Internship Report entitled:

«Benchmarking Bi-objective Black-box Algorithms»

Illia Sylenko
Dates: May 15th - September 30th 2016
Presentation: September 14th 2016

Supervisors:
Dimo Brockhoff
chef de recherche INRIA Lille
Anne Auger
researcher at INRIA Saclay

University tutor:
Filippo Santambrogio
Laboratoire de Mathématiques d’Orsay,
University Paris-Sud

Master degree in Optimization, University Paris-Sud
https://webens.math.u-psud.fr/-optimization-?lang=fr

September 26, 2016
## Contents

1 Internship context .................................................. 3

2 Introduction to bi-objective black-box optimization .......................... 3

3 Experimental setup .................................................. 5
   3.1 Benchmarking platform ........................................... 5
   3.2 Algorithm introduction ........................................... 6

4 Results ........................................................................ 9
   4.1 Introducing restarts .................................................. 9
   4.2 Comparing the choice of population sizes ......................... 12
   4.3 The "ipop" algorithm of variated population size ................. 16
   4.4 Searching for the best values of parameters for the stopping
       criteria and generating restarts of the algorithm ............... 20
   4.5 Comparing the different variations of "ipop" algorithm .......... 24

5 Conclusions ............................................................... 26

References .................................................................... 28
1 Internship context

I have done my internship in the TAO team of Inria Saclay, which is research group interested in the interplay of Machine Learning and Optimization. The achieved goal of my internship was to familiarize myself with the COCO platform (Comparing Continuous Optimizers Platform) for Benchmarking, to test the bi-objective algorithm NSGA-II with this platform, analyze the results and improve a little bit the algorithm. Except learning how to do Benchmarking for black-box algorithms, I have also learned how to use LaTeX for the first time.

In more details about the structure of the internship, first of all I was exploring the `gamultiobj` function of the algorithm NSGA-II in Matlab, and, equipping it with restarts, I had to figure out the most suitable values of the parameters for the stopping criteria, in order to have the amount of restarts that will have the best impact on the performance of the algorithm. This was done via benchmarking on the Coco platform, which is a Benchmarking platform for single- and bi-objective algorithms. The Benchmarking provided the comparison of the convergence of different algorithms in different cases of objective functions and in the overall case, which helped to decide which is the algorithm that should be used. The conclusion was that the parameter TolFun for the function `gamultiobj` should not be kept as the default one, but it has to be made bigger in order to have a better result for the algorithm with restarts. Also, I was investigating the choice of the population size of the algorithm (which was now NSGA-II with restarts), and after observing the performance, we concluded that the choice of population size depends very much on the amount of budget that is used. In order to have a general-case algorithm and to improve even more the final results of the algorithms with bigger budgets, the solution was to consider the impact of different population sizes in different stages of the run of the algorithm, and to combine them in a good way, so that the population size changes dynamically during the algorithm. This resulted in making the so-called ”ipop” (from “increasing population size”) algorithm, where the population size increases (factorized by 2) after each restart of the function `gamultiobj`. The performance of the ”ipop” algorithm was then tested with Coco platform again, and investigated on the choice of the initial population size. Eventually, the benchmarking proved the ”ipop” algorithm to be better than the algorithms with a constant population size.

2 Introduction to bi-objective black-box optimization

In the context of the internship we considered bi-objective, unconstrained minimization problems of the form
\[
\min_{x \in \mathbb{R}^D} f(x) = (f_1(x), f_2(x))
\]

where \(D\) is the problem dimension. \(f_1 : \mathbb{R}^D \rightarrow \mathbb{R}\) and \(f_2 : \mathbb{R}^D \rightarrow \mathbb{R}\) are the two objective functions, and the min operator is related to the standard dominance relation. A point \(x \in \mathbb{R}^D\) is dominating another point \(y \in \mathbb{R}^D\) if \(f_1(x) \leq f_1(y)\) and \(f_2(x) \leq f_2(y)\) hold and at least one of the inequalities is strict.

Solutions which are not dominated by any other solution in the search space are called Pareto-optimal. These Pareto-optimal solutions constitute the Pareto set [4]. The image of the Pareto set in the objective space is called Pareto front. As the functions are conflicting in the sense of the problem being non-trivial, so that improving the value of one objective function results in worsening the value of the other, the Pareto front in the coordinate system of the two objective functions will then look like some monotone decreasing line. A demonstration example of how the Pareto front can be drawn is on the picture (Figure 1)[6]:

![Figure 1: Pareto Front](image)

To solve the bi-objective optimization problem implies finding (approximating) the Pareto set.

Bi-objective optimization is a special case of multi-objective optimization, which is a big area of research concerned with mathematical problems involving more than one objective function to be optimized simultaneously. Multi-objective optimization has been applied in many fields of science, including economics, engineering and logistics, where optimal decisions need to be taken in accordance with trade-offs between two or more conflict-
ing objectives. Examples of multi-objective problems could be minimizing cost while maximizing comfort while buying a car, or maximizing performance whilst minimizing fuel consumption and emission of pollutants of a vehicle.[5]

3 Experimental setup

3.1 Benchmarking platform

As for benchmarking algorithms, I had to use a COmparing Continuous Optimisers platform (COCO), which was created especially to check the convergence of some black-box optimization algorithms (single or bi-objective). The platform gives the convergence results (in plots) for some build-in functions of different behavior and in different dimensions. As for the bi-objective optimization (bbob-biobj), the platform includes 55 different bi-objective test functions in continuous domain, which are the combinations of some well-known functions from the single-objective noiseless test suite (bbob). These functions are all of different nature (e.g. separable, moderate, ill-conditioned, weakly-structured, multi-modal), which provides the diversity and generality of the test suite. The benchmarking platform provides the results of the work of an algorithm not only for each concrete function, but also the results for the function groups, as well as a general result for all of the given functions together. Except this, the bbob-biobj test suite of COCO provides the results for six different dimensions (2D, 3D, 5D, 10D, 20D and 40D) and also on a number of possible set instances. The instances are different independent runs of the algorithm on one function. Usually during the internship, the selected amount of instances was 5.

More exactly about bbob-biobj test suite, a multi-objective problem is represented by a combination of single objective functions with representative difficulties, which are typically observed in real-world problems. Having 24 bbob functions for the single objective case, it implies having 300 different combinations of these function in bi-objective case. To reduce this number of bi-objective functions to make combinations, the 24 bbob functions were reduced to 10, so that there are two functions representing each of the following classes: separable functions; functions with low or moderate conditioning; functions with high conditioning and unimodal; multi-modal functions with adequate global structure; multi-modal functions with weak global structure. All combinations of these 10 functions create 55 test-functions for bbob-biobj test suite.

The plots that the COCO platform provides are Runtime distributions per function and dimension; Runtime distributions per group and dimension; average runtime with dimension; and average runtime for selected targets. I was more interested in Runtime distribution plots, which show the dependency between the empirical cumulative distribution function (ECDF) on a
number of function evaluations. ECDF presents the runtimes to reach the target values, which are based on target precision and a reference hypervolume indicator value, which is an approximation of hypervolume indicator value on the Pareto set.[8]. Runtime distributions values can be displayed considering the aggregation over target values and over subclasses of problems, or all problems.[9]

In the COCO platform, we display average runtimes and the empirical distribution function of runtimes. When displaying runtime distributions, we consider the aggregation over target values and over subclasses of problems, or all problems.

As for the other software I was using Matlab 2015a and its Global Optimization Toolbox, and I also had to connect and use the servers of Inria, because sometimes it took really long time to run the benchmarking of an algorithm. It would for example take several days to run and collect the data for a single algorithm on the budget 1e5.

3.2 Algorithm introduction

The algorithm to do the Benchmarking on was chosen to be NSGA-II, which is considered to be the baseline algorithm in the multi-objective optimization domain and is one of the most cited Multi-Objective Evolutionary Algorithms, if not the most cited one - it has more than 20,000 citations in Google scholar. The paper that is cited is [1], written by Indian authors in 2002. ((The most prominent features of NSGA-II are its low computational complexity, elitist approach and a method for diversity that does not need additional parameters[7].))

NSGA-II is an improved version of NSGA (Nondominated Sorting Genetic Algorithm), which addresses three main criticisms of NSGA: high computational complexity of nondominated sorting; nonelitism approach; and the need for specifying a sharing parameter[1]. NSGA-II is a fast nondominated sorting approach with $O(MN^2)$ computational complexity (where $N$ is the number of objectives and $M$ is the population size). The algorithm selects $N$ best solutions from combined parent and offspring populations, using presented in the paper[1] new selection operator.

«The objective of the NSGA algorithm is to improve the adaptive fit of a population of candidate solutions to a Pareto front constrained by a set of objective functions. The algorithm uses an evolutionary process with surrogates for evolutionary operators including selection, genetic crossover, and genetic mutation. The population is sorted into a hierarchy of sub-populations based on the ordering of Pareto dominance. Similarity between members of each sub-group is evaluated on the Pareto front, and the resulting groups and similarity measures are used to promote a diverse front of non-dominated solutions.» [3] The algorithm approximates the Pareto front adding more and more points to the archive (the set of non-dominated points.
that have been discovered and selected at some point of the run of the algorithm), improving it continuously, and advancing to the Pareto front. All this evolution of the archive has been stored during the run of the algorithm, and I have plotted it with Matlab to show how it evolved during the run of the algorithm on some functions, and what is the final result for the approximated Pareto front. The Figure 2 shows the evolution of the archive for the sphere function, which is the first function of Coco Benchmarking platform, and Figure 3 displays the evolution of the archive for Combination of two Gallagher functions with 101 peaks, which is the last one (55th) function in Coco platform. For both tested functions the algorithm was run on budget $1e4$ (can be considered as the number of function evaluations). Some more information more about the second 55-th function: it is a function from the class of weakly-structured - weakly-structured combination, both objective functions are non-separable and highly multi-modal. Position and height of all 101 optima in each objective function are unrelated and randomly chosen and thus, no global structure is present.[4]

![Figure 2: Archive plot for the sphere function](image)

The dots of the blue color are the ones that have been added first, and of the yellow color - the most recent ones in the archive.
The genetic algorithm NSGA-II is present as a build-in algorithm in the Global Optimization Toolbox of Matlab, where it can be called by a function `gamultiobj`. `gamultiobj` finds a local Pareto set of the objective functions defined. The algorithm was tested in 2D objective space of 55 combinations of objective functions of different behavior within the COmparing Continuous Optimizer framework (COCO) [2]. The test is done on different dimensions of the search space: 2, 3, 5, 10 and 20.

`gamultiobj` uses three different criteria to determine when to stop the solver. The solver stops when any one of the following stopping criteria is met. It stops when the maximum number of generations is reached (by default this number is ‘200*numberOfVariables’). `gamultiobj` also stops if the average relative change in the best fitness function value over the StallGenLimit generations (default is 50) is less than or equal to tolerance specified as a parameter in options.TolFun. The third criterion is the maximum time limit in seconds (default is Inf). The stopping criteria of the function tolerance (TolFun) was the one of the values in the algorithm, along with the population size, that was most investigated during the internship. The default value of function tolerance is 1e-6 in Matlab 2015a, however it already changes in Matlab 2016a, in the updated Global Optimization Toolbox. In Matlab 2016a the name of the parameter is changed to FunctionTolerance, and is now by default 1e-4 (with the default value MaxStallGenerations, which was referred as StallGenLimit in previous version, increased to 100
from 50). Except this distinction, the reference to ‘the average relative change in the best fitness function value’ was replaced by the ‘average change in the spread of the Pareto front’. Having said that, in my case I was using Matlab 2015a during my internship.

4 Results

4.1 Introducing restarts

During the internship I was using slightly modified version of the algorithm, in particular I had a ”budget” value of some quantity of allowed function evaluations, which was set, and then I was restarting the function `gamultiobj` each time when the stopping criteria of it was met. The final stopping criteria of the algorithm that generates restarts was that the number of function evaluations cannot exceed budget value.

Not only the following approach allows to control the number of function evaluations, if it is important to have some cost boundaries, but also this way the whole algorithm can be run longer and in the end produce better results when the budget is set to be bigger. To demonstrate this, on the Figure 4 there are provided comparison plots of the simple `gamultiobj` algorithm without restarts and the algorithm with restarts and a set budget of 1e4.

![Figure 4: Algorithms without restarts comped to the ones with restarts](image-url)
The data presented is describing the empirical cumulative distribution function (ECDF), and showing an averaged result of all of the 55 functions available for benchmarking. This information is provided in the top left corner, together with the indication that the dimension of the problem presented is 2D. In order to demonstrate more things, here are present the algorithms for the population sizes 4 an 256. The purple and dark blue lines correspond to the algorithms without restarts with population sizes 4 and 256 accordingly; and the yellow and light blue lines are of modified algorithm with restarts and population sizes 4 and 256 accordingly. The crosses correspond to the points where the whole algorithm has finished. The lines that still continue after crosses are describing the best estimation of further running with more budget, however for this part there is no actual data. (More formally about, the cross on the ECDF plots of COCO represents the median of the maximal length of the unsuccessful runs to solve the problems aggregated within the ECDF.[9])

As for the plots of population size 4 (two lower ones), one can observe that the two pairs of compared algorithms run similarly until the first restart (until the cross, which is a point where the function \texttt{gamultiobj} has stopped), which happens much earlier for the population size 4 than for the population size 256. One of the lines finishes here, and for the algorithm that at this point makes a restart - a line continues until a cross at the point \(1e4\) - which is a finish due to exhaustion of the budget. For the population size 256, both algorithms - equipped with restarts and without, were forced to be stopped near the point of \(1e4\).

Now, when the algorithm is equipped with restarts, we can look at the stopping criteria of function \texttt{gamultiobj} not in the perspective of when the algorithm can be stopped, but rather when a next restart can be generated.

Restarting numerical optimization algorithms has been proven to be crucial for good algorithm performance in the single-objective case, and recently, the investigation of online convergence detection also gained interest in the MO field, although the focus there has been less on restarts itself, but in not burning function evaluations.

Generating new restarts of the algorithm at the right time can prevent it from being a little bit ‘stuck’, and thus the suitable stopping criteria need to be met. In Matlab function \texttt{gamultiobj}, there is a stopping criteria implemented depending on a convergence. Namely, the algorithm stops whenever the average relative change in the best fitness function value over StallGenLimit generations (which is some number of maximum stall generations) is less than or equal to a value TolFun. Those two parameters: StallGenLimit and TolFun, can be tuned in the function \texttt{gaoptimset} which is preceding the function \texttt{gamultiobj} and creates the option structure for generic algorithm. Doing so, one can set at which stage of convergence of the best fitness function the algorithm should stop. It was however discovered during the internship, that it is more effective to be changing TolFun
- a value of Function Tolerance (which is a positive scalar) rather than StallGenLmit. The default value of this parameter (TolFun) in the Global Optimization Toolbox of Matlab is 1e-6.

To have an idea in which way changing the time of a restart can impact the results of an algorithm, one can address to the Figure 5. It contains the two algorithms without restarts (this time with population sizes 8 and 128), and also two algorithms with restarts and with increased TolFun value from 1e-6 to 1e-3 in order to have restarts earlier (also with the same population sizes respectively).

![Figure 5](image)

**Figure 5:** Algorithms without restarts comped to the ones with restarts and increased TolFun value to 1e-3

This time the algorithms with restarts not only run for longer, but also they have increased ECDFs comparing to the previous case. Now the two lines go together until the separation point, which is the point where the first restart of the algorithm with TolFun=1e-3 happened. The thing that should be pointed out, is that the restarted algorithm continues to work better than the one where function `gamultiobj` has not been stopped (because the stopping criteria arrives later for TolFun value 1e-6). Also, here the algorithm without restarts on the population size of 128 was actually forced to stop at the number of evaluations 1e4 manually as well, without this criteria it would be supposed to continue on.
It was an important part of the task of the internship to figure out which parameters of stopping criteria for the function `gamultiobj` to use in order to have the best possible results for the algorithm with restarts, however I decided to explore this topic in full at the final stage of the report, firstly focusing on the choice of the population sizes. But I should point out, that from now on, all my work during the internship will be concerned about the NSGA-II algorithm with restarts.

4.2 Comparing the choice of population sizes

As it could be seen in the previous plots, in the function `gamultiobj` one can set a population size on which the algorithm will perform. Intuitively, with small population sizes the algorithm should give better results on smaller budgets, but in the long run the better idea is to chose a bigger population size. Using the COCO platform for benchmarking, I was able to compare different population sizes and to see the resulting ECDF, and it was the starting point of the internship. The population sizes were taken as factors of two from 4 up till 256. The comparison of these results can be seen on the following plots of Figure 6:

Figure 6: A comparison of algorithms using different population sizes with the TolFun value set to 1e-3

This is again the result for an aggregation of all the functions from the
bbob-biobj test suite. I should point out that these results are shown for the algorithms where I have changed the TolFun value to 1e-3 - the one that has been already used successfully in the previous plot, and which is supposed to be a better choice for the value of the parameter (for the case of generating restarts of course).

Indeed, the plots show that on different stages of the process, different population sizes are performing better than the others, and there is no optimal population size for the general case, that will always be better for any set budget. That makes one think about the idea of dynamically changing the population size depending on which stage of the process the algorithm is. About the implementation of this "increasing-population" algorithm is the next paragraph.

But going back to the current state, and concluding from the given plots - while different budgets set the choice for the preferred population size, the choice of a budget on the Figure 6 was 1e4, and at the finish of the algorithm, the higher population sizes produced better result in this case. This might be also seen on the demonstrations of the archive for the different population size choices, which will show how well is Pareto Front actually approximated and what is the impact of well-chosen population size. For example, on the Figures 7 and 8 are presented two archives for the sphere function on the budget 1e4: one of them is with population size 4, and another one is with population size 128.
Figure 7: An archive for the sphere function for the NSGA-II algorithm with budget $1e4$ and population size 4
Figure 8: An archive for the sphere function for the NSGA-II algorithm with budget 1e4 and population size 128

As for the algorithms with population sizes that are even higher than 256, they are not likely to be giving significantly improved results, but I thought that it would be interesting to run them anyway - they can be observed on Figure 9.
Figure 9: A comparison of algorithms that are using much higher population sizes

4.3 The "ipop" algorithm of variated population size

As it was seen from the comparison plots of NSGA-II with different population sizes, using a low population size is better for the shorter run length, however for the long run lengths, the algorithm with a high population size eventually will start to converge faster. The task at this point was to have an algorithm, that would behave well in general, despite this distinction. In order to have a general algorithm, the idea was to combine suitable leap areas given by different population sizes, dynamically increasing the population size after each restart. In this case, at the beginning the algorithm will use a low population size, and have better convergence at the small number of evaluations, and then later the population size will be changed, and it will also give a greater speed-up of the algorithm, just like it is seen on the plots of Figure 6 that the behavior of algorithm with bigger population sizes improves after some point. This kind of population size adaptation algorithm is referred to as "ipop" (stands for 'increasing population size'), and its concept was to start the algorithm on some small population size, and increase it by a factor of two after each restart of the algorithm. It can be considered as a sequence of runs of the `gamultiobj` function of Matlab, each one with a different, bigger population size until the exhaustion of the
The described "ipop" algorithm is not very hard to implement. After running it and collecting the output data, it was possible to plot the way the population size changes during the evolution of the "ipop", and at which points of number of evaluations. The description of this is on Figure 10. For the displayed algorithm, "ipop" was chosen with the starting population size 4 (I will refer it as "ipop-4" for simplicity), and ran on a sphere function (the 1st function of 55) in 2-dimensional space, using all default values, except TolFun=1e-3.

As displayed, even when the TolFun value is set to 1e-3, the first restart happens not very early, which is approximately on 1e2 function evaluations, so the population size 4 was kept for probably too long. And it would be even worse situation if the TolFun was of the default value 1e-6. That is when it is even more relevant to have an earlier restart, setting the parameters for the stopping criteria of `gamultiobj`. And it is also displayed that the maximal value of population size, that the "ipop" algorithm reaches is 32 for the budget 1e4. The actual ECDF plots of the very same "ipop-4" algorithm on the same function with same parameters is shown on Figure 11 - and it also provides the comparison of "ipop" to the algorithms that do not change
the population size:

Figure 11: Algorithm of "ipop-4" on the sphere function with the comparison to the algorithms of constant population size

On the Figure 11 it is possible to see how the "ipop-4" algorithm follows the same path as the algorithm with the population size 4, until the point, which is the same as the point of change on the previous Figure 10 - the point of a restart. After it, the population size has changed to 8, and it also can be seen from Figure 11 that the run of the algorithm afterwards is much improved comparing to the one with constant population size 4.

Another plot on Figure 12 shows the aggregated results, where all the functions from 1 to 55 were taken into account, and again, the performance of the "ipop" algorithms ("ipop-4" and "ipop-8") are compared to the algorithms with constant population size.
Figure 12: Overall comparison of "ipop" algorithm to the ones with constant population sizes for the TolFun=1e-3

As it has been noted, The TolFun value of *gamultiobj* function in this case of "ipop" algorithm is even more essential, because the shorter is the run of *gamultiobj* function is - the faster the population sizes will change, and the better it is, considering that it takes some time, as for it was shown in the previous case. For example, if to compare on another plot in Figure 13 all the same algorithms, but with the TolFun 1e-6, then in this case it takes longer to change the population sizes, as a result the population size in the "ipop" algorithm stays the initial one for too long, which is not beneficial for the algorithm.
Concluding what has been said, it is of major importance to change the parameters for stopping criteria of the `gamultiobj` function from the default ones, in order to improve the results of the generic algorithm equipped with restarts and with increasing population size. About this is the next paragraph.

### 4.4 Searching for the best values of parameters for the stopping criteria and generating restarts of the algorithm

The question of suitable parameters for stopping criteria was also answered with the help of Benchmarking on Coco platform. It was clear that in order to generate restarts earlier one should increase the value of Function Tolerance, or indeed decrease the value of Maximum Stall Generations (StallGenLimit).

After all different tests the conclusion obtained is that the value of $5 \times 10^{-3}$ is probably the most suitable of all, as for the algorithm that generates restarts and increases the population size by 2 after each of them, however the value of $1 \times 10^{-3}$ up to even bigger values until $1 \times 10^{-1}$ still remain well-chosen, giving approximately similar results. Interestingly enough, the change of StallGenLimit (more precisely, decreasing it from the default 50 to 25 in
order to have an earlier restart) is not giving as good results in improving the work of the algorithm the way the increase of TolFun value does. In addition, decreasing of StallGenLimit is improving the results more in the case when the TolFun value is already increased. Compare on the plots: Figure 14 for increasing the value of TolFun from the default one $1e^{-6}$ to $1e^{-3}$; Figure 15 for comparison of the values of TolFun $1e^{-3}$ and bigger; and Figure 16 for comparison of different StallGenLimit values with the fixed TolFun.

![Figure 14: Comparison of different values of TolFun](image)

The Figure 14 shows the difference between different TolFun values for the algorithm "ipop-8". Whereas for the Figure 15, there is data present for the two different algorithms: "ipop-8" and "ipop-16".

Again, it is interesting to notice that on the Figure 14 all the lines follow the same pattern at the beginning, until the first restart of the algorithm which has the TolFun value $1e^{-3}$, and then the slope there significantly increase because of the change of the population size to 8.
Continuing the explication of the plots, one can deduce from the Figure 15, where all the lines are very close by, that at some parameter value (around TolFun=1e-3) it is not very much influential to increase it to generate even earlier restarts for "ipop" while having StallGenLimit value unchanged and equal to 50. So then, based on this empirical data, it can be concluded that the optimal choice of TolFun should be approximately 5e-3 (as there can be seen a slight improvement of the algorithm with this value than the 1e-3 one on the plots) when running this kind of algorithm.
The Figure 16 includes the plots of "ipop" with various initial population sizes: 4, 8 and 16. Unfortunately, the plots that have been converted into a format that makes it possible to add them into a document are slightly cut, which sometimes makes it incomprehensible to read the actual names of the algorithms displayed. In the case of Figure 16, the missing information is about the StallGenLimit parameter. To give an idea which is where, whenever there are two lines close to each other, the one that shows in some period of times better ECDF corresponds to the algorithm with StallGenLimit=25, and the worse line - is connected to the this value being 50.

So eventually, for the StallGenLimit value, there is also even more of improvement of already improved algorithm when decreasing it from the default one 50 to 25. So the combination of tuning those two parameters gives eventually the following resulting improvement for the "ipop" algorithm. However, there are still left some parameters that have an impact. For example, in this paragraph there were used different initial population sizes for the "ipop". A comparison between them is in the next paragraph.
4.5 Comparing the different variations of "ipop" algorithm

Because the initial population size can be staying too long unchanged, which is slowing down the algorithm, it was also needed to decide with which initial population size to start the algorithm. It was another issue that was eventually answered with the benchmarking. The plot that provides the comparison of different choices is shown on the Figure 17, where the algorithm is with the value of TolFun=1e-3. As it can be seen, here as well it is unclear which one is the best one, as all them may be suitable in different cases. The different budget sizes are defining what algorithm will give a better result at the exhaustion of it - this dependency can be directly figured out from the plot. Although, if it is needed to generalize after all, it would probably be right to choose the option in the middle, for example "ipop-8".

![Figure 17: Comparison of "ipop" algorithms using different initial population sizes](image-url)

Except "ipop" there is also a concept of "bipop", where in this case the population size increases by a factor of two every other restart, otherwise it is the initial one. So, for example, if we start with population size 4, the sequence of population sizes after each restart would be 4, 8, 4, 16, 4, 32 and so on... The algorithm "bipop" was implemented as well, rather to check how relevant it can be in the given situation. In the end it gave quite similar
results as "ipop" algorithm, but without much of improvement.

Another thing that was considered is to increase "ipop" with the factor of 4 instead of factor of 2. Then, if for example, starting with population size 4, and at the moment of the first restart surpassing the domain where the value 8 would be good, it can be proceeded with population size 16 instead of 8. This algorithm actually has proved to be good idea, which is concluded from the plot on the Figure 18

Figure 18: Comparison of "ipop-4" with increase by 4 (light blue) and different TolFun values to "ipop-4" with increase by 2
5 Conclusions

Basically the purpose of my internship was to do different manipulations with the generic algorithm NSGA-II and benchmark it, obtaining also the results that might answer the question of how to improve the way to use this algorithm.

I started the internship with the already implemented in Matlab function `gamultiobj`, which corresponds to this algorithm, equipped with restarts. After obtaining the data for different population sizes, it was clear that in this case a very good option will be to use "ipop" algorithm with increasing population size. For the "ipop" the most crucial was to tune the timing of the restarts, so that the population sizes can be changed at a suitable time. For that, the parameters of the stopping criteria of `gamultiobj` function had to be changed: the results obtained were that the approximate values of parameters TolFun and StallGenLimit should be around $5e^{-3}$ and 25 accordingly. At the very final stage I also discovered that the increase by factor of 4 for the "ipop" might work even better. This was all done with the help of many comparison plots provided by the Benchmarking platform Coco.

On the last plot I would like to show how different look the plots of the simple `gamultiobj` function with all the default parameters on population sizes 8 and 128, and the plot of "ipop" with the increase by a factor of 4, initial population size 4 and all the well-tuned parameters for the stopping criteria and eventually the restarts:
Figure 19: “ipop” algorithm with increment of 4, and TolFun 5e-3 comparing to the unequipped Matlab function `gamultiobj` with population sizes 4 and 128.

The internship was very beneficial for me, and during the internship I have learned a lot of new stuff: I gained some knowledge in the setting of Multiobjective Blackbox Optimization, learned how to do Benchmarking on the special Coco platform, and as well I have learned how to use LaTex.

For the remaining plans until the finish of the internship on the 30th of September, we plan to write and submit for a conference a paper which will include some of the obtained results.
References


