Comparative Assessment of Algorithms and Software for Global Optimization

CHAROENCHAI KHOMPATRAPORN and **ZELDA B. ZABINSKY**
Industrial Engineering, Box 352650, University of Washington,
Seattle, Washington, 98195-2650, USA
ckhom@u.washington.edu, zelda@u.washington.edu

JÁNOS D. PINTÉR
Pinter Consulting Services Inc., and Dalhousie University,
PCS: 129 Glenforest Drive, Halifax, NS, Canada B3M 1J2
jdpinter@is.dal.ca

June 4, 2001

Abstract. The thorough evaluation of global optimization algorithms and software demands devotion, time and (hardware) resources, in addition to professional objectivity. This general remark is particularly valid with respect to global optimization (GO) software since GO literally encompasses "all" mathematical programming models. It is easy not only to fabricate very challenging test problems, but also to find realistic GO problems that pose a formidable task for any algorithm of today (and of tomorrow).

A report on computational experiments should ideally cover a large number of aspects, including detailed description and practical background of the models; related earlier work; solution approach; algorithm implementation and parameterization; hardware platform(s), operating system(s), and software environment; an exact description of all performance measures; report of successes and failures; analysis of solver parameterization effects; statistical characteristics for randomized problem-classes; and a summary of results in tabular and/or graphical forms.

An extensive inventory of classical GO test problems, as well as often much harder test suites have been suggested more recently. This paper will review several prominent test collections, discuss comparison issues, and present illustrative numerical results. A second paper will perform a comparative study using ideas presented here and discussions from the Stochastic Global Optimization workshop to be held in New Zealand, June 2001.

Key words: Global optimization; Analysis and comparison of numerical algorithms.

1. Introduction

In formulating and solving a quantitative decision model, one of the very first questions to consider is: what optimization approach and — eventually — software to use? Ideally, we want to choose the method that would be most *suitable* for solving the problem at hand. The word "suitable" could imply different interests for different users. For example, one practitioner may emphasize the speed of the algorithm to find a "good" solution, even though the solution found is not guaranteed to be the optimum. Another expert may give more attention to rigorous guarantees and to the accuracy of the solution obtained by the algorithm. Since there is clearly no one universal algorithm that performs best in all categories of optimization problems, numerical experiments to compare new algorithms and software with existing ones is an important area complementing the theoretical analysis of optimization methods. In this paper we will discuss key computational issues related to comparing global optimization strategies and solvers.

Modern optimization algorithms typically utilize iterative techniques. These algorithms, in combination with recent advances in computer technology, allow practitioners to address problems that could not have been solved otherwise. It may be beneficial to first reflect on a definition of what constitutes an algorithm. For example, Kronsjø [23] defines an algorithm as "a procedure consisting of a finite set of unambiguous rules which specify a finite sequence of operations that provides the solution to a problem, or to a specific class of problems." This definition reveals interesting features that should be emphasized:

- The algorithm must be rigorously and precisely defined so that it eliminates ambiguity.
- The algorithm must provide a solution (even if it may not be the best solution) to the problem after a finite number of steps, or equivalently after a reasonable amount of time.
- It is essential that an algorithm is applicable to classes of problems rather than just to a particular problem instance.

This list generally agrees with the suggestions by Arora [2, 3] and Bazaraa, Sherali, and Shetty [4] on the attributes of a good optimization method. More specifically, they propose that a good algorithm has following attributes:

- *Generality.* The algorithm should be insensitive to secondary details of the problem structure: that is, the algorithm should converge without any further restrictions on the structure of the problem instance (assuming, of course, that the problem-class belongs to the scope of the algorithm).
- *Efficiency*. The algorithm should (i) keep the amount of calculations as small as possible so that the actual computation time is realistic; (ii) require relatively few iterations and thus converge quickly; and (iii) be insensitive to the initial starting point and other specifications.
- *Trustworthiness*. The algorithm should be reliable and solve the given problem within a reasonable degree of accuracy, which can be specified by the user.

- *Ease of Use*. The application of the algorithm should be relatively easy to understand, even by less experienced users. The algorithm should have as few parameters that require tuning as possible.

Tradeoffs between these attributes are usually inevitable. When robustness and ease of use increase, efficiency typically decreases and vice versa. Often a computational penalty such as CPU time increase has to be paid in order to achieve a robust and easy to use algorithm [49]. On the other hand, an algorithm tailored to a specific class of problems with common structure may be more efficient for that particular problem class than an algorithm that could be applied to a general class of problems, but a lot less efficient — or even unusable — when applied to problems outside that class. Consult, for instance, the related discussions by Neumaier [33] and Pintér [38].

These attributes are also interrelated. For instance, algorithms that are very sensitive to the selection of the tuning parameters are often problem dependent, and hence not sufficiently general. Thus, there are tradeoffs between generality, efficiency, trustworthiness, and ease of use.

Once an algorithm is developed, the next phase is to demonstrate that it is a suitable method — not only in theory, but also in practice. The next section discusses how researchers have been appraising the "goodness" of optimization algorithms.

2. A Review of Comparative Characteristics

It is a challenging task to compare optimization algorithms because there are many aspects of the algorithms that need to be addressed. The first and probably the most important issue for any fair comparison is the selection of objective, empirical and reproducible measures of merit.

2.1. MEASURES OF MERIT

Based on the proposed attributes of a good optimization algorithm discussed previously, we can establish a guideline for merit measures as outlined below.

2.1.1. General Applicability

Dimensionality. The complexity of algorithms is usually interpreted in relation to the size of the problems. In general, as model dimensionality becomes larger, algorithms take longer time to solve them. A common experience is that once the dimension of the problems reaches a certain (algorithm-specific) level, the algorithm practically can no longer solve them [23]. A simple explanation of this experience is that the computational burden and the memory storage requirements, especially for global optimization problems, can dramatically increase with growing dimensionality [51]. Although global optimization problems are known to be NP-complete [52], the hope is that practitioners will be able to address large dimensional problems in some manner.

In order to acquire practical solutions to large dimensional global optimization problems, heuristic methodologies must also be considered. Heuristic and stochastic approaches that have been successfully applied to integer (linear) programming may be

adapted to (continuous) global optimization. Some of the measures such as the absolute or relative deviation from the optimum value, useful in evaluating heuristics for integer programming may also be applied to global optimization.

Local Minima. The number of local minima as well as their location and depth can affect the performance of global optimization algorithms. Problems with relatively few local minima are expected to be easier to solve than problems with many local minima. The distribution of the local minima may also influence how algorithms perform. If the local minima are closely concentrated in only a few areas on the feasible region, detection of one local minimum may easily lead to finding other local minima, whereas if the local minima are all scattered throughout the feasible region, it is generally more difficult to find the global one.

Törn, Ali, and Viitanen [50] discuss the impact of distribution of minima on algorithm performance. The phrase *embedded global minimum* case is used to describe the situation when the global minimum is close to the other local minimizers. Detecting these local minima often leads the algorithm to eventually detect the global minimum. The opposite situation is termed the *isolated global minimum* case. Local search techniques can be rightfully expected to perform poorly in the latter case.

2.1.2. Efficiency

Computation time, normally reported as program execution time or CPU time, is nevertheless an important evidence revealing how an algorithm performs on a specific problem. However, as Kronsjø [23] and numerous others caution, the execution time of an algorithm can be greatly affected by programming skills and the hardware characteristics of the machine used. Skillful changes in coding may not prominently alter the underlying algorithm, but can significantly influence the speed of program execution. Moreover, execution times of any two algorithms are not directly comparable if the timing collected are not from machines with the same specifications.

With the above in mind, there is a line of thought suggesting that instead of relying on execution time it may be better to count the number of the objective function evaluations needed by the algorithm to numerically converge to the optimum within some degree of accuracy. However, the number of the objective function evaluations also depends on programming skills. Some algorithms may also have other computational requirements (such as finding the inverse of the Hessian matrix) that are captured by solely counting objective function evaluations. Therefore a joint report combining both execution time and number of function evaluations is probably a better approach.

Thus far we have discussed the *time complexity* of an algorithm. The time complexity of an algorithm is basically the time required to execute the algorithm. There are two other types of complexity that should also be considered, namely *space complexity* and *computational complexity.* Space complexity is the memory space required by an algorithm to complete the entire execution. It is clear that a smaller memory requirement is preferred to a large memory requirement for economic reasons. It is possible that some methods applied to high dimensional problems imply huge memory requirements that exceed the limitations of most (if not all) computers. Therefore space complexity should be considered as a part of algorithm performance.

Computational complexity in practical terms refers to the number of arithmetic or logical operations that an algorithm requires to solve a given problem. In numerical computations, *algebraic* and *analytic complexity* (two branches of computational complexity) should be distinguished. Algebraic complexity indicates a known bound on the number of arithmetic operations required by the algorithm to achieve the solution. This may not be a practically usable measure in the case of GO problems because theoretical convergence typically needs an infinite number of search points and function evaluations. On the other hand, analytic complexity focuses on how much computation is needed to yield a solution with a certain degree of accuracy. Thus, in terms of computational complexity, a better algorithm is the one requiring fewer arithmetic operations to achieve a solution with a predetermined degree of accuracy.

There is also another type of complexity called *theoretical complexity.* Commonly, the theoretical complexity is categorized by the theoretical convergence. For an algorithm to be rigorous, it must guarantee convergence. The next issue is to examine the algorithm's theoretical convergence speed, referred to as the *order of convergence*. A lower order of convergence implies (at least, in theory) greater speed. Hence, algorithms with lower order of convergence are preferred over those with a higher order of convergence. Exact definitions regarding the order of convergence can be found in textbooks on nonlinear optimization, e.g. Bazaraa, Sherali, and Shetty [4], and Brent [6].

2.1.3. Trustworthiness

The trustworthiness or reliability of an algorithm is often characterized using both theoretical analysis and computational results. Rigorous methods may provide deterministic guarantees of accuracy after a finite number of iterations, but require a theoretically infinite number of iterations for absolute convergence. Discussions of rigorous GO methods can be found, for instance in [22, 32, 39].

A number of global optimization algorithms guarantee the convergence to the solution only in probability. This implies that the results obtained in a finite number of iterations or a given time may not be the optimum, and can provide (at best) *statistical* bounds on the optimal value. Of course, one needs to balance speed (execution time or number of function evaluations) against the degree of assurance regarding the results. In certain cases and applications, strict solution guarantees are indispensable. In many other situations, however, practicality dictates the acceptance of good quality solutions obtained by a limited computational effort.

Törn and Zilinskas [51] suggested a comparison by using the success ratio $g = s/m$, where m is the total number of times that the algorithm (started from random initial points) is applied to a problem, and s is the number of times that the algorithm successfully finds the optimum. Then careful numerical experiments can be conducted and a quantitative comparison can be drawn among probabilistic algorithms using the success ratio.

Note that the solution accuracy *per se* is also a numerical issue, particularly for continuous optimization problems. Algorithms should prevent rounding errors which limit the accuracy of the solution. Examples illustrating the effect of rounding errors can be found, e.g. in Brent [6].

2.1.4. Ease of Use

The effort of preparing the input data and model formulation — in order to use a given algorithm (implementation) — should also be taken into consideration [4]. An algorithm that needs extensive input data preparation, such as data sorting or complicated data format conversion, is less desirable because this can be a time consuming process.

In general, users want to be able to use optimization software as simply as possible. Algorithms that are straightforward to use are more attractive than those that are difficult to comprehend on the user level. Ease of understanding leads also to easier and proper implementation: this can enhance result reproducibility. Hence, a very important goal in algorithm and software development is that average expert or non-expert alike should be able to grasp the ideas underlying the algorithm with not much difficulty.

From a practitioner's point of view, the ideal global optimization algorithm is basically a *black box* that will output the final solution without too much preparation by the user. Therefore any algorithm that depends upon a large number of (externally set) independent parameters is less user friendly. Observe also that the introduction of too many algorithm parameters, which usually require pre-calibration or "tuning" for every new problem, indicates strong problem-dependency, and hence renders the algorithm less useful. Again, this point implies nontrivial compromises between user-friendliness and sophisticated implementations. A good compromise is often to provide preset (default) parameterizations which will work acceptably well even for the inexperienced user, while advanced users have the option of overriding these settings.

2.2. GLOBAL OPTIMIZATION MODEL TYPES AND CLASSIFICATIONS

Global optimization problems are very heterogeneous. As noted previously, GO encompasses the usual categories of mathematical programming models, specifically including both linear models and the broad nonlinear category.

There are no universally accepted ways to categorize nonlinear problems. For instance, Törn and Zilinskas [51] classified global optimization test problems as follows:

Type A: Unconstrained global optimization problems
- (A1) Solvable problems
- (A2) General unconstrained problems

Type B: Constrained global optimization problems
- (B1) Special form problems
- (B2) General constrained problems.

These authors suggested further reading of (B2) type problems in Pardalos and Rosen [34], whereas they addressed the other types of problems. This tentative classification is obviously not too sophisticated or detailed, and can be refined in many ways. Consult, for instance [20]. Also electronically available articles by Pintér [36, 37] provide concise, but fairly detailed nomenclature of GO model-types, as well as a review of the most frequently used solution approaches and GO software.

Törn, Ali, and Viitanen [50] also attempted to categorize problems from a pragmatic solution point of view regarding the number of modes and the ease/difficulty in finding them. They use the following categories:

(a) unimodal
(b) easy multimodal
(c) moderately difficult multimodal, and
(d) difficult multimodal problems.

The degree of difficulty is determined by the embeddedness of the global mimimum relative to the minimizers, and the probability of missing the region of attraction of the global minimum. Problems with isolated global minimum/minima and with smaller chance of finding the region of attraction to the global minimum are (rightfully) considered more difficult.

Especially in the past decade there has been very significant progress in GO methodology, but there are and will always remain models of extreme complexity. As Pintér [38] emphasizes, it is not difficult to construct GO problems which pose a tremendous challenge to any correct GO method, whether today or tomorrow. Luckily, many practical GO problems are much less intimidating than such purely mathematical constructions, but many practically motivated models are still very difficult. For instance, Ratschek and Rokne [40] analyzed a circuit design model described by a system of — seemingly not too complicated — nonlinear equations in just 9 variables. The verified solution of this model to a specified significant accuracy (a few years ago) took a collective work power of tens of workstations and several months of total runtime. Notwithstanding the apparent (potential) difficulty of GO models, global optimization techniques have been successfully tailored and applied to a large variety of complex engineering design problems. Examples can be found in [1, 9, 15, 17, 18, 28, 38, 41, 53].

2.3. TEST PROBLEMS

Perhaps the most comprehensive sources of global optimization test problems today are the volumes compiled by Floudas and Pardalos [14] and by Floudas et. al. [15]. The latter book is a significantly expanded version of the first one. Floudas and Pardalos [14] classify GO test problems into four main categories:

(i) quadratic programming problems
(ii) quadratically constrained problems
(iii) general nonlinear programming problems, and
(iv) real-world application problems.

Note that in this classification (i) is contained by (ii); the latter is contained by (iii); while (iv) may contain elements from any of the classes above. The second volume [15] classifies test problems into 14 (again, partly overlapping) categories all of which are mathematically related. Examples of prominent categories in the second volume are quadratically constrained problems, mixed-integer nonlinear models, semidefinite programming, as well as dynamic programming problems.

One should also note here the close theoretical connection between integer programming (IP) and continuous GO models. Integer models can be directly transformed into GO models since each disjunctive binary relation can be represented by a reverse convex continuous constraint (consult, for instance [21, 35]). This fact implies that, at least in theory, IP models may also serve to test GO strategies. Interested readers

may consult [25] as a good collection of problems closely related to the famous traveling salesman model.

There is also a significant issue to consider regarding standard academic test problems vs. real-world problems. The eventual goal of global optimization is applicability to real-world problems which could be massively nonlinear, complex, and high-dimensional, and/or have an unusual structure. Solving and reporting the results on a particular problem or a small class of real-world problems may not be useful for purposes of comparison, unless the problem is of true significance, and the test conditions and results can be directly reproduced. Academic test problems are sometimes either a bit too simplistic or have a "fabricated" structure, but others may well represent (possibly simplified) real problems. Researchers have been proposing new classes of test problems regularly: Schoen [46], Mathar and Zilinskas [27], or several randomized problem-classes discussed in Pintér [35] are examples.

To conclude this section, note that the selection of test problems in itself poses a difficult philosophical question: which problem sets can be chosen as the *true benchmark* that algorithms should be tested against? Further related discussions can be found, e.g. in [29, 33, 38].

2.4. REPORTING TEST RESULTS

Most works available in the published literature do not report all types of evaluation criteria mentioned earlier, but only a combination of some of the measures. This is probably due to the very serious resource demands of properly detailed testing. Another natural issue is scientific objectivity. Törn, Ali, and Viitanen [50] mention that comparisons of algorithms in the literature are often not quite fair, because of some, even unavoidable subjectivity and parameter tuning to the test suite. A frequently observed example is the selection of stopping criteria that are chosen in light of the (known or pre-set) solution: such choice will typically shed too favorable light on the algorithm in question.

Table 1 lists some criteria and other details that are often reported in the literature, to indicate the performance of global optimization (or other) algorithms on selected test problems. Let us discuss briefly the entries of this table. Typically, there exist some pointers available regarding the motivation to select a particular test problem, or class of problems. Test problems are sometimes chosen because they are pertinent to some real-world applications. In other cases, they are used because there exist comparable benchmark results using other algorithms. The model functional form, dimensionality, and feasible region are commonly reported, whereas the numbers of local and global minima may be unknown (in complex tests and in many practical problems), or simply omitted from the report.

The best function value found is obviously a very important measure. There are many difficult problems, such as the traveling salesman problem (TSP), molecular architecture, and over-determined systems of equations, where the optimal solutions is unknown and hence the best function value found is of primary interest.

Table 1. Performance Comparison Aspects of GO Algorithms.

ASPECTS	NOTES
1. Test Problem	Practical motivation, or original reference.
2. No. of Variables / Constraints	Smaller is often (much) easier.
3. Feasible Region	Smaller is usually (but not always) easier.
4. No. of Local Minima	Fewer is easier. More embedded is easier.
5. No. of Global Minima	More is easier if only one needs to be found. More is difficult if all have to be found.
6. Best Function Value Found	Closer to optimality is better.
7. CPU Time	Less is better.
8. No. of Function Evaluation	Fewer is better.
9. Accuracy	Higher is better.
10. Avg. No. of Iterations per Replication (in Multiple Replications)	Fewer is better.
11. No. of Replications	More is better.
12. Success Rate	Higher the better.
13. Tuning Parameters	Fewer is better. Less sensitive is better.
14. Stopping Criteria	May vary.
15. Platform	May vary.
Additional Comments	Summary, recommendations can be provided.

The CPU time and the actual number of function evaluations seem to be the other most frequently reported measures in the literature. This is probably because time is a standard unit with physical meaning. For a given hardware platform, one can compare the efficiency among algorithms by simply looking at the CPU time used by the algorithms. Different platforms can be compared by using certain benchmark evaluations, but this may lead to unwanted biases.

The accuracy of an algorithm is usually pre-set by its user. A high level of accuracy is generally preferable, but there has not yet been a common agreement on the general level of accuracy that should be used as the benchmark. In practice, the level of accuracy is dictated by the actual model and data. The average number of major iterations is a bit less often reported. One of the reasons is that this measure could be dependent also on the programming skills of the algorithm developer. Some algorithms are designed to use a large number of iterations, but in each iteration the objective function is evaluated only once, whereas other algorithms may exploit many objective function evaluations per major iteration, but need a lot fewer of such iterations. Practical applications often involve computationally expensive objective function calculations, which may dominate the computational overhead in a larger number of iterations. Thus, the total number of function evaluations is a widely accepted and relevant measure.

Deterministic methods are expected to be able to reproduce identical results. However, success rate for deterministic methods may be relevant when parameters such as initial starting point could affect the quality of the solution. The number of replications and success rate pertain to stochastic global optimization (SGO) algorithms. Since SGO algorithms guarantee finding the global optimum in probability, their robustness needs to be tested via properly chosen statistical tools.

Many numerical algorithms have a few parameters that must be “tuned” heuristically. Examples include the *mutation rate* in genetic algorithms, and the *cooling schedule* in

simulated annealing algorithms. The stopping criteria used in one algorithm often differ from those of another algorithm. For example, one can mention a pre-assigned constraint satisfaction or Kuhn-Tucker condition satisfaction accuracy. Other pragmatic numerical criteria include the maximal number of model function evaluations, the number of evaluations without "noticeable" solution improvement, and/or the maximum execution time. These parameters are usually decided somewhat arbitrarily, and this makes the comparison of algorithms difficult.

A practical issue that contributes to the complication of evaluating algorithms is the variety of available computer hardware and software platforms. Program execution times are basically incomparable, unless the algorithms are implemented on the same hardware platform with the same configurations. This issue becomes even more complicated when algorithms can utilize parallel processing on several platforms. Further differences may appear even between identical or similar algorithm implementations when using different programming languages or compilers, not to mention the added burden of user-friendly but resource-intensive program interface features.

In many cases, the performance of algorithms is depicted graphically to illustrate their progress or convergence rate as a function of iterations completed for given test problems. For the SGO approach, the average and standard deviation characteristics as well as the best and worst cases encountered are also frequently reported when solving the same model repeatedly. Such information can be useful in the statistical analysis of SGO methods.

Without going into details beyond the scope of this work, observe that the criteria listed are often (partially) conflicting. Consider, for instance, the tradeoff between accuracy required and a pre-set maximal number of objective function evaluations. In such cases, concepts and techniques used in multi-objective optimization (specifically including the selection of non-dominated, Pareto optimal algorithms for a given set of test problems) can be brought to the subject. Such analysis can lead to insights as to the strengths and weaknesses of optimization algorithms.

3. Algorithm Implementations and Comparative Numerical Experiments

It is important to recognize the difference between an algorithm and its actual software implementation. Although the latter is basically a machine-readable form of the underlying algorithm, the software itself is not the algorithm. The effectiveness of a software implementation may not directly reflect the effectiveness of the corresponding algorithm because the coding skills of the developer can greatly affect the performance of the software. However, in order to compare algorithms for a real-world setting, one needs to compare both the algorithms and their implementations, as opposed to the algorithms alone. Several important aspects of algorithm comparison have been discussed in the previous section. Here we will focus on the issues arising from algorithm comparison through software. Part of this section is based on Reklaitis, Ravindran, and Ragsdell [41].

3.1. A BRIEF HISTORY OF COMPARATIVE ASSESSMENTS IN CONSTRAINED OPTIMIZATION

In 1968, Colville [8] made a pioneering attempt at comparing the performance of algorithms by sending out eight test problems to developers of 30 nonlinear optimization codes. The participants were required to submit the result of the "best effort" on each problem and the corresponding program execution time. Characteristics of the Colville set of test problems are summarized in Table 2.

This work was obviously a significant step towards establishing objective evaluation and comparison criteria. Eason [12] observed, however that Colville's study contains three major flaws. First, the execution times collected in the study are not comparable because the effect caused by the difference in the compilers and platforms running the algorithms was not removed. Second, the participants could apply their codes as many times as they liked and only the best results were reported. Hence, if an independent investigator would like to apply the same code to the same problem, he/she may not be able to reproduce the reported results. Third, no two participants reported the same accuracy of their results since this aspect was not pre-specified.

Eason and Fenton [13] performed a comparative study of 20 optimization codes using 13 test problems, which are summarized in Table 3. The study primarily focused on penalty-type methods and all of the computations were performed on one computer. The major inadequacies of this study were due to (i) failure to include other powerful methods available of the time and (ii) shortcomings in the difficulty of the problems.

Another major comparative study of nonlinear programming (NLP) methods was implemented by Sandgren [43, 44]. The experimental procedure employed in the study was:

1. Assembly of solver codes and test problems
2. Elimination of codes using 14 preliminary test problems
3. Application of the remaining codes to the full suite of test problems
4. Removal of the test problems on which fewer than 5 codes were successful
5. Aggregation of the test results
6. Preparation of individual and composite utility curves

Table 2. Colville [8] Problem Set (Source: [41]).

	Problem Name and/or Source	Number of Variables	Number of Inequality Constraints (I)	Number of Equality Constraints (E)	Total Number of Constraints (I+E)	Total Number of Bounds on Variables
1.	Shell	5	10	0	10	5
2.	Shell	15	5	0	5	10
3.	Mylander / Res. Analysis Corp.	5	6	0	6	10
4.	Wood / Westinghouse	4	0	0	0	8
5.	Efroymson / Esso	6	4	0	4	0
6.	Huard / Electricite de France	6	0	4	4	12
7.	Gauthier / IBM France	16	0	8	8	32
8.	Colville / IBM	3	14	0	14	6

Table 3. Eason and Fenton [13] Problem Set (Source: [41]).

	Problem Name and/or Source	Number of Variables	Number of Inequality Constraints (I)	Number of Equality Constraints (E)	Total Number of Constraints (I+E)	Total Number of Bounds on Variables
1.	Colville #1	5	10	0	10	5
2.	Post office parcel problem	3	2	0	2	6
3.	Colville #3	5	6	0	6	10
4.	Colville #4	4	0	0	0	8
5.	Rosenbrock	2	0	0	0	4
6.	Colville #5	6	0	4	4	12
7.	Beightler / Journal bearing	2	1	0	1	4
8.	Siddall / Flywheel	3	2	0	2	6
9.	Siddall / Chemical reactor	3	9	0	9	4
10.	Mischke / Gear train	2	0	0	0	4
11.	Mischke / CAM design	2	2	0	2	4
12.	Eason / Mechanism synthesis	4	0	0	0	8
13.	Eason / Gear train	5	4	0	4	3

The purpose of step 2 in the experimental procedure was to avoid the possibilities of wasted effort for codes that did not have the potential to solve the full suite of the test problems. Sandgren was thus prioritizing thrustworthiness of the method over measures of performance. Sandgren's test problem set included problems 2, 7, and 8 of the Colville problem set, all 13 problems from the Eason and Fenton problem set, eight problems from the Dembo problem set which is shown in Table 4, a welded beam problem, and six industrial design application problems. The Dembo models are geometric programming problems which is a rather difficult class to solve by general NLP methods. After the removal of several test problems (in step 4 of the procedure), there were only 23 problems left. A summary of the codes used in Sandgren's study can be found in [41, 43, 44]. It is interesting to note that in Sandgren's study, the print routines were removed from the basic iterative loop so that accurate execution time of the algorithm itself can be obtained.

Table 4 : Dembo [11] Problem Set (Source: [41]).

	Problem Name and/or Source	Number of Variables	Number of Inequality Constraints (I)	Number of Equality Constraints (E)	Total Number of Constraints (I+E)	Total Number of Bounds on Variables
1.	Gibbs free energy	12	3	0	3	24
2.	Colville #3	5	6	0	6	10
3.	Alkylation process model (Bracken and McCormick)	7	14	0	14	14
4.	Practor design (Rijckaert)	8	4	0	4	16
5.	Heat Exchanger (Avriel)	8	6	0	6	16
6.	Membrane Separation (Dembo)	13	13	0	13	26
7.	Membrane Separation (Dembo)	16	19	0	19	32
8.	Beck and Ecker	7	4	0	4	14

The amount of execution time was the main performance measure used in Sandgren's study. He ranked the codes based on the relative number of problems solved within a series of specified time limits. The limits are based on a fraction of the average time for all codes on each problem. Moreover, the execution time to find the solution within a pre-specified accuracy for a problem was normalized by dividing it by the average execution time on that problem. This normalization allows direct comparison among algorithms. A generic example on Sandgren's ranking is shown in Table 5.

To read Table 5, the second column from the left indicates that Code A could solve 7 problems in 25% of the average execution time, 13 problems in 50% of the average execution time, and so on. Using the number of problems solved within specified average execution time levels as the comparison basis, fast codes can be easily identified. From the table, Codes A and B dominate by being consistently faster than Codes C and D. However, Code A is faster than Code B at the 50% of the average execution time level, while after the 75% level, Code B worked faster than Code A. Similar comparisons can be performed between other codes.

The last computational study of NLP codes discussed in this paper was performed by Schittkowski [47]. The study includes 20 codes on 180 randomly generated problems with predetermined characteristics and multiple starting points. An important difference between the Schittkowski and Sandgren studies is that quadratic programming methods were also included in Schittkowski's study [41].

The codes were evaluated based upon their 1) efficiency, 2) reliability, 3) global convergence, 4) ability to solve degenerate problems, 5) ability to solve ill-posed problems, 6) ability to solve indefinite problems, 7) sensitivity to starting points, 8) sensitivity to problem variations, and 9) ease of use. Each code was applied to all 180 test problems. The data were collected in the same manner as in the Sandgren study. Schittkowski then weighed the 9 criteria according to Saaty's priority theory [45] which is outlined by Lootsma [26]. Schittkowski's weighting scheme is listed in Table 6. Using this weighting scheme as the measures he suggested a ranking of the codes analyzed.

Several criteria in Schittkowski's weighting scheme have been mentioned in the beginning of this paper. Others may adopt different weighting factors or schemes, but this study does imply the need to consider several measures of performance. An interesting observation from this study is that it supports the theory that the reliability of a code may reflect coding skills more than the quality of the algorithm itself [41].

Table 5: Number of Problems Solved at Accuracy Level $e = 10^{-4}$ within a Percentage of Normalized Execution Time.

CODES	FRACTION OF NORMALIZED EXECUTION TIME					
	0.25[a]	0.50	0.75	1.00	1.50	2.50
Code A	7	13	14	16	16	16
Code B	0	9	14	17	19	20
Code C	0	1	2	3	4	6
Code D	0	0	0	0	3	9
⋮	⋮	⋮	⋮	⋮	⋮	⋮

[a] Times the average execution time.

Table 6: Schittkowski's [47] Weighting Scheme (Source: [41]).

Criteria	Weights
1) Efficiency	0.32
2) Reliability	0.23
3) Global Convergence	0.08
4) Ability to Solve Degenerate Problems	0.05
5) Ability to Solve Ill-posted Problems	0.05
6) Ability to Solve Indefinite Problems	0.03
7) Sensitivity to Slight Problem Variations	0.03
8) Sensitivity to Starting Points	0.07
9) Ease of Use	0.14

Future computational comparison studies for global optimization can use these and similar experiences. Specifically, they should consider multiple measures of performance, a comprehensive choice of test problems, and a well-defined procedure to conduct the evaluations. Note in this context that a large number of further test problems, such as randomized systems of nonlinear equations, random clustering problem-instances, as well as detailed case studies, are discussed in [35]. For an extensive myriad of other practically motivated tests and case studies, consult e.g. [1, 7, 9, 10, 17, 18, 19, 24, 28, 30, 31].

4. An Illustrative Comparison of Algorithms: IHR and LGO

For illustration, two global optimization methods — namely the Improving Hit-and-Run (IHR) algorithm and the Lipschitz Global Optimizer (LGO) program system — will be briefly analyzed and *qualitatively* compared. (A more detailed comparative study will appear in our forthcoming work.)

4.1. IMPROVING HIT-AND-RUN

Improving Hit-and-Run [54] is a sequential random search algorithm. It is an extension of the original Hit-and-Run algorithm developed by Smith [48]. The IHR procedure first generates a random point X_k which may depend upon the immediate previous point or several previous points. Then the algorithm generates a random direction D_k and a random step size s_k along the random direction. If the function value at the new point X_{k+1} is better than that of the previous point, i.e. $f(X_k+s_k\, D_k) < f(X_k)$, then the algorithm moves to the new point; otherwise it remains at the current point. The algorithm iterates until certain stopping criteria have been met. The basic IHR procedure is summarized below.

IHR was developed for global optimization problems where the feasible region is contained in a bounded region (typically upper and lower bounds on all variables). It has been shown that IHR will converge with probability one to the global optimum for a very broad class of problems [51]. It has also been shown that the expected number of function evaluations for quadratic problems is of order $O(n^{5/2})$. Many tests have been performed on global optimization problems with 25 or more variables. Several enhancements to IHR have been reported, for example in [42], but they will not be discussed here.

Improving Hit-and-Run Algorithm

Step 0. Initialize the iteration counter $k = 0$ and let $X_0 \in S$, $Y_0 = f(X_0)$.

Step 1. Generate a random direction vector D_k from a uniform distribution on the surface of the unit hypersphere.

Step 2. Generate a step size s_k by sampling uniformly in S along the direction vector D_k from the current point X_k.

Step 3. Update the point according to the following scheme:

$$X_{k+1} = \begin{cases} X_k + s_k D_k & \text{if } f(X_k + s_k D_k) < Y_k \\ X_k & \text{otherwise} \end{cases}$$

and set $Y_{k+1} = f(X_{k+1})$.

Step 4. Stop if the stopping criteria have met. Otherwise increase k by 1 and return to Step 1.

One feature of IHR is that it tends to make very quick improvements, and then the sequence of points slows to converge. Figure 1 shows a graph of how IHR algorithm performs for a specific maximization problem. This particular problem contains one global optimum with the corresponding objective function value of 10. In the figure, three IHR algorithm variations are compared against the number of iterations and the best objective function value found so far. The best objective function value found so far is an average of five successful replications, meaning that these representative replications must converge to a certain degree of accuracy.

From Figure 1 one can see that variation A converges the fastest until approximately the 190th iteration. After this iteration variation A starts to slowly converge and eventually performs worst after 1000 iterations. Variation C converges slowest until approximately the 270th iteration after which it starts to converge faster than variation A. In fact, after about the 450th iteration, variations B and C converge at about the same rate. Lines 1 and 2 indicate where variation A crosses variation B and variation A crosses variation C, respectively.

Figure 1 reveals two important aspects in numerical comparison of algorithms. First, it emphasizes the importance of stopping criteria. Often times, in practice the algorithms are set to stop after a certain number of iterations. This number of iterations is often arbitrarily decided by the user. Hence, some algorithms could be forced to prematurely stop even though they would perform better later on. Second, the figure stresses the consequence of the degree of accuracy. If the required degree of accuracy in the above comparison was at the average objective function value of 100 after 1000 iterations, variation A would be considered as a failure even though it converges the fastest in the beginning.

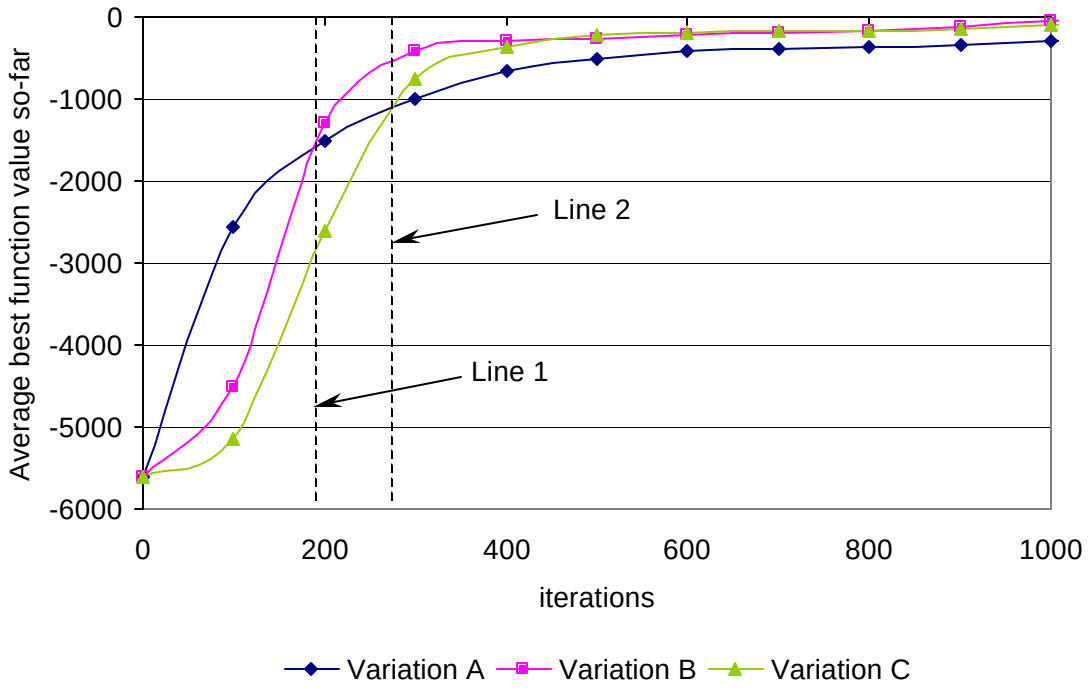

Figure 1: Performance Comparison of Three IHR Variations

4.2. THE LGO (LIPSCHITZ GLOBAL OPTIMIZER) PROGRAM SYSTEM

LGO is a professionally developed and maintained software system which (as of 2001) has been used in some 15 countries by academia and private industry. The theoretical background of LGO is discussed in [35]; note, however, that algorithmic improvements as well as numerous user interface features have been added to LGO in recent years. For a current description, consult [38]. The software has been recently peer-reviewed by Benson and Sun [5].

The core LGO solver system provides a range of *global* and *local* (nonlinear) optimization procedures that can be used in a flexible manner. One of the principal global scope solvers is an adaptive partition (branch-and-bound) algorithm outlined below. Note that for the sake of numerical efficiency, in addition to the globally convergent algorithm outlined here, efficient local search methods are also incorporated in LGO.

Considering now the entries in Table 1, the LGO software can be applied to *almost arbitrary* continuous models defined by Lipschitz functions. In practice, most (merely continuous) GO models are also handled well. To use LGO, the user needs to set only a handful parameters such as the maximum number of function evaluations, or the required constraint satisfaction accuracy. Even some of these parameters can simply be left at their default setting, in most cases, assuming that the optimization model is reasonably well-scaled. In order to enhance its efficiency, LGO generates also random sample points. The pseudo-random number generator can optionally be reset by the user, thereby enabling the reproduction of results. The current standard LGO implementation is intended to seek for a unique global solution, but customizations to find multiple solutions can also be made available.

LGO: Partition Algorithm Scheme

Step 0. Initialize()
Define
iteration count $k = 0$;
initial feasible set D;
set of active subsets $D_1 = \{D\}$
initial lower bound estimate of optimum $lb_1 = -infinity$
initial upper bound estimate of optimum $ub_1 = infinity$
current optimal solution estimate x_{opt} – *undefined*
numerical tolerance: "acceptable" discrepancy between lower and upper bounds, given by parameter *eps*.

Step k. (main iteration cycle)
Set $k = k+1$
Given
selected active partition subset: D_k
current lower bound estimate of optimum: $lb = lb_k$
current upper bound estimate of optimum: $ub = ub_k$
Execute the following steps:

Partition(D_k) (partition currently selected subset)
The partition operator leads to new subsets: D_{ki}, $i = 1, ..., i_k$, which form a partition of D_k; replace D_k by this new collection of subsets.

Do $i = 1, i_k$
    Sample(D_{ki}) (generate sample points in D_{ki})
    Update x_{opt} and $ub = f(x_{opt})$, whenever possible
    Bound(D_{ki} , $bound_{ki}$) (aggregate search information; generate optimum/bound estimate in subset D_{ki})
    Update *lb*, considering all currently active subsets and bounds
End Do

Fathom Test ($bound_i$, x_{opt}) (eliminate subsets when their lower bound exceeds $f(x_{opt})$)

Termination Test (*lb*, *ub*, *eps*): **If** $ub - lb < eps$ **Then Stop; Else Continue**

Select() (select best subset for further partition)
Go to Step k (Return to next main iteration cycle)

Stop: Report estimated solution x_{opt} and $f(x_{opt})$.

The PC version of LGO has a fully MS Windows-style version embedded into an integrated development environment (IDE), with dialogs and model visualization options). A simple "command-line" implementation; and a "silent mode" version can also be provided: the "silent" version can be seamlessly built into user applications. The two latter versions are available not only for workstations, but also can be connected to other

modeling environments: see, for instance, the LGO engine for MS Excel, as an advanced Solver option in [16]. The LGO IDE supports seamless communication with a large variety of MS Windows platforms.

In an extensive number of tests, including real-world applications received from clients, LGO has proved to be a competitive professional software product, both in terms of execution speed (number of function evaluations) and accuracy. The largest client models solved so far had hundreds of variables and constraints; the corresponding runtimes were at most in the order of hours when using moderately-fast Pentium processor based machines. For details, please consult for instance [35] and [38].

5. Conclusions

Several measures of computational performance for global optimization algorithms and software have been discussed. We propose the use of several measures in comparing methods and discuss some of the difficulties of conducting a fair comparison. Highlights of earlier studies restricted to constrained nonlinear programming have been summarized, which may be used to guide future studies on global optimization algorithms and software implementation.

In the forthcoming second part of this paper we will conduct a systematic comparative study of IHR and LGO. We invite both test problems and software products to be submitted to any of the authors to make this study as informative and representative as possible.

6. Acknowledgements

The authors would like to thank Yanfang Shen and Eva H. Dereksdottir for preparing Figure 1. The work of Zelda B. Zabinsky and Charoenchai Khompatraporn has been partially supported by NSF Grant No. DMI-9820878. The work of János D. Pintér has been partially supported by grants received from the National Research Council of Canada (NRC IRAP Project No. 362093) and from the Hungarian Scientific Research Fund (OTKA Grant No. T 034350).

References

1. Argonne National Laboratories (1993) *MINPACK-2 Test Problem Collection.* See also the accompanying notes titled “Large-scale optimization: Model problems,” by B.M. Averick and J.J. Moré. (See http://www-c.mcs.anl.gov/home/more/tprobs/html)

2. Arora, J.S. (Editor) (1997) *Guide to Structural Optimization*, ASCE Manuals and Reports on Engineering Practice No. 90, American Society of Civil Engineers, New York.

3. Arora, J.S. (1990) “Computational Design Optimization: A Review and Future Directions,” *Structural Safety 7*, 131-148.

4. Bazaraa, M.S., Sherali, H.D., and Shetty, C.M. (1993) *Nonlinear Programming: Theory and Algorithms* (Second Edition), John Wiley and Sons, New York.

5. Benson, H.P., and Sun, E. (2000) "LGO — Versatile Tool for Global Optimization," *OR/MS Today 27 (5)*, 52-55.

6. Brent, R.P. (1973) *Algorithms for Minimization without Derivatives*, Prentice-Hall, New Jersey.

7. Bomze, I.M., Csendes, T., Horst, R., and Pardalos, P.M. (Editors) (1997) *Developments in Global Optimization*, Kluwer Academic Publishers, Dordrecht / Boston / London.

8. Colville, A.R. (1968) "A Comparative Study of Nonlinear Programming Codes," *Technical Report. No. 320-2949*, IBM Scientific Center, New York.

9. Corliss, G.F., and Kearfott, R.B. (1999) "Rigorous Global Search: Industrial Applications," *Developments in Reliable Computing*, edited by Csendes, T., 1-16, Kluwer Academic Publishers, Dordrecht / Boston / London.

10. De Leone, Murli, A., Pardalos, P.M., and Toraldo, G. (Editors) (1998) *High Performance Software for Nonlinear Optimization: Status and Perspectives*, Kluwer Academic Publishers, Dordrecht / Boston / London.

11. Dembo, R.S. (1974) "A Set of Geometric Programming Test Problems and Their Solutions," Dept. Management Sci., University of Waterloo, Ontario, Canada, Working Paper.

12. Eason, E.D. (1977) "Validity of Colville's Time Standardization for Comparing Optimization Codes," *ASME Des. Eng. Tech. Conf., Paper No. 77-DET-116*, Chicago.

13. Eason, E.D., and Fenton, R.G. (1974) "A Comparison of Numerical Optimization Methods for Engineering Design," *ASME J. Eng. Ind. Ser. B 96(1)*, 196-200.

14. Floudas, C.A., and Pardalos, P.M. (1990) *A Collection of Test Problems for Constrained Global Optimization Algorithms*, Lecture Notes in Computer Science 455, Springer-Verlag, Berlin / Heidelberg / New York.

15. Floudas, C.A., Pardalos, P.M., Adjiman, C.S., Esposito, W.R., Gümüs, Z.H., Harding, S.T., Klepeis, J.L., Meyer, C.A., and Schweiger, C.A. (1999) *Handbook of Test Problems in Local and Global Optimization*, Kluwer Academic Publishers, Dordrecht / Boston / London.

16. Frontline Systems (2001) *Premium Solver Platform — Solver Engines. User Guide*, Frontline Systems, Inc., Incline Village, Nevada. (For a description of the LGO Global Solver Engine, see also http://www.frontsys.com/ lgoeng.htm)

17. Grossmann, I.E. (Editor) (1996) *Global Optimization in Engineering Design*, Kluwer Academic Publishers, Dordrecht / Boston / London.

18. Hendrix, E.M.T. (1998) *Global Optimization at Work*, Ph.D. disseration, LU Wageningen, the Netherlands.

19. Hock, W., and Schittkowski, K. (1987) *Test Examples for Nonlinear Programming Codes*, Lecture Notes in Economics and Mathematical Systems 187, Springer-Verlag, Berlin / Heidelberg / New York.

20. Horst, R., and Pardalos, P.M. (Editors) (1995) *Handbook of Global Optimization*, Kluwer Academic Publishers, Dordrecht / Boston / London.

21. Horst, R., and Tuy, H. (1996) *Global Optimization — Deterministic Approaches* (Third Edition), Springer-Verlag, Berlin / Heidelberg / New York.

22. Kearfott, R.B. (1996) *Rigorous Global Search: Continuous Problems*, Kluwer Academic Publishers, Dordrecht / Boston / London.

23. Kronsjø, L. (1987) *Algorithms: Their Complexity and Efficiency* (Second Edition), John Wiley and Sons, New York.

24. Laguna, M., and González-Velarde, J-L. (Editors) (2000) *Computing Tools for Modeling, Optimization and Simulation*, Kluwer Academic Publishers, Boston / Dordrecht / London.

25. Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., and Shmoys, D.B. (Editors) (1985) *The Traveling Salesman Problem*, John Wiley and Sons, New York.

26. Lootsma, F.A. (1980) "Ranking of Nonlinear optimization Codes According to Efficiency and Robustness," in *Konstruktive Methoden der Finiten Nichtlinearen Optimierung*, edited by Collatz L., Meinardus, G., and Wetterling, W., Birkhäuser, Basel, Switzerland, 157-158.

27. Mathar, R., and Zilinskas, A. (1994) "A Class of Test Functions for Global Optimization," *Journal of Global Optimization 5*, 195-199.

28. Mistakidis, E.S., and Stavroulakis, G.E. (1997) *Nonconvex Optimization. Algorithms, Heuristics and Engineering Applications of the F.E.M.*, Kluwer Academic Publishers, Dordrecht / Boston / London.

29. Mittelmann, H.D., and Spellucci, P. (2001) *Decision Tree for Optimization Software*. (See http://plato.la.asu.edu/guide.html)

30. Mockus, J., Eddy, W., Mockus, A., Mockus, L., and Reklaitis, G. (1996) *Bayesian Heuristic Approach to Discrete and Global Optimization*, Kluwer Academic Publishers, Dordrecht / Boston / London.

31. Moré, J.J., Garbow, B.S., and Hillström, K.E. (1981) "Testing unconstrained optimization software," *ACM Transactions on Mathematical Software 7*, 17-41.

32. Neumaier, A. (1990) *Interval Methods for Systems of Equations*, Cambridge University Press, Cambridge.

33. Neumaier, A. (2001) *Global Optimization*. (See http://solon.cma.univie.ac.at/~neum/glopt.html)

34. Pardalos P.M., and Rosen, J.B. (1987) *Constrained Global Optimization: Algorithms and Applications*, Lecture Notes in Computer Science 268, Springer-Verlag, Berlin / Heidelberg / New York.

35. Pintér, J.D. (1996a) *Global Optimization in Action*, Kluwer Academic Publishers, Dordrecht / Boston / London.

36. Pintér, J.D. (1996b) "Continuous Global Optimization Software: A Brief Review," *Optima 52*, 1-8. (For a WWW copy, see e.g. http://plato.la.asu.edu/gom.html)

37. Pintér, J.D. (1999) "Continuous Global Optimization: An Introduction to Models, Solution Approaches, Tests and Applications," *Interactive Transactions in Operations Research and Management Science 2*, No. 2. (See http://catt.bus.okstate.edu/itorms/pinter/)

38. Pintér, J. D. (2001) *Computational Global Optimization in Nonlinear Systems — An Interactive Tutorial*, Lionheart Publishing, Atlanta, GA. (See http://www.lionhrtpub.com/ books/ globaloptimization.html)

39. Ratschek, H., and Rokne, J. (1988) *New Computer Methods for Global Optimization*, Ellis Horwood, Chichester.

40. Ratscheck, H., and Rokne, J. (1993) "Experiments using Inverval Analysis for Solving a Circuit Design Problem," *Journal of Global Optimization 3*, 501-518.

41. Reklaitis, G.V., Ravindran, A., and Ragsdell, K.M. (1983) *Engineering Optimization Methods and Applications*, John Wiley and Sons, New York.

42. Romeijn, H.E., and Smith, R.L. (1994) "Simulated Annealing for Constrained Global Optimization," *Journal of Global Optimization 5*, 101-126.

43. Sandgren, E. (1977) "The Utility of Nonlinear Programming Algorithms," Ph.D. Dissertation, Purdue University, University microfilm, 300 North Zeeb Road, Ann Arbor, MI, Document No. 7813115.

44. Sandgren, E., and Ragsdell, K.M. (1980) "The Utility of Nonlinear Programming Algorithms: A Comparative Study—Part 1 and 2," *ASME J. Mech. Des. 102(3)*, 540-551.

45. Saaty, T.L. (1977) "A Scaling Method for Priorities in Hierarchical Structures," *J. Math. Psych. 15*, 234-281.

46. Schoen, F. (1993) "A Wide Class of Test Functions For Global Optimization," *Journal of Global Optimization 3*, 133-138.

47. Schittkowski, K. (1980) *Nonlinear Programming Codes: Information, Tests, Performance*, Lecture Notes in Economics and Mathematical Systems 183, Springer-Verlag, Berlin / Heidelberg / New York.

48. Smith, R.L. (1984) "Efficient Monte Carlo Procedures for Generating Random Feasible Points Uniformly Over Bounded Regions," *Operations Research 32*, 1296-1308.

49. Thanedar, P.B., Arora, J.S., Li, G.Y., and Lin, T.C. (1990) "Robustness, Generality and Efficiency of Optimization Algorithms for Practical Applications," *Structural Optimization 2*, 203-212.

50. Törn, A., Ali, M.M., and Viitanen, S. (1999) "Stochastic Global Optimization: Problem Classes and Solution Techniques," *Journal of Global Optimization 14*, 437-447.

51. Törn, A., and Zilinskas, A. (1989) *Global Optimization*, Lecture Notes in Computer Science 350, Springer-Verlag, Berlin / Heidelberg / New York.

52. Vavasis, S.A. (1995) "Complexity Issues in Global Optimization," *Handbook of Global Optimization*, edited by Horst, R. and Pardalos, P.M., 27-41, Kluwer Academic Publishers, Dordrecht / Boston / London.

53. Zabinsky, Z.B. (1998) "Stochastic Methods for Practical Global Optimization," *Journal of Global Optimization 13*, 433-444.

54. Zabinsky, Z.B., Smith, R.L., McDonald, J.F., Romeijn, H.E., and Kaufman, D.E. (1993) "Improving Hit-and-Run for Global Optimization," *Journal of Global Optimization 3*, 171-192.