A Comparison for Small-Scale Software Nonlinear Optimization Software (CSSNOS)

PRELIMINARY PROJECT OUTLINE

by A.R. Conn[1] and Ph.L. Toint[2]

Report 94/4 February 16, 1994

[1] IBM T.J. Watson Research Center, P.O.Box 218, Yorktown Heights, NY 10598, USA
Email : arconn@watson.ibm.com

[2] Department of Mathematics, Facultés Universitaires ND de la Paix, 61, rue de Bruxelles, B-5000 Namur, Belgium, EC
Email : pht@math.fundp.ac.be

Current reports available by anonymous ftp from the directory "pub/reports" on thales.math.fundp.ac.be (internet 138.48.4.14)

Keywords : Nonlinear optimization, numerical algorithms, software, testing.

A Comparison for Small-Scale Nonlinear Optimization Software (CSSNOS)
Preliminary Project Outline

A.R. Conn* Ph.L. Toint

February 16, 1994

Abstract

This note presents the outline of a project to test small-dimensional nonlinear optimization software. Both the motivation and aspects of the practical organization are discussed.

1 Motivation

The optimization literature continues publishing numerous papers devoted to the design, study or implementation of algorithms for small-scale nonlinear optimization problems. However, it is our opinion that too little is known about the relative computational merits of these approaches and methods. Furthermore, it is very difficult for a researcher with a new idea to compare their results with the best results available today for the class of problems of interest. The main reason for this difficulty is the lack of a recent comparison between available software for small-scale nonlinear optimization. Indeed, the important and much referred to work by Hock and Schittkowski [6] is now 13 years old, and, of course, did not compare many of today's most popular codes.

At the Il Ciocco NATO conference in September 1993, the idea was raised that the time might have come for another broad comparison. It was also argued that the availability of CUTE [1] would make such an enterprise realistic, both from the point of view of selecting a suitable test set and providing a relatively easy way to interface a variety of different software packages into a single environment. Discussion with other participants at the conference indicated that the idea was at least worth investigating. In particular, it was agreed that, for at least the following reasons, a new comparison of small-scale optimization software would be useful:

- it would provide to the research community an indication of the "state of the art" in this area against which new approaches could/should be compared,

*This research was supported in part by the Advanced Research Projects Agency of the Departement of Defense and was monitored by the Air Force Office of Scientific Research under Contract No F49620-91-C-0079. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

- it would encourage the research community to focus its attention on (classes of) problems that are not solved satisfactorily by any software,
- it would help in convincing practitioners and researchers from other disciplines that solving this type of problem can actually be considered as a viable alternative to other modelling approaches,
- it would provide the badly missing comparison between various contemporary codes, including some very popular ones (UNCMIN [7] and NPSOL [5], for instance).

It is the purpose of the present note to elaborate on the more practical and organizational aspects of these preliminary discussions. It is hoped that it will clarify some of the choices that have to be made in such a context and constitute a basis for the necessary preparatory debate. It does not pretend to be complete or even to touch on all important issues.

2 A general outline of the project

The idea of the CSSNOS project is currently based on the following points.

1. The comparison would be conducted (or, at least, overseen, see Section 3 below) by a relatively large group of acknowledged experts.
2. The comparison should be open to accept software submissions from all willing parties, under a format and protocol to be defined. It would be adequately publicized and timed to permit a wide participation.
3. CUTE (Constrained and Unconstrained Testing Environment) would constitute the environment in which the comparison would be conducted. This is to say that
    - the test problems used would be selected from the CUTE collection (with the possibility of adding problems to the collection),
    - the software would be compared on these problems via the use of the CUTE software interfacing tools, or via any other supplied interface capable of automatically handling the problems selected from the CUTE collection (which are in SIF, a specialized format for the input of nonlinear optimization problems [1] [2]).
4. The actual comparison process should be automated as much as possible, which would serve three purposes:
    - the actual testing environment could be transferred to potential software submitters for preparation,
    - the amount of personal intervention (and thus of bias) during the testing procedure would be kept to a minimum,
    - the same environment could be reused subsequently for possibly updating the results of the comparison in later years.

5. The primary objective of the comparison would be to test optimization software for small problems, written in Fortran and running on sequential machines. Although comparing software for large scale problems and/or on parallel machines would be of interest, it is felt that this would require much more initial investment and would suppose some (for now unavailable) consensus on how to represent the structure of large problems and/or what parallel architectures/systems are important to consider at this point. Thus is would seem natural to start with small problems and a sequential environment.

We now discuss some issues that have to be examined if a project like the one outlined above is to be organized in practice.

3 A two-level involvement

It seems practically difficult to ensure the minimum amount of bias by a suitably large group of optimizers whilst at the same time maintaining operational efficiency in managing the project. We assume the management will require a smaller group of 4 or 5 people. It is thus envisaged that involvement in the CSSNOS project could take place at two different levels.

- The *core* group would consist of a limited number of participants managing the project, working on the testing environment, running the tests and preparing the reports
- The *council* would gather a larger number of people that are willing to give opinions, suggestions and recommendations based on documents emanating from the core group, including the present text. Their role would be to ensure that the testing procedure and reporting is done with minimum bias, and thus that the final results of the projects can be accepted by the international research community as valid. Practically, this would imply writing "internal referee's reports" on the submitted documents with well-defined deadlines.

The two authors of the present text are of course open to other organizational suggestions.

A side issue is that it would be expected from the CSSNOS participants that they would submit software produced by them for the comparison, if applicable.

4 The comparison basis

It seems fairly natural to base the comparison on both reliability and efficiency of the competing software.

Reliability could be measured as the number of problems in the test set that are successfully solved by the particular piece of software. Efficiency is more difficult to determine. One idea would be to measure

- the CPU time (suitably normalized, see below),
- the number of function (and derivatives) evaluations.

All these quantities can be obtained directly from CUTE, or from the operating system primitives (on a Unix machine). It should be noted that execution times for the solution of a single small-scale problem are typically rather small, leading to possible gross inaccuracies in the evaluation of the time really needed to solve the problem, due to variation in system load and other external influences. Two ways around this difficulty seem possible:

- solve each problem several times, until the accumulated CPU time exceeds a prescribed threshold, and then compute the average running time,
- consider only the amount of CPU time needed to solve the entire test set.

In the latter case, one would expect that random variations in timing would disappear in the aggregated measure. The precise definition of how the timing should be conducted needs to be discussed further.

On the other hand, it does not seem adequate to count the number of iterations, because different software would typically be based on possibly very different algorithms, where the notion of iteration could be quite variable.

Finally, it should be remembered that testing other software features is also of interest. For instance, one might be interested to compare the level of final accuracy that can be reached by different packages. Whether this can be done in a purely automated way for a variety of packages is still unclear.

5 The test problem selection

Since the CUTE collection is the largest collection of test problems currently available, it seems adequate to choose test cases for the comparison from that particular collection. Of course, if some additional problems are judged useful or especially interesting, nothing prevents their inclusion both in CUTE and in the CSSNOS test set.

It would be important to establish a priori the list of problems considered, as potential submitters have to know on what basis their product is going to be evaluated. This selection will immediately face some difficulties.

- Problems will have to be selected on the basis of their intrinsic interest.
- A choice will have to be made for problems whose number of variables (or constraints) is parametrizable. Testing all possible instances seems excessive, and a possible compromise would be to select three particular cases of very small (below 10), small (around 50) and medium (just below 100) size.

The complexity of problem selection should not be underestimated.

It would also be interesting, as has already been pointed out by several researchers, to use the test problems not only from their standard starting points, but also from starting points situated further away from the solution. In particular, it has been suggested to perform tests from points whose components are obtained by multiplying the components of the original starting point by factors of 10 and 100. This creates a (minor) practical problem, as these variants have to be determined

- either as independent problem files,
- or by a suitable scaling of the default starting point in the software driver program,

and, moreover, this expansion may cause feasibilty problems. An important open question is what class of problems should be considered. Should the comparison be restricted to generally constrained problems, or should one also consider purely unconstrained or bound constrained problems? It seems at first sight that more than one class would be interesting, but the amount of work is of course substantially larger. Suggested classes include:

- purely unconstrained problems,
- problems with simple bound constraints,
- linearly constrained problems,
- general constrained problems,

but this classification might require further discussion.

6 The requested format for software submissions

It would be requested from potential submitters to submit

- a source for their optimization code and (ideally) suitable documentation,
- a driver program using the CUTE tools for setup and evaluations,
- any other specification file or equivalent that would be needed for correct execution of the program,
- the log of a preliminary run with the CSSNOS environment, as a proof that the supplied material fits into the testing framework.

The driver program should call (after optimization) a "report" tool. In the call to this tool, the driver program should pass

- the software name,
- a run termination code (SUCCESS, FAILED),
- the value of the objective function at the best point found.

The input-output activity of the code should be kept to a minimum (none if possible).

This submission should be done by ftp or by email, if at all possible. In all other cases, a PC diskette (MSDOS format) would be requested.

The submission should indicate to what class (amongst the predetermined classes of problems) the submitted software is applicable. It should be requested that machine dependent constants in the submitted code are set appropriately for a machine with IEEE arithmetic.

Every submission should be acknowledged

- at receipt,
- after a test run within the environment, indicating that the submission is effectively compatible with the CSSNOS environment.

Each submitted software package could be used only within the framework of the comparison project and deleted after its completion, if so requested by the submitter.

A difficulty arises in that it is expected that several packages will be presented with a potentially wide choice of algorithmic options (finite difference gradients, exact or approximate Hessians, etc). It is clearly impossible to test every possible option of each submitted package. To circumvent this problem and to somewhat limit the work load associated with testing, it would be required that

- each instance of a submitted software package is considered as a submission of its own, with a clearly identifiable name and associated option file (if any),
- at most 5 algorithmic instances of the same code could be submitted (at the choice of the submitter),
- the "default instance" of each piece of software should be included in the submitted variants.

Another difficulty is that some codes are only available in single or double precision arithmetic. It should be requested that the double precision arithmetic version be submitted, if available.

A final difficulty is that the CUTE tools are, for now, only available in Fortran, which automatically restricts the comparison to codes written in this language. This difficulty could probably be partly overcome by producing a C version of CUTE (using f2c), therefore allowing C codes to be submitted as well. However, this possibility has to be investigated in more detail before a firm commitment can be made.

7 The practical call for submissions

The project should be publicized in professional journals (MP, SIOPT, JOTA, etc) as well as in Newsletters of various types. A poster on Opt-Net and NANET will also be submitted.

Once the CSSNOS environment is ready, a delay of about 6 months would be given for potential submitters to prepare and verify their material for compatibility with the environment.

It is envisioned that the final report on the comparison will be available within 6 months after the submission deadline.

8 Some comments on the automation of the testing procedures

As indicated above, it is crucial to automate the testing procedure as completely as possible. Two successive stages can be distinguished:

- the actual testing stage,
- the comparison stage.

The testing stage is the stage at which each item of submitted software would actually be tried on the test problems and statistics of its performance recorded. The idea would be to automate the running of the successive tests within a shell script. After each run, the script would collect the relevant information

- from the information output by the "report tool",
- from the system timer,

and gather them into a single statistics record appended to a result file. There should be such a result file associated with each piece of software (softx.res, say). It is known from our experience in testing the LANCELOT package [2] that this approach is practical (see [3] and [4]).

Note that, in order to obtain numbers of function, gradient and Hessian evaluations as well as problem name and size, a new common block must be introduced in the CUTE tools to remember these. They can then be exploited by the "report tool" whose purpose is to write the final evaluation counts from the CUTE environment in a suitable form. The "report tool" should write a small summary of the current execution in a specific file (STATS.d, say). This file should contain

- the date of the test run
- the software name [from arguments]
- the problem identification (name + dimensions) [from the modified CUTE tools]
- the result of the run (SUCCESS, FAILED) [from arguments]
- the number of function evaluations [from the modified CUTE tools]
- the number of gradient evaluations [from the modified CUTE tools]
- the number of Hessian evaluations [from the modified CUTE tools]
- the value of the objective function at the solution found [from arguments].

As the notion of iteration is potentially extremely variable from one piece of software to the next, no upper limit on the number of iterations would be imposed. Handling absence of or too slow convergence should be done by requiring a uniform upper limit on the execution time.

Some suitable accuracy level must also be determined for all runs (10^{-5} both in optimality and feasibility?)

In addition to the statistics already mentioned, the environment should also monitor the cases where some runtime error occurs, causing the software to crash. One possible way to do this (on a Unix sytem) would be to simultaneously check

- the abscence of the STATS.d file,
- the fact that the used CPU time is below the limit allowed,
- the presence of a "core" file.

Occurrence of a crash would be checked after each execution and reported in the complete statistics file for the relevant software.

The comparison stage occurs when all the actual test runs are completed and the softx.res file is available for all the submitted software. The next phase of automation would then analyse the performance and reliability data contained in all those files. Again, experience with LANCELOT testing indicates that this could be achieved in two successive phases: the reliability analysis and the performance analysis.

The reliability analysis would first establish the reliability measures for each competing package and then suitably flag all unsuccessful runs and all runs converging to alternate solutions, when such solutions exist for the corresponding problem. The performance analysis is harder to define. Two types of complementary comparisons are indeed possible:

- compare the performance indices of every piece of software on a subset of test problems that have been successfully solved by a sufficiently large proportion (50%?) of the software submitted,
- rank the software according to the number of times it is best, second best, etc., with the convention that unsuccessful runs fall in the least favorable category. The number of such categories has to be established.

The evaluation procedure should automatically produce

- tables of comparative reliability and performance measures,
- tables of detailed results in LaTeX format, for inclusion in the final report,
- suitable graphics (or at least input files for a graphics package such as QUATTRO).

It is important that the automated procedures allow for an adaptable number of software packages to be tested and a changing list of test problems.

9 The computer dependent issues

All tests should be run on a single machine, or on several machines with an identical configuration and a comparable load, in order to ensure consistency of CPU times across the comparison. A Unix-based workstation with IEEE arithmetic appears to be a reasonable candidate for this choice, for several reasons.

1. It is a widely available configuration, which makes the test more realistic for the average user than if it were run on the Cray, say.

2. The Unix shell script facilities, as well as the availability of tools like awk, grep, sed, etc make this kind of platform very attractive from the point of view of setting up the CSSNOS environment.

3. Such machines are today sufficiently fast to ensure that the complete testing would not take an inordinate amount of time.

The upper limit on execution time is also dependent on the platform where the tests would be run.

Because the comparison might be updated in later years, very probably on a different platform, it is important to design an *optimization benchmark* relative to which all performances would be measured. The design of such a benchmark has to be discussed, but a possible choice would be to select a particular optimization software package (frozen at that time as to version, options etc.) and a particular set of test problems, and to consider the time required for the complete solution of these problems by this software as the desired yardstick. Of course, this benchmark and the value of the yardstick should be documented in the final report.

Note that availability of memory should not be a crucial point for the solution of small-scale problems.

10 A database of performances

An interesting potential development of the project would be to build a database where performance of various packages would be recorded, test problem by test problem. If this database is widely available to researchers, this would guarantee that new ideas could be tested on known problems and compared with the state-of-the-art performance. This possible comparison should be extremely helpful for authors in evaluating the relevance of their algorithmic ideas, and for referees in their task to assess that relevance.

Again, this development requires further thought (content, format, distribution, maintenance, submission of new data,...), but also seem rather attractive.

11 Temporary conclusions

As stated above, the purpose of this note is merely to raise a number of practical questions involved in the organization of the comparison project, as well as to clarify some of its basic options. The issues requiring further debate include

- the precise definition of the test set (and the associated problem classes),
- how to handle software written in C,
- the formal stopping conditions for the test runs,

- the definition of a suitable CPU time benchmark,
- the details of the timing procedures,
- the procedures to vary the problems' starting points,
- the presentation of the comparison's results,
- the best way to partition the workload amongst the partners in the project.

Clearly, this list is not exhaustive and (probably much) additional discussion is required before the project can be officially launched. It is thus important that interested parties manifest their opinions about this project. The authors of this note are grateful in advance for these undoubtedly fruitful contributions and look forward to an intellectually challenging and practically useful project.

A realisation of many of the ideas alluded to in this document, in the context of testing LANCELOT alone, is given in [4].

12 Aknowledgements

The ideas expressed in this note have benefited from helpful discussions with M. Biggs, A. Buckley, R. Fourer, N. Gould and J. Nocedal.

References

[1] I. Bongartz, A. R. Conn, Nick Gould, and Ph. L. Toint. CUTE: Constrained and Unconstrained Testing Environment. *ACM Transactions on Mathematical Software*, (to appear), 1994.

[2] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. *LANCELOT: a Fortran package for large-scale nonlinear optimization (Release A)*. Number 17 in Springer Series in Computational Mathematics. Springer Verlag, Heidelberg, Berlin, New York, 1992.

[3] A. R. Conn, Nick Gould, and Ph. L. Toint. Intensive numerical tests with LANCELOT (Release A): the complete results. Technical Report 92/15, Department of Mathematics, FUNDP, Namur, Belgium, 1992.

[4] A. R. Conn, Nick Gould, and Ph. L. Toint. Numerical experiments with the LANCELOT package (Release A) for large-scale nonlinear optimization. Technical Report 92/16, Department of Mathematics, FUNDP, Namur, Belgium, 1992.

[5] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. User's guide for NPSOL (version 4.0): A Fortran package for nonlinear programming. Technical Report SOL86-2, Department of Operations Research, Stanford University, Stanford, California 94305, USA, 1986.

[6] W. Hock and K. Schittkowski. *Test Examples for Nonlinear Programming Codes*. Springer Verlag, Berlin, 1981. Lectures Notes in Economics and Mathematical Systems 187.

[7] J.E. Koontz, R.B. Schnabel, and B.E. Weiss. A modular system of algorithms for unconstrained minimization. *ACM Transactions on Mathematical Software*, 11:419–440, 1985. Also available as Technical Report CU-CS-240-82, Department of Computer Science, University of Colorado, Boulder, CO.