Project 8

Qidi Han

April 28, 2020

Question 1

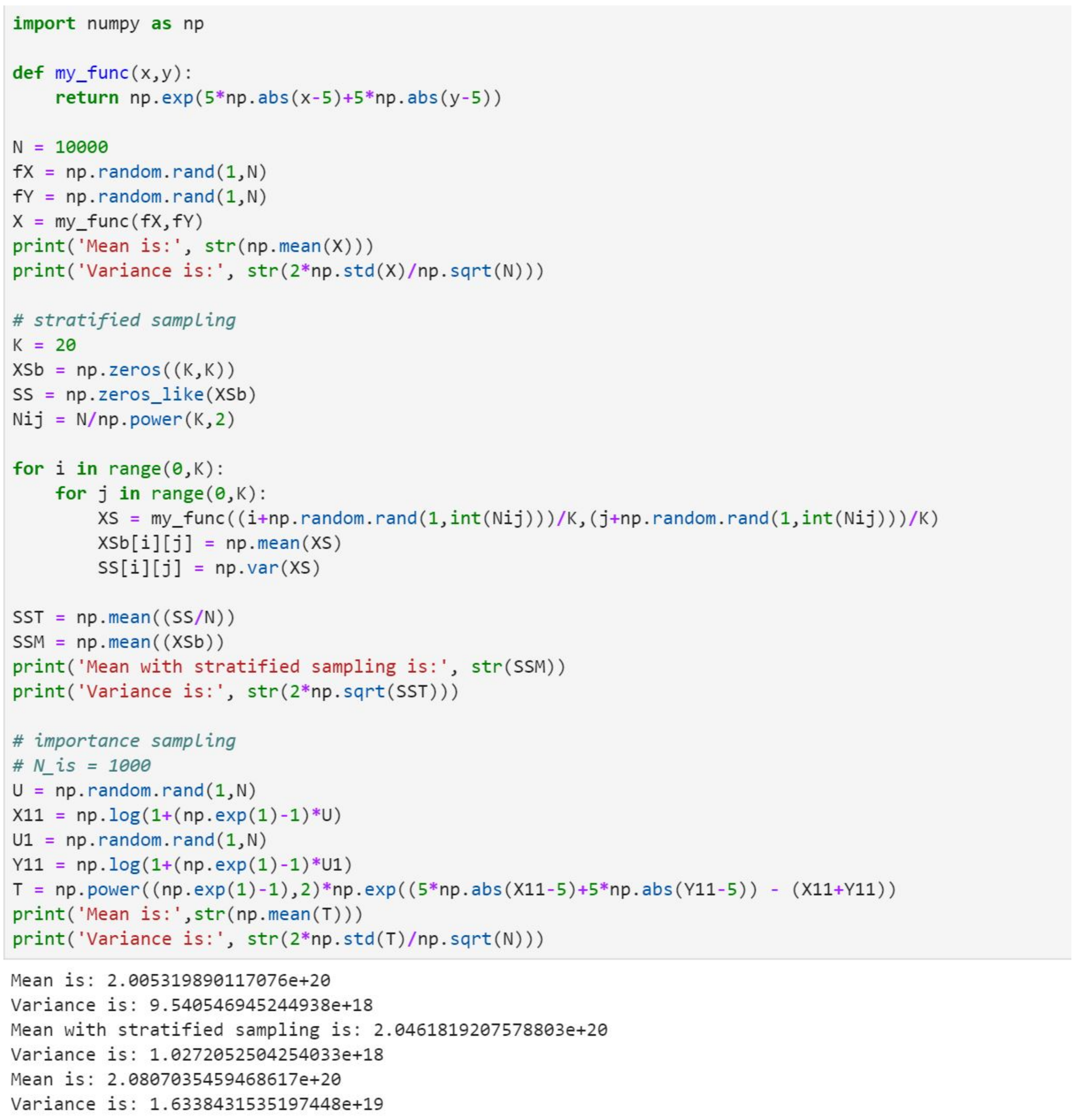
Use a total sample budget of n=1000 samples to obtain Monte Carlo estimates and variances for the following integrals in two dimensions (x1 and x2). Then implement stratification and importance sampling in your Monte Carlo estimation simulation using the same sample budget. Compare the three different Monte Carlo integral estimates, the quality of the estimates, and their sample variances.

Theory

In this question, there are three case for both equation,such as Monte Carlo estimates, stratified sampling and importance sampling. For the Monte Carlo, we will set the equation first, and generate two normal variable number, next fit this two number into the equation. For the Stratified sampling and importance sampling, I just use the formula that provide in the class. Then I calculate the mean and variance for each sampling.

Algorithm 1: Code for random vector X having multivariate Gaussian distribution

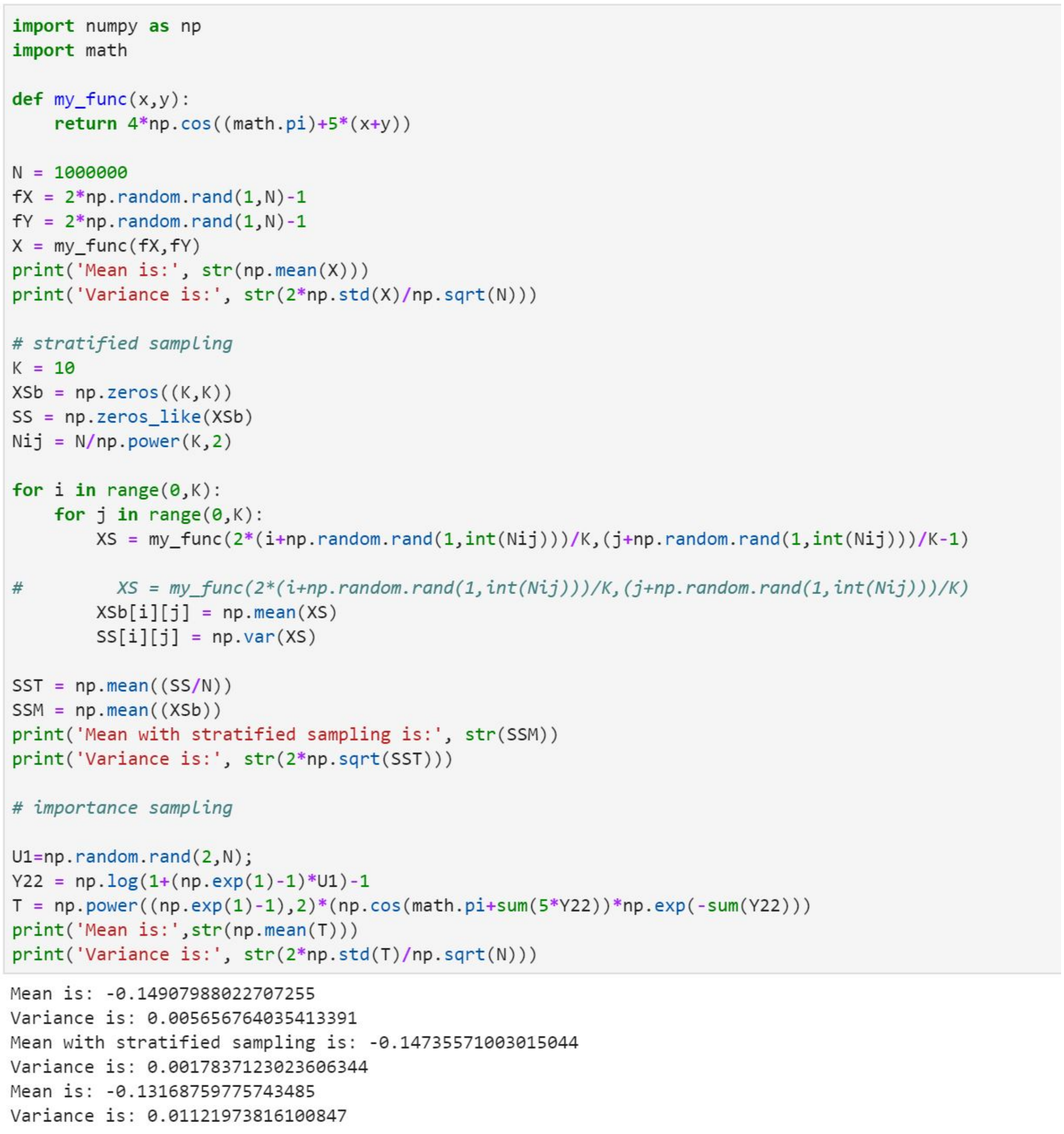
```
/* Formula for Monte Carlo */
fX = np.random.rand(1,N)
fY = np.random.rand(1,N)
X = my$_f unc(fX, fY)$/* Formula for stratified sampling */
XSb = np.zeros((K,K))
SS = np.zeros$_l ike(XSb)$
Nij = N/np.power(K,2)
for $i$ from 0 to 10000 do
    for $j$ from 0 to 10000 do
        XS = my$_f unc((i + np.random.rand(1, int(Nij)))/K, (j + np.random.rand(1, int(Nij)))/K)$
        XSb[i][j] = np.mean(XS)
        SS[i][j] = np.var(XS)
/* Formula for importance sampling */
U = np.random.rand(2,N$_i s$)
X$_i s = np.log(1 + (np.exp(1) - 1) * U)$
T = np.power((np.exp(1)-1),2)*np.exp((np.power(np.sum(X$_i s, axis = 0), 2)) - np.sum(X_i s, axis = 0))$

```



```python
import numpy as np

def my_func(x,y):
    return np.exp(5*np.abs(x-5)+5*np.abs(y-5))

N = 10000
fX = np.random.rand(1,N)
fY = np.random.rand(1,N)
X = my_func(fX,fY)
print('Mean is:', str(np.mean(X)))
print('Variance is:', str(2*np.std(X)/np.sqrt(N)))

# stratified sampling
K = 20
XSb = np.zeros((K,K))
SS = np.zeros_like(XSb)
Nij = N/np.power(K,2)

for i in range(0,K):
    for j in range(0,K):
        XS = my_func((i+np.random.rand(1,int(Nij)))/K,(j+np.random.rand(1,int(Nij)))/K)
        XSb[i][j] = np.mean(XS)
        SS[i][j] = np.var(XS)

SST = np.mean((SS/N))
SSM = np.mean((XSb))
print('Mean with stratified sampling is:', str(SSM))
print('Variance is:', str(2*np.sqrt(SST)))

# importance sampling
# N_is = 1000
U = np.random.rand(1,N)
X11 = np.log(1+(np.exp(1)-1)*U)
U1 = np.random.rand(1,N)
Y11 = np.log(1+(np.exp(1)-1)*U1)
T = np.power((np.exp(1)-1),2)*np.exp((5*np.abs(X11-5)+5*np.abs(Y11-5)) - (X11+Y11))
print('Mean is:',str(np.mean(T)))
print('Variance is:', str(2*np.std(T)/np.sqrt(N)))
```

```
Mean is: 2.005319890117076e+20
Variance is: 9.540546945244938e+18
Mean with stratified sampling is: 2.0461819207578803e+20
Variance is: 1.0272052504254033e+18
Mean is: 2.0807035459468617e+20
Variance is: 1.6338431535197448e+19
```

Figure 1: Code and result for part a



```
import numpy as np
import math

def my_func(x,y):
    return 4*np.cos((math.pi)+5*(x+y))

N = 1000000
fX = 2*np.random.rand(1,N)-1
fY = 2*np.random.rand(1,N)-1
X = my_func(fX,fY)
print('Mean is:', str(np.mean(X)))
print('Variance is:', str(2*np.std(X)/np.sqrt(N)))

# stratified sampling
K = 10
XSb = np.zeros((K,K))
SS = np.zeros_like(XSb)
Nij = N/np.power(K,2)

for i in range(0,K):
    for j in range(0,K):
        XS = my_func(2*(i+np.random.rand(1,int(Nij)))/K,(j+np.random.rand(1,int(Nij)))/K-1)

#         XS = my_func(2*(i+np.random.rand(1,int(Nij)))/K,(j+np.random.rand(1,int(Nij)))/K)
        XSb[i][j] = np.mean(XS)
        SS[i][j] = np.var(XS)

SST = np.mean((SS/N))
SSM = np.mean((XSb))
print('Mean with stratified sampling is:', str(SSM))
print('Variance is:', str(2*np.sqrt(SST)))

# importance sampling

U1=np.random.rand(2,N);
Y22 = np.log(1+(np.exp(1)-1)*U1)-1
T = np.power((np.exp(1)-1),2)*(np.cos(math.pi+sum(5*Y22))*np.exp(-sum(Y22)))
print('Mean is:',str(np.mean(T)))
print('Variance is:', str(2*np.std(T)/np.sqrt(N)))
```

```
Mean is: -0.14907988022707255
Variance is: 0.005656764035413391
Mean with stratified sampling is: -0.14735571003015044
Variance is: 0.0017837123023606344
Mean is: -0.13168759775743485
Variance is: 0.01121973816100847
```

Figure 2: Code and result for part b

Explanation

All the mean of part a is around 2 from the figure 1. If we do double integral, we will receive the answer also around 2. The mean of the part b is around -0.14 from the figure 2. The answer of the part b is also -0.14. Therefore, we use the formula correctly.

Question 2

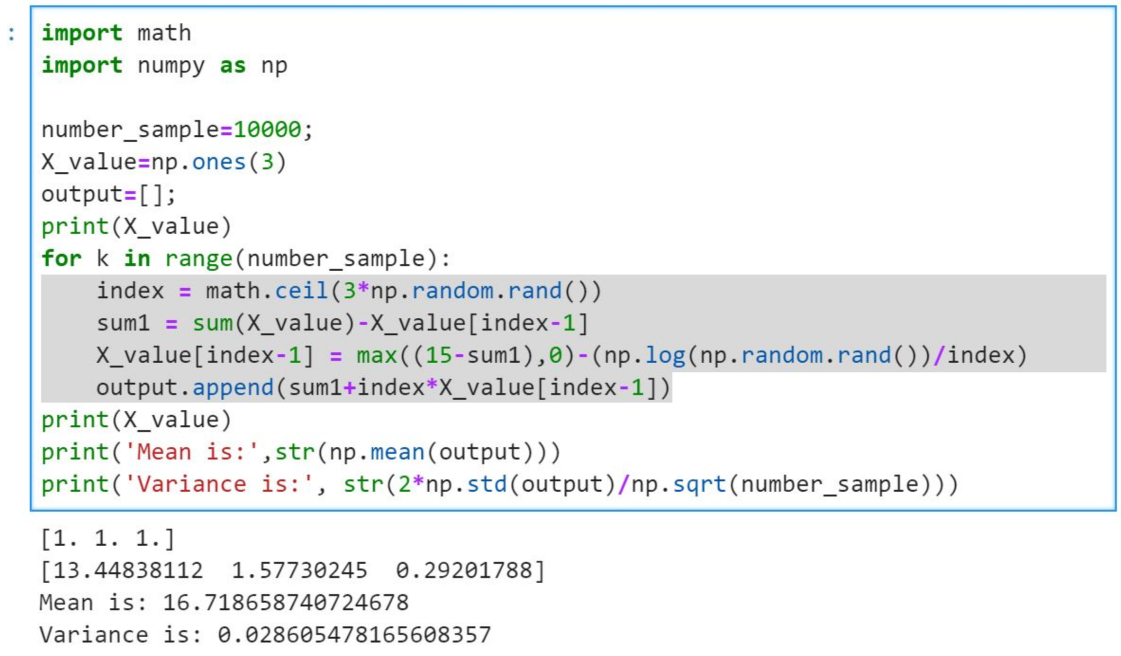
Let Xi,i=1,2,3 be independent exponential with mean 1. Use Gibbs sampling to estimate the part a and part b.

Theory

Metropolis-Hastings is a method with component by component update. There are basically three steps of our question. The first step is set the x1,x2 and x3 as 1, then random choose initial coordinate, we will use the sum to subtract the value in the choosing coordinate. The second step is using the equation x1+2x2>15-3*X2 by using the exponential distribution. Then, we will record this changing value. Finally, we will repeat the step 1 and 2 with 10000 times, and calculate the mean and variance for this changing data. For the part b, we just have to use minimum value to subtract the exponential distribution.

Algorithm 2: Second Question.

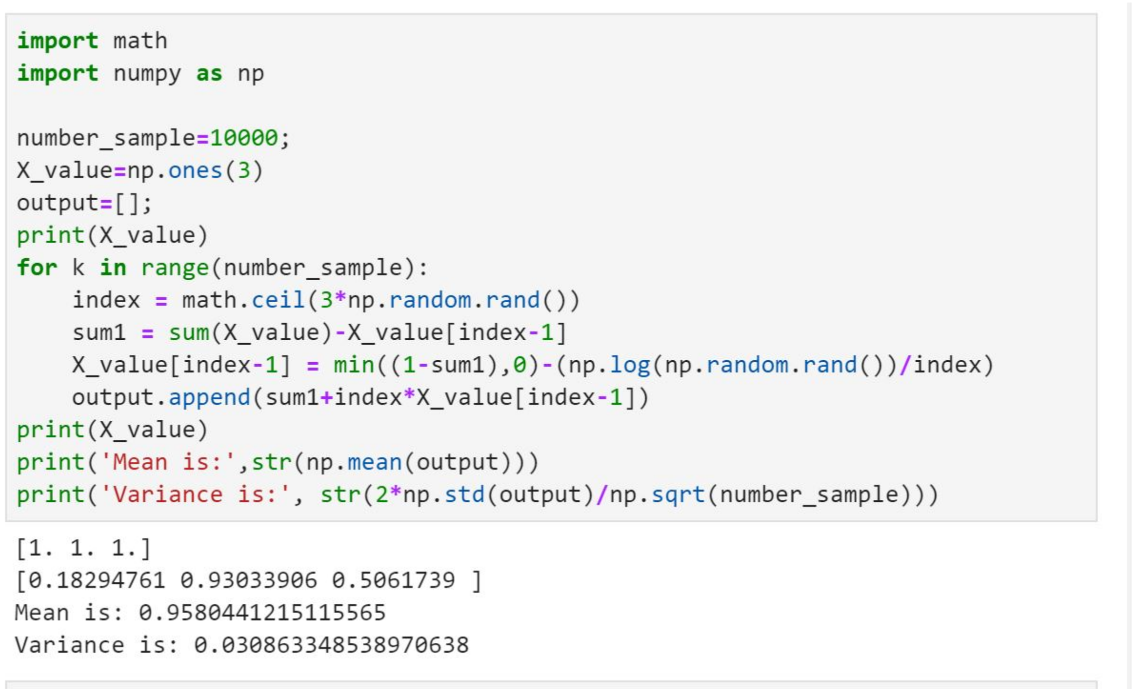
```
/* Code for Gibbs sampler */
for i from 0 to 10000 do
    index = math.ceil(3*np.random.rand())
    sum1 = sum(X_value) − X_value[index − 1]
    X_value[index − 1] = max((15 − sum1), 0) − (np.log(np.random.rand())/index)
    output.append(sum1+index*X_value[index − 1])
```



```
import math
import numpy as np

number_sample=10000;
X_value=np.ones(3)
output=[];
print(X_value)
for k in range(number_sample):
    index = math.ceil(3*np.random.rand())
    sum1 = sum(X_value)-X_value[index-1]
    X_value[index-1] = max((15-sum1),0)-(np.log(np.random.rand())/index)
    output.append(sum1+index*X_value[index-1])
print(X_value)
print('Mean is:',str(np.mean(output)))
print('Variance is:', str(2*np.std(output)/np.sqrt(number_sample)))
```

```
[1. 1. 1.]
[13.44838112  1.57730245  0.29201788]
Mean is: 16.718658740724678
Variance is: 0.028605478165608357
```

Figure 3: Code and result for part a



```
import math
import numpy as np

number_sample=10000;
X_value=np.ones(3)
output=[];
print(X_value)
for k in range(number_sample):
    index = math.ceil(3*np.random.rand())
    sum1 = sum(X_value)-X_value[index-1]
    X_value[index-1] = min((1-sum1),0)-(np.log(np.random.rand())/index)
    output.append(sum1+index*X_value[index-1])
print(X_value)
print('Mean is:',str(np.mean(output)))
print('Variance is:', str(2*np.std(output)/np.sqrt(number_sample)))
```

```
[1. 1. 1.]
[0.18294761 0.93033906 0.5061739 ]
Mean is: 0.9580441215115565
Variance is: 0.030863348538970638
```

Figure 4: Code and result for part b

Explanation

The expectation of part a is around 16.7, it higher than 15 about 1.7 in our question, but this is reasonable. The variance of this part is only 0.0286. Therefore, I think the result is correct. For the part b, the mean of each iteration is around 0.958, it's less than 1 by only 0.042. The variance of this result also very small. Therefore, the experiment is successful.

Question 3

The Schwefel function is a standard optimization benchmark because it has many local minimas and a single global minimum. The Schwefel function is given by 418.9829*2-X*np.sin(np.sqrt(abs(X)))-Y*np.sin(np.sqrt(abs(Y)))

Theory

In this question, I will plot the 2-dimensional schwefel surface. Then I will find the global minimum of the surface using the simulated annealing. the original data point is start (0,0). There are three different cooling schedules, such as exponential, polynomial and logarithmic. For each cooling schedules, I will run different iteration counts[50,200,1000,10000]. for the algorithm part, I will describe the steps for the question.

Algorithm 3: .

```
/* Step for Cooling Schedules */
1. Choose x0 w/ c(x0)>0 and initial temperature T0
2. Generate candidate X(t+1) by sampling from the jump distribution q(y|xt)
3. Compute Boltzman function =exp(c(x$_1^*(t+1), x_2^*(t+1)) - c(x_1(t), x_2(t)))/kT$
4. Accept X(t+1)(X(t+1)-X(t+1)) if C(x(t+1)) <=C(X(t))
5. Use different cooling schedules
6. Goto step 2
```

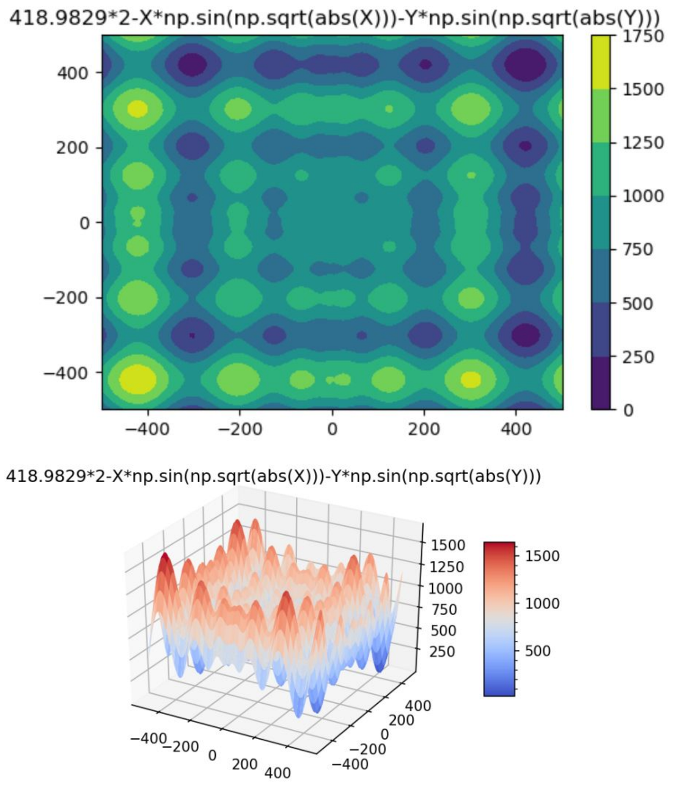


Figure 5: 2-dimensional Schwefel Surface

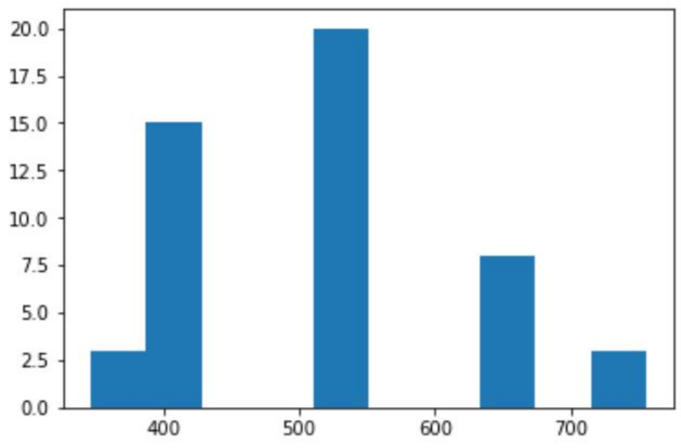


Figure 6: Histogram of Exponential Cooling Schedules when N=50

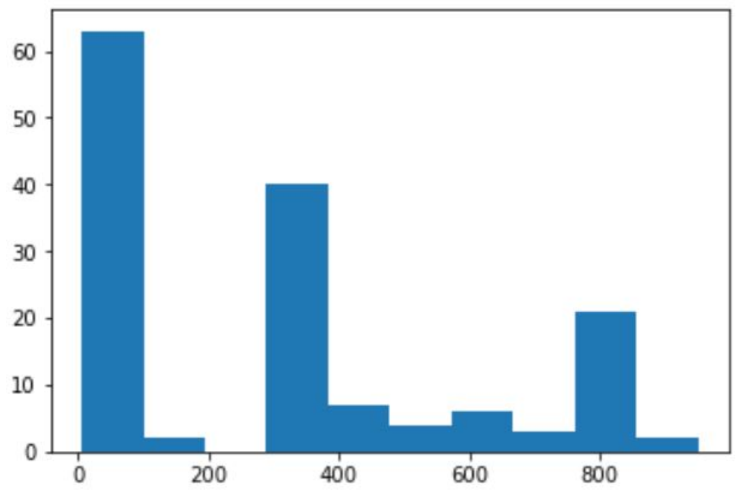


Figure 7: Histogram of Exponential Cooling Schedules when N=200

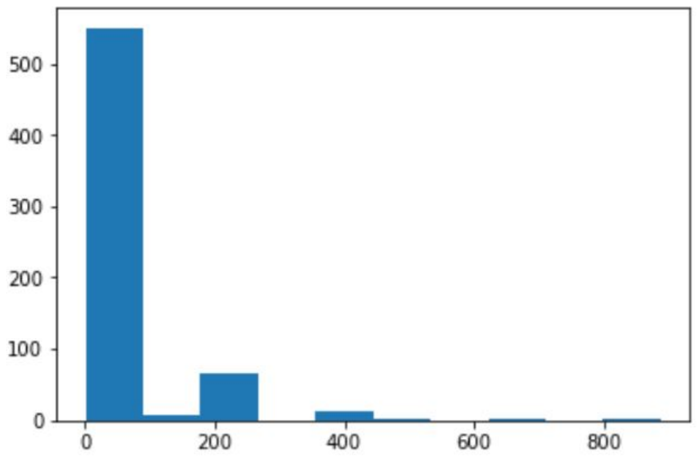


Figure 8: Histogram of Exponential Cooling Schedules when N=1000

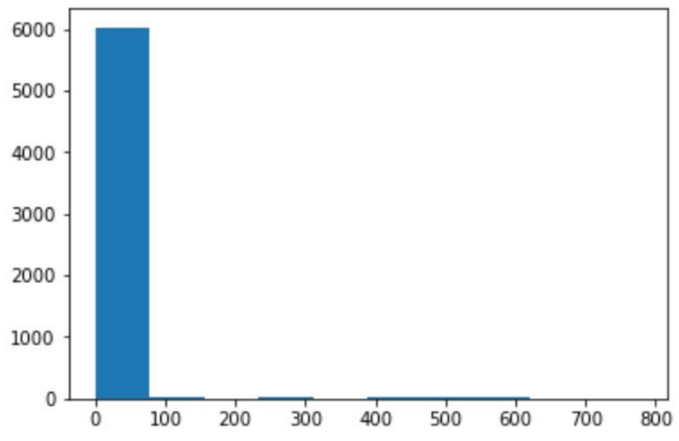


Figure 9: Histogram of Exponential Cooling Schedules when N=10000

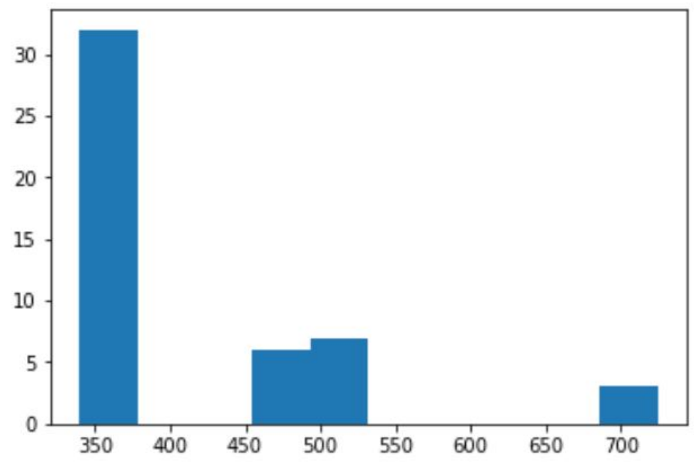


Figure 10: Histogram of logarithmic Cooling Schedules when N=50

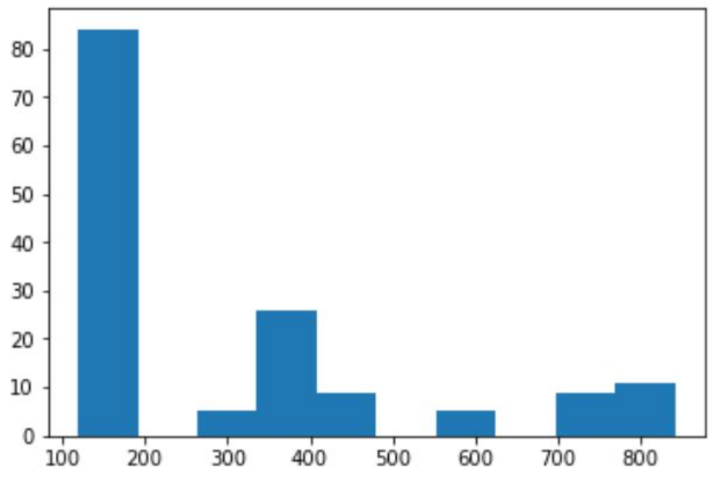


Figure 11: Histogram of logarithmic Cooling Schedules when N=200

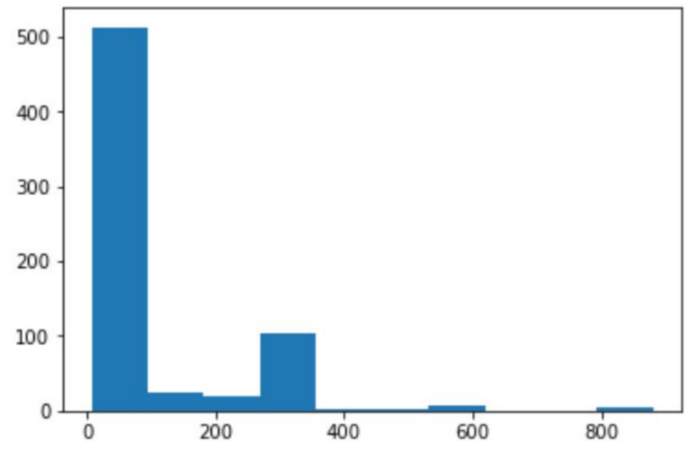


Figure 12: Histogram of logarithmic Cooling Schedules when N=1000

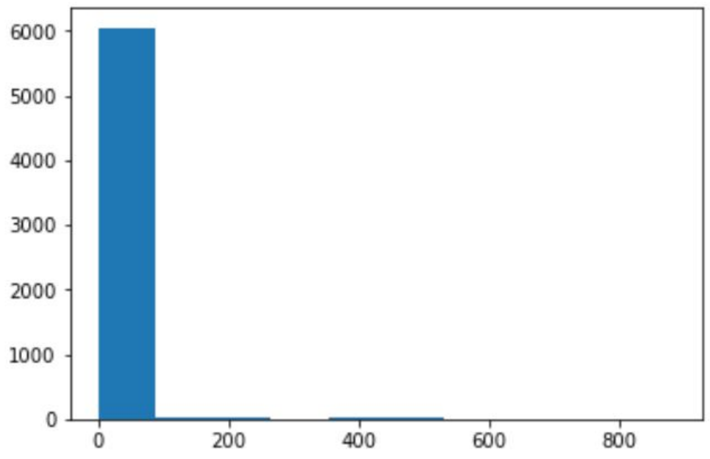


Figure 13: Histogram of logarithmic Cooling Schedules when N=10000

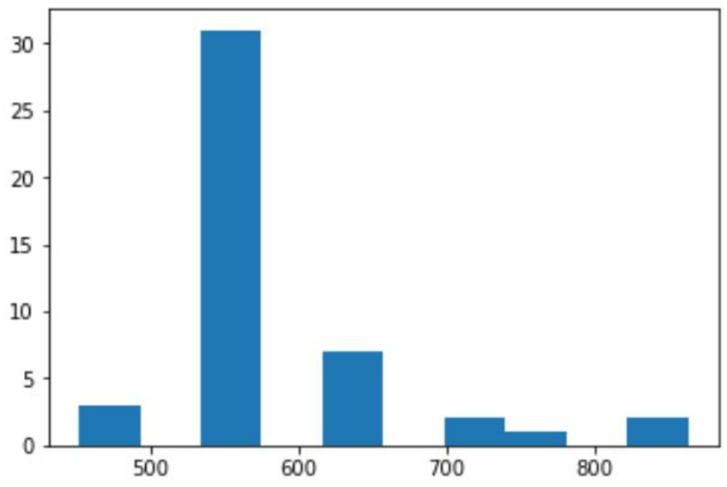


Figure 14: Histogram of Polynomial Cooling Schedules when N=50

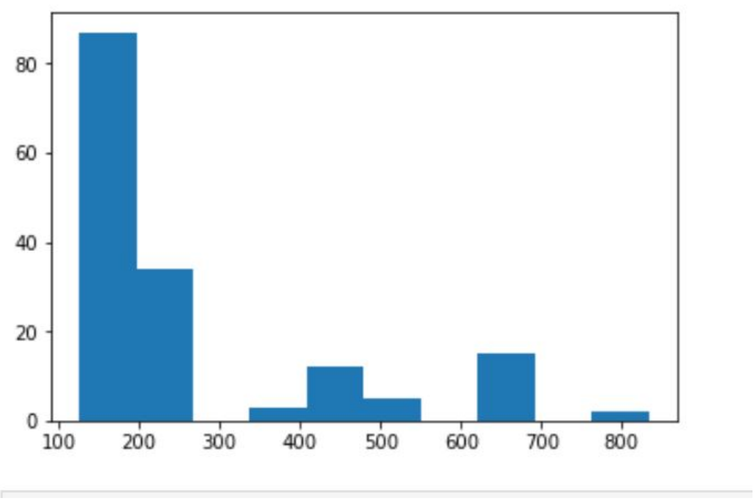


Figure 15: Histogram of Polynomial Cooling Schedules when N=200

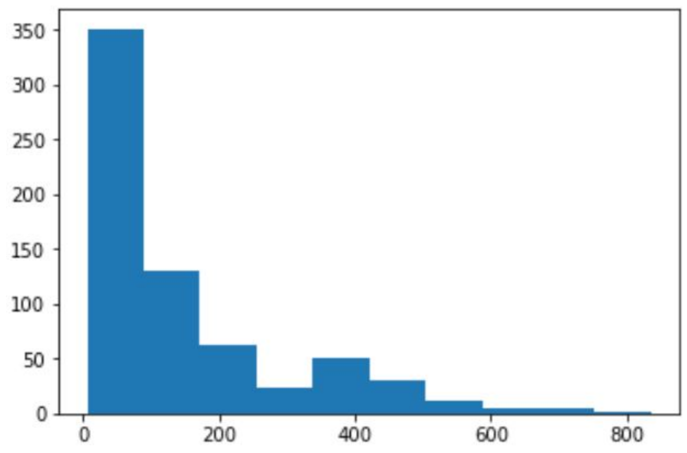


Figure 16: Histogram of Polynomial Cooling Schedules when N=1000

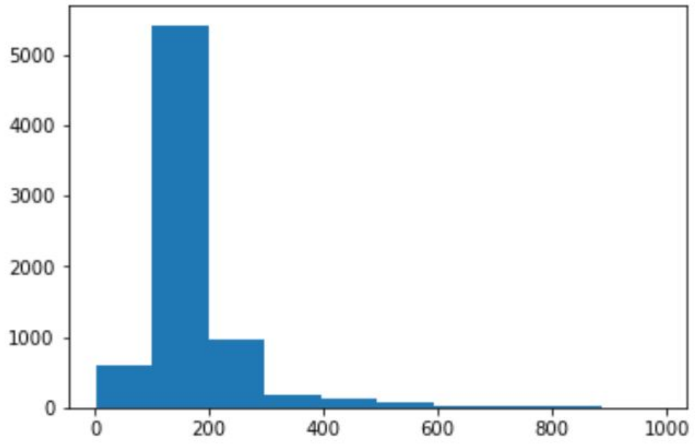
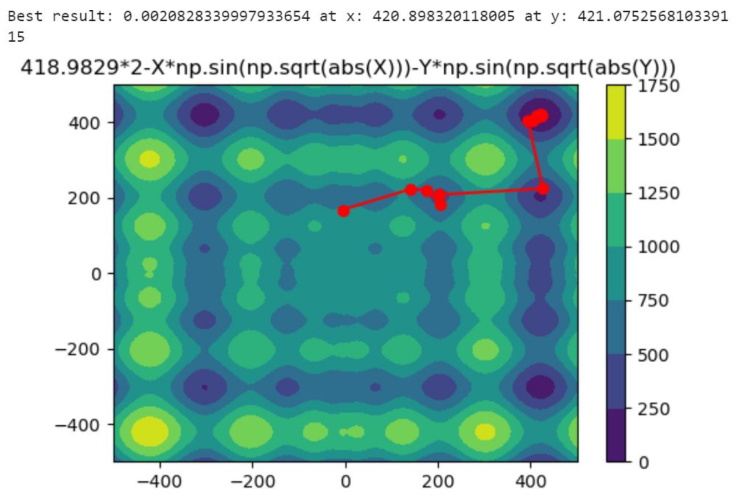
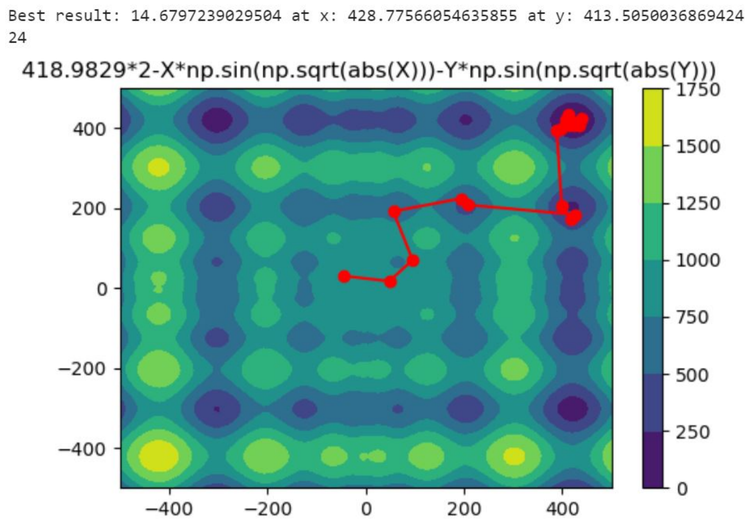


Figure 17: Histogram of Exponential Cooling Schedules when N=10000



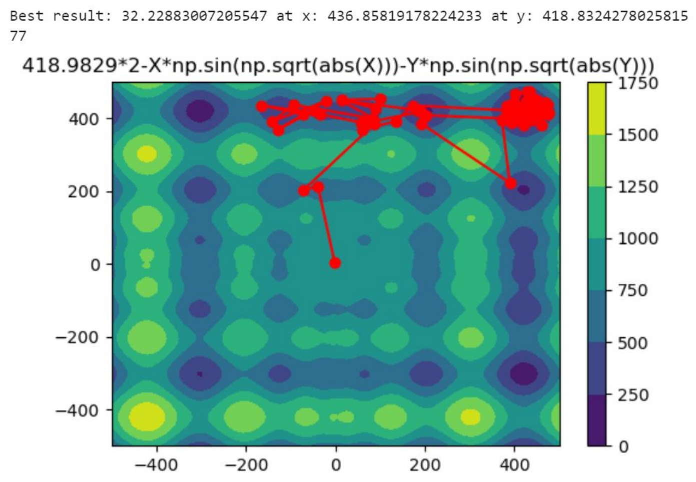
```
Best result: 0.0020828339997933654 at x: 420.898320118005 at y: 421.0752568103391
15
```

Figure 18: Best estimate of global minimum of Exponential Cooling Schedules



```
Best result: 14.6797239029504 at x: 428.77566054635855 at y: 413.5050036869424
24
```

Figure 19: Best estimate of global minimum of polynomial Cooling Schedules



```
Best result: 32.22883007205547 at x: 436.85819178224233 at y: 418.8324278025815
77
```

Figure 20: Best estimate of global minimum of logarithmic Cooling Schedules

Explanation

When, I determine the global minimum, I think the darkest area represent the global minimum. When the N is small, the minimum value of the distribution of each cooling schedules is looks like normal distribution, but when the N is becoming a large value, the distribution will more spread. From my perspective, I think it's because when the iterations become larger, the global value will come more obvious. Figure 18,19,20 shows the global minimum value is around (420,420). The logarithmic cooling schedules give us the longest path, its converge very slowly, even the global minimum is found. The exponential cooling schedules give us the short path. it's the fast one to reach the global minimum.

Question 4

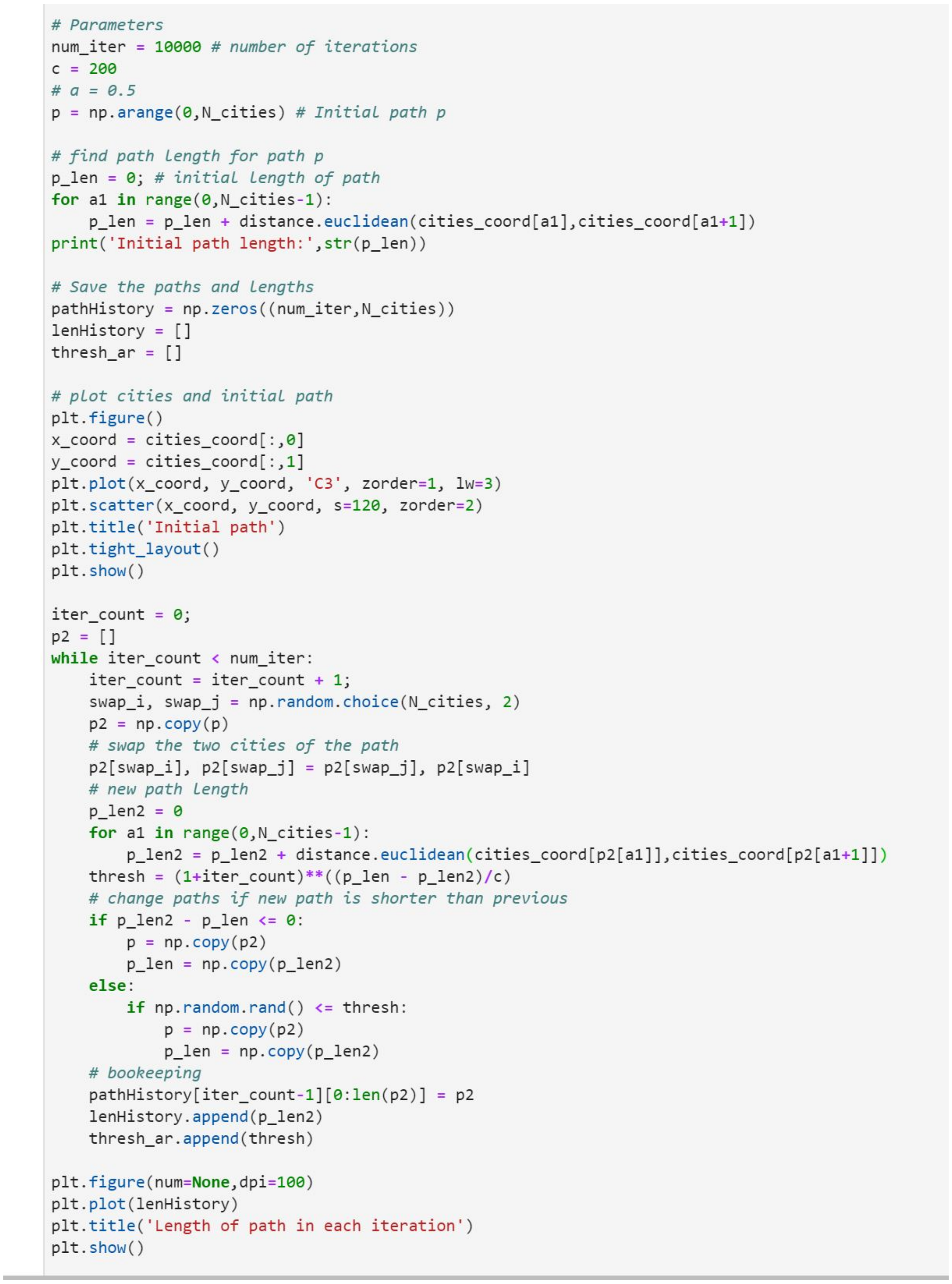
. You and a friend decide to take a summer road trip through every state capitol in the contiguous United States (the 48 states excluding Hawaii and Alaska) now that you have finished another tough year at USC. Refer to the addendum data files that contain names and (x,y) coordinates (these are based on cylindrical projection onto the plane) of all 50 US state capitals. Use a simulated annealing simulation to determine a minimal path between these 48 state capitals (removing Juneau, Alaska and Honolulu, Hawaii).

Theory

In this question, I will read the data from the txt file, then I will find the local for cities named Alaska and Hawaii. After the pre-process, we will have 48 coordinates. Then I will use euclidean distance to estimate the road distance between two cities. I will plot initial figures, convergence rate and the best road in x-y axis. I will use different iteration to run the program, to see the convergence rate in minimum iterations. I will discuss the step in next section, and I will discuss the result in the explanation part.

Algorithm 4: .

```
/* Step of experiment                                          */
1. Initialize the path and compute the cost function
2. Compute P(t+1) by using the Euclidean distance
3. Repeat the process until connect all the cities
```



```python
# Parameters
num_iter = 10000 # number of iterations
c = 200
# a = 0.5
p = np.arange(0,N_cities) # Initial path p

# find path length for path p
p_len = 0; # initial length of path
for a1 in range(0,N_cities-1):
    p_len = p_len + distance.euclidean(cities_coord[a1],cities_coord[a1+1])
print('Initial path length:',str(p_len))

# Save the paths and lengths
pathHistory = np.zeros((num_iter,N_cities))
lenHistory = []
thresh_ar = []

# plot cities and initial path
plt.figure()
x_coord = cities_coord[:,0]
y_coord = cities_coord[:,1]
plt.plot(x_coord, y_coord, 'C3', zorder=1, lw=3)
plt.scatter(x_coord, y_coord, s=120, zorder=2)
plt.title('Initial path')
plt.tight_layout()
plt.show()

iter_count = 0;
p2 = []
while iter_count < num_iter:
    iter_count = iter_count + 1;
    swap_i, swap_j = np.random.choice(N_cities, 2)
    p2 = np.copy(p)
    # swap the two cities of the path
    p2[swap_i], p2[swap_j] = p2[swap_j], p2[swap_i]
    # new path length
    p_len2 = 0
    for a1 in range(0,N_cities-1):
        p_len2 = p_len2 + distance.euclidean(cities_coord[p2[a1]],cities_coord[p2[a1+1]])
    thresh = (1+iter_count)**((p_len - p_len2)/c)
    # change paths if new path is shorter than previous
    if p_len2 - p_len <= 0:
        p = np.copy(p2)
        p_len = np.copy(p_len2)
    else:
        if np.random.rand() <= thresh:
            p = np.copy(p2)
            p_len = np.copy(p_len2)
    # bookeeping
    pathHistory[iter_count-1][0:len(p2)] = p2
    lenHistory.append(p_len2)
    thresh_ar.append(thresh)

plt.figure(num=None,dpi=100)
plt.plot(lenHistory)
plt.title('Length of path in each iteration')
plt.show()
```

Figure 21: Code for this question

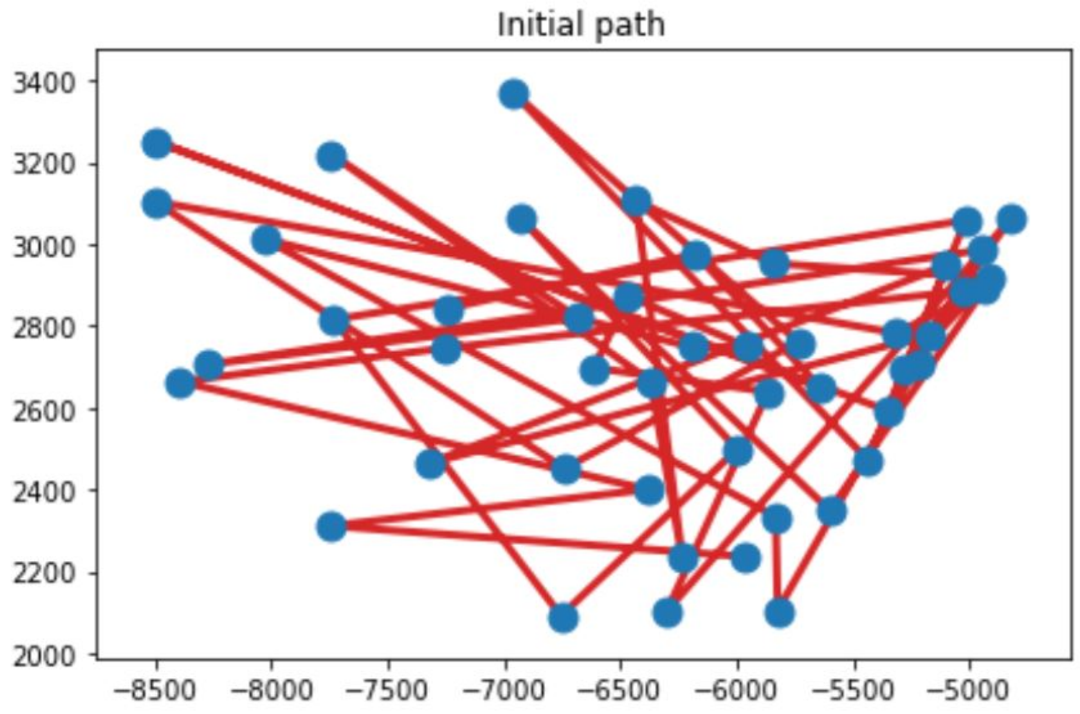


Figure 22: Initial path between each cities

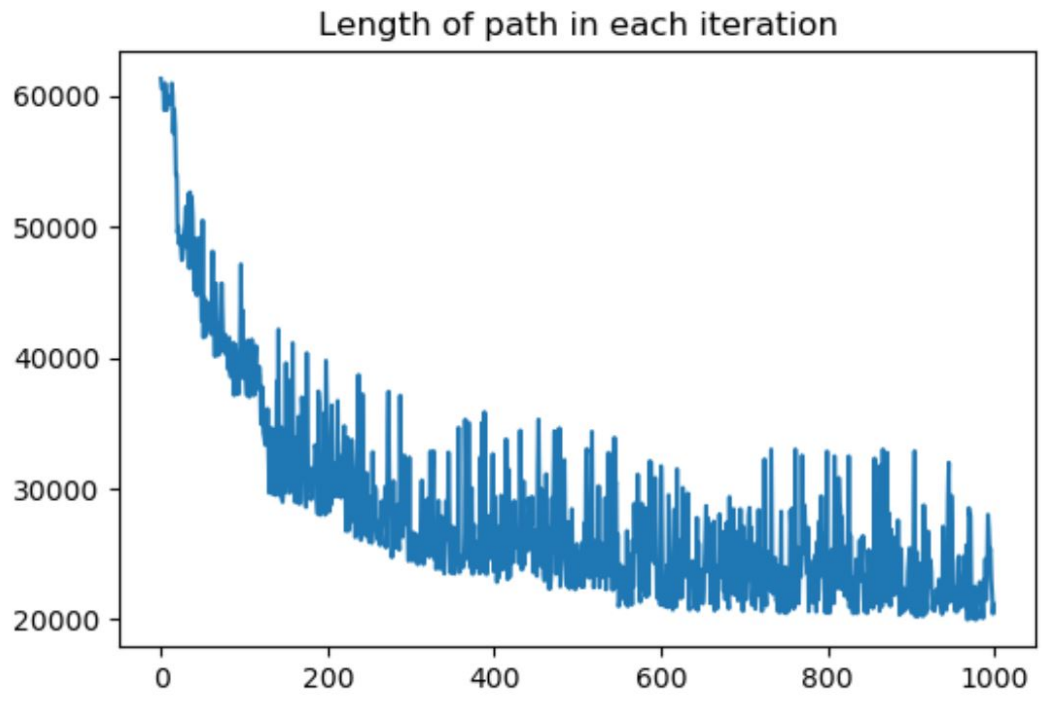


Figure 23: Iteration equals 1000

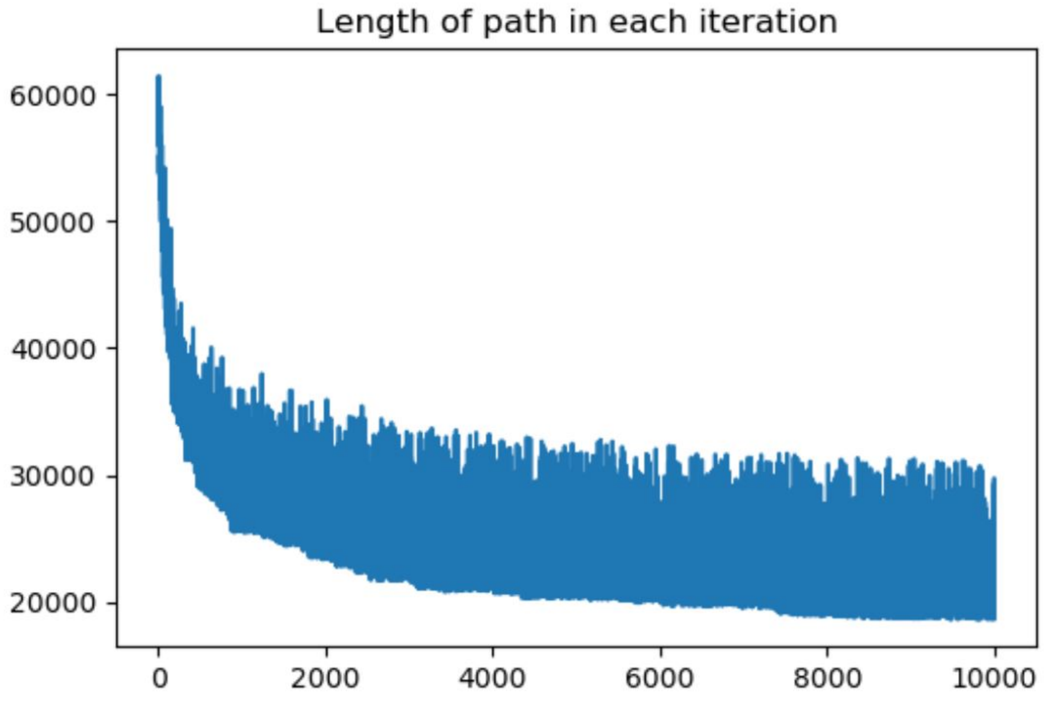


Figure 24: Iteration equals 10000

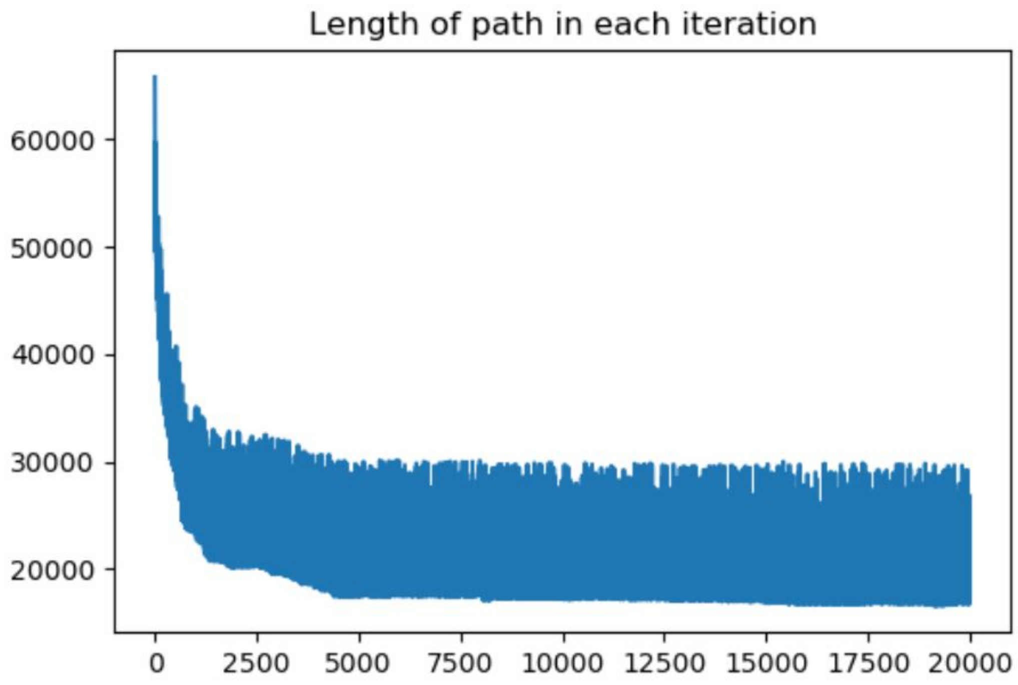


Figure 25: Iteration equals 20000

```
[ 3  4 33 40 39 35 17  5 36 18 29 19 11 46 12 13 24 38 44 34 47 25  1 28
 41  9 23 31 20 42 16 26 27  6 43 30 37 45 32 14  8  7  0 21 15  2 22 10]
```

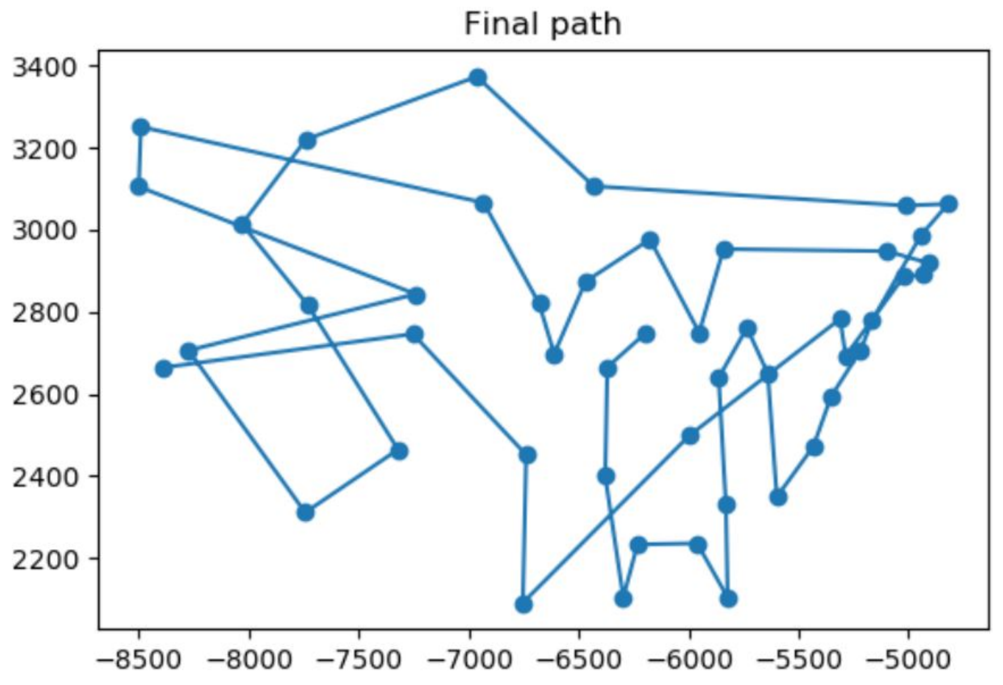


Figure 26: Final path with 10000 iteration

Explanation

For the 1000 iteration, it seems that the convergence still not happen at 1000 iteration. For the 10000, it seems that the convergence is happened at 6000 iterations. For the 20000, it seems that the convergence is happened at 6000 iterations. So, i think 6000 iteration will give us a good path. At this iteration, the length of the path will be around 18000. From Figure 26, it can see that we have a good answer for the best path from Sacramento California. The path shows us a very clear way between each cities. We can just follow the path.