CONF-770340--1

MATHEMATICAL SOFTWARE PRODUCTION

W. R. Cowell and L. D. Fosdick

Prepared for

Symposium on Mathematical Software
The Mathematics Research Center
University of Wisconsin
Madison, Wisconsin
March 28-30, 1977

U of C-AUA-USERDA

ARGONNE NATIONAL LABORATORY, ARGONNE, ILLINOIS

operated under contract W-31-109-Eng-38 for the
U. S. ENERGY RESEARCH AND DEVELOPMENT ADMINISTRATION

DISCLAIMER

The facilities of Argonne National Laboratory are owned by the United States Government. Under the terms of a contract (W-31-109-Eng-38) between the U. S. Energy Research and Development Administration, Argonne Universities Association and The University of Chicago, the University employs the staff and operates the Laboratory in accordance with policies and programs formulated, approved and reviewed by the Association.

MEMBERS OF ARGONNE UNIVERSITIES ASSOCIATION

The University of Arizona
Carnegie-Mellon University
Case Western Reserve University
The University of Chicago
University of Cincinnati
Illinois Institute of Technology
University of Illinois
Indiana University
Iowa State University
The University of Iowa
Kansas State University
The University of Kansas
Loyola University
Marquette University
Michigan State University
The University of Michigan
University of Minnesota
University of Missouri
Northwestern University
University of Notre Dame
The Ohio State University
Ohio University
The Pennsylvania State University
Purdue University
Saint Louis University
Southern Illinois University
The University of Texas at Austin
Washington University
Wayne State University
The University of Wisconsin

Begin typing here. You may type over these words. Type squarely.

Mathematical Software Survey

by

W. R. Cowell and C. L. Lawson

Type within Solid Blue Lines

ABSTRACT

Locally constructed collections of mathematical routines are gradually being replaced by mathematical software that has been produced for broad dissemination and use. The process of producing such software begins with algorithmic analysis, and proceeds through software construction and documentation, to extensive testing, and finally to distribution and support of the software products. These are demanding and costly activities which require such a range of skills that they are carried out in collaborative projects. The costs and effort are justified by the utility of high quality software, the efficiency of producing it for general distribution, and the benefits of providing a conduit from research to applications.

In this paper we first review certain of the early developments in the field of mathematical software. Then we examine the technical problems that distinguish software production as an intellectual activity, problems whose descriptions also serve to characterize ideal mathematical software. Next we sketch three mathematical software projects with attention to their emphasis, accomplishments, organization, and costs. Finally, we comment on possible future directions for mathematical software production, as extrapolations of the present involvement of universities, government laboratories, and private industry.

I. Introduction

The term "mathematical software" refers to computer programs which perform the basic mathematical computations of science and engineering. As with most broad concepts, an understanding of what mathematical software is may be best inferred from reading the subject matter, for instance the proceedings of the Purdue Mathematical Software Symposia [1, 2] and the contents of the ACM Transactions on Mathematical Software (TOMS). It will be apparent that programs to approximate functions, solve equations, analyze experimental data, etc., are included while compilers, assemblers, and operating systems are not, although many of the practical problems of producing and using mathematical software stem directly from such "systems" software.

Before about 1970, most mathematical software was written by users or individuals closely associated with users and was available to workers in a particular group or at a particular installation. Early efforts to develop mathematical software for a wider audience (e.g., [3, 4, 5]) led to increased interest in the production process and stimulated several major projects which have matured during the past half-dozen years. Consequently, mathematical software intended for general distribution has begun to replace "home-made" routines while libraries of mathematical routines are being accepted as software products in the same sense as compilers and operating systems. In Section V we shall comment on the stability of this trend.

Mathematical software production is not a simple extension of do-it-yourself programming. Programs intended for public distribution must perform as advertised across a broad range of input data, compilers, operating systems, and hardware characteristics. The nature of their usage must be clearly specified in functional terms and they must detect and recover from (or at least report) anomalous situations. Production of software of such quality requires detailed analysis and planning, extensive testing, and comprehensive documenting. These, together with distribution and maintenance of the software products, are demanding and costly activities.

From an accounting standpoint, the costs of mathematical software production are justified in terms of wide distribution which results in a saving of expensive effort. However, there are more profound questions of science resource management that help provide meaningful justification of the costs. As computer power has increased, so have both the complexity of the computing environment and the scope of problems undertaken by users. Now expert knowledge is required for effective utilization of present computing power. We can no longer expect that chemists, mechanical engineers, and physicists will have enough special knowledge about computing to develop the best algorithms for their own use. Roughly fashioned software will sometimes do the job but in many other cases (e.g., (1) the success of singular value decomposition for dealing with difficult problems in data analysis; (2) the development of ordinary differential equations algorithms for stiff systems), expertly crafted algorithms were the difference between solving a problem or not solving it. There is no obvious end to the growth of computing power and hence to the need for expert attention to the expanding problem set. This demand for expertise has inspired advances in algorithm construction and analysis whose embodiment in widely-distributed software is one of the fruits of mathematical software production.

We must also view the production of mathematical software from the standpoint of the mathematicians and computer scientists who construct algorithms in the course of their research. Software provides the means whereby algorithms are utilized and thus deserves the attention that a responsible scientist gives to the use of his creations and discoveries. In the case of the proof of a mathematical theorem or the discovery of an elementary particle, the simple reporting of the information is generally a fulfillment of this responsibility. But new ideas in algorithms are intimately associated with machines and the relationship is so complex that software is one of the means of communi-

cating ideas. From this standpoint, mathematical software production bridges the gap between algorithmic research and effective machine implementations.

We hope that readers will recognize how many of these problems of mathematical software production are intellectually challenging and worthy of the best efforts of talented individuals. To cultivate such interest we shall first review selected early developments in mathematical software, then examine current activities, and finally offer a few predictions about the future.

II. The Evolution of Mathematical Software Production

Criteria for good software have changed with experience and technology. Broadly speaking, we have always insisted that good software be accurate and efficient, but over the years the meanings of these terms have changed and a more explicit standard for good software is still evolving. Looking back several machine generations to the late 1940's and early 1950's, the computers were much slower and had very small memories. High level languages could not be supported on such machines, so programming was done at a very low level, not far from machine language. The small memories forced programmers to employ various tricks permitting multiple use of memory cells.

Good programs generally employed such tricks since they allowed more powerful programs to fit in the limited memory. Indeed it was amazing how much computing power could be packed into a small amount of memory. For example, the assembly program written for the ILLIAC by David Wheeler occupied just 25 words of memory. It was primitive by current standards but it did permit such things as decimal addresses, decimal constants, relative and absolute addressing, and block loading of memory.

The tricks employed to cope with the small memories resulted in code which was difficult to understand and which sometimes imposed strange constraints on the user. For example, in the description of a Runge-Kutta subroutine, G1, for the EDSAC library [6], the following comment is found:

Begin typewriter Your ... ...

> "...G1 should be placed in the upper half of the store to obtain maximum accuracy (the ideal position is 386 onwards). This is because one of the orders forms part of a constant which thus depends slightly on the location of the routine. In normal use the effect is quite negligible..."
>
> Another example from the same source is this comment:
>
> "...The conversion of decimal fractions is slightly simpler if the least significant digit is read first and subroutines R5 and R7 are designed in this way. The number tape can, however, be punched in the ordinary way with the most significant digit first and reversed during the process of copying onto the final tape..."

With our present notions of good programs we would exclude programs with these properties. However, at the time, these programs were among the best and they exerted a strong influence on program library development because of the way they were injected into the bloodstream of scientific computing. First, the principal contributors to the EDSAC library, Wilkes, Wheeler, and Gill, wrote the unique book [6], quoted above, the first in which a subroutine library was published. It was widely read and the authors visited the small number of computing laboratories of that time and contributed their ideas. In visiting the University of Illinois for extended periods, David Wheeler and Stanley Gill wrote a good portion of the ILLIAC library. The order code of the ILLIAC was copied on other machines: SILLIAC (University of Sidney), CYCLONE (Iowa State University), MISTIC (Michigan State University), ORDVAC* (Aberdeen Proving Ground), MUSASINO I (Nippon Telegraph and Telephone Public Corp.). Through this process, the ILLIAC library was transported to these other machines and became the basis for their program libraries. Here we see the first instance of the distribution of software as a consequence of the existence of several copies of a machine.

54
55
56
57
58
59

*This machine was completed slightly earlier than the ILLIAC.

The days of the one thousand word memory soon disappeared but a residue of tricky programming remained. It is finally becoming widely recognized [7] that clarity of programming style is a criterion for good software. Programming languages, made practical by larger memories and faster machines, permitted programs to be written in a fashion more clearly understood by humans. Also, these languages appeared for a time to make machine independent programming possible, whetting appetites for the wide distribution of programs. In this atmosphere, style and portability emerged as criteria for good software.

Another important factor influencing our notions about software was the enormous growth in the number of computers and the size of the user community. User groups were formed in response to the need for exchange of software and other forms of information, the first and largest being SHARE for IBM computers. For some years a group within SHARE was responsible for a program "library" which was essentially a repository and exchange point for programs whose quality varied so widely that all the programs in the library became suspect. Attempts were made to impose standards and to critique the programs without much success. One fact emerging from this experience is that a collection of high quality software cannot be produced through such a process.

As attitudes toward programming evolved, the publication of algorithms in professional journals reflected the state of the art. The early articles emphasized programming techniques rather than algorithms as illustrated by a 1949 program for solving a Laplace boundary value problem on a rectangle [8]. As indicated by the following excerpt the intent of this article was to introduce UNIVAC programming, rather than to communicate a program to potential users:

> "It is believed that there is wide interest in the question of instructing high-speed electronic computers now under design to perform the sequences of operations pertinent to selected

Begin typing here.

> problems. "This article, submitted by members of the staff of a company engaged in the development and construction of electronic digital computers, is considered to be a useful introduction to the use of the instruction code for the computer therein discussed."

Around this period there were a number of articles dealing with the solution of particular problems on computers, especially problems in the area of linear algebra. (The inversion of a matrix of order 38 on the Aiken relay calculator was reported in 1948 [9]; the computing time was 59½ hours.) In January 1953, Joe Wegstein initiated the publication of a bibliography of coding procedures in MTAC [10]. Later, in February, 1960, through his efforts, the regular publication of algorithms in the *Communications of the ACM* began. As with the example cited earlier, it seems that the main purpose of this was to illustrate and promulgate the use of a new tool: in this case, the programming language Algol. However, by 1966, it was recognized that the algorithms appearing in *CACM* were a valuable resource and they were collected and published separately in *Collected Algorithms of the ACM*. The Algorithms Department of *CACM* was transferred to the new *ACM Transactions on Mathematical Software* in 1975. Other journals, too, *(BIT, Computer Journal, Numerische Mathematik)* began the publication of algorithms.

The programs available from the published literature have been of mixed quality. While the refereeing process these journals have used represents more control on the programs submitted for publication than in the case of SHARE, it has not been sufficient to assure a collection of high quality software. Refereeing is a voluntary activity and it is extremely difficult to find good people willing to devote the time and energy required to thoroughly check the programs submitted. Also, authors have rights and opinions that must be respected; therefore, there are limits to the rules on style that one can impose in this

environment. Certainly, the publication of programs has been an important mechanism for the exchange of algorithmic ideas and programming techniques but, unfortunately, it has been less successful as a mechanism for communicating executable programs. To support this assertion, we remark that from June, 1975 when the algorithm distribution service started in the *Transactions on Mathematical Software*, making published programs available in machine readable form (tape or cards), to December, 1976, only 44 orders have been received for the 18 algorithms published; i.e., less than three requests per algorithm.

Users' expectations have significantly changed over a quarter-century, including attitudes toward the ease of use. A program which is the fastest and most accurate will be ignored even if there are a small number of impediments to its use. The commercial developers of software understand this, but many researchers do not and they will expend great effort to produce a program embodying the very best algorithm only to see the fruit of their labor ignored. For example, consider the matter of consistency among the subroutines in a library. It is common for several subroutines from a library to be used within a program. If the global variables for these subroutines are not defined in a consistent way, then passing information from one program to another becomes error prone and awkward, and users are repelled.

Reliability has always been a criterion for good software but now it is a far more significant issue. Programs are larger and much more complex than they were in the early years, so the problem of determining whether or not they are error free is a problem of major proportions. At the same time computers are being used in very sensitive situations where an error is extremely costly. In the early years, it was sufficient to employ a few debugging tools and run some test cases. Under the present circumstances, the production of reliable software requires a disciplined approach to the design and creation of the software itself, as well as very sophisticated tools to assure the reliability of the product.

Efficiency too has always been a criterion for judging software, but the notion of what constitutes efficiency has changed. In the early years, efficient use of memory was a matter of primary concern as we have already mentioned. Memory is now so large and cheap that its efficient use is of less concern. Naturally, speed of computation is important, but machines have become very fast and so this component of efficiency is less important, relatively speaking, than it once was. The big change in our notion of efficiency is the importance we assign to the human time involved in using a program. If large amounts of human effort are required to use a program, even though it executes twice as fast, say, as any other program for the same task, then that program is inefficient in a very real sense. Similar remarks apply to the maintenance of a program. Thus the early preoccupation with raw speed and minimal use of memory has been replaced with a growing concern for the interrelationship between man and machine.

III. Intellectual Challenges

By about 1970 there was enough experience with software to enable those who had acquired a taste for quality to articulate their requirements. The check list would look something like the following:

(1) Algorithmic foundations. The algorithm employed should be the best available for the intended purpose, as demonstrated by careful analysis.

(2) Style. The program must be written so as to clearly exhibit the logical organization of the computation. System dependencies should be minimal, localized, and well-identified.

(3) Documentation. The documentation must be clear and thorough. It must be organized in a way that permits an occasional user to obtain necessary information easily.

(4) Reliability. The operation of the program must be consistent with its documentation; there must be no surprises in its use. Provision must be made to verify that all conditions on input data are met. Extensive testing must show evidence of satisfactory numerical performance.

In addition, if the software is part of a distributed collection, the requirements include:

(5) Consistency. Documentation and conventions for use of programs in the collection must be consistent.

(6) Maintenance. Responsibility for maintenance must be assumed. This includes responding to users queries, making modifications, extensions, and correcting errors.

Attitudes in the early 1950's tended to discriminate between the "intellectual" task of algorithm formulation and the "clerical" task of coding. The challenges implied by the above requirements for high quality software makes such a division of labor and difference of status an anachronism. The computer science community has come to realize that the brainpower needed to create such software is as great, if somewhat differently oriented, as that which provides the algorithmic foundations. Moreover, there are significant research questions associated with software creation. We suggest several in the remainder of this section.

Consider the issue of good documentation. How does one guarantee the consistency of documentation with the software it purports to describe, given the complexities of a typical program and the fact that it will change from time to time? It seems reasonable to consider some automatic way of assuring this consistency. One avenue of approach might be to use the ideas of stepwise refinement [11]. In particular, let the initial description of the program be the documentation or some part of it; then, by a well-defined process, refine this initial

description into the code itself. Another approach might be to use ideas from the work on proving programs correct. In this approach one might let relevant parts of the documentation play the role of assertions, or conversely, and attempt to prove automatically from the code that these assertions are correct. Another aspect of the documentation problem is the automatic preparation of documentation for different versions of a program corresponding to different computing environments.

Programming style is a matter of considerable interest to programmers these days. As we have already observed, our notions of good style have changed. In the early years, that style was good which used the least amount of memory. (Now we seem to wish to use the smallest number of go to's!) The ultimate objective of the style formalized as "structured programming" is to produce programs which are less likely to contain errors. The guidelines for structured programming are heuristics which people believe programmers should follow to meet this objective. One way this approach could be put on a firmer foundation would be to tie it to the relative ease of automatic error detection. For example, one technique for automatic detection of errors in a program uses data flow analysis [12]. Certain programming practices greatly complicate data flow analysis (e.g., the use of EQUIVALENCE), reducing the error detection capability. It would seem reasonable to recommend that programmers avoid the use of such constructs. Redundancy is another aid to error detection. Languages which permit greater redundancy would, in this sense, encourage a better programming style.

The new machine architectures and microprogrammable machines pose new and challenging problems for the algorithm designer. It is becoming possible to mould a machine to a class of problems through micro-programming. Perhaps it would be fruitful to think of designing algorithms with this additional degree of freedom in mind. Here we see the need for an important, but all too rare, combination of talents in numerical analysis, algorithm design, and computer architecture.

Articles in the area of software reliability have appeared with increasing frequency in recent years and there have been several conferences and a recent book devoted to this subject [13, 14, 15, 16]. We are still a long way from being able to prove the correctness of a sizeable numerical routine in a formal sense. However, it does seem that good progress can be made in the area of informal proofs and, if we are ever to achieve truly reliable software, more serious attention should be given to this subject (e.g., see [17]).

There is a very interesting connection between optimization techniques and error detection in programs that is worth mentioning here. In data flow analysis the flow graph of a program is examined and certain patterns of data flow are recognized [12]. Some patterns are symptomatic of errors; however, if these patterns arise on a path which is not executable then they are of no interest. Thus it becomes important to distinguish the executable paths. One attack on this problem has been described by Clarke [18] where it is shown that the problem reduces to a non-linear programming problem, and frequently to a linear programming problem.

The exploration of such problems may appropriately be labeled computer science research. It has become fashionable to call the application of the knowledge thus acquired "software engineering." We shall see in the remainder of the paper how software engineering links to numerical analysis and to a delivery system to form integrated production projects that are carried out in the real world where economies and politics leave their mark.

IV. Projects to Produce Mathematical Software

Because the people who can contribute the required skills are dispersed geographically and have various primary sources of financial support, mathematical software projects are organized as collaborative endeavors involving individuals in universities, government laboratories, and private enterprises.

Begin typing here. Your pages will be typeset. Type accurately.

The accomplishments of a mathematical software project can be listed under the following three headings:

1. Utility:
   Software provided by the project is readily available to and usable by the scientific and engineering public.
2. Research Exploitation:
   The project is a conduit for the flow of research results in numerical analysis and algorithmics into applications.
3. Software Production Improvement:
   Production tools and techniques have been developed and knowledge about the production process has been acquired by the project.

Every successful mathematical software project we have observed could claim accomplishments under all three headings although each project is characterized by a particular distribution of emphasis. Opinion will vary among individuals on a particular project as to the relative importance of these categories. We believe that all three orientations are necessary to mathematical software production and that the success of a project (assuming adequate funding and physical resources) is largely measured by the extent to which differently motivated individuals have merged their efforts to everyone's satisfaction. These differing perspectives will emerge as we describe three representative mathematical software projects, namely NATS (National Activity to Test Software), NAG (Numerical Algorithms Group), and IMSL (International Mathematical and Statistical Libraries), Inc.

NATS

NATS [19] commenced in early 1971 when grants from the National Science Foundation were awarded to Argonne National Laboratory, The University of Texas at Austin, and Stanford University. The objectives of the project were (1) to assemble, test, certify, disseminate, and support packages of mathematical software for eigensystem computation and function approximation; (2) to explore the methodology,

costs, and resources required to do (1). The second objective was primary and was approached by carrying out the first as a prototypical effort. Therefore, NATS was a study of the means of mathematical software production, more explicitly so than any other project we know. By focusing on two computational areas with such intensity that great care could be taken in code construction, testing, and documentation, NATS attempted to establish benchmarks and guidelines for mathematical software production. By organizing the project as a university-government laboratory collaboration, NATS explored the interfaces and division of responsibilities among several collaborating institutions.

The software products of NATS were called "systematized collections" [20]. There were two, EISPACK [21, 22] and FUNPACK [23] in the areas of eigensystem computation and function approximation respectively. Two releases of each collection have been distributed, the second release, in each case, extending the capabilities of the first. EISPACK is available in six machine versions (IBM 360-370, CDC 6000-7000, Univac 1108-1110, Honeywell 6070, DEC PDP-10, Burroughs 6700) and has been distributed to approximately 450 installations. FUNPACK exists in three machine versions (IBM 360-370, CDC 6000-7000, Univac 1108-1110) and has been distributed to about 165 installations. Thus, a significant computational resource has been produced. It is limited in the number of computational areas it covers but is of very high quality within its scope.

The algorithms realized by the NATS software were the culmination of years of work. The eigensystem algorithms published in Algol [24], are generally acknowledged as representing state-of-the-art methods, supported by penetrating error analysis. Release 2 of EISPACK also contains the algorithm discussed in [25]. In the function approximation area, the algorithms were expressed as running Fortran codes that had been developed for use at Argonne National Laboratory. However, such codes are highly machine-dependent and a significant amount of work

was required to prepare different machine versions, assemble a modularized package for each machine, test and fine-tune the software, and write the documentation required for widely distributed software. As the work progressed, NSF grants to the University of Kentucky and Jet Propulsion Laboratory brought collaboration with those institutions in the function approximation area. By producing EISPACK and FUNPACK the NATS project made analytic and algorithmic research results in these areas of computation readily accessible to scientific users for application in many fields.

An important form of collaboration was field-testing in which some two dozen computer installations at universities and research laboratories, representing the machines for which the package was being developed, cooperated by running tests supplied by the software developers and by making the software locally available for application to "real" problems.

Satisfactory performance at the test sites leads to "certification" of the codes; i.e., issuance of a warranty that the codes have performed satisfactorily in extensive testing and assurance of the continued interest of the developers. Certified NATS software is distributed by the Argonne Code Center.

The importance of coordination from some central point becomes very real when one considers the information flow among the various collaborators: numerical analysts supplying algorithms expressed as code developed on different machines in different locations, field test representatives attempting to implement code on a different machine than used by the developers, numerical analysts and software specialists writing documentation, the enormous clerical task of accurately processing many hundreds of requests for a package, and finally a continuing responsive point of contact for questions about the codes and their performance. These are the operational issues after the project has been organized and commitments made. They are preceded by an exploratory effort to determine the feasibility of a particular package development, involve those

who could contribute, and seek funds to support the activity. Indeed a significant investment is required before a major mathematical software project can begin. Argonne National Laboratory played the central coordinating role for these pre-NATS organizational and operational activities.

The clerical burden on the project stimulated the development of computer based techniques for managing the large volume of source code and documentation for several computers. From this beginning there evolved a system, now called TAMPR [26, 27], which analyzes Fortran source programs, constructs an abstract form of the code which can be manipulated to produce alternate versions (e.g., for different computers, with various subroutine options, in greater or less precision, using real or complex arithmetic) which are reconstituted as running Fortran. This innovative work on the production process was intimately associated with the mathematical software development activities of NATS. It has long range research goals but is applicable in a very practical sense. It is an example of the delicate balance between short-term and long-term needs. If NATS had been under greater pressure to meet production deadlines, TAMPR necessarily would have focused on helping to meet those deadlines, probably to the detriment of the system's generality and its larger computer science implications. On the other hand, an attack on program transformation problems without relevance to real production needs could have led to more abstract formulations of less help in the production process.

The NATS project, having lived out its prototypical life, is now deceased. Its products, EISPACK and FUNPACK, have been well-received and its descendents are alive and well. These include the LINPACK project [28, 29], a collaborative effort among Argonne, the University of Maryland, the University of New Mexico, the University of California at San Diego, and the test sites to produce a systematized collection of linear systems routines; the

MINPACK project [30] aimed, in the long term, at producing a systematized collection of codes to solve non-linear optimization problems and systems of non-linear equations; and the research in program transformation exemplified by TAMPR.

NATS was supported by funds from the National Science Foundation and the Energy Research and Development Administration. We offer the following information (see [31]) to convey a sense of the resources required for such a project.

EISPACK, Release 1

Duration - 34 months.

Total Personnel Effort - Senior Professional Staff (Ph.D. or equivalent): 96 months;
Professional Staff (M.S. or equivalent): 16 months;
Clerical Staff: 14 months.

Cost - $528,000

Size - 34 routines totalling about 6000 source cards for each machine version;
The IBM version includes a control program of about 2500 cards.

EISPACK, Release 2

Duration - 41 months (About 16 months overlap with work on Release 1).

Total Personnel Effort - Senior Professional Staff: 60 months;
Professional Staff: 32 months;
Clerical Staff: 17 months.

Cost - $371,000

Size - 70 routines plus certified drivers totalling about 12,000 source cards for each machine version;
The IBM version includes a control program of about 3500 cards;
Machine-readable documentation requires about 12,000 cards.

NAG

In 1970 a group of British universities, all users of the ICL 1906A, initiated joint action to produce a numerical software library for that machine. Now run from a central office in Oxford, the NAG (Numerical Algorithms Group) project [32] has considerably enlarged its original scope. It aims at the creation of a comprehensive numerical software library that can readily be implemented in virtually any scientific computing environment. The current library is available in Fortran and Algol 60 (an Algol 68 version is under construction) for a number of machines manufactured by Burroughs, CDC, DEC, Honeywell, IBM, ICL, Prime, Siemens, Telefunken, Univac, and Varian. There is widespread use of the library, especially in British universities. NAG maintains more than 100 copies in 8 countries [33].

The NAG library project remains a collaborative endeavor among British universities and government research laboratories, notably the National Physical Laboratory and the Atomic Energy Research Establishment at Harwell. Its activities have been subsidized in part by the government; however, it has become a non-profit corporation which will attempt to achieve financial self-sufficiency by renting its library products and services.

The NAG project emphasizes utility. It was created in response to the need for a product and it retains that orientation. The current release (Mark 5) contains over 300 routines (in each of Fortran and Algol 60). A new Mark of the library is issued approximately once a year.

Software for the NAG library originates with contributors who decide upon the coverage of some area of computation, select the methods, and who write, test, and document the software. The contributors are experts in the computational area under development and are usually from one of the cooperating universities or research laboratories. Their software and documents are subject to validation by other experts who review the material for algorithmic merit and usability. It is thus largely the judgment of the

contributors and validators which determines the way NAG functions as a conduit for algorithmic research.

Validated routines are examined by the NAG central office, using various software aids, for adherence to language standards, formatting, and general software performance. The product of these contribution-validation-examination activities is known as the "contributed library." It is not distributed as certified software in the NATS sense but passes through an implementation phase for each machine range, an activity overseen by a coordinator for that machine.

Accepted implementations are returned to the central office for inclusion in a master library file system which retains a complete history of each piece of software in its various incarnations. Information in these files has proved very valuable in determining programming standards to promote portability [34]. This type of case study and the development of the master library file system [35] are among the specific contributions of NAG to software production tools and techniques.

Perhaps the greatest methodological contribution of NAG has been its organization as a collaborative enterprise. It has become a national effort that brings a great deal of the best analytic and programming talent in Britain to focus. The project now has 22 full time staff in the central office and associated universities. Some 120 people work in part-time and voluntary capacities. During the period June 1, 1970 to May 31, 1976, the total economic cost of the NAG project is estimated to have been £1,025,000 [36]. (£1 ≈ $1.70) The effort during the same period is estimated to have been 152 man-years which, in the United States, would have cost approximately $6,000,000, about three and a half times as much.

Considering its success, is NAG a good model for government-sponsored mathematical software production in North America? We think not, at least it was not in 1970, because the emulation of NAG would have brought North Americans into conflict with the way government funds for mathematical software development had been dispensed on this

side of the Atlantic. Under the rubric of "mathematics research" or "computing support," such funds were dispersed among many groups serving the interests of various agencies and sub-agencies. By 1970 when "mathematical software" assumed an identity, nearly every major computing establishment in North America had a mathematical subroutine library. Effort was duplicated, quality variable, and attempts to transport codes were rarely completely successful; nevertheless, the local codes had the virtue of familiarity and most users would not abandon their local software for anything less than codes of the quality of EISPACK.

Meanwhile, in Europe, first class algorithmic research offered a superb foundation for software development but less work had been done toward developing software libraries for a variety of computers. Both NAG and NATS entered this picture, NAG dedicated to satisfying immediate needs for mathematical software through collaborative action and NATS determined to show the benefits of collaboration in producing very high quality packages. NATS was limited in breadth of coverage; NAG software varied in quality though it was, in general, highly competitive with other software then available. (Later Marks of the NAG library show a steady improvement in quality while NATS-like projects, e.g., LINPACK and MINPACK, have entered other computational areas.)

A government-sponsored NAG-like project in North America would, therefore, have been in conflict with the decentralized way mathematical software had been produced on this continent. To gain acceptability, its product library would have needed to be EISPACK-like in each computational area. Even if algorithmic research were sufficiently mature to permit this, the effort would have been very expensive because (1) it costs more to do this work in North America than it does in Britain, and (2) the cost curve rises very sharply as one approaches EISPACK-like quality. The concentration of funds required to produce such a library would not have been politically

acceptable. The same discussion shows, incidentally, why a NATS-like project would have been unacceptably extravagant in Britain in 1970.

IMSL

IMSL (International Mathematical and Statistical Libraries), Inc. is a for-profit corporation in Houston, Texas which offers to lease a proprietary mathematical software library containing about 400 subroutines to users of Burroughs, CDC, DEC, Honeywell, IBM, Univac, and Xerox computers [37]. The corporate intent is to provide software that realizes the state-of-the science in methods and algorithms. The emphasis at IMSL is on utility: satisfying the needs of customers.

The company was established in 1970 by scientist-managers who had earlier helped develop program libraries for IBM. The founders were aware of the local library syndrome discussed above. Since they were using private capital they were not bound by the decentralized approach of government agencies. However, they were faced with the problem of marketing their product in the climate created by that approach. Moreover, they were not possessed of the capital to produce EISPACK-like codes across a broad spectrum. Their response in this tightly-constrained situation was to produce a library that was comprehensive in mathematics and statistics (thereby covering a broader base than most local libraries), to keep its quality as high as possible by involving expert advisers, to underscore their responsiveness to customer problems, and to encourage trial of the library by keeping the subscription price fairly low.

We believe that IMSL assumed substantial risk and that the success of the venture from a business standpoint is still not assured. At its lowest point, the company showed a net loss of about $350,000 in 1972. It broke into the black in the third quarter of 1976. Techniques that are being developed to moderate problems of portability will significantly enhance the ability of IMSL and other producers of mathematical software to disseminate that

software widely; indeed IMSL has, with support from NSF, carried out a portability study that resulted in a system called the Converter [38] used by the company and available to the public. This improvement of software production techniques is of central interest to the company in breaking down barriers to acceptance of its product. We note that both NATS and NAG have also regarded portability as a crucial issue.

IMSL is far more centralized than NATS or NAG. We underscored the collaborative nature of the latter projects which are coordinated, rather than directed, from a central place. IMSL certainly draws on the expertise of the numerical analysis community through its board of advisers but this partnership is limited to a flow of technical advice on request. The structure of the library, and the tactics and overall strategy of the company are, naturally, the responsibility of the corporate officers who answer to the investors.

We believe that the company has demonstrated its ability to act as a conduit for research results. IMSL's source for methods, algorithms, and software is primarily published material in leading journals as well as doctoral theses and contributions from advisers. The President of IMSL has declared that the company's role is "to quickly move research results on algorithms and software development into programs which operate ... in a scientific environment." [39]

Cost and income figures quoted by IMSL [39] add to our information about the expenses involved in mathematical software development. The company spends approximately $2000 for each code for the first implementation while the cost of moving that software to a new environment is about one-sixth of the original cost. The gross revenue derived per code per year per customer is $3.75. It is expected that this figure will decrease as the library grows while the cost of production will decrease as more sophisticated portability techniques are employed. A gross comparison of these costs with the EISPACK development

costs quoted earlier indicates that the NATS costs were about three times as great as IMSL costs for a given volume of code, further evidence that exceptionally high quality software is exceptionally expensive.

V. Trends in Mathematical Software Production

In this section we shall confine our attention to the North American scene in an attempt to predict future patterns of mathematical software production. The following trends are already apparent and will combine to exert a strong influence:

(1) The demand for good general-purpose mathematical software is increasing among the user public;

(2) Relationships between the quality and cost of mathematical software are becoming better understood by producers and users;

(3) Computer-based production techniques, now under development, show promise of improving quality and decreasing cost.

Users have become more receptive to mathematical software produced for dissemination. EISPACK and FUNPACK are acceptable because they are of very high quality while IMSL and other libraries offer good quality and broad coverage. The responsiveness of the developers of these packages to user problems has helped overcome the resistence to non-local software. This acceptance is still very fragile. Users are uncertain about the highest price they are willing to pay, either in terms of subscription fees or the toleration of imperfections. These are closely related since, as we have already remarked, the cost of production rises sharply as one approaches NATS-like quality. If users do not become disillusioned because of some particularly disappointing software that appears, then there is a good chance that mathematical software production for a mass audience will remain viable long enough for improved production techniques to strengthen the whole enterprise. We believe that users and producers are becoming sharp enough in their evaluation of cost-quality relationships to sustain the present momentum.

Three areas of production technique research show particular promise of providing the tools needed to start an escalation of good mathematical software, cost reduction, and user confidence. These are (a) computer-aided analysis and transformation of source programs, (b) testing methods, and (c) networks.

Example of program analysis systems are DAVE [40], PFORT [41], FACES [42], PET [43]. Examples of program transformation systems are TAMPR [26, 27] and the Program Generator [44]. Analysis systems permit a probe of the structural details of complex programs with identification of anomalous and erroneous constructions. The transformation systems also analyze the structure of a program but for the purpose of transforming it in various ways. For example, programs from various sources may be brought into conformance with a set of formatting standards. Such systems may use a master program or set of instructions to automatically generate software tailored to a particular machine or with variations in program structure. By storing and maintaining only the master programs, developers and distributors have far fewer data to manage in error-prone information processing operations.

Testing methods are being developed that take into account a fundamental distinction between two classes of software [45]. The first class (called "precision-bound" software; for example, numerical eigenanalysis, linear systems, and function approximation) suffers primarily from round-off error due to the finiteness of machine number representations. This type of error is rather well understood theoretically (see [46]) and the testing techniques used in the NATS project appear adequate in principle. The second class ("heuristic-bound" software; for example, the numerical treatment of quadrature, ordinary differential equations, and non-linear optimization) is a victim of errors derived from the manner of simulating analytic constructs, in particular convergence. These errors dominate round-off so that such software requires an approach to testing which permits careful statements to be made about the cost of obtaining reliability at some specified confidence level. A testing

methodology along these lines has been developed for quadrature routines [47]. Further work will be essential to the production of high quality heuristic-bound software, accompanied by meaningful statements about performance.

Computer networks will facilitate interaction among geographically dispersed collaborators [48]. Such networks will also enable users to obtain both information about software applicable to their needs and the software itself. Experimentation along these lines is under way at several ERDA laboratories.

We are optimistic that tools based on this research will facilitate the passage of algorithmic ideas into useful software. However, we do not believe that future mathematical software projects will simply be streamlined versions of earlier efforts. Rather we foresee certain basic changes in institutional roles, the principal one being a shift of government support away from NATS-like efforts in which a government laboratory acts as a mediator between university research and end-users. The effectiveness of such programs has been demonstrated and we have argued for their expansion [31]. However, the required concentration of resources remains politically unacceptable. Our hope now is that present trends will lead to commercially viable production activities which draw heavily on expertise in the research community. We predict that the government will encourage these trends by supporting state-of-the-science studies and algorithmic and systems research, buying the mathematical software products and developing the use of these products in government programs. In this view of the future, the actual production will be carried out through commercial enterprises which maintain a working partnership with universities and research laboratories, often taking the form of interaction with individual scientists.

We must, however, admit a second possible future for, as we have already suggested, the commercialization of mathematical software production is in a somewhat precarious situation and its growth is not assured. After all, the same improved techniques that make the production of

commercial libraries more feasible also make it easier for small groups in universities and research laboratories to do what they have always done - produce mathematical software for their own use and for limited distribution within a specialized community. We rate such a "cottage industry" approach as inferior to "mass production" that strives to be a conduit for the best research results. The real future will probably be a mixture of commercialization and local developments in a proportion impossible to foresee. In any case, the mathematical software production events of the last half-dozen years have raised standards and influenced tastes so that the state of scientific computing is the better for it.

REFERENCES

1. Rice, John R. (Ed.) *Mathematical Software*, Academic Press (1971), 515 pp.

2. *Mathematical Software II - Informal Proceedings of a Conference.* Purdue University, May 29-31, 1974. 324 pp.

3. Battiste, E. L. The production of mathematical software for a mass audience. In *Mathematical Software*, John R. Rice, ed., Academic Press (1971), 121-130.

4. Traub, J. F. High quality portable numerical mathematics software. In *Mathematical Software*, John R. Rice, ed., Academic Press (1971) 131-139.

5. Newbery, A. C. R. The Boeing library and handbook of mathematical routines. In *mathematical software*, John R. Rice, ed., Academic Press (1971) 153-169.

6. Wilkes, Maurice V.; Wheeler, David J.; and Gill, Stanley. *The Preparation of Programs for an Electric Digital Computer*, Addison-Wesley (1951), 167 pp.

7. Kernighan, Brian W. and Plauger, P. J. *The Elements of Programming Style*, McGraw-Hill (1974), 147 pp.

8. Snyder, Francis E. and Livingston, Hubert M. Coding of a Laplace boundary value problem for the UNIVAC. *Math. Tables and Other Aids to Computation.* (now Mathematics of Computation) 3, 25 (Jan. 1949), 341-350.

9. Mitchell, Herbert F., Jr. Inversion of a matrix of order 38. *Math. Tables and Other Aids to Computation* (now Mathematics of Computation) 3, 23 (July 1948), 161-166.

10. Todd, John. Bibliography of coding procedures. *Math. Tables and Other Aids to Computation* (now Mathematics of Computation) 7, 41 (Jan. 1953), 47-48.

11. Wirth, Niklaus. Program development by stepwise refinement. *Comm. ACM* 14, 4 (1971), 221-227.

12. Fosdick, Lloyd D. and Osterweil, Leon J. Data flow analysis in software reliability. *ACM Comp. Surv.* 8, 3 (September 1976), 305-330.

13. *Proceedings IEEE Symposium on Computer Software Reliability,* New York City, April 30-May 2, 1973.

14. *Proceedings International Conference on Reliable Software,* Los Angeles, Ca., April 21-23, 1975. 567 pp. IEEE Cat. No. 75CHO940-7CSR.

15. Hetzel, William C. (Ed.) *Program Test Methods,* Prentice-Hall (1973), 311 pp.

16. Myers, Glenford J. *Software Reliability: Principles & Practices,* Wiley (1976), 360 pp.

17. Hull T. E.; Enright, W. H.; Sedgwick, A. E. The correctness of numerical algorithms. *Proc. ACM Conference on Proving Assertions About Programs,* New Mexico State University, Las Cruces, N. M., January 1972, pp. 66-73.

18. Clarke, Lori. A system to generate test data and symbolically execute programs. *IEEE Trans. on Software Engineering* SE-2, 3 (Sept. 1976), 215-222.

19. Boyle, J. M.; Cody, W. J.; Cowell, W. R.; Garbow, B. S.; Ikebe, Y.; Moler, C. B.; and Smith, B. T. NATS, A collaborative effort to certify and disseminate mathematical software, *Proceedings 1972 National ACM Conference,* 630-635.

20. Smith, B. T.; Boyle, J. M.; Cody, W. J. The NATS approach to quality software. In *Software for Numerical Mathematics,* D. J. Evans, ed., Academic Press (1974), 393-405.

21. Garbow, B. S. EISPACK - A package of matrix eigensystem routines. *Computer Physics Communications* 7, 4 (April 1974), 179-184.

22. Smith, B. T.; Boyle, J. M.; Dongarra, J. J.; Garbow, B. S.; Ikebe, Y.; Klema, V. C.; Moler, C. B. *Matrix Eigensystem Routines - EISPACK Guide*, Lecture Notes in Computer Science, 6, 2nd Edition. Springer-Verlag (1976).

23. Cody, W. J. The FUNPACK package of special function subroutines. *ACM Trans. on Math. Soft.* 1, 1 (March, 1975), 13-25.

24. Wilkinson, J. H. and Reinsch, C. *Handbook for Automatic Computation, Volume II, Linear Algebra*, Part 2. Springer-Verlag (1971).

25. Moler, C. B., and Stewart, G. W. An algorithm for generalized matrix eigenvalue problems. *SIAM Journ. of Numer. Anal.* 10, 2 (April, 1973) 241-256.

26. Boyle, J. M. and Dritz, K. W. An automated programming system to facilitate the development of quality software. In *Information Processing 74*, North Holland Pub. Co. (1974), pp. 542-546.

27. Dritz, Kenneth W. Multiple program realizations using the TAMPR system. In *Proceedings of the Workshop on Portability of Numerical Software*, To appear.

28. The LINPACK Prospectus and Working Notes, Applied Mathematics Division, Argonne National Laboratory, Argonne, Ill. 60439.

29. Stewart, G. W. Research, development, and LINPACK. These Proceedings.

30. Brown, K.; Minkoff, M.; Hillstrom, K.; Nazareth, L.; Pool, J.; and Smith, B. Progress in the development of a modularized package of algorithms for optimization problems. In *optimization in Action*, L. C. W. Dixon, ed., Academic Press (1976), pp. 185-211.

31. Cowell, Wayne R. and Fosdick, Lloyd D. A program for development of high quality mathematical software, Report #CU-CS-079-75 (Sept. 1975), Department of Computer Science, University of Colorado, Boulder, Colorado, 80302.

32. Ford, B. and Sayers, D. K. Developing a single numerical algorithms library for different machine ranges. *ACM Trans. on Math. Soft.* 2, 2 (June, 1976) 115-131.

33. Annual Report of the Numerical Algorithms Group: 1 June, 1975 to 31 May, 1976, NAG Central Office, 13 Banbury Road, Oxford OX2 6NN.

34. Bentley, J. and Ford B. On the enhancement of portability within the NAG project. In *Proceedings of the Workshop on Portability of Numerical Software*, To Appear.

35. DuCroz, J. J.; Hague, S. J.; Siemieniuch, J. L. Aids to portability within the NAG project. In *Proceedings of the Workshop on Portability of Numerical Software*, To Appear.

36. Ford, B. Private Communication (1977).

37. *IMSL Numerical Computation Newsletter*, issue no. 1 (January, 1972), International Mathematical and Statistical Libraries, Inc., Sixth Floor, GNB Bldg., 7500 Bellaire Blvd., Houston, Texas 77036.

38. Aird, T. J. The IMSL Fortran Converter: An approach to solving portability problems. In *Proceedings of the Workshop on Portability of Numerical Software*, To Appear.

39. Battiste, E. L. Mathematical software and the private sector, Talk at ACM Annual Conference (1976).

40. Osterweil, Leon J. and Fosdick, Lloyd D. DAVE - a validation, error detection and documentation system for FORTRAN programs. *Software-Practice and Experience* 6 (1976), 473-486.

41. Ryder, Barbra G. The PFORT verifier, *Software-Practice and Experience* 4 (1974), 359-378.

42. Ramamoorthy, C. V. and Ho, Siu-Bun F. Testing large software with automated software evaluation systems. *IEEE Trans. on Software Engineering* 1, 1 (March 1975), 46-58.

43. Stucki, Leon G. and Foshee, Gary L. New assertion concepts for self-metric software validation. pp. 59-65 in reference 14.

44. Voevodin, V.; Gaisaryan, S.; Kabanov, M. Automated program generation. In *Numerical Analysis in Fortran*, Vol. 1. Moscow State University Press (1973) (In Russian).

45. Cowell, Wayne. The validation of mathematical software. In *Proceedings of INFOPOL-76, International Conference on Data Processing,* To Appear.

46. Wilkinson, J. H. Modern error analysis. *SIAM Review* 13, 4 (October, 1971) 548-568.

47. Lyness, J. N. and Kaganove, J. J. A technique for comparing automatic quadrature routines. *Computer Journal,* To Appear.

48. Greenberger, M.; Aronofsky, J.; McKenney, J. L.; Massy, W. F., eds. *Networks for Research and Education.* M.I.T. Press (1974).

Work supported in part by the U.S. Energy Research and Development Administration and the National Science Foundation.

Applied Mathematics Div.
Argonne National Lab.
Argonne, Illinois 60439

Dept. of Computer Science
Univ. of Colorado
Boulder, Colorado 80302