MSc Robotics, Autonomous and Interactive Systems

Bio-Inspired Computation MSc Coursework 1

Multilayer Perceptron & Black-Box Optimisation

Camille SALLABERRY & Artem LUKIANOV

14 November 2016

Heriot-Watt University

Chapter 1

The optimisation of a MultiLayer Perceptron (MLP)

1.1 Running a MultiLayer Perceptron

1.1.1 Elaborate a MultiLayer Perceptron model

Due to the fact that there are a lot of useful tools and functions in Matlab both of group member agreed to use it for this coursework.

To elaborate an MLP model for each chromosome, the cell structure proposed by Matlab was used. All of the MLP is defined in one cell named ”Population” where each line represents one chromosome and includes the index of this chromosome, a cell of n-weight matrices, the approximation results of the function for each pair and the average fitness value.

- During the process the number of inputs and outputs stayed the same: two inputs and one output.
- The index of the chromosome is used to rank the population depending of their fitness value.
- Inputs are created randomly between -1 and 1. The function chosen for approximation is the sphere function : $a^2 + b^2$
- The weight matrices are created randomly using Matlab function ”Weight”. Each matrix contains the value of the weights for each input neuron.
- To calculate the fitness of each chromosome, first of all fitness for each pair needs to be calculated, then an average of those values can be found.

1.1.2 Genetic Algorithm (GA) : Mutation & Crossover

To optimise the MLP, it was decided to develop two types of mutation and one type of crossover.

- The first created GA was a mutation involving replacing some values of the weight matrices with new random values. This function is called ”MutationRandom” and is used to mutate the half of the population with the worst fitness value.
- The second type of mutation was based on swapping a column of a weight matrix with another column of the same matrix. Similar to the previous step only half of the population with the worst fitness has been used for mutation.
- For the Cross-mutation function, the idea was to create a crossover of the population by selecting the best individuals and use them to produce children. At first, it did not perform good enough and clones could be produced. After adding mutation to children the results became much better.

For each of the mutations, the fitness value is between 0.11 and 0.2 depending on the inputs but for the cross-mutation, the minimum value is around 0.3. After testing those three algorithms separately, it was decided to use the mutation in general and sometimes the cross-mutation.

1.2 Optimisation of the MLP

1.2.1 Working process

The working principle of the code is the following :
At the beginning:

- Generate randomly two values of input for each pair input/output
- For each chromosome, create randomly n-weight matrices
- Run the MLP and get a first approximation of the desired function & a fitness value for each chromosome.
- Rank the population depending of their fitness (from the best to the worst)

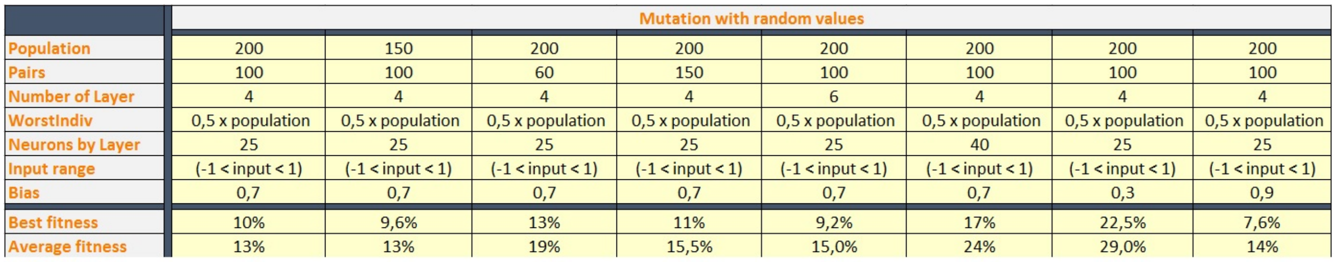
For each generation:

- Create a sub-population made from the worst or the best individuals depending on the GA used.
- If it is a mutation, randomly pick a column in a weight matrix and replace it with a new column or swap two column of a same matrix. If it is a cross-mutation, create children by using the weight matrices of two parents from the best part of the population.
- Run the MLP for the new population (children or mutate population) and get them a new fitness value.
- Compare the worst actual population with the new one and keep the best of them.
- Rank the population depending of their fitness (from the best to the worst)

In order to test the different combinations of GA, parameters of a the acceptable MLP have been used and improved throughout the process. The number of generation was set to 300.

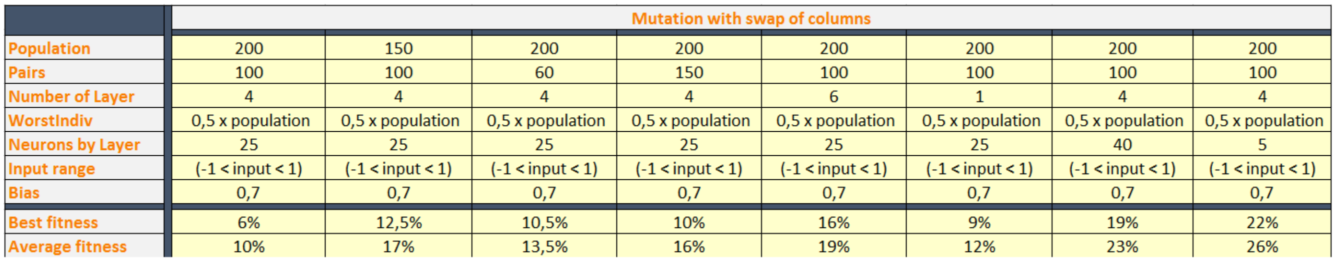
1.2.2 Optimisation using one mutation and the cross-mutation

For the first test, the function "MutationRandom" was used. By changing some parameters such as the number of pairs, the number of Layer or the bias, some evolutions around the best and average fitness values can be observed. The table below represents the results of the first mutation algorithm.



	Mutation with random values							
Population	200	150	200	200	200	200	200	200
Pairs	100	100	60	150	100	100	100	100
Number of Layer	4	4	4	4	6	4	4	4
WorstIndiv	0,5 x population	0,5 x population	0,5 x population	0,5 x population	0,5 x population	0,5 x population	0,5 x population	0,5 x population
Neurons by Layer	25	25	25	25	25	40	25	25
Input range	(-1 < input < 1)	(-1 < input < 1)	(-1 < input < 1)	(-1 < input < 1)	(-1 < input < 1)	(-1 < input < 1)	(-1 < input < 1)	(-1 < input < 1)
Bias	0,7	0,7	0,7	0,7	0,7	0,7	0,3	0,9
Best fitness	10%	9,6%	13%	11%	9,2%	17%	22,5%	7,6%
Average fitness	13%	13%	19%	15,5%	15,0%	24%	29,0%	14%

Figure 1.1: Evolution of the best fitness value & the fitness average



	Mutation with swap of columns							
Population	200	150	200	200	200	200	200	200
Pairs	100	100	60	150	100	100	100	100
Number of Layer	4	4	4	4	6	1	4	4
WorstIndiv	0,5 x population	0,5 x population	0,5 x population	0,5 x population	0,5 x population	0,5 x population	0,5 x population	0,5 x population
Neurons by Layer	25	25	25	25	25	25	40	5
Input range	(-1 < input < 1)	(-1 < input < 1)	(-1 < input < 1)	(-1 < input < 1)	(-1 < input < 1)	(-1 < input < 1)	(-1 < input < 1)	(-1 < input < 1)
Bias	0,7	0,7	0,7	0,7	0,7	0,7	0,7	0,7
Best fitness	6%	12,5%	10,5%	10%	16%	9%	19%	22%
Average fitness	10%	17%	13,5%	16%	19%	12%	23%	26%

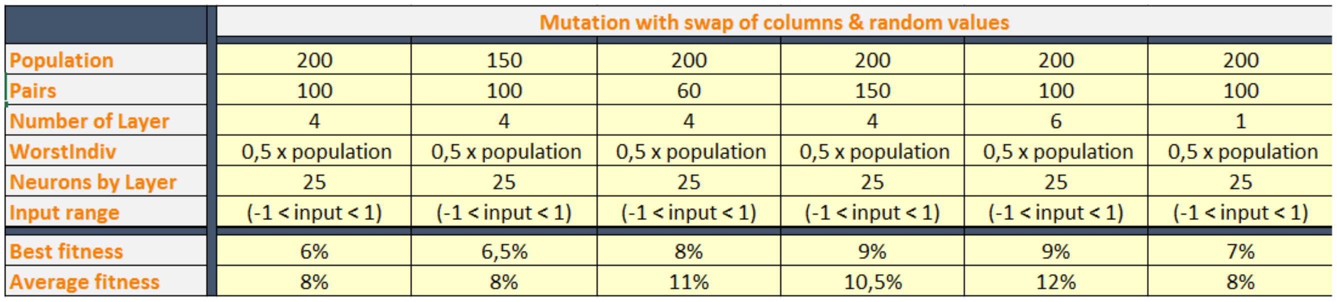
Figure 1.2: Evolution of the best fitness value & the fitness average

It can be seen from the first table that the most important changes happened when the number of neurons or the value of the bias has been changed. After a number of manipulation for each parameter, it can be noticed that the most significant changes occur when the bias is changed. Therefore, it was decided to keep the value of 0.7 for the next tests.

After the second test it can be seen that by changing the number of neuron in each layer, the accuracy of the fitness changes significantly. It was decided to keep 25 neuron in each layer for the future tests, which is the most optimum value.

1.2.3 Using the two mutations

After testing with one crossover and one mutation, it was decided to use two mutations.



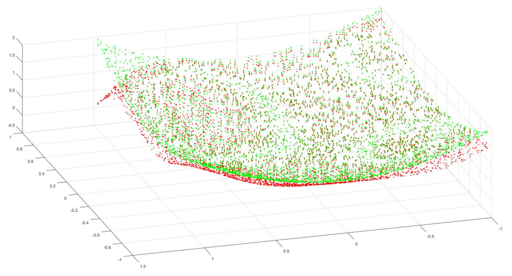
	Mutation with swap of columns & random values					
Population	200	150	200	200	200	200
Pairs	100	100	60	150	100	100
Number of Layer	4	4	4	4	6	1
WorstIndiv	0,5 x population	0,5 x population	0,5 x population	0,5 x population	0,5 x population	0,5 x population
Neurons by Layer	25	25	25	25	25	25
Input range	(-1 < input < 1)	(-1 < input < 1)	(-1 < input < 1)	(-1 < input < 1)	(-1 < input < 1)	(-1 < input < 1)
Best fitness	6%	6,5%	8%	9%	9%	7%
Average fitness	8%	8%	11%	10,5%	12%	8%

Figure 1.3: Evolution of the best fitness value & the fitness average

It can be seen that the results are clearly better by using the two mutations. The change in the accuracy is not significant, however, the best models obtained are the models with 4 layers or less. The best value found during testing is within 6 % accuracy of the desired value.

1.2.4 Testing with the three GA

Using the cross-mutation function together with the two types of mutation, the values obtained are very similar to the previous test. The best accuracy is 6 % and the best fitness average is 8 %. After repeating this test several times, it was confirmed that crossover does not improve our MLP.



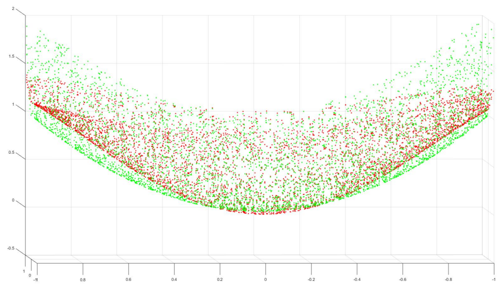


Figure 1.4: Comparison between the desired function (green) and the approximation (red) for two mutations (left) and the three GA (right)

1.3 Results & Conclusions

1.3.1 Best optimisation

After testing different combinations of GA, it was found that the best Multilayer Perceptron can be generated by using only two mutations, giving us a ”good” approximation with a fitness accuracy of 6 %. The value lower than that was not found, therefore, it could be possible that local minimum is reached. The parameters for our MLP are:

- Population of 200 chromosomes & a hundred of pairs
- One or four layers of 25 neurons
- A bias value of 0.7
- An input range of [-1 1]

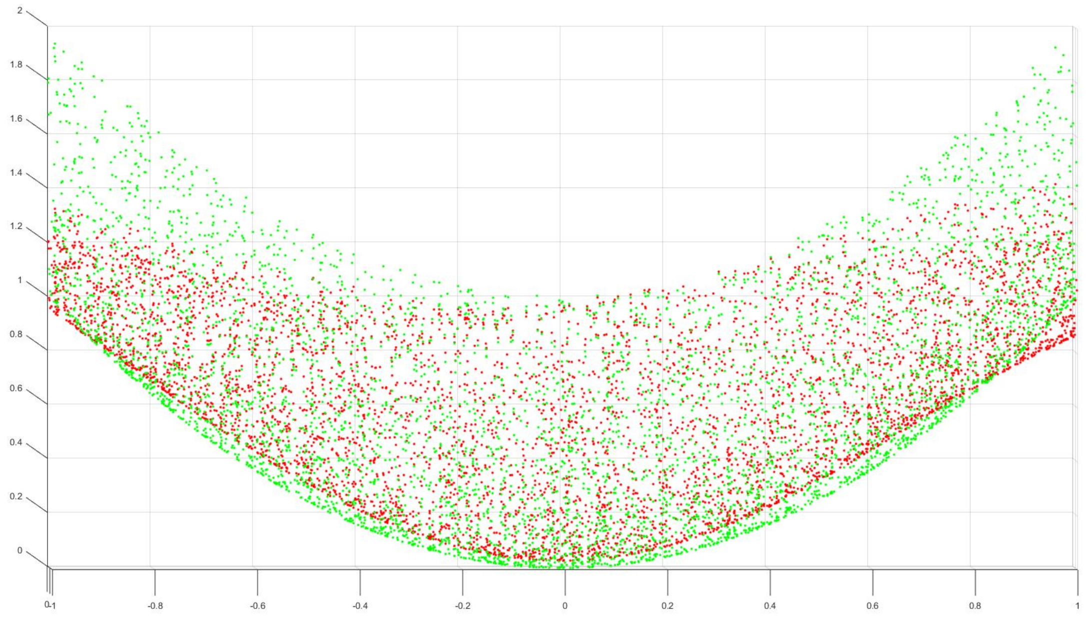


Figure 1.5: Best Approximation of the desired function

1.3.2 Bias

During the optimisation of the MLP a major problem has been encountered. The best obtained fitness value could not be reduced in many different cases. The most optimum solution found for this problem was to use the bias, a coefficient added to the input before the activation function: the fitness value was able to drop from 30 % to around 10 %. The figure below represents an approximation without using the bias coefficient.

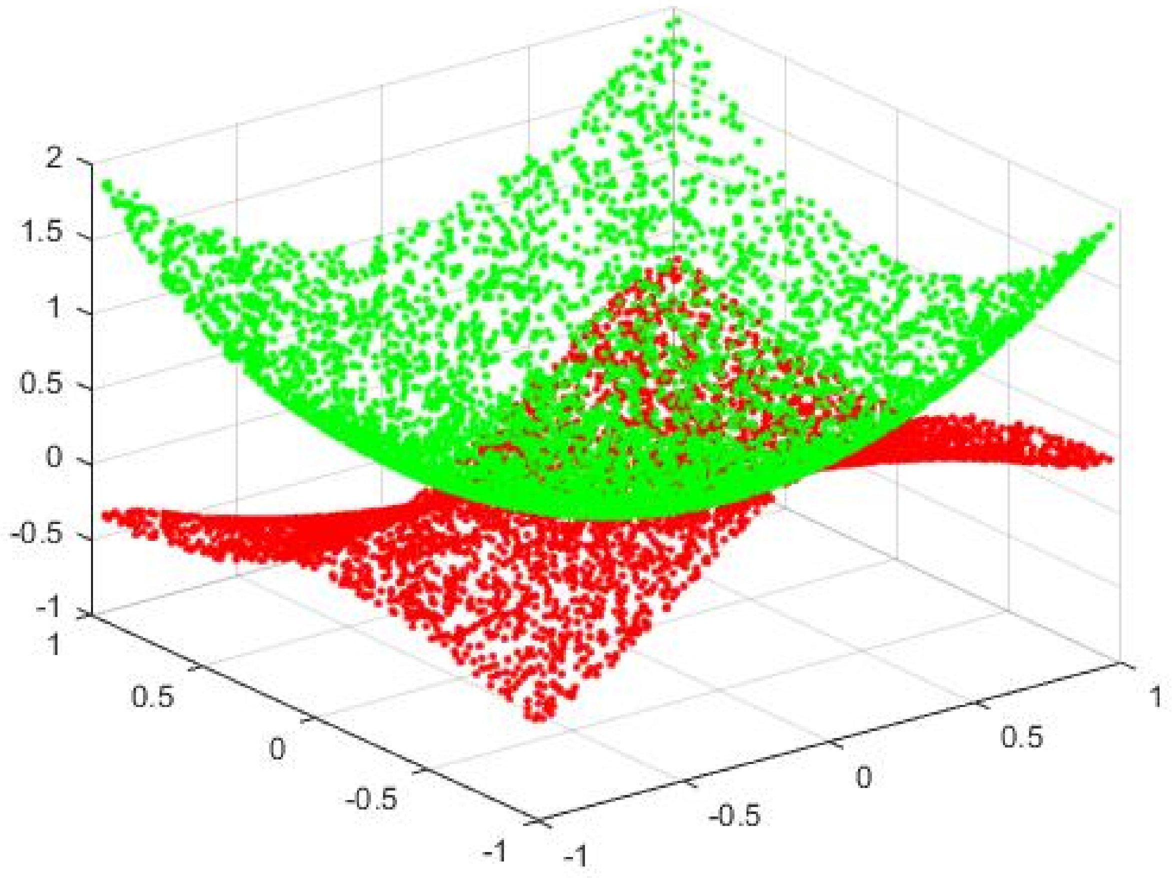


Figure 1.6: Approximation function without using the bias coefficient

1.3.3 Crossover

During the development of the GA crossover, a number of clones could be found in our population. The problem was that the crossover function created many children and after several generations, some children had the same weight matrices, even if the elements from their parents were picked randomly. To solve this problem, it was decided to mutate children by changing a weight matrix chosen randomly. Although some improvements have been achieved, they were still not good enough.

1.4 Minimise the approximate function

After the MLP has been optimised and the best chromosome has been generated, the minimum of the function has to be found using *Minimize.m* function.

At first:

- Create a population
- For each individual use the MLP of the best chromosome found previously
- Generate an approximate output of the function for each pair
- Rank them depending on the value of the *realout* (from lowest to highest)

In every generation for each individual:

- Add to each value of the pair a random value between -0.2 and 0.2
- Generate a new approximation (*realout*)
- Compare the new approximation with the previous one. If the new one is smaller, keep the new pair of input and the new *realout*.
- At the end of each generation, sort them depending on the value of the *realout*

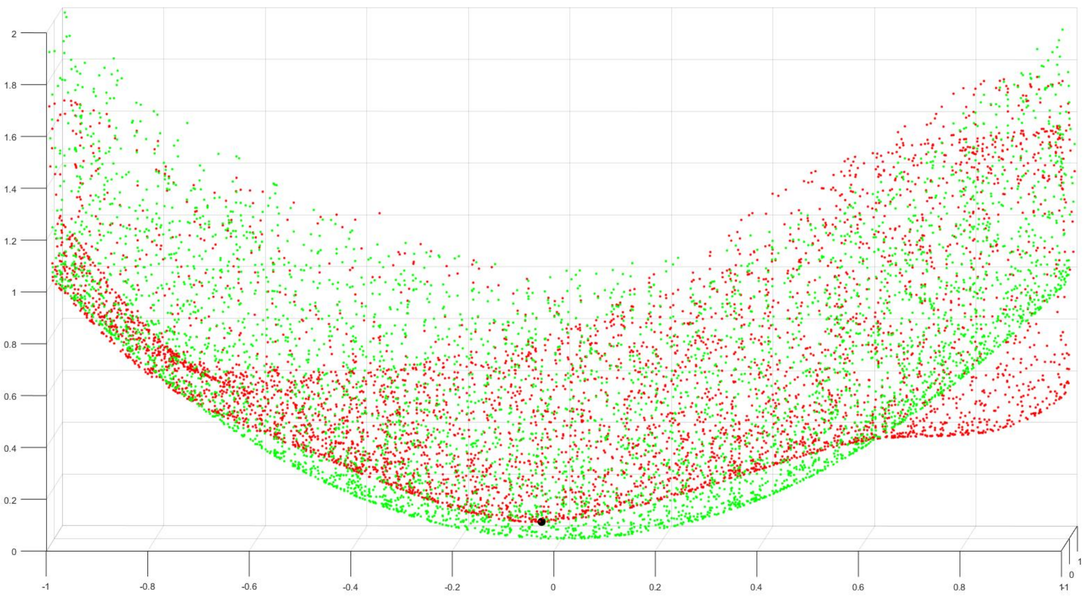


Figure 1.7: Minimum value of real out = 0.0818 (black point) and the corresponding pair $[-0.0344 - 0.3467]$

Chapter 2

Running an evolutionary algorithm with COCO

2.1 Step-by-step development of the evolutionary algorithm

2.1.1 Selection process

The first tested method of selection is a simple tournament selection with the following working principle:

1. Organise a tournament of random size
2. Choose a random lucky individual from the population and send it to participate in the tournament
3. Repeat stage 2 until the tournament size is full
4. Tournament is happening and all participants are sorted from the best to the worst
5. After the tournament is finished random number of top champions are chosen to be the best ultimate champions who can proceed to the next stage.

This group of champions is a starting population for the further sequence of experiments.

2.1.2 Crossover

Simple crossover is not very efficient and after many generations stops working due to the fact that all population is replaced with clones of the best individual. Population update was also one of the reasons to that, in this case those children who are better than their parents will proceed to the next generation together with parents whose children are not better than them. The code of this and previous step can be found in a file step1-2_tournament_and_crossover.m

2.1.3 First implementation of mutation

The working principle of this algorithm is the following:

1. Generate a population of random solutions
2. For each generation evaluate fitness of the population and sort individuals.
3. Choose a random individual
4. Mutate this individual depending on the mutate rate
5. Evaluate the fitness of this mutant. If the mutant is better than the worst individual in the population, then replace the worst with the new individual. Otherwise do nothing.
6. Go to stage 2 until a certain number of generations is reached or if a good individual is found.

The settings for this algorithm were the following: population size = 100, mutation rate = 0.8, max number of generations = 1000, dimension = 2. It has been tested with two dimensions on the Sphere function with an approximation of a global value of 0.001. Most of the time the global value can be achieved within 1000 generations with only 100 individuals in the population. However, sometimes it could take more than 10000 generations to achieve the target. Therefore, further improvement needs to be made. If the population size is increased to 1000 individuals in 6 out of 10 tests the global value can be achieved in the first generation. In other tests, it is guaranteed that the global value can be achieved within 2000 generations. The code of this step can be found in the file step3_mutation.m

2.1.4 Rank selection with variable crossover and mutation

The working principle of this algorithm is the following:

1. Generate a population of random solutions
2. Create an array of probabilities assigned to each place in population. The probability value depends on the population size. Probability of the first place is 1 and will decrease for each other further places with a rate of 1/popsize. In other words, probability of a place = probability of the previous − 1/popsize
3. Evaluate fitness of the population and sort individuals from the best (first place) to the worst (last place).
4. For each dimension make a random number of crossover operations depending on the crossover range (for example 1-5). During this crossover operation, places of two randomly selected values within a dimension are swapped.
5. Sort individuals depending on the new fitness after the crossover operation
6. Target a random individual.
7. Depending on the probability assigned to the place of this individual it can be chosen or not, if not go to stage 6.
8. Mutate this individual depending on the mutate rate.
9. Evaluate the fitness of this mutant. If the mutant is better than the worst individual in the population, then replace the worst with the new individual. Otherwise do nothing.
10. Go to stage 3 until a certain number of generations is reached or if a good individual is found.

The settings for this algorithm were the following: population size = 100, mutation rate = 0.8, max number of generations = 1000, dimension = 2, crossover range = 5. Using the same sphere function as in the previous case, this time the same goal of 0.001 can be achieved within a hundred generations almost all the time. In this case, an increase in population size to 1000 keeps the rate of achieving the goal in the first generation the same as in the previous test. On the other hand, it requires more resources and computation time to reach the goal if it was not reached in the first generation. It is guaranteed that the goal can be achieved within 200 generations. The code of this step can be found in the file step4_rank_selection_cros_mut.m

2.2 Running COCO

2.2.1 Testing mutation only on COCO

An algorithm described in step 3 is added to COCO and tested on the first three functions from the benchmarks function list. For this test the following settings have been used: dimension = 2, population size = 1000, number of generations = 1000, mutation rate = 0.8.

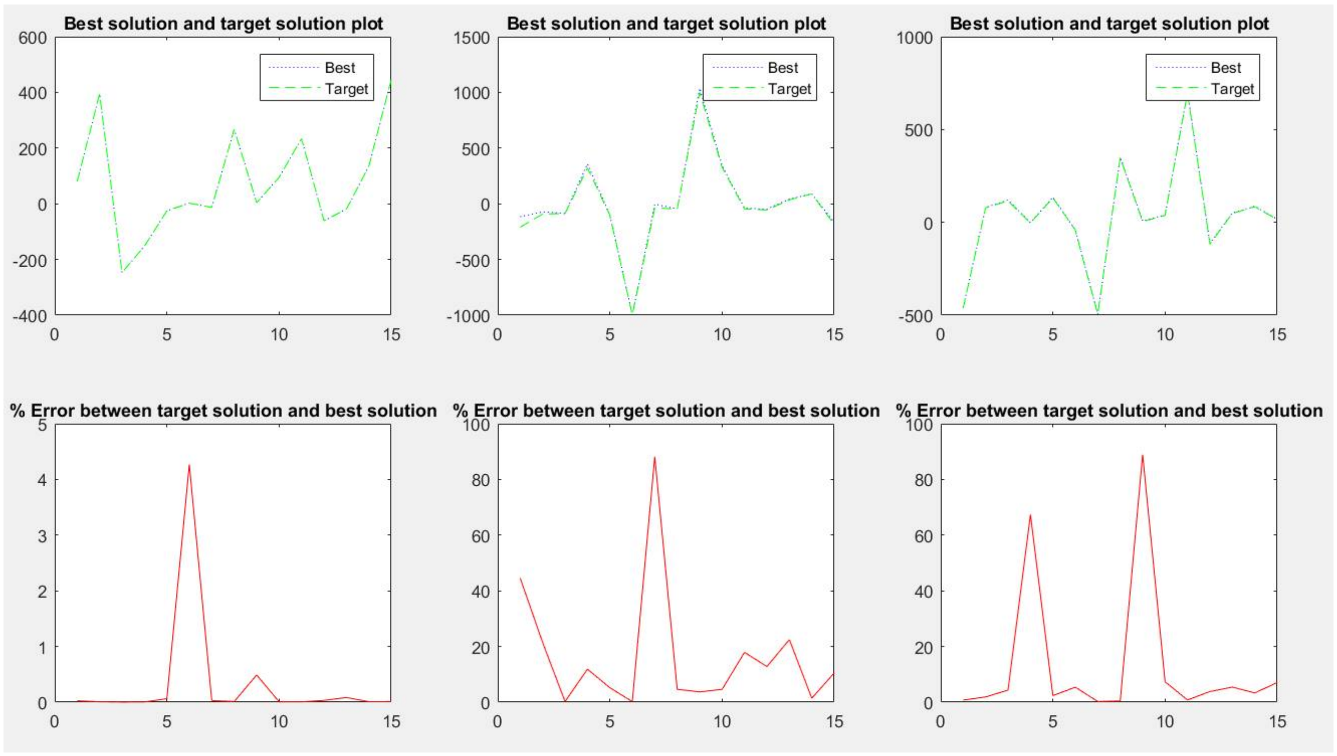


Figure 2.1: Results obtained after testing the first three benchmarks functions

It can be seen that for the first sphere function the best solution is very close to the target solution and the error is very small. However, the next two function are more complicated and the difference between the best and target solution is significant.

2.2.2 Testing improved algorithm on COCO

An algorithm described in step 4 is added to COCO and the following results are produced.

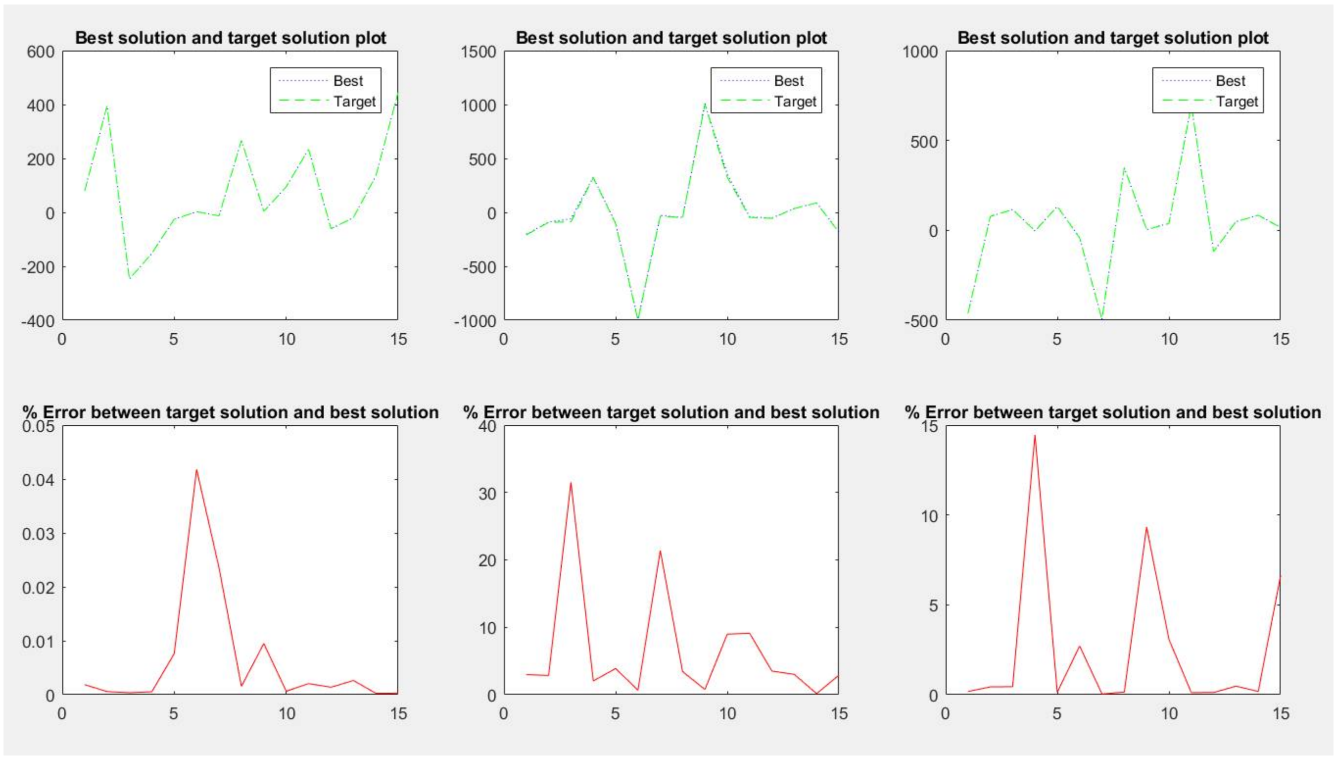


Figure 2.2: Results obtained after testing the first three benchmarks functions

It can be seen from Figure 2 that the error is decreased, which means that the best solution is now much closer to the target solution. The overall performance is increased significantly and 8 out of 15 instances in the first function could almost reach the target solution within the range of 0.001. This algorithm has been tested on all of the function and results can be found in the folder ”Final test”. The higher dimension the higher the error and it takes more time a resources to produce good results. In order to operate in 40-D the algorithm needs to be improved, in particular, crossover, mutation and selection methods must be more specific depending on the behaviour of the function so that all the search space is explored and a very good solution is generated.

GitHub link: https://github.com/HWBIC/Coursework-1