EE511 Project 5 Summary

SilongHu email:silonghu@usc.edu

This summary is divided into 3 parts: Implementation, Conclusion and Source Code

I. Implementation

Problem 1. [Monte Carlo]

-Use these random samples to estimate the area of the inscribed quarter circle. Use this area estimate to estimate the value of pi. Do k=50 runs of these pi-estimations. Plot the histogram of the 50 pi-estimates.

To evaluate the area, we generate 2-dimensional uniform distribution, and calculate the x^2 + y^2.

```
Pi_array = np.zeros((50,1))
for k in range(0,50):
    Pi = 0
    a = np.random.uniform(0,1,size=(100,2))
    for i in range(0,100):
        if a[i][0] ** 2 + a[i][1] ** 2 <= 1:
            Pi += 1
# Because Pi      Real_Pi
#         ----  = --------
#          100        4

    Pi_array[k] = Pi/25.0
```

Code Piece 1. Area Estimate

And we generate the histogram of the 50 pi-estimates.

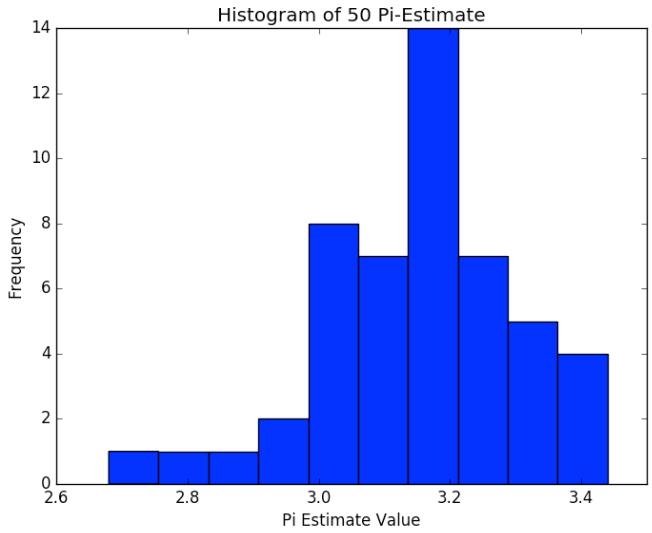


Figure 1. Histogram of 50 Pi-Estimates

We could see that the value near 3.1 and 3.2 are most, which is close to Pi (3.1415926)

-Repeat the experiment with different numbers of uniform samples, n (using k=50 for all these runs). Plot the sample variance of the pi-estimates for these different values of n. What is the relationship the estimate variance and Monte Carlo sample size?

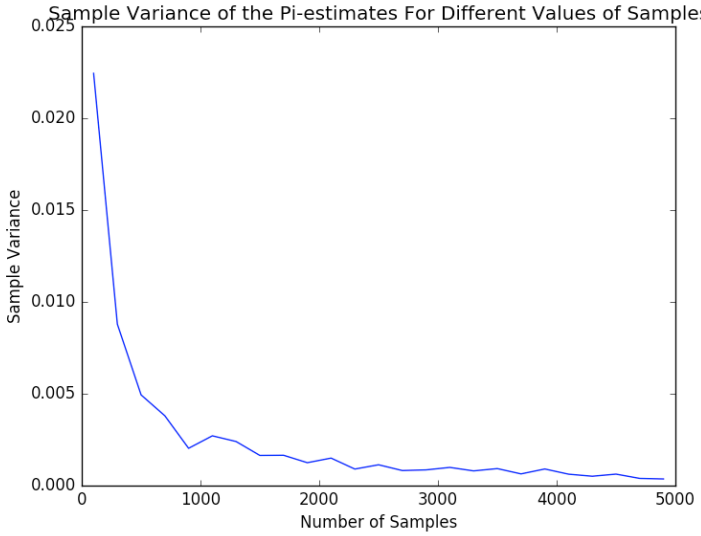


Figure 2. Sample Variance of the Pi-Estimate

As figure 2 shown, when we increase the number of samples, the estimate variance would be decreased. Thus, the more Monte Carlo sample size, the less estimate variance.

-Adapt your Monte Carlo solution to provide integral and error estimates for the function: $g(x, y) = |4x - 2| \times |4y - 2|$ x, y in [0, 1]

We use our Monte Carlo solution to calculate the integral of the 2-dimensional function, which returns the integral value, error, and estimate variance.

```
def integrateMC(func, dim, limit, expected, N):
    I = 1. / N
    tol = []
    for n in range(dim):
        I *= (limit[n][1] - limit[n][0])

    for k in range(N):
        x = []
        for n in range(dim):
            x.append(random.uniform(limit[n][0], limit[n][1]))
        tol.append(func(x))
    est_var = 0
    m = np.mean(tol)
    for nums in tol:
        est_var += (nums - m) ** 2

    return I * np.sum(tol), abs(I * np.sum(tol) - expected),est_var
```

Code Piece 2. MC Integral Function

```
Using n =  50
Result =  0.908060269802 estimated variance =  0.0177706967632
Known result =  1  error =  0.0919397301979  =  9.19397301979 %

Using n =  100
Result =  0.989650471514 estimated variance =  0.00952117307469
Known result =  1  error =  0.0103495284859  =  1.03495284859 %

Using n =  500
Result =  1.00338836451 estimated variance =  0.00163581131085
Known result =  1  error =  0.00338836451463  =  0.338836451463 %

Using n =  1000
Result =  0.998770385059 estimated variance =  0.000814094686684
Known result =  1  error =  0.001229614941  =  0.1229614941 %
```

Figure 3. Results of Different Sample Numbers.

As shown in the result above-mentioned, the more sampling numbers we use, the more accuracy to get the real value. And the estimated variance and error would be smaller.

Problem 2. [Variance Reduction Methods for Monte Carlo]

In this problem, we solve the stratification first:

```
*************Function A real value:  20.007707388987143 *****************
For minimum estimate:
Stratified integral value of a:  18.4249987859 estimate variance of a:  0.325044061394
For closest value:
Stratified integral value of a:  20.0039932734 estimate variance of a:  0.439141034923
MC integral a: 20.1627737575 MC estimate variance of a:  0.50189838926
*************Function B real value:  0.0 *****************
For minimum estimate:
Stratified integral value of b:  -0.0512758683068 estimate variance of b:  0.000458239128688
For closest value:
Stratified integral value of b:  3.65423229627e-05 estimate variance of b:  0.000481769260107
MC integral b: -0.0245695042935 MC estimate variance of b:  0.000471904721511
*************Function C real value:  1.0 *****************
For minimum estimate:
Stratified integral value of c:  0.943702974998 estimate variance of c:  0.000594235817186
For closest value:
Stratified integral value of c:  0.999988328437 estimate variance of c:  0.000683936919439
MC integral c: 1.00339488786 MC estimate variance of c:  0.000793189405802
```

Figure 4. Results of Stratified Sampling.

In the above 3 2-dimensiaonal functions, we only divide x from [0,sigma] and [sigma,1]. In the Cos function, we uniformly divide x as the same way but from [-1,1]. For each sigma from [0.001, 0.999] in their function, we calculate the separate value of them. And finally we choose 2 solutions: one for closest value, and the other for minimum variance.

As shown in figure 4, there exists that the value is closer to the real value than MC method, and estimate variance is also decreased. Thus, the stratified method indicates the variance reduction.

And Now, we focus on Importance Sampling, in this case, we divide all the 2-dimensional function into 1-dimensional function, and then compute their integral value and the estimate variance. For each 1-dimensional function, I choose the following g(x):

```
def g_a1(x):
    return  0.792 * 2.5 **(abs(x[0] - 0.5))
def g_b1(x):
    return 2.93 /math.sqrt(2 * math.pi * 1) * math.exp( - x[0] ** 2/ 2)
def g_c1(x):
    return  -1. * (x[0] - 1) ** 2 + 1.33
```

```
Importance Sampling Value of A:  19.02103233  Estimate variance of IS of A:  0.420548560395
MC integral a: 20.6286582798 MC estimate variance of a:  0.507863859659
Importance Sampling Value of B:  -0.0491385288239  Estimate variance of IS of B:  0.000588033040281
MC integral b: -0.0301926383672 MC estimate variance of b:  0.000503119940096
Importance Sampling Value of C:  1.92256922701  Estimate variance of IS of C:  0.00741799271153
MC integral c: 1.00217071378 MC estimate variance of c:  0.000789104465489
```

Figure 5. Results of Importance Sampling

In this case, we can see the estimate variance reduction effect is not good as stratified sampling, because the function we choose would impact on the value of variance.

Extra Credit: [MCMC for Optimization]

-Plot a contour plot of the surface for the 2-D surface

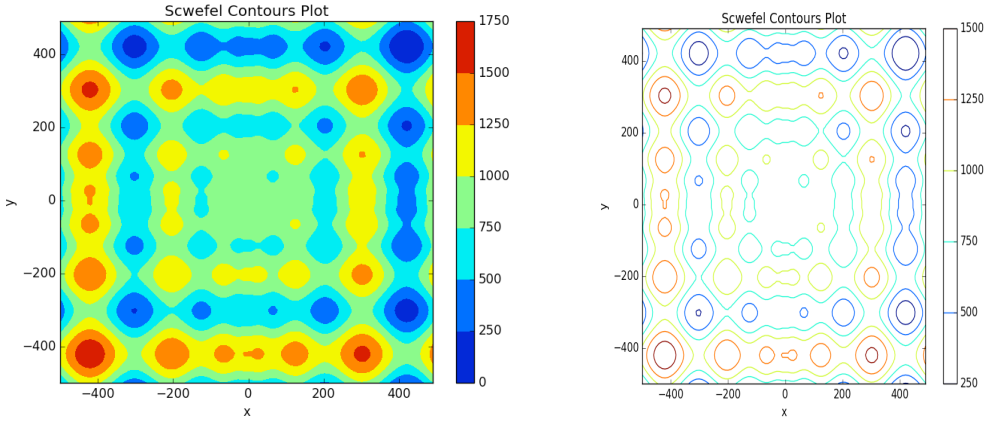


Figure 6. Contour Plot with/without color bar

-Implement a simulated annealing procedure to find the global minimum of this surface

-Choose your best run and overlay your 2-D sample path on the contour plot of the Schwefel function to visualize the locations your optimization routine explored.

We solve these 2 questions combined. Part of code are shown.

```
while Temperature > Tolerance:

    # Choosing the Cooling style
    Temperature = DecayScale * Temperature #快速降温 For 3.2
    #Temperature = (0.88 ** (iteration + 1) ) * Temperature #Exponential
    #Temperature = Temperature / ( 1 + alpha * math.log( 1 + iteration,math.e)) #logarithmic
    #Temperature = Temperature / (1 + 0.1 * iteration) # polynomial
```

Code Piece 3. Cooling Schedule

```
    #Metropolis Process
if (Obj(PreX,PreY) - Obj(NextX,NextY) > 0):
    PreX = NextX
    PreY = NextY
    AcceptPoints += 1
else:
    changer = -1. * (Obj(NextX,NextY) - Obj(PreX,PreY))/Temperature
    rnd = random.random()
    p1 = math.exp(changer)
    if p1 > rnd:
        PreX = NextX
        PreY = NextY
```

Code Piece 4. Metropolis Process

When we finish run the process, we find the least point to iterate to the minimum value, in this case, the minimum value is 0, and we only use 13 steps to get the expected value. To see the figure clearly, we didn't use the color bar. In the figure 7, the point finally gather to the minimum value range.

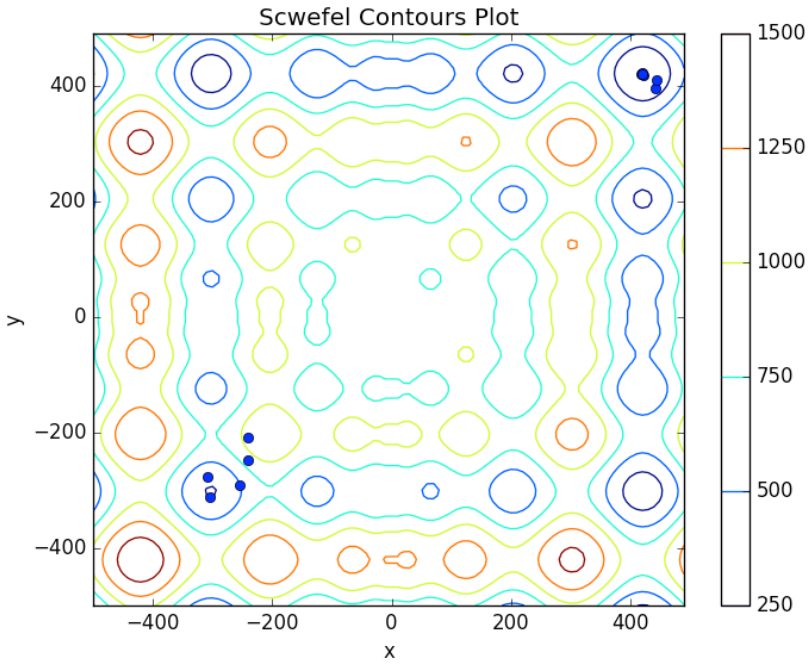
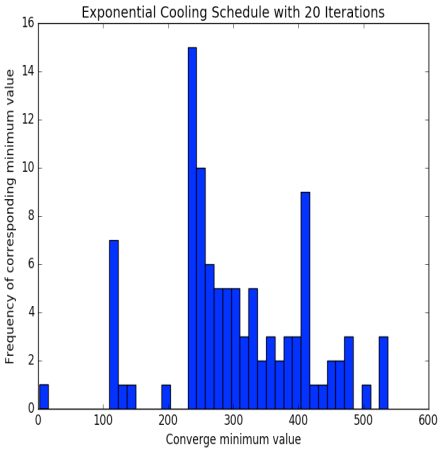
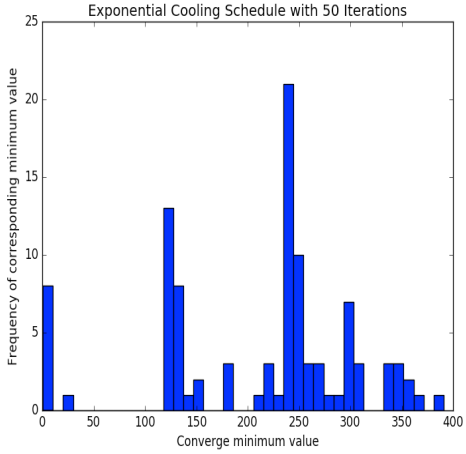


Figure 7. Contour Plot with Routing Points

-Explore the behavior of the procedure starting from the origin with an exponential, a polynomial, and a logarithmic cooling schedule. Run the procedure for t={20, 50, 100, 1000} iterations for k=100 runs each. Plot a histogram of the function minima your procedure converges to.





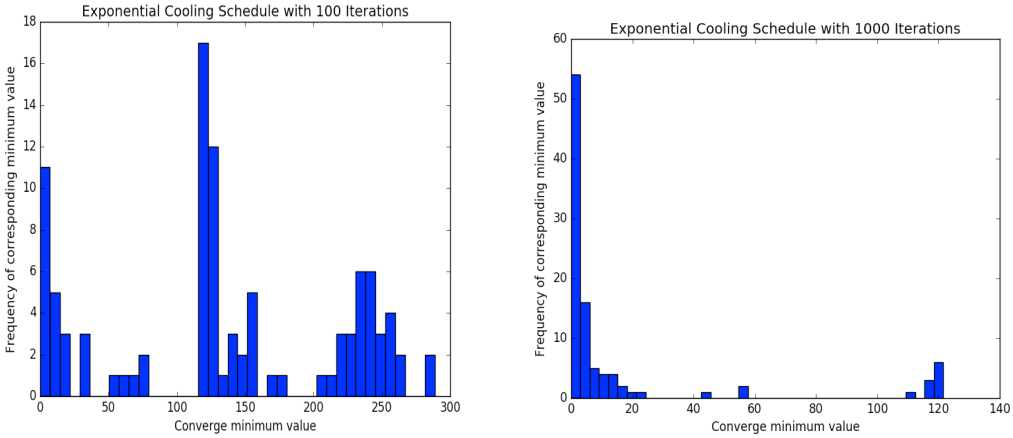


Figure 8. Exponential Cooling Schedule Histogram

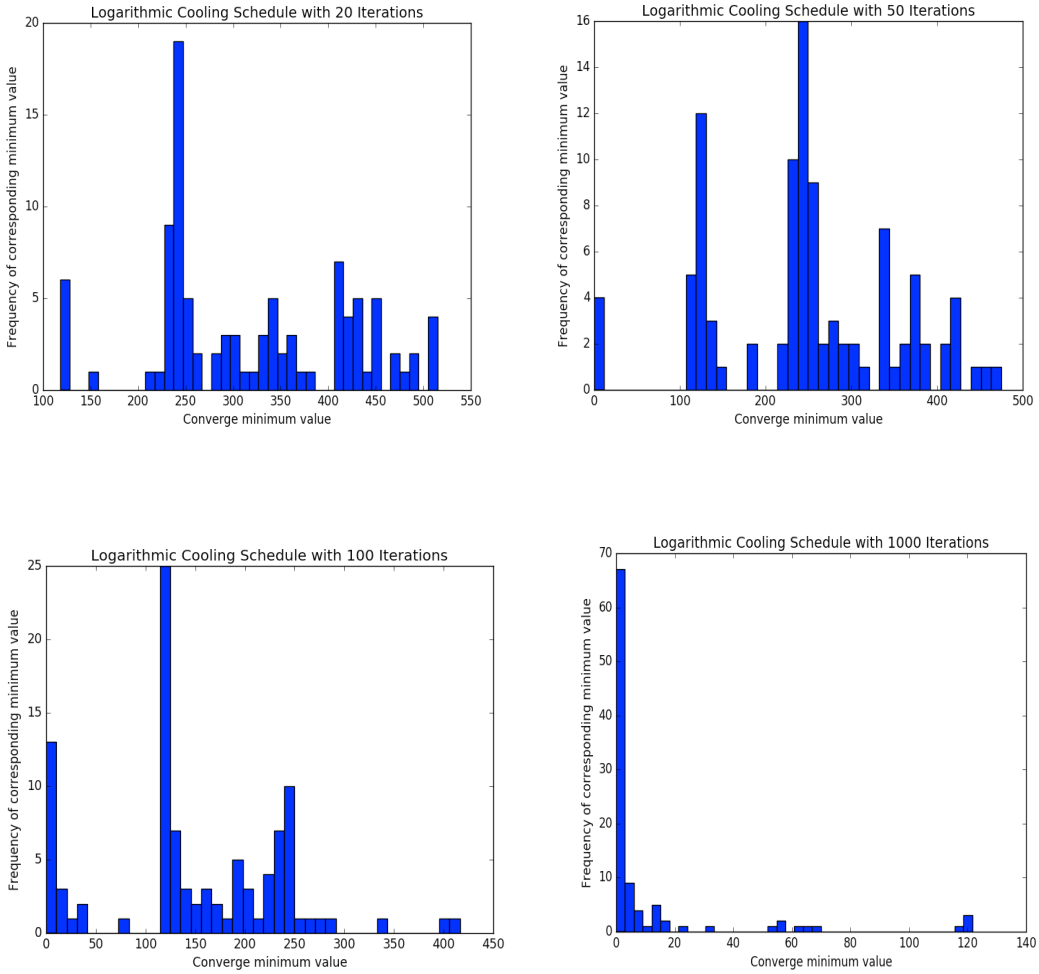


Figure 8. Logarithmic Cooling Schedule Histogram

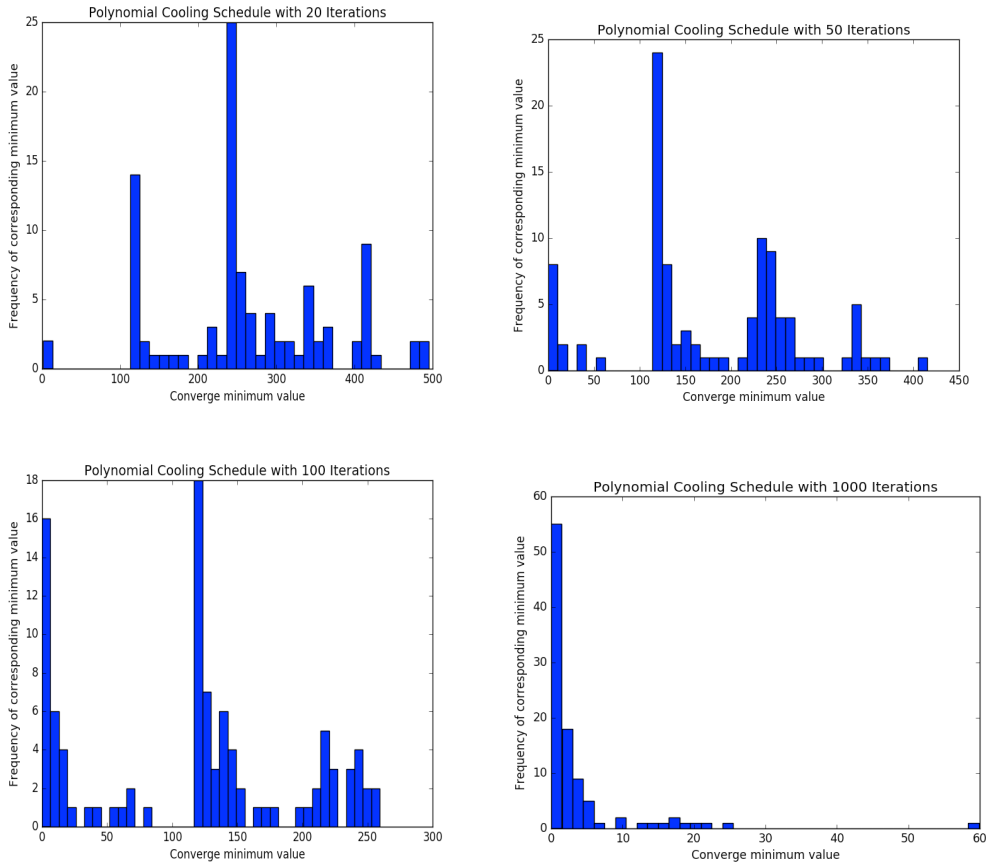


Figure 9. Polynomial Cooling Schedule Histogram

As shown above, the Logarithmic and Polynomial Cooling schedule has faster converge rate than the Exponential one. For specific cooling schedule, the more markov chain iteration, the more frequency to get the minimum value which is 0 in our case.

II. Conclusion

In this project, we learned an important Monte Carlo Simulation method to calculate the integral value of multi-dimensional function.

We also learned the plot on python, and stratified, importance sampling which can reduce the variance.

For Annealing Simulation, we find the minimum value via different cooling schedule, and each converge rate is different. The best path use the least step.

III. Source Code

1. MC_1.py

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

Pi_array = np.zeros((50,1))
```

```
for k in range(0,50):
    Pi = 0
    a = np.random.uniform(0,1,size=(100,2))
    for i in range(0,100):
        if a[i][0] ** 2 + a[i][1] ** 2 <= 1:
            Pi += 1
# Because Pi    Real_Pi
#      ----  = --------
#       100       4

    Pi_array[k] = Pi/25.0

plt.hist(Pi_array)  # plt.hist passes it's arguments to np.histogram
plt.title("Histogram of 50 Pi-Estimate")
plt.xlabel("Pi Estimate Value")
plt.ylabel("Frequency")
plt.show()
```

2. MC_2.py

```
import numpy as np
import matplotlib.pyplot as plt
#a = np.array(a)

Pi_array = np.zeros((50,1))
N = list(range(100,5000,200))
Var_array = []
#Var_array = np.zeros((50,1))
for nums in N:
    for k in range(0,50):
        Pi = 0
        a = np.random.uniform(0,1,size=(nums,2))
        for i in range(0,nums):
            if a[i][0] ** 2 + a[i][1] ** 2 <= 1:
                Pi += 1
# Because Pi    Real_Pi
#      ----  = --------
#       100       4

        Pi_array[k] = Pi/(nums/4.0)
#    Var_array.append(np.var(Pi_array))
    Est_var = 0
    mean = np.mean(Pi_array)
    for unit in Pi_array:
        Est_var += (unit - mean)**2
```

```
    Var_array.append(Est_var/49.0)

plt.plot(N,Var_array)
plt.title("Sample Variance of the Pi-estimates For Different Values of Samples")
plt.xlabel("Number of Samples")
plt.ylabel("Sample Variance")
plt.show()
```

3. MC_3.py

```
import random
import numpy as np

def integrateMC(func, dim, limit, expected, N):
    I = 1. / N
    tol = []
    for n in range(dim):
        I *= (limit[n][1] - limit[n][0])

    for k in range(N):
        x = []
        for n in range(dim):
            x.append(random.uniform(limit[n][0], limit[n][1]))
        tol.append(func(x))
    est_var = 0
    m = np.mean(tol)
    for nums in tol:
        est_var += (nums - m) ** 2

    return I * np.sum(tol), abs(I * np.sum(tol) - expected), est_var * I / (N -
1)

def integrand(x):
    return (abs(4*x[0] - 2) * abs(4*x[1] - 2))

def sampler():
    while True:
        y = random.uniform(0.,1.)
        x = random.uniform(0.,1.)
        yield (x,y)

dim, limit = 2, [[0, 1], [0, 1]]
N = 100  # starting value for number of MC samples
x, y, y2 = [], [], []  # for plotting
n = 2  # the number of times each integration is performed to
# define average and SDOM
```

```
domainsize = 1 # x,y belongs to [0,1]
expected = 1 # Calculate the integal value of g(x,y)
for nmc in [50, 100, 500, 1000]:
    random.seed(1)
    result, error,est_var = integrateMC(integrand, dim,limit,expected, nmc)
    #diff = abs(result - expected)
    print ("Using n = ", nmc)
    print ("Result = ", result, "estimated variance = ", est_var)
    print ("Known result = ", expected, " error = ", error, " = ", 100.*
error/expected, "%")
    print (" ")
```

4. stratified.py

```
from __future__ import division
import random
import numpy as np
import math
import warnings

warnings.simplefilter("error")

def integrateMC(func, dim, limit, N):
    I = 1. / N
    tol = []
    for n in range(dim):
        I *= (limit[n][1] - limit[n][0])

    for k in range(N):
        x = []
        for n in range(dim):
            x.append(random.uniform(limit[n][0], limit[n][1]))
        tol.append(func(x))
    est_var = 0
    m = np.mean(tol)
    for nums in tol:
        est_var += (nums - m) ** 2
    return I * np.sum(tol), est_var * I / (N - 1)


def SS(func, dim, limit, N, sigma):
    # for 2 dimension, it's has 2 part,
    # We divide X into [0,sigma] and [sigma,1], remain Y same
    I = 1. / N
```

```
    N_a = 0.
    N_b = 0.
    tol_a = []
    tol_b = []

    # We only deal with X

    for n in range(dim):
        if n==0:
            N_a = N * (limit[n][1] * sigma - limit[n][0])/(limit[n][1] - limit[n][0])
            N_b = N * (limit[n][1] - limit[n][1] * sigma) / (limit[n][1] -
limit[n][0])



    for n in range(dim):
        I *= (limit[n][1] - limit[n][0])

    for k in range(int(N_a)):
        x = []
        for n in range(dim):
            if n == 0:
                x.append(random.uniform(limit[n][0], sigma * limit[n][1]))
            else:
                x.append(random.uniform(limit[n][0], limit[n][1]))

        tol_a.append(func(x))

    for k in range(int(N_b)):
        x = []
        for n in range(dim):
            if n == 0:
                x.append(random.uniform(sigma * limit[n][1], limit[n][1]))
            else:
                x.append(random.uniform(limit[n][0], limit[n][1]))

        tol_b.append(func(x))
    #print (tol_a)
    #print (np.var(tol_b))
    est_var = float(sigma - limit[0][0])/(limit[0][1] - limit[0][0]) / N *
np.var(tol_a) + \
              float(limit[0][1] - sigma)/(limit[0][1] - limit[0][0]) / N *
np.var(tol_b)
    val =  I * np.sum(tol_a) + I * np.sum(tol_b)
    diff = abs(np.var(tol_a) - np.var(tol_b))
```

```
    return val, est_var, diff

def SS_2(func, dim, limit, N, sigmax, sigmay):
    # for 2 dimension, it's has 4 part
    N_a = N * sigmax * sigmay
    N_b = N * sigmax * (1 - sigmay)
    N_c = N * (1 - sigmax) * sigmay
    N_d = N * (1 - sigmax) * (1 - sigmay)
    I_a = 1. / N_a
    I_b = 1. / N_b
    I_c = 1. / N_c
    I_d = 1. / N_d
    tol_a = []
    tol_b = []
    tol_c = []
    tol_d = []

    for k in range(int(N_a)):
        x = []
        for n in range(dim):
            if n == 0:
                x.append(random.uniform(0, sigmax))
            else:
                x.append(random.uniform(0, sigmay))
        # print 'x',x
        tol_a.append(func(x))

    for k in range(int(N_b)):
        x = []
        for n in range(dim):
            if n == 0:
                x.append(random.uniform(0, sigmax))
            else:
                x.append(random.uniform(sigmay, 1))
        # print 'x2',x
        tol_b.append(func(x))

    for k in range(int(N_c)):
        x = []
        for n in range(dim):
            if n == 0:
                x.append(random.uniform(sigmax, 1))
            else:
                x.append(random.uniform(0, sigmay))
```

```
        # print 'x2',x
        tol_c.append(func(x))

    for k in range(int(N_d)):
        x = []
        for n in range(dim):
            if n == 0:
                x.append(random.uniform(sigmax, 1))
            else:
                x.append(random.uniform(sigmay, 1))
        # print 'x2',x
        tol_d.append(func(x))
    # print 'tol_a: ',np.sum(tol_a),' mean_of_a:',I_a * np.sum(tol_a)
    # print 'tol_b: ',np.sum(tol_b),' mean_of_b:',I_b * np.sum(tol_b)
    est_var = sigmax * sigmay / N * np.var(tol_a) + (1 - sigmay) * sigmax / N *
np.var(tol_b) + \
              (1 - sigmax) * sigmay / N * np.var(tol_c) + (1 - sigmay) * (1 - sigmax)
/ N * np.var(tol_d)

    val = sigmax * sigmay* I_a * np.sum(tol_a) + sigmax * (1-sigmay)* I_b *
np.sum(tol_b) + \
          (1-sigmax)*sigmay*I_c * np.sum(tol_c) + (1-sigmax)*(1-sigmay)*I_d *
np.sum(tol_d)
    diff = abs(np.var(tol_a) - np.var(tol_b)) + abs(np.var(tol_a) - np.var(tol_c))
+ \
           abs(np.var(tol_a) - np.var(tol_d))
    return val, est_var, diff

def f_a(x):
    return math.exp(5 * abs(x[0] - 0.5) + 5 * abs(x[1] - 0.5))

def f_b(x):
    return math.cos(math.pi + 5 * (x[0] + x[1]))

def f_c(x):
    """ integrand function """
    return (abs(4 * x[0] - 2) * abs(4 * x[1] - 2))


dim = 2
limit_a = [[0, 1], [0, 1]]
limit_b = [[-1, 1], [-1, 1]]
limit_b1 = [[-1, 0], [-1, 1]]
limit_b2 = [[0, 1], [-1, 1]]
```

```
limit_c = [[0, 1], [0, 1]]
N = 1000  # starting value for number of MC samples

#****************** Function A Operation ************************

real_a = 4./25 * (math.exp(2.5) - 1) ** 2
dict_a = {}
dict_a2 = {}
for sigma in np.arange(0.001, 1, 0.001):
    S_a = SS(f_a, dim, limit_a, N, sigma)
    #dict_a[S_a[2]] = S_a[0],S_a[1]
    dict_a[S_a[1]]= S_a[0] # find the minimum estimate variance
    dict_a2[abs(S_a[0] - real_a)] = S_a[0],S_a[1] # find the closed value
print ("**************Function A real value: ",real_a,"******************")
a = (min(dict_a, key=float))
print ("For minimum estimate: ")
print ("Stratified integral value of a: ", dict_a[a] \
      ,"estimate variance of a: " , a )

a2 = (min(dict_a2, key=float))
print ("For closest value: ")
print ("Stratified integral value of a: ", dict_a2[a2][0]  \
      ,"estimate variance of a: " , dict_a2[a2][1] )


v_a = integrateMC(f_a, dim, limit_a, N)
print ("MC integral a:", v_a[0], "MC estimate variance of a: ",v_a[1])


#****************** Function B Operation ************************

real_b = 0.0
dict_b = {}   # find the minimum estimate variance
dict_b2 = {}  # find the closed value
for sigma in np.arange(-0.9, 0.9, 0.001):
    S_b = SS(f_b, dim, limit_b, N, sigma)
    #S_b2 = SS(f_b, dim, limit_b2, N, sigma)
    dict_b[S_b[1]]= S_b[0] # find the minimum estimate variance
    dict_b2[abs(S_b[0])] = S_b[0],S_b[1] # find the closed value

print ("**************Function B real value: ",real_b,"******************")
b = (min(dict_b, key=float))
print ("For minimum estimate: ")
print ("Stratified integral value of b: ", dict_b[b] \
```

```
      ,"estimate variance of b: " , b )

b2 = (min(dict_b2, key=float))
print ("For closest value: ")
print ("Stratified integral value of b: ", dict_b2[b2][0]  \
      ,"estimate variance of b: " , dict_b2[b2][1] )

v_b = integrateMC(f_b, dim, limit_a, N)
print ("MC integral b:", v_b[0], "MC estimate variance of b: ",v_b[1])


#****************** Function C Operation *************************
real_c = 1.0

dict_c = {}
dict_c2 = {}
for sigma in np.arange(0.001, 1, 0.001):
    S_c = SS(f_c, dim, limit_c, N, sigma)
    dict_c[S_c[1]]= S_c[0] # find the minimum estimate variance
    dict_c2[abs(S_c[0] - real_c)] = S_c[0],S_c[1] # find the closed value
print ("**************Function C real value: ",real_c,"*****************")
c = (min(dict_c, key=float))
print ("For minimum estimate: ")
print ("Stratified integral value of c: ", dict_c[c] \
      ,"estimate variance of c: " , c )

c2 = (min(dict_c2, key=float))
print ("For closest value: ")
print ("Stratified integral value of c: ", dict_c2[c2][0]  \
      ,"estimate variance of c: " , dict_c2[c2][1] )

v_c = integrateMC(f_c, dim, limit_c, N)
print ("MC integral c:", v_c[0], "MC estimate variance of c: ",v_c[1])
```

5. ImportanceSampling.py

```
from __future__ import division
import numpy as np
import random
import scipy.interpolate as interpolate
import math
import matplotlib.pyplot as plt
from scipy.stats import norm

def inverse_transform_sampling(data, n_bins=50, n_samples=1000):
```

```
    hist, bin_edges = np.histogram(data, bins=n_bins, density=True)
    cum_values = np.zeros(bin_edges.shape)
    cum_values[1:] = np.cumsum(hist*np.diff(bin_edges))
    inv_cdf = interpolate.interp1d(cum_values, bin_edges)
    r = np.random.rand(n_samples)
    return inv_cdf(r)

def f_a(x):
    return math.exp(5 * abs(x[0] - 0.5) + 5 * abs(x[1] - 0.5))

def f_b(x):
    return math.cos(math.pi + 5 * (x[0] + x[1]))

def f_c(x):
    """ integrand function """
    return (abs(4 * x[0] - 2) * abs(4 * x[1] - 2))

def integrateMC(func, dim, limit, N):
    I = 1. / N
    tol = []
    for n in range(dim):
        I *= (limit[n][1] - limit[n][0])

    for k in range(N):
        x = []
        for n in range(dim):
            x.append(random.uniform(limit[n][0], limit[n][1]))
        tol.append(func(x))
    est_var = 0
    m = np.mean(tol)
    for nums in tol:
        est_var += (nums - m) ** 2
    return I * np.sum(tol), est_var * I / (N - 1)

def integrate(func, dim, limit, N):
    I = 1. / N
    tol = []
    for n in range(dim):
        #print (limit[n][1])
        I *= (limit[n][1] - limit[n][0])

    for k in range(N):
        x = []
        for n in range(dim):
```

```
            x.append(random.uniform(limit[n][0], limit[n][1]))
        #print ("x: ",x)
        #print ('func: ',func(x))
        tol.append(func(x))
    return tol


u1 = []
u2 = []
tol = []
dim = 2
limit_a = [[0, 1], [0, 1]]
limit_b = [[-1, 1], [-1, 1]]
limit_c = [[0, 1], [0, 1]]
N = 1000  # starting value for number of MC samples


#********************* Function A Operation ************
## Let's try 1-D formula:

def g_a1(x):
    return  0.792 * 2.5 **(abs(x[0] - 0.5))

def f_a1(x):
    return math.exp(5 * abs(x[0] - 0.5))

data_a = integrate(g_a1, dim, limit_a, N)
#print ("g_a1 integral value: ",np.mean(data_a))

g_dista = inverse_transform_sampling(data_a,50,1000)
ua = []
for i in range(1000):
    ua.append(random.random())
g_dista = g_dista * ua
h_a = []
for items in g_dista:
    h_a.append(f_a1([items])/g_a1([items]))

data_a1 = integrate(g_a1, dim, limit_a, N)
g_dista1 = inverse_transform_sampling(data_a1,50,1000)
ua1 = []
for i in range(1000):
    ua1.append(random.random())
```

```
g_dista1 = g_dista1 * ua1
h_a1 = []
for items in g_dista1:
    h_a1.append(f_a1([items])/g_a1([items]))

h_a = np.array(h_a) * np.array(h_a1)
print ('Importance Sampling Value of A: ', np.mean(h_a),' Estimate variance of
IS of A: ',np.var(h_a)/N)

v_a = integrateMC(f_a, dim, limit_a, N)
print ("MC integral a:", v_a[0], "MC estimate variance of a: ",v_a[1])
#************************************************************


#********************* Function B Operation ******************
#Separate Cos(-Pi + 5(x + y)) = -Cos(5x)Cos(5y) + Sin(5x)Sin(5y)
def g_b1(x):
    return 2.93 /math.sqrt(2 * math.pi * 1) * math.exp( - x[0] ** 2/ 2)

def g_b2(x):
    return 2.93 / math.sqrt(2 * math.pi * 1) * math.exp(- x[0] ** 2 / 2)

def f_b1(x):
    return math.cos(5 * x[0])

def f_b2(x):
    return math.sin(5 * x[0])

    # The Cos part
data_b1 = integrate(g_b1, dim, limit_b, N)
#print ("g_a1 integral value: ",np.mean(data_b1))


g_distb1 = inverse_transform_sampling(data_b1,50,1000)
ub1 = []
for i in range(1000):
    ub1.append(random.random())
g_distb1 = g_distb1 * ub1
h_b1 = []
for items in g_distb1:
    h_b1.append(f_b1([items])/g_b1([items]))

g_distb11 = inverse_transform_sampling(data_b1,50,1000)
ub11 = []
```

```
for i in range(1000):
    ub11.append(random.random())
g_distb11 = g_distb11 * ub11
h_b11 = []
for items in g_distb11:
    h_b11.append(f_b1([items])/g_b1([items]))

h_b1 = np.array(h_b1) * np.array(h_b11)

# The Sin part
data_b2 = integrate(g_b2, dim, limit_b, N)
#print ("g_a1 integral value: ",np.mean(data_b1))
g_distb2 = inverse_transform_sampling(data_b2, 50, 1000)
ub2 = []
for i in range(1000):
    ub2.append(random.random())
g_distb2 = g_distb2 * ub2
h_b2 = []
for items in g_distb2:
    h_b2.append(f_b2([items]) / g_b2([items]))

g_distb21 = inverse_transform_sampling(data_b2, 50, 1000)
ub21 = []
for i in range(1000):
    ub21.append(random.random())
g_distb21 = g_distb21 * ub21
h_b21 = []
for items in g_distb21:
    h_b21.append(f_b2([items]) / g_b2([items]))

h_b2 = np.array(h_b2) * np.array(h_b21)

print ('Importance Sampling Value of B: ', np.mean(-h_b1 + h_b2),' Estimate
variance of IS of B: ',np.var(-h_b1 + h_b2)/N)

v_b = integrateMC(f_b, dim, limit_a, N)
print ("MC integral b:", v_b[0], "MC estimate variance of b: ",v_b[1])
#*****************************************************************

#********************* Function C Operation *******************
def g_c1(x):
    return  -1. * (x[0] - 1) ** 2 + 1.33

def f_c1(x):
```

```
    return abs(4 * x[0] - 2)

data_c = integrate(g_c1, dim, limit_c, N)
#print ("g_c1 integral value: ",np.mean(data_c))

g_distc = inverse_transform_sampling(data_c,50,1000)
uc = []
for i in range(1000):
    uc.append(random.random())
g_distc = g_distc * uc
h_c = []
for items in g_distc:
    h_c.append(f_c1([items])/g_c1([items]))

data_c1 = integrate(g_c1, dim, limit_c, N)
g_distc1 = inverse_transform_sampling(data_c1,50,1000)
uc1 = []
for i in range(1000):
    uc1.append(random.random())

g_distc1 = g_distc1 * uc1
h_c1 = []
for items in g_distc1:
    h_c1.append(f_c1([items])/g_c1([items]))

h_c = np.array(h_c) * np.array(h_c1)
print ('Importance Sampling Value of C: ', np.mean(h_c),' Estimate variance of
IS of C: ',np.var(h_c)/N)

v_c = integrateMC(f_c, dim, limit_c, N)
print ("MC integral c:", v_c[0], "MC estimate variance of c: ",v_c[1])
#************************************************************
```

6. SA.py

```
import math
import random
import numpy as np
import matplotlib.pyplot as plt
from deap import benchmarks
try:
    import numpy as np
except:
    exit()
```

```
XMAX = 500
YMAX = 500

alpha = 1.1
def Obj(x,y):
    value = 418.9829 * 2 - x * math.sin(math.sqrt(abs(x))) - y *
math.sin(math.sqrt(abs(y)))
    return value


value = []
N = 100
MarkovLength = 1000 #Iteration time

allroute_x = []
allroute_y = []

for _ in range(N):

    DecayScale = 0.95
    StepFactor = 0.2 #
    Temperature = 1e5
    Tolerance = 1e-5
    #AcceptPoints = 0.0
    iteration = 0

    route_x = []
    route_y = []
    rnd = random.random()
    # Initial solution: PreBextX, preBestY , Current best value: BestX, BestY
    # Prepared solution: Prex, prey
    PreX = -XMAX * random.random()
    PreY = -YMAX * random.random()

    PreBestX = PreX
    PreBestY = PreY

    PreX = -XMAX * random.random()
    PreY = -YMAX * random.random()

    BestX = PreX
    BestY = PreY

    route_x.append(BestX)
```

```
    route_y.append(BestY)
    while Temperature > Tolerance:

        # Choosing the Cooling style
        #Temperature = DecayScale * Temperature #快速降温 For 3.2
        #Temperature = (0.88 ** (iteration + 1) ) * Temperature #Exponential
        #Temperature = Temperature / ( 1 + alpha * math.log( 1 + iteration,math.e))
#logarithmic
        Temperature = Temperature / (1 + 0.1 * iteration) # polynomial
        AcceptPoints = 0.0
        i = 0
        while i < MarkovLength :
            p = 0 # Choose the right range value
            while p == 0:
                NextX = PreX + StepFactor * XMAX * (random.random() - 0.5)
                NextY = PreY + StepFactor * YMAX * (random.random() - 0.5)
                if((NextX >= -XMAX) and (NextX <= XMAX) and (NextY >= -YMAX) and
(NextY <= YMAX)):
                    p = 1

            if(Obj(BestX,BestY) > Obj(NextX,NextY)):
                #Preserve the pervious best solution
                PreBestX = BestX
                PreBestY = BestY

                #Update the best solution
                BestX = NextX
                BestY = NextY
                route_x.append(BestX)
                route_y.append(BestY)

                #Metropolis Process
            if (Obj(PreX,PreY) - Obj(NextX,NextY) > 0):
                PreX = NextX
                PreY = NextY
                AcceptPoints += 1
            else:
                changer = -1. * (Obj(NextX,NextY) - Obj(PreX,PreY))/Temperature
                rnd = random.random()
                p1 = math.exp(changer)
                if p1 > rnd:
                    PreX = NextX
                    PreY = NextY
                    AcceptPoints += 1
```

```
            i = i + 1
        iteration += 1

    value.append(Obj(BestX,BestY))
    allroute_x.append(route_x[:])
    allroute_y.append(route_y[:])
'''
#*************** Question 3.1 and 3.4 ******************
index = []
for i in range(N):
    if value[i] < 0.1:
        index.append(i)
print (index)
ki = {}
#it belongs to minimum value, then find minimum route length
for it in index:
    ki[it] = len(allroute_y[it])

print (ki)
a = min(ki, key=ki.get)
k = len(allroute_y[a])

X = np.arange(-500, 500, 10)
Y = np.arange(-500, 500, 10)
X, Y = np.meshgrid(X, Y)
Z = np.zeros(X.shape)
def schwefel_arg0(sol):
    return benchmarks.schwefel(sol)[0]

for i in range(X.shape[0]):
    for j in range(X.shape[1]):

        Z[i, j] = schwefel_arg0((X[i, j], Y[i, j]))

plt.figure()
contour = plt.contour(X, Y, Z)
plt.colorbar(contour)
plt.title('Scwefel Contours Plot')
plt.xlabel('x')
plt.ylabel('y')
plt.plot(allroute_x[a],allroute_y[a],'ob')
#plt.plot(allroute_x[a][0:int (k/3)],allroute_y[a][0:int(k/3)],'ow')
#plt.plot(allroute_x[a][int(k/3):int( 2 * k/3)],allroute_y[a][int(k/3):int( 2
```

```
* k/3)],'^b')
#plt.plot(allroute_x[a][int(2 * k/3):],allroute_y[a][int(2 * k/3):],'*k')
plt.show()


#*********************************************************
'''

#*************** Question 3.3 Start ************************
plt.figure()
axes = plt.gca()
#axes.set_xlim([0,1400])
#axes.set_ylim([0,30])
plt.hist(value,bins = 40)
plt.xlabel('Converge minimum value')
plt.ylabel('Frequency of corresponding minimum value')
str = "Polynomial Cooling Schedule with "+str(MarkovLength)+ " Iterations"
#str = "Logarithmic Cooling Schedule with "+str(MarkovLength)+ " Iterations"
#str = "Exponential Cooling Schedule with "+str(MarkovLength)+ " Iterations"
plt.title(str)
plt.show()

#*************** Question 3.3 End **************************
```