Hw 5 – Advanced Analytics and Metaheuristics

Student: Sai Teja Kanneganti

Question 1: Genetic Algorithm Implementation

(a)

i. Create code to generate chromosomes in the initial population.

1. Create Initial population:

```
def initializePopulation():   # n is size of population; d is dimensions of chromosome
  population = []
  populationFitness = []

  for i in range(populationSize):
     population.append(createChromosome(dimensions, lowerBound, upperBound))
```

- initializePopulation() function is called to initialize the population.
- An empty list population is initialised to store chromosomes.
- populationSize refers to number of chromosomes in a population.
- createChromosome(dimensions, lowerBound, upperBound) function is called and it creates a chromosome.
- Created chromosome is appended to the population.

2. Create Chromosome:

```
def createChromosome(d, lBnd, uBnd):
    x = []
    for i in range(d):
        x.append(myPRNG.uniform(lBnd, uBnd))  # creating a randomly located solution
    return x
```

- createChromosome has 3 parameters(d,lBnd,uBnd), d refers to dimension of the chromosome(number of values a chromosome can have), lBnd, uBnd are lower and upper bound of each values in a chromosome.
- Later for each of the dimensions, a value between lower bound and upper bound is appended to the chromosome. Thus, a chromosome with d dimensions is created.
- In a similar way, a population is initialised with no of chromosomes = population size.

ii. Create code to mutate chromosomes.

```
def mutate(x):
  rnd = myPRNG.random()
  if rnd < mutationRate:
     mut_gene = myPRNG.randint(0,dimensions-1)
     x[mut_gene] = myPRNG.uniform(lowerBound, upperBound)
  return x
```

- X is a chromosome that is to be mutated.
- Chromosomes in children population are subjected to mutate function.
- Mutate generates a random number between 0 and 1 and assign it to rnd,When the rnd is less than Mutation Rate then mutation occurs.
- We mutate the randomly selected dimension(mut_gene) with a random number between lower bound and upper bound and then generate a random integer between 0 to dimension-1(referring to dimension) and assign to mut_gene.

iii. Implement logic for crossover rate and mutation rate.

```
def breeding(matingPool):
  children = []
  childrenFitness = []
  for i in range(0, populationSize - 1, 2):
     prob = myPRNG.random()  # random number between 0 and 1
     if prob > crossOverRate:
        child1, child2 = matingPool[i], matingPool[i + 1]
     else:
        child1, child2 = crossover(matingPool[i], matingPool[i + 1])

     child1 = mutate(child1)
     child2 = mutate(child2)
     children.append(child1)
     children.append(child2)
     childrenFitness.append(evaluate(child1))
     childrenFitness.append(evaluate(child2))
  tempZip = zip(children, childrenFitness)
  childVals = sorted(tempZip, key=lambda tempZip: tempZip[1])
  return childVals
```

1. **Logic for Crossover rate:**

- Crossover rate is used in the breeding function. We take two chromosomes from the population for the crossover.

- We generate a random probability and if that probability is greater than crossover rate then crossover doesn't occur then these two chromosomes will be replicated in children population. If random probability is not greater than crossover rate then crossover occurs.

2. Logic for Mutation Rate:

```
def mutate(x):
  rnd = myPRNG.random()
  if rnd < mutationRate:
    mut_gene = myPRNG.randint(0,dimensions-1)
     x[mut_gene] = myPRNG.uniform(lowerBound, upperBound)
  return x
```

- Chromosomes in children population are subjected to mutate function.
- Mutate function generates a random number between 0 and 1 and assign it to rnd, When the rnd is less than Mutation Rate then mutation occurs.
- We mutate the randomly selected dimension(mut_gene) with a random number between lower bound and upper bound and then generate a random integer between 0 to dimension-1(referring to dimension) and assign to mut_gene.

iv. Implement some type of elitism in the insertion step.

```
def insert(pop, kids):
    BestPop = []
    combined = pop + kids
    combined = sorted(combined, key=lambda combined: combined[1])
    d = len(pop)
    for i in range(d):
        BestPop.append(combined[i])
    return BestPop
```

- In insert (commonly known as elitism), has both population and kids as parameters.
- We combine population and kids to a single array and sort that array based on their fitness values.
- We took the best chromosomes from the combined population and passed to the next generation.

v. Complete/modify any other logic as you see fit.

- **We add mutation function:**

```
def mutate(x):
  rnd = myPRNG.random()
  if rnd < mutationRate:
     mut_gene = myPRNG.randint(0,dimensions-1)
```

```
        x[mut_gene] = myPRNG.uniform(lowerBound, upperBound)
    return x
```

- **Insert function:**

```
def insert(pop, kids):
    BestPop = []
    combined = pop + kids
    combined = sorted(combined, key=lambda combined: combined[1])
    d = len(pop)
    for i in range(d):
        BestPop.append(combined[i])
    return BestPop
```

- **Used crossover rate in breeding function:**

```
def breeding(matingPool):
    children = []
    childrenFitness = []
    for i in range(0, populationSize - 1, 2):
        prob = myPRNG.random()  # random number between 0 and 1
        if prob > crossOverRate:
            child1, child2 = matingPool[i], matingPool[i + 1]
        else:
            child1, child2 = crossover(matingPool[i], matingPool[i + 1])
        child1 = mutate(child1)
        child2 = mutate(child2)
        children.append(child1)
        children.append(child2)
        childrenFitness.append(evaluate(child1))
        childrenFitness.append(evaluate(child2))
    tempZip = zip(children, childrenFitness)
    childVals = sorted(tempZip, key=lambda tempZip: tempZip[1])
    return childVals
```

- bestSolutionInPopulation

1. Added parameter j to the function.
2. Printed best chromosome, best fitness value for every generation (generation = j+1).

```
def bestSolutionInPopulation(pop,j):
    print("Best solution found after",j+1,"generations is:",pop[0])
    print("Best Fitness value after",j+1,"generations is:", pop[0][1])
```

(b) Empirically decide on parameters for population size, stopping criterion, crossover rate, mutation rate, selection, and elitism.

Population size:

In most of the cases, we observed that fitness value is better with increase in population size. With increase in population size it allows us to pick more chromosomes in the random space. So more space is explored, since population size refers to number of chromosomes in a given population. Here the stopping criteria is to get best fitness value irrespective of number of generations.

S.No	Dimensions	Population size	Number of generations	Crossover rate	Mutation rate	Fitness value
1	200	10	476	0.8	0.3	38,320
2	200	15	614	0.8	0.3	30,452
3	200	25	827	0.8	0.3	23,974
4	200	60	2198	0.8	0.3	2307
5	200	150	3570	0.8	0.3	59.38

Stopping criterion:

There can be 2 stopping criterion:

1. **Fixing number of generations:**

Here program terminates after repeating for specified number of generations and displays best chromosome found so far.

S.No	Dimensions	Population Size	Number of Generations	Crossover rate	Mutation rate	Fitness value
1	200	20	100	0.8	0.3	57,705
2	200	20	200	0.8	0.3	47,812
3	200	20	500	0.8	0.3	28,582
4	200	20	1000	0.8	0.3	22,178
5	200	20	2000	0.8	0.3	18,152

2. **Program terminates if there is no improvement in the fitness value:**

Our program terminates if there is no improvement in fitness value for next 1000 generations and prints generation number and best fitness value.

S.No	Dimensions	Population Size	Number of Generations	Crossover rate	Mutation rate	Fitness value
1	200	10	476	0.8	0.3	38,320
2	200	15	614	0.8	0.3	30,452
3	200	25	827	0.8	0.3	23,947
4	200	32	1386	0.8	0.3	10,265
5	200	60	2198	0.8	0.3	2307

Crossover rate:

- From the below function we can observe that more the crossover rate, more is the probability for crossover and less is the chance for duplicates. So, by increasing crossover rate we can reduce duplicates and explore more space.
- We can also observe that, solution converges quickly to better optimum value for higher crossover rate.

```
for i in range(0, populationSize - 1, 2):
    prob = myPRNG.random()  # random number between 0 and 1
    if prob > crossOverRate:
        child1, child2 = matingPool[i], matingPool[i + 1]
    else:
        child1, child2 = crossover(matingPool[i], matingPool[i + 1])
```

- Here the stopping criteria is to get best fitness value irrespective of number of generations.

S.No	Dimensions	Population size	Number of generations	Crossover rate	Mutation rate	Fitness value
1	200	150	3570	0.8	0.3	59.38
2	200	150	4992	0.9	0.3	6.36
3	200	150	6672	0.95	0.3	3.44
4	200	150	16,662	0.99	0.3	0.26

Mutation rate:

- Chromosomes in children population are subjected to mutate function.
- Mutate function generates a random number between 0 and 1 and assign it to rnd,When the rnd is less than Mutation Rate then mutation occurs.
- We mutate the randomly selected dimension(mut_gene) with a random number between lower bound and upper bound and then generate a random integer between 0 to dimension-1(referring to dimension) and assign to mut_gene.
- So if the mutation rate is too low,probability for mutation to occur is too low. So, I varied mutation rate and found better results when mutation rate is between 0.20 and 0.40.

```
def mutate(x):
  rnd = myPRNG.random()
  if rnd < mutationRate:
     mut_gene = myPRNG.randint(0,dimensions-1)
     x[mut_gene] = myPRNG.uniform(lowerBound, upperBound)
  return x
```

selection:

- Selection process used is tournament selection. In tournament selection k chromosomes are randomly selected from a population and best chromosome is selected based on evaluation function, and that chromosome is sent to the mating pool.Here I have used k=3.
- This process takes place till the chromosomes in mating pool equals to population size and the mating pool generated is used for breeding.

```
def tournamentSelection(pop, k):
    matingPool = []

    while len(matingPool) < populationSize:
    ids= [myPRNG.randint(0,populationSize-1) for i in range(k)] competingIndividuals=[pop[i][1] for i in
ids]
    bestID = ids[competingIndividuals.index(min(competingIndividuals))]
    matingPool.append(pop[bestID][0])

     return matingPool
```

Elitism:

```
def insert(pop, kids):
    BestPop = []
    combined = pop + kids
    combined = sorted(combined, key=lambda combined: combined[1])
    d = len(pop)
    for i in range(d):
```

```
        BestPop.append(combined[i])
    return BestPop
```

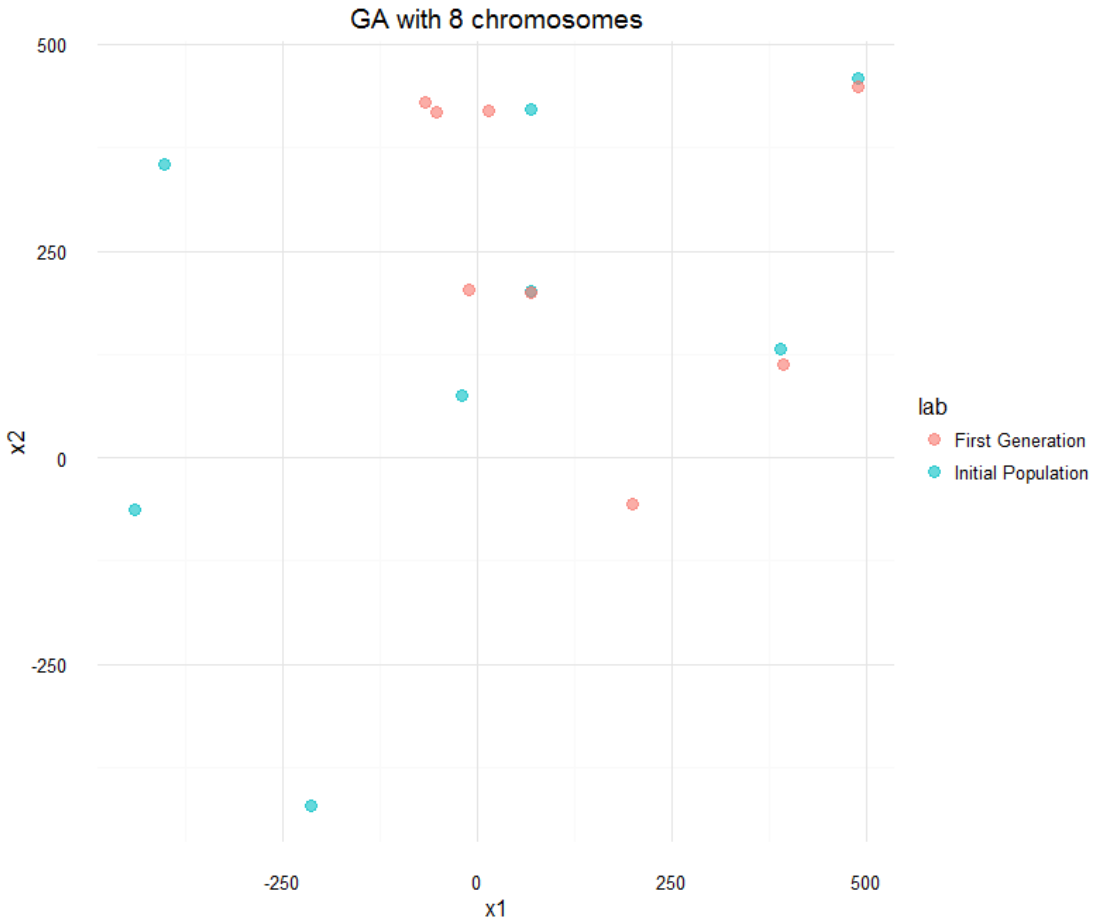
- In insert (commonly known as elitism), has both population and kids as parameters.
- We combine population and kids to a single array and sort that array based on their fitness values.
- We took the best chromosomes from the combined population and passed to the next generation.

(c) Solve the 2D Schwefel.

i. Create a small population of 8 chromosomes and depict their locations on a graph for the initial random set and the first generation.

Plot with following specifications:

- dimensions = 2
- Population Size = 8
- Crossover rate = 0.8
- Mutation rate = 0.3



ii. Solve the problem as best as possible and provide information on the quality of the best solution and the performance of the approach overall.

Best Solution:

Best fitness value found for 2 dimensional Schwefel function is 0.0027.
It has following properties:

- Fitness value: 0.0027
- Best Chromosome: [420.98032189447486, 421.1146422819745]
- Population:100
- No.of Generations to get best value:14
- Crossover rate:0.8
- Mutation rate:0.3

Information on the quality of the best solution and the performance of the approach overall:

- It is observed that better solutions are found with higher population size(population size close to 100). With increase in population size it allows us to pick more chromosomes in the random space. So more space is explored, since population size refers to number of chromosomes in a given population.
- We can also observe that the solution converges quickly with increase in population size.
- 2 dimensional Schwefel function converges quickly when compared to higher dimensional Schwefel function. Here we can observe that best solution is obtained in 14 generations.
- Selection process used is tournament selection. In tournament selection k chromosomes are randomly selected from a population and best chromosome is selected based on evaluation function, and that chromosome is sent to the mating pool.Here I have used k=3.This process takes place till the chromosomes in mating pool equals to population size and the mating pool generated is used for breeding.
- More the crossover rate, more is the probability for crossover and less is the chance for duplicates. So, by increasing crossover rate we can reduce duplicates and explore more space.
- If the mutation rate is too low, the probability for mutation to occur is too low. So, I varied mutation rate and found better results when mutation rate is between 0.20 and 0.40.

Other solutions:

- Here the stopping criteria is to get best fitness value irrespective of number of generations.

S.No	dimensions	Population Size	Number of Generations	Crossover rate	Mutation rate	Fitness value
1	2	100	14	0.8	0.3	0.0027

2	2	120	15	0.8	0.3	0.0027
3	2	150	10	0.8	0.3	0.0027
4	2	25	62	0.8	0.2	0.0057
5	2	20	23	0.8	0.2	0.2748
6	2	20	317	0.8	0.2	0.33
7	2	40	67	0.8	0.2	0.632
8	2	30	185	0.8	0.2	0.068
9	2	35	35	0.8	0.2	0.9

(d) Solve the 200D Schwefel problem as best as possible. Provide information on the quality of the solution and the performance of the approach overall.

Best Solution:

Best fitness value found for 200 dimensional Schwefel function is 0.26

It has following properties:

- Fitness value: 0.26
- Population:150
- No.of Generations to get best value:16,608
- Crossover rate:0.99
- Mutation rate:0.3

Information on the quality of the best solution and the performance of the approach overall:

- It is observed that better solutions are found with higher population size(population size close to 150). With increase in population size it allows us to pick more chromosomes in the random space. So more space is explored, since population size refers to number of chromosomes in a given population.
- We can also observe that the solution converges quickly with increase in population size.
- 200 dimensional Schwefel function takes much time to converge when compared to lower dimensional Schwefel function. Here we can observe that best solution is obtained in 16608 generations.
- Selection process used is tournament selection. In tournament selection k chromosomes are randomly selected from a population and best chromosome is selected based on evaluation function, and that chromosome is sent to the mating pool. This process takes place till the chromosomes in mating pool equals to population size and the mating pool generated is used for breeding.

- More the crossover rate, more is the probability for crossover and less is the chance for duplicates. So, by increasing crossover rate we can reduce duplicates and explore more space.
- If the mutation rate is too low,probability for mutation to occur is too low. So, I varied mutation rate and found better results when mutation rate is between 0.20 and 0.40.

Other Solutions:

- Here the stopping criteria is to get best fitness value irrespective of number of generations.

S.No	dimensions	Population size	Number of generations	Crossover rate	Mutation rate	Fitness value
1	200	150	16608	0.99	0.3	0.26
2	200	150	6672	0.95	0.3	3.44
3	200	150	4992	0.9	0.3	6.36
4	200	200	4434	0.9	0.3	11.06
5	200	150	3570	0.8	0.3	59.38
6	200	100	3497	0.8	0.3	96.67
7	200	60	2198	0.8	0.3	2307
8	200	45	1449	0.8	0.3	8658
9	200	32	1386	0.8	0.3	10,265
10	200	33	1308	0.8	0.3	11,097
11	200	38	1064	0.8	0.3	15,207
12	200	37	1404	0.8	0.3	10,534
13	200	36	1101	0.8	0.3	14,416
14	200	34	862	0.8	0.3	19,158
15	200	35	1037	0.8	0.3	15,599
16	200	40	435	0.8	0.3	32,453
17	200	30	766	0.8	0.3	21,908
18	200	25	827	0.8	0.3	23,974
19	200	20	571	0.8	0.3	26,440

20	200	15	614	0.8	0.3	30,452
21	200	10	476	0.8	0.3	26,440

Question 2: Particle Swarm Optimization Implementation

(a) We placed limits to the positions between -500 and 500, also limited the velocity to stay in the range of -1 to 1, and set as a stopping criteria the number of iterations.

(b) The first 3 positions and velocities for our 5 particles are the following:

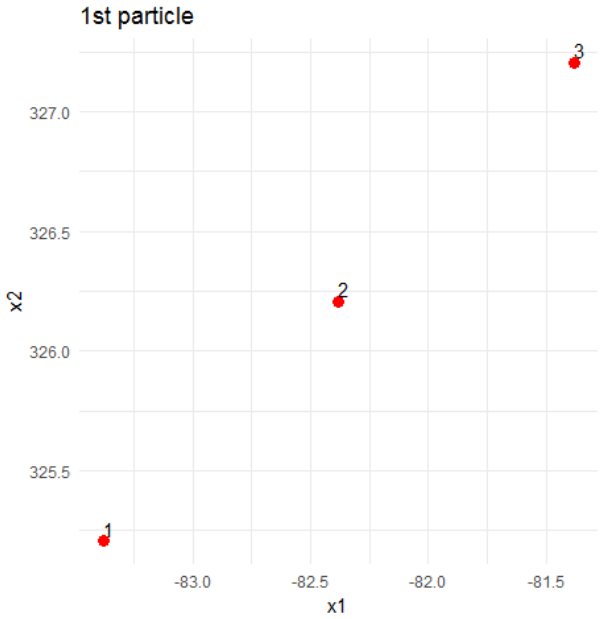
Position: for each particle we can see their positions evolving on each iteration. The global best position per iteration is shown in the right, and the particle position that is the current optimal for the iteration is highlighted in yellow.

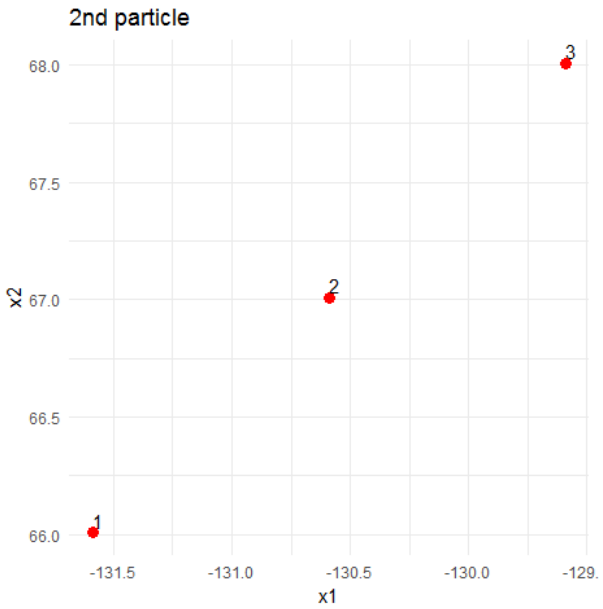
	Particle 1	Particle 2	Particle 3	Particle 4	Particle 5	Global best
Pos 1	[-83.38012745, 325.20650925]	[-131.58831051, 66.00816873]	[-375.73311572, 62.07848808]	[53.22108557, 458.0647851]	[478.63999557, 3.93536811]	[53.22108557, 458.0647851]
Pos 2	[-82.38012745, 326.20650925]	[-130.58831051, 67.00816873]	[-374.73311572, 63.07848808]	[52.93088834, 457.2473733]	[477.63999557, 4.93536811]	[52.93088834, 457.2473733]
Pos 3	[-81.38012745, 327.20650925]	[-129.58831051, 68.00816873]	[-373.73311572, 64.07848808]	[52.82055428, 456.9365903]	[476.63999557, 5.93536811]	[52.82055428, 456.9365903]

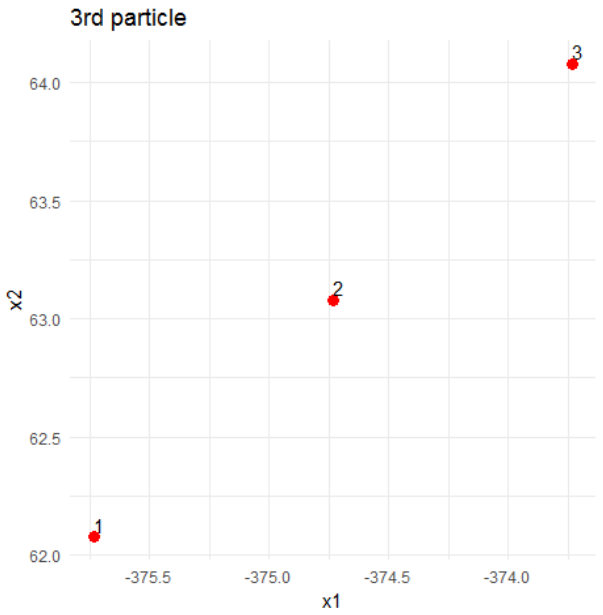
Velocity: for each particle we can see their velocities evolving on each iteration. Note that the velocity for particle 4 doesn't vary as much as for the other particles. This is because this particle is the optimal solution (for initial, iteration 1 and 2) or close to it (iteration 3), so the social component is very low.

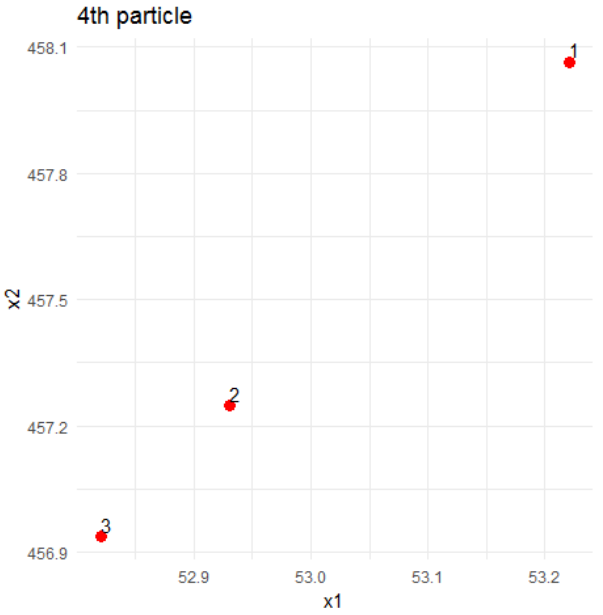
	Particle 1	Particle 2	Particle 3	Particle 4	Particle 5
Vel 1	[-0.97966166, -0.40272029]	[-0.6126773 , -0.67662435]	[-0.13412746, -0.65131288]	[-0.29019723, -0.8174118]	[-0.17576121, -0.70370766]
Vel 2	[1., 1.]	[1., 1.]	[1., 1.]	[-0.29019723, -0.8174118]	[-1., 1.]
Vel 3	[1., 1.]	[1., 1.]	[1., 1.]	[-0.11033406, -0.310783]	[-1., 1.]

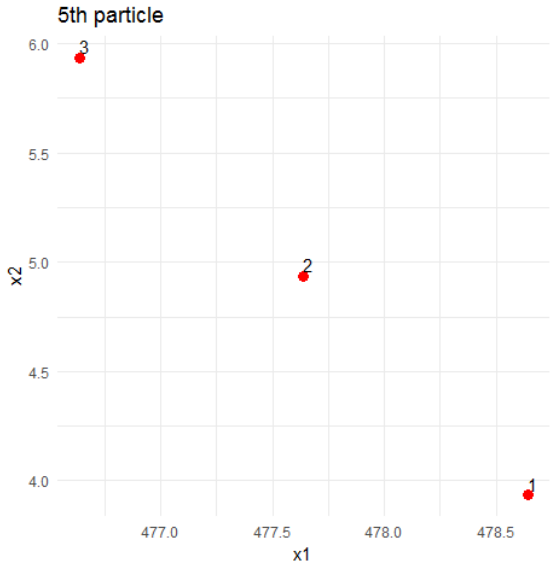
Following, we present several plots of the each of the particles in those 3 initial positions.


1st particle
327.0
326.5
326.0
325.5
x2
-83.0
-82.5
-82.0
-81.5
x1
1
2
3


2nd particle
68.0
67.5
67.0
66.5
66.0
x2
-131.5
-131.0
-130.5
-130.0
-129.
x1
1
2
3


3rd particle
64.0
63.5
63.0
62.5
62.0
x2
-375.5
-375.0
-374.5
-374.0
x1
1
2
3


4th particle
458.1
457.8
457.5
457.2
456.9
x2
52.9
53.0
53.1
53.2
x1
1
2
3



(c) We implemented a ring neighborhood in which has as a parameter the number of neighborhoods. The neighbors of a neighborhood are selected by their indexes, which will be a multiple of the number of neighborhoods. As an example, let's say the number of neighborhoods is 2, then will we have our swarm list like this (for 10 particles in the population):

0	1	2	3	4	5	6	7	8	9

Here the blue ones are one neighbor and the red ones are other neighbor. Then, one ring is 0-2-4-6-8 and other ring is 1-3-5-7-9. The indexes of each neighborhood can be described mathematically as this:

neighborhood 1: 2*i

neighborhood 2: 2*i + 1

Then, with these results we can get the neighbors of a particle with index '2*i'. Considering k = 2, the neighbors of the particle '2*i' are the particles in the indexes '2*(i-1)' and '2*(i+1)'. In general, this formula would contain 'num_neighborhoods*i'.

For example, if we have need the neighbors of particle 3, they will be particle 1 and particle 5. However, we have to check when we get the neighbors of extreme like 0 or 9. What we have to do is once the array is finished we start from the other extreme.

(d) Implementing and solving the Schwefel problem we got the following results:

Problem	PSO	GA
2 Dimensions	367.6384857879614	0.0027
200 Dimensions	79755.11909629653	0.26

From the above table, we can see that Genetic Algorithm gave the best result for both 2D and 200D.

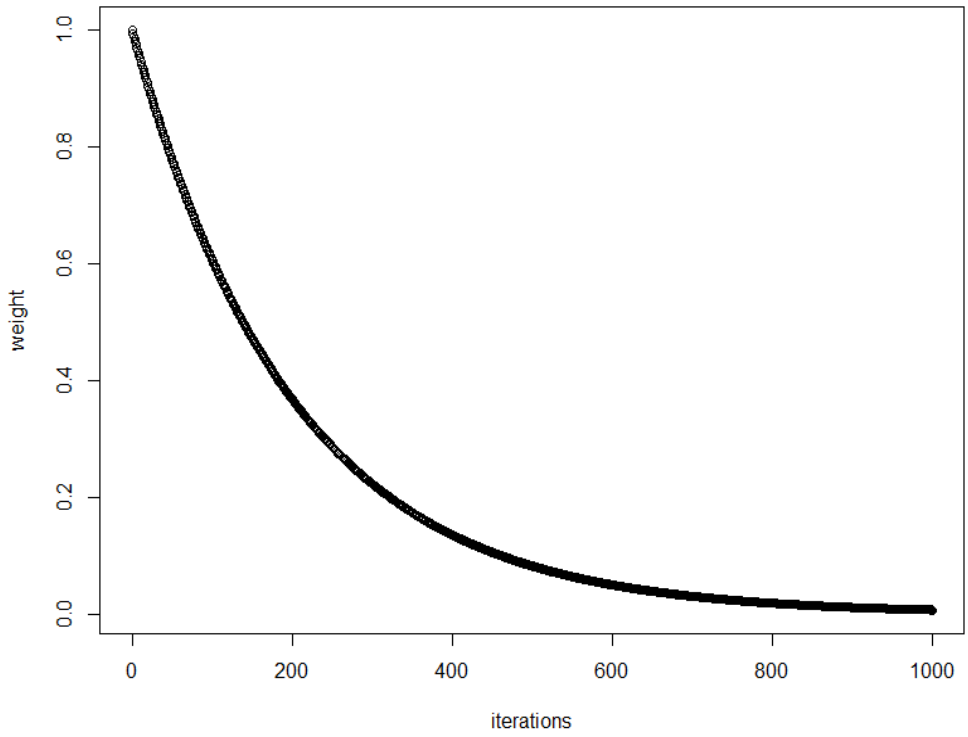
The tuned parameters were:

vel_max = 1
vel_min = -1
theta1 = 1.8 (for cognitive component)
theta2 = 0.5 (for social component)
Max number of iterations = 1000

And we used the weight function for the inertia:

```
1 / math.exp(0.005 * num_iter)
```

Which has the following behavior across the iterations:



Question 3: Guided Local Search

For guided local search, we create several functions that will be needed for guided local search. We create the **initial solution** function, as follows:

```
def initialSolution(dim):
  x = []
  for i in range(dim):
     x.append(myPRNG.uniform(posMin, posMax))
  return x
```

The function will generate initial solution for n dimension with random value between -500 and 500.

For **evaluation** function, we use the same function as in the PSO and GA.

```
def evaluate(x):
          val = 0
          d = len(x)
          for i in range(d):
            val = val + x[i] * math.sin(math.sqrt(abs(x[i])))
          val = 418.9829 * d - val
          return val
```

We create **neighborhood** function as follows:

```
def neighborhood(dim, x, N, lower, upper):
  neighbors = []
  prob = 0.3 # probability to decide whether add/subtract
  for i in range(N):
     neighbor = []
     for j in range(dim):
        if prob < myPRNG.random():
           # add random value
           pos = x[j] + myPRNG.uniform(lower,upper)
        else:
           # subtract initial solution with random value
           pos = x[j] - myPRNG.uniform(lower,upper)
        # set the solution to boundary if infeasible
        if pos < posMin:
           posVal = posMin
        elif pos > posMax:
           posVal = posMax
        else:
           posVal = pos
        neighbor.append(posVal)
     neighbors.append(neighbor[:])
  return neighbors
```

The neighborhood is created by add or subtract random value (with specified lowerbound and upperbound) to the initial solution based on random probability. If the solution is infeasible, we will use the lowerbound and upperbound of the domain (so that the solution is feasible).

The features that we used for GLS are as follows:

1. The euclidian distance between the solution and local optimum is less than 10.
2. The position contain 0

We create **indicator function for feature**, the value will be 1 if the solution has the feature, and 0 otherwise. The function is as follow:

```
def features(x, s):
        nFeatures = 2 #number of features
        # euclidian distance
        eD = scipy.spatial.distance.euclidean(x, s)
        # calculate indicator result for feature 1
        if eD < 10 : f1 = 1
        else: f1 = 0
        # calculate indicator result for feature 2
        if 0 in s: f2 = 1
        else: f2 = 0
        feature = [f1,f2]
        return feature
```

We create function to **initialize penalty value** as follows:

```
def initialP(nFeature):
        p = []
        for i in range(nFeature):
        p.append(0)
        return p
```

The initial value for each penalty will be 0.

We create function to calculate the value of **lambda**, as follows:

```
def getLambda(x,f):
        alpha = 0.5
        lmbda = alpha * (evaluate(x)/len(f))
        return lmbda
```

GLS use different evaluation function, we create function for **extended move evaluation** as follows:

```
def evaluateGLS(s,f,p):
        value = 0
        aF = np.array(f)
        aP = np.array(p)
        fp = np.multiply(aF,aP)
        value = evaluate(s) + (getLambda(s,f)*(sum(fp)))
        return value
```

We also create function to calculate **utility**, as follows:

```
def utility(x,f,p):
        util = f * (evaluate(x)/(1+p))
        return util
```

Then, we create a function for **gls** that utilize all the previous functions that we created.

```
def gls(dim,steps,N,lower,upper):
    itermax = steps
    iter = 0
    nFeature = 2
    x_curr = initialSolution(dim) # initial solution
    x_best = x_curr[:]
    xBest = x_best[:]
    p = initialP(nFeature) # initial penalty for feature
    while iter < itermax:
        Neighborhood = neighborhood(dim, x_curr, N, lower, upper) # neighborhood for x_curr
        for s in Neighborhood:
            feat_best = features(x_curr, x_best) # feature for x_best
            f_best = evaluateGLS(x_best, feat_best, p) # extended move evaluation for x_best
            featS = features(x_curr,s) # feature for solution s
            f_s = evaluateGLS(s,featS,p) # extended move evaluation for solution s
            if f_s < f_best:
                x_best = s
        sStar = x_best
        if evaluateGLS(sStar,features(x_curr,sStar),p) < evaluateGLS(x_curr,features(x_curr,x_curr),p):
            x_curr = sStar
            if(evaluate(x_curr)<evaluate(xBest)):
                xBest = x_curr
        else:
            util = []
            for i in range(nFeature):
                u = utility(x_curr,features(x_curr,x_curr)[i],p[i])
                util.append(u)
            index, value = max(enumerate(util), key=operator.itemgetter(1)) # get the feature that has the
maximum utility
            p[index] = p[index] + 1
            xBest = x_curr
            fBest = evaluate(xBest)
        iter = iter + 1
    return xBest, fBest
```

To call the gls function, we need to specify several parameters as follows:

dim(dimension, from 2 to 200);
steps(number of iteration);
N(neighborhood size);
lower(lowerbound value to generate random value to add/substract to the value of initial solution);
upper(upperbound value to generate random value to add/substract to the value of initial solution)

GLS Result

The best result that we got from GLS are as follows:

Dimension	Iteration	Neighborhood Size	lowerbound	upperbound	Fitness value
2	500	500	-500	500	0.009096686
200	10	50000	-500	500	71687.90495

Result Comparison of Genetic Algorithm, Particle Swarm Optimization, and Guided Local Search

After implementation of the three methods, we get the best result for each of the method as follow:

	Fitness value for each method		
Dimension	GA	PSO	GLS
2D	0.0027	367.6384858	0.00909669
200D	0.26	79755.1191	71687.9049

From the above table, we can see that Genetic Algorithm gave the best result for 2D and 200D.