**EE 511**

**PROJECT # 5**

BY

PRANAV GUNDEWAR

USC ID: 4463612994

EMAIL: [gundewar@usc.edu](mailto:gundewar@usc.edu)

**Problem 1 MCMC for Sampling**

Monte Carlo methods are a broad class of [computational](https://en.wikipedia.org/wiki/Computation) [algorithms](https://en.wikipedia.org/wiki/Algorithm) that rely on repeated [random sampling](https://en.wikipedia.org/wiki/Random_sampling) to obtain numerical results. Their essential idea is using [randomness](https://en.wikipedia.org/wiki/Randomness) to solve problems that might be deterministic in principle. They are often used in [physical](https://en.wikipedia.org/wiki/Physics) and [mathematical](https://en.wikipedia.org/wiki/Mathematics) problems and are most useful when it is difficult or impossible to use other approaches. Monte Carlo methods are mainly used in three distinct problem classes: [optimization](https://en.wikipedia.org/wiki/Optimization), [numerical integration](https://en.wikipedia.org/wiki/Numerical_integration), and generating draws from a [probability distribution](https://en.wikipedia.org/wiki/Probability_distribution).

**Summary:**

* I implemented a Metropolis-Hastings algorithm to generate samples from this distribution and plotted sample paths for different initial points.
* I have plotted sample paths for the algorithm using Cauchy, normal and logistic PDFs.

**Approach:**

* Metropolis–Hastings algorithm is a Markov chain Monte Carlo (MCMC) method for obtaining a sequence of random samples from a probability distribution for which direct sampling is difficult.
* This sequence can be used to approximate the distribution or to compute an integral such as an expected value.
* Generate: Generate a candidate x for the next sample by picking from the distribution.
* Calculate acceptance ratio alpha.

\alpha = \text{min} \left (1,\frac{p(x^*)}{p(x^{(t-1)})} \times c\right ) 

proposal correction factor  c = \frac{q(x^{(t-1)} | x^*) }{q(x^*|x^{(t-1)})}

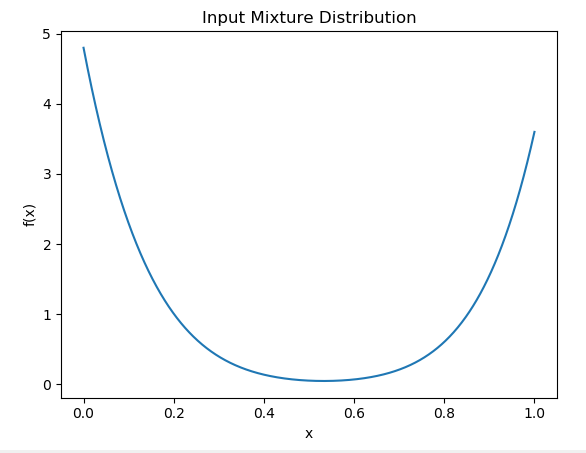
* Generate uniform random number u between 0 and 1.

If u <= alpha accept candidate. Else reject candidate.

**Result and Analysis:**

I have used uniform and Cauchy pdf as proposal because of their symmetric nature. I varied the variance of proposal pdfs and found that low variance **PDFs converge faster** than PDFs with high variance. From the plots shown below, we can conclude that samples sampled using proposal PDF closely resembles the input mixture distribution. The generated samples do converge to the mixture.

For low variance PDFs, I got convergence very early than when I made variance high.



**Figure 1:** Input Mixture Distribution

|  |  |
| --- | --- |
|  |  |

**Figure 2:** MCMC Sampling using Cauchy Distribution

|  |  |
| --- | --- |
|  |  |

**Figure 3:** MCMC Sampling using Normal Distribution

|  |  |
| --- | --- |
|  |  |

**Figure 4:** MCMC Sampling using Logistic Distribution

**Problem 2 MCMC for Optimization**

**Summary:**

* Plotted a contour plot of the surface for the 2-D surface
* Implement a simulated annealing procedure to find the global minimum of this surface using different types of temperature cooling.
* I plotted a histogram of the function minima for all procedures.

**Approach:**

The annealing schedule is defined by the call temperature(*r*), which should yield the temperature to use, given the fraction *r* of the time budget that has been expended so far**.**

* Let *s* = *s*0
* For *k* = 0 through *k*max (exclusive):
  + *T* ← temperature(*k* ∕ *k*max)
  + Pick a random neighbour, *s*new ← neighbour(*s*)
  + If *P*(*E*(*s*), *E*(*s*new), *T*) ≥ random(0, 1):
    - *s* ← *s*new
* Output: the final state *s*

|  |  |  |  |
| --- | --- | --- | --- |
| **$\displaystyle g(\vec{x})$** | **$\displaystyle =$** | **$\displaystyle (2\pi T)^{-\frac{D}{2}}\exp\left( -\frac{\left(\Delta{\vec{x}}\right)^2}{2T}\right),$** | **(5.13)** |
| **$\displaystyle h(\vec{x})$** | **$\displaystyle =$** | **$\displaystyle \frac{1}{1+\exp\left( \frac{E_{k+1}-E_k}{T}\right)},$** | **(5.14)** |
| **$\displaystyle T(k)$** | **$\displaystyle =$** | **$\displaystyle \frac{T_0}{\ln{k}}$** |  |

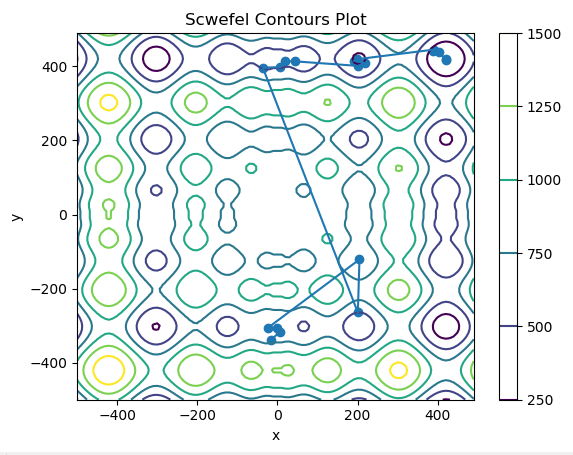
I have used logarithmic, exponential and polynomial temperature cooling functions for annealing.

**Result and Analysis:**

|  |  |
| --- | --- |
|  |  |

**Figure 5:** Contour plot for Schwefel Function

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 22 steps to get the expected value.



**Figure 6:** Contour plot for Routing Points

The final path has been plotted. We can observe finally algorithm achieve global minima at location around (400,320).

|  |  |
| --- | --- |
|  |  |
|  |  |

**Figure 7:** Exponential Cooling Schedule

|  |  |
| --- | --- |
|  |  |
|  |  |

**Figure 8:** Logarithmic Cooling Schedule

|  |  |
| --- | --- |
|  |  |
|  |  |

**Figure 9:** Polynomial Cooling Schedule

From above figures, polynomial and logarithmic has better convergence. I have gotten more zeros as minima in those. The best path uses very least and optimum curve.

**Test estimate integration-**

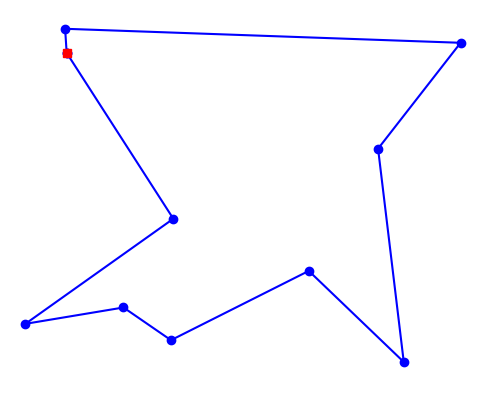
The last function is very complex and hard to find a proper pdf or stratified region. I use uniform stratification sampling method which uses uniform PDF to calculate random numbers between -5 to 5

**Problem 3 MCMC for Optimal Path**

**Summary:**

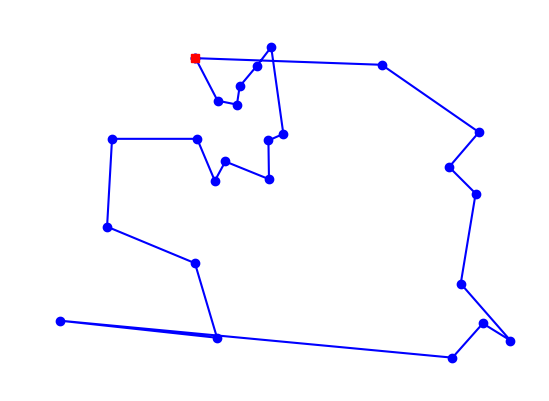
* Plotted a contour plot of the surface for the 2-D surface
* Implement a simulated annealing procedure to find the global minimum of this surface.

**Result and Analysis:**





**Figure 10:** 10 City tours



**Length = 3556 for 25 cities**

I coded for 10 cities and 25 cities. By changing different parameters for cooling schedule, I vary the rate of convergence.

**However, due to limitation of my laptop, it freezes for cities more than 50 cities and hence I have only included plots until 25**.

**CODES:**

"""

@author: Pranav Gundewar

Project #5: Optimization & Sampling via MCMC

Q1- MCMC for Sampling

"""

#%%

#Importing Libraries

import matplotlib.pyplot as plt

import numpy as np

from scipy.stats import beta, cauchy, norm, logistic

#%%

def function(x):

"""

Define Input mixture Distribution to generate samples

unnormalized density function

"""

return (0.6\*beta.pdf(x, 1, 8) + 0.4\*beta.pdf(x, 9, 1))

def proposal(x):

"""

This is symmetric proposal PDF to plot the sample path

"""

# return cauchy.rvs(loc=x, scale=0.15, size=1)

# return norm.rvs(loc=x, scale=0.2, size=1)

return logistic.rvs(loc=x, scale=1, size=1)

def metropolis(function, proposal, old):

"""

basic metropolis algorithm, needs symmetric proposal distribution.

"""

new = proposal(old)

alpha = np.min([function(new)/function(old), 1])

u = np.random.uniform()

# \_cnt\_ indicates if new sample is used or not.

cnt = 0

if (u <= alpha):

old = new

cnt = 1

return old, cnt

def run\_chain(f, proposal, initial, n):

"""

f:unnormalized density function to sample

proposal:proposal distirbution

start:initial start of the Markov Chain

n:length of the chain

"""

count = 0

samples = [initial]

# while (count != n-1):

## for i in range(n):

# initial, c = metropolis(f, proposal, initial)

# count = count + c

# if c==1:

# samples.append(initial)

# return samples

for i in range(n):

initial, c = metropolis(f, proposal, initial)

count = count + c

if i%1 is 0:

samples.append(initial)

return samples, count

#%%

#x = np.linspace(0, 1, 1000)

#plt.figure(1)

#plt.plot(x,function(x))

#plt.xlabel('x'); plt.ylabel('f(x)')

#plt.title('Input Mixture Distribution')

#%%

while True:

x0 = np.random.uniform()

if function(x0) != 0:

break

samples = run\_chain(function, proposal, x0, 50000)

#%%

plt.figure(2)

plt.plot(samples[0],function(samples[0]),'o')

plt.xlabel('x'); plt.ylabel('f(x)')

plt.title('Sample Path for Logistic Proposal PDF')

#%%

plt.figure(3)

#plt.plot(samples[0],function(samples[0]))

#plt.xlabel('x'); plt.ylabel('f(x)')

#plt.title('Sample Path for Logistic Proposal PDF')

"""

@author: Pranav Gundewar

Project #5: Optimization & Sampling via MCMC

Q2- MCMC for Optimization

"""

#%%

# Importing Libraries

import math

import random

import numpy as np

import matplotlib.pyplot as plt

from deap import benchmarks

global str

X = 500

Y = 500

alpha = 1.1

def schwefel(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 = 100 #Iteration time

final\_x = []

final\_y = []

for \_ in range(N):

Step = 0.1

Temperature = 1e5

tol = 1e-5

iteration = 0

route\_x = []

route\_y = []

rnd = random.random()

OldX = -X \* random.random()

OldY = -Y \* random.random()

BestX = OldX

BestY = OldY

route\_x.append(BestX)

route\_y.append(BestY)

while Temperature > tol:

# Choosing the Cooling style

#Temperature = (0.88 \*\* (iteration + 1) ) \* Temperature #Exponential

# Temperature = Temperature / ( 1 + alpha \* math.log( 1 + iteration,math.e)) #logarithmic

Temperature = Temperature / (1 + 0.15\*iteration) # polynomial

accept = 0.0

i = 0

while i < MarkovLength :

p = 0 # Choose the right range value

while p == 0:

NewX = OldX + Step \* X \* (random.random() - 0.5)

NewY = OldY + Step \* Y \* (random.random() - 0.5)

if((NewX >= -X) and (NewX <= X) and (NewY >= -Y) and (NewY <= Y)):

p = 1

if(schwefel(BestX,BestY) > schwefel(NewX,NewY)):

#Preserve the pervious best solution

PreBestX = BestX

PreBestY = BestY

#Update the best solution

BestX = NewX

BestY = NewY

route\_x.append(BestX)

route\_y.append(BestY)

#Metropolis Process

if (schwefel(OldX,OldY) - schwefel(NewX,NewY) > 0):

OldX = NewX

OldY = NewY

accept += 1

else:

changer = -1. \* (schwefel(NewX,NewY) - schwefel(OldX,OldY))/Temperature

rnd = random.random()

p1 = math.exp(changer)

if p1 > rnd:

OldX = NewX

OldY = NewY

accept += 1

i = i + 1

iteration += 1

value.append(schwefel(BestX,BestY))

final\_x.append(route\_x[:])

final\_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)

key = {}

#%%

#it belongs to minimum value, then find minimum route length

for i in index:

key[i] = len(final\_y[i])

print (key)

#%%

a = min(key, key=key.get)

#k = len(final\_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(final\_x[a],final\_y[a])

plt.scatter(final\_x[a],final\_y[a])

#plt.plot(final\_x[a][0:int (k/3)],final\_y[a][0:int(k/3)],'ow')

#plt.plot(final\_x[a][int(k/3):int( 2 \* k/3)],final\_y[a][int(k/3):int( 2 \* k/3)],'^b')

#plt.plot(final\_x[a][int(2 \* k/3):],final\_y[a][int(2 \* k/3):],'\*k')

plt.show()

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#%%

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Question 3.3 Start \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

plt.figure()

axes = plt.gca()

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 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

"""

@author: Pranav Gundewar

Project #5: Optimization & Sampling via MCMC

Q1- MCMC for Sampling

"""

import itertools

import random

import matplotlib.pyplot as plt

import time

import numpy as np

import math

def alltours\_tsp(cities):

"Generate all possible tours of the cities and choose the shortest tour."

return shortest\_tour(alltours(cities))

def shortest\_tour(tours):

"Choose the tour with the minimum tour length."

return min(tours, key=tour\_length)

def alltours(cities):

"Return a list of tours, each a permutation of cities, but each one starting with the same city."

start = first(cities)

return [[start] + Tour(rest)

for rest in itertools.permutations(cities - {start})]

def first(collection):

"Start iterating over collection, and return the first element."

return next(iter(collection))

Tour = list # Tours are implemented as lists of cities

def tour\_length(tour):

"The total of distances between each pair of consecutive cities in the tour."

return sum(distance(tour[i], tour[i-1])

for i in range(len(tour)))

# Cities are represented as Points, which are a subclass of complex numbers

class Point(complex):

x = property(lambda p: p.real)

y = property(lambda p: p.imag)

City = Point

def distance(A, B):

"The distance between two points."

return abs(A - B)

def Cities(n, width=900, height=600, seed=42):

"Make a set of n cities, each with random coordinates within a (width x height) rectangle."

random.seed(seed \* n)

return frozenset(City(random.randrange(width), random.randrange(height))

for c in range(n))

def plot\_tour(tour):

"Plot the cities as circles and the tour as lines between them. Start city is red square."

start = tour[0]

plot\_lines(list(tour) + [start])

plot\_lines([start], 'rs') # Mark the start city with a red square

def plot\_lines(points, style='bo-'):

"Plot lines to connect a series of points."

plt.plot([p.x for p in points], [p.y for p in points], style)

plt.axis('scaled'); plt.axis('off')

def plot\_tsp(algorithm, cities):

"Apply a TSP algorithm to cities, plot the resulting tour, and print information."

# Find the solution and time how long it takes

t0 = time.clock()

tour = algorithm(cities)

t1 = time.clock()

assert valid\_tour(tour, cities)

plot\_tour(tour); plt.show()

print("{} city tour with length {:.1f} in {:.3f} secs for {}"

.format(len(tour), tour\_length(tour), t1 - t0, algorithm.\_\_name\_\_))

def valid\_tour(tour, cities):

"Is tour a valid tour for these cities?"

return set(tour) == set(cities) and len(tour) == len(cities)

def nn\_tsp(cities):

"""Start the tour at the first city; at each step extend the tour

by moving from the previous city to the nearest neighboring city, C,

that has not yet been visited."""

start = first(cities)

tour = [start]

unvisited = set(cities - {start})

while unvisited:

C = nearest\_neighbor(tour[-1], unvisited)

tour.append(C)

unvisited.remove(C)

return tour

def nearest\_neighbor(A, cities):

"Find the city in cities that is nearest to city A."

return min(cities, key=lambda c: distance(c, A))

#plot\_tsp(alltours\_tsp, Cities(10, seed=332))

#plot\_tsp(nn\_tsp, Cities(40, seed=332))

def reverse\_segment(input\_tour, i, j):

"Reverse segment tour[i:j] of a tour"

input\_tour[i:j] = reversed(input\_tour[i:j])

def swap\_cities(input\_tour, i, j):

"Swap two cities at index i and j in a tour"

# save city1

city1 = input\_tour[i]

# save city2

city2 = input\_tour[j]

new\_tour = input\_tour[:]

# swap

new\_tour[j] = city1

new\_tour[i] = city2

return new\_tour

def change\_tour(input\_tour):

"Change a tour for tsp iteration"

indices = range(len(input\_tour))

# take two random indices to swap

c1 = np.random.choice(indices)

c2 = np.random.choice(indices)

new\_tour = change\_path(input\_tour, c1, c2)

return new\_tour

change\_path=swap\_cities

# We've constructed our own simulated annealing function for tsp but we don't

# really need to make any changes. So we'll just comment the regular sa :-)

def sa\_tsp(energyfunc, initials, epochs, tempfunc, iterfunc, proposalfunc):

"""Run simulated annealing on a tsp."""

# Accumulate results in the same form as initals

accumulator=[]

# Our initial state is in initials['solution']

best\_solution = old\_solution = initials['solution']

# Our initial temperature is in initials['T']

T=initials['T']

# Our initial length (i.e. number of iterations per epoch)

# is in initals['length']

length=initials['length']

# initialize the energy of our current state by running the

# energy function on our initial solution

best\_energy = old\_energy = energyfunc(old\_solution)

# keep track of accepted proposals and total iterations

accepted=0

total=0

for index in range(epochs):

#print("Epoch", index)

# if we're past the first index, we need

# to update our cooling schedule and iteration

# schedule

if index > 0:

T = tempfunc(T)

length=iterfunc(length)

#print("Temperature", T, "Length", length)

# run through the iterations for each epoch

for it in range(length):

# keep track of total proposals

total+=1

# get a new proposal and calculate its energy

new\_solution = proposalfunc(old\_solution)

new\_energy = energyfunc(new\_solution)

# Use a min here as you could get a "probability" > 1

alpha = min(1, np.exp((old\_energy - new\_energy)/T))

if ((new\_energy < old\_energy) or (np.random.uniform() < alpha)):

# Accept proposed solution

accepted+=1.0

accumulator.append((T, new\_solution, new\_energy))

# we have a new candidate for optimum (minimum)

if new\_energy < best\_energy:

# Replace previous best with this one

best\_energy = new\_energy

best\_solution = new\_solution

best\_index=total

best\_temp=T

old\_energy = new\_energy

old\_solution = new\_solution

else:

# Keep the old stuff

accumulator.append((T, old\_solution, old\_energy))

best\_meta=dict(index=best\_index, temp=best\_temp)

print("frac accepted", accepted/total, "total iterations", total, 'bmeta', best\_meta)

return best\_meta, best\_solution, best\_energy, accumulator

initial\_cities = Cities(25)

initial\_tour = list(initial\_cities)

length\_func1 = lambda temperature: np.max((np.floor(np.sqrt(temperature)).astype(int),1))

length\_func2 = lambda length: max(int(math.ceil(1.2\*length)), 10)

length\_func = length\_func1

temp\_func = lambda t: 0.8\*t

init\_length = length\_func(100)

inits=dict(solution=initial\_tour, length=init\_length, T=3.0)

print(inits)

bmeta, bs, be, out = sa\_tsp(tour\_length, inits, 10000, temp\_func, length\_func, change\_tour);

nn\_sol = nn\_tsp(initial\_cities)

plot\_tour(nn\_sol)

plot\_tour(bs)