```python
import numpy as np
import matplotlib.pyplot as plt
from functions import *
from evophase import *

global LOW, HIGH
LOW = 0.3
HIGH = 0.6

class Evolution:
    #Initiate parameters
    def __init__(self, population_tot, crossover_rate, mutation_rate, generations, N, target):
        self.population_tot = population_tot
        self.crossover_rate = crossover_rate
        self.mutation_rate = mutation_rate
        self.generations = generations
        self.N = N
        self.target = target

        # Spacing between antennas
        self.pop_dis = np.ndarray(shape=(self.population_tot, self.N-1))

        # Non-uniform symetrically spaced LAA
        for i in range(self.population_tot):
            a = np.random.uniform(low = LOW, high=HIGH)
            b = np.random.uniform(low = LOW, high=HIGH)
            self.pop_dis[i][4] = a;
            self.pop_dis[i][0] = a
            self.pop_dis[i][1] = b
            self.pop_dis[i][3] = b

            self.pop_dis[i][2] = np.random.uniform(low = LOW, high=HIGH)

        # Phase Shift of the signal of each antenna element.
        self.pop_pha = np.random.uniform(size = (self.population_tot, self.N), low = 0, high = 2*np.pi) # randians

        # Amplitude of the signal of each antenna element.
        self.pop_amp = np.random.uniform(size=(self.population_tot, self.N), low = 1, high=1)

        # Fitness scores
        self.pop_fit = np.zeros(self.population_tot, dtype = np.float)

        self.evolve()

    def results(self):
        # Return the best solution
        q = np.argmin(self.pop_fit)
        return self.pop_dis[q], self.pop_pha[q], self.pop_amp[q]

    def evolve(self):
        # probabilities for crossover. To avoid calculationg random numbers every itteration.
        crossovers = np.random.uniform(size=(self.generations), low = 0, high=1)
        mutations = np.random.uniform(size=(self.generations), low = 0, high=1)
        # Probabilities for which parameter to mutate or to undergo crossover
        choice = np.random.uniform(size=(self.generations), low = 0, high=1)

        for i in range(self.generations):

            for dna in range(self.population_tot):
                score = fitness(self.pop_dis[dna], self.pop_pha[dna], self.pop_amp[dna], self.target)
                self.pop_fit[dna] = score;
            # Index of fittest solution
            index = np.argmin(self.pop_fit)
            # Copy fitteset solution over to next genertation (first and last index of array)
            self.pop_dis[0] = self.pop_dis[index]
            self.pop_pha[0] = self.pop_pha[index]
            self.pop_dis[-1] = self.pop_dis[index]
            self.pop_pha[-1] = self.pop_pha[index]

            # Optimize the phase shift for the fittest solution.
            evo = EvoPhase(5, 0.5, 1, 2000, self.N, self.pop_dis[0], self.target)
            self.pop_pha[0] = evo.results()

            # Stop simulating when the generation matches the maximun no of generations.
            if i == self.generations-1:
                print('Finished')
                break

            # Itterations per generations
            for _ in range(1):
                if crossovers[i] < self.crossover_rate:
                    # crossover occurs (Two parent solutions)
                    s_1 = selection(self.pop_fit)
                    s_2 = selection(self.pop_fit)

                    # Only share spacing DNA
                    if choice[i] <= 0.5:
                        self.pop_dis[s_1] = (self.pop_dis[s_1] + self.pop_dis[s_2] )/2
                    # Only share phase DNA
                    elif choice[i] > 0.5:
                        self.pop_pha[s_1] = (self.pop_pha[s_1] + self.pop_pha[s_2] )/2

                if mutations[i] < self.mutation_rate:
                    # mutation occurs
                    s_1 = selection(self.pop_fit)
                    # Only mutate spacing DNA
                    if choice[-i] <= 0.5:
                        rs = np.random.uniform(low = LOW, high=HIGH)
                        a = np.random.randint(low=0, high=self.N-1)
                        # SYMETRICAL SPACING
                        self.pop_dis[s_1][a] =  rs
                        #self.pop_dis[s_1][(self.N-2 - a)] =  rs
                    # Only mutate phase DNA
                    elif choice[-i] > 0.5:
                        self.pop_pha[s_1][np.random.randint(low=0, high=self.N)] =  np.random.uniform(low = 0, high=2*np.pi)
```