AIL 7022: Reinforcement Learning
Assignment 1

Ankur Kumar (2025AIB2557)
Indian Institute of Technology Delhi

```
A1
├── Q1
│   ├── env.py   -- environment setup for FootballSkills env
│   ├── PI_and_VI.py   -- combined policy iteration and value iteration solution
│   ├── VI_non_stat_env.py   -- value iteration in non-stationary environment
│   └── mod_VI.py   -- modified value iteration algorithm solution
└── Q2
    └── or_gym
        ├── knapsack_sol.py   -- knapsack problem solution (problem - part 1 and part 2)
        ├── knapsack_sol2.py   -- alternate knapsack solution (problem - part 3)
        ├── portfolio_sol.py   -- portfolio optimization solution (problem - part 1 and part 2)
        └── portfolio_sol2.py   -- portfolio optimization solution (problem - part 3)
```

Figure 1: Directory structure of assignment code files.

1. Policy and Value Iteration

1.1. Stationary Environment Analysis

The implementation of the Policy Iteration and Value Iteration algorithms can be found in the repository. The number of calls made to the transition function (to obtain the probability and next state) are:
Policy Iteration (PI): 338,400
Value Iteration (VI): 86,800
These numbers are intuitive: in Value Iteration, the policy evaluation step is performed only once per policy update, rather than being repeated until convergence as in Policy Iteration. This significantly reduces the number of calls made to the transition function during the implementation.

Comparing Policies by PI and VI:
The policy and value function obtained from Policy Iteration and Value Iteration are identical, which empirically verifies the theoretical result that both algorithms converge to the optimal policy. However, it is important to note that obtaining the exact same policy depends on certain implementation details, such as using the same tie-breaking rule in both algorithms. Furthermore, the policies are identical only for non-terminal states, since the action in all terminal states is *None*.
Computing Reward for different runs with different seeds:
We used the learned policy to evaluate the reward over 20 different seed values. Here, the total reward is defined as the sum of the rewards obtained at each step along the trajectory:

$$R_{\text{total}} = \sum_{t=0}^{T} r_t$$

It is important to note that this is different from the *Return* G_t, which is the discounted sum of rewards:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

The mean and standard deviation of the total rewards across N runs were computed using the following formulas, here $N = 20$:

$$\mu = \frac{1}{N} \sum_{i=1}^{N} R_i \qquad \sigma = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (R_i - \mu)^2}$$

Results:

- Policy Iteration: $\mu = 47.00$, $\sigma = 22.36$
- Value Iteration: $\mu = 47.00$, $\sigma = 22.36$

Comparison of PI and VI with different Gamma(γ)
See Table 1 for a detailed comparison of Policy Iteration (PI) and Value Iteration (VI) across different values of γ.

γ	**PI calls**	**VI calls**	**PI (Mean, Std)**	**VI (Mean, Std)**	**Notes**
0.95	338,400	86,800	47.00, 22.36	47.00, 22.36	Policies identical (non-terminal states).
0.5	132,000	75,600	32.00, 58.72	32.00, 58.72	Policies identical (non-terminal states).
0.3	116,800	47,600	32.00, 58.72	32.00, 58.72	Policies not identical.

Table 1: Comparison of Policy Iteration (PI) and Value Iteration (VI) across different γ values.

In the table we observe that the number of transition function calls decreases as γ is reduced. This is intuitive because γ controls how strongly future states influence the current state. For smaller γ, the impact of distant rewards diminishes quickly, meaning there is less information to propagate through the state space. As a result, the value function stabilizes more quickly and the algorithm converges in fewer iterations, thereby reducing the total number of transition function calls.

We also observe that as γ decreases, the mean reward is reduced while the standard deviation increases. This happens because with a smaller γ, the agent becomes more short-sighted, focusing only on immediate rewards rather than considering long-term outcomes. This reflects that the agent prioritizes short-term gains or terminates quickly, sometimes resulting in much worse or much better outcomes — hence larger variance.

For $\gamma = 0.3$, we observe that both the value functions and the resulting policies obtained from Policy Iteration and Value Iteration differ for some non-terminal states.
This happens because:

- A smaller discount factor γ reduces the influence of future rewards, so value propagation is weaker and both algorithms converge more quickly.
- With a looser tolerance (ϵ), the algorithms may terminate prematurely, before the value function has fully stabilized.
- Even small discrepancies in $V(s)$ can affect the greedy choice of actions.

When we reduce the tolerance (enforce a stricter convergence condition), these discrepancies diminish, and both algorithms converge to the same policy and value function.
To view the animated demonstration of an agent following the optimal policy, click here for Policy Iteration and here for Value Iteration.

1.2. Non stationary Environment Analysis

The implementation of the Value Iteration in degrading pitch environment can be found in the repository.
The total number of calls made to the transition function in this case is **4,368,000**.
When Value Iteration was implemented without including time as part of the state, only **173,600** calls were made. The significant reduction in calls arises because, in the first case, the state space is 40 times larger due

to the additional time dimension.

See Table 2 for a comparison between Value Iteration with and without time included as part of the state. The table highlights the significant increase in the number of transition function calls when time is added to the state space, as well as the differences in the resulting mean and standard deviation of the obtained rewards. We also observe that the reward is lower in the normal VI case (without time in state). This happens because

Method	**Transition Calls**	**Mean Reward**	**Std Dev**
Value Iteration (normal)	173,600	24.50	46.42
Value Iteration (with time in state)	4,368,000	39.25	40.98

Table 2: Comparison of transition calls and performance metrics for Value Iteration with and without time in state.

the agent is forced to take a fixed action in a given state, irrespective of the probability of slipping, which in the original formulation was dependent on time. As a result, the agent has less flexibility to optimize its behavior, leading to lower returns.
To view the animated demonstration of an agent following the policy learned by Value Iteration with time in state, click here and to view the policy learned when Value iteration algorithm is implemented without time in state in degrading pitch case click here.

1.3. Modified Value Iteration

As pointed out in the question, if the values of all successors of a state remain unchanged, then the Bellman backup for that state will not change its value either. Updating such states repeatedly is redundant.
To improve the Value Iteration Algorithm we did the following :-

- Instead of updating all states every iteration, only states whose successors' values have changed significantly are considered for backup in the next iteration.
- A set *(states to update)* is maintained, containing states that need to be updated based on value changes in their successors.
- States whose successors' values do not change from one iteration to the next are skipped (not backed up), avoiding redundant computation.

This modification reduces the number of transition function calls considerably. With Modified Value Iteration, only **40,082** calls are made, which significantly lowers computation time and speeds up the algorithm. This is substantially fewer than the calls required by Policy Iteration and Value Iteration, as reported in Section 1. Watch a rollout of the optimal policy over the environment here.

2. Dynamic Programming

2.1. Online Knapsack Problem

2.1.1. Formulating Online Knapsack as a MDP

To solve the Online Knapsack Problem using Policy Iteration (PI) and Value Iteration (VI), we first need an **MDP formulation** of the problem. At each time step, the environment presents an item, and the agent chooses between two actions: *Accept* or *Reject*. If the agent accepts an item, its weight is added to the current knapsack weight and its value contributes to the total knapsack value. The objective is to maximize the total value while respecting the constraint that the knapsack's weight cannot exceed its maximum capacity.

We formulate the state of the MDP as:

$$s = \big[\,\text{knapsack weight } (N),\ \text{item index } (i),\ \text{time step } (t)\,\big]$$

where,

$$N \in [0, 200], \quad i \in [0, 200), \quad t \in [0, N]$$

The variable i encodes the information about the weight, value, and capacity constraint of the current item presented.
The variable t (time step) is important because it indicates when an episode terminates, thereby providing the MDP with information about how many decision steps remain. This helps the MDP in learning a policy that

accounts for the horizon of the problem.

In this problem, the transition dynamics assume that all items have an equal probability of appearing at the next time step. This assumption is crucial, as it allows us to express the value function update rule for each state in both policy iteration and value iteration.

2.1.2. Policy Iteration for Knapsack MDP

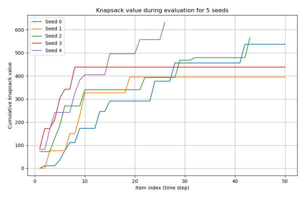
The implementation of the Policy Iteration algorithm for the Online Knapsack Problem can be seen in the code. As shown in Table 3, the total value of the knapsack is obtained using the optimal policy derived from Policy Iteration. The same result is illustrated in Figure 2(a).

Furthermore, the three heatmaps provide a visualization of the state space and the corresponding value function after convergence. Each heatmap places a different variable on the x-axis to highlight how the optimal value function varies across states, while the y-axis consistently represents the current knapsack weight. All heatmaps are generated for the initial time step ($t = 0$):

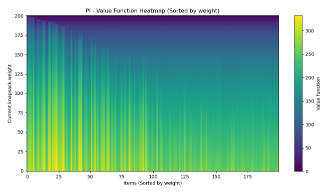
- **Sorted weights on X axis:** Figure 2(b) shows the state space with item weights sorted along the x-axis. We observe that the value function is higher for states corresponding to smaller item weights, indicating that the optimal policy is more likely to accept lighter items, as they contribute more effectively to increasing the overall knapsack value.
- **Sorted values on X axis:** Figure 2(c) shows the state space with item values sorted along the x-axis. We observe that the value function is higher for states with larger item values, which indicates that the optimal policy favors selecting items that contribute greater reward to the knapsack.
- **Sorted weight to value ratio on X axis:** Figure 2(d) presents the state space with the weight-to-value ratio of items on the x-axis. In this case, the value function is higher for items with a lower weight-to-value ratio, reflecting the optimal policy's tendency to prefer items that provide greater value relative to their weight.

Seed	Value Iteration	Policy Iteration
0	540	540
1	400	400
2	570	570
3	440	440
4	630	630

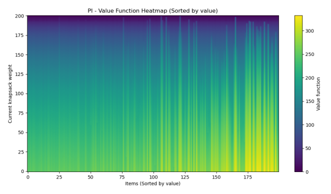
Table 3: Comparison of total knapsack values obtained by Value Iteration and Policy Iteration.



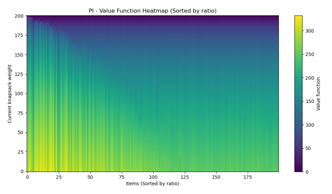
(a) Knapsack value over time steps



(b) Sorted by weight on x axis



(c) Sorted by value on x axis



(d) Sorted by weight to value ratio

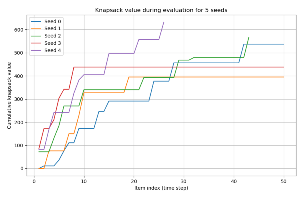
Figure 2: Plots for Policy iteration.

2.1.3. Value Iteration for Knapsack MDP

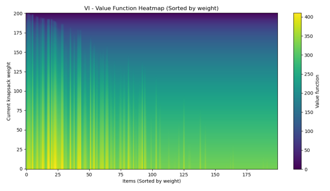
The implementation of the Value Iteration algorithm for the Online Knapsack Problem can be seen in the code.

Table 3 presents the total knapsack value obtained by the optimal policy using Value Iteration. Figure 3(a) illustrates the total knapsack value across time steps. Figure 3(b) shows the heatmap of the value function with item weights sorted along the x-axis. Figure 3(c) presents the heatmap with item values sorted on the x-axis, while Figure 3(d) shows the heatmap with the weight-to-value ratio sorted on the x-axis.

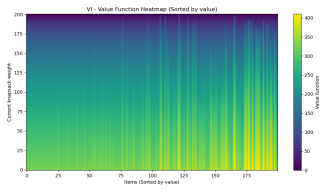
The plots and knapsack values are identical for both Policy Iteration and Value Iteration, since both methods converge to the same optimal policy. Therefore, for a fixed random seed, they produce identical results.



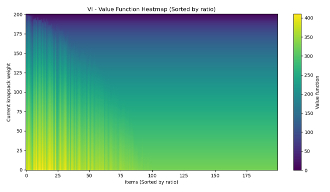
(a) Knapsack value over time steps



(b) Sorted weights on x axis



(c) Sorted Values on x axis



(d) Sorted weight to value ratio

Figure 3: Plots for Value Iteration.

2.1.4. VI for different time steps

Figure 4 compares the heatmaps of the value function across different numbers of Value Iteration steps for all three cases:
(i) item weights sorted on the x-axis,
(ii) item values sorted on the x-axis, and
(iii) weight-to-value ratio sorted on the x-axis.

2.2. Portfolio Optimization

2.2.1. MDP Formulation

We model the trading problem as a Markov Decision Process (MDP). The main components are defined as follows:

- **States:** A state s_t at time step t is represented as a tuple

  $$s_t = (t, c_t, h_t),$$

  where $t \in \{0, 1, \ldots, T\}$ is the current time step, c_t is the amount of cash held, and h_t is the number of units of the asset (*Unobtainium*) held. The state space therefore combines time, cash balance, and asset holdings.

- **Actions:** At each time step the agent may choose an action $a_t \in \{-2, -1, 0, 1, 2\}$ corresponding to selling two units, selling one unit, doing nothing, buying one unit, or buying two units. Each transaction incurs a fixed cost of one unit of cash.

- **Transitions:** The asset price at each step is determined externally from the given price sequence. After taking an action, the agent's cash and holdings are updated accordingly:
  - If the agent buys, cash decreases by the purchase cost and holdings increase.
  - If the agent sells, cash increases from the sale and holdings decrease.

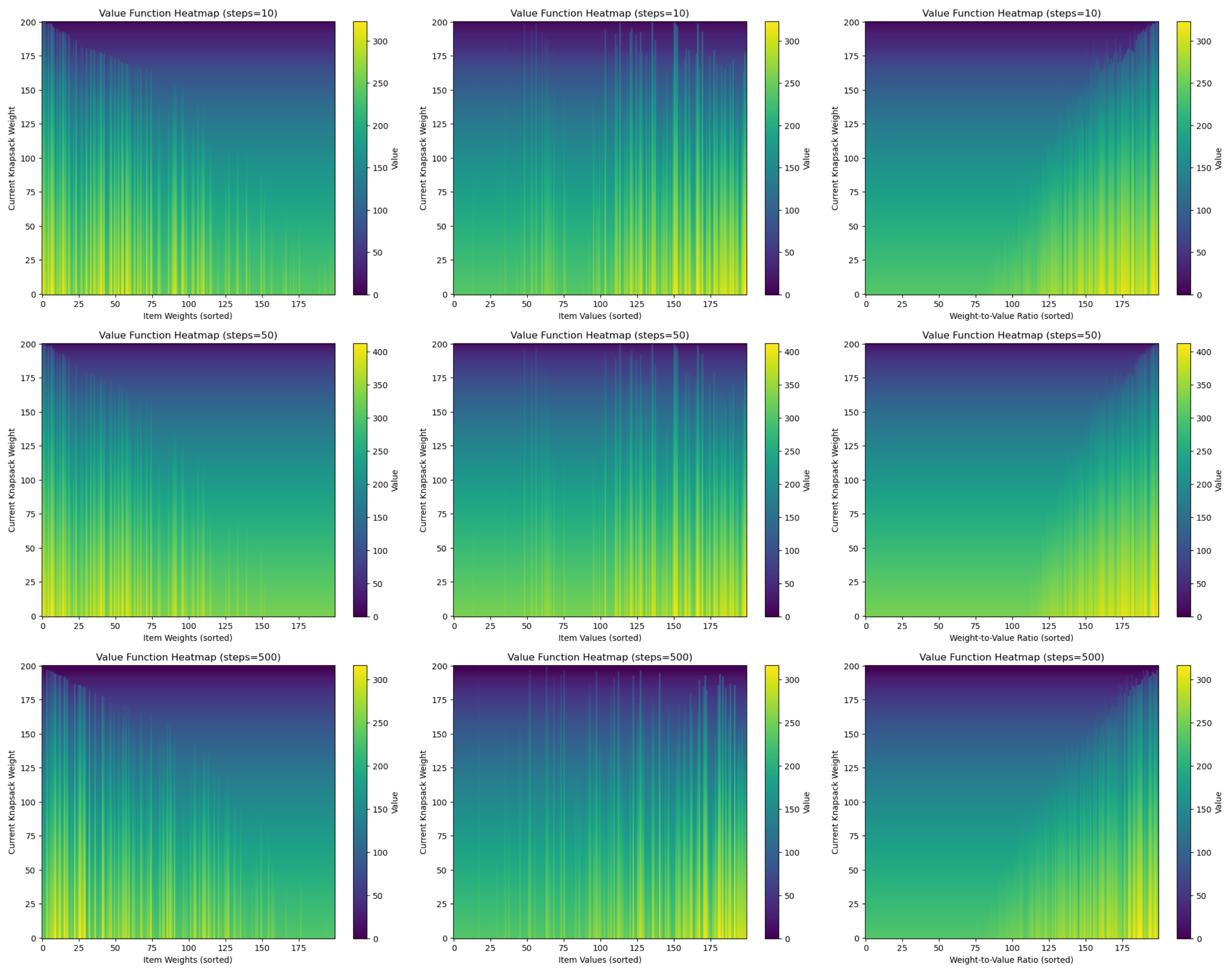


Figure 4: Heatmaps for value function with 3 different step size.

- If the agent does nothing, both cash and holdings remain unchanged.

The episode terminates if the holdings ever become negative, or after the final time step T.

2.2.2. Policy Iteration

The implementation of the Policy Iteration algorithm for the Portfolio optimization is available in the code repository.
Table 4 presents a comparison of execution times between Policy Iteration and Value Iteration.
Figure 5 illustrates the wealth evolution across an episode for all three configurations, along with the convergence plot of the Policy Iteration algorithm for $\gamma = 1$ and Figure 6 shows it for $\gamma = 0.999$

Configuration	$\gamma = 1.0$		$\gamma = 0.999$	
	VI Time (s)	PI Time (s)	VI Time (s)	PI Time (s)
Config 1	1.0545	2.1933	0.9591	2.0406
Config 2	0.7548	1.6972	0.6267	1.3152
Config 3	0.7177	1.7451	0.5947	1.4775

Table 4: Execution times of Value Iteration (VI) and Policy Iteration (PI) for the three configurations under different discount factors γ.

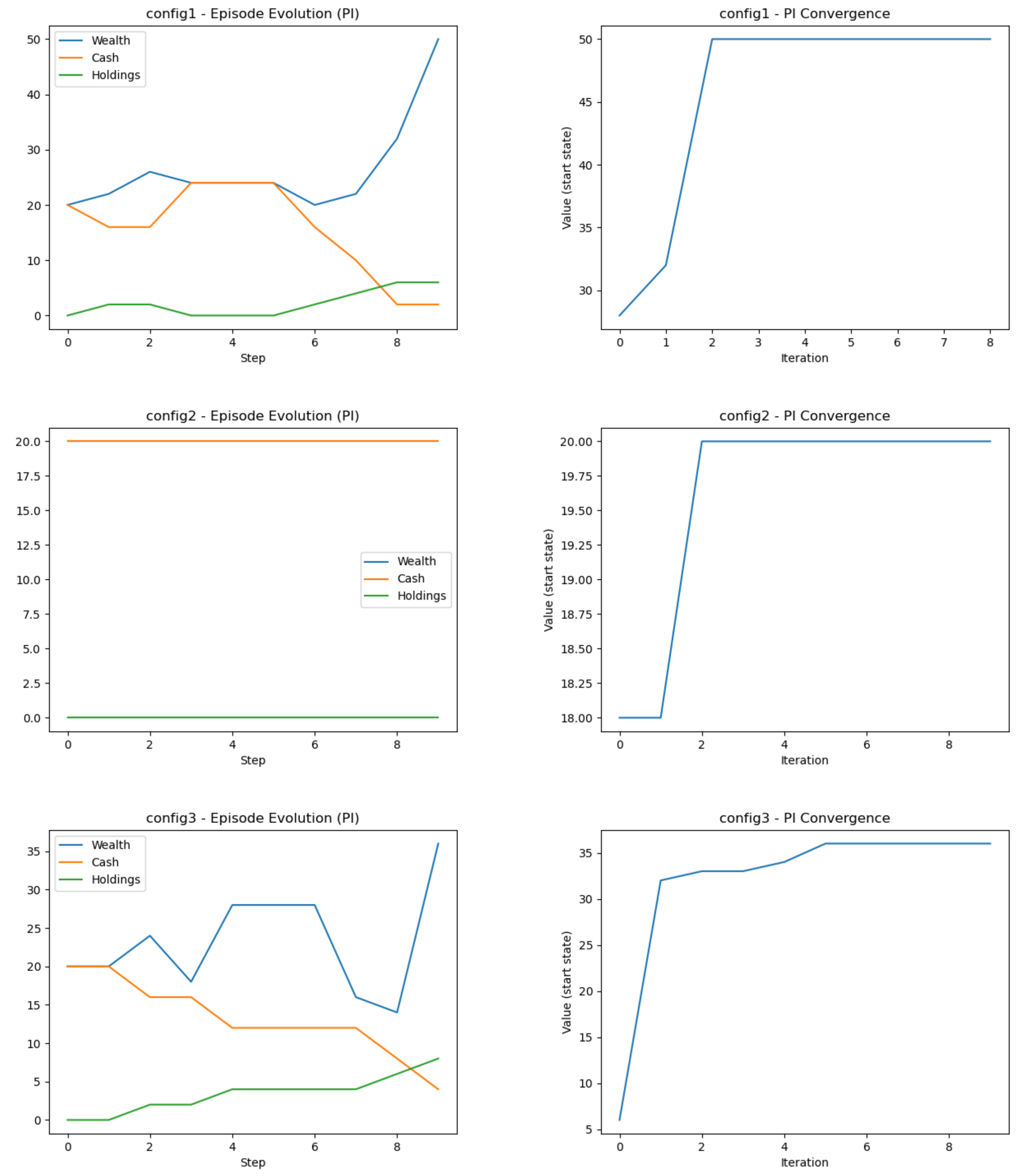


Figure 5: Policy Iteration results across three price configurations. Left column: wealth evolution across an episode. Right column: convergence plots showing value function differences during training. $\gamma = 1$

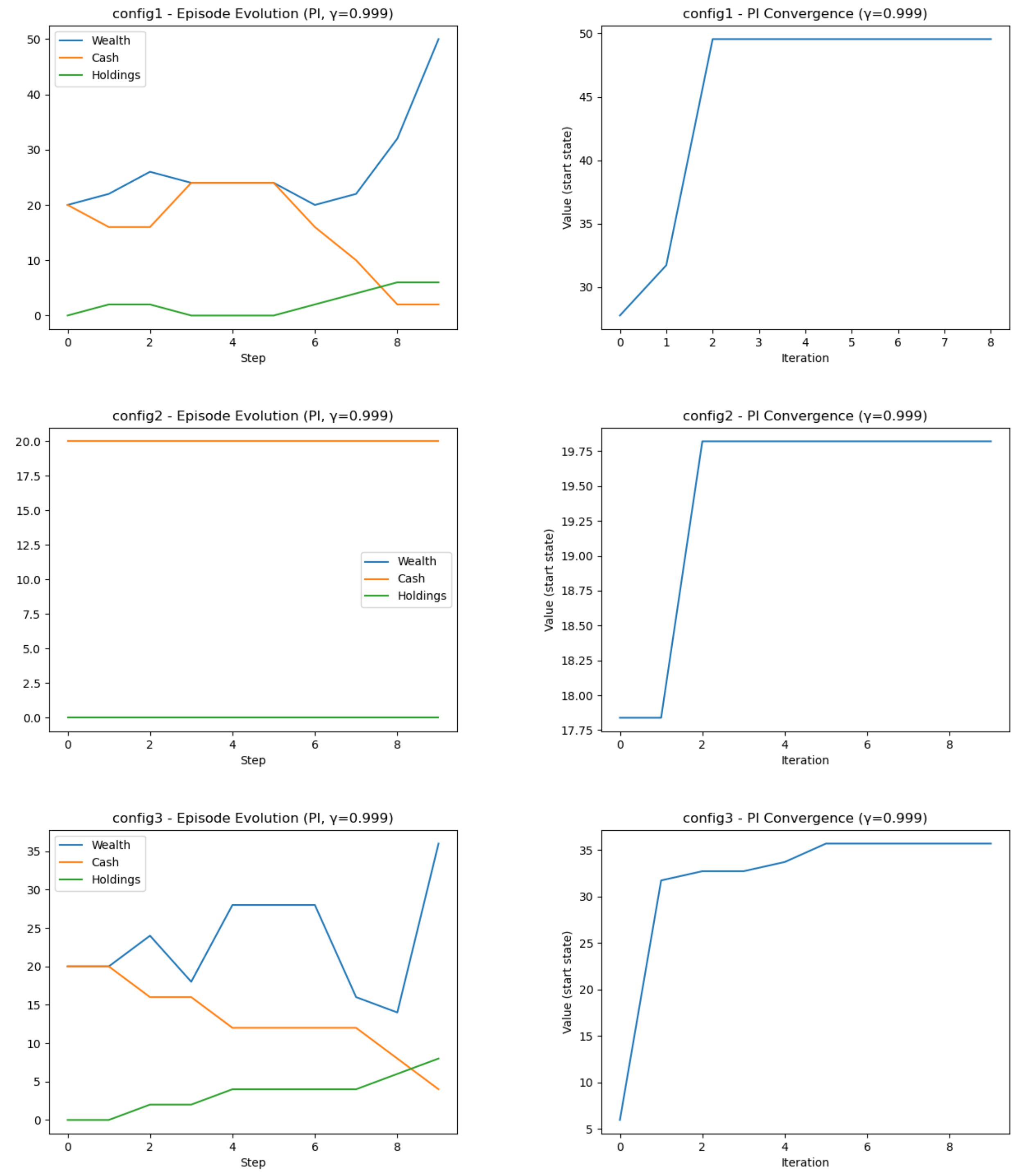


Figure 6: Policy Iteration results across three price configurations. Left column: wealth evolution across an episode. Right column: convergence plots showing value function differences during training. $\gamma = 0.999$

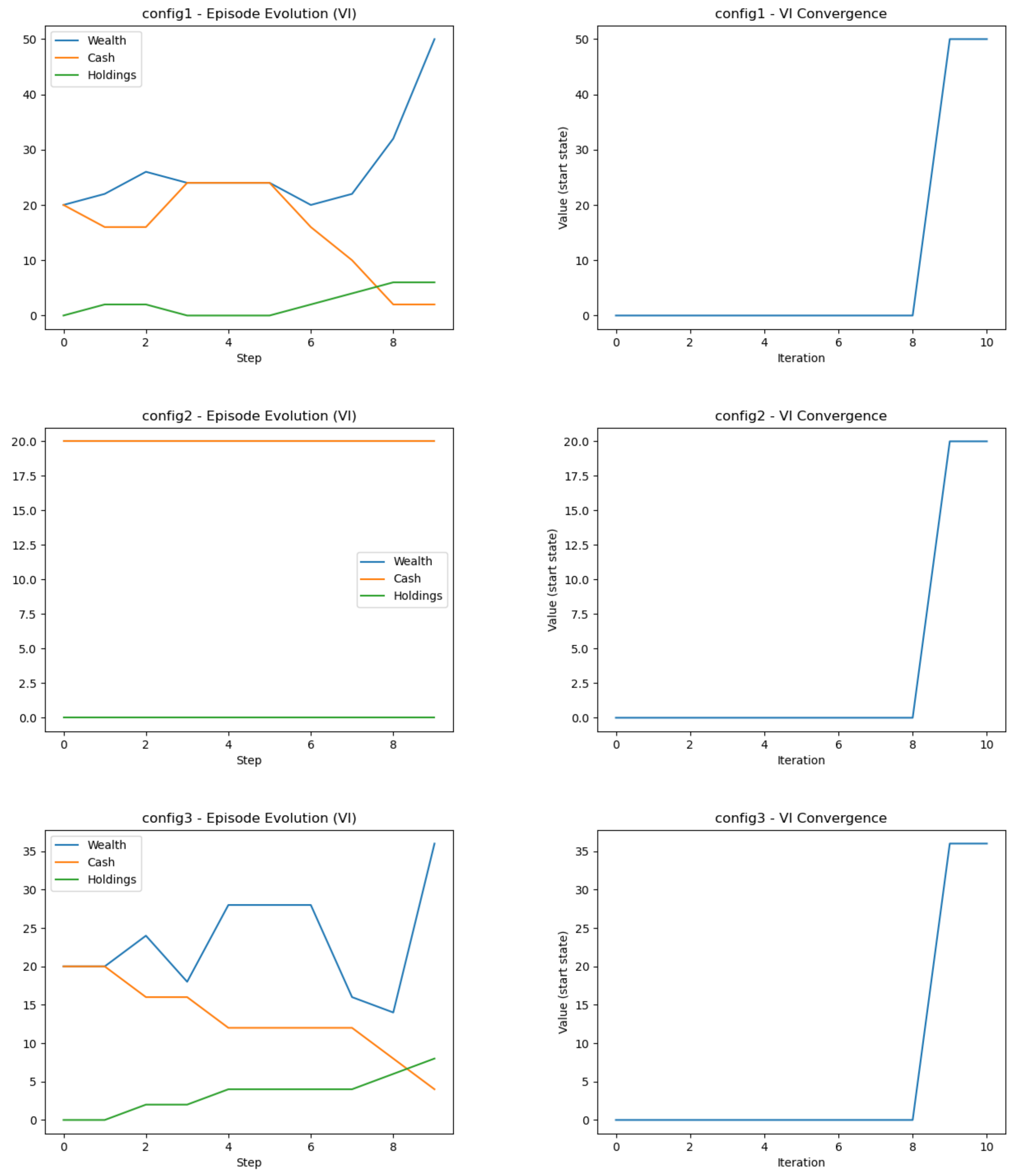


Figure 7: Value Iteration results across three price configurations. Left column: wealth evolution across an episode. Right column: convergence plots showing value function differences during training. $\gamma = 1$

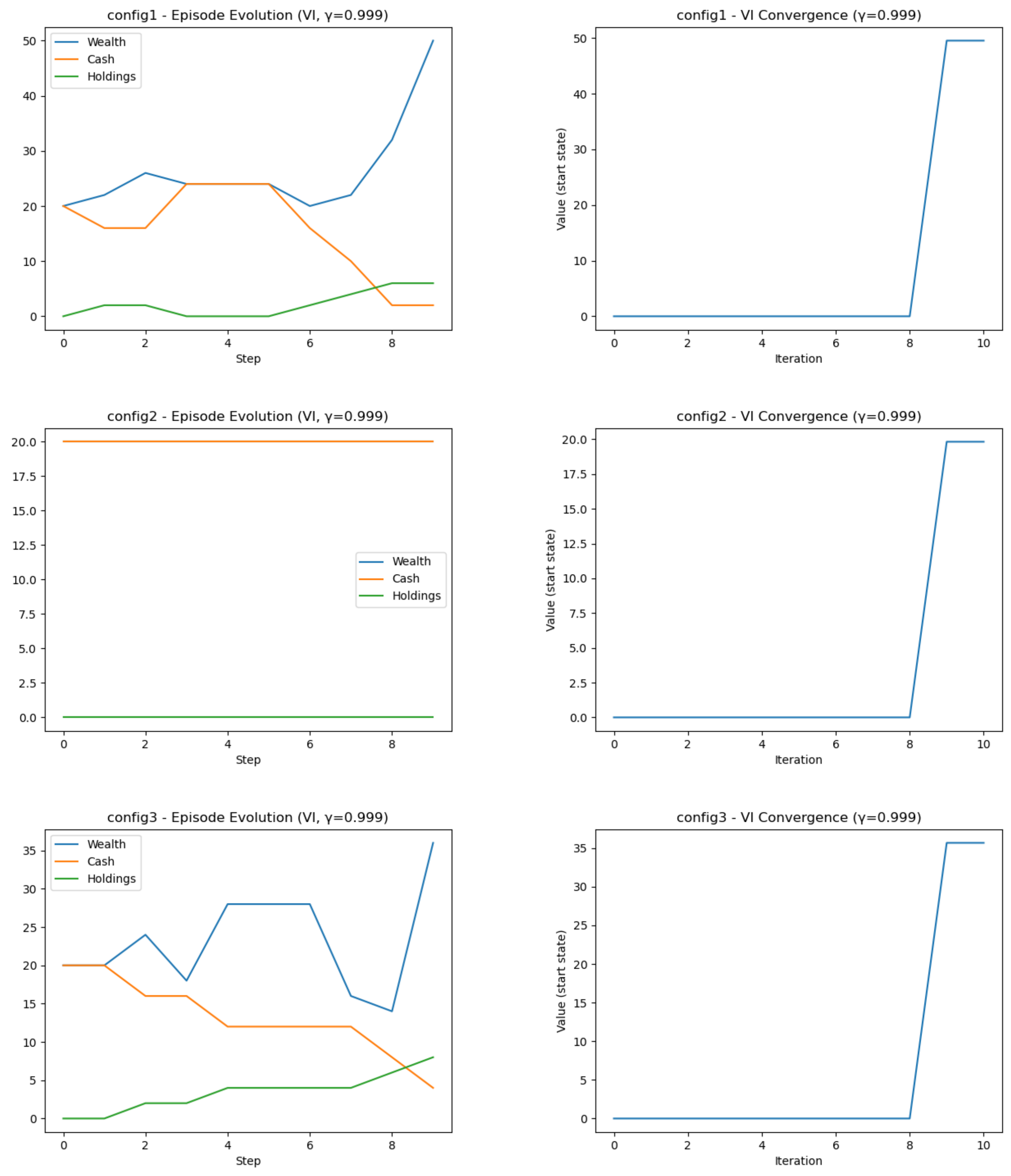


Figure 8: Value Iteration results across three price configurations. Left column: wealth evolution across an episode. Right column: convergence plots showing value function differences during training. $\gamma = 0.999$

2.2.3. Value Iteration

The implementation of the Value Iteration algorithm for the Portfolio optimization is available in the code repository.
Table 4 presents a comparison of execution times between Policy Iteration and Value Iteration.
Figure 7 illustrates the wealth evolution across an episode for all three configurations, along with the convergence plot of the Value Iteration algorithm for $\gamma = 1$ and Figure 8 shows it for $\gamma = 0.999$ In all three episodes, we observe that both Policy Iteration and Value Iteration achieve the same final wealth, indicating that both algorithms converge to the same optimal policy.
The convergence plots reveal an interesting difference between the two algorithms. In Value Iteration, the wealth at the initial iteration remains unchanged for most of the iterations and then jumps sharply to the optimal value in the last few iterations. In contrast, Policy Iteration shows a gradual increase in wealth across iterations, reflecting the incremental improvements in the policy evaluation and policy improvement steps. This behavior illustrates the different mechanisms by which the two algorithms propagate value information through the state space, even though they ultimately converge to the same optimal policy.

2.2.4. Price Variance and Its Effect on Policy Iteration

In the environment, the price of the asset at each time step can be stochastic. This is modeled by sampling the price from a normal distribution:

$$\text{price}_t \sim \mathcal{N}(\mu_t, \sigma^2)$$

where μ_t is the mean price at time step t and σ^2 is the variance.

Effect on Policy Iteration:

- With $\sigma^2 = 0$, the environment is deterministic, and each state-action pair leads to a single predictable next state. Policy Iteration converges quickly.
- With $\sigma^2 > 0$, each state-action pair can lead to multiple possible next states depending on the sampled price. The value of a state becomes the *expected value over all possible next prices.*
- During policy evaluation, the algorithm must average over these possible outcomes (or approximate the expectation via multiple samples), which increases computational cost.
- As a result, Policy Iteration takes longer to converge in environments with stochastic asset prices compared to deterministic ones.

To incorporate stochasticity in prices, we extend the state space by explicitly modeling the price as a random variable. For each timestep t, we know the expected price μ_t and assume Gaussian noise with variance $\sigma^2 = 1$. We discretize this Gaussian distribution into a finite set of price states.

Concretely, for every timestep we construct a *Gaussian probability window* centered at μ_t. Within this window, we generate (e.g., 100) discrete price points, and for each such point we compute its probability mass (via the Gaussian pdf/CDF). These discrete prices and their associated probabilities are stored in lists.

During policy iteration, when performing value function updates, we use these price supports and probabilities to compute the expected next-state value:

$$V(s) \;\leftarrow\; \max_a \sum_{p \in \text{PriceSupport}_t} \Pr(p) \cdot \Big(r(s, a, p) + \gamma V(s') \Big)$$

This way the stochastic effect of Gaussian-distributed prices is fully captured in the Bellman updates.

Figure 9 shows the evolution of Δ (the maximum value difference across states) while training with policy iteration.

Figure 10 illustrates the trajectory of wealth accumulation over time along with the evolution of cash and holdings.

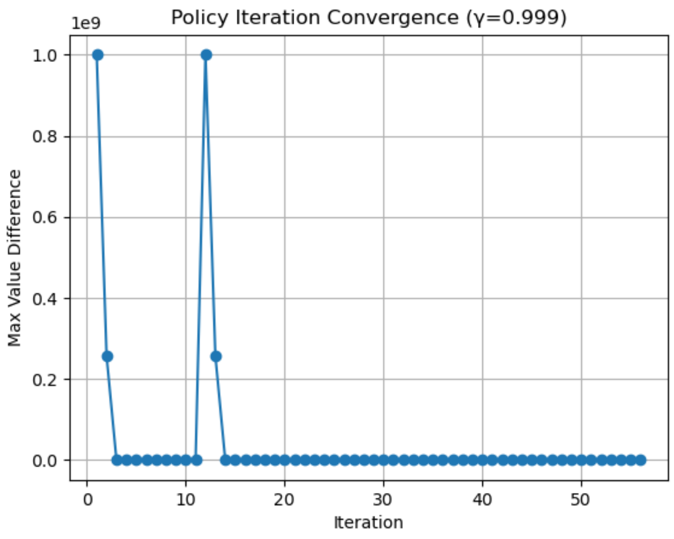


Figure 9: Deltas during policy evaluation plotted against iteration count. The logarithmic scale highlights how the maximum value difference Δ decreases rapidly as the policy evaluation converges.

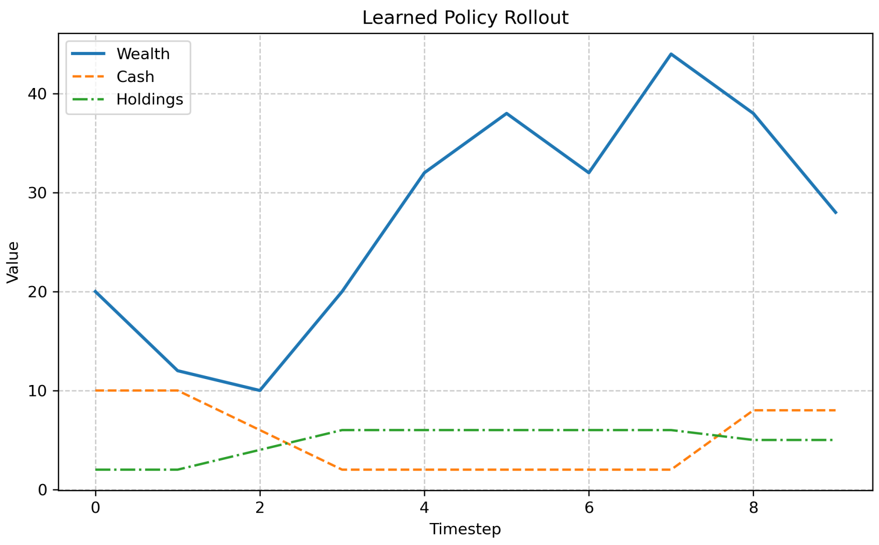


Figure 10: Evolution of wealth, cash, and holdings when executing the learned policy. The plot illustrates how the agent allocates resources dynamically to maximize final wealth.