JOURNAL OF RESEARCH of the National Bureau of Standards
Volume 83, No. 6, November–December

A Critical Review of Comparisons of Mathematical Programming Algorithms and Software (1953–1977)

Richard H. F. Jackson*

Center for Applied Mathematics, National Bureau of Standards, Washington, D. C. 20234

and

John M. Mulvey*

Graduate School of Business Administration, Harvard University, Boston, Massachusetts 02163

June 27, 1978

Since the introduction of general-purpose computers during the early 1950's, competing techniques of mathematical programming have been developed, analyzed empirically, and compared. In this paper we survey fifty articles (spanning the period 1953–1977) which report the computational testing of mathematical programming algorithms. Our intention is to document the performance measures which were used, the standards which were maintained, and the forms in which the results were reported for these experiments. Trends in methodology are noted, and suggestions for improving the current state of affairs are offered.

Keywords: Code comparison; comparison of mathematical programming software; evaluation; testing.

1. Introduction

Computational experiments with mathematical programming techniques have taken place throughout the past twenty-five years. In 1953, for instance, Hoffman et al. [A1][1] compared the numerical efficiency of the simplex method, the fictitious play method of G. Brown, and the relaxation method of T. S. Motzkin for solving several symmetric matrix games. Since 1968, an increasing number of experimental studies have been published in scholarly journals. Unfortunately, the methodology for conducting such computational experiments has not received systematic study, and no set of generally accepted guidelines has been available.

The purpose of this paper is to survey a substantial sample of the published articles and working papers in this field. The fifty papers chosen for the study are listed in Appendix A. Our intention is to document the methods employed in conducting these experiments, including the controls maintained, the performance measures used, the extent of a priori experiment design, and the forms in which the results were reported.

In selecting articles, we attempted to locate as many bona fide computational experiments as possible. We rejected studies presenting "new" algorithms, even when extensive computational testing was provided. We wished to restrict attention to papers whose primary contribution was presentation of the empirical evidence. On the basis of its abstract, title, and/or introduction, each of the selected papers indeed qualified as the result of computational experiments. A majority of them evaluate mathematical programming algorithms; some evaluate tactical choices within an algorithm; others evaluate the computational efficiencies of mathematical programming software. All of them present empirical results as evidence of superior or inferior performance, a feature distinguishing them from publications giving only theoretical bounds or estimates for computational effort.

The papers that were deliberately left out of this survey are numerous; no doubt this is true of unintentional omissions as well. Studies that were clearly inferior were excluded; we did not want to single out inadequate efforts since we felt these were best left undisturbed. A number of works were not included because they were

* The authors are members of the Committee on Algorithms (COAL) of the Mathematical Programming Society. COAL is concerned with computational aspects of mathematical programming. A related paper is "Guidelines for Reporting Computational Experiments in Mathematical Programming," which may be requested from John M. Mulvey. John M. Mulvey's present address: School of Engineering/Applied Science, Princeton University, Princeton, New Jersey 08540.

[1] Figures in brackets indicate literature references at the end of this paper.

not accessible to us or because we were unaware of them at the time of the survey. Such papers might well be included in a revision of this survey.

As indicated in table 1.1, most of the major areas of mathematical programming are represented, as are the journals which most frequently publish results about mathematical programming. There are ten articles each on linear programs and on integer programs. The categories of unconstrained optimization, network, nonlinear programs, and shortest paths have five to seven articles apiece. Quadratic programs, knapsack problems, systems of nonlinear equations and geometric programs are represented by one to three articles each. Twenty-six of the fifty articles appeared in refereed journals, whereas eleven are chapters of books or were presented in proceedings of conferences. Eleven papers are mainly recent technical reports, many of which have been subsequently submitted for publication. Two theses were included.

TABLE 1.1. *Articles selected for the critical review.*

Areas of MP Represented	Number
Linear Programs	10
Integer Programs	10
Unconstrained	7
Shortest Paths	6
Nonlinear Programs	5
Networks (Min-Cost Flow)	5
Geometric Programs	3
Systems of Nonlinear Equations	2
Quadratic Programs	1
Knapsack Problems	1
	50

Journals Represented	Number
Mathematical Programming	8
Management Science	4
Communications of the ACM	3
Operations Research	2
Journal of the ACM	2
SIAM Journal of Numerical Analysis	2
Journal of the SIAM	1
ACM Transactions on Mathematical Software	1
OPSEARCH	1
Australian Computer Journal	1
Computer Journal	1
	26

Methods of Publication	Number
Published Journals	26
Chapters in Books & Proceedings of Conferences	11
Technical Reports*	11
Theses and Dissertations	2
	50

* Many have been subsequently submitted for publication.

Figure 1.1 presents a histogram of the selected articles as a function of publication date. Corresponding to the three commonly recognized "generations" of computers, we have identified three fairly distinct generations of computational experiments. We were able to locate only one article that appeared prior to the introduction of programming languages (FORTRAN was officially introduced in 1959)—the paper of Hoffman et al. [A1], which appeared in the first generation. Seven papers were found with publication dates between 1960 and 1966—corresponding to the second generation. The majority (42) of the fifty papers appeared after the introduction of third-generation computers, such as IBM's 360 series, in 1968. Note that there seems to be an upward trend in the number of papers per year.

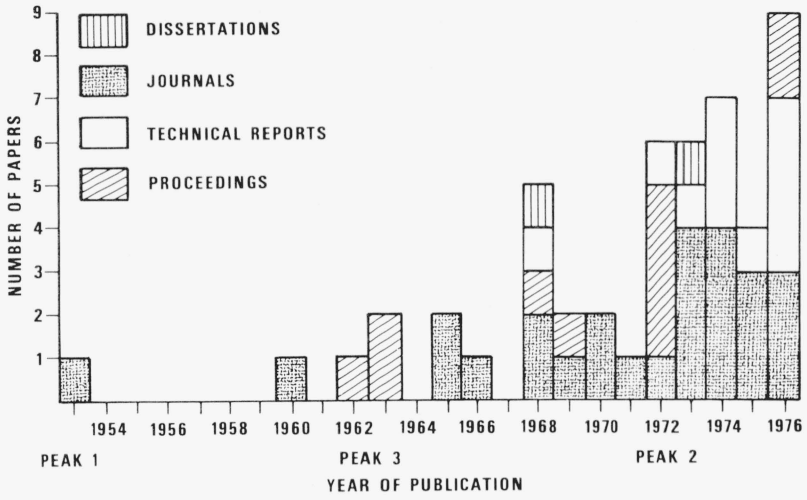

FIGURE 1.1. *Distribution of the sample over time.*

This survey is arranged according to the following topics: elements of the experiment; experiment design; and empirical results. Section 2 addresses information pertaining to the elements of the experiment, i.e., the algorithms and codes which were employed. Were the algorithms presented in unambiguous detail or were they referred to by name only? Could the computer software be reproduced by other researchers? Were descriptions of the information structures provided?

The experiment designs are reviewed in section 3. A careful experiment design, in which goals and the plan for their achievement are clearly laid out, is the keystone of experimental analysis. We shall see that one of the principles of experimental investigation, viz, replication, is woefully neglected.

Section 4 takes up the issue of how the results of the experiments were presented. For instance, the performance measures that were used are noted. Incidences of extrapolation and speculation are also indentified. Section 5 offers suggestions for improving the state of computational experimentation, a topic taken up more fully in Crowder, et al. [2]. References for the surveyed papers are given in appendix A, and appendix B contains the form used in evaluating these papers. A copy of the unabridged data (suitably disguised) is available to interested researchers.

2. The Elements of the Experiment

In this section, we are concerned with the amount of detail with which the elements of the computational experiments were described. We define these elements to include the algorithms themselves and the related computer software. However, we decline involvement in the debate over where an algorithm ends and its computer implementation begins. This review uncovered many instances of confusion over this distinction, and two papers [A1, C11] are noteworthy in their discussions of the matter. It is interesting to note that [A1] is the earliest paper in our sample and [C11] is one of the latest. In our survey, if an idea was presented as part of the solution technique or methodology, it was identified as part of the algorithm. On the other hand, if the idea appeared explicitly in the code and was presented in that light, we included it in our analysis of the software. This dichotomy may not apply generally, but it proved workable for our purposes.

2.1 Algorithms

Certainly there are many ways to describe an algorithm, and as expected, we encountered various levels of detail in presentation. Most descriptions fall into one of the following categories: (1) flowchart; (2) primarily mathematical; (3) verbal; or (4) reference to other documentation. Our concern, however, was whether enough information was provided so that a reader could understand all of the algorithms to the extent that he could

repeat the experiment, given sufficient resources. (Repeatability here assumes all other aspects of the experiment were sufficiently documented.) Thus, when a paper described one algorithm in detail and referenced the remaining algorithms, we assumed that sufficient detail was provided. When, on the other hand, any algorithm was referred to by name only, we assumed that sufficient detail was lacking. Figure 2.1 indicates our estimate of the number of papers providing sufficient detail for a determined reader to be able to understand and repeat the experiment.

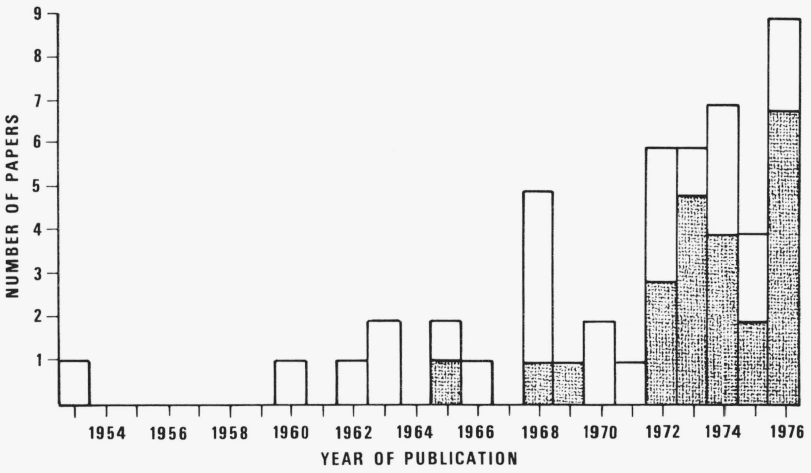

FIGURE 2.1. *Number of papers that provided "sufficient" discussion of the algorithm.*
(Shown as a portion of all papers in our study.)

2.2 Software

Since the papers are reports of computational experiments, we feel that it is crucial for the codes employed to be described adequately. Table 2.1 shows the frequency with which different methods of presentation were used. The numbers of papers sum to 54 since some papers used more than one method.

TABLE 2.1. *Frequencies of methods of software presentation.*

Method	Number of papers
Referred to by name only	29
Referenced to other descriptions	13
Extensive discussion	2
Brief discussion	7
Extensive flow chart	1
Brief flow chart	1
Listings included	1

In many cases, for example codes that solve large-scale transportation problems, the computer implementation has a critical effect on the outcome of the experiments since the empirical results are highly sensitive to the method of implementation. Indications of data structures within the codes is one test of software discussion completeness. Figure 2.2 depicts the number of papers that mentioned this topic, which is clearly considered more important in some areas of mathematical programming than in others. A related subject is the total storage requirements of the computer software. Five papers gave the amount of storage required; eight papers mentioned the amount of working storage, and one paper referred to offline) storage.

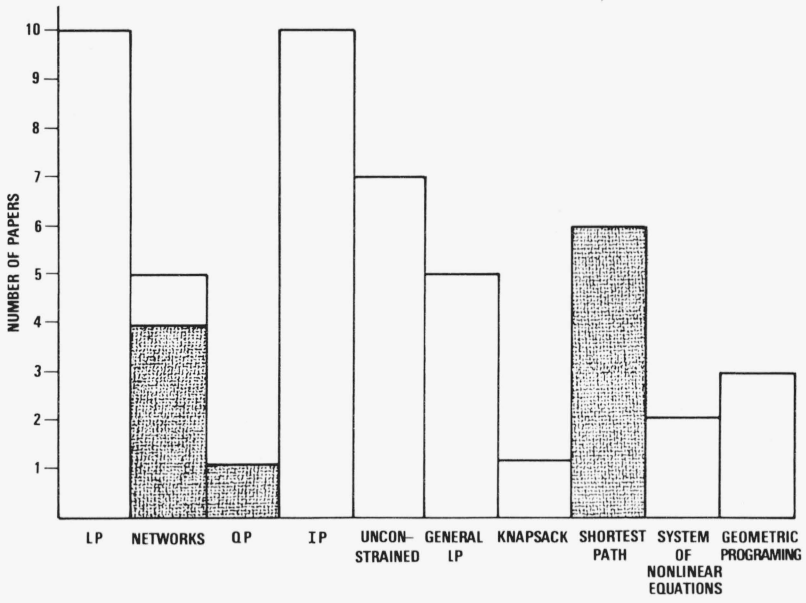

FIGURE 2.2. *Number of papers that included a discussion of data structures, displayed by areas of mathematical programming.*

As another indication of the lack of attention paid to the software elements, we note that just one paper presented the time required for input/output. In the folklore of computational experimentation, it is generally assumed that problems are solved without using external (offline) memory; however, we could not determine the extent to which this tradition was followed. Further evidence of the lack of attention is provided by the fact that in twenty-eight papers the computer language for at least one of the codes used could not be determined. (Of the twenty-nine papers that mentioned language for at least one of the codes, 79% used FORTRAN, 14% used machine language, and 7% used ALGOL.)

Other important topics regarding the computer software are portability, ease of use, and availability. One paper [B7] discussed the ease of use of the software used, and is noteworthy in its attempt to quantify this subjective measure. Two papers advanced portability as a performance indicator; one of these tested it. Availability of the software was mentioned in nine papers.

Finally, we note the number of papers discussing tolerance settings, since tolerance choice is another factor that can affect the empirical results. Ten papers addressed this topic. Figure 2.3 presents a histogram over time of these papers as a portion of all relevant papers. Techniques employing integer arithmetic, such as shortest paths and minimum cost-flow networks, are not affected by tolerance settings and were omitted.

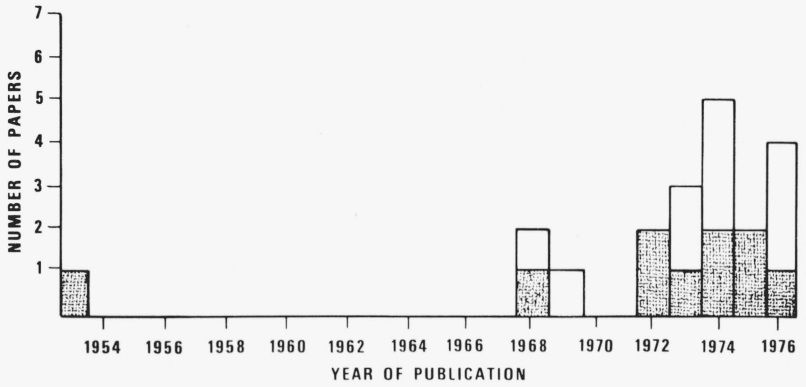

FIGURE 2.3. *Number of papers that discussed tolerance settings.*

2.3 Problem Class

We examine next how each paper characterized the class of problems to which it referred. In most cases (80%), the problem class was referred to by generic name only (e.g., integer programming problems), or by a brief mathematical definition. The remaining 20% included a more complete discussion. In addition, eight of the papers discussed appropriate areas of application for using the technique.

An area of importance in experimental design is the statistical relationship between problem class (the population) and specific representatives (the sample) of that class chosen for analysis. Some mathematical programmers feel that this relationship is the key to improving algorithm and code comparisons. Current research centers on viewing the problem class as a well-defined population from which "random" samples of test problems can be chosen. This approach allows statistical inferences to be drawn about code performance on problems other than the ones used. Five papers addressed this topic.

3. Experiment Design

We begin this section on experiment design with the following quote from Design of Experiments, A Realistic Approach [1].

> "Unfortunately, there are cases in which the sole purpose of the experiment is to 'prove' what the experimenter already 'knew'. This type of experiment frequently is conducted so that the 'known' result will occur no matter whether it should or not. This type of experiment cannot be condoned by persons seeking the truth. On the other hand, a worse condition may exist where people run experiments fully intending to be honest but being completely unaware of their incompetency in conducting experiments intelligently. Frequently, experiments are run so that the effect of the factor of interest is disguised by the effect of another factor not considered. This latter factor is then ignored or considered unimportant, yet, in the long run, it is the real cause."

The degree to which researchers controlled the factors affecting the outcome of their experiments will be investigated next. As we shall see, there is considerable room for improvement in the design of computational experiments. Indeed, one of our questions asked whether the goals of the experiment were clearly defined. Only one-third of the papers contained statements of objectives extending beyond general statements about comparison or evaluation.

We identified three areas that contain factors significantly affecting the outcome: (1) the problems on which the codes were tested, (2) the computer environment, and (3) the controls with which these two areas were governed.

As previously mentioned, the essential question involves how much understanding and control of the important aspects of experiment design were indicated in the papers. For example, we cannot directly discover how well the computer environment was understood by the experimenter. Nevertheless, we feel that if a basic component such as name of the compiler was not specified, there is reason to suspect that other influential variables were not considered when the experiment was designed.

3.1 Test Problems

There has been considerable debate in recent years over whether hand-picked problems (either arising in practice, or specially constructed) or randomly generated problems should be used in a computational study. We will not discuss this issue, but it is interesting to note the totals shown in table 3.1, where the breakdown of hand-picked and generated problems is given for each of the ten areas of mathematical programming represented in our study. The use of randomly generated problems appears to be more common in certain areas, such as networks (minimum cost flow and shortest path), than in others, such as constrained mathematical programs. Random generation is favored when large-scale problems (many variables) are used, and hand-picked examples otherwise.

Regardless of the types of test problems used, there is a need to provide sufficient information so that the experiment can be replicated by other researchers and properly evaluated by referees. Thus, when randomly generated problems are used, an important question is whether the generator is available. There were twenty-three papers that used randomly generated problems. In twelve of these, the authors developed their own

TABLE 3.1. *Number of times hand-picked and randomly generated problems were used.*

	Hand-Picked	Generated	Both
Linear Programs	5	4	1
Networks	0	5	0
Quadratic Programs	0	1	0
Integer Programs	7	2	1
Unconstrained	7	0	0
General Nonlinear Programs	2	2	1
Knapsack	0	0	1
Shortest Path	1	4	1
System of Nonlinear Equations	2	0	0
Geometric Programming	3	0	0

generators; in one case, it was developed elsewhere; and in ten other papers the origin of the generator was not divulged. Although eighteen of the twenty-three discussed the method of generation, only four stated that the generator was publicly available.

In the case of the thirty-two articles that used hand-picked problems, sixteen of these provided a reference to the origins of the problems, twenty included a description of each problem, and nineteen provided references to other descriptions. Twenty-two of the thirty-two mentioned that their test problems were available to other researchers.

It is well known in many areas of optimization that the density of the coefficient matrix plays an important role in code performance. Of the twenty-three studies that used test-problem generators, eleven papers indicated that the density could be regulated, eight indicated that the density could not be regulated, and five papers did not mention the topic. Although the density of the coefficient matrix for hand-picked problems is not directly controllable, in only six cases (out of thirty-two) were the densities of test problems provided.

Another important aspect of the experiment design is the number of test problems to be solved, i.e., the sample size. Table 3.2 displays, as a function of time, the number of papers that used various sample sizes. Notice the trend to larger samples.

TABLE 3.2. *Temporal trend in sample size.*

NUMBER OF PROBLEMS	1954	1956	1958	1960	1962	1964	1966	1968	1970	1972	1974	1976
GREATER THAN 100					1					1	1	1	3	6
51-100								1		1		2	1	1	1
21-50	1						1	2	1				1		2
11-20									1	1	2		2		
1-10				1	2	1	1	2		2	1	3	2		

YEARS OF PUBLICATION

A related topic involves the blocking of test cases according to size or type, and their subsequent replication within each category according to analysis of variance principles. Figure 3.1 presents a graph of the papers employing this concept of experiment design.

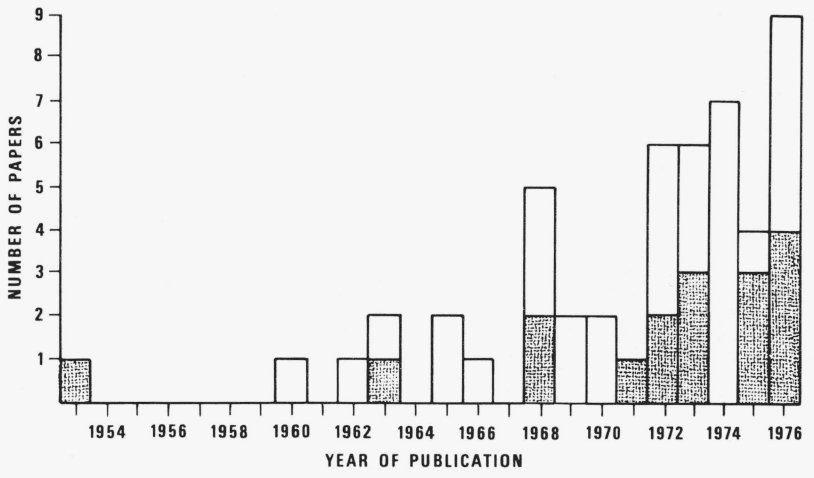

FIGURE 3.1. *Number of papers that used the replication-within-category concept of experiment design.*
(Shown as a portion of all papers in our study.)

As the next topic, we chose the questions of how and why the number and the type of problems were chosen. Somewhat surprisingly, there were twelve papers (out of fifty) that provided a form of justification. These justifications ranged from lack of availability of any other problems to statistically based arguments involving sampling theory.

As a final item, we explored the question of whether preprocessing of the test problems (e.g., data sorting or scaling, reorganizing the coefficient matrix) had been performed prior to optimization. Of the fifty papers we surveyed, four indicated that preprocessing had occurred. Two other papers addressed this topic, but did not explicitly deal with the manner of preprocessing. Since preprocessing can have an influential effect on experimental results, we believe that future articles should be more explicit in dealing with this issue.

3.2 Computer Environment

Next, we turn to the computer environment and the degree to which it was described. Generally, a minimum of information about the computer was provided in the papers. For instance, although the word-length of the computer affects the accuracy of the results and has an impact on the total processing time, only seven of the fifty papers provided the word-length (available precision) of the computer. An area of recent interest, the multiprogrammability of the machines was mentioned in five papers. The operating system was identified in four of the fifty papers, the machine size (core storage) was reported in four cases, and the compiler was left anonymous in all but seven. Each of these, though unimportant in some cases, may affect code performance; their omission certainly limits the replicability of the experiment. Incredibly, as many as seven papers did not name the computer used! We can note that although papers meeting these criteria for computer-environment informativeness were few, they appeared mainly in the last several years.

The use of standardized timers is an idea proposed by Coleville [B5] as an attempt to improve the comparability of experimental results when codes and problems are run on different machines. The idea is to time a "standard" code (in Coleville's case it was a matrix inversion routine) and use that value to normalize results across computers. Five of the papers we reviewed employed standardized time: two in unconstrained optimization, two in general nonlinear programming, and one in geometric programming. The dates of publication range uniformly from 1968 through 1976.

3.3 Experiment Controls

In considering the experiment controls, we identified those factors that introduce variability in the outcome of computational experiments. Each paper was evaluated with respect to the manner in which the factors were controlled.

The first factor was "computer." We simply wanted to know if the same computer was used for all test runs. The results are as follows:

yes	32	na*	0
no	11	cna**	7

* Not applicable.
** Could not ascertain.

The next factor identified was "compiler." Since the same codes on the same machine will produce different results with different compilers, we asked whether the same complier was used for each code. The results are:

yes	8	na	0
no	8	cna	34

If the same set of test problems is not run on each code, an uncontrolled parameter ("problem set") is introduced which may cause erroneous inferences. It is of less concern when randomly generated problems are used and the results are reported as the mean or the median of a large number of similar problems. The results of asking whether the same problems were used on all codes are:

yes	42	na	1
no	5	cna	2

It would be interesting to know how often the same programmer was used to develop the codes for evaluation. However, in most cases, we were unable to answer this question. The results are as follows:

yes	7	na	0
no	15	cna	28

Using different computer languages can produce varying results. In fact, if machine language is used, one might even say that comparisons with codes containing high level language are inappropriate. The results of asking whether the same language was used for all codes are:

yes	20	na	0
no	6	cna	24

A factor that may introduce variability in results of computational comparisons is the workload of the computer when operating in a multiprogrammable environment. Unfortunately, this topic was addressed all too infrequently. We asked whether an attempt was made to run under the same workload:

yes	2	na	1
no	13	cna	34

Note that multiprogramming is a recent phenomenon and is not applicable to the older studies; we could not determine the extent of multiprogramming in most of the more recent studies.

Tolerances (e.g., pivot tolerances and convergence tolerances) were also investigated. It is encouraging to have discovered that almost 25% of the relevant papers addressed this point. The statistics for using identical tolerances for each run are:

yes	10	na	11
no	4	cna	25

Another parameter whose effect is often overlooked is the zero tolerance, i.e., the definition of an ϵ-range for the value zero. Asking whether the zero tolerance was held constant for all codes on all runs, we discovered the following results:

yes	9	na	10
no	1	cna	30

One paper attempted to ascertain the effects of altering the zero tolerance.

Another confounding variable is the use of different starting points within the algorithms. We recognize that in many codes available today, the algorithm for locating a starting point is built-in. However, if codes with different initial-solution algorithms are compared, the evaluation becomes complicated by an additional factor. When we asked whether starting points were the same in each run, we sought answers for all areas of MP, and if the point was not addressed, we made no assumptions. The results follow:

yes	13	na	2
no	1	cna	34

4. Empirical Results

This section deals with the empirical results of the experiments as they were reported in the reference articles. We summarize how these results were presented and used, and point out noteworthy instances of good and bad methodology. Since many of these studies were condensed for publication, especially the articles appearing in refereed journals, and since we did not have access to the original reports, we were unable to evaluate thoroughly whether the data justify the conclusions stated. Thus, we could not critique the experiments with the same degree of understanding as could a conscientious referee. Instead we acted as investigative reporters.

4.1 Performance Measures

The initial topic considered was the measures of performance used in evaluating algorithm and software effectiveness. A summary of the most frequently employed measure, central processing times, is provided in Table 4.1. Sixty-eight percent of the papers (34) used processing times as a performance indicator. Although the majority of these papers (24/34) reported the processing times for individual problems a sizable portion (14/34) used the average or the median processing times as an indicator of performance.

TABLE 4.1. *How processing times were reported.*

	Central Tendency (mean or median)	Standard Deviation	Individual Problems Discussed	Worst Case Analysis
No	36	47	26	48
Yes	14	3	24	2

Note that many experiments used central processing time without regard for input/output, even though the input/output portions may be two to three times as time consuming as the optimization elements (see Himmelblau, [5]). For nineteen of the thirty-four papers, we could not determine whether input/output was included as part of the reported processing times. On the other hand, thirteen papers mentioned that input/output was not included. Only two papers provided the computer times required for input/output processing.

There is a potential source of difficulty with using central processing as the sole performance indicator; there are many other objectives that have a bearing on the usefulness of the technique, for example the relative amount of storage that is required for each code. But the storage requirements were mentioned in only seven of the fifty papers. Three of these seven papers considered the number of non-zero elements which were stored as a criterion for comparison.

An important consideration that is often left aggregated within the measurement of input/output times is that of preprocessing. As mentioned in Section 3.1, preprocessing refers to computations performed prior to the "official" start of a program. These computations may constitute a significant percentage of the total. In the area of networks, for example, the preprocessing (sorting) of arcs by cost coefficients is usually accomplished during the input stage. Unfortunately, we could not ascertain the extent of preprocessing that occurred because the input/output specifications were not reported for these experiments. Not one paper fully described the extent of preprocessing that took place.

The number of function evaluations occurring during program execution is another frequently used measure of performance. Tables 4.2 and 4.3 provide summary information regarding this measure. Nine papers addressed this issue, primarily in the area of unconstrained optimization. Some researchers are reluctant to use function evaluations as a sole measure of performance. Thus, three of the nine papers that counted function evaluations also reported processing time. In four papers, a standard unit of work was defined and used.

The number of iterations to solve a problem is another measure of performance that is closely aligned with the previous one, and is relatively independent of the computer used. See Table 4.4 for a summary of how often this criterion was used.

TABLE 4.2. *Papers that used operation count or number of function evaluations as a performance measure.*

Subclass	Total	Number Using Function Evaluations
Linear Programs	10	1
Networks	5	0
Quadratic Programs	1	0
Integer Programs	10	0
Unconstrained	7	4
Nonlinear Programs	5	1
Knapsack	1	0
Shortest Path	6	1
Nonlinear Equations	2	2
Geometric Programs	3	0

TABLE 4.3. *Methods of reporting function evaluations.*

Method	Number
Central Tendency	4
Standard Deviation	1
By Problem	7
Worst Case	0

TABLE 4.4. *The number of iterations as a measure of performance.*

	Central Tendency	Spread	By Problems	Worst Case
General LP's	3	2	7	0
Integer Programs	0	0	8	0
General Networks	2	0	0	0
Quadratic Programs	1	0	1	0
Unconstrained	0	0	2	0

In earlier works, it was thought that the number of pivots would be an invariant and unbiased measure of performance; a belief that has persevered to the present. Unfortunately, the average time to conduct a pivot may vary considerably. (See Mulvey [7]) for evidence showing a 17-fold variation in average pivot time for a single problem. Thus, conclusions drawn about code performance based on results from the single indicator "number of pivots" are likely to be misleading. The pivot strategy and the scheme for storing the basis have a profound effect, and should be controlled during a computational experiment.

Numerical accuracy is occasionally considered more important than efficiency (as measured by processing time, number of function evaluations, or number of iterations). Many users of mathematical programs prefer an efficient method which is occasionally inaccurate over a relatively inefficient method that provides accurate answers. Design engineers who solve small-scale nonlinear programming problems often express this preference. We should note that methods employing integer arithmetic throughout are not influenced by accuracy considerations, except when computer word size is exceeded, and these methods were excluded.

A variety of definitions of numerical accuracy exists: Himmelblau [5] provides a sample. Since we were primarily concerned with how many articles addressed numerical accuracy, we did not distinguish among these definitions. Table 4.5 indicates the percentage of experiments which treated numerical accuracy, by type of article. In total, numerical accuracy was considered in 14% of the articles. Reference [C5] is particularly noteworthy for pointing out potential difficulties with setting the zero tolerance in the program.

TABLE 4.5. *Numerical accuracy as a measure of performance.*

	Accuracy as Related to CPU or Iteration	Accuracy as Related to Convergence	Accuracy as a Function of 0 Tolerances	Formal Error Analysis
Applicable Refereed Articles	1	1	0	0
Applicable Reports and Theses	2	0	0	0
Applicable Books and Proceedings	0	1	1	0
Theses and Dissertations	1	0	0	0

Robustness was the next measure of performance considered. We defined robustness as the percentage of problems solved by each method as related to the total number of problems attempted. Eighteen percent of the papers considered robustness. Since our definition of robustness depends upon the termination criterion used, whenever a program ends prematurely it is impossible to estimate how long it would have taken to reach the "optimal" solution. A variety of suggestions have been made for resolving this problem (see Gill and Murray, [4]).

In another noteworthy article [A21], robustness was further refined as a measure of performance, and given the name reliability. A formal definition of reliability was provided, and the algorithms were ranked according to this criterion.

It is clearly important to understand why a method failed on particular problems. Three out of fifty papers went into a detailed analysis of this point, whereas twelve out of fifty papers simply counted how many problems could be solved by each code.

Program set-up time for problems and codes was the final measure of performance addressed. Two papers in the area of general nonlinear programming attempted to quantify and measure this criterion for competing techniques.

4.2. Statistical Methods

Statistical methods have been used in analyzing computational experiments. The frequency of use in shown in Figure 4.1. With statistical sampling, inferences can be made about the performance of the techniques for

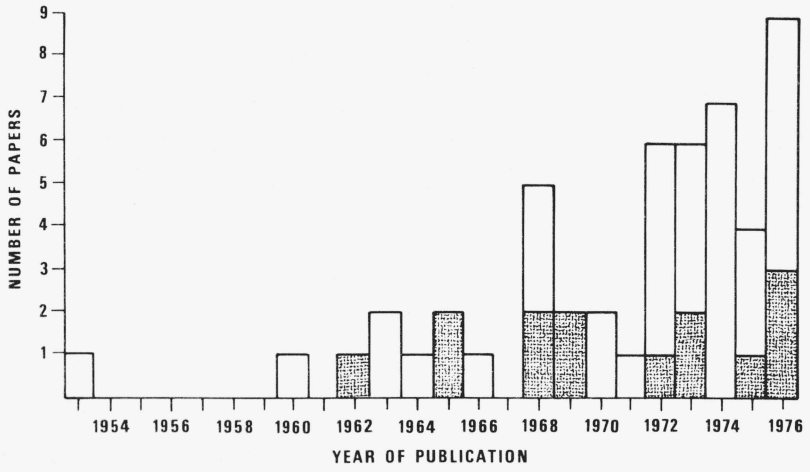

FIGURE 4.1. *Use of statistical methods.*

a population of test problems. Of course, the assumptions that such a "statistical" population exists and that a "random" sampling procedure can be developed must be made. These assumptions might not be generally appropriate. As mentioned earlier, the current lively debate over whether computational experiments should be conducted with carefully selected real-world problems or with randomly generated problems illustrates the lack of a clear resolution for this dilemma. In Table 4.6, we indicate the statistical methods used within the fifty experiments. As shown, regression analysis was the statistical method most frequently employed. It was usually used to estimate a function relating computational difficulty to initial problem parameters.

TABLE 4.6. *Survey of statistical methods employed.*

Category	Hypothesis Tests	Regression Analysis	ANOVA	Factorial Design	Latin Squares	Significance Tests	Non-Parametric Tests	Other
Refereed Papers	0	3	0	1	0	1	0	3
Chapters of Books and Proceedings	1	2	0	0	1	0	0	1
Working Papers	0	1	1	0	0	0	0	1
Theses and Dissertations	0	0	1	1	0	1	1	0
Total	1	6	2	2	1	2	1	5

4.3 Mathematical Checks

Without indication to the contrary, researchers and journal referees generally have faith that a code has "solved" a problem when it terminates. Integer programming is an exception to this rule. As shown in Table 4.7, only 6% of the papers provided proof of optimality, and 34% provided the final objective-function values for the problems solved. Hence, it is difficult to know whether or not optimality was reached. Many articles failed to report the final values of the objective function or to say where these values could be obtained. This omission seriously undermines the conclusions drawn, since replication of the results is impossible. In addition, the data may not be compatible because different codes may terminate at different solution values. The omission of the degree of satisfaction obtained for the Kuhn-Tucker conditions further degrades the results. The paper of Crowder et al. [2] is an attempt to correct this problem by requiring certain minimal standards for publication. These authors believe that the values of the objective function and the Kuhn-Tucker "residuals" should be included in the published article or in a supplementary unabridged report.

TABLE 4.7. *Data at termination.*

	Proof of Conditions Provided	Objective Function Values
no	47	33
yes	3	17

4.4 Reporting of Empirical Evidence

Empirical evidence was reported primarily in tables of summary data. These talbes take many forms, of course. Table 4.8 summarizes the frequencies of the most popular methods. There seems to be a trend to include more technical information in the published articles.

TABLE 4.8. *Presentation of empirical evidence.*

	Summary Statistics	By Problem	Value of Objective Function vs. Iteration Count	Density of Arrays as Function of Iteration	Other
Total	20	34	6	0	11

Graphics are another means for reporting empirical evidence. Again, a variety of forms are included. Table 4.9 shows the incidence of various graphs. In five instances, there was very little empirical evidence provided; a general discussion of coding enhancements replaced the usual evaluation discussion.

TABLE 4.9. *Types of graphs used.*

	Value of O.F. vs. Time or Iteration	Size of Problem vs. Time or Iteration	# Function Evaluations vs. Time	Problem Topologies
Total	4	5	4	3

4.5. Interpretation of Results

In this section we analyze how and why the researchers used the empirical evidence provided in their papers.

Understanding and predicting code performance were fundamental concerns, and twenty-four papers stated or implied this objective for their experiment. In seven papers, the domain of applicability for each code was established by means of the computational experiments. Six papers indicated possible improvements in the software, whereas forty-two papers used the empirical evidence to demonstrate the relative rankings of the techniques tested; however, the results were compared with previous work in only eleven cases. Generally the ranking scheme was not formally defined, but in six cases a weighted average was employed.

The most glaring weakness, in our opinion, was the lack of concise statements indicating the purpose of the computational experiment and the limitations of the study. Only 30% of the papers provided this. Five papers gave a reference to a more comprehensive report of the experiment.

5. Suggestions for Future Work

The evidence shows that the development of a methodology for testing and evaluating mathematical software is at any early stage, and the field of mathematical programming is no exception. The recent creation of a journal (ACM's Transactions on Mathematical Software) to disseminate information about mathematical software indicates an increasing interest in this subject. Groups have been formed—one in statistics (The Committee on Evaluation of Statistical Program Packages (see Francis [3])) and one in *Mathematical Programming* (COAL, [6])—concerning themselves with computational aspects of their respective disciplines. Other such groups will no doubt arise.

This paper shows that a consensus for conducting computational experiments has not been reached, although patterns can be detected within certain areas of mathematical programs. On the whole, the more recent experiments appear improved in methodology only slightly over their predecessors. The 1953 paper [A1] of Hoffman was one of the most thorough evaluations.

We believe that fundamental research in the area of computer-algorithm performance is long overdue. We further believe that developmental work in the area of computational evaluation is greatly needed. There are several obvious needs: (1) compact, portable problem generators which build test problems possessing controllable realistic structure; (2) a modeling language for generating problems that takes into account the inherent structure of a class of problems; (3) ideas for reducing the computational burden of testing large-scale examples; (4) suitable performance indicators; and (5) an aware group of researchers. This paper is concerned with point (5), but the others are also important and should be subjects for future work.

6. References

[1] Anderson, V. L., and McLean, R. A., Design of Experiments: A Realistic Approach (M. Dekker, Inc., New York, 1974).

[2] Crowder, Harlan P., Dembo, R. S., and Mulvey, J. M., Guidelines for reporting computational experiments in mathematical programming, Working Paper No. HBS 77-8 (Graduate School of Business, Harvard University, 1976), submitted to Mathematical Programming.

[3] Francis, Ivor, Heiberger, R. M., and Velleman, P. F., Criteria and considerations in the evaluation of statistical program packages, The American Statistician, **29** (February 1, 1975).

[4] Gill, P. E., and Murray, W., Numerical Methods for Constrained Optimization (Academic Press, New York, 1974).

[5] Himmelblau, D. M., Applied Nonlinear Programming (McGraw-Hill, New York, 1972).
[6] Mathematical Programming, **9**, 131–135 (August 1975).
[7] Mulvey, J. M., Pivot strategies for primal-simplex network codes (to appear in the Journal of the ACM, 1978).

Appendix A: The Papers Reviewed

The references in this bibliography are grouped into four sections according to the form of publication: (A) papers appearing in refereed journals; (B) papers appearing in the proceedings of conferences and chapters from books; (C) technical reports;* and (D) theses and dissertations. Within each section the references are generally ranked according to the date of publication.

Section A

[1] Hoffman, A., Mannos, M., Sokolowsky, D., Wiegmann, N., Computational experience in solving linear programs, J. SIAM, **1**, 17–33 (1953).
[2] Dickson, J., and Frederick, F., A decision rule for improved efficiency in solving linear programming problems with the simplex algorithm, CACM, **3**, 509–512 (Sept. 1960).
[3] Mueller, R., and Cooper, L., A comparison of the primal-simplex and primal-dual algorithms for linear programming, CACM, **8**, No. 11, 682–686 (Nov. 1965).
[4] Srinivasan, V., An investigation of some computational aspects of integer programming, JACM, **12**, No. 4, 525–535 (Oct. 1965).
[5] Box, M. J., A comparison of several current optimization methods, and the use of transformations in constrained problems, Computer Journal, **9**, 67–77 (1966).
[6] Gue, R., Liggett, J., and Cain, K., Analysis of algorithms for the zero-one programming problem, CACM, **11**, No. 12, 837–844 (Dec. 1968).
[7] Bennett, J. M., Cookley, P. C., and Edwards, J., The performance of an integer programming algorithm with test examples, The Australian Computer Journal, **1**, No. 3, 182–185 (Nov. 1968).
[8] Trauth, C., and Woolsey, R., Integer linear programming: A study in computational efficiency, Management Science, **15**, No. 9, 481–493 (May 1969).
[9] Ravindran, A., Computational aspects of Lemke's complementarity algorithm applied to linear programs, Opsearch, **7**, 241–262 (1970).
[10] Bard, Y., Comparison of gradient methods for the solution of nonlinear parameter-estimation problems, J. SIAM Numerical Analysis, **7**, No. 1, 157–186 (March 1970).
[11] Benichou, M., Gauthier, J., Girodet, D., Hentges, G., Ribiere, G., and Vincent, O., Experiments in mixed-integer linear programming, Mathematical Programming, **1**, 76–94 (1971).
[12] Braitsch, R., A computer comparison of four quadratic programming algorithms, Management Science, **15**, No. 11, 631–643 (July 1972).
[13] Srinivasan, V., and Thompson, G., Benefit-cost analysis of coding techniques for the primal transportation algorithm, JACM, **20**, No. 2, 193–213 (April 1973).
[14] Asaadi, J., Computational comparison of some nonlinear programs, Mathematical Programming, **4**, 144–154 (1973).
[15] Brent, R., Some efficient algorithms for solving systems of nonlinear equations, J. SIAM Numerical Analysis, **10**, No. 2, 327–344 (April 1973).
[16] Mitra, G., Investigation of some branch and bound strategies for the solution of mixed integer linear programs, Mathematical Programming, **4**, 155–170 (1973).
[17] Breu, R., and Burdet, C., Branch and bound experiments in zero-one programming, Mathematical Programming Study 2: Approaches to Integer Programming (North-Holland, Amsterdam, Netherlands, 1–50, 1974).
[18] Glover, F., Karney, D., Klingman, D., and Napier, A., A computational study of start procedures, basis change criteria, and solution algorithms for transportation problems, Management Science, **20**, No. 5, 793–813 (Jan. 1974).

* Many of these reports have been submitted for publication. Unknown to us, some may already have appeared; for this oversight we apologize. One paper [C7] was unfortunately placed in this category. Because the National Bureau of Standards' Technical Notes are refereed publications, [C7] should have been included under Section A; this was recognized too late for revision.

[19] Pape, U., Implementation and efficiency of Moore-algorithms for the shortest route problem, Mathematical Programming, **7**, 212–222 (1974).

[20] Barr, R., Glover, F., and Klingman, D., An improved version of the out-of-Kilter method and a comparative study of computer codes, Mathematical Programming, **7**, 60–86 (1974).

[21] Shanno, D., and Phua, K., Effective comparison of unconstrained optimization techniques, Management Science, **22**, No. 3, 321–330 (Nov. 1975).

[22] Sharp, J., and Welling, P., A simulation study of the error produced by approximation in separable concave programming, Mathematical Programming Study 4: Computational Practice in Mathematical Programming (North-Holland, Amsterdam, Netherlands, 133–141, 1975).

[23] Fayard, D., and Plateau, G., Resolution of the zero-one knapsack problem: comparison of methods, Mathematical Programming, **8**, 272–307 (1975).

[24] Golden, B., Shortest path algorithms: A comparison, Operations Research, **24**, No. 6, 1164–1168 (Nov.–Dec. 1976).

[25] Mahendrarajah, A., and Fiala, F., A comparison of three algorithms for linear zero-one programs, ACM-TOMS, **2**, No. 4, 331–334 (Dec. 1976).

[26] Chen, Der-San, and Zionts, S, Comparisons of some algorithms for solving the group theoretic integer programming problem, Operations Research, **24**, No. 6, 1120–1128 (Nov.–Dec. 1976).

Section B

[1] Wolfe, P., and Cutler, L., Experiments in linear programming, in R. Graves and P. Wolfe (eds.), Recent Advances in Mathematical Programming (McGraw-Hill, New York, pp. 177–200, 1963).

[2] Smith, D., and Orchard-Hays, W., Computational efficiency in product form LP codes, in R. Graves and P. Wolfe (eds.), Recent Advances in Mathematical Programming (McGraw-Hill, New York, pp. 211–218, 1963).

[3] Kuhn, H., and Quandt, R., An experimental study of the simplex method, Symposia on Applied Mathematics, **15**, American Mathematical Society, Providence, RI, 107–124 (1962).

[4] Bougoin, M., and Heurgon, E., Study and comparison of algorithms of the shortest path through planned experiments, Project Planning by Network Analysis, North-Holland, Amsterdam, Netherlands, 106–118 (1969).

[5] Colville, A., A comparative study of nonlinear programming codes, IBM Scientific Center Report No. 320-2949, 487–500 (June 1968).

[6] Abadie, J., and Guigou, J., Numerical experiments with the GRG method, in F. Lootsma, Numerical Methods for Nonlinear Optimization (Academic Press, New York, pp. 529–536, 1972).

[7] Himmelblau, D., A uniform evaluation of unconstrained optimization techniques, in F. Lootsma, Numerical Methods for Nonlinear Optimization (Academic Press, New York, pp. 69–97, 1972).

[8] Parkinson, J., and Hutchinson, D., An investigation into the efficiency of variants on the simplex method, in F. Lootsma, Numerical Methods for Nonlinear Optimization (Academic Press, New York, pp. 115–135, 1972).

[9] Dembo, R., and Mulvey, J., On the analysis and comparison of mathematical programming algorithms and software, Technical Report No. HBS 76-19, Harvard Business School, Boston, MA, 1976 (to appear in the Proceedings of the SIGMAP-NBS Bicentennial Conference on Mathematical Programming, Nov. 1976).

[10] Schlick, F., and Nazareth, L., A performance profile study of three unconstrained optimization routines (to appear in the Proceedings of the SIGMAP-NBS Bicentennial Conference on Mathematical Programming, Nov. 1976).

[11] Sargent, R., and Sebastian, D., Numerical experience with algorithms for unconstrained minimization, in F. Lootsma, Numerical Methods for Nonlinear Optimization (Academic Press, New York, pp. 45–68, 1972).

Section C

[1] Yakimovsky, Y., Experiments on the comparative efficiency of various variants of the standard simplex algorithm, Technical Report No. 72-17 (Stanford University, Stanford, CA, Aug. 1972).

[2] Gochet, W., Loute, E., and Solow, D., Comparative computer results of three algorithms for solving prototype geometric programming problems, CORE Discussion Paper No. 7409 (University of Louvain, Brussels, Belgium, April 1974).

[3] McCoy, P., and Tomlin, J., Some experiments on the accuracy of three methods of updating the inverse in the simplex method, Technical Report SOL 74-21 (Stanford University, Stanford, CA, Dec. 1974).

[4] Hitchner, L., A comparative investigation of the computational efficiency of shortest path algorithms, Technical Report No. ORC 68-25 (University of California, Berkeley, CA, Nov. 1968).

[5] Brocklehurst, E., and Dennis, K., A comparison of six algorithms for dense linear programs, NPL Report NAC 51 (National Physical Laboratory, Teddington, Middlesex, England, June 1974).

[6] Rijckaert, M., and Martens, X., A comparison of generalized geometric programming algorithms, Report CE-RM-7503 (University of Louvain, Brussels, Belgium, 1975).

[7] Gilsinn, J., and Witzgall, C., A performance comparison of labeling algorithms for calculating shortest path trees, Technical Note 772 (National Bureau of Standards, Washington, DC, May 1973).

[8] Shier, D., A computational study of a class of K shortest path algorithms, Working Paper (National Bureau of Standards, Washington, DC, April 1976).

[9] Lin, B., and Rardin, R., Controlled experimental design for comparison of integer programming algorithms, Report No. J-76-25 (Georgia Institute of Technology, Atlanta, GA, Sept. 1976).

[10] Hillstrom, K., A simulation test approach to the evaluation and comparison of unconstrained nonlinear optimization algorithms, Report No. ANL-76-20 (Argonne National Laboratory, Argonne, IL, Feb. 1976).

[11] Dembo, R., The current state-of-the-art of algorithms and computer software for geometric programming, Working Paper No. 88 (Yale University, New Haven, CN, Nov. 1976).

Section D

[1] Zanakis, S., Experimental comparison of nonlinear programming algorithms in deriving maximum likelihood estimates for the three-parameter Weibull distribution, Ph.D. Dissertation (Pennsylvania State University, University Park, PA, 1973). (Available from University Microfilms.)

[2] Lee, Shao-ju, An experimental study of the transportation algorithm, Master's Thesis (University of California, Los Angeles, CA, 1968).

Appendix B: The Questionnaire

2.2.1.2 By mathematical form only? - - - - - [36]
2.2.2 Application areas discussed? - - - - - [37]
2.3.3 Any discussion of differences in problem class across technique? - - - - - [38]

2.3 Software

2.3.1 Was method of presentation same for all codes? - - - - - [45]
2.3.2 Methods used:
2.3.2.1 Referenced by name only? - - - - - [46]
2.3.2.2 Reference to other descriptions in literature? - - - - - [47]
2.3.2.3 Extensive discussion? - - - - - [48]
2.3.2.4 Brief discussion? - - - - - [49]
2.3.2.5 Extensive flowchart? - - - - - [50]
2.3.2.6 Brief flowchart? - - - - - [51]
2.3.2.7 Were listings included? - - - - - [52]
2.3.3 Were data structures discussed? - - - - - [55]
2.3.4 Were storage requirements given?
2.3.4.1 For code? - - - - - [56]
2.3.4.2 For working storage? - - - - - [57]
2.3.4.3 For offline storage? - - - - - [58]
2.3.4.4 Total only? - - - - - [59]
2.3.5 Were any codes commercially produced? - - - - - [60]
2.3.6 Were all codes commercially produced? - - - - - [61]
2.3.7 Was programming language unmentioned for any of the codes? - - - - - [62]
2.3.8 What languages were used?
2.3.8.1 FORTRAN? - - - - - [63]
2.3.8.2 ALGOL? - - - - - [64]
2.3.8.3 PL/1 - - - - - [65]
2.3.8.4 Machine level? - - - - - [66]
2.3.8.5 Any codes with mixtures? - - - - - [67]
2.3.9 Any reference to more documentation of codes? - - - - - [68]
2.3.10 Were codes liberally annotated with comments? - - - - - [69]
2.3.11 Was code portability mentioned? - - - - - [70]
2.3.12 Was code portability tested? - - - - - [71]
2.3.13 Was ease of use of codes discussed? - - - - - [72]
2.3.14 Was availability of codes mentioned? - - - - - [73]
2.3.14.1 Were any codes proprietary? - - - - - [74]
2.3.15 Was output of codes mentioned? - - - - - [75]
2.3.15.1 Sample output includes? - - - - - [76]
2.3.16 Any discussion of why tolerances were chosen? - - - - - [77]

3. Experimental Design - - - - - [5]

3.1 The Problems

3.1.1 Were randomly generated problems used? - - - - - [20]
3.1.2 Did authors develop their own generators? - - - - - [21]
3.1.3 Was generator availability mentioned? - - - - - [22]
3.1.4 Was method of generation discussed? - - - - - [23]
3.1.5 Was distribution of problem parameters mentioned? - - - - - [24]
3.1.6 Was density under control of the generator? - - - - - [25]
3.1.7 Were the ranges for the parameters given? - - - - - [26]
3.1.8 Was the total number of problems run mentioned explicitly? - - - - - [27]
3.1.9 Total number of problems run:

3.1.9.1 In the range 1–10? -----[28]
3.1.9.2 In the range 11–20? -----[29]
3.1.9.3 In the range 21–50? -----[30]
3.1.9.4 In the range 51–100? -----[31]
3.1.9.5 Greater than 100? -----[32]
3.1.10 Was there a breakdown by size or type with replications? -----[33]
3.1.11 Number of sizes or types:
3.1.11.1 In the range 1–10? -----[34]
3.1.11.2 In the range 11–20? -----[35]
3.1.11.3 In the range 21–50? -----[36]
3.1.11.4 In the range 51–100? -----[37]
3.1.11.5 Greater than 100? -----[38]
3.1.12 Number of replications:
3.1.12.1 In the range 3–5? -----[39]
3.1.12.2 In the range 5–10? -----[40]
3.1.12.3 Greater than 10? -----[41]
3.1.13 Was there any discussion of why the number of problems was chosen? -----[42]
3.1.14 Were real world (RW) problems used? -----[50]
3.1.15 Was there a discussion of origins of RW problems? -----[51]
3.1.16 Was there a description of each RW problem? -----[52]
3.1.16.1 Was size given for each? -----[53]
3.1.16.2 Was density mentioned for each? -----[54]
3.1.16.3 Was the range of parameters given? -----[55]
3.1.17 Was there a reference to other descriptions? -----[60]
3.1.18 Were any of the RW problems specially tailored? -----[61]
3.1.19 Were the RW problems mentioned to be available? -----[62]
3.1.20 Number of RW problems used:
3.1.20.1 In the range 1–10? -----[63]
3.1.20.2 In the range 11–20? -----[64]
3.1.20.3 In the range 21–50? -----[65]
3.1.20.4 In the range 51–100? -----[66]
3.1.20.5 Greater than 100? -----[67]
3.2.21 Was any preprocessing performed on the problems? -----[68]
-----[1]
-----[3]
-----[5]

3.2 The Computer Environment

3.2.1 Characteristics of the machine:
3.2.1.1 Was the name listed? -----[20]
3.2.1.2 Was core storage available reported? -----[21]
3.2.1.3 Was the operating system mentioned? -----[22]
3.2.1.4 Was the compiler named? -----[23]
3.2.1.5 Multi-programmability mentioned? -----[24]
3.2.1.6 Was word length given? -----[25]
3.2.2 Was the run time of day considered? -----[26]
3.2.3 Were standardized times used? -----[27]

3.3 Experiment Controls

3.3.1 Were the goals of the experiment clearly defined? -----[40]
3.3.2 Was same computer used for all runs? -----[41]

3.3.3 Was same compiler used for all codes? -----[42]
3.3.4 Were the same problems used for all codes? -----[43]
3.3.5 Was same language used for all codes? -----[44]
3.3.6 Were codes programmed by the same person? -----[45]
3.3.7 Was an attempt made to run under same workload? -----[46]
3.3.8 Were same tolerances used for each problem? -----[47]
3.3.9 Was the effect of zero tolerances considered? -----[48]
3.3.10 Were starting points the same for each run? -----[49]
3.3.11 Were same termination criteria used for each run? -----[50]
3.3.12 Termination criteria used:
3.3.12.1 Optimum achieved? -----[51]
3.3.12.2 Closeness to formal bound? -----[52]
3.3.12.3 Lack of reduction in objective function? -----[53]
3.3.12.4 Number of iterations exceeded? -----[54]
3.3.12.5 Computer time exceeded? -----[55]
3.3.13 Were all problems generated on same machine? -----[56]

4. Empirical Results -----[1]
-----[5]

4.1 Measures of Performance

4.1.1 Processing times:
4.1.1.1 Method of Reporting: Central tendency? -----[20]
4.1.1.2 Method of Reporting: Spread? -----[21]
4.1.1.3 Method of Reporting: By problem? -----[22]
4.1.1.4 Method of Reporting: Worst case? -----[23]
4.1.1.5 Were I/O time indicated? -----[24]
4.1.1.6 Was total processing time reported *with no* segregation of I/O? -----[25]
4.1.1.7 Was preprocessing time considered? -----[26]
4.1.2 Functional evaluations (operations count):
4.1.2.1 Any definition? -----[30]
4.1.2.1.1 Standard units of work? -----[31]
4.1.2.1.2 Horner units? -----[32]
4.1.2.1.3 Comparable function evaluators? -----[33]
4.1.2.2 Method of Reporting: Central tendency? -----[34]
4.1.2.3 Method of Reporting: Spread? -----[35]
4.1.2.4 Method of Reporting: By problem? -----[36]
4.1.2.5 Method of Reporting: Worst case? -----[37]
4.1.3 Number of iterations:
4.1.3.1 Method of Reporting: Central tendency? -----[40]
4.1.3.2 Method of Reporting: Spread? -----[41]
4.1.3.3 Method of Reporting: By problem? -----[42]
4.1.3.4 Method of Reporting: Worst case? -----[43]
4.1.4 Was numerical accuracy considered? -----[44]
4.1.4.1 As a function of CPU time or number of iterations? -----[45]
4.1.4.2 As a function of convergence tolerances? -----[46]
4.1.4.3 As a function of zero tolerances? -----[47]
4.1.4.4 Any formal error analysis? -----[48]
4.1.5 Was robustness considered? -----[50]
4.1.5.1 Discussion of why problems were not solved? -----[51]
4.1.5.2 Detailed formal analysis of robustness? -----[52]

4.1.5.3 Number of problems solved (**only**)? -----[53]

4.1.6 Was reliability analyzed? -----[55]

4.1.6.1 Definition given? -----[56]

4.1.6.2 Any empirical evidence shown? -----[57]

4.1.7 Were storage requirements considered? -----[59]

4.1.7.1 Number of non-zero elements? -----[60]

4.1.8 Was programmer set-up time for problems and codes counted? -----[62]

4.2 Any Statistical Methods Used?

4.2.1 Hypothesis testing? -----[65]

4.2.2 Regression analysis? -----[66]

4.2.3 Analysis of variance? -----[67]

4.2.4 Full factorial design? -----[68]

4.2.5 Latin squares? -----[69]

4.2.6 Significance testing? -----[70]

4.2.7 Non-parametric sign test? -----[71]

4.2.8 Other -----[72]

4.3 Mathematical Checks

4.3.1 Were the K-T conditions verified? -----[74]

4.3.2 Were the objective function values provided? -----[75]

4.4 Methods of Presenting Empirical Evidence

4.4.1 Tables

4.4.1.1 Summary statistics? -----[10]

4.4.1.2 By problem? -----[11]

4.4.1.3 Value of objective function versus iteration count? -----[12]

4.4.1.4 Density of arrays as a function of iteration count? -----[13]

4.4.1.5 Other -----[14]

4.4.2 Graphics

4.4.2.1 Value of the objective function vs. time or iteration? -----[16]

4.4.2.2. Size of problem vs. time or iteration? -----[17]

4.4.2.3 Function evaluation vs. time or iteration? -----[18]

4.4.2.4 Problem topologies shown? -----[19]

4.4.3 Discussion **only**

4.5 Interpretation of Results

4.5.1 Understanding and predicting code performance

4.5.1.1 As a function of problem parameters? -----[25]

4.5.1.2 Matching algorithms (codes) with types of problems? -----[26]

4.5.2 Determining the domain of applicability for each code? -----[27]

4.5.3 Indicating possible improvements in codes? -----[28]

4.5.4 Discussing which method works best? (Evaluation) -----[29]

4.5.4.1 Were these results compared with other previous experiments? -----[35]

4.5.4.2 Types of rating schemes:

4.5.4.2.1 Domination? -----[37]

4.5.4.2.2 Weighted averaging? -----[38]

4.5.4.2.3 Totally arbitrary? -----[39]

4.5.4.2.4 Statistically superior? -----[40]

4.5.5 Extrapolating to wider classes of problems? -----[45]

4.5.5.1 Formal (mathematical)? - - - - - [46]
4.5.5.2 Informal (speculation)? - - - - - [47]
(No basis)

4.6 **Were the limitations of the study indicated?** - - - - - [50]
4.7 **Was there a reference to a more comprehensive report of this experiment?** - - - - - [55]
4.8 **Discussion of whether the goals of the experiment were reached?** - - - - - [60]