# Finding global minimum using genetic algorithms

Stamate Valentin 2B4

November 15, 2020

## Abstract

In this raport we will study the behaviour of a genetic algorithm solving a common optimization problem. Here will be explinations on how it works, the parameters, and based of the results concluzion.

## 1 Introduction

The raport contain the interpretation of the results returned by a genetic algorithm solving a common problem. This way I made more sections for clarity: methods: there is the description of the algorithm used as well as how a candidate is reprezented, stop condition etc., description: contains details about the input size, precizion, repetions etc. results: tables with results and also few graphics to interpret the results, comparisons: a discution on how well the algorithm behaved, conclusion and a bibliografy: a list with all the sources I used for making the algorithm.

### 1.1 Motivation

Genetic algorithms are inspired from nature and so far every species have evolved in such a way that they fit in the enviroment they are. So, making the same thing in code with a population we may expect a similar result with an enviroment set by us. The motivation is to see if such an algorithm give good results. If it does, this approach can be used in optimization problems that don't have a deterministic solution in polinomial time.

## 2 Methods

The algorithm used is a genetic algorithm. The idea is to have a random population at the beginning that evolves over time. This way we can select a random number of candidates and multiply them two by two making two childrens that are added to population. For every generation the candidates are picked in such a way that the best of them have a higher chance of being selectd for the next generation keeping the population constant. The selection of the population is made with *tourney selection*.

A member of population can be reprezented as a class with fields for assigning the radom selection and current fitness and the genome. The genome is reprezented as a bitstring that contains all the components for a point, because the mutations can be made very easy and we work with data.

The fitness for every candidate is calculated this way $P(i).fit = 1/(-min(P) * cd + F(P(i).gene) + 100)$ where, $cd$ is 1 if $min(P)$ is negative, 0 otherwise $min(P)$ is the minimum value found within members, $P$ is the population and $F(gene)$ is the value of the function with the point as a bitstring. This way we keep a positive fitness even if the function takes negative numbers and we don't need to know the global minimum.

We add 100 to the formula to avoid having 1/0 and to assing a fitness of 0.01 for the best candidate when $cd$ is 1.

The stop condition is when the number of generations reaches 1000.

Finally, the mutation is made for every candidate and every genome component has a probability of 0.1% to be changed. Without it, if we have a population with the same genome, it won't change.

## 3 Experimental Setup

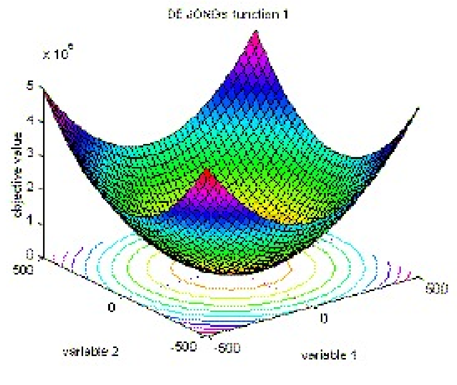
To have a better idea of how the algorithm is behaving we use different inputs and functions: as function dimention 2, 5, 10, 30 and functions: **DeJong's function**, **Schwefel's function**, **Rastrigin's function** and **Michalewicz's function**. Each function have a different number of global minimums. The precision for the experiment is 5 so $\varepsilon = 0.00001$. The mutations probability is 0.1% as explained earlier. The number of iterations is the same number as the stop condition witch is 1000 iterations/generations and the sample size is 30. The population size is 100 and the crossover probability is 20%.

CPU : Intel i5 - 8265U with 4 phisical and 8 virtual cores.

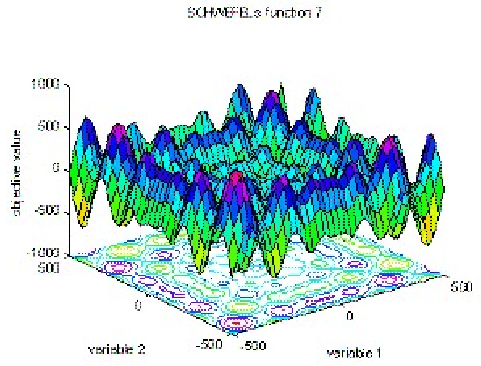
Software: Visual Studio Code

Programming Language: C++ 11

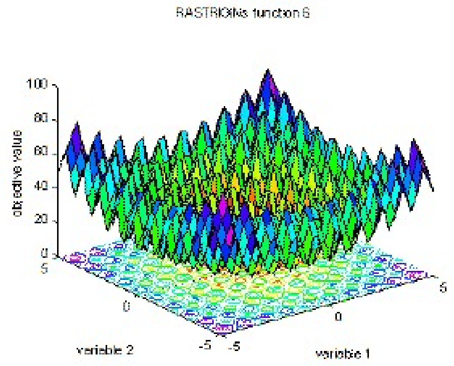
The reprezentation of the functions have two dimensions and below evey image is the funtion definition, the interval and the global minimum.



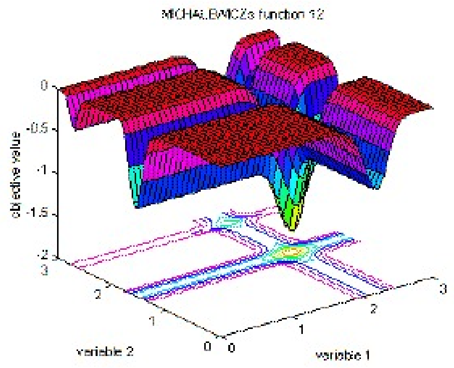
De Jong's function: $f_n(x) = \sum_{i=1}^{n} ix_i^2 \ \ -5.12 \leq x_i \leq -5.12, \ \ f(x) \geq 0$



Schwefel's function: $f_n(x) = \sum_{i=1}^{n} -x_i sin(\sqrt{|x_i|}), -500 \leq x_i \leq 500, f_n(x) \geq -n*418.9829$



Rastrigin's function: $f_n(x) = 10n + \sum_{i=1}^{n}(x_i^2 - 10cos(2\pi x_i)), -5.12 \leq x_i \leq 5.12, f(x) \geq 0$



Michalewicz's function: $f_n(x) = -\sum_{i=1}^{n} sin(x_i)(sin(ix_i^2/\pi))^2 0, f_5(x) \geq -4.687, f_{10}(x) \geq -9.66$

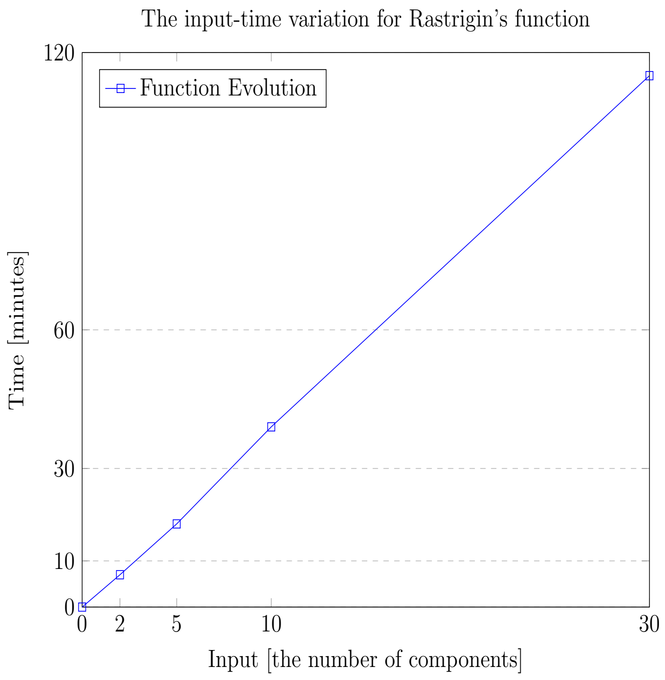
## 4 Results and Comparisons

| 2 dimensions | | | | |
|---|---|---|---|---|
| Function | Best Value | Mean | StDev | Duration |
| De Jong | 0.00496 | 0.50765 | 0.84868 | 7min 6s |
| Schwefel | −837.96 | −815.6729 | 51.61495 | 9min 25s |
| Rastrigin | 0.00067 | 2.70680 | 2.51859 | 7min 8s |
| Michalewicz | −0.80122 | −0.41647 | 0.33797 | 6min 51s |

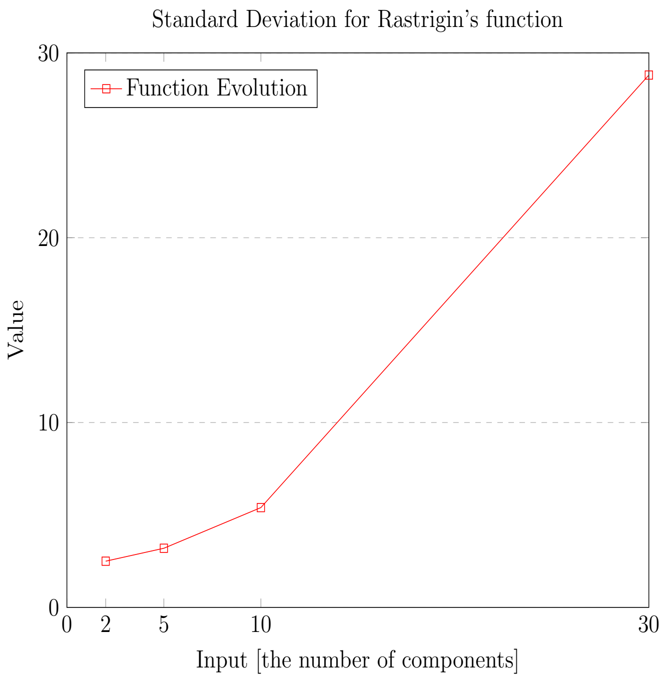
| 5 dimensions | | | | |
|---|---|---|---|---|
| Function | Best Value | Mean | StDev | Duration |
| De Jong | 0.23029 | 1.48307 | 1.07994 | 18min 40s |
| Schwefel | −2092.64 | −1932.087 | 113.9483 | 24min 14s |
| Rastrigin | 0.81863 | 7.84385 | 3.28231 | 18min 46s |
| Michalewicz | −2.15064 | −1.38632 | 0.44637 | 18min 9s |

| 10 dimensions | | | | |
|---|---|---|---|---|
| Function | Best Value | Mean | StDev | Duration |
| De Jong | 1.20608 | 2.75918 | 1.14485 | 39min 42s |
| Schwefel | −3993.25 | −3600.308 | 181.7103 | 51min 9s |
| Rastrigin | 11.2655 | 18.72519 | 5.41844 | 39min 54s |
| Michalewicz | −4.48764 | −2.81137 | 0.80355 | 38min 36s |

| 30 dimensions | | | | |
|---|---|---|---|---|
| Function | Best Value | Mean | StDev | Duration |
| De Jong | 13.1106 | 20.1773 | 4.79230 | 1h 54min 41s |
| Schwefel | −11169 | −10252.52 | 850.5401 | 1h 55min 19s |
| Rastrigin | 74.966 | 132.1712 | 28.80329 | 1h 55min 22s |
| Michalewicz | −9.07803 | −7.16802 | 1.01378 | 1h 51min 36s |



We observe that the time increases linear with input witch is a good thing because it can be used in practice for bigger inputs also taking into consideration that the running time is low for such a problem.



The standard deviation looks like it's increasing exponentially but it's normal because once with the increase of input also the domain increases and we didn't increase the other parameters.

Now, let's run the same algorithm but with mutation probability of 1% and 0.01% instead of 0.1% to see what changes.

| Rastrigin's function with 30 dimensions | | | |
|---|---|---|---|
| Mut. Prob. | Best Value | Mean | StDev |
| 0.01% | 111.938 | 174.0999 | 39.59821 |
| 0.1% | 74.966 | 132.1712 | 28.80329 |
| 1% | 297.874 | 335.4668 | 29.83531 |

We see that the result is the best when mutation probability is set to 0.1%, better than when is set to 1%. So making it smaller means better results? Not necessary. Decresing it down to 0.01% there is almost no mutation and only crossover induce variation. This is why we get the worst result.

Let's make another test but this time changing the probabilities one by one. The sample size is 10.

| Rastrigin's function with 10 dimensions | | | | |
|---|---|---|---|---|
| Mut. Prob. | Crossover | Best Value | Mean | StDev |
| 0.1% | 20% | 12.7081 | 17.00872 | 4.63378 |
| 0.1% | 100% | 8.47128 | 16.63453 | 5.27540 |
| 75% | 20% | 91.6142 | 110.7948 | 11.07916 |

The first row is from my normal results picking only 10 elements from sample. So, making the crossover 100% we see that we get a better result. The crossover is made within all the population, meaning that there will be more candidates to choose and the next generation will contain more good candidates. Looking at the third row with mutation probability of 75% the population suffers may mutation and if this probabilty would be increased to 100% we willl obtain an algoritm similar to *random search*. This is why we get the worst results here.

## 5 Conclusions

We study a genetic algorithm solving the minimum of a function problem. The results are very good for such an alogrithm even if it doesn't give the best results for big inputs, they are still relevant. The time is pretty low and increases linear, so we can approximate how much it's gonna take for greater inputs. As a conclusion, genetic algorithms can be use in practice for solving many optimization problems because even if they don't give the best result at a certain time they give a very good approximation of it.

## References

[1] https://stackoverflow.com/questions/39102028/how-to-return-vector-of-pointers-and-ow

[2] https://stackoverflow.com/questions/11134497/constant-sized-vector

[3] https://stackoverflow.com/questions/1380463/sorting-a-vector-of-custom-objects

[4] https://profs.info.uaic.ro/~eugennc/teaching/ga/

[5] https://www.youtube.com/watch?v=9zfeTw-uFCw&list=
PLRqwX-V7Uu6bJM3VgzjNV5YxVxUwzALHV&ab_channel=TheCodingTrain

[6] Seminar 5 notes

[7] http://www.geatbx.com/docu/fcnindex-01.html#P89_3085

[8] http://www.geatbx.com/docu/fcnindex-01.html#P150_6749

[9] http://www.geatbx.com/docu/fcnindex-01.html#P140_6155

[10] http://www.geatbx.com/docu/fcnindex-01.html#P204_10395