Homework 2 - 290 - Francois Porcher

Francois Porcher

Osctober 2022

1 Problem 1: Snakes and Ladders

The average number of throws before winning the game is 36. (See the appendix for the code and the graphs).

2 Problem 2: Two Markov Decision Processes

Let M_1, M_2 2 Markov Decision Processes with corresponding reward functions R_1, R_2, such that $R_2(s) = R_1(s) + c$ for all states s and c constant.

Then we have:

$$\begin{aligned}
V_2^*(s) &= \max_a Q^{\pi^*}(s, a) \\
&= \max_a E_{\pi^*}\left(R_{2,t} \mid s_t = s, a_t = a\right) \\
&= \max_a E_{\pi^*}\left(\sum_{k=0}^{\infty} \gamma^k r_{2,t+k+1} \mid s_t = s, a_t = a\right) \\
&= \max_a E_{\pi^*}\left(\sum_{k=0}^{\infty} \gamma^k (r_{1,t+k+1} + c) \mid s_t = s, a_t = a\right) \\
&= \max_a E_{\pi^*}\left(\sum_{k=0}^{\infty} \gamma^k r_{1,t+k+1} \mid s_t = s, a_t = a\right) + \max_a E_{\pi^*}\left(\sum_{k=0}^{\infty} \gamma^k c \mid s_t = s, a_t = a\right) \\
&= V_1^*(s) + \frac{c}{1-\gamma}
\end{aligned} \tag{1}$$

because the second term of the sum does not depend on the actions a, states s and policy function π.

The Optimization problem for the second markov decision process is just the same Optimization problem for markov decision process 1 + a constant that does not depend on the policy.

From this we can deduce that the optimal policy π^* is the same for the two Markov Decision Processes.

3 Lilypads

3.1 Definition of the problem

3.1.1 State Space

The State Space is $\{0, 1, .., n\}$ corresponding to the $n+1$ lilypads.

3.1.2 Action Space

The frog can do two actions: croak sound A or sound B.

3.1.3 Transition Probabilities

If the frog choses to croak sound A, the transition probabilities are:

$$\begin{cases} p_{i,i+1} = \frac{n-i}{n} & \forall i \in \{0, ..n-1\} \\ p_{i,i-1} = \frac{i}{n} & \forall i \in \{1, ..n\} \end{cases}$$

If the frog choses to croak sound B, the transition probabilities are:

$$p_{i,j} = \frac{1}{n} \quad \forall j \neq i$$

3.1.4 Rewards

I tried several kind of reward functions:

- Give a reward only when reaching a terminal state
  - Give 1 as a reward if the frog reaches n.
  - Give -1 as a reward if the frog reaches 0.
  - Else give 0.
- Give a reward for depending on the state i. The reasonning behind this is that the closer to the terminal state, the higher the reward should be.
  $r(i) = \frac{i}{n}, \forall i \in \{0, .., n\}$

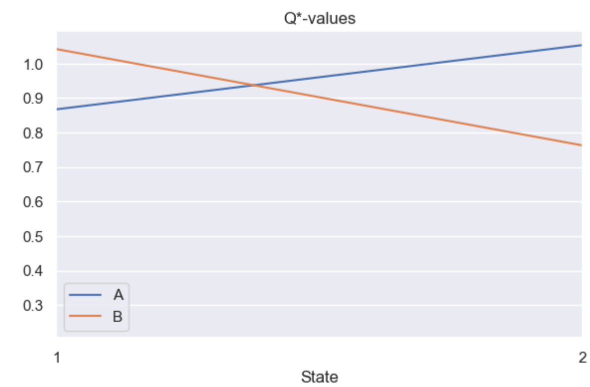
After experimenting with the reward functions, it turns out the second reward function works better.

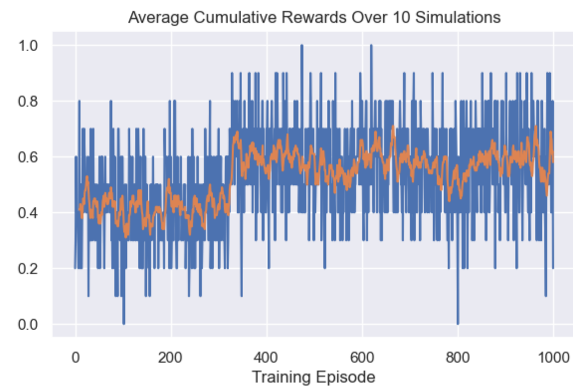
3.2 Results

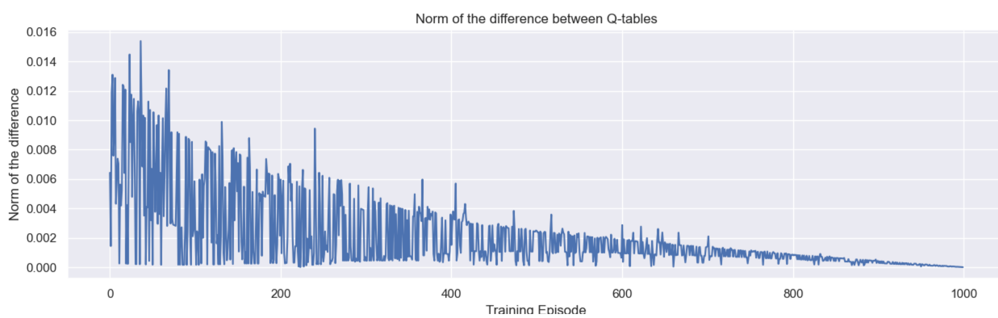
We can make several remarks about the following graphs:

- For the learning rate α, I chose to set the initial value at 0.01 and linearly decrease to zero with the number of iterations
- For ϵ, I picked the initial value 1 and make it linearly decrease toward 0 toward the number of iteration.
- We can see that the optimal policy is to pick choice A when the number of the lilypad is small, and B when it is large. This is coherent with the intuition, because when close to n, choice A will almost surely make the frog go to the previous state.
- I plotted the cumulative reward curve. We can think that it tends to increase with the number of training episodes.
- Finally we plot $||Q_{old} - Q_{new}||$, the norm of the difference between the Q-tables Q_{old} and Q_{new}. As the number of training episodes grow, we see that the Q table evolves less and less with time.

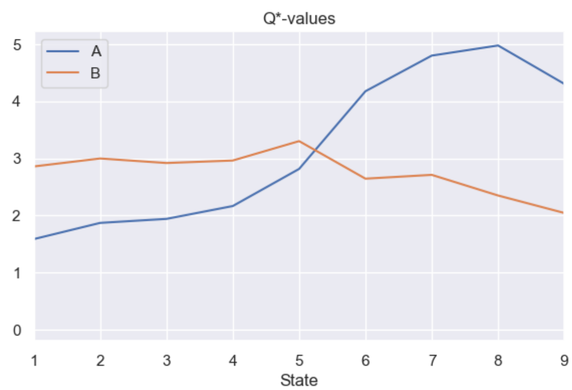
3.2.1 n = 3


Q*-values
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
A
B
1
2
State

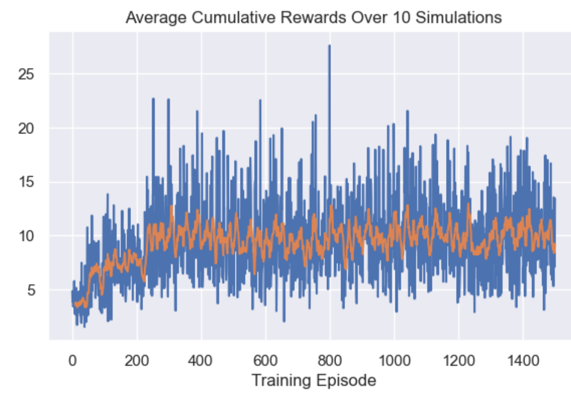

Average Cumulative Rewards Over 10 Simulations
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000
Training Episode


Norm of the difference between Q-tables
Norm of the difference
0.016
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
0
200
400
600
800
1000
Training Episode

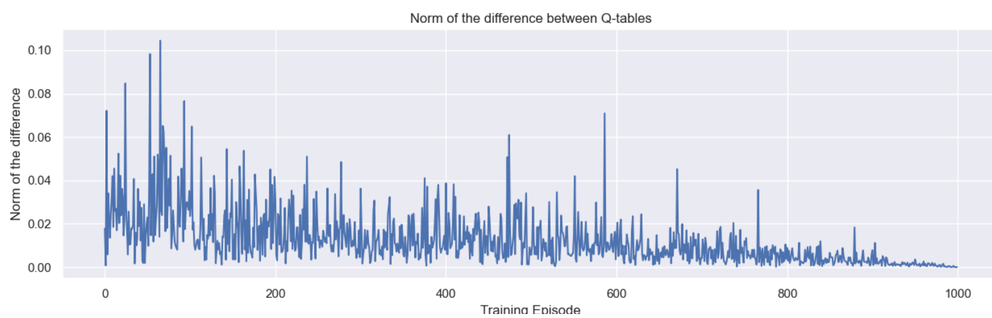
3.2.2 n = 10



Q*-values
A
B
State

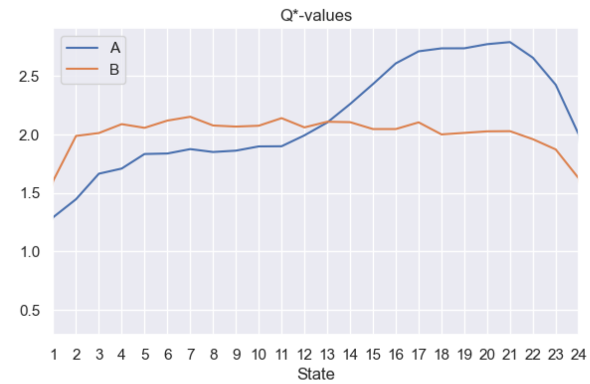


Average Cumulative Rewards Over 10 Simulations
Training Episode

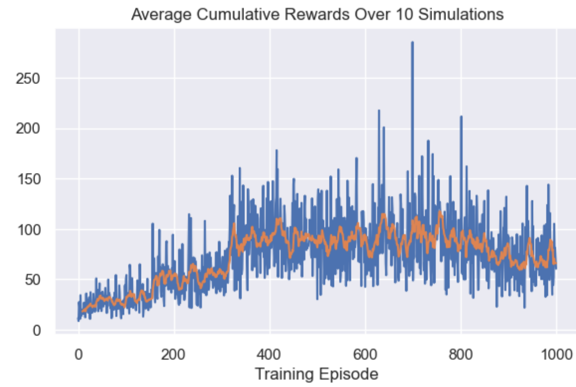


Norm of the difference between Q-tables
Norm of the difference
Training Episode

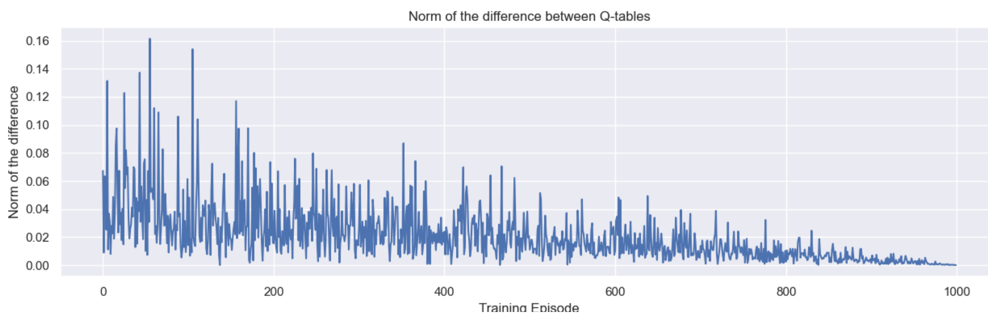
3.2.3 n = 25



Q*-values
A
B
2.5
2.0
1.5
1.0
0.5
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
State



Average Cumulative Rewards Over 10 Simulations
250
200
150
100
50
0
0 200 400 600 800 1000
Training Episode



Norm of the difference between Q-tables
Norm of the difference
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0 200 400 600 800 1000
Training Episode

4 MDP

In this problem, we want to execute a large buy order of size X.

4.1 State Space

We can divide state variables into 2 types (source: Reinforcement Learning for Optimized Trade Execution):

1. Trading agent state variable
   - Amount of time remaining for the complete execution (We divide the horizon H into T distinct points at which the policy is allowed to observe the state and take action)
   - Amount of shares we still need to buy at time t: X_t ($X_0 = X$)
2. Market variables
   - Bid/Ask Spread: s
   - Mid-Price move: Δm
   - Spread
   - Spread Volatility
   - Bid/Ask Volume imbalance: $\frac{\text{Bid Volume}}{\text{Ask Volume}}$
   - Time difference from last traded point
   - Signed Volume
   - Volume traded in the last N points
   - Volume traded in the last N ticks
   - Volatility

   I expect the Bid/Ask Volume imbalance to be a very significative feature because it renders the current market imbalance and in which direction the price is going to move.

   I also expect the Current Volume/Liquidity and Spread to be the most significant features because they are directly correlated with the cost of transactions.

4.2 Action Space

Let's consider that the size of every order (Limit Order and Market Order) is constant of size o, with $o << X$.

Tha agent can either place an agressive order (M.O of size o), place a limit order of size o in one of the 5 depth of the Best Ask in the Limit Order Book, wait (do nothing), or cancel an existing Limit Order.

To sum up:

- Market Order of size o
- Place Buy Limit Order of size o in Best Ask 1 (Depth 1)

- Place Buy Limit Order of size o in Best Ask 2 (Depth 2)
- Place Buy Limit Order of size o in Best Ask 3 (Depth 3)
- Place Buy Limit Order of size o in Best Ask 4 (Depth 4)
- Place Buy Limit Order of size o in Best Ask 5 (Depth 5)
- Wait (Do nothing)
- Cancel an existing Limit Order

4.3 Transition Probabilities

The transition probabilities are given by a simulator.

4.4 Reward

Let's define:

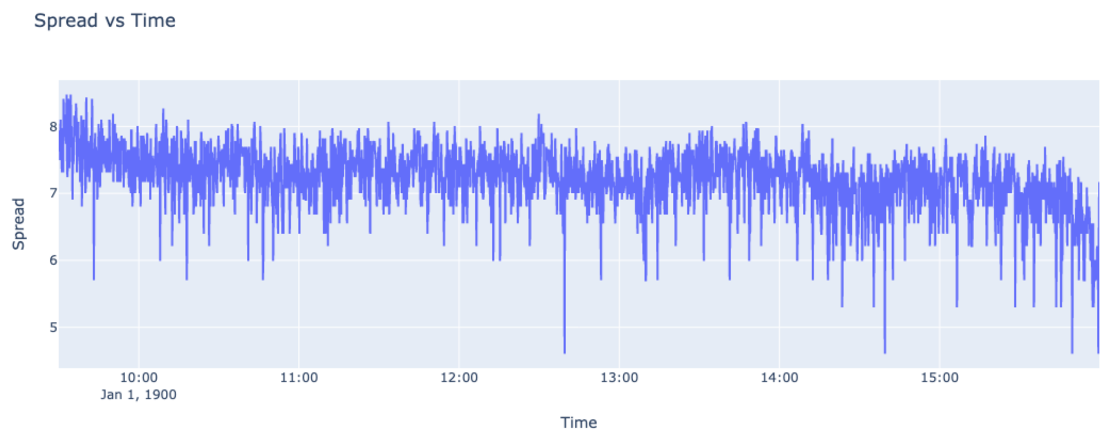
- R_t be the reward received at time t.
- $f_{t,i}$ the number of orders who where executed at time t for corresponding price $p_{t,i}$.
- $p_{t,i}$ the price at which $f_{t,i}$ orders were executed at time t
- m_t the mid price at time t.

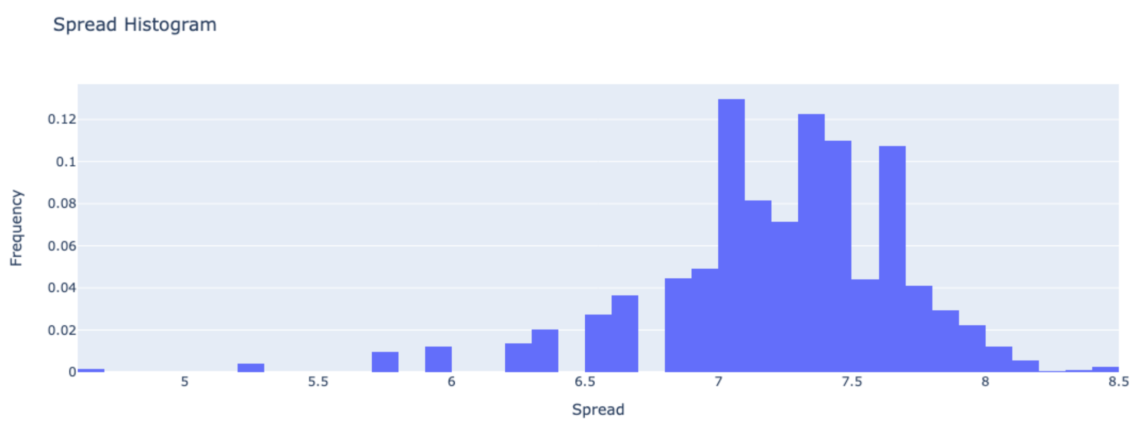
$$\boxed{R_t = \sum_i (p_{t,i} - m_t) f_{t,i}}$$

Indeed, a limit order placed at time t_1 could be executed at a much later time t_2, along with other limit orders with a different price. Thus, we have to sum over the different prices of execution.

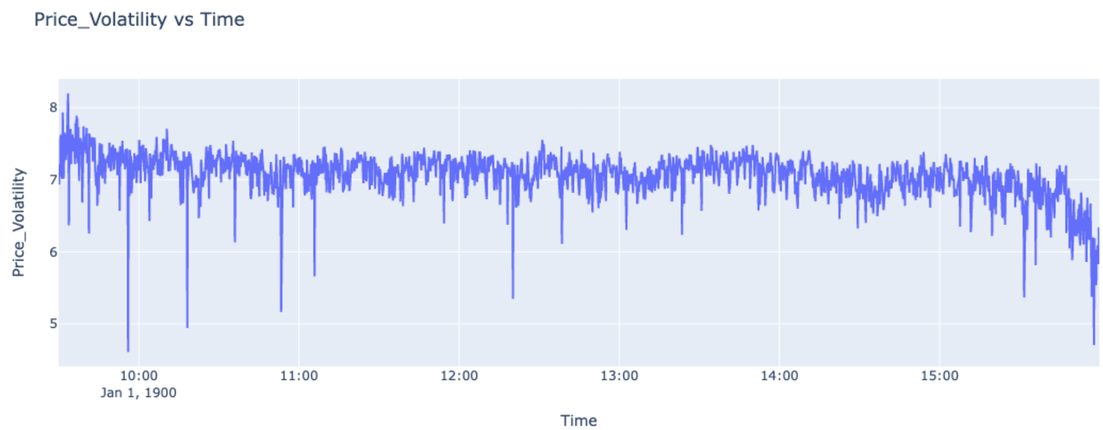
4.5 Plotting features

4.5.1 Spread

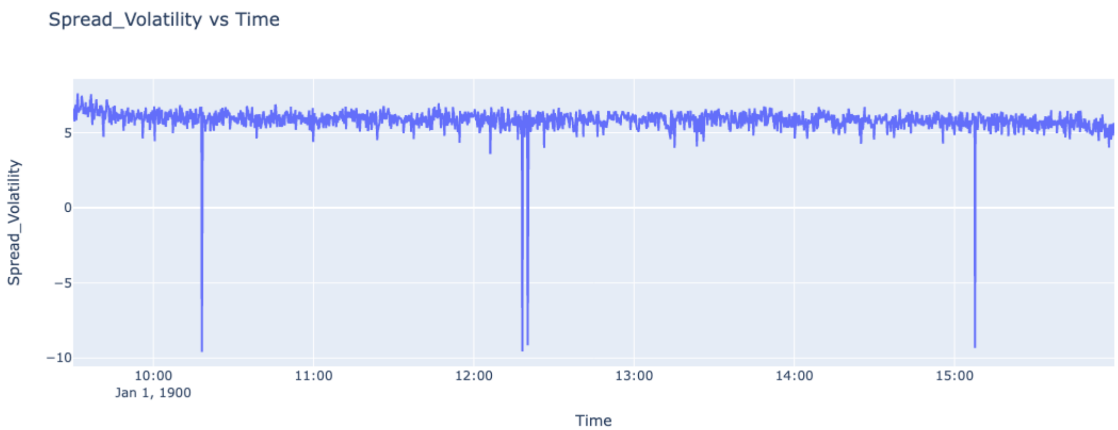


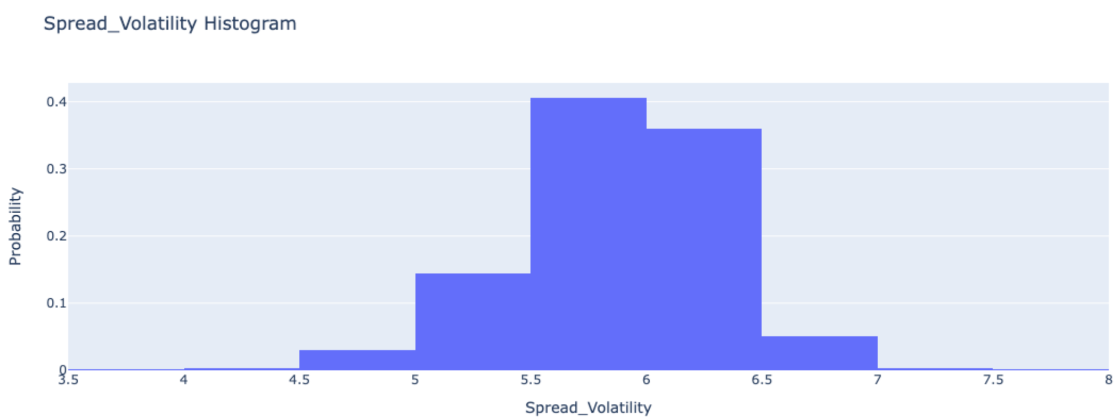


4.5.2 Price Volatility

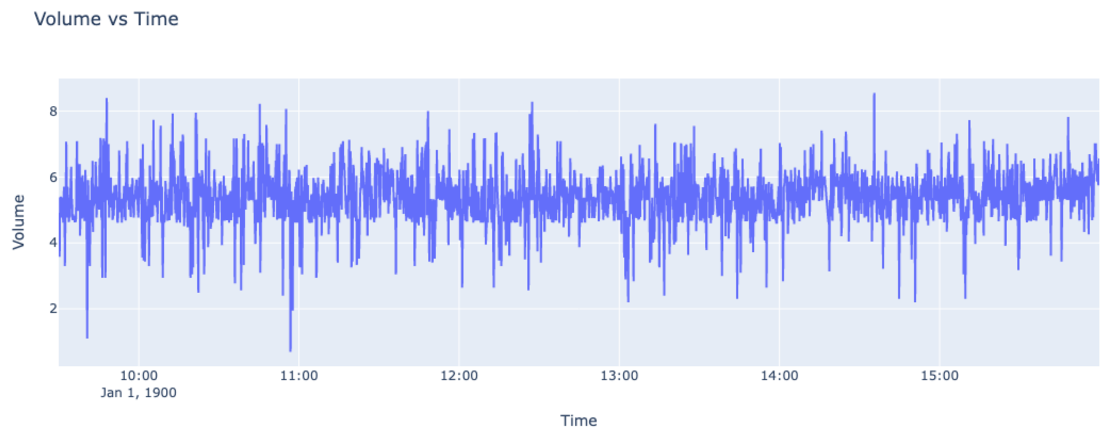


4.5.3 Spread Volatility

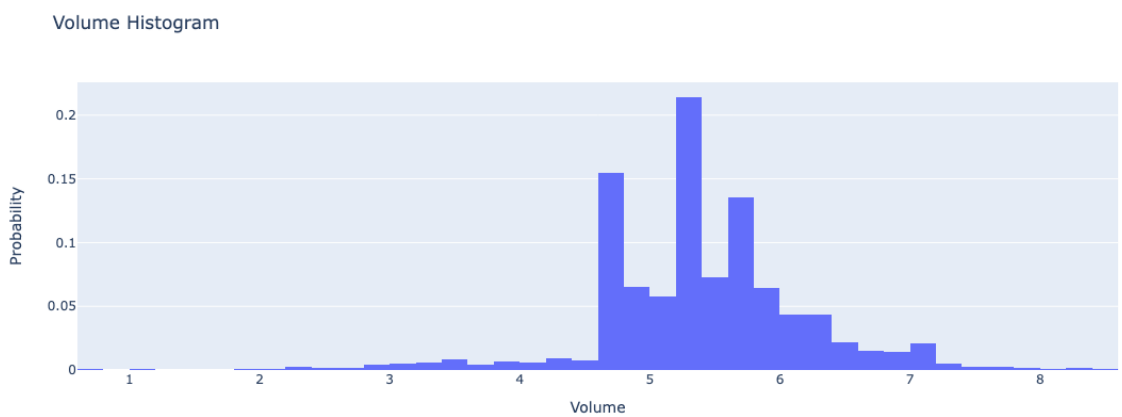




4.5.4 Volume

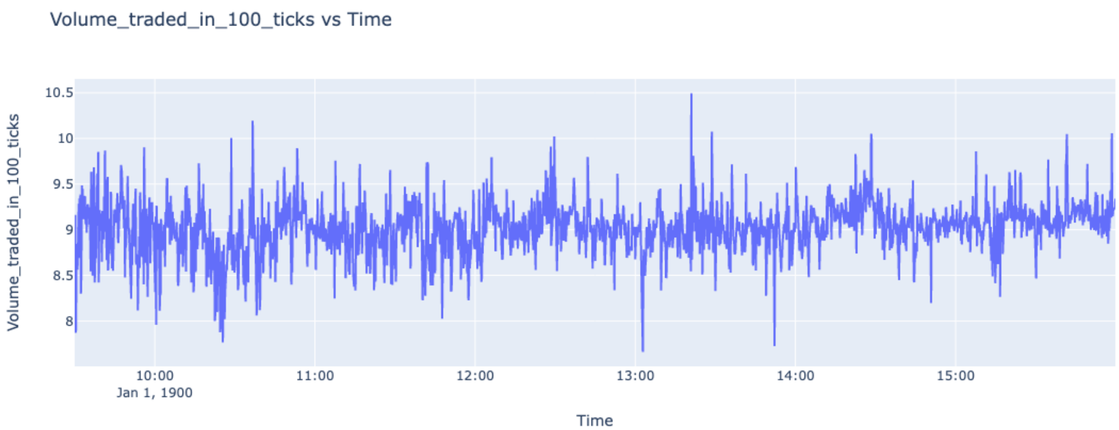


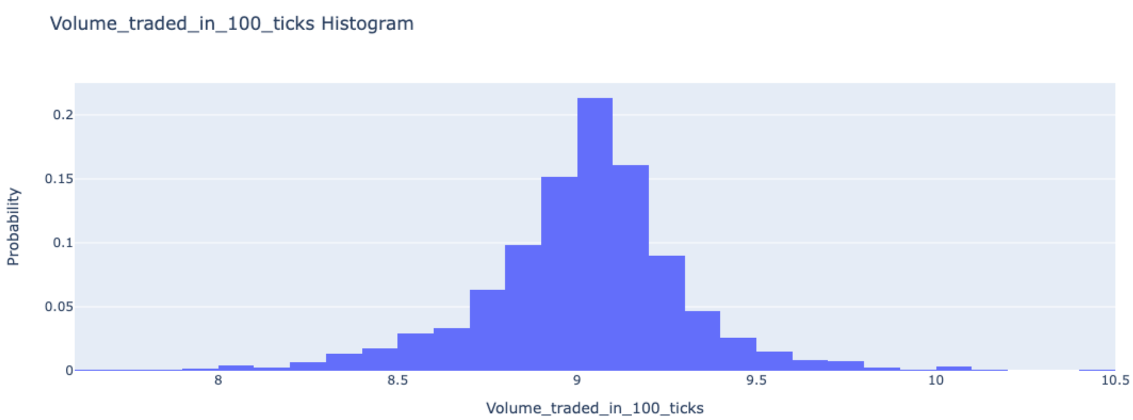
Volume vs Time
8
6
4
2
Volume
10:00
Jan 1, 1900
11:00
12:00
13:00
14:00
15:00
Time



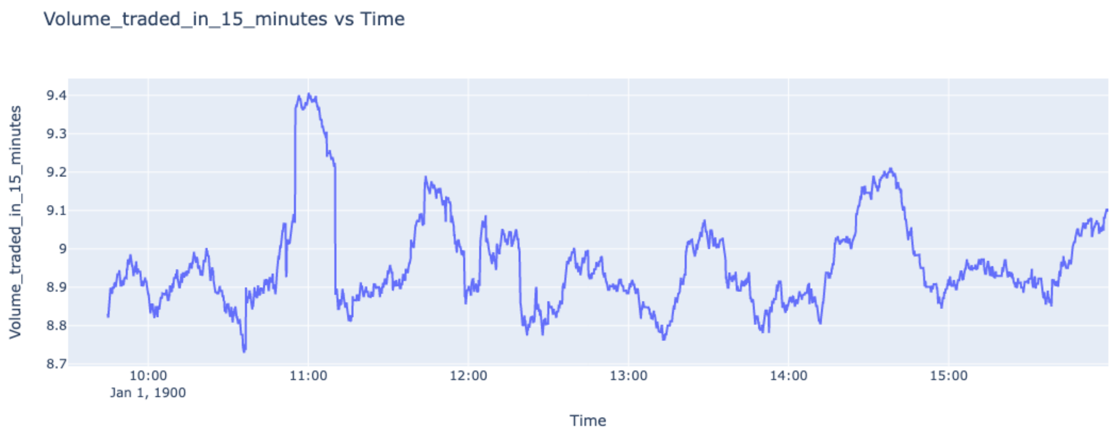
Volume Histogram
0.2
0.15
0.1
0.05
0
Probability
1
2
3
4
5
6
7
8
Volume

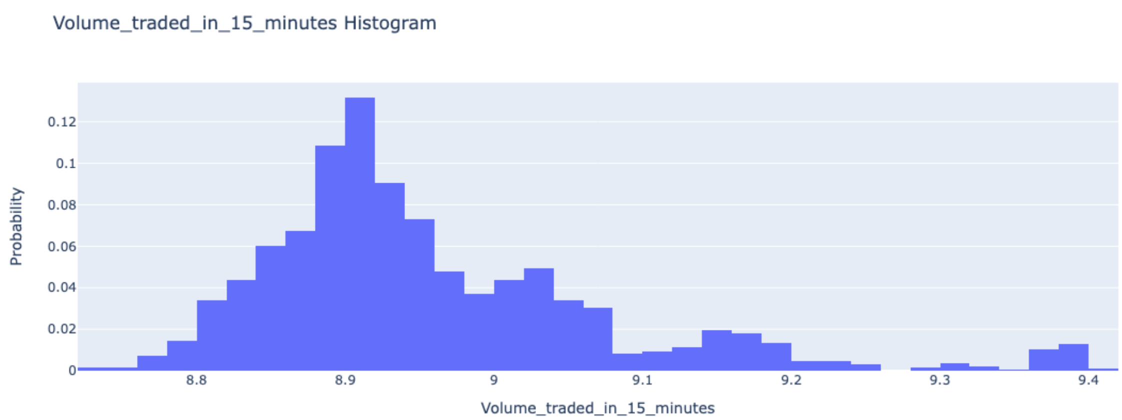
4.5.5 Volume traded during last 100 ticks



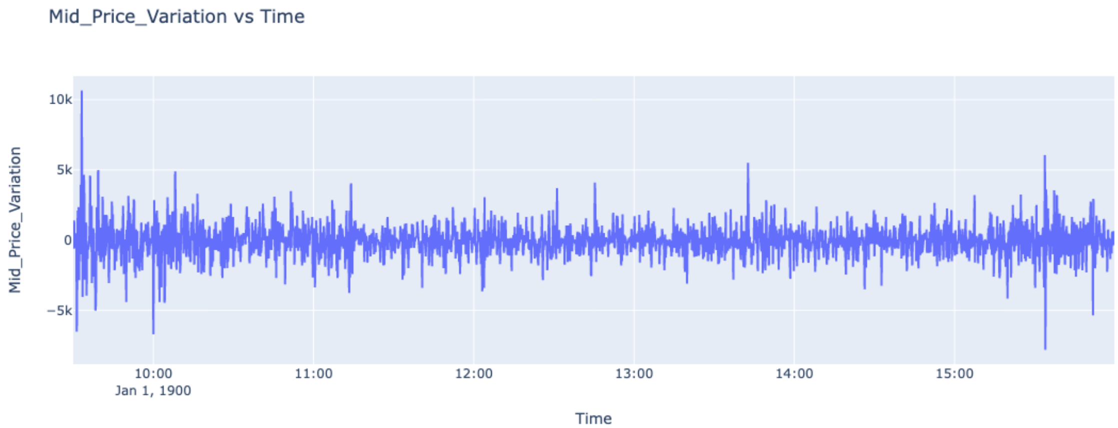


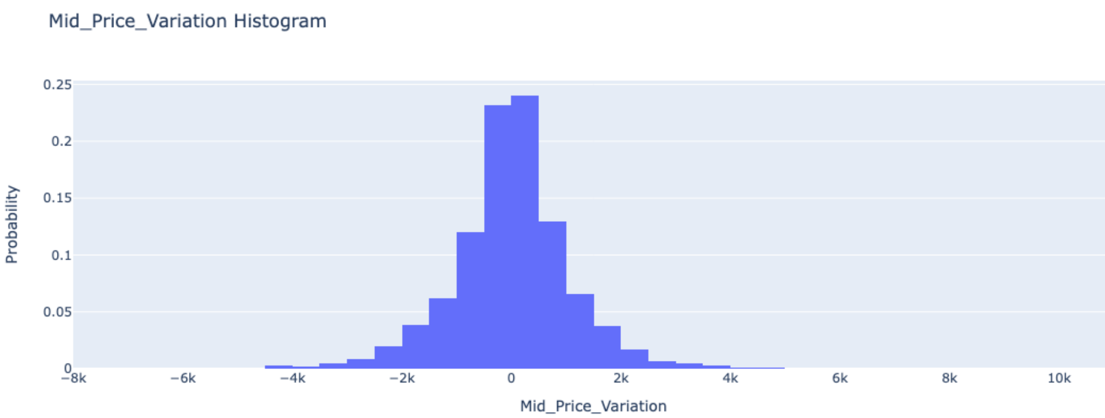
4.5.6 Volume traded in last 15 minutes





4.5.7 Mid Price Variation





snakes_and_ladders

October 21, 2022

1 Libraries

[1]:
```
import random
import matplotlib.pyplot as plt
plt.style.use('seaborn')
import numpy as np
import pandas as pd
import plotly.graph_objects as go
```

2 Snakes and Ladders

[2]:
```
def roll_dice():
    """Rolls a dice and returns the number"""
    return random.randint(1, 6)
```

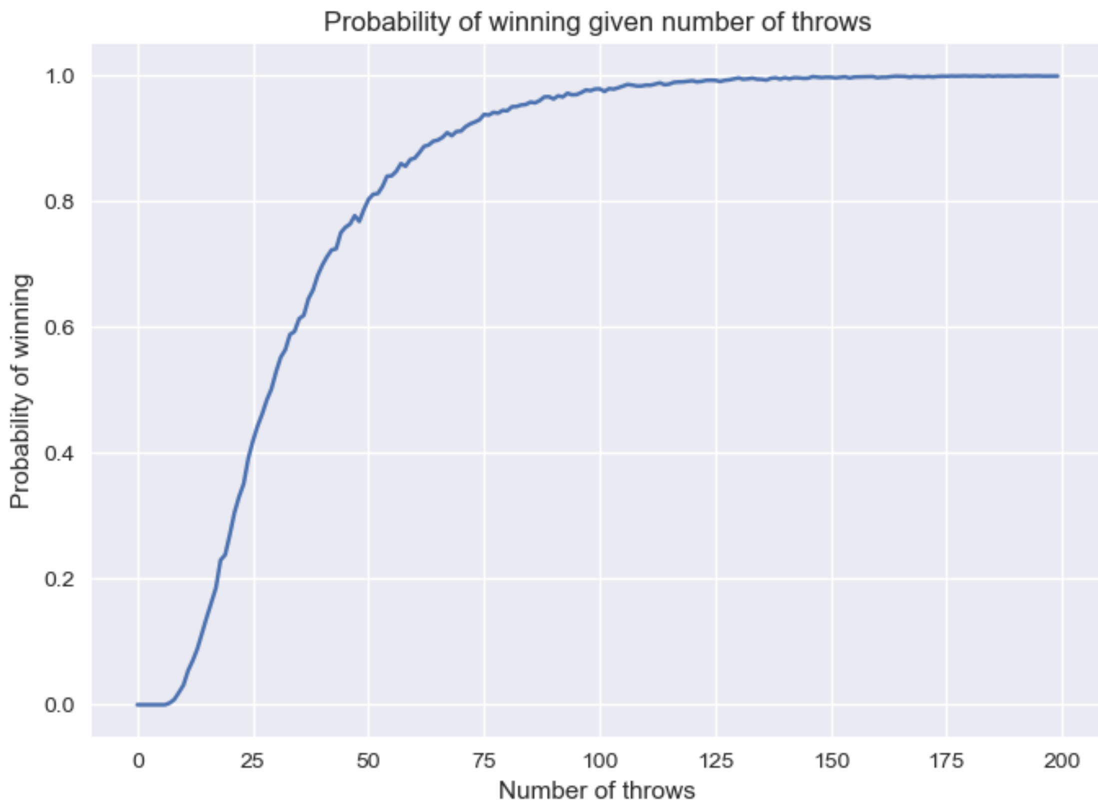
[3]:
```
# Define the board
dict_rules = {
    1 :38,
    4 : 14,
    9 : 31,
    16 : 6,
    21 : 42,
    28 : 84,
    36 : 44,
    47 : 26,
    49 : 11,
    51 : 67,
    56 : 53,
    62 : 19,
    64 : 60,
    71 : 91,
    80 : 100,
    87 : 24,
    93 : 73,
    95 : 75,
    98 : 78
}
```

[4]:
```
def simulate_snake_and_ladders_game(X):
    """_summary_
    After X throws, return a boolean: True if game is won, False otherwise
    """
    position = 0

    for i in range(X):
        dice_roll = roll_dice()
        position = position + dice_roll

        if position in dict_rules:
            position = dict_rules[position]

        if position >= 100:
            return True

    return False
```

[5]:
```
n_simulations = 5000
X = list(range(200))
proportion_games_won = []
for x in X:
    nb_games_won = 0
    for i in range(n_simulations):
        if simulate_snake_and_ladders_game(x):
            nb_games_won += 1

    proportion_games_won.append(nb_games_won / n_simulations)
```

[6]:
```
def plot_probability_winning_given_number_of_throws(X, proportion_games_won):
    plt.plot(X, proportion_games_won)
    plt.xlabel('Number of throws')
    plt.ylabel('Probability of winning')
    plt.title("Probability of winning given number of throws")
    plt.show()
```

[7]:
```
plot_probability_winning_given_number_of_throws(X, proportion_games_won)
```



[8]:
```python
def min_nb_roll_dice_before_winning_game():
    """_summary_
    After X throws, return a boolean: True if game is won, False otherwise
    """
    nb_dice_roll = 0

    position = 0

    while position < 100:
        dice_roll = roll_dice()
        nb_dice_roll += 1
        position = position + dice_roll

        if position in dict_rules:
            position = dict_rules[position]


    return nb_dice_roll
```

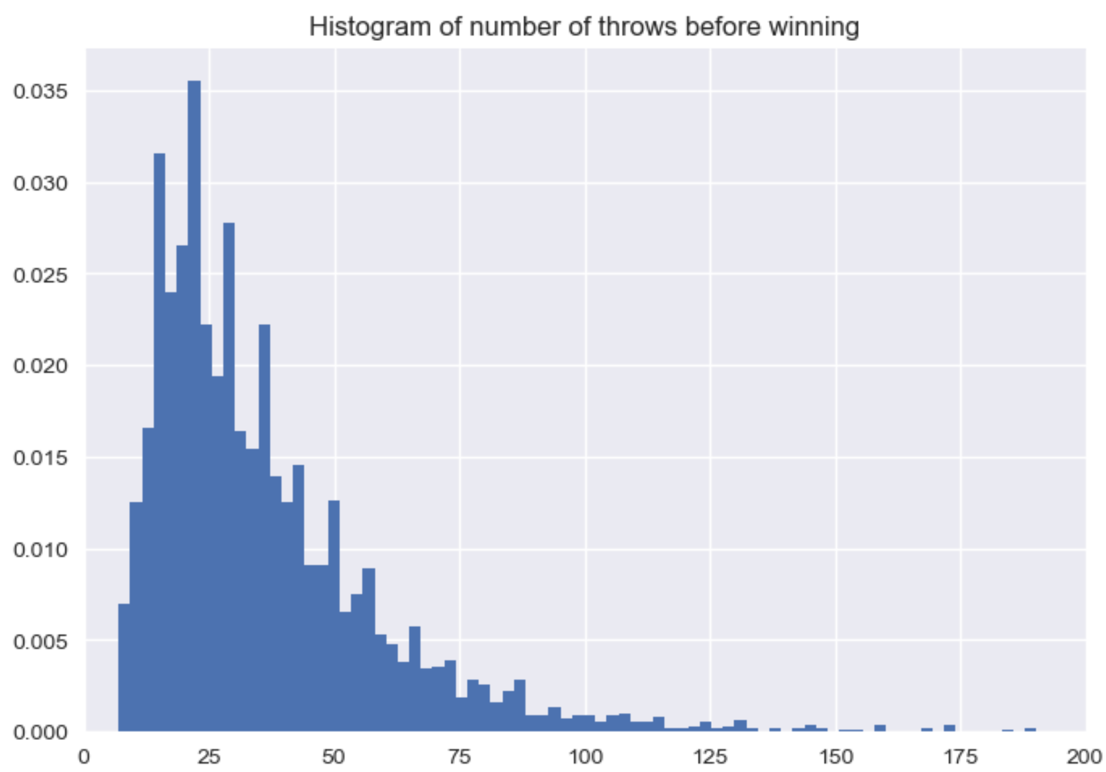
[42]:
```
n_simulations = 5000
X = []

for i in range(n_simulations):
    X.append(min_nb_roll_dice_before_winning_game())
```

[43]:
```
# Expectation of the number of throws before winning the game
print("The average number of throws before winning the game is: ", np.mean(X))
```

```
The average number of throws before winning the game is:  35.5894
```

[31]:
```
def plot_histogram_from_list(X):
    plt.hist(X, bins=100, density=True)
    plt.title("Histogram of number of throws before winning")
    plt.xlim(0, 200)
    plt.show()
```

[32]:
```
plot_histogram_from_list(X)
```



[ ]:

qlearning_frog_updated

October 21, 2022

[1]:
```
import seaborn as sns
sns.set_theme()
```

[3]:
```
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 = []
```

```python
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)
    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 simulate(self, num_simulation):
    simulated_rewards = []
    number_of_games_won = 0
```

```
        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

                if state == self.n:
                    number_of_games_won += 1

            simulated_rewards.append(reward)
        #return cumlated rewards over num_episode simulations for a given policy
        proportion_of_games_won = number_of_games_won/num_simulation

        return np.mean(simulated_rewards), proportion_of_games_won

    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
            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)

                print("reward = ", reward)
                state = state_new
```

```
                print("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)
        elif self.hyperparameter_scheme == 3:
            return self.my_hyperparameter_scheme_3(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 my_hyperparameter_scheme_3(self, i, num_episode):
        eps =  float(self.initial_eps*(num_episode - i))/num_episode
        alpha = float(self.initial_alpha*(num_episode - i))/num_episode
        return [eps, alpha]

    def q_learning(self, num_episode):
        self.list_norm_differences = []
        self.list_proportion_of_games_won = []

        for i in range(num_episode):
            old_Q = self.Q.copy()
            if i%50 == 0:
```

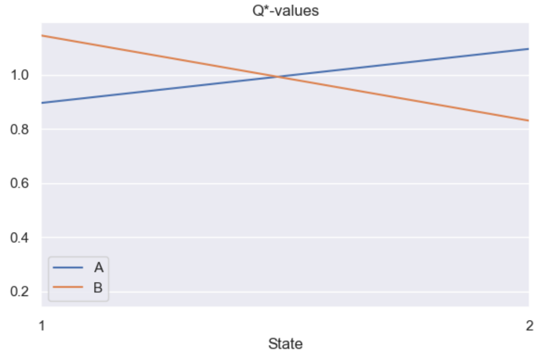
```
                print("Episode: ", i)
            self.state = np.random.randint(1, self.n - 1)

            #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
            reward_obtained_during_simulation, proportion_of_games_won = self.
↪simulate(self.num_simulations)
            self.simulated_rewards.append(reward_obtained_during_simulation)
            self.list_proportion_of_games_won.append(proportion_of_games_won)



            norm_difference = np.linalg.norm(self.Q - old_Q)
            self.list_norm_differences.append(norm_difference)


    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 {} Simulations".format(self.
↪num_simulations))
```

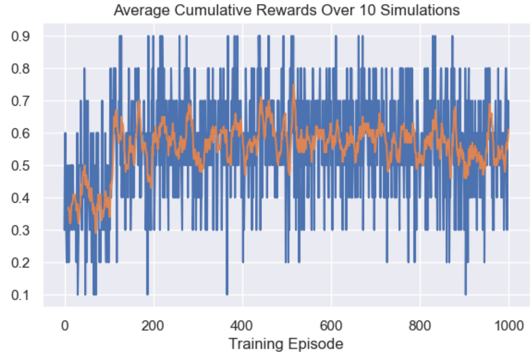
```
        plt.plot(pd.Series(self.simulated_rewards))
        plt.plot(pd.Series(self.simulated_rewards).rolling(10).mean())
        plt.xlabel('Training Episode')
        plt.show()
```

1 n = 3

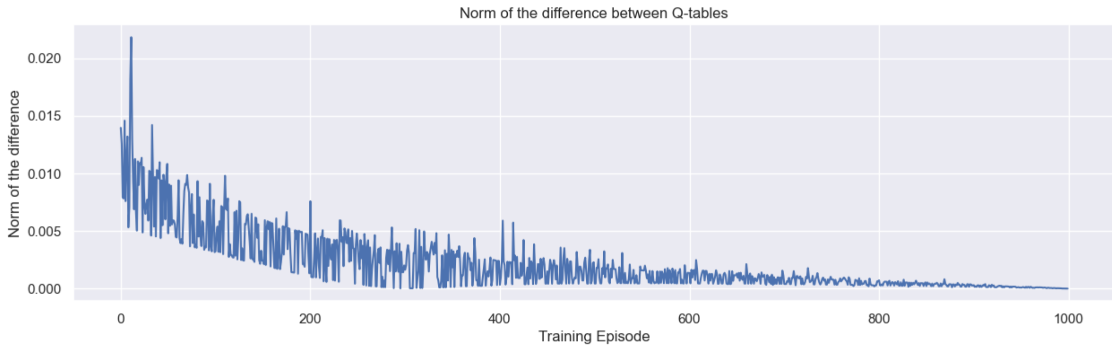
[7]:
```
myFrog_3 = Frog(n=3, gamma = 0.999, initial_alpha = 0.01, initial_eps = 0.999,␣
 ↪num_simulations = 10, hyperparameter_scheme = 3, rewards_choice = 2)
myFrog_3.q_learning(1000)
myFrog_3.all_plots()
```

```
Episode:  0
Episode:  50
Episode:  100
Episode:  150
Episode:  200
Episode:  250
Episode:  300
Episode:  350
Episode:  400
Episode:  450
Episode:  500
Episode:  550
Episode:  600
Episode:  650
Episode:  700
Episode:  750
Episode:  800
Episode:  850
Episode:  900
Episode:  950
```





[8]:
```
fig = plt.figure(figsize=(15, 4))
plt.plot(myFrog_3.list_norm_differences)
plt.title("Norm of the difference between Q-tables")
plt.xlabel('Training Episode')
plt.ylabel('Norm of the difference')
plt.show()
fig.savefig('norm_of_difference_3.png')
```

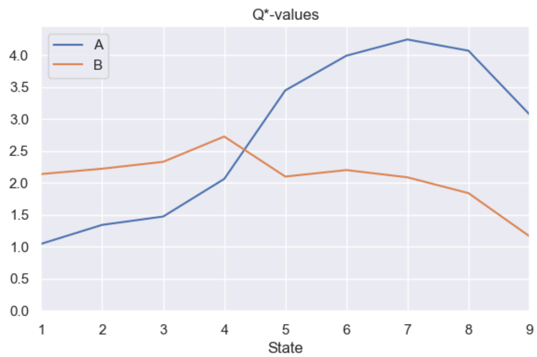


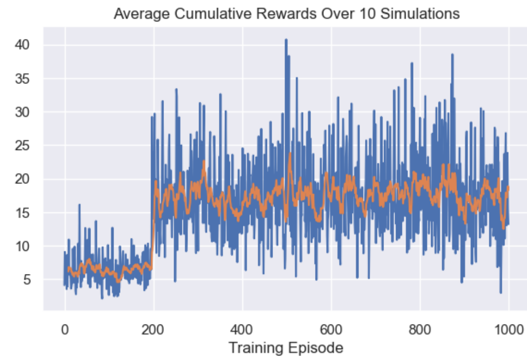
2 n = 10

[33]:
```
myFrog_10 = Frog(n=10, gamma = 0.999, initial_alpha = 0.01, initial_eps = 0.999,␣
↪num_simulations = 10, hyperparameter_scheme = 3, rewards_choice = 1)
myFrog_10.q_learning(1000)
myFrog_10.all_plots()
```

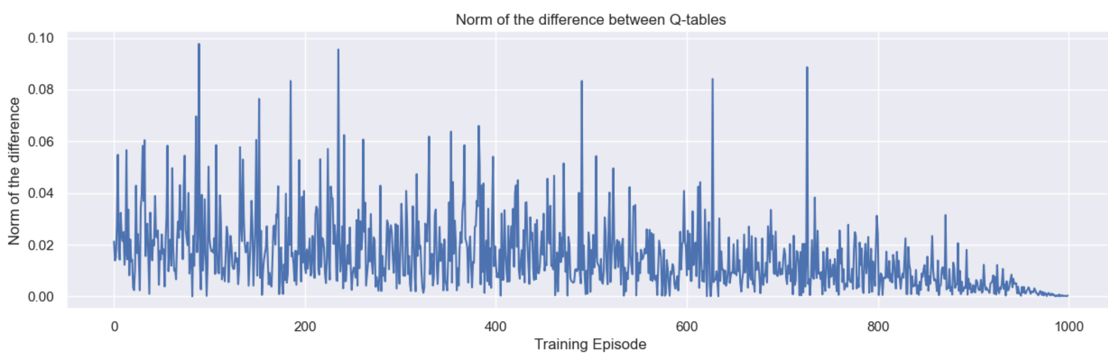
```
Episode:  0
Episode:  50
Episode:  100
Episode:  150
Episode:  200
Episode:  250
Episode:  300
Episode:  350
Episode:  400
Episode:  450
Episode:  500
Episode:  550
Episode:  600
Episode:  650
Episode:  700
Episode:  750
Episode:  800
Episode:  850
Episode:  900
```

Episode: 950





[34]:
```
fig = plt.figure(figsize=(15, 4))
plt.plot(myFrog_10.list_norm_differences)
plt.title("Norm of the difference between Q-tables")
plt.xlabel('Training Episode')
plt.ylabel('Norm of the difference')
plt.show()
fig.savefig('norm_of_difference_10.png')
```

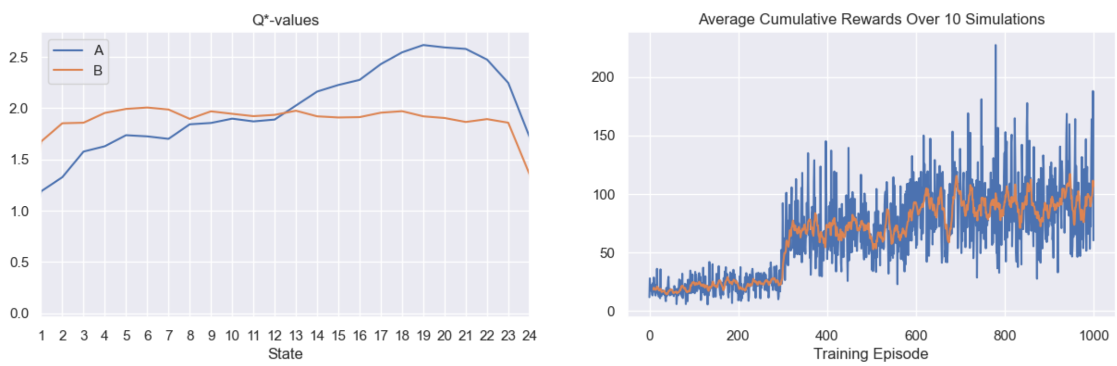


3 n = 25

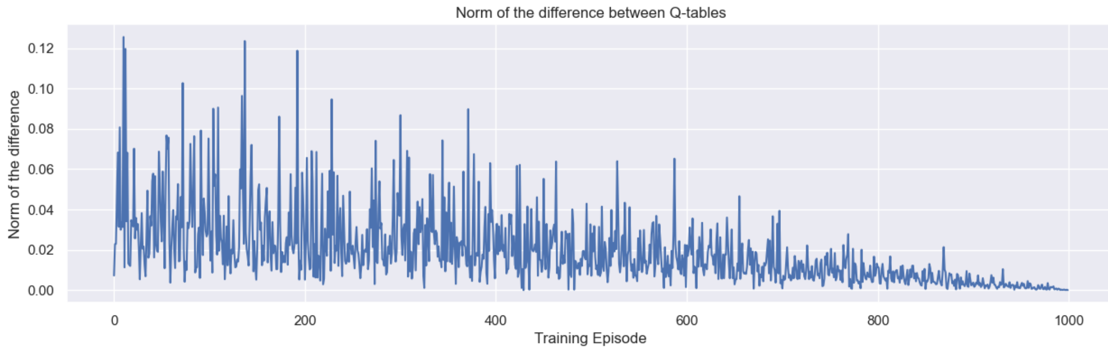
[35]:
```
myFrog_25 = Frog(n=25, gamma = 0.999, initial_alpha = 0.01, initial_eps = 0.999,␣
↪num_simulations = 10, hyperparameter_scheme = 2, rewards_choice = 1)
myFrog_25.q_learning(1000)
myFrog_25.all_plots()
```

Episode: 0
Episode: 50
Episode: 100
Episode: 150

```
Episode:  200
Episode:  250
Episode:  300
Episode:  350
Episode:  400
Episode:  450
Episode:  500
Episode:  550
Episode:  600
Episode:  650
Episode:  700
Episode:  750
Episode:  800
Episode:  850
Episode:  900
Episode:  950
```



[36]:
```python
fig = plt.figure(figsize=(15, 4))
plt.plot(myFrog_25.list_norm_differences)
plt.title("Norm of the difference between Q-tables")
plt.xlabel('Training Episode')
plt.ylabel('Norm of the difference')
plt.show()
fig.savefig('norm_of_difference_25.png')
```


Norm of the difference between Q-tables
Norm of the difference
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0
200
400
600
800
1000
Training Episode

[ ]:

reinforcement_learning_lbo

October 21, 2022

1 Libraries

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
import plotly.graph_objects as go
import datetime
import seaborn as sns
plt.style.use('seaborn')
```

2 Reading the data

[2]:
```
df_message = pd.read_csv('LOBSTER_SampleFile_AAPL_2012-06-21_5/
 ↪AAPL_2012-06-21_34200000_57600000_message_5.csv', index_col=0, header = None)
df_orderbook = pd.read_csv("LOBSTER_SampleFile_AAPL_2012-06-21_5/
 ↪AAPL_2012-06-21_34200000_57600000_orderbook_5.csv", header = None)
```

[3]:
```
df_message = df_message.reset_index()
df_message.columns = ['Time', 'Type', 'OrderID', 'Size', 'Price', 'Direction']
df_orderbook.columns = ['Ask Price 1', 'Ask Size 1', 'Bid Price 1', 'Bid Size␣
 ↪1', 'Ask Price 2', 'Ask Size 2', 'Bid Price 2', 'Bid Size 2', 'Ask Price 3',␣
 ↪'Ask Size 3', 'Bid Price 3', 'Bid Size 3', 'Ask Price 4', 'Ask Size 4', 'Bid␣
 ↪Price 4', 'Bid Size 4', 'Ask Price 5', 'Ask Size 5', 'Bid Price 5', 'Bid Size␣
 ↪5']
```

[4]:
```
df = pd.merge(left = df_message, right = df_orderbook, left_index = True,␣
 ↪right_index = True)
df['index_time'] = pd.to_datetime(df['Time'], unit='s')
df['index_time_precise'] = df['index_time'].dt.strftime("%H:%M:%S.%f")
df['index_time'] = df['index_time'].dt.strftime("%H:%M:%S")
df.drop(['OrderID', 'Direction', 'Type'], axis = 1, inplace = True)

df
```

[4]:
```
                Time  Size    Price  Ask Price 1  Ask Size 1  Bid Price 1  \
0       34200.004241    18  5853300      5859400         200      5853300
```

```
1       34200.004261    18  5853200      5859400          200      5853300
2       34200.004447    18  5853100      5859400          200      5853300
3       34200.025552    18  5859100      5859100           18      5853300
4       34200.025580    18  5859200      5859100           18      5853300
...              ...   ...      ...          ...          ...          ...
301582  57599.444020   103  5776000      5776700          300      5776000
301583  57599.444020    11  5776000      5776700          300      5775400
301584  57599.444020    48  5776100      5776100           48      5775400
301585  57599.913118    48  5776100      5776700          300      5775400
301586  57599.913118    52  5776350      5776700          300      5775400

        Bid Size 1  Ask Price 2  Ask Size 2  Bid Price 2  ...  Ask Price 4  \
0               18      5859800         200      5853000  ...      5868900
1               18      5859800         200      5853200  ...      5868900
2               18      5859800         200      5853200  ...      5868900
3               18      5859400         200      5853200  ...      5861000
4               18      5859200          18      5853200  ...      5859800
...            ...          ...         ...          ...  ...          ...
301582          11      5776800         200      5775400  ...      5777000
301583         410      5776800         200      5775300  ...      5777000
301584         410      5776700         300      5775300  ...      5776900
301585         410      5776800         200      5775300  ...      5777000
301586         410      5776800         200      5775300  ...      5777000

        Ask Size 4  Bid Price 4  Bid Size 4  Ask Price 5  Ask Size 5  \
0              300      5850100          89      5869500          50
1              300      5851000           5      5869500          50
2              300      5853000         150      5869500          50
3              200      5853000         150      5868900         300
4              200      5853000         150      5861000         200
...            ...          ...         ...          ...         ...
301582        1624      5775200         460      5777100         400
301583        1624      5775100         500      5777100         400
301584         160      5775100         500      5777000        1624
301585        1624      5775100         500      5777100         400
301586        1624      5775100         500      5777100         400

        Bid Price 5  Bid Size 5  index_time  index_time_precise
0           5849700           5    09:30:00     09:30:00.004241
1           5850100          89    09:30:00     09:30:00.004260
2           5851000           5    09:30:00     09:30:00.004447
3           5851000           5    09:30:00     09:30:00.025551
4           5851000           5    09:30:00     09:30:00.025579
...             ...         ...         ...                 ...
301582      5775100         500    15:59:59     15:59:59.444019
301583      5775000        2755    15:59:59     15:59:59.444019
301584      5775000        2755    15:59:59     15:59:59.444019
```

```
301585       5775000       2755    15:59:59     15:59:59.913117
301586       5775000       2755    15:59:59     15:59:59.913117

[301587 rows x 25 columns]
```

[5]:
```
def plot_data(df):
    fig = go.Figure()

    fig.add_trace(go.Scatter(x = df['index_time'], y = df['Ask Price 5'], name =␣
 ↪'Ask Price 5'))
    fig.add_trace(go.Scatter(x = df['index_time'], y = df['Bid Price 5'], name =␣
 ↪'Bid Price 5'))
    fig.add_trace(go.Scatter(x = df['index_time'], y = df['Price'], name =␣
 ↪'Price'))
    fig.update_layout(title = 'Price vs Time', xaxis_title = 'Time', yaxis_title␣
 ↪= 'Price')

    fig.show()

plot_data(df)
```

3 Creation of features without resampling data

[6]:
```
# Creation of features
# Spread
df['Spread'] = df['Ask Price 1'] - df['Bid Price 1']

# Bid Ask Volume imbalance
df['Bid_Ask_imbalance'] = df['Bid Size 1'] / df['Ask Size 1']

# Price Volatility
df['Price_Volatility'] = df['Price'].rolling(100).std()

# Spread Volatility
df['Spread_Volatility'] = df['Spread'].rolling(100).std()

# Volume traded in the last 100 ticks
df['Volume_traded_in_100_ticks'] = df['Size'].rolling(100).sum()
```

4 Creation of features with resampling data

[7]:
```
df["Time"]=df["Time"].apply(lambda x: datetime.datetime.strptime(str(datetime.
 ↪timedelta(seconds=x)), "%H:%M:%S.%f"))
df = df.set_index("Time").groupby(pd.Grouper(freq='10S')).last()
```

[8]:
```
# Mid price move
df['Mid_Price'] = (df['Ask Price 1'] + df['Bid Price 1']) / 2
df['Mid_Price_Variation'] = df['Mid_Price'].diff()

# Liquidity
df['Volume'] = df['Ask Size 1'] + df['Bid Size 1']

# Volume traded in the last 15 minutes
df['Volume_traded_in_15_minutes'] = df['Size'].rolling(90).sum()
```

5 Stationnarize features

[9]:
```
df.columns
```

[9]:
```
Index(['Size', 'Price', 'Ask Price 1', 'Ask Size 1', 'Bid Price 1',
       'Bid Size 1', 'Ask Price 2', 'Ask Size 2', 'Bid Price 2', 'Bid Size 2',
       'Ask Price 3', 'Ask Size 3', 'Bid Price 3', 'Bid Size 3', 'Ask Price 4',
       'Ask Size 4', 'Bid Price 4', 'Bid Size 4', 'Ask Price 5', 'Ask Size 5',
       'Bid Price 5', 'Bid Size 5', 'index_time', 'index_time_precise',
       'Spread', 'Bid_Ask_imbalance', 'Price_Volatility', 'Spread_Volatility',
       'Volume_traded_in_100_ticks', 'Mid_Price', 'Mid_Price_Variation',
       'Volume', 'Volume_traded_in_15_minutes'],
      dtype='object')
```

[10]:
```
columns_to_normalize = ['Size',
                        'Price',
                        'Ask Price 1',
                        'Ask Size 1',
                        'Bid Price 1',
                        'Bid Size 1',
                        'Ask Price 2',
                        'Ask Size 2',
                        'Bid Price 2',
                        'Bid Size 2',
                        'Ask Price 3',
                        'Ask Size 3',
                        'Bid Price 3',
                        'Bid Size 3',
                        'Ask Price 4',
                        'Ask Size 4',
                        'Bid Price 4',
                        'Bid Size 4',
                        'Ask Price 5',
                        'Ask Size 5',
                        'Bid Price 5',
                        'Bid Size 5',
                        'Spread',
```

```
                              'Bid_Ask_imbalance',
                              'Price_Volatility',
                              'Spread_Volatility',
                              'Volume_traded_in_100_ticks',
                              'Volume',
                              'Volume_traded_in_15_minutes']

df[columns_to_normalize] = np.log(df[columns_to_normalize])
```

6 Plot features

6.1 Spread

[11]:
```
def plot_spread(df):
    fig = go.Figure()

    fig.add_trace(go.Scatter(x = df.index, y = df['Spread'], name='Spread'))
    fig.update_layout(title = 'Spread vs Time', xaxis_title = 'Time',␣
 ↪yaxis_title = 'Spread')
    fig.show()
```

[12]:
```
plot_spread(df)
```

[13]:
```
def plot_hist_spread(df):
    fig = go.Figure()

    fig.add_trace(go.Histogram(x = df['Spread'], name='Spread',␣
 ↪histnorm='probability', nbinsx=60))

    fig.update_layout(title = 'Spread Histogram', xaxis_title = 'Spread',␣
 ↪yaxis_title = 'Frequency')
    fig.show()

plot_hist_spread(df)
```

7 Bid/Ask Imbalance

[14]:
```
def plot_bid_ask_imbalance(df):
    fig = go.Figure()

    fig.add_trace(go.Scatter(x = df.index, y = df['Bid_Ask_imbalance'],␣
 ↪name='Bid_Ask_imbalance'))
    fig.update_layout(title = 'Bid_Ask_imbalance vs Time', xaxis_title = 'Time',␣
 ↪yaxis_title = 'Bid_Ask_imbalance')
    fig.show()
```

```
plot_bid_ask_imbalance(df)
```

[15]:
```
def plot_hist_Bid_Ask_imbalance(df):
    fig = go.Figure()

    fig.add_trace(go.Histogram(x = df['Bid_Ask_imbalance'],␣
 ↪name='Bid_Ask_imbalance', histnorm='probability', nbinsx=50, xbins=dict( #␣
 ↪bins used for histogram
        start=0,
        end=400,
    )))

    fig.update_layout(title = 'Bid_Ask_imbalance Histogram', xaxis_title =␣
 ↪'Bid_Ask_imbalance', yaxis_title = 'Probability')
    fig.show()

plot_hist_Bid_Ask_imbalance(df)
```

8 Price Volatility

[16]:
```
def plot_price_volatility(df):
    fig = go.Figure()

    fig.add_trace(go.Scatter(x = df.index, y = df['Price_Volatility'],␣
 ↪name='Price_Volatility'))
    fig.update_layout(title = 'Price_Volatility vs Time', xaxis_title = 'Time',␣
 ↪yaxis_title = 'Price_Volatility')
    fig.show()
plot_price_volatility(df)
```

[17]:
```
def plot_hist_Price_Volatility(df):
    fig = go.Figure()

    fig.add_trace(go.Histogram(x = df['Price_Volatility'],␣
 ↪name='Price_Volatility', histnorm='probability', nbinsx=50, xbins=dict( # bins␣
 ↪used for histogram
        start=0,
        end=400,
    )))

    fig.update_layout(title = 'Price_Volatility Histogram', xaxis_title =␣
 ↪'Price_Volatility', yaxis_title = 'Probability')
    fig.show()

plot_hist_Price_Volatility(df)
```

```

```

9 Spread Volatility

[18]:
```
def plot_spread_volatility(df):
    fig = go.Figure()

    fig.add_trace(go.Scatter(x = df.index, y = df['Spread_Volatility'],␣
 ↪name='Spread_Volatility'))
    fig.update_layout(title = 'Spread_Volatility vs Time', xaxis_title = 'Time',␣
 ↪yaxis_title = 'Spread_Volatility')
    fig.show()
plot_spread_volatility(df)
```

[19]:
```
def plot_hist_Spread_Volatility(df):
    fig = go.Figure()

    fig.add_trace(go.Histogram(x = df['Spread_Volatility'],␣
 ↪name='Spread_Volatility', histnorm='probability', nbinsx=50, xbins=dict( #␣
 ↪bins used for histogram
        start=0,
        end=400,
    )))

    fig.update_layout(title = 'Spread_Volatility Histogram', xaxis_title =␣
 ↪'Spread_Volatility', yaxis_title = 'Probability')
    fig.show()

plot_hist_Spread_Volatility(df)
```

[ ]:
```

```

10 Volume_traded_in_100_ticks

[20]:
```
def plot_Volume_traded_in_100_ticks(df):
    fig = go.Figure()

    fig.add_trace(go.Scatter(x = df.index, y = df['Volume_traded_in_100_ticks'],␣
 ↪name='Volume_traded_in_100_ticks'))
    fig.update_layout(title = 'Volume_traded_in_100_ticks vs Time', xaxis_title␣
 ↪= 'Time', yaxis_title = 'Volume_traded_in_100_ticks')
    fig.show()
plot_Volume_traded_in_100_ticks(df)
```

[21]:
```
def plot_hist_Volume_traded_in_100_ticks(df):
    fig = go.Figure()

    fig.add_trace(go.Histogram(x = df['Volume_traded_in_100_ticks'],␣
 ↪name='Volume_traded_in_100_ticks', histnorm='probability', nbinsx=50,␣
 ↪xbins=dict( # bins used for histogram
        start=0,
        end=400,
    )))

    fig.update_layout(title = 'Volume_traded_in_100_ticks Histogram',␣
 ↪xaxis_title = 'Volume_traded_in_100_ticks', yaxis_title = 'Probability')
    fig.show()

plot_hist_Volume_traded_in_100_ticks(df)
```

[ ]:

11 Volume

[22]:
```
def plot_Volume(df):
    fig = go.Figure()

    fig.add_trace(go.Scatter(x = df.index, y = df['Volume'], name='Volume'))
    fig.update_layout(title = 'Volume vs Time', xaxis_title = 'Time',␣
 ↪yaxis_title = 'Volume')
    fig.show()
plot_Volume(df)
```

[23]:
```
def plot_hist_Volume(df):
    fig = go.Figure()

    fig.add_trace(go.Histogram(x = df['Volume'], name='Volume',␣
 ↪histnorm='probability', nbinsx=50, xbins=dict( # bins used for histogram
        start=0,
        end=400,
    )))

    fig.update_layout(title = 'Volume Histogram', xaxis_title = 'Volume',␣
 ↪yaxis_title = 'Probability')
    fig.show()

plot_hist_Volume(df)
```

12 Volume_traded_in_15_minutes

[24]:
```
def plot_Volume_traded_in_15_minutes(df):
    fig = go.Figure()

    fig.add_trace(go.Scatter(x = df.index, y =␣
 ↪df['Volume_traded_in_15_minutes'], name='Volume_traded_in_15_minutes'))
    fig.update_layout(title = 'Volume_traded_in_15_minutes vs Time', xaxis_title␣
 ↪= 'Time', yaxis_title = 'Volume_traded_in_15_minutes')
    fig.show()
plot_Volume_traded_in_15_minutes(df)
```

[25]:
```
def plot_hist_Volume_traded_in_15_minutes(df):
    fig = go.Figure()

    fig.add_trace(go.Histogram(x = df['Volume_traded_in_15_minutes'],␣
 ↪name='Volume_traded_in_15_minutes', histnorm='probability', nbinsx=50,␣
 ↪xbins=dict( # bins used for histogram
        start=0,
        end=400,
    )))

    fig.update_layout(title = 'Volume_traded_in_15_minutes Histogram',␣
 ↪xaxis_title = 'Volume_traded_in_15_minutes', yaxis_title = 'Probability')
    fig.show()

plot_hist_Volume_traded_in_15_minutes(df)
```

13 Mid Price Variation

[26]:
```
def plot_Mid_Price_Variation(df):
    fig = go.Figure()

    fig.add_trace(go.Scatter(x = df.index, y = df['Mid_Price_Variation'],␣
 ↪name='Mid_Price_Variation'))
    fig.update_layout(title = 'Mid_Price_Variation vs Time', xaxis_title =␣
 ↪'Time', yaxis_title = 'Mid_Price_Variation')
    fig.show()
plot_Mid_Price_Variation(df)
```

[27]:
```
def plot_hist_Mid_Price_Variation(df):
    fig = go.Figure()

    fig.add_trace(go.Histogram(x = df['Mid_Price_Variation'],␣
 ↪name='Mid_Price_Variation', histnorm='probability', nbinsx=50,))
```

```
    fig.update_layout(title = 'Mid_Price_Variation Histogram', xaxis_title =␣
 ↪'Mid_Price_Variation', yaxis_title = 'Probability')
    fig.show()

plot_hist_Mid_Price_Variation(df)
```

[ ]:

```

```