2.1 Test Problems

In any computational study, it is important to obtain a wide range of test problems on which the study should be performed. It should be pointed out that unlike the case of its linear counterpart, in nonlinear integer programming, the size of a problem is only one of many factors in determining its overall complexity and this factor is often dominated by the problem structure. The degree of nonlinearity of the objective and the constraint functions contributes significantly to the complexity of the problem.

Since to our knowledge, no experimental study of this nature has been carried out, we considered, among others, test problems from the various past studies on nonlinear continuous algorithms such as Colville, Eason and Fenton, Himmelblau, and Sandgren and Ragsdell. Among the available test problems, we considered only those problems where certain variables can be treated as integer variables; e.g., those problems having some of the variables with reasonably tight bounds and feasible regions with integer points. It serves no purpose to consider those problems that have no integer points in their feasible region or where the variables take on large values. A total of fourteen real-world problems were selected from these studies as well as other literature sources. We also generated a set of seven additional test problems using the technique of Rosen and Suzuki. One problem which was intuitively constructed was also taken as a test problem. Therefore, we have a total collection of 22 test problems. The test problems range from two to sixteen continuous variables, and two to ten integer variables. Out of 22 problems, four were of the mixed nonlinear integer programming type, and the remaining were pure nonlinear integer programming problems. The number of constraints varied from zero to eight. Additional problems of greater difficulty were also considered, but the excessive computational cost made it impractical for us to use them as test problems. A complete Fortran listing of test problems can be found in Gupta and Omprakash.

3. RESULTS AND ANALYSIS

Each of the 27 strategies was tested on each of the 22 test problems on the CDC 6500 Computer system at Purdue University. Except for one problem, labeled problem 19, each strategy was able to solve every problem in less than 240 seconds of execution time. Problem 19 turned out to be more difficult to solve, as two of the strategies (no. 4 and 6) could not find optimal solutions within a time limit of 500 seconds. Two other strategies (no. 24 and 27) generated subproblems for which the code OPT even failed to find continuous optimal solutions. Although the remaining 23 strategies could find optimal solutions to this problem, some strategies took almost 500 seconds to reach the final solution. The fact that four strategies were unable to reach the final solutions to one of the 22 test problems should not underestimate either the importance of these strategies or the code OPT itself for they were successful on all other problems.

3.1 A Strategy Ranking Criterion

Since the significant cost of executing a nonlinear integer program is the solution time, we decided to use it as the initial indicator of the performance of the strategies. It was found that our test problems had a wide variation in their execution times that ranged from less than 0.5 second to almost 500 seconds. This paper presents the average solution times, the worst solution times and the best solution times. It also gives the ratio of the worst to the best solution times.

Normalized Solution Time (NST)

As we would like to consider all the test problems equally important for our studies, the following procedure was used to normalize the solution time.

1. Let t_{ij} denote the solution time for the i-th strategy to solve the j-th problem, and n_j denote the number of strategies that were able to reach the final optimal solution for the j-th problem within a specified time limit of 500 seconds. $(i = 1, 2, \ldots, 27;\ j = 1, 2, \ldots, 22)$.

2. Form the sum $S_j = \sum_i t_{ij}$ where the sum is taken over all those strategies i which could solve the problem j; thus, the average solution time for the j-th problem $A_j = S_j/n_j \quad (j = 1, 2, \ldots, 22)$.

3. Form the normalized solution times (T_{ij}) for the i-th strategy to solve the j-th problem as:

$$T_{ij} = t_{ij}/A_j$$

Since the average nomalized solution time for each problem is one unit, all the test problems are now comparable as equally important problems, and we would therefore be able to make direct comparisons. It is clear that a strategy with lower (less than one) normalized solution times would have a better rating than a strategy with relatively higher (greater than one) normalized solution times. The following procedure is used to evaluate the performance of the strategies:

1. First compute the normalized solution times (NST_{ij}) for each strategy and test problem.

2. Form the sum $ST_i = \sum_j NST_{ij}$, and the average $AV_i = ST_i/N_i$, where N_i is the number of problems solved by strategy i.

3. Rank the average normalized solution times for each strategy, the best strategy being the one with the lowest value of AV_i.

The ranking and average nomalized solution times are shown in this paper.

The following observations can be made:

1. None of the top 12 strategies invokes either branching from the newest node or implementation of heuristic A. As a matter of fact, these 12 strategies are formed by the exclusive and exhaustive combination of all the other selection strategies (A 3x2x2 design).

2. The top four strategies invoke the most fractional integer variable as the branching variable selection strategy, and the next eight strategies use the other two branching variable selection strategies. The pattern of the most fractional integer variable selection strategy being better than the other two strategies continues when a combination of node selection and heuristic selection is fixed, and therefore we may conclude that most fractional integer variable strategy is a relatively better strategy than the other two strategies for the variable selection.

Analysis of Variance

In order to make statistical comparisons, the statistical package SPSS was used to analyze the data provided by the normalized solution times for the purpose of the analysis of variance. Before considering the table of analysis of variance, it is of interest to examine the overall mean values of the three selection parameters. The results shown in this paper are clearly consistent with the observations made earlier. Most fractional integer variable selection strategy performs the best among the branching variable selection strategies. While the other two branching variable selection strategies perform about the same. Among the node selection strategies, selecting from the newest node turns out to be the worst. The strategy using estimations seems to perform better than the one selecting the node with the lowest bound.

With these broad observations in mind, the Analysis of Variance can be examined in Table 1. The results shown there indicate that significant differences exist among the means of solution times for all the three main effects of selection strategies. This was only to be expected in light of the preceding discussion. The table also indicates that there are no significant differences due to two-way interactions. This shows that the three selection parameters are independent of one another.

4.2 Another Strategy Ranking Criterion

In the above ranking procedure, we have compared the normalized solution times which are computed from their corresponding actual solution times. The solution time often depends on the following:

BRANCH AND BOUND EXPERIMENTS IN NONLINEAR MIXED INTEGER PROGRAMMING

Omprakash K. Gupta

Washington State University
Pullman, Washington 99164

and

A. Ravindran

Purdue University
School of Industrial Engineering
West Lafayette, Indiana 47907

The branch and bound principle has long been established as an effective computational tool for solving mixed integer linear programming problems. This paper investigates the computational feasibility of branch and bound methods in solving nonlinear mixed integer programming problems. The efficiency of a branch and bound method often depends on the rules used for selecting the branching variables and branching nodes. Among others, the concepts of pseudo-costs and estimations are implemented in selecting these parameters. Since the efficiency of the algorithm also depends on how fast an upper bound on the objective minimum is attained, heuristic rules are developed to locate an integer feasible solution to provide an upper bound. The different criteria for selecting branching variables, branching nodes and heuristics form a total of 27 branch and bound strategies. These strategies are computationally tested and the empirical results are presented.

1. EXPERIMENTAL DESIGN

The primary variables of interest in the experimental work were the branching variable selection and node selection strategies. Three strategies for selecting the branching variable were (1) lowest index-first, (2) most fractional integer variable and (3) use of pseudo costs. Three strategies for selecting the branching node were (1) Branch from node with lowest bound, (2) Branch from the newest node and (3) the use of estimators. The three strategies of each of these two selection categories formed a combination of nine strategies. With each combination, we implemented three options on the heuristics to provide an upper bound: (1) no heuristic, (2) heuristic A, and (3) heuristic B. Thus we had a total of 27 branch and bound strategies. This comprises an experimental design with three levels of treatment. A computer code BBNLMIP was developed capable of invoking each of the 27 strategies.

the procedures examined in this Section are polynominal algorithms of the same degree. However, the computational results indicate marked difference in running times among procedures. For this reason, the authors examined how computational time T varies with n, the number of nodes, by statistically fitting a curve of the form $T = an^b$ to the data using least squares.

Section 5 focuses on the comparison of three or more heuristics at one time. A discussion of how to apply the Friedman test, a nonparametric analog to the classical ANOVA test of homogeneity is presented. An expected utility approach is also tested. The assymetric or directed TSP is studied in detail in Section 6. In Section 7, the central issue is the development of point and interval estimates for the globally optimal solution using extreme-value distribution theory. Finally, in Section 8, some additional topics, which primarily concern non-Euclidean TSP's are discussed. These include how to obtain a measure of quality that is invariant to data changes and how to use multidimentional scaling to convert a non-Euclidean TSP to one with Euclidean distances.

In summary, this paper explores a wide variety of topics regarding the empirical analysis of TSP heuristics. The emphasis is on the development of formal (statistical) methods for comparing and analyzing TSP heuristics.

This paper can be obtained by writing to Bruce L. Golden at the University of Maryland. It will appear in The Traveling Salesman Problem, E. Lawler, J. K. Lenstra and A. Rinnooy Kan, Eds.

References

[Adrabinski & Syslo 1980]; "Computational Experiments with Some Heuristic Algorithms for the Traveling Salesman Problem," Technical Report #N-78, Institute of Computer Science, Wroclaw University, Wroclaw, Poland.

[Golden, Bodin, Doyle & Stewart 1980]; "Approximate Traveling Salesman Algorithms," Operations Research, 28, 694-711.

[Mosteller & Rourke 1973]; Sturdy Statistics, Addison-Wesley, Reading, Massachusetts.

[Stewart 1981]; "New Algorithms for Deterministic and Stochastic Vehicle Routing Problems," Doctoral Dissertation, University of Maryland.

1. The particular computer system used for the study.

2. The particular computer program used to solve the problem. In our case, the computer code BBNLMIP involves the following:

   a. The branch and bound computer code where all the locigal operations of the branch and bound are performed except for solving the nonlinear continuous problems, and

   b. The nonlinear code OPT [Gabriele and Ragsdell] which solves all the intermediate continuous problems.

Since the computational time depends on the computer system, the branch and bound code, and the nonlinear code (OPT, in our case), the ranking of the strategies might as well depend on these factors. Hence, another ranking criterion is also used to make comparisons where these factors would not have any effect.

This criterion compares the number of nonlinear continuous problems solved instead of the solution times. The continuous problems which are solved under a particular branch and bound strategy in fact define the corresponding branch and bound tree and therefore these problems would remain the same irrespective of the computer system, the branch and bound code, and the nonlinear algorithm used to solve the problem. The total number of nonlinear continuous problems solved by each strategy were tabulated and a procedure similar to one used for normalizing the solution times was used to normalized number of nonlinear continuous problems solved for each problem. The strategies were ranked according to their average value of normalized number of nonlinear problems (NNLP).

Table 1

Analysis of Variance (Normalized Solution Time)

Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	Significance
Main Effects	12.000	6	2.000	30.621	.001
Variable Selection	1.243	2	.621	9.513	.001
Node Selection	5.783	2	2.892	44.272	.001
Heuristic Selection	4.937	2	2.469	37.796	.001
2-Way Interaction	.229	12	.019	.293	.991
Variable - Node	.076	4	.019	.292	.883
Variable - Heuristic	.010	4	.002	.037	.997
Node - Heuristic	.142	4	.036	.545	.703

Source of Variation	Sum of Squares	D.F.	Mean Square	F-Ratio	Significance
3-Way Interactions	.251	8	.031	.480	.870
Variables-Node-Heuristic	.251	8	.031	.480	.870
Explained	12.480	26	.480	7.349	.001
Residual	36.773	563	.065		
Total	49.253	589	.084		

It should be noted that the top 12 strategies still remain the same. None of them make use of either branching from the newest node or implementing heuristic A. Strategies 16 and 10 are still the top two strategies. The rankings of the other strategies do not alter significantly. The overall means of the selection parameters were computed and are also given in this paper. The results are very much simliar to those of obtained previously. The findings obtained by previous ranking procedure are confirmed by the findings based on this criterion.

5. CONCLUSIONS

In this paper, we have investigated the feasibility of applying the branch and bound approach to nonlinear mixed integer programming problems. It has been shown that branch and bound methods can be implemented as useful computational tools in solving such problems. Among others, the concepts of pseudo-costs and estimations are invoked in selecting branching variables and branching nodes. The results show that among the three branching variable selection criteria that we have tested, the option of selecting the integer variable with the most fractional value is the best for the problem set tested. Among the node selection strategies, the criterion of selecting the newest node is the worst. It should be noted that considerable effort is made in computing pseudo-costs and estimations at the beginning of each run, and therefore, their full strength is probably not exploited in relatively less complex test problems.

Two heuristics have also been developed to provide fast upper bounds on the objective. However, the results do not seem to indicate that they have any significant computational advantage. It should be noted that the various strategies are compared with respect to how soon the optimal solutions are reached and not how soon the upper bounds are formed. The results were drastically different when the strategies were compared only for finding the upper bounds. It was found that heuristic B was able to find bounds on the average within 60% of the time when compared with the option of invoking no heuristics. Heuristic A was not as good, and it took almost 90% of the average time of strategies without heuristics. This indicates that these heuristics, particularly heuristic B, have a computational advantage in finding upper bounds. For details on this analysis of heuristics, readers are referred to Gupta and Omprakash.

REFERENCES

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

Eason, E. D., and Fenton, R. G., "A Comparison of Numerical Optimization Methods for Engineering Design," ASME J. of Engineering for Industry, Vol. 96, No. 1, 1974, pp. 196-200.

Gabriele, G. A., and Ragsdell, K. M., "OPT, A Nonlinear Programming Code in Fortran IV," The Modern Design Series, Vol. 1, Purdue Research Foundation, 1976.

Gupta, Omprakash K., "Branch and Bound Experiments in Nonlinear Integer Programming," Ph.D. Thesis, School of Industrial Engineering, Purdue University, West Lafayette, Indiana, December 1980.

Himmelblau, D. M., Applied Nonlinear Programming, McGraw-Hill, New York.

Rosen, J. B., and Suzuki, S., "Construction of Nonlinear Programming Test Problems," Communications of the ACM, Vol. 8, No. 2, 1965, pp. 113.

Sandgren, E., and Ragsdell, K. M., "The Utility of Nonlinear Programming Algorithms: A Comparative Study - Parts I and II," ASME Journal of Mechanical Design, Vol. 102, No. 3, 1980, pp. 540-551.

Whereas, worst-case analysis and probabilistic analysis of heuristics are performed in a scientific manner, empirical analysis is often performed in a haphazard and subjective way. Our hope is that this paper will help correct this situation.

When we think about the empirical analysis of heuristics, one of the following two scenarios usually come to mind. Either we are interested in comparing several heuristics on a set of test problems to determine the best one, or we are interested in finding out how far from optimality heuristic solutions lie. In either case, statistical inference plays a major role.

While we devote most of this paper to a discussion of ways in which to obtain high-quality solutions, heuristics possess attributes in addition to quality of solution that we feel should be kept in mind. These attributes are well-recognized as valid criteria for the comparison of heuristic algorithms (see [Ball & Magazine 1981]) and include:

(a) Running Time;
(b) Ease of Implementation;
(c) Flexibility
(d) Simplicity.

The running time required by a heuristic to solve a given problem is often a crucial consideration in choosing between competing approaches. Ease of implementation is also an important consideration. Difficult-to-code algorithms that require substantial amounts of computer time may not be worth the effort if they only marginally outperform an easy-to-code algroithm that is extremely efficient. Flexibility refers to the ability of an algorithm to handle problem variations. A TSP heuristic that can solve only undirected TSP's is, clearly, not as flexible as one than can solve both directed and undirected TSP's. With regard to simplicity, simply-stated algorithms are more appealing to the user than cumbersome algorithms and they more readily lend themselves to various kinds of analysis.

Section 1 of this paper provides the necessary motivation for the paper. In Section 2, a statistical method for comparing two heuristics at a time (the Wilcoxon test) is introduced and illustrated with respect to two heuristics and four measures of the quality of the solution. The measures of quality are the number of times the heuristic is best or tied for best, the average percentage above the lower bound, the average rank among the heuristics, and the worst ratio of the solution to the lower bound.

Section 3 provides a synopsis of computational experience gathered in previous studies on TSP heuristics. In Section 4, a new heuristic for the Euclidean TSP is presented. Extensive computational results comparing this new algorithm to those previously presented are analyzed using a variety of statistical tests including the sign test of Mosteller and Rourke. Each of

THE EMPIRICAL ANALYSIS OF TSP HEURISTICS

by

B. L. Golden

University of Maryland at College Park
College of Business & Management
College Park, Maryland 20742

W. R. Stewart

College of William and Mary
School of Business
Williamsburg, Virginia 23185

We focus on the empirical analysis of TSP heuristics. We interpret empirical analysis to mean analysis originating in or based on computational experience. The number of articles that are devoted, at least partially, to this topic is enormous. Since the 1950's, hundreds of procedures have been suggested for solving the TSP; these procedures have, in general, been tested on a small set of benchmark problems. The bibliography in the Traveling Salesman Problem edited by E. Lawler, et al bears witness to this observation. With this fact in mind, we make no attempt at being encyclopedic. We do, however, seek to accomplish, in a coherent fashion, the following:

1. discuss some guidelines for the empirical analysis and comparison of TSP heuristics;

2. discuss ad hoc methods for comparing TSP heuristics;

3. discuss statistical methods for comparing TSP heuristics;

4. summarize and update recent computational studies ([Golden, Bodin, Doyle & Stewart 1980] and [Adrabinski & Syslo 1980]);

5. present the results of a new computational comparison of Euclidean - TSP heuristics ([Stewart 1981]);

6. describe algorithms specifically designed for the directed (i.e., asymmetric) TSP;

7. study statistical inference techniques for assessing deviations from optimality.

COMPUTER CODES FOR INTEGER PROGRAMMING IN THE 1980'S*

A. H. Land

London School of Economics
Houghton Street
London W.C.2, United Kingdom

and

S. Powell

University of Kent at Canterbury
Rutherford College
Kent CT 2 7NX, United Kingdom

The authors of this paper made a survey in 1977 of all the computer codes they could locate, which purported to solve linear programming (LP) problems with some or all variables constrained to take on integer values (ILP). This survey is published in [1] and included both commercial codes and academic codes. They found, however, that the huge variety of data specificaitons and methods for the latter group prevented them from doing more than simply listing the information supplied to them by the authors of each of the academic codes. In this paper they have chosen not to seek and list new academic codes but to concentrate their attention on the commercial codes.

It turned out that all of the commercial codes listed in the 1977 survey were basically LP codes with a branch and bound (BB) [2,3,4,5] facility 'added on'. The enormous amount of research effort which has gone into developing other methods for integer programming is barely represented in the commercial codes. This is not to say that there has been no research in the development of the commercial codes, but only that it has been within the broad strategy of BB with LP as the fathoming procedure. Presumably one reason for this bias is that the paying customer demand comes from users who feel the need to incorporate some discrete restrictions into their LP models: for mixed integer programming (MIP) rather than for pure integer problems (PIP). Another reason is that BB generally provdes a usable answer, even when it fails to reach and prove an optimal solution.

Whatever the reasons, the situation is fundamentally unchanged in 1981, and this survey is concerned principally with recording developments in the commercial MIP scene since 1977. For that reason the authors have omitted from this survey codes which are not currently under development, even

*The editor of this newsletter has taken the liberty of summarizing the above paper for the COAL Newsletter and takes full responsibility for any oversights in the presentation.

though they may still be available to users. The survey covers six maintained, commercially available MIP codes. Table 1 gives the name of the code, the name of the responsible organization, and the computers on which the code runs, and the data at which it was first introduced.

This survey reports on the various capabilities of each of these codes. It reports on the types of variables the codes can handle, (such as continuous, semi-continuous, binary and integer) whether it can handle special ordered sets of type one and type two, the types of nodes in the branching tree and the various branching strategies of the codes, the stopping rules for the branch and bound search, how the codes detect which subproblems are considered not worth solving, and the solution method used for the linear programming subproblems. It also describes the defaults built into each of these codes, the tolerance settings used by the codes, the methods available to the user for specifying parts of the tree, the input and interactive capabilities of the codes, and the ability of the user to access the component parts of these packages.

The survey describes in great detail the various approaches to branching that each of the codes uses including descriptions of the types of penalties, pseudo-costs, priorities, choice of node to develop, backtrack strategies and candidature rules. Besides providing the reader with the specifics about each of the codes, it presents the reader with an overview of the general structure of large-scale integer programming packages.

The authors conclude "Progress has been made more in changes of detail than by any dramatic leaps. The emphasis continues to be on MIP rather than PIP, and on obtaining 'good' solutions quickly rather than on proving optimality. It is interestng that LAMPS and Sciconic are becoming available on a range of mini-computers, as well as on the large main-frames which were required for any sophisticated system in 1977. Oddly enough, this may enable users to let the problem run on 'until Monday morning' in the hope of proving optimality or of finding a better solution, in a way that would not be economic on a large computer."

This paper can be obtained by writing to either of the authors.

References

[1] A. Land and S. Powell, Computer codes for problems of integer programming, Annals of Discrete Mathematics 5 (1979) 221-269.

[2] M. Benichou, J-M. Gauthier, P. Girodet, G. Hentges, G. Ribiere and O. Vincent, Experiments in mixed integer linear programming, Mathematical Programming 1 (1971) 76-94.

[3] J.J.H. Forrest, J.P.H. Hirst and J.A. Tomlin, Practical solutions of large mixed integer programming problems with UMPIRE, Management Science 20 (Jan. 1974) 736-773.

developers candidly admit that this was not a design goal. MINOS is not capable of handling integer variables, but again, was not intended to do so. MINOS appears to be a significant improvement over other contemporary codes, and a most useful tool for the modeller/analyst.

XS, is also exactly what it is claimed to be: an experimental testbed for state-of-the-art optimization research. It is very easy to use, but is not intended primarily for the casual, inexperienced user. Nor is XS specifically designed for nonlinear models. XS provides many flexible file editing, problem representation, solution, and report options, but is designed for efficient custom applications to particular classes of models. The default nonlinear feature of XS, operating with one of the problem representation "templates," provides a quick-turnaround modelling environment.

XS is designed to accommodate problems with no analytic gradients, poor data resolution, approximated functions, and all the attendant difficulties of real-life optimization at large-scale. Problems should be formulated for XS with realistic range intervals and meaningful penalties associated with contraints. Accordingly, XS can encounter difficulties with artificial test problems presented in strictest equality form with default (rather than modelled) penalties.

The large-scale nonlinear integer capability of XS, combined with basis factorization (e.g., GUB), decomposition facilities, etc., make it truly unique in the field of large-scale optimization. It is capable of solving problems that no other system in the world known to the author can attempt. By achieving this, XS has more than fulfilled the goals of its developers.

REFERENCES

1. **Pierskalla, W., and Ratliff, H. D., "Reporting Computational Experiences in Operations Research," Operations Research, v. 29., pp. xi-xiv, March-April 1981.**

2. **International Business Machines Report SC28-6852-2, IBM OS FORTRAN IV (H Extended) Compiler Programmer's Guide, November 1974.**

3. **International Business Machines Report SH20-0968-1, Mathematical Programming System-Extended (MPSX), and Generalized Upper Bounding, IBM Corporation, New York, 1974.**

4. **Systems Optimization Laboratory, Department of Operations Research, Stanford University Technical Report SOL 80-100, MINOS Distribution Documentation, by M. A. Saunders, September 1980.**

5. **Systems Optimization Laboratory, Department of Operations Research, Stanford University Technical Report SOL 77-9, MINOS User's Guide, by B. A. Murtagh and M. A. Saunders, February 1977.**

6. **Systems Optimization Laboratory, Department of Operations Research, Stanford University Technical Report SOL 80-14, MINOS/Augumented User's Manual, by B. A. Murtagh and M. A. Saunders, June 1980.**

Although relatively little was required of the user for initial setup on XS, becoming familiar with the techniques of using "pointer" variables required time; especially with the dearth of documentation.

Interestingly, as the user becomes more experienced with both systems, MINOS becomes easier to use with the exception of the gradient functions, while in some ways XS becomes more difficult as the profound tuning capability of the system becomes apparent and places more demands on the talents of the user/analyst (i.e., appropriate problem formulation).

2. Debug Output

Both systems are capable of producing voluminous debug listings to assist the user, and the only limitation is the knowledge and patience of the user in their interpretation.

3. Failure Mode

Both codes give comprehensive output to indicate why they fail. As long as the user is sophisticated enough in system use to understand the diagnostics, he can usually intuit the cause.

4. Robustness

Default values exist for all parameters in the SPECS file for MINOS and experience has shown these values to be very robust with little tuning required other than specifying the problem-specific size parameters. Although the explicit statement of the objective function gradient is recommended for MINOS [Ref 6: p. 18], experimentation with the objective function differencing option has revealed no significant change in solution values or CPU time between the explicit gradient and the numerical differencing representation.

Many of the XS tuning parameters are dependent on the scaling of the problem, and their robustness is in direct proportion with the user's ability to provide a well-scaled problem. This just requires reasonable care in the original problem formulation, but for complex test problems presented in completed form, reformulation and scaling can be vexing.

G. Summary

After 15 months of intensive use of both codes, it is apparent that both systems have achieved what their designers intended. MINOS is a well-documented, easy-to-use code that reliably achieves excellent results on the general (NLP) problem while demanding only moderate skill of the user. Its default parameters are robust and require minimal tuning to achieve satisfactory results. Although its input files can be somewhat cumbersome to manage, they are straightforward and unlikely to cause a confidence crisis for the inexperienced user. Constraint gradients, on the other hand, can be arduous to prepare and debug, even for simple test problems. MINOS does not have some of the more sophisticated file editing and solution options, but its

[4] A.M. Geoffrion and R.E. Marsten, Integer programming algorithms: a framework and state-of-the-art survey, Management Science 18 (1972) 465-491.

[5] G. Mitra, Investigation of some branch and bound strategies for the solution of mixed integer linear programs, Mathematical Programming 4 (1973) 155-170.

Table 1

Code	Organization	Computer	Date
Apex III	Control Data	Cyber 70 series, models 72, 73, 74 and 76; 6000 series, 7600 series	1975
Bloodhound	Ketron Inc.	IBM S/370, models 145 and above. IBM 303X, Amdahl 470vx	?
LAMPS	AMS Ltd.	DEC20 series, DEC/VAX 11/780, PRIME 550 upwards, Perkin-Elmer 3200 series, Harris	1980
MIP/370	IBM	370 series	1973
MPS	Honeywell	Series 60 (level 66)	1975
Sciconic	Scicon Ltd.	Univac 1100 series, DEC/VAX 11/780	1976

TOOLPACK
Architectural Design: The Users' Perspective

Leon J. Osterweil[1]
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

Stephen Hague
Numerical Algorithms Group, Ltd.
256 Banbury Road
Oxford OX2 7DE

Webb Miller[2]
Department of Computer Science
University of Arizona
Tucson, AZ 85721

April 15, 1982

[1]Supported by National Science Foundation grant number MCS-8000017, and Department of Energy grant number DE-AC02-80ER10718.

[2]Supported by National Science Foundation grant number MCS-7926441.

Pivots (minor iterations) for MINOS represent classical basis exchanges (a superbasic variable becomes basic, replacing a basic variable which becomes nonbasic). XS pivots are counted in several varieties, including logical cases in which no basis change or objective function improvement takes place but the logical composition of the solution changes. XS usually requires more of its pivots than does MINOS, but apparently executes each of them faster.

Iteration counts do not appear to be adequate general measures of efficiency for dissimilar optimization algorithms, or even alternate implementations of the same algorithm.

E. Number of Function Evaluations

For the objective function precision specified in these tests, MINOS has generally required less function (/gradient) calls than XS. This difference can be crucial for problems in which function generation requires the vast majority of computing effort. However, a paradox is apparent in that such problems often have no closed-form derivatives: desirable for (quasi-Newton) second-order algorithms.[1] In most cases, XS provides 2-3 decimal place precision in the objective function value with less function evaluations than MINOS, but MINOS produces 3-5 place precision with less function calls than XS. MINOS usually requires more function evaluations than XS to yield a feasible solution.

For most classes of functions (polynominal, exponential, etc.), the gradient is of the same class as the function; the function calls (and therefore the required CPU time) to determine gradient information are approximately the same for both analytic and numerical methods.

F. User Friendliness

Because of its documentation and intended use, it was expected from the start of the study that MINOS would be the system most amenable to the user and the results bear this out.

1. Ease of Setup

This was one area in which MINOS was not necessarily better. The requirements for providing analytic gradients proved to be very time consuming, notwithstanding their aforementioned use as a trouble-shooting aid. In addition, the need to refer to a number of documents for constructing the SPECS and MPS files (until "templates" were constructed) was a handicap to the new user. Once the programming aids were in place, construction of a new problem became quite straightforward with a minimum of outside reference required.

---

[1]A conversation late in this research with Professor G. Vanderplatz, Dept. of Mechanical Engineering, Naval Postgraduage School, revealed that his widely used optimization codes for engineering design problems have been evaluated using the number of function calls as the <u>exclusive</u> gauge of efficiency, that closed-form gradients are rarely available for such problems, and that empirical engineering data resolution may often be as poor as ±20 percent.

II. CONCLUSIONS

A. Algorithm Capacilities

1. Type of Problems

MINOS is capable of reliably solving both the general (LP) and (NLP) problems with any combination of linear and nonlinear constraints, but cannot accommodate integer variables.

XS has the same capabilities as MINOS with the addition of the use of integer variables. XS can also employ decomposition and basis factorization.

2. Growth Possibilities

There is no inherent maximum problem size for either MINOS of XS. For sheer capacity, they are both limited by the computer memory requirements for their working arrays. However, as noted earlier for MINOS, in large problems (when the number of superbasic variables exceeds 100 or 200) the shift to the conjugate gradient algorithm, which consumes less memory, results in a significant decrease in the theoretical rate of convegence of the algorithm [Ref. 5: p. 10].

B. CPU Time

MINOS was a bit quicker in solving many of the problems of this study. While XS yields 2-3 decimal place precision in the objective function faster than MINOS, MINOS usually prevails in 3-5 place efficiency (even when using first-order conjugate gradient options).

C. Storage Requirements

For linear programs containing m general constraints, roughly 100(m)-bytes of memory are required for workspace by MINOS. If there are many nonlinear variables, additional memory may be required. This workspace size may be adjusted by changing the size of one array in the main program for MINOS or by a non-FORTRAN routine that allocates storage at run-time. The choice of method is machine-dependent and guidelines are provided to the user in the documentation [Ref.4].

XS, used strictly in-core, requires a region of approximately 56(MN) + 8(NR) + 200K-bytes where MN is the total rows + cols, and NR is the size of the distinct real value pool. Storage requirements for nonlinear problems known to this writer are not a significant consideration for XS, or for MINOS.

D. Number of Iterations

Each major iteration of MINOS creates a local linearization of the nonlinear program, and then solves it after addition of a quadratic (augmented Lagrangian) objective function. XS simply solves local linearizations (with augmentation of the linear penalty function and local trust region). MINOS usually requires less of its iterations than does XS, but evidently works harder on each.

1. Introduction

The purpose of this document is to set forth the architectural design for the user interface of an ensemble of software tools and a software system to meet the agreed-upon objectives of the Toolpack project. Accordingly the document reviews the project's goals, then presents the Toolpack architecture.

The ideas presented here are the results of considerable thought and of numerous discussions with people both within and outside of the Toolpack group. They represent a firm architectural blueprint with many elaborative details, some of which can still be viewed as tentative and changeable. Those ideas and details that are still changeable will be indicated as such.

2. Objectives for Toolpack

From the time of the project's inception it has been agreed that the purpose of Toolpack is to provide strong, comprehensive tool support to programmers who are producing, testing, transporting, or analyzing moderate size mathematical software written in Fortran. Careful consideration of the feasibility and desirability of a number of possible strategies for fulfilling that basic purpose has resulted in agreement on the following particular objectives.

1. The mathematical software whose production, testing, transportation and analysis will be supported by Toolpack is to be written in a dialect of Fortran 77. This dialect is to be carefully chosen to span the needs of as broad and numerous a user community as is practical.

2. Toolpack is to provide cost effective support for the production by up to 3 programmers of programs whose length is up to 5000 lines of source text. It may be less effective in supporting larger projects.

3. Toolpack is to provide cost effective support for the analysis and transporting of programs whose length is up to 10,000 lines of source text. It may be less effective in supporting larger projects.

4. Toolpack is to support users working in either batch or interactive mode, but may better support interactive use.

5. Toolpack is to be highly portable, making only weak assumptions about its operating environment. It will be designed, however, to make effective use of large amounts of primary and secondary memory, whenever these resources can be made available.

3. Overall Strategy of the Toolpack Project

The tool capabilities to be supported by Toolpack must be powerful, efficient and easy to use. Many tools have suffered misuse and rejection because they insufficiently met these criteria. Consequently the major goals of this project are to determine how best to satisfy all three criteria and to then build and distribute the appropriate software.

In order to make this determination two contrasting approaches will be pursued and evaluated. The first, and more conventional, approach will be to build and distribute individual tools designed and implemented to offer power, efficiency and ease-of-use. Many of these tools will be "second-generation" tools, whose utility and desirability have been established through experience with previously built prototypes.

The second, more innovative and speculative, approach will be to build a completely integrated collection of the individual tools — the *Integration System for Tools* (IST). IST will feature a modular internal design and a clean user interface. The internal design is to provide power through flexibility, and efficiency through intertool cooperation. The user interface is to greatly simplify the communication between the user and Toolpack. In order to address these objectives, the IST design incorporates some rather speculative innovations, thereby inviting risk. Thus careful evaluation of IST is planned, and will include comparisons with the stand-alone tools. Design iterations, involving feedback from a sequence of releases, are planned to follow a schedule outlined in Section 7.

4. The Toolpack Tool Ensemble

The Toolpack group is in agreement that the following tool capabilities constitute a sound basis for a programming environment that supports the production of high quality Fortran programs:

0. A compiling/loading system
1. A Fortran-intelligent editor
2. A formatter
3. A structurer
4. A dynamic testing and validation aid
5. A dynamic debugging aid
6. A static error detection and validation aid
7. A static portability checking aid
8. A documentation generation aid
9. A program transformer

A compiling/loading capability is generally available on host operating systems. Thus no tool development effort in this area is proposed. The extensive tool development described below is, however, planned for areas 1 to 9.

4.1. Fortran-Intelligent Editor

A powerful editor [Hagu 81] will be included in Toolpack to assist the programmer in producing Fortran source code. This editor will offer a range of general text manipulation facilities, including the usual capabilities for inserting, deleting, locating and transforming arbitrary strings of characters. In addition, the editor will provide the following facilities for constructing and modifying Fortran programs.

d. Robustness

As an aid to the inexperienced user, codes should be robust in their default parameters to minimize the amount of "tuning" that need be done on most problems. At the same time, the parameters must have sufficient scope and power to allow the experienced user to exploit the structure of difficult problems where interaction by the analyst is required in order to achieve a solution.

C. Test Problems

This study has been handicapped, as have other similar efforts, by the lack of suitable test problems to test the full capabilities of the codes. In this case, the need has been for large (several hundred variables or more) real-life problems. The author has been unable to secure such problems for which publication release is available and for which the data is in a form that is readily usable by both codes. This is a continuing and widely recognized problem which has prompted the development of a number of nonlinear artificial problem generators. However, although these generators can produce arbitrarily large problems, it has been the experience of the developers of the XS System that the randomly generated problems produced by the generators are not realistic tests for optimization codes because they possess none of the specialized structure that is routinely found in real-life problems and which, in fact, is the very thing capitalized upon by good codes to produce their excellent results on large problems.

Therefore, for the purposes of this test, the emphasis has been placed on real-life, or at least well-known problems, as opposed to generated problems, at the sacrifice of size. Thirteen problems were selected for this study, of which at least nine have been previously published. The variables in the problems range from two to 793 and the constraints from one to 401. The problems contain a mix of linear, nonlinear, equality, and inequality constraints. Also included are two problems with (12 and 100) integer variables which have never before been formally solved as nonlinear integer and nonlinear mixed integer programs.

D. Computer System

All computing has been completed in the W. R. Church Computer Center of the Naval Postgraduate School, Monterey, California on its installed IBM 3033 computers using new VM/SP timesharing system. The load modules for both optimization systems and for each respective problem generation subroutine were generated with the FORTRAN IV (H Extended) compiler using the OPTIMIZE (2) option [Ref. 2]. All problems have been solved interactively in real-time on the computer system using precompiled load modules of the respective optimization systems linked with code and parameter data for the individual problems. This research has promoted development of an extensive real-time library of service routines to aid in the preparation, execution, and interpretation of large optimization problems. Although neither optimization system has a truly interactive solution algorithm, each can be used with interactive parameter settings and with real-time monitoring of solution progress, providing a fertile research environment.

Because of the region requirements of the FORTRAN compiler used on the host computer (see Section I.D), one megabyte of default virtual memory was used for all problems in a single-step procedure. Comments concerning problem-dependent memory requirements of each system will be made in Chapter IV.

3. Number of Iterations

The number of major interations (linearizations) and pivots required to reach solution is given for each problem, with the caveat that the nature of an "iteration" varies considerably between the algorithms. The specific nature of these iterations is discussed in Sections II.A.1 and II.B.1 of the thesis.

4. Number of Function Evaluations

The number of function evaluations to reach solution is listed for each algorithm. However, since this number includes both objective function and constraint evaluations, as well as gradient calculations in the case of MINOS, different amounts of information may be obtained on each function call and this may, therefore, be a deceptive comparison.

5. User Friendliness

One of the primary goals of this study is to evaluate the ability of a user familiar with some optimization theory but with little experience with the individual codes to set up and successfully solve a problem. Because of the codes' different design motivations, it was expected from the beginning of the study that MINOS would be far superior in this regard.

a. Ease of Setup

One measure of the flexibility of a problem-solving system is the ease with which it can be adapted by the general user to the particular problem/data structure at hand.

b. Debug Output

During initial debugging of a problem, varying quantities and types of diagnostic information may be required to isolate a particular error. The ability of each code to provide a tailored output in concise, readable form for the user will be evaluated.

c. Failure Mode

Since the perfect optimization code has yet to be developed, one measure of a code's performance is its ability to fail "gracefully," leaving the user in a posture from which he can recover without all of his effort being wasted. The information given to the user when each code fails is examined to evaluate its usefulness in further problem exploration.

- The user will be able to abbreviate Fortran keywords; these abbreviations will be automatically expanded by the editor.
- The editor will assure that various fields of Fortran statements are placed in the proper columns.
- As with the Cornell Program Synthesizer [Teit 81] and the Mentor system [Donz 80], the Toolpack editor will prompt the user for anticipated constructs. Moreover, the subsequent incoming statement will be checked for syntactic correctness and certain kinds of semantic consistency, e.g., the usage of a variable against declarations of the variable.
- The user will be able to search for occurrences of specified variables or labels. The editor will be able to distinguish those occurrences from occurrences of the same string of characters in other contexts, e.g., comments.
- It will be possible to confine searching and replacement operations to fixed domains of a program, such as a particular DO loop or a particular subroutine. For example, it will be possible to change all occurrences of a given variable (say *X*) to another variable (say *Y*) within a specified subroutine.

4.2. Formatter

Toolpack will provide a tool to put Fortran programs into a canonical form. In particular, the formatting tool, called Polish-X [Fosd 81] will have the following capabilities.

- Variables and operators will be set off by exactly one space on either side, except in certain cases, e.g., subscripts.
- DO loop bodies and IF statement alternatives will be indented.
- Statement labels will optionally be put in regular increasing order.
- It will be possible to optionally align the lefthand and righthand margins of statements.
- It will be possible to insert ON and OFF markers to indicate that Polish-X is to leave certain sections of the program unaltered.

4.3. Structurer

The ability to infer and emphasize the underlying looping structure of a program is useful. The failure of Fortran 77 to supply suitable constructs for doing so has left a significant void in the language. Hence a tool is to be provided that will recast Fortran 77 program loops as, for example, DO WHILE loops, either simulated in Fortran 77 by canonical constructs or realized explicitly according to the rules of RATFOR [Kern 75], EFL [Feld 79a] or SFTRAN [JPL 81]. This tool will, moreover, be able to automatically upgrade many Fortran 66 GO TO's to Fortran 77 IF-THEN-ELSE constructs. Such structuring often improves readability and comprehensibility, and serves as valuable documentation. The structuring capability in Toolpack will be closely patterned after the UNIX **struct** command [Bake 77].

4.4. Dynamic Testing and Validation Aid

Toolpack will contain a facility for automatically inserting instrumentation probes into Fortran programs and for creating useful intermediate output from these probes at run time. This facility will enable the user to capture and view a variety of trace and summary information. It will be possible, for example, to capture a program's statement execution sequence or to generate a histogram of the relative frequencies of execution of the various statements. Similarly, it will be possible to capture and study subroutine execution sequences and histograms, or variable evolution histories.

It will be possible for the user to implant in the subject program monitors for certain kinds of errors. For example, the user will be able to specify that either all or certain specified arrays are to be monitored to be sure that the subscripts by which they are referenced stay within declared bounds.

This facility will also incorporate a capability for checking the outcome of an execution against specifications of intent fashioned by the user. The specifications of intent are to be embodied in assertions, stated in comments and expressed in a flexible assertion language. These assertions will be expanded into executable code by the Toolpack dynamic testing facility. Once an assertion violation is detected, relevant information about the program status at the time of the violation will be automatically saved.

A system, called Newton [Feib 81], is being developed to provide the functional capabilities just outlined.

The dynamic testing capability will make it all too easy to produce very large amounts of output. There is some sentiment among Toolpack group members that tool support should be provided to aid the process of inferring useful diagnostic and documentation information from this raw output. This support would treat the output from the dynamic execution as a data base of information, and would consist of data base management aids and report generators to organize and format the diagnostic output for ease of understanding. There are currently no firm plans to construct such a tool.

4.5. Dynamic Debugging Aid

Debugging is facilitated by the ability to scrutinize to arbitrary levels of detail the progress of the execution of a program that is behaving incorrectly. Thus the Newton system, classified above as a dynamic testing and validation aid, can be viewed as a debugging aid as well.

In addition, however, debugging is assisted by snapshot, breakpoint and single-step-execution capabilities. These are also to be provided by Newton. They will enable a user to suspend execution at designated sites or on designated conditions. While execution is suspended the user will be able to examine the current values of variables, the execution history to date and the source text.

4.6 Static Error Detection and Validation Aid

Toolpack will furnish flexible capabilities for statically detecting a wide range of errors and, where possible, for proving the absence of errors. These tools will offer the users the ability to easily select from a range of capabilities that includes those supplied by the DAVE data flow analysis system [Oste 76].

The general non-separable nonlinear (NLP) problem can be stated as:

minimize	$f(x)$	(objective function)
subject to	$\underline{r} \le g(x) \le \bar{r}$	(ranged constraints)
	$\underline{b} \le x \le \bar{b}$	(bounds on variables)

where

x = variables;

$f(x)$ = general non-separable, nonlinear function;

$g(x)$ = general non-separable, nonlinear constraint;

$\underline{r}$, $\bar{r}$ = upper and lower constraint ranges;

$\underline{b}$, $\bar{b}$ = upper and lower variable bounds.

B. Comparison Criteria

In any study of this nature, one of the primary concerns is the criteria with which the codes are to be objectively compared. In this case, the guidelines recently published in Operations Research [Ref. 1] will be used with some modification to prevent comparisons that are not valid because of the somewhat different nature of the two codes. These criteria, to be elaborated in Chapter IV, are listed below.

1. Algorithm Capabilities

This section contains a general overview of the types and classes of problems for which each code is designed and comments on the growth capabilities of each code.

2. CPU (Compute) Time

The CPU times listed are the virtual CPU times required for each problem running with precompiled load modules for each primary system code and do not include the linkage editor times. Since the problems have been run interactively on a virtual memory computer system, these times will vary somewhat from run to run depending upon computer loading. Extensive experience on the host computer with these problems indicates that the listed CPU times are valid within one per cent. Although the actual "clock," or "response," time (as opposed to CPU time) varies as a function of system loading; empirical evidence gathered while conducting this study suggests that a useful rule-of-thumb is that actual clock time is approximately four times as long as CPU time for the virtual memory time-sharing system used. This should provide a reasonable estimate of the response times to be expected for problems of this study.

This is the first independent comparison of either code and is intended to serve both as an evalutaion of each and as a guide to the potential user concerned with the applicability of each code to the individual problem with which he might be faced.

Two caveats should be kept in mind while reading this evaluation. First, the codes are quite different in intended use. MINOS is intended as an academic production code and is designed to be readily distributed and applied by a wide variety of users. Extensive documentation and reliable performance have been paramount concerns in the development of MINOS. On the other hand, XS is used as an advanced experimental testbed for optimization research. The fully instrumented version used in this comparison is a prototype designed to be used almost exclusively by its originators and their co-workers for a wide range of problems, such as large mixed integer and linear formulations and especially for decomposition problems. As such, it is in a continual state of flux and varies considerably in its content (hopefully in an improving direction) from month to month. All results from XS are from the most recent prototype system at the time of publication cutoff for this thesis with no specialization for nonlinear programming. Academic and industrial production versions of XS are typically customized to the application at hand and thoroughly documented for routine use.

Second, although both systems are "large-scale" nonlinear codes which have been successfully used on many large, real-life problems, because of their intended day-to-day application, their characteristics are not the same, nor are they intended to be. Therefore, any differences between them in speed or capability may be attributable to design intention rather than relative deficiencies in the algorithms, underlying data structures, or implementation.

A. General Problem Statement

The general linear programming (LP) problem can be stated as:

minimize	$c^T x$	(objective function)
subject to	$\underline{r} \le A x \le \bar{r}$	(ranged constraints)
	$\underline{b} \le x \le \bar{b}$	(bounds on variables)

where:

x = variables;

c^T = cost coefficients;

A = constraint matrix coefficients;

$\underline{r}$, $\bar{r}$ = upper and lower constraint ranges;

$\underline{b}$, $\bar{b}$ = upper and lower variable bounds.

The structures of modern modular compilers and of the DAVE II system suggest that the static analysis of a program can be organized into the following progression of analytic steps: lexical analysis, syntactic analysis, static semantic analysis and data flow analysis. Thus the Toolpack static analysis capability will be subdivided into individually selectable capabilities offering these levels of analytic power.

The lexical analysis step will accept as input the program source text, and convert it into the corresponding list of lexical tokens. Illegal tokens, such as unknown keywords or variables that are too long, will be detected in this process and reported.

The syntactic analysis step will require the list of lexical tokens as input. This process will construct a parse tree representation of the user's program and a symbol table. In the process of doing this, syntactic errors, such as illegal expressions or malformed statements, will be detected and reported.

The static semantic analysis step will build upon the output of the first two static analyzers and will produce a number of structures designed to represent and elucidate the functioning of the program. These structures will facilitate the checking and cross-checking that can detect such errors as mismatched argument and parameter lists, unreachable code segments, inconsistencies between variable declaration and usage, and improper DO loop nesting and specification.

The data flow analysis step will rest upon the semantic information and flowgraph structures built by the other three analyzers, and will produce reports about the references and definitions affected by each statement and subprogram of the user's program. These reports will then be the basis for analytic scans of all possible program execution sequences. These scans will produce reports about whether there is any possibility of referencing a program variable before it has been defined, or defining a program variable and then never referencing it.

There is some sentiment among Toolpack group members that a tool is needed for centralizing and coordinating error reporting from these four static analysis tools. Such a tool would be similar in purpose to the error reporting tool discussed in Section 4.4. Here too, there is currently no firm plan to build such a tool.

4.7. Static Portability Checking Aid

Toolpack will furnish a capability for statically determining whether or not a given Fortran 77 program is written in such a dialect and style as to facilitate transporting the program. This capability will be modeled after the PFORT Verifier [Ryde 74], a very successful and useful program for checking the portability of Fortran 66 programs. Such portability obstacles as use of statement types not defined in the language standard (e.g., NAMELIST), assumptions about word lengths (e.g., packing of multiple characters in a word without use of the CHARACTER data type), and use of non-portable machine constants will be detected and reported.

Certain interprocedural checks not done by the PFORT Verifier, but supported by DAVE, will be incorporated into the Toolpack portability checker. For example, Fortran programs sometimes rely for correct execution upon assumptions about the parameter passing mechanism of the compiler on which the programs were developed. Data flow analysis determines the treatment of every parameter and COMMON variable by every subprogram with sufficient precision that nonportable parameter passing practices can be detected.

The functional capabilities needed to create this portability aid are quite similar to those needed by the static error detection and validation aid just described. A primary difference is that this tool, as opposed to most other Toolpack tools, will need to analyze and support a restricted Fortran 77 dialect, as opposed to a liberalized, extended dialect.

4.8. Documentation Generation Aid

The Toolpack group recognizes the importance of high quality program documentation and the desirability of tool support for the process of creating it. The group believes that the static and dynamic analysis capabilities already described create items of information that are useful program documentation. A documentation aid might well draw upon this information and facilitate its availability. In addition, the documentation aid might assist the preparation of user-generated documentation. No specification for this tool capability is currently available.

4.9. Program Transformer

There is general agreement among Toolpack group members that it is highly desirable to produce a program transformation tool as part of the Toolpack project. It is agreed that the tool should offer such capabilities as assistance in translating one dialect of Fortran to another, assistance in altering a program to perform its computations at a different level of precision, and facilities for creating special-purpose control or data structures.

There is currently little agreement, however, about the tradeoffs this tool should make between power, rigor, efficiency and usability. Three specific tools have been proposed — a template processor system, a macro processor system, and a correctness-preserving transformation system.

The template processor [Ward 81] is designed to enable the user to define Fortran language extensions by establishing data structure "templates". These templates can be named in the body of a Fortran program along with program data objects. The program data objects are then taken as arguments to be imbedded in the template description. The effect of this is that the user can employ and manipulate complex data structures in a source program without having to define those data structures within the program. This also leaves open the possibility that an expert could establish these complex data structure templates for users who lack the expertise to create the structures, but who, nevertheless, have a need for them. The template processor is designed to be extremely easy to use, but does little to guarantee that the Fortran statements it generates are correct or efficient.

The macro processor, called BIGMAC II [Myer 81], is similar in operation to the template processor. It enables users to define macros (code skeletons) which can then be expanded into actual bodies of code with the incorporation of parameters supplied to the macro at an invocation site in a user's program. Here, too, the macros can be written by an expert and made available to casual users, much like a library subprogram. BIGMAC II macros are more complicated and difficult to write than templates, but have the advantage of assisting the writer in preparing

COMPUTATIONAL ADVANCES IN LARGE SCALE NONLINEAR OPTIMIZATION

Summary of Thesis

by

Dennis Ross Dean*

Naval Postgraduate School
Monterey, California

Abstract

This is a comparison of two state-of-the-art large-scale nonlinear optimization systems exhibiting unprecedented problem solution capabilities both in size of problem handled and method of solution. These codes are MINOS, developed by B. A. Murtagh and M. A. Saunders, and XS, developed by G. G. Brown and G. W. Graves. The codes are evaluated with respect to their problem solving capabilities and potential for practical application by analysts. Computational results are presented for thirteen nonlinear and nonlinear mixed integer test problems with from two to 793 variables (12 to 100 integer variables) and one to 401 constraints.

I. INTRODUCTION

This study is a comparison of two state-of-the-art nonlinear programming codes that are designed to accommodate problems of thousands of variables and constraints. While there are many codes designed to handle the general linear programming (LP) problem and its specializations, there are significantly fewer systems designed to reliably solve the much more difficult nonlinear programming (NLP) problem. Of these, very few are capable of solving "large-scale" problems: larger than, say, a thousand constraints or a thousand variables or more. Most of these large-scale codes are internal, proprietary systems developed by companies for the solution of their specific industrially related problems; the codes used by petroleum refiners for chemical process control provide singular examples of such contributions.

One of the codes evaluated in this study is MINOS (Modular In-core Nonlinear Optimization System) developed by B. A. Murtagh, the University of New South Wales, and M. A. Saunders, Stanford University. The other is XS (S System) developed by G. G. Brown, Naval Postgraduate School, and G. W. Graves, University of California at Los Angeles.

*The editor of this Newsletter has abstracted portions of this thesis. A complete copy of this thesis can be obtained from the Naval Postgraduate School. Dennis Dean's Thesis advisor was Gerald G. Brown.

32. K. L. Weldon, One-to-one graphical strategies in multivariate data analysis, manuscript, 1981.

33. P. E. Gill and W. Murray, Performance evaluation for optimization software, in [7].

34. P. Nelson, Workshop on benchmark problems and code availability -- a workshop summary, in [4].

35. J. R. Rice, Machine and compiler effects on the performance of elliptic PDE software, Computer Science Report CSD-TR 359, Purdue Univ., 1981.

36. J.C.P. Bus, A methodological approach to testing of NLP - software, presented at Conference on Testing and Validating Algorithms and Software, Boulder, Colorado, Jan. 1981

efficient and correct Fortan programs. These differences spring, essentially, from the ability of BIGMAC II macros to acquire information about their invoking environments, to communicate with each other, and to produce output Fortran code that can be implanted in a few different strategic places in the source code of the invoking program.

The correctness-preserving transformation system, called TAMPR [Boyl 74], is the most powerful and sophisticated of the three proposed transformation systems. TAMPR constructs a parse-tree representation of the subject Fortran program, enables the user to analyze and transform the tree, and finally translates the transformed tree back to equivalent Fortran source code. TAMPR scrutinizes the transformation rules to be sure that the transformations that they specify do not alter the functionality of the subject program. This aspect of TAMPR makes it the safest of the three transformation systems. In addition, because the user is completely free to analyze and transform as much of the tree as desired, TAMPR has virtually limitless transformation power. The main drawbacks to this system are that, at least in prototype form, it appears to be very expensive to use and requires that the user be highly skilled and conversant with mathematical formalism.

In order to assist the Toolpack group in evaluating these three alternatives, it is likely that all will be made available as part of Toolpack so that a large and diverse user community can compare and evaluate them.

4.10. Additional Capabilities

Support from individual Toolpack group members has been expressed for eventual inclusion of a preprocessor for RATFOR [Kern 75], EFL [Feld 79a] or SFTRAN [JPL 81], for a document preparation aid like ROFF, for a source text version control facility, for a tape archiving program and for a general-purpose macro processor as advocated in [Mill 82]. Decisions about inclusion of such capabilities in Toolpack will hinge upon perceived user demand.

5. The Toolpack Integation System for Tools

5.1 Overview

As noted earlier, there is considerable support among the Toolpack group for the construction of a software system that effectively integrates the capabilities enumerated in Section 4. Accordingly the group is implementing such a system, called the *Integration System for Tools* (IST), and will evaluate and upgrade the system through a series of planned releases.

The primary motivating goal of the IST architectural design is to create a programming environment where usage is as straightforward and natural as possible. In particular, IST attempts to relieve the user of having to understand the intricasies and idiosyncrasies of individual Toolpack tools. It also lightens the burden of combining and coordinating these tools. The user is encouraged to express needs in terms of the requirements of the actual software job. IST is designed to then ascertain which tools are necessary, properly configure those tools, and present the results of using the tools to the user in a convenient form.

IST is a vehicle for establishing and maintaining a file system containing the information important to the user, and for using that file system to both furnish input to needed tools and capture the output from those tools. The file system will be managed primarily by means of a tree-structured directory system and a modular set of file accessing and updating primitives. It appears likely that existing portable Fortran packages supporting a tree-structured directory system [Hans 80a] and a set of I/O primitives [Hans 80b] will be used to implement much of the IST file system. However, competitive capabilities are under development within the Toolpack group and will be evaluated as they are completed.

This file system is to be initialized at the start of a programming project and remain and grow throughout the lifetime of the project. There is no reason why several users may not each access this file system, though we will make the implicit assumption that it is accessed by one user at a time, or by more than one user in non-interfering ways.

The IST file system is potentially quite large and may contain a diversity of stored entities. Source code modules would certainly reside in the file system, but so would more arcane entities such as token lists and flowgraph annotations. In order to keep IST's user image as straightforward as possible, this design proposes that most file system management be done automatically and internally to the IST, out of the sight and sphere of responsibility of the user. The Toolpack mechanism to automatically manage the IST file system will draw heavily on experience with the UNIX **make** facility [Feld 79b]. This approach will allow extensive and economical re-use of intermediate files.

The user will be encouraged to access only a relatively small number of files — those such as source code modules and test data sets that are of direct concern. The user may create, delete, alter and rename these files, as well as manipulate them with a set of commands that selectively and automatically configure and actuate the components of the Toolpack tool ensemble. The commands are designed to be easy to understand and use. They conceal the sometimes considerable tool mechanisms needed to effect the user's requests.

5.2. Basic IST File System Entities

In order to encourage and facilitate the preceding view, IST will support the naming, storage, retrieval, editing and manipulation of the following categories of entities, which should be considered to be the basic objects of IST. The five categories that are identified make it straightforward for the user to designate a body of code as the object of analysis and testing, to select various degrees of rigor and thoroughness in analyzing and testing that code, or to encapsulate commonly used sequences of operations.

5.2.1. Program Units:

An IST *program unit* (PU) is a Fortran main program, subroutine or function.

16. D. Kahaner, Comparison of numerical quadrature formulas, in Mathematical Software, ed. by J. Rice, Academic Press, London and New York (1971), pp. 229-259.

17. W. J. Kennedy and J. E. Gentle, Statistical Computing, Marcel Dekker, Inc., New York, 1980.

18. M. Lentini and V. Pereyra, An adaptive finite difference solver for nonlinear two point boundary value problems with mild boundary layers, SIAM J. Numer. Anal. 14 (1977), pp. 91-111.

19. F. A. Lootsma, Performance evaluation of non-linear programming codes from the viewpoint of the decision maker, in [7], pp. 285-297.

20. J. N. Lyness, Performance profiles and software evaluation, in [7], pp. 51-58.

21. J. A. Nelder, Experimental design and statistical evaluation, in [7], pp. 309-316.

22. V. Pereyra, Solucion numerica de ecuaciones diferenciales con valores de frontera, Acta Cientifica Venezolana 30 (1979), pp. 7-22.

23. P. M. Prenter and R. D. Russell, Orthogonal collocation for elliptic partial differential equations, SIAM J. Numer. Anal. 13 (1976), pp. 923-939.

24. S. Pruess, private communication, 1980.

25. J. R. Rice, Methodology for the algorithm selection problem, in [7], pp. 301-308.

26. R. D. Russell, Efficiencies of B-spline methods for solving differential equations, Proc. 5th Conference on Numerical Mathematics, Manitoba (1975), pp. 599-617.

27. R. D. Russell, Global codes for BVODES and their comparison, to appear in Proc. of Workshop on Numerical Integration of Differential Equations, Springer Lecture Notes.

28. R. D. Russell and J. M. Vaah, A comprison of global methods for linear two-point boundary value problems, Math of Comp. 29 (1975), pp. 1-13.

29. L. F. Shampine, Discussion on performance evaluation in ordinary differential equations, in [7], pp. 215-217.

30. L. F. Shampine, H. A. Watts and S. M. Davenport, Solving nonstiff ordinary differential equations -- the state of the art, SIAM Review 18 (1976), pp. 376-411.

31. A. Weiser, S. L. Eisenstat and M. H. Schultz, On solving elliptic equations to moderate accuracy, SIAM J. Numer. Anal. 17 (1980), pp. 908-929.

VI. REFERENCES

1. U. Ascher and R. D. Russell, Reformulation of boundary value problems into "standard" form to appear in SIAM Review.

2. U. Ascher, J. Christiansen and R. D. Russell, Collocation software for boundary value ODE's, 1979, to appear in Trans. on Math. Software.

3. P. Bjorstadt, A biharmonic equation solver, Proc. of Conference on Elliptic Problem Solvers, B. L. Buzbee, ed., 1980.

4. B. Childs, M. Scott, J. W. Daniel, E. Denman and P. Nelson, Codes for Boundary Value Problems in Ordinary Differential Equations, Springer Lecture Notes in Computer Science, New York, 1979.

5. H. Crowder, R. S. Dembo, and J. M. Mulvey, On reporting computational experiments with mathematical software, ACM Trans. on Math. Software, 5 (1979), 193-203.

6. B. Ford, G. S. Hodgson, and D. K. Sayers, Evaluation of numerical software intended for many machines -- is it possible?, in [7], pp. 317-330.

7. L. D. Fosdick, ed., Performance Evaluation of Numerical Software, Proc. of IFIP TC 2.5, North Holland Pub. Co., Amsterdam, 1979.

8. L. D. Fosdick, Detecting errors in programs, in [7], pp. 77-88.

9. E. O. Frind and G. F. Pinder, A collocation finite element method for potential problems in irregular domains, Int. J. Num. Math. Eng., 14 (1979), 681-701.

10. P. W. Gaffney, Using packaged software for solving two differntial equation problems that arise in plasma physics, Proc. of Conference on Elliptic Problem Solvers, B. L. Buzbee, ed., 1980.

11. W. M. Gentleman, Discussion of general aspects of performance evaluation, in [7], pp. 89-92.

12. K. L. Hiebert, A comparison of software which solves systems of nonlinear equations, Sandia Laboratories Report SAND 80-0181, Albuquerque, N.M., 1980.

13. K. L. Hiebert, A comparison of nonlinear least squares software, SANDIA Laboratories Report SAND 79-0483, Albuquerque, N.M., 1979.

14. E. N. Houstis, R. E. Lynch, T. S. Papatheodoru and J. R. Rice, Evaluation of numerical methods for elliptic partial differential equations, J. Comp. Phys. 27 (1978), pp. 323-350.

15. W. Kahan and J. Palmer, On a proposed floating-point standard, SIGNUM Newsletter special issue "The proposed IEEE Floating-Point Standard", October 1979, pp. 13-21.

5.2.2. Program Unit Groups:

An IST *program unit group* (PUG) is a set of IST program units that are to be analyzed or tested together. Other program unit groups may also be named as constituents of a program unit group, as long as no circularity is implied by such definitions, and a PU may belong to several PUG's. Optionally, a program unit group may contain the specification that certain transformations be automatically applied to the code. Among other things, this will facilitate programming in higher level languages such as RATFOR [Kern 75], EFL [Feld 79a] and SFTRAN [JPL 81].

5.2.3. Test Data Collections:

An IST *test data collection* (TDC) is a collection of one or more complete sets of test data for exercising one or more IST program unit groups. Each test data set may have associated with it a specification of the expected output.

5.2.4. Options Files:

An IST tool *options file* (OF) specifies which of the many available options are to be in force for a particular invocation of a Toolpack/IST tool. We see, for example, the need for *test options files* (TOF's) to specify dynamic testing options and *formatter options files* (FOF's) to specify source text formatting options, among others. It is expected that some standard options files will be created initially by the individual toolmakers and automatically incorporated as part of a newly installed IST. These standard options files could be altered to meet the needs of individual users and installations. In addition, entirely new options files can be built. It should be stressed, however, that tool options can be specified directly as part of a tool invocation command. Options so specified may either replace or supplement an options file specification.

5.2.5. Command Files:

An IST *command file* contains a sequence of IST commands that can be directed to the command interpreter simply by specifying the file name. Command files make it unnecessary to repeat a sequence of commands every time the same job must be done. This capability is supplied as a convenience and is intended to supplement, not replace, one-at-a-time command invocation. When an IST command file is invoked, it will be possible to specify certain items, like file names, that are left unspecified in the command file.

5.3 The IST Command Language

The Toolpack group is sensitive to the fact that user reaction to IST is likely to be strongly influenced by the character of the IST command language. The group recognizes that this language should be easy to use, forgiving of errors, flexible and, above all, that it conceal as completely as possible the elaborate implementation mechanism that underpins IST. The purpose of this section is to give a very brief and high-level view of the command language that is currently being implemented.

It is important for the reader to understand that the exact syntax of the command language has not yet been specified. Furthermore, it is unlikely that the final syntax will be agreed upon before there has been significant "hands-on" experience with some alternatives. Thus this document will present only an abstract view of the command language. This view is intended to convey the general tone, not the specifics, of the final IST command language.

The user of Toolpack/IST will employ the command language to specify the tools to be invoked, the source files to be processed and the options to be in force. The form of a command is

command_name source_file_name options_specification

where the three fields are delineated for illustrative purposes here by spaces. The command is processed by the Toolpack command interpreter, whose job it is to assemble all needed input files by whatever means are necessary (including automatically invoking other tools), to invoke the specified tool, to create and store the output files generated by the tool, and to present the user with a brief report on the outcome of this process.

The *command_name* field contains the name of the command that the user wishes to invoke. Some of the commands will correspond directly to the tool capabilities outlined in Section 4. Thus there will be commands like EDIT (Intelligent editor), FORMAT, INSTRUMENT, ANALYZE, and STRUCTURE. (The exact names will probably differ from this list, and abbreviations will be allowed.) Toolpack/IST will also support a number of utility commands. For example, the user will be able to inspect the contents of the file system and to create, delete, display, compare and modify specific files. Default naming conventions and summary reports from the main Fortran-processing tools will lead the user directly to files containing the tool's output; the utility tools will facilitate the perusing and distilling of those output files.

The *source_file_name* field of a command designates a list of one or more PU's or PUG's to which the command is to apply. Some tools, such as the parser and formatter are designed to operate on a single PU at a time. In such cases it is the job of the command interpreter to convert PUG names into lists of PU names. Other tools, such as the dynamic analysis tool and the data flow analysis component of the static analyzer, are designed to operate on entire PUG's. In those cases the command interpreter will see to it that the relevant information associated with the PU's comprising the PUG is integrated and assembled into files associated with the PUG.

The *options_specification* field designates the options that are to be in force during the command's execution. (If the field is empty, the command interpreter will supply a default options file.) The field can contain the name of an options file, an explicit options list, or both. An options file can be either system-defined or created by the user. An explicit options list is a list of pairs — the name of an option and a value for that option. In cases where both an options file and an explicit options list are specified, the explicit options will supplement the options in the file, if there is no conflict, or override the file-specified options when there is a conflict.

5.4. Advantages of the IST Approach

The following advantages, for the user, of a completely integrated tool system appear to the Toolpack group to outweigh the effort and risk, for the group, attending the production of such a system.

Incidentally, lack of precise design criteria does little to defer the typical user from forming a subjective preference for one code over another. In our experience, such preferences are readily made, often because one code is misused, and continued use of the codes can cause preference changes equally readily.

After evaluation criteria have somehow been chosen, a number of difficulties remain. One of these is insuring that objective results are presented, e.g. that one does not intensively test a code and present only its most favourable runs. On the other hand, fine tuning a code itself to perform well for the test problems may be desirable if their characteristics are representative of those for problems which one wishes to solve in practice.

Given evaluation criteria and numerical results, the problem of adequately displaying those results then arises. In [12], [13], Chernoff faces are used to display the results visually. Unfortunately, it is not obvious what criteria are reasonable for selecting the various facial features and the measurement scales for their display. It could be useful to consider alternative methods for displaying the data, such as star plots [32].

In any evaluation of mathematical software, it is important to set out goals which can be realistically achieved. Given the considerable difficulties discussed in the previous section, we feel that it is normally inappropriate to perform comparisons of general codes like BVODE codes aimed at finding the "best" one. The context is in many ways similar to that in [12], [13], namely (1) to draw conclusions about the general state of current codes, including where they are basically adequate and lacking, and (2) to test special features to indicate to users which code may be most appropriate for their special needs. While these purposes could probably be achieved for BVODE codes, the wide diversity of codes and problems would necessitate exhaustive experimentation for an extensive set of test problems, which to our knowledge has yet to be complied.

Comparing COLSYS and PASVA3 on a relatively small set of 20 BVODES, it was difficult to come to a consensus on when one outperforms the other. For us, this is partly because of our natural inclinations to emphasize evaluation criteria corresponding to the design criteria for our own codes, but it is also because both codes typically performed very well on the problems. As a result, our view is that efficient codes should generally be considered as complementary rather than competitive (see also [33] in another context). In addition to the extra insights that come from familiarity with two codes, higher degree of confidence in one's results when they are produced by different codes is perhaps the biggest advantage (see [34]). The use of reproduceability of numerical results by different methods has traditionally been a way of verifying their trustworthiness, and we recommend its use when feasible and economical.

3g. Form of Driver - Given the complexity of BVODES and the variety of possible inputs, the form of a driver is not necessarily straightforward. Often, unsuccessful runs or poor performance are the result of subtle errors. The effect of possible simplifications, such as finite difference approximations to analytic Jacobians, is another factor whose possible importance should be considered.

V. CONCLUSIONS

In the evaluation of mathematical software, it is important to realize that the underlying methods or even algorithms are not being compared, but only specific implementions. Indeed, order of magnitude improvements are sometimes made in BVODE codes by changing one part. (See [36] for the nonlinear programming case.) Care must be taken not to mistake limitations from a code's design for limitations of the method. For example, global methods have generally been considered impractical for solving Schrodinger equations which are BVODES involving a large number of differential equations because standard implementations (like COLSYS and PASVA3) would have prohibitive storage requirements. However, in the typical situation where only the missing conditions at ∞ are desired, by judiciously saving of only part of the information at each step, little more storage than for the successful initial value approaches is required [27].

Comparing codes is extremely difficult in an area of high complexity, such as BVODES, since these codes have options and purposes which are often different and incompatible. The difficulties seem to us to be in many senses insurmountable when comparing codes like COLSYS and PASVA3. Even though they have somewhat the same philosophies, selection of elementary evaluation criteria often necessitates going against the criteria of the designers of one code. Matters would be much worse if initial value type codes were considered too. For example, with most shooting codes one would have to make a subjective evaluation of the lack of opportunity to provide a good initial solution and mesh when available, lack of a global error estimate, and lack of automatic output point selection for the solution. While such features make the codes more complicated and expensive, one is often willing to pay this price for added robustness and flexibility, so evaluation of such features is at best intricate.

The importance of applying statistical techniques in evaluation of mathematical software has naturally been emphasized on several occasions, c.f.[25]. Much more experience needs to be gained in the area of experimental design for software evaluation [21]. Recommendations to follow when reporting numerical experiments in [5] are most valuable and in many contexts fairly complete. However, for the BVODE case the difficulties are to a large extent yet to be dealt with, and the previous section gives many specific examples of this. Many of these difficulties should arise in the evaluation of other types of differential equations, including IVODES, as the software capabilities are expanded. Since the experimental design for a specific software evaluation is not a priori well-defined and will undergo change in the course of the effort, we recommend saving information concerning all test runs. In this way information of unforeseen importance may be available, and in any case some indication of the codes ease of use is available.

5.4.1. Portability of Documentation and Experience

The Toolpack group is committed to producing stand-alone tools that realize the capabilities outlined in Section 4. Unfortunately, each particular computing environment will require the user to perform certain local incantations to invoke a tool. This implies a reliance upon local documentation, which may not be readily available, and poses a nuisance for the Toolpack user who must move to a different machine.

IST will provide a uniform programming environment for the use of Toolpack capabilities. This assures that essentially complete documentation, prepared as part of the Toolpack project, will be available to the user, and enhances the portability of programmers and programming experience.

5.4.2. Flexibility

Each component of the Toolpack tool ensemble will provide a considerable range of user options. For instance, the formatter will accept specifications for the depth of loop indenting or the increment between consecutive statement labels, directives to move FORMAT statements to the end of a program unit, and so on. IST command arguments, options files and command files provide mechanisms for accessing this flexibility, while avoiding both the inefficiency of recompiling a tool to change an option and the clumsiness of inserting directives like

```
C$      INDENT=3,INCREMENT=10,MOVE=NO
```

into every subject program.

5.4.3. Efficiency

The IST architecture allows implementations of Toolpack/IST to realize considerable efficiency through re-use of intermediate files. An example will serve to illustrate the basic idea.

Suppose that *MYPROG* is a PUG consisting of a number of PU's. The command

```
ANALYZE  MYPROG  OPT=CALLGRAPH
```

might report that the variable X is of single precision in the statement *CALL SAM(X)* occurring at line 37 of program unit *GEORGE*, whereas the dummy argument of *SAM* is double precision. To produce this diagnostic message Toolpack first assembles certain information about each individual PU in *MYPROG*, then checks for inter-unit inconsistencies. Toolpack/IST retains this information about the PU's (assuming that sufficient computer memory is available), whereas the stand-alone static analysis tool discards it.

Suppose that *GEORGE* is now edited to remove the inconsistency, say by declaring X to be double precision, and then the command

```
ANALYZE  MYPROG  OPT=CALLGRAPH
```

is repeated. Only the source text of *GEORGE* is accessed by IST, then compared with the still-valid information about the other PU's in *MYPROG*.

As stated earlier, the Toolpack/IST mechanism for keeping track of the validity of intermediate files draws heavily on experience with the UNIX make facility [Feld 79b]. This mechanism will be completely invisible to the casual Toolpack/IST user;

the user need not even know that the intermediate files exist.

5.4.4. Ease of Use

Even a relatively casual user of Toolpack/IST will find it easy to do many useful things that are difficult, on many computer systems, for the user of stand-alone tools. IST command files, which allow the user to encapsulate common sequences of commands, are a major contributor to this advantage, as the following example illustrates.

Suppose that a user of Toolpack/IST discovers that his particular approach to program development frequently results in the following sequence of activities.

1. A PUG is analyzed for certain errors that are inexpensive to uncover.
2. If no errors are found at step 1, the PUG is subjected to a more thorough and costly static analysis. (Information gleaned by Toolpack/IST at step 1 will not be rederived, so this phased approach to static analysis is not uneconomical.)
3. If no errors are found at step 2, the PUG is tested on a standard battery of test problems (using the default TOF).
4. The results computed during step 3 are compared with the expected results. (Specifications of the expected output can be associated with an IST test data collection.)
5. If the answers are correct, the program is formatted according to the options in the file *MY.STYLE.FOF*.

The user can encapsulate this sequence of activities into an IST command file like the following, which leaves unspecified the names of the PUG and the TDC. (The particular command syntax is for illustrative purposes only, and in no way represents a concrete proposal.)

```
ANALYZE  arg1  OPT=PORTABILITY,CALLGRAPH
IF  ERROR=TRUE  STOP
ANALYZE  arg1  OPT=DAVE
IF  ERROR=TRUE  STOP
TEST  arg1  arg2
COMPARE  arg1/DEFAULT/arg2/OUTPUT  arg2/EXPECTED
IF  IDENTICAL=TRUE  FORMAT  arg1  MY.STYLE.FOF
```

If these commands are saved under the name *TEST.AND.FORMAT*, then the command

```
TEST.AND.FORMAT  MYPROG  STANDARD.TDC
```

will result in activities 1 to 5 being performed on the PUG *MYPROG* and the TDC *STANDARD.TDC*.

6. Examples of Toolpack's Use

6.1 Program Construction

Figure 1 provides a framework for illustrating the ability of the Toolpack/IST system, as presented here, to aid in the process of producing Fortran programs. The diagram depicts what is thought to be a reasonable procedure for code creation. It is

3e. Program Parameters. The codes have many input parameters which determine the amount of information to be utilized, and different uses for them can vitally affect conclusions. One such parameter is the user's requested tolerance. COLSYS performs best (in a relative sense) if nonzero tolerances TOL_j in (4.1) are restricted to low order derivatives. Also, the tolerances chosen directly affect the mesh selection strategy for COLSYS, sometimes giving unusual results.

For both codes, continuation (a solution for one value of a parameter in the BVODE is used to obtain the initial approximation for a solution of the BVODE with a new value of the parameter) is easy to perform externally. Several options are available with only one of the codes, e.g. for PASVA3 internal Euler-continuation or for COLSYS using fixed mesh points, changing the fixed order k, or using a "sensitive problem" switch. Ignoring such parameters can give very misleading impressions about the capabilities of codes.

When solving most nontrivial BVODES, the important input parameters include the initial solution profile. This involves defining an initial solution and mesh, and the importance of the latter is frequently overlooked. It is not difficult to provide striking examples. In one test COLSYS is faster with an initial equal-spaced mesh of 5 subintervals, but slower with 15. In another test, PASVA3 behaves much differently on the problem when a finer mesh is provided. Consequently, it may be extremely difficult to decide which runs, corresponding to different inputs, are fair to report. A code may perform exceptionally well (or bad) with a certain solution or mesh, but this may only be discovered after exhaustive experimentation.

3f. Test Problems - Considerable effort has been expended in collecting test problems in order to evaluate many types of mathematical software, but the task of collecting a reasonably extensive set for BVODES still remains. It is suggested in [20] that problems involving parameters which control their level of difficulty can be run for several values of the parameters and in this way determine performance profiles. Naturally, the increase in difficulty of a given problem affects each code differently.

Recall that one of the biggest problems in compiling a set of problems to test BVODE software is that many important types of problems are convertible, and in different ways, to forms acceptable by the two codes [22], [1]. These types include problems with integral constraints, eigenvalues, infinite intervals, interfaces, singularities, switching points, constant delays, non-separated BC, and multi-point BC. A code's performance on a problem can be critically affected by the form of the problem used, and to some extent the efficiency of a code depends upon its ability to treat a problem with minimal modification. For example, in one test COLSYS performs favorably on the high order equation while PASVA3 converts to a first order system, and a second test PASVA3's ability to treat a non-separated BC enables it to solve a delay equation with one transformation instead of the two required for COLSYS [1]. We should mention here that a new generation code, PASVA4, is available on a testing basis. PASVA4 will solve many of these problems without conversion, gaining in this fashion a considerable increase in efficiency over PASVA3.

3b. User Feedback - Many mathematical software libraries prohibit user feedback under normal situations. We feel that it is generally desirable to have user feedback when solving complex problems such as BVODES. The two codes share this philosophy and provide through options information about the current mesh (and solution for COLSYS) and an account of how the nonlinear interation is proceeding. If comparison with codes using initial value-type approaches were being done, these output capabilities could be very difficult to evaluate.

3c. Error Estimation - Unlike most current IVODE solvers, COLSYS and PASVA3 provide global error estimates, although their philosophies are somewhat different. If e is the actual (absolute) error and $\tilde{e}$ is the estimated error, then e is usually accurate to within 1-2 digits for PASVA3 while usually

$.1 \leq \left|\frac{e - \tilde{e}}{e}\right| \leq 10$ for COLSYS. An exception frequently occurs when the solution has only 1-2 digits of accuracy, and the asymptotic behaviour upon which the error estimate is based is not yet valid. In any event, the evaluation of results for a given problem can be significantly affected by the emphasis placed upon the accuracy of the global error estimate.

3d. Termination Criteria - The codes use different stopping criteria. If u is the exact solution to the BVODE, (3.1)-(3.2) $\tilde{u}$ is the approximate solution, and TOL_j is the requested user tolerance for the jth derivative ($1 \leq j \leq m-1$), then COLSYS attempts to satisfy

$$(4.1) \quad |u^{(j)} - \tilde{u}^{(j)}|_i \leq TOL_j(1+|u^{(j)}|_i), \quad (1 \leq i \leq N)$$

where $|u|_i$ is the maximum norm for u(x) over the ith subinterval. PASVA3 uses the absolute error tolerance TOL for all solution components (all solution derivatives for (3.1)-(3.2)), so it attempts to satisfy

$$(4.2) \quad |\underline{v}-\tilde{\underline{v}}| \leq TOL$$

at the mesh points, where $\underline{v} = (u,u',\ldots,u^{(m-1)})^T$ and $\tilde{\underline{v}} = (\tilde{u},\tilde{u}',\ldots,\tilde{u}^{(m-1)})^T$. (The termination strategies are natural extensions when the BVODE involves a system of equations). A consequence is that when running the same BVODE for the two codes with a given TOL, completely different results can be obtained. Comparison is also troublesome because PASVA3 is providing a discrete solution satisfying (4.2) at the mesh points, while COLSYS is providing a continuous solution satisfying (4.1) in the (function) maximum norm. COLSYS is usually providing significantly better accuracy at the mesh points than globally although no error estimate for this is provided by the code.

not claimed here that this activity diagram presents a uniquely plausible scenario for program development. Rather, we believe that the IST architecture is eminently capable of supporting a diversity of code creation procedures, of which Figure 1 depicts one possibility. Determining the architecture's effectiveness will require direct experience with its applications in a variety of modes of use.

Comments on the Major Activities of Figure 1.

[1] Create regimen of test cases and required results:

An editor is used to create these TDC's, which are stored in the file system under a name supplied by the user.

[2] Compose new source text:

A text editor is used to create source code, which is stored in the IST file system indexed by PU name and version identification. The user may also define PUG's as sets of PU versions and transformation specifications, thereby creating other file system entities.

[2A] Revise existing source text:

If the required modifications are either minor or have no fixed pattern, a general purpose text editor would be an appropriate tool. If systematic restructuring is needed, the Fortran-intelligent editor or one of the other transforming tools may be more suitable.

[3] Standardize text:

The user may at this point wish to standardize the source text. For most users, standardization will mean applying a text formatter (alias "polisher", alias "pretty printer"). It may also involve the use of a structurer to improve the program's control structure. In the case of a large subprogram library, this stage may be a more comprehensive imposition of various programming conventions, such as declaring all variables explicitly or revising the order of declarations. There is to be an automatic purge of unpolished versions of the PU from the file system, unless the user directs that the polished version be saved under a new version name.

[4] Perform static analysis:

The user requests ANALYZE and specifies a PUG and a level of thoroughness for analysis. A data base of analytic results is created for perusal with the IST browsing facilities.

[5] Set up test runs:

This is basically an editing activity. The user assembles TOF's, which specify types and thoroughness of dynamic monitoring. A special TOF editor might be used to build new TOF's or to modify old ones. The user may modify or create TDC's here as well, and a source text editor may be used to inject new assertions into the source text. We should leave open the opportunity for incorporating test data

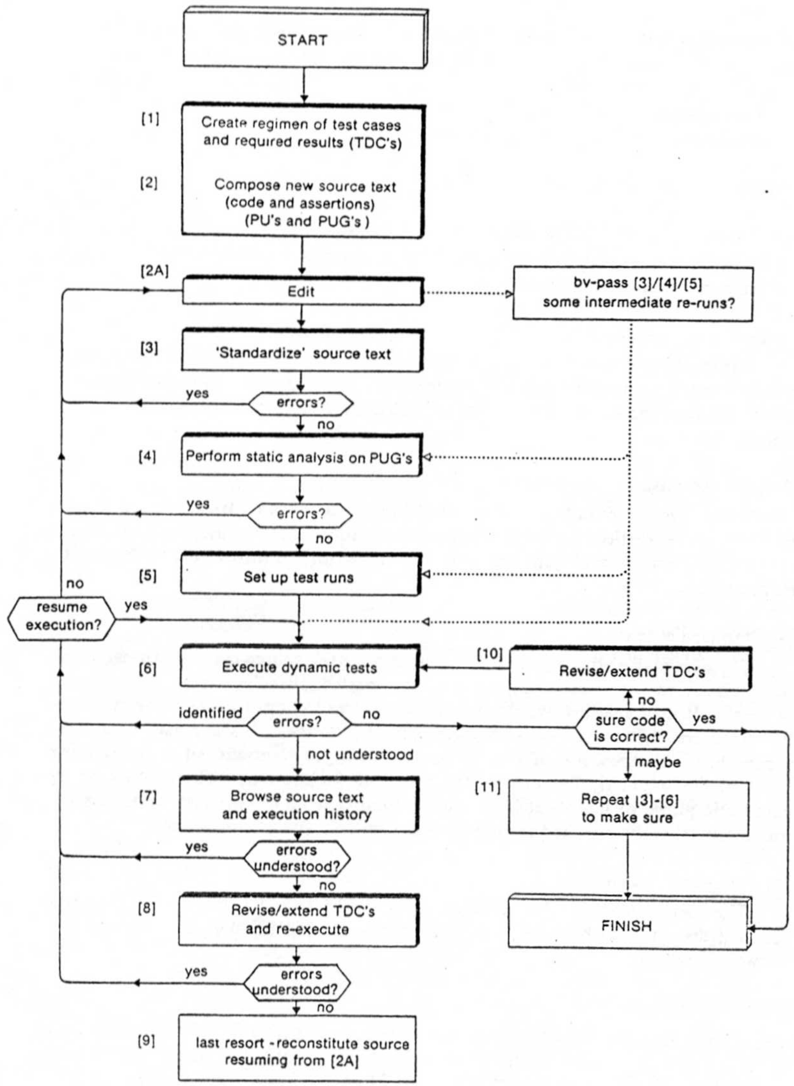

Figure 1.

1d. Program Correctness -Various approaches for debugging and for verifying program correctness have been proposed [8]. It is unrealistic though to expect that software evaluation will always discover the errors in large codes. A good code can be "error resistent" and perform well anyway [29], but a hope that unknown errors do not have the potential to cause grief in special situations is only that.

2a. Ease of Use - While one can subjectively compare code documentations, an obvious problem with evaluating relative ease of use is that of selecting a method of measurement. Both COLSYS and PASVA3 gave our students difficulties when learning the various capabilities, but ability to solve strightforward problems came quickly. Certain general statements can be made about where these difficulties arise, e.g. that errors in drivers are usually located in the Jacobians for nonlinear BVODES, but beyond this it is difficult to be specific about relative ease of use.

2b. Robustness - The robustness of codes, or degree to which they can solve a large class of problems and exit gracefully if not, is considered to be increasingly important in the trade-off with efficiency. For BVODES, this has motivated development of adaptive mesh selection and sophisticated nonlinear iteration strategies, frequently resulting in more flexible and easier-to-use codes. While certain indicators of the robustness of codes, such as frequency of occurrence of destructive overflows/underflows are easy to measure for a given set of runs, it is not straightforward to compare codes for difficult problems. One important reason is that it is a nontrivial task to insure that a comparable amount of initial information is given to the different codes as we shall see later for specific examples. Another reason is that ease of use decreases as generality of a code and/or the number of options increase. While one user may emphasize general ease of use another may be willing to solve their particular problem at virtually any cost or inconvenience.

3a. For of the Approximate Solution - Despite the fact that COLSYS and PASVA3 are more like each other than they are like the other BVODE codes, their design criteria (and hence their purposes) are in many ways quite different. A good example is the form used for the approximate solution, where PASVA3 produces a discrete solution defined at the mesh points and COLSYS provides a spline solution. If a discrete solution was desired by a user, the superconvergent mesh values could be used for COLSYS, and in fact extrapolating with these values is valid [24] even though not explicitly done by the code. Similarly, if a continuous solution was desired, the discrete solution provided by PASVA3 could be interpolated. The relative cost to preserve high accuracy in the latter case would probably be largest for smooth problems and coarse meshes. For run #2, e.g., PASVA3 appears preferable if an approximate solution at 20 points with accuracy 10^{-4} is sufficient, but more investigation would be necessary if a continuous solution is needed. General conclusions about the codes' relative performances would require dealing with this difference in a fairly subjective manner.

<u>1b. Storage</u> - Relative storage requirements are slightly affected by the machine precision used, but the measurement problem itself for a given example is nontrivial. COLSYS, for example, will use different mesh selection strategies, depending upon whether or not the alloted storage is a limiting factor. If it is, the code may solve a problem using less storage than if unlimited storage is provided. If storage is limited, accuracy may of course still be acceptable in a given situation even though the requested tolerance TOL is not attained.

<u>1c. Protability</u> - A large code's portability is almost impossible to ensure without actually testing it in each environment. Although COLSYS is written in ANSI FORTRAN [2], minor modifications were necessary before it ran successfully on the Burroughs 6700. The effects of using different compilers is of course well known; recall the problem of determining a machine's unit roundoff, where asking "If (1.+U .EQ. 1.)" is done differently for the IBM FORTRAN G and FORTRAN H compilers (the latter cleverly "optimizes" by comparing U to 0 directly).

Gaffney [10], considering stiff IVODE solvers, has the following striking examples of inconsistencies in a code's performance: (i) results in double precision are worse than those in single precision on one machine; (ii) qualitatively different integration results are obtained on different machines; (iii) as a parameter $\varepsilon \to 0$ in an IVP, making it more difficult, the solution is sometimes computed efficiently and sometimes not, but the transition is erratic as $\varepsilon \to 0$; (iv) results on the GRAY become 20% worse after it is updated to require commutative multiplication.

Various FORTRAN constructions cause difficulties for certain machines, e.g. block data subprograms for CDC [12]. By merely arranging DO loops (not in the most "natural" order) to allow for vector processing, the time for linear system solution in a biharmonic equation solver [3] is reduced to less than 1/7 of the original time when using a GRAY with this option. Investigating the effects of machines and compilers on elliptic PDE software, Rice [35] finds average relative performance to vary 30-50 per cent. Obviously, then, for complex software not only, <u>can</u> these portability questions be important, but they <u>are</u> important.

Writing good software increasingly required knowledge of machine architectures and their capabilities. FORTRAN, the dominant mathematical software language has unfortunately lost a main virtue at being a realistic abstraction of how machines work [11]. Should use of other languages increase, relative evaluation would naturally become even more unwieldy. Inconsistent results on different machines can show inadequacies of a code itself or of a machine's compiler or arithmetic, underlining the desirability of a floating point arithmetic standard [15]. While many-machine evaluation of software has been successfully done for large program libraries [6], it is an undertaking of different proportions for complex codes.

generation aids, perhaps using execution path specifications produced by the data flow analyzer.

[6] Execute dynamic test(s);

Each test run can be specified by a triple (PUG, TDC, TOF) of file system entity names. Test runs are made, with results going into a data base for later perusal. This involves automatic instrumenting, compiling, link editing (including fetching of run-time libraries to support monitoring), creating data bases of results, and presenting requested results to the user.

[7] Browse source text and test execution results:

This involves use of the IST browsing facilities to help the user identify and understand errors well enough to fix them.

6.2 Library Implementation

In a second illustration of how Toolpack might be used we consider the implementation of a large subprogram library on different machines. Figure 2 depicts a sequence of activities modeled largely on present NAG Library implementation practice [Hagu 79]; a source text library and test software are assembled, tested and standardized on one system (the "base" system), and then transported to "target" systems for *implementation.* Whereas Figure 1 is broadly applicable to NAG library activities (contribution, validation, standardization, integration) prior to implementation, in Figure 2 the emphasis is on certifying the performance of the transferred software in a new computing environment. The application of a wide range of static and dynamic analysis tests, as envisioned in Figure 1, will probably not be repeated *en masse* in the implemention phase. However, as the subsequent commentary on the major stages in Figure 2 will point out, it is important that such facilities are available if required.

The critical difference between the circumstances implied by Figure 2 and the present pre-Toolpack situation is the availability of a set of uniformly designed reliable tools, either integrated into a system or as standalone utilities, in all relevant environments. In the future a preliminary activity of library personnel before embarking on active technical cooperation with any new collaborator (contributor, implementer or otherwise) may be to ensure that the Toolpack system, or at least a relevant subset of it, is available on the computer(s) used by the other person.

Comments on the Major Activities in Figure 2

[N1] Retrieve and assemble source text:

The appropriate test program, test data collections and library subprograms are assembled from master files.

[N2] Make anticipated changes:

Using a Fortran-intelligent editor or some other transforming device, make anticipated systematic changes to the assembled source text. The TDC's may also

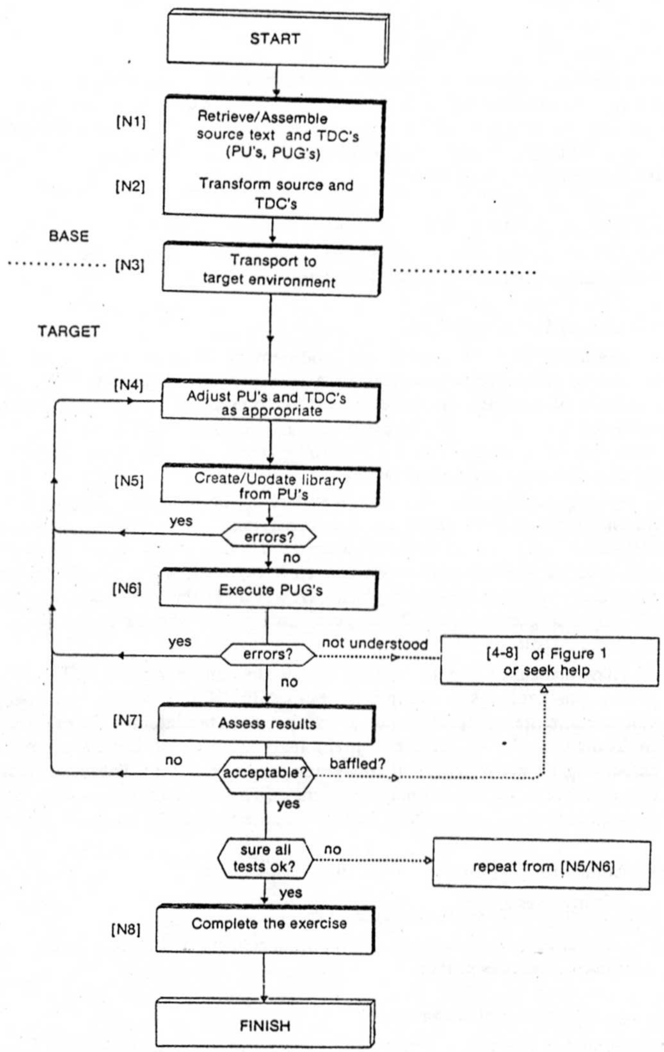

Figure 2.

III. <u>COLSYS AND PASVA3</u>

Many of the recent BVODE codes are initial value type and use available robuts IVP software, but COLSYS and PASVA3 use global techniques which in some sense calculate the solution simultaneously over the entire interval. Since they are described in detail elsewhere [2], [18], only the features relevant to our discussion are treated here.

IV. <u>COMPARISON OF CODES</u>

Although COLSYS and PASVA3 have been used extensively to solve diverse sets of problems, little substantial work on comparing them (or others) to make recommendations concerning the appropriate use for each has been completed. Indeed, attempts to compare several codes have been made, but the efforts have usually been abandoned due to problems in deciding upon objective criteria for comparison. With complex software there is the additional difficulty that familiarity with a code is necessary before it can be fairly assessed. Here we discuss many of the issues to consider in undertaking such a comparison.

Conclusions drawn from a comparison of codes are strictly only for the set of test problems used. The effort in producing a standard set of test problems for BVODEs has been inadequate, one reason being that such problems arise in many different forms [22], [13]. There is also controversy over whether a realistic set can be drawn exclusively from artificial problems, or it "real world" problems are necessary.

Before making any software evaluation, performance criteria must be selected. So long as the software has been designed with basically the same purpose in mind, as is the case for previously-mentioned comparisons, such a selection can be fairly straightforward. However, in an area of high complexity like BVODEs, the difficulties that arise are substantial.

We delineate the possible performance criteria for BVODE codes and discuss problems that arise with their use. They are roughly split up into (1) general "objective" (more quantitative) criteria, (2) general "subjective" (more qualitative) criteria, and (3) subjective criteria particularly relevant to BVODEs.

<u>1a. Timing</u> - The speeds of codes are typically measured using CPU time, although in [5] the desirability of separating I/O time and estimating its variability due to multiprogramming environments is pointed out. COLSYS and PASVA3 use similar I/O features, in marked contrast to the initial-value type codes. For BVODE codes which can require large amounts of storage, not only are times variable from one environment to another because of such factors as paging algorithms, but completely different measures such as the amount of user time required in an interactive environment could be overriding considerations.

The first type of comparison is frequently used to gauge the methods' potential, although even in this restricted context it is not uncommon for conclusions to be reversed (e.g. [28] and [26]). Operation counts for collocation and the Ritz method for solving elliptic PDES (partial differential equations) have been made in [23], [14], [9], and [31]. Each contains modifications of the algorithms giving improved operation counts over the previous results. The second and third view collocation as the more efficient, both in terms of these counts and the authors' resulting codes. The fourth improves upon the counts for Ritz to conclude that it is more efficient. While it is not necessarily true that the superiority of an algorithm in (1) implies the superiority for a code based upon the algorithm, it certainly does not imply the superiority of the code designed from a variant of the algorithm.

Thus, (1) serves a useful purpose in helping one think out how to implement methods, but ultimately the practical comparison of methods come from evaluation of the codes themselves, with the understanding that conclusions are not necessarily valid for future implementations.

There have previously been quite useful software comparisons for numerical quadrature [16], (scalar) nonlinear equation solvers [ ], and codes for IVODEs (initial value problems for ODEs) [30]. In these cases, the codes were intended to solve the same problems and the design criteria were basically the same. The difficulty of evaluation and comparison increases, however, as the complexity increases (see [12], [13] for comparison of nonlinear equation solvers and nonlinear optimization codes).

The problem of evaluating BVODE codes can be particularly intricate, for several reasons. Robust codes have only appeared quite recently [4], and they are being frequently modified as practical experience dictates implementation changes. More importantly, BVODEs are of sufficient complexity that the resulting codes do not lend easily to comparisons. BVODEs arise in many forms [22], [1], and the codes deal with them in different ways. Also, solution of these problems necessitates many types of numerical considerations, and in this sense we say that the area is of "high complexity". In Figure 1 below, BVODE is connected to the other areas of numerical analysis which must be considered in the use of (at least) a large class of the methods; the result is that almost every possible one is included. This leads to BVODE codes having rather different purposes, as we see for the two considered in the next sections.

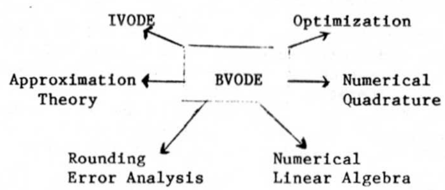

Figure 1.

require changes. The amended software constitutes the "predicted" version.

[N3] Transport the new system:

Stages [N1] and [N2] can take place within the Toolpack file system. The programs and data are now extracted for transfer to a new machine. Because of the scale of the software involved, this transfer is likely to involve magnetic tapes for some time, but may eventually become primarily a direct file-to-file transfer instead.

[N4] Adapt received software:

Depending on the effectiveness of stage [N2] and the outcome of subsequent testing, some adaptation may be necessary, which may involve use of the Fortran-intelligent editor or some other program transformer.

[N5] Create/update library:

This involves using the library compilation features of the host system. A calling dependency index generated by a Toolpack callgraph facility may be useful here. Whereas in individual small-scale program development a source text library mechanism provided by Toolpack is likely to be adequate, in the context of library implementation the desire is to test a pre-compiled library of subprograms.

[N6] Execute tests:

For the reasons given earlier, this testing is probably performed directly under the host system's compiling/linking/executing facilities. If errors occur during testing of particular programs, however, it may be useful to apply the various static and dynamic testing aids under Toolpack to try to determine the causes of the errors. Those errors may be introduced during any of stages [N1] to [N6] or may have arisen at an even earlier stage and only now be manifested.

[N7] Assess results:

For mathematical software in particular, assessing the acceptability of divergence in results in different computing environments cannot easily be automated. Currently there is no intention to include a numerical results comparison utility (as suggested in [Nuge 80] for example), but even a general file comparison utility, which will be supplied with early releases of Toolpack/IST, can be useful in checking results when tests are rerun in (what purports to be) the same environment.

[N8] Complete the exercise:

This involves a number of tasks including the preparation of a release tape and supporting user documents. Unless stage [N4] involves no changes, it is advisable to perform a mechanical comparison of the source text as received in stage [N3], and as modified after [N4]. The generated summary of changes will provide a useful record of the extent of the textual divergence between predicted and corrected versions. The summary, in machine-readable form, may be returned to the originating site for possible incorporation into a source text version control facility within the Toolpack system.

7. IST Implementation Schedule

It is recognized that the IST represents a foray into tool integration on a somewhat higher level and larger scale than previously attempted, and in support of a user community that has not previously been well supported by such tool ensembles. For these reasons there is justifiably considerable doubt that optimal toolsets, integration strategies and user interfaces can be arrived at without significant experimentation and iteration. Hence the IST implementation plan calls for a succession of releases, designed to be increasingly comprehensive, increasingly widely circulated and iteratively well adapted to the perceived and expressed needs and desires of actual users.

The tentative schedule of releases is as follows:

Release -1: Date: November 15, 1981
Audience: Toolpack group only.
Description: Command interpreter plus a few representative tools and tool fragments, some in prototype form.

Release 0: Date: July, 1982
Audience: Toolpack group and selected test sites.
Description: Command interpreter and fairly complete suite of tools described in Section 4. Some of the tools may be in prototype form.

Release 1: Date: Spring, 1983
Audience: First unrestricted public release.
Description: Command interpreter and complete suite of tools.

It is anticipated that by 1983 we will become aware of needs for tools and capabilities that are currently unanticipated. Hence we foresee a continuing sequence of releases incorporating a growing number of tools and functional capabilities. We also expect that such central features as the command language and file system will undergo continuing change and improvement as we benefit from user experience.

Acknowledgments

Our thanks go to Wayne Cowell and Burt Garbow for reading several drafts of this document with great care and for makingmany helpful suggestions. We are also grateful to Gilles Garcia of IMSL for preparing the figures.

References

[Bake 77] B.S. Baker, "An Algorithm for Structuring Flowgraphs," *Journal of the ACM* 24 (January 1977) 98-120.

[Boyl 76] J. Boyle and M. Matz, "Automating Multiple Program Realizations," in: *Proceedings of the Symposium on Computer Software Engineering*, Polytechnic Institute of New York, April 20-22, 1976.

DIFFICULTIES OF COMPARING COMPLEX MATHEMATICAL SOFTWARE: GENERAL COMMENTS AND THE BOUNDARY VALUE PROBLEM FOR ORDINARY DIFFERENTIAL EQUATIONS

by

V. Pereyra* and R. D. Russell**

*Department Computacion, University Central de Venezuela, Caracas

**Department of Mathematics, Simon Fraser University, Burnaby, B.C.
Supported in part by NSERC (Canada) Grant A8781, and by Department Computacion, UCV, while visiting Venezuela under a UNESCO contract.

I. INTRODUCTION

The proliferation of mathematical software has increased the need to develop methods of evaluating this software, in order that the relative merits can be determined in a given environment. The importance of this topic has certainly been realized, as evidenced by the recent IFIP meeting [7], research articles (see especially [5]) and formation of groups to deal with this issue for optimization and statistical software (e.g. see [19] and [17, p 57]).

In this paper, we discuss how hard a comparison of codes really is in a "high complexity area" as defined in §2. The approach is reasonably general and the focus is on the difficulties of comparison. They are considerable and generally not sufficiently appreciated. Discussion of these difficulties is by no means exhaustive, however, as we center on examples from BVODE (boundary value problem for ordinary differential equation) software. In particular we discuss two BVODE codes, COLSYS and PASVA3, which are currently the two most general codes based upon global methods.

One of our purposes is to show that there are serious questions which must be answered before a precise evaluation of the relative merits of BVODE codes can be made. A proper perspective on what one can realistically hope to achieve is necessary (see [12, 13] and §5). Among other conclusions, we emphasize is that it is more appropriate to view codes (here, BVODE codes) as complementing one another rather than competing.

II. BRIEF HISTORY OF COMPARISONS

It is instructive to see how numerical analysts have typically compared numerical methods, and their corresponding successes and failures. Three types of comparison are common: (1) comparison of algorithms using operation counts (possibly incorporating constants in a priori error estimates), (2) comparison of codes, and (3) comparison of methods in some usually unspecified more general sense which results in a linear ordering, viz. the conclusion is made that one method is "better than" another.

empirical evaluations, organized and conducted international conferences on software and testing methodologies, served as repositories of information regarding all of the above, published proceedings of our conferences, and of course contributed to our newsletter. I want to take this opportunity to thank all the members of COAL (current and past) as well as all the members of the Friends of COAL for making my tenure as chair so rewarding and so much fun.

Richard H. F. Jackson,

Chairman

Members of COAL: Past and Present

Michele Benichou	Freerk Lootsma*
Jacques Bus*	Robert Meyer**
Harlan Crowder**	John Mulvey*
Jan de Jung	Richard O'Neill**
Ron Dembo	Susan Powell**
Phillip Gill	Ken Ragsdell**
Harvey Greenberg*	Ron Rardin**
Karla Hoffman**	Patsy Saunders**
Jerry Kreuser	Klaus Schittkowski**
Leon Lasdon**	Robert Schnabel**
Claude Lemarechal**	Jan Telgen**
Melanie Lenard	John Tomlin*

*Previous chairperson

**Current member of COAL

[Donz 80] V. Donzeau-Gouge, G. Huet, G. Kahn and B. Lang, "Programming Environments Based on Structured Editors: the MENTOR Experience," Technical Report, INRIA, France, May 1980.

[Feib 81] J. Feiber, R.N. Taylor, L.J. Osterweil, "Newton — A Dynamic Program Analysis Tool Capabilities Specification," Tech. Report #CU-CS-200-81, Dept. of Computer Science, University of Colo., Boulder, Colo., March 1981.

[Feld 79a] S.I. Feldman, "The Programming Language EFL," Computing Science Tech. Report #78, Bell Laboratories, Murray Hill, New Jersey, June 1979.

[Feld 79b] S.I. Feldman, "Make — A Program for Maintaining Computer Programs," *Software — Practice and Experience* 9 (April 1979) 255-265.

[Fosd 81] L.D. Fosdick, "POLISH-X Transformations," University of Colorado, Dept. of Computer Science, Tech. Report CU-CS-203-81 (May 1981).

[Hagu 79] S.J. Hague, "Implementation of the NAG Library — Principles and Practice," *Statistical Software Newsletter* 4, Gesellschaft fuer Strahlen and Umweltforshung, Munich (1978).

[Hagu 81] S.J. Hague, "The Provision of Editors for the Manipulation of Fortran," Toolpack Document SJH 11112 (November 1981). Available from Applied Mathematics Division, Argonne National Laboratory, Argonne, Ill. 60439.

[Hans 80a] D.R. Hanson, "A Portable File Directory System," *Software — Practice and Experience* 10 (August 1980) 623-634.

[Hans 80b] D.R. Hanson, "The Portable I/O System PIOS," University of Arizona, Dept. of Computer Science, Tech. Report #80-6a (April 1980, revised December 1980).

[JPL 81] "SFTRAN III Programmer's Reference Manual," JPL Document # 1846-98, Jet Propulsion Laboratory, Pasadena, Calif., April 1981.

[Kern 75] B.W. Kernighan, "Ratfor — A Preprocessor for a Rational Fortran," *Software — Practice and Experience* 5 (October 1975) 395-406.

[Mill 82] W.C. Miller, "Tools for Numerical Programming," in: *The Relationship Between Numerical Computation and Programming Languages*, J. Reid, ed., North-Holland, 1982.

[Myer 81] E.W. Myers and L.J. Osterweil, "BIGMAC II: A Fortran Language Augmentation System," *Proceedings of the Fifth International Conference on Software Engineering*, (March 1981) 410-421.

[Nuge 80] S.M. Nugent, "Automatic Comparison of Numerical Result Files," in *Proceedings of the Workshop on Software Adaptation and Maintenance*, R. Ebert, J. Luegger and L. Goecke, eds. North-Holland, 1980.

[Oste 76] L.J. Osterweil and L.D. Fosdick, "DAVE — A Validation, Error Detection, and Documentation System for FORTRAN Programs," *Software — Practice and Experience* 6 (September 1976) 473-486.

[Ryde 74] B.G. Ryder, "The PFORT Verifier," *Software — Practice and Experience* 4 (October 1974) 359-377.

[Teit 81] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Comm. ACM* 24 (September 1981) 563-573.

[Ward 81] W. Ward and J. Rice, "A Simple Macro Processor," Toolpack Document WW/JR 10921 (September 1981). Available from Applied Mathematics Division, Argonne National Laboratory, Argonne, Ill. 60439.

Chairman's Comments

At the MPS Symposium in 1973 at Stanford, Michele Balinski, John Tomlin, Jim Kalan, Jerry Kreuser and I met for lunch to discuss the state of computational testing in the mathematical programming community. The result was the formation of the Working Committee on Algorithms. (The "Working" was later dropped to quit implying that other committees did not work.)

The topics discussed at that meeting eventually grew into a group of goals and objectives for the Committee on Algorithms. They are listed on the inside cover of this newsletter. I consider it a priviledge to have been a member of this committee since its inception, and to have been able to observe, as well as participate in, its growth during those years.

As I prepare to step down as chair, I must say I am pleased to turn over the office to Karla Hoffman who has worked tirelessly as editor of this Newsletter and is responsible for making it so valuable to those interested in computational mathematical programming. Further, I am happy to report that Jan Telgen has accepted editorship of the COAL Newsletter. Since joining the committee, Jan, too, has worked hard to further its goals, and I feel he is an excellent choice to take over from Karla. I am confident that the committee is in good hands for the next 3 year term.

Since its inception, the members of COAL, who, for the record, are listed below, have all worked to improve the state of computational testing.

They have organized computationl MP sessions at national and international conferences, collected and disseminated test problems and codes, performed research in the methodology for conducting

COMMITTEE ON ALGORITHMS of the
MATHEMATICAL PROGRAMMING SOCIETY

COAL OBJECTIVES

The Committee on Algorithms is involved in computational developments in mathematical programming. There are three major goals: (1) ensuring a suitable basis for comparing algorithms, (2) acting as a focal point for computer programs that are available for general calculations and for test problems, and (3) encouraging those who distribute programs to meet certain standards of portability, testing, ease of use, and documentation.

NEWSLETTER OBJECTIVES

The newsletter's primary objective is to serve as a forum for the Friends of COAL. Through an informal exchange of opinions, members have an opportunity to share their experiences. To date, our profession has not developed a clear understanding on the issues of how computational tests should be carried out, how the results of these tests should be presented in the literature, or how mathematical programming algorithms should be properly evaluated and compared. These issues will be addressed in the newsletter.

DISTRIBUTION OF THIS NEWSLETTER

This newsletter is mailed to every member of the Mathematical Programming Society and to all "friends" of COAL. If you are not presently receiving this newsletter and would like to, please write to the editor requesting that your name be added to the list of "friends" of COAL. There is currently no charge for this newsletter.

Mathematical Progamming Society
Committee on Algorithms Newsletter

No. 7: September 1982 — Karla L. Hoffman, Editor

Contents:

DR. KARLA L. HOFFMAN
UNITED STATES DEPARTMENT OF COMMERCE
NATIONAL BUREAU OF STANDARDS
CENTER FOR APPLIED MATHEMATICS
WASHINGTON, D.C. 20234

T. Steihaug
Dept. Mathematical Sciences
Rice University
P.O. Box 1892
Houston, TX 77001
USA

PRIORITY MAIL