Project 8

Wenjun Li

April 29, 2020

Question 1

Use a total sample budget of $n = 10000$ samples to obtain Monte Carlo estimates and variances for the following integrals in two dimensions $(x_1 and x_2)$. 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.

a. $exp(\sum_{i=1}^{2} 5|x_i - 5|) for x \leq x_i \leq 1$.

b. $cos(\pi + \sum_{i=1}^{2} 5x_i) for -1 \leq x_i \leq 1$.

Results and Analysis

In question 1, we need to implement Monte Carlo estimation, stratification sampling and importance sampling with a sample budget of n = 1000 to obtain their estimated mean and variances for the given functions. After simulation, we should compare the different Monte Carlo integral estimates, the quality of estimates and their sample variances.

Because we are asked to do simulation in 2-D, we can simplify the functions as $exp(\sum_{i=1}^{2} 5|x_i - 5|) = exp(5(5-x_1+5(5-x_2)))$ for x_1, x_2 $\in$ [0, 1] and $cos(\pi+\sum_{i=1}^{2} 5x_i) = cos(\pi+5x_1+5_2)$ for x_1, x_2 $\in$ [-1, 1]. For Monte Carlo estimates, I sample random variables within [0, 1] and [-1, 1] respectively and then obtain their Monte Carlo estimates directly. For stratified sampling, I set K = 20; for importance sampling, I set the number of importance as 1000. Simulation results are given below:

Monte Carlo: Mean = 2.202e+20; Standard Deviation = 3.367e+19
Stratified Sampling: Mean = 2.060e+20; Standard Deviation = 2.294e+18
Importance Sampling: Mean = 1.863e+20; Standard Deviation = 3.705e+19

As we can see from the above results, we can conclude that 1) Monte Carlo and stratified sampling produce a similar mean while stratified sampling produces a much smaller variance; 2) Importance sampling has a relatively smaller mean, but it has a similar variances as that of Monte Carlo.

Python Code

Listing 1: Code of Question 1

```python
import numpy as np


def my_func_a(x1, x2):
    return np.exp(5*(5-x1) + 5*(5-x2))


def my_func_b(x1, x2):
    return np.cos(np.pi + 5*x1 + 5*x2)


# part (a)
N = 1000
fX = np.random.rand(1, N)
fY = np.random.rand(1, N)
X = my_func_a(fX, fY)
print('Mean is:', str(np.mean(X)))
print(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(K):
    for j in range(K):
        XS = my_func_a((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(2 * np.sqrt(SST))


# importance sampling
N_is = 1000
U = np.random.rand(2, N_is)
X_is = np.log(1 + (np.exp(1) - 1) * U)
T = (np.e-1) ** 2 * np.exp(5 * (5 - X_is[0]) + 5 * (5 - X_is[1]) - (X_is[0] + X_is
    [1]))
print('Mean with importance sampling is:', str(np.mean(T)))
print(2 * np.std(T) / np.sqrt(N))

```

```

# part (b)
fX = np.random.uniform(-1, 1, N)
fY = np.random.uniform(-1, 1, N)
X = my_func_b(fX, fY)
print('Mean is:', str(np.mean(X)))
print(2 * np.std(X) / np.sqrt(N))


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

for i in range(K):
    for j in range(K):
        XS = my_func_b((i + np.random.uniform(-1, 1, int(Nij))) / K, (j + np.
            random.uniform(-1, 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(2 * np.sqrt(SST))


# importance sampling
U = 2 * np.random.rand(2, N_is) - 1
X_is = np.log(1 + U)
T = (np.e-1) ** 2 * np.cos(np.pi + 5 * X_is[0] + 5 * X_is[1] - (X_is[0] + X_is[1])
    )
print('Mean with importance sampling is:', str(np.mean(T)))
print(2 * np.std(T) / np.sqrt(N))
```

Question 2

Let $X_i, i = 1, 2, 3$ be independent exponentials with mean 1. Use Gibbs sampling to estimate the following:

a. $E[X_1 + 2X_2 + 3X_3 | X_1 + 2X_2 + 3X_3 > 15]$.

b. $E[X_1 + 2X_2 + 3X_3 | X_1 + 2X_2 + 3X_3 < 1]$.

Results and Analysis

In this question, we need to use Gibbs sampling to estimate the given expectations. Since Gibbs sampling ensures that samples will converge to the target density if the initial state has some non-zero probability. Also, we can consider a equivalent condition for part (a): $Y_1+2Y_2+3Y_3 > 15$ where $Y_k\ Exp(\frac{1}{k})$ for $k = 1, 2, 3$ since $aX\ Exp(\frac{1}{a})$ if $X\ Exp(1)$. Therefore, I set $X_1 = 2X_2 = 3X_3 = 5$, so that $X_1 + 2X_2 + 3X_3 = 15$. In this way, it will be easier for me to decide whether the inequality condition is satisfied or not. Similarly, I initialize $X_1 = 2X_2 = 3X_3 = 1/3$ for part (b). For simulation details, I use a sample budget of n = 1000 and repeat the simulation for 10 times. The simulation results are given below:

Part (a) $E[X_1 + 2X_2 + 3X_3|X_1 + 2X_2 + 3X_3 > 15] = 17.70$
Part (b) $E[X_1 + 2X_2 + 3X_3|X_1 + 2X_2 + 3X_3 < 1] = 0.0018$

From the results, we can tell that Gibbs sampling is quite useful in conditional sampling and conditional expectations.

Python Code

Listing 2: Code of Question 2

```python
import numpy as np


# part (a)
N = 100
Expectation_hist = []

for trails in range(N):
    # define initial x
    x = [[5], [5], [5]]

    # simulation in 1 trial
    for i in range(1000):
        sum = 0
        temp = 0
        randint = np.random.randint(3)

        for j in range(3):
            if j == randint:
                continue
            sum += (j+1) * x[j][-1]

        # define the rest of (15-sum)
        rest = 15 - sum

        # generate one random variable
        while temp * (randint+1) <= rest:
            temp = np.random.exponential(1)
        x[randint].append(temp)

    # averaged x
    Expectation_hist.append(np.mean(np.mean(x[0]) + 2*np.mean(x[1]) + 3*np.mean(x
        [2])))

print('The averaged Expectation in {} trails is: {}'.format(N, np.mean(
    Expectation_hist)))


# part (b)
Expectation_hist_b = []

for trails in range(N):
    # define initial x
    x_b = [[0.33], [0.33], [0.33]]

    # simulation in 1 trial
    for i in range(100):
        sum = 0
        temp = 0
        randint = np.random.randint(3)

        for j in range(3):
            if j == randint:
                continue
            sum += (j+1) * x_b[j][-1]

        # define the rest of (1-sum)
        rest = 1 - sum

        # generate one random variable
        while temp * (randint+1) > rest:
            temp = np.random.exponential(1)
        x_b[randint].append(temp)

    # averaged x
    Expectation_hist_b.append(np.mean(np.mean(x_b[0]) + 2*np.mean(x_b[1]) + 3*np.
        mean(x_b[2])))

```

```
print('The averaged Expectation in {} trails is: {}'.format(N, np.mean(
    Expectation_hist_b)))
```

Question 3

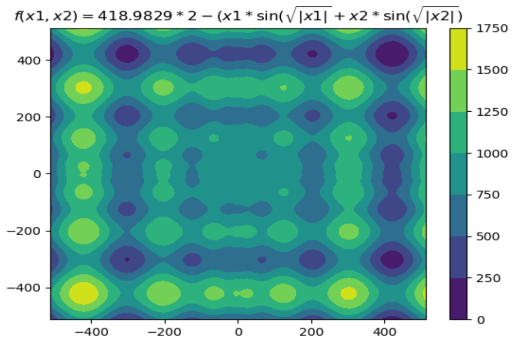
The Schwefel function is a standard optimization benchmark because it has many local minimas and a single global minimum. The Schwefel function is given by

$$f(x) = 418.9829d - \sum_{i=1}^{d} x_i \sin(\sqrt{|x_i|})$$

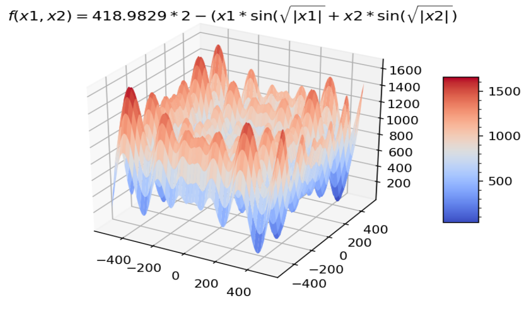
over the hpercube -500 $\leq x_i \leq$ 500 for i = 1, ..., d where d is the dimension. For this problem consider the surface in 2-dimensions (d = 2). Draw a contour plot of the 2-dimensional Schwefel surface. Find the global minimum of the surface using simulated annealing. Begin each simulation at the origin (0, 0). Compare the behavior of your simulation when using exponential, polynomial, and logarithmic cooling schedules. Run your algorithm for with different iteration counts (e.g. N=50, 200, 1000, 10000) and generate a histogram showing the computed global minimum for each case. For the simulation that achieves the best estimate of the global minimum overlay the 2-D sample path on a contour plot of the surface and comment on the behavior of the search.

Results and Analysis

In this question, we need to first plot a 2-D contour plot and a 3-D plot of the 2-D Schwefel function, which is $f(x) = 418.9829 * 2 - (x_1 * \sin(\sqrt{|x_1|} + x_2 * \sin(\sqrt{|x_2|})$. The plots are shown in Figure 1.



(a) 2-D



(b) 3-D

Figure 1: 2-D and 3-D Contour Plot of the Schwefel Function

After draw the plots, we need to find the global minimum of the surface using simulated annealing starting from the origin (0, 0) with three different cooling schedules, i.e. exponential, polynomial and logarithmic cooling schedules.

Python Code

Listing 3: Code of Question 3

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

from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter


def f(x):
    fx = 418.9829*2 - (x[0]*np.sin(np.sqrt(abs(x[0]))) + x[1]*np.sin(np.sqrt(abs(x
        [0]))))
    return fx


N_r = 512
x = np.linspace(-N_r, N_r, 100)
y = np.linspace(-N_r, N_r, 100)
X, Y = np.meshgrid(x, y)
Z = 418.9829*2 - (X*np.sin(np.sqrt(abs(X))) + Y*np.sin(np.sqrt(abs(Y))))

# contour plot
plt.figure(num=None, dpi=100)
plt.contourf(X, Y, Z)
plt.title('$f(x1,x2) = 418.9829*2 - (x1*\sin(\sqrt{|x1|} + x2*\sin(\sqrt{|x2|})$')
plt.colorbar()
plt.savefig('Contour_Plot_2D.png')
plt.show()

# 3-D plot
cplot = plt.figure(num=None, dpi=150)
ax = cplot.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm)
cbar = cplot.colorbar(surf, shrink=0.5, aspect=5)
cbar.minorticks_on()
plt.title('$f(x1,x2) = 418.9829*2 - (x1*\sin(\sqrt{|x1|} + x2*\sin(\sqrt{|x2|})$')
plt.savefig('Contour_Plot_3D.png')
plt.show()


# find 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). Use the Euclidean distance to estimate the distance between two cities given (x, y)-coordinates:

$$d(c_1, c_2 = \sqrt{(x_1 - x_2)^2 + (x_1 - x_2^2))}$$

Begin your tour in Sacramento, California. Initialize your simulation with an arbitrary trip through the remaining state capitols. Choose new candidate circuits by randomly choosing two cities and then reversing the path between them. Estimate the distance of the shortest path that visits each of the 48 state capital cities. Plot the best path on the x-y axis and comment on the graphical path. Generate a plot of the total tour distance as a function of the simulation time (step number). Comment on the number of iterations required to estimate this path and on the convergence rate of the estimate toward your minimum.

Results and Analysis

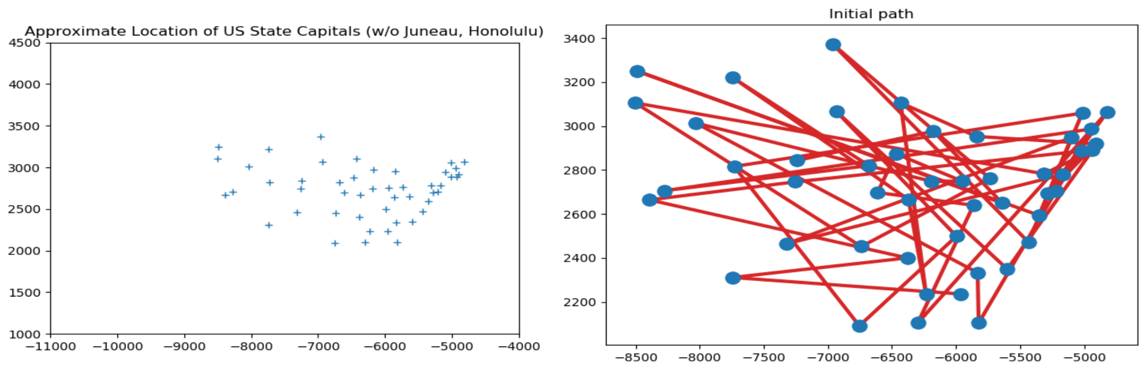
In question 4, we are going to estimate the shortest path of Traveling Salesman Problem (TSP). Since TSP is a NP-hard problem, we can only try to iteratively find shorter path. In this question, we will use simulated annealing. Simulated annealing starts with an initial path p^0 with cost (route length) $l(p^0)$. Then, we will have path p^t and cost $l(p^t)$ at time t. During the iterative procedure, simulated annealing tries to find the next candidate path p' by randomly swapping the order of two cities in p^t. And we accept $p^{t+1} = p'$ only if $l(p') \leq l(p^t)$ and remain $p^{t+1} = p^t$ if $l(p') > l(p^t)$. But, we might stuck in some local optimal path because we are too 'greedy' and never accept any worse paths. Therefore, we should sometimes 'take risk' by accepting $p^{t+1} = p'$ though p' is worse than p^t. The idea of taking risk makes this algorithm less greedy because it can find some better path by visiting bad places temporarily.

The algorithm of solving TSP with simulated annealing can be described as below:

1. Initialize the path p^0 and calculates its cost $l(p^0)$.

2. Generate a candidate path p' by swapping two cities in p^t randomly and iteratively find the next path p^{t+1} from p^t via:
if $l(p') \leq l(p^t)$, then $p^{t+1} = p'$;
else $p^{t+1} = \begin{cases} p' withprobability \text{q} \\ p^t withprobability 1-\text{q} \end{cases}$
$whereqdecreasesastheiterationsgoesandcommonchoicesofa$are :q $= (\text{t}+1)^{\frac{l(p^t)-l(p')}{c}}$ and $q = e^{\frac{l(p^t)-l(p')}{c*a^t}}$ with a $\in (0, 1)$.

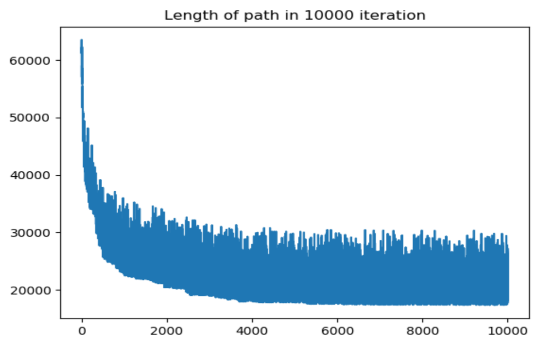
Before solving this TSP problem, let's look at the approximate location map of US state capitals, which is shown in Figure 2. The initial path is also given in Figure 2, which is a random path covering all state capitals. The length of initial path is 61429 mils, which is clearly much longer than the shortest path in this TSP problem.



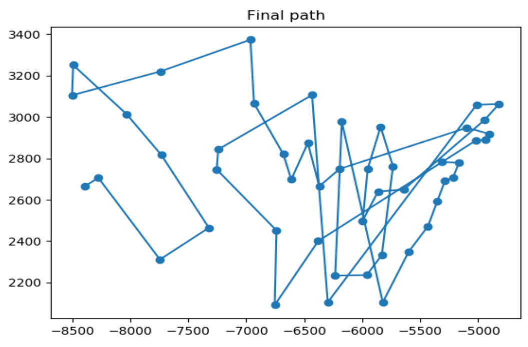
(a) Location Map of US State Capitals (b) Initial Path

Figure 2:

After I implement the simulated annealing algorithm on this TSP problem, the length path $l(p)$ start to decrease quickly and remarkably. As is shown in Figure 3., the path length drop quickly in the beginning and start to converge after around 4000 iterations. The final path length (shortest path length) is around 17500 miles, which is only one fourth of the initial path length. Also, we can derive this conclusion from the overlaid final path in Figure 3. (b). Starting from Sacramento, California, the final path visits all 48 state capitals once and the total length is much shorter than that of the initial path.



(a) Location Map of US State Capitals



(b) Initial Path

Figure 3:

Python Code

Listing 4: Code of Question 4

```
import numpy as np
from sklearn.metrics import pairwise_distances
import matplotlib.pyplot as plt
from scipy.spatial import distance


N_cities = 48
cities_coord = []

# read file and store city coordinates
# ...... Note: I have to delete this read file part, OR, LaTex won't compile
    properly ......


# remove Juneau, Alaska (index = 1) and Honolulu, Hawaii (Index = 10)
del cities_coord[10]     # delete Honolulu
del cities_coord[1]      # delete Juneau
cities_coord = np.array(cities_coord)
print(cities_coord.shape)

# draw capital map
x_coord_capital = cities_coord[:, 0]
y_coord_capital = cities_coord[:, 1]
plt.figure()
plt.plot(x_coord_capital, y_coord_capital, '+')
plt.xlim(-11000, -4000)
plt.ylim(1000, 4500)
plt.title('Approximate Location of US State Capitals (w/o Juneau, Honolulu)')
plt.show()


# distance matrix: euclidean dist between every point
dist_mat = pairwise_distances(cities_coord, cities_coord, metric='euclidean')

# Parameters
num_iter = 100  # number of iterations
c = 1
# 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 += 1
    # Create path p2 by randomly swap two cities index of two cities for the new
        path
    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 = np.power((1 + iter_count), ((p_len - p_len2) / c))

    # change paths if new path is shorter than previous
    if p_len2 - p_len <= 0:
        #         p[:] = p2[:]
        p = np.copy(p2)
        p_len = np.copy(p_len2)

    #  or change paths with probability thres
    else:
        if np.random.rand() <= thresh:
            p = np.copy(p2)
            p_len = np.copy(p_len2)

    # bookkeeping
    pathHistory[iter_count - 1][0:len(p2)] = p2
    lenHistory.append(p_len2)
    thresh_ar.append(thresh)


# plot length v.s. iterations
plt.figure(num=None, dpi=100)
plt.plot(lenHistory)
plt.title('Length of path in {} iteration'.format(num_iter))
plt.show()

# plot final path
ind_f = pathHistory[-1, :].astype(int)
x_coord_f = cities_coord[ind_f, 0]
y_coord_f = cities_coord[ind_f, 1]
plt.figure(num=None, dpi=100)
plt.title('Final path')
plt.plot(x_coord_f, y_coord_f, '-o')
plt.show()
```

Below is the read file part in the python codes.

with open(`uscap_xy.txt`, 'r') as f:
for eachline in f:
temp = eachline.split()
temp = list(map(float, temp))
cities$_c oord.append(temp)$