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

scopefactors = list()
xline = list()
yline = list()
zline = list()
def DSA(lb, ub, dim, iters, func, convergence):

    PopSize = 3**dim #every dimension will exponentially increase the number of 
particles
    particles = numpy.zeros((PopSize,dim)) #create a matrix of the particles, ex. 2 
dimensions will have one explorer particles and eight scope particles
    fit = numpy.zeros(PopSize) #array of the fitness of each particle
    best = numpy.zeros(dim) #position of the best particle (with the miniumum value)
    bestfit = float("inf") #best value (minimum)

    pos = numpy.random.uniform(0,1,(dim))*(ub-lb)+lb #randomly places the explorer 
particle within the upper and lower bounds
    #convergence_curve=numpy.zeros(iters)
    restart = 0 #to avoid resetting the explorer again after one iteration
    #repeated = 0 #used track repeats, to tighten the scope if the scope is too big
    countdown = iters*PopSize*dim #total number of k values needed
    v = contractRetract(countdown,1,100,0.05)
    T = 100 #initial temperature
    cooldown = 0.95 #temperature cooldown rate

    for i in range(dim):
            particles[0,i] = pos[i].item()

    #THIS WHOLE SECTION UPDATES THE "particles" matrix
    for l in range(iters): #for every iteration

        for i in range(dim): #for each dimensions, ex. x coord -> y coord -> z coord
            switch = 0 #trinary, since the scope can only to stay, subtracted, or 
added from the explorer's coordinate
            count = (3**(dim-1))/(3**i)
            counter = 0
            for j in range(0, PopSize): #for every particle
                k = contractRetract(countdown,1,100,0.05)/v #the input values are 
arbitrary, the v makes the output proportional
                countdown -= 1
                if k < 0: #keeps the output positive
                    k = -1*k
                scopefactors.append(k)

                #each particle will get a random scopefactor depending on iterations 
and the convergence factor
                ScopeFactor = numpy.random.uniform(0,k)*((ub-lb)/(l+1)**convergence)
                #scopefactors.append(ScopeFactor)
                if switch == 0:
                    counter +=1
                    particles[j,i] = particles[0,i].item()
                elif switch == 1 : #first scope particle gets the positive movement 
from the explorer's current position
                    counter +=1
                    positivestep = particles[0,i].item() + ScopeFactor #move in a 
positive direction (North or East in a 2-dimensional position)
                    if positivestep <= ub: #the new step is within the upper bound 
of the problem
                        particles[j,i] = positivestep
                    else:
                        particles[j,i] = ub
                    switch = 1
                elif switch == 2: #second scope particle gets the negative movement 
from the explorer's current position
                    counter +=1
                    negativestep = particles[0,i].item() - ScopeFactor #move in a 
negative direction (South or West in a 2-dimensional position)
                    if negativestep >= lb: #the new step is within the lower bound 
of the problem
                        particles[j,i] = negativestep
                    else:
                        particles[j,i] = lb
                if counter == count:
                    counter = 0
                    switch = (switch+1)%3


        best, fit = calcBestFitness(particles, PopSize, dim, bestfit, func) 
#calcuate the best (position of the best particle), fit (array of all particles' 
fitness)
        oldbestfit = bestfit
        #bestfit = min(fit) #get the minimum fitness "Beam search"
        bestfit = simulated_annealing(fit,oldbestfit,T) #less greedy"
        T = cooldown*T #temperature cooldown
        xline.append(best[0])
        yline.append(best[1])
        zline.append(bestfit)
        for i in range(dim):
            particles[0,i] = best[i]
    print("Best Solution: ", best, " Value: ", bestfit)
```

```
def simulated_annealing(fit,oldbestfit, T):
    p = 0
    bestfit = min(fit)
    if bestfit <= oldbestfit:
        return bestfit
    else:
        p = exp(-(bestfit-oldbestfit)/T)
        if random.uniform(0,1) > p:
            return bestfit
        else:
            return oldbestfit


def calcBestFitness(particles, PopSize, dim, bestfit, func):
    fit = numpy.zeros(PopSize)
    best = numpy.zeros(dim)

    for i,pos in enumerate(particles):

        fitness = func(pos)

        fit[i] = fitness

    best = particles[numpy.argmin(fit)] #get the particle with the lowest fitness

    return best, fit

def contractRetract(x,A,B,C):
    return A*(x**2)+ x*B*(math.cos(C*math.pi*x))

############################### FUNCTIONS
###################################################
def function1(x): #f(0,0,...0) = 0
    total=0
    for i in range(len(x)):
        total+= (x[i])**2
    return total

def function2(coord): #Beale Function: f(3,0.5) = 0
    x = coord[0]
    y = coord[1]

    f = (1.5-x+(x*y))**2+(2.25-x+(x*(y**2)))**2+(2.625-x+(x*(y**3)))**2
    return f

def function3(coord): #Levi Function: f(1,1) = 0
```

```
    x = coord[0]
    y = coord[1]
    pi = math.pi
    f =
((math.sin(3*pi*x))**2)+((x-1)**2)*(1+(math.sin(3*pi*y))**2)+((y-1)**2)*(1+(math.sin
(2*pi*y))**2)
    return f
def function4(coord): #Eggholder function f(512, 404.2319) = -959.6407
    x = coord[0]
    y = coord[1]
    f =(-(y + 47.0)*np.sin(np.sqrt(abs(x/2.0 + (y + 47.0))))- x *
np.sin(np.sqrt(abs(x - (y + 47.0)))))
    return f

#For positional dimension, it is one dimension less than the actual function search
space.
convergence = 0
for i in range(1): #run the test 10 times
    DSA(-512,512,2,100, function4,convergence)


############################## PLOT
##############################################################

plt.plot(scopefactors)
plt.title('Contract-Retract')
#plt.title('Convergence Rate: '+ str(convergence))
plt.xlabel('Iteration per particle')
plt.ylabel('k-value')
plt.show()
x = np.linspace(-10, 10, 30)
y = np.linspace(-10, 10, 30)

X, Y = np.meshgrid(x, y)
#Z = function3(coord)
fig = plt.figure()
#ax = fig.add_subplot(111, projection='3d')
ax = plt.axes(projection='3d')
# Data for a three-dimensional line
ax.plot3D(xline, yline, zline, 'blue')
#ax.contour3D(X, Y, Z, 50, cmap='binary')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');

fig.show()
```