EE511 Project 8

Name: Yijing Jiang

Email: yijingji@usc.edu

USC ID: 5761477714

Question 1

Explain:

- General Monte Carlo: comput the mean of all the samples as the estimation of integral.
- Stratification sampling:
  - divide the whole range into k regions
  - in each region, generate some samples and compute their mean and var
  - combine all the mean and var with conditional probability as the final mean and var.
- Importance sampling: find a helper function similar to the given function, and sample with this function to find the mean and var

Result:

- **For Part a:**
  - **Stratified method vs i.i.d.:**
    - Stratified method has similar mean with i.i.d. method, and both are similar to the real value of integral.
    - Stratified method has far more smaller variance than i.i.d. methos, which means that stratified method has a better concentration and its results will more around the mean.
  - **Importance method vs i.i.d. :**
    - estimate quality of importance method is worse than i.i.d, and will float in a large region, which may have relation with the choice of helper function
    - variance of these two method is similar, because we still compute the mean of all the samples

	mean	var
Actual	2.0460289486955774e+20	
Stratified	2.0377577756696193e+20	5.383600940663645e+18
i.i.d	2.046865009978822e+20	2.9136408259908362e+19
Importance	1.6127482080596845e+19	3.078581743789789e+19
i.i.d	1.9871437010370724e+20	2.8370162223680737e+19

- **For Part b:**
  - **Stratified method vs i.i.d.:**
    - Stratified method has better estimation on the integral than i.i.d. method, and both are similar to the real value of integral.
    - Stratified method still has far more smaller variance than i.i.d. methos, which means that stratified method has a better concentration and its results will more around the mean. This difference is related to and proportional to the size of each budget. With a larger budget, the difference will be smaller; with a smaller budget, the variance will samller, thus the difference between stratified and i.i.d. will be larger.
  - **Importance method vs i.i.d. :**
    - estimate quality of importance method is similar to i.i.d, and will float in a large region.
    - variance of these two method is also similar

	estimation = mean*2	var
Actual	-0.1471257223261162	
Stratified	-0.15091256428863392	0.08661534053406464
i.i.d	-0.1007305781370157	0.44380461364581697
Importance	-0.11417785639703303	0.014039379643824722
i.i.d	-0.14523433889385984	0.04409203311897025

Code 1a :

```
import numpy as np
from scipy.integrate import dblquad

#(a)
#standard integral value

area = dblquad(lambda x, y: np.exp(5*np.abs(x-5)+5*np.abs(y-5
)), 0, 1, 0, 1)
target = area[0]
print("Actual result is: ",str(target))

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

#estimate by stratified sampling
N = 1000
K = 10
XSb = np.zeros((K,K))
SS = np.zeros_like(XSb)
Nij = N/np.power(K,2)
PijX = []
```

```
PijY = []

for i in range(0,K):
    for j in range(0,K):
        PijXt = np.random.rand(1,int(Nij))
        PijYt = np.random.rand(1,int(Nij))
        PijX.append(PijXt)
        PijY.append(PijYt)
        XS = my_func((i+PijXt)/K,(j+PijYt)/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(2*np.sqrt(SST))

PijX = np.reshape(np.array(PijX),(N,))
PijY = np.reshape(np.array(PijY),(N,))
X = my_func(PijX,PijY)
print('Mean with i.i.d. is:', str(np.mean(X)))
print(2*np.std(X)/np.sqrt(N))

# importance sampling
N_is = 1000
U = np.random.rand(2,N_is)
X_is = (1/10)*np.log(U*(np.exp(10)-1)+1)
T = my_func(X_is[0],X_is[1])/(10*np.exp(10*X_is)/
(np.exp(10)-1))
print('Mean is:',str(np.mean(T)))
print(2*np.std(T)/np.sqrt(N_is))

X = my_func(U[0],U[1])
print('Mean with i.i.d. is:', str(np.mean(X)))
print(2*np.std(X)/np.sqrt(N_is))
```

Code 1b :

```
import numpy as np
from scipy.integrate import dblquad


#(b)
#standard integral value
area = dblquad(lambda x, y: np.cos(np.pi+5*x
+5*y), -1, 1, -1, 1)
target = area[0]
print("Actual result is: ",str(target))

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

```
13
14 #estimate by i.i.d
15 N = 1000
16 K = 10
17 XSb = np.zeros((K,K))
18 SS = np.zeros_like(XSb)
19 Nij = N/np.power(K,2)
20 PijX = []
21 PijY = []
22
23 for i in range(0,K):
24     for j in range(0,K):
25         PijXt = np.random.rand(1,int(Nij))
26         PijYt = np.random.rand(1,int(Nij))
27         PijX.append(PijXt)
28         PijY.append(PijYt)
29         XS = my_func((i+PijXt)/K,(j+PijYt)/K)*10
30         XSb[i][j] = np.mean(XS)
31         SS[i][j] = np.var(XS)
32
33 SST = np.mean((SS/N))
34 SSM = np.mean((XSb))
35 print('Mean with stratified sampling is:', str(SSM))
36 print('estimation with stratified is:', str(2*SSM))
37 print(2*np.sqrt(SST))
38
39 PijX = np.reshape(np.array(PijX),(N,))
40 PijY = np.reshape(np.array(PijY),(N,))
41 X = my_func(PijX,PijY)*10
42 print('Mean with i.i.d. is:', str(np.mean(X)))
43 print('estimation with i.i.d is:', str(2*np.mean(X)))
44 print(2*np.std(X)/np.sqrt(N))
45
46 # importance sampling
47 N_is = 1000
48 U = np.random.rand(2,N_is)*2-1
49 X_is = np.log(2+(np.exp(1)-1)*U)
50 T = ((np.power((np.exp(1)-1),2)*my_func(X_is[0],X_is[1]) -
    np.sum(X_is,axis=0)))/10
51 print('Mean is:',str(np.mean(T)))
52 print('estimation with importance is:', str(2*np.mean(X)))
53 print(2*np.std(T)/np.sqrt(N_is))
54
55 X = my_func(U[0],U[1])
56 print('Mean with i.i.d. is:', str(np.mean(X)))
57 print('estimation with i.i.d is:', str(2*np.mean(X)))
58 print(2*np.std(X)/np.sqrt(N_is))
```

Question 2a

Explain:

- Gibbs sampling can be used to estimate joint distribution, the general step is:
  - To estimate the joint distribution of X(x1, x2, …): f(x1, x2,…)
  - Begin with some initial sample X(i)
  - generate Xj(i+1), which is jth element in X(j+1), based on current rest samples by using conditional probability:
  - update X(i)
- For part a, estimate E[X1+2X2+3X3 | X1+2X2+3X3>15], we always need to satisfy the second condition when choosing the initial states and updating the samples:
  - Set the initial point as X(0) = [5,5,5] to meet the condition, compute sum = X1(0)+2X2(0)+3X3(0)
  - Randomly choose one variable Xi
  - New sample Xi(t+1) needs to satisfy the condition, thus needs to be greater than some number th.
  - Generate Xi(t+1) from distribution f(xi|xi>th). We can compute that the pdf and cdf
  - Generate u from U[0,1]
  - Xi(t+1) = th - ln(1-u)
  - update samples
  - compute the mean of samples

Results

- **Estimate 10 times:**

18.076373927701574

17.97949061923062

17.986445718744946

18.02036085666643

18.060713384044504

17.995198527609837

17.952857681298543

17.96384178813549

18.00268537548646

17.995960499808746

Code:

```
import numpy as np
import random


for t in range(10):
    xt = [5,5,5]
    sum = []

    for i in range(10000):
        #current sum
        sumt = xt[0] + 2*xt[1] + 3*xt[2]
        sum.append(sumt)
        #randomly choose one
        r = random.randint(0, 2)
        th = 15
        for j in range(3):
            if (j != r) :
                th = th - (j+1) * xt[j]
        th = th/(r+1)
        u = np.random.uniform(0,1)
        xt[r] = th - np.log(1-u)
    print(np.mean(sum))
```

Question 2b

Explain:

- For part b, estimate E[X1+2X2+3X3 | X1+2X2+3X3<1], we always need to satisfy the second condition when choosing the initial states and updating the samples:
  - Set the initial point as X(0) = [0,0,0] to meet the condition, compute sum = X1(0)+2X2(0)+3X3(0)
  - Randomly choose one variable xi
  - New sample Xi(t+1) needs to satisfy the condition, thus needs to be smaller than some number th.
  - Generate Xi(t+1) from distribution f(xi|xi<th). We can compute that pdf and cdf.
  - Generate u from U[0,1]
  - Xi(t+1) = F^(-1)
  - update samples
  - compute the mean of samples

Results

- **Estimate 10 times:**

0.7305861337071193

0.7318951323022207

0.7219646575946972

0.7279717357993816

0.726729003842508

0.728599721790243

0.7298903139473145

0.7234092618008163

0.7262838270063867

0.7212395654471767

Code:

```
1   import numpy as np
2   import random
3
4
5   for t in range(10):
6       xt = [0,0,0]
7       sum = []
8
9       for i in range(10000):
10          #current sum
11          sumt = xt[0] + 2*xt[1] + 3*xt[2]
12          sum.append(sumt)
13          #randomly choose one
14          r = random.randint(0, 2)
15          th = 1
16          for j in range(3):
17              if (j != r) :
18                  th = th - (j+1) * xt[j]
19          th = th/(r+1)
20          u = np.random.uniform(0,1)
21          xt[r] = -np.log(1-u*(1-np.exp(-th)))
22      print(np.mean(sum))
```

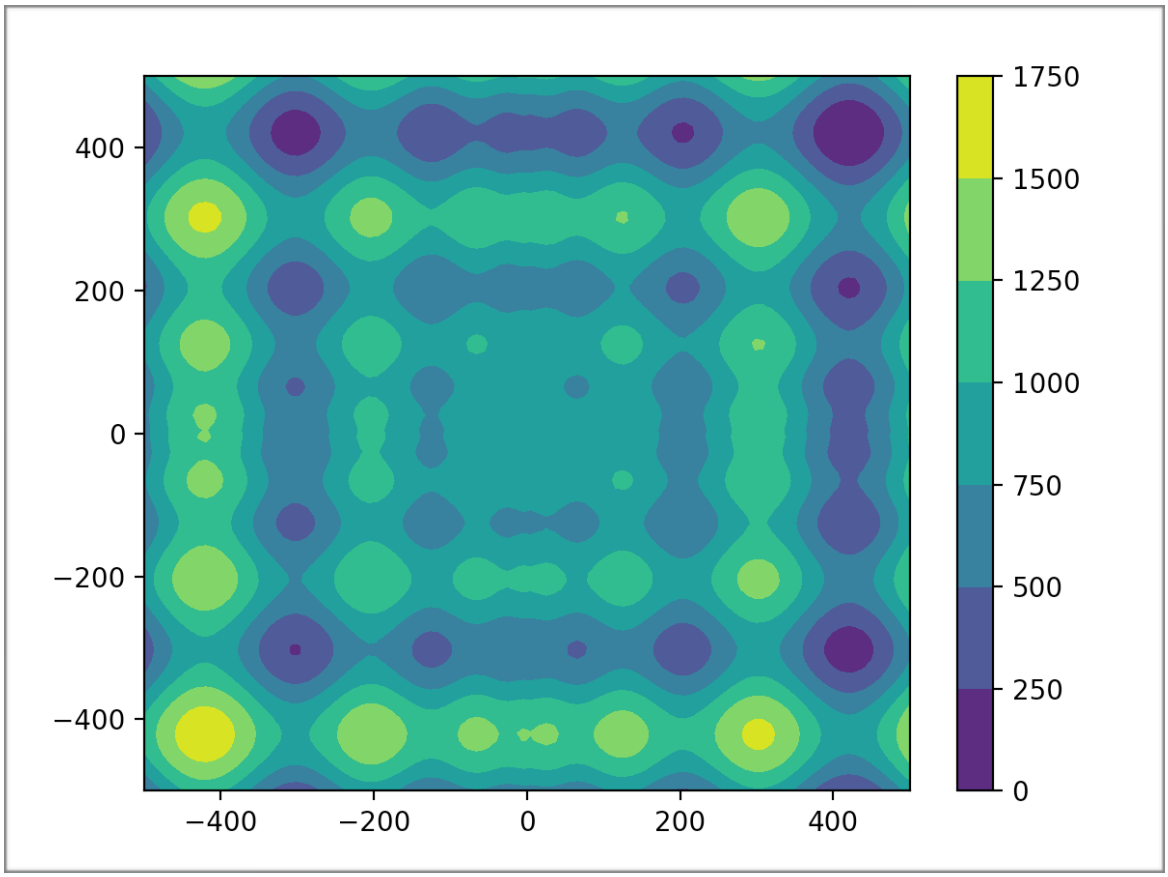
Question 3

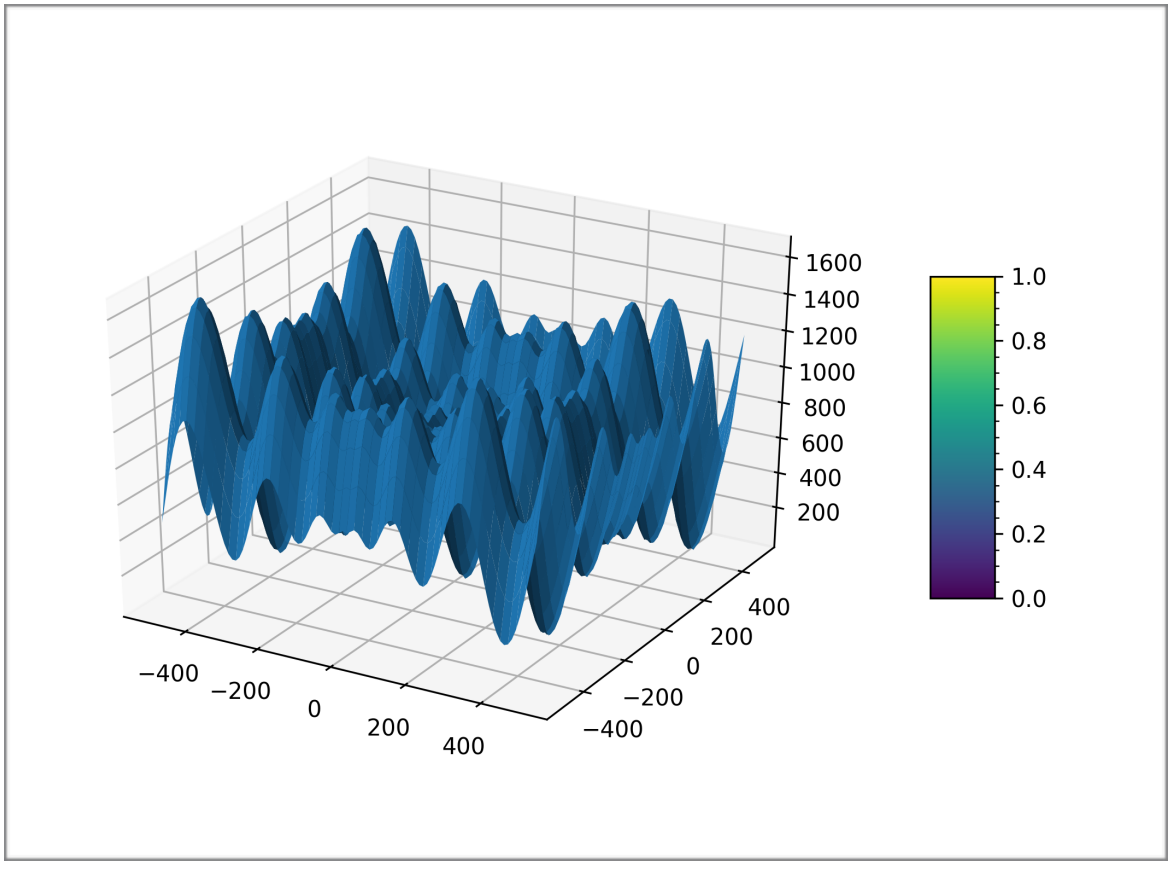
Explain:

- **Simulated Annealing can approximate the global minimum of a given function:**
  - Initial:
    - choose initial X0 (0,0)
    - choose initial **T0**
  - Choose X' from Gaussian around Xt (last sample):
    - Generate X' ~ N(Xt, **sigma**) by **"jump distribution"**
  - Accept or reject X':
    - compute alpha = exp(f(Xt)-f(X')/**Tt**)
    - if f(X') < f(Xt): accept
    - else accept with probability alpha:
      - generate u form U[0,1]
      - if u < alpha, accept; else, reject
  - Update Tt to T(t+1) by **"cooling schedule"**
  - Repeate above steps for N = 50, 200, 1000, 10000 times
- **Cooling schedule:**
  - decrease:
    - log: T(t) ~ 1/log(t+1)
  - increase:
    - polynomial: T(t) ~ (t+1)^2
    - exp: T(t) ~ exp(|t|^0.5)
- **Finding process will be affected by parameters: T0, sigma, and the cooling schedule.**

Results:

- **Draw contour plot of 2D Schwefel surface:**

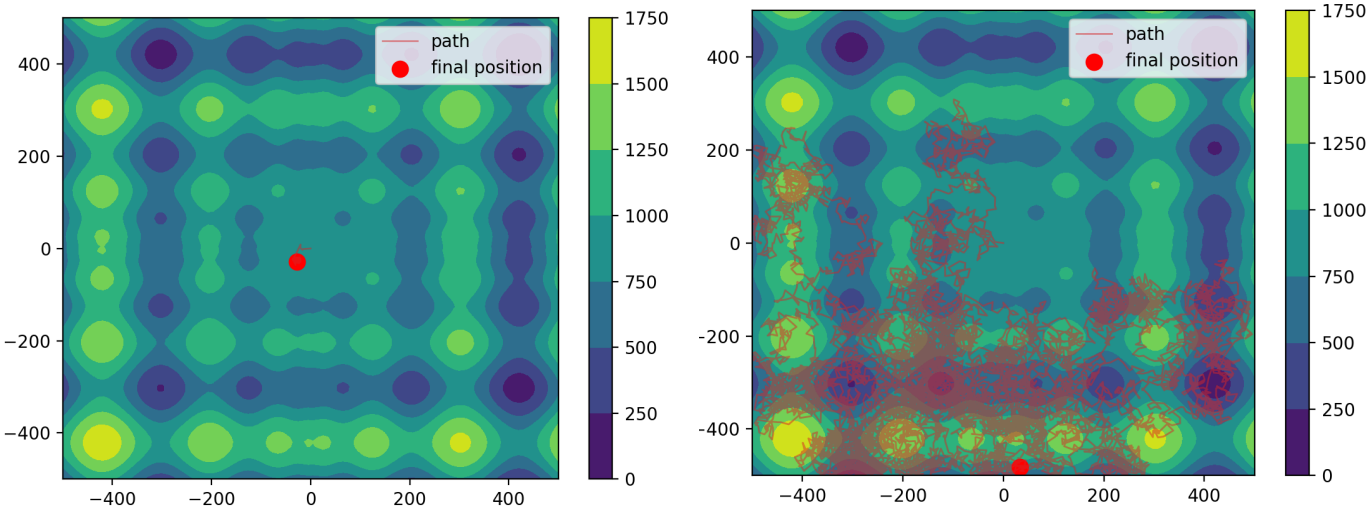




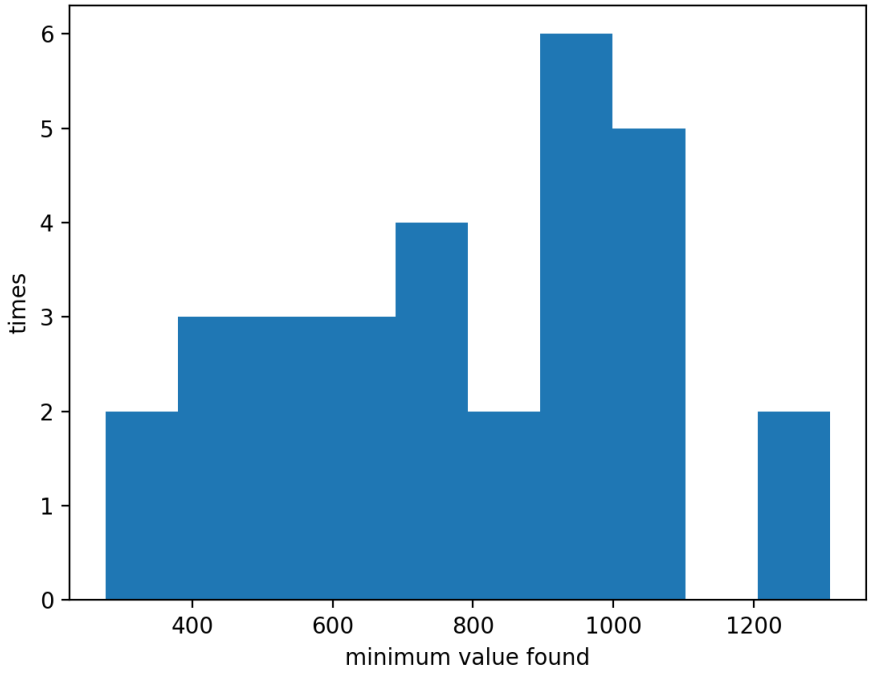
- **Find general results by using different cooling schedule and with different iteration:**

cooling schedule	N = 50	N = 200	N = 1000	N = 10000
log	910.0570	853.9814	809.3436	127.0168
poly	844.8050	1041.7045	458.3127	287.0942
exp	846.2282	861.2941	485.1856	276.6301

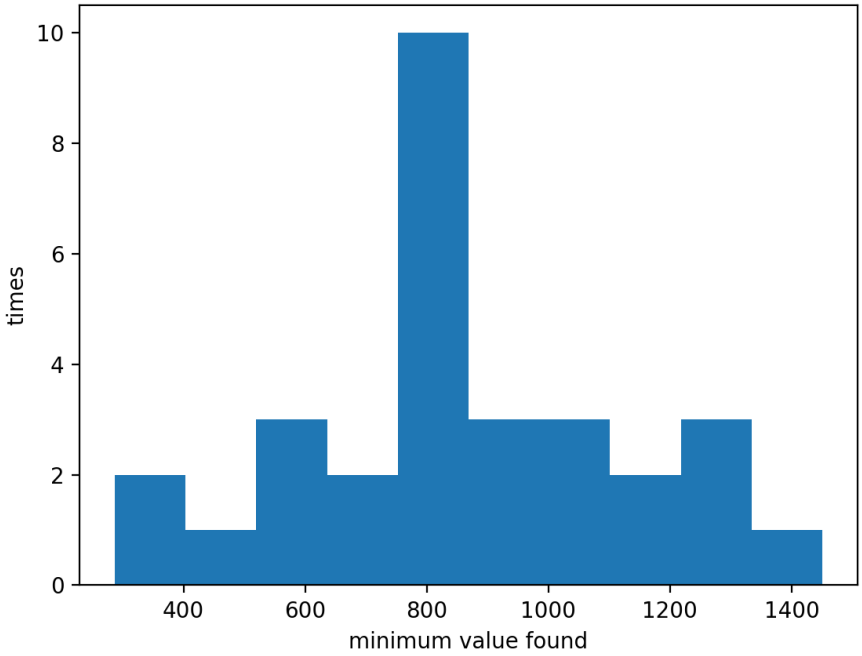
- Iteration affects the number of steps the program takes to explore the image. If iteration is small(left: log with 50 iteration), only several points are taken. Otherwise(right: log with 10000 iteration without adjusting proper parameters), many places are explored.



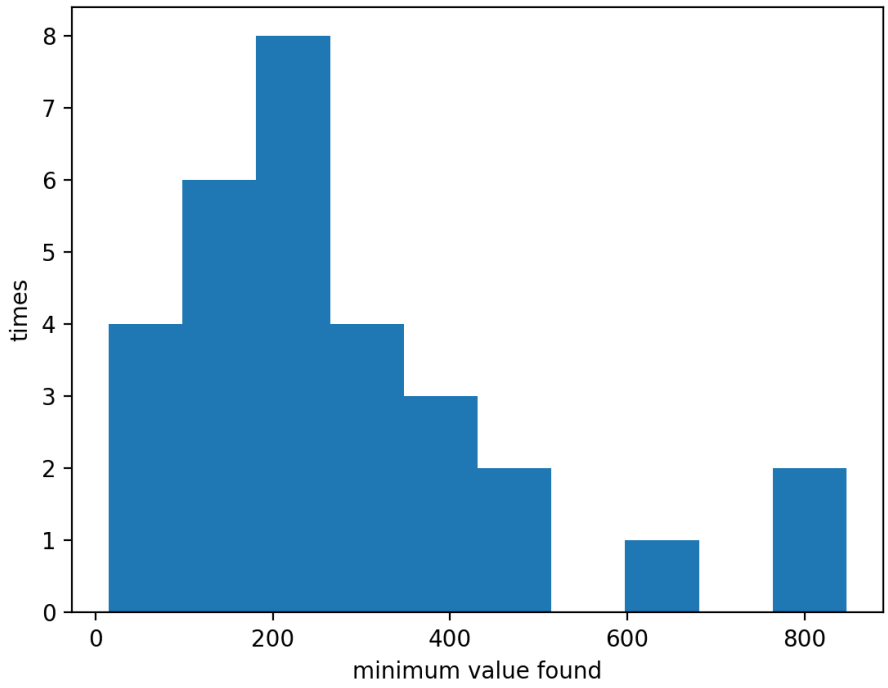
- **Plot histogram of different cooling schedule when N = 10000 for 30 times:**
  - **exp with T0 = 100, sigma = 10, Tt ~ exp(|t|^0.5)**
    - **minimum value in 30 times = 276.630**



- **polynomial with T0 = 100, sigma = 10, Tt ~ (t+1)^2**
  - **minimum value in 30 times = 287.0942**



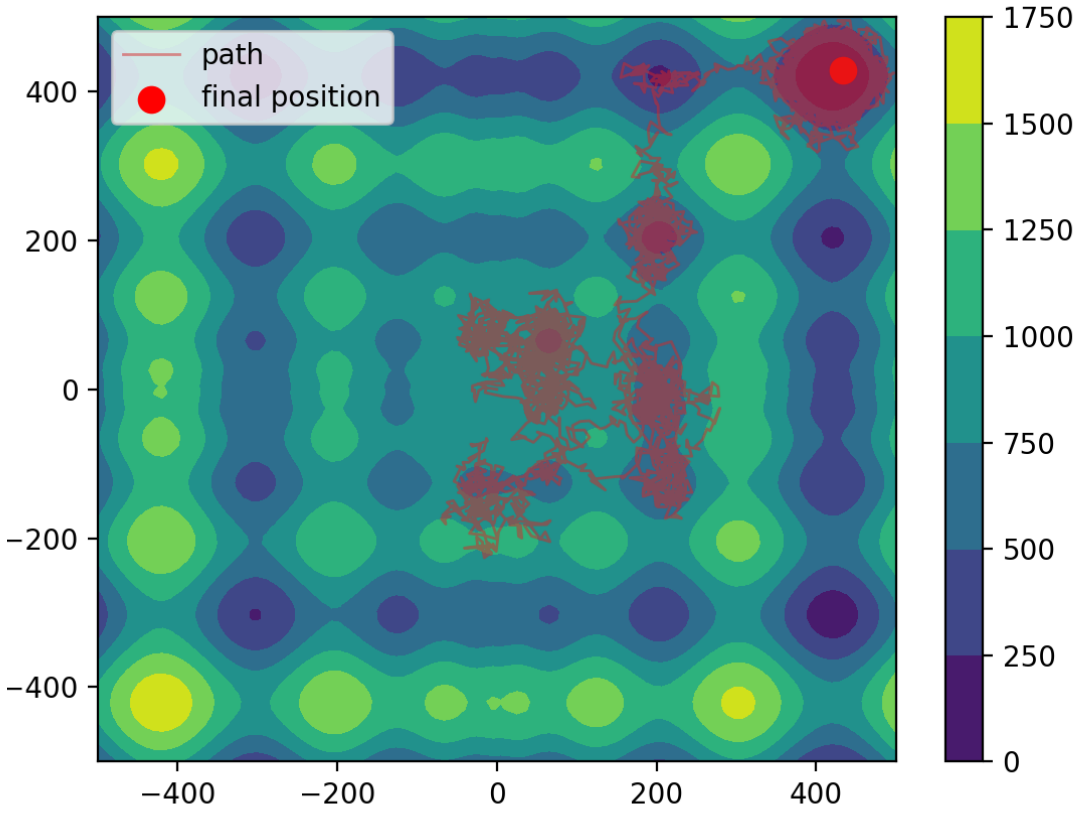
- **polynomial with T0 = 100, sigma = 10, Tt ~ 1/log(t+1)**
  - **minimum value in 30 times = 14.9686**
  - **somtimes will also have very bad results**



- Here, log cooling schedule has a better result in finding the global minimum.

- For the special of our target image, there are many global minimum. For the rest two cooling schedule with my parameters here, they always can not quickly escape from a local minimum or can not search deep for the global minimum.

- **Best result of log cooling schedule with T0 = 100, sigma = 10, N = 10000:**
  - **minimum value = 14.9686**
  - **For the search of minimum value in a image, there are two aspects:**
    - First is **exploring**, which means search for as many as places in the image.
    - Second is **exploiting**, which means digging in one place and find more information in one place.
  - **From searching path below, we can find that for this best finding:**
    - it will spend several iteration in local minimum, but will not stay for too long time and can leave the local minimum quickly.
    - But when it encountered with global minimum, it can stay here and spend far more time to expoit it, thus can find the global minimum.



- **During the progress of finding the best parameters, we can find how the T0 and sigma will affect the final path and the ability of above two aspects:**

- **For T0:**
  - too small T0 will lead to a small alpha, thus most of u generated from U[0,1] will be rejected, which means little points will be accepted. So our searching will be very slow.
  - too large T0 will lead to a big alpha, thus most of u will be accepted, which means many points will be accepted. So if we encountered with the global minimum, it will also leave quickly without further searching, thus can not find the global minimum.
- **For sigma**, sigma controls the jump step to next point. So if it is too large, our points will jump all around the image, thus difficult to search for one region. If it is too small, we will move very slow and just in a very small region, thus we can not seach for more place to find the global minimum.

Code:

```
1   import numpy as np
2   import matplotlib
3   import matplotlib.pyplot as plt
4
5   from mpl_toolkits.mplot3d import Axes3D
6   from matplotlib import cm
7
    from matplotlib.ticker import LinearLocator, FormatStrFormatt
    er
8
9   def f(x):
10      return 418.9829*2 -
     (x[0] * np.sin(np.sqrt(np.abs(x[0]))) + x[1] * np.sin(np.sqr
    t(np.abs(x[1]))))
11
12  def Tlog(Taph, t):
13      return 1/(Taph*np.log(t))
14
15  def Tpoly(t):
16      return t**2
17
18  def Texp(Taph, t):
19      return np.exp(Taph*np.sqrt(t))
20
21  N_r = 500
22  x = np.linspace(-N_r,N_r,100)
23  y = np.linspace(-N_r,N_r,100)
24  X, Y = np.meshgrid(x, y)
25  Z = f([X,Y])
26
27  plt.figure(num=None,dpi=100)
28  plt.contourf(X,Y,Z)
```

```
plt.colorbar()

cplot = plt.figure(num=None,dpi=150)
ax = cplot.add_subplot(111,projection='3d')
surf = ax.plot_surface(X,Y,Z)
cbar = cplot.colorbar(surf, shrink=0.5, aspect=5)
cbar.minorticks_on()
plt.show()

N = 10000 # try: 50,200,1000,10000

X0 = [0,0]
Xt = [0,0]
Xt1 = [0,0]
Xpath = np.array([[0,0]])

T0 = 100
Taph = 1

std = [10,10] # standard deviation of jump function (normal)--find best

min = np.zeros((30,))
for i in range (30):
    for t in range(N):
        Xt1[0] = np.random.normal(Xt[0], std[0])
        Xt1[1] = np.random.normal(Xt[1], std[1])
        while (Xt1[0]<-500 or Xt1[0]>500):
            Xt1[0] = np.random.normal(Xt[0], std[0])
        while (Xt1[1]<-500 or Xt1[1]>500):
            Xt1[1] = np.random.normal(Xt[1], std[1])
        aph = np.exp((f(Xt)-f(Xt1))/Tt)
        u = np.random.uniform(0,1)
        #update Xt
        if u <= aph:
            Xt = np.array(Xt1)
            Xpath = np.concatenate((Xpath, [Xt1]), axis=0)
        #update Tt
        Tt = T0*Texp(Taph, t+2)
        min[i] = f(Xt)

minV = np.min(min)

plt.plot(Xpath[:,0], Xpath[:,1], 'C3', linewidth=1, alpha = 0.5, label = 'path')

plt.scatter(Xt[0],Xt[1], marker = 'o', color = 'r', s = 80, label = 'final position')
plt.legend()
plt.show()

print(minV)
```

```
76 plt.hist(min)
77 plt.xlabel('minimum value found')
78 plt.ylabel('times')
79 plt.show()
```

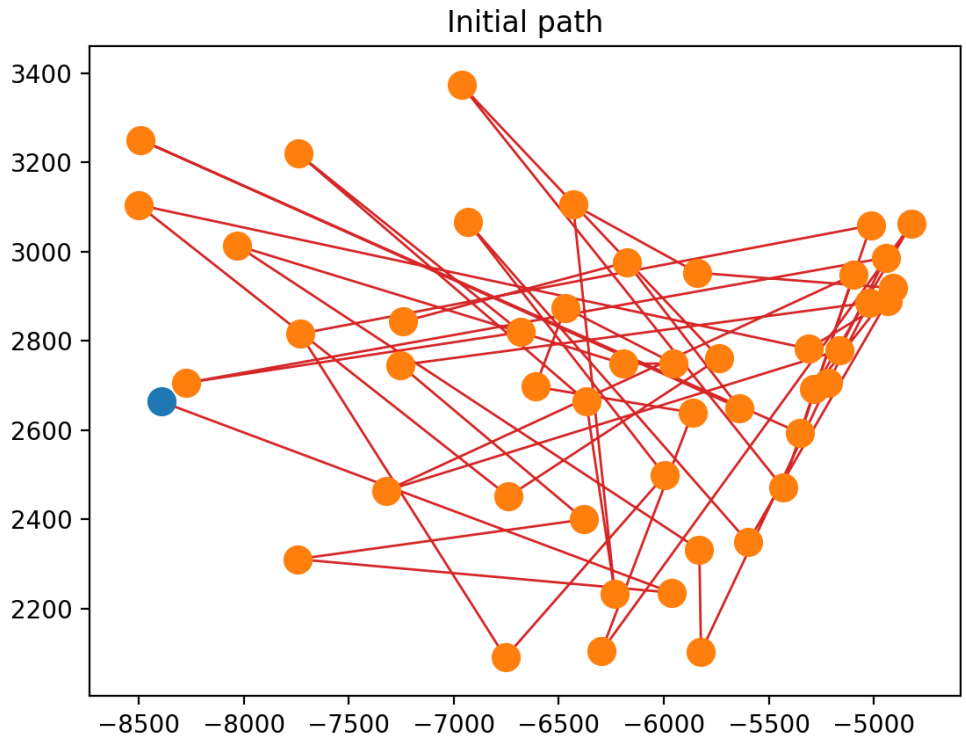
Question 4

Explain:

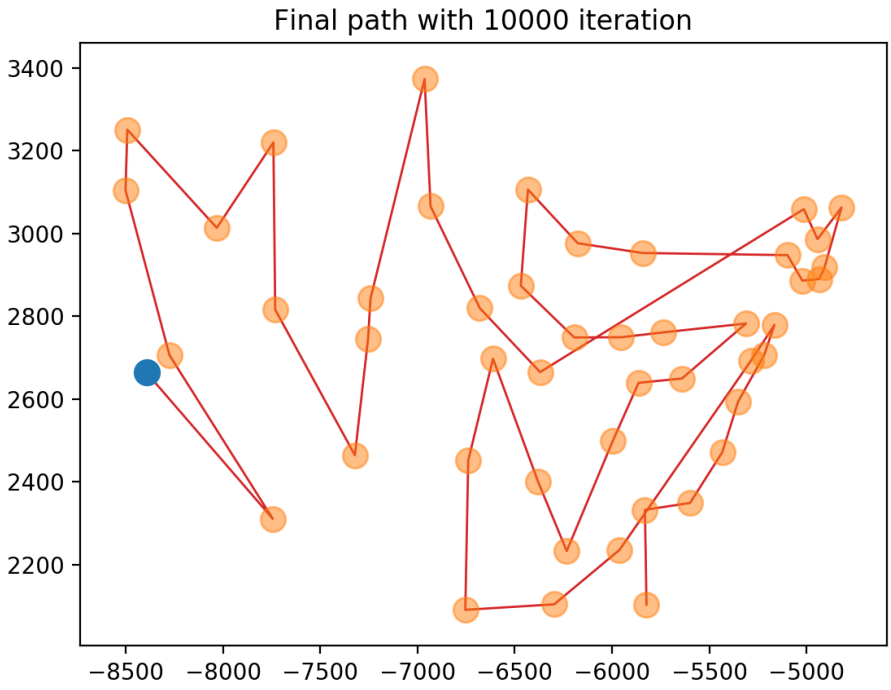
- **Slove TSP by using Metropolis-Hastings method in discrete case.**
- **Here we have 48 cities to consider with. Set the start city as CA, and find the minimum path to travel through all the cities:**
  - use the arbitrary order of CA and the rest 47 cities as the initial path, compute path l(p0)
  - in every iteration:
    - generate new path p' by randomly choose two cities (expect CA) and swap there position
    - comput path l(p')
    - accept or reject p':
      - compute q = (t+1)^[((l(pt)-l(p'))/c], where pt is last path, p' is the new path, c is a parameter
      - if l(p') < l(pt), accept p' as next path
      - else, accept p' with probability q:
        - generate u from U[0,1], if u<p', accept p'; else reject p'
  - update path with p'

Results:

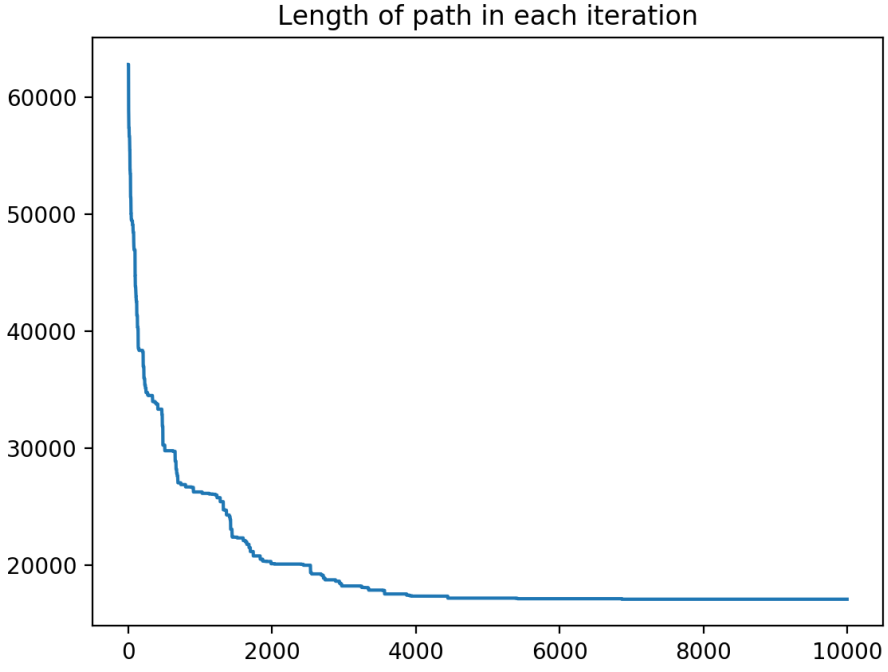
- **Initial path: CA is in blue point**



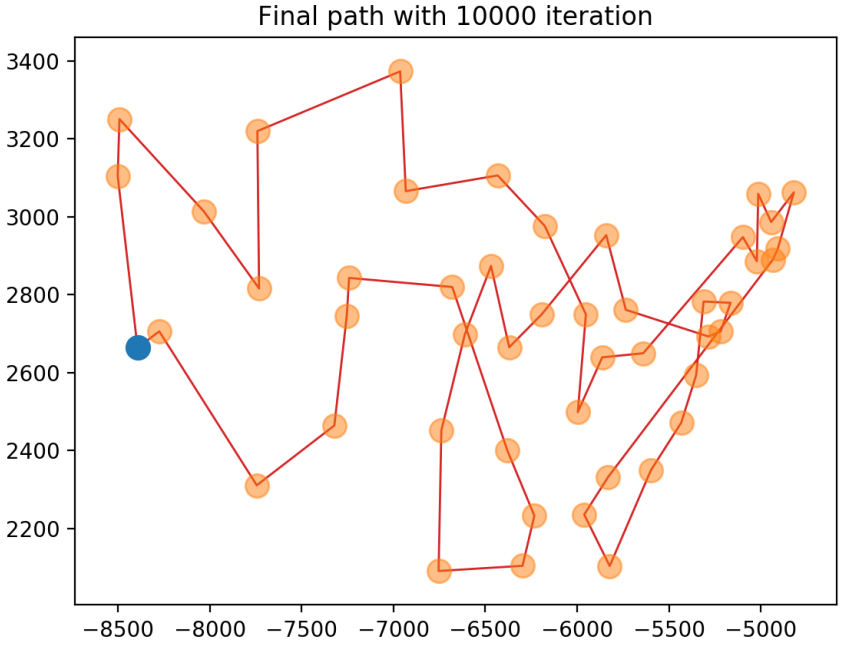
- **Best path when the final destination is not CA (with c =1 and 10000 iterations)**
  - **len = 15876**
  - Compare with the initial path, we can find that this path starts from CA and travles roughly from left to right, which may be one choice in real life.
  - But not strictly from left to right, in the middle part, our path failed to find the nearest cities and traveled in circle.



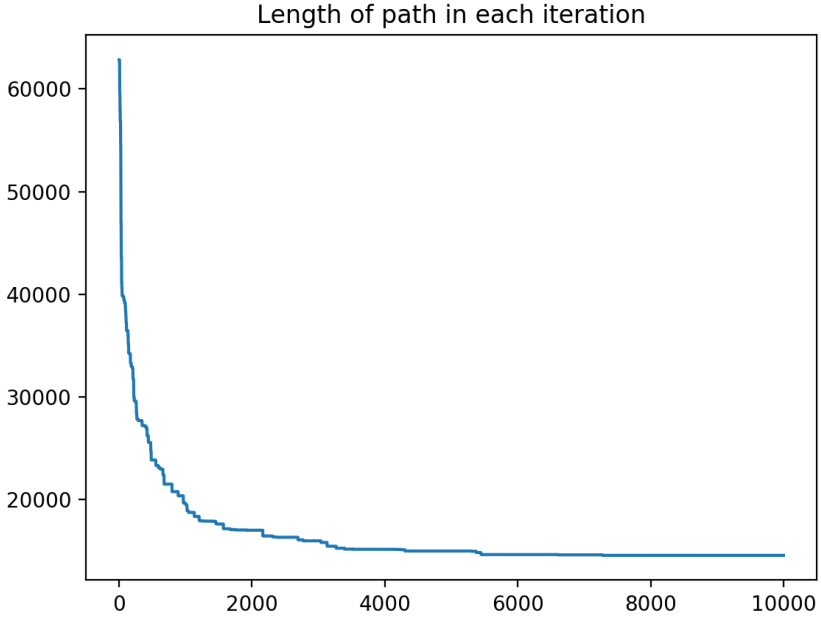
- **change of tour distance in simulation time:**
  - roughly converge at 5000 iterations
  - rate of converge is kind of logarithmically, which means it converge to the minimum path very fast.



- **Best path when the final destination is CA (with c =1 and 10000 iterations)**
  - **len = 14590**
  - Compare with the path above, we can find that this path starts from CA and travles from south to north and left to right, which may also be one choice in real life.
  - But not strictly, in the right part, our path failed to find the nearest cities and the path still has many intersect points, which means that we are in a local minimum of the path, or not find the best minimum around the global minimum.



- **change of tour distance in simulation time:**
  - roughly converge at 8000 iterations
  - rate of converge is logarithmically, which means it converge to the minimum path very fast.



Code:

```
1  from sklearn.metrics import pairwise_distances
2  from sklearn.utils.random import sample_without_replacement
3  import numpy as np
4  import matplotlib.pyplot as plt
5
6  from scipy.spatial import distance
7
8  def totalDist(xy_CA, xy, p):
9      p_len = distance.euclidean(xy_CA, xy[p[0]])
10     for i in range(0,46):
11         p_len = p_len + distance.euclidean(xy[p[i]], xy[p[i
   +1]])
12     return p_len
13
14
15 N_cities = 50
16
17 filename = '/Users/mac-pro/Desktop/20SPRING/511/project 8/
   uscap_xy.txt'
18 xy = np.loadtxt(filename)
19 print(xy.shape)
20
21 filename = '/Users/mac-pro/Desktop/20SPRING/511/project 8/
   uscap_name.txt'
22 with open(filename) as fp:
23    names = fp.readlines()[0:50]
24 #remove Alaska and Hawaii
25 index_Alaska = names.index('Juneau, Alaska\n')
26 xy = np.delete(xy,index_Alaska,0)
27 names.remove('Juneau, Alaska\n')
28 index_Hawaii = names.index('Honolulu, Hawaii\n')
29 xy = np.delete(xy,index_Hawaii,0)
30 names.remove('Honolulu, Hawaii\n')
31 #find position of CA as xy_CA
32 index_CA = names.index('Sacramento, California\n')
33 xy_CA = xy[index_CA]
34 xy = np.delete(xy,index_CA, 0)
35 names.remove('Sacramento, California\n')
36
37 #initial path:
38 N_rest = 47
39 p = np.arange(0, N_rest)
40 p_len = totalDist(xy_CA, xy, p)
41
42 xy_init = np.concatenate(([xy_CA], xy), axis = 0)
43 plt.plot(xy_init[:,0], xy_init[:,1], 'C3', zorder=1, lw=1)
44 plt.scatter(xy_CA[0], xy_CA[1], s=120, zorder=2)
45 plt.scatter(xy[:,0], xy[:,1], s=120, zorder=2)
46 plt.title('Initial path')
47 plt.show()
```

```
48
49
50
51  T = 20
52  p_lenT = []
53  pT = []
54  for i in range (T):
55      N = 10000
56      c = 1
57      pt = np.copy(p)
58      p_lent = p_len + distance.euclidean(xy_CA, xy[46])
59      #record total len
60      p_lenHistory = np.zeros((N+1,))
61      p_lenHistory[0] = p_lent
62      #record path
63      p_pathHistory = []
64      p_pathHistory.append(np.copy(p))
65
66      for t in range (1,N+1):
67          #random swap
68          i, j = np.random.choice(N_rest,2)
69          p2 = np.copy(pt)
70          p2[i], p2[j] = p2[j], p2[i]
71          #new distence
72
            p_len2 = totalDist(xy_CA, xy, p2) + distance.euclidea
    n(xy_CA, xy[p2[46]])
73          #compute q
74          q = np.power((t+1), (p_lent - p_len2)/c)
75          #check if accept
76          if p_len2 < p_lent:
77              pt = np.copy(p2)
78              p_lent = p_len2
79          else:
80              u = np.random.uniform(0,1)
81              if u < q:
82                  pt = np.copy(p2)
83                  p_lent = p_len2
84
85          p_lenHistory[t] = p_lent
86          p_pathHistory.append(np.copy(pt))
87      p_lenT.append(p_lent)
88      pT.append(pt)
89
90  plt.plot(p_lenHistory)
91  plt.title('Length of path in each iteration')
92  plt.show()
93
94  plt.plot(p_lenT)
95  plt.show()
96
97  p_lent = np.min(p_lenT)
```

```
pt = pT[np.argmin(p_lenT)]

xy_final = []
xy_final.append(xy_CA)
for i in range(N_rest):
    xy_final.append(xy[pt[i]])
xy_final.append(xy_CA)
xy_final = np.array(xy_final)

print(p_lent)

plt.plot(xy_final[:,0], xy_final[:,
1], 'C3', zorder=1, lw=1)
plt.scatter(xy_CA[0], xy_CA[1], s=120, zorder=2)

   plt.scatter(xy_final[1:48,0], xy_final[1:48,1], s=120, alpha
   = 0.5, zorder=2)
plt.title('Final path with 10000 iteration')
plt.show()
```

Reference:

① **https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.scatter.html**

② **https://zs.symbolab.com/solver/multiple-integrals-calculator/**

③ **https://en.wikipedia.org/wiki/Rate_of_convergence**

④ **https://docs.scipy.org/doc/numpy/reference/generated/numpy.histogram.html**