Final Pj:bandit-arms

- Group Menber:
  - Bo Shen, 2022533182
  - Junhui Ge, 2022533047
  - Yang Wang, 2022533092

Part I: Classical Bandit Algorithms

Problem 1

Since

$$(oracle\ value)_j = N * \theta_j$$

Hence, here are the oracle value.

Arm j	1	2	3
θ_j	0.7	0.5	0.4
Oracle Value	3500	2500	2800

Problem 2, 3

In this problem, θ_j is the value in Problem 1.
We use

$$delta = |(oracle\ value)_{max} - (algorithm's\ reward)|$$

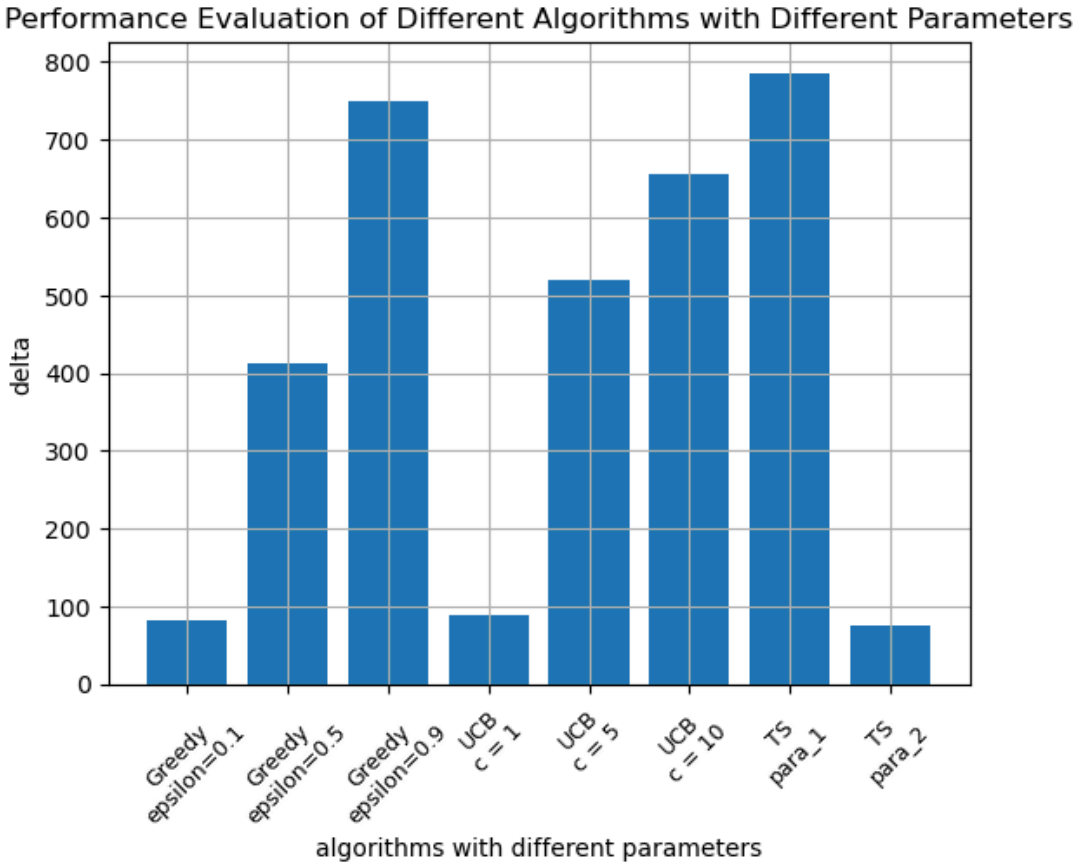
to evaluate the performance of different algorithm. The smaller the delta, the better the performance of the algorithm.

Here are the results of three algorithm with different parameters in a simulation.

- ϵ-greedy Algorithm
  - $\epsilon = 0.1$, $delta = 82.155$
  - $\epsilon = 0.5$, $delta = 413.525$
  - $\epsilon = 0.9$, $delta = 749.36$
- UCB Algorithm
  - $c = 1$, $delta = 88.610$
  - $c = 5$, $delta = 519.09$
  - $c = 10$, $delta = 654.785$
- TS Algorithm
  - $\{(\alpha_1, \beta_1) = (1, 1), (\alpha_2, \beta_2) = (1, 1), (\alpha_3, \beta_3) = (1, 1)\}$, $delta = 786$

- $\{(\alpha_1, \beta_1) = (601, 401), (\alpha_2, \beta_2) = (401, 601), (\alpha_3, \beta_3) = (2, 3)\}$, $delta = 75.0$

Here are the bar chart.



Problem 4

Random θ_j array

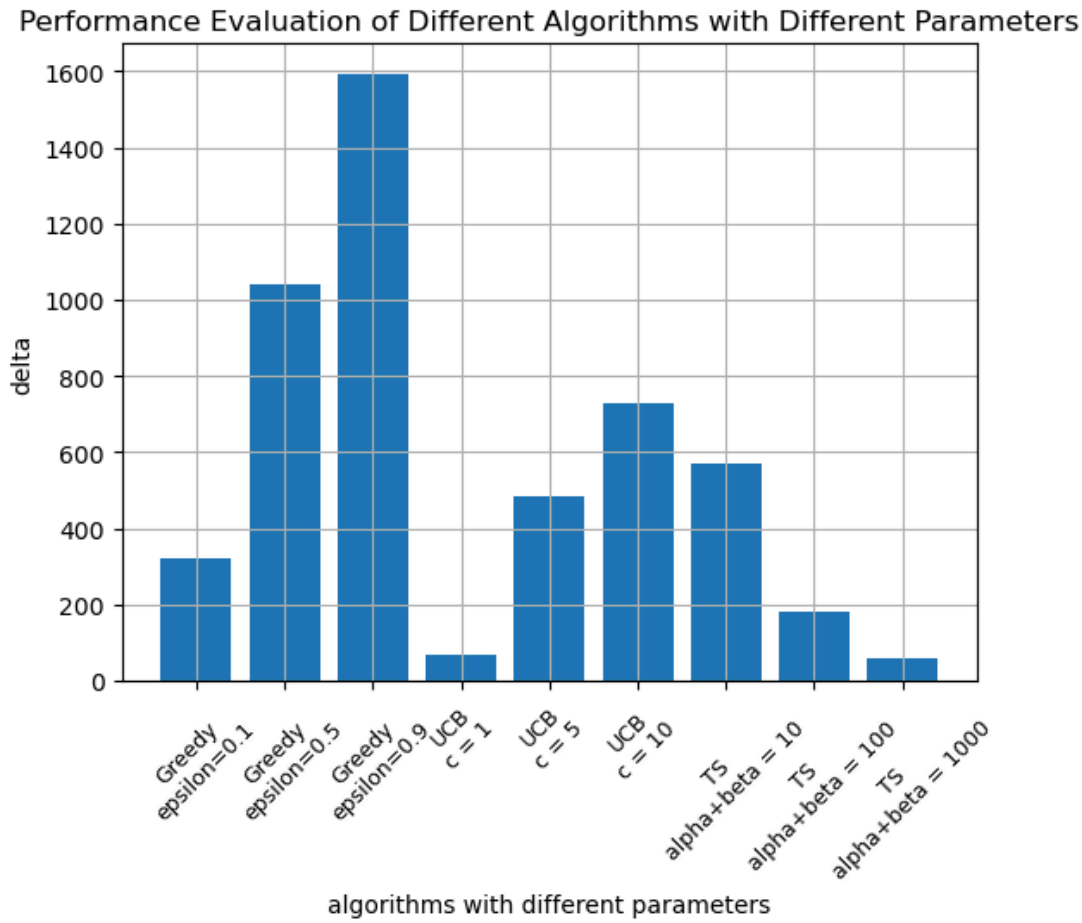
Firstly, we randomly generate 50 groups of $[\theta_1, \theta_2, \theta_3]$ and use the final average delta to evaluate performances of different algorithm with different parameters.

Here are the average results of three algorithm with different parameters for 50 simulation.

- ϵ-greedy Algorithm
  - $\epsilon = 0.1$, $average\ delta = 319.2$
  - $\epsilon = 0.5$, $average\ delta = 1038.6$
  - $\epsilon = 0.9$, $average\ delta = 1592.0$
- UCB Algorithm
  - $c = 1$, $average\ delta = 67.84$
  - $c = 5$, $average\ delta = 2485.9$
  - $c = 10$, $average\ delta = 727.7$
- TS Algorithm
  - $\{\alpha_i + \beta_i = 10, for\ i = 1, 2, 3\}$, $average\ delta = 570.1$
  - $\{\alpha_i + \beta_i = 100, for\ i = 1, 2, 3\}$, $average\ delta = 181.2$

- $\{\alpha_i + \beta_i = 1000, for\ i = 1, 2, 3\}$, $average\ delta = 57.60$

Here are the bar chart.



From the chart, we can find that:

- For ϵ-greedy Algorithm,
  - $\epsilon = 0.1$ performs best in $\epsilon \in \{0.1, 0.5, 0.9\}$
- For UCB Algorithm,
  - $c = 1$ performs best in $c \in \{1, 5, 10\}$
- For TS Algorithm,
  - $\{\alpha_i + \beta_i = 1000, for\ i = 1, 2, 3\}$ performs best in $\alpha_i + \beta_i \in \{10, 100, 1000\}$
- For all the algorithm,
  - $\{\alpha_i + \beta_i = 1000, for\ i = 1, 2, 3\}$ performs best in these three algorithm

θ_j array for specific data features

However, to find the most appropriate parameters for the θ_j array for specific data features, such as mean, standard deviation, we conducted further experiments for these three algoritms.

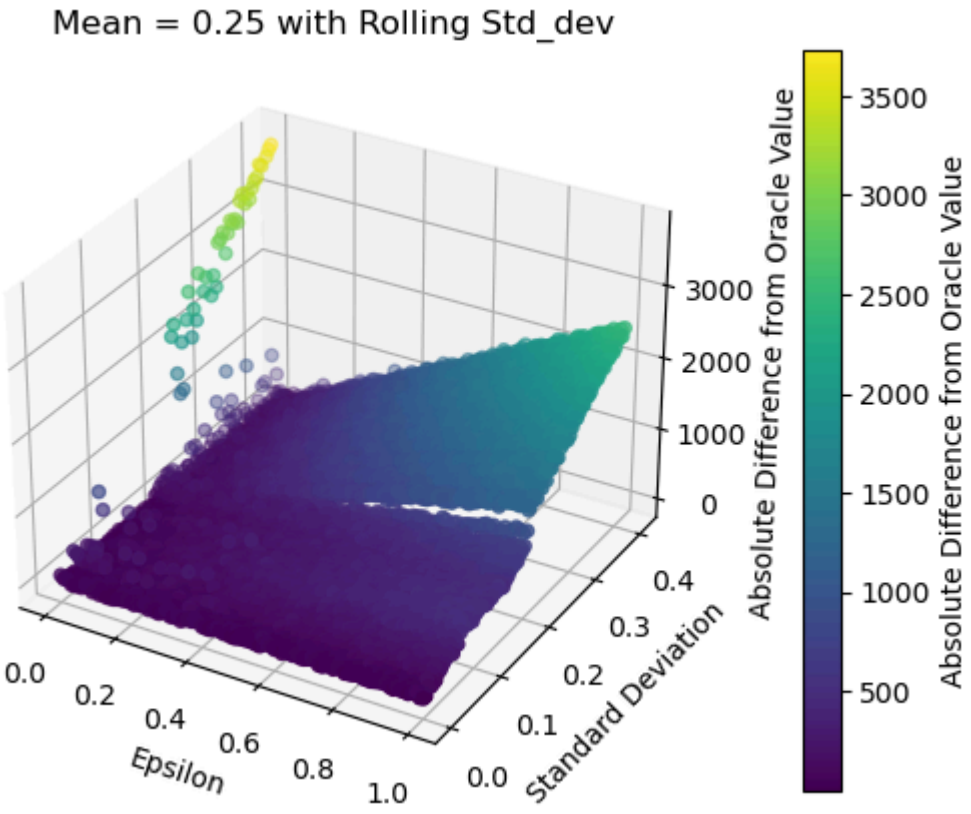
We set static $mean = 0.25, 0.5, 0.75$ with increasing std_dev in $(0, \sqrt{\frac{1}{6}})$.($\sqrt{\frac{1}{6}}$ is the maximum standard deviation for 3 numbers between 0 and 1). Simultaneously,

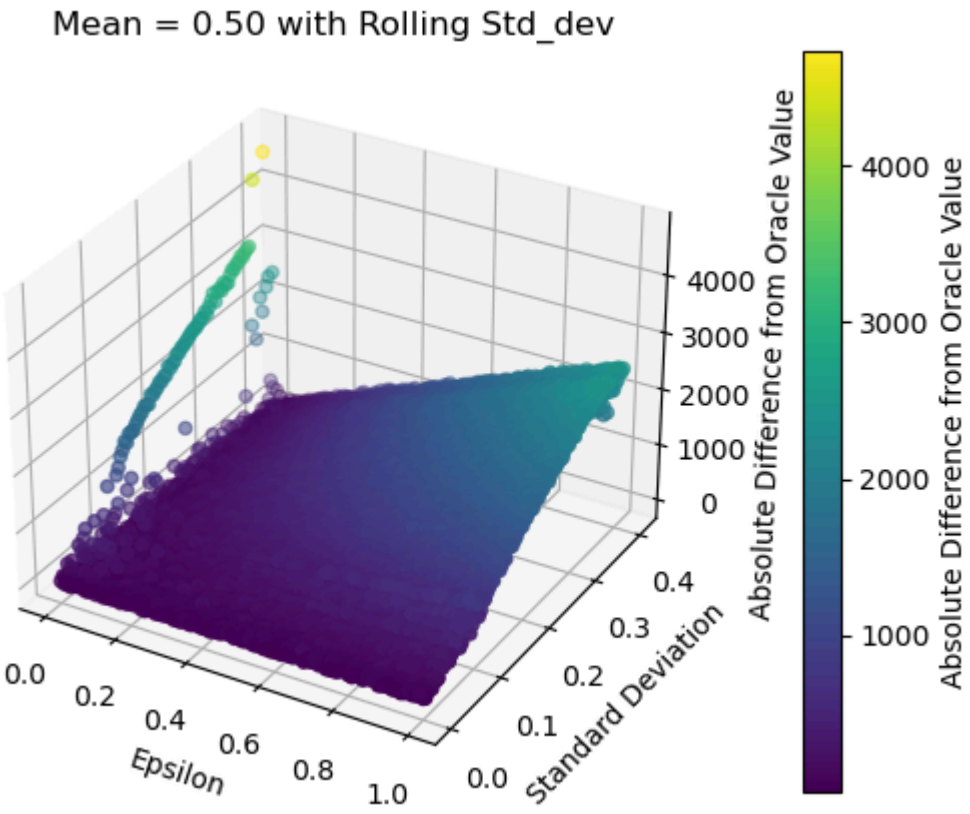
We set static $std\_dev = \frac{1}{4}\sqrt{\frac{1}{6}}, \frac{2}{4}\sqrt{\frac{1}{6}}, \frac{3}{4}\sqrt{\frac{1}{6}}$, with increasing $mean$ in $(0, 1)$.

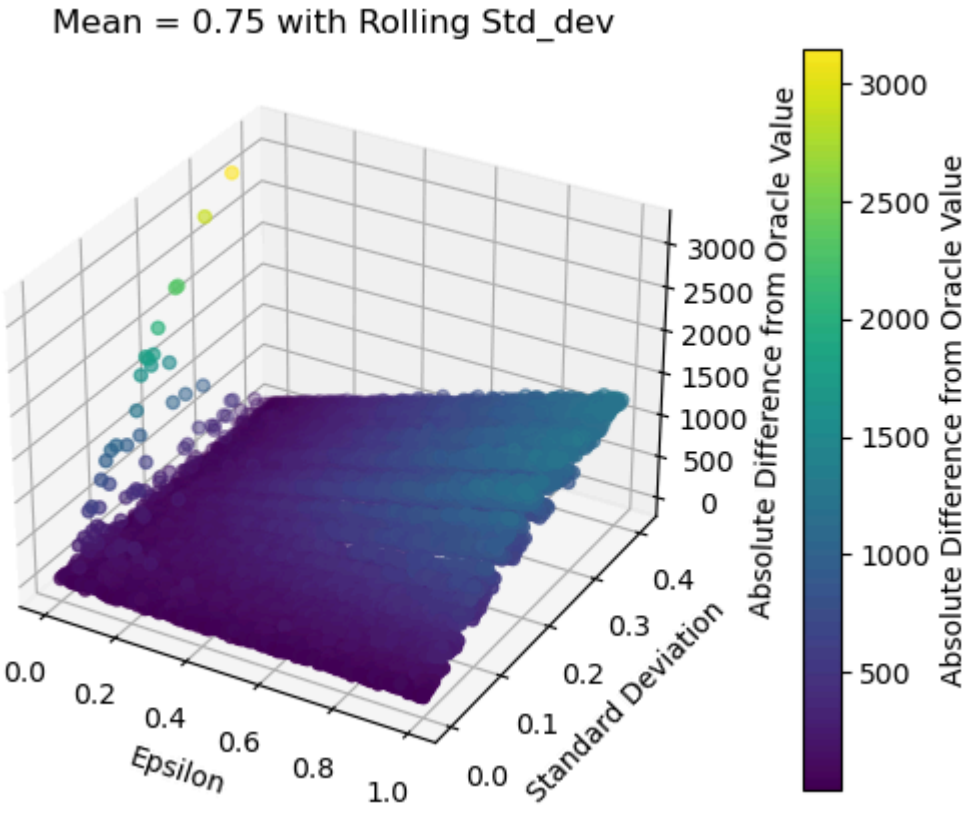
So for each algorithm, there are six charts showing the relationship between delta, parameter, and mean or standar d deviation.

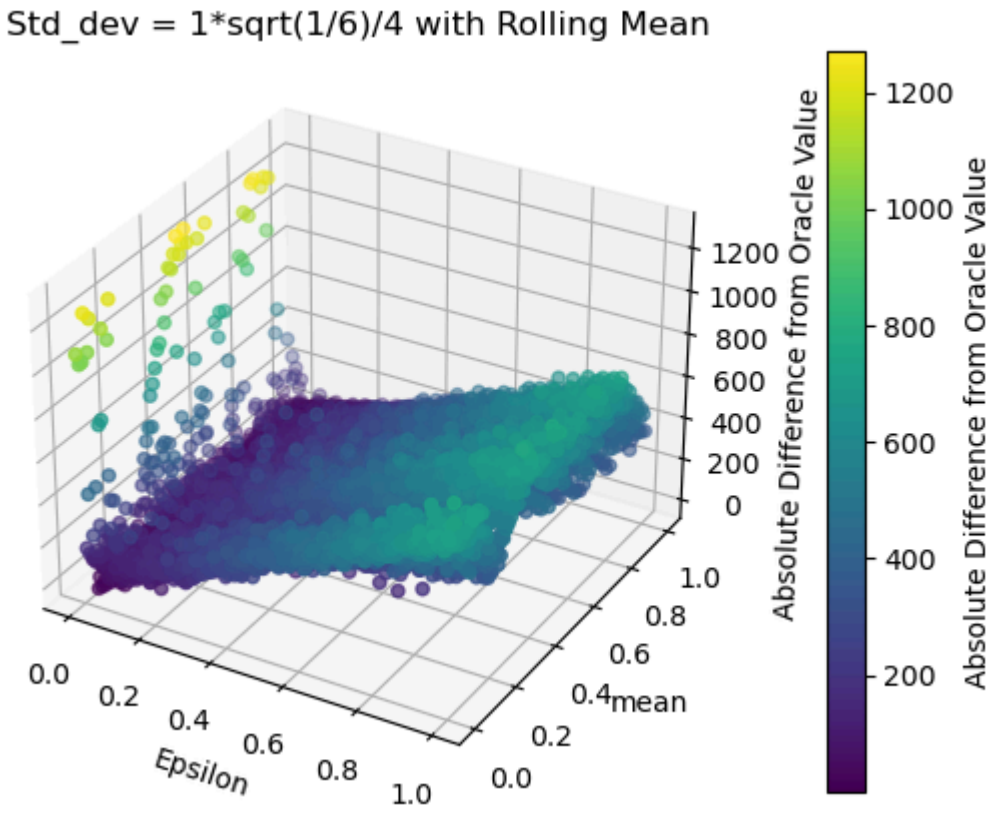
Here are results for three algorithm.

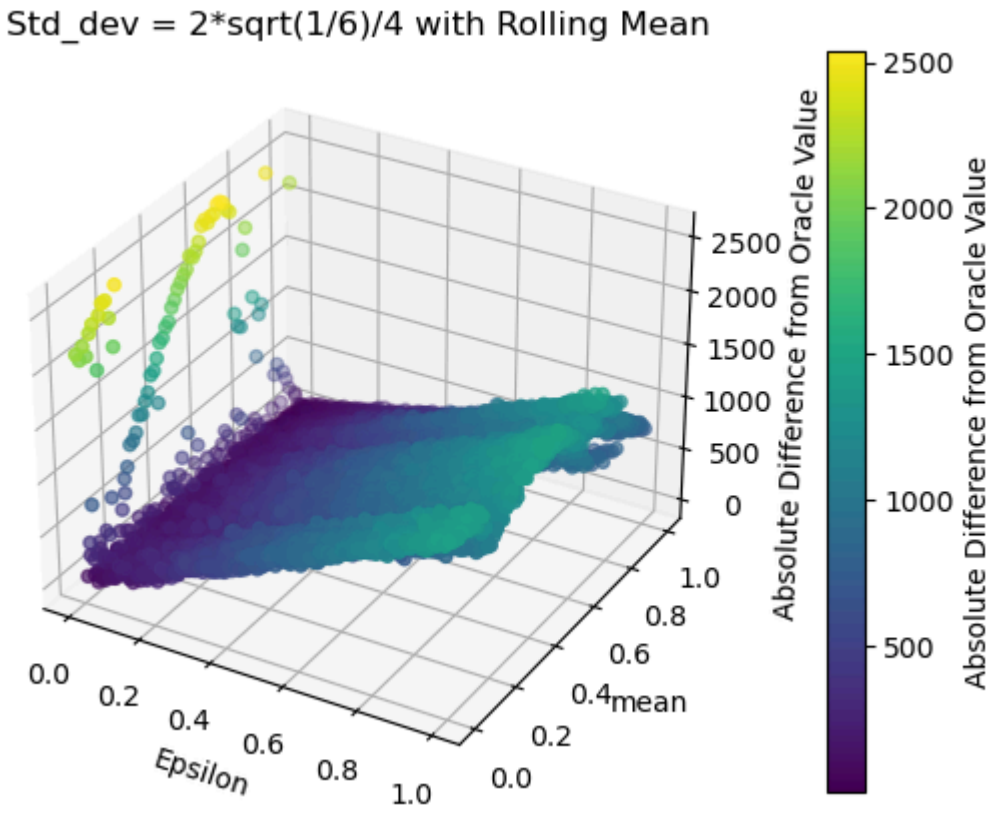
ϵ-greedy Algorithm

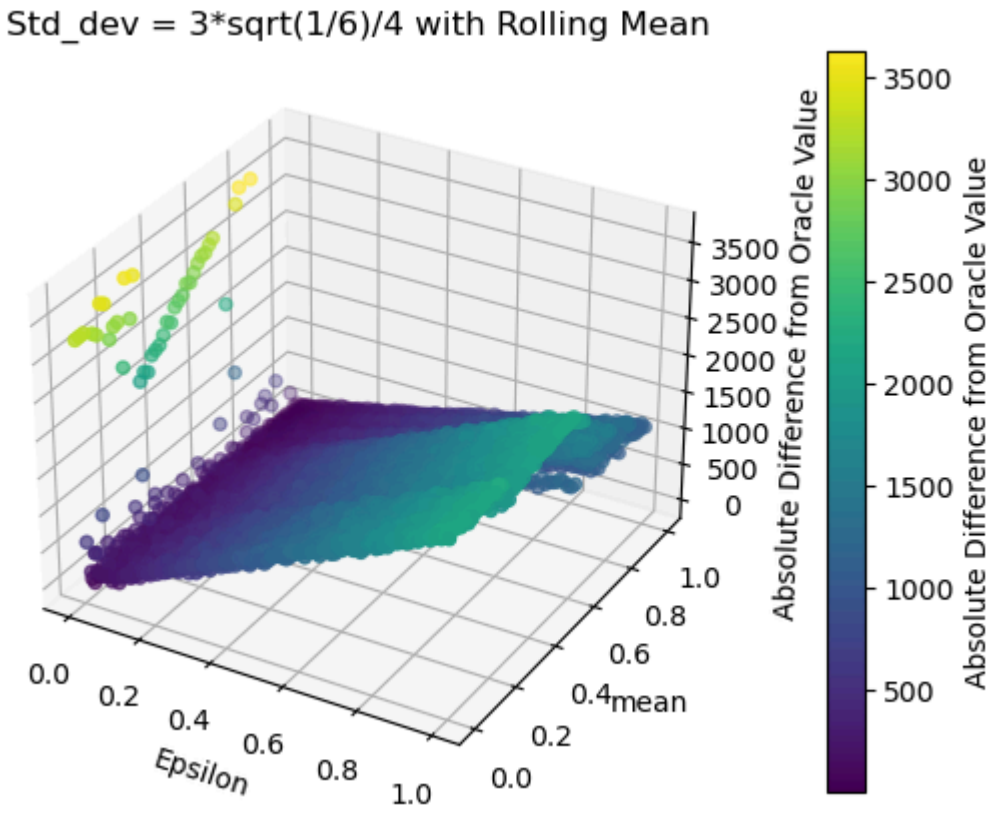



Mean = 0.50 with Rolling Std_dev
Epsilon
Standard Deviation
Absolute Difference from Oracle Value
Absolute Difference from Oracle Value


Mean = 0.75 with Rolling Std_dev
Epsilon
Standard Deviation
Absolute Difference from Oracle Value
Absolute Difference from Oracle Value


Std_dev = 1*sqrt(1/6)/4 with Rolling Mean
Epsilon
mean
Absolute Difference from Oracle Value
0.0
0.2
0.4
0.6
0.8
1.0
0
200
400
600
800
1000
1200


Std_dev = 2*sqrt(1/6)/4 with Rolling Mean
Epsilon
mean
Absolute Difference from Oracle Value
0.0
0.2
0.4
0.6
0.8
1.0
0
500
1000
1500
2000
2500

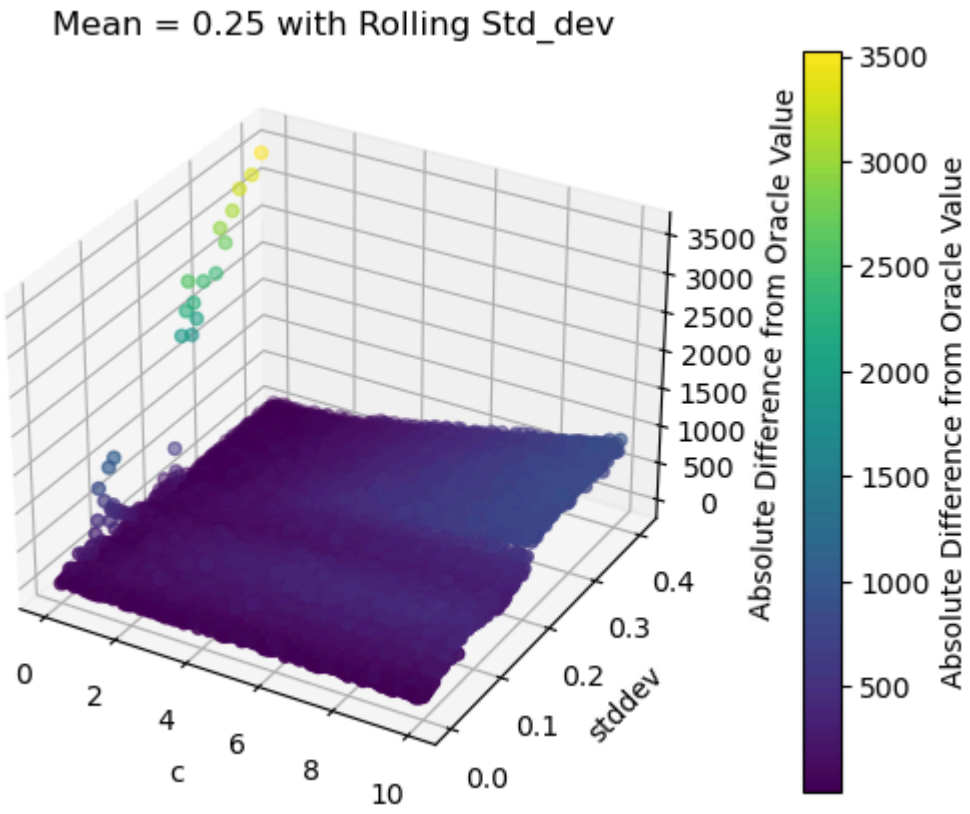


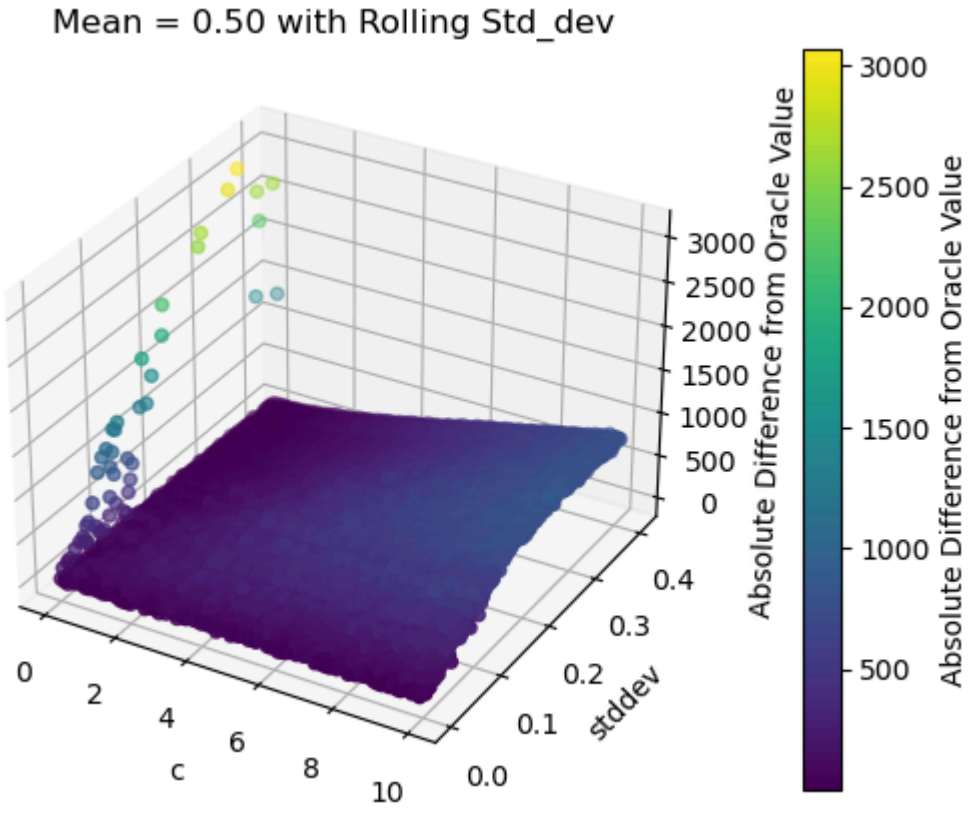
From these six 3D charts, we find that: When the mean is fixed, the larger the standard deviation of θ_j under the same ϵ, the larger the $delta$ generally.

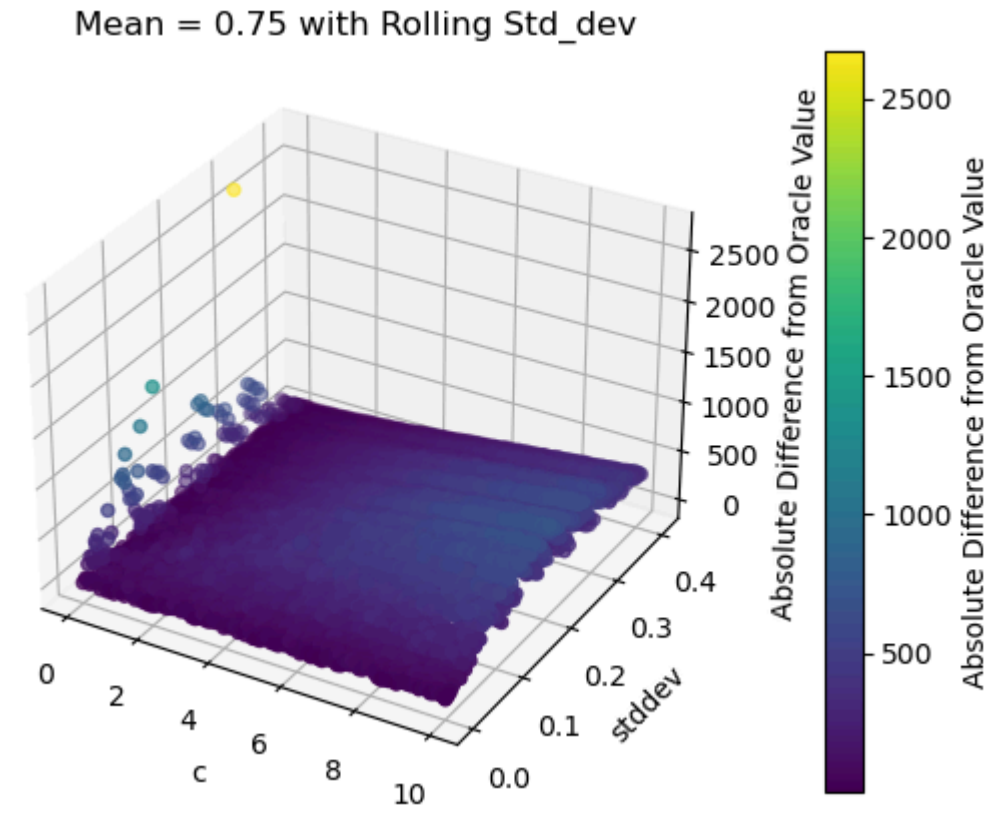
When the standard deviation is fixed, the mean of θ_j under the same ϵ has no significant impact on the $delta$.

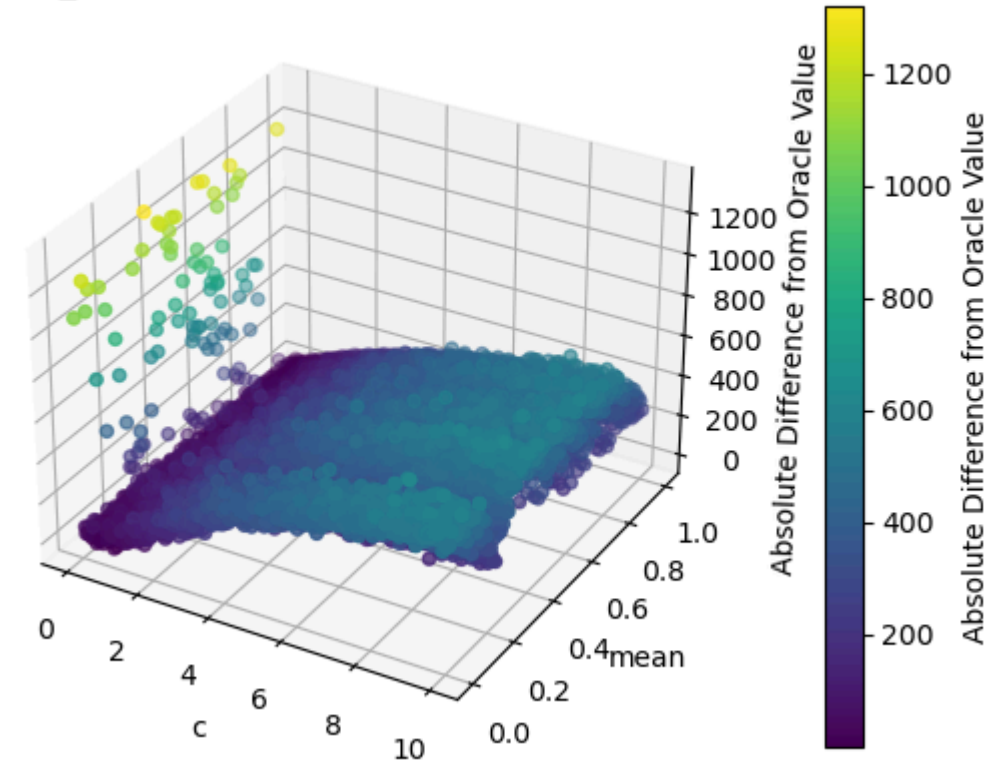
When observing ϵ, we observe a curve with a significant $delta$ around $\epsilon = 0$. This is because the ϵ is too small, causing the machine to stick to the initial selection and not explore. As ϵ increases, the $delta$ also increases. This is because the machine lacks the exploitation of existing high-quality information. So we need to choose the appropriate parameters ϵ.

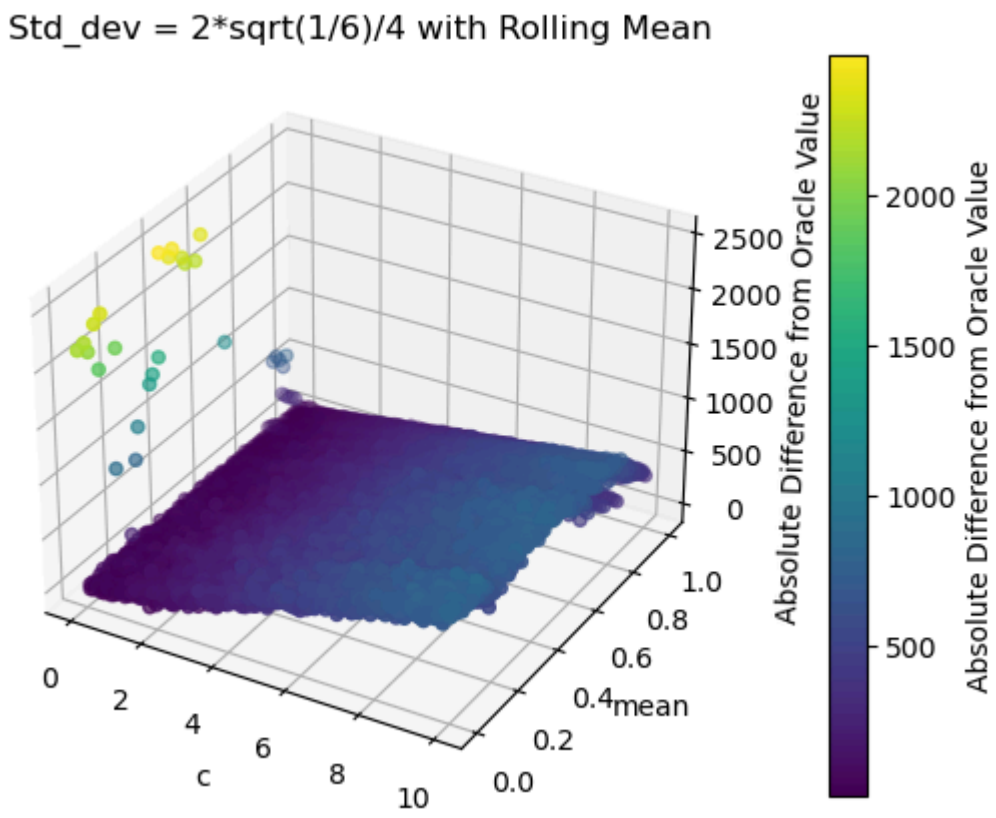
UCB Algorithm

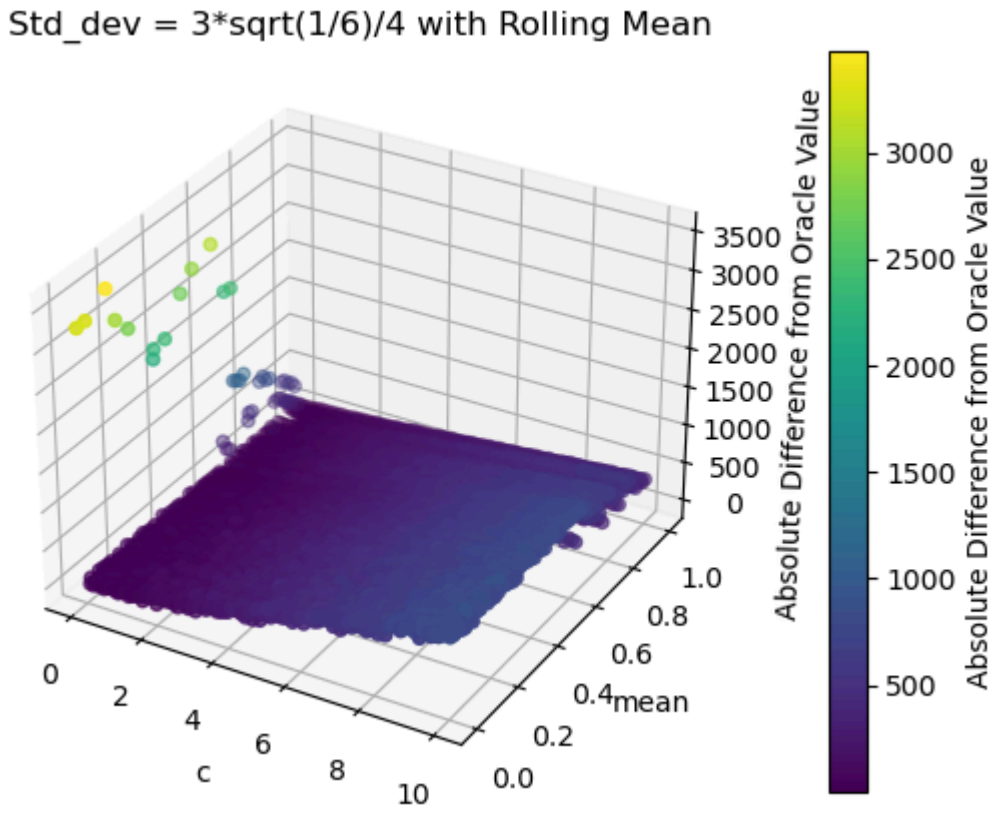

Mean = 0.25 with Rolling Std_dev
Absolute Difference from Oracle Value
c
stddev
0
2
4
6
8
10
0.0
0.1
0.2
0.3
0.4
0
500
1000
1500
2000
2500
3000
3500


Mean = 0.50 with Rolling Std_dev
Absolute Difference from Oracle Value
c
stddev
0
2
4
6
8
10
0.0
0.1
0.2
0.3
0.4
0
500
1000
1500
2000
2500
3000


Mean = 0.75 with Rolling Std_dev
Absolute Difference from Oracle Value
2500
2000
1500
1000
500
0
0.4
0.3
0.2
0.1
0.0
stddev
0
2
4
6
8
10
c
Absolute Difference from Oracle Value
2500
2000
1500
1000
500


Std_dev = 1*sqrt(1/6)/4 with Rolling Mean
Absolute Difference from Oracle Value
1200
1000
800
600
400
200
0
1.0
0.8
0.6
0.4
0.2
0.0
mean
0
2
4
6
8
10
c
Absolute Difference from Oracle Value
1200
1000
800
600
400
200



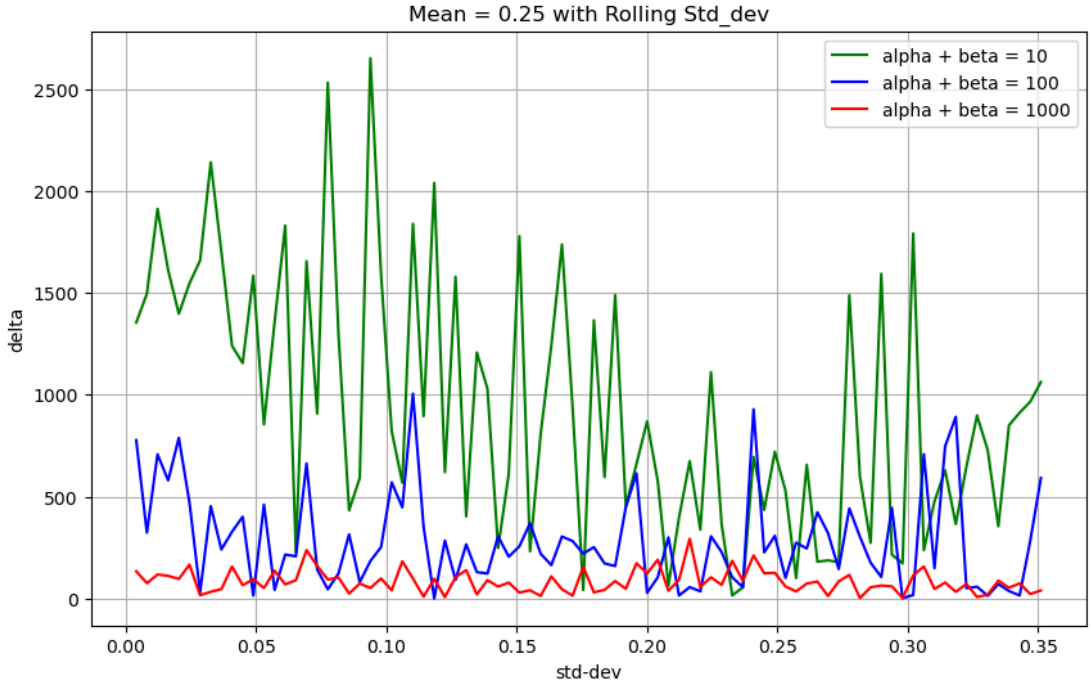


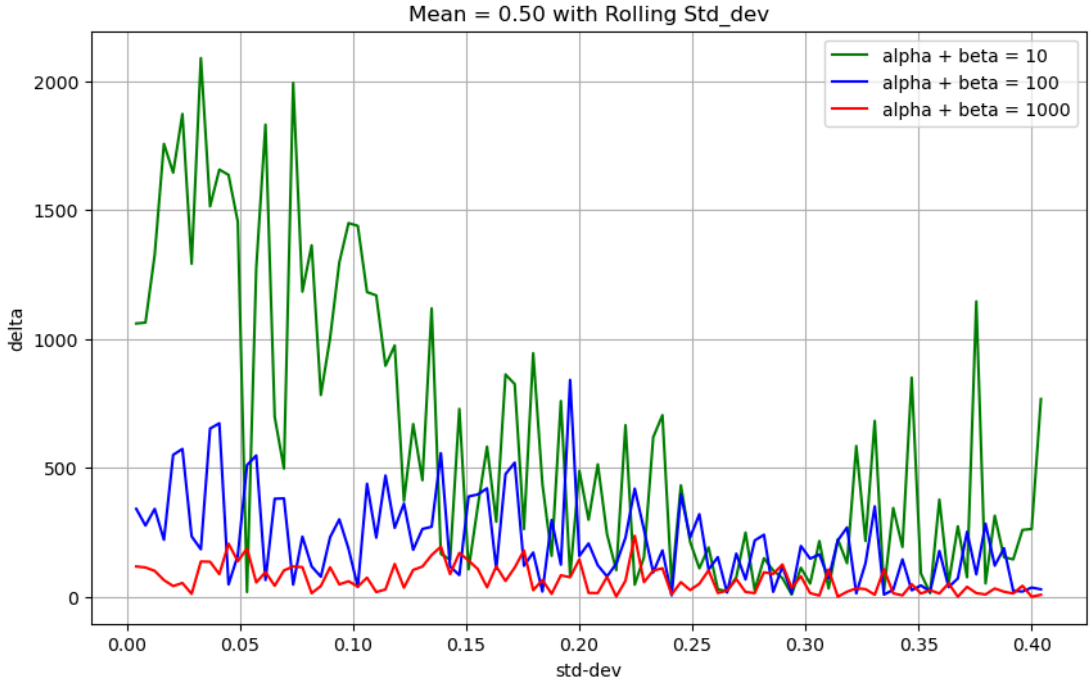
From these six 3D charts, we find that: When the mean is fixed, the larger the standard deviation of θ_j under the same c, the greater the $delta$ generally.

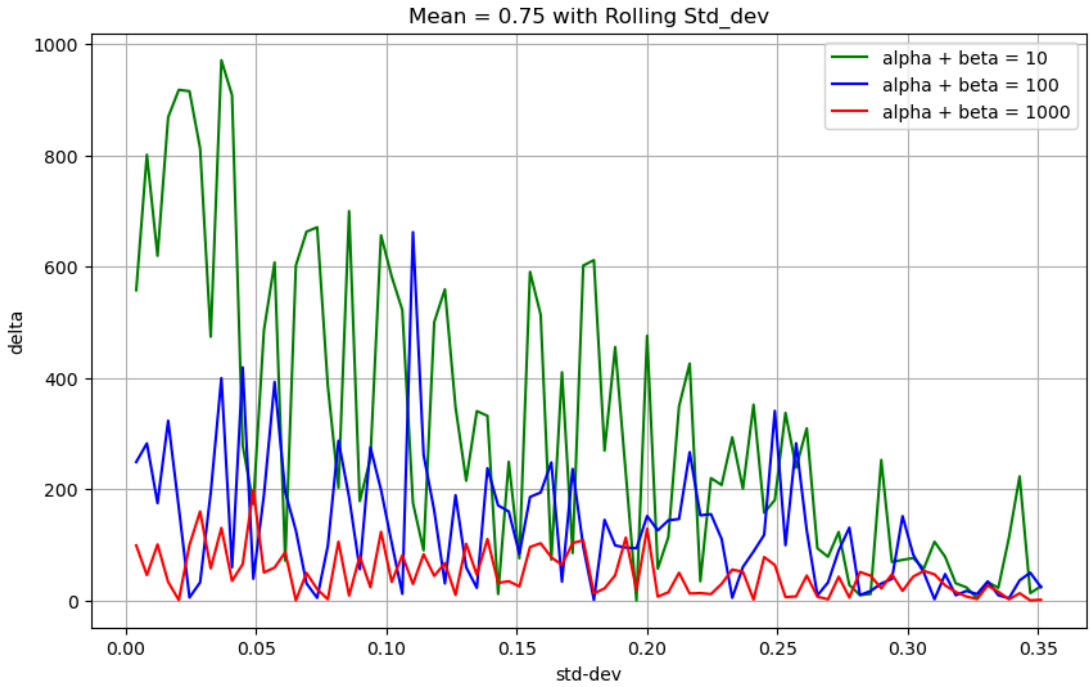
When the variance is fixed, the mean of θ_j under the same c has no significant impact on the $delta$.

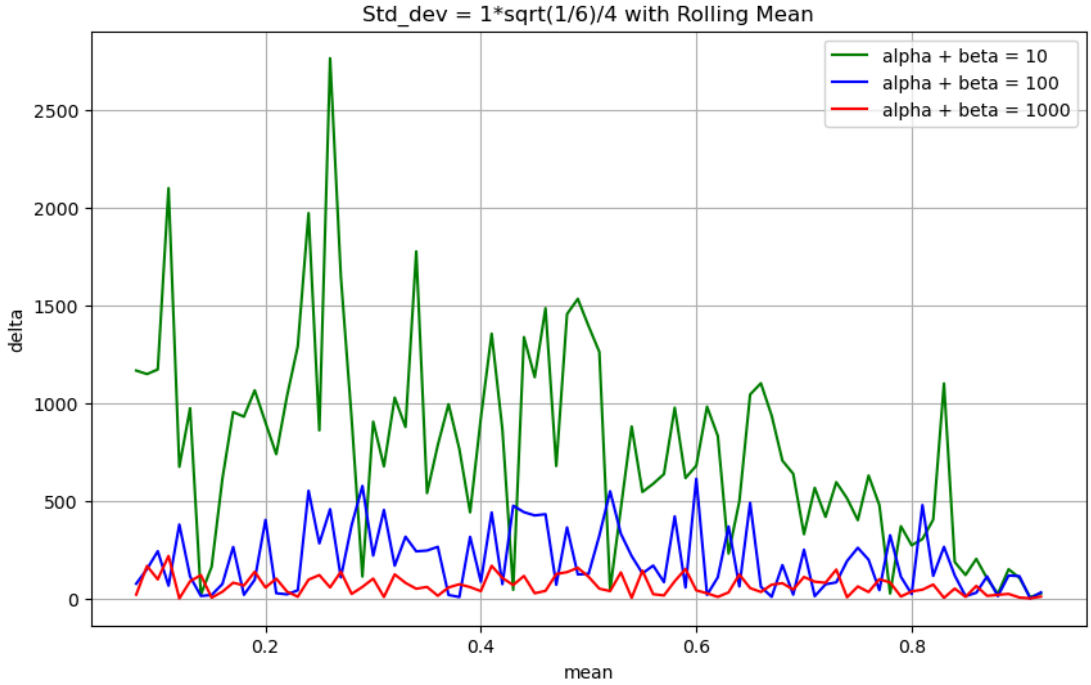
When observing c, we observe a curve with a significant $delta$ around $c = 0$. This is because c is too small, causing the machine to be limited to the initial choice and not explore. As c increases, the $delta$ also increases. This is because the machine lacks the exploitation of existing high-quality information. So we need to choose the appropriate parameters c.

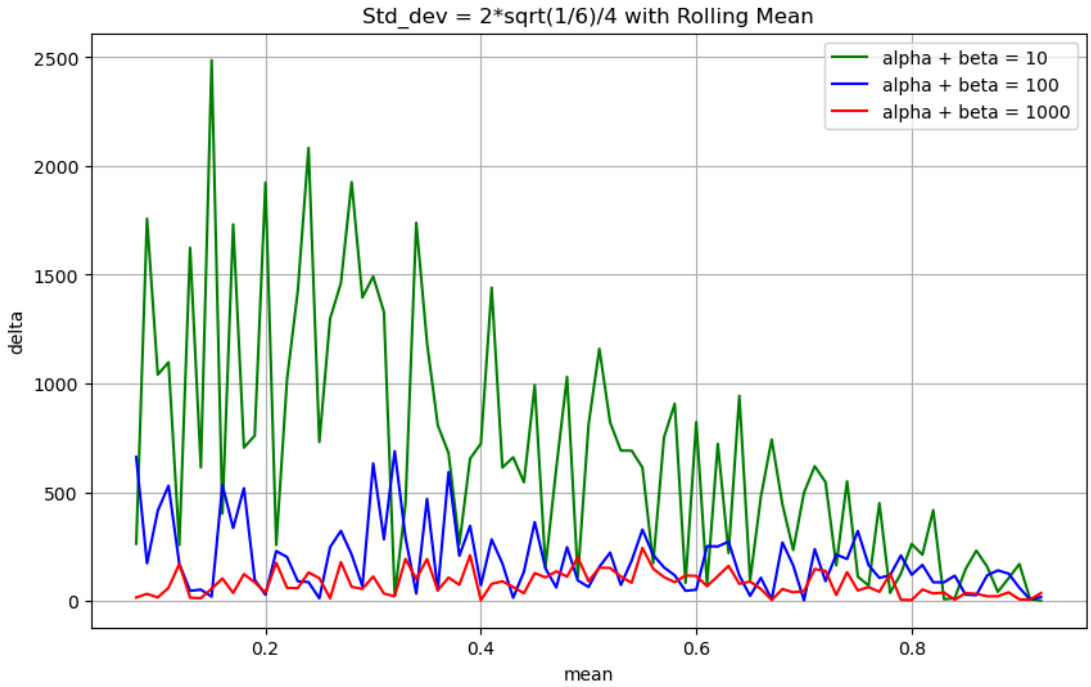
TS Algorithm

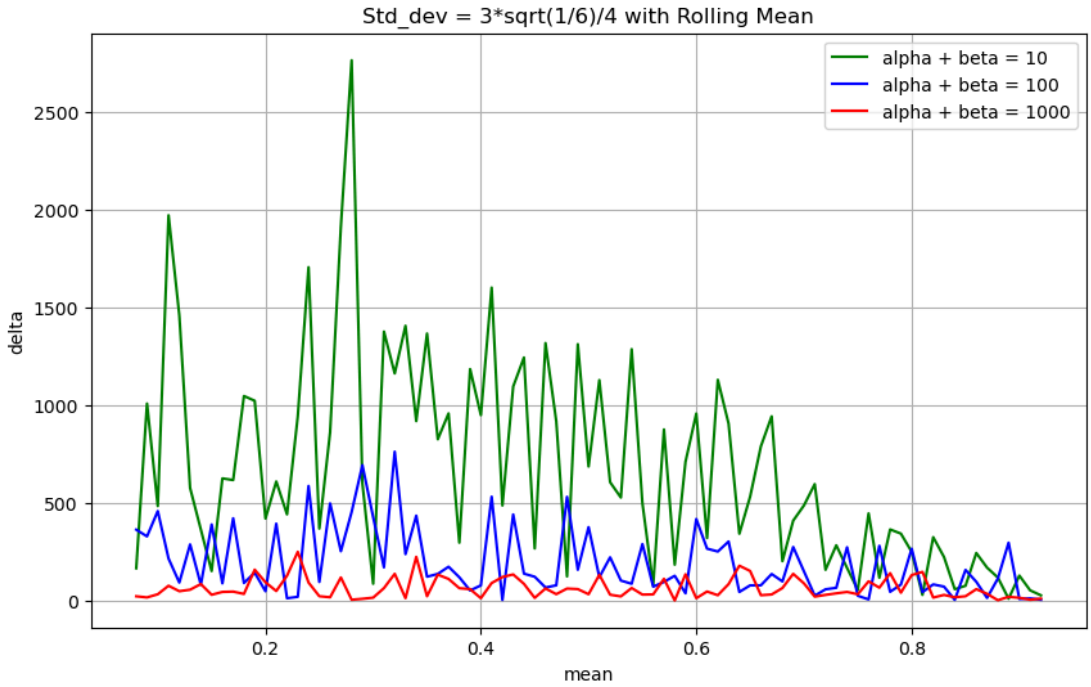





Mean = 0.75 with Rolling Std_dev
alpha + beta = 10
alpha + beta = 100
alpha + beta = 1000
delta
std-dev
0
200
400
600
800
1000
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35


Std_dev = 1*sqrt(1/6)/4 with Rolling Mean
alpha + beta = 10
alpha + beta = 100
alpha + beta = 1000
delta
mean
0
500
1000
1500
2000
2500
0.2
0.4
0.6
0.8



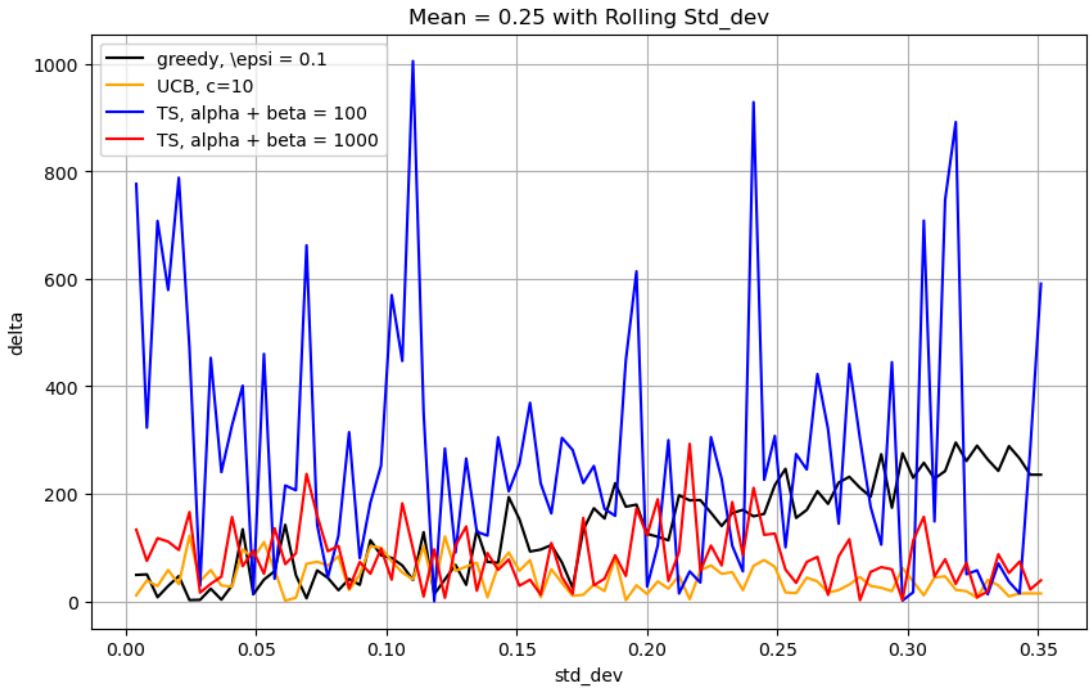


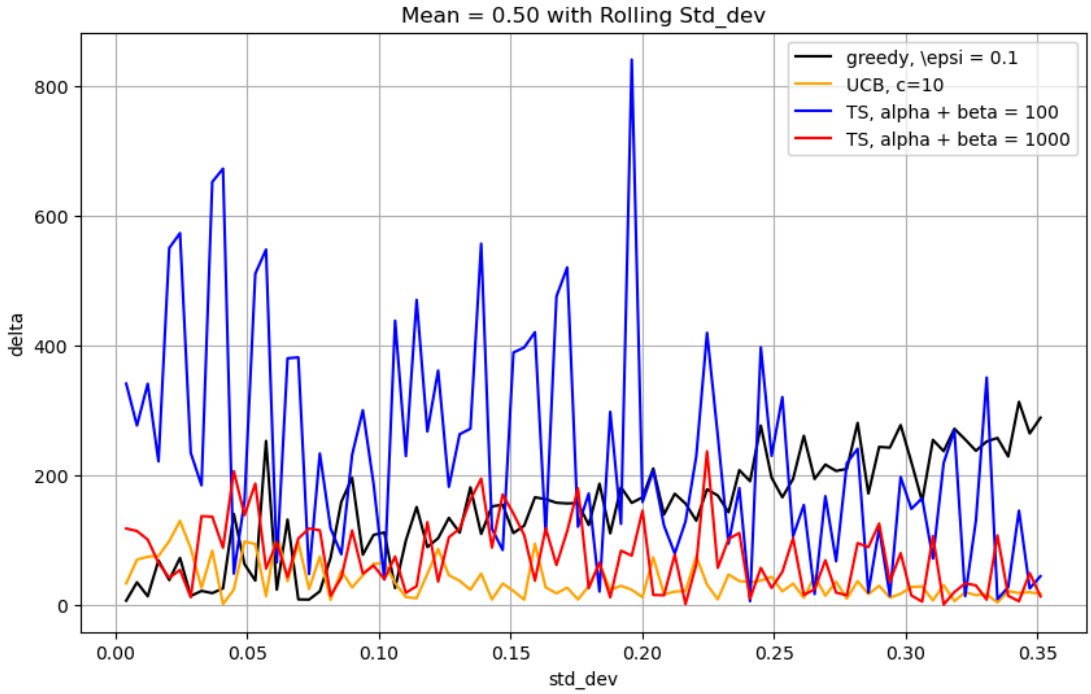
From these six line charts, we find that:

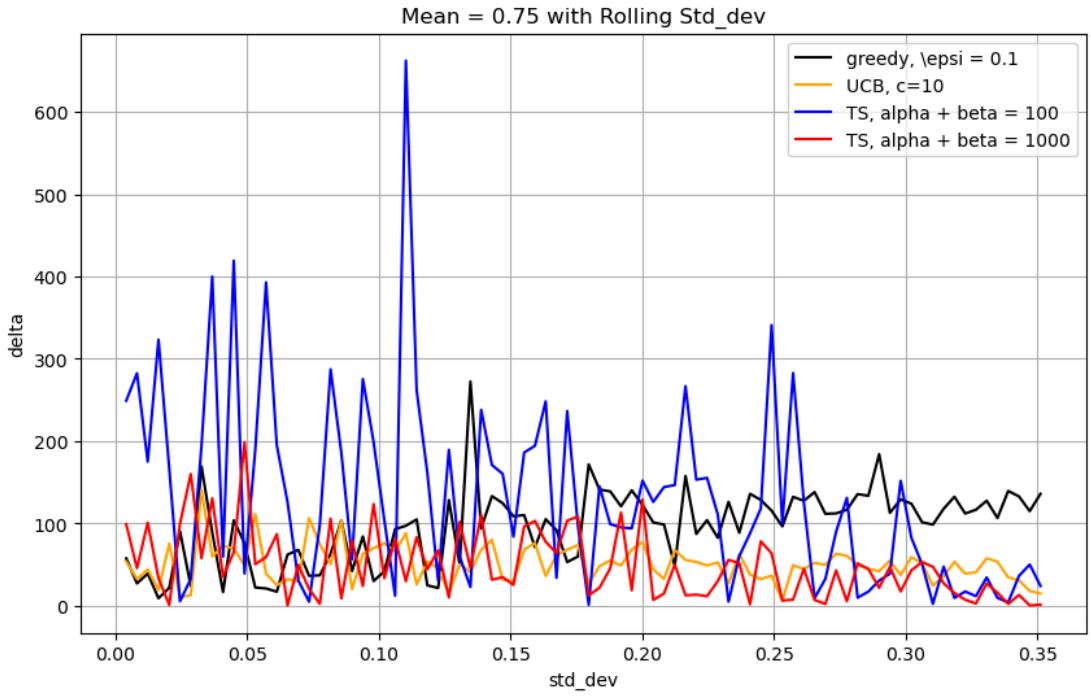
- Compare three line charts with $mean$ of 0.25, 0.5, and 0.75, the average delta tends to decrease as the $mean$ increases by a step of 0.25.
- Compare three line charts with std_dev of $\frac{1}{4}\sqrt{1/6}$, $\frac{2}{4}\sqrt{1/6}$, $\frac{3}{4}\sqrt{1/6}$, the average delta tends to decrease as the std_dev increases by a step of $\frac{1}{4}\sqrt{1/6}$.
- Although we conducted 20 experiments for a single group of θ_j and used the average value of test results to plot, when K was small, the image line still had relatively large oscillations. On the whole, the larger the K is, the smaller the oscillation amplitude of the image.

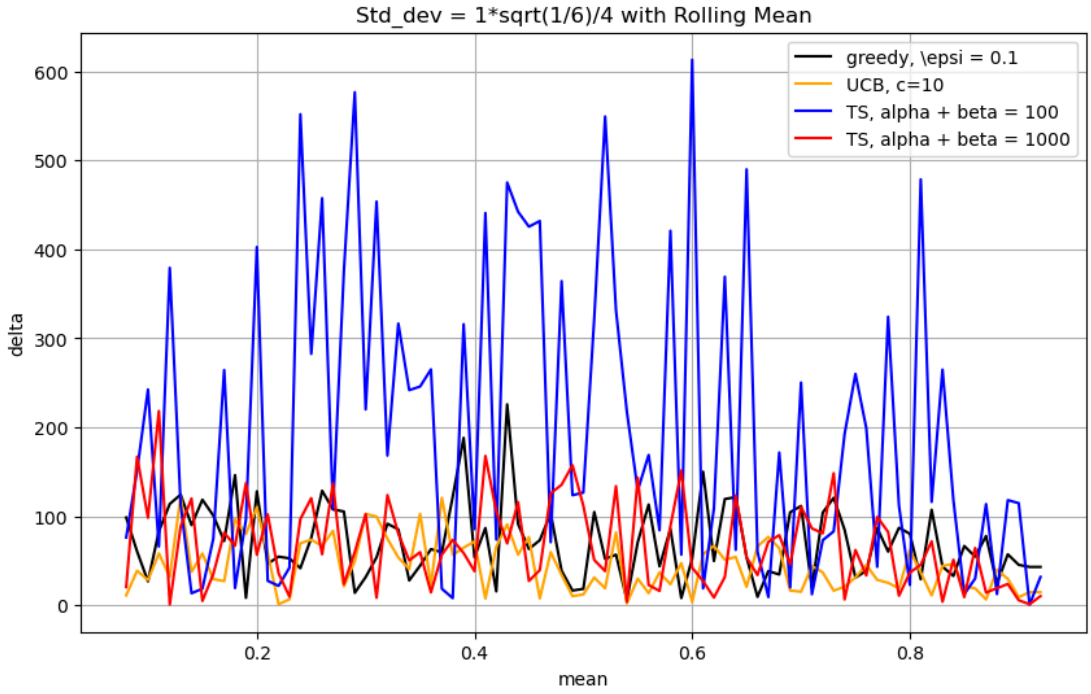
Based on the experimental results, we find that for θ_j, whether randomly selected with specific data features, the result is: the larger K is, the smaller $delta$ is.

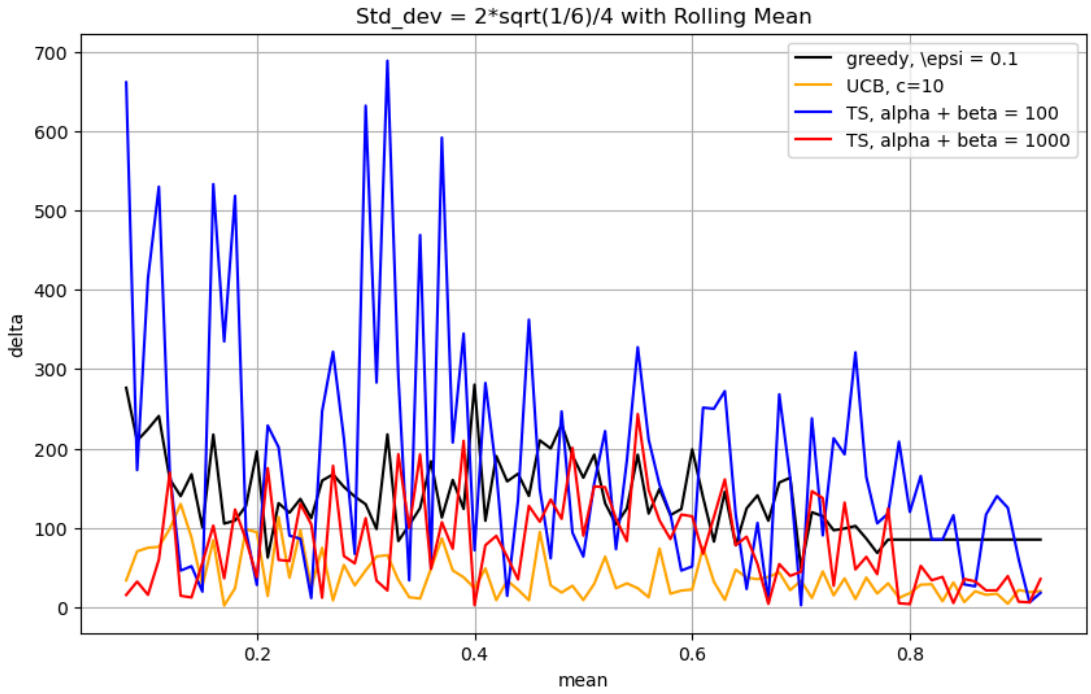
Algorithm Comparation

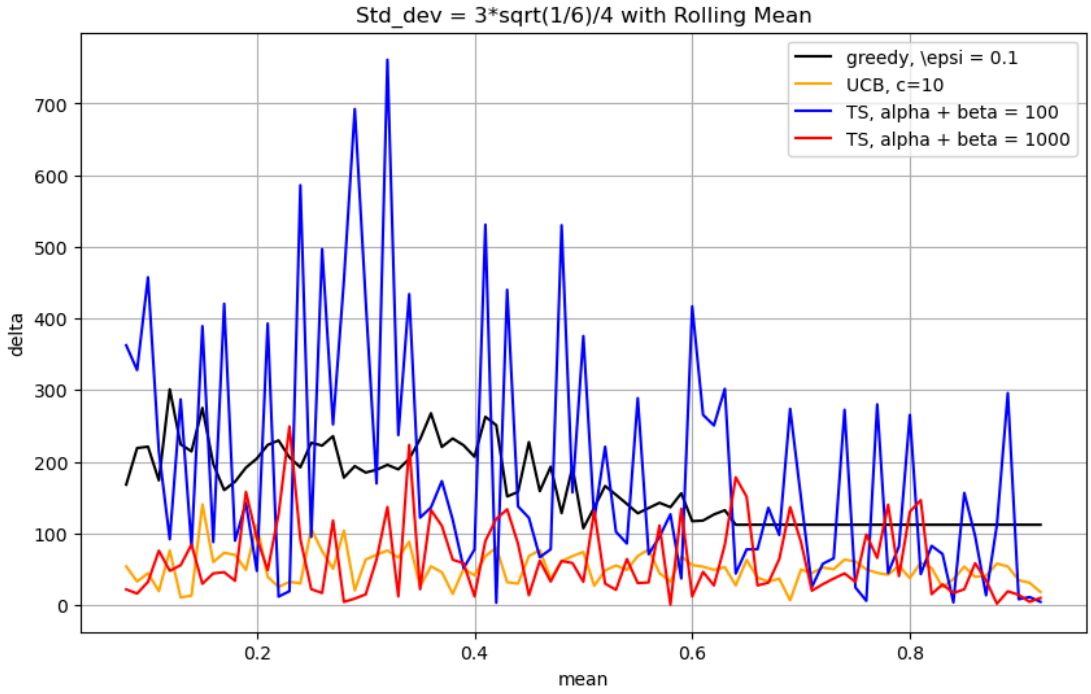





Mean = 0.75 with Rolling Std_dev
greedy, \epsi = 0.1
UCB, c=10
TS, alpha + beta = 100
TS, alpha + beta = 1000
delta
std_dev
0
100
200
300
400
500
600
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35


Std_dev = 1*sqrt(1/6)/4 with Rolling Mean
greedy, \epsi = 0.1
UCB, c=10
TS, alpha + beta = 100
TS, alpha + beta = 1000
delta
mean
0
100
200
300
400
500
600
0.2
0.4
0.6
0.8





According to the chart, we can find that:

- TS Algorithm with 1000 has the best performance in most situation.
- ϵ-greedy with $\epsilon = 0.1$ has the best performance in most situation.
- UCB Algorithm with $c = 1$ has the best performance in most situation.
- UCB Algorithm with $c = 1$ performed best among the three algorithms with a narrow margin

The impact of parameters

ϵ for ϵ-greedy Algorithm

The ϵ parameter in the ε-greedy algorithm is crucial for balancing exploration and exploitation. Here's a concise summary of its impact:

High ε: Encourages exploration by selecting actions randomly, useful for understanding the environment early on.

Low ε: Favors exploitation by choosing the best-known action, leveraging existing knowledge for better performance.

Convergence: A high ε can slow convergence to the optimal strategy, while a low ε might lead to premature exploitation without adequate exploration.

Stability: Very low ε values might make the algorithm overly sensitive to initial conditions, potentially overlooking better strategies.

ε Decay: Gradually reducing ε over time allows for more exploration initially, shifting to exploitation as the algorithm learns more.

Complexity: Complex environments may require a more cautious reduction in ε to ensure thorough exploration.

Multi-Armed Bandit: The choice of ε is critical, with variants like UCB offering theoretically better performance.

Practical Use: ε should be set based on specific problem requirements and may be adjusted according to observed performance.

In essence, the ε parameter is key to the ε-greedy algorithm's effectiveness, influencing how an agent learns and adapts to its environment.

c for UCB Algorithm

The parameter c in the UCB (Upper Confidence Bound) algorithm significantly influences the balance between exploration and exploitation.

Here's a concise summary of its effects: Exploration: A higher c increases exploration, favoring arms with higher uncertainty.

Utilization: A lower c decreases exploration, favoring arms with the highest estimated average reward.

Convergence Speed: A higher c may slow convergence, while a lower c can speed it up, provided sufficient exploration has occurred.

Stability: The c value should match the environment's stability; more dynamic environments might require a higher c.

Performance: Theoretically, increasing c reduces regret, but the optimal c depends on the specific problem.

Problem Complexity: More complex problems might necessitate a larger c to ensure thorough exploration.

C Decay: Adjusting c over time can help shift the focus from exploration to exploitation.

Practical Use: The c value should be set based on experience and may be refined based on performance feedback.

In essence, the c parameter is pivotal in the UCB algorithm, guiding its learning process and overall performance.

α_j and β_j for TS Algorithm

In the Thompson Sampling Algorithm, the priors are Beta distributions with parameters (α_j, β_j), and the algorithm is attempting to use sampling experiment results to modify the Beta distribution to fit the Bernoulli distribution of θ_j.

Denote $K = \alpha_j + \beta_j$ as the number of prior trials.
In our experiment, we investigated the influence of different orders of magnitude of the priori K on the gaps($delta$) between the algorithm outputs and the oracle value.

Here is our research result:

- The larger the order of magnitude of K, the smaller the $delta$.
  - The larger the K is, the closer our Beta distribution is to the Bernoulli distribution of θ_j. So experiments of TS Algorithm with larger K will result in stable, relatively small deltas, like $K = 1000$.

After experiments about random θ_j and specific θ_j, we get the research result, so it is applicable to general situations.

Problem 5

General Understanding

Exploration is to obtain effective information from several trails. Exploitation means use known information to continue make the most beneficial choice.

For ϵ-greedy Algorithm

ϵ can be considered as an exploration-exploitation coefficient: the larger ϵ, the more the algorithm tends to explore; The smaller the ϵ, the more inclined the algorithm is to exploit it.

Find balance in exploration-exploitation interval, so find the best parameter ϵ.

For UCB Algorithm

c can be considered as an exploration-exploitation coefficient: the larger c, the more the algorithm tends to explore; The smaller the c, the more inclined the algorithm is to exploit it. In the UCB algorithm, this coefficient is process independent and is a constant.

Find balance in exploration-exploitation interval, so find the best parameter c .

For TS Algorithm

Exploration is to continuously modify the beta distribution by pulling result to fit the Bernoulli distribution of theta, and exploitation is to continuously pull the arm with the highest mean of Bernoulli distribution according to the highly fitted beta distribution.

There is no clear distinction between the stages of exploration and exploitation. With the increase of (K + # of experiments), the TS algorithm gradually turns from exploration to exploitation.

Coefficient $E = \frac{K + \#\ of\ finished\ pulling}{K + \#\ of\ total\ pullings}$ can be used to represent the degree of exploration-exploitation. The closer E is to 0, the closer it is to exploration; the closer E is to 1, the closer the algorithm is to exploitation.

Problem 6

in this problem, we consider two basic cases

- 1. $\sum_{i=1}^{3} \theta_i = k$,where k is a constant
- 2. $\exists \text{ 0<i<j<3 such that } \theta_i = \theta_j $ namely two or three bandit arms share a distribution of reward

Case 1:

To exploit

$\sum_{i=1}^{3} \theta_i = k$, 0<k<3 .

We modify the original algorithm in two steps.

Step 1

1.initialization:

for t = 1,2,3

$I(t) = t$, $\hat{\theta}_i = r_{I(t)}$

end for

$\hat{\theta}_i = \frac{k\hat{\theta}_i}{\sum_i \theta_i}, 0 \leq i < 3$

2. recursion:

for time slot t,

suppose

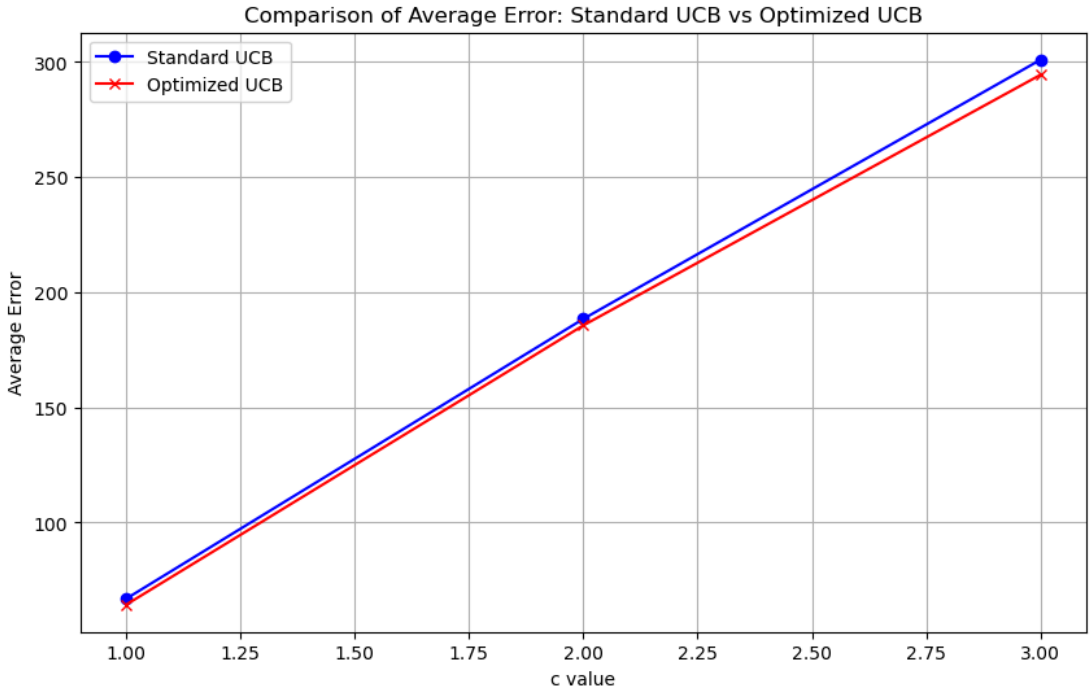
$\hat{\theta}(I(t)) = \hat{\theta}(I(t)) + \delta_{I(t)}$

if t > m,where m is a time slot

$$\begin{cases} \hat{\theta}(I(t)) = \frac{\delta_{I(t)}}{\delta_{I(t)+1}}\hat{\theta}(I(t)), \\ \hat{\theta}(i) = \frac{\delta_{I(t)}}{\delta_{I(t)+1}}\hat{\theta}(i), \\ \hat{\theta}(j) = \frac{\delta_{I(t)}}{\delta_{I(t)+1}}\hat{\theta}(j), \\ \mathrm{i}\ ,\ \mathrm{j} \neq \mathrm{I(t)} \end{cases}$$

It turns out when we add m as a delay time to normalize the overall reward, the optimization is way better.

We conceive that the constraints of m boosts optimization implies that the exploration is severely disturbed with an early assumption of fixed sum of reward.



Case 2:

To exploit

$\exists\ 0<\mathrm{i}<\mathrm{j}<3 \text{ such that } \theta_i = \theta_j.$

It is trivial case when $\theta_1 = \theta_2 = \theta_3$

Because any policy is the potimal

However, when there are two arms with same distribution of reward

From the lesson we learn from the previous optimization, we need to wait for some time slots until the exploration is sufficient and estimation is relieable

And now suppose we know that $\theta_m = \theta_n$

It means that the exploration of these two arms should be viewed as an integrity

So in UCB algorithm, we can optimize the recusion equation as :

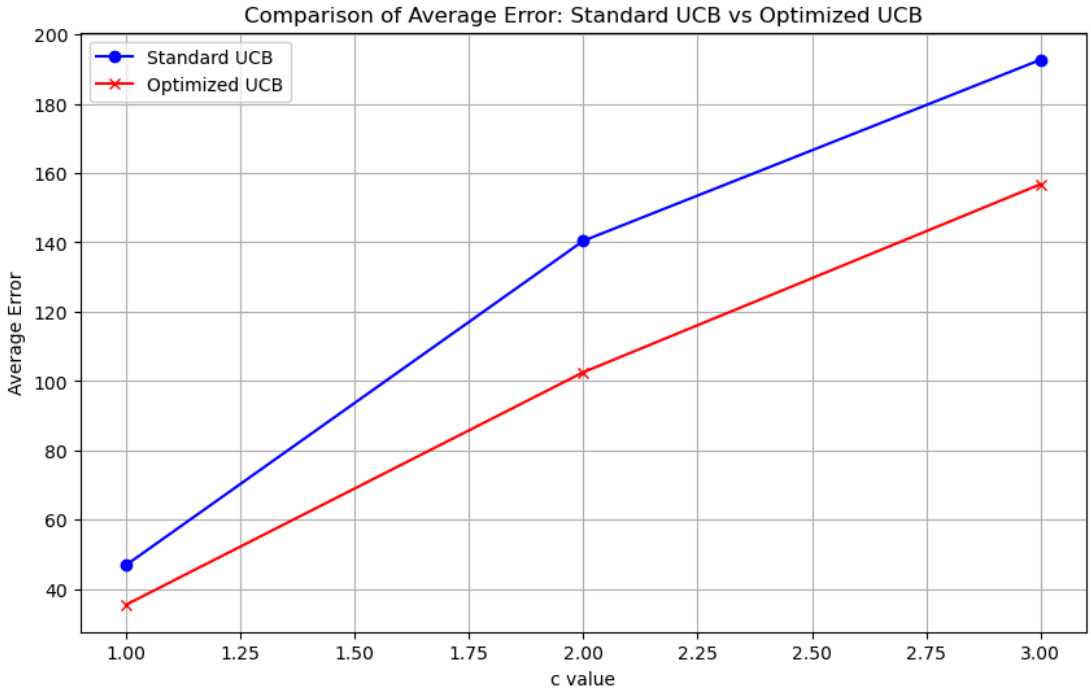
$$I(t) = \arg\max_{j=1,2,3}(\hat{\theta}(j) + c\sqrt{(\frac{2log(t)}{count(j)}}), max(\hat{\theta}(m), \hat{\theta}(n)) + c\sqrt{(\frac{2log(t)}{count(m+n)}}))$$

When t > m and $\theta_m - \theta_n < \delta$

After the unification, whenever arm m or n is chosen, each time the estimation

$$\theta_m = \theta_n, count_m += 1, count_n += 1$$

Becasue only after m time slots can we have some concepts of whcich two arms have the same reward distribtion

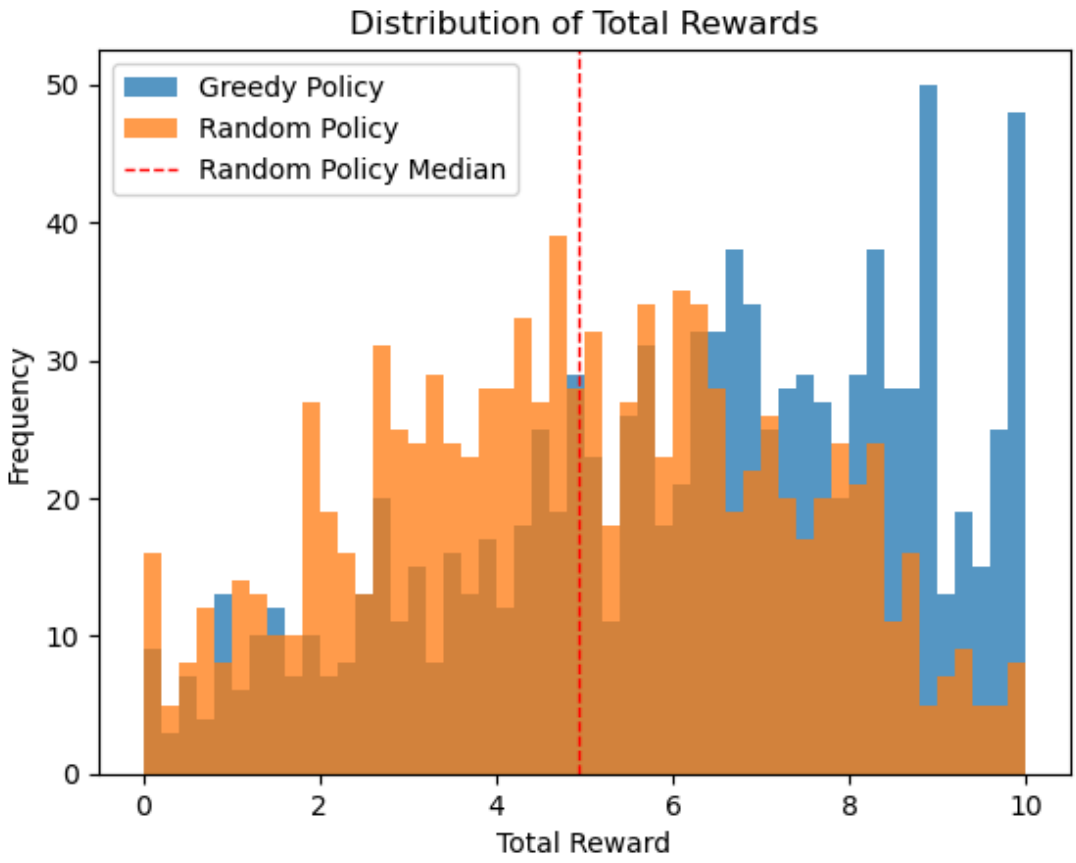


In [ ]:

Part II: Bayesian Bandit Algorithms

Problem 1

Here is the simulation result, compared with random simulation.



Problem 2

2. this policy might not be optimal let's suppose this scenerio:

define $R_j(\alpha, \beta)$: the overall reward of choosing arm j under $Beta(\alpha, \beta)$
$\alpha_1 = 1, \beta_1 = 1$
$\alpha_2 = 10^{100} + 1, \beta_2 = 10^{100} - 1$
according to the choose larger expectaion policy,
we would choose arm 2 in the first step

$$R_2(10^{100}+1, 10^{100}-1) = \frac{R_2(10^{100}+1, 10^{100})}{2} + \frac{\gamma + R_2(10^{100}+2, 10^{100}-1)}{2}$$
$$= \frac{R_2(10^{100}+1, 10^{100})}{2} + \frac{R_2(10^{100}+1, 10^{100}+1) + R_2(10^{100}+2, 10^{100}) + \gamma^2 + 2\gamma}{4}$$

bacause the expectaion of $\theta_1 < \theta_2$
however, if we simply choose arm 1 in the first step
and choose arm 2 whenever a failure occur

$$R_1(1,1) = \frac{R_2(10^{100}+1, 10^{100})}{2} + \frac{\gamma + R_1(2,1)}{2}$$
$$= \frac{R_2(10^{100}+1, 10^{100}-1)}{2} + \frac{\frac{1}{2}R_2(10^{100}+1, 10^{100-1}) + R_1(3,1) + \gamma^2 + \frac{3}{4}\gamma}{3}$$

suppose on the contrary that
$R_2(10^{100}+1, 10^{100}-1) > R_1(1,1)$
$\because R_1(3,1) > R_2(10^{100}+2, 10^{100})$
$\therefore \frac{\gamma^2 + 2\gamma}{4} > \frac{\gamma^2 + \frac{3}{4}\gamma}{3}$
$\therefore \gamma < 0$
, which is contradictoy
indicating chooosing the larger expectation
is not the optimal policy

Problem 3:

according to the conjunacy of Beta distribution after a bonoulli trial
the overall reward of pulling arm 1 :$R_1(\alpha_1, \beta_1)$
according to LOTP
$=$ reward in case of winning $*$ P_{winning} $+$ reward in case of losing $*$ P_{losing}
Case 1: the first pulling wins at arm 1
$R^1_{\text{winning}} = 1 + \gamma(max(R^2_1, R^2_2))$
$P_{\text{winning}} = \dfrac{\alpha_1}{\alpha_1 + \beta_1}$
Case 2: the first pulling loses at arm 1
$R^1_{\text{winning}} = \gamma(max(R^2_1, R^2_2))$
$P_{\text{winning}} = \dfrac{\beta_1}{\alpha_1 + \beta_1}$
Case 3: the first pulling wins at arm 2
$R^1_{\text{winning}} = 1 + \gamma(max(R^2_1, R^2_2))$
$P_{\text{winning}} = \dfrac{\alpha_2}{\alpha_2 + \beta_2}$
Case 4: the first pulling loses at arm 2
$R^1_{\text{winning}} = \gamma(max(R^2_1, R^2_2))$
$P_{\text{winning}} = \dfrac{\beta_2}{\alpha_2 + \beta_2}$

Problem 4:

Define $R_1(\alpha_1^n, \beta_1^n)$: the total largest reward of pulling arm 1 after n pullin

$R_2(\alpha_2^n, \beta_2^n)$: the total largest reward of pulling arm 2 after n pullin

let $m = \alpha_1 + \beta_1 + \alpha_2 + \beta_2$

where $\alpha_1^n + \beta_1^n + \alpha_2^n + \beta_2^n = \alpha_1 + \beta_1 + \alpha_2 + \beta_2 + n = m + n$

since $\gamma < 1$

$$\lim_{n\to\infty} R_1(\alpha_1^n, \beta_1^n) = \lim_{n\to\infty} R_2(\alpha_2^n, \beta_2^n) = 0$$

$$\therefore R(\alpha_1^n, \beta_1^n, \alpha_2^n, \beta_2^n) = 0$$

$$\therefore R_1(\alpha_1^{n-1}, \beta_1^{n-1}) = \frac{\alpha_1^{n-1}}{\alpha_1^{n-1} + \beta_1^{n-1}}$$

$$\therefore R_2(\alpha_2^{n-1}, \beta_2^{n-1}) = \frac{\alpha_2^{n-1}}{\alpha_2^{n-1} + \beta_2^{n-1}}$$

so we can set n as the starting point of this equation

Case 1: the n pulling wins at arm 1

$$\because R_1(\alpha_1^n, \beta_1^n) = R_2(\alpha_2^n, \beta_2^n)$$

$$\therefore R_1(\alpha_1^{n-1} + 1, \beta_1^{n-1}) = R_2(\alpha_2^{n-1}, \beta_2^{n-1})$$

$$\because R_1(\alpha_1^{n-1} + 1, \beta_1^{n-1}) = \frac{\alpha_1^{n-1} + 1}{\alpha_1^{n-1} + 1 + \beta_1^{n-1}}$$

$$\therefore \frac{\alpha_1^{n-1} + 1}{\alpha_1^{n-1} + 1 + \beta_1^{n-1}} = \frac{\alpha_2^{n-1}}{\alpha_2^{n-1} + \beta_2^{n-1}}$$

$$\therefore \frac{\alpha_1^n}{\alpha_1^n + \beta_1^n} = \frac{\alpha_2^n}{\alpha_2^n + \beta_2^n};$$

let

$$\frac{\alpha_1^n}{\alpha_2^n} = \frac{\beta_1^n}{\beta_2^n} = k$$

where k is a integer

in case of n-2

$$R_1(\alpha_1^{n-2}, \beta_1^{n-2}) = \frac{\alpha_1^{n-1}}{\alpha_1^{n-1} + \beta_1^{n-1}}$$

$$R_2(\alpha_2^{n-2}, \beta_2^{n-2}) = \frac{\alpha_2^{n-1}}{\alpha_2^{n-1} + \beta_2^{n-1}}$$

Problem 5

Appendix 1: How to obtain θ_j with specific data features

Since we want to figure out the relationship between θ_j array with exact data features like mean and standard deviation and different parameters, we have to design an algorithm to get a θ_j array through specific mean and standard deviation.

Since except θ_j having to be between 0 and 1, there is no other limitation on value of θ_j, we can obtain θ_1 through sample from $Unif(0,1)$.

Now with known θ_1, $m = mean$, $\sigma = std\_dev$, here is the equation set:

$$\begin{cases} (\theta_1+\theta_2+\theta_3)/3 = m \\ [(\theta_1-m)^2+(\theta_2-m)^2+(\theta_3-m)^2]/3 = \sigma^2 \\ 0<\theta_1,\theta_2,\theta_3<1 \end{cases} \tag{62}$$

then, we can get

$$2\theta_3 + (2\theta_1 - 6m)\theta_3 + 2\theta_1^2 + 6m^2 - 3\sigma^2 - 6\theta_1 m = 0$$

According to symmetry, θ_2 and θ_3 are the two roots of the above equation.

Hence, by solving the equation, we can get θ_2 and θ_3, and with sampled θ_1, we obtain a θ_j array with specific mean and standard deviation.

Appendix 2: Python codes and figures for corresponding problems

Part I: Classical Bandit Algorithms

Algorithm Implementation

Theta Generate: randomly and specificaly

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

class ThetaSequence:
    def __init__(self, seq: np.ndarray, n_elements: int):
        self.seq = seq
        self.n_elem = n_elements
        self.mean = np.mean(seq)
        self.var = np.var(seq)
        self.max_theta = np.max(seq)
        self.std_dev = np.std(seq)

## randomly generate
def generate_theta_sequence_set(n_elem, n_sets):
    theta_seq_set = []
    for _ in range(n_sets):
        theta_seq = ThetaSequence(np.random.uniform(0, 1, size=n_elem), n
        theta_seq_set.append(theta_seq)
    return np.array(theta_seq_set)

## 用theta组的数值特征，判断不同参数对不同特征的适应性
## 由数学关系得到具有特征 theta1，均值，标准差的thetaSeq
## 不需要输入数字，因为只能生成三个
## 部分注释内容为测试语句
def specific_theta_generate(mean, std_dev, theta1=np.random.uniform(0, 1)
    coff_a = 2
```

```
    coff_b = 2 * theta1 - 6 * mean
    coff_c = 6 * (mean ** 2) - 3 * (std_dev ** 2) + (2 * theta1 ** 2) - 6
    count = 0
    while count < 1000:
        count += 1
        delta = coff_b ** 2 - 4 * coff_a * coff_c
        # print("delta problem")
        if delta >= 0:
            theta2 = (-coff_b + np.sqrt(delta)) / (2 * coff_a)
            theta3 = (-coff_b - np.sqrt(delta)) / (2 * coff_a)
            if 0 <= theta2 <= 1 and 0 <= theta3 <= 1:
                # print([theta1, theta2, theta3])
                return ThetaSequence(np.array([theta1, theta2, theta3]),3
            # print("0,1 problem: {}".format([theta1, theta2, theta3]))

        theta1 = np.random.uniform(0, 1)
        coff_b = 2 * theta1 - 6 * mean
        coff_c = 6 * (mean ** 2) - 3 * (std_dev ** 2) + (2 * theta1 ** 2)
    else:
    # 如果无法找到合法的 theta1、theta2 和 theta3，则返回 None
        # print("Mean: {}, Std_Dev: {}".format(mean, std_dev))
        return None

## 静均值，动标准差
def theta_generate_static_mean_rolling_std_dev(mean, start_std_dev, end_s
    theta_set = []
    for rolling_std_dev in np.arange(start_std_dev, end_std_dev, span):
        theta_group = specific_theta_generate(mean, rolling_std_dev)
        if not theta_group == None:
            theta_set.append(theta_group)
    return np.array(theta_set)

## 静标准差，动均值
def theta_generate_static_std_dev_rolling_mean(std_dev, start_mean, end_m
    theta_set = []
    for rolling_mean in np.arange(start_mean, end_mean, span):
        theta_group = specific_theta_generate(rolling_mean, std_dev)
        if not theta_group == None:
            theta_set.append(theta_group)
    return np.array(theta_set)
```

ϵ-greedy Algorithm & UCB Algorithm

The implementations of these two algorithms are in the following specific problems, including their corresponding chart plot codes.

TS Algorithm

In [ ]:

```
def beta_prio_generate(theta_seq_set: np.ndarray, min_n_trails: int):
  para_set_min = [[],[],[]]
  for theta_seq in theta_seq_set:
    ## 以下针对一组thetaSeq
    for k in range(3):
      para = []
      for theta in theta_seq.seq:
        res = np.random.binomial(1, theta, (10 ** k)*min_n_trails)
        ## +1 保证参数不为零
        para.append([np.sum(res)+1, (10 ** k)*min_n_trails - np.sum(res)+
```

```
      para_set_min[k].append(para)
  return para_set_min


## Just get reward
def Thompson_Sampling_once(n_arms, n_trials, beta_prio):
  sampled_theta = np.zeros(n_arms)
  reward = np.zeros(n_arms, dtype=int)
  for _ in range(n_trials):
    for i in range(n_arms):
      sampled_theta[i] = np.random.beta(beta_prio[i][0],beta_prio[i][1])
    max_arm = np.argmax(sampled_theta)
    cor_reward = np.random.binomial(1, sampled_theta[max_arm])
    beta_prio[max_arm][0] += cor_reward
    beta_prio[max_arm][1] +=  (1 - cor_reward)
    reward[max_arm] += cor_reward

  for i in range(n_arms):
    ## final theta update
    sampled_theta[i] = np.random.beta(beta_prio[i][0],beta_prio[i][1])
  return np.sum(reward)


def packaged_Thompson_Sampling(theta_set, n_arms, n_experiments, n_slots,
  final_reward_para_exp = [[], [], []]
  for i in range(len(theta_set)):
    ## 对第i组theta做n_experiments次实验，取平均
    reward = [[], [], []]

    for _ in range(n_experiments):
      for j in range(len(para)):
        reward[j].append(Thompson_Sampling_once(n_arms, n_slots, para[j][

    for k in range(len(para)):
      final_reward_para_exp[k].append(np.mean(np.array(reward[k])))

  delta = [[], [], []]
  mean, std_dev = [], []
  for i in range(len(theta_set)):
    for j in range(len(delta)):
      delta[j].append(abs(final_reward_para_exp[j][i] - theta_set[i].max_
    mean.append(theta_set[i].mean)
    std_dev.append(theta_set[i].std_dev)
  return delta, mean, std_dev
```

Some of Chart Plot

In [ ]:

```
## 注意！有更新!
def plot_n_lines(n_line, x, y_axis, title, x_label, y_labels, colors):
  fig, ax = plt.subplots(figsize=(10, 6))

  for i in range(n_line):
    if len(y_axis[i]) < len(x):
      for _ in range(len(y_axis[i]),len(x)+1):
        y_axis[i].append(y_axis[i][len(y_axis[i])-1])
    ax.plot(x, y_axis[i][:len(x)], color=colors[i], label=y_labels[i])

  ax.set_title(title)
  ax.set_xlabel(x_label)
```

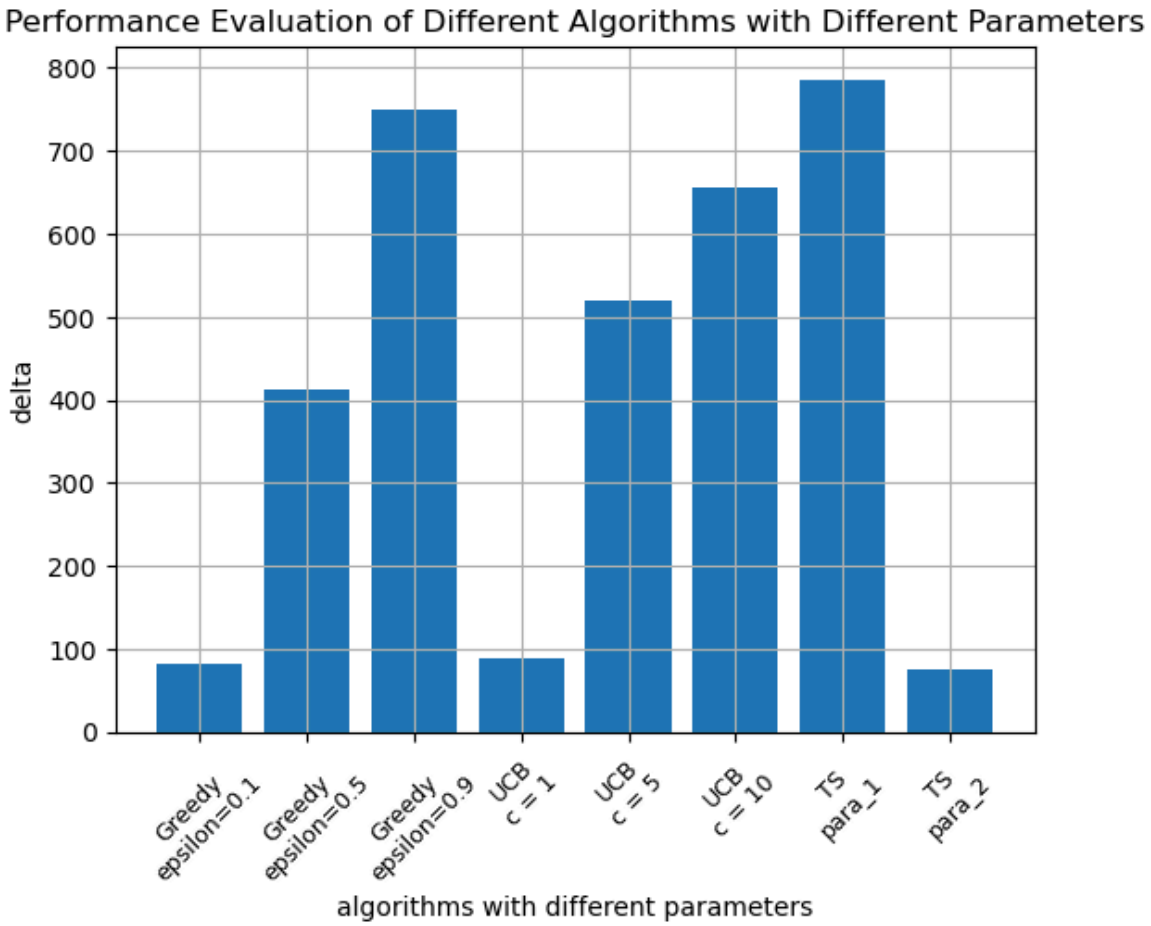
```
  ax.set_ylabel('delta')

  ax.legend()

  plt.grid(True)
  plt.show()

  return


def plot_bar_chart(x_axis, y_axis, title, x_labels, y_label = 'delta'):
  plt.bar(x_axis,y_axis)

  plt.xticks(x_axis, x_labels,rotation=45, fontsize=9)
  plt.xlabel('algorithms with different parameters')
  plt.ylabel(y_label)
  plt.title(title)
  plt.grid(True)
  plt.show()
```

Problem 2,3

In [ ]:

```
n_arms = 3
n_experi = 200
n_slot = 5000
theta = ThetaSequence(np.array([0.7,0.5,0.4]),n_arms)
delta = [0,0,0,0,0,0]

## greedy
## UCB
## 结果放入delta
theta = np.array([0.7, 0.5, 0.4])  # 固定的theta值
num_experiments = 200  # 实验次数
num_pulls_per_experiment = 5000  # 每次实验的摇臂次数

# 初始化UCB和epsilon-greedy算法的误差列表
ucb_errors = []
epsilon_greedy_errors = []

# 设置UCB的c值和epsilon-greedy的epsilon值
ucb_cs = [1, 5, 9]
epsilons = [0.1, 0.5, 0.9]
# 初始化存储奖励和oracle value的数组
rewards = np.zeros((len(ucb_cs), num_experiments))
oracle_values = np.zeros((len(ucb_cs), num_experiments))
for epsilon in epsilons:
    epsilon_greedy_total_rewards = np.zeros(num_experiments)
    for experiment in range(num_experiments):
        # 重置计数器和总奖励
        counts = np.zeros_like(theta)
        total_rewards = 0
        for _ in range(num_pulls_per_experiment):
            arm = np.random.randint(len(theta)) if np.random.rand() < eps
            counts[arm] += 1
            reward = np.random.binomial(1, theta[arm])
            total_rewards += reward
        epsilon_greedy_total_rewards[experiment] = total_rewards

    # 计算epsilon-greedy算法的平均误差
```

```
    epsilon_greedy_average_error = np.mean(epsilon_greedy_total_rewards -
    epsilon_greedy_errors.append((epsilon, epsilon_greedy_average_error))

for epsilon, error in epsilon_greedy_errors:
    if epsilon == 0.1:
        delta[0] = np.abs(error)
    elif epsilon == 0.5:
        delta[1] = np.abs(error)
    elif epsilon == 0.9:
        delta[2] = np.abs(error)
# 模拟UCB算法
for i, c in enumerate(ucb_cs):
    for experiment in range(num_experiments):
        total_reward = 0
        theta_estimates = np.zeros_like(theta)
        counts = np.ones_like(theta)  # 初始化选择次数
        for trial in range(num_pulls_per_experiment):
            if trial < 3:  # 前三次随机选择
                arm = np.random.randint(len(theta))
            else:
                # 计算UCB值
                ucb_values = theta_estimates + c * np.sqrt((2 * np.log(tr
                arm = np.argmax(ucb_values)

            reward = 1 if np.random.rand() < theta[arm] else 0
            total_reward += reward
            counts[arm] += 1
            theta_estimates[arm] += (reward - theta_estimates[arm]) / cou

        rewards[i, experiment] = total_reward
        oracle_values[i, experiment] = num_pulls_per_experiment * np.max(

# 计算平均误差
average_rewards = np.mean(rewards, axis=1)
average_oracle_values = np.mean(oracle_values, axis=1)
average_errors = np.abs(average_rewards - average_oracle_values)
delta[3]=average_errors[0]
delta[4]=average_errors[1]
delta[5]=average_errors[2]
# 打印每个c值对应的平均误差

# 模拟epsilon-greedy算法


 # 也是为了防出不了图，而用来占位置的数
theta = ThetaSequence(np.array([0.7,0.5,0.4]),n_arms)
beta_prio_setted = [[[1,1],[1,1],[1,1]],[[601,401],[401,601],[2,3]]]
delta.append(abs(Thompson_Sampling_once(n_arms, n_slot, beta_prio_setted[
delta.append(abs(Thompson_Sampling_once(n_arms, n_slot, beta_prio_setted[


x = np.arange(8)
x_labels = ['Greedy\nepsilon=0.1', 'Greedy\nepsilon=0.5', 'Greedy\nepsilo
title = 'Performance Evaluation of Different Algorithms with Different Pa
plot_bar_chart(x,delta,title,x_labels)
print("Delta in order: {}".format(delta))
```




```
Delta in order: [82.155, 413.525, 749.36, 88.61000000000013, 519.094999999
9998, 654.7849999999999, 786.0, 75.0]
```

Problem 4: Random θ_j array

Theta Generate

In [ ]:
```
n_elem = 3
n_set = 50
random_theta_set = generate_theta_sequence_set(n_elem,n_set)
```

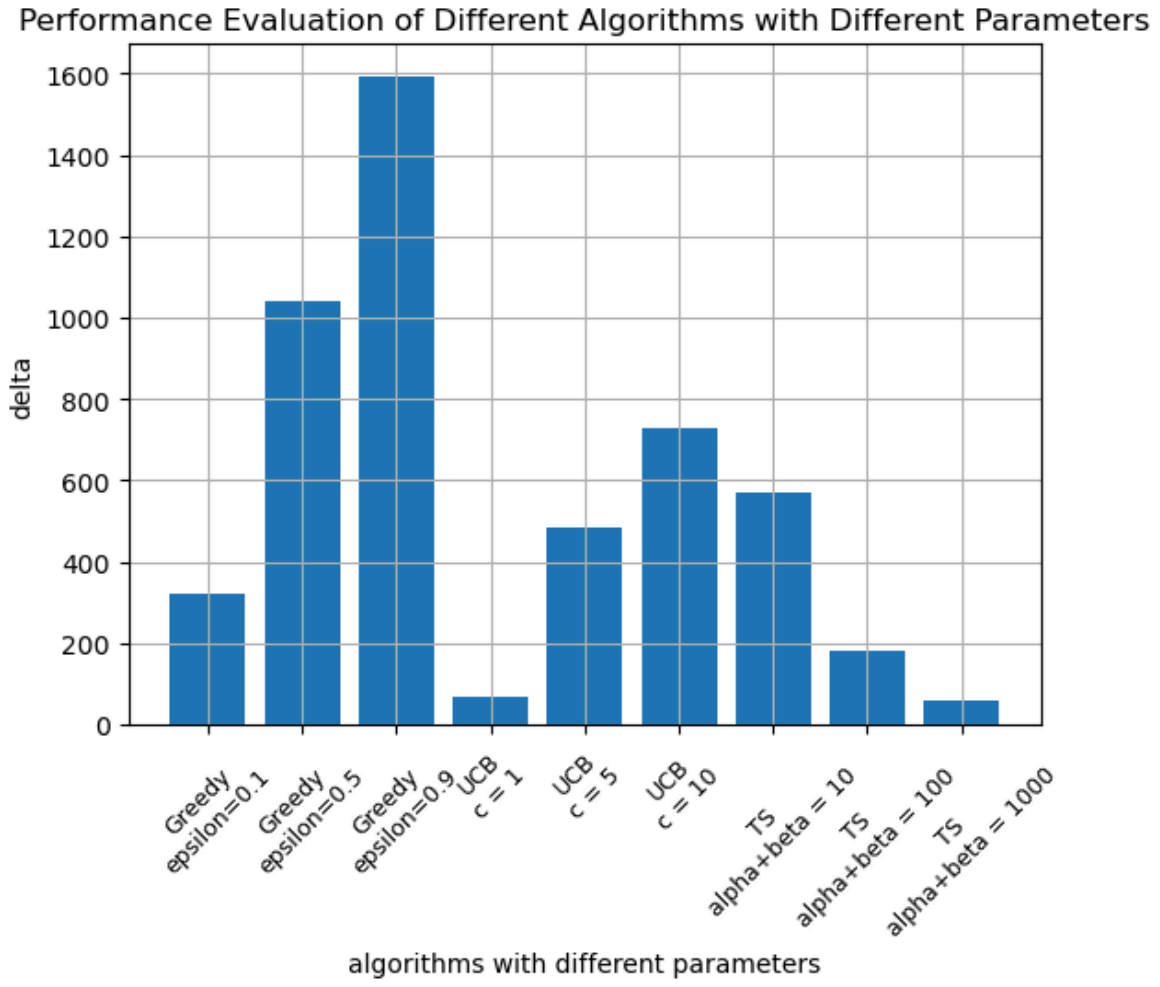
TS Algorithm

In [ ]:
```
n_arms = 3
n_experi = 20
n_slot = 5000

n_line = 3
labels = ['alpha + beta = 10','alpha + beta = 100','alpha + beta = 1000']
line_color = ['green','blue','red']

para_exp = beta_prio_generate(random_theta_set, min_n_trails=10)
y_delta, _, _ = packaged_Thompson_Sampling(random_theta_set, n_arms, n_ex
average_y_delta = []
for set in y_delta:
  average_y_delta.append(np.mean(set))
# print(average_y_delta)
```

ϵ-greedy Algorithm, UCB Algorithm, and Chart Plot

In [ ]:
```
y = [0, 0, 0, 0, 0, 0]
num_experiments = 200  # 实验次数
num_pulls_per_experiment = 5000  # 每次实验的摇臂次数

# 初始化UCB和epsilon-greedy算法的误差列表
ucb_errors = []
epsilon_greedy_errors = []

# 设置UCB的c值和epsilon-greedy的epsilon值
ucb_cs = [1, 5, 9]
epsilons = [0.1, 0.5, 0.9]
# 初始化存储奖励和oracle value的数组
rewards = np.zeros((len(ucb_cs), num_experiments))
oracle_values = np.zeros((len(ucb_cs), num_experiments))
for epsilon in epsilons:
    epsilon_greedy_total_rewards = np.zeros(num_experiments)
    for experiment in range(num_experiments):
        theta = np.random.uniform(0, 1, 3)
        counts = np.zeros_like(theta)
        total_rewards = 0
        for _ in range(num_pulls_per_experiment):
            arm = np.random.randint(len(theta)) if np.random.rand() < eps
            counts[arm] += 1
            reward = np.random.binomial(1, theta[arm])
            total_rewards += reward
        epsilon_greedy_total_rewards[experiment] = total_rewards

    # 计算epsilon-greedy算法的平均误差
    epsilon_greedy_average_error = np.mean(epsilon_greedy_total_rewards -
    epsilon_greedy_errors.append((epsilon, epsilon_greedy_average_error))

for epsilon, error in epsilon_greedy_errors:
    if epsilon == 0.1:
        y[0] = np.abs(error)
    elif epsilon == 0.5:
        y[1] = np.abs(error)
    elif epsilon == 0.9:
        y[2] = np.abs(error)
# 模拟UCB算法
for i, c in enumerate(ucb_cs):
    for experiment in range(num_experiments):
        theta = np.random.uniform(0, 1, 3)
        total_reward = 0
        theta_estimates = np.zeros_like(theta)
        counts = np.ones_like(theta)  # 初始化选择次数
        for trial in range(num_pulls_per_experiment):
            if trial < 3:  # 前三次随机选择
                arm = np.random.randint(len(theta))
            else:
                # 计算UCB值
                ucb_values = theta_estimates + c * np.sqrt((2 * np.log(tr
                arm = np.argmax(ucb_values)

            reward = 1 if np.random.rand() < theta[arm] else 0
            total_reward += reward
            counts[arm] += 1
            theta_estimates[arm] += (reward - theta_estimates[arm]) / cou

        rewards[i, experiment] = total_reward
```

```
        oracle_values[i, experiment] = num_pulls_per_experiment * np.max(

# 计算平均误差
average_rewards = np.mean(rewards, axis=1)
average_oracle_values = np.mean(oracle_values, axis=1)
average_errors = np.abs(average_rewards - average_oracle_values)
y[3]=average_errors[0]
y[4]=average_errors[1]
y[5]=average_errors[2]

y.extend(average_y_delta)

x = range(9)
# ！！！注意 ！！！
title = 'Performance Evaluation of Different Algorithms with Different Pa
x_labels = ['Greedy\nepsilon=0.1', 'Greedy\nepsilon=0.5', 'Greedy\nepsilo

plot_bar_chart(x,y,title,x_labels)
print("Delta in order: {}".format(y))
```




```
Delta in order: [319.20600491207665, 1038.6015153933872, 1591.995161955955
2, 67.84205839151946, 485.9048645726152, 727.778160322468, 570.06966845654
19, 181.25160398542076, 57.59999900190664]
```

Problem 4: θ_j array with specific mean and standard deviation

Theta Generate

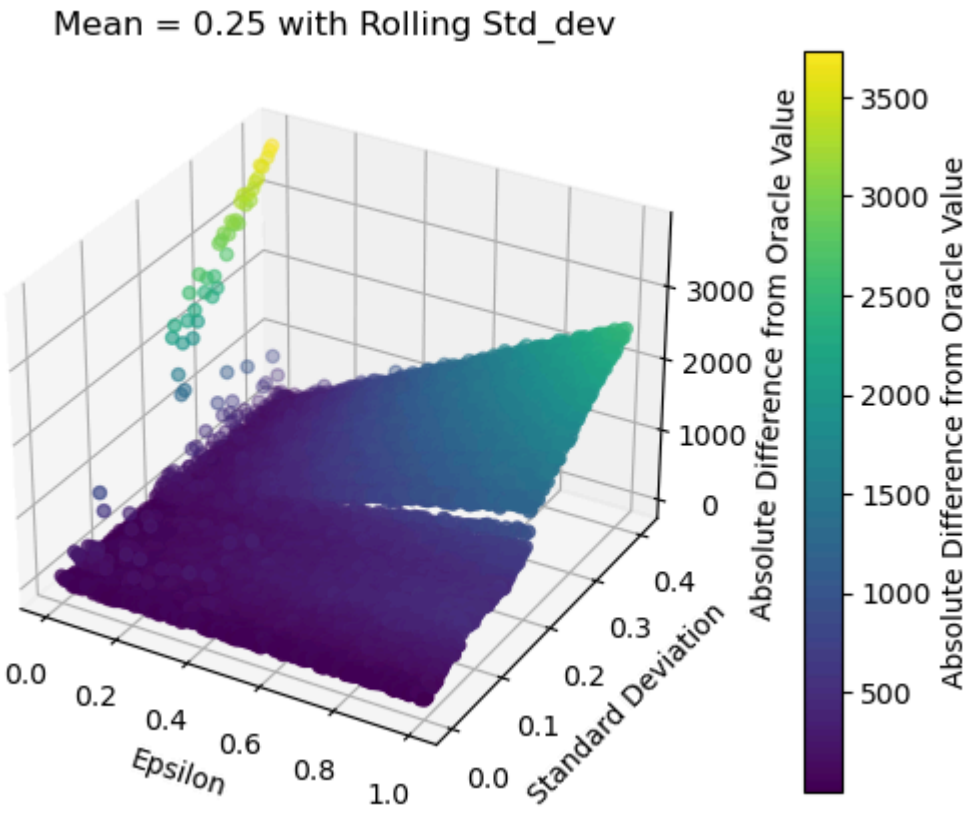
In [ ]:
```
max_mean = 1
max_std_dev = np.sqrt(1/6)

static_mean_rolling_std_dev_set = []
for static_mean in np.arange(max_mean/4, max_mean-0.01, max_mean/4):
  static_mean_rolling_std_dev_set.append(theta_generate_static_mean_rolli


static_std_dev_rolling_mean_set = []
for static_std_dev in np.arange(np.sqrt(1/6)/4, np.sqrt(1/6) - 0.00001, n
  static_std_dev_rolling_mean_set.append(theta_generate_static_std_dev_ro
```

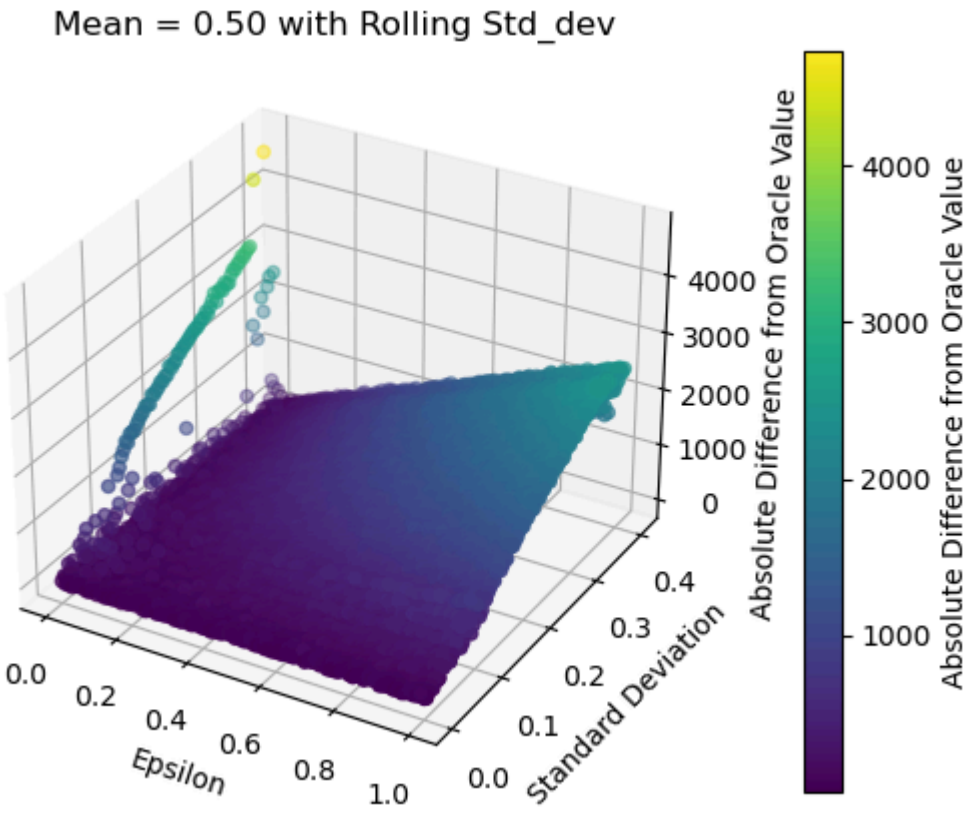
ϵ-greedy Algorithm

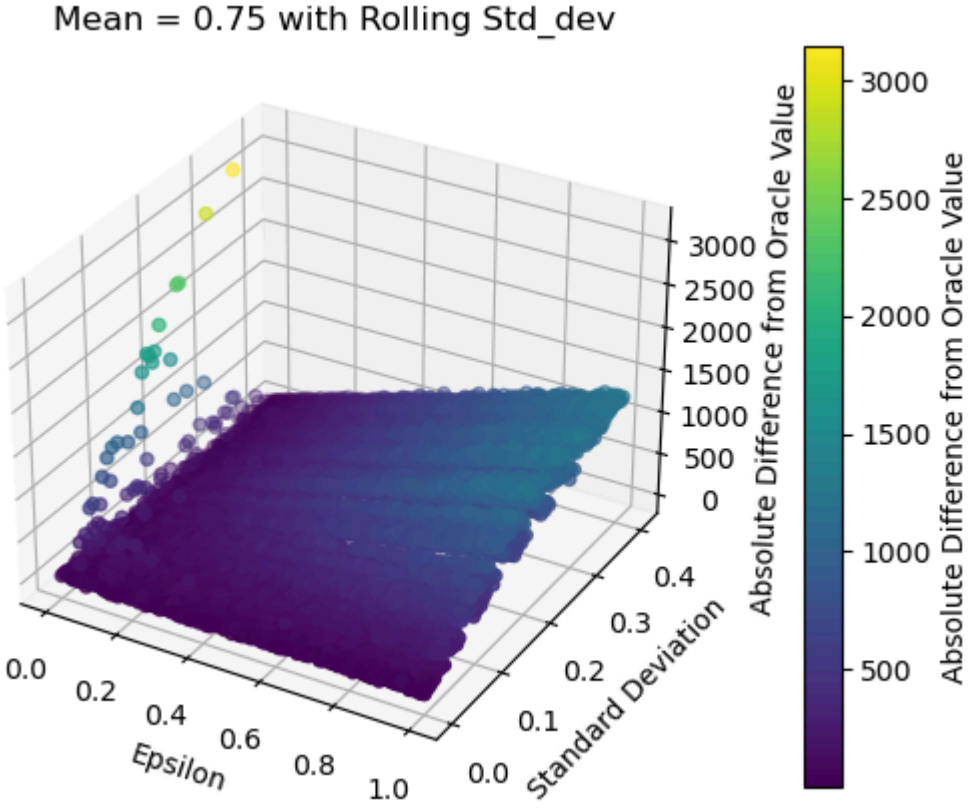
In [ ]:
```
##### 两个画图中的std_dev是什么
y_delta_greedy_static_mean = []
y_delta_greedy_static_std_dev = []

num_epsilons = 101
epsilon_values = np.linspace(0,1, num_epsilons)
# num_std_devs = 41
# std_devs = np.linspace(0, np.sqrt(1/6), num_std_devs)
num_trials = 5000
# rewards = np.zeros((num_epsilons, num_std_devs))

mean_word = ['Mean = 0.25 with Rolling Std_dev', 'Mean = 0.50 with Rollin
std_dev_word = ['Std_dev = 1*sqrt(1/6)/4 with Rolling Mean','Std_dev = 2*


mean=[0.25,0.5,0.75]
for z in range(len(mean)):
    theta = static_mean_rolling_std_dev_set[z]
    num_std_devs = len(theta)
    rewards = np.zeros((num_epsilons, num_std_devs))
    oracle_values = np.zeros((num_epsilons, num_std_devs))
    std_devs = np.linspace(0, np.sqrt(1/6), num_std_devs)

    for i, epsilon in enumerate(epsilon_values):
        for j in range(len(theta)):
            count=[0,0,0]
            theta_estimates=[0,0,0]
            total_reward = 0
            reward=0
            arm=0
            for _ in range(num_trials):
                if np.random.rand() < epsilon:
                    arm = np.random.choice([0, 1, 2])
                else:
                    arm = np.argmax(theta_estimates)
                if np.random.rand() < theta[j].seq[arm]:
                    reward = 1
                else:
                    reward = 0
                count[arm] += 1
                theta_estimates[arm] += (1/count[arm]) * (reward - theta_
                total_reward += reward
            oracle_value = num_trials * theta[j].max_theta
```

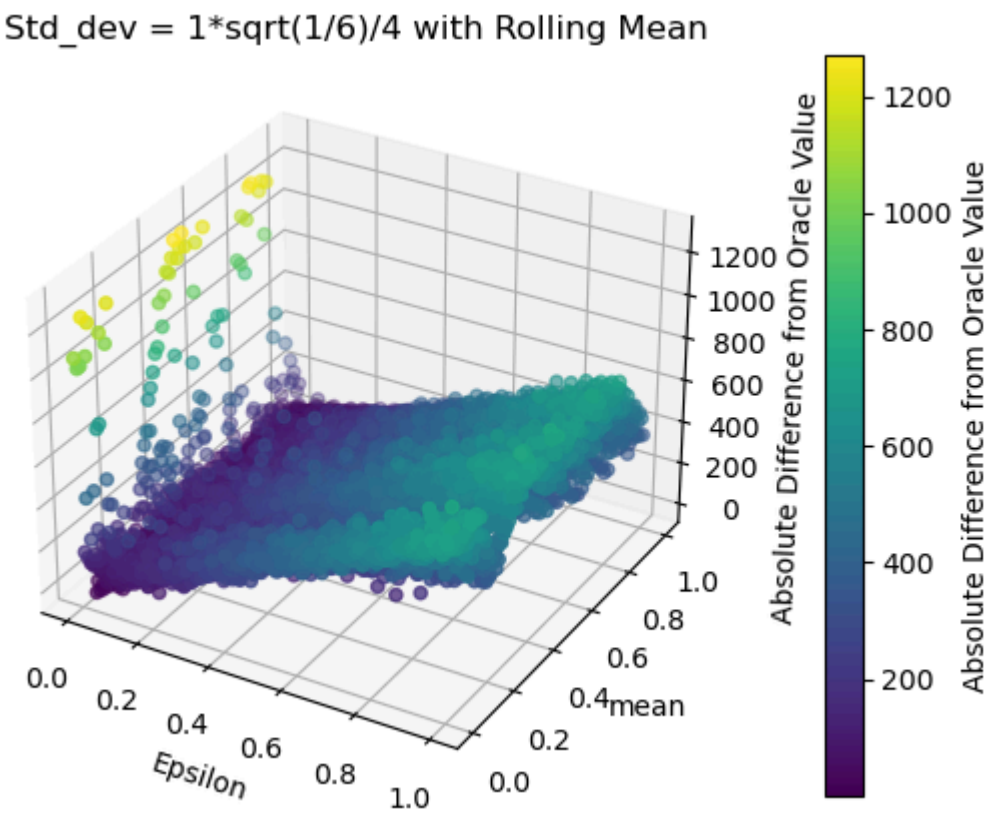
```
            rewards[i, j] = total_reward
            oracle_values[i, j] = oracle_value
    rewards_differences = np.abs(rewards - oracle_values)

    res_reward  = [num for num in rewards_differences[10] if abs(num) > 1
    y_delta_greedy_static_mean.append(res_reward)


    epsilon_values_2d = np.repeat(epsilon_values[:, np.newaxis], num_std_
    std_devs_2d = np.tile(std_devs[np.newaxis, :], (num_epsilons, 1))
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    scatter = ax.scatter(epsilon_values_2d.flatten(), std_devs_2d.flatten
                         c=rewards_differences.flatten(), cmap='viridis',
    cbar = fig.colorbar(scatter, ax=ax, pad=0.1)
    cbar.set_label('Absolute Difference from Oracle Value')
    ax.set_xlabel('Epsilon')
    ax.set_ylabel('Standard Deviation')
    ax.set_zlabel('Absolute Difference from Oracle Value')
    ax.set_title(mean_word[z])

    plt.show()




num_epsilons = 101
epsilon_values = np.linspace(0,1, num_epsilons)
num_trials = 5000

var=[0.102,0.204,0.306]
for z in range(len(var)):
    theta = static_std_dev_rolling_mean_set[z]
    num_mean = len(theta)
    rewards = np.zeros((num_epsilons, num_mean))
    means = np.linspace(0, 1, num_mean)
    oracle_values = np.zeros((num_epsilons, num_mean))

    for i, epsilon in enumerate(epsilon_values):
        for j in range(len(theta)):
            count=[0,0,0]
            theta_estimates=[0,0,0]
            total_reward = 0
            reward=0
            arm=0
            for _ in range(num_trials):
                if np.random.rand() < epsilon:
                    arm = np.random.choice([0, 1, 2])
                else:
                    arm = np.argmax(theta_estimates)
                if np.random.rand() < theta[j].seq[arm]:
                    reward = 1
                else:
                    reward = 0
                count[arm] += 1
                theta_estimates[arm] += (1/count[arm]) * (reward - theta_
                total_reward += reward
            oracle_value = num_trials * theta[j].max_theta
            rewards[i, j] = total_reward
```

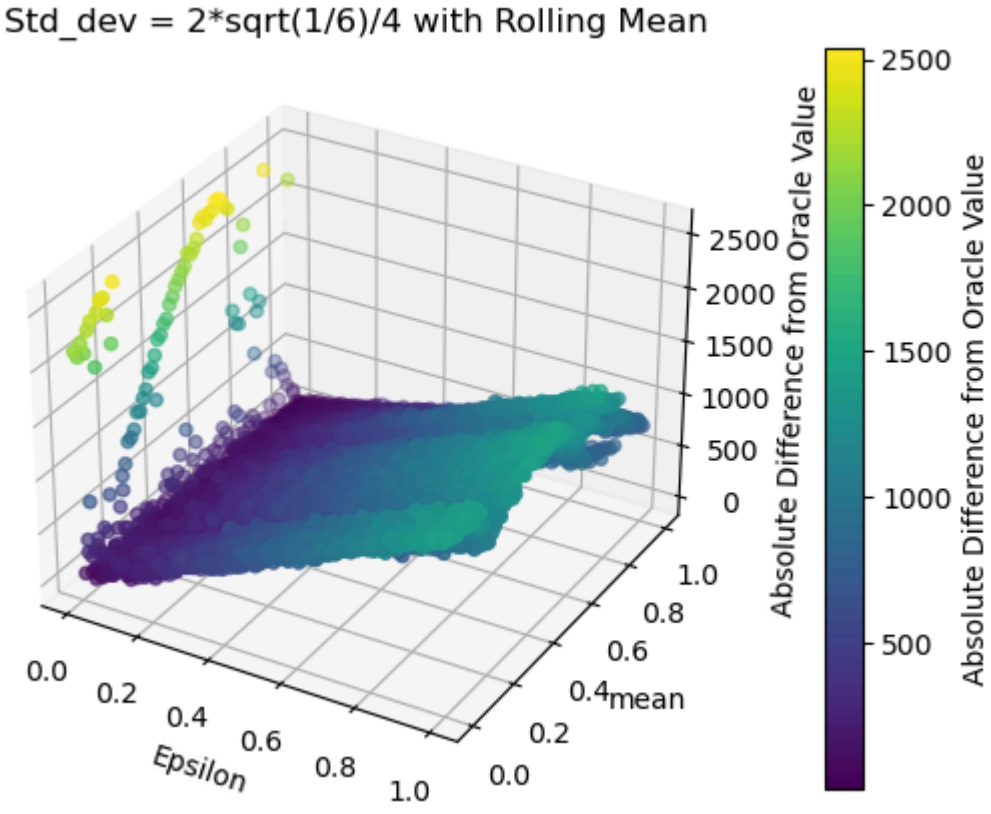
```
            oracle_values[i, j] = oracle_value
    rewards_differences = np.abs(rewards - oracle_values)

    res_reward  = [num for num in rewards_differences[10] if abs(num) > 1
    y_delta_greedy_static_mean.append(res_reward)


    epsilon_values_2d = np.repeat(epsilon_values[:, np.newaxis], num_mean
    means_2d = np.tile(means[np.newaxis, :], (num_epsilons, 1))
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    scatter = ax.scatter(epsilon_values_2d.flatten(), means_2d.flatten(),
                         c=rewards_differences.flatten(), cmap='viridis',
    cbar = fig.colorbar(scatter, ax=ax, pad=0.1)
    cbar.set_label('Absolute Difference from Oracle Value')
    ax.set_xlabel('Epsilon')
    ax.set_ylabel('mean')
    ax.set_zlabel('Absolute Difference from Oracle Value')
    ax.set_title(std_dev_word[z])
    plt.show()

#共400次个点
```

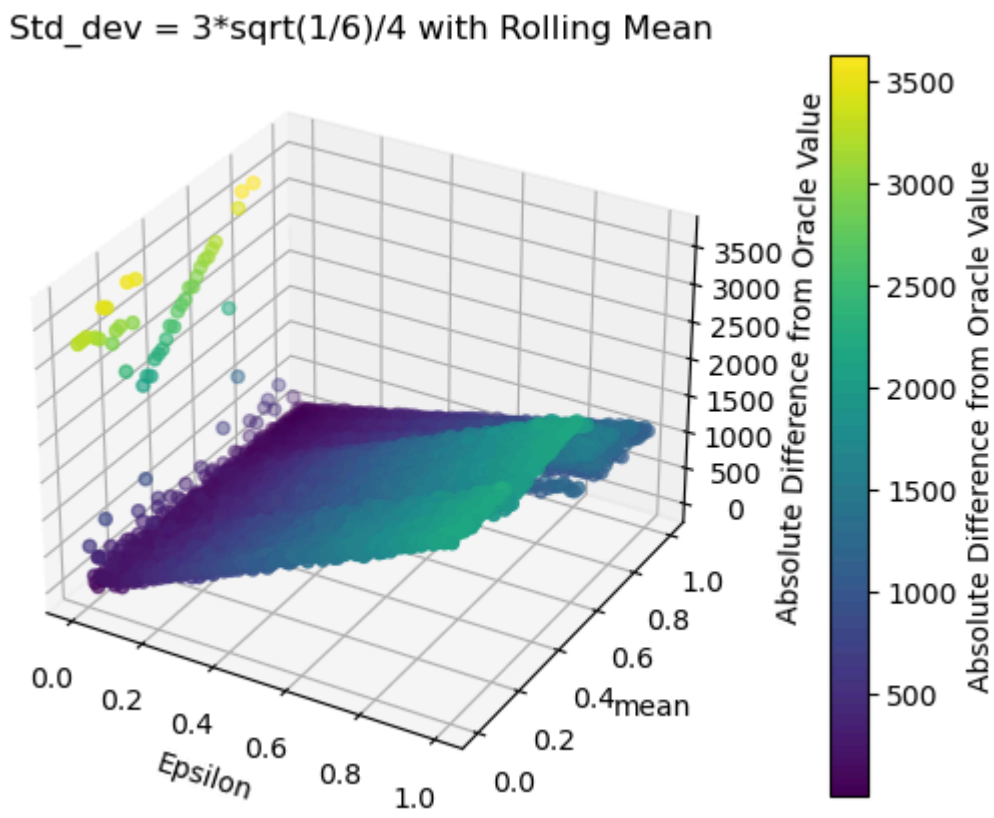



Mean = 0.50 with Rolling Std_dev
Epsilon
Standard Deviation
Absolute Difference from Oracle Value
Absolute Difference from Oracle Value


Mean = 0.75 with Rolling Std_dev
Epsilon
Standard Deviation
Absolute Difference from Oracle Value
Absolute Difference from Oracle Value


Std_dev = 1*sqrt(1/6)/4 with Rolling Mean
Absolute Difference from Oracle Value
Epsilon
mean
0.0
0.2
0.4
0.6
0.8
1.0
0
200
400
600
800
1000
1200


Std_dev = 2*sqrt(1/6)/4 with Rolling Mean
Absolute Difference from Oracle Value
Epsilon
mean
0.0
0.2
0.4
0.6
0.8
1.0
0
500
1000
1500
2000
2500

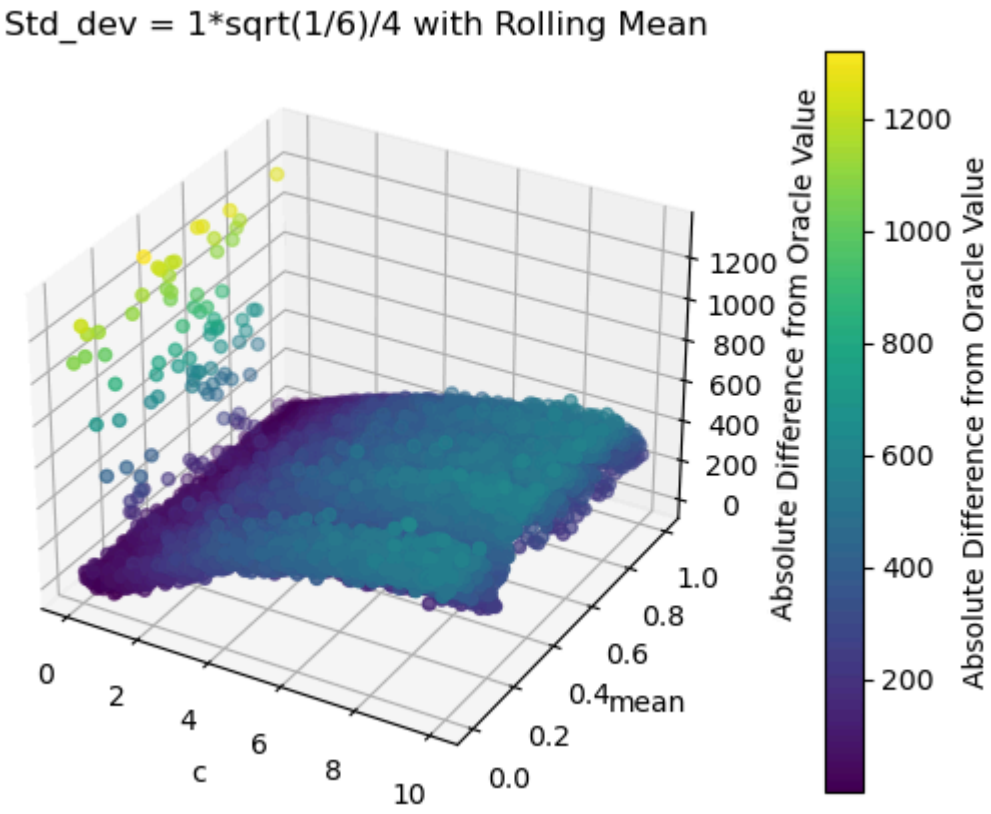


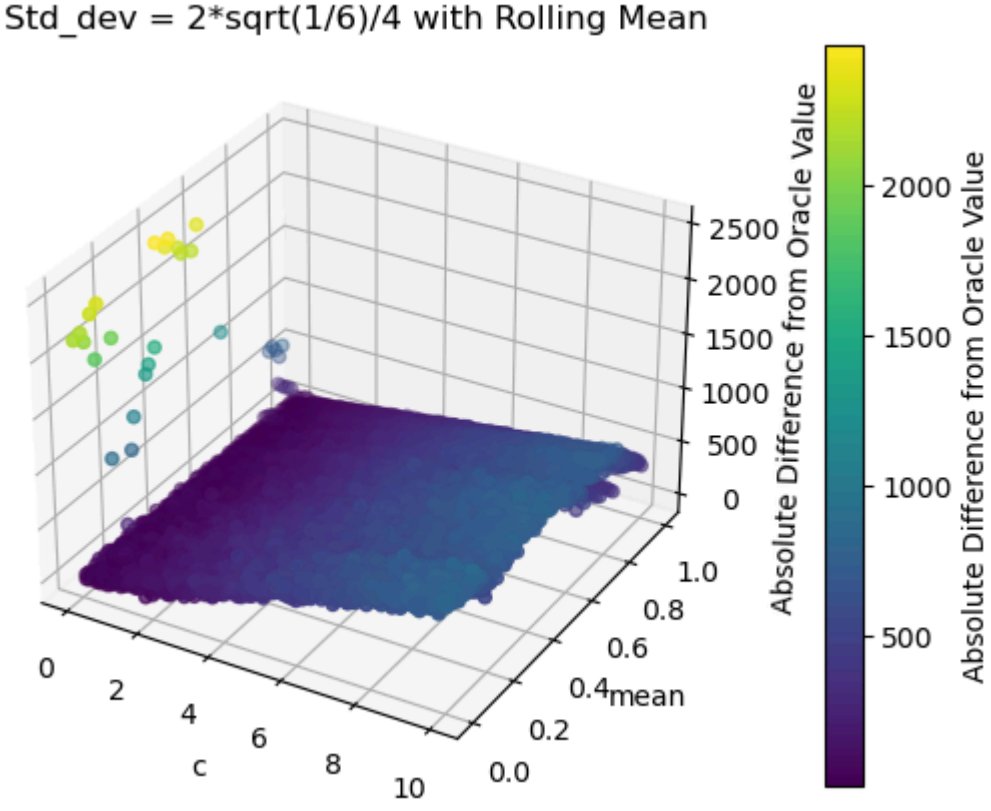
UCB Algorithm

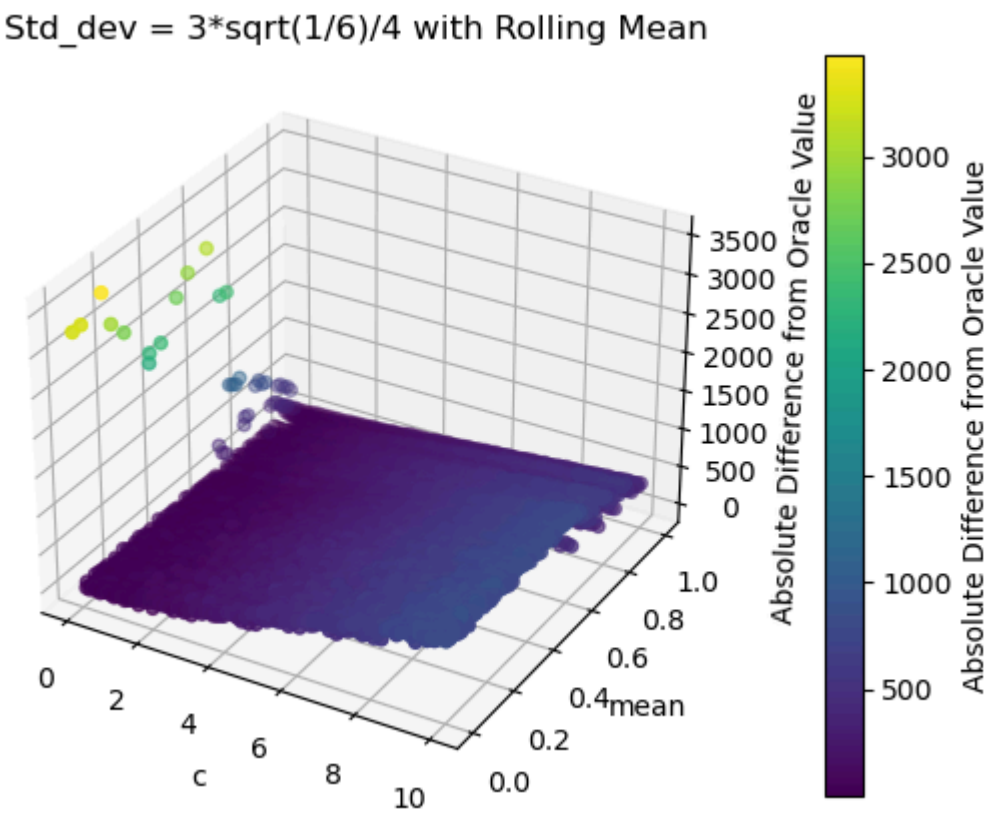
In [ ]:
```
y_delta_UCB_static_std_dev = []
y_delta_UCB_static_mean = []


mean_word = ['Mean = 0.25 with Rolling Std_dev', 'Mean = 0.50 with Rollin
std_dev_word = ['Std_dev = 1*sqrt(1/6)/4 with Rolling Mean','Std_dev = 2*


num_cs = 101
c_values = np.linspace(0,10, num_cs)
num_trials = 5000
# rewards = np.zeros((num_cs, num_std_devs))
# oracle_values = np.zeros((num_cs, num_std_devs))
var=[0.102,0.204,0.306]
for z in range(len(var)):
    theta = static_std_dev_rolling_mean_set[z]
    num_mean = len(theta)
    means = np.linspace(0,1,num_mean)

    rewards = np.zeros((num_cs, num_mean))
    oracle_values = np.zeros((num_cs, num_mean))


    for i, c in enumerate(c_values):
        for j in range(len(theta)):
            total_reward = 0
            reward=0
            arm=0
            k=-1
            theta_estimates=[0,0,0]
            for s in range(num_trials):
                k+=1
```

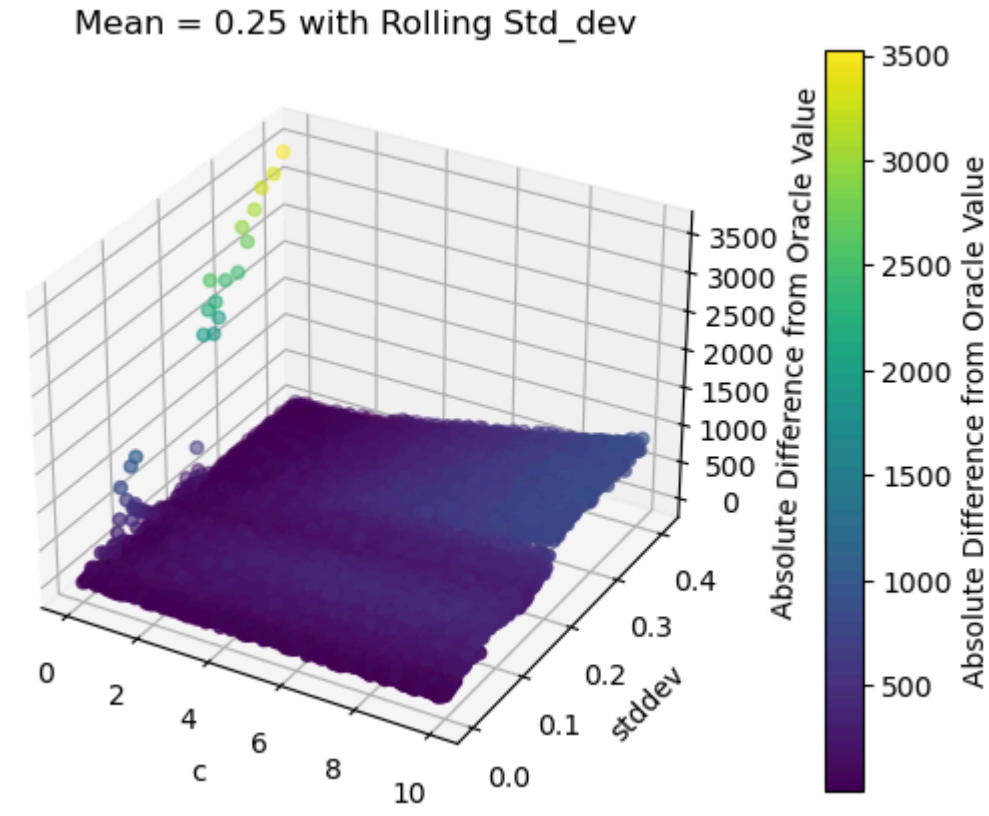
```
                if k < 3:
                    count=[1,1,1]
                    if np.random.rand() < theta[j].seq[k]:
                        theta_estimates[k]+=1
                else:
                    arm = np.argmax(theta_estimates+c*np.sqrt(2*np.log10(
                    if np.random.rand() < theta[j].seq[arm]:
                        reward = 1
                    else:
                        reward = 0

                    total_reward += reward

                count[arm] += 1

                theta_estimates[arm] += (1/count[arm]) * (reward - theta_

            oracle_value = num_trials * theta[j].max_theta
            rewards[i, j] = total_reward

            oracle_values[i, j] = oracle_value

    rewards_differencesU = np.abs(rewards - oracle_values)
    res_reward  = [num for num in rewards_differencesU[10] if abs(num) >
    y_delta_UCB_static_std_dev.append(res_reward)


    c_values_2d = np.repeat(c_values[:, np.newaxis], num_mean, axis=1)
    means_2d = np.tile(means[np.newaxis, :], (num_cs, 1))
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    scatter = ax.scatter(c_values_2d.flatten(), means_2d.flatten(), rewar
                         c=rewards_differencesU.flatten(), cmap='viridis',
    cbar = fig.colorbar(scatter, ax=ax, pad=0.1)
    cbar.set_label('Absolute Difference from Oracle Value')
    ax.set_xlabel('c')
    ax.set_ylabel('mean')
    ax.set_zlabel('Absolute Difference from Oracle Value')
    ax.set_title(std_dev_word[z])
    plt.show()



num_cs = 101
c_values = np.linspace(0,10, num_cs)
num_trials = 5000
mean=[0.25,0.5,0.75]
for z in range(len(mean)):
    theta = static_mean_rolling_std_dev_set[z]

    num_std_devs = len(theta)
    std_devs = np.linspace(0, np.sqrt(1/6), num_std_devs)
    rewards = np.zeros((num_cs, num_std_devs))
    oracle_values = np.zeros((num_cs, num_std_devs))

    for i, c in enumerate(c_values):
        for j in range(len(theta)):
            total_reward = 0
            reward=0
            arm=0
```

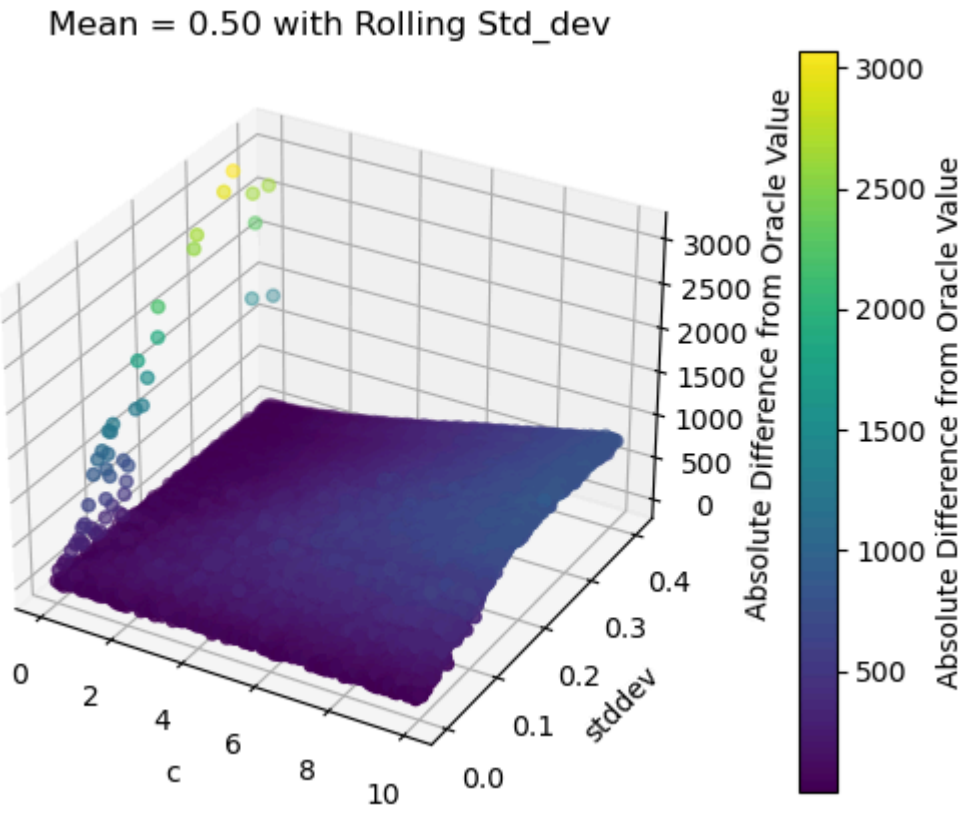
```
            k=-1
            theta_estimates=[0,0,0]
            for s in range(num_trials):
                k+=1
                if k < 3:
                    count=[1,1,1]
                    if np.random.rand() < theta[j].seq[k]:
                        theta_estimates[k]+=1
                else:
                    arm = np.argmax(theta_estimates+c*np.sqrt(2*np.log10(
                    if np.random.rand() < theta[j].seq[arm]:
                        reward = 1
                    else:
                        reward = 0

                    total_reward += reward

                count[arm] += 1

                theta_estimates[arm] += (1/count[arm]) * (reward - theta_

            oracle_value = num_trials * theta[j].max_theta
            rewards[i, j] = total_reward

            oracle_values[i, j] = oracle_value

    rewards_differencesU = np.abs(rewards - oracle_values)

    res_reward  = [num for num in rewards_differencesU[10] if abs(num) >
    y_delta_UCB_static_mean.append(res_reward)

    c_values_2d = np.repeat(c_values[:, np.newaxis], num_std_devs, axis=1
    std_devs_2d = np.tile(std_devs[np.newaxis, :], (num_cs, 1))
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    scatter = ax.scatter(c_values_2d.flatten(), std_devs_2d.flatten(), re
                         c=rewards_differencesU.flatten(), cmap='viridis',
    cbar = fig.colorbar(scatter, ax=ax, pad=0.1)
    cbar.set_label('Absolute Difference from Oracle Value')
    ax.set_xlabel('c')
    ax.set_ylabel('stddev')
    ax.set_zlabel('Absolute Difference from Oracle Value')
    ax.set_title(mean_word[z])

    plt.show()
```

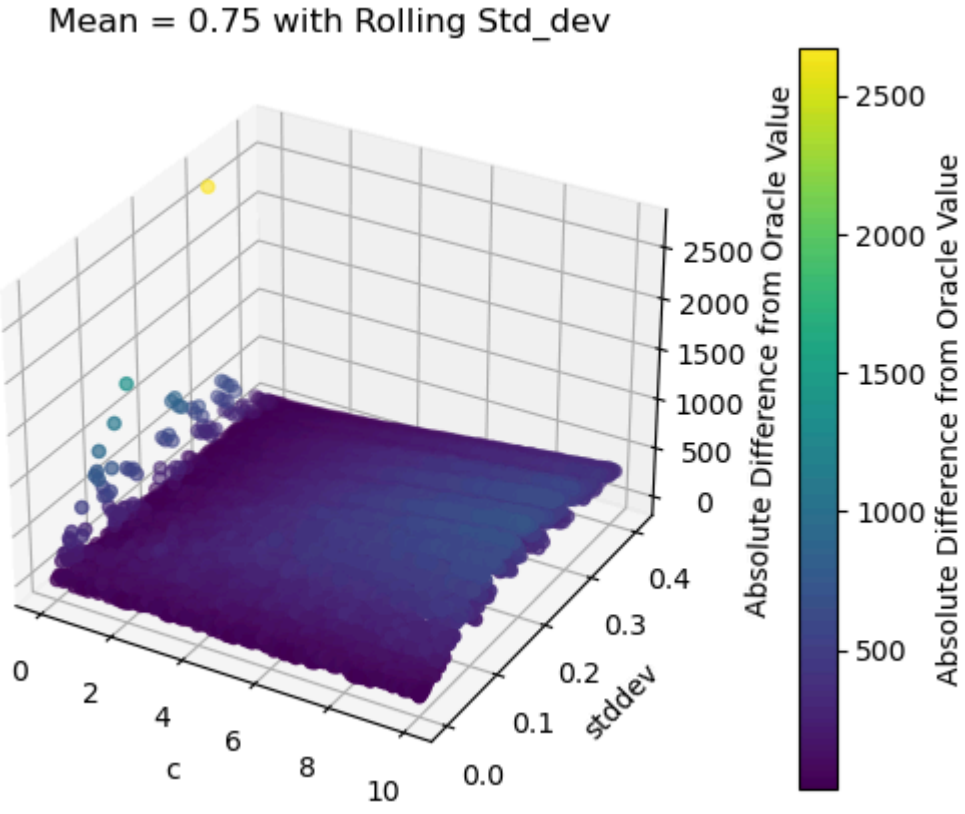

Std_dev = 1*sqrt(1/6)/4 with Rolling Mean
Absolute Difference from Oracle Value
c
mean
0
2
4
6
8
10
0.0
0.2
0.4
0.6
0.8
1.0
0
200
400
600
800
1000
1200


Std_dev = 2*sqrt(1/6)/4 with Rolling Mean
Absolute Difference from Oracle Value
c
mean
0
2
4
6
8
10
0.0
0.2
0.4
0.6
0.8
1.0
0
500
1000
1500
2000
2500


Std_dev = 3*sqrt(1/6)/4 with Rolling Mean
Absolute Difference from Oracle Value
3500
3000
2500
2000
1500
1000
500
0
c
0
2
4
6
8
10
mean
0.0
0.2
0.4
0.6
0.8
1.0


Mean = 0.25 with Rolling Std_dev
Absolute Difference from Oracle Value
3500
3000
2500
2000
1500
1000
500
0
c
0
2
4
6
8
10
stddev
0.0
0.1
0.2
0.3
0.4

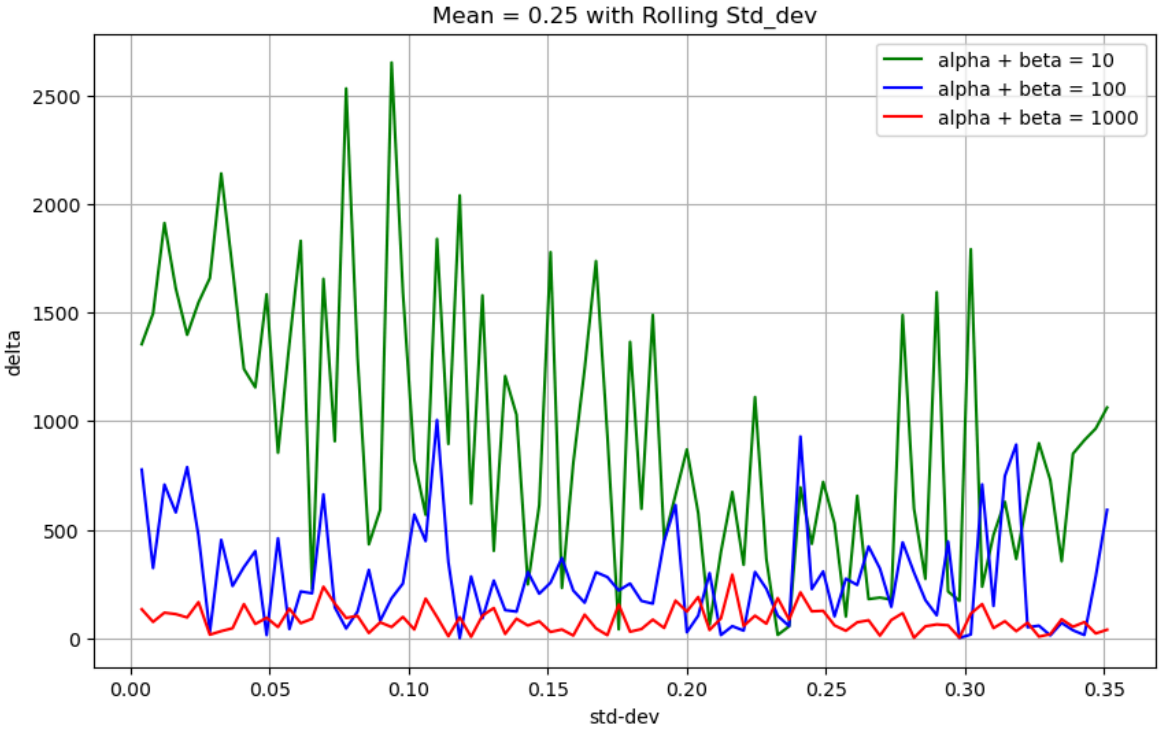


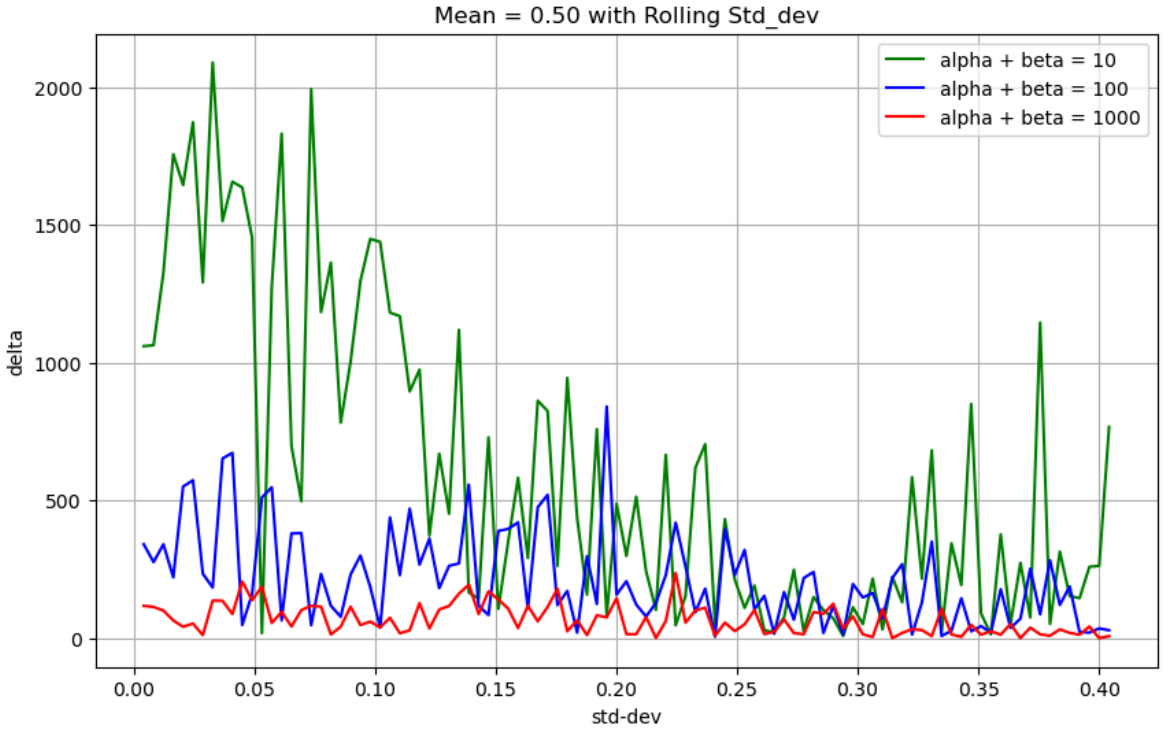


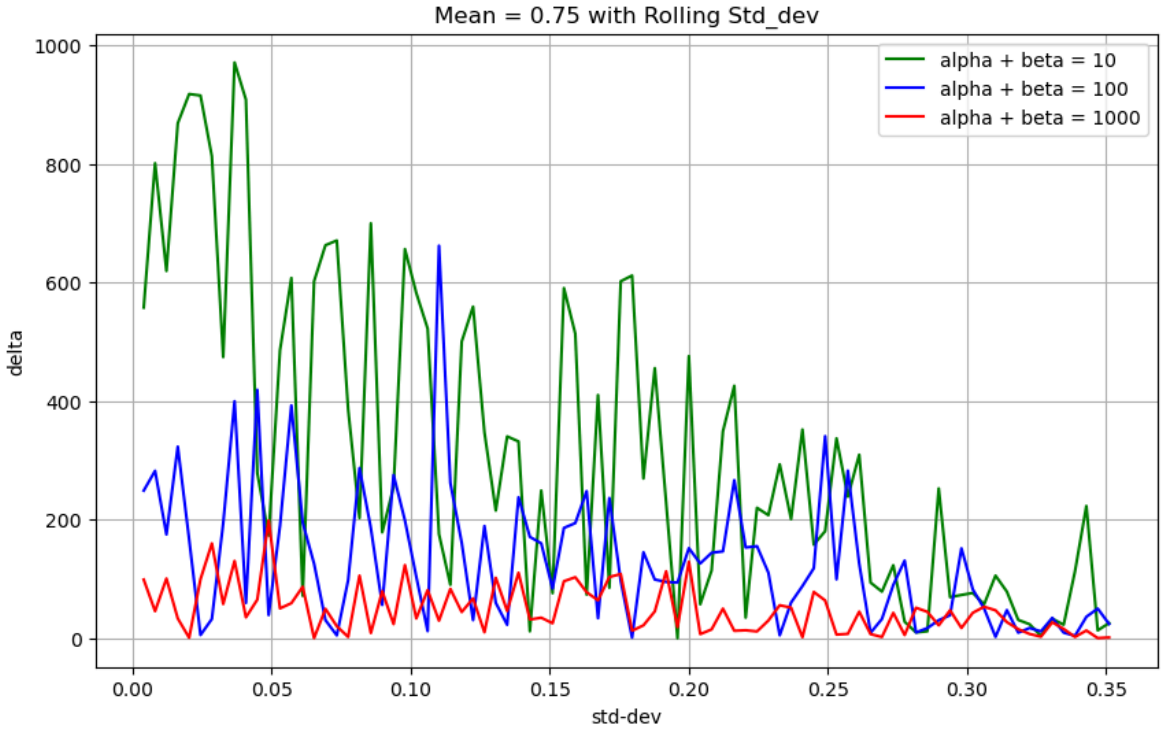
TS Algorithm

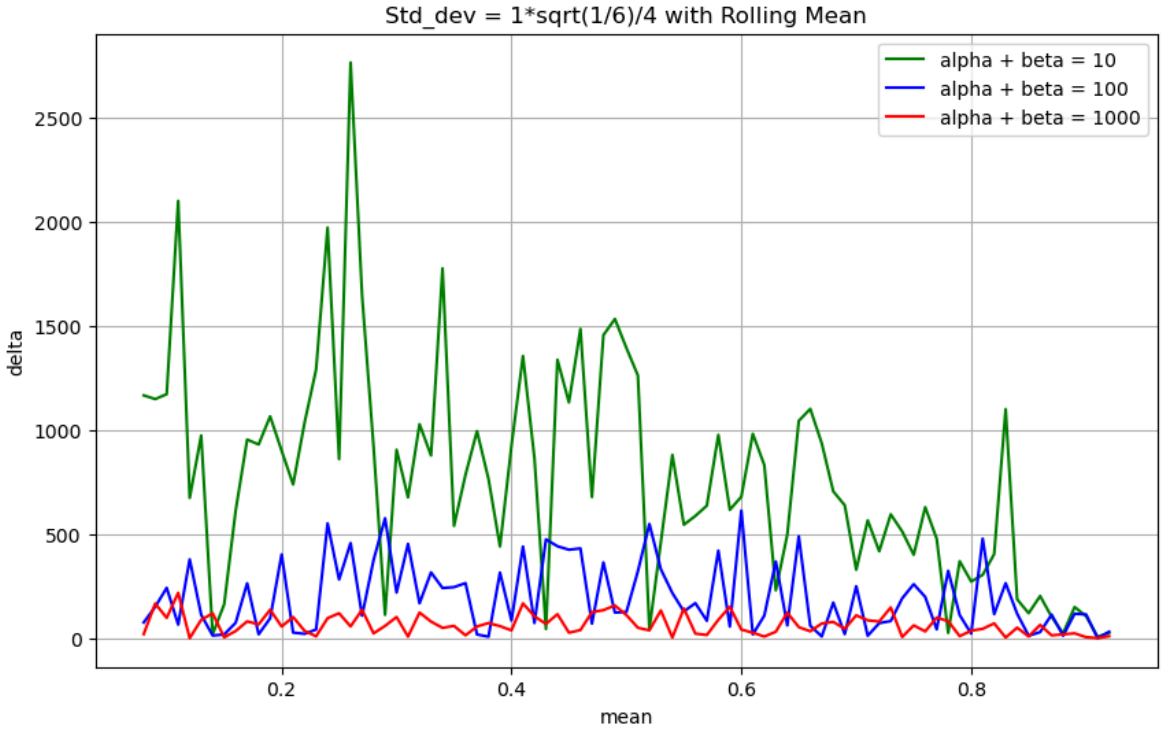
In [ ]:
```
n_arms = 3
n_experi = 10
n_slot = 5000
y_delta_static_mean = [[],[],[]]
y_delta_static_std_dev = [[],[],[]]
x_std_dev = None
x_mean = None
```

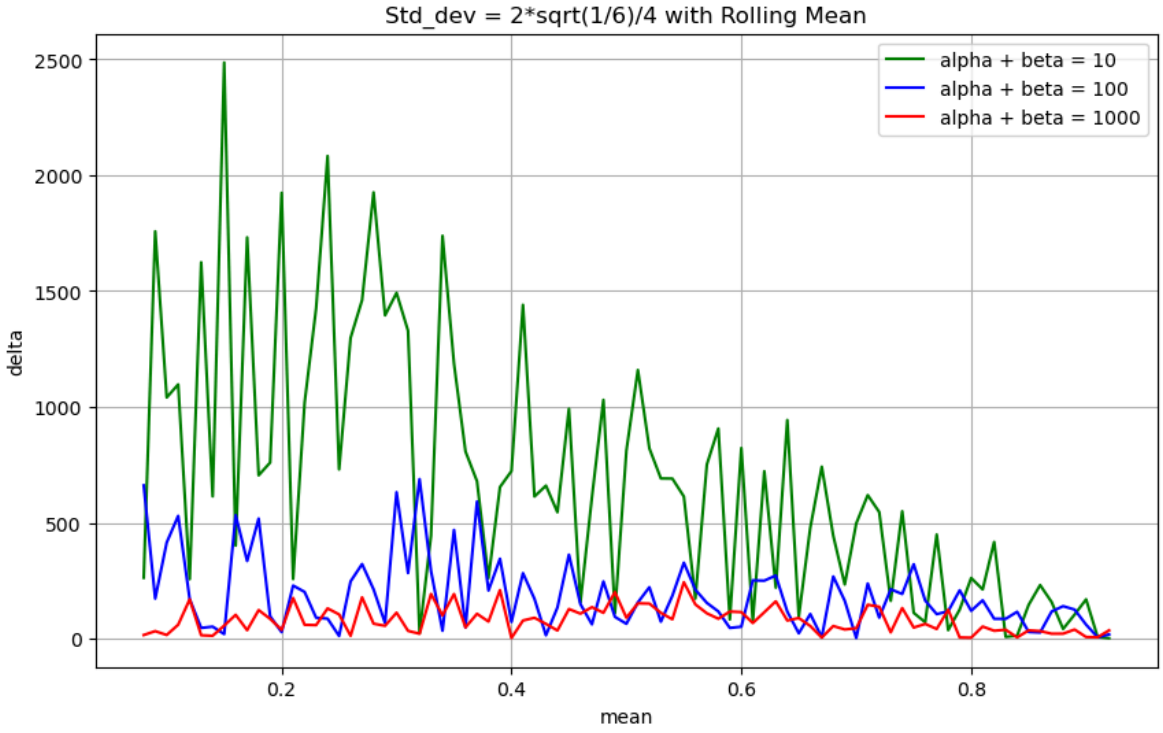
```
n_line = 3
labels = ['alpha + beta = 10','alpha + beta = 100','alpha + beta = 1000']
line_color = ['green','blue','red']
mean_word = ['Mean = 0.25 with Rolling Std_dev', 'Mean = 0.50 with Rollin
std_dev_word = ['Std_dev = 1*sqrt(1/6)/4 with Rolling Mean','Std_dev = 2*

for i in range(len(static_mean_rolling_std_dev_set)):
  para_exp = beta_prio_generate(static_mean_rolling_std_dev_set[i], min_n
  y_delta_static_mean[i], _, x_std_dev = packaged_Thompson_Sampling(stati
  plot_n_lines(n_line, x_std_dev, y_delta_static_mean[i], mean_word[i] ,'

for i in range(len(static_std_dev_rolling_mean_set)):
  para_exp = beta_prio_generate(static_std_dev_rolling_mean_set[0], min_n
  y_delta_static_std_dev[i], x_mean, _ = packaged_Thompson_Sampling(stati
  plot_n_lines(n_line, x_mean, y_delta_static_std_dev[i], std_dev_word[i]
```

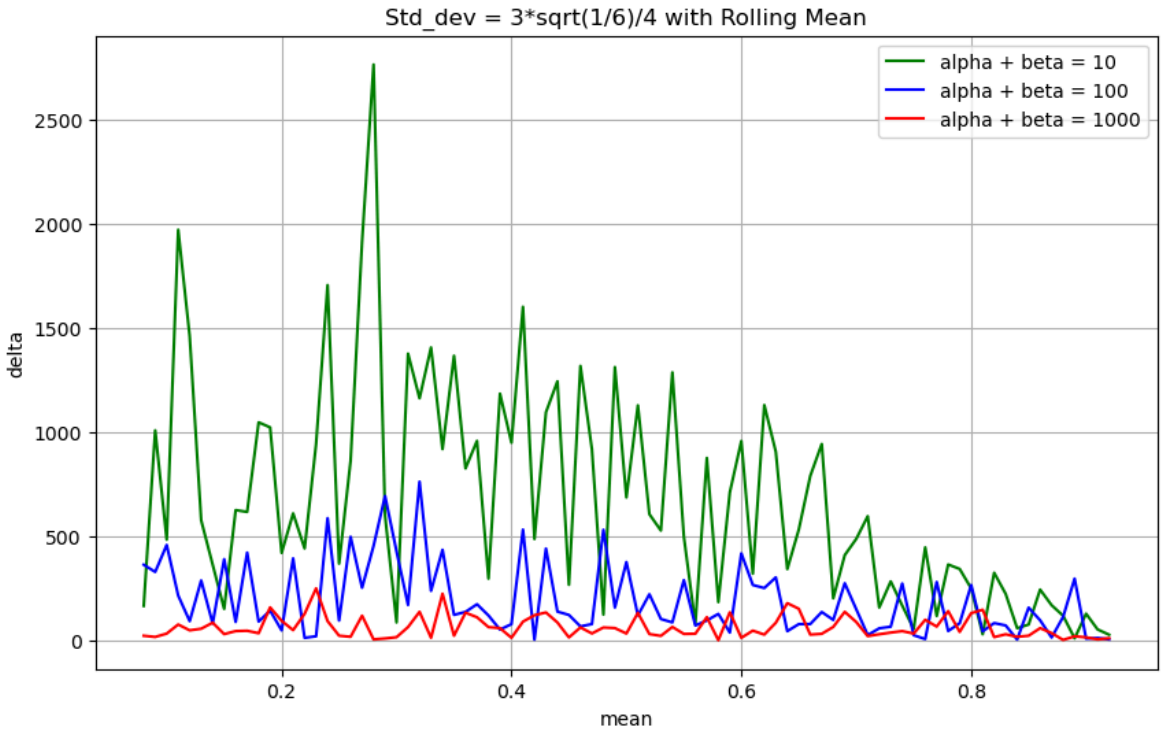



Mean = 0.50 with Rolling Std_dev
alpha + beta = 10
alpha + beta = 100
alpha + beta = 1000
delta
std-dev
0
500
1000
1500
2000
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40


Mean = 0.75 with Rolling Std_dev
alpha + beta = 10
alpha + beta = 100
alpha + beta = 1000
delta
std-dev
0
200
400
600
800
1000
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35

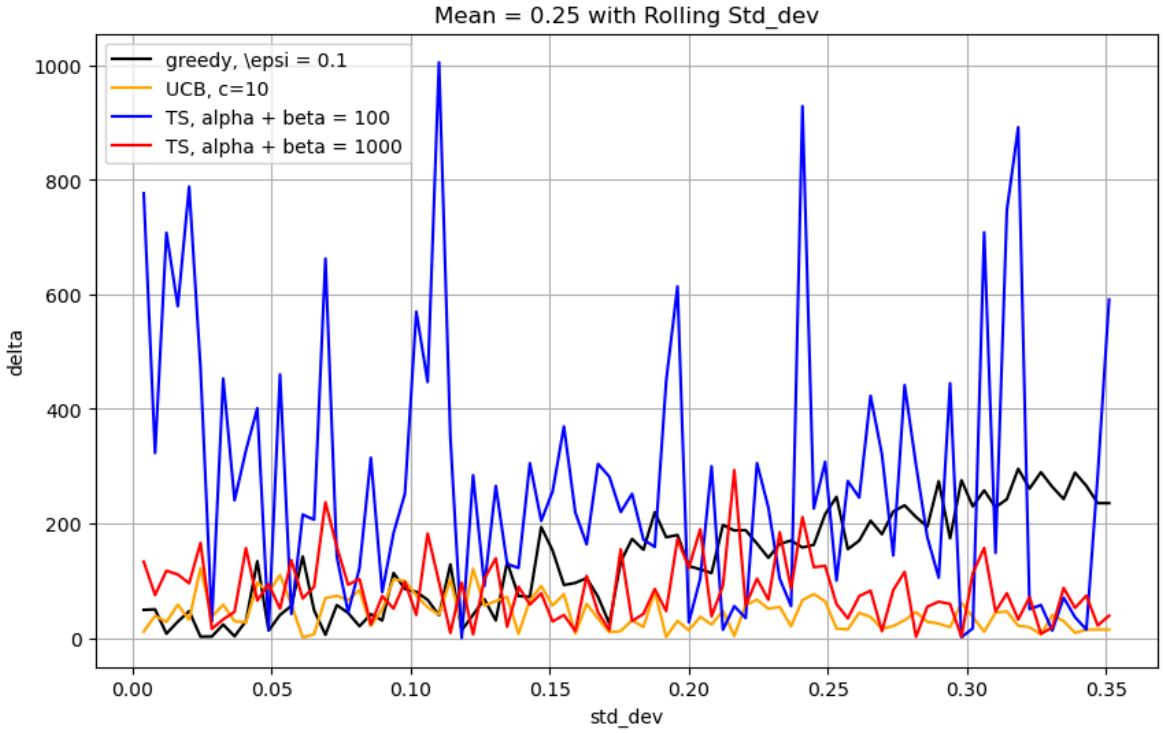

Std_dev = 1*sqrt(1/6)/4 with Rolling Mean
alpha + beta = 10
alpha + beta = 100
alpha + beta = 1000
delta
mean
0
500
1000
1500
2000
2500
0.2
0.4
0.6
0.8

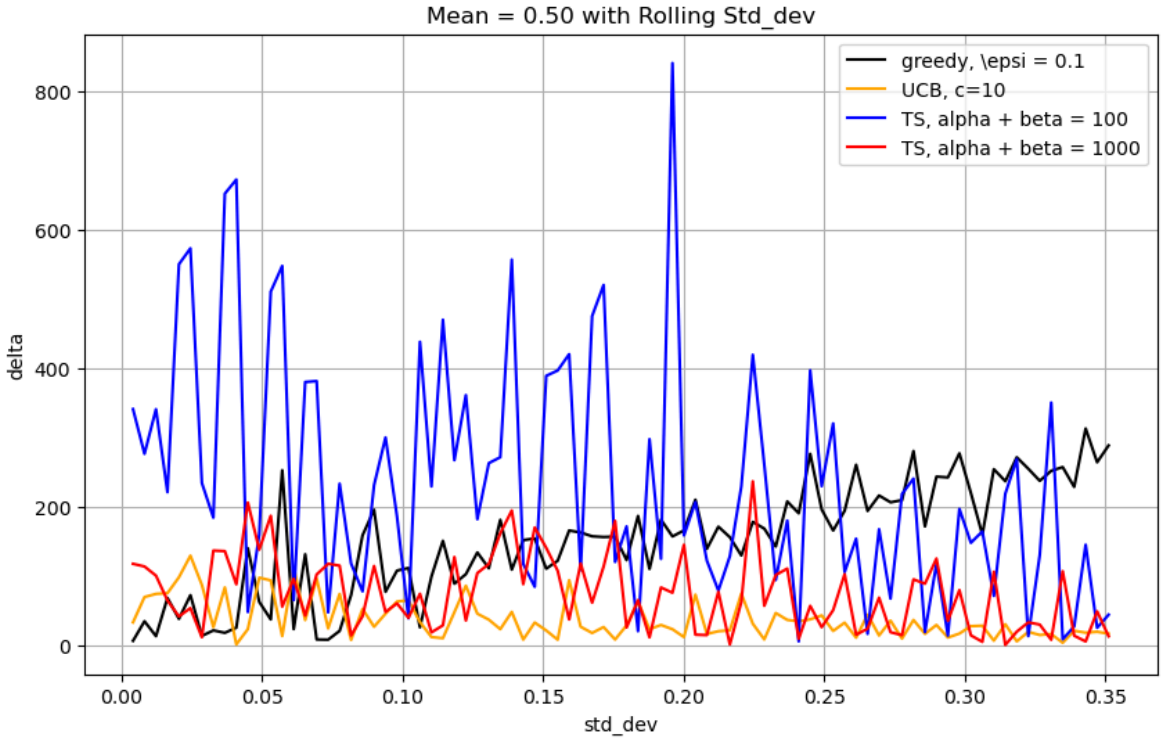

Std_dev = 2*sqrt(1/6)/4 with Rolling Mean
alpha + beta = 10
alpha + beta = 100
alpha + beta = 1000
delta
mean
0
500
1000
1500
2000
2500
0.2
0.4
0.6
0.8

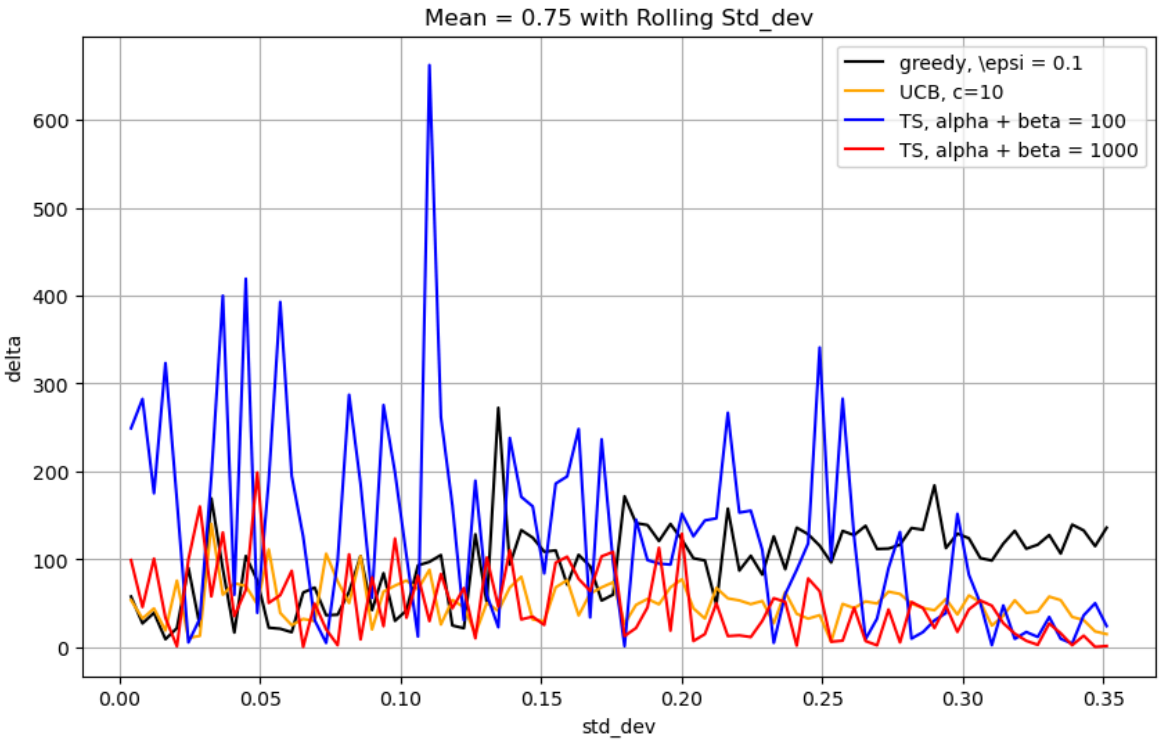


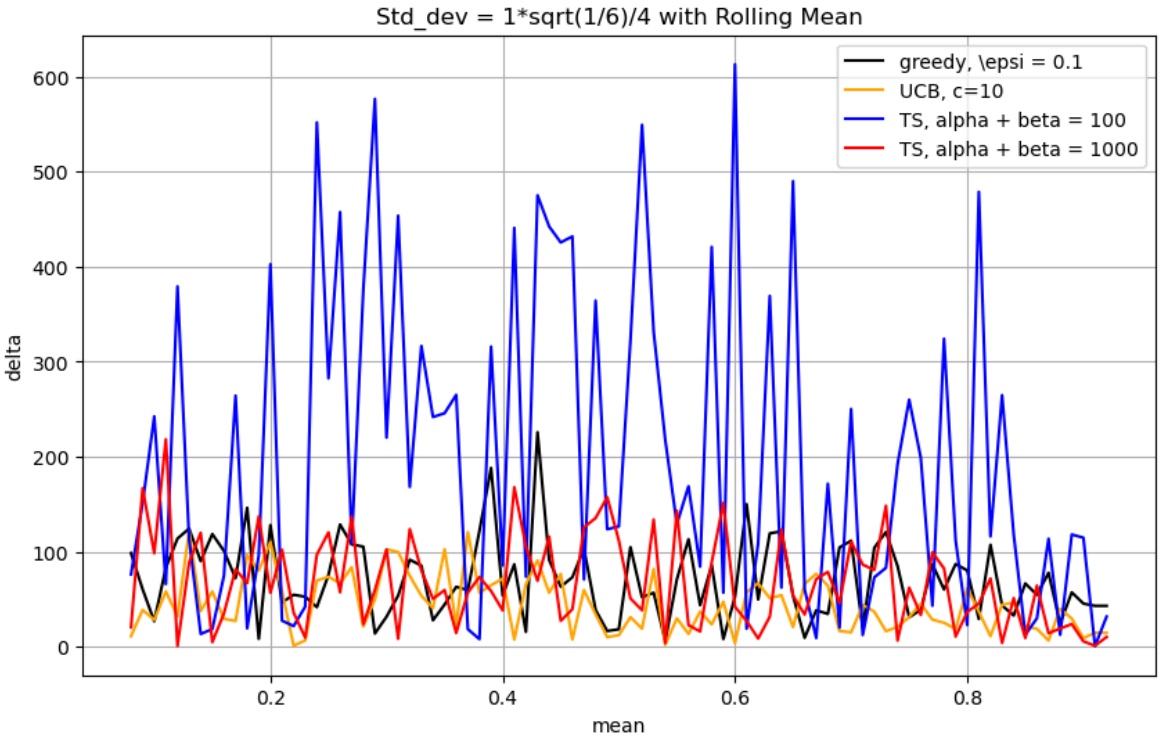
Total Comparation

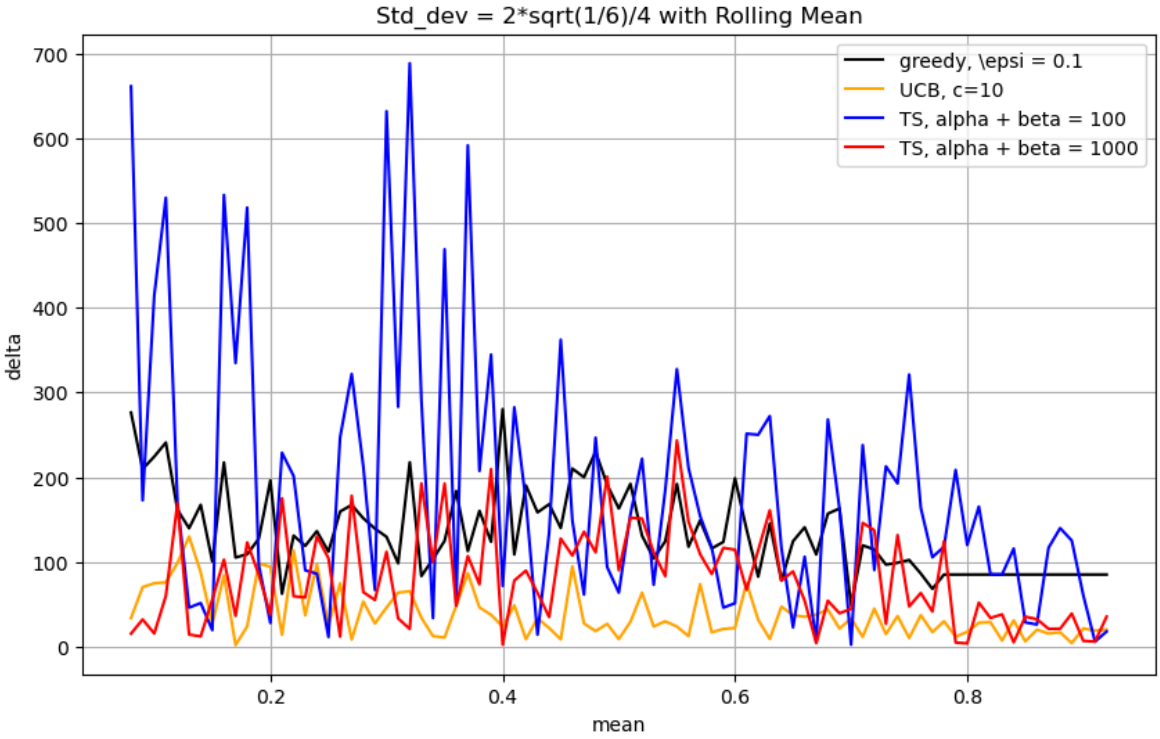
In [ ]:
```
## 把greedy, UCB的放进去
n_line = 4
y_labels = ['greedy, \epsi = 0.1','UCB, c=10','TS, alpha + beta = 100','T



for i in range(3):
      y_delta_greedy_static_std_dev.append(y_delta_greedy_static_mean[3+i

y_delta_total_comp_static_mean = [[y_delta_greedy_static_mean[i],y_delta_
y_delta_total_comp_static_std_dev = [[y_delta_greedy_static_std_dev[i],y_
line_color = ['black','orange','blue','red']

for i in range(3):
  plot_n_lines(n_line, x_std_dev, y_delta_total_comp_static_mean[i], mean

for i in range(3):
  plot_n_lines(n_line, x_mean, y_delta_total_comp_static_std_dev[i], std_
```

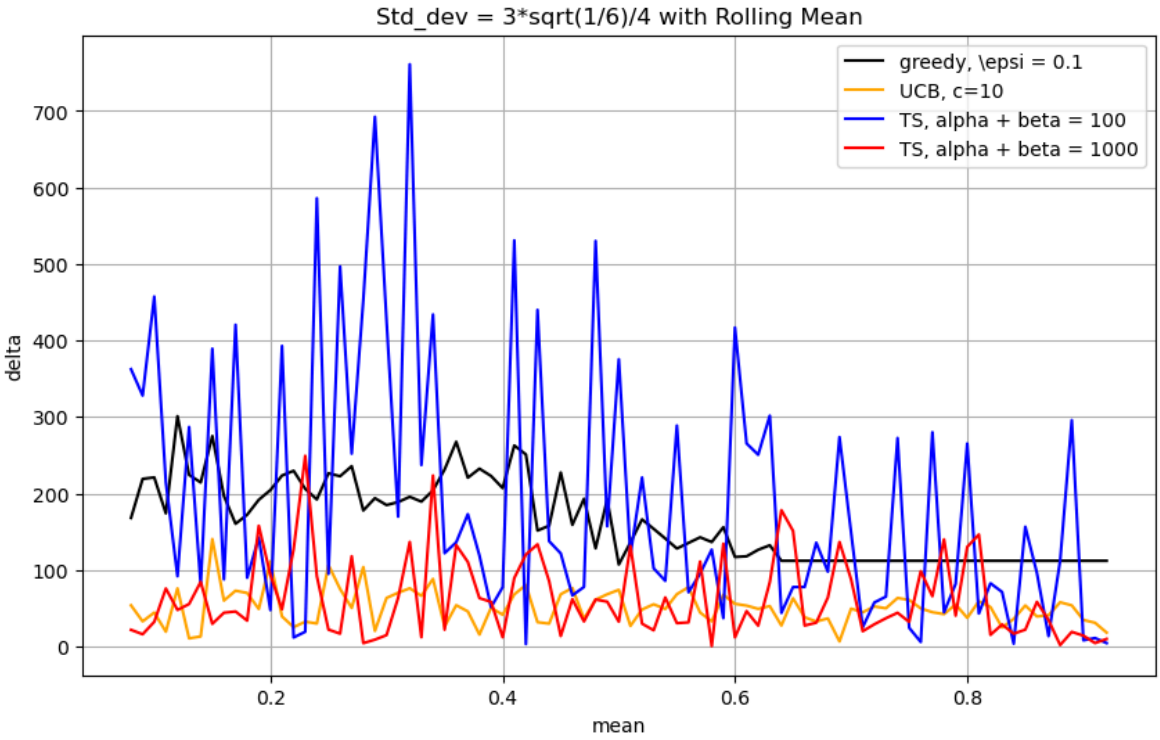

Mean = 0.25 with Rolling Std_dev
greedy, \epsi = 0.1
UCB, c=10
TS, alpha + beta = 100
TS, alpha + beta = 1000
delta
std_dev
0
200
400
600
800
1000
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35


Mean = 0.50 with Rolling Std_dev
greedy, \epsi = 0.1
UCB, c=10
TS, alpha + beta = 100
TS, alpha + beta = 1000
delta
std_dev
0
200
400
600
800
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35


Mean = 0.75 with Rolling Std_dev
greedy, \epsi = 0.1
UCB, c=10
TS, alpha + beta = 100
TS, alpha + beta = 1000
delta
std_dev
0
100
200
300
400
500
600
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35


Std_dev = 1*sqrt(1/6)/4 with Rolling Mean
greedy, \epsi = 0.1
UCB, c=10
TS, alpha + beta = 100
TS, alpha + beta = 1000
delta
mean
0
100
200
300
400
500
600
0.2
0.4
0.6
0.8





Problem 6

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
# Initializing the y values for errors from the provided code
y = [0, 0, 0, 0, 0, 0]

num_experiments = 200  # Number of experiments
num_pulls_per_experiment = 5000  # Number of pulls per experiment
k = 1  # Total probability sum for the arms
m = 2000
# Initializing UCB and optimized UCB algorithm error lists
ucb_errors = []
```

```
optimized_ucb_errors = []

# Setting UCB c values and epsilon-greedy epsilon values
ucb_cs = [1, 2, 3]
#epsilons = [0.1, 0.5, 0.9]

# Initializing arrays to store rewards and oracle values
rewards = np.zeros((len(ucb_cs), num_experiments))
oracle_values = np.zeros((len(ucb_cs), num_experiments))

# Simulating the optimized UCB algorithm
for i, c in enumerate(ucb_cs):
    for experiment in range(num_experiments):
        theta = np.random.uniform(0, 1, 3)
        while (theta[0] + theta[1] > k):
            theta = np.random.uniform(0, 1, 3)
        theta[2] = k - theta[0] - theta[1]

        total_reward = 0
        theta_estimates = np.zeros_like(theta)
        counts = np.ones_like(theta)  # Initialize selection counts

        for trial in range(num_pulls_per_experiment):
            if trial < 3:  # Random selection for the first three trials
                arm = np.random.randint(len(theta))
                reward = 1 if np.random.rand() < theta[arm] else 0
                total_reward += reward
                theta_estimates[arm] = reward
            if trial == 2:
                if (np.sum(theta_estimates))>0:
                    theta_estimates = theta_estimates * k / np.sum(theta_
            else:
                # Calculate UCB values
                ucb_values = theta_estimates + c * np.sqrt((2 * np.log(tr
                arm = np.argmax(ucb_values)
                reward = 1 if np.random.rand() < theta[arm] else 0
                total_reward += reward
                counts[arm] += 1
                theta_estimates[arm] += (reward - theta_estimates[arm]) /
                if ((np.sum(theta_estimates))>0 & trial > m):
                    theta_estimates = theta_estimates * k / np.sum(theta_

        rewards[i, experiment] = total_reward
        oracle_values[i, experiment] = num_pulls_per_experiment * np.max(

# Calculate average errors for the optimized UCB algorithm
average_rewards = np.mean(rewards, axis=1)
average_oracle_values = np.mean(oracle_values, axis=1)
average_errors = np.abs(average_rewards - average_oracle_values)
optimized_ucb_errors = average_errors

# Simulating the standard UCB algorithm
for i, c in enumerate(ucb_cs):
    for experiment in range(num_experiments):
        theta = np.random.uniform(0, 1, 3)
        while (theta[0] + theta[1] > k):
            theta = np.random.uniform(0, 1, 3)
        theta[2] = k - theta[0] - theta[1]

        total_reward = 0
```

```
        theta_estimates = np.zeros_like(theta)
        counts = np.ones_like(theta)  # Initialize selection counts

        for trial in range(num_pulls_per_experiment):
            if trial < 3:  # Random selection for the first three trials
                arm = np.random.randint(len(theta))
            else:
                # Calculate UCB values
                ucb_values = theta_estimates + c * np.sqrt((2 * np.log(tr
                arm = np.argmax(ucb_values)

            reward = 1 if np.random.rand() < theta[arm] else 0
            total_reward += reward
            counts[arm] += 1
            theta_estimates[arm] += (reward - theta_estimates[arm]) / cou

        rewards[i, experiment] = total_reward
        oracle_values[i, experiment] = num_pulls_per_experiment * np.max(

# Calculate average errors for the standard UCB algorithm
average_rewards = np.mean(rewards, axis=1)
average_oracle_values = np.mean(oracle_values, axis=1)
average_errors = np.abs(average_rewards - average_oracle_values)
ucb_errors = average_errors

# Plotting the results
plt.figure(figsize=(10, 6))
plt.plot(ucb_cs, ucb_errors, label='Standard UCB', marker='o', color='blu
plt.plot(ucb_cs, optimized_ucb_errors, label='Optimized UCB', marker='x',
plt.xlabel('c value')
plt.ylabel('Average Error')
plt.title('Comparison of Average Error: Standard UCB vs Optimized UCB')
plt.legend()
plt.grid(True)
plt.show()
```

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt
# Initializing the y values for errors from the provided code
y = [0, 0, 0, 0, 0, 0]

num_experiments = 200  # Number of experiments
num_pulls_per_experiment = 5000  # Number of pulls per experiment
k = 1  # Total probability sum for the arms
m = 2000
delta = 0.1
# Initializing UCB and optimized UCB algorithm error lists
ucb_errors = []
optimized_ucb_errors = []

# Setting UCB c values and epsilon-greedy epsilon values
ucb_cs = [1, 2, 3]
#epsilons = [0.1, 0.5, 0.9]

# Initializing arrays to store rewards and oracle values
rewards = np.zeros((len(ucb_cs), num_experiments))
oracle_values = np.zeros((len(ucb_cs), num_experiments))
not_unified = 1
equal_arm = [0,0]
# Simulating the optimized UCB algorithm
```

```
for i, c in enumerate(ucb_cs):
    for experiment in range(num_experiments):
        theta = np.random.uniform(0, 1, 3)
        theta[1] = theta[2]
        total_reward = 0
        theta_estimates = np.zeros_like(theta)
        counts = np.ones_like(theta)  # Initialize selection counts

        for trial in range(num_pulls_per_experiment):
            if trial < 3:  # Random selection for the first three trials
                arm = np.random.randint(len(theta))
                reward = 1 if np.random.rand() < theta[arm] else 0
                total_reward += reward
                theta_estimates[arm] = reward
            else:
                # Calculate UCB values
                if (trial > m):
                    diff = [abs(theta_estimates[0]-theta_estimates[1]),ab
                    min_diff = np.min(diff)
                    if (min_diff > delta and not_unified):
                        not_unified = 0
                        if min_diff == diff[0]:
                            sum_counts = counts[0] + counts[1]
                            mean = (theta_estimates[0]+theta_estimates[1]
                            counts[0] = sum_counts
                            counts[1] = sum_counts
                            theta_estimates[0] = mean
                            theta_estimates[1] = mean
                            equal_arm = [0,1]
                        elif min_diff == diff[1]:
                            sum_counts = counts[2] + counts[1]
                            mean = (theta_estimates[2]+theta_estimates[1]
                            counts[2] = sum_counts
                            counts[1] = sum_counts
                            theta_estimates[2] = mean
                            theta_estimates[1] = mean
                            equal_arm = [2,1]
                        else:
                            sum_counts = counts[2] + counts[0]
                            mean = (theta_estimates[0]+theta_estimates[2]
                            counts[0] = sum_counts
                            counts[2] = sum_counts
                            theta_estimates[0] = mean
                            theta_estimates[2] = mean
                            equal_arm = [0,2]
                    ucb_values = theta_estimates + c * np.sqrt((2 * np.log(tr
                    arm = np.argmax(ucb_values)
                    reward = 1 if np.random.rand() < theta[arm] else 0
                    total_reward += reward
                    if(not not_unified and (arm == equal_arm[0] or arm == equ
                        counts[equal_arm[0]] += 1
                        counts[equal_arm[1]] += 1
                        theta_estimates[equal_arm[0]] += (reward - theta_esti
                        theta_estimates[equal_arm[1]] = theta_estimates[equal
                    else:
                        counts[arm] += 1
                        theta_estimates[arm] += (reward - theta_estimates[arm
        rewards[i, experiment] = total_reward
        oracle_values[i, experiment] = num_pulls_per_experiment * np.max(
```

```
# Calculate average errors for the optimized UCB algorithm
average_rewards = np.mean(rewards, axis=1)
average_oracle_values = np.mean(oracle_values, axis=1)
average_errors = np.abs(average_rewards - average_oracle_values)
optimized_ucb_errors = average_errors

# Simulating the standard UCB algorithm
for i, c in enumerate(ucb_cs):
    for experiment in range(num_experiments):
        theta = np.random.uniform(0, 1, 3)
        theta[1] = theta[2]
        total_reward = 0
        theta_estimates = np.zeros_like(theta)
        counts = np.ones_like(theta)  # Initialize selection counts

        for trial in range(num_pulls_per_experiment):
            if trial < 3:  # Random selection for the first three trials
                arm = np.random.randint(len(theta))
            else:
                # Calculate UCB values
                ucb_values = theta_estimates + c * np.sqrt((2 * np.log(tr
                arm = np.argmax(ucb_values)

            reward = 1 if np.random.rand() < theta[arm] else 0
            total_reward += reward
            counts[arm] += 1
            theta_estimates[arm] += (reward - theta_estimates[arm]) / cou

        rewards[i, experiment] = total_reward
        oracle_values[i, experiment] = num_pulls_per_experiment * np.max(

# Calculate average errors for the standard UCB algorithm
average_rewards = np.mean(rewards, axis=1)
average_oracle_values = np.mean(oracle_values, axis=1)
average_errors = np.abs(average_rewards - average_oracle_values)
ucb_errors = average_errors

# Plotting the results
plt.figure(figsize=(10, 6))
plt.plot(ucb_cs, ucb_errors, label='Standard UCB', marker='o', color='blu
plt.plot(ucb_cs, optimized_ucb_errors, label='Optimized UCB', marker='x',
plt.xlabel('c value')
plt.ylabel('Average Error')
plt.title('Comparison of Average Error: Standard UCB vs Optimized UCB')
plt.legend()
plt.grid(True)
plt.show()
```

Part II: Bayesian Bandit Algorithms

Problem 1

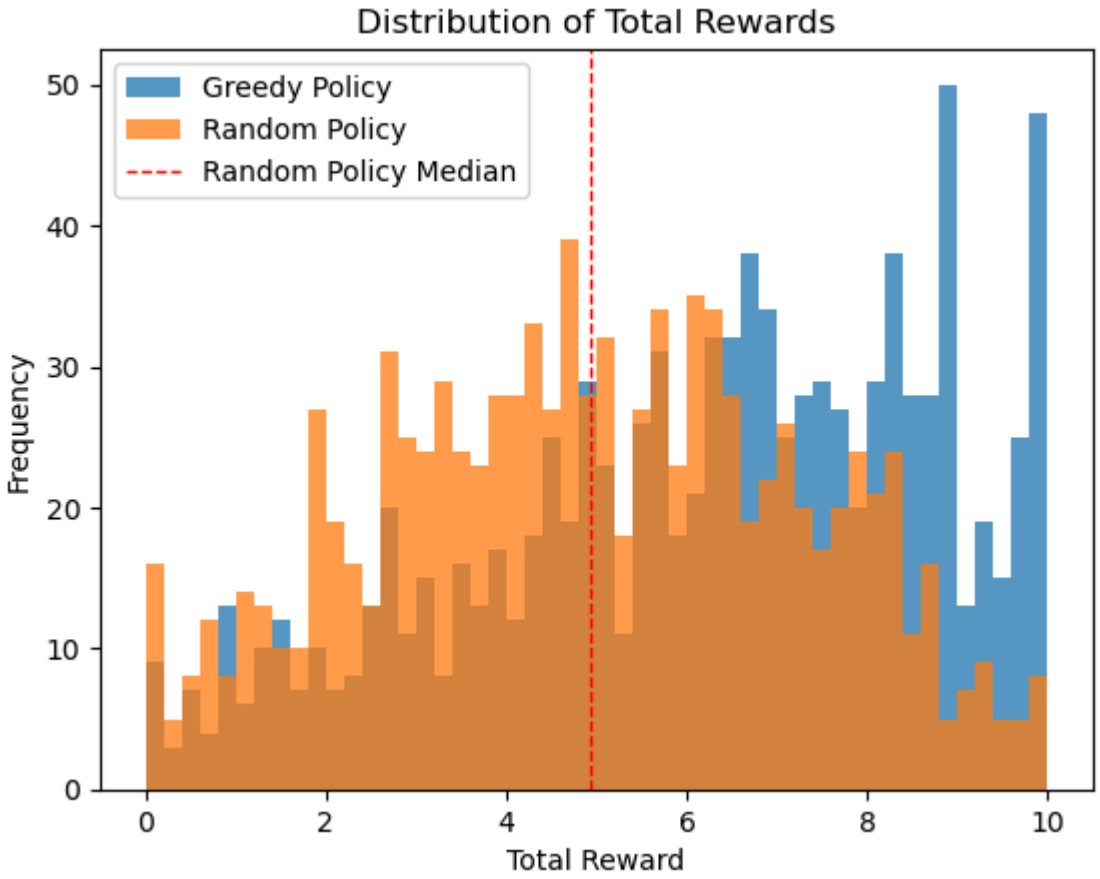
In [ ]:

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

# Parameters
gamma = 0.9  # Discount factor
```

```
num_simulations = 1000  # Number of simulations to run
num_pulls = 1000  # Number of pulls in each simulation
alpha1, beta1 = 1, 1  # Initial parameters for arm 1
alpha2, beta2 = 1, 1  # Initial parameters for arm 2

# Function to simulate the greedy policy
def simulate_greedy_policy(alpha1, beta1, alpha2, beta2, num_pulls, gamma
    alpha = [alpha1, alpha2]
    beta = [beta1, beta2]
    total_reward = 0

    for t in range(1, num_pulls + 1):
        # Calculate the expected values of theta for each arm
        expected_theta1 = alpha[0] / (alpha[0] + beta[0])
        expected_theta2 = alpha[1] / (alpha[1] + beta[1])

        # Select the arm with the highest expected theta
        chosen_arm = 0 if expected_theta1 > expected_theta2 else 1

        # Simulate the result of pulling the chosen arm
        success = np.random.rand() < (alpha[chosen_arm] / (alpha[chosen_a

        # Update the parameters of the chosen arm's Beta distribution
        if success:
            alpha[chosen_arm] += 1
            reward = gamma ** (t - 1)
        else:
            beta[chosen_arm] += 1
            reward = 0

        # Accumulate the reward
        total_reward += reward

    return total_reward

# Function to simulate the random selection policy
def simulate_random_policy(alpha1, beta1, alpha2, beta2, num_pulls, gamma
    alpha = [alpha1, alpha2]
    beta = [beta1, beta2]
    total_reward = 0

    for t in range(1, num_pulls + 1):
        # Randomly select an arm
        chosen_arm = np.random.choice([0, 1])

        # Simulate the result of pulling the chosen arm
        success = np.random.rand() < (alpha[chosen_arm] / (alpha[chosen_a

        # Update the parameters of the chosen arm's Beta distribution
        if success:
            alpha[chosen_arm] += 1
            reward = gamma ** (t - 1)
        else:
            beta[chosen_arm] += 1
            reward = 0

        # Accumulate the reward
        total_reward += reward

    return total_reward
```

```
# Run the simulations for both policies
greedy_rewards = []
random_rewards = []
for _ in range(num_simulations):
    greedy_rewards.append(simulate_greedy_policy(alpha1, beta1, alpha2, b
    random_rewards.append(simulate_random_policy(alpha1, beta1, alpha2, b

# Analyze the results
mean_greedy_reward = np.mean(greedy_rewards)
std_greedy_reward = np.std(greedy_rewards)
mean_random_reward = np.mean(random_rewards)
std_random_reward = np.std(random_rewards)
median_random_reward = np.median(random_rewards)

print(f"Greedy Policy - Mean Total Reward: {mean_greedy_reward}, Std Dev:
print(f"Random Policy - Mean Total Reward: {mean_random_reward}, Std Dev:
print(f"Random Policy - Median Total Reward: {median_random_reward}")

# Plot the distribution of rewards
plt.hist(greedy_rewards, bins=50, alpha=0.75, label='Greedy Policy')
plt.hist(random_rewards, bins=50, alpha=0.75, label='Random Policy')
plt.axvline(median_random_reward, color='r', linestyle='dashed', linewidt
plt.title("Distribution of Total Rewards")
plt.xlabel("Total Reward")
plt.ylabel("Frequency")
plt.legend()
plt.show()
```

```
Greedy Policy - Mean Total Reward: 6.182453165133446, Std Dev: 2.536737935
2641657
Random Policy - Mean Total Reward: 4.947482064611808, Std Dev: 2.340358950
0273387
Random Policy - Median Total Reward: 4.933902558328084
```


Distribution of Total Rewards
Greedy Policy
Random Policy
Random Policy Median
Frequency
Total Reward
0
10
20
30
40
50
0
2
4
6
8
10