Nonlinear Integer Programming

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 \( A_j = S_j/n_j \) (\( j = 1, 2, \ldots, 22 \)).

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 normalized 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 \( ST_j = \sum_{i} NST_{ij} \), and the average \( AV_j = ST_j/N_j \), where \( N_j \) 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 \( AV_j \).

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

The following observations can be made:
The primary argument of interest in the experimental work...
TSP Heuristics

the procedures examined in this Section are polynomial 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 asymmetric 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 multidimensional 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


Nonlinear Integer Programming

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 RNLMP involves the following:

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

   b. The nonlinear code OPT [Gabriele and Ragade] 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 (NLNP).

Table 1

Analysis of Variance (Normalized Solution Time)

<table>
<thead>
<tr>
<th>Source of Variation</th>
<th>Sum of Squares</th>
<th>D.F.</th>
<th>Mean Square</th>
<th>F-Ratio</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Effects</td>
<td>12,000</td>
<td>6</td>
<td>2,000</td>
<td>30.621</td>
<td>.001</td>
</tr>
<tr>
<td>Variable Selection</td>
<td>1,243</td>
<td>2</td>
<td>621</td>
<td>9.513</td>
<td>.001</td>
</tr>
<tr>
<td>Node Selection</td>
<td>5,783</td>
<td>2</td>
<td>2,892</td>
<td>44.272</td>
<td>.001</td>
</tr>
<tr>
<td>Heuristic Selection</td>
<td>4,937</td>
<td>2</td>
<td>2,469</td>
<td>37.796</td>
<td>.001</td>
</tr>
</tbody>
</table>

2-Way Interaction

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable - Node</td>
<td>.076</td>
<td>4</td>
<td>.019</td>
<td>.282</td>
<td>.883</td>
</tr>
<tr>
<td>Variable - Heuristic</td>
<td>.010</td>
<td>4</td>
<td>.002</td>
<td>.037</td>
<td>.997</td>
</tr>
<tr>
<td>Node - Heuristic</td>
<td>.142</td>
<td>4</td>
<td>.036</td>
<td>.545</td>
<td>.703</td>
</tr>
</tbody>
</table>

Source of Variation

<table>
<thead>
<tr>
<th>Sum of Squares</th>
<th>D.F.</th>
<th>Mean Square</th>
<th>F-Ratio</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-Way Interactions</td>
<td>.251</td>
<td>8</td>
<td>.031</td>
<td>.680</td>
</tr>
<tr>
<td>Variables-Node-Heuristic</td>
<td>.251</td>
<td>8</td>
<td>.031</td>
<td>.680</td>
</tr>
</tbody>
</table>

Explained       | 12,480 | 26     | .680     | 7.349  | .001         |
| Residual       | 36,773 | 563   | .065     |         |              |
| Total          | 49,253 | 589   | .084     |         |              |

26
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 [Adrabiński & Sysło 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 7NN, 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 specifications 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 provides 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 non-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 CMAL Newsletter and takes full responsibility for any oversights in the presentation.
This paper can be obtained by writing to either of the authors.

Economics a la large

The economic order quantity in a given market is determined by the following factors:
- Supply and demand factors
- Cost factors
- Market conditions
- Government regulations
- Technological advances
- Consumer behavior

In order to satisfy the above factors, a comprehensive approach to economic order quantity is required.

REFERENCES


Equation 5.1: Economic Order Quantity

EOQ = \sqrt{2DS/H}

where:
- E = Economic order quantity
- D = Demand rate
- S = Ordering cost per order
- H = Holding cost per unit per year

Example 5.1: A company has a demand rate of 100 units per year, an ordering cost of $50 per order, and a holding cost of $2 per unit per year. What is the Economic Order Quantity?

EOQ = \sqrt{2 \times 100 \times 50 / 2} = 100 units

The Economic Order Quantity is calculated as 100 units.
Large-scale Nonlinear Optimization

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 SPRES 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.

5. 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

\[\text{IP Codes}\]


<table>
<thead>
<tr>
<th>Code</th>
<th>Organization</th>
<th>Computer</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apex III</td>
<td>Control Data</td>
<td>Cyber 70 series, models 72, 73, 74 and 76; 6000 series, 7600 series</td>
<td>1975</td>
</tr>
<tr>
<td>Bloodhound</td>
<td>Ketron Inc.</td>
<td>IBM S/370, models 145 and above. IBM 303X, Andahl 470vx</td>
<td>?</td>
</tr>
<tr>
<td>LAMPS</td>
<td>ANS Ltd.</td>
<td>DEC20 series, DEC/VAX 11/780, PAPR 550 upwards, Perkin-Elmer 3200 series, Harris</td>
<td>1980</td>
</tr>
<tr>
<td>MIL/370</td>
<td>IBM</td>
<td>370 series</td>
<td>1973</td>
</tr>
<tr>
<td>MPS</td>
<td>Honeywell</td>
<td>Series 60 (level 66)</td>
<td>1975</td>
</tr>
<tr>
<td>Sciconic</td>
<td>Scicon Ltd.</td>
<td>Univac 1100 series, DEC/VAX 11/780</td>
<td>1976</td>
</tr>
</tbody>
</table>
II. CONCLUSIONS

A. Algorithm Capabilities

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 or 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 convergence 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 any 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 these criteria and to then build and distribute the
Because of the region requirements of the FORTRAN compiler used on the host computer (see Section 1.9), 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 iterations (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 11.4.1 and 11.8.1 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 to use the MINOS system with relative ease. However, the individual codes to set up and 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.

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 left-hand and right-hand 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 SFRTRAN [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
The general non-separable nonlinear (NLP) problems can be stated as:

\[
\begin{align*}
\text{minimize} & \quad f(x) \\
\text{subject to} & \quad \begin{cases}
\sum \leq b & \text{general non-separable, non-linear constrains} \\
\sum \leq b & \text{upper and lower constant ranges}
\end{cases}
\end{align*}
\]

where \( \sum \) = variables;
\( f(x) \) = general non-separable, non-linear function;
\( \sum \leq b \) = general non-separable, non-linear constrains;
\( \sum \leq b \) = upper and lower constant ranges.

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

\[
\begin{align*}
\text{minimize} & \quad f(x) \\
\text{subject to} & \quad \begin{cases}
\sum \leq b & \text{general non-separable, non-linear constrains} \\
\sum \leq b & \text{upper and lower constant ranges}
\end{cases}
\end{align*}
\]

where \( \sum \) = variables;
\( f(x) \) = general non-separable, non-linear function;
\( \sum \leq b \) = general non-separable, non-linear constrains;
\( \sum \leq b \) = upper and lower constant ranges.

I. Comparison Criteria

A. Algorithm Capabilities

This section contains a general survey of the type and range of problems for which each code is designed and compared in the general group.

1. CPU (Computer) Time

(b) The CPU times listed are the virtual CPU times required for each problem and are not to be interpreted as actual CPU times. Since performance of problem-solving has been done on the test computer only, the listed experiences will be valid within each one.

(c) Although the actual CPU times are valid within one percent, CPU times are saved to include the interprocessor times. Since the problems have been run in a distributed system, the actual CPU times will vary, and the listed CPU times are valid within each one percent.

(d) The CPU times are in general actual CPU times. Although the actual CPU times are valid within one percent, CPU times are saved to include the interprocessor times. Since the problems have been run in a distributed system, the actual CPU times will vary, and the listed CPU times are valid within each one percent.

II. Debugging

4.5 Dynamic Debugging Aid

Debugging is facilitated by the ability to scrutinize to arbitrary levels of detail the execution of a program that is behaving incorrectly. Thus the Newtopack debugger assists the user in generating and examining diagnostic output for ease of understanding. There are currently no firm plans to embed such a tool.

4.6 Static Error Detection and Validation Aid

Toolpack is designed to detect errors at the dataflow analysis level. The tools also work in a range of capabilities that includes those supplied by the DAVE data flow analysis system [One 76].
This is the first independent comparison of either code and is intended to serve both as an evaluation 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:

\[
\begin{align*}
\text{minimize} & \quad c^T x \\
\text{subject to} & \quad \mathbf{r} \leq A x \leq \mathbf{r} \\
& \quad \mathbf{b} \leq x \leq \mathbf{b} \\
\end{align*}
\]

where:

- \( x \) = variables;
- \( c^T \) = cost coefficients;
- \( A \) = constraint matrix coefficients;
- \( \mathbf{r}, \mathbf{r} \) = upper and lower constraint ranges;
- \( \mathbf{b}, \mathbf{b} \) = 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, syntax 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
school. Lunches are prepared by Central Office. Therefore, in the event of an emergency, school officials can provide food to the students. A. Dr. Robert J. Smith's contact number is 555-1234.

The exact location of the nearest emergency shelter is available.

Conclusion

Summary of Text

Conclusion

REFERENCES

ABSTRACT

Introduction

Methods

Results

Discussion

References

Appendices
efficient and correct Fortran 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 [Boyd 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 73], 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 Integration 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 intricacies 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
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 SFRTRAN [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 for generic 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-shot-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
34. 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 implementations. Indeed, order of magnitude improvements are sometimes made in BVODES 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 COLSTS and PASYA3) would have prohibitive storage requirements. However, in the typical situation where only the missing conditions at are desired, by judiciously saving 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 areas insurmountable when comparing codes like COLSTS and PASYA3. 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, especially [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 inanciations 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

```
C5 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;
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 interaction 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 \( e = \text{the actual (absolute) error} \) and \( \hat{e} = \text{the estimated error} \), then \( e \) is usually accurate to within 1-2 digits for PASVA3 while usually \( 0.1 < \frac{|e - \hat{e}|}{e} < 10 \) 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, \((3.1) - (3.2)\) \( u \) is the approximate solution, and \( \text{TOL} \) is the requested user tolerance for the \( j \)th derivative \((1 \leq j < m-1)\), then COLSYS attempts to satisfy

\[
(4.1) \quad \left| u^{(j)} - \hat{u}^{(j)} \right|_1 < \text{TOL}_j (1+|u^{(j)}|_1), \quad (1 \leq j \leq N)
\]

where \( |u|_1 \) is the maximum norm for \( u(x) \) over the \( i \)th subinterval. PASVA3 uses the absolute error tolerance \( \text{TOL} \) for all solution components (all solution derivatives for \((3.1) - (3.2))\), so it attempts to satisfy

\[
(4.2) \quad \|v - \hat{v}\| < \text{TOL}
\]

at the mesh points, where \( v = (u, u', \ldots, u^{(m-1)})^T \) and \( \hat{v} = (\hat{u}, \hat{u}', \ldots, \hat{u}^{(m-1)})^T \). (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 \( \text{TOL} \), 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.

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 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.
Storage - Relative storage requirements are slightly affected by the machine precision used, but the measurement problem itself for a given example is nontrivial. COLUMNS, for example, will use different mesh selection strategies, depending upon whether or not the allocated 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.

Portability - A large code's portability is almost impossible to ensure without actually testing it in each environment. Although COLUMNS 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 (1.0+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 ε → 0 in an IVP, making it more difficult, the solution is sometimes computed efficiently and sometimes not, but the transition is erratic as ε → 0; (iv) results on the CRAY 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 more "natural" order) to allow for vector processing, the time for linear system solutions in a biharmonic equation solver [3] is reduced to less than 1/7 of the original time when using a CRAY 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 can these portability questions be important, but they are important.

Writing good software increasingly requires 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, undermining 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.

TLPACK

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 object 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 [Hag 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 at this point in the implementation 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
Cooperation

In a cooperation environment, an interesting environment could be created. The cooperation environment could be created by following the guidelines, creating different situations, and ensuring that the environment is consistent enough to provide meaningful outcomes. To achieve this, we can create multiple environments, each with its own characteristics. This can help us understand the dynamics of cooperation, and how it can be used to create meaningful outcomes.

The guidelines for cooperation include:

1. Consistency of Goals
2. Openness and Transparency
3. Trust and Communication
4. Shared Understanding of the Problem

These guidelines help ensure that the cooperation environment is effective and meaningful. By following these guidelines, we can create environments that are conducive to cooperation and meaningful outcomes.
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 [1], 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 IVODE 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, IVODEs are of sufficient complexity that the resulting codes do not lend easily to comparisons. IVODEs 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, IVODE 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 IVODE codes having rather different purposes, as we see for the two considered in the next sections.

Figure 1.
Chairman's Comments

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 Freerik Lootsma*
Jacques Bus* Robert Meyer**
Harlan Crowder** John Mulvey*
Jan de Jung Richard O'Neill**
Ron Denbo Susan Powell**
Phillip Gill Ken Ragade**
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. Huef, G. Kaha and B. Lang, "Programming


performed research in the methodology for control of

confinement, collection, and transportation of hazardous "waste" and "spinoff".

They have, however, contributed to the development of national and international

milestones that will lead to the successful implementation of the system of confinement and storage.

Since the inception of the committee, the membership has grown, for the record, are listed

in the committee's minutes. The committee's minutes are kept on file in the office of the Chairman.

The committee's minutes are also available to the public on request.

In conclusion, I would like to express my appreciation to the members of the committee for the work they have performed. Their contributions have been invaluable in the progress of the project.
COMMITTEE ON ALGORITHMS of the
MATHEMATICAL PROGRAMMING SOCIETY

CHAIRMAN
Richard H. F. Jackson
Center for Applied Mathematics
National Bureau of Standards
Washington, D. C. 20234
(301) 921-3855

EDITOR OF THE NEWSLETTER
Karla L. Hoffman
Center for Applied Mathematics
National Bureau of Standards
Washington, D. C. 20234
(301) 921-3855

Jacques C. P. Bus
Haagwintel 61
1391 ZZ
Abcoude, The Netherlands

Patsy B. Saunders
Center for Applied Mathematics
National Bureau of Standards
Washington, DC 20234
(301) 921-3855

Harlan P. Crowder
IBM Thomas J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
(914) 945-1710

Klaus Schittkowski
Institute für Angewandte Mathematik und Statistik
Universität Würzburg
D-67 Würzburg
West Germany

Jan L. de Jung
Amerikalaan 83
5691 EC Son
Nederland

Jan Telgen
RAOBANK NEDERLAND
Department of Applied Mathematics
Laan Van Eikenum 9 (ZL-C-170)
3705 AR Zeist
The Netherlands

Leon S. Lasdon
Department of General Business
School of Business Administration
Austin, TX 78712
(512) 471-3322

EX OFFICIO MEMBERS
J. Abadie, Chairman, MPS
29, Boulevard Edgar-Quintet
75014 Paris, France

A. C. Williams, MPS Exec. Comm.
Mobil Oil Co. Technical Center
P. O. Box 1025
Princeton, NJ 08540

Richard F. O'Neill
EI 612, Room 4447
Department of Energy
Washington, DC 20461

Philip Wolfe, Vice-Chairman, MPS
IBM Research 33-2
P. O. Box 218
Yorktown Heights, NY 10598

John M. Mulvey
School of Engineering/Applied Science
Princeton University
Princeton, NJ 08540
(609) 452-5423

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.

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.
Committee on Algorithms Newsletter

Mathematical Programming Society

No. 17 September 1982

Karla L. Hoffman, Editor