HOMEWORK 2

Loïc Jannin

September 2023

1 Problem 1

Consider the Snakes and Ladders game (single-player version) that was covered in class. We conducted 5000 simulations to analyze the game and answer two key questions:

1.1 Probability Distribution Function (PDF) of Finishing the Game in X Dice Rolls

To understand the distribution of the number of dice rolls required to finish the game, we ran the simulation 5000 times. For each simulation, we recorded the number of dice rolls needed to reach the finish line at position 100. The histogram below illustrates the Probability Distribution Function (PDF) of finishing the game in X dice rolls:

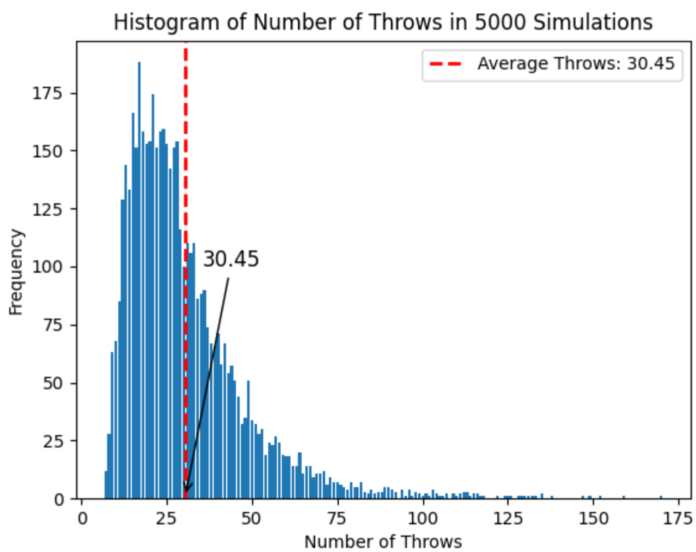

Figure 1: Histogram of Number of Throws in 5000 Simulations

In the histogram, the x-axis represents the number of throws (X), and the y-axis represents the frequency (i.e., how often that number of throws occurred) in the 5000 simulations. The red dashed line indicates the average number of throws required to finish the game, which is approximately 30.45

1.2 Expected Number of Dice Rolls Needed to Finish the Game

The average number of dice rolls needed to finish the game, also known as the expected value, was calculated from the simulation results. We found that the expected number of throws is approximately 30.45 based on the 5000 simulations. This means that, on average, it takes around 30.45 dice rolls to complete the game in our simulation.

2 Problem 2

Optimality of Policy in Markov Decision Processes

Consider two Markov Decision Processes, M1 and M2, with corresponding reward functions $R1$ and $R2$. We are given that M1 and M2 are identical except that the rewards for R2 are shifted by a constant from the rewards for R1, i.e., for all states s, $R2(s) = R1(s) + c$, where c does not depend on s.

We aim to prove that the optimal policy must be the same for both Markov Decision Processes.

Definition: Markov Decision Process (MDP).

A Markov decision process is a 4-tuple (S, A, P_a, R_a), where:

- S is a set of states called the state space.
- A is a set of actions called the action space.
- $P_a(s, s')$ is the probability that action a in state s at time t will lead to state s' at time $t+1$.
- $R_a(s, s')$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state s' due to action a.

The state and action spaces may be finite or infinite, for example, the set of real numbers. Some processes with countably infinite state and action spaces can be reduced to ones with finite state and action spaces.

Definition: Policy Function.

A policy function π is a (potentially probabilistic) mapping from the state space S to the action space A.

Definition: Total cumulative rewards.

We define the return G_t from state S_t as :

$$G_t = \sum_{n=t+1}^{\infty} \gamma^{i-t-1} R_i \tag{1}$$

Definition: State-value function under a policy π.

$$V^{\pi}(s) = E_{\pi}(G_t | s_t = s) = E_{\pi}\left(\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| s_t = s\right) \tag{2}$$

2.1 Proof

Let π_1^* be the optimal policy for M1 and π_2^* be the optimal policy for M2. We need to show that $\pi_1^* = \pi_2^*$. For a given state s, we have π_1^* and π_2^* respectively given by the following equations:

$$V^{\pi_1^*}(s) = \max_{\pi_1} V^{\pi_1}(s) \tag{3}$$

$$V^{\pi_2^*}(s) = \max_{\pi_2} V^{\pi_2}(s) \tag{4}$$

Since $R2(s) = R1(s) + c$ for all states s, we can substitute $R2$ into the equation for M2:

$$V^{\pi_2^*}(s) = \max_{\pi 2} \sum_{k=0}^{\infty} \gamma^k E_{\pi 2}[R1(s_{t+k+1}) + c | s_t = s]$$

We can split the sum into two parts:

$$V^{\pi_2^*}(s) = \max_{\pi 2} \sum_{t=0}^{\infty} \gamma^t E_{\pi 2}[R1(s_t) | s_0 = s] + \sum_{t=0}^{\infty} \gamma^t c$$

Notice that the second term, $\sum_{t=0}^{\infty} \gamma^t c$, is a constant that does not depend on the policy $\pi 2$. Since the optimal policy is the one that maximizes the expected cumulative reward, this constant term does not affect the policy's optimality.

Hence, we have proven that the optimal policy must be the same for both Markov Decision Processes, M1 and M2.

3 Problem 3

Optimizing Frog's Escape in an MDP

3.1 Problem Statement:

Consider an array of $n+1$ lilypads on a pond, numbered 0 to n. A frog sits on a lilypad other than the lilypad numbered 0 or n. When on lilypads i ($1 \leq i \leq n-1$), the frog can croak one of two sounds, A or B.

- If it croaks A when on lilypad i ($1 \leq i \leq n-1$), it is thrown to lilypad $i-1$ with probability $\frac{i}{n}$ and is thrown to lilypad $i+1$ with probability $\frac{n-i}{n}$.
- If it croaks B when on lilypad i ($1 \leq i \leq n-1$), it is thrown to one of the lilypads $0, \ldots, i-1, i+1, \ldots, n$ with uniform probability $\frac{1}{n}$.

A snake, located on lilypad 0, will eat the frog if the frog lands on lilypad 0. The frog can escape the pond (and hence, escape the snake!) if it lands on lilypad n.

3.2 Modeling:

To model this problem as an MDP (Markov Decision Process) and derive the Optimal Action Value Function Q^*, we need to define the following components:

- **State Space (S)**: The state space will represent the lilypad on which the frog is currently located. It ranges from 0 to n. States 0 and n are terminal states.
- **Action Space (A)**: The action space corresponds to the frog's choice of croak (A or B) when on lilypads 1 to $n-1$. $A = \{a, b\}$
- **Transition Probabilities (P)**: We define the transition probabilities based on the frog's croak choice and the movement probabilities provided in the problem statement. $P_a(s, s')$ is the probability that action a in state s at time t will lead to state s' at time $t+1$. For all $i \in \{1, 2, \ldots, n-1\}$ and for all $j \neq i \in \{0, 1, 2, \ldots, n\}$, we have:

$$P_a(i,j) = \begin{cases} \frac{i}{n}, & \text{if } j = i-1 \\ \frac{n-i}{n}, & \text{if } j = i+1 \\ 0, & \text{else} \end{cases}$$

$$P_b(i,j) = \frac{1}{n}$$

- **Rewards (R)**: The rewards assign values to different states and actions. In this case, we aim to maximize the probability of reaching lilypad n while avoiding lilypad 0 (being eaten by the snake). Different types of rewards can be considered. A natural approach would be to assign a reward of 0 to every lilypad from 1 to $n-1$, a reward of 1 to lilypad n, and a reward of -1 to lilypad 0. But during the second part of the problem, we will experiment with different reward functions and analyze the results.

With these components, we can formulate the MDP and solve for the Optimal Action Value Function Q^*.

3.3 Results:

Below are the results of a Python code that models the MDP and calculates Q^* for $n = 3, 10$, and 25, along with plots of $Q^*(s, a')$ as a function of the states of the MDP for each action a'. The code used is the code provided in class, but with some changes to play with the results. Additionally, this code plots the average cumulative reward of our trained agent. This code experiments with different hyperparameters to try and find the most efficient model for training our agent.

The parameters we're going to adjust are:

- The reward function.
- The value of alpha.
- The value of epsilon.
- The value of gamma.

As a reminder, the value of epsilon dictates the tendency of our agent to take a random action (here, to transition to a random state through action B) instead of following the best policy given the Q function. Alpha is the learning rate, generally set between 0 and 1. Setting the alpha value to 0 means that the Q-values are never updated, and nothing is learned. If we set alpha to a high value like 0.9, learning can occur quickly. Gamma is the discount factor set between 0 and 1. This models the fact that future rewards are worth less than immediate rewards.

3.3.1 Sensitivity to reward formulation.

First, we will examine the impact of the reward function. We propose four different reward functions:

1. **Reward 1:** State i gives $\frac{i}{n}$ as a reward.
2. **Reward 2:** Every state gives 0 except state n, which gives 1.
3. **Reward 3:** Every state gives 0 except state n, which gives 1, and state 0, which gives -1.
4. **Reward 4:** State i gives $1 - 1/i^2$ as a reward.

We will plot the optimal Q-values for each state and action, and then we will plot the average cumulative rewards over 10 simulations. We first use hyperparameter scheme 1, which means that the learning rate alpha linearly decreases from 1 to 0 during training with each episode. Then we use hyperparameter scheme 2 which means that the learning rate alpha linearly decreases from 1 to 0 during training with each episode.

Remarks:

1. The reward scheme 3 with hyperparameter scheme 2 seems to not have reached its optimal value in the previous simulations. We performed additional simulations with 2000 runs to investigate further; the results are far better (Figure 8).

2. We observe that the reward functions significantly impact the learning process of the agent. Reward scheme 2 led to the fastest convergence in both hyperparameter schemes (Figure 3 and 4).

3. Hyperparameter scheme 2, where the learning rate alpha linearly decreases from 1 to 0 during training with each episode, and epsilon keeps its value, shows promising results in terms of convergence (Figure 1 vs 2, Figure 7 vs. 8). The choice of epsilon, which affects the exploration vs. exploitation trade-off, plays a crucial role in the agent's learning. Further experimentation may help fine-tune this hyperparameter.

4. The results from the additional 2000 simulations for reward scheme 3 with hyperparameter scheme 2 show improved convergence, suggesting that more simulations can provide better insights into the agent's performance.

5. Future work may involve exploring different combinations of hyperparameters to optimize the training process further.

6. Understanding the impact of reward functions on the agent's behavior is essential for designing effective reinforcement learning systems.

7. It is evident that in reward function 3, where state 0 incurs a negative reward, the Q-value of action A at state 1 reaches its lowest point. This reflects the significant influence of the adverse consequence associated with state 0's negative reward and, consequently, the frog's cautious disposition under this reward scheme.

8. When examining scenarios where the "core states" receive positive rewards (as in reward functions 1 and 4), a noteworthy observation emerges. In such cases, the frog exhibits a greater propensity to employ croak A, as remaining in the pond yields favorable rewards, as illustrated in Figures 1, 2, 7, and 8.

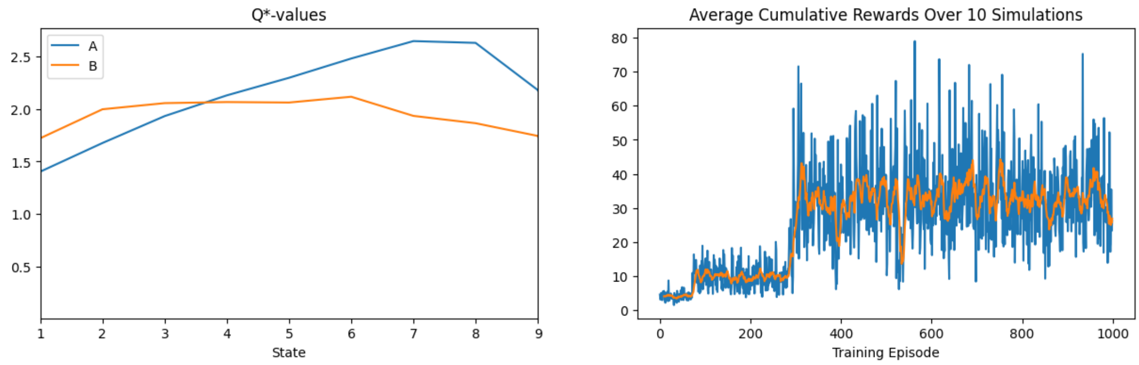

Figure 1: Results for reward 1, hyperparameters scheme 2.

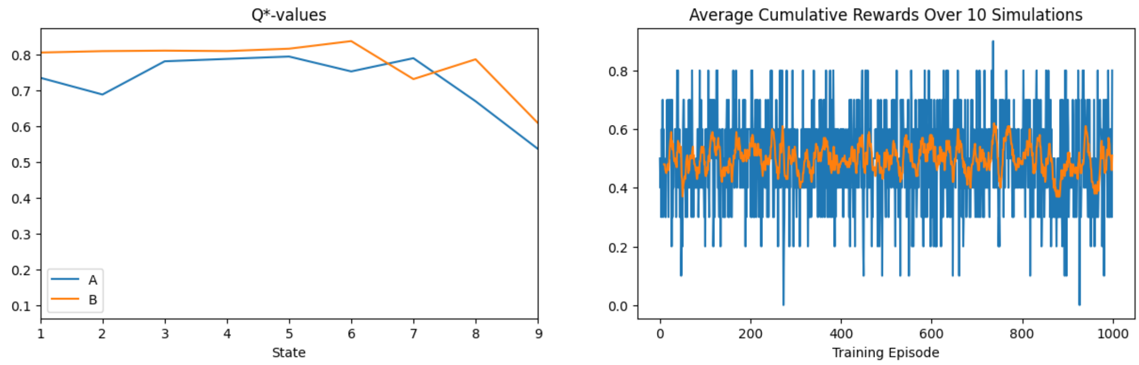

Figure 2: Results for reward 2, hyperparameters scheme 1.

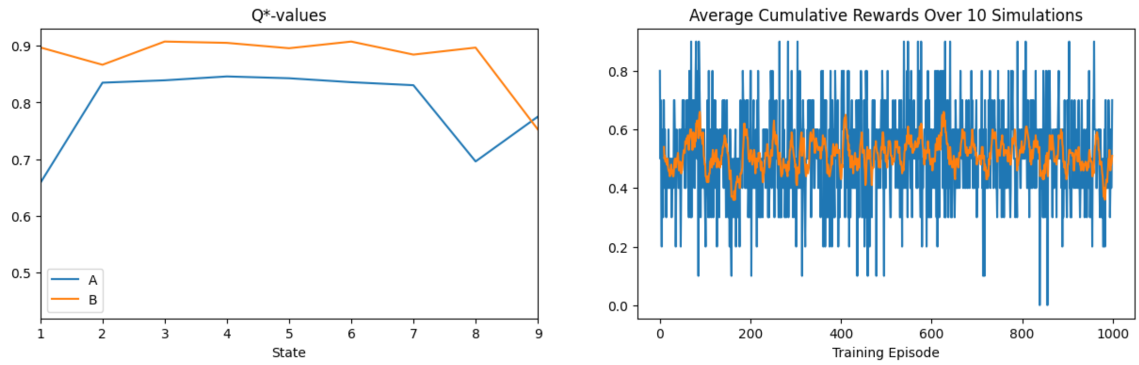

Figure 3: Results for reward 2, hyperparameters scheme 2.

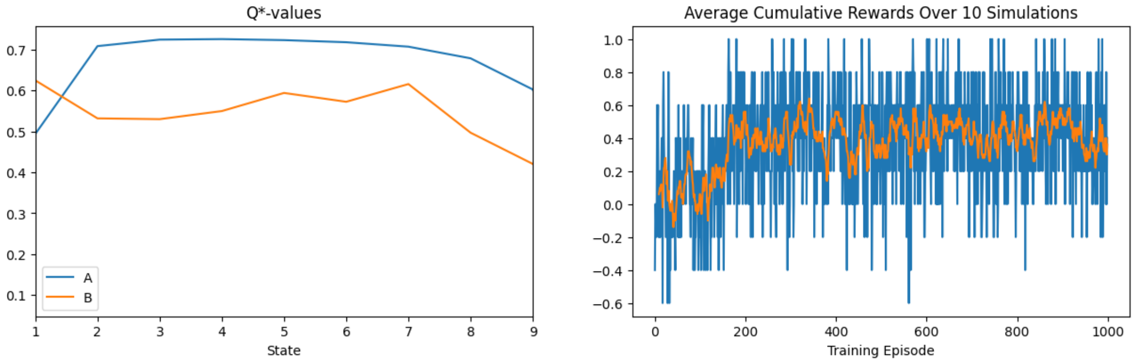

Figure 4: Results for reward 3, hyperparameters scheme 1.

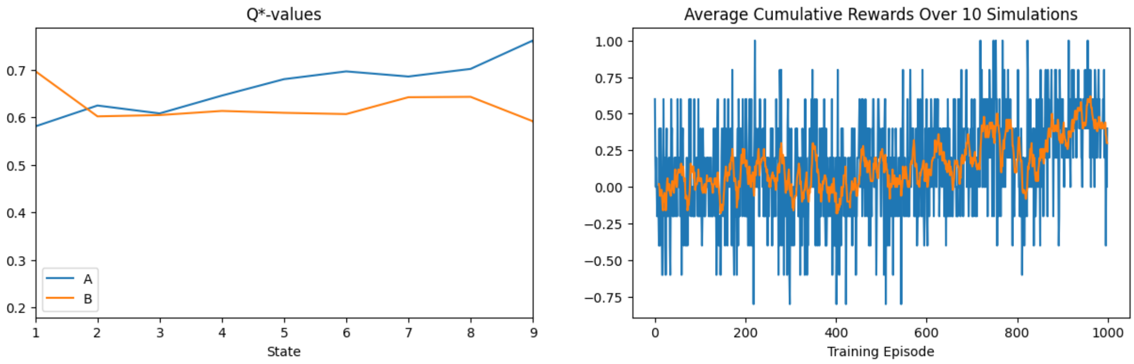

Figure 5: Results for reward 3, hyperparameters scheme 2.

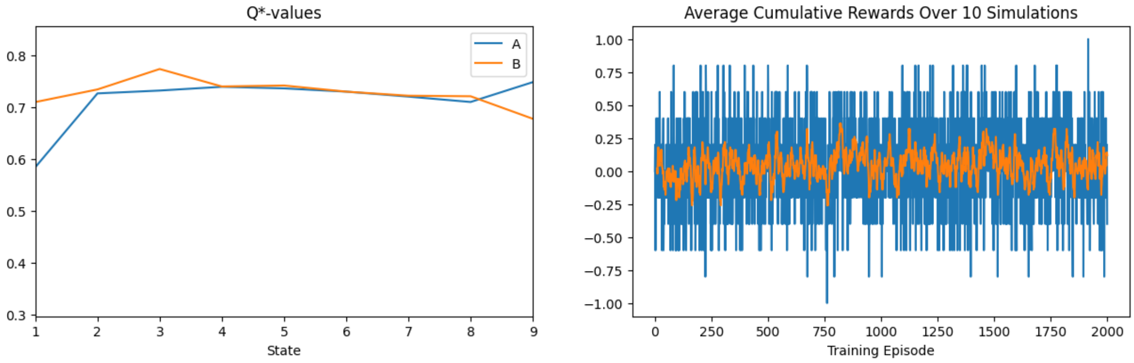

Figure 6: Results for reward scheme 3, hyperparameters scheme 2, and 2000 simulations.

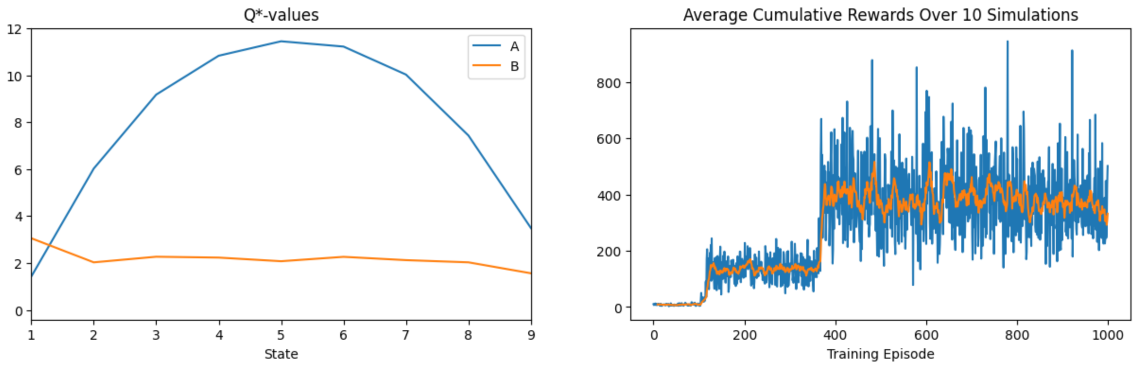

Figure 7: Results for reward scheme 4, hyperparameters scheme 1.

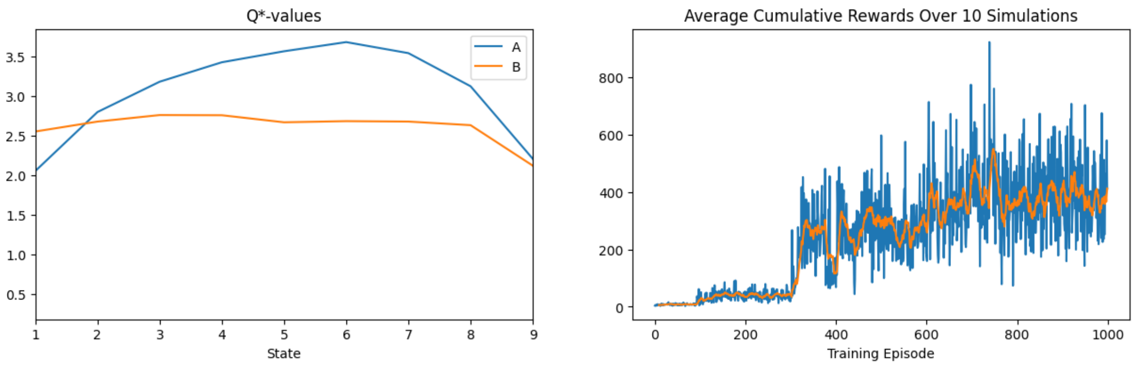

Figure 8: Results for reward scheme 4, hyperparameters scheme 2.

4 Problem 4

Using the depth 5 Amazon data from LOBSTER, we formulated an MDP (Markov Decision Process) to find an optimal execution strategy of a large client buy order of size X using a model-free RL (Reinforcement Learning) method. In this market replay regime, transition probabilities are implied by historical data with the assumption of no market impact. The execution agent makes decisions every 10 seconds. We provide clear mathematical notation for the proposed state space, action space, and rewards of the MDP.

4.1 State Space

Each state $s \in S$ is a vector of attributes (state variables) that describes the current configuration of our system. We use the term 'state' rather than 'observed state' due to the complexity of the environment, but we will assume that the Markov property is respected.

- State Variable 1: **Remain Inventory**. Noted "I". Gives the number of share left we have to buy.
- State Variable 2: **Imbalance**. Noted "Imb". Given by $Imb = \frac{BidVolume}{AskVolume}$
- State Variable 3: **Volatility**. Noted σ. Given by $\sigma = \sqrt{\text{Annualization Factor} \times \text{Var(Log Returns)}}$ Computes the volability for the last 100 ticks.
- State Variable 4: **Spread.** Noted Sp, given by the formula: $Sp = AskPrice - BidPrice$
- State Variable 5: **Traded Volumes.** Noted V, gives the sum of the traded volume during the last 10s.
- State Variable 6: **Best Bid Price.**
- State Variable 7: **Best Ask Price.**
- State Variable 8: **Last event** Dummy variables that can take values 1;2;3;4;5 given the last event, lobster notation.
- State Variable 9: **10s Returns**
- State Variable 10: **1s Returns**

4.2 Action Space

The action space defines the possible actions that the execution agent can take at each time step. Actions represent different execution strategies or trading decisions. The action space A includes:

- Action a: **Buy Limit order**. Action 'a' involves the placement of a limit order for any remaining unexecuted shares at a price equal to the current bid price plus 'a', with a fixed volume of 10 per cent the total share volume to buy. This action effectively cancels any existing outstanding limit orders we may have, aligning with the practices supported by actual exchanges. It's worth noting that 'a' can be either a positive or negative value, with 'a=0' indicating an entry at the current bid price. In reality, the action state is not limited to a single action 'a'; we can model a continuous action space with all possible values of 'a.' However, a more straightforward approach would be to discretize 'a', and allowing our agent to chose the different buy volumes (choice between 1 per cent, 10, 50) . Another important point is that there's no need for a separate 'market order' action, as a market order is equivalent to choosing a = spread.

4.3 Rewards

The rewards in the MDP play a critical role in providing immediate feedback to the agent based on its actions and the state of the environment. These rewards can be strategically designed to motivate the agent towards the ultimate goal of executing the large client buy order optimally. Since our primary objective is to successfully execute a large buy order, our immediate rewards essentially represent the cash outflows resulting from any (partial) execution of the limit order placed.

To facilitate meaningful comparisons across various training episodes with different buy volumes, it becomes imperative to 'normalize' the rewards for the purpose of comparison. Drawing inspiration from the work of Nevmyvaka, Feng, and Kearns, we will compare our total cash outflow to the ideal cash flow scenario, where we hypothetically purchase all the shares at the initial time midprice.
Let $CF(s,a,s')$ be the outflow realized by the transition betwen state s' and s given action a. Let $Id(V,t_0)$ be the ideal cash flow scenario. The reward function is defined by:

$$R(s,a,s') = \frac{CF(s,a,s')}{Id(V,t_0)}$$

4.4 Results

The selection of state variables is critical for training an effective RL agent. Among the state variables examined, imbalance, volatility, and spread are expected to be particularly useful. Imbalance provides insights into order book dynamics, while volatility and spread impact trading decisions and execution strategies. To address the question of the usefulness of each state variable, we could examine the correlation between each state variable and positive events for our agent, such as identifying favorable future stock prices for our agent to purchase shares. However, a more explicit method for assessing the utility of each variable would involve constructing a decision tree.

Further experimentation and fine-tuning of the MDP components will be necessary to achieve optimal execution results. Predicting which state variables will be the most useful for our MDP without running simulations can be quite challenging. Additionally, it's worth noting that, given the reward function, certain state variables may have more influence than others.(All codes are in annex)

4.5 Plots

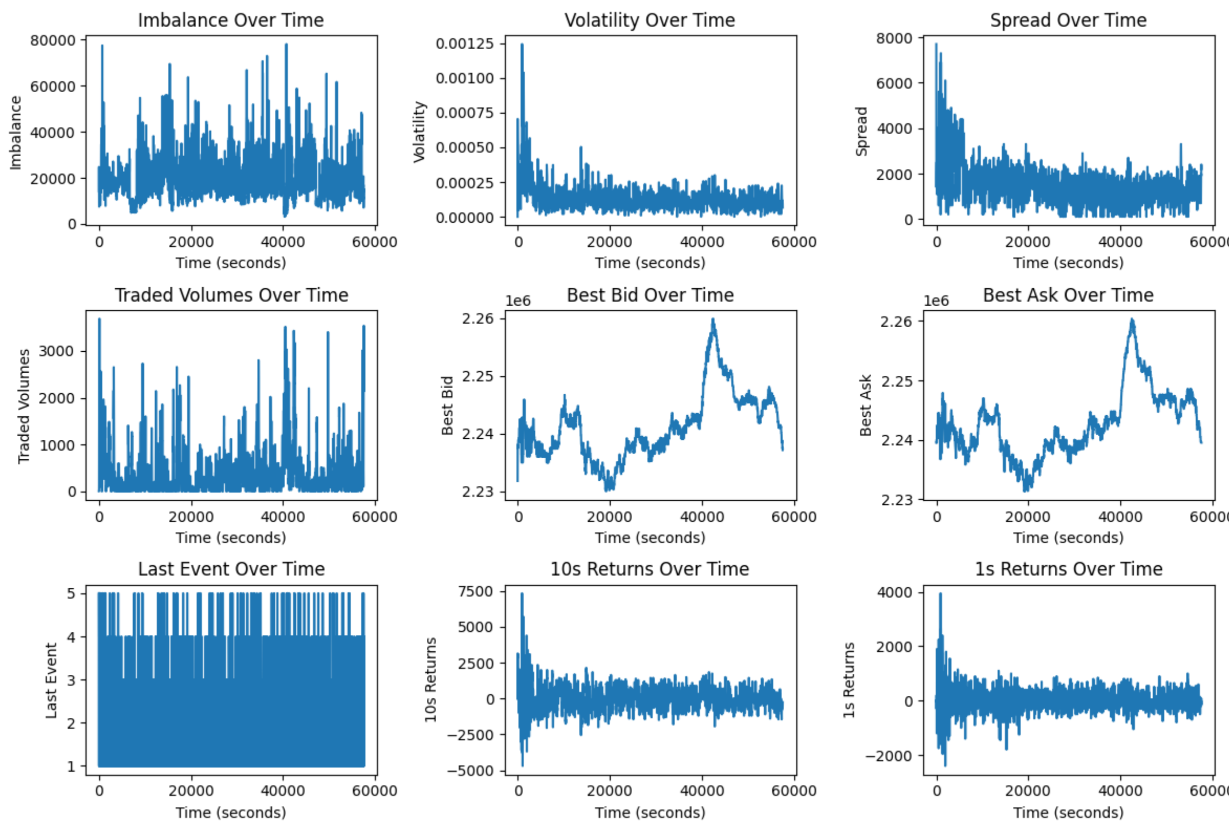

Figure 2: State Space variables as function of time

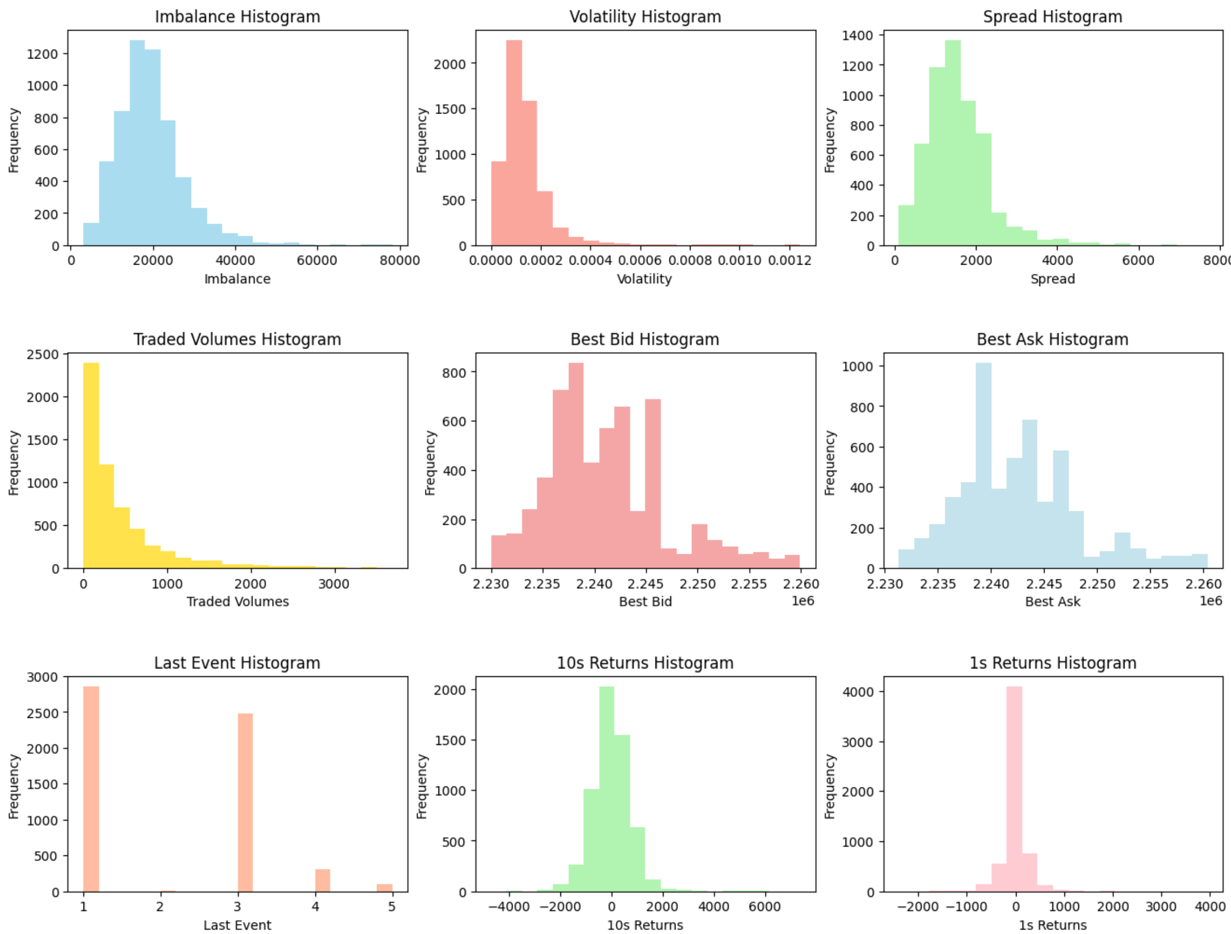

Figure 3: Histogram of States space variables

5 Problem 5

We propose a Markov Decision Process (MDP) formulation for designing an execution agent with deadlines. The goal is to find an optimal execution strategy for executing a large client buy order of size X by time T, ensuring that the entire order is executed by the specified deadline. In this problem, we are required to express the MDP with clear mathematical notation, including the state space, action space, rewards, and transition probabilities model. Once again, this work is inspired by the paper "Reinforcement Learning for Optimized Trade Execution", shared in class.

5.1 State Space

The state space for this MDP consists of 5 state variables that capture the relevant information about the market and the current state of the execution agent. We Limited the state space to 5 state variables for simplicity.

1. State Variable 1: **Remain Inventory**. Noted "I". Gives the number of share left we have to buy. Takes values between X and 0.
2. State Variable 2: **Imbalance**. Noted "Imb". Given by $Imb = \frac{BidVolume}{AskVolume}$
3. State Variable 3: **Volatility**. Noted σ. Given by $\sigma = \sqrt{\text{Annualization Factor} \times \text{Var(Log Returns)}}$ Computes the volability for the last 10s.
4. State Variable 4: **Spread.** Noted Sp, given by the formula: $Sp = AskPrice - BidPrice$
5. State Variable 5: **Elapsed time.** Noted t. Takes values between 0 and T.

Let's mention that any state with a value of I=0 or t=T is a terminal state.

5.2 Action Space

The action space defines the set of possible actions that the execution agent can take at each time step. Actions represent different execution strategies or trading decisions.

1. Action 1: **Buy market order**
2. Action 2: **Buy limit order at price midprice(t) + a.**

For each of these actions, we attach a certain volume V. For complexity purposes, the volume V can take the values: $\frac{I}{2}, \frac{I}{5}, \frac{I}{10}, \frac{I}{100}$.

5.3 Rewards

The rewards in this MDP play a pivotal role in shaping the agent's behavior. When designing the reward function, we have several options to consider. One approach is to impose a significant penalty on any order executed after time T with a substantial negative reward. We can introduce a parameter β to model "how bad it is to exceed T". Re utilizing the former reward function notations (problem 4):

$$R(s, a, s') = \frac{CF(s, a, s')}{Id(V, t_0)} - \beta I_{t>T}$$

Where :

$$I_{t>T} = \begin{cases} 1, & \text{if } t > T \\ 0, & \text{if } t < T \end{cases}$$

Alternatively, we could implement a more nuanced strategy by gradually penalizing orders as the execution time progresses, encouraging the agent to execute a larger portion of the order early on. This reward function could introduce a risk aversion parameter, allowing us to control the rate at which penalization increases.

$$R(s, a, s') = \frac{CF(s, a, s')(1 - \alpha t)}{Id(V, t_0)}$$

To create a more accurate representation of our problem, we may also consider incorporating a constraint in the action space rather than the reward function. For instance, we can enforce that the agent must execute all remaining inventory at the market price when time reaches T. This constraint would ensure that the remaining shares are executed promptly without explicitly relying on the reward function.

5.4 Transition Probabilities Model

Given the complexity of the data we use, the transition probability model is not explicitly defined but rather inferred from the simulation results.

5.5 Market Impact

In addressing market impact within this problem, we face the choice of whether to explicitly model its effects on our execution strategy. If we opt not to model market impact, we should make assumptions, such as executing small orders in a highly liquid market, where our actions have minimal influence on prices. However, modeling market impact realistically can be intricate. It requires training the agent and other agents in the environment to respond to market dynamics and adapt as if trading in a real-world environment. Training the agent solely on the LOBSTER dataset, as seen in Problem 4, does not account for market impact accurately, as it lacks information on how the market reacts to our orders. Incorporating a temporary market impact model into the MDP is a more sophisticated approach but significantly increases complexity. Furthermore, defining the agent's behavior in response to market impact becomes crucial—whether it should minimize impact by executing smaller orders over a more extended period or prioritize faster execution, even with higher price impact. These considerations highlight the complexities and choices involved in handling market impact effectively.

Listing 1: Python code for problem 1

```python
import random as rd
import matplotlib.pyplot as plt

def next_position(position):
    # Entry : Int position, position of the player in the game
    # Output: Int position, position of the player after one play

    # Create a dictionary to map positions after a dice roll
    position_mapping = {
        1: 38, 4: 14, 9: 31, 17: 6, 21: 42, 28: 84, 36: 44, 47: 26, 49: 11,
        51: 67, 56: 53, 64: 60, 71: 91, 87: 24, 80: 100, 93: 73, 95: 75
    }

    # Compute the result of the dice
    dice_result = rd.randint(1, 6)

    # Calculate the new position
    new_position = position + dice_result

    # Check if the new position is in the position mapping
    if new_position in position_mapping:
        return position_mapping[new_position]

    return new_position

def simulation():
    # Entry  -  None
    # Output -  Int : The number of throws needed to arrive at 100


    # Intiatlization
    number_of_throws = 0
    position = 0

    # Play
    while position < 100:
        position = next_position(position)
        number_of_throws += 1

    # We arrived at 100
    return number_of_throws

mean = 0
result_list = []
for i in range(5000):
    result = simulation()
    mean+=result
    result_list.append(result)

```

```python

# Plot the histogram of result_list
plt.hist(result_list, bins=range(min(result_list), max(result_list) + 1),
    align='left', rwidth=0.8)

# Calculate the average number of throws
average_throws = mean / 5000

# Add a vertical line for the average number of throws
plt.axvline(x=average_throws, color='red', linestyle='dashed', linewidth=2,
     label=f'Average Throws: {average_throws:.2f}')

# Set labels and title
plt.xlabel('Number of Throws')
plt.ylabel('Frequency')
plt.title('Histogram of Number of Throws in 5000 Simulations')

# Add a legend
plt.legend()

# Add the x value as text annotation
plt.annotate(f'{average_throws:.2f}', xy=(average_throws, 1), xytext=(
    average_throws + 5, 100), fontsize=12,
             arrowprops=dict(arrowstyle='->', color='black'))

# Show the plot
plt.show()
```

Listing 2: Python code for problem 3 (adapted from class)

```python

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

class Frog:
    def __init__(self, n=10, gamma=0.99, initial_alpha=0.1, initial_eps
        =0.9, num_simulations = 50,
                 hyperparameter_scheme=2, rewards_choice=2
                ):
        self.n = n
        self.gamma = gamma
        self.initial_alpha = initial_alpha
        self.initial_eps = initial_eps
        self.state_space = list(range(n+1))
        self.terminal_space = [0, n]
        self.action_space = ['A', 'B']

        self.hyperparameter_scheme = hyperparameter_scheme
        self.rewards_choice = rewards_choice

```

```
        #initialization - might impact the speed of training (especially in early stages), but should not impact the outcome
        #self.Q = np.zeros([len(self.state_space), len(self.action_space)])
        self.Q = np.random.rand( len(self.state_space), len(self.action_space))

        self.num_simulations = num_simulations
        self.simulated_rewards = []


    def eps_greedy_action(self, eps):
        rv = np.random.uniform(0,1)
        if rv < eps:
            #random action
            return np.random.choice(self.action_space)
        else:
            #follow the best action
            return self.action_space[np.argmax(self.Q[self.state, :])]


    def action_A(self, state):
        rv = np.random.uniform(0,1)
        if rv < float(state)/self.n:
            return state - 1
        else:
            return state + 1

    def action_B(self, state):
        state_space = list(self.state_space)
        state_space.pop(state)
        return np.random.choice(state_space)

    def choose_reward_function(self, state):
        if self.rewards_choice == 1:
            return self.get_reward(state)
        elif self.rewards_choice == 2:
            return self.get_reward2(state)
        elif self.rewards_choice == 3:
            return self.get_reward3(state)
        elif self.rewards_choice == 4:
            return self.get_reward4(state)
        else:
            print("ERROR! Unknown reward function")

    def get_reward(self, state):
        #reward assignment - you can experiment with different rewards
        return float(state)/self.n

    def get_reward2(self, state):
        #reward assignment - you can experiment with different rewards
```

```
        if state == 0:
            return 0
        elif state == self.n:
            return 1
        else:
            return 0

    def get_reward3(self, state):
        #reward assignment
        if state == 0:
            return -1
        elif state == self.n:
            return 1
        else:
            return 0

    def get_reward4(self, state):
        #reward assignment
        if float(state) == 0:
            return 0
        else:
            return 1-1/float(state)**2

    def simulate(self, num_simulation):
        simulated_rewards = []
        for i in range(num_simulation):
            state = np.random.randint(1, self.n - 1)
            reward = 0
            while state not in self.terminal_space:
                #follow the best policy
                action = self.action_space[np.argmax(self.Q[state, :])]
                if action == 'A':
                    state_new = self.action_A(state)
                else:
                    state_new = self.action_B(state)
                reward += self.choose_reward_function(state_new)
                state = state_new
            simulated_rewards.append(reward)
        #return cumlated rewards over num_episode simulations for a given policy
        return np.mean(simulated_rewards)

    def simulate_for_large_n(self, num_simulation):
        #this function might be useful for testing/debugging your code for large n
        simulated_rewards = []
        for i in range(num_simulation):
            state = np.random.randint(1, self.n - 1)
            reward = 0
            num_iter = 0
```

```python
            while state not in self.terminal_space and (num_iter<0.5e7):
                #with a small probability pick action B not to be stuck in the infinite loop traversing the lilypads,
                #otherwise follow the best policy
                rv = np.random.uniform(0,1)
                if rv < 1e-2:
                    action = 'B'
                else:
                    action = self.action_space[np.argmax(self.Q[state, :])]
                if action == 'A':
                    state_new = self.action_A(state)
                else:
                    state_new = self.action_B(state)
                reward += self.choose_reward_function(state_new)
                state = state_new
                num_iter +=1
            if (num_iter<0.5e7):
                simulated_rewards.append(reward)
            else:
                print("Dropped rewards due to large time needed to simulate")
        #return cumlated rewards over num_episode simulations for a given policy (this value is calibrated to n=25)
        return np.mean(simulated_rewards)

    def choose_hyperparameter_scheme(self, i, num_episode):
        if self.hyperparameter_scheme == 1:
            return self.my_hyperparameter_scheme_1(i, num_episode)
        elif self.hyperparameter_scheme == 2:
            return self.my_hyperparameter_scheme_2(i, num_episode)
        else:
            print("ERROR! Unknown hyperparameter scheme")

    def my_hyperparameter_scheme_1(self, i, num_episode):
        if i<500:
            eps =  self.    initial_eps
            alpha = float(self.initial_alpha*(num_episode - i))/num_episode
        else:
            eps = 0
            alpha = float(self.initial_alpha*(num_episode - i))/num_episode/10
        return [eps, alpha]

    def my_hyperparameter_scheme_2(self, i, num_episode):
        eps =  self.initial_eps
        alpha = float(self.initial_alpha*(num_episode - i))/num_episode
        return [eps, alpha]

    def q_learning(self, num_episode):
        for i in range(num_episode):
```

```
            print("training episode", i)
            self.state = np.random.randint(1, self.n - 1)

            #my hyperparameter scheme - feel free to implement your own
            [eps, alpha] = self.choose_hyperparameter_scheme(i, num_episode)

            while self.state not in self.terminal_space:
                #epsilon-greedy action selection
                action = self.eps_greedy_action(eps)
                #follow action to a new state
                if action == 'A':
                    state_new = self.action_A(self.state)
                else:
                    state_new = self.action_B(self.state)
                #get reward at a new state
                reward = self.choose_reward_function(state_new)
                #Q-update
                self.Q[self.state, self.action_space.index(action)] += alpha * (reward + self.gamma * np.max(self.Q[state_new, :]) - self.Q[self.state, self.action_space.index(action)])
                self.state = state_new

            #now simulated rewards for the fixed Q table
            self.simulated_rewards.append(self.simulate(self.num_simulations))


    def all_plots(self):
        plt.figure(figsize=(15, 4))
        plt.subplot(121)
        plt.title("Q*-values")
        plt.plot(self.Q[:,0], label='A')
        plt.plot(self.Q[:,1], label='B')
        plt.xlabel('State')
        plt.xlim((1, self.n-1))
        plt.xticks(range(1, self.n))
        plt.legend()
        plt.subplot(122)
        plt.title("Average Cumulative Rewards Over 10 Simulations")
        plt.plot(pd.Series(self.simulated_rewards))
        plt.plot(pd.Series(self.simulated_rewards).rolling(10).mean())
        plt.xlabel('Training Episode')
        plt.show()
```

Listing 3: Python code for problem 4

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

```python


# Initialization of the orderbook and message file:
message_file = 'AMZN_2012-06-21_34200000_57600000_message_5.csv'
orderbook_file = 'AMZN_2012-06-21_34200000_57600000_orderbook_5.csv'
message_data = pd.read_csv(message_file,header=None)
orderbook_data = pd.read_csv(orderbook_file,header=None)

# Initialization of inventory, lets say we want to buy 100 shares
I = 100

def get_mean_for_tens():
    # This function returns the average number of tick between every 10s
    # It avoid the 10s and 10s volatility function to have a tremendous complexity by looking for the last
    # 10s tick time among all the data
    n=0
    tick_list = []
    previous_row = None

    for index, row in message_data.iterrows():
        if previous_row is not None:
            if row[0]>34200+n*10 and  previous_row[0]<34200+n*10:
                tick_list.append(index)
                n+=1
        else :
            tick_list.append(index)
            n+=1
        previous_row=row

    for i in range(len(tick_list)-1):
        tick_list[-i]=tick_list[-i]-tick_list[-i-1]
    tick_list.pop(0)
    mean = np.mean(tick_list)
    print(mean)

# Retreive all the states variables
def get_remain_inventory():
    return I

def get_midprice(time, orderbook_data):
    # Entry:
    #        The orderbook lobster file from a given stock on df format: orderbook_data
    #        The ticker time at format integer X: t
    # Outpout:
    #        The midprice at ticker time t
    midprice = (float(orderbook_data.iloc[time,0]) + float(orderbook_data.iloc[time,2])) / 2
    return midprice
```

```python
def get_midprice_physical(t, orderbook_data, message_data):
    #               --- midprice_physical ---
    # Entry:
    #         The orderbook lobster file from a given stock on df format:
        orderbookFile
    #         The message lobster file from a given stock on df format:
        messageFile
    #         The physical time at format XXXX.XXXXXXXX: t
    # Outpout:
    #         The midprice at time t


    # Finds the index of the last transaction made before time = t
    filtered_data = message_data[message_data.iloc[:, 0] < t]
    tickerT = len(filtered_data)


    # To avoid an indice out of bounds error
    if tickerT==len(message_data):
        return (float(orderbook_data.iloc[tickerT-1,0]) + float(
            orderbook_data.iloc[tickerT-1,2])) / 2

    # This index gives us the best bid and best ask at the given time
    midprice = (float(orderbook_data.iloc[tickerT,0]) + float(
        orderbook_data.iloc[tickerT,2])) / 2
    return midprice

def get_imbalance(time,orderbook_data):
    # Entry:
    #         The orderbook lobster file from a given stock on CVS format:
        orderbookFile
    #         The ticker time at format integer X: t
    # Outpout:
    #         The imballance at ticker time t
    askVol = 0
    bidVol = 0
    for i in range(10):
        askVol += orderbook_data.iloc[time,2*i+1]
        bidVol += orderbook_data.iloc[time,2*i]
    return bidVol/askVol

def get_volatility(time,orderbook_data):
    # Entry : tick Time
    # This function computes the volability of 10 tick returns during the
        last 100 ticks
    # During the first min, lets say at time = 10 the function
    # returns the volability during the first 10s

    # Init
```

```
    start_time = max(0 ,time - 100)
    t = start_time
    midprice_list =[]

    # Computes the 1s return
    while t <= time:
        row = orderbook_data.iloc[t]
        midprice = (row[0]+row[2])/2
        midprice_list.append(midprice)
        t+=10

    # Compute the returns
    midprice_return_list = []
    for i in range(1, len(midprice_list),10):
        midprice_return_list.append(midprice_list[i]-midprice_list[i-1])

    # Computes the var and volatility
    sigma = np.std(np.log(midprice_list))
    return sigma

def get_spread(time,orderbook_data):
    #                --- spread_physical ---
    # Entry:
    #        The orderbook lobster file from a given stock on df format: orderbookFile
    #        The ticker time at format n: t
    # Outpout:
    #        The spread at ticker time t

    # This index gives us the best bid and best ask at the given time
    spread = abs(float(orderbook_data.iloc[time,0]) - float(orderbook_data.iloc[time,2]))
    return spread

def get_traded_volumes(time,message_data):
    # Entry : Tick time
    # This function returns the cumulative traded volumes durnig the last 100 ticks

    start_time = max(0 ,time - 79)
    t = start_time
    cumulated_volume = 0

    while t <= time:
        row = message_data.iloc[t]
        if row[1] in [4, 5]:
            cumulated_volume+=row[3]
        t+=1

    return cumulated_volume
```

```python

def get_best_bid(time,orderbook_data):
    return orderbook_data.iloc[time,2]

def get_best_ask(time,orderbook_data):
    return orderbook_data.iloc[time,0]

def get_last_event(time,message_data):
    return message_data.iloc[time,1]

def get_10s_returns(time,orderbook_data):
    start_time = max(0, time - 79)
    return ( get_midprice(time, orderbook_data) -  get_midprice(start_time, orderbook_data) )

def get_1s_returns(time,orderbook_data):
    start_time = max(0, time - 8)
    return ( get_midprice(time, orderbook_data) -  get_midprice(start_time, orderbook_data) )

def get_stateSpace(time,orderbook_data, message_data):
    # returning the value of the state space at time t
    # The value of I is supposed to be dynamically changed so here we will just use the initial
    # value of I as we dont trade.
    return I, get_midprice(t,orderbook_data),get_imbalance(t,orderbook_data),get_volatility(t,orderbook_data),get_spread(t,orderbook_data),get_traded_volumes(t,message_data),get_best_bid(t,orderbook_data),get_best_ask(t,orderbook_data),get_last_event(t,message_data),get_10s_returns(t,orderbook_data)

# Retrieve functions
def retrieve_values(time, orderbook_data, message_data):
    # Entry : Same as usual
    # Output: diferents list containing the values of each states variables every 10s for t < time

    # Init :
    midprice_list = []
    imbalance_list = []
    volatility_list = []
    spread_list = []
    traded_volumes_list = []
    bid_list = []
    ask_list = []
    last_event_list = []
    tenS_return_list = []
    oneS_return_list = []

    for t in range(0,time,10):
```

```
        print(t)
        midprice_list.append(get_midprice(t,orderbook_data))

        imbalance_list.append(get_imbalance(t,orderbook_data))

        volatility_list.append(get_volatility(t,orderbook_data))

        spread_list.append(get_spread(t,orderbook_data))

        traded_volumes_list.append(get_traded_volumes(t,message_data))

        bid_list.append(get_best_bid(t,orderbook_data))

        ask_list.append(get_best_ask(t,orderbook_data))

        last_event_list.append(get_last_event(t,message_data))

        tenS_return_list.append(get_10s_returns(t,orderbook_data))

        oneS_return_list.append(get_1s_returns(t,orderbook_data))
    return imbalance_list, volatility_list, spread_list,
        traded_volumes_list, bid_list, ask_list, last_event_list,
        tenS_return_list, oneS_return_list

# Plot function
def plot_values_over_time(time, orderbook_data, message_data):
    # Call the retrieve_values function to get the lists of state variables
    imbalance_list, volatility_list, spread_list, traded_volumes_list,
        bid_list, ask_list, last_event_list, tenS_return_list,
        oneS_return_list = retrieve_values(time, orderbook_data,
        message_data)

    # Create a time vector for plotting (assuming time increments of 10
        seconds)
    time_vector = list(range(0, time, 10))

    # Plot each state variable
    plt.figure(figsize=(12, 8))

    plt.subplot(3, 3, 1)
    plt.plot(time_vector, imbalance_list, label='Imbalance')
    plt.title('Imbalance Over Time')
    plt.xlabel('Time (seconds)')
    plt.ylabel('Imbalance')

    plt.subplot(3, 3, 2)
    plt.plot(time_vector, volatility_list, label='Volatility')
    plt.title('Volatility Over Time')
    plt.xlabel('Time (seconds)')
    plt.ylabel('Volatility')
```

```
    plt.subplot(3, 3, 3)
    plt.plot(time_vector, spread_list, label='Spread')
    plt.title('Spread Over Time')
    plt.xlabel('Time (seconds)')
    plt.ylabel('Spread')

    plt.subplot(3, 3, 4)
    plt.plot(time_vector, traded_volumes_list, label='Traded Volumes')
    plt.title('Traded Volumes Over Time')
    plt.xlabel('Time (seconds)')
    plt.ylabel('Traded Volumes')

    plt.subplot(3, 3, 5)
    plt.plot(time_vector, bid_list, label='Best Bid')
    plt.title('Best Bid Over Time')
    plt.xlabel('Time (seconds)')
    plt.ylabel('Best Bid')

    plt.subplot(3, 3, 6)
    plt.plot(time_vector, ask_list, label='Best Ask')
    plt.title('Best Ask Over Time')
    plt.xlabel('Time (seconds)')
    plt.ylabel('Best Ask')

    plt.subplot(3, 3, 7)
    plt.plot(time_vector, last_event_list, label='Last Event')
    plt.title('Last Event Over Time')
    plt.xlabel('Time (seconds)')
    plt.ylabel('Last Event')

    plt.subplot(3, 3, 8)
    plt.plot(time_vector, tenS_return_list, label='10s Returns')
    plt.title('10s Returns Over Time')
    plt.xlabel('Time (seconds)')
    plt.ylabel('10s Returns')

    plt.subplot(3, 3, 9)
    plt.plot(time_vector, oneS_return_list, label='1s Returns')
    plt.title('1s Returns Over Time')
    plt.xlabel('Time (seconds)')
    plt.ylabel('1s Returns')

    plt.tight_layout()
    plt.show()

def plot_histograms(time, orderbook_data, message_data):
    # Call the retrieve_values function to get the lists of state variables
    imbalance_list, volatility_list, spread_list, traded_volumes_list,
        bid_list, ask_list, last_event_list, tenS_return_list,
```

```
        oneS_return_list = retrieve_values(time, orderbook_data,
        message_data)

    # Create a time vector for plotting (assuming time increments of 10
        seconds)
    time_vector = list(range(0, time, 10))

    # Create a larger figure for improved readability
    plt.figure(figsize=(16, 12))

    # Adjust the subplot layout for better formatting
    plt.subplots_adjust(hspace=0.5)

    plt.subplot(3, 3, 1)
    plt.hist(imbalance_list, bins=20, color='skyblue', alpha=0.7)
    plt.title('Imbalance Histogram')
    plt.xlabel('Imbalance')
    plt.ylabel('Frequency')

    plt.subplot(3, 3, 2)
    plt.hist(volatility_list, bins=20, color='salmon', alpha=0.7)
    plt.title('Volatility Histogram')
    plt.xlabel('Volatility')
    plt.ylabel('Frequency')

    plt.subplot(3, 3, 3)
    plt.hist(spread_list, bins=20, color='lightgreen', alpha=0.7)
    plt.title('Spread Histogram')
    plt.xlabel('Spread')
    plt.ylabel('Frequency')

    plt.subplot(3, 3, 4)
    plt.hist(traded_volumes_list, bins=20, color='gold', alpha=0.7)
    plt.title('Traded Volumes Histogram')
    plt.xlabel('Traded Volumes')
    plt.ylabel('Frequency')

    plt.subplot(3, 3, 5)
    plt.hist(bid_list, bins=20, color='lightcoral', alpha=0.7)
    plt.title('Best Bid Histogram')
    plt.xlabel('Best Bid')
    plt.ylabel('Frequency')

    plt.subplot(3, 3, 6)
    plt.hist(ask_list, bins=20, color='lightblue', alpha=0.7)
    plt.title('Best Ask Histogram')
    plt.xlabel('Best Ask')
    plt.ylabel('Frequency')

    plt.subplot(3, 3, 7)
```

```python
    plt.hist(last_event_list, bins=20, color='lightsalmon', alpha=0.7)
    plt.title('Last Event Histogram')
    plt.xlabel('Last Event')
    plt.ylabel('Frequency')

    plt.subplot(3, 3, 8)
    plt.hist(tenS_return_list, bins=20, color='lightgreen', alpha=0.7)
    plt.title('10s Returns Histogram')
    plt.xlabel('10s Returns')
    plt.ylabel('Frequency')

    plt.subplot(3, 3, 9)
    plt.hist(oneS_return_list, bins=20, color='lightpink', alpha=0.7)
    plt.title('1s Returns Histogram')
    plt.xlabel('1s Returns')
    plt.ylabel('Frequency')

    # Show the plots
    plt.show()
```