Cory Mckiel

CS 4320 Evolutionary Computation

2/18/20

# Project 1

Notes:

The range of f(x,y,x) = x^2 + y^2 + z^2 over the domain [-1,5] is [0,75]. Because this is a minimization problem, I would like to reward Individuals that result in a value closer to the range's minimum. To accomplish this, considering the normal sampling process assigns a higher percent of selection to evaluations of higher values, I reverse the mapping of function results for a given Individual. Let x be the result of plugging Individual i into f(x,y,z), my evaluation method returns 75-x. For a perfect value of x=0, eval() returns 75, the functions max value over the domain. This makes sampling and selection easier while only adding one line of code into eval(). All numerical results in this paper assume 75 is the maximum fitness possible for an Individual, 0 the lowest.

The average population fitness of the initially random population for my algorithm is on average 50 for 30 runs of the program. After generation 10 this average increases to 64. For the remaining intervals of 10 this average hovers around 64. I expected a continual increase in average population fitness for each interval of 10 generations. Knowing that the average for the completely random initial populations was 50, we do see that the algorithm is performing better that randomness, but just doesn't improve past a certain threshold. I suspect that a crossover rate of 80% and a mutation rate of 10% are too high and that the best solutions do not often survive unaltered from generation to generation.

## Code: (EvoAlg.java, Population.java, Individual.java)

## EvoAlg.java:

```
//***********************************************************************************
*************
//Author: Cory Mckiel
//Date Created: Feb 15, 2020
//Last Modified: Feb 15, 2020
//Associated Files: Population.java Individual.java
//IDE: IntelliJ
//Java SDK: 12
//Program Description: This is an evolutionary algorithm designed to solve analytic
functions.
//It is easily configurable for multiple population sizes, individual sizes, crossover
```

```
and
//mutation rates, etc. With changes to specific functions different problems can be
tackled.
//This program is meant be a modular and easily configurable platform for evolutionary
//algorithms.
//Current Function: f(x,y,z) = x^2 + y^2 + z^2
//Current Objective: Minimize
//*************************************************************************************
*************

import java.util.Random;

public class EvoAlg {
    public static void main(String [] args) {
        //*******PARAMETERS*******//
        //Change this to alter the number of individuals.
        //Only works for even numbers.
        int pop_size = 30;
        //Change this to alter the number of genes in each individual.
        //Changing this without altering other code is dangerous.
        int genome_size = 3;
        //Select the domain here.
        double [] domain = {-1, 5};
        //Change this seed each time you'd like to produce a different
        //run. Record this seed for reproducibility of results.
        long seed = 303631435;
        //Change to alter the probability of crossover
        double p_c = 0.8;
        //Change this to alter mutation rate
        double p_m = 0.1;
        //The number of generations
        int generations = 51;
        //The Random used by the entire program.
        Random r = new Random(seed);
        //Used to keep track of the best of run solution
        //across all generations.
        Individual bor = new Individual(genome_size);

        //***MAIN EVOLUTIONARY
ALGORITHM*****************************************************************
        int t = 0;
        //Create the initial population.
        Population pop = new Population(pop_size, genome_size, domain[0], domain[1],
r.nextLong());
        //Evaluate the initial population
        eval(pop);
        //Sample the initial population
        sample(pop);
        //While we still have generations to complete.
        while (t < generations) {
            //Create a new population to temporarily store selected individuals.
            Population next_pop = new Population(pop_size);
            //Fill up this new population. Two individuals are selected at a time
            //and undergo crossover and mutation. These two individuals are placed
            //in the first half and second half of the new population.
            //Therefore only need to iterate half the size of the population.
            for (int i = 0; i < pop_size/2; i++) {
                //Create a copy of the current population to pass into the
                //select, crossover and mutate methods because they corrupt the
                //original population because Java is pass by reference.
                Population pop_clone = new Population(pop);
                //Create an array of individuals to store results of selection,
crossover, and mutation.
```

```
                Individual [] individuals;
                individuals = mutate(crossover(select(pop_clone, r), select(pop_clone,
r), r, p_c), r, p_m, domain[0], domain[1]);
                //Add the first individual to the first half of the population.
                next_pop.setIndividual(new Individual(individuals[0]), i);
                //Add the second individual to the second half of the population.
                next_pop.setIndividual(new Individual(individuals[1]),
(pop_size/2)+i);
            }
            //The new population is complete, assign it as the previous generation
            //to move to the new gen.
            pop = new Population(next_pop);
            //Evaluate the new generation.
            eval(pop);
            //Sample the new generation.
            //F is total pop fitness.
            double F = sample(pop);
            //Get the fittest individual from the generation.
            Individual gen_fittest = new Individual(getFittest(pop));
            //Get the weakest individual from the generation.
            Individual gen_weakest = new Individual(getWeakest(pop));
            //Compare it to the best of run.
            if (bor.getFitness() < gen_fittest.getFitness()) {
                bor = new Individual(gen_fittest);
            }
            //At intervals of 10, print the total gen fitness.
            if (t % 10 == 0) {
                System.out.println("GENERATION " + t);
                System.out.println("Total fitness: " + F);
                System.out.println("Fittest Individual: ");
                gen_fittest.print();
                System.out.println("Weakest Individual: ");
                gen_weakest.print();
                System.out.println("Average Fittnes: " + F/pop_size);
                System.out.println();
            }
            //Increment the generation number.
            t += 1;
        }
        //When everything is done print the resulting solution.
        //pop.print();
        //double bor = getFittest(pop);
        //System.out.println("The BOR is: " + bor);
        System.out.println();
        bor.print();
    }

    //The eval function assigns a fitness value to each individual in the
    //Population. This value is based off the function f(x,y,z)= x^2 + y^2 + z^2.
    //In our case the Domain of the function is [-1,5] making the range [0,75].
    //Because we are minimizing we would like to reward the smallest values.
    //During sampling the highest fitness number is rewarded. To fix this
    //contradiction we remap small values to high ones and vice versa
    //by subtracting the calculated function value from the ranges max
    //value. Small function values result in very little being subtracted
    //from the max range, 75, which means that the fitness value is very
    //high for small function values.
    //Ex:
    //  function value = 0.05 (This is a good value we would like to reward)
    //  fitness value = 75 - 0.05 = 74.95 (This is a very high value that will
    //                                      be rewarded in sampling)
    private static void eval(Population population) {
        for (int i = 0; i < population.getPop_size(); i++) {
            Individual individual = population.getIndividual(i);
            double x_sq = Math.pow(individual.getGene(0), 2);
            double y_sq = Math.pow(individual.getGene(1), 2);
            double z_sq = Math.pow(individual.getGene(2), 2);
            double function_value = x_sq + y_sq + z_sq;
            double fitness_value = 75 - function_value;
            individual.setFitness(fitness_value);
        }
    }

    //The sample method generates the statistics for correct
    //population selection.
    private static double sample(Population population) {
        //Total population fitness
        double F = 0;
        //Rolling prob accumulates the probabilities
        //for insertion into pop_cumulative_prob in
        //the Population class.
        double rolling_prob = 0;
        //Calculate the total population fitness.
        for (int i = 0; i < population.getPop_size(); i++) {
            F = F + population.getIndividual(i).getFitness();
        }
        //System.out.println("Pop Fit: " + F + "");
        //Calculate each individuals probability of selection.
        for (int i = 0; i < population.getPop_size(); i++) {
            Individual individual = population.getIndividual(i);
            double prob_selection = individual.getFitness() / F;
            individual.setProb_selection(prob_selection);
        }
        //Calculate the cumulative probability table directly
        //used for selecting individuals.
        for (int i = 0; i < population.getPop_size(); i++) {
            rolling_prob = rolling_prob +
population.getIndividual(i).getProb_selection();
            population.setPop_cumulative_probs(rolling_prob, i);
        }
        //Ensure the last entry is always 1.0, that way something will
        //always be selected.
        population.setPop_cumulative_probs(1.0, population.getPop_size()-1);

        //Return the total population fitness so
        //it can be measured by caller to track
        //Population fitness over generations.
        return F;
    }

    //select takes in the population and the random to make a selection
    //of a new individual fro the population.
    private static Individual select(Population population, Random random) {
        double r = random.nextDouble();
        //System.out.println("select rand: " + r);
        //Loop through the cumulative probability table and return
        //the first individual associated with the index of the table
        //that has a probability of selection greater than r.
        for (int i = 0; i < population.getPop_size(); i++) {
            double prob_select = population.getPop_cumulative_probs(i);
            if (r < prob_select) {
                return population.getIndividual(i);
            }
        }
        //Don't return anything if nothing was selected.
        //If we reach here something went wrong.
        System.out.println("Failed to select an Individual.");
        System.exit(-1);
        return new Individual(3);
    }

    //crossover takes in two individuals and using random and probability of
crossover(p_c)
    //randomly crosses the genes between each other. There can either be a head swap
or a
    //tail swap in the current configuration of 3 genes.
    //ex headswap:
    // i1: <1 2 3> i2: <4 5 6> --> i1: <4 2 3> i2: <1 5 6>
    private static Individual[] crossover(Individual i1, Individual i2, Random random,
double p_c) {
        double r = random.nextDouble();
        //System.out.println("Crossover rand: " + r);
        Individual [] individuals = new Individual[2];
        if (r < p_c) {
            //Crossover occurs
            //Pick the point as 0 or 1.
            int cross_pt = random.nextInt(2);
            if (cross_pt == 0) {
                //Swap the head
                double temp0 = i1.getGene(0);
                i1.setGene(i2.getGene(0), 0);
                i2.setGene(temp0, 0);
            }
            if (cross_pt == 1) {
                //Swap the tail
                double temp2 = i1.getGene(2);
                i1.setGene(i2.getGene(2), 2);
                i2.setGene(temp2, 2);
            }
        }
        //Pack it up and send it home.
        individuals[0] = i1;
        individuals[1] = i2;
        return individuals;
    }

    //mutate takes two individuals at a time. It is designed to have the output of
crossover
    //be its input. It also takes in the random. If it made it's own random then there
wouldn't be
    //reproducible results. It takes in the probability of mutation and the bounds for
the genes.
    //If a mutate is triggered, mutate will replace the gene with a random value over
the domain.
    //It returns the same number of Individuals that it takes in.
    private static Individual[] mutate(Individual[] individuals, Random random, double
p_m, double gene_lower_bound, double gene_upper_bound) {
        double r;
        //Loop through the individuals
        for (int i = 0; i < individuals.length; i++) {
            //Loop through an individual's genes
            for (int j = 0; j < individuals[i].getGenome_size(); j++) {
                r = random.nextDouble();
                //System.out.println("mutation rand: " + r);
                if (r < p_m) {
                    //Mutation occurs
                    double gene_value = gene_lower_bound + (gene_upper_bound -
gene_lower_bound) * random.nextDouble();
                    //Assign a random value from domain to gene.
                    individuals[i].setGene(gene_value, j);
                }
            }
        }
        //Pack it up and send it home.
        Individual[] ind = {individuals[0], individuals[1]};
        return ind;
    }

    public static Individual getFittest(Population population) {
        double max = -1;
        Individual individual = new
Individual(population.getIndividual(0).getGenome_size());
        for (int i = 0; i < population.getPop_size(); i++) {
            if (max < population.getIndividual(i).getFitness()) {
                max = population.getIndividual(i).getFitness();
                individual = new Individual(population.getIndividual(i));
            }
        }
        return individual;
    }

    public static Individual getWeakest(Population population) {
        double min = 100;
        Individual individual = new
Individual(population.getIndividual(0).getGenome_size());
        for (int i = 0; i < population.getPop_size(); i++) {
            if (min > population.getIndividual(i).getFitness()) {
                min = population.getIndividual(i).getFitness();
                individual = new Individual(population.getIndividual(i));
            }
        }
        return individual;
    }
}
```

**Population.java:**

```
//*************************************************************************************
************
//Author: Cory Mckiel
//Date Created: Feb 15, 2020
//Last Modified: Feb 15, 2020
//Associated Files: EvoAlg.java Individual.java
//IDE: IntelliJ
//Java SDK: 12
//Program Description: This is an evolutionary algorithm designed to solve analytic
functions.
//It is easily configurable for multiple population sizes, individual sizes, crossover
and
//mutation rates, etc. With changes to specific functions different problems can be
tackled.
//This program is meant be a modular and easily configurable platform for evolutionary
//algorithms.
//*************************************************************************************
************

import java.util.Random;

public class Population {
    //The population is represented by an array of Individuals.
    private Individual [] population;
    //The cumulative probability array will be set during
    //Sampling the population. It is used to select individuals
    //for the next population.
    private double [] pop_cumulative_probs;
    //The number of Individuals in the population.
    private int pop_size;
    //This seed is used for the reproduction of results,
    //Using the same seed results in the exact same initial population.
    private long pop_seed;

    //Use this Constructor to make a blank population.
    Population(int pop_size) {
        this.pop_size = pop_size;
        population = new Individual[this.pop_size];
        pop_cumulative_probs = new double[this.pop_size];

        pop_seed = 0;
    }

    //Constructor for copying Population object
    Population(Population population) {
        pop_size = population.getPop_size();
        this.population = new Individual[pop_size];
        pop_cumulative_probs = new double[pop_size];
        pop_seed = population.getPop_seed();

        for (int i = 0; i < pop_size; i++) {
            this.population[i] = new Individual(population.getIndividual(i));
            pop_cumulative_probs[i] = population.getPop_cumulative_probs(i);
        }
    }

    //Constructor. Takes in the population size, Genome size, the bounds for the
allowable gene values,
    //and the seed used for 'random' generation of the population.
    Population(int pop_size, int genome_size, double gene_lower_bound, double
gene_upper_bound, long pop_seed) {
        this.pop_size = pop_size;
        this.pop_seed = pop_seed;
        population = new Individual[this.pop_size];
        pop_cumulative_probs = new double[this.pop_size];

        //Initialize the Individuals in the population.
        for (int i = 0; i < this.pop_size; i++) {
            population[i] = new Individual(genome_size);
        }
        //Call this init function to 'randomly' generate values
        //for each gene and each individual within the population.
        init(genome_size, gene_lower_bound, gene_upper_bound);
    }

    //init takes the gene size and the allowable gene value bounds. It then 'randomly'
    //selects allowable values for each gene in each individual to serve as the
initial
    //population.
    private void init(int genome_size, double gene_lower_bound, double
gene_upper_bound) {
        Random r = new Random(pop_seed);
        for (int i = 0; i < pop_size; i++) {
            for (int j = 0; j < genome_size; j++) {
                //Generate double within the bounds.
                double gene_value = gene_lower_bound + (gene_upper_bound -
gene_lower_bound) * r.nextDouble();
                population[i].setGene(gene_value, j);
            }
        }
    }

    //***GETTERS AND SETTERS***//
    public  void setIndividual(Individual individual, int i) {
        if (i >= pop_size || i < 0) {
            System.out.println("Invalid Population index in 'setIndividual' from
caller");
            System.exit(-1);
        }
        population[i] = individual;
    }

    public Individual getIndividual(int i) {
        if (i >= pop_size || i < 0) {
            System.out.println("Invalid Population index from caller.");
            System.exit(-1);
        }
        return population[i];
    }

    public void setPop_cumulative_probs(double prob, int i) {
        if (i >= pop_size || i < 0) {
            System.out.println("Invalid cumulative_prob index from caller.");
            System.exit(-1);
        }
        pop_cumulative_probs[i] = prob;
    }

    public double getPop_cumulative_probs(int i) {
        if (i >= pop_size || i < 0) {
            System.out.println("Invalid cumulative_prob index from caller.");
            System.exit(-1);
        }
        return pop_cumulative_probs[i];
    }

    public int getPop_size() {
        return pop_size;
    }

    public long getPop_seed() {
        return pop_seed;
    }

    //This method prints out an entire population of individuals.
    public void print() {
        for (int i = 0; i < pop_size; i++) {
            System.out.println("\nPopulation member: " + i);
            population[i].print();
        }
        /*System.out.println("\n\nCumulative Probs:");
        for (int i = 0; i < pop_size; i++) {
            System.out.println(pop_cumulative_probs[i]);
        }*/
    }
}
```

## Individual.java:

```
//*************************************************************************************
************
//Author: Cory Mckiel
//Date Created: Feb 15, 2020
//Last Modified: Feb 15, 2020
//Associated Files: Population.java EvoAlg.java
//IDE: IntelliJ
//Java SDK: 12
//Program Description: This is an evolutionary algorithm designed to solve analytic
functions.
//It is easily configurable for multiple population sizes, individual sizes, crossover
and
//mutation rates, etc. With changes to specific functions different problems can be
tackled.
//This program is meant be a modular and easily configurable platform for evolutionary
//algorithms.
//*************************************************************************************
************

public class Individual {
    //The genome is the entire vector of genes.
    //Each genome represents a potential solution to the problem.
    private double [] genome;
    //The fitness is a measure of how well the genome solves
    //the problem. It is calculated by the eval() function.
    private double fitness;
    //The probability of selection is a ratio of this
    //individuals fitness compared to the total
    //population fitness. Higher values tend to be
    //selected more often because they are better
    //solutions.
    private double prob_selection;
    //The genome_size is the number of genes it
    //takes to form a solution to a function.
    //eg. f(x,y,z) needs 3 genes.
    private int genome_size;

    //Constructor
    Individual(int size) {
        genome_size = size;
        genome = new double[genome_size];
        fitness = 0;
        prob_selection = 0;
    }

    //Constructor for copying individuals
    Individual(Individual individual) {
        genome_size = individual.getGenome_size();
        genome = new double[genome_size];
        fitness = individual.getFitness();
        prob_selection = individual.getProb_selection();
        for (int i = 0; i < genome_size; i++) {
            genome[i] = individual.getGene(i);
        }
    }

    //*****GETTERS AND SETTERS*****//
    public double[] getGenome() {
        return genome;
    }
```

```java
    public double getGene(int index) {
        //Bounds checking
        if (index < genome_size && index >= 0) {
            return genome[index];
        }
        else {
            return -1;
        }
    }

    public void setGenome(double[] genome) {
        this.genome = genome;
    }

    public void setGene(double value, int index) {
        if (index < genome_size && index >= 0) {
            genome[index] = value;
        }
    }

    public double getFitness() {
        return fitness;
    }

    public void setFitness(double fitness) {
        this.fitness = fitness;
    }

    public double getProb_selection() {
        return prob_selection;
    }

    public void setProb_selection(double prob_selection) {
        this.prob_selection = prob_selection;
    }

    public int getGenome_size() {
        return genome_size;
    }

    //This method prints out all information associated
    //with an individual.
    public void print() {
        System.out.print("Ind: <");
        for (int i = 0; i < genome_size; i++) {
            System.out.print(genome[i] + ", ");
        }
        System.out.println(">");
        System.out.println("Fitness: " + fitness);
        System.out.println("Probability of Selection: " + prob_selection);
    }
}
```

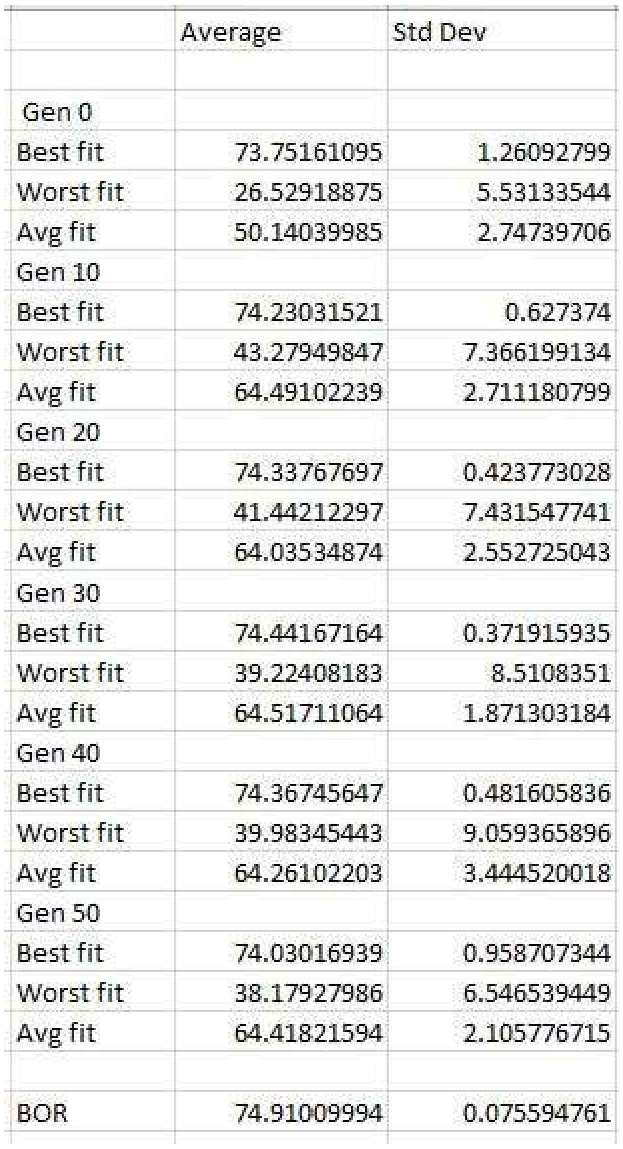
**Example Output:**

```
GENERATION 0

Total fitness: 1734.8413640394922

Fittest Individual:

Ind: <0.13179322931166593, -0.228446419798211, -0.48483134925752935, >

Fitness: 74.6953813407661

Probability of Selection: 0.04305602972645384

Weakest Individual:

Ind: <2.9530892300979525, 4.1017459124070825, 3.8558411602725, >

Fitness: 34.587433415879694

Probability of Selection: 0.019936943015553057

Average Fittnes: 57.828045467983074


GENERATION 10

Total fitness: 2021.9015404945324

Fittest Individual:

Ind: <0.3182795160502869, 0.23339664011868888, 0.7629585792000444, >

Fitness: 74.26211836446915

Probability of Selection: 0.036728849984606834

Weakest Individual:

Ind: <4.8973888921417705, 4.412114214112885, -0.8240123986783374, >

Fitness: 30.869833767573816

Probability of Selection: 0.015267723550981336

Average Fittnes: 67.39671801648441


GENERATION 20

Total fitness: 1808.6756327329742
```

```
Fittest Individual:
Ind: <-0.5218357316516755, -0.16992518852982674, -0.8240123986783374, >
Fitness: 74.01981646629903
Probability of Selection: 0.04092487073232275
Weakest Individual:
Ind: <1.0061731251186972, 3.7246647850781187, 4.602297283251788, >
Fitness: 38.933347597661054
Probability of Selection: 0.021525887170178414
Average Fittnes: 60.28918775776581

GENERATION 30
Total fitness: 1866.0153667694822
Fittest Individual:
Ind: <-0.20989039514207342, -0.01335433949066589, -0.5174118266161742, >
Fitness: 74.68805268532158
Probability of Selection: 0.04002542209211515
Weakest Individual:
Ind: <3.0392932780412405, 4.272961117413624, 4.6070290359596395, >
Fitness: 26.27978312094944
Probability of Selection: 0.014083369080955649
Average Fittnes: 62.200512225649405

GENERATION 40
Total fitness: 1708.6530246337657
Fittest Individual:
Ind: <0.30823063930938877, -0.01335433949066589, 0.15448427120249097, >
Fitness: 74.88095014455872
```

```
Probability of Selection: 0.043824550136859285

Weakest Individual:

Ind: <4.5695689421713785, 3.1089978094429833, 1.987075828373968, >

Fitness: 40.50470195591339

Probability of Selection: 0.02370563325142927

Average Fittnes: 56.95510082112552


GENERATION 50

Total fitness: 1967.0015201987874

Fittest Individual:

Ind: <0.02533149032329529, 0.41838957720906333, 0.7837669602761659, >

Fitness: 74.21001782926028

Probability of Selection: 0.03772748371936212

Weakest Individual:

Ind: <-0.7568610152438682, 4.192698149479641, 3.075385516698357, >

Fitness: 47.390447554636

Probability of Selection: 0.024092735601875217

Average Fittnes: 65.56671733995958



Ind: <-0.20989039514207342, -0.01335433949066589, -0.18628241264319012, >

Fitness: 74.9210665463837

Probability of Selection: 0.039887876661411546


Process finished with exit code 0
```

## Data Collected for 30 Runs:



| | Average | Std Dev |
|---|---|---|
| | | |
| Gen 0 | | |
| Best fit | 73.75161095 | 1.26092799 |
| Worst fit | 26.52918875 | 5.53133544 |
| Avg fit | 50.14039985 | 2.74739706 |
| Gen 10 | | |
| Best fit | 74.23031521 | 0.627374 |
| Worst fit | 43.27949847 | 7.366199134 |
| Avg fit | 64.49102239 | 2.711180799 |
| Gen 20 | | |
| Best fit | 74.33767697 | 0.423773028 |
| Worst fit | 41.44212297 | 7.431547741 |
| Avg fit | 64.03534874 | 2.552725043 |
| Gen 30 | | |
| Best fit | 74.44167164 | 0.371915935 |
| Worst fit | 39.22408183 | 8.5108351 |
| Avg fit | 64.51711064 | 1.871303184 |
| Gen 40 | | |
| Best fit | 74.36745647 | 0.481605836 |
| Worst fit | 39.98345443 | 9.059365896 |
| Avg fit | 64.26102203 | 3.444520018 |
| Gen 50 | | |
| Best fit | 74.03016939 | 0.958707344 |
| Worst fit | 38.17927986 | 6.546539449 |
| Avg fit | 64.41821594 | 2.105776715 |
| | | |
| BOR | 74.91009994 | 0.075594761 |

Where an ideal value looks like 74.9999

All seeds used to determine these averages have been saved and are available upon request.