Codes

THIS CODE IS FOR THE GRAPHING PORTION

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

#functions for project
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

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

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

def f4(x):
   """F8 Rosenbrock's saddle"""
   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

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))
```

```
    # Calculate first exp
    exp1 = -20.0 * (np.e ** (-0.2 * sum1))
    exp2 = np.e ** sum2

    # Calculate final result
    result = exp1 - exp2 + 20 + np.e
    return result

def f6(x):
        sum1, sum2, sum3 = 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):
                sum2 += (a ** k) * np.cos(2 * np.pi * (b ** k) * (x[i] + 0.5))
                sum3 += (a ** k) * np.cos(2 * np.pi * (b ** k) * 0.5)
        sum1 += sum2 - (len(x) * sum3)
        return sum1

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

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

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)))

#graphs for part 1
```

```
#Function 1
X = np.linspace(-100, 100, 100)          # points from 0..10 in the x axis
Y = np.linspace(-100, 100, 100)          # points from 0..10 in the y axis
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()

#Function 2
X = np.linspace(-100, 100, 100)          # points from 0..10 in the x axis
Y = np.linspace(-100, 100, 100)          # points from 0..10 in the y axis
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()

#Function 3
X = np.linspace(-100, 100, 100)          # points from 0..10 in the x axis
Y = np.linspace(-100, 100, 100)          # points from 0..10 in the y axis
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()

#Function 4
X = np.linspace(-100, 100, 100)          # points from 0..10 in the x axis
Y = np.linspace(-100, 100, 100)          # points from 0..10 in the y axis
```

```
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()

#Function 5
X = np.linspace(-100, 100, 100)         # points from 0..10 in the x axis
Y = np.linspace(-100, 100, 100)         # points from 0..10 in the y axis
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()

#Function 6
X = np.linspace(-100, 100, 100)         # points from 0..10 in the x axis
Y = np.linspace(-100, 100, 100)         # points from 0..10 in the y axis
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()

#Function 7
X = np.linspace(-100, 100, 100)         # points from 0..10 in the x axis
Y = np.linspace(-100, 100, 100)         # points from 0..10 in the y axis
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()

#Function 8
X = np.linspace(-100, 100, 100)          # points from 0..10 in the x axis
Y = np.linspace(-100, 100, 100)          # points from 0..10 in the y axis
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()

#Function 9
X = np.linspace(-100, 100, 100)          # points from 0..10 in the x axis
Y = np.linspace(-100, 100, 100)          # points from 0..10 in the y axis
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()
```

THIS CODE IS FOR PARTICLE SWARM OPTIMIZATION

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


class Particle:
    def __init__(self, dim=10):
        pass
        self.__dim = dim

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

        # number of particles in swarm
        self.nPart = 100

        # Control Parameters
        self.epsError = 1
        self.maxGen = 3000
        self.w = 0.2
        self.phiP = 0.2
        self.phiG = 0.1
        self.default = -1

        # Function to be minimised
        self.problem = func

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

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


        # Initial positions
        if initPos!=None:
            self.initPos = np.array(initPos).reshape((self.default,self.dim))
        else:
            self.initPos = initPos


    def __initPart(self):
```

```
    """Initiate particles.
    """

    # Create particles
    self.Particles = []
    for i in range(self.nPart):
        self.Particles.append( Particle(self.dim) )

    # Initiate pos and fit for particles
    for part in self.Particles:

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

            # If nothing left on initial pos
            if len(self.initPos) == 0:
                self.initPos = None

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

        # Initial fitness
        part.fitness = self.problem(part.pos)
        part.bestFit = part.fitness
        part.bestPos = part.pos

    # Global best fitness
    self.globBestFit = self.Particles[0].fitness
    self.globBestPos = self.Particles[0].pos
    for part in self.Particles:
        if part.fitness < self.globBestFit:
            self.globBestFit = part.fitness
            self.globBestPos = part.pos

def update(self):

    for part in self.Particles:

        # Gen param
        rP, rG = np.random.random(2)

        w, phiP, phiG = self.w, self.phiP, self.phiG

        # Update velocity
        v, pos = part.vel, part.pos
        part.vel = self.w*v + phiP*rP*(part.bestPos-pos) + phiG*rG*(self.globBestPos-pos)
```

```
        # New position
        part.pos += part.vel

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

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

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

        # Comparing to local best
        if part.fitness < part.bestFit:
            part.bestFit = part.fitness

        # Comparing to global best
        if part.fitness < self.globBestFit:
            self.globBestFit = part.fitness
            self.globBestPos = part.pos

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

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

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

        # Perform search
        self.update()

         #Acceptably close to solution
        if self.globBestFit < self.epsError:
            return self.globBestPos, self.globBestFit

        # next gen
        NFC += 1
        self.listOfPos.append(self.globBestFit)
    # Search finished
```

```
        return self.globBestPos, self.globBestFit, self.listOfPos

####################################

if __name__ == "__main__":
#############RUNS#######################
    N = 100
    outputFile = open('output9.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)
    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]])

    ##############################
    # Visual results representation---uncomment for plotting performance
    py.figure()
    py.plot(g[2])
    py.xlabel("NFC")
    py.ylabel("Best Fit Performance")
    py.title("PSO Performance Vs NFC")
    py.show()
```

THIS CODE IS FOR DIFFERENTIAL EVOLUTION

```
from __future__ import division, print_function
from algorithmChecker import *

import numpy as np
from numpy.random import random as _random, randint as _randint


class DiffEvolOptimizer(object):

    def __init__(self, fun, bounds, npop, F=0.8, C=0.9, seed=None, maximize=False):
        if seed is not None:
            np.random.seed(seed)

        self.fun = fun
        self.bounds = np.asarray(bounds)
        self.npop = npop
        self.F = F
        self.C = C

        self.ndim  = (self.bounds).shape[0]
        self.m  = -1 if maximize else 1

        bl = self.bounds[:, 0]
        bw = self.bounds[:, 1] - self.bounds[:, 0]
        self.population = bl[None, :] + _random((self.npop, self.ndim)) * bw[None, :]
        self.fitness = np.empty(npop, dtype=float)
        self._minidx = None

    def step(self):
        """Take a step in the optimization"""
        rnd_cross = _random((self.npop, self.ndim))
        for i in xrange(self.npop):
            t0, t1, t2 = i, i, i
            while t0 == i:
                t0 = _randint(self.npop)
            while t1 == i or t1 == t0:
                t1 = _randint(self.npop)
            while t2 == i or t2 == t0 or t2 == t1:
                t2 = _randint(self.npop)

            v = self.population[t0,:] + self.F * (self.population[t1,:] - self.population[t2,:])
```

```
            crossover = rnd_cross[i] <= self.C
            u = np.where(crossover, v, self.population[i,:])

            ri = _randint(self.ndim)
            u[ri] = v[ri]

            ufit = self.m * self.fun(u)

            if ufit < self.fitness[i]:
                self.population[i,:] = u
                self.fitness[i] = ufit

    @property
    def value(self):
        """The best-fit value of the optimized function"""
        return self.fitness[self._minidx]

    @property
    def location(self):
        """The best-fit solution"""
        return self.population[self._minidx]

    @property
    def index(self):
        """Index of the best-fit solution"""
        return self._minidx


    def iteroptimize(self, ngen=100):

        for i in xrange(self.npop):
            self.fitness[i] = self.m * self.fun(self.population[i,:])

        for j in xrange(ngen):
            self.step()
            self._minidx = np.argmin(self.fitness)
            #print("Fitness Value: " + str(self.fitness))
            yield self.population[self._minidx,:], self.fitness[self._minidx]
            #print("Fitness Value: " + str(self.fitness[self._minidx]))
    def __call__(self, ngen=1):
        return self.iteroptimize(ngen)

from de import DiffEvolOptimizer
import matplotlib.pyplot as plt
import numpy as np
import csv


# setup the optimization
ngen, npop, ndim = 3000, 100, 10
```

```
limits = [[-5, 5]] * ndim
ax = plt.subplot(2, 2, 2)
de = DiffEvolOptimizer(f1, limits, npop)
outputFile = open('output.csv', 'w')
outputWriter = csv.writer(outputFile)
outputWriter.writerow(['Function 1'])
outputWriter.writerow(['Run','Best Fit', 'Best Solution'])
for i in range(6):
    g=[]
    de.iteroptimize()
    print("Best Fit Location: " + str(de.location))
    print("Best Fit Solution: " + str(de.value))
    g.append(str(de.location))
    g.append(str(de.value))
    outputWriter.writerow([[i+1],g[0], g[1]])
# store all the values during iterations for plotting.
pop = np.zeros([ngen, npop, ndim])
loc = np.zeros([ngen, ndim])
for i, res in enumerate(de(ngen)):
    loc[i,:] = de.value.copy()
print("Best Fit Location: " + str(de.location))
print("Best Fit Solution: " + str(de.value))

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