Issues on the Optimisation of Evolutionary Algorithms Code

Pierre Collet[1]
CMAP
École Polytechnique
91128 Palaiseau cedex - France
Pierre.Collet@polytechnique.fr

Jean Louchet
ENSTA
25 Bd Victor
75015 Paris - France
Jean.Louchet@ensta.fr

Evelyne Lutton
INRIA
Projet FRACTALES
78150 Le Chesnay - France
Evelyne.Lutton@inria.fr

Abstract - The aim of this paper is to show that the common belief, in the evolutionary community, that evaluation time usually takes over 90% of the total time, is far from being always true. In fact, many real-world applications showed a much lower percentage. This raises several questions, one of them being the balance between fitness and operators computational complexity: what is the use of elaborating smart evolutionary operators to reduce the number of evaluations if as a result, the total computation time is increased ?

I. Introduction

In a recent research work, we compared two evolutionary libraries with the help of testbench functions[20]. During the extensive tests that were involved, we came across the fact that the time spent by an evolutionary algorithm in the evaluation/fitness function was not as high as expected.

In fact, it appeared that the overhead induced by genetic operators (selection, mutation, recombination, etc.) was far from being negligible in many cases. Therefore, it came down to us that people may not always be optimising the right piece of code.

After shortly discussing EA code optimisation, we will describe four very different evolutionary algorithms where the evaluation function takes from 98% of the total time down to 26% only.

We will then use common testbenches to try to show that in some cases, over-optimising evolutionary operators may not be a nice idea after all.

II. Optimising the EA code

Many people have been working on the optimisation of the code of Evolutionary Algorithms, with the result that many papers are available on the subject. As scientists, researchers concentrated on the optimisation of the algorithm itself using the widespread knowledge on this field. As an example, many research works were conducted on optimising the size of the population [15], [2]: Fuchs[11] uses testbenches to find the optimal population size to reduce the number of evaluations, Colin Reeves[25] tries to find the minimum population size with no performance loss.

However, with a smaller population size, convergence to local optima tends to appear rapidly. One is therefore tempted to improve (read complexify) genetic operators to fight against premature convergence [28].

All in all, most people try to minimise the number of evaluations by all possible means [24], on the fair ground that the evaluation function of their problem is extremely demanding in CPU time.

Moreover, one often refers to a received theory that the CPU time spent in the evaluation function usually accounts for more than 90% of the total time.

In fact, empirical and experimental comparisons prove just as difficult as theoretical demonstrations [33], [32]. Many factors need to be taken into account to evaluate the efficiency of a given algorithm for a given problem and many test sets have been introduced over the years [6], [1], [26], [5], [22], [9], [16], [31]. Some studies were focussed on the fitness value as an efficiency measure, other ones consider scalability [32], computational efficiency (in term of number of function evaluations), parameters setting [10], [21], quality achievable within a given time [34]. Few tests were really considering computational complexity issues, directly related to CPU time.

A recent study comparing two evolutionary libraries (GALib and EO) on testbench functions using the EASEA compiler[2] [7], [20] showed that the two libraries were giving very similar results although EO[8], a modern library making extensive use of C++ templates, was using more CPU time to handle genomes. While this was not at first a great concern, closer examination revealed that in many cases, the figure of 90% was not reached, even with relatively complex evaluation functions.

[1]European Commission IST Programme 1999-12679 (Future and Emerging Technologies).

[2]EASEA (EAsy Specification of Evolutionary Algorithms) is a language dedicated to evolutionary libraries now developed within the DREAM European Project. Its aim is to relieve the programmer of the task of learning how to use evolutionary libraries and object-oriented programming by only specifying the algorithm in a `.ez` source file.

III. Experimental results

As testbenches may not reflect reality, we looked back at real applications rather than benchmark functions. Although in some cases, evaluation took well over 90% of the total time, we discovered that in many implementations, the percentage was much smaller indeed.

In all cases, the programmers of these experiments have been trying to minimise evaluation time, sometimes by adding more information than necessary in the genome, or by using elaborate recombination or mutation operators.

The extra time spent in the evolutionary algorithm improved the exploration of the search space. However, the induced extra CPU-cost could have been used to get more evaluations done. The following experiments show that a balance must be found.

A. Inverse problem for affine IFS

Iterated functions system (IFS) theory is an important topic in the domain of fractals, mainly for image compression applications.

A major challenge of both theoretical and practical interest is the resolution of the so-called inverse problem. Except for some particular cases, no exact solution is known. Some solutions exist based on deterministic or stochastic optimization methods, but as the function to be optimised is extremely complex, most of them make some *a priori* restrictive hypotheses. Solutions based on GA or ES have been presented for affine IFS [13], [23], and more recently on a GP for non affine IFS [3].

The following test case is a simple implementation in EASEA of an EA for the resolution of the inverse problem for affine IFS, with a variable number of functions. The attractor simulation stage is however still very expensive, meaning that the fitness function takes most of the total time.

The standard EASEA output reads:

```
Score = 2130.927620  image size: 256x256
Elapsed time: 118.77s for 14516 evaluations.
Time spent in evaluation function:
      116.3s = 97.9204% of total time.
POP_SIZE=100, NB_GEN=500
```

With 97.92% of the total time spent in the evaluation function, this example behaves as expected.

B. The Fly Algorithm

This example [17], taken from Image Processing (stereovision) uses the "Parisian Evolution" paradigm, in which each individual in the population represents a part of the solution rather than the solution itself in a way similar to some artificial life simulations. It processes image data from two digital cameras in order to build a three-dimensional representation of the scene. Here, the algorithm evolves a population of "flies" which are 3-D points in space, using an Evolution Strategy.

A fly's fitness value is a measurement of the consistency of the projections of the fly on the two images so that the flies' fitness values are statistically biased in favour of those lying on the surfaces of visible objects.

Once evolved, the population of flies spreads onto the surfaces of the objects and the model of the scene is the fly population taken as a whole.

As is often the case with Parisian Evolution, population size is large, typically 5000 flies. Faster processing may be achieved using down to 1000 flies, at the expense of much less accurate results. Genetic operators are standard: ranking, gaussian mutation and barycentric crossover. On the other hand, each fitness calculation requires local image processing to correlate the neighbourhoods of the fly's projections on each image, including some projective geometry to calculate the local gradient norm and 25 pixel grey value square differences per fly.

The genome only consists of 3 integers corresponding to the three x, y, z coordinates of the fly. Our tests gave the following average values on 200 generation runs for two population sizes:

```
population = 1000
elapsed time: 1560 milliseconds
evaluation (including gradient)   78.4 %
genetic operators (incl. selection)   21.6 %

population = 5000
elapsed time: 4690 milliseconds
evaluation (including gradient)   66.0 %
genetic operators (incl. selection)   34.0 %
```

On this example, even though the chromosome is very small, the contribution of genetic operators to the total time is far from negligible, probably due to the conjugation of the selection function (ranking) and a large population.

C. Mass and Generalised Spring Physical Model Identification

Mechanical models, using point masses and generalised springs, are used in several applications such as synthetic music, man-machine gestural communication and image animation, in order to model dynamically deformable objects. They essentially use two-extremities ("binary") bonds as in Cordis-Anima [19], but may also use three-extremities ("ternary") bonds [18]. The inverse problem consisting of finding mechanical parameters (spring stiffness, lengths at rest, non-linearities, damping coef-

ficient values) in order to reproduce a given kinematic trajectory, has been first solved using a custom-designed adaptive Evolution Strategy using multiple cost functions and hybridised with a gradient descent function [18]. Recent extensions [29] allow to enrich the viscoelastic model through the addition of "muscle" bonds, i.e.: active neural-controlled bonds able to add external energy to the system.

The complexity of the task strongly depends on the number of bonds, the genome length (i.e.: the number of bond parameters to be identified), and the number of kinematic frames used in the reference sequence. Each evaluation needs a complete mechanical simulation of the sequence, involving physical forces calculations and motion integration.

Here are the results of three representative experiments:

Test sequence: `horse1`. This is a real-world sequence of a real horse walking at a trot. The horse is represented with 35 masses, 53 binary bonds (39 springs with 4 parameters and 14 dampers with 2 parameters) plus 3 ternary bonds (2 params each), giving a total chromosome length of 3840 bytes:

```
Sequence length: 53 frames.
generations: 1010
population: 1000.
184 parameters
evaluation  73.4  %
genetic operators  26.6 %
```

Test sequence: `horse2`. This example is based on the same real-world horse sequence, with 35 masses, only using binary bonds (39 springs with 4 parameters and 14 dampers with 5 params) for a chromosome length of 4464 bytes:

```
Sequence length: 53 frames.
generations: 1010
population: 500
226 parameters
evaluation  69.17 %
genetic operators   30.83 %
```

In spite of the complexity of the fitness calculation, the share of genetic operators is still high, again around 30%.

Test sequence: `muscle`. This final test uses a synthesised reference sequence generated by 3 masses on 10 images, including three "muscle" bonds (4 connection weights + 4 mechanical parameters = 8 parameters each). Unlike the two previous examples, this test sequence was devised in the framework of the development of the muscle model, but does not represent any realistic object due to its low number of masses and bonds.

```
Sequence length: 10 frames
generations: 1010
population: 2000
24 parameters
evaluation  11.03 %
genetic operators   88.97%
```

With a small size problem and a short fitness function, evaluation percentage drops down to 11 %.

D. PPSN VI

This real-world example comes from the PPSN VI conference organisation. The conference received 156 papers to be reviewed by 178 reviewers. As such a massive assignment represents several days' work for a conference organiser, it was decided that an evolutionary algorithm should be given the task to automatically assign papers to reviewers [4].

As each paper needed to be reviewed by four reviewers, each reviewer needed to read an average of $155 \times 4/178 = 3.48$ papers. It was decided that the EASEA program would assign 3 reviewers per paper, and that a fourth one would be added manually.

Such an assignment is quite difficult to construct because it is not possible to give any reviewer any paper to review. A strong constraint must not be violated (a reviewer must not review a paper of which he is an author) and many soft constraints need to be satisfied, that can be included as penalties within a fitness function:

- It is preferable that all reviewers should have 3 to 5 papers to review (rather than 0 or 12).
- It is preferable that reviewers should not know authors personally. This is of course very difficult for a computer to tell. However, e-mail addresses fields can be compared,
- The assignment should take reviewers' preferences into account (titles and abstracts were provided to the reviewers before the assignment so that they could express their preferences).
- The program should try to match papers and reviewers with a maximum of keywords in common.

All in all, four databases contained the necessary information:

1. A reviewers database, containing the name, the e-mail address, and a list of keywords corresponding to the fields in which they are competent.
2. A papers database, containing the identification number, the title, the list of authors, a list of e-mail addresses and a list of keywords.
3. A file containing the reviewer ID, along with a list of paper IDs he would rather review.

As much information was needed to attribute the penalties, the genome of each individual was quite huge:

```
Match { int KeywContribution;
        int Will_Unwill_Contribution;
        int KwMatch[4];
        int ReviewerId[4];
        int Id;
        int Nb;
        int reviewer[4];
}
Genome { Match paper[156];
         int NbPapersPerReviewer[178];
         int Distribution[10];
         int NbPapersPerReviewerContribution;
         int Will_Unwill_Contribution;
         int TotalKeywContribution;
}
```

On a Pentium architecture, one instance of the `Match` class takes 64 bytes, giving a total size of 10 748 bytes for the genome.

The best result was obtained with 5 million generations of 40 individuals, for a total of around 200 million evaluations.

In this case, evaluations were very simple (adding penalties depending on the violated constraints), and the time spent in the evaluation function only amounted to 26.4% of the total time.

As the total time was 41 hours and 40 minutes on a PENTIUM II 300 Mhz, more than 32 hours were spent on genome handling routines, mutation and recombination included.

IV. Testbenches with EASEA

The previous examples showed that it is possible to find real world problems with completely different proportions of evaluation time over total time. We therefore reverted to standard benchmarks implemented in EASEA to minutely explore this area. Many such functions are available (Sphere, Ackley Path, Griewangk, Rastrigin, Rosenbrock, Schwefel, Weierstrass, etc. [30]) and their EASEA implementation can be found on the EASEA web page (http://www-rocq.inria.fr/EASEA/).

A. Choosing test functions

Some functions are extremely simple (Sphere) and only take micro-seconds to evaluate, when more elaborate ones (Weierstrass) include a loop allowing to tune the precision of the result (and at the same time the evaluation time). Moreover, these functions can be run with different dimensions, meaning that the size of the genome can be changed while still having meaningful results.

We therefore discarded very fast test functions (like Sphere, where the evaluation time never goes beyond 10-15% of the total time) and very long functions (Weierstrass with a long loop) where evaluation time never comes under 97% of the total time.

Some functions showed the very interesting property of keeping a nearly constant evaluation percentage whatever the dimension. What happens is that as the dimension increases, the evolutionary algorithm needs to deal with larger genomes to clone, mutate, recombine, which would lead to a smaller evaluation percentage. However, as dimension increases, evaluation also takes more time, tending to counterbalance the previous effect.

Some test functions (Schwefel and Griewangk) were doubly interesting in that their nearly constant evaluation percentage was around 60-70%, which happened to be the value for the Fly and Mass/Spring model algorithms, and which in our experience seems to be a not so infrequent value.

If some odd behaviour were to appear on EAs showing this relatively high evaluation percentage, it would also appear on algorithms like PPSN, where only 26% of the total time was devoted to evaluation.

As we wanted to explore the balance between operator complexity and evaluation time, we decided to compare EAs with and without "optimised" EA operators. We therefore decided to test a very powerful tool described by Schwefel in [27]: the log-normal self-adaptive gaussian mutation operator. This smart operator —originally created for Evolution Strategies— was designed in order to explore locally, as well as globally, a bounded search space.

Log-normal mutation allows to make variable sized random jumps. It may become self-adaptive in the sense that a sigma mutation parameter is also evolved by the EA, thus adapting the exploration range of this mutation accordingly to the evolution state of each individual. This operator makes the algorithm more efficient in escaping from local maxima.

Its implementation is simple: sigma mutation parameters are added to each component of the genome, (thus doubling the genome size if each gene was a floating point value). The sigma parameters are evolved exactly in the same way as the original genome: they are recombined and mutated (gaussian mutation of fixed radius). The mutation of the original genome components is a log-normal mutation, with the corresponding variable sigma encoded in the genome.

B. Explaining the tests and the results

Schwefel's smart mutation operator drastically cuts down the number of evaluations by improving the exploration of the search space thanks to its self-adapting capabilities. The achieved effect is spectacular when very costly evaluation functions are involved: the little added time

to compute the sigma mutation parameters is counterbalanced with drastic savings in CPU evaluation time. The trade-off is however not so beneficial with our selected functions.

We conducted the following experiment to show the trade-off:

1. We ran the benchmarks on a wide range of dimensions (note the log scale) with a standard mutation operator with a *fixed number* of evaluations (24,151), and stored the results.
2. We ran the same benchmarks over the same set of dimensions with Schwefel's mutation operator *until we got a smaller or equal* fitness value (these functions must be minimised).

Griewangk with standard mutation

Dim.	Eval. #	Total	Eval. %	Fitness
10	24151	3.84s	39.1%	0.515
31	24151	4.56s	45.2%	4.187
100	24151	6.76s	54.8%	42.010
316	24151	14.10s	59.4%	252.430
1000	24151	37.13s	65.3%	843.642
3162	24151	111.24s	65.3%	3169.142
10000	24151	356.98s	64.7%	10845.621

Griewangk with adaptive mutation

Dim.	Eval. #	Total	Eval. %	Fitness
10	4351	1.21s	38.1%	0.479
31	4471	2.03s	30.0%	4.053
100	6151	5.28s	18.7%	41.120
316	8311	18.70s	14.5%	247.330
1000	12151	81.62s	14.2%	840.689
3162	16111	336.23s	13.2%	3158.232
10000	24151	1616.57s	13.3%	10820.232

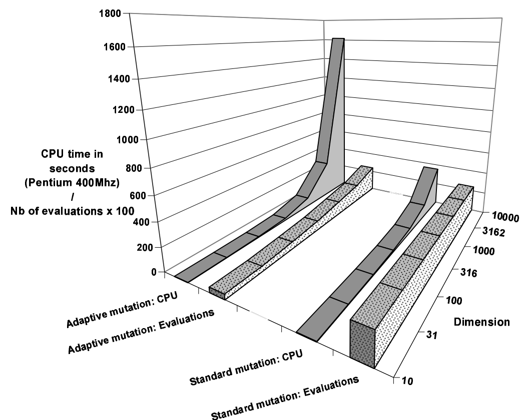

Figure 1: Tests on the Griewangk function

It is quite clear that the adaptive mutation operator works really well: it drastically reduces the number of evaluations for equal or better fitness values, and as far as *total time* is concerned, the EA is therefore much faster when the problem is relatively simple to solve (until dimension 316). However, when things get tough —i.e.: when the problem gets so difficult than the smart mutation operator is not efficient any more— having a complex and therefore costly mutation function becomes a huge handicap. For dimension 10000, adaptive mutation is more than four times slower (cf. figure 1).

If we look at the evaluation time percentage, the algorithm with the non adaptive mutator stabilises around 65% —i.e.: at about the same percentage as the Fly and Mass/Spring model algorithms— while the evaluation time percentage of the algorithm with adaptive mutation drops to 13% of the total time. This may be what happened to the PPSN VI example quoted previously.

Schwefel with standard mutation

Dim.	Eval. #	Total	Eval. %	Fitness
10	24151	3.79s	48.6%	1209
31	24151	4.45s	39.5%	7315
100	24151	6.63s	53.5%	27965
316	24151	12.74s	58.2%	114133
1000	12151	33.12s	59.8%	369134
3162	24151	97.39s	60.3%	1246990
10000	24151	313.22s	58.8%	4086970

Schwefel with adaptive mutation

Dim.	Eval. #	Total	Eval. %	Fitness
10	24151	5.11s	33.0%	1162
31	9871	3.79s	23.2%	7223
100	6031	5.17s	14.9%	27499
316	3271	7.32s	17.2%	113995
1000	3991	26.09s	13.1%	368914
3162	4351	87.55s	12.2%	1242240
10000	8311	543.43s	12.1%	4084840

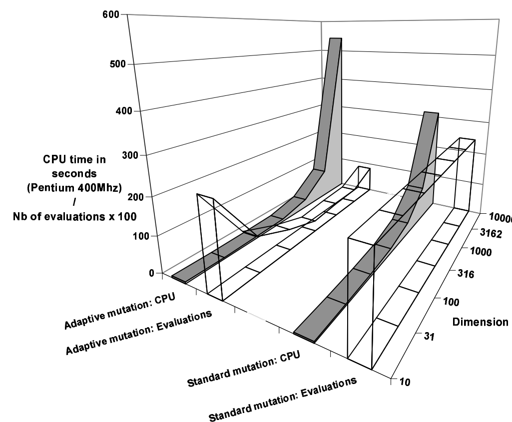

Figure 2: Tests on the Schwefel function

The symptoms look the same on Schwefel's function. Although, for a strange reason, it does not seem to handle the very simple cases very well (maybe until 31), the Schwefel adaptive mutator is most effective until and beyond dimension 3162 when again things get tougher. The simple mutation wins again beyond dimension 10000 (cf. figure 2).

As far as evaluation time percentage is concerned, we also see a big drop down to 12% of total time, while the non-adaptive function remains stable at around 59%.

V. Conclusion

When it is obvious that the fitness function contains intensive and very long calculations, (above 95% of the total time, as for the Inverse IFS problem), trying to minimise the number of evaluations is the right way to go, whatever this involves concerning the evolutionary operators. Saving one single evaluation may be worth it.

However, if the algorithm takes ages to complete and the evaluation function does not seem that CPU-consuming, great care should be taken when "optimising" the evolutionary part of the algorithm. Going for smart operators may save time, if and only if the added "intelligence" is really effective over the problem at hand. Otherwise, if the problem is simply too complex, streamlining evolutionary operators might be the right way to go, to get as many evaluations as possible and to let Darwin handle the case.

The latest version of EASEA 0.6c now displays evaluation time percentage as a default feature, so that users can have a rough idea on what part of the code they should optimise. This paper suggests that with an evaluation time below 60% of total time, "optimising" the EA by writing more complex operators may be the wrong thing to do.

Acknowledgements

We would like to thank Amine Boumaza and Bogdan Stanciulescu for the data on the Flies algorithm and the Mass/Spring Model.

Bibliography

[1] D. Ackley, "A Connexionist Machine for Genetic Hillclimbing", Kluwer Academic Publishers, 1987.

[2] J.T. Alander , "On Optimal Population Size of Genetic Algorithms", Proc. of CompEuro'92, pp65-70, IEEE Comp. Soc. Press., 1992.

[3] P. Collet, E. Lutton, F. Raynal, M. Schoenauer, Polar IFS + parisian genetic programming = efficient IFS inverse problem solving, *Genetic Programming and Evolvable Machines Journal*, 1(4):339–361, 2000. October.

[4] P. Collet, E. Lutton, M. Schoenauer, "PPSN VI Reviewer and Papers: an EASEA Match", *INRIA Research Report RR-4177*, March 2001.

[5] Y. Davidor, A Naturally Occurring Niche and Species Phenomenon: The model and first results. In L. Booker and R. Belew, eds, Proc of the Fourth ICGA, pp 257-263, Morgan-Kauffman, 1991.

[6] K. A. De Jong, "The Analysis of the behavior of a class of genetic adaptive systems", PhD Dissertation, Univ. Michigan, Ann Arbor. MI, 1975.

[7] P. Collet, E. Lutton, M. Schoenauer, J. Louchet, "Take it EASEA", PPSN VI, Paris - France, Sept. 16-20, 2000. LNCS 1917, Springer Verlag.
EASEA home page: http://www-rocq.inria.fr/EVO-Lab/ .

[8] M. Keijzer, J. J. Merelo, G. Romero, M. Schoenauer, "Evolving Objects: a general purpose evolutionary computation library", EA'01: the 5th nternational conference on Artificial Evolution. Springer Verlag 2001.
EO home page: http://www.cmap.polytechnique.fr/EO .

[9] S. Forrest, M. Mitchell, "Relative Building-Block Fitness and the Building Block hypothesis", in L. D. Whitley, ed, FOGA 2, pp 109-126, Morgan-Kauffman, 1993.

[10] O. Francois, C. Lavergne, "Design of Evolutionary Algorithms – A statistical perspectiv", IEEE Trans. on Evolutionary Computation, vol 5, Number 2, pp 129-148, Apr. 2001.

[11] M. Fuchs, "Large Population are Not Always the Best Choice in Genetic Programming", GECCO'99, Morgan-Kaufmann, pp1033-1038

[12] *GAlib home page:*
http://www.mit.edu/people/moriken/doc/galib .

[13] B. Goertzel. Fractal image compression with the genetic algorithm. *Complexity International*, 1, 1994.

[14] D. Goldberg, K. Deb, "A comparative analysis of selection schemes used in genetic algorithms" FOGA 1, pp 69-93, 1991.

[15] J. J. Grefenstette, "Optimisation of Control Parameters for Genetic Algorithms", IEEE Trans. on Systems, Man and Cybernetics, pp122-128, 1986.

[16] A. Griewangk, "Generalized descent for global optimization". Journal of optimization theory and applications. vol 34, number 1, pp 11-39. 1981.

[17] J. Louchet, "Using an Individual Evolution Strategy for Stereovision", Genetic Programming and Evolvable Machines, Vol. 2, No2, March 2001, Kluwer Academic Publishers, 101-109.

[18] J. Louchet, "An Evolutionary Algorithm for Physical Motion Analysis", British Machine Vision Conference, York, BMVA Press, pp.701-710, September 1994

[19] A. Luciani, S. Jimenez, J.L. Florens, C. Cadoz, O. Raoult, "Computational Physics: a Modeller Simulator for Animated Physical Objects", Proc. Eurographics Conference, Wien, Sept 1991, Elsevier

[20] E. Lutton, P. Collet, J. Louchet, "EASEA Comparisons on Test Functions : GALib Versus EO", EA01: 5th conference on Artifial Evolution, 2001, Le Creusot, France, September 2001.

[21] B. Miller, "Noise, Sampling, and Efficient Genetic Algorithms", IlliGAL Report No. 97001, May 1997.

[22] H. Mühlenbein, M. Schomisch, J. Born. The parallel Genetic Algorithm as Function Optimizer. In L. Booker and R. Belew, eds, Proc of the Fourth ICGA, pp 271-278, Morgan-Kauffman, 1991.

[23] D. J. Nettleton and R. Garigliano. Evolutionary algorithms and a fractal inverse problem. *Biosystems*, 33:221–231, 1994.

[24] J. Periaux, B. Mantel, M. Sefrioui, B. Stoufflet, J.-A. Desideri, S. Lanteri, N. Marco, "Evolutionary Computational Methods for complex design in aerodynamics", 36th American Institute of Aeronautics and Astronautics Conference, Reno, 1997.

[25] C. Reeves, "Using Genetic Algorithms with Small Populations", S. Forrest Editor, Proceedings of the 5th ICGA, San Mateo, California, Morgan Kaufmann, 1993.

[26] J. D. Schaffer, D. Whitley, L. J. Eshelman, R. Das, "A study of Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization", in J. D. Schaffer, ed, Proc. of the Third ICGA, pp 51-60, Morgan-Kauffman, 1989.

[27] H-P. Schwefel, "Numerical Optimization of Computer Models", Wiley, Chichester, 1981.

[28] M. Sefrioui, J. Periaux, J.G. Ganascia, "Fast Convergence Thanks to Diversity", L.J. Fogel, P.J. Angeline, T. Baeck Eds, Proceedings of the 5th Annual Conference on Evolutionary Programming, San Diego, IEEE Comp. Soc. Press, MIT Press, 1999.

[29] B. Stanciulescu, J. Louchet, "Evolving Physical Models to Understand Motion in Image Sequences", ESIT'2000, September 14-15, Aachen, Germany

[30] A test function set: http://www.geatbx.com/docu/fcnindex.html

[31] A. Torn, A. Zilinska, "Global Optimization". LNCS 350. Springer Verlag Berlin.

[32] D. Whitley, S. B. Rana, J. Dzubera, K. E. Mathias, "Evaluating Evolutionary Algorithms", Artificial Intelligence, vol 85, number 1-2, pp 245-276, 1996.

[33] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization", IEEE Trans. on Evolutionary Computation, vol 1, pp 67-83, Apr. 1997.

[34] B-T. Zhang, J-J KIM, "Comparison of Selection Methods for Evolutionary Optimization", Evolutionary Optimization, vol2, number 1, pp 55-70, 2000, citeseer.nj.nec.com/articles/zhang00comparison.html