Project 5 : Optimization & Sampling via MCMC

Author: Ishan Mohanty

USC ID: 4461-3447-18

NET ID: imohanty

Email: imohanty@usc.edu (mailto:imohanty@usc.edu)

/##

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 behavior of your algorithm.

Import Libraries and Other Dependencies

In [754]:

```
import numpy as np
from scipy.stats import beta
from scipy.stats import norm
from scipy.stats import lognorm
from scipy.stats import cauchy
import matplotlib.pyplot as plt
```

In [755]:

```
def stationary_pdf(x):
    fx = 0.60*beta.pdf(x, 1, 8)+0.40*beta.pdf(x,9,1)
    return fx
```

In [756]:

```
def proposal_pdf(x,var,func_type):
    if func_type == 'normal':
        gx = norm.rvs(loc=x, scale=var)
    elif func_type == 'cauchy':
        gx = cauchy.rvs(loc=x,scale=var)
    elif func_type == 'log_normal':
        gx = lognorm.rvs(var,loc=x,scale=1)
    else:
        print('wrong proposal pdf! please re-enter')
    return gx
```

In [757]:

```
def mh_algo(prev_sample,prop_sample,func_type,var):
    acc_count = 0
    x_cand = prop_sample
    if func_type == 'symmetric':
        acc_prob = np.min([1,stationary_pdf(x_cand) / stationary_pdf(prev_sample)])
    else:
        g_num = proposal_pdf(x_cand,var,'log_normal')
        g_denum = proposal_pdf(prev_sample,var,'log_normal')
        acc_prob = np.min([1,( stationary_pdf(x_cand)*g_num ) / (stationary_pdf(prev_

    u = np.random.uniform()
    if u < acc_prob:
        acc_count = 1
        prev_sample = x_cand
    return acc_count,prev_sample
```

In [758]:

```
while True:
    init_sample = np.random.uniform()
    if init_sample != 0:
        break
```

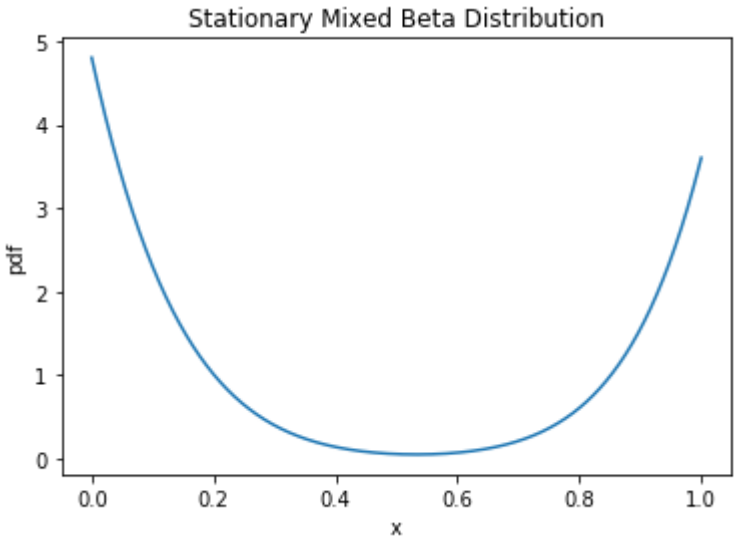
Stationary Beta Distribution fx

In [759]:

```
x = np.linspace(0,1,10000)
plt.figure()
plt.plot(x,stationary_pdf(x))
plt.xlabel('x')
plt.ylabel('pdf')
plt.title('Stationary Mixed Beta Distribution')
```

Out[759]:

```
Text(0.5,1,'Stationary Mixed Beta Distribution')
```



Convergence Plots for Standard Normal Gaussian Distribution

Standard Normal with mu = 0 and Variance is 0.1

In [760]:

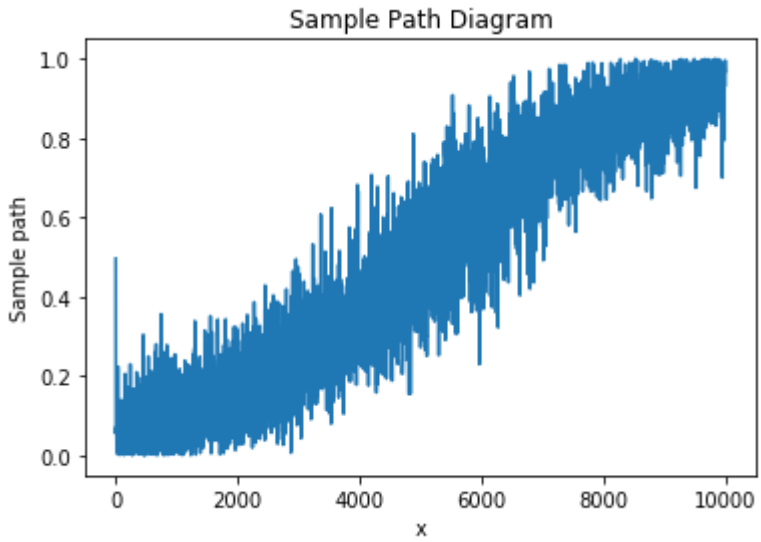
```
x = np.linspace(0,1,10000)
samples = [init_sample]
count = 0
for i in range(10000):
    prop_sample = proposal_pdf(x[i],0.1,'normal')
    c,prev_sample = mh_algo(samples[i],prop_sample,'symmetric',0)
    samples.append(prev_sample)
    count+=c
```

In [761]:

```
plt.figure()
samples = samples[0:10000]
x = np.arange(0,len(samples),1)
plt.plot(x,samples)
plt.title('Sample Path Diagram')
plt.xlabel('x')
plt.ylabel('Sample path')
```

Out[761]:

```
Text(0,0.5,'Sample path')
```

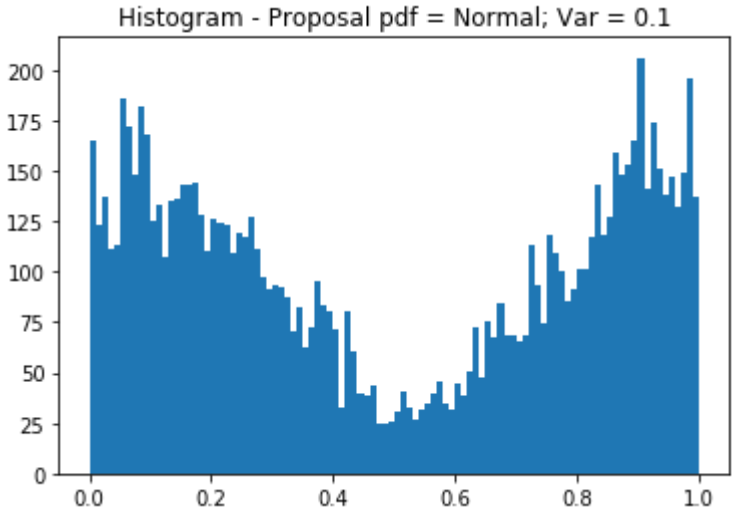


In [762]:

```
plt.figure()
plt.hist(samples, bins=100)
plt.title('Histogram - Proposal pdf = Normal; Var = 0.1')
```

Out[762]:

```
Text(0.5,1,'Histogram - Proposal pdf = Normal; Var = 0.1')
```



Standard Normal with mu = 0 and Variance is 0.1

In [763]:

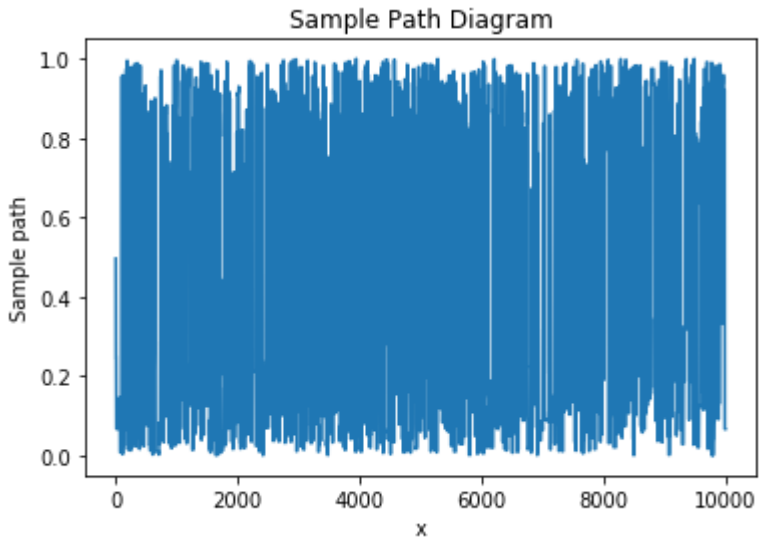
```
x = np.linspace(0,1,10000)
samples = [init_sample]
count = 0
for i in range(10000):
    prop_sample = proposal_pdf(x[i],1,'normal')
    c,prev_sample = mh_algo(samples[i],prop_sample,'symmetric',0)
    samples.append(prev_sample)
    count+=c
```

In [764]:

```
plt.figure()
samples = samples[0:10000]
x = np.arange(0,len(samples),1)
plt.plot(x,samples)
plt.title('Sample Path Diagram')
plt.xlabel('x')
plt.ylabel('Sample path')
```

Out[764]:

```
Text(0,0.5,'Sample path')
```

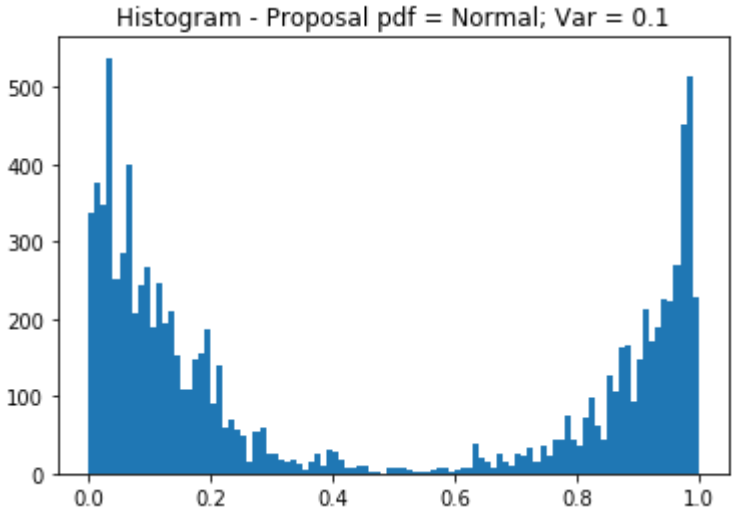


In [765]:

```
plt.figure()
plt.hist(samples, bins=100)
plt.title('Histogram - Proposal pdf = Normal; Var = 0.1')
```

Out[765]:

```
Text(0.5,1,'Histogram - Proposal pdf = Normal; Var = 0.1')
```



LogNormal assymetric Distribution

LogNormal with s parameter = 0.1

In [766]:

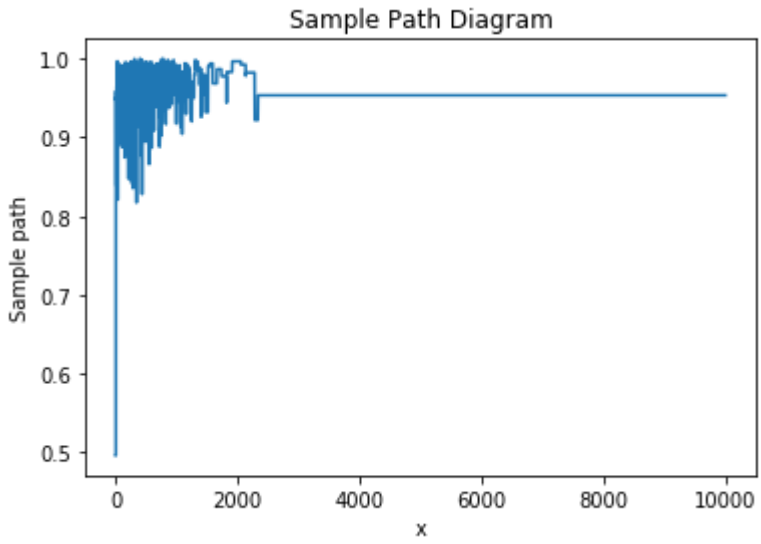
```
x = np.linspace(0,1,10000)
log_samples = [init_sample]
count = 0
for i in range(10000):
    prop_sample = proposal_pdf(x[i],0.1,'log_normal')
    c,prev_sample = mh_algo(log_samples[i],prop_sample,'asymmetric',0.1)
    log_samples.append(prev_sample)
    count+=c
```

In [767]:

```
plt.figure()
log_samples = log_samples[0:10000]
x = np.arange(0,len(log_samples),1)
plt.plot(x,log_samples)
plt.title('Sample Path Diagram')
plt.xlabel('x')
plt.ylabel('Sample path')
```

Out[767]:

```
Text(0,0.5,'Sample path')
```

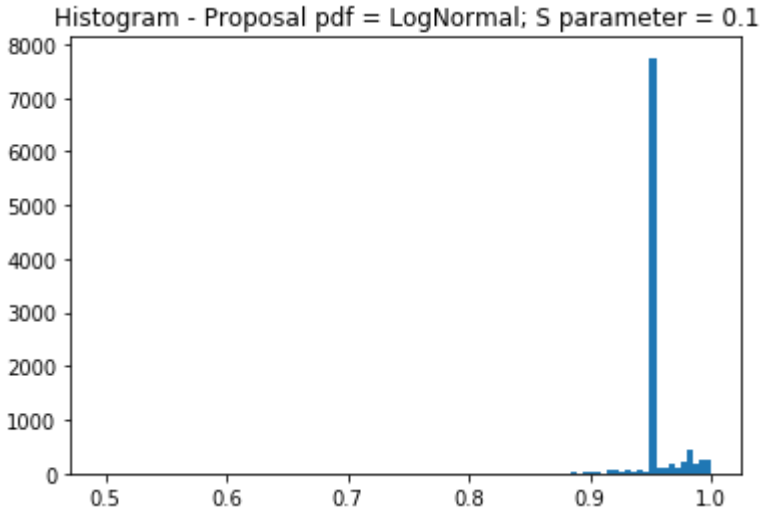


In [768]:

```
plt.figure()
plt.hist(log_samples, bins=100)
plt.title('Histogram - Proposal pdf = LogNormal; S parameter = 0.1')
```

Out[768]:

```
Text(0.5,1,'Histogram - Proposal pdf = LogNormal; S parameter = 0.1')
```



LogNormal with s parameter 20 , s acts like a scaling or variance factor

In [769]:

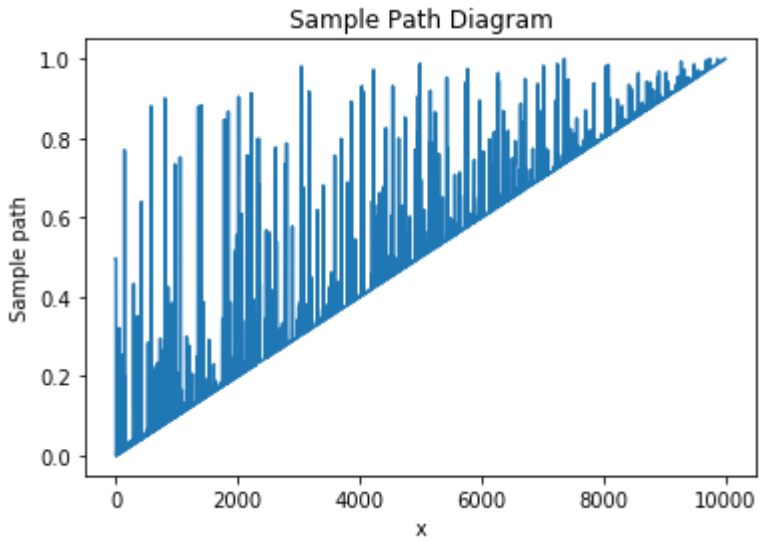
```
x = np.linspace(0,1,10000)
log_samples = [init_sample]
count = 0
for i in range(10000):
    prop_sample = proposal_pdf(x[i],20,'log_normal')
    c,prev_sample = mh_algo(log_samples[i],prop_sample,'asymmetric',20)
    log_samples.append(prev_sample)
    count+=c
```

In [770]:

```
plt.figure()
log_samples = log_samples[0:10000]
x = np.arange(0,len(log_samples),1)
plt.plot(x,log_samples)
plt.title('Sample Path Diagram')
plt.xlabel('x')
plt.ylabel('Sample path')
```

Out[770]:

```
Text(0,0.5,'Sample path')
```

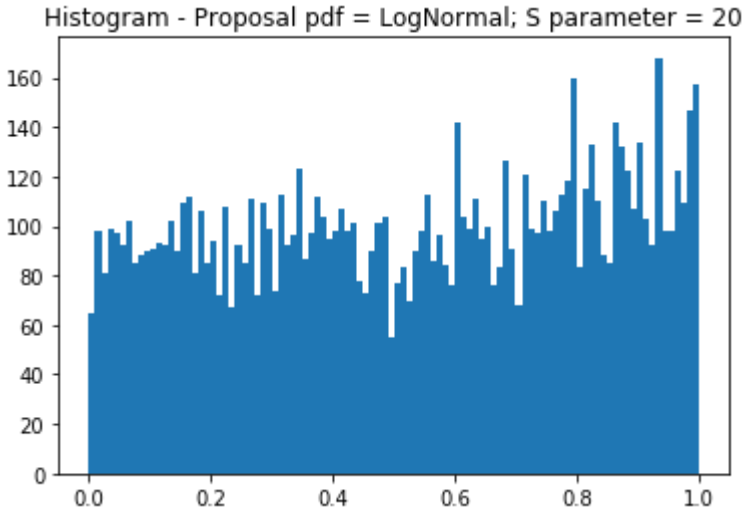


In [771]:

```
plt.figure()
plt.hist(log_samples, bins=100)
plt.title('Histogram - Proposal pdf = LogNormal; S parameter = 20')
```

Out[771]:

```
Text(0.5,1,'Histogram - Proposal pdf = LogNormal; S parameter = 20')
```



Cauchy - with variance 0.1

In [772]:

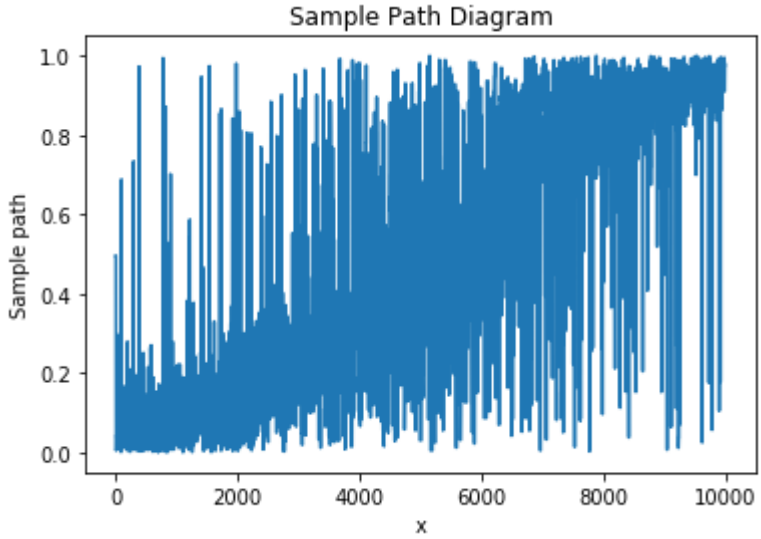
```
x = np.linspace(0,1,10000)
cauchy_samples = [init_sample]
count = 0
for i in range(10000):
    prop_sample = proposal_pdf(x[i],0.1,'cauchy')
    c,prev_sample = mh_algo(cauchy_samples[i],prop_sample,'symmetric',0)
    cauchy_samples.append(prev_sample)
    count+=c
```

In [773]:

```
plt.figure()
cauchy_samples = cauchy_samples[0:10000]
x = np.arange(0,len(cauchy_samples),1)
plt.plot(x,cauchy_samples)
plt.title('Sample Path Diagram')
plt.xlabel('x')
plt.ylabel('Sample path')
```

Out[773]:

```
Text(0,0.5,'Sample path')
```

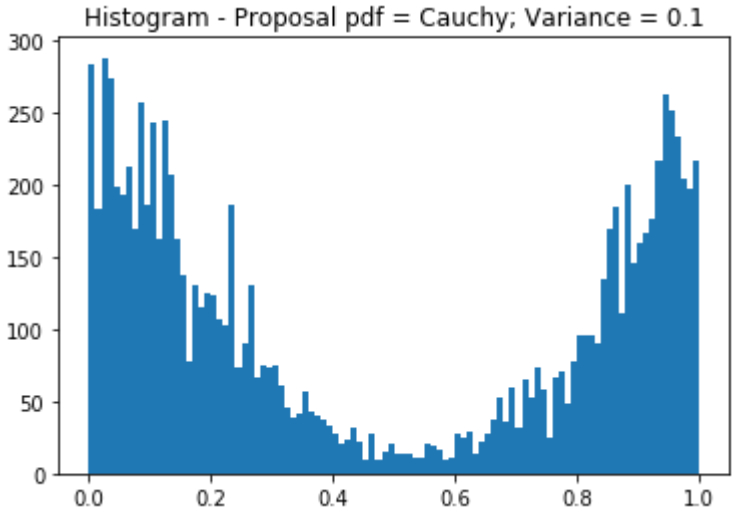


In [774]:

```
plt.figure()
plt.hist(cauchy_samples, bins=100)
plt.title('Histogram - Proposal pdf = Cauchy; Variance = 0.1')
```

Out[774]:

```
Text(0.5,1,'Histogram - Proposal pdf = Cauchy; Variance = 0.1')
```



Cauchy - with variance 1

In [775]:

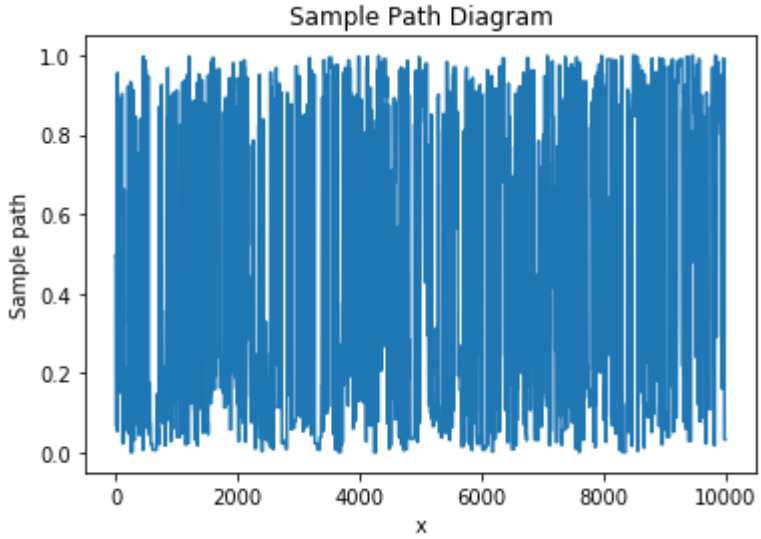
```
x = np.linspace(0,1,10000)
cauchy_samples = [init_sample]
count = 0
for i in range(10000):
    prop_sample = proposal_pdf(x[i],1,'cauchy')
    c,prev_sample = mh_algo(cauchy_samples[i],prop_sample,'symmetric',0)
    cauchy_samples.append(prev_sample)
    count+=c
```

In [776]:

```
plt.figure()
cauchy_samples = cauchy_samples[0:10000]
x = np.arange(0,len(cauchy_samples),1)
plt.plot(x,cauchy_samples)
plt.title('Sample Path Diagram')
plt.xlabel('x')
plt.ylabel('Sample path')
```

Out[776]:

```
Text(0,0.5,'Sample path')
```

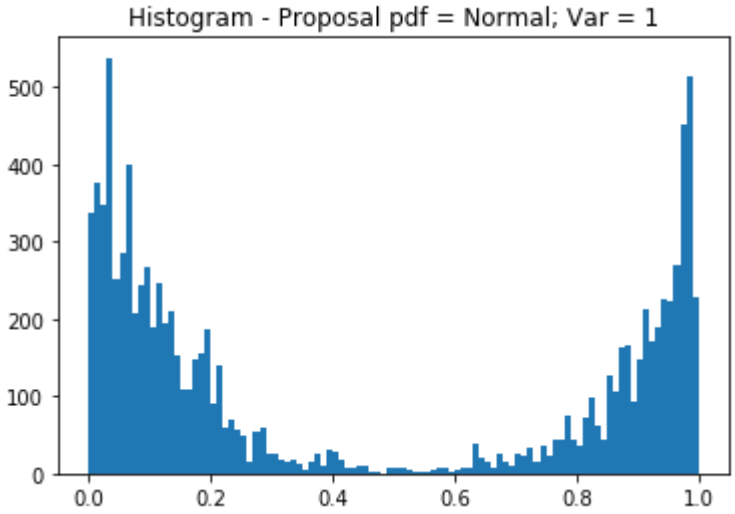


In [777]:

```
plt.figure()
plt.hist(samples, bins=100)
plt.title('Histogram - Proposal pdf = Normal; Var = 1')
```

Out[777]:

```
Text(0.5,1,'Histogram - Proposal pdf = Normal; Var = 1')
```



Summary:

1. Metro-Hastings Algorithm enables us to build a Markov Chain which has a suitable stationary distribution.
2. Mixed beta distribution f(x) was generated according to the following distribution: 60% in a Beta(1,8) distribution and 40% in a Beta(9,1) distribution.
3. Metro-Hastings Algorithm:
   A. choose an initial sample random value xt , t=0.
   B. get the proposed sample x* from the proposal pdf normal , cauchy or log-normal with different variances.
   C. calculate acceptance probability alpha = min(1 , f(x* / xt) g(x* / xt) / f(xt / x*)*g(xt /x*))*
   D. generate u [0,1]
   E. if u < alpha , accept xt+1 = x* , proposed else, xt+1 = xt , previous sample
   F. repeat process for other sample iterations
4. Check for convergence with the histogram. Repeat the different configurations with different variance values for the normal, cauchy and log-normal distributions.
5. Here Normal and Cauchy are symmetric but log-normal is assymetric.
6. In symmetric cases, g(x* / xt) = g(xt / x*). Hence the ratio of g = 1.

Results:

1. Sample paths have been plotted and we see that different starting points don't really make that much of a difference to convergence points.
2. variance was taken as 0.1 and 1 for the symmetrical distributions normal and cauchy. we see that we get really good convergence for high variance values and poorer convergence for low variance values. High variance helps in mixing well with distribution and helps to approximate the fx distibution better.
3. the lognormal assymetric function does not do that good a job in approximating the beta distribution but we see that higher scaling parameters S gives better results than lower ones.

Problem 2.

The n-dimensional Scwefel function

f ($\vec{x}$)=418.9829 n−Σ i=1 n* xi*sin √|xi|, xi∈[−500,500]

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 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.

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.

In [608]:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
from scipy.stats import cauchy
import random
```

In [609]:

```
def cost_function(x1,x2):
    cost = (418.9829 * 2) - ((x1 * np.sin(np.sqrt(abs(x1)))) + (x2 * np.sin(np.sqrt(a
    return cost
```

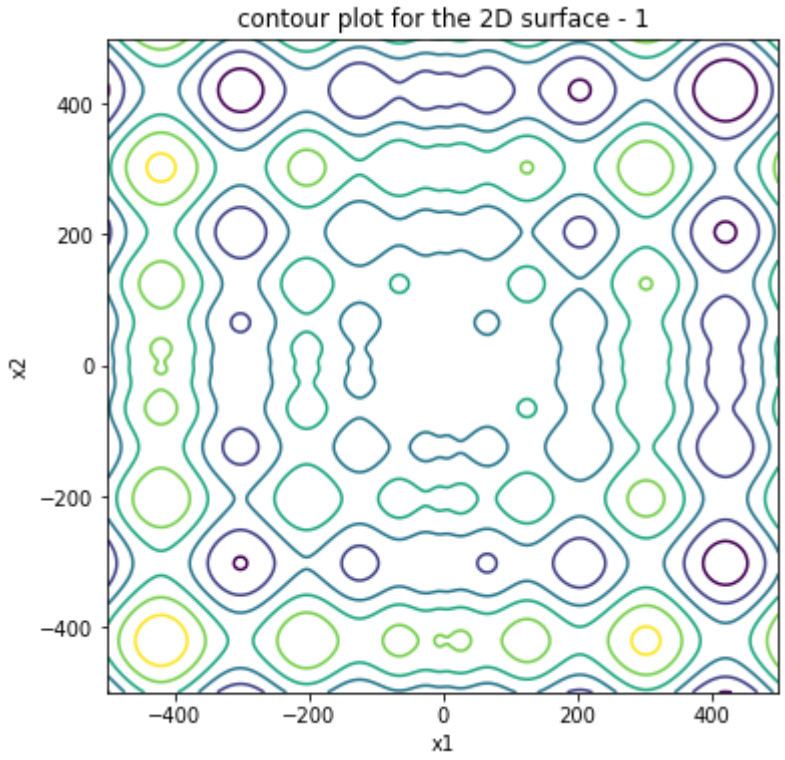
Part 1 Surface Plot

In [625]:

```
def surface_plot(title,temp_type,temp):
    x1 = np.linspace(-500, 500, 200)
    x2 = np.linspace(-500, 500, 200)
    x, y = np.meshgrid(x1, x2)
    plt.figure(figsize=(6,6))
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.title(title+" "+temp_type+" "+str(temp))
    plt.contour(x,y, cost_function(x,y))
```

In [626]:

```
surface_plot('contour plot for the 2D surface','-',1)
```



Part 2 Simulated Annealing for Global minimum

In [627]:

```
def proposal(state,func_type,var):
    if func_type == 'normal':
        s = norm.rvs(loc=state,scale=var)

    if func_type == 'cauchy':
        s = cauchy.rvs(loc=state,scale=var)

    if s[0] < -500:
        s[0] = -500
    if s[1] < -500:
        s[1] = -500
    if s[0] > 500:
        s[0] = 500
    if s[1] > 500:
        s[1] = 500

    return s
```

In [628]:

```
def decay_temp(temp_type,i,init_temp):
    if temp_type == 'exp':
        decay_t = init_temp*( 0.80 ** (i + 1) )
    elif temp_type == 'log':
        alpha = 1.1
        decay_t = init_temp/(1+alpha*np.log(1+i))   #sometimes just log(1+i) or 1+log
    elif temp_type == 'poly':
        alpha = 0.1
        decay_t = init_temp/(1+alpha*i)
    else:
        print('incorrect cooling function')
    return decay_t
```

In [629]:

```
def anneal(max_iter,init_temp,func_type,var,temp_func):
    initial = np.random.uniform(-500,500,2)
    samples = [initial]
    count = 1
    for i in range(max_iter):
        prev_state = samples[count-1]
        new_state = proposal(prev_state,func_type,var)
        t_decay = decay_temp(temp_func,i,init_temp)
        cost_prev = cost_function(prev_state[0], prev_state[1])
        cost_new = cost_function(new_state[0], new_state[1])
        acc_prob = np.min([1, np.exp(-1 * (cost_new - cost_prev)/ t_decay)])
        if acc_prob > np.random.uniform():
            count+=1
            samples.append(new_state)
    return samples
```

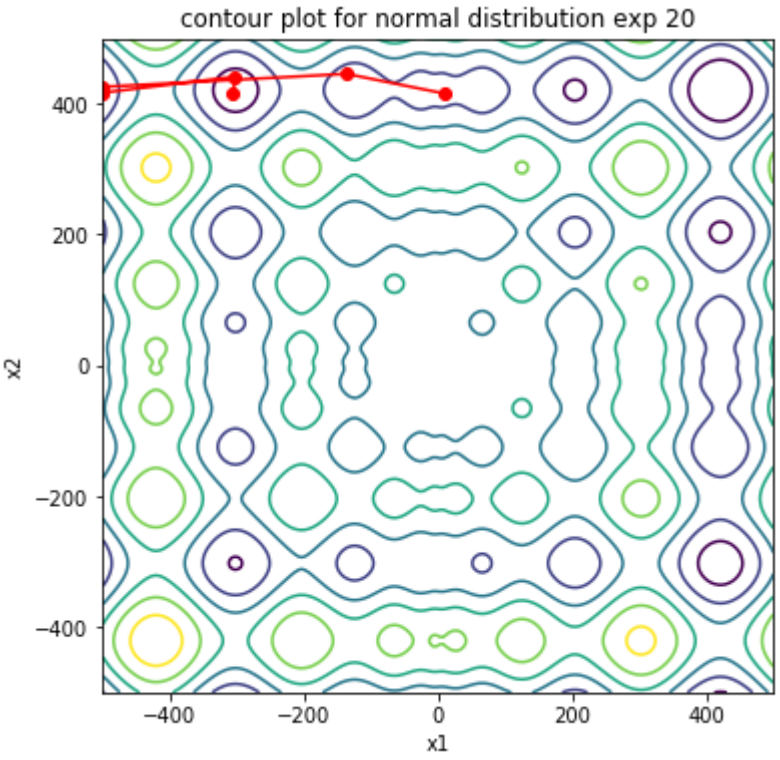
In [630]:

```
def convergence_info(init_temp,prop_func,var,temp_type,title):
    accepted_samples = anneal(100,init_temp,prop_func,var,temp_type)
    surface_plot(title,temp_type,init_temp)
    accepted_samples = np.array(accepted_samples)
    plt.plot(accepted_samples[:,0],accepted_samples[:,1], marker='o', color='r')
    plt.show()
    print("Convergence achieved at: ", accepted_samples[-1])
    print("Number of Points connecting the path",len(accepted_samples))
    return accepted_samples
```

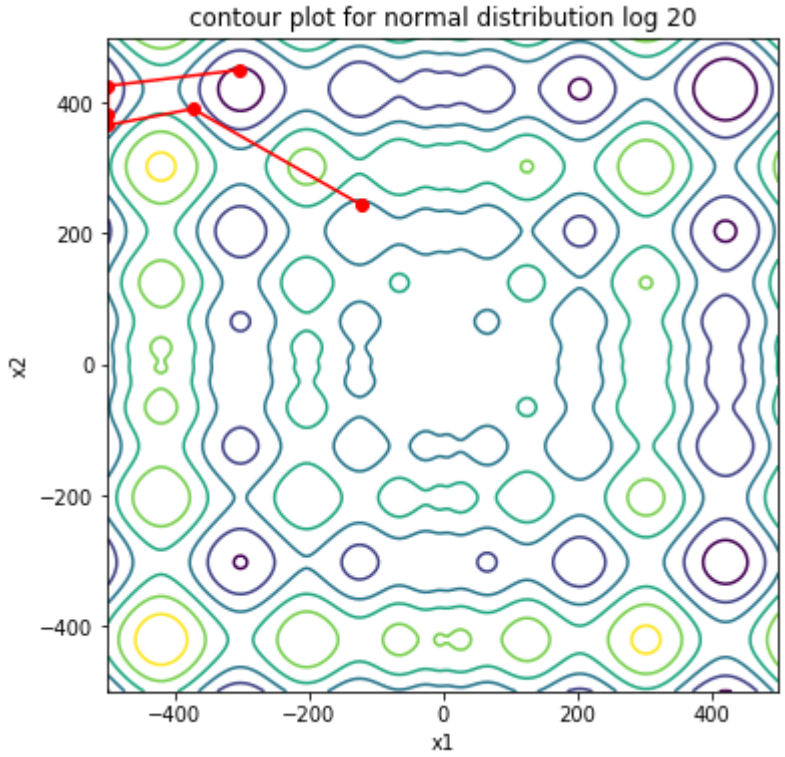
Standard Normal Distribution

In [631]:

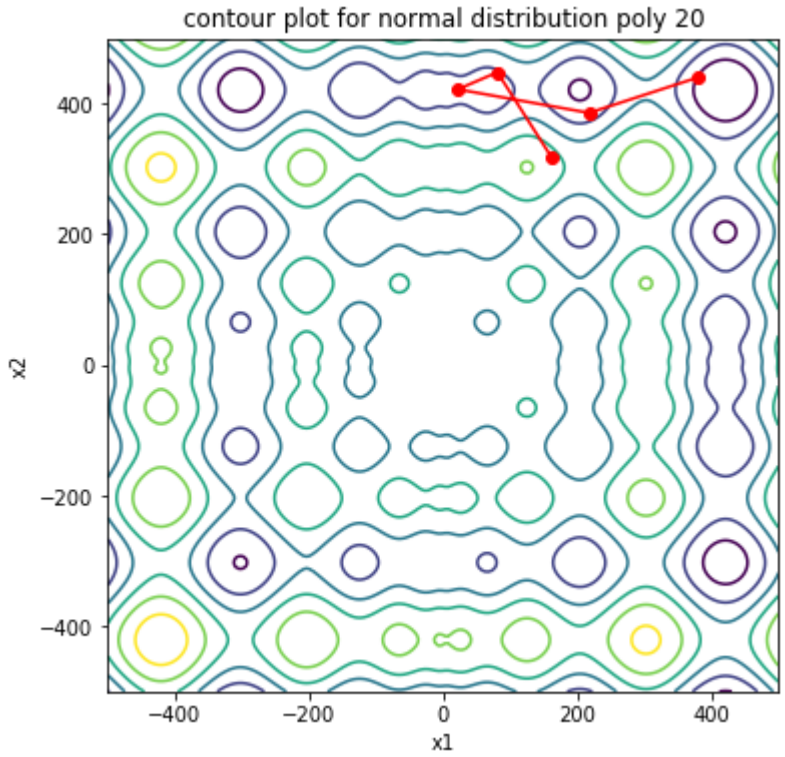
```
hist_samples_exp = []
hist_samples_log = []
hist_samples_poly = []
t = [20,50,100,1000]
for temp in t:
    hist_samples_exp.append(convergence_info(temp,'normal',200,'exp','contour plot fc
    hist_samples_log.append(convergence_info(temp,'normal',200,'log','contour plot fc
    hist_samples_poly.append(convergence_info(temp,'normal',200,'poly','contour plot
```



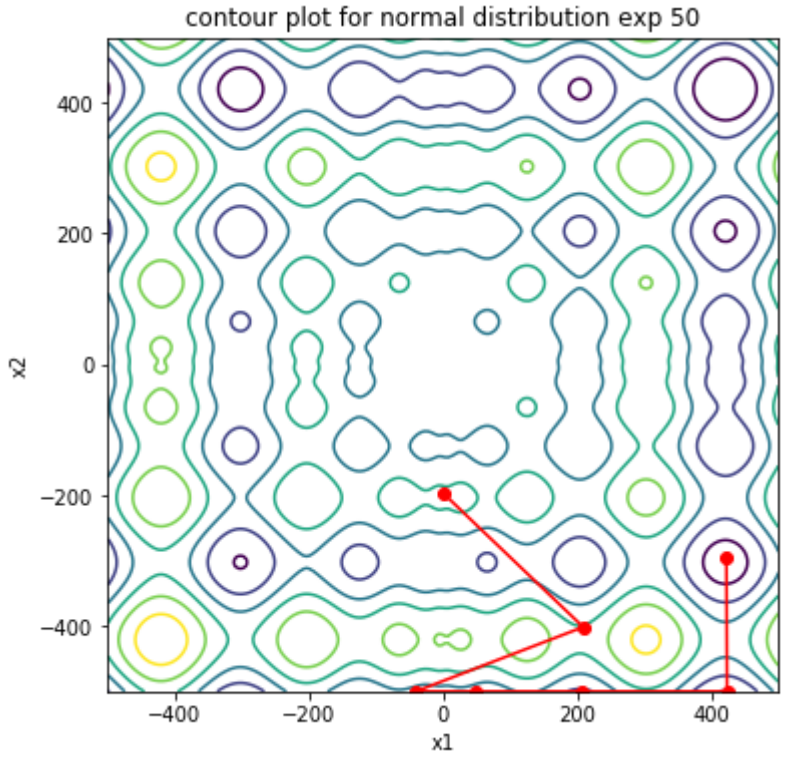

```
Convergence achieved at:  [-305.8944933   417.58352028]
Number of Points connecting the path 6
```



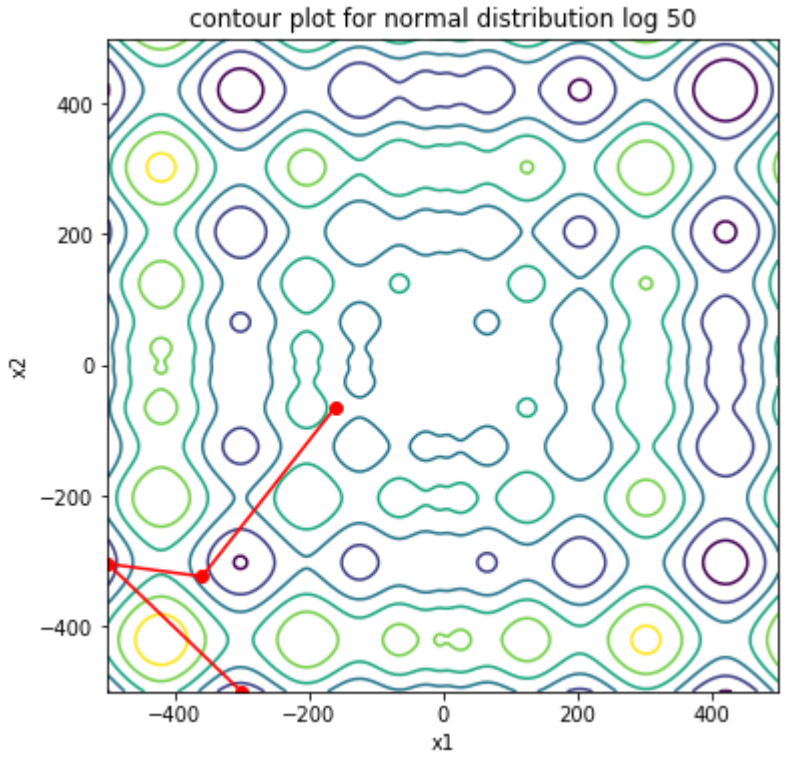

```
Convergence achieved at:  [-302.88078191  450.41896398]
Number of Points connecting the path 6
```



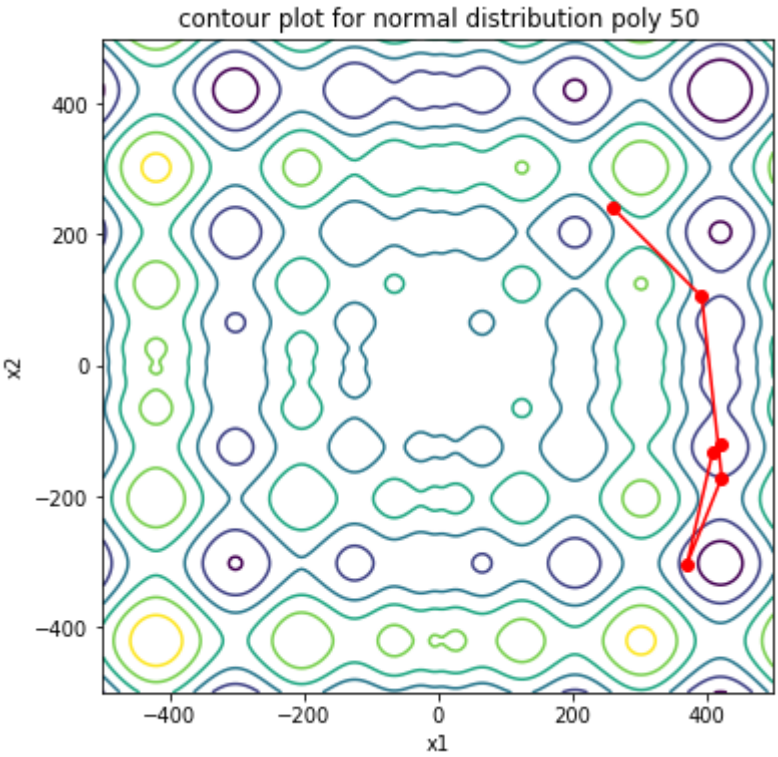

```
Convergence achieved at:  [378.61736049 439.75469076]
Number of Points connecting the path 5
```



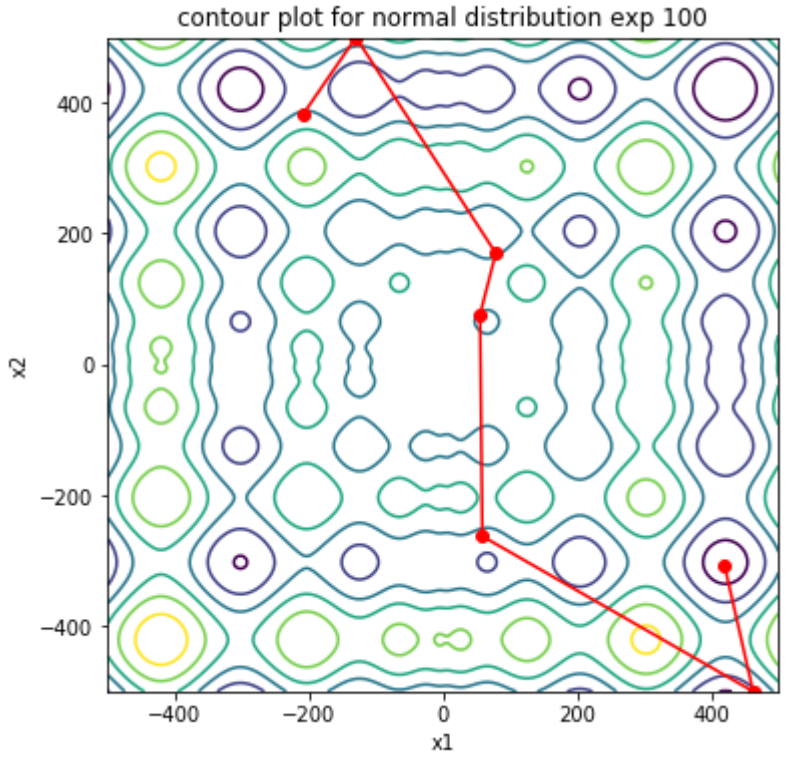

```
Convergence achieved at:  [ 422.35663483 -296.00664676]
Number of Points connecting the path 7
```



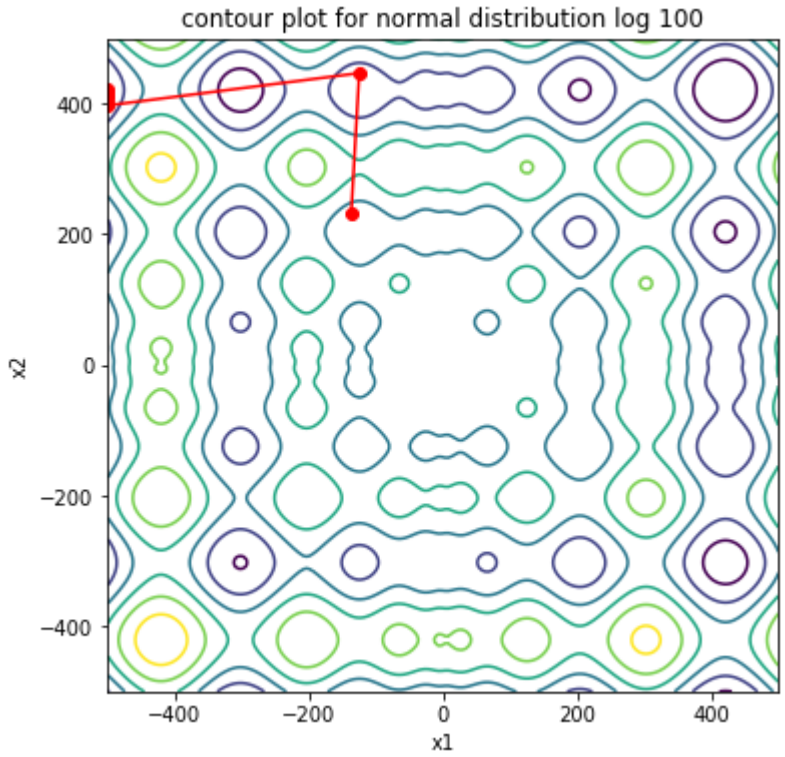
```
Convergence achieved at:  [-301.25472947 -500.        ]
Number of Points connecting the path 4
```



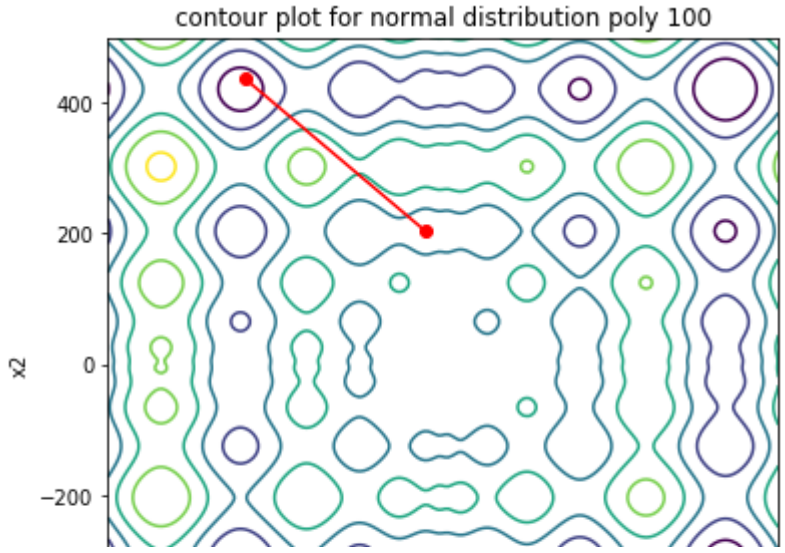

```
Convergence achieved at:  [ 420.81409908 -121.62593997]
Number of Points connecting the path 6
```



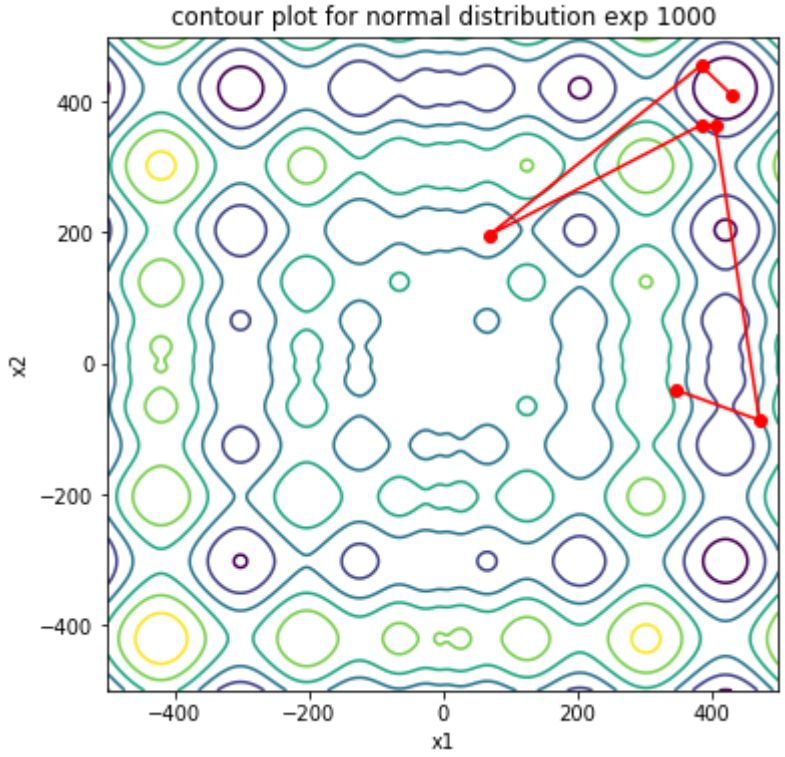

```
Convergence achieved at:  [ 419.21822936 -305.96245128]
Number of Points connecting the path 7
```



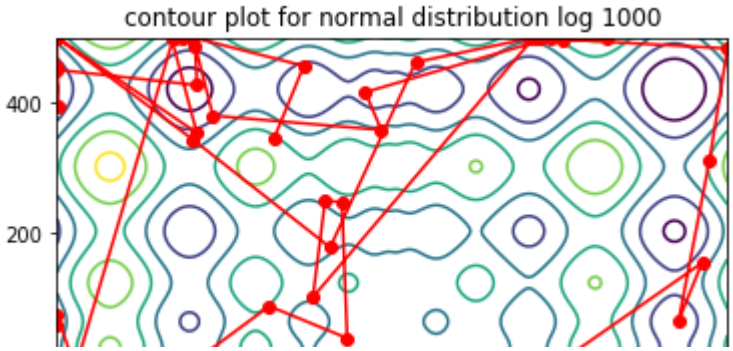

```
Convergence achieved at:  [-500.          410.51662575]
Number of Points connecting the path 8
```



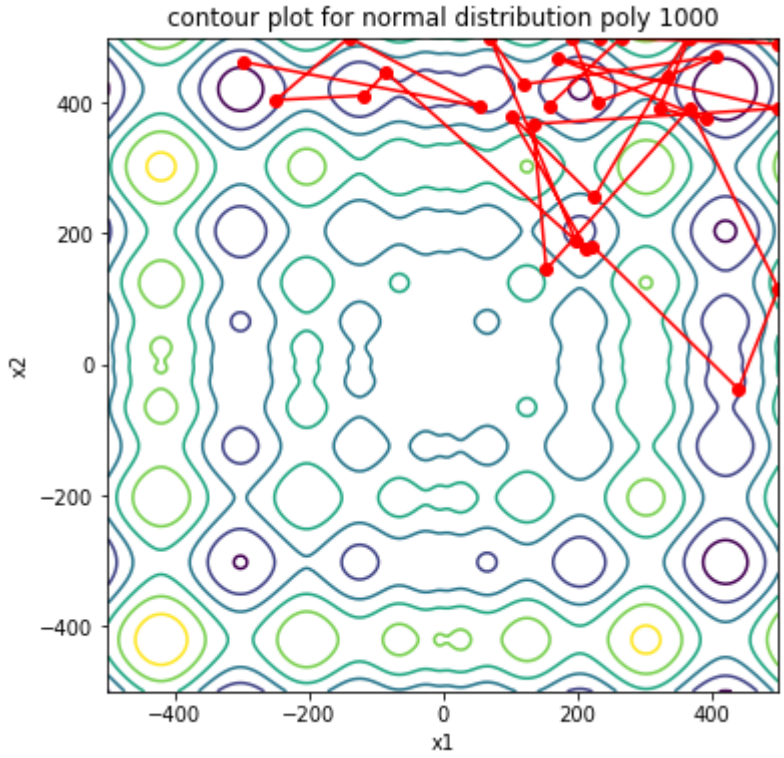

```
Convergence achieved at:  [-295.69250726  436.8854802 ]
Number of Points connecting the path 2
```




```
Convergence achieved at:  [430.9271412 409.735382 ]
Number of Points connecting the path 7
```



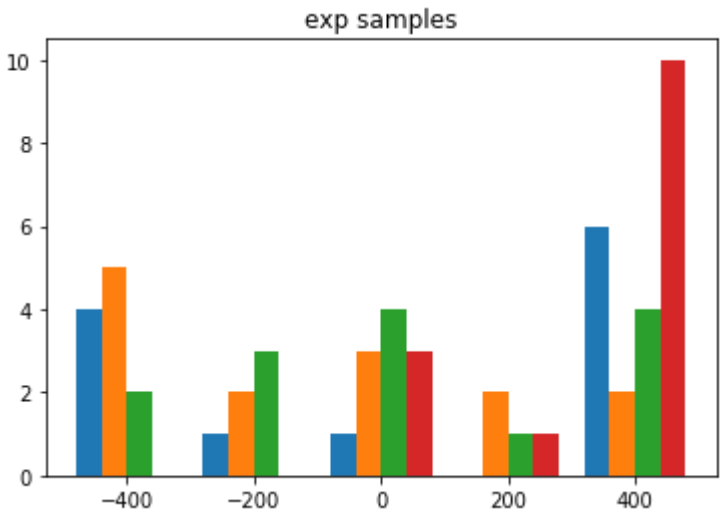

```
Convergence achieved at:  [-500.          60.44837378]
Number of Points connecting the path 48
```




```
Convergence achieved at:  [-299.57866518  462.84529769]
Number of Points connecting the path 32
```

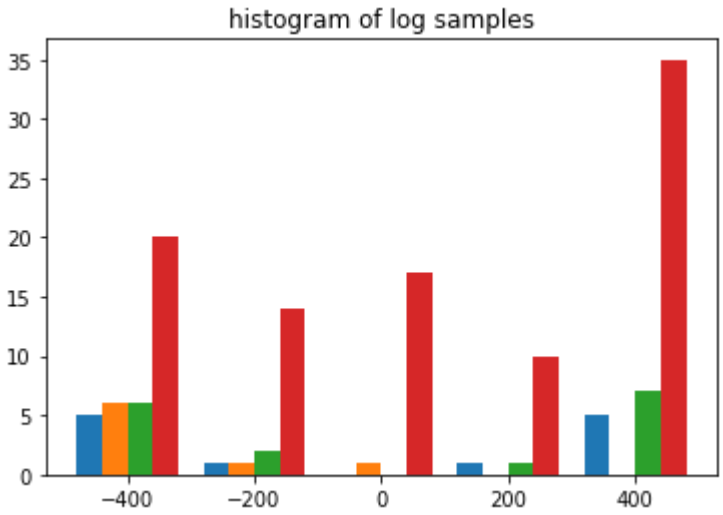
In [635]:

```
plt.figure()
plt.hist(hist_samples_exp,bins=5)
plt.title('histogram of exp samples')
plt.show()
```



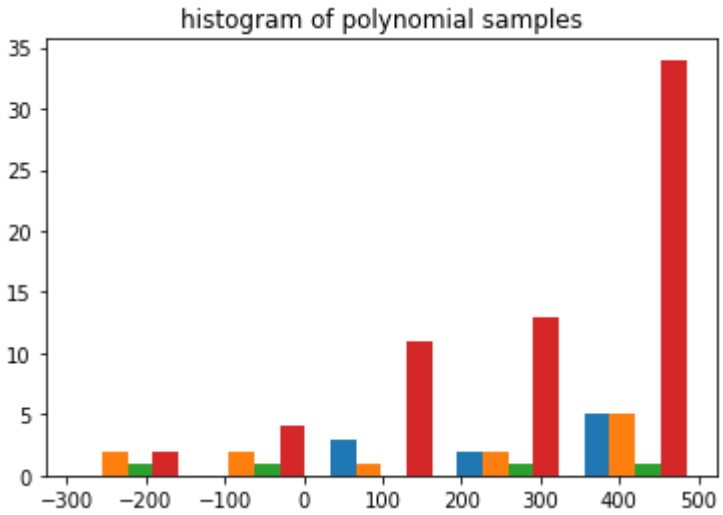
In [636]:

```
plt.figure()
plt.hist(hist_samples_log,bins=5)
plt.title('histogram of log samples')
plt.show()
```



In [637]:

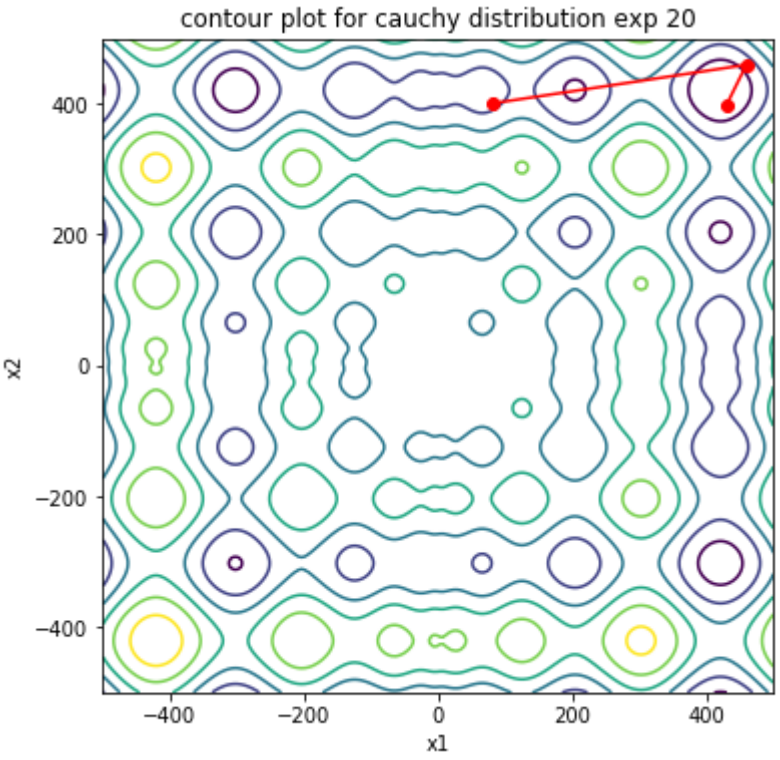
```
plt.figure()
plt.hist(hist_samples_poly,bins=5)
plt.title('histogram of polynomial samples')
plt.show()
```



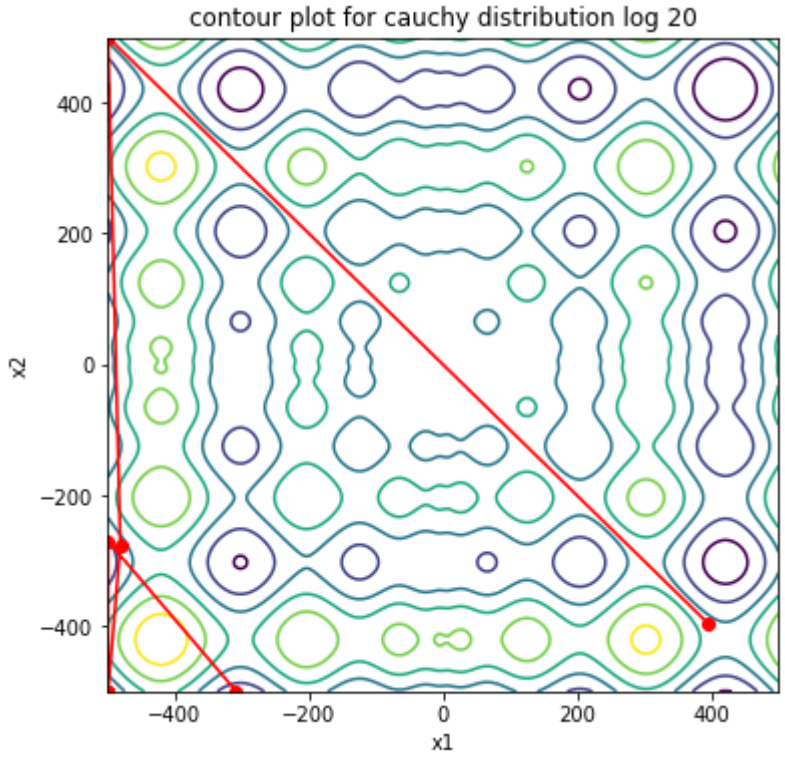
Cauchy Distribution

In [646]:

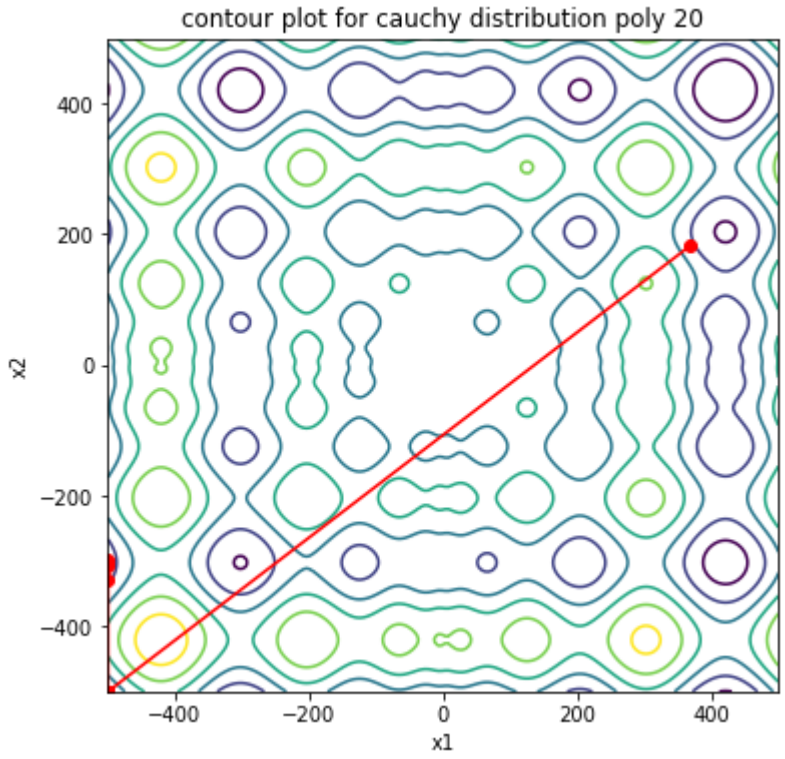
```
hist_samples_exp_cau = []
hist_samples_log_cau = []
hist_samples_poly_cau = []
t = [20,50,100,1000]
for temp in t:
    hist_samples_exp_cau.append(convergence_info(temp,'cauchy',200,'exp','contour plc
    hist_samples_log_cau.append(convergence_info(temp,'cauchy',200,'log','contour plc
    hist_samples_poly_cau.append(convergence_info(temp,'cauchy',200,'poly','contour p
```



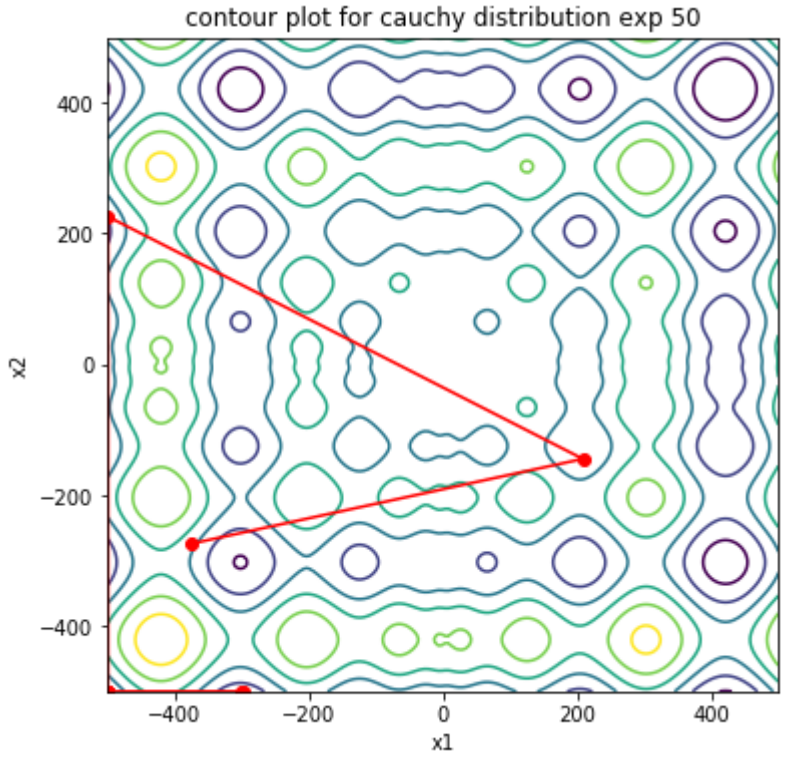

```
Convergence achieved at:  [429.32865807 396.91860961]
Number of Points connecting the path 3
```



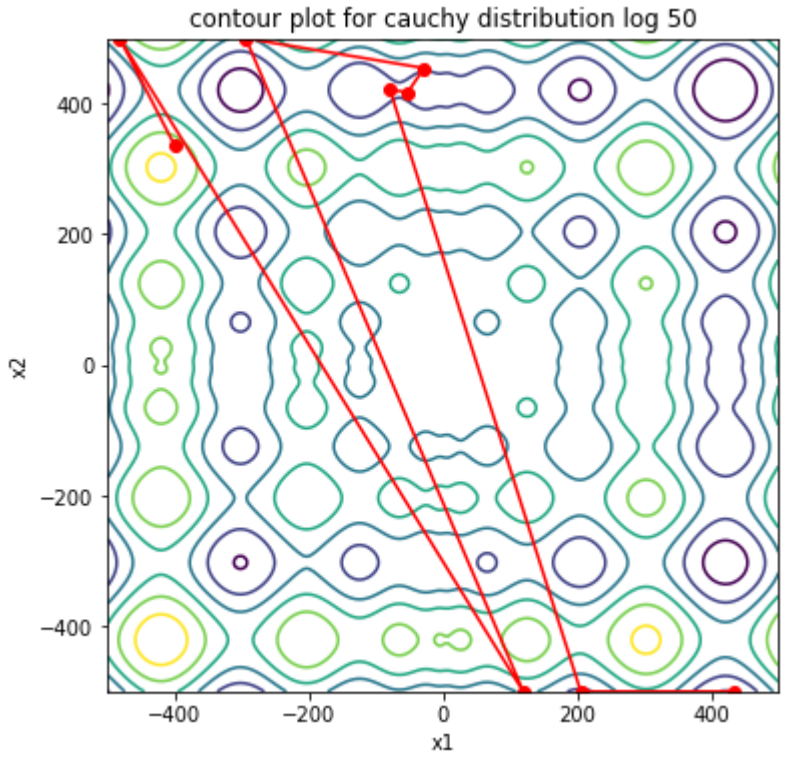

```
Convergence achieved at:  [-308.70734355 -500.        ]
Number of Points connecting the path 10
```



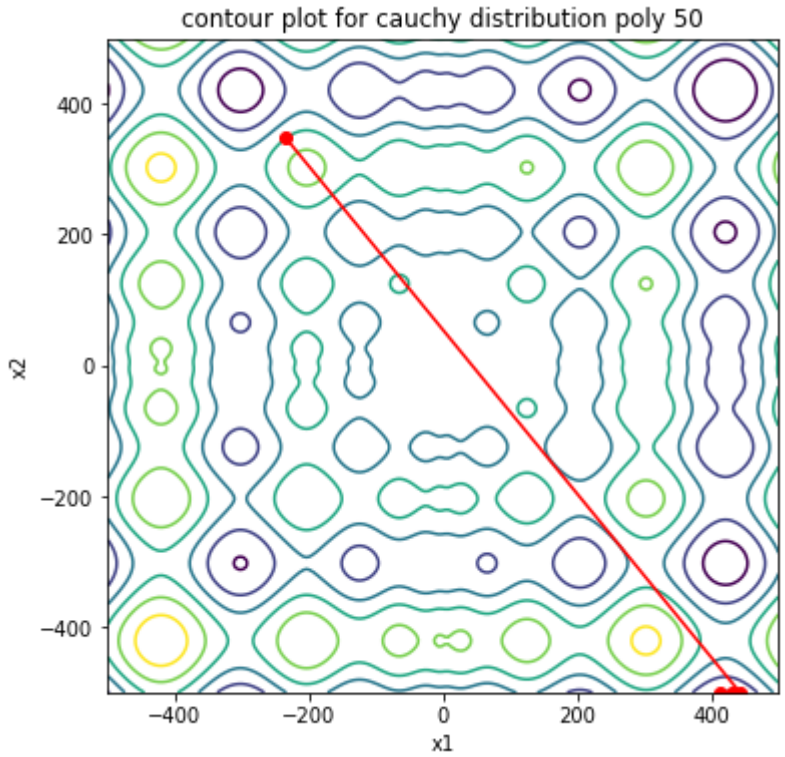

```
Convergence achieved at:  [-500.         -301.71819985]
Number of Points connecting the path 24
```



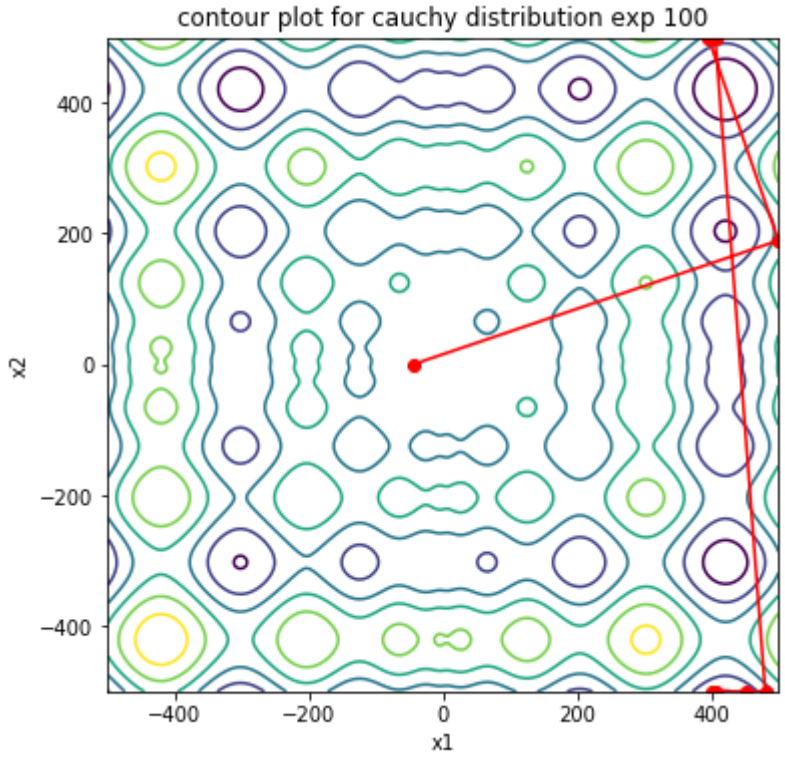

```
Convergence achieved at:  [-299.76524547 -500.        ]
Number of Points connecting the path 16
```



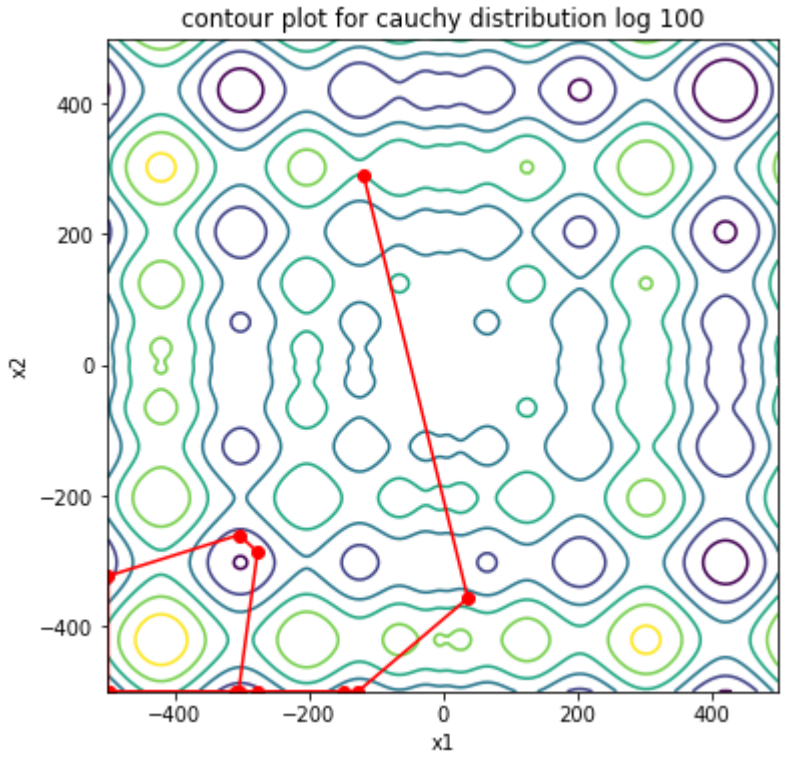
```
Convergence achieved at:  [ 432.5181126 -500.        ]
Number of Points connecting the path 9
```



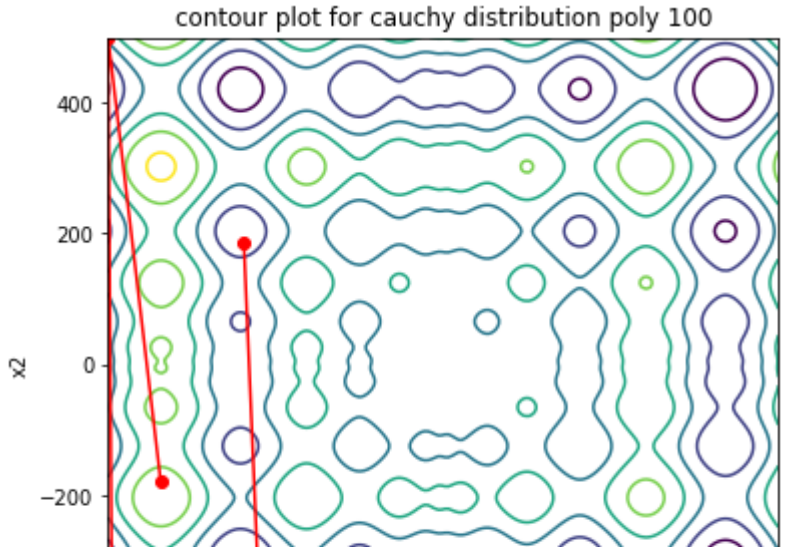

```
Convergence achieved at:  [ 429.63522071 -500.        ]
Number of Points connecting the path 4
```



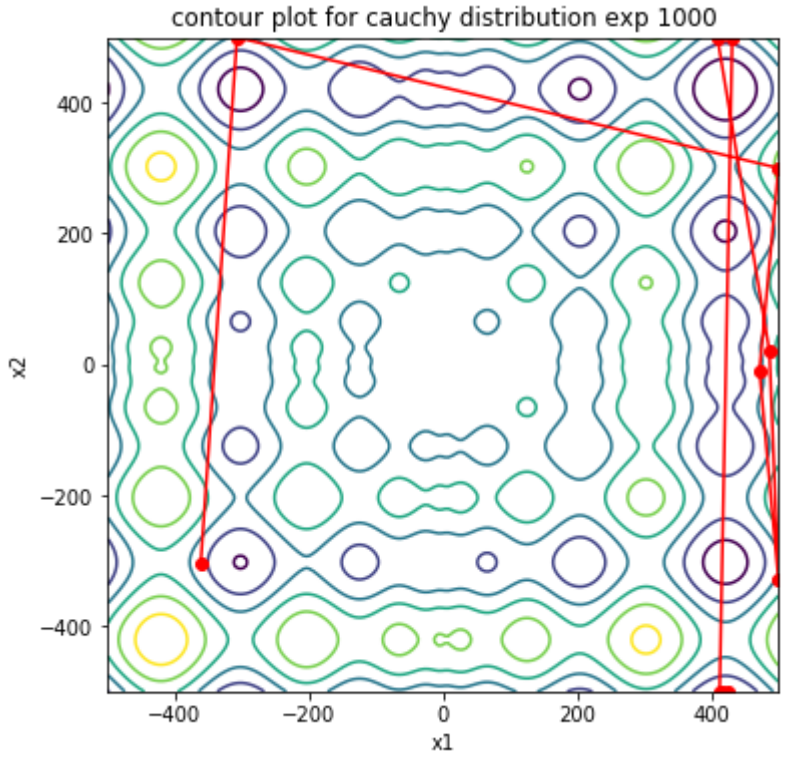

```
Convergence achieved at:  [ 406.92227858 -500.        ]
Number of Points connecting the path 8
```



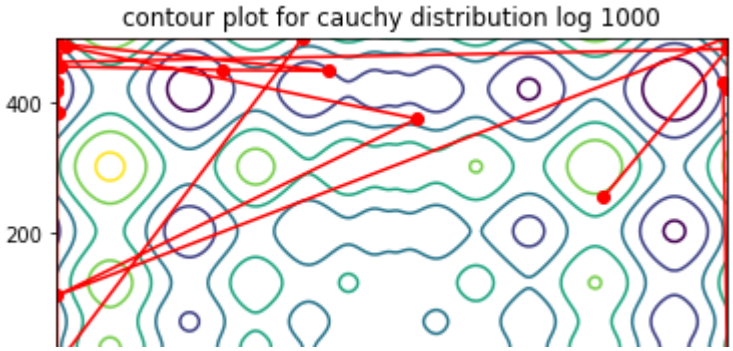

```
Convergence achieved at:  [-310.3590452 -500.        ]
Number of Points connecting the path 23
```



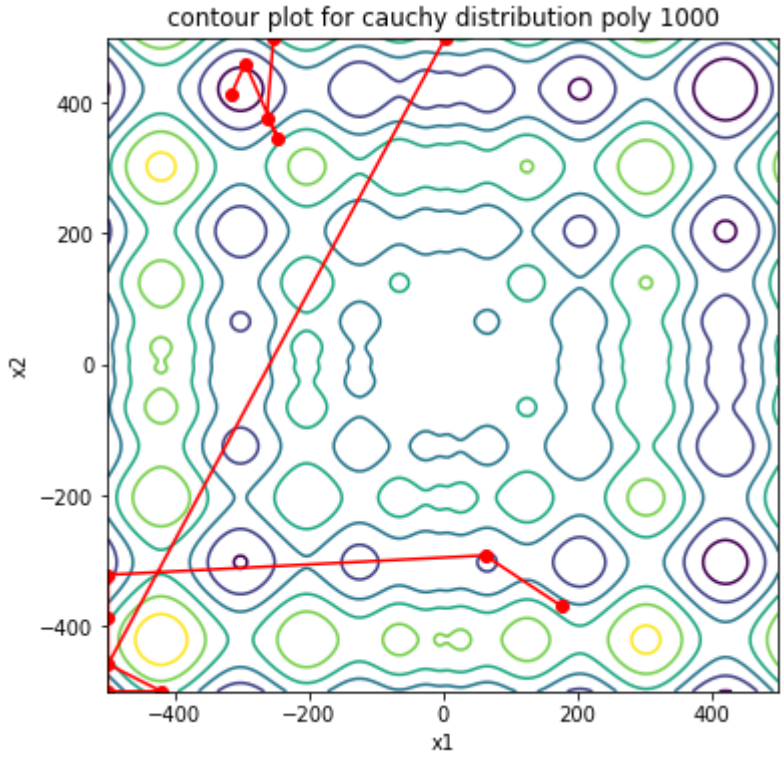

```
Convergence achieved at:  [-297.40269012  185.81359683]
Number of Points connecting the path 7
```




```
Convergence achieved at:  [ 423.42813971 -500.        ]
Number of Points connecting the path 11
```



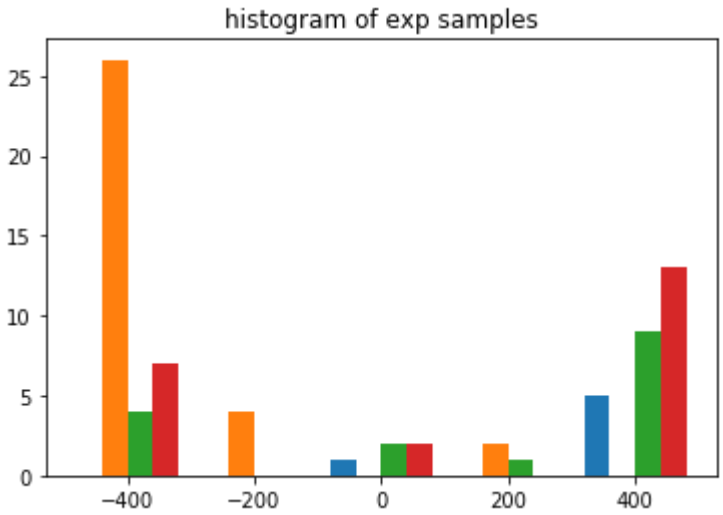

```
Convergence achieved at:  [-290.72912733 -297.98203033]
Number of Points connecting the path 35
```




```
Convergence achieved at:  [-316.08773094  413.55223951]
Number of Points connecting the path 14
```

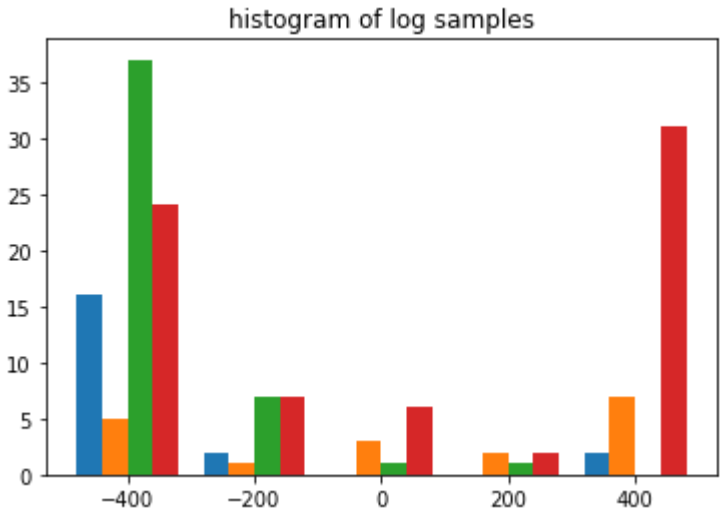
In [647]:

```
plt.figure()
plt.hist(hist_samples_exp_cau,bins=5)
plt.title('histogram of exp samples')
plt.show()
```



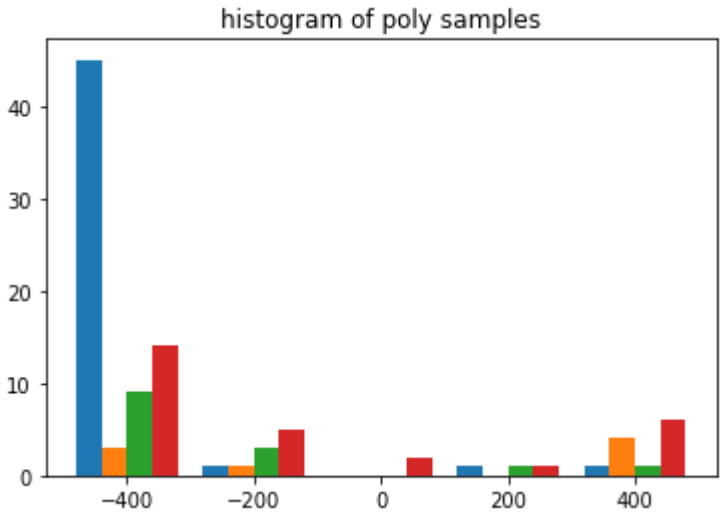
In [648]:

```
plt.figure()
plt.hist(hist_samples_log_cau,bins=5)
plt.title('histogram of log samples')
plt.show()
```



In [649]:

```
plt.figure()
plt.hist(hist_samples_poly_cau,bins=5)
plt.title('histogram of poly samples')
plt.show()
```



Best global minimum solution for standard normal distribution is the exponential cooling schedule, Temp :100

Best global minimum solution for standard normal distribution is the exponential cooling schedule Temp: 20

Summary

- 1. The contour plot of the scwefel equation is shown above.
- 2.Sample initial point x_0
- 3.A proposal pdf like the cauchy pdf and the normal pdf was used to generate a candidate sample

$$y \sim q(y/x_t)$$

- 4.Gibbs ratio is defined as shown below:

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

- 5.The temperature is initialized 20
- 6. Accept y with a probability $\alpha(T_t)$
- 7. Substitue T_t according to any of the three cooling schedules - logarithmic, polynomial and exponential
- 8.this process was repeated for 100 runs till we converge to a global minimum
- 9.The entire procedure is repeated for different initial temperatures example 20, 50, 100 and 1000
- 10.The histogram of the function minima and the best run that converged were plotted.

Results

1. For standard Normal distribution: best global minimum - Convergence achieved at: [430.9271412 409.735382]

Number of Points connecting the path 7

cooling schedule - exponential temperature: 100

2. For Cauchy Distribution:

   best global minimum - Convergence achieved at: [429.32865807 396.91860961]

   Number of Points connecting the path 3

   cooling schedule - exponential temperature: 20
3. The results are displayed above for the best histogram and the best overlay run.

Problem 3:

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 $\vec{c}=(c1, ...c\ N\ )$ 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.

1. Write a function to capture the objective function of the TSP problem: $D(\square c\ )=\Sigma\ i=1\ N-1\ \|ci+1-ci\|$
2. Start with a random path through all N cities $\vec{c}0$ (a random permutation of the cities), an initial high temperature T0, and a cooling schedule Tk=f (T0 , k).
3. Randomly pick any two cities in your current path. Swap them. Use the difference between the new and old path length to calculate a Gibbs acceptance probability. Update the path accordingly.
4. Update your annealing temperature and repeat the previous city swap step. Run the simulated annealing procedure “to convergence.”
5. Plot the values of your objective function from each step. 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.

In [281]:

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

In [263]:

```
def objective_cost_function(N,city_tour,c_loc):
    total_cost = 0
    for i in range(N-1):
        start = city_tour[i]
        dest = city_tour[i+1]
        c_loc = np.asarray(c_loc)
        city_start = c_loc[start]
        city_dest = c_loc[dest]
        diff = city_dest - city_start
        total_cost += np.linalg.norm(diff,ord=2)
    return total_cost
```

In [264]:

```
def city_locations(N):
    city_loc = []
    for i in range(N):
        boundary = range(0,1000)
        city_loc.append(random.sample(boundary,2))
    return city_loc
```

In [265]:

```
def decay_temp(temp_type,i,init_temp):
    if temp_type == 'exp':
        decay_t = init_temp*( 0.80 ** (i + 1) )
    elif temp_type == 'log':
        alpha = 1.1
        decay_t = init_temp/(1+alpha*np.log(1+i))   #sometimes just log(1+i) or 1+log
    elif temp_type == 'poly':
        alpha = 0.1
        decay_t = init_temp/(1+alpha*i)
    else:
        print('incorrect cooling function')
    return decay_t
```

In [266]:

```
def travel_anneal(N,num_runs):
    final_cost = []
    c_loc = city_locations(N)
    city_tour = random.sample(range(0,N),N)
    init_tour = city_tour[:]

    for i in range(num_runs):
        sw_pos =  random.sample(range(N),2)
        sw_city_tour = city_tour[:]
        sw_city_tour[sw_pos[1]] = city_tour[sw_pos[0]]
        sw_city_tour[sw_pos[0]] = city_tour[sw_pos[1]]

        cost = objective_cost_function(N,city_tour,c_loc)
        sw_cost = objective_cost_function(N,sw_city_tour,c_loc)

        t_decay = decay_temp('exp',i,100)

        acc_prob = np.min([1, np.exp(-1 * (sw_cost - cost)/ t_decay)])
        if acc_prob > np.random.uniform():
            city_tour = sw_city_tour[:]
            final_cost.append(sw_cost)

    return city_tour, init_tour, final_cost , c_loc
```

In [267]:

```
def plot_city_tour(num_cities,x,y,color,title):
    plt.figure()
    plt.plot(x,y, marker='o', color=color)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.grid('on')
    plt.title(title+" for "+str(num_cities)+" cities")
```

N = 10 cities

In [269]:

```
num_cities = 10
start = time.time()
city_tour, init_tour, final_cost, c_loc = travel_anneal(num_cities,1000)
end = time.time()
print("Time for the TSP algorithm to run: ",end - start," seconds")
prev_x = list(zip(*[c_loc[init_tour[i % num_cities]] for i in range(num_cities+1)]))[
prev_y = list(zip(*[c_loc[init_tour[i % num_cities]] for i in range(num_cities+1)]))[
new_x = list(zip(*[c_loc[city_tour[i % num_cities]] for i in range(num_cities+1)]))[0
new_y = list(zip(*[c_loc[city_tour[i % num_cities]] for i in range(num_cities+1)]))[1
```

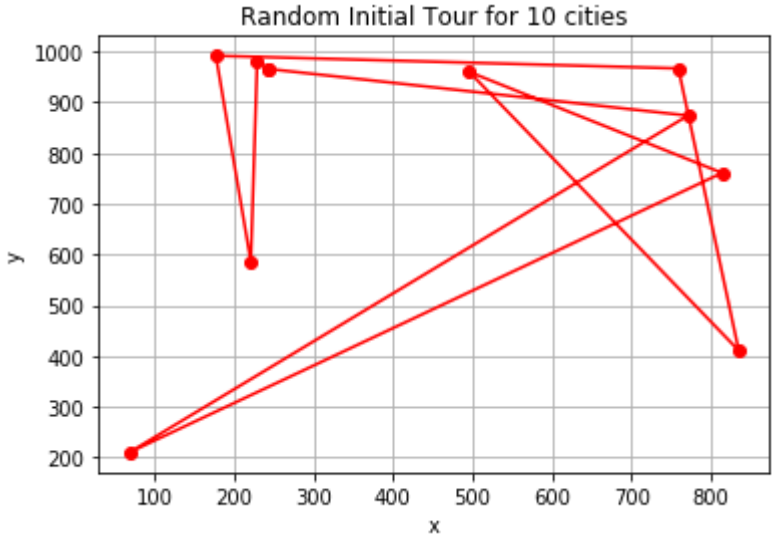
```
Time for the TSP algorithm to run:  0.20295166969299316  seconds
```

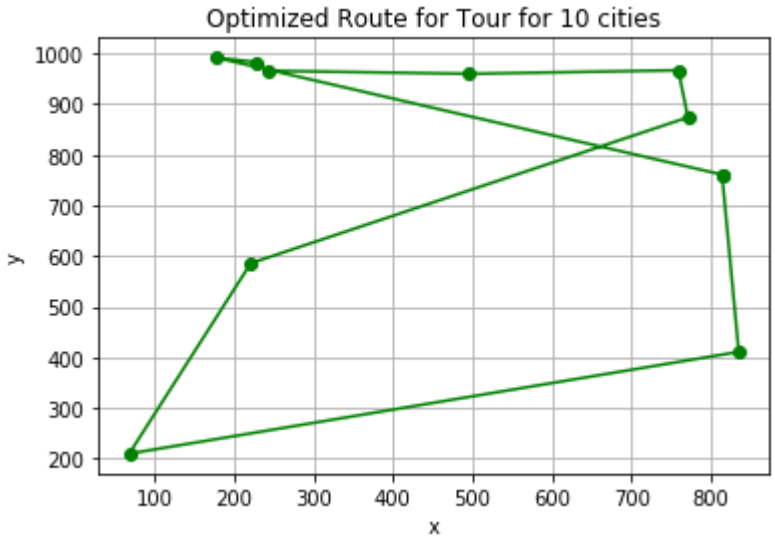
In [270]:

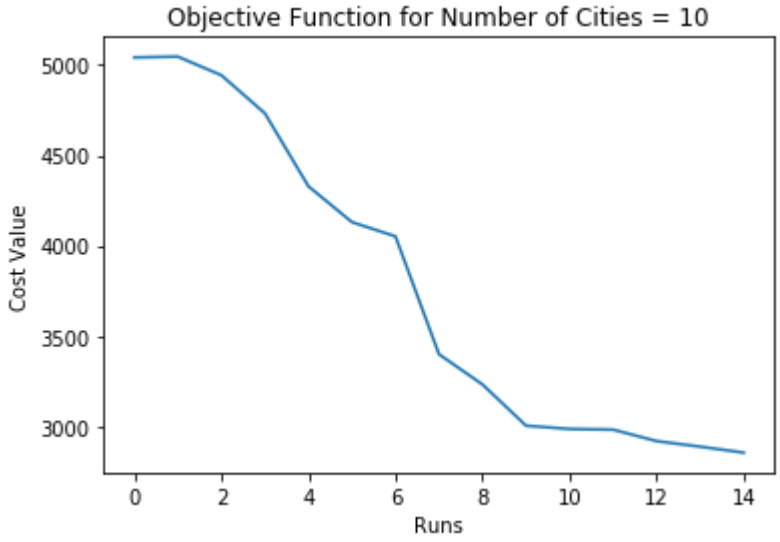
```
plot_city_tour(num_cities,prev_x,prev_y,'r','Random Initial Tour')
plot_city_tour(num_cities,new_x,new_y,'g','Optimized Route for Tour')
plt.figure()
plt.plot(final_cost)
plt.xlabel('Runs')
plt.ylabel('Cost Value')
plt.title('Objective Function for Number of Cities = '+str(num_cities))
```

Out[270]:

```
Text(0.5,1,'Objective Function for Number of Cities = 10')
```







N = 40 cities

In [273]:

```
num_cities = 40
start = time.time()
city_tour, init_tour, final_cost, c_loc = travel_anneal(num_cities,1000)
end = time.time()
print("Time for the TSP algorithm to run: ",end - start," seconds")
prev_x = list(zip(*[c_loc[init_tour[i % num_cities]] for i in range(num_cities+1)]))[
prev_y = list(zip(*[c_loc[init_tour[i % num_cities]] for i in range(num_cities+1)]))[
new_x = list(zip(*[c_loc[city_tour[i % num_cities]] for i in range(num_cities+1)]))[0
new_y = list(zip(*[c_loc[city_tour[i % num_cities]] for i in range(num_cities+1)]))[1
```

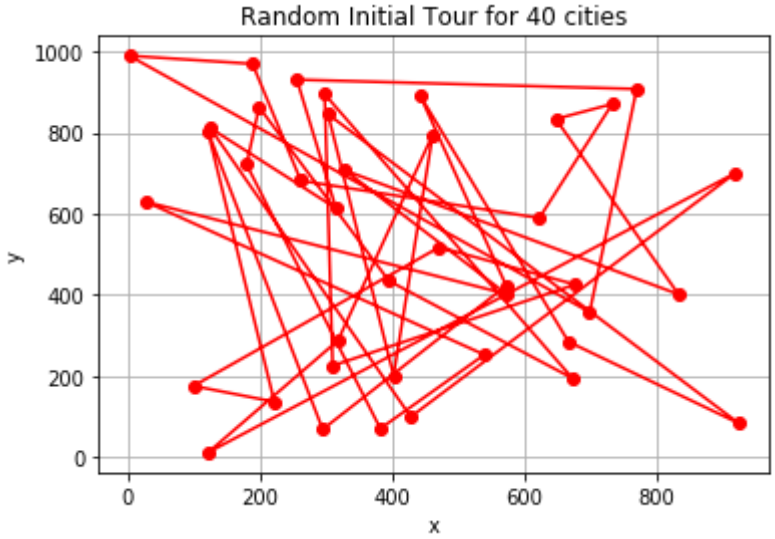
```
Time for the TSP algorithm to run:  0.6092278957366943  seconds
```

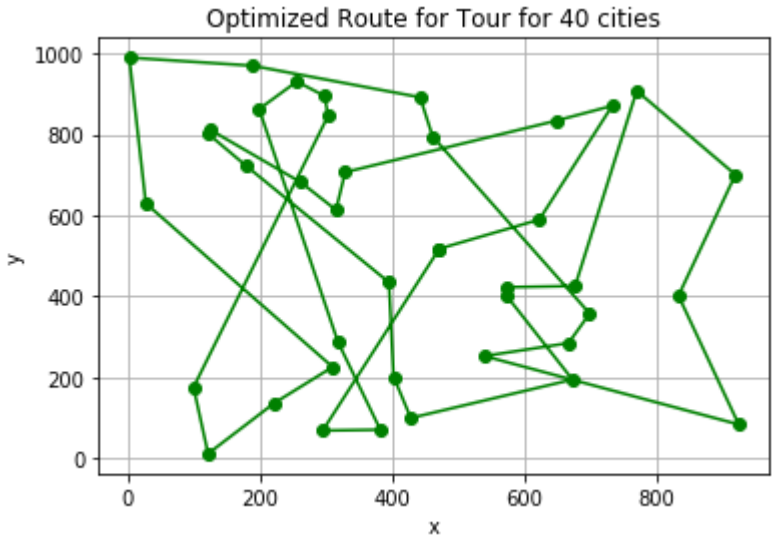
In [274]:

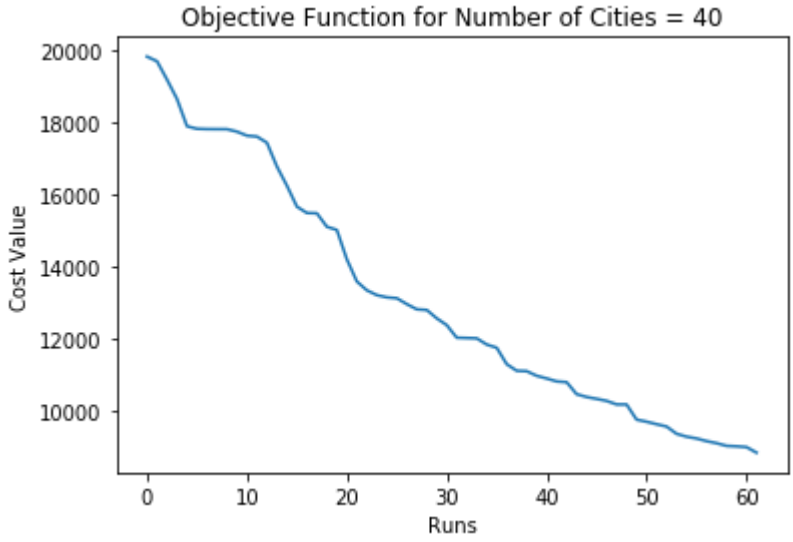
```
plot_city_tour(num_cities,prev_x,prev_y,'r','Random Initial Tour')
plot_city_tour(num_cities,new_x,new_y,'g','Optimized Route for Tour')
plt.figure()
plt.plot(final_cost)
plt.xlabel('Runs')
plt.ylabel('Cost Value')
plt.title('Objective Function for Number of Cities = '+str(num_cities))
```

Out[274]:

```
Text(0.5,1,'Objective Function for Number of Cities = 40')
```







N=400 cities

In [276]:

```
num_cities = 400
start = time.time()
city_tour, init_tour, final_cost, c_loc = travel_anneal(num_cities,1000)
end = time.time()
print("Time for the TSP algorithm to run: ",end - start," seconds")
prev_x = list(zip(*[c_loc[init_tour[i % num_cities]] for i in range(num_cities+1)]))[
prev_y = list(zip(*[c_loc[init_tour[i % num_cities]] for i in range(num_cities+1)]))[
new_x = list(zip(*[c_loc[city_tour[i % num_cities]] for i in range(num_cities+1)]))[0
new_y = list(zip(*[c_loc[city_tour[i % num_cities]] for i in range(num_cities+1)]))[1
```

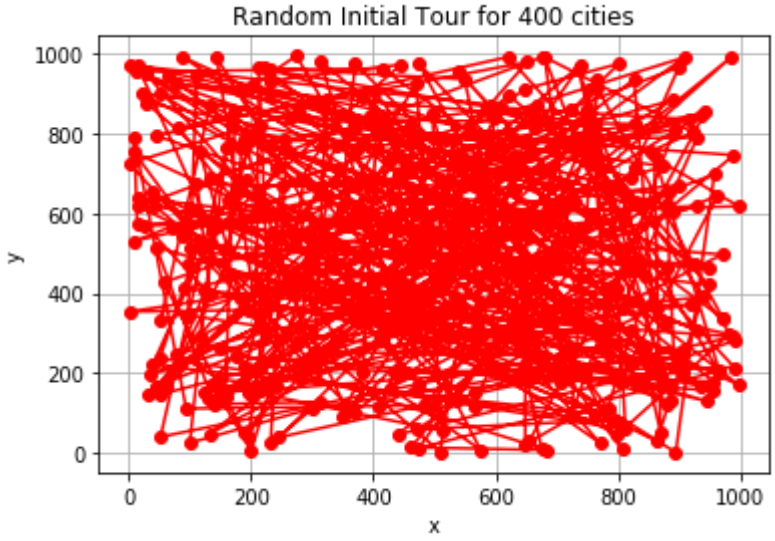
```
Time for the TSP algorithm to run:  6.704432487487793  seconds
```

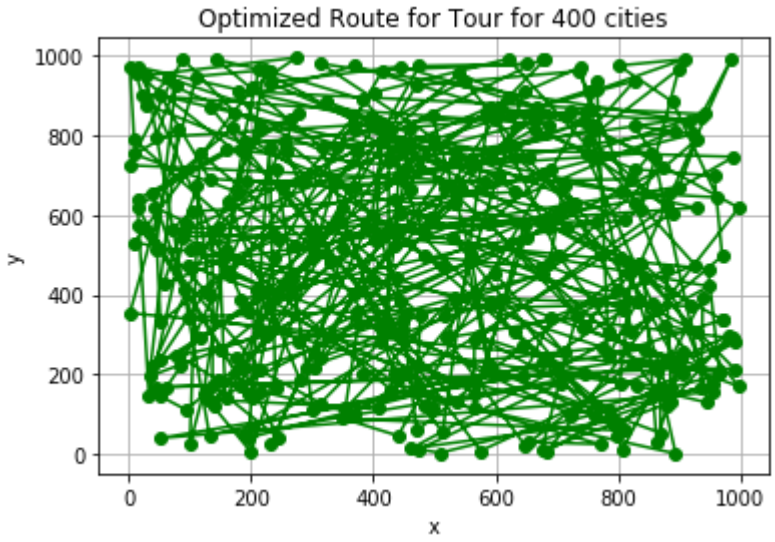
In [277]:

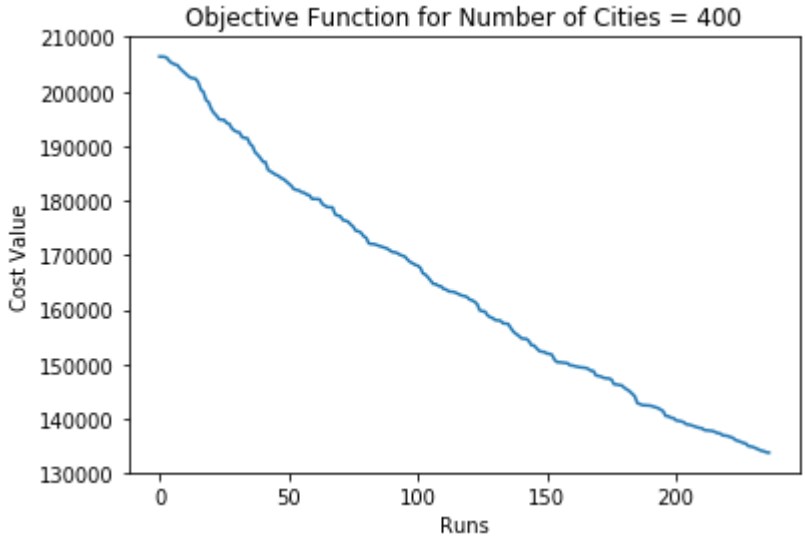
```
plot_city_tour(num_cities,prev_x,prev_y,'r','Random Initial Tour')
plot_city_tour(num_cities,new_x,new_y,'g','Optimized Route for Tour')
plt.figure()
plt.plot(final_cost)
plt.xlabel('Runs')
plt.ylabel('Cost Value')
plt.title('Objective Function for Number of Cities = '+str(num_cities))
```

Out[277]:

```
Text(0.5,1,'Objective Function for Number of Cities = 400')
```







N=1000

In [279]:

```
num_cities = 1000
start = time.time()
city_tour, init_tour, final_cost, c_loc = travel_anneal(num_cities,1000)
end = time.time()
print("Time for the TSP algorithm to run: ",end - start," seconds")
prev_x = list(zip(*[c_loc[init_tour[i % num_cities]] for i in range(num_cities+1)]))[
prev_y = list(zip(*[c_loc[init_tour[i % num_cities]] for i in range(num_cities+1)]))[
new_x = list(zip(*[c_loc[city_tour[i % num_cities]] for i in range(num_cities+1)]))[0
new_y = list(zip(*[c_loc[city_tour[i % num_cities]] for i in range(num_cities+1)]))[1
```

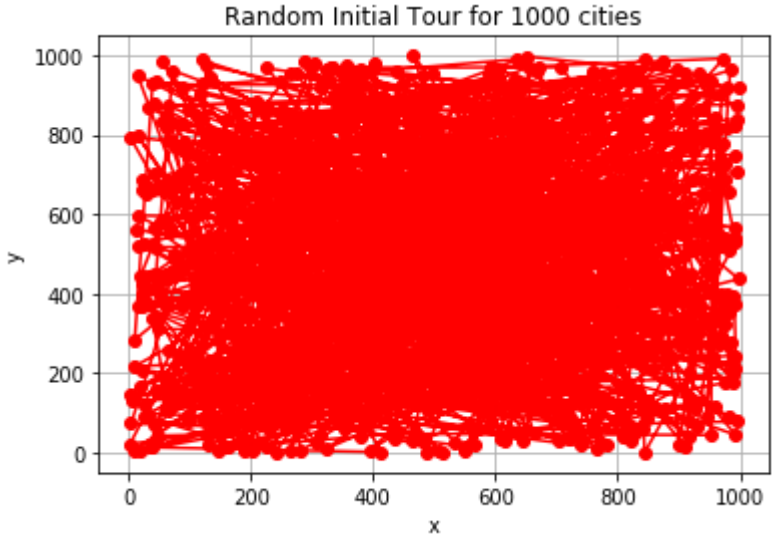
```
Time for the TSP algorithm to run:  14.408300638198853  seconds
```

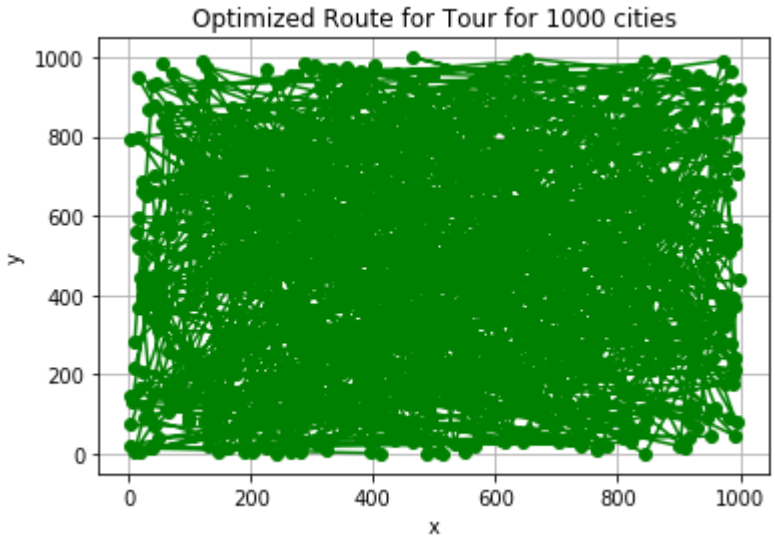
In [280]:

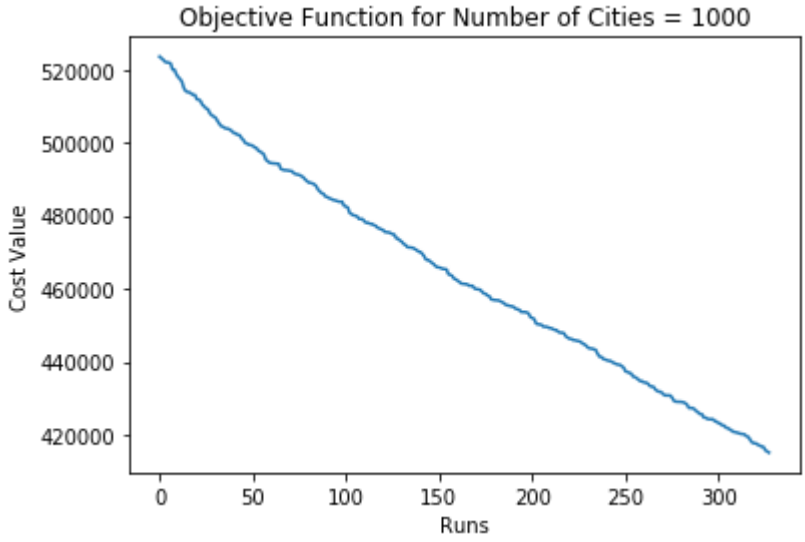
```
plot_city_tour(num_cities,prev_x,prev_y,'r','Random Initial Tour')
plot_city_tour(num_cities,new_x,new_y,'g','Optimized Route for Tour')
plt.figure()
plt.plot(final_cost)
plt.xlabel('Runs')
plt.ylabel('Cost Value')
plt.title('Objective Function for Number of Cities = '+str(num_cities))
```

Out[280]:

```
Text(0.5,1,'Objective Function for Number of Cities = 1000')
```







Summary:

1. The TSP algorithm is one of the most fascinating algorithms in the NP domain.

2. The TSP Algorithm has the following algorithm written in a pseudo-code format:
   A. Choose N = 10,40,400 and 1000 cities randomly points from the [0,1000]x[0,1000] rectangle
   B. These points represent the city locations.
   C. define an l2-norm objective function between a pair of cities D(□c)=Σ i=1 N−1 ‖ci+1−ci‖.
   D. Random path is chosen through all the N cities and an initial high temperature T0 is selected and a cooling schedule Tk=f (T0 , k) is defined.
   E. we pick two cities at random and swap their positions. we then calculate cost of each of the swapped configuration and the previous path cost.
   F. this is used in the cooling schedule and the accepted probability is calculated.
   G. we then follow the metro-hasting algorithm for selecting the new configurations untill convergence.

3. Objective function, Initial TSP city tour and Final TSP city tour is plotted and compared.

Results:

1. The values of the objective function were plotted and it was observed that as the number of cities increase the objective function becomes smoother that is , it takes more iterations to decrease rather than the cost falling sharply.

2. As Number of cities increases the algorithmic time increases.

3. The best cooling schedule was seen to be exponential schedule = T = T_init*(0.8^k)