Sina Mahbobi

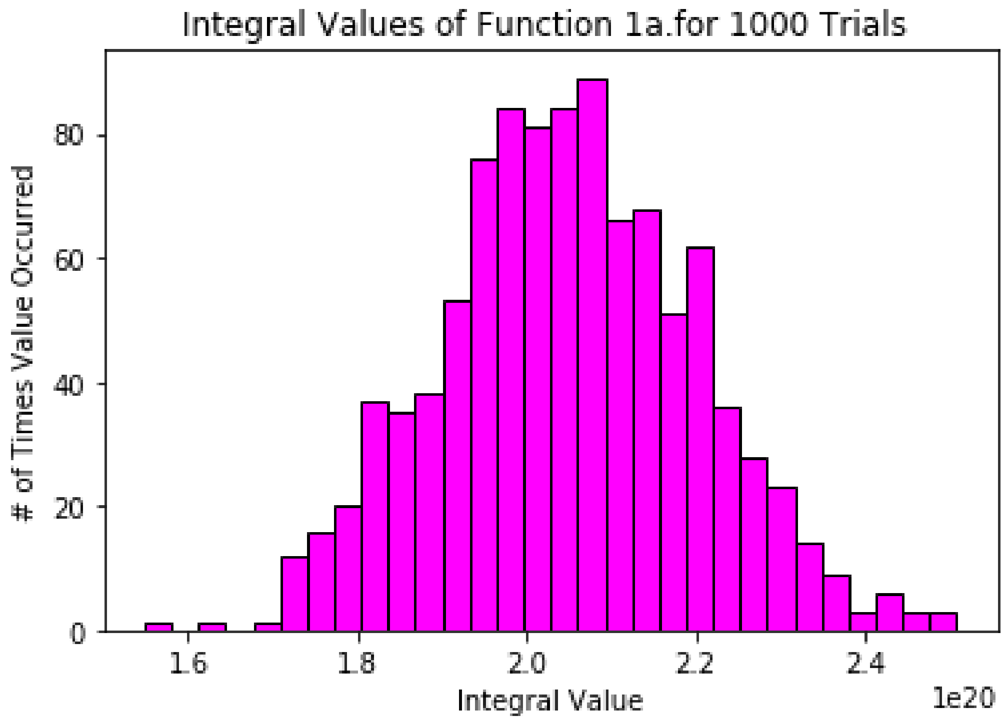
April 27, 2020

# 1 EE511 Project 8

### 1.0.1 Question 1

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
trials = 1000
samples = 1000
integrals = np.empty(trials)
integral_range = 1
def func_1(x1,x2):
    return np.exp(5* (np.abs(x1-5) + np.abs(x2-5)))

for i in range(0,trials):
    integral_sum = 0
    x1_rand = np.zeros(samples)
    x2_rand = np.zeros(samples)
    for j in range(0,samples):
        x1_rand[j] = np.random.rand()
        x2_rand[j] = np.random.rand()
        integral_sum += func_1(x1_rand[j], x2_rand[j])
    integral_sum = integral_sum * integral_range / samples
    integrals[i] = (integral_sum)
plt.hist(integrals, bins = 30, edgecolor = 'black', facecolor = 'magenta' )
plt.xlabel("Integral Value")
plt.ylabel("# of Times Value Occurred")
plt.title("Integral Values of Function 1a.for 1000 Trials ")
plt.style.use('fivethirtyeight')
plt.show()
```



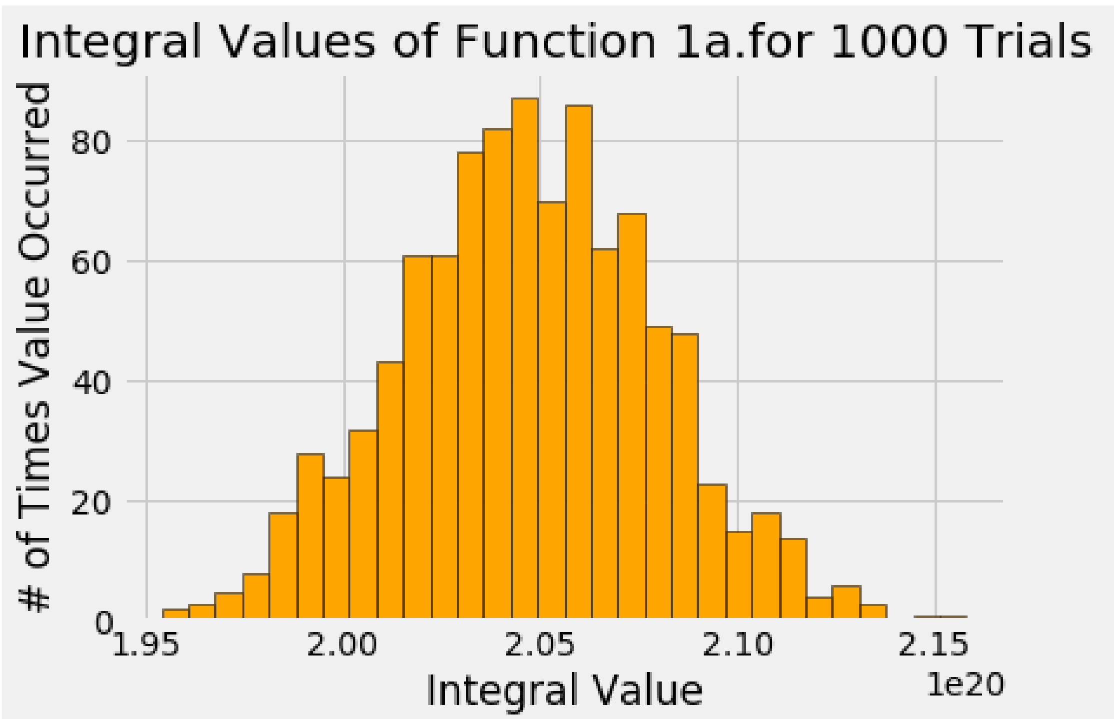

[3]:
```
print("True answer is 2.04e20")
print("Mean from 1000 trials: ", np.mean(integrals))
print("Variance from 1000 trials: ", np.var(integrals))
```

```
True answer is 2.04e20
Mean from 1000 trials:  2.0485767126098238e+20
Variance from 1000 trials:  2.259202305532061e+38
```

**Function 1a with Stratification**

[4]:
```
k = 10
integrals_strat = np.empty(trials)
n_ij = int(samples/(k*k)) #stratified into partitions of 10 with equal␣
 ↪probability for each xi
#print(n_ij)
for sample in range(0,1000):
    integral_strat = 0

    for i in range (0, k):
        for j in range(0, k):
            for l in range(0, n_ij):
                x1 = i + np.random.rand()
                x2 = j + np.random.rand()
                integral_strat += func_1(x1/k, x2/k)
```

```
    integrals_strat[sample] = (integral_strat/samples)
plt.hist(integrals_strat, bins = 30, edgecolor = 'black', facecolor = 'orange' )
plt.xlabel("Integral Value")
plt.ylabel("# of Times Value Occurred")
plt.title("Integral Values of Function 1a.for 1000 Trials ")
plt.style.use('fivethirtyeight')
plt.show()
```

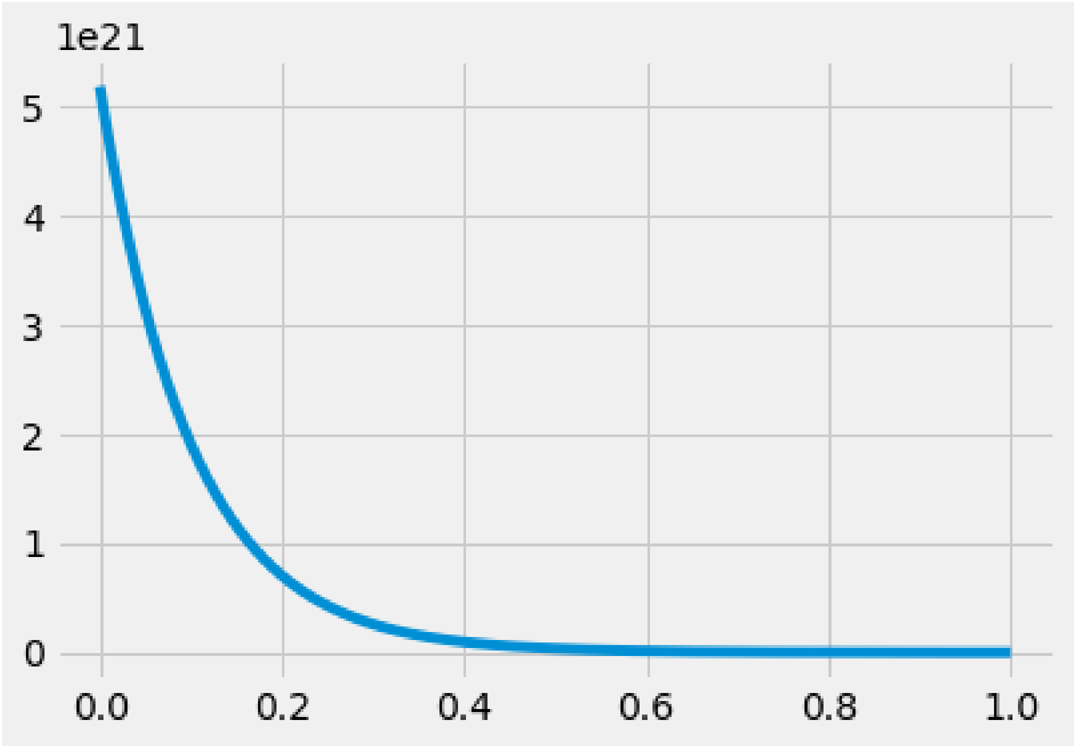



[5]:
```
print("True answer is 2.04e20")
print("Mean with stratified sampling from 1000 trials: ", np.
 ↪mean(integrals_strat))
print("Variance with stratified sampling from 1000 trials: ", np.mean(np.
 ↪var(integrals_strat)))
```

```
True answer is 2.04e20
Mean with stratified sampling from 1000 trials:  2.0470904919133936e+20
Variance with stratified sampling from 1000 trials:  1.0688941023125679e+37
```

#### 1a Importance Sampling

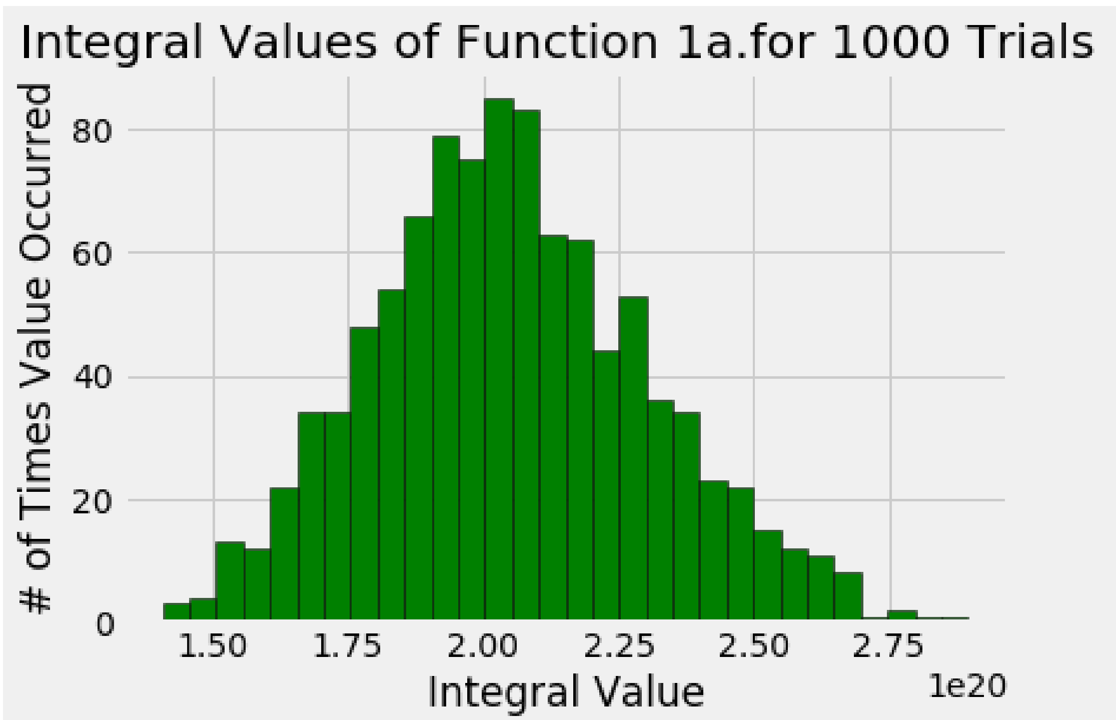
[6]:
```
x1 = np.linspace(0,1,100000)
x2 = np.linspace(0,1,100000)
y = np.exp(5* (np.abs(x1-5) + np.abs(x2-5)) )
#print(np.mean(y))
#print(np.var(y))
plt.plot(x1,y)
plt.show()
#for i in range(0,samples):
```




- Most of the contribution is for x1 and x2 is between 0 and 0.4, however this is at a large scale, 1e21, so even the smaller values have some contribution.

[7]:
```
integrals_importance = np.empty(trials)
integral_range = 1

for i in range(0,trials):
    integral_sum_imp = 0
    x1_rand = np.zeros(samples)
    x2_rand = np.zeros(samples)
    for j in range(0,samples):
        x1_rand[j] = np.log(1+ (np.exp(1)-1)*np.random.rand())
        x2_rand[j] = np.log(1+ (np.exp(1)-1)*np.random.rand())
        #scaling uniform values to fit range around 0 to 0.4
```

```
        integral_sum_imp += ((np.exp(1)-1) * (np.exp(1)-1) *np.exp(50 - 6 *␣
 ↪(x1_rand[j]+ x2_rand[j])))
    integral_sum_imp = integral_sum_imp * integral_range/1000
    integrals_importance[i] = (integral_sum_imp)
plt.hist(integrals_importance, bins = 30, edgecolor = 'black', facecolor =␣
 ↪'green' )

plt.xlabel("Integral Value")
plt.ylabel("# of Times Value Occurred")
plt.title("Integral Values of Function 1a.for 1000 Trials ")
plt.style.use('fivethirtyeight')
plt.show()
imp_var = 2*np.std(integrals_importance)/np.sqrt(samples)
```




[8]:
```
print("True answer is 2.04e20")
print("Mean with importance sampling from 1000 trials: ", np.
 ↪mean(integrals_importance))
print("Variance with importance sampling from 1000 trials: ", str(imp_var))
```

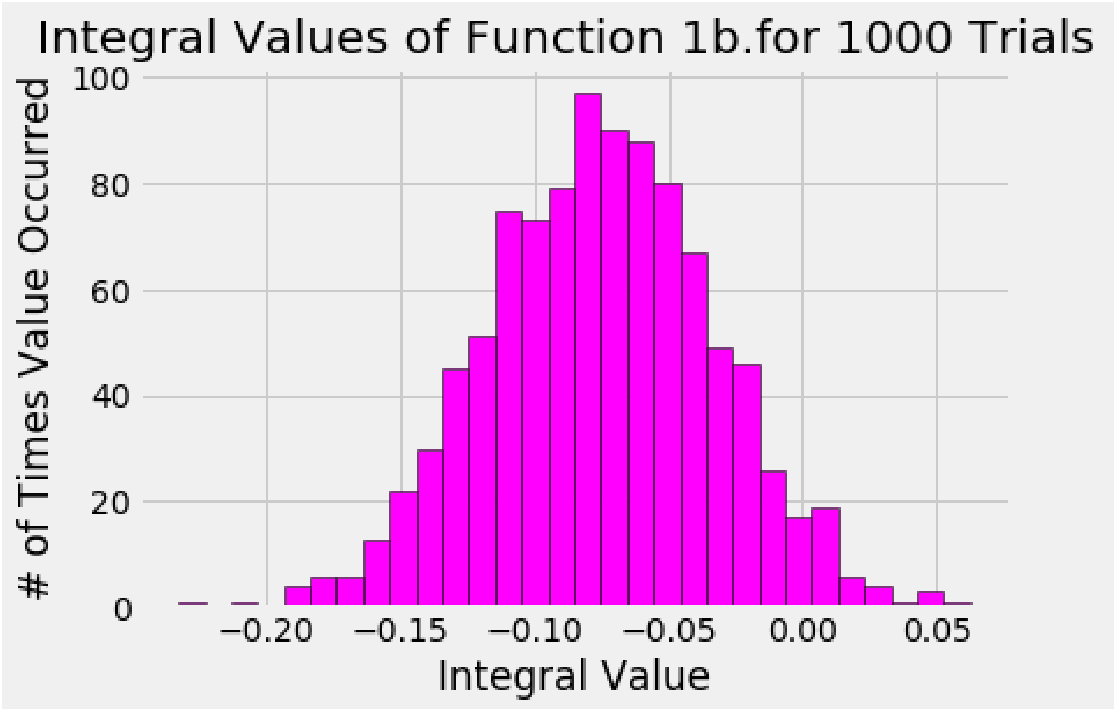
```
True answer is 2.04e20
Mean with importance sampling from 1000 trials:  2.0495795452807833e+20
Variance with importance sampling from 1000 trials:  1.6205463336303475e+18
```

### 1.0.2 1a Analysis

- The first graph is one where a standard Monte Carlo method was used. It determines the mean with good accuracy, but the variance is a little large, but the large variance makes sense given how large the values we are calculating are
- In the 2nd graph, where we used stratified sampling, the variance is reduced by a factor of 10. In the grand scheme of things however, this is not super significant since the variance is still incredible large.
- In the 3rd graph, we use importance sampling. Here, we get significant improvement of variance. The amount of variance drops by factor of 1e19, over 50% in reduction.

## 1.1 1b

[9]:
```
integrals_1b = np.empty(trials)
integral_range_1b = 2
def func_2(x1,x2):
    return np.cos(np.pi+ 5*x1+5*x2)


for i in range(0,trials):
    integral_sum_1b = 0
    x1_rand = np.zeros(samples)
    x2_rand = np.zeros(samples)
    for j in range(0,samples):
        x1_rand[j] = 2*np.random.rand()-1
        x2_rand[j] = 2*np.random.rand()-1
        integral_sum_1b += func_2(x1_rand[j], x2_rand[j])
    integral_sum_1b = integral_sum_1b*2 / samples
    integrals_1b[i] = (integral_sum_1b)



plt.hist(integrals_1b, bins = 30, edgecolor = 'black', facecolor = 'magenta' )
plt.xlabel("Integral Value")
plt.ylabel("# of Times Value Occurred")
plt.title("Integral Values of Function 1b.for 1000 Trials ")
plt.style.use('fivethirtyeight')
plt.show()
```

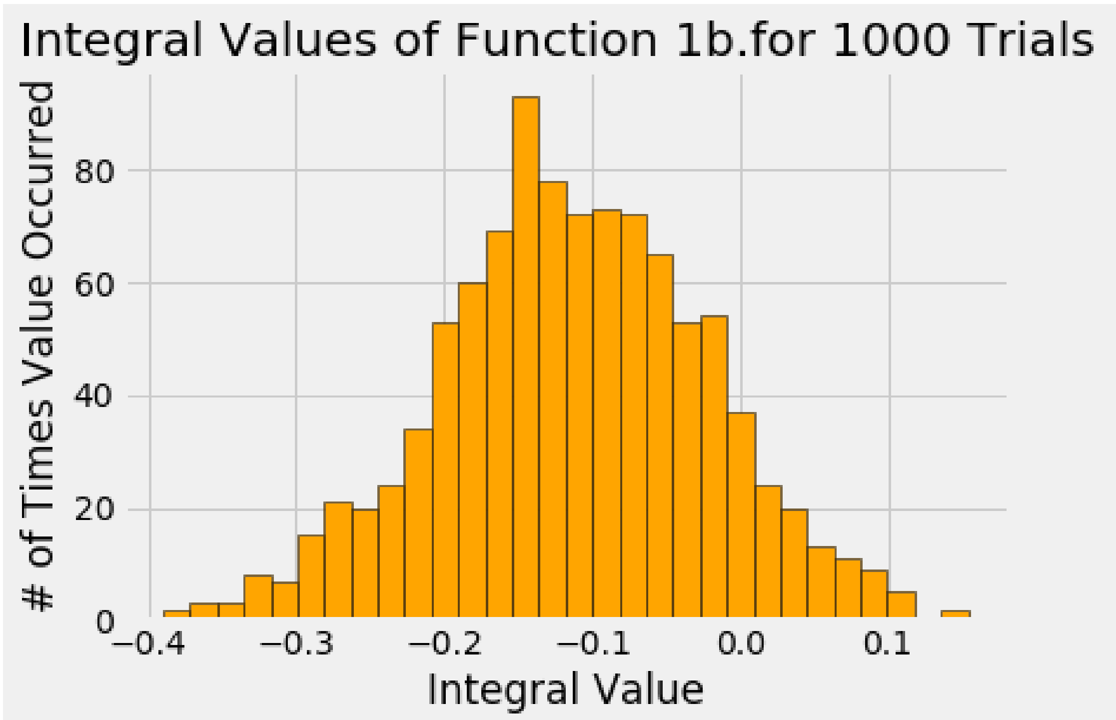



[10]:
```
print("True answer is -0.147")
print("Mean from 1000 trials: ", np.mean(integrals_1b))
print("Variance from 1000 trials: ", np.var(integrals_1b))
```

```
True answer is -0.147
Mean from 1000 trials:  -0.07571515303129409
Variance from 1000 trials:  0.001817137574374486
```

[11]:
```
k = 10
integrals_strat_1b = np.empty(trials)
n_ij = int(samples/(k*k)) #stratified into partitions of 10 with equal␣
↪probability for each xi
#print(n_ij)
for sample in range(0,1000):
    integral_strat_1b = 0

    for i in range (0, k):
        for j in range(0, k):
            for l in range(0, n_ij):
                x1 = i + np.random.uniform(-1,1)
                x2 = j + np.random.uniform(-1,1)
                integral_strat_1b += func_2(x1/k, x2/k)

    integrals_strat_1b[sample] = (-integral_strat_1b/100)
```

```
plt.hist(integrals_strat_1b, bins = 30, edgecolor = 'black', facecolor =␣
 ↪'orange' )
plt.xlabel("Integral Value")
plt.ylabel("# of Times Value Occurred")
plt.title("Integral Values of Function 1b.for 1000 Trials ")
plt.style.use('fivethirtyeight')
plt.show()
```

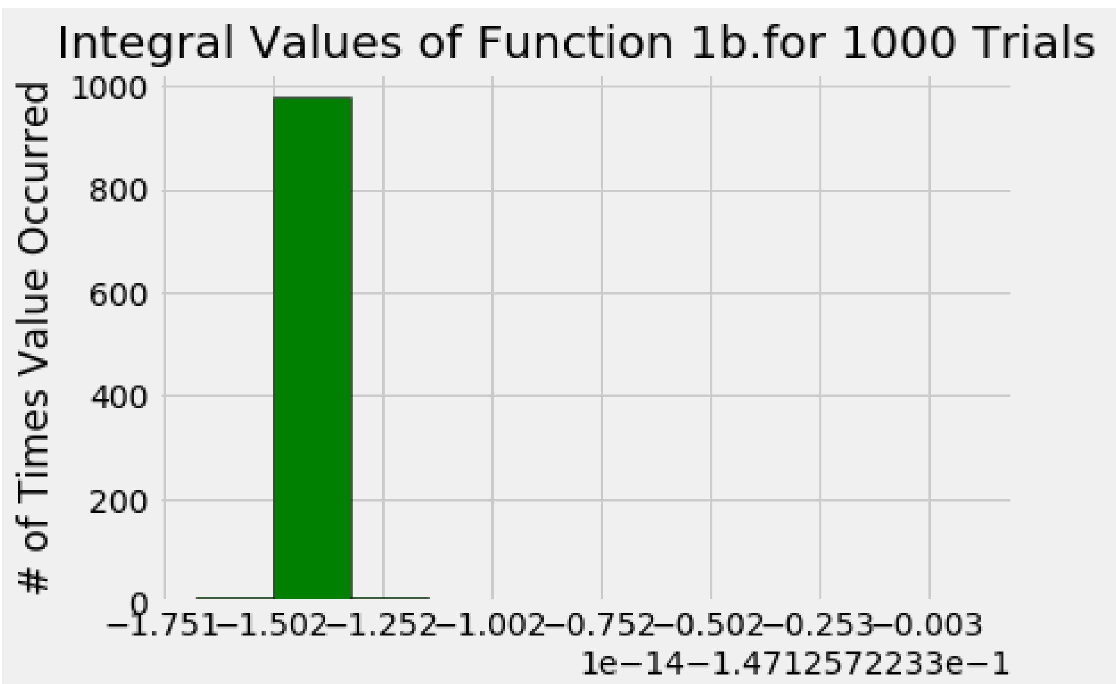



[12]:
```
print("True answer is -0.147")
print("Mean from 1000 trials: ", np.mean(integrals_strat_1b))
print("Variance from 1000 trials: ", np.var(integrals_strat_1b))
```

```
True answer is -0.147
Mean from 1000 trials:  -0.11351836966657512
Variance from 1000 trials:  0.008159318728183181
```

[13]:
```
integrals_importance_1b = np.empty(trials)
integral_range = 2

for i in range(0,trials):
    integral_sum_imp_1b = 0
    x1_rand = np.zeros(samples)
    x2_rand = np.zeros(samples)
    for j in range(0,samples):
```

```
        x1_rand[j] = np.arcsin(2*np.random.rand()*np.sin(5) - np.sin(5))/5  #np.
↪log(1+ (np.exp(1)-1)*np.random.rand())
        x2_rand[j] = np.arcsin(2*np.random.rand()*np.sin(5) - np.sin(5))/5 #np.
↪log(1+ (np.exp(1)-1)*np.random.rand())
        #scaling uniform values to fit range around 0 to 0.4
        integral_sum_imp_1b += (( (2*np.sin(5)/5) * (2*np.sin(5)/5) \
           * np.cos(np.pi + 5*(x1_rand[j] + x2_rand[j]))) / np.cos(5␣
↪*(x1_rand[j] + x2_rand[j])) )

    integral_sum_imp_1b = integral_sum_imp_1b /1000
    integrals_importance_1b[i] = (integral_sum_imp_1b)
plt.hist(integrals_importance_1b, edgecolor = 'black', facecolor = 'green' )

#plt.xlabel("Integral Value")
plt.ylabel("# of Times Value Occurred")
plt.title("Integral Values of Function 1b.for 1000 Trials ")
plt.style.use('fivethirtyeight')

plt.show()
imp_var_1b = 2*np.std(integrals_importance_1b)/np.sqrt(samples)
```




```
[14]: print("True answer is -0.147")
print("Mean with importance sampling from 1000 trials: ", np.
 ↪mean(integrals_importance_1b))
```

```
print("Variance with importance sampling from 1000 trials: ", str(imp_var_1b))
```

```
True answer is -0.147
Mean with importance sampling from 1000 trials:  -0.1471257223261144
Variance with importance sampling from 1000 trials:  3.763975351866937e-17
```

### 1.1.1 1b Analysis

- The first graph is one where a standard Monte Carlo method was used. It does not determine the mean with good accuracy where the estimated value is only half the true value, but the variance is very small, which makes sense given the true value of the integral is small as well, around -0.147.
- In the 2nd graph, where we used stratified sampling, we get an improvement in the mean estimation. It comes to within a few hundreths of the true mean, but here we get slightly worse variance.
- In the 3rd graph, we use importance sampling. Here, we get significant improvement of variance from stratified sampling. We get a reduction in variance of over 13 orders of magnitude, and the estimation of the mean comes to within a ten thousandth of the true value.

[ ]:

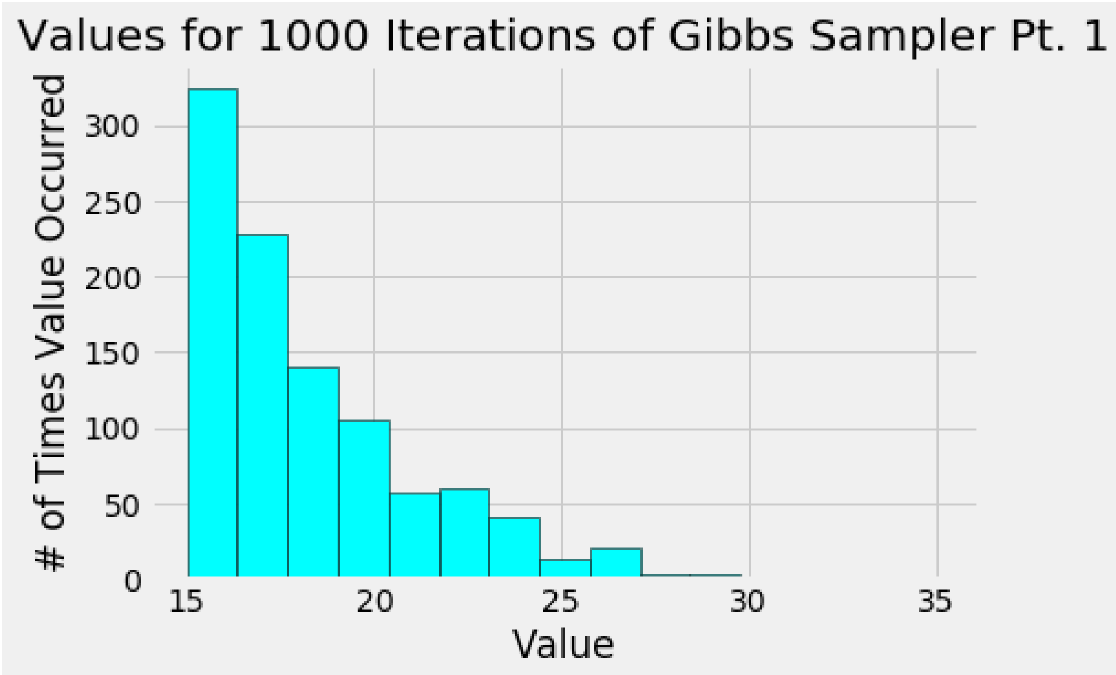
### 1.1.2 Question 2

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
iterations = 1000
trials = 10
sample1 = np.zeros((3,iterations))
sample1_vals = np.zeros(1000)

while 1: #initialize
    x = np.random.exponential(size = 3)
    if (x[0] + 2*x[1] + 3* x[2] > 15):
        sample1[0,0] = x[0]
        sample1[1,0] = x[1]
        sample1[2,0] = x[2]
        break

sample1_vals[0] = sample1[0,0] + 2* sample1[1,0] + 3* sample1[2,0]
for i in range(1,iterations):
    u = np.random.random()
    if ( u < (1/3) ):
        while 1:
            temp0 = np.random.exponential()
            if( temp0 + 2*x[1] + 3*x[2] > 15):
                x[0] = temp0
```

```
                break
    elif( u < (2/3) ):
        while 1:
            temp1 = np.random.exponential()
            if( x[0] + 2* temp1 + 3*x[2] > 15):
                x[1] = temp1
                break
    else:
        while 1:
            temp2 = np.random.exponential()
            if( x[0] + 2* x[1] + 3*temp2 > 15):
                x[2] = temp2
                break
    sample1[0,i] = x[0]
    sample1[1,i] = x[1]
    sample1[2,i] = x[2]

    sample1_vals[i] = sample1[0,i] + 2* sample1[1,i] + 3* sample1[2,i]
sample1_mean = np.mean(sample1_vals)
```

[7]:
```
plt.hist(sample1_vals, bins = 15, edgecolor = 'black', facecolor = 'cyan' )
#plt.xticks(np.arange(70,120))
plt.xlabel("Value")
plt.ylabel("# of Times Value Occurred")
plt.title("Values for 1000 Iterations of Gibbs Sampler Pt. 1")
plt.style.use('fivethirtyeight')
plt.show()
print("Expectation of 1000 iterations: ", str(sample1_mean))
```

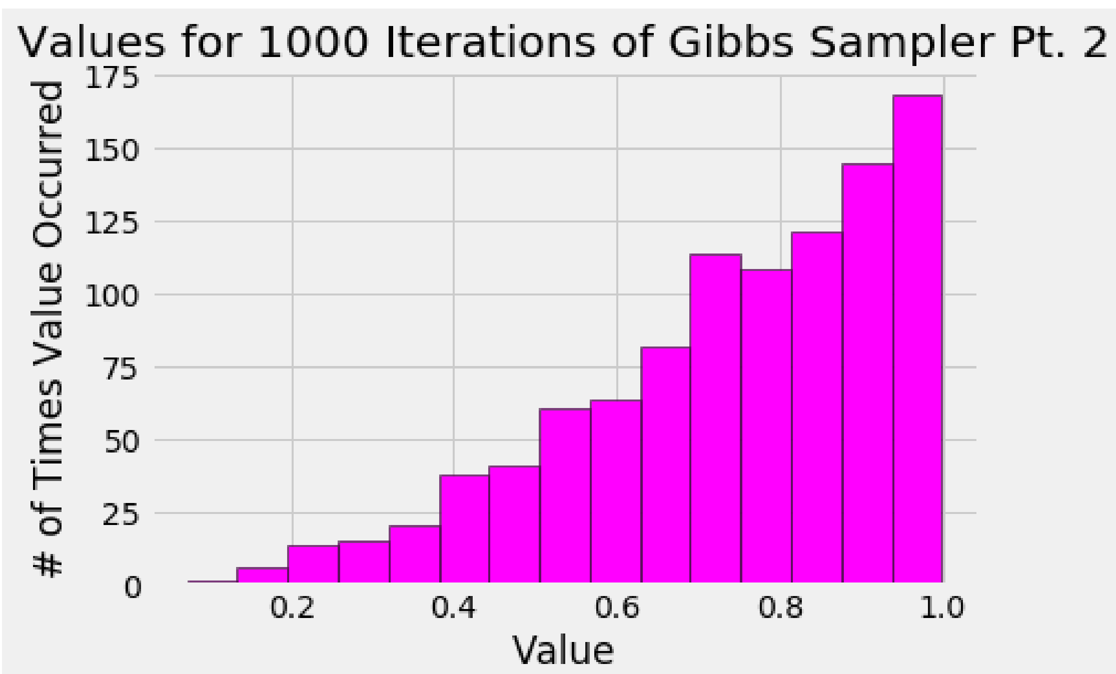



```
Expectation of 1000 iterations:  18.23771709100648
```

[4]:
```
sample2 = np.zeros((3,iterations))
sample2_vals = np.zeros(1000)

while 1: #initialize
    x = np.random.exponential(size = 3)
    if (x[0] + 2*x[1] + 3* x[2] < 1):
        sample2[0,0] = x[0]
        sample2[1,0] = x[1]
        sample2[2,0] = x[2]
        break

sample2_vals[0] = sample2[0,0] + 2* sample2[1,0] + 3* sample2[2,0]
for i in range(1,iterations):
    u = np.random.random()
    if ( u < (1/3) ):
        while 1:
            temp0 = np.random.exponential()
            if( temp0 + 2*x[1] + 3*x[2] < 1):
                x[0] = temp0
                break
    elif( u < (2/3) ):
        while 1:
            temp1 = np.random.exponential()
```

```
            if( x[0] + 2* temp1 + 3*x[2] < 1):
                x[1] = temp1
                break
    else:
        while 1:
            temp2 = np.random.exponential()
            if( x[0] + 2* x[1] + 3*temp2 < 1):
                x[2] = temp2
                break
    sample2[0,i] = x[0]
    sample2[1,i] = x[1]
    sample2[2,i] = x[2]

    sample2_vals[i] = sample2[0,i] + 2* sample2[1,i] + 3* sample2[2,i]
sample2_mean = np.mean(sample2_vals)
```

[8]:
```
plt.hist(sample2_vals, bins = 15, edgecolor = 'black', facecolor = 'magenta' )
#plt.xticks(np.arange(70,120))
plt.xlabel("Value")
plt.ylabel("# of Times Value Occurred")
plt.title("Values for 1000 Iterations of Gibbs Sampler Pt. 2")
plt.style.use('fivethirtyeight')
plt.show()
print("Expectation of 1000 iterations: ", str(sample2_mean))
```



```
Expectation of 1000 iterations:  0.7414614256522195
```

**Question 2 Analysis**

- The expected value after 1000 iterations for the first conditional 18.24
- The expected value after 1000 iterations for the 2nd conditional 0.74
- For both these processes, I used a random number generator to decide which random variable of X1, X2, X3, would be updated, and which two would be held constant. So if u < 1/3, it picks X1, else if u < 2/3 it picks X2, and else it picks X3.

- From there, I set a temp variable equal to a value randomly generated from the exponential distribution, if the conditional held true with the temp variable substituted in, that temp variable would be set to the random variable being updated, and this was done for 1000 iterations.

[ ]:

```

```

### 1.1.3 Question 3 Log Cooling

[1]:

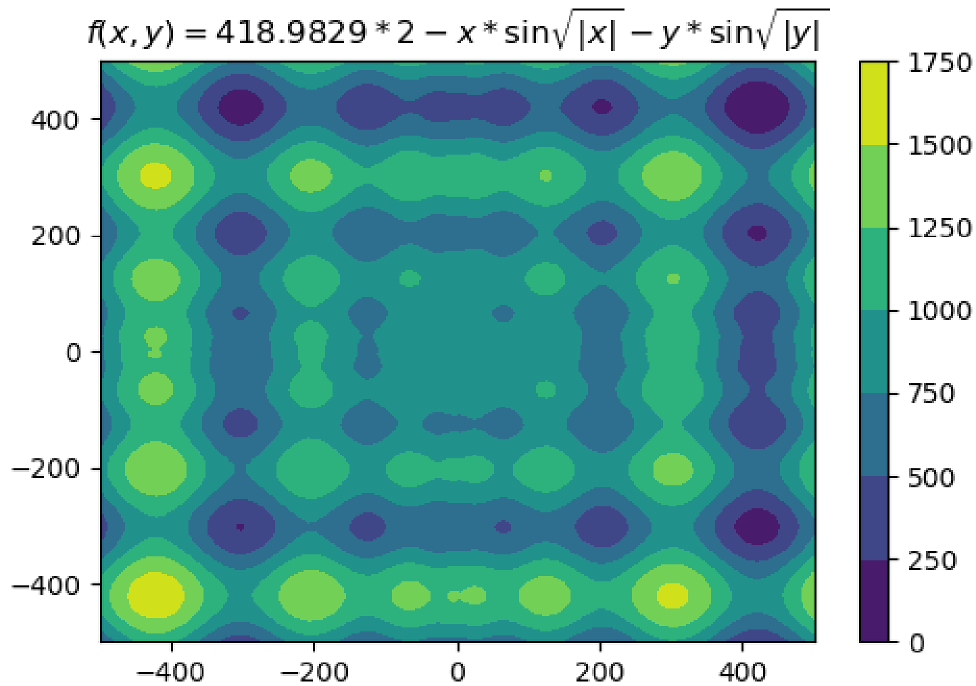
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
```

[2]:

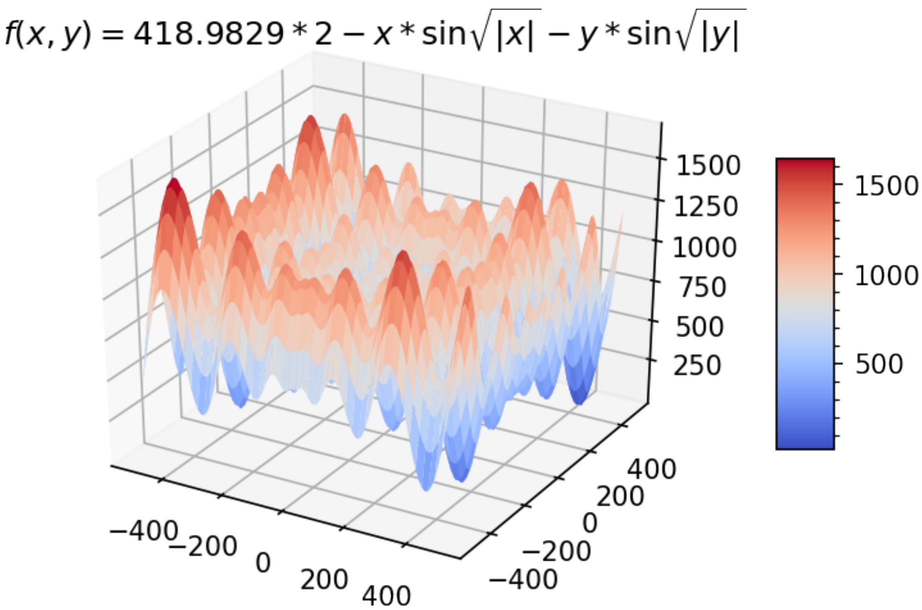
```
def schwefel_2d(x,y):
    return 418.9829 * 2 - (x*np.sin(np.sqrt(np.abs(x))) + y * np.sin(np.sqrt(np.
 ↪abs(y))) )
```

[3]:

```
N_r = 500
x = np.linspace(-N_r,N_r,100)
y = np.linspace(-N_r,N_r,100)
X, Y = np.meshgrid(x, y)
Z = schwefel_2d(X,Y)

plt.figure(num=None,dpi=100)
plt.contourf(X,Y,Z)
plt.title('$f(x,y) = 418.9829 * 2 -x*\sin \sqrt{|x|} - y *\sin \sqrt{|y|}$')
plt.colorbar()
plt.show()
```

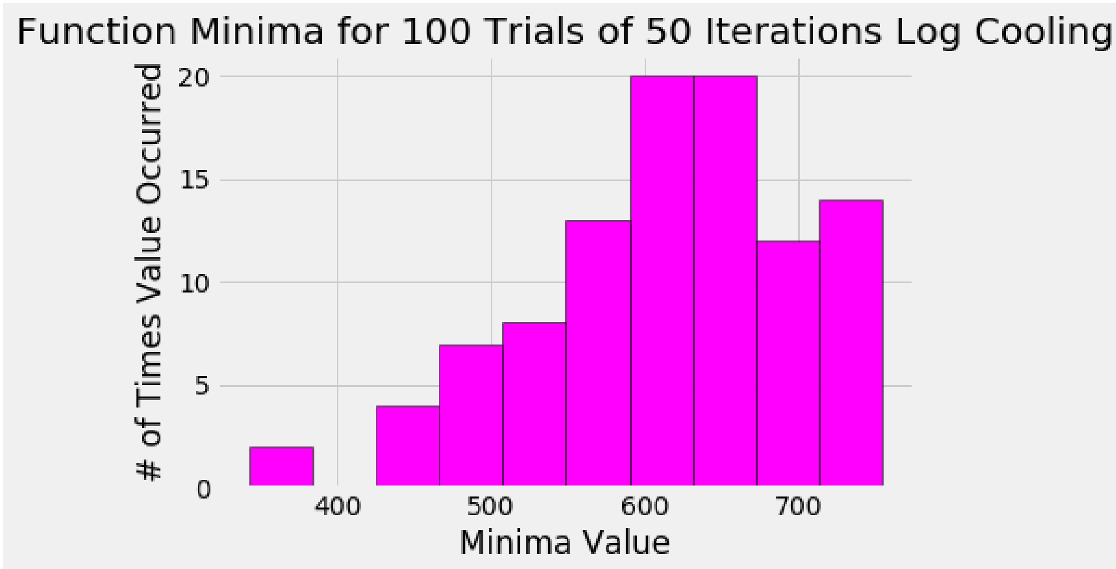



[4]:
```
cplot = plt.figure(num=None,dpi=150)
ax = cplot.add_subplot(111,projection='3d')
surf = ax.plot_surface(X,Y,Z,cmap=cm.coolwarm)
cbar = cplot.colorbar(surf, shrink=0.5, aspect=5)
cbar.minorticks_on()
plt.title('$f(x,y) = 418.9829 * 2 -x*\sin \sqrt{|x|} - y *\sin \sqrt{|y|}$')
plt.show()
```

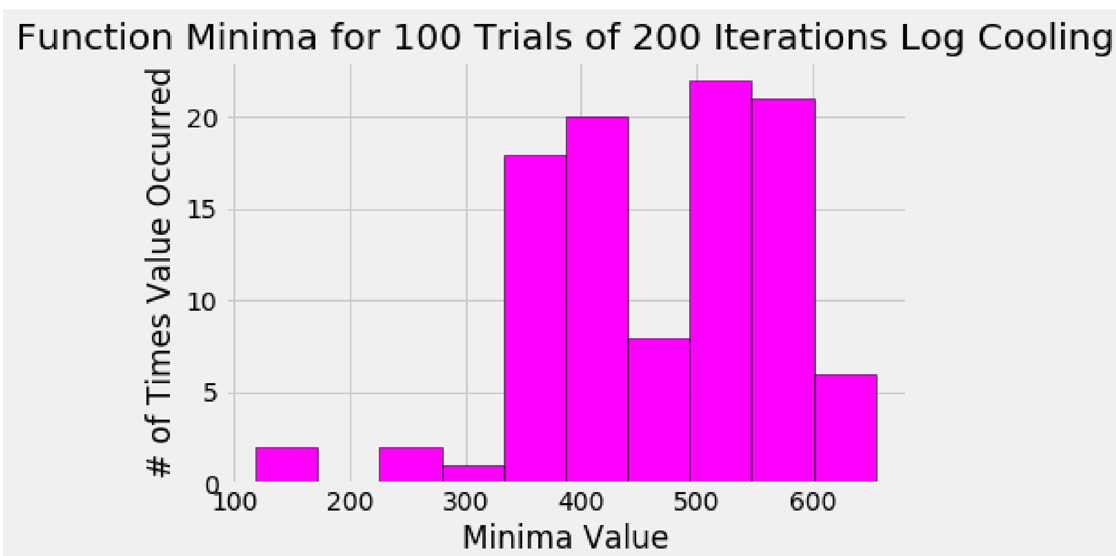



[32]:
```
X0 = 0
Y0 = 0
T0 = 500
N = [50, 200, 1000,10000]
trials = 100

for n in range(0,len(N)):
    min_outputs = np.zeros(trials)
    min_x = np.zeros(trials)
    min_y = np.zeros(trials)
    for trial in range(0,trials):

        X = np.zeros(N[n])
        Y = np.zeros(N[n])
        xy_output = np.zeros(N[n])
        X[0] = 0
        Y[0] = 0
        T = T0
        xy_output[0] = schwefel_2d(X[0], Y[0])
        for i in range(1,N[n]):
            while 1:
                X_temp = X[i-1] + np.random.normal(0,25)
                Y_temp = Y[i-1] + np.random.normal(0,25)
                if ((np.abs(X_temp) < 500) and (np.abs(Y_temp) < 500) ):
                    break
            alpha = np.exp((schwefel_2d(X[i-1], Y[i-1]) - schwefel_2d(X_temp,␣
 ↪Y_temp))/T)
```

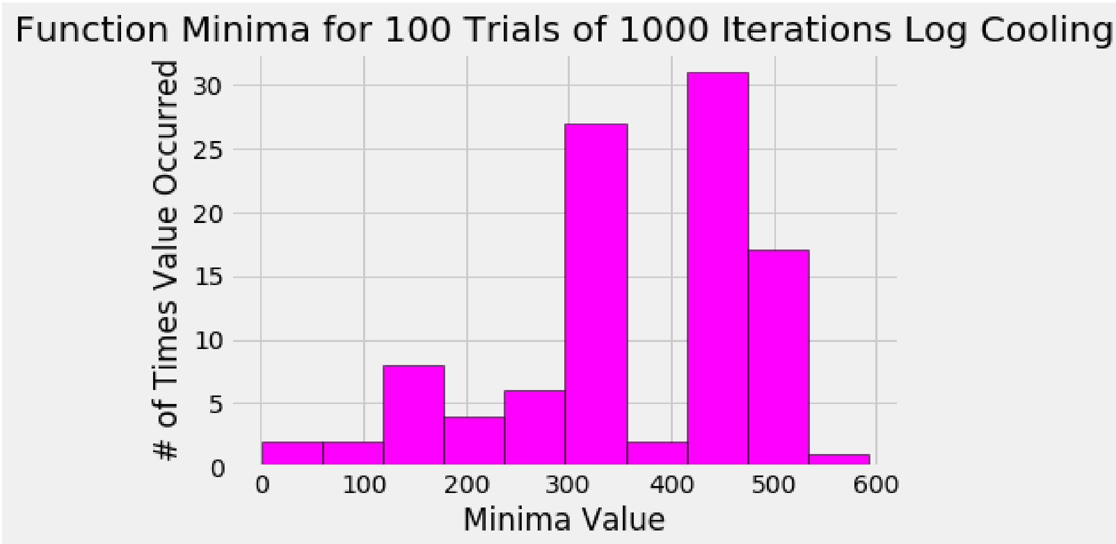
```
            if(schwefel_2d(X_temp, Y_temp) <= schwefel_2d(X[i-1], Y[i-1])):
                X[i] = X_temp
                Y[i] = Y_temp
                xy_output[i] = schwefel_2d(X_temp, Y_temp)
            elif (np.random.uniform() < alpha):
                X[i] = X_temp
                Y[i] = Y_temp
                xy_output[i] = schwefel_2d(X_temp, Y_temp)
            else:
                X[i] = X[i-1]
                Y[i] = Y[i-1]
                xy_output[i] = schwefel_2d( X[i-1], Y[i-1])
            T = T0/np.log(i+1)
        min_outputs[trial] = min(xy_output)

        min_x[trial] = X[np.argmin(xy_output)]
        min_y[trial] = Y[np.argmin(xy_output)]
    #print(min_outputs)
    plt.hist(min_outputs, bins = 10, edgecolor = 'black', facecolor = 'magenta' )
#plt.xticks(np.arange(70,120))
    plt.xlabel("Minima Value")
    plt.ylabel("# of Times Value Occurred")
    plt.title("Function Minima for 100 Trials of {number} Iterations Log␣
↪Cooling".format(number=N[n]))
    plt.style.use('fivethirtyeight')
    plt.show()
    x_min_trials = min_x[np.argmin(min_outputs)]
    y_min_trials = min_y[np.argmin(min_outputs)]
    print("Minimum X,Y pair from 100 trials for {number} Iterations: ({X}, {Y})".
↪format(X = x_min_trials,
            Y = y_min_trials, number=N[n]))
    print("Function minima value for X,Y pair: ", min(min_outputs))
```



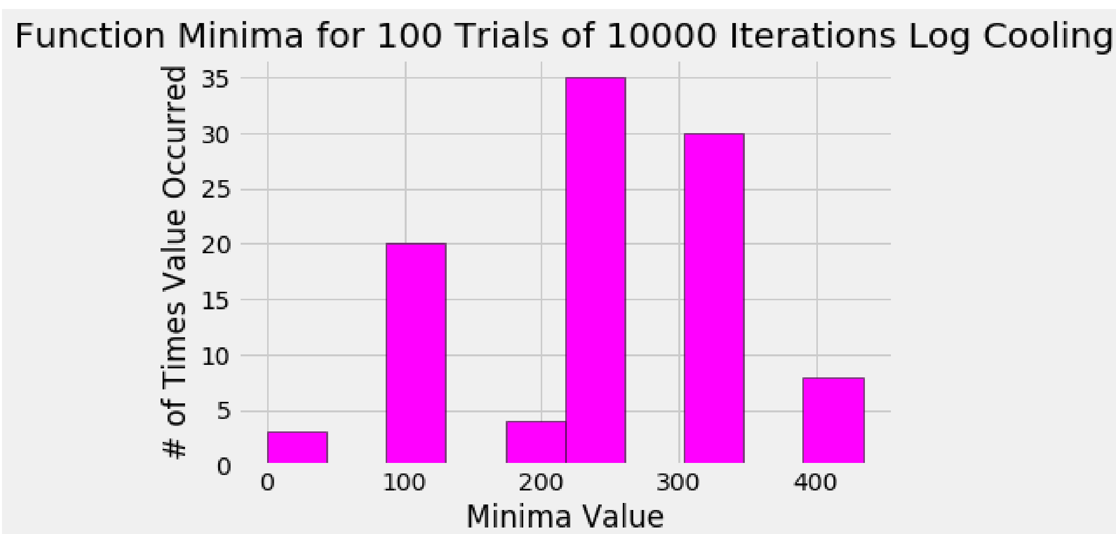

Minimum X,Y pair from 100 trials for 50 Iterations: (210.0199055089765, -298.56540610088445)
Function minima value for X,Y pair:  342.468431428446




Minimum X,Y pair from 100 trials for 200 Iterations: (-301.8231250418515, 422.04807781673105)
Function minima value for X,Y pair:  118.64776278272791



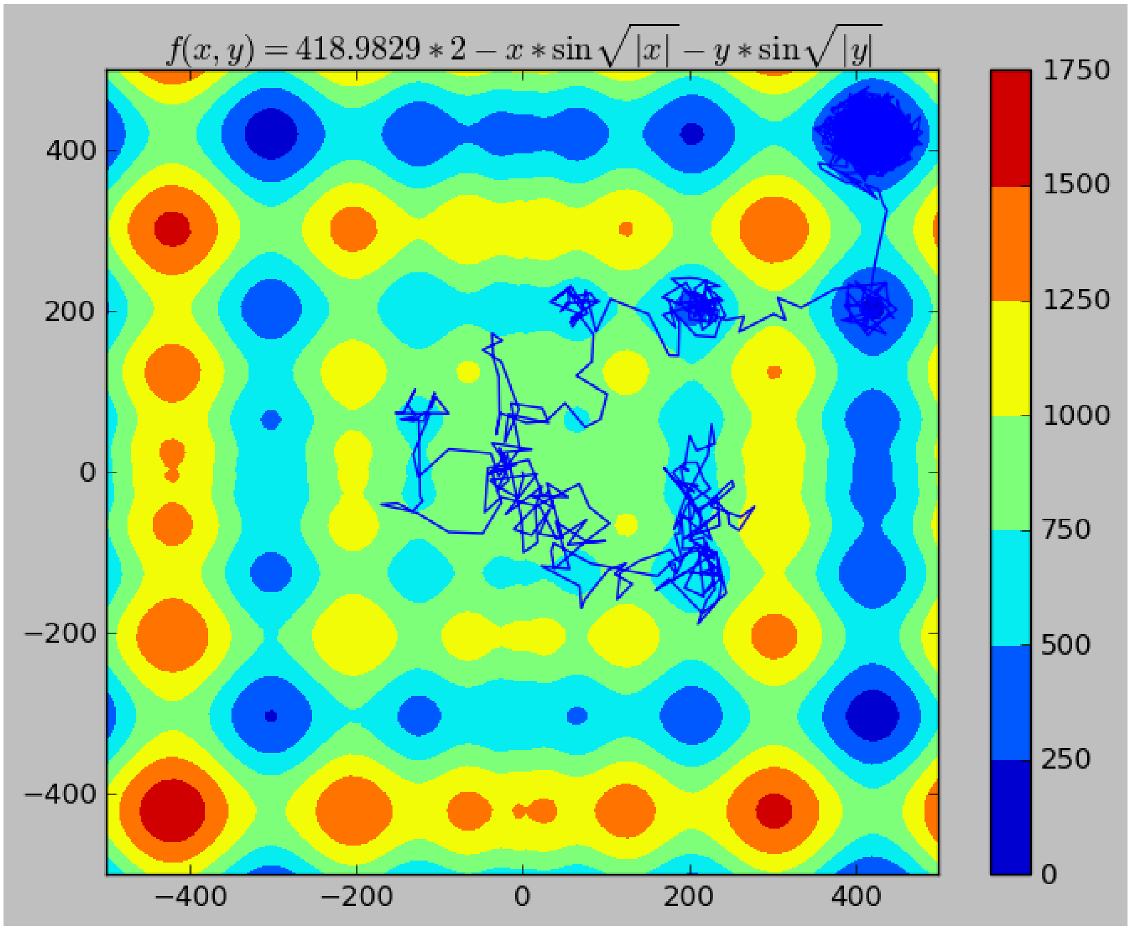

```
Minimum X,Y pair from 100 trials for 1000 Iterations: (421.7041203154593,
420.78889906942567)
Function minima value for X,Y pair:  0.07236091219147056
```




```
Minimum X,Y pair from 100 trials for 10000 Iterations: (420.9790620399202,
421.0199101834157)
Function minima value for X,Y pair:  0.00036920452168942575
```

[62]:
```
X = np.zeros(10000)
Y = np.zeros(10000)
xy_output = np.zeros(10000)
```

```python
X[0] = 0
Y[0] = 0
T = T0
xy_output[0] = schwefel_2d(X[0], Y[0])
for i in range(1,10000):
    while 1:
        X_temp = X[i-1] + np.random.normal(0,25)
        Y_temp = Y[i-1] + np.random.normal(0,25)
        if ((np.abs(X_temp) < 500) and (np.abs(Y_temp) < 500) ):
            break
    alpha = np.exp((schwefel_2d(X[i-1], Y[i-1]) - schwefel_2d(X_temp, Y_temp))/T)

    if(schwefel_2d(X_temp, Y_temp) <= schwefel_2d(X[i-1], Y[i-1])):
        X[i] = X_temp
        Y[i] = Y_temp
        xy_output[i] = schwefel_2d(X_temp, Y_temp)
    elif (np.random.uniform() < alpha):
        X[i] = X_temp
        Y[i] = Y_temp
        xy_output[i] = schwefel_2d(X_temp, Y_temp)
    else:
        X[i] = X[i-1]
        Y[i] = Y[i-1]
        xy_output[i] = schwefel_2d( X[i-1], Y[i-1])
    T = T0/np.log(i+1)

N_r = 500
x = np.linspace(-N_r,N_r,100)
y = np.linspace(-N_r,N_r,100)
x, y = np.meshgrid(x, y)
Z = schwefel_2d(x,y)

plt.style.use('classic')
plt.figure(num=None,dpi=100)
plt.contourf(x,y,Z)
plt.title('$f(x,y) = 418.9829 * 2 -x*\sin \sqrt{|x|} - y *\sin \sqrt{|y|}$')
plt.plot(X,Y)
plt.colorbar()
plt.show()
```




**Question 3 (Log Cooling) Analysis**

- The first 4 Histograms are for 100 trials at the 4 different iteration values: 50, 200, 1000, and 10000 using logarithmic temperature cooling.
- As evident from the histograms, as the number of iterations increase, the number of trials with which the global minima is approached increases, and the best estimate of the 100 trials at that iteration count gets better and better.
- The countour map above is overlayed with the path of a trial that approaches the global minima. As one can tell, it jumps from local minima to local minima, eventually settling closer and closer to the global minima.

```
[ ]:
```

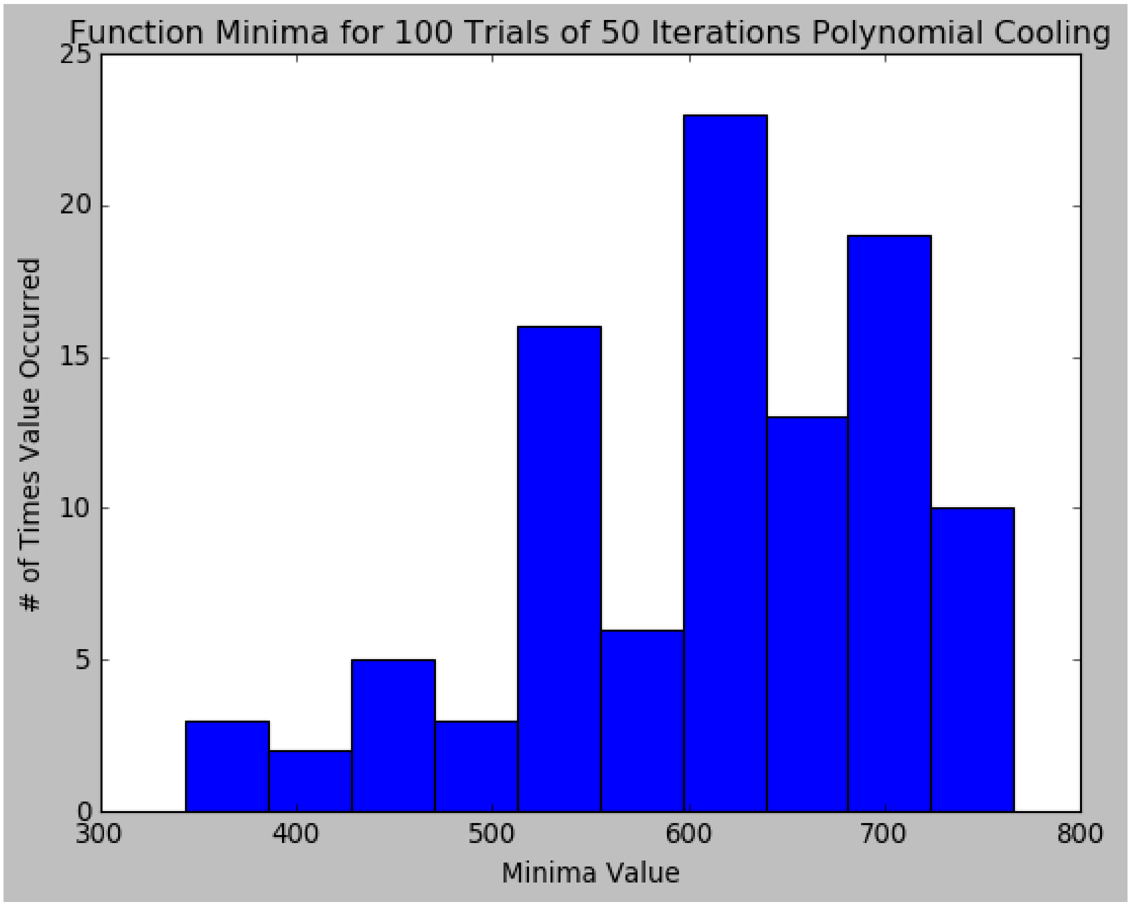
### 1.1.4 Question 3 Polynomial Cooling

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
```

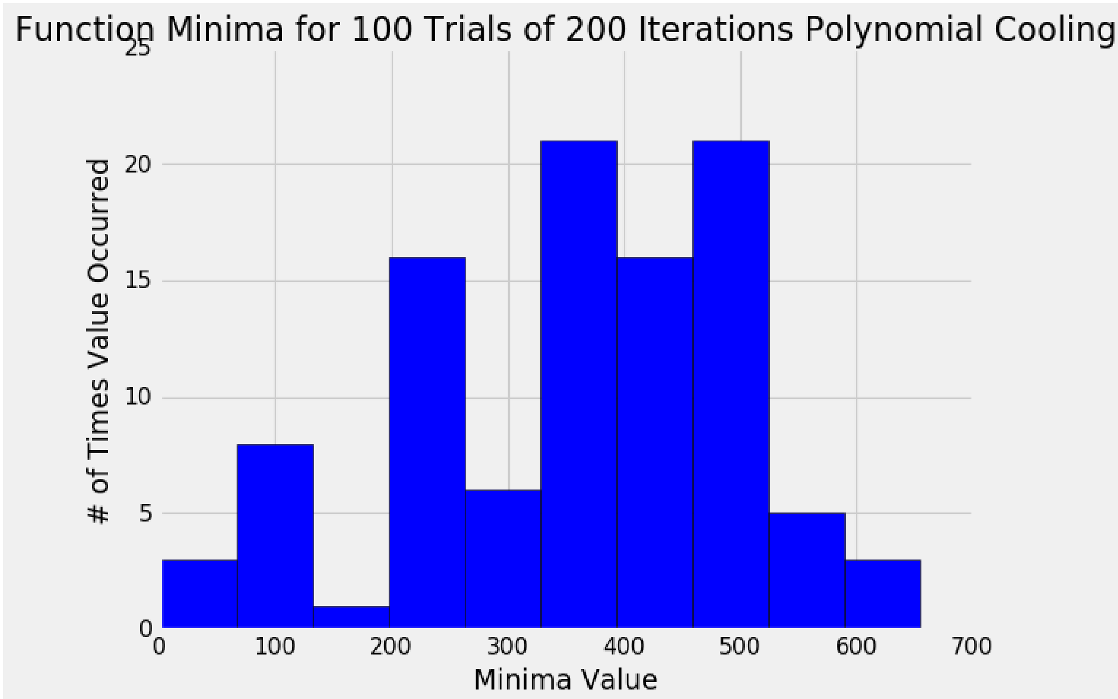
[2]:
```
def schwefel_2d(x,y):
    return 418.9829 * 2 - (x*np.sin(np.sqrt(np.abs(x))) + y * np.sin(np.sqrt(np.
 ↪abs(y))) )
```

[27]:
```
X0 = 0
Y0 = 0
T0 = 500
N = [50, 200, 1000,10000]
trials = 100

for n in range(0,len(N)):
    min_outputs = np.zeros(trials)
    min_x = np.zeros(trials)
    min_y = np.zeros(trials)
    for trial in range(0,trials):

        X = np.zeros(N[n])
        Y = np.zeros(N[n])
        xy_output = np.zeros(N[n])
        X[0] = 0
        Y[0] = 0
        T = T0
        xy_output[0] = schwefel_2d(X[0], Y[0])
        for i in range(1,N[n]):
            while 1:
                X_temp = X[i-1] + np.random.normal(0,25)
                Y_temp = Y[i-1] + np.random.normal(0,25)
                if ((np.abs(X_temp) < 500) and (np.abs(Y_temp) < 500) ):
                    break
            alpha = np.exp((schwefel_2d(X[i-1], Y[i-1]) - schwefel_2d(X_temp,␣
 ↪Y_temp))/T)

            if(schwefel_2d(X_temp, Y_temp) <= schwefel_2d(X[i-1], Y[i-1])):
                X[i] = X_temp
                Y[i] = Y_temp
                xy_output[i] = schwefel_2d(X_temp, Y_temp)
            elif (np.random.uniform() < alpha):
                X[i] = X_temp
                Y[i] = Y_temp
```

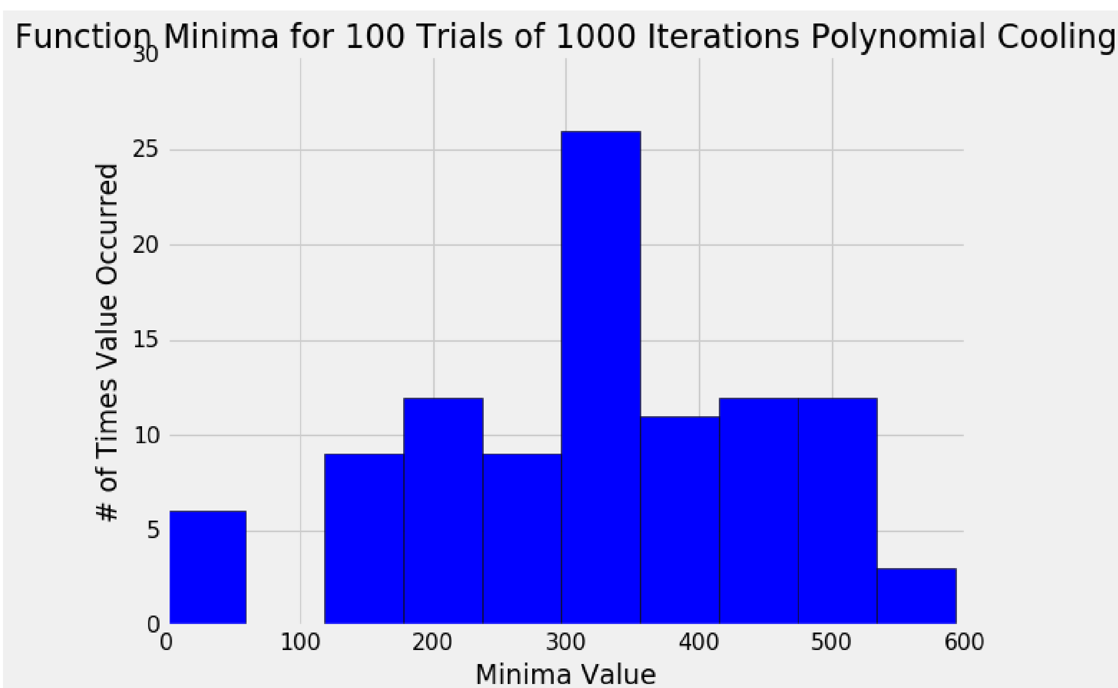
```
                xy_output[i] = schwefel_2d(X_temp, Y_temp)
            else:
                X[i] = X[i-1]
                Y[i] = Y[i-1]
                xy_output[i] = schwefel_2d( X[i-1], Y[i-1])
            T = T0/(.0001*i*i)
        min_outputs[trial] = min(xy_output)

        min_x[trial] = X[np.argmin(xy_output)]
        min_y[trial] = Y[np.argmin(xy_output)]
    #print(min_outputs)
    plt.hist(min_outputs, bins = 10, edgecolor = 'black', facecolor = 'blue' )
#plt.xticks(np.arange(70,120))
    plt.xlabel("Minima Value")
    plt.ylabel("# of Times Value Occurred")
    plt.title("Function Minima for 100 Trials of {number} Iterations Polynomial␣
 ↪Cooling".format(number=N[n]))
    plt.style.use('fivethirtyeight')
    plt.show()
    x_min_trials = min_x[np.argmin(min_outputs)]
    y_min_trials = min_y[np.argmin(min_outputs)]
    print("Minimum X,Y pair from 100 trials for {number} Iterations: ({X}, {Y})".
 ↪format(X = x_min_trials,
                Y = y_min_trials, number=N[n]))
    print("Function minima value for X,Y pair: ", min(min_outputs))
```



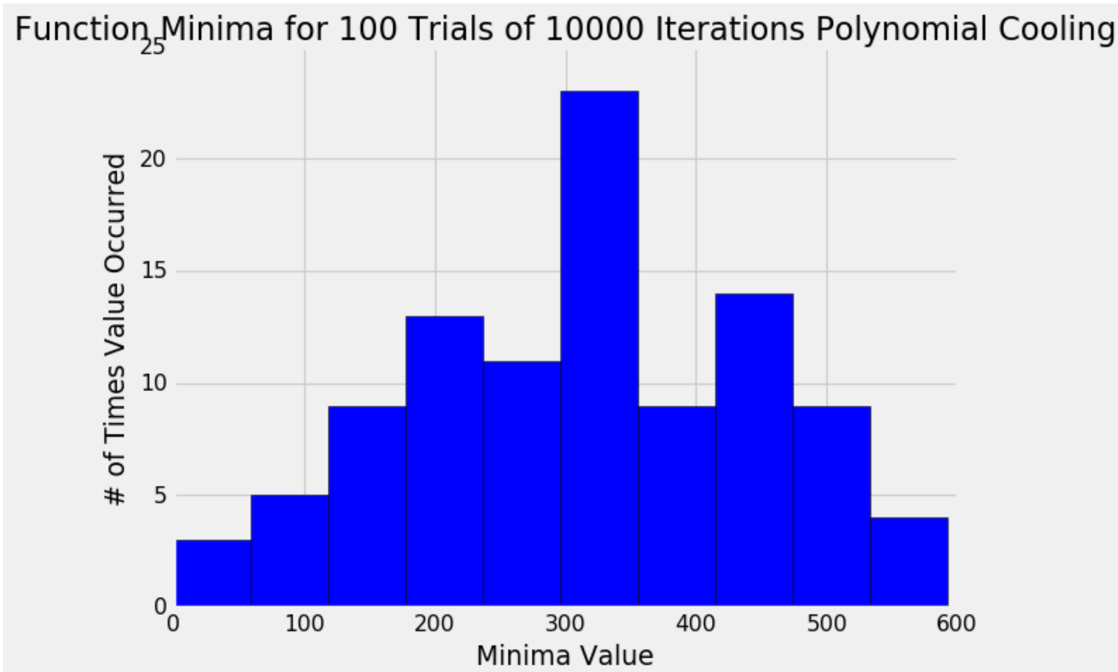

```
Minimum X,Y pair from 100 trials for 50 Iterations: (-297.6363483838687,
197.57759988184904)
Function minima value for X,Y pair:  343.50408774437153
```




```
Minimum X,Y pair from 100 trials for 200 Iterations: (422.44599194841527,
418.26210296529035)
Function minima value for X,Y pair:  1.1986228483795003
```



```
Minimum X,Y pair from 100 trials for 1000 Iterations: (420.5653945932271,
420.8139958217626)
Function minima value for X,Y pair:  0.023572859132514168
```

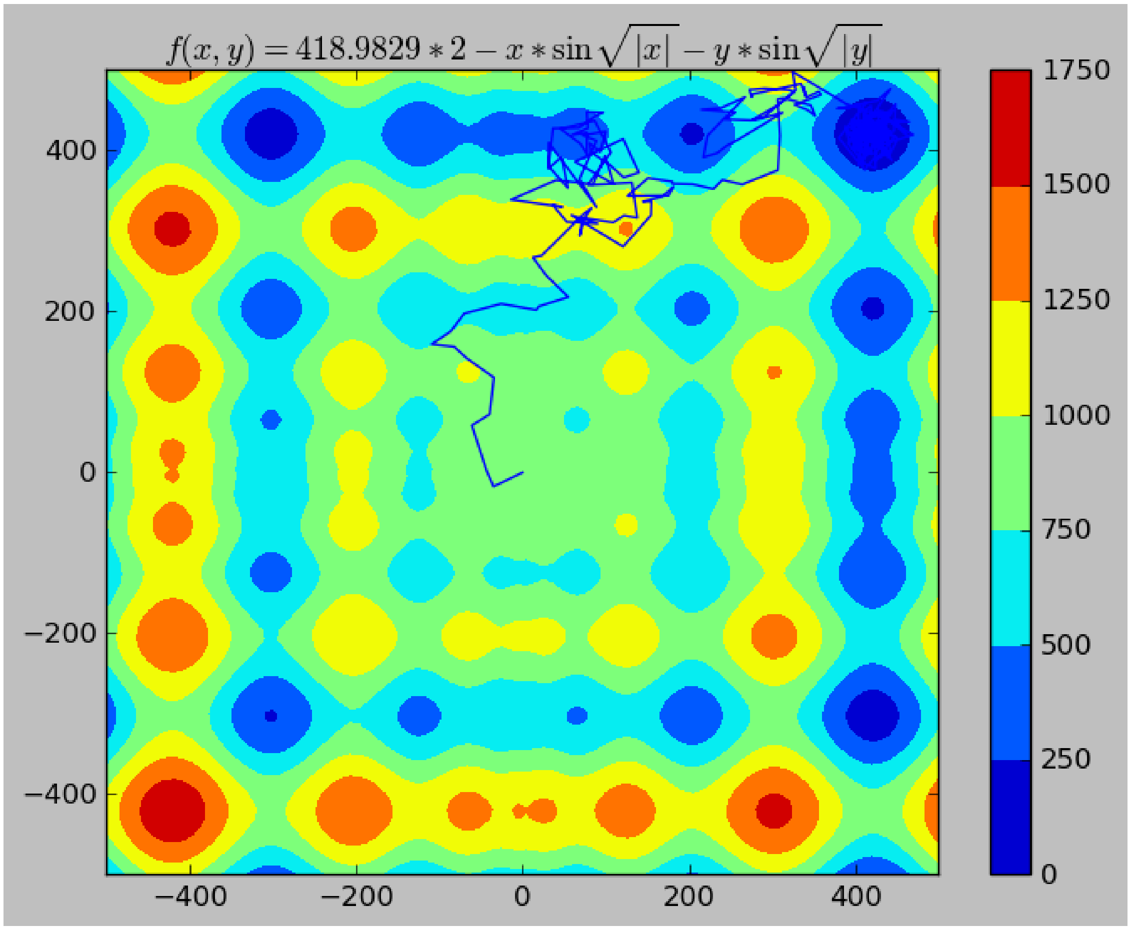



```
Minimum X,Y pair from 100 trials for 10000 Iterations: (420.83053851568417,
421.1122338663152)
Function minima value for X,Y pair:  0.0050336989070274285
```

[26]:

```
X = np.zeros(10000)
Y = np.zeros(10000)
xy_output = np.zeros(10000)
X[0] = 0
Y[0] = 0
T = T0
xy_output[0] = schwefel_2d(X[0], Y[0])
for i in range(1,10000):
    while 1:
        X_temp = X[i-1] + np.random.normal(0,25)
        Y_temp = Y[i-1] + np.random.normal(0,25)
        if ((np.abs(X_temp) < 500) and (np.abs(Y_temp) < 500) ):
            break
    alpha = np.exp((schwefel_2d(X[i-1], Y[i-1]) - schwefel_2d(X_temp, Y_temp))/T)

    if(schwefel_2d(X_temp, Y_temp) <= schwefel_2d(X[i-1], Y[i-1])):
```

```
        X[i] = X_temp
        Y[i] = Y_temp
        xy_output[i] = schwefel_2d(X_temp, Y_temp)
    elif (np.random.uniform() < alpha):
        X[i] = X_temp
        Y[i] = Y_temp
        xy_output[i] = schwefel_2d(X_temp, Y_temp)
    else:
        X[i] = X[i-1]
        Y[i] = Y[i-1]
        xy_output[i] = schwefel_2d( X[i-1], Y[i-1])
    T = T0/(.0001 *i*i)

N_r = 500
x = np.linspace(-N_r,N_r,100)
y = np.linspace(-N_r,N_r,100)
x, y = np.meshgrid(x, y)
Z = schwefel_2d(x,y)

plt.style.use('classic')
plt.figure(num=None,dpi=100)
plt.contourf(x,y,Z)
plt.title('$f(x,y) = 418.9829 * 2 -x*\sin \sqrt{|x|} - y *\sin \sqrt{|y|}$')
plt.plot(X,Y)
plt.colorbar()
plt.show()
```




**Question 3 (Polynomial Cooling) Analysis**

- The first 4 Histograms are for 100 trials at the 4 different iteration values: 50, 200, 1000, and 10000 using polynomial temperature cooling.
- As evident from the histograms, as the number of iterations increase, the number of trials with which the global minima is approached increases, and the best estimate of the 100 trials at that iteration count gets better and better.
- The countour map above is overlayed with the path of a trial that approaches the global minima. As one can tell, it jumps from local minima to local minima, eventually settling closer and closer to the global minima.
- Compared to logarithmic cooling and exponential cooling, polynomial cooling seems to jump between less local minima before settling on the global minima.
- Polynomial cooling also has less trials where the global minima is approached compared to logarithmic cooling or exponential.
- This could be due to the fact that the polynomial picked decays at rate that may be too quick.

```
[ ]:
```

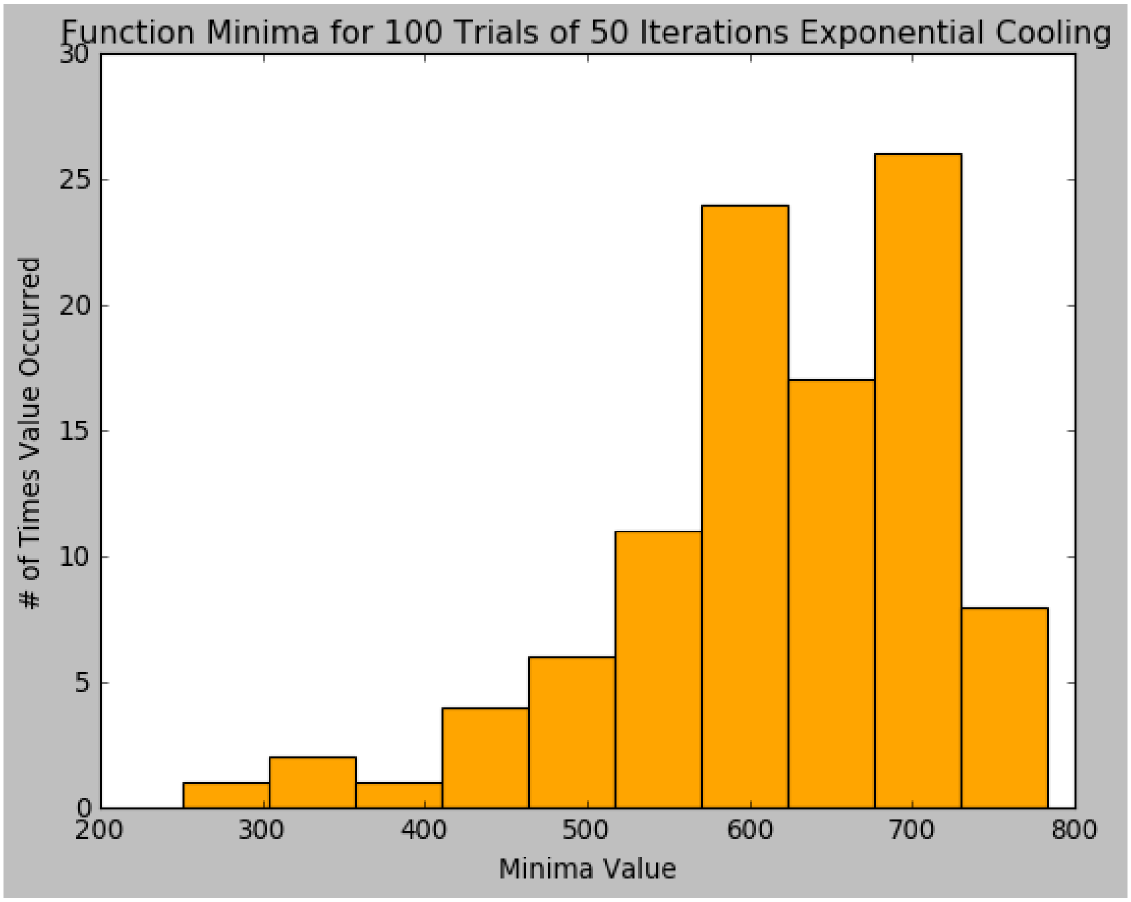
### 1.1.5 Question 3 Exponential Cooling

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
```

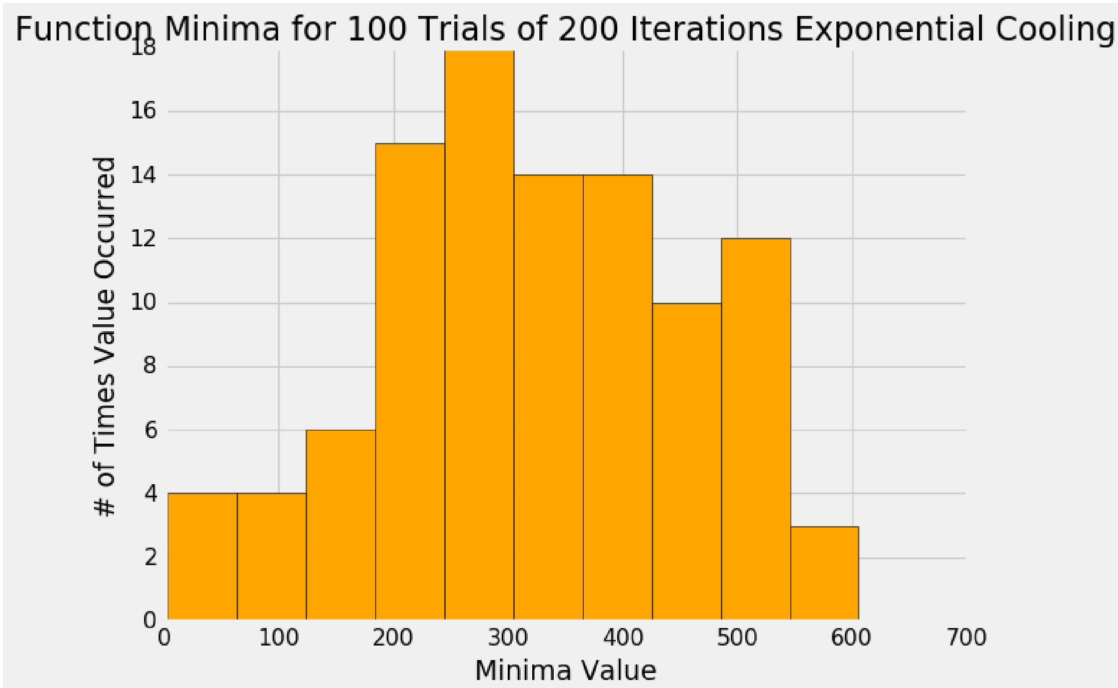
[2]:
```
def schwefel_2d(x,y):
    return 418.9829 * 2 - (x*np.sin(np.sqrt(np.abs(x))) + y * np.sin(np.sqrt(np.
 ↪abs(y))) )
```

[10]:
```
X0 = 0
Y0 = 0
T0 = 500
N = [50, 200, 1000,10000]
trials = 100

for n in range(0,len(N)):
    min_outputs = np.zeros(trials)
    min_x = np.zeros(trials)
    min_y = np.zeros(trials)
    for trial in range(0,trials):

        X = np.zeros(N[n])
        Y = np.zeros(N[n])
        xy_output = np.zeros(N[n])
        X[0] = 0
        Y[0] = 0
        T = T0
        xy_output[0] = schwefel_2d(X[0], Y[0])
        for i in range(1,N[n]):
            while 1:
                X_temp = X[i-1] + np.random.normal(0,25)
                Y_temp = Y[i-1] + np.random.normal(0,25)
                if ((np.abs(X_temp) < 500) and (np.abs(Y_temp) < 500) ):
                    break
            alpha = np.exp((schwefel_2d(X[i-1], Y[i-1]) - schwefel_2d(X_temp,␣
 ↪Y_temp))/T)

            if(schwefel_2d(X_temp, Y_temp) <= schwefel_2d(X[i-1], Y[i-1])):
                X[i] = X_temp
                Y[i] = Y_temp
                xy_output[i] = schwefel_2d(X_temp, Y_temp)
            elif (np.random.uniform() < alpha):
                X[i] = X_temp
                Y[i] = Y_temp
```

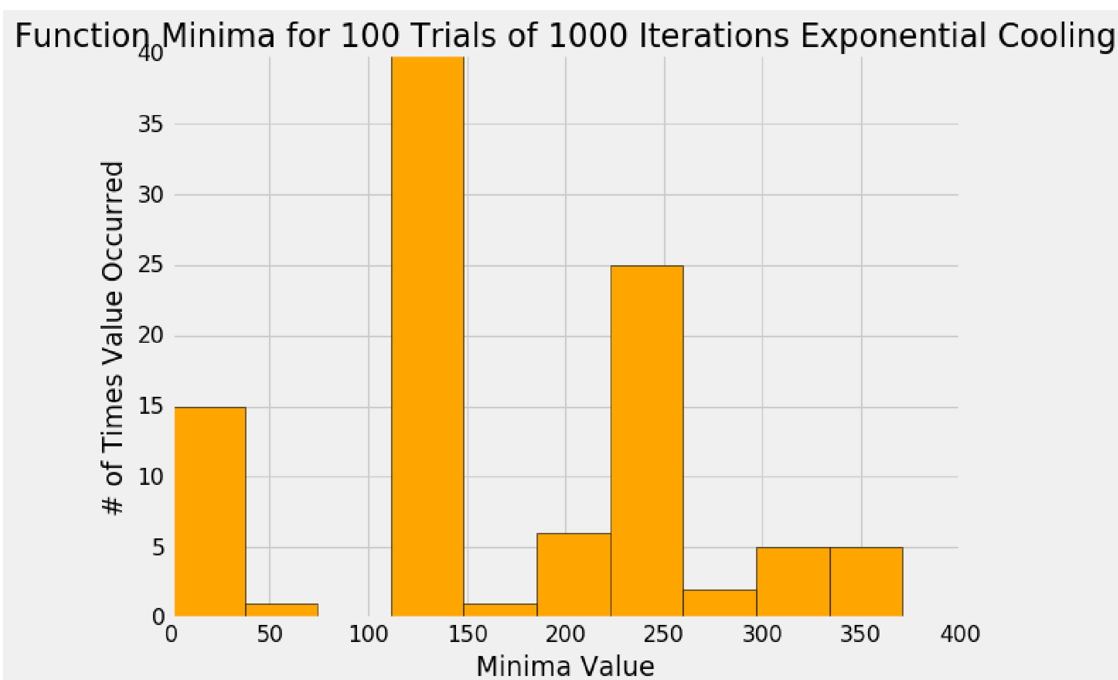
```
                xy_output[i] = schwefel_2d(X_temp, Y_temp)
            else:
                X[i] = X[i-1]
                Y[i] = Y[i-1]
                xy_output[i] = schwefel_2d( X[i-1], Y[i-1])
            T = T0/np.exp(.001*i)
        min_outputs[trial] = min(xy_output)

        min_x[trial] = X[np.argmin(xy_output)]
        min_y[trial] = Y[np.argmin(xy_output)]
    #print(min_outputs)
    plt.hist(min_outputs, bins = 10, edgecolor = 'black', facecolor = 'orange' )
#plt.xticks(np.arange(70,120))
    plt.xlabel("Minima Value")
    plt.ylabel("# of Times Value Occurred")
    plt.title("Function Minima for 100 Trials of {number} Iterations Exponential␣
 ↪Cooling".format(number=N[n]))
    plt.style.use('fivethirtyeight')
    plt.show()
    x_min_trials = min_x[np.argmin(min_outputs)]
    y_min_trials = min_y[np.argmin(min_outputs)]
    print("Minimum X,Y pair from 100 trials for {number} Iterations: ({X}, {Y})".
 ↪format(X = x_min_trials,
                Y = y_min_trials, number=N[n]))
    print("Function minima value for X,Y pair: ", min(min_outputs))
```



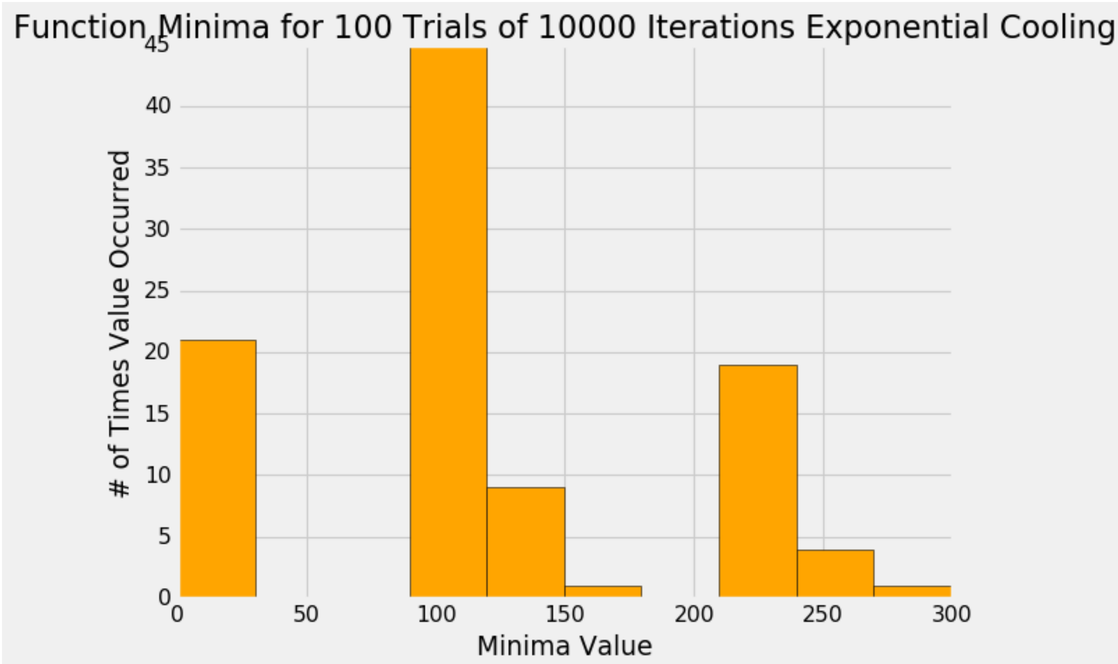

```
Minimum X,Y pair from 100 trials for 50 Iterations: (-291.90996124627645,
-300.71293913544986)
Function minima value for X,Y pair:  251.3681112237325
```




```
Minimum X,Y pair from 100 trials for 200 Iterations: (422.79905399852424,
425.7369830452089)
Function minima value for X,Y pair:  3.2938814891480206
```



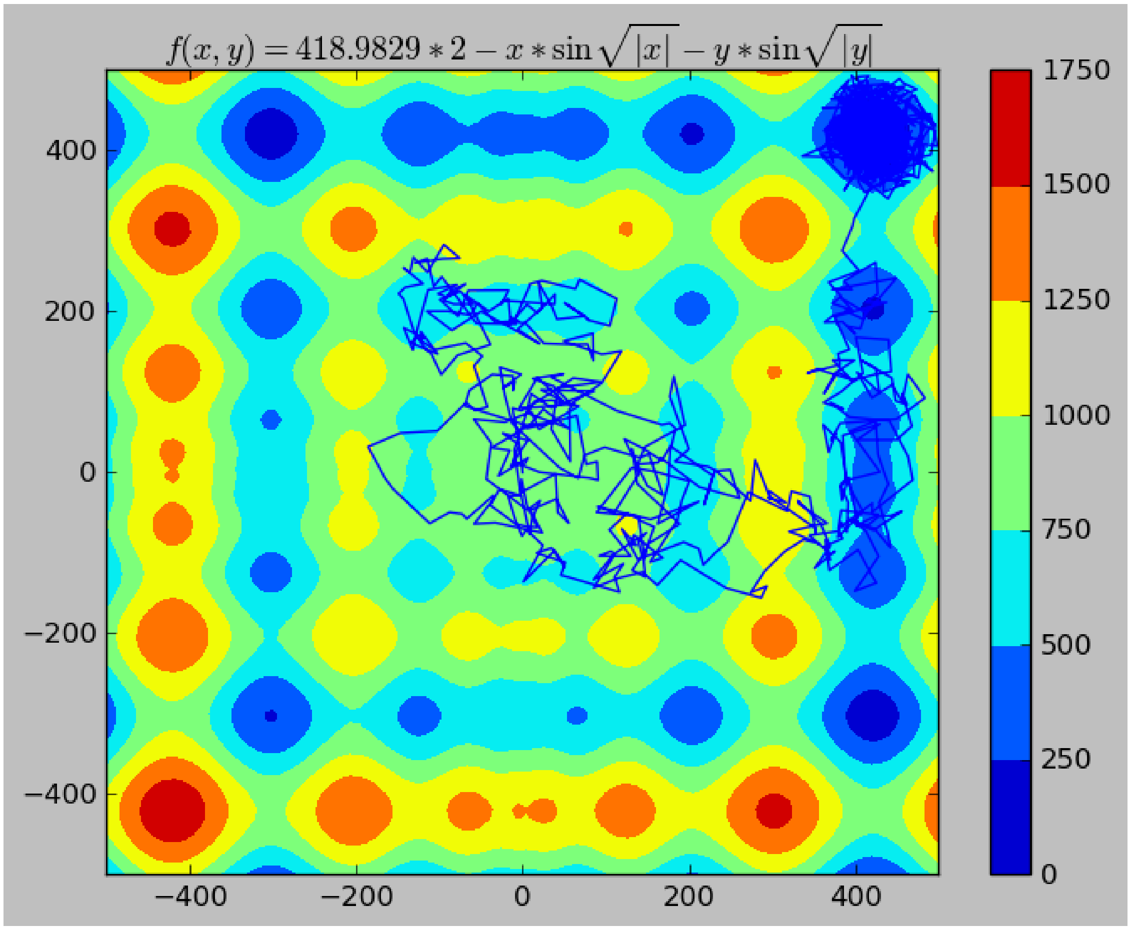
```
Minimum X,Y pair from 100 trials for 1000 Iterations: (421.51973151656676,
421.44387743663594)
Function minima value for X,Y pair:  0.06683113280178077
```




```
Minimum X,Y pair from 100 trials for 10000 Iterations: (420.96092163228565,
421.0485183728245)
Function minima value for X,Y pair:  0.0008361835002688167
```

[9]:
```
X = np.zeros(10000)
Y = np.zeros(10000)
xy_output = np.zeros(10000)
X[0] = 0
Y[0] = 0
T = T0
xy_output[0] = schwefel_2d(X[0], Y[0])
for i in range(1,10000):
    while 1:
        X_temp = X[i-1] + np.random.normal(0,25)
        Y_temp = Y[i-1] + np.random.normal(0,25)
        if ((np.abs(X_temp) < 500) and (np.abs(Y_temp) < 500) ):
            break
    alpha = np.exp((schwefel_2d(X[i-1], Y[i-1]) - schwefel_2d(X_temp, Y_temp))/T)

    if(schwefel_2d(X_temp, Y_temp) <= schwefel_2d(X[i-1], Y[i-1])):
        X[i] = X_temp
        Y[i] = Y_temp
```

```
        xy_output[i] = schwefel_2d(X_temp, Y_temp)
    elif (np.random.uniform() < alpha):
        X[i] = X_temp
        Y[i] = Y_temp
        xy_output[i] = schwefel_2d(X_temp, Y_temp)
    else:
        X[i] = X[i-1]
        Y[i] = Y[i-1]
        xy_output[i] = schwefel_2d( X[i-1], Y[i-1])
    T = T0/np.exp(.001*i)

N_r = 500
x = np.linspace(-N_r,N_r,100)
y = np.linspace(-N_r,N_r,100)
x, y = np.meshgrid(x, y)
Z = schwefel_2d(x,y)

plt.style.use('classic')
plt.figure(num=None,dpi=100)
plt.contourf(x,y,Z)
plt.title('$f(x,y) = 418.9829 * 2 -x*\sin \sqrt{|x|} - y *\sin \sqrt{|y|}$')
plt.plot(X,Y)
plt.colorbar()
plt.show()
```




**Question 3 (Exponential Cooling) Analysis**

- The first 4 Histograms are for 100 trials at the 4 different iteration values: 50, 200, 1000, and 10000 using exponential temperature cooling.
- As evident from the histograms, as the number of iterations increase, the number of trials with which the global minima is approached increases, and the best estimate of the 100 trials at that iteration count gets better and better.
- The countour map above is overlayed with the path of a trial that approaches the global minima. As one can tell, it jumps from local minima to local minima, eventually settling closer and closer to the global minima.
- Compared to logarithmic cooling, exponential cooling seems to jump between more local minima before settling on the global minima.
- Exponential cooling also has more trials where the global minima is approached compared to logarithmic cooling.
- Exponential cooling also descends at a quicker rate which dould explain why this seems to be the case, so the alpha values shrink faster and faster.

[ ]:

### 1.1.6 Question 4

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import pairwise_distances
from sklearn.utils.random import sample_without_replacement
from scipy.spatial import distance
```

[2]:
```
data_set_path_1 = "../data/uscap_name.txt"
data_set_path_2 = "../data/uscap_xy.txt"
data1 = pd.read_csv(data_set_path_1, sep=",", header=None)
data1.columns = ["City", "State"]
data2 = pd.read_csv(data_set_path_2, sep = "\s+", header=None)
data2.columns = ["X-Coordinate", "Y-Coordinate"]
data2
frames = [data1,data2]
data = pd.concat(frames, sort = 'False', axis = 1)
data = data.drop(data.index[1])
data = data.drop(data.index[9])
data = data.reset_index(drop=True)
data

b, c = data.iloc[0], data.iloc[3]


temp = data.iloc[0].copy()
data.iloc[0] = c
data.iloc[3] = temp
data  #this swap makes sacramento first location
```

[2]:

| | City | State | X-Coordinate | Y-Coordinate |
|---|---|---|---|---|
| 0 | Sacramento | California | -8392.976246 | 2664.025176 |
| 1 | Phoenix | Arizona | -7743.816805 | 2311.143387 |
| 2 | Little Rock | Arkansas | -6379.680295 | 2400.107649 |
| 3 | Montgomery | Alabama | -5961.513053 | 2236.041996 |
| 4 | Denver | Colorado | -7253.950857 | 2745.804159 |
| 5 | Hartford | Connecticut | -5021.665662 | 2885.918649 |
| 6 | Dover | Delaware | -5218.571379 | 2705.918983 |
| 7 | Tallahassee | Florida | -5822.883103 | 2104.087378 |
| 8 | Atlanta | Georgia | -5830.983188 | 2332.669658 |
| 9 | Boise | Idaho | -8031.517820 | 3013.520309 |
| 10 | Springfield | Illinois | -6194.452160 | 2748.850124 |
| 11 | Indianapolis | Indiana | -5952.431603 | 2749.381607 |
| 12 | Des Moines | Iowa | -6468.796015 | 2873.753598 |
| 13 | Topeka | Kansas | -6611.764205 | 2697.494770 |
| 14 | Frankfort | Kentucky | -5863.673038 | 2639.266057 |

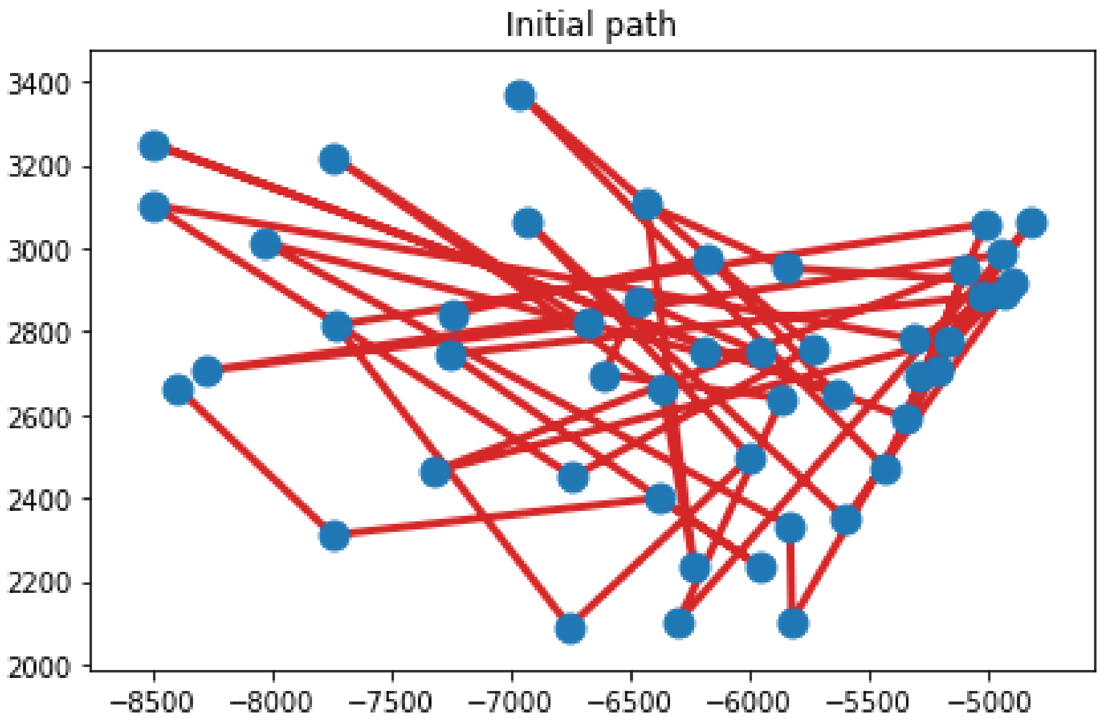
```
15      Baton Rouge        Louisiana  -6297.394751   2104.521990
16          Augusta            Maine  -4820.477118   3062.564136
17        Annapolis         Maryland  -5285.898333   2692.861561
18           Boston    Massachusetts  -4907.692362   2918.269240
19          Lansing         Michigan  -5841.810479   2952.699610
20       Saint Paul        Minnesota  -6432.391858   3105.850152
21          Jackson      Mississippi  -6232.912673   2233.171900
22   Jefferson City         Missouri  -6369.879835   2665.223916
23           Helena          Montana  -7740.582230   3219.568143
24          Lincoln         Nebraska  -6679.847274   2819.784977
25      Carson City           Nevada  -8274.473795   2705.851822
26          Concord    New Hampshire  -4943.734526   2986.321077
27          Trenton       New Jersey  -5165.325085   2779.147951
28         Santa Fe       New Mexico  -7321.692800   2464.451053
29           Albany         New York  -5097.970699   2947.609263
30          Raleigh   North Carolina  -5433.544922   2471.621041
31         Bismarck     North Dakota  -6963.392322   3372.790406
32         Columbus             Ohio  -5734.984919   2761.217902
33    Oklahoma City         Oklahoma  -6739.245293   2451.673744
34            Salem           Oregon  -8500.781583   3104.544866
35       Harrisburg     Pennsylvania  -5311.771620   2782.467859
36       Providence     Rhode Island  -4934.959722   2889.826009
37         Columbia   South Carolina  -5599.167231   2349.252618
38           Pierre     South Dakota  -6932.808784   3065.634125
39        Nashville        Tennessee  -5996.398211   2498.844733
40           Austin            Texas  -6754.101276   2091.295490
41   Salt Lake City             Utah  -7731.295151   2815.973108
42       Montpelier          Vermont  -5014.406471   3058.615664
43         Richmond         Virginia  -5352.150228   2593.851273
44          Olympia       Washington  -8491.378907   3250.427165
45       Charleston    West Virginia  -5640.506753   2649.784006
46          Madison        Wisconsin  -6176.077619   2976.276571
47         Cheyenne          Wyoming  -7241.366809   2842.979010
```

[3]:
```
# initial path

coordinates = data.iloc[:,[2,3]].values
num_cities = 48

# end of random coordinate generation
# --------------------------------------------------

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

#coordinates
```

[ ]:

[4]:
```
# Parameters
p_len = 0
iterations = 20000 # number of iterations
c = 1
# a = 0.5
p = np.arange(0,num_cities) # Initial path p
for a1 in range(0,num_cities-1):
    p_len = p_len + distance.euclidean(coordinates[a1],coordinates[a1+1])
print('Initial path length:',str(p_len))

# Save the paths and lengths
pathHistory = np.zeros((iterations,num_cities))
lenHistory = []
thresh_ar = []

# plot cities and initial path
plt.figure()
x_coord = coordinates[:,0]
y_coord = coordinates[:,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()
```

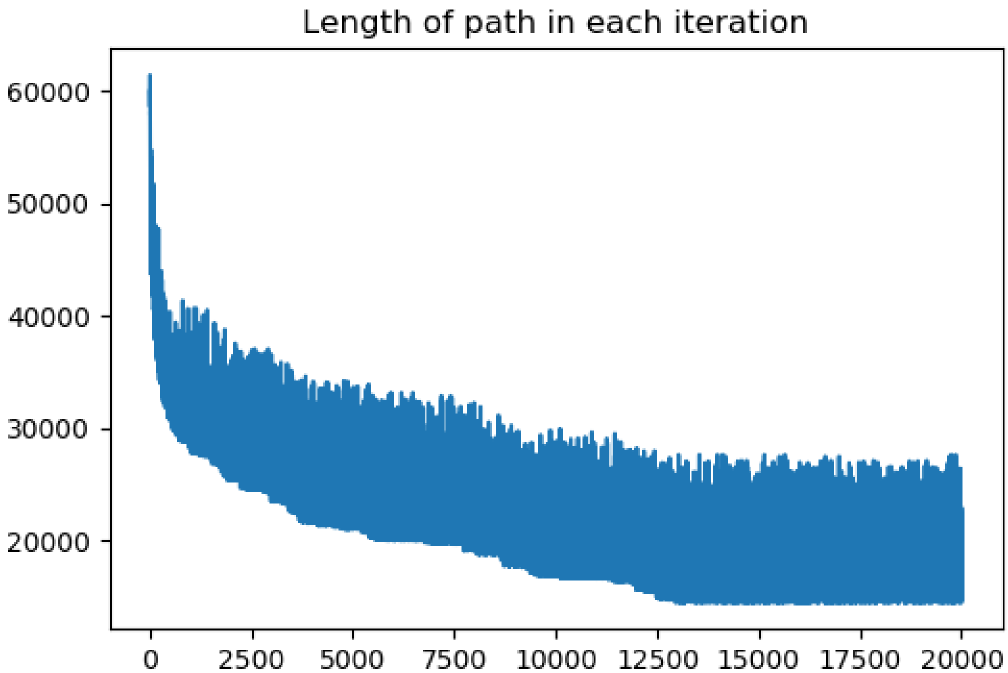
```
Initial path length: 59050.047779598455
```

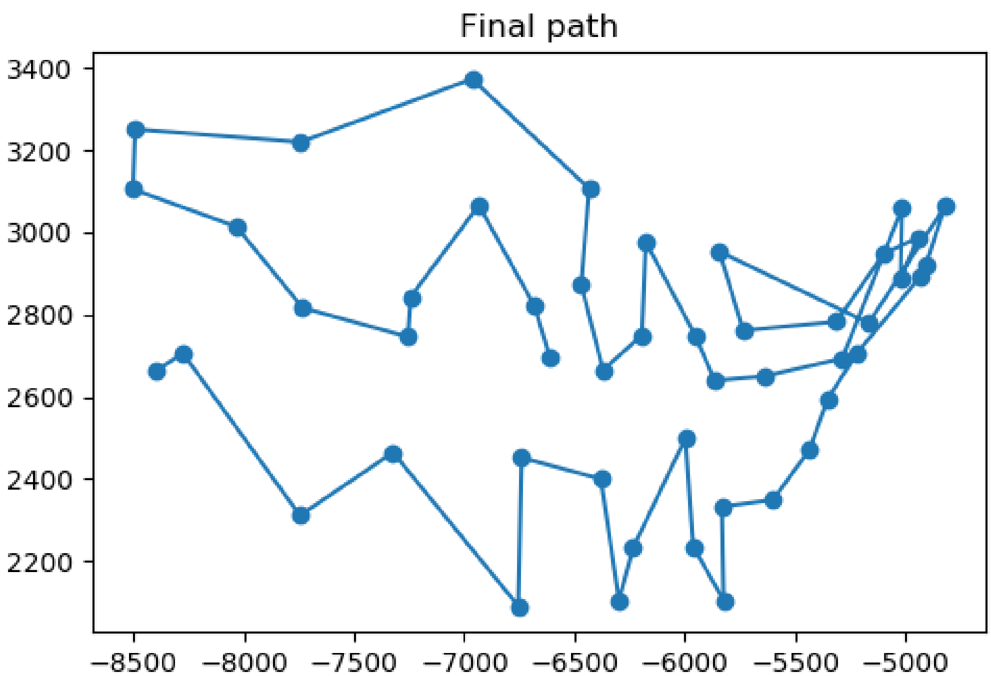



[5]:
```python
iter_count = 0;
p2 = []
while iter_count < iterations:
    iter_count = iter_count + 1;
    # Create path p2 by randomly swap two cities
    # index of two cities for the new path
    swap_i, swap_j = np.random.choice(num_cities, 2)
    p2 = np.copy(p)
    # swap the two cities of the path
    p2[swap_i], p2[swap_j] = p2[swap_j], p2[swap_i]

    # new path length
    p_len2 = 0

    for a1 in range(0,num_cities-1):
        p_len2 = p_len2 + distance.
↪euclidean(coordinates[p2[a1]],coordinates[p2[a1+1]])

    thresh = np.power((1+iter_count),((p_len - p_len2)/c))
#     altearnative formula for q
#     thresh = np.exp((p_len - p_len2)/(c*np.power(a,iter_count)))

    # 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)

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

plt.figure(num=None,dpi=100)
plt.plot(lenHistory)
plt.title('Length of path in each iteration')
plt.show()

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

```
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:18: RuntimeWarning:
overflow encountered in power
```


Length of path in each iteration
60000
50000
40000
30000
20000
0
2500
5000
7500
10000
12500
15000
17500
20000


Final path
3400
3200
3000
2800
2600
2400
2200
−8500
−8000
−7500
−7000
−6500
−6000
−5500
−5000

[ ]:

[6]:
```
cities_order = []
for i in range(0,len(x_coord_f)):
    cities_order.append(data.loc[data['X-Coordinate'] == x_coord_f[i] , 'City'].
 ↪iloc[0])
print(cities_order)
```

```
['Sacramento', 'Carson City', 'Phoenix', 'Santa Fe', 'Austin', 'Oklahoma City',
'Little Rock', 'Baton Rouge', 'Jackson', 'Nashville', 'Montgomery',
'Tallahassee', 'Atlanta', 'Columbia', 'Raleigh', 'Richmond', 'Dover',
'Providence', 'Boston', 'Augusta', 'Trenton', 'Lansing', 'Columbus',
'Harrisburg', 'Albany', 'Concord', 'Hartford', 'Montpelier', 'Annapolis',
'Charleston', 'Frankfort', 'Indianapolis', 'Madison', 'Springfield', 'Jefferson
City', 'Des Moines', 'Saint Paul', 'Bismarck', 'Helena', 'Olympia', 'Salem',
'Boise', 'Salt Lake City', 'Denver', 'Cheyenne', 'Pierre', 'Lincoln', 'Topeka']
```

### 1.1.7 Question 4 Analysis

- The array list above marks the optimal path calculated when starting from Sacramento.
- The graph above it lists all the coordinates and the direction of the path taken from Sacramento to its endpoint in Topeka, Kansas.
- Look at the graph that indicates the length of the path at each iteration, at around 15000 iterations the length of the path begins to flatten and converge to the optimal answer. The simulation shown here is at 20000 iterations.
- The randomized initial path length begins at value of around 60000, and through the iterations begins to drop until it hovers at around 30000.

[ ]:
```
```