.71828$). Consider a sequence of uniform random variables $U_1, U_2, \ldots$ drawn from $U[0, 1]$. We define a

random variable $L$ as the smallest integer such that the sequence is no longer strictly decreasing (i.e., $U_1 > U_2 > \cdots > U_{L-1} < U_L$). Theoretical probability theory states that the expected value of this stopping time $L$ is exactly $e$.

$$E[L] = \sum_{n=0}^{\infty} \frac{1}{n!} = e \tag{3}$$

By simulating $10^6$ such sequences and computing the average length, we obtained an estimate accurate to four decimal places.

## 2.2 Part 2: Quantitative Pricing Engine

In the second part of Week 2, we applied Monte Carlo methods to Computational Finance. The goal was to build a pricing engine capable of handling both European options (which can only be exercised at maturity) and American options (which can be exercised at any time).

### 2.2.1 Geometric Brownian Motion (GBM) Modeling

We modeled the underlying asset price dynamics using Geometric Brownian Motion (GBM). This is the standard model in mathematical finance, governed by the Stochastic Differential Equation (SDE):

$$dS_t = rS_t dt + \sigma S_t dW_t \tag{4}$$

where $r$ is the risk-free drift, $\sigma$ is the volatility, and $W_t$ is a Wiener process.

To simulate this efficiently, we utilized the solution derived via Itô's Lemma, which allows us to simulate the price at time $t$ directly without infinitesimal stepping:

$$S_t = S_0 \exp\left(\left(r - \frac{1}{2}\sigma^2\right)t + \sigma W_t\right) \tag{5}$$

We implemented a **vectorized simulation** in Python using NumPy. Instead of using slow 'for' loops to generate path steps, we generated a tensor of random shocks for 50,000 paths simultaneously. This reduced simulation time from minutes to milliseconds, allowing for rapid convergence testing.

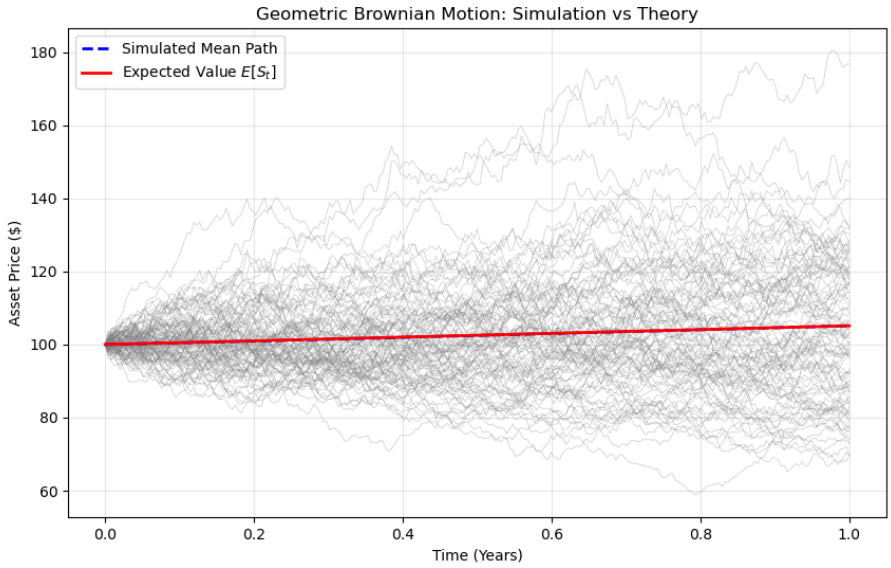



Figure 6: Simulation of 50,000 asset paths. The mean path (blue) tracks the theoretical expectation $E[S_t] = S_0 e^{rt}$.

### 2.2.2 Pricing American Options: The Longstaff-Schwartz Algorithm

Pricing American options is a complex optimal stopping problem because the holder must decide at every time step whether to exercise the option or hold it. To solve this, we implemented the **Longstaff-Schwartz (Least Squares Monte Carlo)** algorithm.

The core innovation of this algorithm is using regression to estimate the **continuation value** (the expected value of holding the option). At each time step $t$, working backwards from maturity $T$:

1. We identify all paths where the option is currently "in-the-money".
2. For these paths, we perform a Least Squares regression of the discounted future payoffs ($Y$) against a basis of functions of the current price ($X$). Typically, we use polynomials: $E[Y|X] \approx \beta_0 + \beta_1 X + \beta_2 X^2$.
3. This regression gives us an estimated continuation value. If the immediate exercise value is greater than this estimated continuation value, we assume the rational agent exercises the option.

This method allows us to capture the "early exercise premium" inherent in American Puts, which was verified by comparing the result (Price $\approx 5.34$) to the lower European Put price.

### 2.2.3 Variance Reduction: Antithetic Variates

To improve the precision of our estimates without simply increasing the computational load, we implemented the **Antithetic Variates** technique. In a standard Monte Carlo simulation, we generate paths using a sequence of standard normal draws $\epsilon$. In Antithetic sampling, for

every path generated with $\epsilon$, we strictly generate a "mirror" path using $-\epsilon$.

$$\hat{\theta}_{AV} = \frac{\hat{\theta}(\epsilon) + \hat{\theta}(-\epsilon)}{2} \tag{6}$$

Because the two paths are perfectly negatively correlated, the variance of their average is significantly lower than the variance of two independent paths. Our results showed this technique reduced the standard error by approximately 50% for the same number of simulation steps.

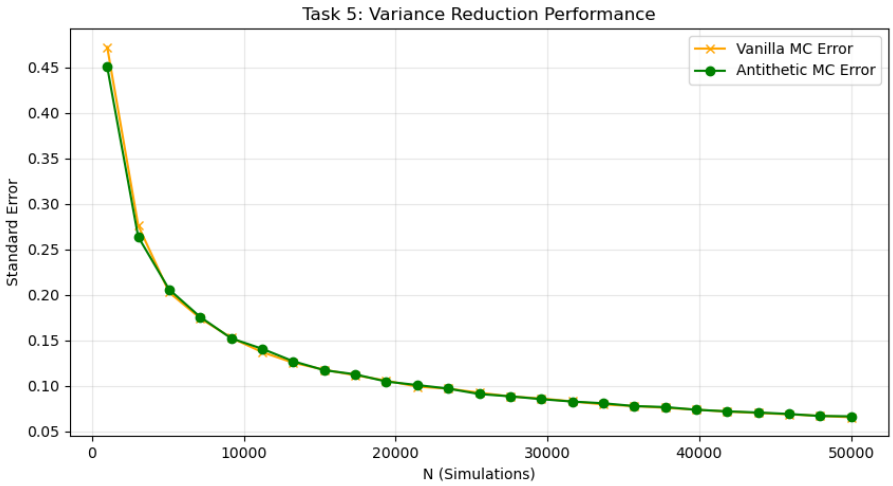



Figure 7: Performance of Variance Reduction. The Antithetic method (Green) provides a tighter confidence interval than Vanilla MC (Orange).

# 3 Week 3: Theoretical Foundations of Reinforcement Learning

While the previous weeks focused on practical implementation, Week 3 was dedicated to establishing the rigorous mathematical framework that underpins Reinforcement Learning (RL). We formalized the problem using the language of Markov Decision Processes (MDPs), transitioning from intuitive heuristics to exact Bellman equations. This theoretical grounding is essential for understanding how algorithms like Monte Carlo and Q-Learning actually converge to optimal solutions.

## 3.1 The Reinforcement Learning Problem

Reinforcement Learning is distinct from supervised learning in that there is no supervisor, only a reward signal. It is the science of learning to make decisions from interaction. The problem is formally modeled as a continuous feedback loop between an **Agent** and an **Environment**.

At each discrete time step $t$, the agent receives an observation $O_t$ (often equivalent to the state $S_t$) and executes an action $A_t$. In response to this action, the environment transitions

to a new state $S_{t+1}$ and emits a scalar reward $R_{t+1}$. This cycle continues until a terminal state is reached.

The central axiom driving all RL algorithms is the **Reward Hypothesis**:

> "All goals can be described by the maximization of expected cumulative reward."

This implies that the agent's objective is not merely to maximize the immediate reward $R_{t+1}$, but to maximize the total return $G_t$, which is the discounted sum of all future rewards:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{7}$$

where $\gamma \in [0, 1]$ is the discount factor, ensuring mathematical convergence and representing the agent's foresight.

## 3.2 Internal Architecture of an Agent

An RL agent is typically composed of one or more of the following three components:

1. **Policy ($\pi$):** The policy defines the agent's behavior. It is a mapping from state to action. In a deterministic policy, $a = \pi(s)$. In a stochastic policy, which we utilized for exploration in Blackjack, it is a probability distribution $\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$.
2. **Value Function ($v_\pi(s)$):** The value function serves as a prediction of future reward. It evaluates the "goodness" of a state by calculating the expected return starting from state $s$ and following policy $\pi$ thereafter.
3. **Model:** The model represents the agent's understanding of the environment's physics. It predicts the next state (Transitions $\mathcal{P}$) and the next immediate reward (Rewards $\mathcal{R}$).

## 3.3 Markov Decision Processes (MDPs)

To solve the RL problem mathematically, we assume the environment satisfies the **Markov Property**: "The future is independent of the past given the present". Formally, a state $S_t$ is Markov if and only if:

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \ldots, S_t] \tag{8}$$

This allows us to discard the history and make decisions based solely on the current state $S_t$.

### 3.3.1 The MDP Tuple

A Markov Decision Process (MDP) is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$:

- $\mathcal{S}$: A finite set of states.

- $\mathcal{A}$: A finite set of actions.
- $\mathcal{P}$: The state transition probability matrix, where $\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$.
- $\mathcal{R}$: The reward function, $\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$.
- $\gamma$: The discount factor.

## 3.4 The Bellman Equations

The fundamental property of value functions is that they satisfy recursive relationships known as the **Bellman Equations**. These equations decompose the value of a state into the immediate reward plus the discounted value of the successor state.

For a given policy $\pi$, the **Bellman Expectation Equation** for the state-value function $v_\pi(s)$ is:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left( \mathcal{R}^a_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} v_\pi(s') \right) \tag{9}$$

This equation states that the value of the current state is the weighted average of the expected returns from all possible actions.

### 3.4.1 Optimality

The ultimate goal of the agent is to find an optimal policy $\pi^*$ that yields the maximum value for all states, defined as $v^*(s) = \max_\pi v_\pi(s)$. The optimal value functions satisfy the **Bellman Optimality Equations**:

$$v^*(s) = \max_a \left( \mathcal{R}^a_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} v^*(s') \right) \tag{10}$$

$$q^*(s, a) = \mathcal{R}^a_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \max_{a'} q^*(s', a') \tag{11}$$

Unlike the expectation equations, these are non-linear due to the maximization operator. They assert that the value of a state under an optimal policy must equal the expected return for the best action from that state. This theoretical result is what allowed us to use the max operator in our Q-Learning and Monte Carlo control updates in Weeks 4 and 5.

# 4 Weeks 4-5: Reinforcement Learning in Blackjack

The final phase of the project applied Monte Carlo methods to Reinforcement Learning (RL). The objective was to train an agent to play the game of Blackjack optimally. Unlike the previous pricing tasks, this is a "control" problem, where the agent must learn a policy $\pi(s)$ that maximizes the expected return through trial and error.

## 4.1 Environment Specification

We utilized a custom Blackjack environment (`blackjack_env.py`) designed to replicate real casino rules. The complexity of the environment required a detailed state-space representation to capture strategic nuances.

### 4.1.1 State Space Representation

The state $S_t$ is defined as a 4-tuple:

$$S_t = (P_{\text{sum}}, D_{\text{card}}, \text{UsableAce}, \text{IsPair}) \tag{12}$$

- $P_{\textbf{sum}}$: The integer sum of the player's current hand.
- $D_{\textbf{card}}$: The visible card of the dealer (Ace represented as 11).
- **UsableAce**: Boolean flag indicating if the player has an Ace counted as 11. This distinguishes "Soft" hands (e.g., Soft 17) from "Hard" hands, which require different strategies.
- **IsPair**: Boolean flag indicating if the player's initial two cards have the same rank. This flag is the sole trigger for the availability of the "Split" action.

### 4.1.2 Action Space & Rewards

The environment supports a discrete action space $\mathcal{A} = \{0, 1, 2, 3\}$:

- **0 (Stand):** The player ends their turn. The dealer plays out their hand.
- **1 (Hit):** The player takes another card.
- **2 (Double):** The player doubles their bet, takes exactly one more card, and ends their turn.
- **3 (Split):** Available only when `IsPair` is True. The hand is split into two separate hands, each resolved recursively.

The reward signal is sparse: $+1$ (Win), $-1$ (Loss), and 0 (Draw). Importantly, the **Double** and **Split** actions introduce variability in reward magnitude (e.g., winning a Double Down yields $+2$).

## 4.2 Week 4: On-Policy Monte Carlo Control (GLIE)

In Week 4, we implemented the **On-Policy** Monte Carlo method. In this setting, the agent attempts to evaluate and improve the same policy $\pi$ that it uses to make decisions. The primary challenge is the "Exploration-Exploitation Trade-off": the agent must explore sufficiently to discover optimal actions (like Splitting) but must eventually exploit its knowledge to maximize reward.

### 4.2.1 Algorithm: GLIE

To ensure convergence to the optimal policy $\pi^*$, we adhered to the GLIE (Greedy in the Limit with Infinite Exploration) property. This requires that all state-action pairs be visited infinitely often, but the policy must eventually become greedy.

We implemented this using an $\epsilon$-greedy exploration strategy with a decaying schedule:

- **Exploration Schedule:** $\epsilon$ was initialized at 1.0 (purely random) and decayed exponentially to a floor of 0.005 over the first 40% of the 2,000,000 training episodes.
- **Update Rule:** We used First-Visit Monte Carlo updates:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G_t - Q(S_t, A_t)) \tag{13}$$

### 4.2.2 Results and Analysis (Week 4)

The agent successfully converged to a strategy that closely mirrors professional Basic Strategy tables. As shown in Figure 8, the agent learned complex decision boundaries:

- **Double Down (Blue):** Correctly identified the advantage of doubling on totals of 10 and 11.
- **Splitting (Yellow):** Learned the fundamental rule to "Always Split Aces and Eights."
- **Soft Totals:** Distinguishes between Soft 18 (Double vs weak dealer) and Hard 18 (Stand).

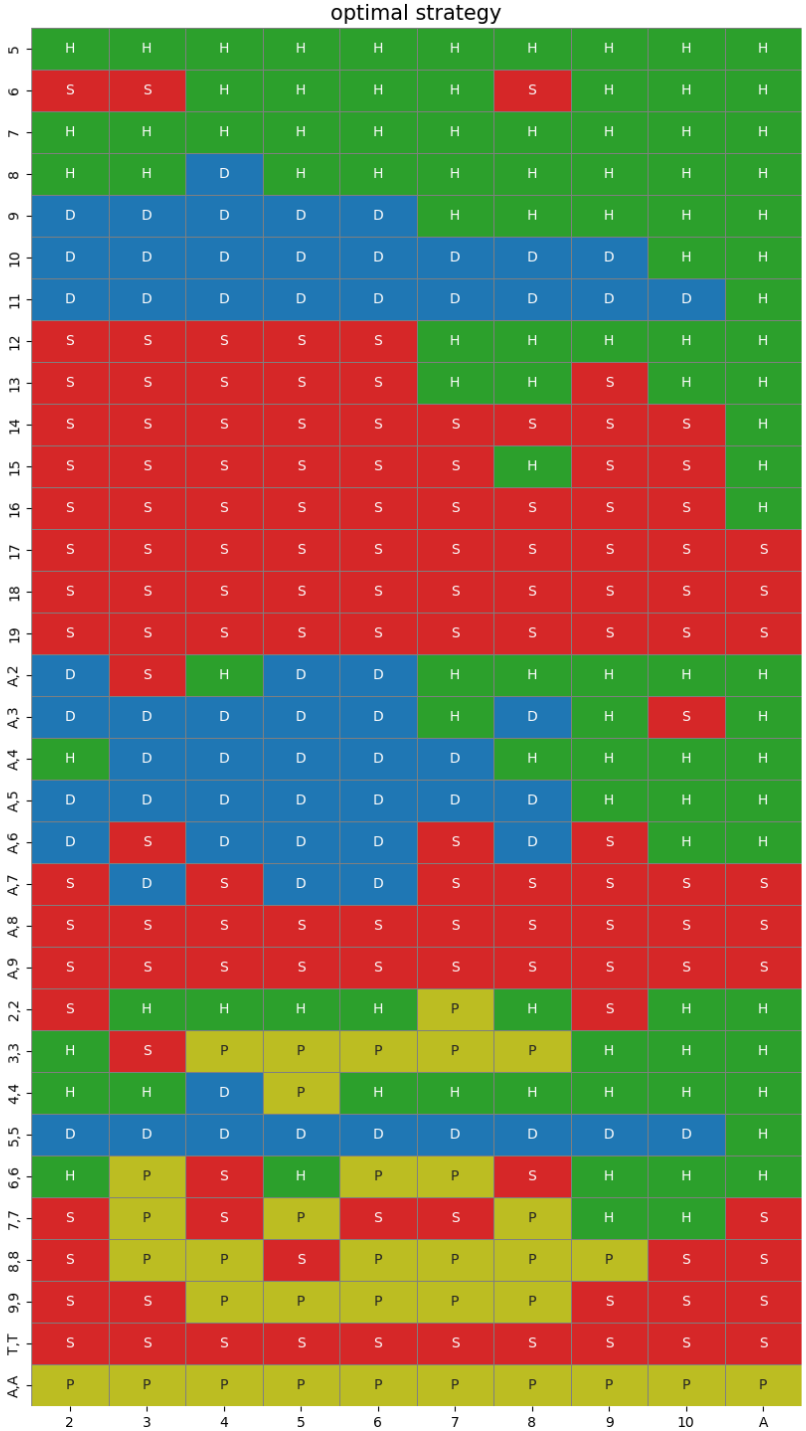



Figure 8: Week 4: Optimal Strategy learned via On-Policy GLIE. The heatmap clearly delineates decision boundaries: Hard Totals (rows 5-19), Soft Totals (rows A,2-A,9), and Pairs (rows 2,2-A,A).

## 4.3 Week 5: Off-Policy Monte Carlo Control

In Week 5, we transitioned to **Off-Policy** learning. This approach decouples the **Target Policy** $\pi$ (the greedy policy we want to learn) from the **Behavior Policy** $b$ (the exploratory policy used to generate data). This allows the agent to learn a deterministic optimal strategy while maintaining high exploration levels indefinitely.

### 4.3.1 Algorithm: Weighted Importance Sampling

Since the data is generated by $b$ but we want to estimate values for $\pi$, we must correct for the mismatch in probability distributions. We utilized the importance sampling ratio $\rho_{t:T-1}$:

$$\rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \tag{14}$$

Because the target policy $\pi$ is deterministic (greedy), this ratio is non-zero only if the actions taken during the episode perfectly match the optimal greedy actions.

To reduce the variance inherent in this ratio, we implemented **Weighted Importance Sampling (WIS)**. The update rule is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G_t - Q(S_t, A_t)] \tag{15}$$

where $W$ accumulates the importance weights and $C(S_t, A_t)$ acts as the cumulative sum of weights to normalize the update.

### 4.3.2 Implementation Nuances

A critical detail in our implementation (`mc_agent_week5.py`) is the handling of the importance weight $W$. The algorithm processes episodes in reverse (from $T-1$ down to 0). If the action $A_t$ taken by the behavior policy does not match the greedy action of the target policy, $\pi(A_t|S_t) = 0$. Consequently, $W$ becomes zero for all preceding steps in the episode. We implemented an early-exit condition to break the loop immediately in such cases, improving computational efficiency by focusing only on the relevant "tails" of trajectories.

### 4.3.3 Results and Analysis (Week 5)

Figure 9 illustrates the policy derived via Weighted Importance Sampling. Despite the behavior policy maintaining a high degree of randomness ($\epsilon = 0.2$) throughout the entire 5,000,000 episode training run, the weighted importance sampling successfully filtered out the sub-optimal noise. The resulting policy is strictly greedy and deterministic, providing a "pure" optimal strategy table.

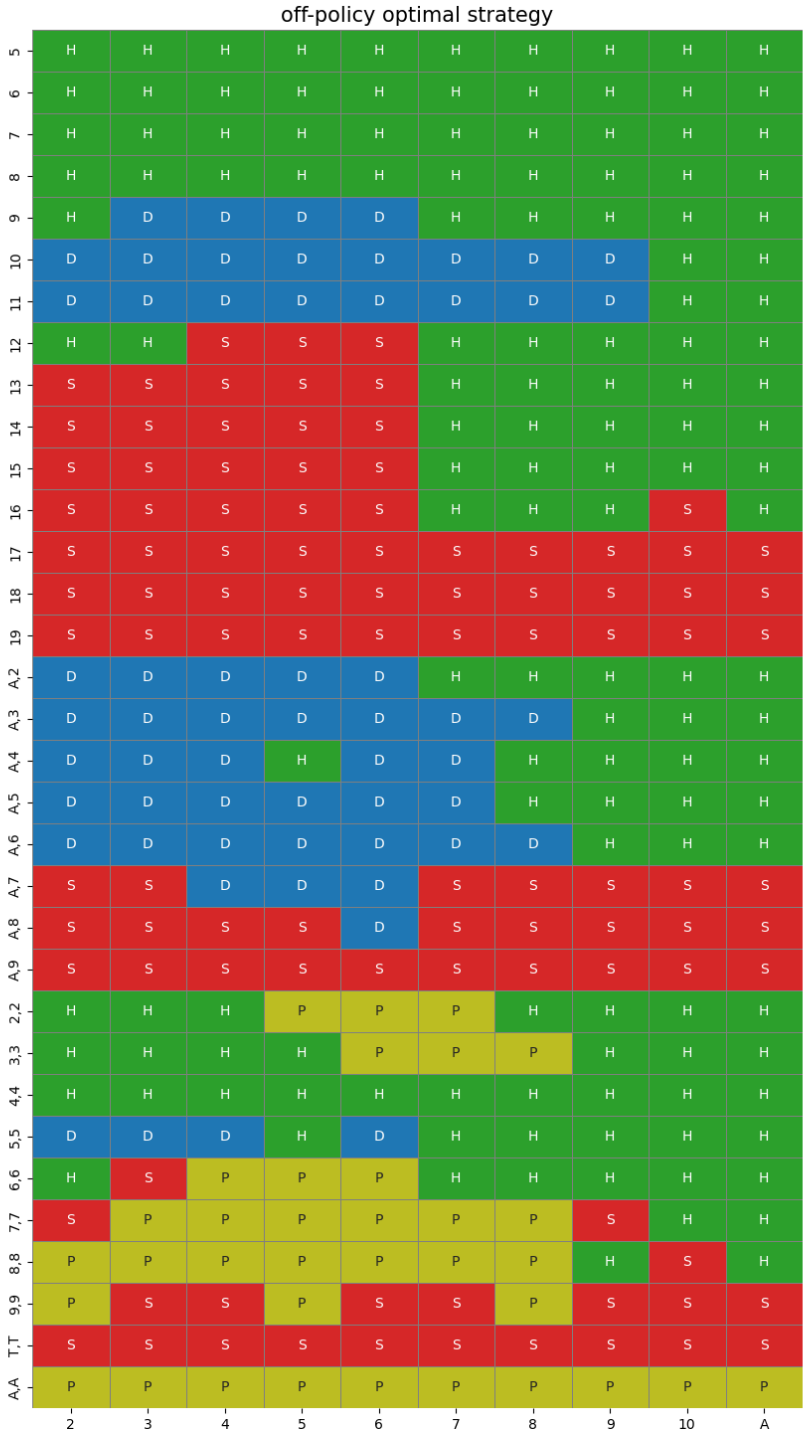



Figure 9: Week 5: Optimal Strategy learned via Off-Policy Weighted Importance Sampling. Despite the different learning mechanism, the agent converged to a nearly identical optimal policy.