Design and Analysis of Algorithms (SOFE 3770U)
Final Project

Comparative Study

Differential Evolution vs. Particle Swarm Optimization

Submission Date: 05/12/2016

Karan Chandwaney (100472699)
Mohammed Maaz Ahmed (100522349)
Saisudan Bashkaran (100554275)

Table of Contents

Plots of 3D Benchmark Functions 3

Functions Definition – code 8

Particle Swarm Optimization – algorithm code 13

Differential Evolution – algorithm code 17

Performance Plots – Particle Swarm Optimization 20

- D = 10 20
- D = 30 23
- D = 50 26

Performance Plots – Differential Evolution 29

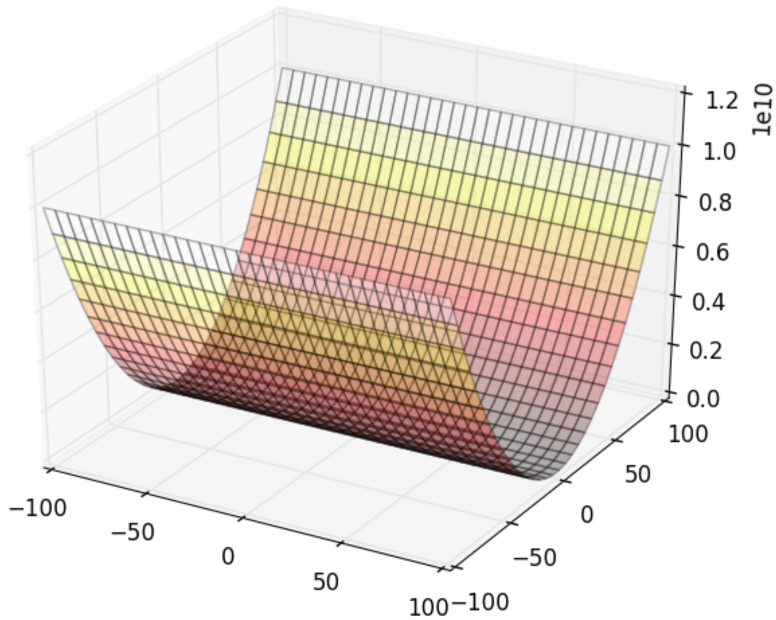
- D = 10 29
- D = 30 32
- D = 50 35

Comparison Table for D = 30 39

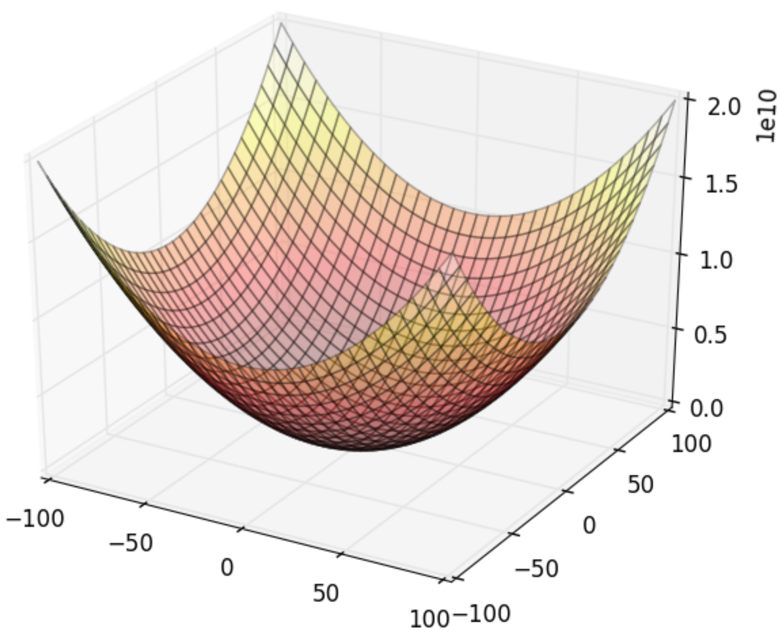
Conclusion 40

Plots of 3D Benchmark Functions

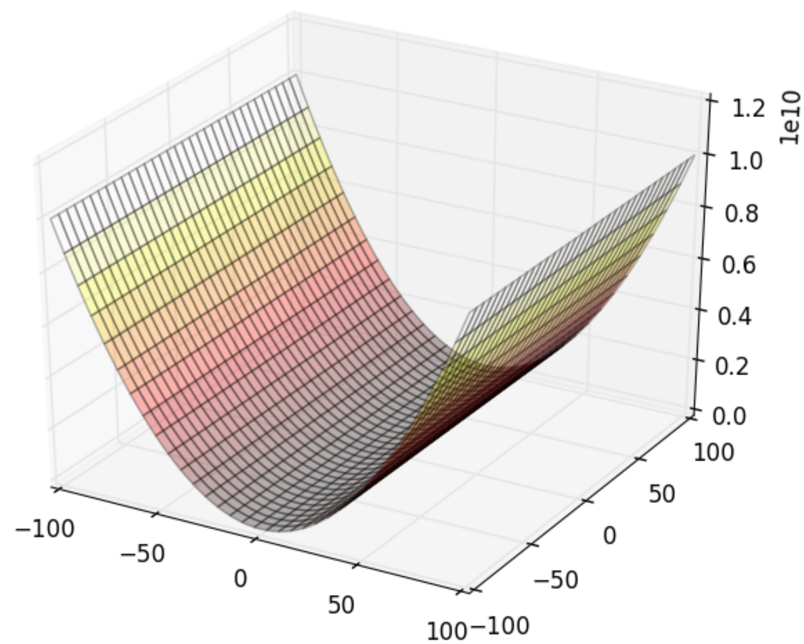
1. High Conditioned Elliptic Function:



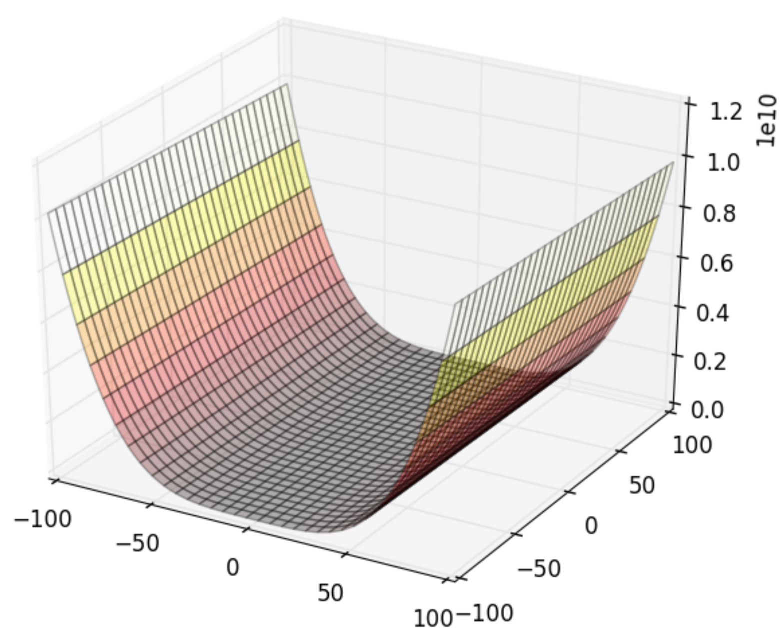
2. Bent Cigar Function:



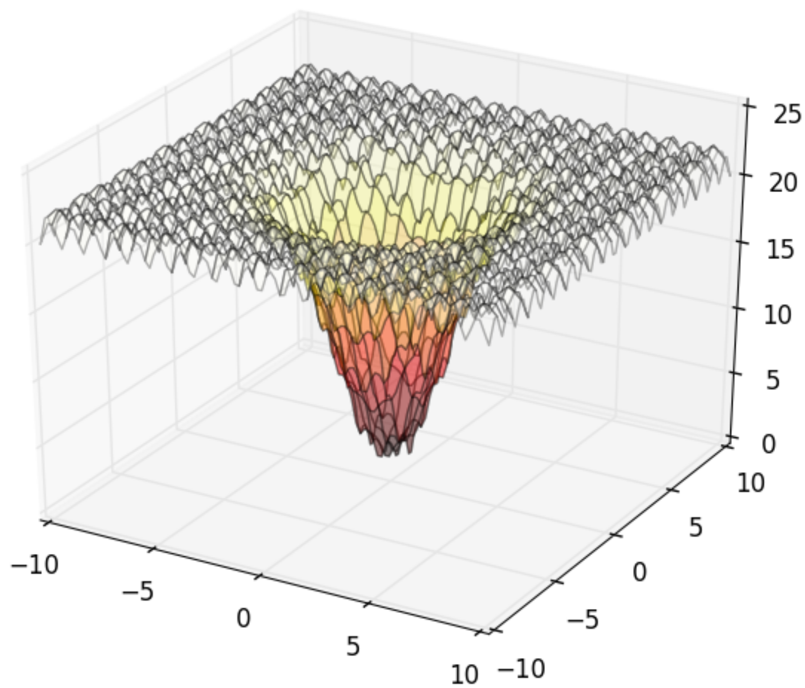
3. Discus Function:



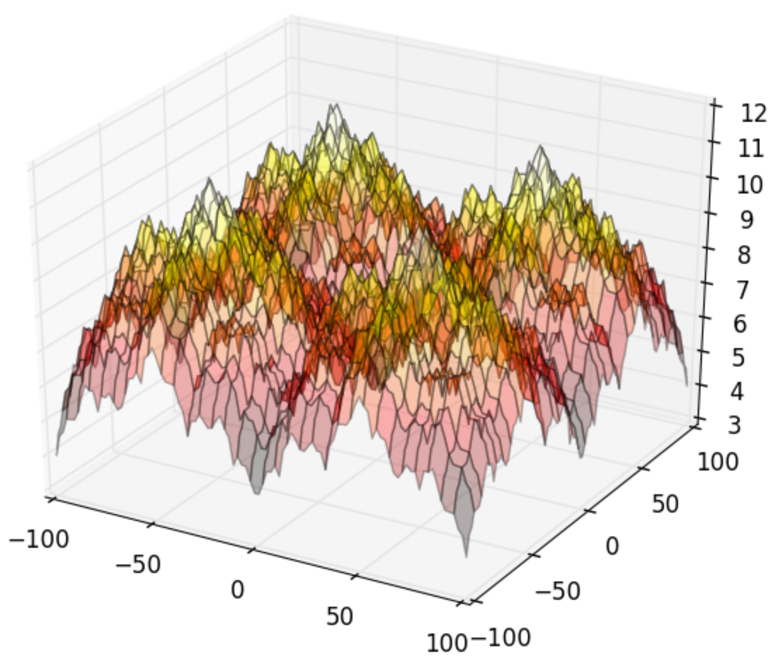
4. Rosenbrock's Function:



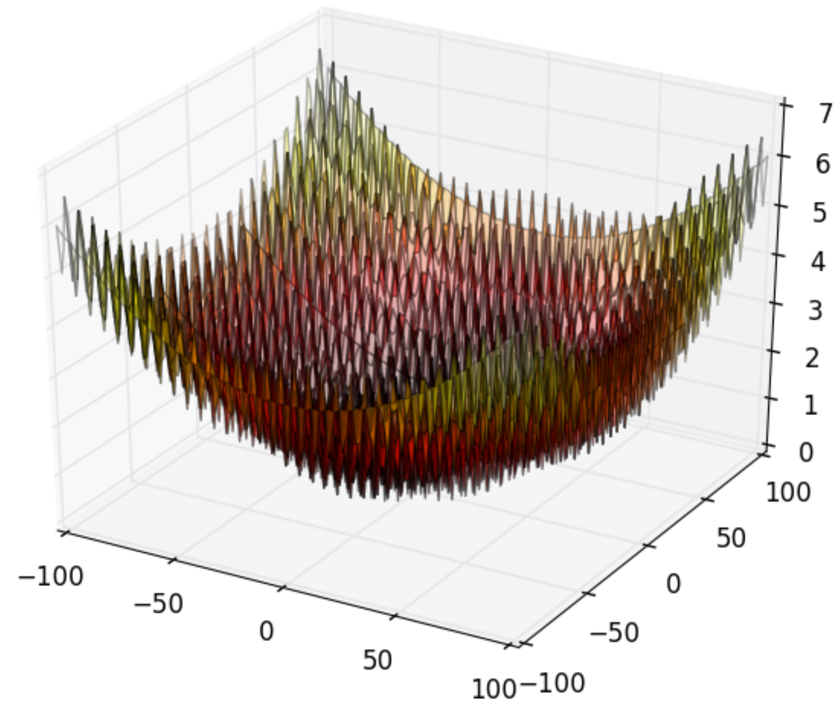
5. Ackley's Function:



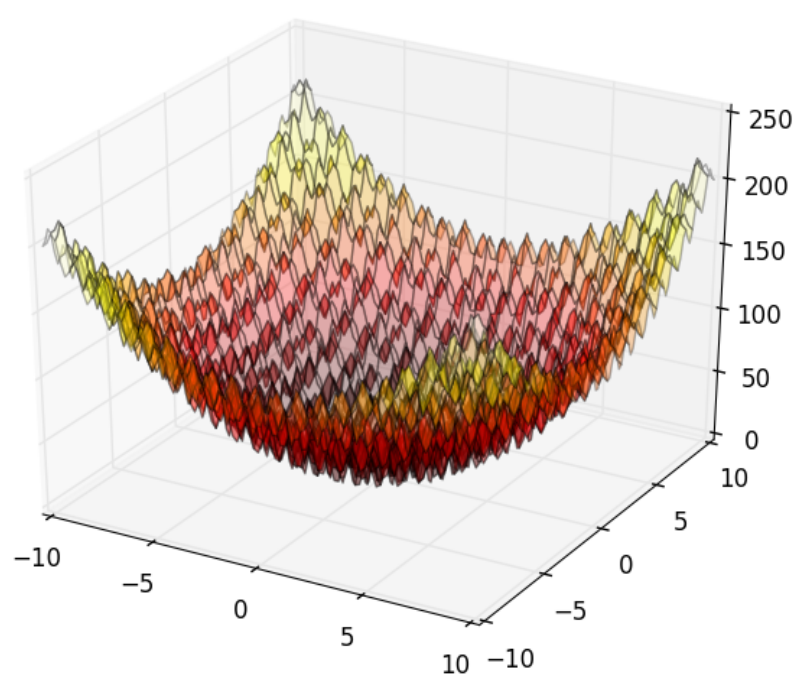
6. Weierstrass Function:



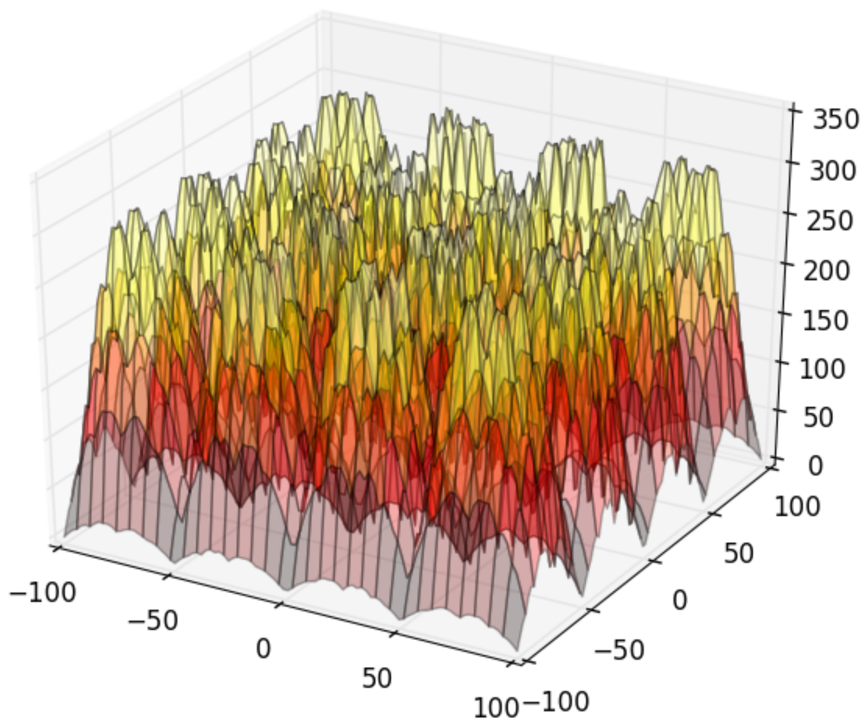
7. Griewank's Function:



8. Rastrigin's Function:



9. Katsuura Function:



Functions Definition – code

Written in Python 2

functions.py

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import matplotlib.pyplot as plt
import numpy as np

'''Define the 9 Benchmark functions and plot them'''

def f1(x):
    """High Conditioned Elliptic Function"""
    sum = 0.0
    for i in range(1, len(x)+1):
        sum += (10**6)**((i-1) / (len(x) - 1)) * x[i-1]**2
    return sum

#High conditioned elliptic
X = np.linspace(-100, 100, 100)
Y = np.linspace(-100, 100, 100)
X, Y = np.meshgrid(X, Y)                # create meshgrid
Z = f1([X, Y])                          # Calculate Z

# Plot the 3D surface for first function from project
fig = plt.figure()
ax = fig.gca(projection='3d')          # set the 3d axes
ax.plot_surface(X, Y, Z,
                rstride=3,
                cstride=3,
                alpha=0.3,
                cmap='hot')
plt.show()

def f2(x):
    """Bent Cigar Function"""
    sum = 0.0
    sum += x[0]**2
    for i in range(2, len(x)+1):
        sum += x[i-1]**2
    sum *= (10**6)
    return sum

#Bent cigar
X = np.linspace(-100, 100, 100)
Y = np.linspace(-100, 100, 100)
X, Y = np.meshgrid(X, Y)                # create meshgrid
Z = f2([X, Y])                          # Calculate Z

# Plot the 3D surface for first function from project
```

```
fig = plt.figure()
ax = fig.gca(projection='3d')          # set the 3d axes
ax.plot_surface(X, Y, Z,
                rstride=3,
                cstride=3,
                alpha=0.3,
                cmap='hot')
plt.show()

def f3(x):
    """Discus Function"""
    sum = 0.0
    sum += (x[0]**2)*(10**6)
    for i in range(2, len(x)+1):
        sum += x[i-1]**2
    return sum

#Discus
X = np.linspace(-100, 100, 100)
Y = np.linspace(-100, 100, 100)
X, Y = np.meshgrid(X, Y)                # create meshgrid
Z = f3([X, Y])                          # Calculate Z

# Plot the 3D surface for first function from project
fig = plt.figure()
ax = fig.gca(projection='3d')          # set the 3d axes
ax.plot_surface(X, Y, Z,
                rstride=3,
                cstride=3,
                alpha=0.3,
                cmap='hot')
plt.show()

def f4(x):
   """Rosenbrock's Function"""
   sum = 0.0
   for i in range(len(x)-1):
      sum += 100*((x[i]**2)-x[i+1])**2+\
      (1-x[i])**2
   return sum

#Rosenbrock's
X = np.linspace(-100, 100, 100)
Y = np.linspace(-100, 100, 100)
X, Y = np.meshgrid(X, Y)                # create meshgrid
Z = f4([X, Y])                          # Calculate Z

# Plot the 3D surface for first function from project
fig = plt.figure()
ax = fig.gca(projection='3d')          # set the 3d axes
ax.plot_surface(X, Y, Z,
```

```
                rstride=3,
                cstride=3,
                alpha=0.3,
                cmap='hot')
plt.show()

def f5(x):
    """Ackley's Function"""
    sum1, sum2 = 0.0, 0.0
    for i in range(0, len(x)):
        sum1 += x[i]**2
    sum1 = sum1 / float(len(x))
    for i in range(0, len(x)):
        sum2 += np.cos(2*np.pi*x[i])
    sum2 = sum2 / float(len(x))

    exp1 = -20.0 * (np.e ** (-0.2 * sum1))
    exp2 = np.e ** sum2

    sum = exp1 - exp2 + 20 + np.e
    return sum

#Ackley's
X = np.linspace(-10, 10, 100)
Y = np.linspace(-10, 10, 100)
X, Y = np.meshgrid(X, Y)               # create meshgrid
Z = f5([X, Y])                         # Calculate Z

# Plot the 3D surface for first function from project
fig = plt.figure()
ax = fig.gca(projection='3d')         # set the 3d axes
ax.plot_surface(X, Y, Z,
                rstride=3,
                cstride=3,
                alpha=0.3,
                cmap='hot')
plt.show()

def f6(x):
    """Weierstrass Function"""
    totalSum, sum1, sum2 = 0.0, 0.0, 0.0
    a = 0.5
    b = 3
    kmax = 20
    for i in range(len(x)):
        for k in range(0, kmax):
            sum1 += (a ** k) * np.cos(2 * np.pi * (b ** k) * (x[i] + 0.5))
            sum2 += (a ** k) * np.cos(2 * np.pi * (b ** k) * 0.5)
    totalSum += sum1 - (len(x) * sum2)
    return totalSum
```

```
#Weierstrass
X = np.linspace(-100, 100, 100)
Y = np.linspace(-100, 100, 100)
X, Y = np.meshgrid(X, Y)                # create meshgrid
Z = f6([X, Y])                          # Calculate Z

# Plot the 3D surface for first function from project
fig = plt.figure()
ax = fig.gca(projection='3d')          # set the 3d axes
ax.plot_surface(X, Y, Z,
                rstride=3,
                cstride=3,
                alpha=0.3,
                cmap='hot')
plt.show()

def f7(x):
    """Griewank's function"""
    sum = 0
    for i in x:
        sum += i * i
    product = 1
    for j in xrange(len(x)):
        product *= np.cos(x[j] / np.sqrt(j + 1))
    return 1 + sum / 4000 - product

#Griewank's
X = np.linspace(-100, 100, 100)
Y = np.linspace(-100, 100, 100)
X, Y = np.meshgrid(X, Y)                # create meshgrid
Z = f7([X, Y])                          # Calculate Z

# Plot the 3D surface for first function from project
fig = plt.figure()
ax = fig.gca(projection='3d')          # set the 3d axes
ax.plot_surface(X, Y, Z,
                rstride=3,
                cstride=3,
                alpha=0.3,
                cmap='hot')
plt.show()

def f8(x):
    """Rastrigin's Function"""
    sum = 0.0
    for i in range(0, len(x)):
        sum += (x[i]**2 - 10 * np.cos(2*np.pi*x[i]) + 10)
    return sum

#Rastrigin's
X = np.linspace(-10, 10, 100)
```

```
Y = np.linspace(-10, 10, 100)
X, Y = np.meshgrid(X, Y)                # create meshgrid
Z = f8([X, Y])                          # Calculate Z

# Plot the 3D surface for first function from project
fig = plt.figure()
ax = fig.gca(projection='3d')          # set the 3d axes
ax.plot_surface(X, Y, Z,
                rstride=3,
                cstride=3,
                alpha=0.3,
                cmap='hot')
plt.show()

def f9(x):
    """Katsuura Function"""
    product = 1
    for i in range(0, len(x)):
        sum = 0
        for j in range(1,33):
            term = np.power(2,j) * x[i]
            sum += np.abs(term - np.round(term))/(np.power(2,j))
        product *= np.power(1 + ((i + 1) * sum), 10.0 / np.power(len(x),
1.2))
    return (10 / len(x) * len(x) * product - (10 / len(x) * len(x)))

#Katsuura
X = np.linspace(-100, 100, 100)
Y = np.linspace(-100, 100, 100)
X, Y = np.meshgrid(X, Y)                # create meshgrid
Z = f9([X, Y])                          # Calculate Z

# Plot the 3D surface for first function from project
fig = plt.figure()
ax = fig.gca(projection='3d')          # set the 3d axes
ax.plot_surface(X, Y, Z,
                rstride=3,
                cstride=3,
                alpha=0.3,
                cmap='hot')
plt.show()
```

Particle Swarm Optimization – algorithm code

Written in Python 2

pso.py

```
import numpy as np
import csv
import pylab as py
from functions import *

class Particle:
    def __init__(self, dim=10):  #set the dimension accordingly currently
D=10
        pass
        self.__dim = dim

class PSO:
    def __init__(self, func, bounds, startPosition=None):

        # Total particles in swarm
        self.Np = 100

        # Control Parameters
        self.error = 1
        self.max_nfc = 3000
        self.w = 0.6
        self.C1 = 2.05
        self.C2 = 2.05
        self.default = 1

        # Minimized function
        self.problem = func

        # Set boundary values
        self.minBound = np.array(bounds[0])
        self.maxBound = np.array(bounds[1])

        #Set Dimensions
        self.dim = len(bounds[0])


        # Set Initial positions for particles
        if startPosition!=None:
            self.startPosition =
np.array(startPosition).reshape((self.default,self.dim))
        else:
            self.startPosition = startPosition


    def __initParticle__ (self):
        """Initiate particles."""
```

```
        #Make particles
        self.Particles = []
        for i in range(self.Np):
            self.Particles.append( Particle(self.dim) )

        # Initiate positions and fit for particles
        for particle in self.Particles:

            # Initial position
            if self.startPosition == None:
                particle.pos = np.random.random(self.dim)*self.maxBound -
self.minBound
            else:
                particle.pos = self.startPosition[0,:]
                self.startPosition = np.delete(self.startPosition, 0,0)

                # If no initial positions left
                if len(self.startPosition) == 0:
                    self.startPosition = None

            # Initial velocity
            particle.Velocity = np.random.random(self.dim)*(self.maxBound
- self.minBound)
            particle.Velocity *= [-1, 1][np.random.random()>0.5]

            # Initial fitness
            particle.fitness = self.problem(particle.pos)
            particle.bestFitness = particle.fitness
            particle.bestPos = particle.pos

        # Global best fitness
        self.bestFitnessGlobal = self.Particles[0].fitness
        self.bestPosGlobal = self.Particles[0].pos
        for particle in self.Particles:
            if particle.fitness < self.bestFitnessGlobal:
                self.bestFitnessGlobal = particle.fitness
                self.bestPosGlobal = particle.pos

    def update(self):

        for particle in self.Particles:

            # Generating the random weights for Best Position influence
            personalWeight = np.random.random(2)
            groupWeight = np.random.random(2)

            w, C1, C2 = self.w, self.C1, self.C2

            # Update velocity
            v, pos = particle.Velocity, particle.pos
```

```
            particle.Velocity = self.w*v + C1* personalWeight
*(particle.bestPos-pos) + C2* groupWeight *(self.bestPosGlobal-pos)

            # New position
            particle.pos += particle.Velocity

            # If pos outside bounds
            if np.any(particle.pos<self.minBound):
                NFC = particle.pos<self.minBound
                particle.pos[NFC] = self.minBound[NFC]
            if np.any(particle.pos>self.maxBound):
                NFC = particle.pos>self.maxBound
                particle.pos[NFC] = self.maxBound[NFC]

            # New fitness
            particle.fitness = self.problem(particle.pos)

        # Global and local best fitness
        for particle in self.Particles:

            # Comparing to local best
            if particle.fitness < particle.bestFitness:
                particle.bestFitness = particle.fitness

            # Comparing to global best
            if particle.fitness < self.bestFitnessGlobal:
                self.bestFitnessGlobal = particle.fitness
                self.bestPosGlobal = particle.pos

    def optimize(self):
        """ Optimisation function.
            Before it is run, initial values should be set."""

        # Initiate particles
        self.__initParticles__ ()
        self.listOfPos = []

        NFC = 0
        while(NFC < self.max_nfc):
            #print "Run: " + str(NFC) + " Best: " +
str(self.bestFitnessGlobal)

            # Perform search
            self.update()

             #Acceptably close to solution
            if self.bestFitnessGlobal < self.error:
                return self.bestPosGlobal, self.bestFitnessGlobal

            # next gen
            NFC += 1
```

```
            self.listOfPos.append(self.bestFitnessGlobal)
        # Search finished
        return self.bestPosGlobal, self.bestFitnessGlobal, self.listOfPos

if __name__ == "__main__":
#----------------RUNS------------------#
    N = 100
    outputFile = open('Results2.csv', 'w')
    outputWriter = csv.writer(outputFile)
    outputWriter.writerow(['Function 6'])
    outputWriter.writerow(['Run','Best Fit', 'Best Solution'])
    t = np.linspace(-100, 100, N)
    minProb = lambda t: f6(t)           #we change functions accordingly
    numParam = 4
    bounds = ([0]*numParam, [10]*numParam)
    pso = PSO(minProb, bounds)
    for i in range(25):
        g = pso.optimize()
        outputWriter.writerow([[i+1],g[0], g[1]])

    py.figure()
    py.plot(g[2])
    py.xlabel("NFC")
    py.ylabel("Best Fit Performance")
    py.title("PSO Performance vs NFC")
    py.show()
```

Differential Evolution – algorithm code

Written in Python 3

de.py (run deTest.py, shown below)

```
from __future__ import division, print_function
import numpy as np
from numpy.random import random as _random, randint as _randint


class DiffEvolveOptimizer(object):

    def __init__(self, function, bound, population, f=0.8, cr=0.9):

        # Set Parameters
        self.function = function
        self.bound = np.asarray(bound)
        self.population = population
        self.f = f
        self.cr = cr
        self.dimension  = (self.bound).shape[0]

        # Set bounds
        lowerbound = self.bound[:, 0]
        upperbound = self.bound[:, 1] - self.bound[:, 0]

        # Set Population
        self.populationtwo = lowerbound[None, :] +
_random((self.population, self.dimension)) * upperbound[None, :]
        self.fitness = np.empty(population, dtype=float)
        self._minidx = None


    def step(self):
        rnd_cross = _random((self.population, self.dimension))
        for i in range(self.population):
            # Creates three unique random values to use
            one, two, three = i, i, i
            while one == i:
                one = _randint(self.population)
            while two == i or two == one:
                two = _randint(self.population)
            while three == i or three == one or three == two:
                three = _randint(self.population)

            v = self.populationtwo[one,:] + self.f *
(self.populationtwo[two,:] - self.populationtwo[three,:])

            # Crossover
            crossover = rnd_cross[i] <= self.cr
            u = np.where(crossover, v, self.populationtwo[i,:])
```

```
            randomthing = _randint(self.dimension)
            u[randomthing] = v[randomthing]

            ufit = self.function(u)

            # Find best fitness
            if ufit < self.fitness[i]:
                self.populationtwo[i,:] = u
                self.fitness[i] = ufit

    # Best Fit Value
    @property
    def value(self):
        return self.fitness[self._minidx]

    # Best Fit Solution
    @property
    def location(self):
        return self.populationtwo[self._minidx]

    # Best Fit Solution Index
    @property
    def index(self):
        return self._minidx

    # Optimizer
    def optimize(self, generation=100):

        for i in range(self.population):
            self.fitness[i] = self.function(self.populationtwo[i,:])

        for i in range(generation):
            self.step()
            self._minidx = np.argmin(self.fitness)
            yield self.populationtwo[self._minidx,:],
self.fitness[self._minidx]

    def __call__(self, generation=1):
        return self.optimize(generation)
```

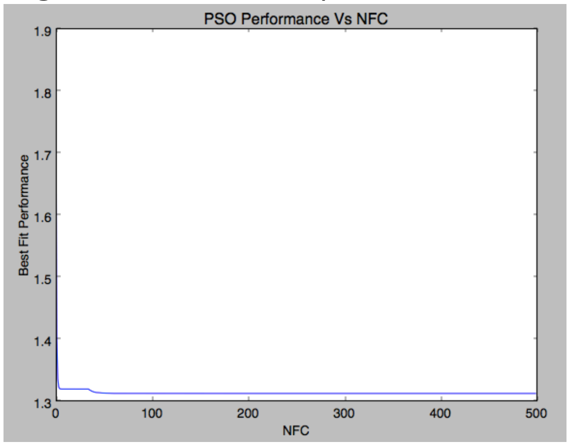
deTest.py

```
from de import DiffEvolveOptimizer
import matplotlib.pyplot as plt
import numpy as np
import csv
from functions import *

# Global Variables
generation = 3000
population = 100
dimension = 10

# Set Parameters
limits = [[-100, 100]] * dimension
ax = plt.subplot(2, 2, 2)

# Optimize Function (will be changed depending on which one will be used)
de = DiffEvolveOptimizer(f1, limits, population)

# Creates an Excel file and writes to it
outputFile = open('output.csv', 'w')
outputWriter = csv.writer(outputFile)
outputWriter.writerow(['Function 1'])
outputWriter.writerow(['Run','Best Fit', 'Best Solution'])

# Preforms multiple runs of optimization
for i in range(25):
    vals = []
    de.optimize()

    print("Best Fit Location: " + str(de.location))
    print("Best Fit Solution: " + str(de.value))
    vals.append(str(de.location))
    vals.append(str(de.value))

    # Also writes to Excel and stores values
    outputWriter.writerow([[i+1],vals[0], vals[1]])

pop = np.zeros([generation, population, dimension])
fvals = np.zeros([generation, dimension])

# Plot graph
plt.figure()
plt.plot(fvals, 'b-')
plt.title('DE Performance vs. NFC')
plt.ylabel('Best fitness error')
plt.xlabel('NFC')
plt.show()
```

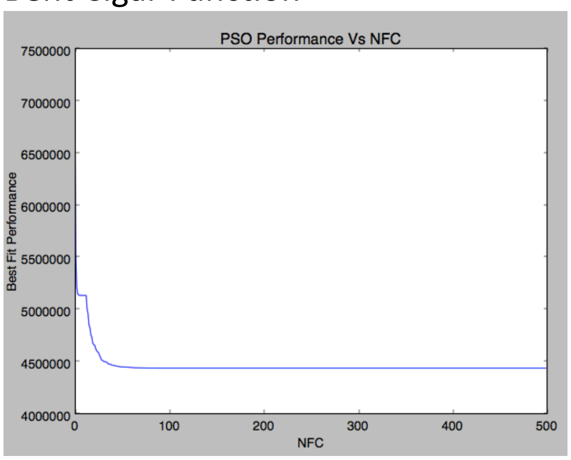
Performance Plots – Particle Swarm Optimization

D = 10

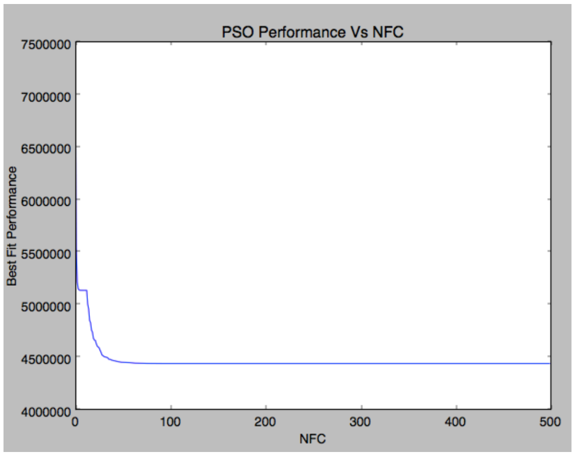
High Conditioned Elliptical Function



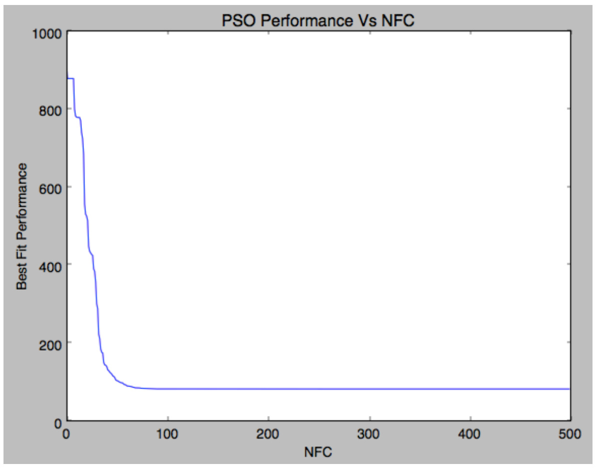
Bent Cigar Function



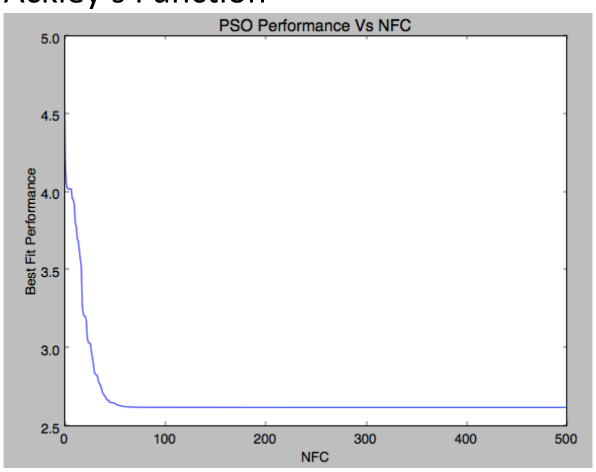
Discus Function



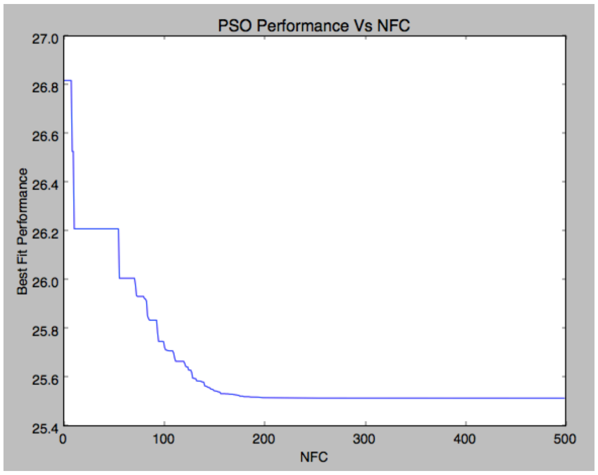
Rosenbrock's Function



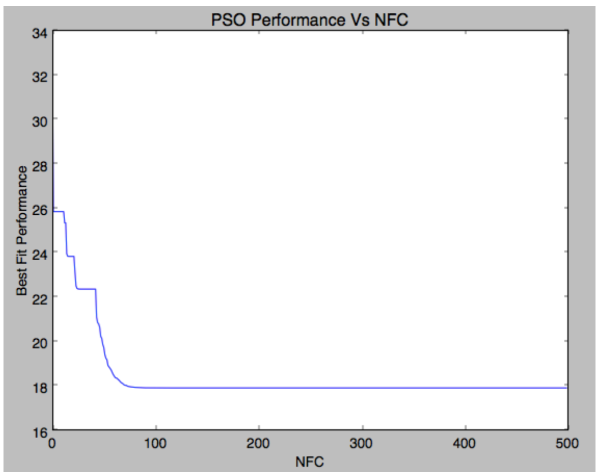
Ackley's Function



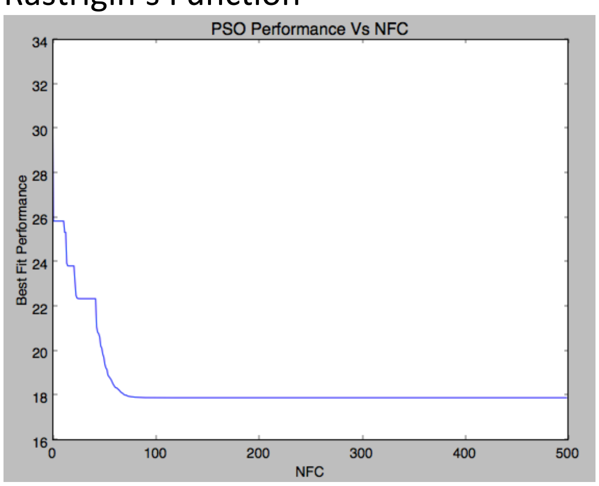
Weierstrass Function



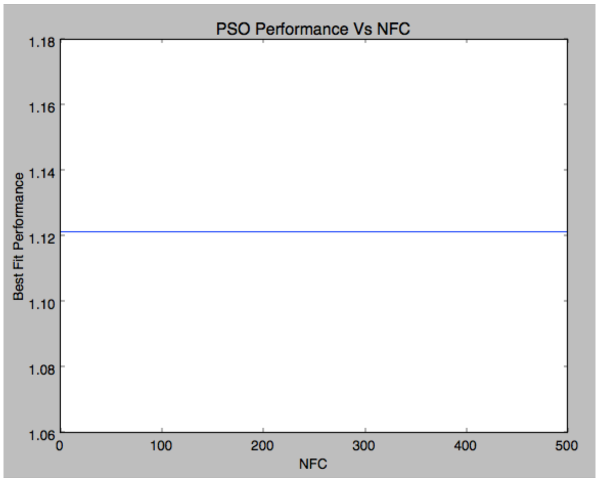
Griewank's Function



Rastrigin's Function

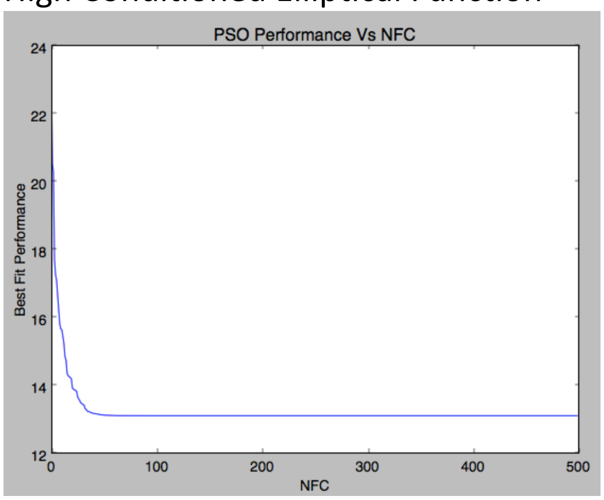


Katsuura Function

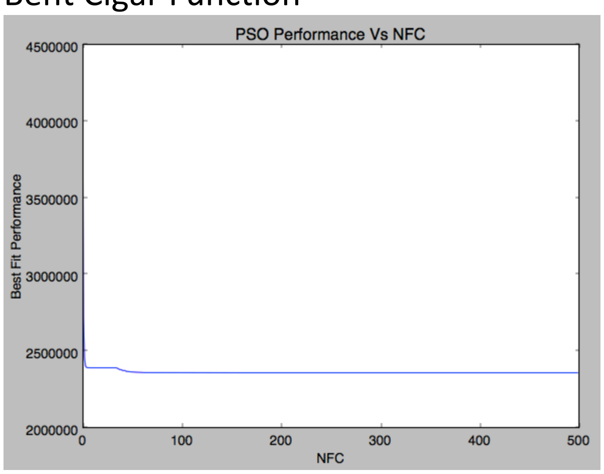


D = 30

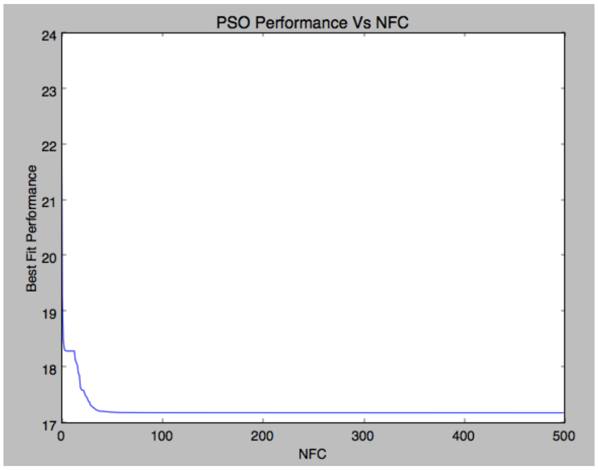
High Conditioned Elliptical Function



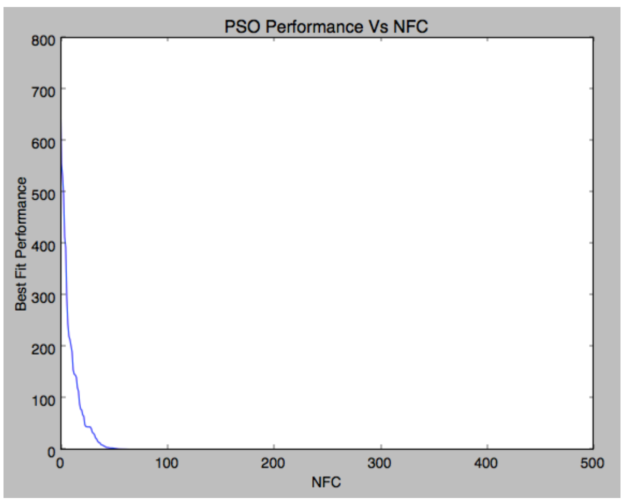
Bent Cigar Function



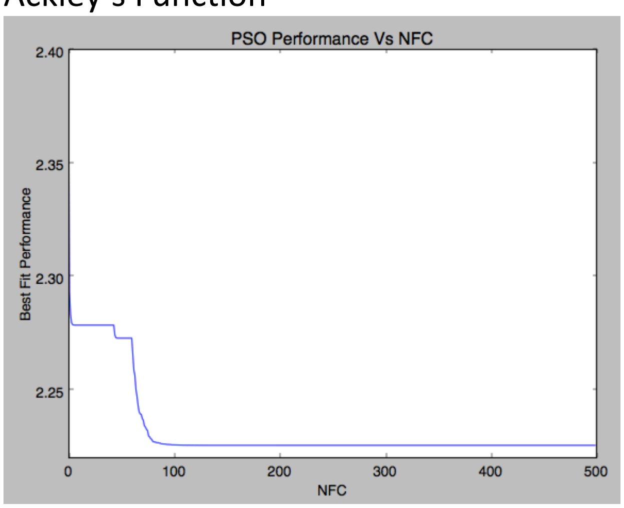
Discus Function



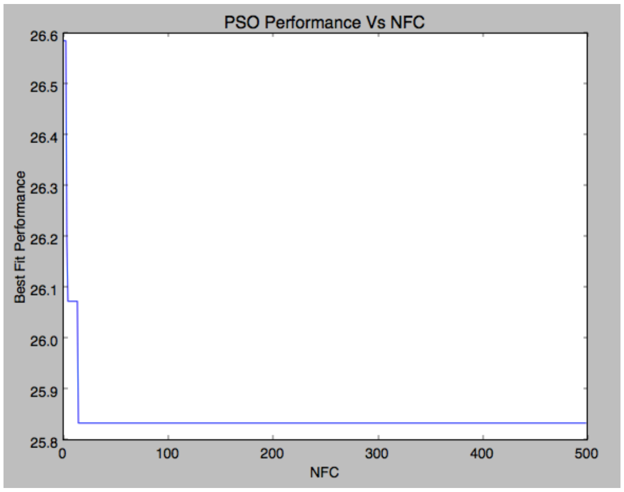
Rosenbrock’s Function



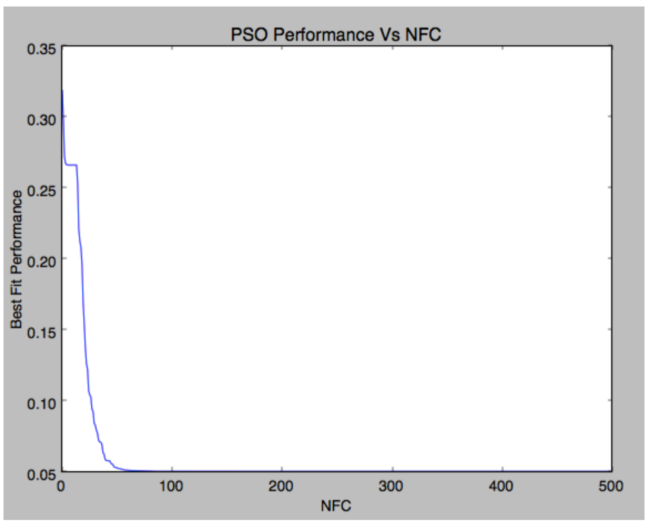
Ackley’s Function



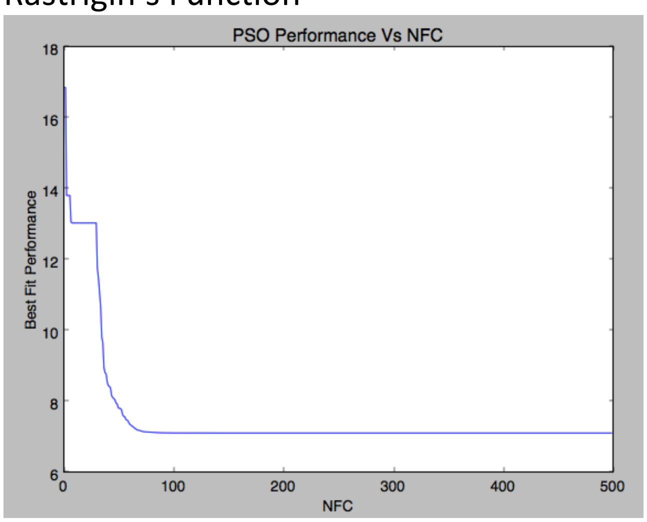
Weierstrass Function



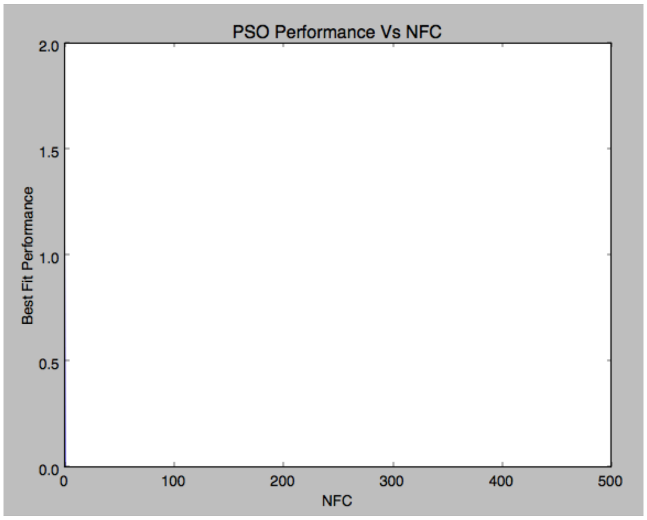
Griewank's Function



Rastrigin's Function

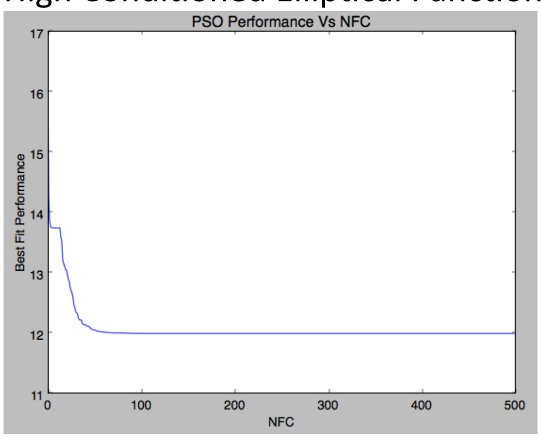


Katsuura Function

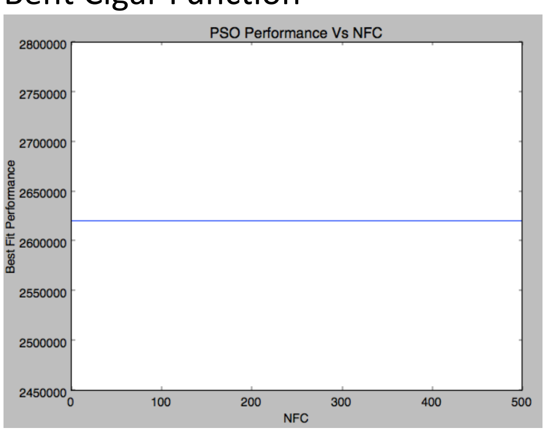


D = 50

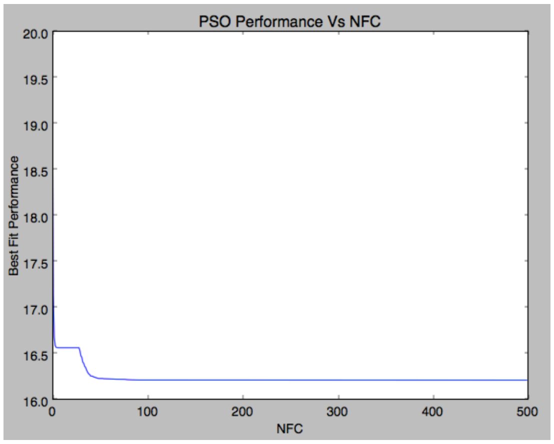
High Conditioned Elliptical Function



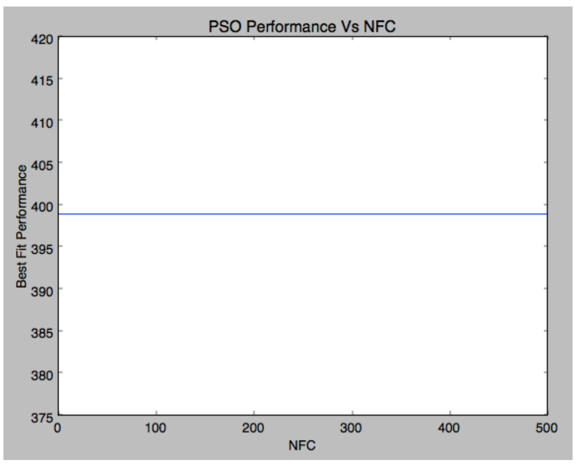
Bent Cigar Function



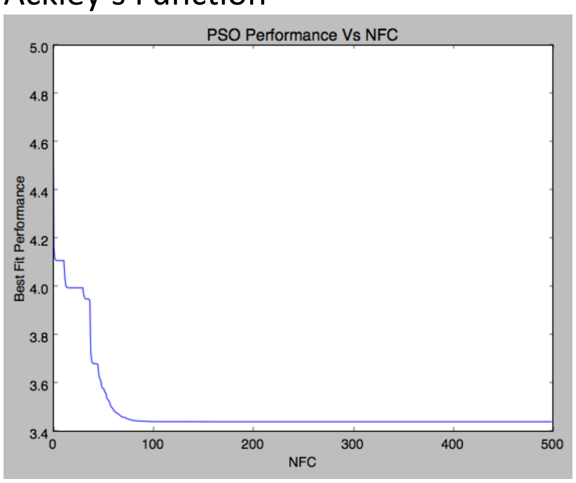
Discus Function



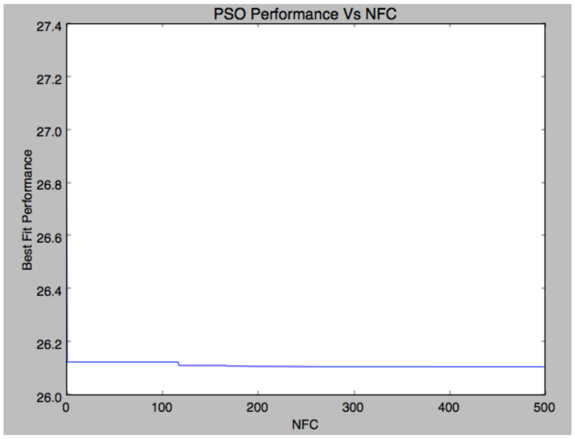
Rosenbrock's Function



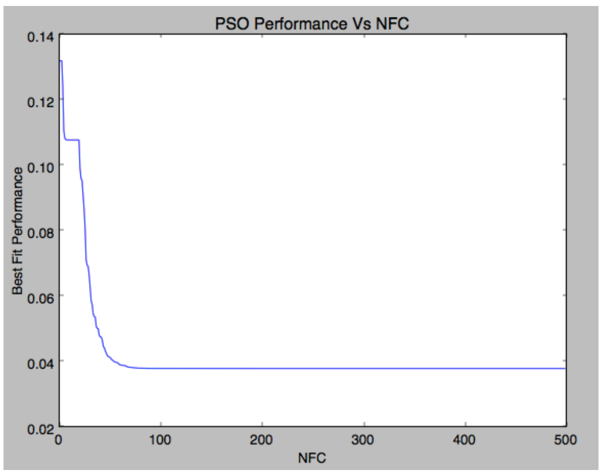
Ackley's Function



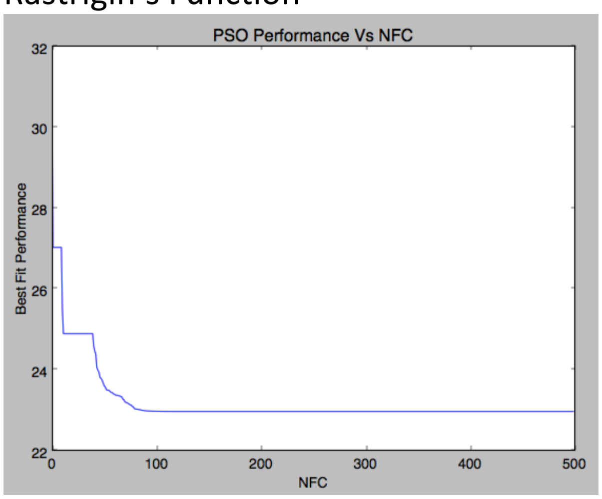
Weierstrass Function



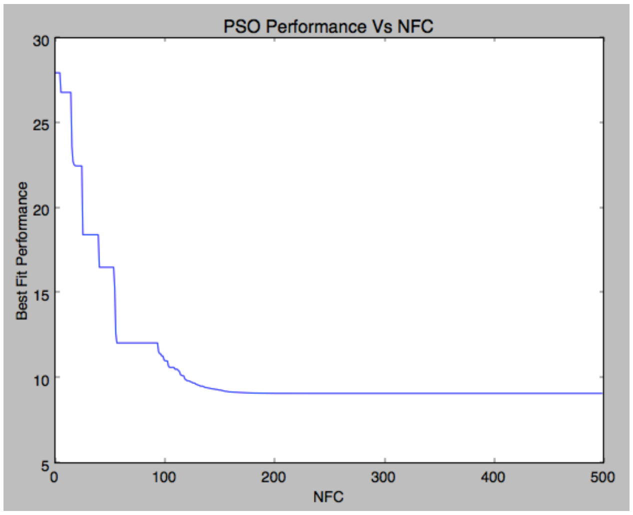
Griewank's Function



Rastrigin's Function



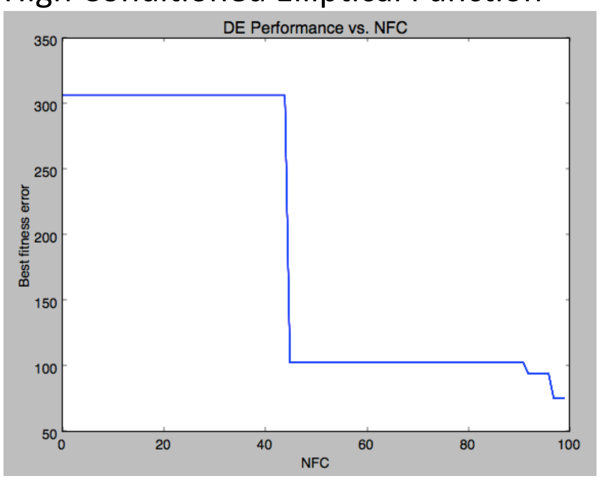
Katsuura Function



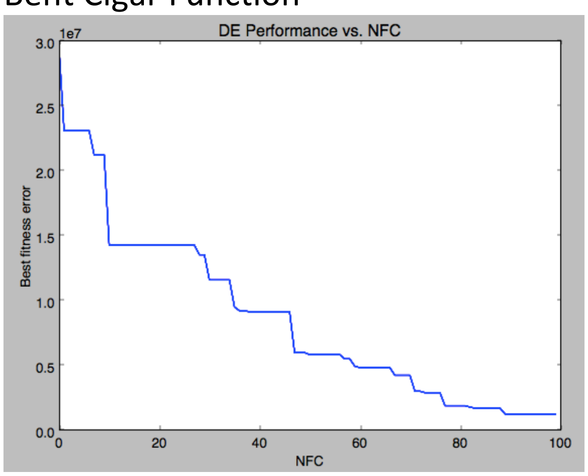
Performance Plots – Differential Evolution

D = 10

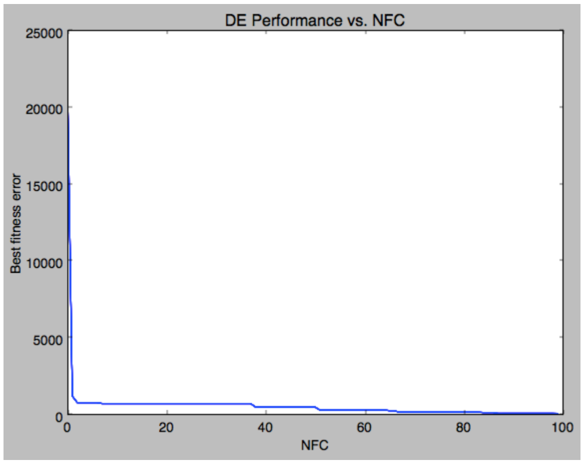
High Conditioned Elliptical Function



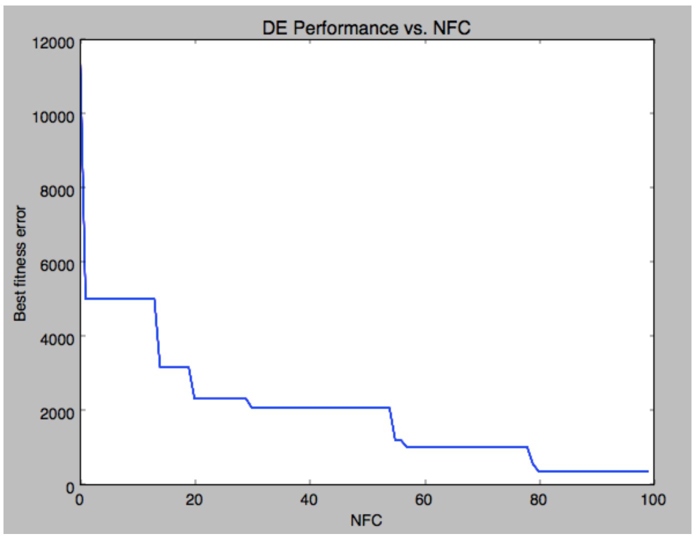
Bent Cigar Function



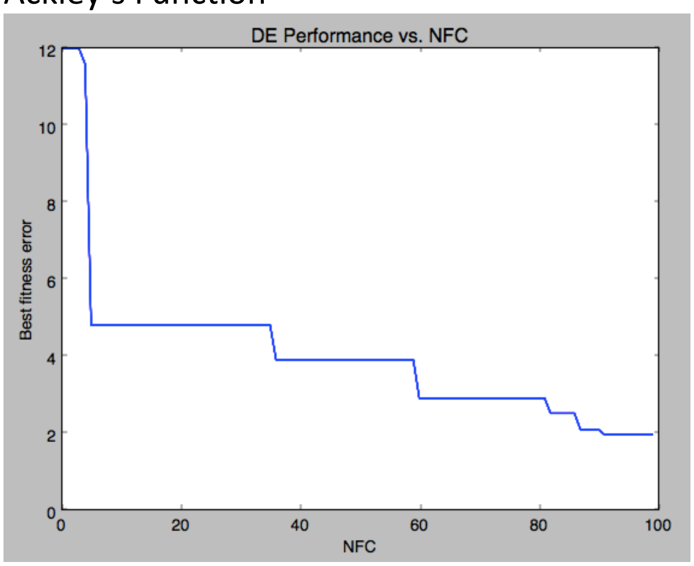
Discus Function



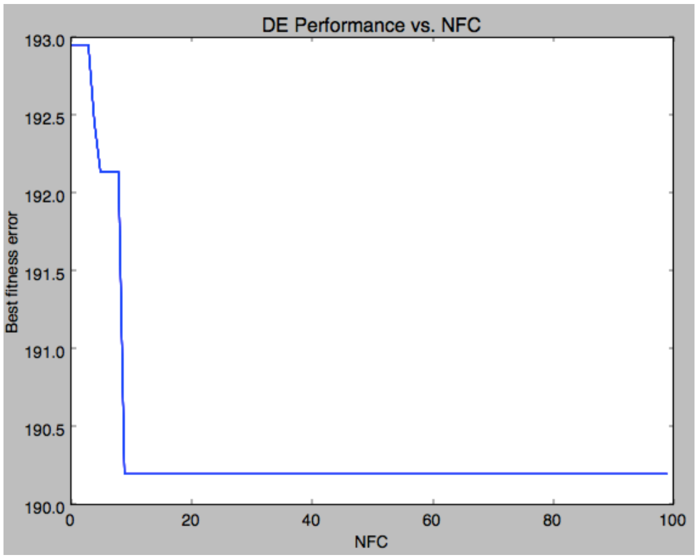
Rosenbrock's Function



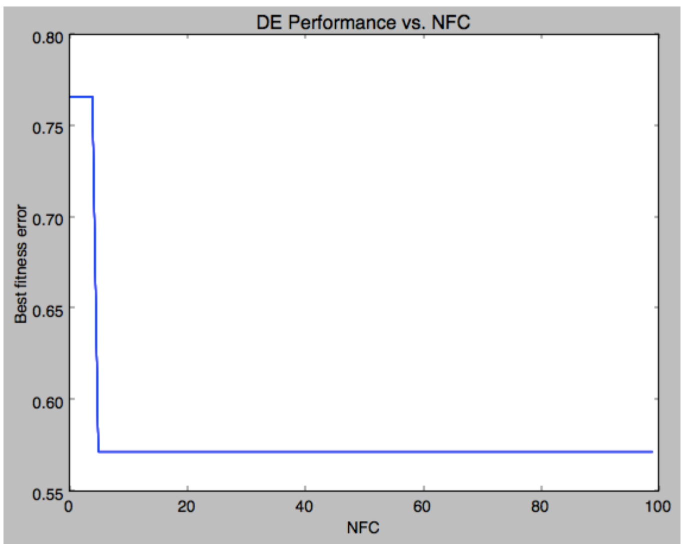
Ackley's Function



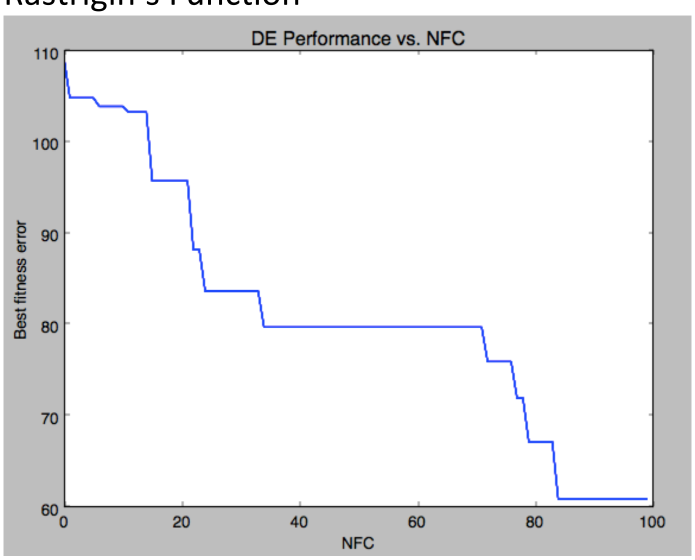
Weierstrass Function



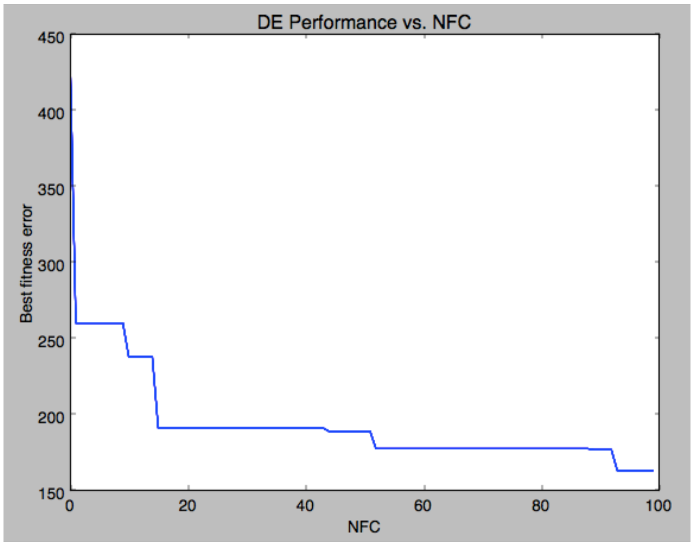
Griewank's Function



Rastrigin's Function

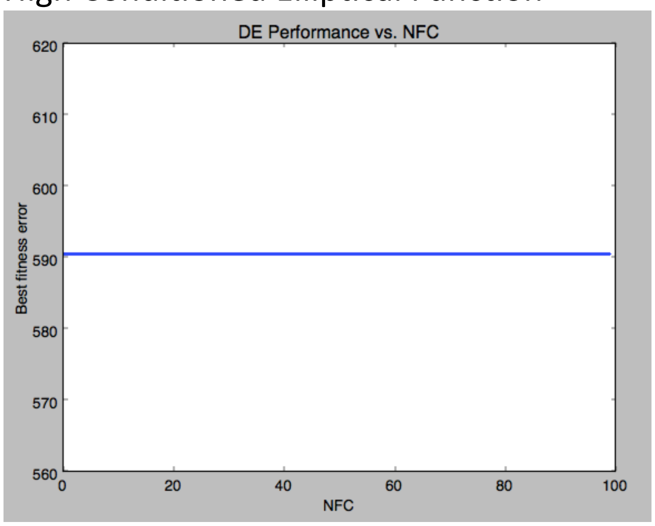


Katsuura Function

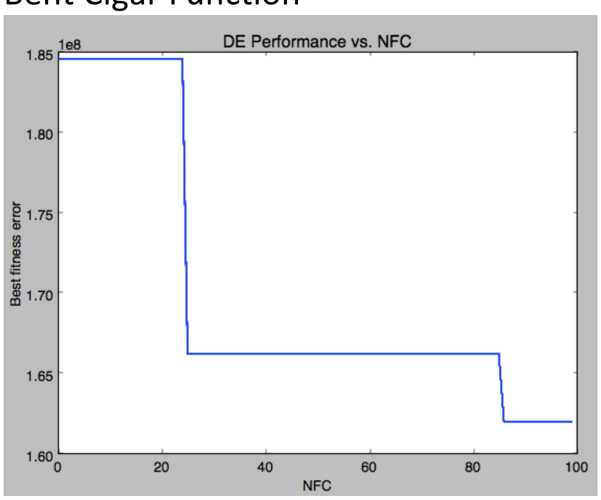


D = 30

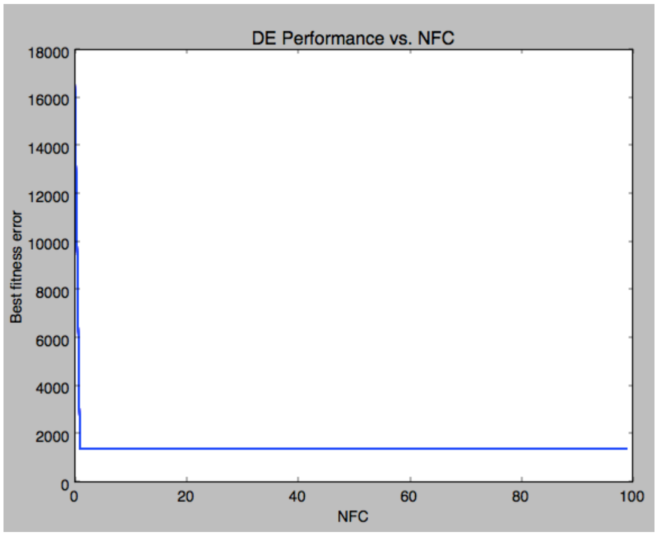
High Conditioned Elliptical Function



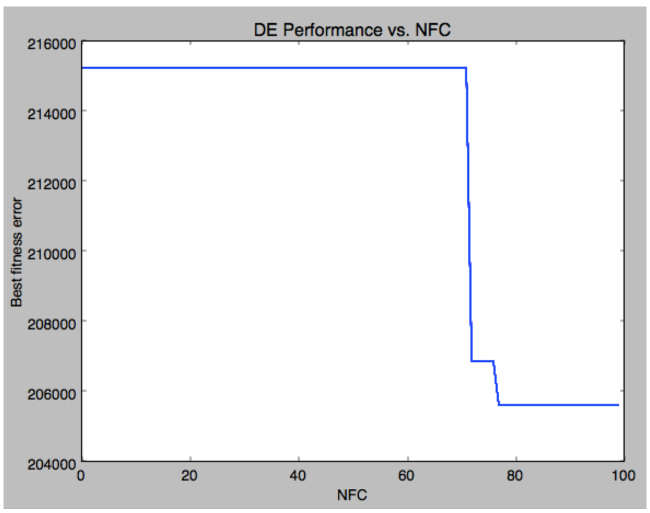
Bent Cigar Function



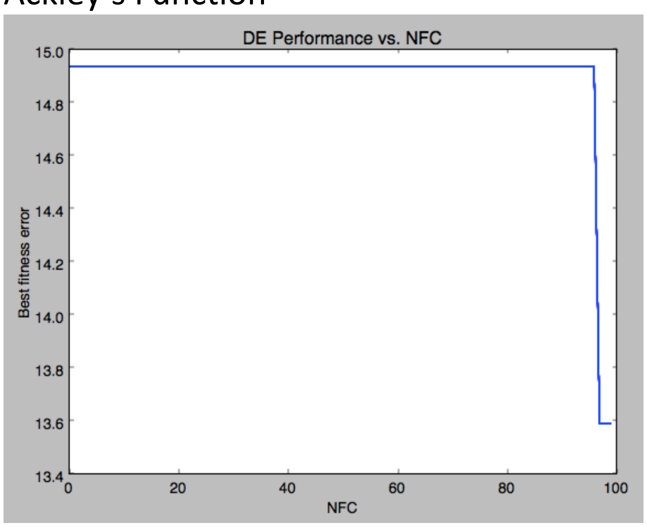
Discus Function



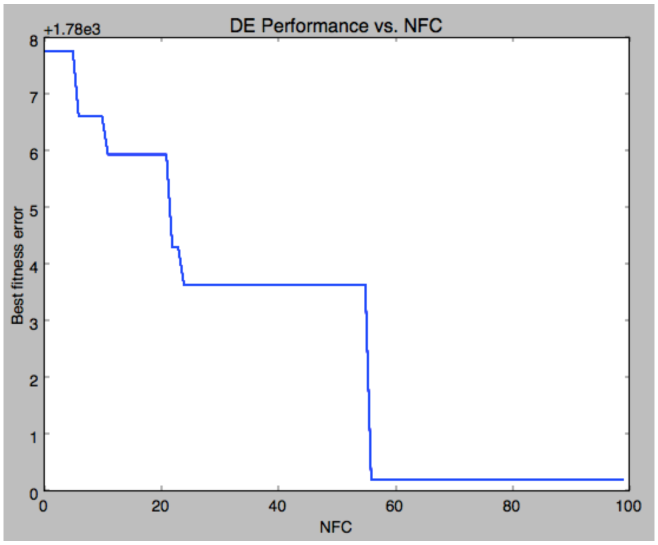
Rosenbrock's Function



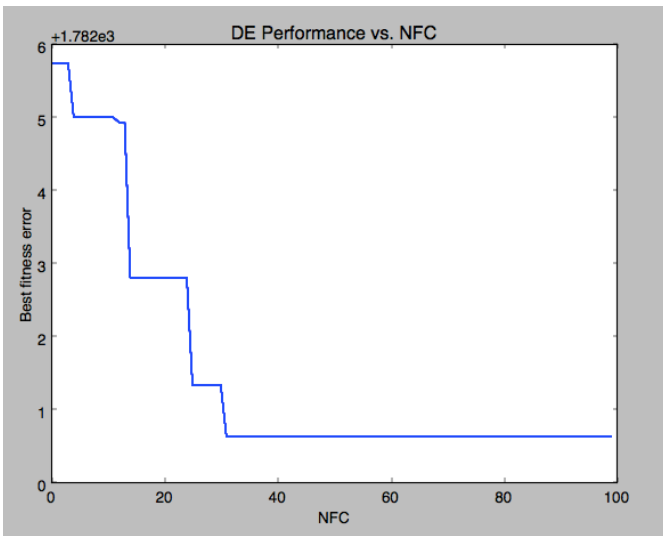
Ackley's Function



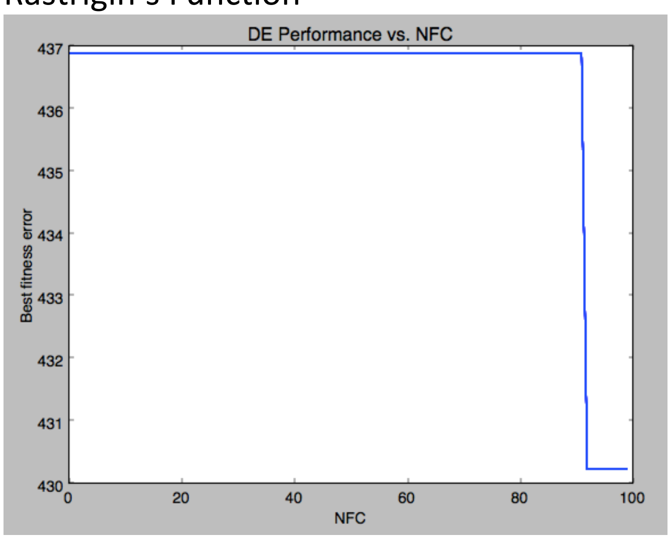
Weierstrass Function



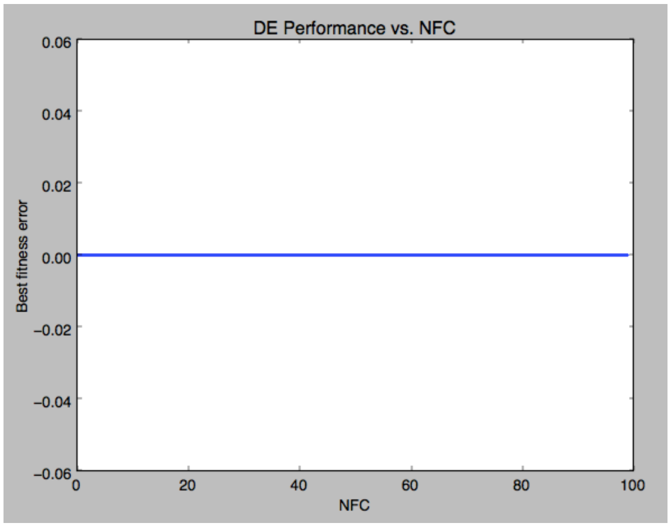
Griewank's Function



Rastrigin's Function

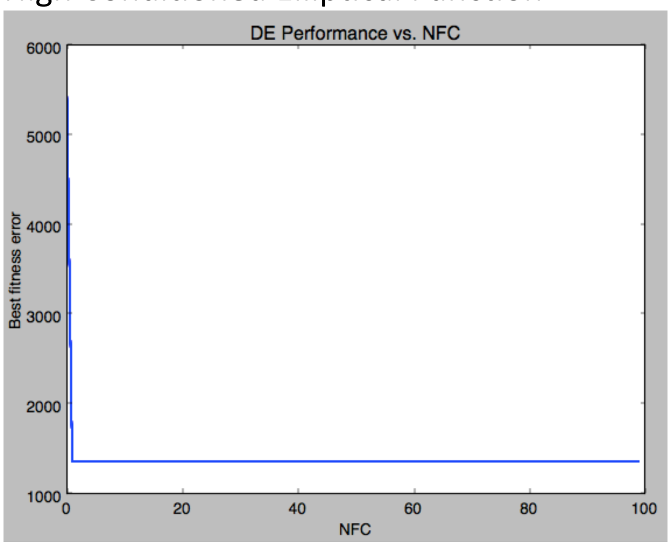


Katsuura Function

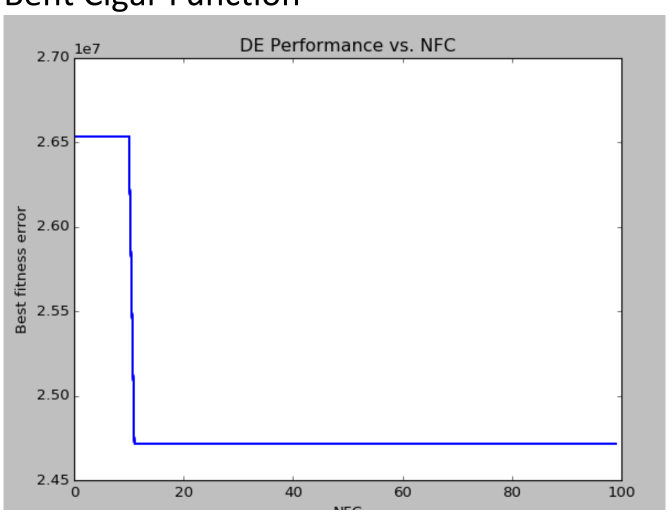


D = 50

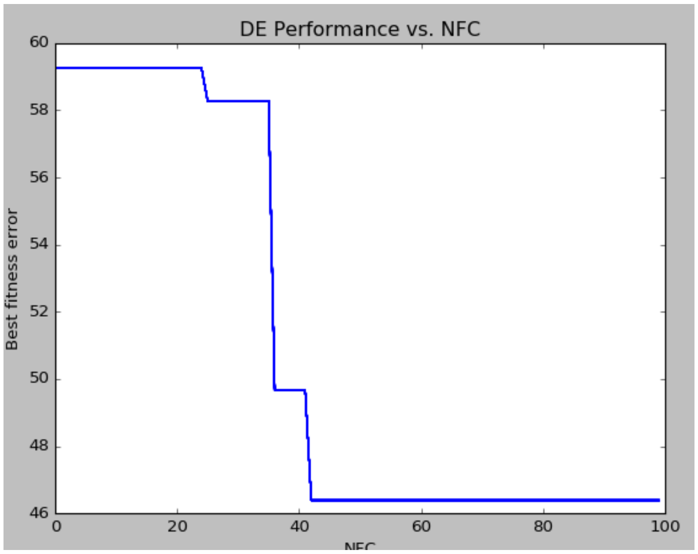
High Conditioned Elliptical Function



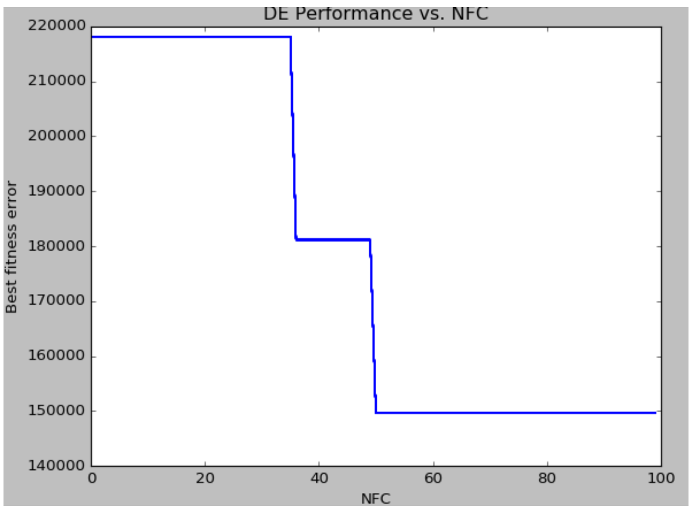
Bent Cigar Function



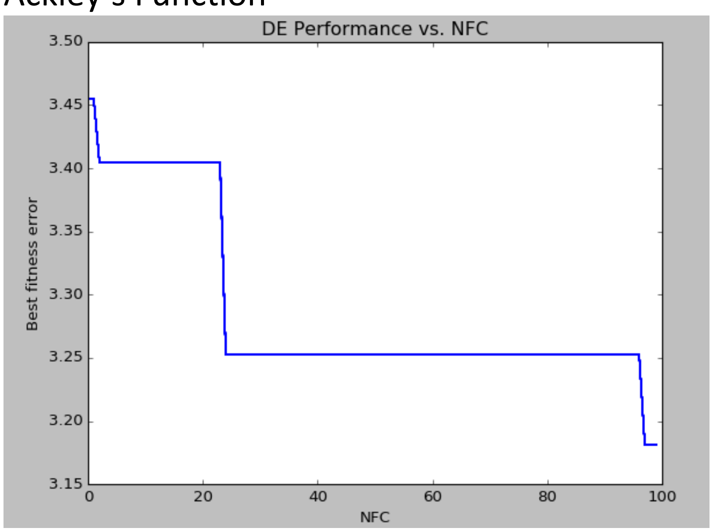
Discus Function



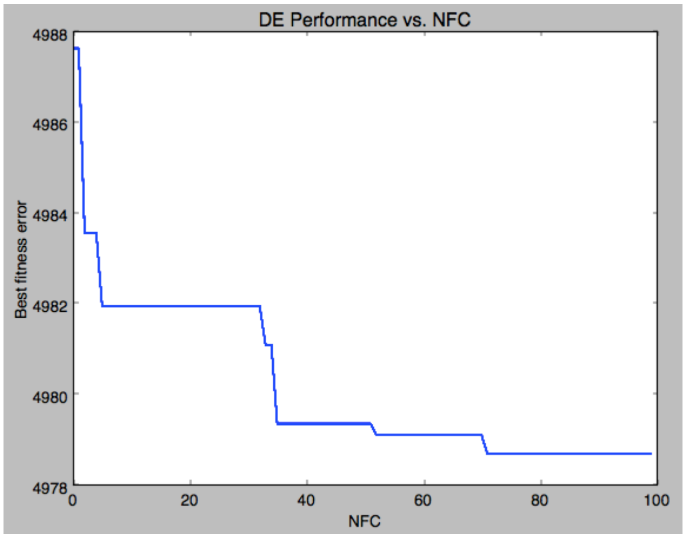
Rosenbrock's Function



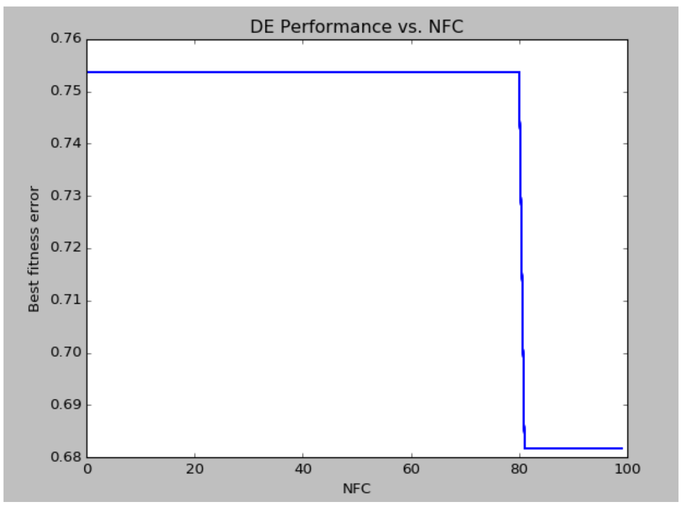
Ackley's Function



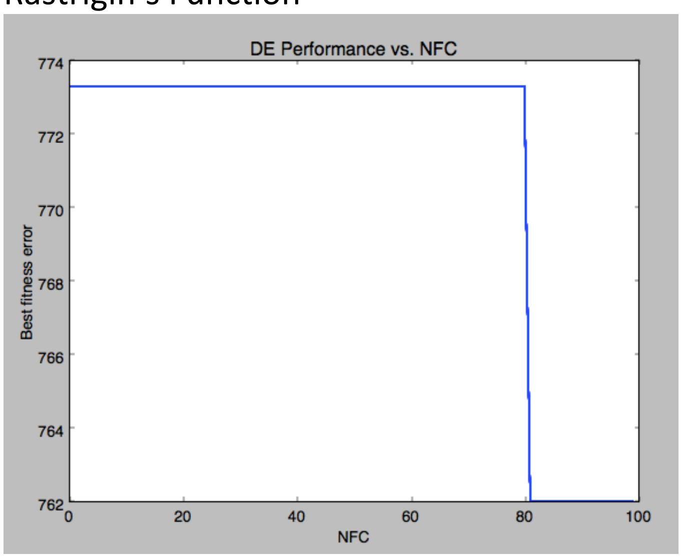
Weierstrass Function



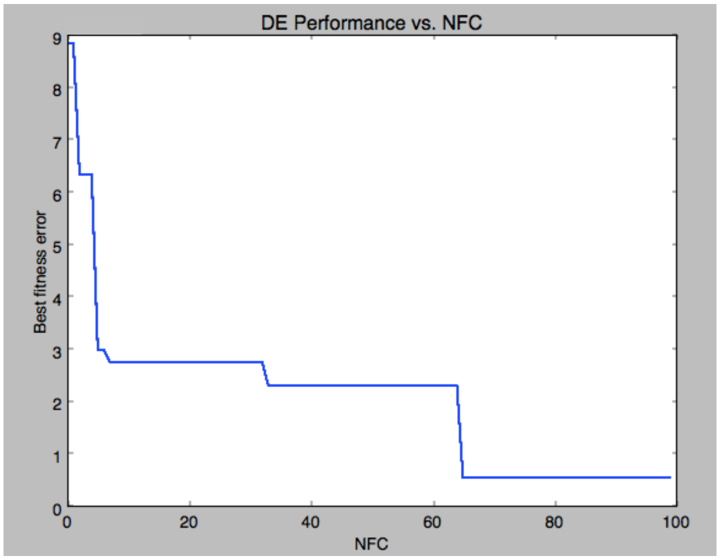
Griewank's Function



Rastrigin's Function



Katsuura Function



Comparison Table for D = 30

Function	Algorithm	Mean	Standard Deviation	Best Value	Worst Value	Best Solution	Fitness Value
f1	DE	135809.5462	154694.7358	10648.96861	456823.1382	10648.94562	[0.31772693 0.21603886 0. 0.43526619]
f1	PSO	5837587.53	3637716.28	98312.7524	12026627.1	98312.2826	[-1.42011038e+01 4.25800510e-01 1.19916602e+01 -2.50307809e+01]
f2	DE	16156874146	17399381284	1008238916	55525182726	1008238577	[1.04310441e-16 2.03558031e-01 7.94805326e-01 1.04492007e+00]
f2	PSO	20.3715843	14.3931186	1.76500896	68.6483713	1.76500915	[14.80774449 5.811829 -5.05645087 -3.61893471 4.31295447]
f3	DE	72373.50876	69585.84618	2797.0478120	197401.1648	2797.047852	[1.0021641 1.0043457 1.00874427 1.01762596]
f3	PSO	176.237184	498.418934	0.00010132	2470.46874	0.00010168	[-9.21135862e-01 5.19681580e+00 -6.64943337e+00 8.44204505e+00]
f4	DE	3013231496	4879391834	20449151.67	15339156479	20449165.48	[1.02279138 1.11720308 1.21003737 1.42514212]
f4	PSO	71.7719687	135.100937	0.86157865	429.471684	0.86158468	[-4.63435744e+00 3.11026965e+00 1.31371838e+01 1.17209034e+02]

f5	DE	20.87396513	0.076573678	20.83639278	21.02418648	20.836451489	[0.06508761 0. 0.12516617 0.17698568]
f5	PSO	6.06123724	3.56894276	0.61682458	16.0974642	0.61682354	[400.03941785 -52.20203135 1415.90927105 -693.88005137]
f6	DE	1780.12143	1.317784894	1784.277934	1778.446547	1784.278165	[1.11797244 1.20138689 1.50099024 2.27207842]
f6	PSO	46.801589	133.526537	0.90874616	576.6943584	0.90942437	[2450.94708641 312.93254199 -148.08908381 -2646.20544362]
f7	DE	4.751589104	4.686582476	1.097259818	17.21613587	1.09725982	[0. 0. 0. 0.]
F7	PSO	0.05785946	0.03844785	0.00020104	0.14530316	0.000008	[-2.72267493 5.57045055 -0.50709748 -0.15482845]
f8	DE	15118.8742	17704.78468	966.4411712	54766.05782	966.4413584	[0.99279254 0.99005337 0.99872935 0.00893607]
f8	PSO	15.0956818	7.83787644	3.00923516	30.8482354	3.007521684	[0.14668588 -5.4294287 2.2943706 -1.02880543

Conclusion

Differential Evolution was shown, in almost all cases to produce higher quality results. Using the Benchmark Functions, it was generally found to be a faster and more efficient algorithm.