EASEA Comparisons on Test Functions : GALib Versus EO

Evelyne LUTTON[1], Pierre COLLET[2], and Jean LOUCHET[3]

[1] Projet Fractales — INRIA, B.P. 105, 78153 Le Chesnay cedex, France,
Evelyne.Lutton@inria.fr, http://www-rocq.inria.fr/fractales/
[2] EEAAX – CMAPX — École Polytechnique, 91128 Palaiseau cedex, France,
Pierre.Collet@Polytechnique.fr, http://www.eeaax.polytechnique.fr
[3] ENSTA, 35 Boulevard Victor, 75011 PARIS, France,
Louchet@ensta.fr, http://www.ensta.fr/~louchet

Abstract. The EASEA[4] language (EAsy Specification of Evolutionary Algorithms) was created in order to allow scientists to concentrate on evolutionary algorithm design rather than implementation. EASEA currently supports two C++ libraries (GALib and EO) and a JAVA library for the DREAM. The aim of this paper is to assess the quality of EASEA-generated code through an extensive test procedure comparing the implementation for EO and GALib of the same test functions.

1 Introduction

Evolutionary algorithms are difficult to implement because of their inherent complexity: the programmer needs to create a data structure and evolve a population, using a problem-specific evaluation function and genetic operators involving choices which may be decisive to the outcome of the algorithm. Moreover, Evolutionary Algorithms are mainly used to solve or optimise complex applications in technical fields sometimes remotely connected to computer science, and scientists needing Evolutionary Algorithms do not always have the skills to implement them.

EASEA (EAsy Specification of Evolutionary Algorithms) is a language specially designed to hide away implementation complexity: the user is only asked to provide problem-specific procedural code (namely the fitness function and the crossover, mutation and initialisation operators).

While most research in the Evolutionary community is devoted to enriching the evolutionary paradigm with new features and concepts [7, 11, 15, 16], we have chosen a pragmatic, application-oriented approach with the development of the EASEA language. It comes with an EASEA compiler which converts `.ez` specification files into C++ files or JAVA files, relieving the user from the burden of programming the evolutionary algorithm.

[4] Research funded by the European Commission IST Programme 1999-12679 (Future and Emerging Technologies).

2 Presentation of EASEA

In theory, a good enough specification language would allow to implement an algorithm using any library capable of implementing the described evolutionary algorithm. Therefore, the EASEA compiler had to be able to generate code for several libraries, not only to prove its generality, but for other reasons as well:

- All libraries have different features. Let us consider GALib, EO and DREAM:
  - GALib is extensively used although its limited flexibility betrays its old design (only one mutator and Xover, no tournament replacement, ...)
  - EO offers a full object-oriented template approach allowing it to be much more versatile, although it is still a young library (v0.9xxx) and it does not naturally support distribution over islands, for instance,
  - DREAM is written over a fully distributed architecture, but it is still a rough prototype, so its evolutionary library is still very minimal.

  By supporting many different underlying libraries, EASEA users have access to a superset of all available features. If a user needs a feature absent from a library, he is directed towards the library hosting the feature.
- Supporting different libraries promotes communication between research projects: A team using GALib may recompile an EASEA file which was created for an EO environment. Results of different teams can then be compared on identical machines and environments.
- Similarly, appending an EASEA description of an algorithm to a research paper or to a web page will allow any user to recompile the program in his own environment provided one of the supported libraries is installed.

When the EASEA project started within the EVO-Lab research action, in January 1999, the GALib [5] C++ library was chosen as a first target, as it was already stable and used by many programmers around the world. The first EASEAv0.1 compiler for GALib was released in september 1999.

Then, the European EO C++ library [3] began to converge towards more stable versions, and EO was chosen as a second target. The EASEA Millennium Edition (v0.6) was the first release (in January 2001) which could indifferently generate code for either GALib or EO.

Finally, the EASEA v0.7 prototype is now able to produce JAVA code for the DREAM (Distributed Resource Evolutionary Algorithm Machine [6]).

3 Time for tests

As a consequence, the same algorithm described in a `.ez` file can be automatically converted into a source file using the GALib C++ library, the EO C++ library or the DREAM JAVA library. This unique tool raises many natural questions, among which:

1. Is the quality of the EASEA code generation equivalent for the different libraries ?

2. What about comparing the respective performances of different libraries ?
3. Is there any difference between an EASEA-coded algorithm and the same algorithm coded by a human programmer using the same library ?

Extensive tests must be elaborated to answer these basic questions.

3.1 Choosing tests

EASEA can create code for two C++ libraries and one JAVA library. As a starting point, it seems more sensible to compare the two C++ implementations, rather than to introduce other unknown factors by adding a JAVA/C++ competition, allowing us to use exactly the same compiler with the same options.

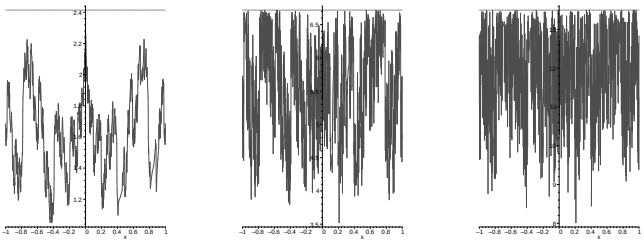
Full competition: EO and GALib are totally different libraries, with different features. A way to compare both libraries would be to use freely all available features of each library to show that solving a particular problem takes X seconds using GALib with GALib-specific algorithms and parameters, and Y seconds using EO with EO-specific algorithms and parameters. Making such a comparison would be very difficult, as it would be necessary to find the optimal way of solving the problem with GALib and demonstrate it is optimal —a very difficult task as there are clearly a great number of different ways to implement a same problem— and do the same with EO, before it is possible to compare execution times between the two libraries. Moreover, this would still rely on the quality of the code generation of the EASEA compiler: Matthew Wall may find a much more efficient way to code the problem as he knows his GALib library inside out, and the same could be said for Marc Schoenauer with the EO library, for instance. Therefore, such a test would have mixed two issues: the quality of the EASEA code generation and the capacities of the two different libraries to solve the problem. EO winning over GALib would have meant that an EASEA-EO implementation would be faster than an EASEA-GALib implementation, even though it could be possible that a Schoenauer-EO implementation would be slower than a Wall-GALib implementation.

Competition on common features: Such a competition tries to compare comparable things as we have already decided to do when we chose not to compare a C++-based library and a JAVA-based library. The idea is to pick up common features between the libraries and compare them over a set of common test functions, using the same `.ez` source file. The possibility that Wall-GALib or Schoenauer-EO implementations give much better results than EASEA-GALib or EASEA-EO implementations is then greatly reduced, as all parameters, all operators (selection, replacement) and all algorithms are imposed beforehand. In such conditions, the EASEA code generation process is very close to human-code generation as EASEA uses man-made templates. In fact, the EO template file was programmed by Marc Schoenauer, which means that apart from the fitness function and mutation/crossover operators (specific to the test function), *an*

EASEA-EO user actually has his evolutionary algorithm coded by Marc Schoenauer. As a conclusion, although the result of this testbench will indeed compare the EASEA-EO and EASEA-GALib implementation of the same evolutionary algorithm, the fact that they are so close to human implementation enables to use EASEA to actually compare the performances of the libraries.

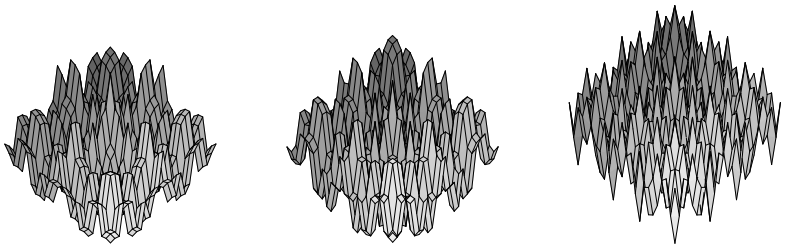
4 Weierstrass-Mandelbrot test-functions

Irregularity has been experimentally and theoretically identified as an EA "difficulty" factor. This is why "fractal" functions, such as the Weierstrass-Mandelbrot ones, have been used in [9, 8] to experimentally confirm theoretical analysis of irregularity influence on deception, taking advantage of their uniform regularity over the search space. Hölder exponents have been established as a relevant measure of irregularity and deception, and as a basis to many fractal analysis methods, especially in the domain of signal analysis.



$h = 0.5$ $h = 0.2$ $h = 0.1$

Fig. 1. 1D Weierstrass Test functions with increasing irregularities, the horizontal line represents the maximal value (attained in 0.)



$h = 0.5$ $h = 0.2$ $h = 0.1$

Fig. 2. 2D Weierstrass Test functions with increasing irregularities, the horizontal line represents the maximal value (attained in 0.)

Weierstrass-Mandelbrot functions, which are defined as:

$$W_{b,h}(x) = \sum_{i=1}^{\infty} b^{-ih} sin(b^i x) \quad \text{with } b > 1 \text{ and } 0 < h < 1$$

depend on a parameter h, which can be viewed as being the global Hölder exponent of the function (it is also equal to $2-d$, where d is the "fractal dimension" of

the graph of the function). Weierstrass-Mandelbrot functions are very irregular for small values of h, and become smoother as h tends to 1, see figures 1 and 2.

Therefore, we used these functions as controlled regularity test functions in the experiments presented below. In the case of maximisation, we compute an upper bound of $W_{b,h}$, which is $MaxVal = \sum_{i=1}^{\infty} b^{-ih}$, and thus maximise:

$$WM_{b,h}(x) = MaxVal - |W_{b,h}(x)| \text{ with } b > 1 \text{ and } 0 < h < 1$$

$W_{b,h}(x) = 0$,thus $WM_{b,h}(x)$ is always positive and is maximal at 0. with value $MaxVal$. In the minimisation case, we directly use $|W_{b,h}(x)|$.
Similarly, for a 2D search space we maximise:

$$WM2D_{b,h}(x) = 2 * MaxVal - |W_{b,h}(x)| - |W_{b,h}(y)| \text{ with } b > 1 \text{ and } 0 < h < 1$$

or minimise $|W_{b,h}(x)| + |W_{b,h}(y)|$.

4.1 Experimental results

The following experimental settings (see `weiermax.ez` in appendix) were used:

- real-encoded genome $x \in [-1., 1.]$,
- plus stragegy (population sizes and number of generation specified for each table),
- tournament selection, with tournament size 2 (labeled "T"), or Roulette Wheel, with no scaling (labeled "RW"),
- barycentric crossover,
- uniform mutation of radius σ (no label), or log normal self-adaptive gaussian mutation (labeled "ad"), [13, 14]

Results in tables 1 to 8 show the mean value over 20 runs.

5 Other test-functions

We used classical functions from `http://www.geatbx.com/docu/fcnindex.html` conveniently scaled so that their global optimum be naught (see tables 5 to 7),

6 Conclusion and future work

The numerous tests conducted in this paper allow to answer some of the questions raised in section 3:

1. All in all, both libraries give very comparable results (although EO's appear to be slightly more accurate —see AckleyPath— maybe due to different implementations of the random number generator). We were not able to explain the results of Rosenbrock 500 and Griewangk 500 where EO results are much worse than GALib (see tables 5 and 6).
   This leads us to one of the main conclusion of this paper, which has shown on a significant number of tests that *EASEA does indeed create comparable evolutionary algorithms using GALib and EO out of the same source file.*

		GALib		EO	
h	Algorithm	best value Mean(σ)	CPU time Mean(σ)	best value Mean(σ)	CPU time Mean(σ)
0.5	T	0.9996(*0.0017*)	0.2435(*0.0073*)	1.0000(*0.0000*)	0.2540(*0.0092*)
0.5	RW	1.0000(*0.0000*)	0.2480(*0.0112*)	1.0000(*0.0000*)	0.2675(*0.0109*)
0.5	T+ad	1.0000(*0.0000*)	0.2605(*0.0097*)	1.0000(*0.0000*)	0.2700(*0.0077*)
0.5	RW + ad	1.0000(*0.0000*)	0.2625(*0.0083*)	1.0000(*0.0000*)	0.2830(*0.0105*)
0.4	T	1.0000(*0.0001*)	0.2610(*0.0054*)	1.0000(*0.0000*)	0.2755(*0.0175*)
0.4	RW	1.0000(*0.0000*)	0.2615(*0.0096*)	1.0000(*0.0000*)	0.2825(*0.0141*)
0.4	T+ad	1.0000(*0.0000*)	0.2750(*0.0059*)	1.0000(*0.0000*)	0.2835(*0.0142*)
0.4	RW + ad	1.0000(*0.0000*)	0.2790(*0.0070*)	1.0000(*0.0000*)	0.2965(*0.0085*)
0.3	T	0.9902(*0.0141*)	0.2680(*0.0081*)	0.9912(*0.0123*)	0.2760(*0.0139*)
0.3	RW	0.9959(*0.0096*)	0.2675(*0.0070*)	0.9956(*0.0103*)	0.2860(*0.0086*)
0.3	T+ad	0.9917(*0.0126*)	0.2850(*0.0087*)	0.9943(*0.0110*)	0.2890(*0.0118*)
0.3	RW + ad	0.9955(*0.0085*)	0.2805(*0.0074*)	0.9963(*0.0079*)	0.3000(*0.0089*)
0.2	T	0.9998(*0.0002*)	0.2615(*0.0079*)	0.9996(*0.0009*)	0.2780(*0.0150*)
0.2	RW	0.9998(*0.0002*)	0.2635(*0.0073*)	0.9998(*0.0002*)	0.2845(*0.0120*)
0.2	T+ad	0.9999(*0.0001*)	0.2775(*0.0077*)	0.9998(*0.0002*)	0.2850(*0.0112*)
0.2	RW + ad	0.9997(*0.0002*)	0.2720(*0.0060*)	0.9998(*0.0003*)	0.2950(*0.0092*)
0.1	T	0.9998(*0.0002*)	0.2595(*0.0080*)	0.9998(*0.0001*)	0.2800(*0.0110*)
0.1	RW	0.9999(*0.0001*)	0.2600(*0.0063*)	0.9998(*0.0002*)	0.2880(*0.0133*)
0.1	T+ad	0.9998(*0.0002*)	0.2840(*0.0066*)	0.9999(*0.0001*)	0.2895(*0.0112*)
0.1	RW + ad	0.9999(*0.0002*)	0.2800(*0.0077*)	0.9999(*0.0001*)	0.3015(*0.0135*)

Table 1. Comparison of GALib and EO performances on 1D Weierstrass test functions to be **maximised**. Population size (50+40) for 50 generations (2050 evaluations). For comparison purposes, all fitness values are normalised so that the maximum is 1. for each test function. This table shows that EO is slightly slower than GALib, although the CPU time is much more variable, for comparable results.

2. This result allows to compare EO and GAlib performance:
   - The EO engine appears to be faster on tournaments than GALib but slower on RouletteWheels.
   - Genome manipulation is much slower with EO than with GALib (due to the extensive use of templates by the EO library, according to Marc Schoenauer) confirmed by a constant overhead for a given genome size. Therefore, genome length impacts EO's performance much more than GALib's.

   As a side effect, these tests show that the usual statement that the fitness function accounts for 90% of the calculation time of an evolutionary algorithm *needs to be qualified*. This can easily be seen on figure 3 : on dimension 500 where 16200 `sphere` evaluations use 3.31 seconds of the 75.6 seconds of the EO adaptive mutation algorithm (4.37% of the total time). Griewangk (also shown on figure 3) only reaches 80% in the best case (GALib non-adaptive) and the decisive 90% value is only attained for Weierstrass 500 (904s for the EA *vs* 839s for evaluation only) Therefore, the overhead induced by the library is far from being negligible on problems using millions of very fast evaluations (scheduling, ...).

		GALib		EO	
h	Algorithm	best value Mean(σ)	CPU time Mean(σ)	best value Mean(σ)	CPU time Mean(σ)
0.5	T	0.9998(*0.0005*)	2.6240(*0.0213*)	1.0000(*0.0000*)	2.5835(*0.0467*)
0.5	RW	1.0000(*0.0000*)	2.6305(*0.0150*)	1.0000(*0.0000*)	2.7830(*0.0517*)
0.5	T+ad	0.9999(*0.0002*)	2.8035(*0.0096*)	1.0000(*0.0001*)	2.6960(*0.0443*)
0.5	RW + ad	1.0000(*0.0001*)	2.8040(*0.0086*)	1.0000(*0.0000*)	2.8845(*0.0565*)
0.4	T	0.9996(*0.0012*)	2.7900(*0.0148*)	0.9995(*0.0018*)	2.7480(*0.0662*)
0.4	RW	1.0000(*0.0002*)	2.8055(*0.0150*)	1.0000(*0.0000*)	2.9395(*0.0329*)
0.4	T+ad	1.0000(*0.0001*)	2.9775(*0.0126*)	0.9999(*0.0002*)	2.8555(*0.0439*)
0.4	RW + ad	0.9998(*0.0002*)	2.9645(*0.0092*)	0.9998(*0.0001*)	3.0620(*0.0275*)
0.3	T	0.9943(*0.0072*)	2.9015(*0.0250*)	0.9988(*0.0023*)	2.7915(*0.0524*)
0.3	RW	0.9993(*0.0031*)	2.8950(*0.0112*)	0.9983(*0.0047*)	3.0050(*0.0285*)
0.3	T+ad	0.9975(*0.0045*)	3.0345(*0.0107*)	0.9966(*0.0061*)	2.9020(*0.0312*)
0.3	RW + ad	0.9974(*0.0009*)	3.0180(*0.0093*)	0.9958(*0.0051*)	3.1380(*0.0331*)
0.2	T	0.9992(*0.0011*)	2.8435(*0.0276*)	0.9994(*0.0005*)	2.7880(*0.0417*)
0.2	RW	0.9985(*0.0013*)	2.8785(*0.0115*)	0.9989(*0.0012*)	2.9880(*0.0273*)
0.2	T+ad	0.9985(*0.0014*)	2.9965(*0.0146*)	0.9983(*0.0014*)	2.8780(*0.0339*)
0.2	RW + ad	0.9965(*0.0020*)	2.9875(*0.0109*)	0.9970(*0.0020*)	3.0815(*0.0255*)
0.1	T	0.9995(*0.0003*)	2.8460(*0.0227*)	0.9992(*0.0006*)	2.8395(*0.0474*)
0.1	RW	0.9988(*0.0006*)	2.9040(*0.0227*)	0.9988(*0.0005*)	3.0115(*0.0300*)
0.1	T+ad	0.9990(*0.0005*)	3.0445(*0.0206*)	0.9988(*0.0007*)	2.9135(*0.0508*)
0.1	RW + ad	0.9986(*0.0008*)	3.0410(*0.0089*)	0.9988(*0.0007*)	3.1175(*0.0417*)

Table 2. Comparison of GALib and EO performances on 2D Weierstrass test functions to be **maximised**. Population size (150+120) for 100 generations (12150 evaluations). For comparison purposes, all fitness values were normalised so that the maximum is 1. for each test function. This table shows that EO is faster than GALib on tournaments, but slower on RouletteWheels, for comparable results.

The general conclusion of this paper is that EASEA allowed to create comparable EAs using different evolutionary libraries out of the same `.ez` files, showing that the concept is working apparently correctly. This sound basis allows to infer that more specific features of a library are equally well implemented by EASEA, which should therefore be considered as a useable EA specification language.

Future work on EASEA testbenches will try to evaluate the quality of JAVA implementations on the DREAM[6] and try to elaborate significant tests on specific features. A consequence of the present work is that potential users can find many implementation examples on the EASEA web page[1], where the EASEA Millennium Edition (v0.6c) compiler and its manual are available.

References

1. *EASEA* Millennium Edition *(v0.6c) page:* `http://www-rocq.inria.fr/EASEA/` .
2. *EVONET home page:* `http://www.evonet.polytechnique.fr` .
3. *EO home page:* `http://eodev.sourceforge.net` .
4. P. Stearns, *ALex & AYacc home page:* `http://www.bumblebeesoftware.com` , Bumblebee Software Ltd.

h	best value Mean(σ)	CPU time Mean(σ)
0.5	0.9820(*0.0096*)	0.0593(*0.0016*)
0.4	0.9712(*0.0100*)	0.0525(*0.0013*)
0.3	0.9643(*0.0072*)	0.0416(*0.0007*)
0.2	0.9996(*0.0004*)	0.0283(*0.0007*)
0.1	0.9999(*0.0001*)	0.0153(*0.0003*)

Table 3. Normalised results on a home-made Random Search on 1D Weierstrass functions for 2050 evaluations (as for table 1). GALib and EO find better results than the RS. A relatively constant overhead of nearly 0.3 seconds is added by both libraries.

h	best value Mean(σ)	CPU time Mean(σ)
0.5	0.5447(*0.0072*)	0.2021(*0.0010*)
0.4	0.4469(*0.0061*)	0.1472(*0.0016*)
0.3	0.3526(*0.0034*)	0.0895(*0.0009*)
0.2	0.2581(*0.0005*)	0.0422(*0.0005*)
0.1	0.1381(*0.0001*)	0.0126(*0.0008*)

Table 4. Normalised results on a home-made Random Search on 2D Weierstrass functions for 12150 evaluations (similar to table 2). CPU times are still very small but on 12150 evaluations, both libraries show a roughly constant overhead of nearly 3 seconds !

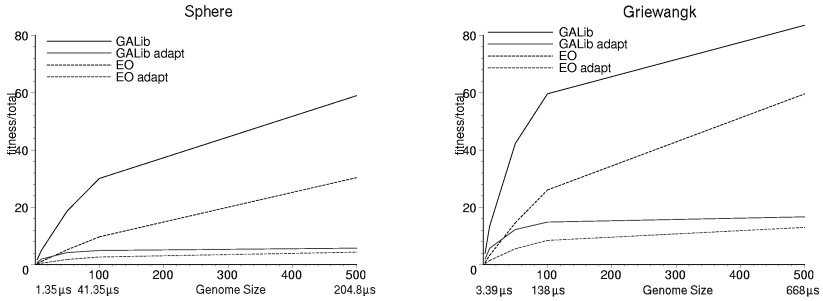


Fig. 3. Percentage of CPU time spent on fitness computation with respect to total CPU time, below the horizontal axis, CPU times for one fitness computation are specified.

5. M. Wall, *GAlib home page:* `http://www.mit.edu/people/moriken/doc/galib` .
6. B. Paechter, T. Baeck, M. Schoenauer, A.E. Eiben, J.J. Merelo, and T. C. Fogarty, "A Distributed Resource Evolutionary Algorithm Machine," Proc. of CEC 2000.
7. I. Landrieu, B. Naudts, "An Object Model for Search Spaces and their Transformations,"EA'99 conference, Springer Verlag LNCS 1829, France, 1999.
8. B. Leblanc and E. Lutton, "Bitwise regularity and GA-hardness", ICEC 98, May 5-9, Anchorage, Alaska.
9. E. Lutton and J. Lévy Véhel, "Hölder functions and Deception of Genetic Algorithms", IEEE trans. on Evolutionary computation, Vol 2, No 2, pp. 56-72, 1998.
10. Z. Michalewicz, "Genetic Algorithms + Data Structures = Evolution Programs", Springer Verlag, 1992.
11. N. J. Radcliffe, "Forma Analysis and Random Respectful Recombination," ICGA'91, pp. 222-229, 1991.
12. N. J. Radcliffe and P. D. Surry, "Fitness variance of formae and performance prediction," FOGA'95, pp. 51-72, Morgan Kaufmann publ., 1995.
13. H.-P. Schwefel, "Collective phenomena in evolutionary systems", *31st annual meeting int. society for general system research*, Vol 2, pp. 1025-1033, Budapest, 1987.

14. H.-P. Schwefel, "Numerical Optimisation of Computer Models". John Wiley & Sons, New-York, 1981. 1995 - 2nd edition.
15. P. D. Surry and N. J. Radcliffe, "Formal Algorithms + Formal Representation = Search Strategies," PPSN'96, Springer Verlag LNCS 1141, pp. 366-375, 1996.
16. P. D. Surry, "A Prescriptive Formalism for Constructing Domain-Specific Evolutionary Algorithms," PhD thesis, Univ. of Edinburgh, 1998.

APPENDIX : `weiermax.ez` file

```
\User declarations :
  #define ITER 50
  #define Abs(x) ((x) < 0 ? -(x) : (x))
  #define MAX(x,y) ((x)>(y)?(x):(y))
  #define MIN(x,y) ((x)<(y)?(x):(y))
  #define SIGMA  0.1                      /*  mutation parameter */
  double h=0.5, MaxTheo=0.;
\end
\User functions:
  double WM(double h,double y){         /* Weierstrass-mandelbrot function */
    double val=0., b=2.;
    for (int i=0;i<ITER;i++) val += pow(b,-(double)i*h) * sin(pow(b,(double)i)*y);
    return (MaxTheo - Abs(val));
  }
\end
\Initialisation function:
  if (argc>1) h = (double)atof(argv[1]);
  else {fprintf(stderr,"Holder exponent h = ? \n");  (void)scanf("%lf",&h);}
  for (int i=0;i<ITER;i++) MaxTheo+= pow(2.,-(double)i*h); // a majoration for the WM function
  fprintf(stderr,"Holder exponent h = %f    Global maximum = %f at 0.\n",h,MaxTheo);
\end
\User classes :
  GenomeClass { double x; }
\end
\GenomeClass::initialiser : // "initializer" is also accepted
  Genome.x=random(-1.,1.);
\end
\GenomeClass::crossover :
  double alpha = (double)random(0.,1.); // barycentric crossover
  if (&child1) child1.x = alpha*parent1.x + (1.-alpha)*parent2.x;
  if (&child2) child2.x = alpha*parent2.x + (1.-alpha)*parent1.x;
\end
\GenomeClass::mutator : // Must return the number of mutations
  Genome.x +=SIGMA*(double)random(-1.,1.);
  Genome.x = MAX(-1.,MIN(1.,Genome.x));             // to stay inside [-1,1]
  return 1;
\end
\GenomeClass::evaluator : // Returns the score
  return (WM(h,Genome.x));
\end
\GenomeClass::display :
  fprintf(stderr,"Best value = %f at x = %f\n",WM(h,Genome.x),x);
\end
\Default run parameters :        // Please let the parameters appear in this order
  Number of generations : 50     // NB_GEN
  Mutation probability : 1       // MUT_PROB
  Crossover probability : 1      // XOVER_PROB
  Population size : 50           // POP_SIZE
  Selection operator : Tournament // RouletteWheel, Deterministic, Ranking, Random
  Offspring population size : 80% // 40%
  Replacement strategy : Plus    //  Comma, SteadyState, Generational
  Discarding operator : Worst    //  Best, Tournament, Parent, Random
  Evaluator goal : Maximise      // Minimise
  Elitism : On                   // Off
\end
```

			GALib		EO	
n	Function	Alg.	best value Mean(σ)	CPU time Mean(σ)	best value Mean(σ)	CPU time Mean(σ)
3	Sphere	T	0.000(*0.000*)	1.268(*0.059*)	0.000(*0.000*)	6.138(*0.124*)
3	Sphere	T+ad	0.000(*0.000*)	2.912(*0.070*)	0.000(*0.000*)	13.46(*0.110*)
10	Sphere	T	0.000(*0.000*)	1.398(*0.044*)	0.000(*0.000*)	6.300(*0.077*)
10	Sphere	T+ad	0.000(*0.000*)	3.631(*0.030*)	0.000(*0.000*)	13.66(*0.166*)
50	Sphere	T	0.079(*0.019*)	1.804(*0.009*)	0.077(*0.016*)	6.502(*0.053*)
50	Sphere	T+ad	0.000(*0.000*)	8.007(*0.043*)	0.003(*0.001*)	18.60(*0.096*)
100	Sphere	T	0.328(*0.051*)	2.239(*0.026*)	0.339(*0.052*)	6.911(*0.069*)
100	Sphere	T+ad	0.046(*0.058*)	13.55(*0.089*)	0.082(*0.025*)	24.91(*0.078*)
500	Sphere	T	3.429(*0.382*)	5.627(*0.076*)	3.261(*0.293*)	10.91(*0.076*)
500	Sphere	T+ad	2.832(*0.728*)	57.48(*0.125*)	2.979(*0.521*)	75.62(*0.158*)
3	AckleyPath	T	0.000(*0.000*)	1.361(*0.032*)	0.000(*0.000*)	6.606(*0.098*)
3	AckleyPath	T+ad	0.000(*0.000*)	2.884(*0.068*)	0.000(*0.000*)	13.47(*0.129*)
10	AckleyPath	T	1.038(*0.826*)	1.557(*0.015*)	0.915(*0.672*)	6.454(*0.047*)
10	AckleyPath	T+ad	0.057(*0.251*)	3.753(*0.068*)	0.000(*0.000*)	13.91(*0.072*)
50	AckleyPath	T	6.139(*0.617*)	2.239(*0.046*)	6.043(*0.569*)	7.114(*0.074*)
50	AckleyPath	T+ad	4.249(*1.289*)	8.435(*0.080*)	4.048(*0.759*)	19.18(*0.137*)
100	AckleyPath	T	7.796(*0.386*)	2.996(*0.039*)	7.633(*0.381*)	7.910(*0.080*)
100	AckleyPath	T+ad	6.632(*1.062*)	14.26(*0.069*)	6.403(*0.900*)	25.88(*0.167*)
500	AckleyPath	T	10.02(*0.364*)	9.030(*0.082*)	9.908(*0.390*)	14.34(*0.069*)
500	AckleyPath	T+ad	10.04(*0.611*)	60.86(*0.160*)	9.885(*0.596*)	79.47(*0.279*)
3	Griewangk	T	0.003(*0.004*)	1.403(*0.037*)	0.002(*0.003*)	7.021(*0.308*)
3	Griewangk	T+ad	0.000(*0.001*)	2.892(*0.073*)	0.002(*0.003*)	15.04(*0.651*)
10	Griewangk	T	0.283(*0.157*)	1.598(*0.069*)	0.126(*0.088*)	6.566(*0.085*)
10	Griewangk	T+ad	0.053(*0.048*)	3.728(*0.101*)	0.012(*0.011*)	14.23(*0.169*)
50	Griewangk	T	8.242(*2.461*)	2.547(*0.041*)	8.129(*1.240*)	7.378(*0.055*)
50	Griewangk	T+ad	1.054(*0.149*)	8.776(*0.084*)	1.190(*0.123*)	19.36(*0.092*)
100	Griewangk	T	29.19(*4.416*)	3.744(*0.041*)	28.94(*5.486*)	8.561(*0.048*)
100	Griewangk	T+ad	5.175(*4.182*)	15.02(*0.031*)	8.097(*2.884*)	26.32(*0.103*)
500	Griewangk	T	299.8(*25.30*)	12.95(*0.061*)	720.0(*31.58*)	18.15(*0.066*)
500	Griewangk	T+ad	239.2(*74.12*)	64.79(*0.146*)	702.7(*55.35*)	83.03(*0.157*)

Table 5. Comparison on a set of test functions to be **minimised**, with a dimension (genome size) > 3. Population size (200+160) for 100 generations (16200 evaluations). The selfadaptive gaussian mutation is much slower although it gives much better results in some cases. On standard tournaments, EO is constantly 5 to 7 seconds slower than GALib, while on selfadaptive mutations, where much more work is done on the genome, EO is in average 6 seconds slower on small genomes (dim 3) and 18 seconds slower on large genomes (dim 500), showing that the overhead of EO over GALib due to genome manipulation (selfadaptation) varies from 0 to 12 seconds.

			GALib		EO	
n	Function	Alg.	best value Mean(σ)	CPU time Mean(σ)	best value Mean(σ)	CPU time Mean(σ)
3	Rastrigin	T	0.049(*0.216*)	1.313(*0.046*)	0.001(*0.004*)	7.317(*0.212*)
3	Rastrigin	T+ad	0.000(*0.000*)	2.735(*0.074*)	0.049(*0.216*)	14.81(*0.1855*)
10	Rastrigin	T	4.374(*2.145*)	1.517(*0.039*)	3.229(*1.333*)	6.667(*0.087*)
10	Rastrigin	T+ad	3.134(*2.801*)	3.577(*0.078*)	2.039(*1.315*)	14.39(*0.146*)
50	Rastrigin	T	179.5(*18.89*)	2.197(*0.042*)	172.5(*20.70*)	7.310(*0.091*)
50	Rastrigin	T+ad	23.35(*6.926*)	8.382(*0.076*)	32.86(*8.383*)	19.61(*0.182*)
100	Rastrigin	T	548.7(*32.55*)	2.945(*0.017*)	528.2(*37.45*)	8.081(*0.070*)
100	Rastrigin	T+ad	143.3(*29.55*)	14.20(*0.071*)	199.5(*37.33*)	26.12(*0.135*)
500	Rastrigin	T	4035(*84.44*)	8.971(*0.086*)	4026(*83.80*)	14.43(*0.072*)
500	Rastrigin	T+ad	2677(*656.5*)	60.78(*0.135*)	2629(*364.2*)	79.33(*0.211*)
3	Rosenbrock	T	1.221(*0.019*)	1.314(*0.024*)	1.217(*0.013*)	6.287(*0.061*)
3	Rosenbrock	T+ad	1.207(*0.000*)	2.754(*0.089*)	1.207(*0.001*)	13.38(*0.146*)
10	Rosenbrock	T	9.563(*0.882*)	1.452(*0.034*)	10.02(*0.454*)	6.381(*0.069*)
10	Rosenbrock	T+ad	9.051(*0.528*)	3.661(*0.074*)	9.115(*0.331*)	13.82(*0.098*)
50	Rosenbrock	T	94.01(*10.05*)	1.930(*0.033*)	99.54(*11.49*)	6.775(*0.038*)
50	Rosenbrock	T+ad	50.10(*0.870*)	8.153(*0.063*)	52.95(*1.987*)	18.76(*0.115*)
100	Rosenbrock	T	252.0(*24.50*)	2.475(*0.042*)	248.7(*24.03*)	7.344(*0.053*)
100	Rosenbrock	T+ad	126.7(*10.53*)	13.77(*0.091*)	144.6(*18.93*)	25.12(*0.107*)
500	Rosenbrock	T	1942(*196.9*)	6.676(*0.083*)	3809(*170.8*)	11.96(*0.047*)
500	Rosenbrock	T+ad	1846(*305.4*)	58.53(*0.128*)	3036(*538.8*)	76.72(*0.195*)
3	Schwefel	T	52.77(*67.20*)	1.345(*0.029*)	35.72(*54.22*)	6.702(*0.279*)
3	Schwefel	T+ad	41.45(*56.49*)	2.706(*0.080*)	22.56(*45.34*)	14.61(*0.203*)
10	Schwefel	T	1518(*242.1*)	1.600(*0.032*)	1626(*236.3*)	6.800(*0.144*)
10	Schwefel	T+ad	1466(*192.5*)	3.640(*0.060*)	1505(*314.7*)	14.97(*0.276*)
50	Schwefel	T	12885(*1010*)	2.510(*0.043*)	13412(*859.6*)	7.747(*0.090*)
50	Schwefel	T+ad	10372(*1255*)	8.670(*0.081*)	13163(*2078*)	20.91(*0.411*)
100	Schwefel	T	29442(*1580*)	3.580(*0.042*)	30894(*1609*)	8.860(*0.129*)
100	Schwefel	T+ad	24860(*3573*)	14.79(*0.018*)	27651(*4078*)	27.85(*0.402*)
500	Schwefel	T	182339(*4546*)	11.99(*0.077*)	183457(*3638*)	17.74(*0.128*)
500	Schwefel	T+ad	157329(*14000*)	63.82(*0.149*)	176812(*13422*)	83.69(*0.492*)
3	Weierstrass	T	0.331(*0.303*)	6.628(*0.412*)	0.312(*0.345*)	12.36(*0.205*)
3	Weierstrass	T+ad	0.242(*0.262*)	8.000(*0.149*)	0.601(*0.338*)	20.04(*0.150*)
10	Weierstrass	T	5.493(*2.093*)	18.32(*0.167*)	5.299(*1.509*)	24.31(*0.397*)
10	Weierstrass	T+ad	4.999(*1.938*)	20.42(*0.061*)	6.525(*1.288*)	32.69(*0.245*)
50	Weierstrass	T	48.87(*7.055*)	85.58(*0.132*)	37.85(*4.086*)	90.48(*0.114*)
50	Weierstrass	T+ad	41.26(*5.506*)	91.81(*0.089*)	42.21(*3.516*)	103.7(*0.117*)
100	Weierstrass	T	93.46(*11.15*)	168.7(*0.115*)	87.36(*4.366*)	173.5(*0.099*)
100	Weierstrass	T+ad	89.67(*5.229*)	181.0(*0.089*)	94.11(*5.757*)	192.7(*0.183*)
500	Weierstrass	T	539.3(*17.25*)	838.6(*3.744*)	524.9(*7.661*)	838.8(*0.133*)
500	Weierstrass	T+ad	547.4(*17.44*)	894.5(*0.147*)	553.0(*14.10*)	904.6(*0.940*)

Table 6. Comparison of GALib and EO performances on a set of test functions to be **minimised**, dimension > 3. Population size (200+160) for 100 generations (16200 evaluations). Results are fairly comparable, (although often slightly better for EO), showing that both libraries work the same way on the same settings. However, results are strangely worse and more erratic (huge sigma) for EO than for GALib on Griewangk 500 and Rosenbrock 500.

n	Function	best value Mean(σ)	CPU time Mean(σ)
3	Sphere	0.0018(*0.0013*)	0.0220(*0.0040*)
10	Sphere	0.4588(*0.0859*)	0.0720(*0.0040*)
50	Sphere	8.9766(*0.4934*)	0.3390(*0.0083*)
100	Sphere	21.8519(*0.7584*)	0.6740(*0.0080*)
500	Sphere	140.8237(*1.9817*)	3.3190(*0.0405*)
3	AckleyPath	4.7354(*0.9722*)	0.0885(*0.0036*)
10	AckleyPath	16.6226(*0.5800*)	0.1775(*0.0043*)
50	AckleyPath	20.3691(*0.0972*)	0.6960(*0.0049*)
100	AckleyPath	20.7216(*0.0458*)	1.3435(*0.0243*)
500	AckleyPath	21.0645(*0.0145*)	6.5055(*0.0619*)
3	Griewangk	0.1525(*0.0638*)	0.0555(*0.0050*)
10	Griewangk	390.2570(*5.1864*)	0.2140(*0.0111*)
50	Griewangk	781.3926(*42.4660*)	1.0760(*0.0196*)
100	Griewangk	1969.6368(*65.2537*)	2.2330(*0.0635*)
500	Griewangk	15265.3389(*139.6805*)	10.8215(*0.0576*)
3	Rastrigin	3.2252(*1.0738*)	0.0415(*0.0036*)
10	Rastrigin	65.8087(*6.6322*)	0.1330(*0.0046*)
50	Rastrigin	651.1677(*17.0618*)	0.6550(*0.0050*)
100	Rastrigin	1449.1632(*26.2853*)	1.3025(*0.0043*)
500	Rastrigin	8349.9958(*71.7607*)	6.5110(*0.0030*)
3	Rosenbrock	1.8106(*0.3473*)	0.0285(*0.0036*)
10	Rosenbrock	1934.5497(*69.1475*)	0.0910(*0.0030*)
50	Rosenbrock	8781.0553(*627.0606*)	0.4390(*0.0030*)
100	Rosenbrock	24720.0889(*1134.4771*)	0.8830(*0.0046*)
500	Rosenbrock	194602.7597(*4972.7807*)	4.3490(*0.0195*)
3	Schwefel	60.1297(*45.7104*)	0.0600(*0.0000*)
10	Schwefel	1869.5170(*159.9681*)	0.1955(*0.0050*)
50	Schwefel	15452.0258(*516.0859*)	0.9660(*0.0049*)
100	Schwefel	34211.7308(*579.2089*)	1.9310(*0.0030*)
500	Schwefel	192109.9503(*1375.4008*)	9.6355(*0.0050*)
3	Weierstrass	0.8275(*0.1597*)	5.0175(*0.0043*)
10	Weierstrass	7.5288(*0.4296*)	16.7660(*0.0482*)
50	Weierstrass	57.4280(*1.1064*)	83.7745(*0.2206*)
100	Weierstrass	125.1542(*1.6515*)	168.5590(*2.6685*)
500	Weierstrass	693.9525(*5.7427*)	839.6965(*0.1667*)

Table 7. Simple random search results for an equivalent number of function evaluations (16200) as in experiments of tables 5 and 6 (**minimisation**). This table shows that the libraries' overhead are constant for a given genome size: approximately 1.3 second for GALib for dimension 3, against 6 to 7 seconds for EO, and 60 seconds for GALib for dimension 500 and adaptive mutation against 73 seconds for GALIb. Here again, the libraries did their job by giving much better results than a simple random search.