# WiDS 5.0 Project Report

Monte Carlo Simulations: From calculating area to Gambling in BlackJack

Piyush Prajapati
WiDS 5.0

30th Jan,2026

## Contents

# 1 The Gambler's Ruin

## 1.1 Problem Setup

In this chapter, we study a Monte Carlo simulation of a stochastic wealth process inspired by repeated fair coin tosses. Each agent starts with an initial bankroll and repeatedly gains or loses one unit at every time step with equal probability. The process terminates for an agent once its wealth reaches zero, modeling a classical *gambler's ruin* scenario.

We simulate:

- $N = 10{,}000$ independent agents,
- each over $T = 1{,}000$ time steps,
- with an initial wealth of 100 units.

At each time step, the outcome is sampled from $\{-1, +1\}$ with equal probability.

## 1.2 Vectorized Monte Carlo Simulation

The simulation is implemented using NumPy vectorization. Let $X_{i,t} \in \{-1, +1\}$ denote the outcome for agent $i$ at time $t$. The wealth process is defined as:

$$W_{i,t} = W_{i,0} + \sum_{k=1}^{t} X_{i,k}, \quad W_{i,0} = 100$$

To enforce the absorbing boundary at zero wealth, we mask trajectories once bankruptcy occurs and set all future values to zero.

This fully vectorized approach enables efficient simulation of all $10^7$ state updates without explicit Python loops.

## 1.3 Trajectory-Level Analysis

Figure 1 shows the evolution of wealth for all agents. Individual trajectories are plotted with low opacity, highlighting the stochastic dispersion over time. Superimposed are:

- the mean wealth trajectory,
- the trajectory of the maximum final wealth,
- the trajectory of the minimum final wealth.

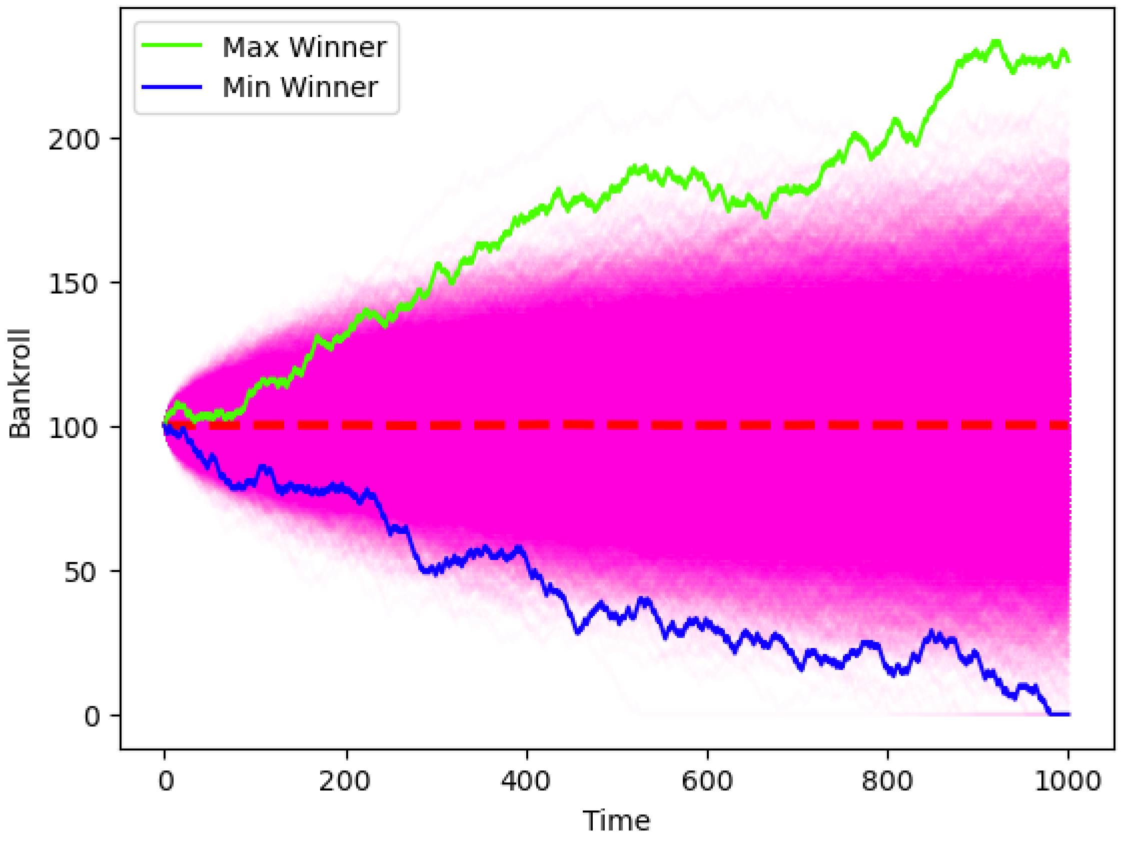

Figure 1: Wealth evolution of 10,000 agents over time. The dashed red curve represents the mean wealth.

Despite the fair nature of the game, the average wealth decreases over time due to the absorbing boundary at zero, demonstrating that fairness does not imply preservation of expected wealth in constrained stochastic processes.

### 1.4 Distribution of Final Wealth

The histogram of final wealth values is shown in Figure 2. The distribution exhibits:

- a heavy concentration at zero (bankrupt agents),
- a long right tail corresponding to surviving agents,
- a noticeable discrepancy between mean and median.

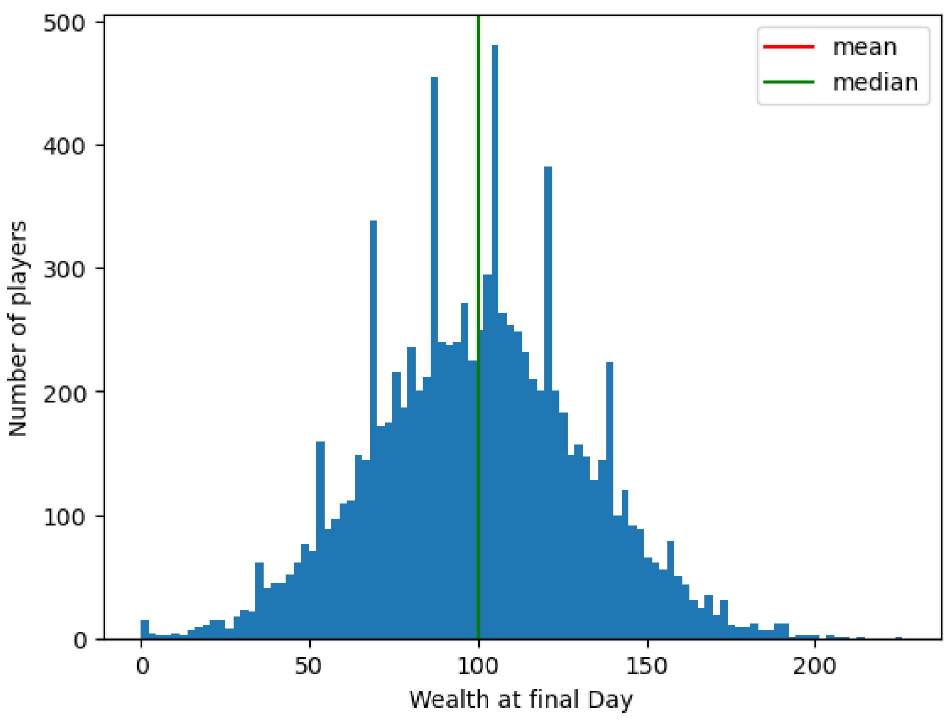

Figure 2: Distribution of wealth at the final time step.

The median wealth is significantly lower than the mean, indicating a positively skewed distribution where a small fraction of agents accumulate disproportionately high wealth.

### 1.5 Connection to the Central Limit Theorem

A sequence of fair coin tosses follows a binomial distribution. According to the Lindeberg–Lévy Central Limit Theorem, the sum of i.i.d. random variables with finite mean and variance converges in distribution to a normal distribution as the number of trials increases.

However, the presence of an absorbing boundary violates the independence assumption after ruin. Consequently:

- the early-time wealth distribution resembles a Gaussian,
- long-term behavior deviates significantly due to truncation at zero.

This highlights a crucial modeling insight: *structural constraints can dominate asymptotic statistical behavior.*

## 2 Modular Blackjack Environment

### 2.1 Problem Objective

The goal of this task is to build a clean, modular, object-oriented Python game that can later be used for large-scale Monte Carlo simulations. The emphasis is not on user interaction, but on separation of concerns, reusable components, and clear state management. Blackjack was chosen due to its well-defined rules, episodic nature, and suitability for probabilistic simulation.

## 2.2 Design Overview

The implementation follows a modular architecture with two clearly separated components:

- **Data Model (cards module):** Represents the physical entities of the game.
- **Game Engine (BlackJack class):** Implements game flow and rules.

This separation ensures that the game logic remains independent of card representation and visualization, which is essential for efficient Monte Carlo experiments.

## 2.3 Data Model

**Card Suits** are represented using Python's `Enum`, preventing invalid values and improving code robustness. Each **Card** stores a suit and rank and implements a readable string representation using Unicode symbols for debugging and visualization.

The **Deck** class models a standard 52-card deck and provides methods to reset, shuffle, and draw cards. It has no knowledge of Blackjack rules, making it reusable for other card-based games.

The **Hand** class stores the cards held by a player or dealer and encapsulates all scoring logic. In particular, it handles the dual value of aces (1 or 11), ensuring that the hand value is always computed correctly without leaking rule logic into the game engine.

## 2.4 Game Engine

The **BlackJack** class controls a single round of play. It maintains the deck, player hand, dealer hand, and game state. At the start of each round, the deck is reset and shuffled, and both participants are dealt two cards, ensuring statistical independence between rounds.

Two actions are supported:

- **Hit:** the player draws a card and may immediately bust.
- **Stand:** the dealer draws cards until reaching a value of at least 17.

Once the round ends, the outcome is resolved as a player win, dealer win, or push.

## 2.5 Policy-Based Interaction

The method `run_match(policy)` allows the game to be driven by an external policy function that maps the player's current hand value to an action. This abstraction decouples decision-making from game mechanics and enables easy integration with Monte Carlo simulations and reinforcement learning agents.

A simple threshold policy was implemented as a baseline:

$$\text{Hit if hand value} < 17, \text{ else Stand.}$$

### 2.6 Relevance to Monte Carlo Simulation

The design satisfies all requirements for scalable simulation:

- clean separation between logic and representation,
- no side effects such as print statements in core logic,
- deterministic state transitions given randomness,
- compatibility with automated policy evaluation.

This environment can therefore be executed millions of times to estimate win rates, evaluate strategies, or train learning agents.

## 3 Monte Carlo Estimation of Mathematical Constants and Integrals

### 3.1 Objective

The objective of this experiment is to develop intuition for Monte Carlo methods by estimating fundamental mathematical constants and definite integrals using random sampling. Emphasis is placed on vectorized NumPy implementations, convergence behavior, and error analysis as the number of samples increases.

### 3.2 Estimating $\pi$ via Random Sampling

The value of $\pi$ was estimated by uniformly sampling points inside a square $[-1,1] \times [-1,1]$ and counting the fraction that fall inside the unit circle:

$$x^2 + y^2 \leq 1$$

Since the area of the square is 4 and the area of the unit circle is $\pi$, the estimator is given by:

$$\hat{\pi} = 4 \cdot \frac{\text{points inside circle}}{N}$$

The simulation was run for sample sizes ranging from $10^1$ to $10^7$. As expected from the Law of Large Numbers, the estimate converges to $\pi$ as $N$ increases. The relative percentage error decreases approximately at a rate proportional to $1/\sqrt{N}$, demonstrating the slow but reliable convergence of Monte Carlo methods.

### 3.3 Estimating Euler's Number $e$

Euler's number was estimated using a probabilistic identity based on uniform random sampling. For a fixed parameter $t > 1$, samples were drawn uniformly from:

$$x \sim \mathcal{U}(1,t), \quad y \sim \mathcal{U}(0,1)$$

Using the relation:

$$\mathbb{P}(y \leq 1/x) = \frac{1}{(t-1)\ln t}$$

an estimator for $e$ can be derived as:

$$\hat{e} = t^{\frac{1}{(t-1)\hat{p}}}$$

The experiment was conducted for $t$ values between 2 and 20 with a fixed sample size of $N = 10{,}000$. The error initially decreases with increasing $t$, but numerical instability and floating-point precision limits prevent further improvement beyond a certain range.

An alternative estimator based on the expected length of decreasing sequences of uniform random variables was also implemented, including a fully vectorized version. This approach further reinforced the connection between Monte Carlo simulation and probabilistic characterizations of constants.

## 3.4 Generic Monte Carlo Integration Framework

A general-purpose Monte Carlo integration function was implemented to estimate areas under arbitrary curves. Given a predicate function defining a region and rectangular bounds, the area is approximated as:

$$\text{Area} \approx (\text{bounding area}) \times \frac{\text{points inside region}}{N}$$

This framework was applied to multiple problems:

- Area of a unit circle (recovering $\pi$),
- Area under $y = x^2$ on $[0, 1]$ (true value $= \frac{1}{3}$),
- Area under $y = e^{-x^2}$ on $[0, 2]$ (related to the Gaussian integral).

In each case, Monte Carlo estimates converge to the analytical solution as the number of samples increases.

## 3.5 Error Analysis and Convergence

For all experiments, error was measured as relative percentage deviation from the true value. Results consistently show:

- convergence guaranteed by the Law of Large Numbers,
- slow convergence rate independent of dimensionality,
- robustness to complex or non-analytic integrands.

These properties highlight why Monte Carlo methods are particularly valuable in high-dimensional settings where deterministic quadrature methods fail.

### 3.6 Conclusion

This set of experiments demonstrates the power and limitations of Monte Carlo methods. While convergence is slow compared to deterministic techniques, Monte Carlo estimation is simple, general, and scales well to complex geometries and probability distributions. The vectorized NumPy implementations further emphasize the importance of efficient numerical computation when performing large-scale simulations.

These insights form the foundation for later applications of Monte Carlo methods in reinforcement learning and control.

# 4 Week 4: Monte Carlo Prediction and Control

## 4.1 Learning Objectives

The goal of this week was to apply Monte Carlo methods to a reinforcement learning problem and understand how agents can learn value functions and optimal policies purely from experience. Using the Blackjack environment built earlier, the objectives were:

- To estimate state-value functions for a fixed policy using Monte Carlo prediction
- To learn an optimal policy using Monte Carlo control with exploration
- To analyze convergence behavior through learning curves and policy visualization

## 4.2 Why Monte Carlo Methods?

Monte Carlo methods are model-free reinforcement learning techniques. Unlike dynamic programming methods, they do not require knowledge of transition probabilities $P(s'|s, a)$ or reward functions. Instead, they rely on complete episodes of experience and learn by averaging observed returns.

Since Blackjack is an episodic task with terminal rewards and stochastic dynamics, Monte Carlo methods are particularly well suited for this problem.

## 4.3 Monte Carlo Prediction

The first experiment focused on Monte Carlo prediction, where the goal is to estimate the state-value function $V^{\pi}(s)$ for a fixed policy $\pi$.

A simple deterministic policy was used:

Hit until the player sum reaches 20, otherwise stand.

For each episode, the sequence of states, actions, and rewards was recorded. The return from time step $t$ was computed as:

$$G_t = \sum_{k=t}^{T} R_k$$

First-visit Monte Carlo prediction was applied. For each state encountered in an episode, only the return following its first occurrence was used to update the value estimate:

$$V(s) = \frac{1}{N(s)} \sum_{i=1}^{N(s)} G_i$$

After training on 10,000 episodes, the estimated value function reflected intuitive Blackjack behavior. States with high player sums (e.g., 20 or 21) had positive expected values, while states with low player sums showed negative expected returns. This confirmed that Monte Carlo prediction successfully captures long-term outcomes using sampled experience alone.

### 4.4 Monte Carlo Control

The next step was Monte Carlo control, where the objective is to learn an optimal policy $\pi^*$ without prior knowledge of the environment dynamics.

Instead of learning state values, action-value functions $Q(s, a)$ were estimated. An $\epsilon$-greedy policy was used to balance exploration and exploitation:

- With probability $\epsilon$, a random action is selected
- With probability $1 - \epsilon$, the action with the highest estimated $Q$-value is chosen

For each episode, returns were computed backward, and first-visit Monte Carlo updates were applied to the action-value function:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left( G_t - Q(s, a) \right)$$

where $\alpha$ is the learning rate.

To encourage sufficient exploration early in training while allowing convergence later, $\epsilon$ was gradually decayed over episodes, with a minimum exploration threshold to avoid premature convergence.

### 4.5 Training Results and Learning Curve

The agent was trained for 500,000 episodes. Performance was evaluated by tracking the total reward obtained in each episode. Due to the high variance inherent in Blackjack, a rolling average over recent episodes was used to visualize learning progress.

The learning curve shows a clear improvement over time:

- Early training exhibited poor performance due to random exploration
- Average rewards steadily increased as the agent refined its policy
- Performance eventually stabilized, indicating convergence

This behavior is consistent with successful Monte Carlo control, where value estimates improve as more experience is accumulated.

## 4.6 Learned Policy Visualization

After training, the learned policy was extracted from the action-value function. For each state, the action with the higher estimated $Q$-value was selected. The policy was visualized over player hand sums and dealer showing cards.

The resulting policy closely resembles the well-known Blackjack basic strategy:

- The agent stands on high player sums
- Hits are preferred when the dealer shows strong cards
- More aggressive play emerges when the risk of busting is low

Importantly, this strategy was not hard-coded. It emerged purely from repeated interaction with the environment and reward feedback.

## 4.7 Challenges and Insights

One of the main challenges was slow convergence, as some state-action pairs occur infrequently. This was mitigated by maintaining a sufficiently high exploration rate early in training.

Another challenge was the high variance of rewards in Blackjack. Individual episodes can result in losses even when optimal actions are taken. Long training runs and rolling-average evaluation were necessary to observe meaningful learning trends.

Overall, this experiment demonstrated that Monte Carlo methods can successfully learn optimal behavior in stochastic environments without explicit models or prior domain knowledge.