Hrishikesh Hippalgaonkar
hippalga@usc.edu

EE 511 Simulation Methods for Stochastic Systems
Project #5

https://github.com/hrishi121/Simulation-methods-for-Stochastic-Systems

❖ Metropolis-Hastings algorithm:

For a probability density π, which is called the *'target'* , and defined on a state space X the Metropolis–Hastings algorithm proposes a procedure to construct a Markov chain on X that is ergodic and stationary with respect to π. Thus, if X(t) ~ π(x), then X(t+1) ~ π(x)—and that therefore converges in distribution to π.

The algorithm can be defined as follows:

1. Choose a *proposal pdf* $q(\cdot\,|x_t)$. This proposal pdf does not have to be reversible
2. Choose an initial sample X_0 such that $f(X_0) \neq 0$
3. Generate a sample $y \sim q(\cdot\,|x_t)$
4. Calculate the acceptance probability

$$\alpha\,(x_t, y) = min\left[1, \quad \frac{f_x(y) \times q(x_t|y)}{f_x(x_t) \times q(y|x_t)}\right]$$

5. Sample U~(0,1)
6. Accept $X_{t+1} \leftarrow y$ if $U < \alpha$, else reject y
7. Repeat from step 3, and continue until it converges to π^*

1. Problem 1

[MCMC for Sampling]

The random variable X has a mixture distribution: 60% in a Beta(1,8) distribution and 40% in a Beta(9,1) distribution.

i. Implement a Metropolis-Hastings algorithm to generate samples from this distribution.

ii. Run the algorithm multiple times from different initial points. Plot sample paths for the algorithm.

Can you tell if/when the algorithm converges to its equilibrium distribution?

Plot sample paths for the algorithm using different proposal pdfs. Comment on the effect of low variance vs high-variance proposal pdfs on the behaviour of your algorithm.

Code:

```
import warnings
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import beta, norm, cauchy, ks_2samp
```

```
def f(x):
    # Define the input mixture distribution
    a1 = 1; b1 = 8; a2 = 9; b2 = 1
    y = (0.6 * beta.pdf(x, a1, b1)) + (0.4 * beta.pdf(x, a2, b2 ))
    return y

def q1(xt):
    # Define a proposal pdf q(x)
    # This function returns a randomly generated sample
    # from the defined Cauchy distribution
    y = cauchy.rvs(loc = xt, scale=0.2, size=1)
    return y

def q2(xt):
    # Define a proposal pdf q(x)
    # This function returns a randomly generated sample
    # from the defined Normal distribution
    y = norm.rvs(loc = xt, scale = 0.5)
    return y

def plots(sample, title):
    y1 = f(sample)
    plt.figure(2)
    plt.plot(sample, y1, 'bo'); plt.xlabel('x'); plt.ylabel('y=f(x)')
    plt.title('Samples drawn from the mixture using MCMC sampling ('+title+')')

    plt.figure(3)
    plt.plot(sample, y1); plt.xlabel('x'); plt.ylabel('y=f(x)')
    plt.title('Sample path for samples drawn')

def fit_test(samples):
    y1 = f(samples)
    rvs1 = norm.pdf(samples)
    rvs2 = cauchy.pdf(samples)
    d, p_value = ks_2samp(y1, rvs1)
    print('The p-value give by the K-S test:', p_value)


# Try to avoid writing this line the program as this shuts
# down few important too
warnings.filterwarnings('ignore')

x = np.linspace(-0.1, 1.1, num = 1000)
y = f(x)
plt.figure(1)
plt.plot(x, y) ; plt.xlabel('x') ; plt.ylabel('y = f(x)')
plt.title('Graph of the mixture distribution')

while True:
    x0 = np.random.rand()
    if f(x0) != 0:
        break

n_samples = 500
sample = np.empty([n_samples])
sample[0] = x0
n_count = 0
while True:
    if (n_count == 0):
        y = q2(sample[0])
    else:
        y = q2(sample[n_count - 1])

    if (y>=0 and y<=1):
```

```
                ratio = f(y) / f(sample[n_count - 1])
                # Acceptance probability
                alpha = min(1, ratio)

                u = np.random.rand()
                if (u <= alpha):
                    n_count = n_count + 1
                    #print(n_count)
                    sample[n_count] = y

            if (n_count == n_samples - 1):
                break

        #plots(sample, 'Cauchy distribution')
        plots(sample, 'Normal distribution')
fit_test(sample)
```

Result:

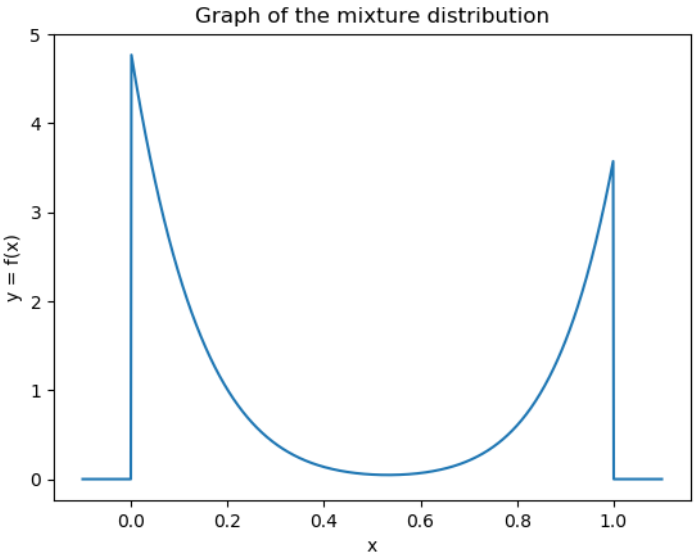
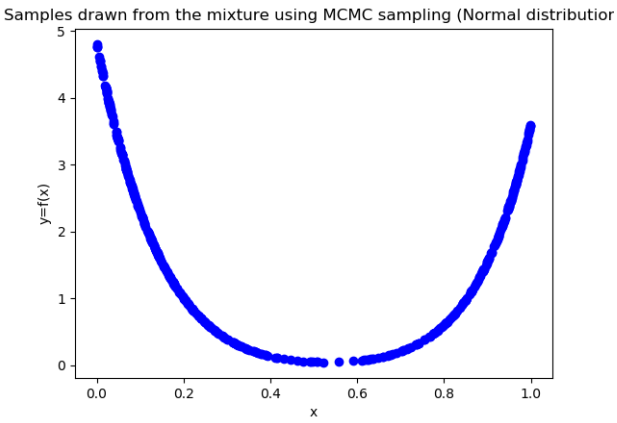
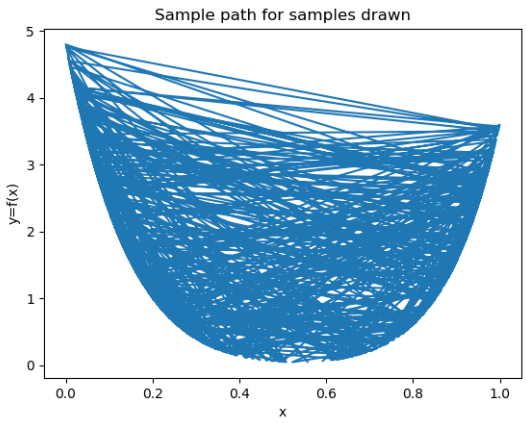


Figure 1. Graph of the actual distribution

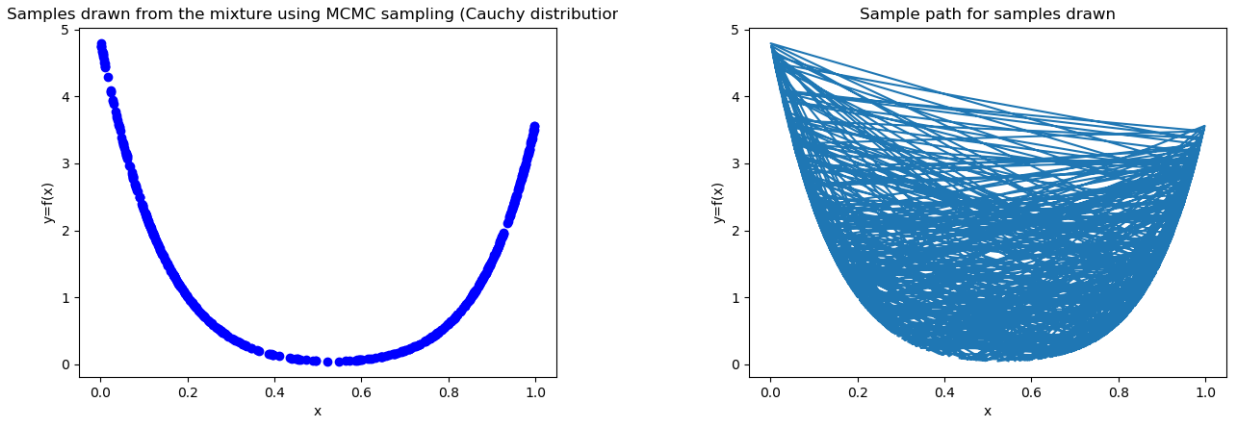


(a) Graph of the samples drawn



(b) Graph of the sample path

Figure 2. Samples draw from the mixture using Normal distribution as proposal pdf



(a) Graph of the samples drawn **(b) Graph of the sample path**

Figure 3. Samples draw from the mixture using Cauchy distribution as proposal pdf

It can be observed in Figure 2a and Figure 3a that the samples drawn almost resemble the graph of the of the original function

If the variance of the proposal pdf is very low, the sampling will become very inefficient and it will take long time to explore the whole parameter space.

On the other hand, if the variance of the proposal pdf is large, the algorithm will reject almost all the samples. Thus, the variance of the proposal pdf must be optimal, so as to explore the parameter space as well as not reject large number of samples.

❖ Simulated Annealing:

Simulated annealing is an optimisation metaheuristic where the goal is to find the global minimum/maximum of a function in a large search space. Simulated makes use of the MCMC sampling and a cooling schedule to find the location of the maxima or minima

The algorithm for simulated annealing is as follows:

1. Choose a *proposal pdf* $q(\cdot\,|x_t)$. This proposal pdf must be reversible. Also, set a cooling schedule $T(t) = g(T_0, t)$
2. Choose an initial sample X_0 such that $f(X_0) \neq 0$
3. Generate a sample $y \sim q(\cdot\,|x_t)$
4. Calculate the acceptance probability

$$\alpha\,(x_t, y) = min\left[1,\ \ exp\left(\frac{g(x_t) - g(y)}{T(t)}\right)\right]$$

5. Sample $U\sim(0,1)$
6. Accept $X_{t+1} \leftarrow y$ if $U < \alpha$, else reject y
7. Repeat from step 3, and continue until it converges to the minima

2. Problem 2

[MCMC for Optimization]

The n-dimensional Schwefel function

is a very bumpy surface with many local critical points and one global minimum. We will explore the surface for the case n=2 dimensions.

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

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

iii. Explore the behaviour 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.

iv. 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.

Code:

```
# -*- coding: utf-8 -*-
"""
Created on Tue May 1 17:06:51 2018

EE511 Project #5, Prof. Osonde Osoba, Spring 2018
@author: Hrishikesh Hippalgaonkar
Tested in Python 3.6.3 :: Anaconda custom (64-bit), Windows 10
Tested in Python 3.5.4 :: Anaconda custom (64-bit), Windows 10

Question 2.
Markov Chain Monte Carlo : Simulated Annealing

"""

import numpy as np
import math
import sys
from math import exp, log
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
from scipy.stats import beta, norm, cauchy
from scipy.stats import multivariate_normal as mvn


def f(x1, x2):
    '''
    Define the input Schwefel function

    '''
    y = 418.9829 * 2 - (    x1*np.sin(np.sqrt(abs(x1)))
                          + x2*np.sin(np.sqrt(abs(x2)))     )
    return y
```

```
def q(x1, x2):
    '''
    Define a proposal pdf q(x)

    '''

    q1 = np.empty([2])
    #q1[0] = norm.rvs(loc = x1, scale = 1)
    #q1[1] = norm.rvs(loc = x2, scale = 1)

    q1[0] = cauchy.rvs(loc = x1, scale=0.2, size=1)
    q1[1] = cauchy.rvs(loc = x2, scale=0.2, size=1)
    return q1

def t1(i):
    # Cooling function : lograthmic
    t = 500/(1 + log(i + 1))
    return t


def t2(i):
    # Cooling function : exponential
    t = 0.8*i
    return t

def t3(i):
    # Cooling function : linear
    t = 500/(1 + 2*i)
    return t

def t4(i):
    # Cooling function : quadratic
    t = 500/(1 + 2*i*i)
    return t


def updt(total, progress):
    """
    Displays or updates a console progress bar.

    Original source: https://stackoverflow.com/a/15860757/1391441
    """
    barLength, status = 20, ""
    progress = float(progress) / float(total)
    if progress >= 1.:
        progress, status = 1, "\r\n"
    block = int(round(barLength * progress))
    text = "\rRuns: [{}] {:.0f}% {}".format(
        "#" * block + "-" * (barLength - block), round(progress * 100, 0),
        status)
    sys.stdout.write(text)
    sys.stdout.flush()


itf = lambda length: math.ceil(1.2*length)


x1_start = 0; x2_start = 0
length = 100
```

```
best_sample = np.array([x1_start, x2_start])
best_run = np.empty([1,1])
best_sample_runs = np.array([x1_start, x2_start])
accept_data = list(); reject_data = list()


for j in range(0, 100):

    updt(100, j + 1)

    t_curr = 20
    samples = np.array([[x1_start, x2_start]])
    accept = 0; reject = 0; count = 0

    for i in range(0, 5000):

        if (i%50 == 0):
            t_curr = t3(i)

        while True:
            y = q(samples[count, 0], samples[count, 0])
            if(y[0]<=500 and y[0]>=-500 and y[1]<=500 and y[1]>=-500):
                break

        f_prev = f(samples[count, 0], samples[count, 0])
        f_curr = f(y[0], y[1])

        ratio = np.exp((f_prev - f_curr)/t_curr)
        alpha = min(1, ratio)
        u = np.random.uniform()

        if((f_curr < f_prev) or (u <= alpha)):
            count += 1; accept +=1
            r = np.array([[y[0], y[1]]])
            samples = np.vstack((samples, r))
            if(f_curr < f_prev):
                best_sample = y

        else:
            reject += 1

    accept_data.append(accept)
    reject_data.append(reject)

    r1 = f(best_sample_runs[0], best_sample_runs[1])
    r2 = f(best_sample[0], best_sample[1])

    if(r1 > r2):
        best_sample_runs = best_sample
        best_run = samples

count = len(best_run)
fy = np.zeros([count])
for i in range(0, count):
    fy[i] = f(best_run[i, 0], best_run[i, 0])
plt.figure(2)
plt.hist(fy); plt.xlabel('Values of Minima'); plt.ylabel('#')
plt.title('Histogram of minima')
```

```
x1 = np.linspace(-500, 500, num = 501)
x2 = np.linspace(-500, 500, num = 501)

xx1, xx2 = np.meshgrid(x1, x2, indexing ='xy')
y = f(xx1, xx2)
plt.figure(3)
plt.contourf(xx1, xx2, y, cmap = 'viridis', alpha = 0.7)
plt.plot(best_run[:,0], best_run[:,0], 'r')
plt.colorbar()
plt.title('Contour plot of the Scwefel function')
plt.xlabel('x1'); plt.ylabel('x2')
```

Result:

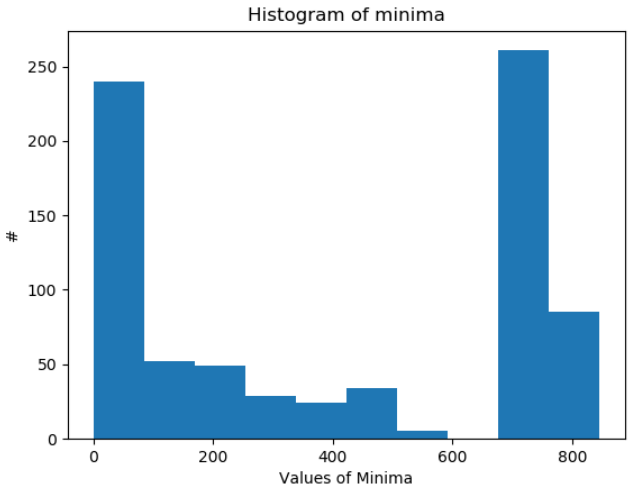


Figure 4. Histogram of function minima (Exponential cooling schedule)

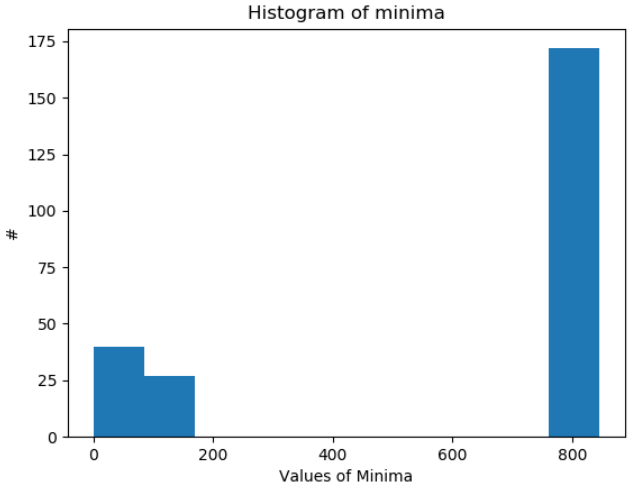


Figure 5. Histogram of function minima (Quadratic multiplicative cooling schedule)

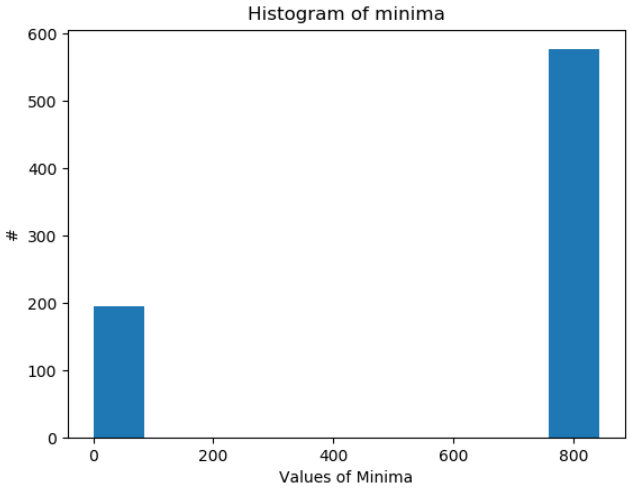


Figure 6. Histogram of function minima (Linear multiplicative cooling schedule)

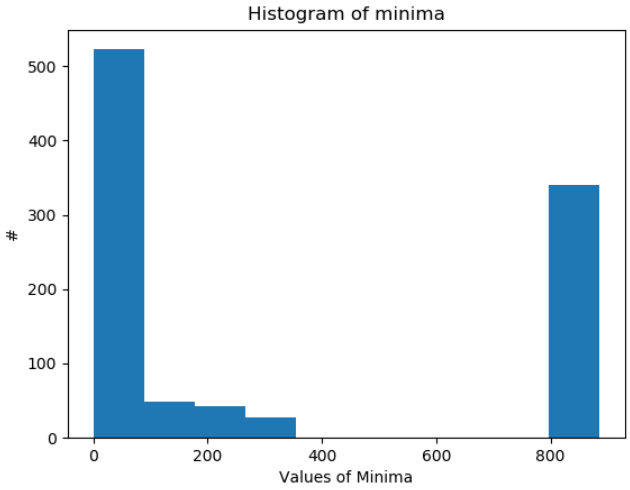


Figure 7. Histogram of function minima (Logarithmic multiplicative cooling schedule)

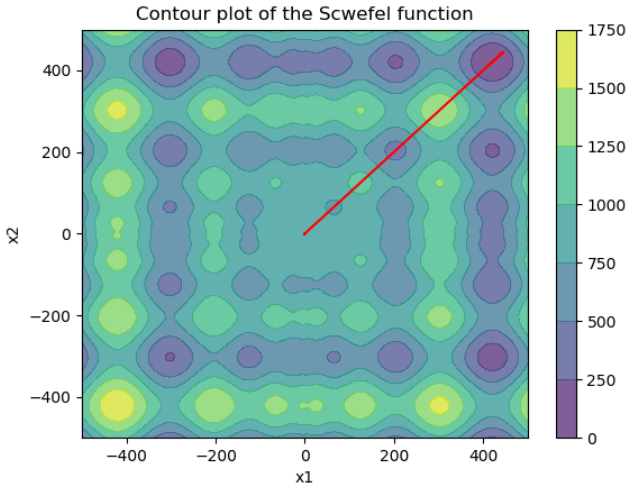


Figure 8. Sample path (shown in red) on the contour plot of the Schwefel function

The cooling schedules that I have used are

1. **Exponential cooling:**

$$T_k = T_0 \alpha^k$$

2. **Logarithmic cooling:**

$$T_k = \frac{T_0}{1 + \alpha \log(1 + k)}$$

3. **Linear multiplicative cooling:**

$$T_k = \frac{T_0}{1 + \alpha k}$$

4. **Quadratic multiplicative cooling:**

$$T_k = \frac{T_0}{1 + \alpha k^2}$$

From figure 7, it can be observed that logarithmic cooling is giving the best results. If the procedure is run for t = {20,50,100} iterations, the algorithm never converges to the minima. For t=1000, the algorithm converges to the minima for all the cooling functions

3. Problem 3

[Optimal Paths]

The famous Traveling Salesman Problem (TSP) is an NP-hard routing problem. The time to find optimal solutions to TSPs grows exponentially with the size of the problem (number of cities). A statement of the TSP goes thus:

> ***"A salesman needs to visit each of N cities exactly once and in any order. Each city is connected to other cities via an air transportation network. Find a minimum length path on the network that goes through all N cities exactly once (an optimal Hamiltonian cycle)."***

A TSP solution $\vec{c} = (c1, \ldots cN)$ is just an ordered list of the N cities with minimum path length. We will be exploring MCMC solutions to small and larger scale versions of the problem.

i. Pick N=10 2-D points in the [0,1000] x [0,1000] rectangle. These 2-D samples will represent the locations of N=10 cities. Plot your final TSP city tour.

ii. Run the Simulated Annealing TSP solver you just developed for N = {40, 400, 1000} cities. Explore the speed and convergence properties at these different problem sizes. You might want to play with the cooling schedules.

Code:

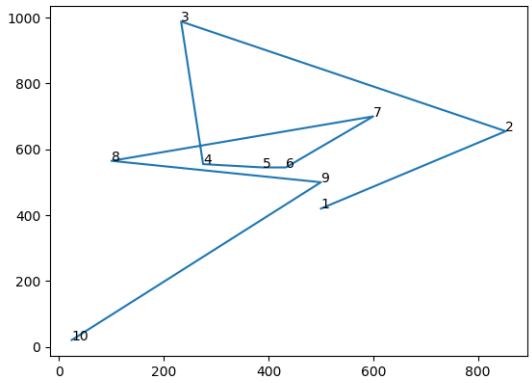
```
# -*- coding: utf-8 -*-
"""
Created on Wed May  2 22:42:44 2018

EE511 Project #5, Prof. Osonde Osoba, Spring 2018
@author: Hrishikesh Hippalgaonkar
Tested in Python 3.6.3 :: Anaconda custom (64-bit), Windows 10
Tested in Python 3.5.4 :: Anaconda custom (64-bit), Windows 10

Question 3.
Markov Chain Monte Carlo: Travelling Salesman Problem using Simulated Annealing
"""


import numpy as np
import math
import copy
import sys
import time
from math import exp, log
import matplotlib.pyplot as plt
import random


def updt(total, progress):
    """
    Displays or updates a console progress bar.

    Original source: https://stackoverflow.com/a/15860757/1391441
    """
    barLength, status = 20, ""
    progress = float(progress) / float(total)
    if progress >= 1.:
        progress, status = 1, "\r\n"
```

```
        block = int(round(barLength * progress))
        text = "\rRuns: [{}] {:.0f}% {}".format(
            "#" * block + "-" * (barLength - block), round(progress * 100, 0),
            status)
        sys.stdout.write(text)
        sys.stdout.flush()

    def t1(i):
        # Cooling function : logarithmic
        t = 500000/(1 + log(i + 1))
        return t

    def t2(i):
        # Cooling function : exponential
        t = 0.8*i
        return t

    def t3(i):
        # Cooling function : linear
        t = 500000/(1 + 2*i)
        return t

    def t4(i):
        # Cooling function : quadratic
        t = 500000/(1 + 2*i*i)
        return t


    def pos_swap(city, loc1, loc2):
        temp_city = copy.copy(city)  # This creates a shallow copy
        temp_city[[loc1, loc2]] = temp_city[[loc2, loc1]]
        return temp_city

    def distance(city):
        dist = 0
        loc = city[:, (0,1)]
        for i in range (1, len(city)):
            dist = dist + np.linalg.norm(loc[i-1,:] - loc[i,:])
        return dist


    def main():

        #t_curr = 50

        city = np.genfromtxt('cities_400.csv', delimiter = ',')
        n = np.linspace(1,400,num = 400); n1 = np.reshape(n,(400,1))
        city = np.hstack((city,n1))
        plt.ion()
        fig, ax = plt.subplots()
        ax.set_ylim([0,1000])
        ax.set_xlim([0,1000])
        plt.title('Travelling salesman problem')


        for i in range(0,8000):

            #updt(1000, i + 1)

    #        if (i%50 == 0):
```

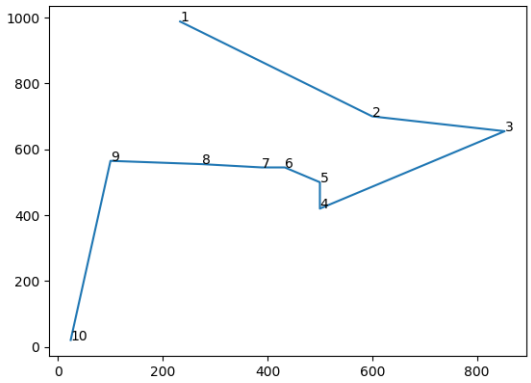
```
        #            t_curr = t2(t_curr)

                t_curr = t3(i)

                pos1 = random.randint(0,399)
                pos2 = random.randint(0,399)

                dist_prev = distance(city)
                temp_city = pos_swap(city, pos1, pos2)
                dist_new = distance(temp_city)
                if i == 0:
                    min_dist = dist_prev

                u = np.random.uniform()
                delta = dist_new - dist_prev
                ratio = np.exp(-delta/t_curr)
                alpha = min(1, ratio)

                if((delta < 0) or (u <= alpha)):
                    city = temp_city
                    min_dist = dist_new
                if(i%50 == 0):
                    print('Epoch:{0}\t Temperature:{1:0.2f} Minimum
      distance:{2:0.2f}'.format(i, t_curr, min_dist))

                time.sleep(0.0001)
                plt.cla()
                ax.plot(city[:,0], city[:,1])
                for q, txt in enumerate(n):
                    ax.annotate(txt, (city[q,0],city[q,1]))
                fig.canvas.draw()
                fig.canvas.flush_events()

            print('\nFinal City Tour:\n')
            print(city)


        if __name__ == "__main__":
    main()
```

Result:

Case 1:
Number of cities: 10 (initial temperature 50)



(a) Initial TSP city tour



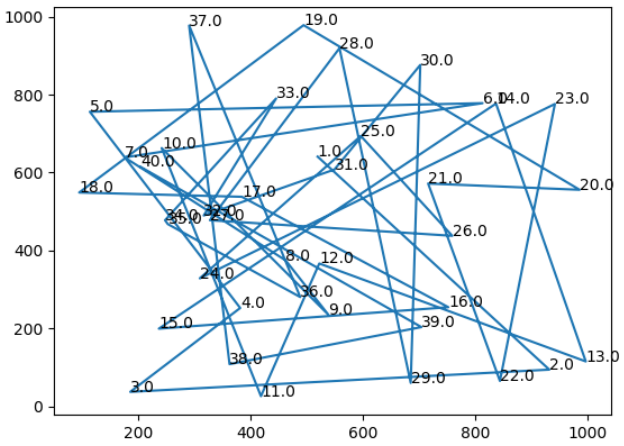
(b) Final optimized TSP city tour

Figure 9. TSP city tour for number of cities = 10

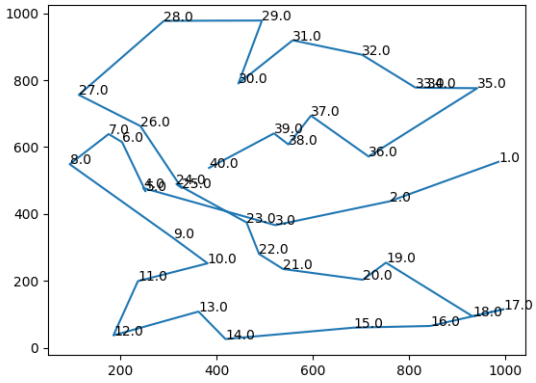
It can be observed that for the case of 10 cities, all the cooling schedules converge to best result in just 400 iterations.

Case 2:

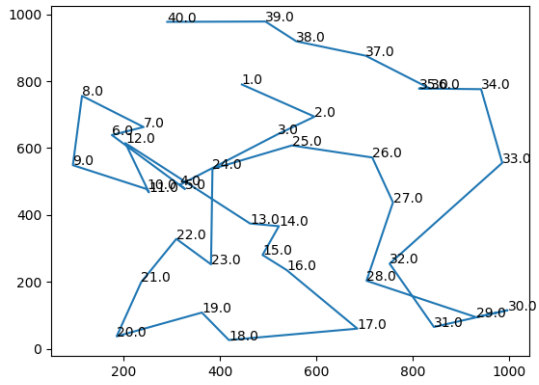
Number of cities: 40 (Initial temperature 500000)



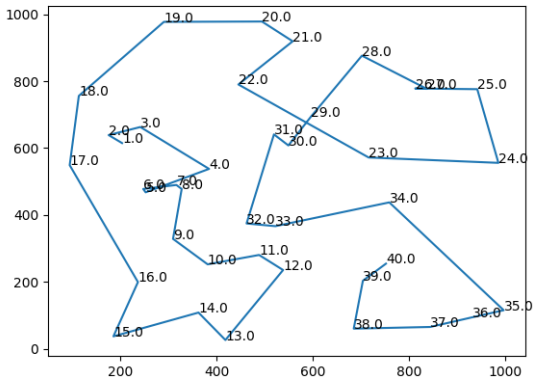
(a) Initial TSP city tour



(b) Final TSP city tour (linear cooling and 8000 iterations)



(c) Final TSP city tour (quadratic cooling and 8000 iterations)



(d) Final TSP city tour (logarithmic cooling and 8000 iterations)

Figure 10. TSP city tour for number of cities = 40

It can be observed that for the case of 40 cities, the logarithmic cooling schedule with 8000 iterations has converged to a good result. But since the initial temperature is only 500000 and number of iterations is only 8000, the algorithm doesn't converge to the best city tour.