UCID-30050
c. 1

PART 1

32

UCID - 30050, Part 1
COMPUTER
DOCUMENTATION

LAWRENCE LIVERMORE LABORATORY
University of California/Livermore, California

CONSTRUCTION OF MATHEMATICAL SOFTWARE PART I: GENERAL DISCUSSION

F. N. Fritsch

August 9, 1972

32

FOREWORD

This is Part I of the five-part report Construction of Mathematical Software. Parts I, III, and V are being issued contemporaneously; parts II and IV will appear later.

The following outline of the complete report lists the topics covered in each of the five parts:

Part I. General Discussion (F. N. Fritsch)

1. Background Material
2. Design Criteria and Tradeoffs
3. Problems of Mathematical Software Distribution at LLL
4. Evaluation of Mathematical Software
5. Summary and Conclusions

Part II. Some Examples of Mathematical Software

1. Software for the Elementary Functions (R. E. von Holdt)
2. Software for Input/Output Conversion (R. E. von Holdt)
3. EISPACK: Software for the Algebraic Eigenvalue Problem (R. P. Dickinson, Jr.)
4. Calculating Padé Approximants (R. L. Pexton)

Part III. The Control of Error in the Gear Package for Ordinary Differential Equations (A. C. Hindmarsh)

Part IV. Nonlinear Least Squares Codes (Brad Johnston)

Part V. Some Application Codes

1. Organization of the HEMP Code (Tokihiko Suyehiro)
2. A Simplistic View of Light Diffusion and the MORSE Code (Nan Davies)
3. An Examination of Some Table Searching Methods Found in Texts and in the Field (Glenn L. Hage)

CONTENTS OF PART I

C709. Construction of Mathematical Software. This course will be a study of the basic principles and tradeoffs involved in the construction of mathematical software. It will also include a detailed examination of several examples of good mathematical software.

Prerequisite: C-Department courses C201 ("Programming Techniques") and C701 ("Elements of Numerical Mathematics") or their equivalents are required. Also, a good working knowledge of the FORTRAN programming language will be essential.

The textbook for the course was Mathematical Software, edited by John R. Rice, Academic Press (New York, 1971). Wherever it appears in this report, "the text" refers to this book.

The course was run seminar-style. It began with a few lectures by the instructor in order to present basic definitions and set forth some of the criteria that must be considered when constructing mathematical software. These lectures were followed by talks by members of the class, interspersed with guest lectures. The object of these presentations was to investigate in some detail the structure of a number of examples of mathematical software, to see how (or if) the criteria mentioned above are met, and to determine how the routines might be improved. There were 23 lectures, each lasting approximately one and one-half hours.

The following schedule for the lectures is in the order in which they were actually presented. Following the title of each talk, a reference to the written version is given in square brackets [ ]. Unless otherwise indicated, these refer to other parts or chapters of this report.

Lecture	Date	Speaker and Topic
1	March 29	F. N. Fritsch Organizational Meeting; Definition of Mathematical Software
2	March 31	J. F. Traub, LLL Consultant* The Bell Laboratories Library Project. [Prof. Traub's talk was essentially an expanded version of his paper on pp. 131-139 of the text.]

*Professor Traub is head of the Computer Science Department at Carnegie-Mellon University. He was head of the Bell Laboratories Library Project until 1970.

Lecture	Date	Speaker and Topic
3	April 10	F. N. Fritsch Problems of Mathematical Software Distribution at LLL. [Part I, Chapter 3.]
4	April 12	F. N. Fritsch Design Criteria and Tradeoffs. [Part I, Chapter 2.]
5	April 17	R. E. von Holdt Software for the Elementary Functions. [Part II, Chapter 1.]
6	April 19	R. E. von Holdt Software for Input/Output Conversion. [Part II, Chapter 2.]
7	April 24	A. C. Hindmarsh A Package for Ordinary Differential Equations Based on the Methods of C. W. Gear [Part III and the references given therein.]
8	April 26	
9	April 28	
10	May 1	
11	May 3	J. H. Bolstad Software for the Solution Systems of Linear Equations. [Based on Forsythe and Moler, Computer Solution of Linear Algebraic Systems, and pp. 347-356 of the text.]
12	May 8	
13	May 10	B. M. Johnston Nonlinear Least Squares Codes. [Part IV.]
14	May 12	
15	May 15	
16	May 17	R. P. Dickinson EISPACK: Software for the Algebraic Eigenvalue Problem. [Part II, Chapter 3.]
17	May 22	F. N. Fritsch Evaluation of Mathematical Software. [Part I, Chapter 4.]
18	May 24	R. L. Pexton Computation of the Padé Table. [Part II, Chapter 4.]
19	May 26	T. Suyehiro Organization of the HEMP Code. [Part V, Chapter 1.]
20	May 31	N. W. Davies A Simplistic View of Light Diffusion and the MORSE Code. [Part V, Chapter 2.]
21	June 2	G. L. Hage An Examination of Some Table Searching Methods Found in Texts and in the Field [Part V, Chapter 3.]

Lecture	Date	Speaker and Topic
22	June 5	R. F. Hausman Four Subroutines for One-Dimensional Function Minimization. [Based on reports UCID-30002 (GOLDEN), 30016 (BOUND), 30038 (QNTRP), and 30039 (LCLMIN).]
23	June 7	F. N. Fritsch Summary: **What Have We Learned About the Construction of Mathematical Software?** [Part I, Chapter 5.]

3. Requirements for a Certificate of Completion

The following rules regarding requirements for full participation were handed out at the beginning of the course:

"Those participants in this seminar who wish to obtain a 'Certificate of Completion' for Computation Department Course C709 must fulfill the following minimum requirements:

1. Maintain regular attendance at meetings of the Seminar;
2. Present a talk (at least 45 minutes in length) on some aspect of the construction of mathematical software;
3. Write a paper in support of your talk.

"Most of the lectures will be discussions of particular examples of mathematical software. The initial source of information may be the textbook or a paper in the computing literature. Possible journals of interest include: BIT, Comm. ACM, Comput. J., J. ACM, Math. Comp., Numer. Math., SIAM J. Numer. Anal. Follow up on the references in your primary source, so that you completely understand the paper. The talk should consider the following questions:

- How has the routine been optimized with regard to the various criteria of accuracy, efficiency, machine independence, etc.?
- How is error control handled?
- How extensively has the algorithm been analyzed?
- How extensively has the routine been tested?
- How good is the documentation?
- How can the routine be improved?

While enough of the mathematical background should be presented so that the audience can understand what the algorithm is doing, remember that we are concentrating on the organization of the routine, rather than its mathematical properties.

"Some topics acceptable for presentation will not fit into this general outline. Feel free to suggest topics.

"The required paper will essentially be a formal written version of the talk. It must be in term paper format. It should include a discussion of the primary source and the references consulted, and include a summary of any additional testing you have performed yourself. It must be typed, double spaced. Any graphs or diagrams included must be neatly drawn, as for inclusion in a technical report."

CHAPTER 2. DESIGN CRITERIA AND TRADEOFFS

1. Definitions

Before beginning our discussion of the design of mathematical software, a few definitions are in order.

1.1. Algorithm. According to Henrici [9, p. 4] an algorithm is "a set of directions to perform mathematical operations designed to lead to the solution of a given problem." Similarly, Conte [4, p. ix] defines an algorithm to be "a complete and unambiguous set of procedures leading to the solution of a mathematical problem." An algorithm is similar to a **recipe** in a good cookbook, which directs the cook to perform certain chemical operations. The restriction "unambiguous" requires that terms like "dash" and "pinch" be avoided, unless they are carefully defined. (Compare Zadeh [19], however, where the concept of "fuzzy algorithm" is introduced.)

Knuth [11, pp. 1-9] goes further to require an algorithm to possess five basic properties:

(1) Finiteness. An algorithm must terminate after a finite number of steps. According to Knuth, a procedure that has all of the characteristics of an algorithm (except that it possibly lacks finiteness) may be called a "computational method."

(2) Definiteness. Each step of an algorithm must be precisely defined; the action to be carried out must be rigorously and unambiguously specified for each case.

(3) Input. An algorithm has zero or more inputs, i.e., quantities that are given to it initially before the algorithm begins.

(4) Output. An algorithm has one or more outputs, i.e., quantities which have a specified relation to the inputs.

(5) Effectiveness. An algorithm is also generally expected to be effective. This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can, in principle, be done exactly and in a finite amount of time by a man using pencil and paper. (Example of a noneffective step: "If $k \geq n$, where n is the largest prime number for which $n + 2$ is also prime, go to step 7." It has not yet been demonstrated that such a number n exists.)

1.2. Implementation of an Algorithm. While we shall frequently identify the concepts "algorithm" and "subprogram" for the purpose of this course, it is important to distinguish between an algorithm and its implementation on a computer. We quote from Cody [2] on this subject:

> It is common to identify...an algorithm with a computer program as is done in the algorithms section of the Communications of the ACM. But we must draw a careful distinction between the mathematical algorithm and the corresponding software, for an algorithm can be embedded in many different computer programs. As host to the algorithm each program manages the flow of information to and from the algorithm and performs certain ancillary services such as detecting and processing error conditions. Different programs may contain, for example, different convergence criteria and may offer the user varying degrees of control over the solution of his problem. A good algorithm is therefore a necessary, but not a sufficient condition for a good subroutine.

1.3. Mathematical Software. According to Cody [2], "Mathematical software is a relatively new term denoting computer programs implementing mathematical algorithms." Cody goes on to state that "A second primary distinction between an algorithm and mathematical software is in documentation. ...Software documentation must include... information on such things as the proper interface with other computer programs and the remedial action taken when improper data or other errors are encountered. An item of mathematical software thus consists of a computer program and its documentation." (Underlining added for emphasis.)

I prefer the above definition to that given by Rice in the preface to our text [15, p. xv]: "Mathematical software is the set of algorithms in the area of mathematics. Its exact scope is not well defined; for example, we might include any algorithms that result from the creation of a mathematical model of nature and an analysis of that model. The scope defined by this book is more narrow and includes only topics of a definite mathematical nature or origin."

1.4. Basic Algorithm. An algorithm that uses a specific method to solve a specific mathematical problem will be called a basic algorithm. This definition is purposely ambiguous, since the notion of whether an algorithm is to be considered "basic" is context-dependent. Algorithms to solve the following mathematical problems would probably be considered basic algorithms in most contexts:

(1) Compute the real zeros of a given function.

(2) Evaluate $\int_a^b f(x)\,dx$, where f is a given function, a and b are given real numbers.

(3) Find the largest eigenvalue of a given matrix.

1.5. Subalgorithm. A subalgorithm is an algorithm to solve an even more specific mathematical problem. It is generally a portion of a basic algorithm. The following are examples of tasks performed by subalgorithms:

(1) Locate an interval in which a given function changes sign.

(2) Transform a matrix to tridiagonal form.

(3) Compute the inner product of two vectors.

1.6. Polyalgorithm. An algorithm that uses some decision logic to determine which of several basic algorithms should be used to solve the problem at hand is called a polyalgorithm. This term was apparently coined by Rice [14]. Polyalgorithms generally occur in the context of automatic numerical analysis.

1.7. Automatic Numerical Analysis. Automatic numerical analysis refers to a routine or system of routines designed to enable one to use a computer to solve mathematical problems, under the assumptions that

(1) the problem will be stated in normal mathematical terms, and
(2) the user will know little or nothing about the numerical analysis aspects of his problem.

Automatic numerical analysis clearly requires that more attention be paid to reliability and the design of the user interface than is done in other types of mathematical software. (See Fig. 2.1.) Very little will be said about this subject in this course. Those interested in more information should refer to Rice [14] and the other papers in the volume containing this paper, and to the papers by de Boor [5] and Gear [8] in our text.

1.8. Driver. A driver is a routine, generally a main program, that sets up a problem, calls one or more basic algorithms (or polyalgorithms) to solve it, and reports the results to the user. There are basically two types of drivers:

(1) Test routines
(2) User interfaces

The latter are routines to handle the input/output required for easy use of a particular algorithm. They are generally more specific and less flexible than the basic algorithms they call upon. (Note that we are now identifying algorithms with the subroutines that implement them.)

1.9. Domain of Applicability. The domain of applicability of a piece of mathematical software is the set of problems that it can successfully solve. For most problem areas, the boundary between the class of successfully solved problems and the class on which a program fails is not strictly defined; the performance will degrade

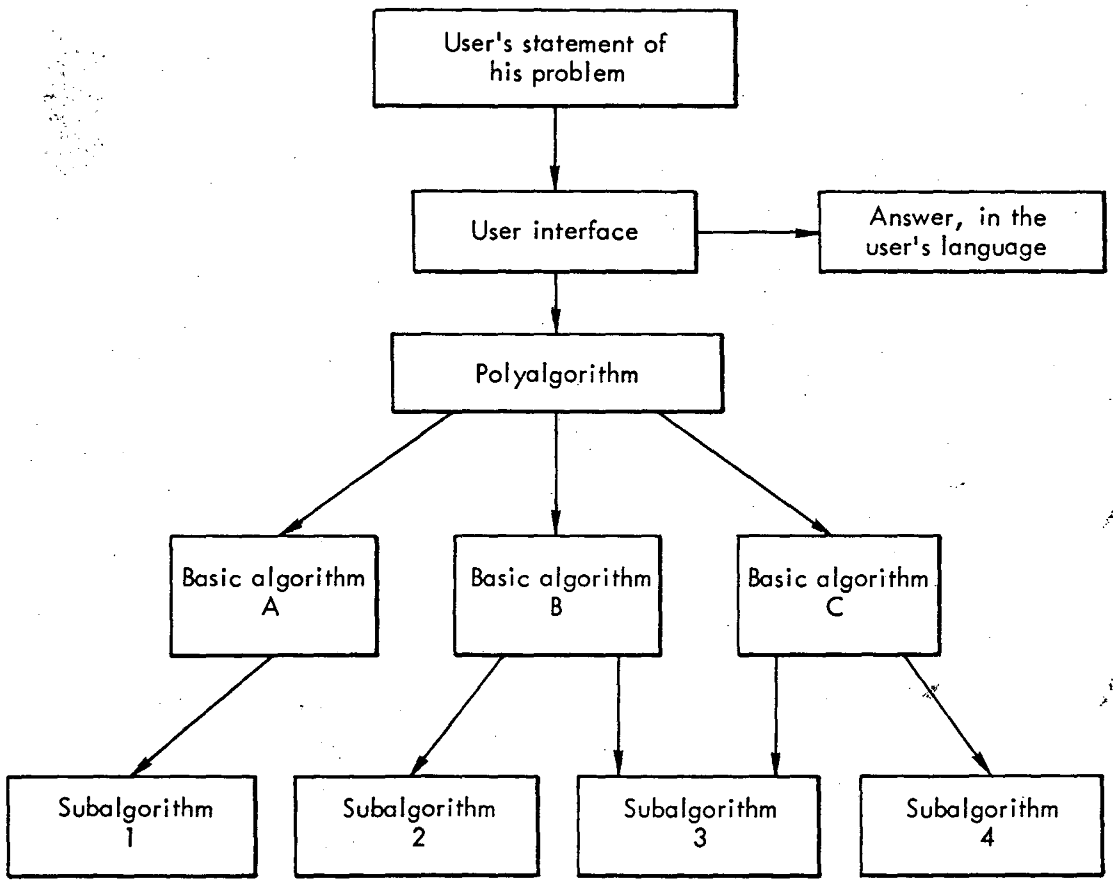


Fig. 2.1. Automatic numerical analysis.

from execellent to unacceptable near this boundary. The domain of applicability is thus a "fuzzy set," in the sense of Zadeh [18].

2. Design Criteria for Mathematical Software

In this section we discuss ten desirable characteristics of mathematical software. These are derived from similar lists in Fritsch and Hausman [7], Kuki *et al.* [13], Rice [14, 16], and elsewhere.

2.1. Good Documentation. The importance of documentation has already been mentioned in section 1.3. Kuki and Cody [12] state, "A program is virtually useless to users other than its author unless it is accompanied by reliable documentation." In another section of this report we read, "The computing public is faced with an ever-growing arsenal of programs prepared and distributed in divers ways. ...(It) is not feasible for individual users to distinguish the good from the bad among them. It is often simpler to build a new program than to search for one in the existing arsenal. For a collection of programs to be useful, users must have reliable information on the programs' efficiency, areas of applicability, areas of comparative strength, etc. Good documentation by the author is supposed to contain this information." The SHARE Evaluation Guidelines [13] further declare that "a program for which no decent documentation is available can be branded useless without further testing."

Kuki and Cody [12] identify four major classes of information that should be present in the documentation of a program. These are:

(1) Statement of Purpose. This is to quickly tell the potential user whether the program can help him solve his problem. A complete statement of the problem that the program solves, together with any important restrictions, should appear here.

(2) Instructions on How to Use It. This section should include information on the status of input variables upon return and any side effects. Optional features and any special environmental requirements should also be clearly described here.

(3) Description of the Algorithm Used.

(4) Performance Information. The documentation should identify the domain of applicability of the program as well as possible. It should contain accurate information on all of the following performance characteristics and clearly indicate the extent to which the program has been tested by the author.

See Fritsch and Hausman [7] for additional thoughts on documentation.

2.2. Reliability. The term "reliability" generally refers to the numerical accuracy of the results produced by a program. We also wish to include here what Cody [2] calls robustness, namely the "ability to recover from abnormal situations without unnecessary termination of the computer run." Furthermore, a reliable piece of mathematical software will either return the correct* answer or else return a wrong* answer with an indication that it is erroneous. It will never "blow up," nor will it quietly return bad results as though nothing has gone wrong.

2.3. Error Control. By error control we refer to either or both of the following:

(1) monitoring and control of accuracy during the course of a computation, or
(2) *a posteriori* accuracy checks.

This may involve the use of special devices, such as interval or significance arithmetic. Whenever practical, an estimate of the error in the results should be returned to the user. Error control is clearly closely related to program reliability.

2.4. Efficiency. Program efficiency refers both to its execution speed and to the optimum allocation of computer resources (program size).

2.5. Flexibility. A flexible piece of mathematical software can handle a wide class of problems.

2.6. Modifiability. Closely related to flexibility is the concept of modifiability: Is the program written in such a way that it is easy to extend (or reduce) the domain of applicability or modify the information that it produces about the problem? This term is often considered synonymous with modularity, the segmenting of a program into a number of relatively small, logically independent blocks, since modularity generally enhances modifiability. Modifiability also requires that the routine be carefully commented and that programming "tricks" be avoided or carefully explained. (See Fritsch and Hausman [7, pp. 10-14] for suggested programming standards.)

2.7. Ease of Use. Here we are considering the design of the interface between the user and the program. Cody [2] refers to this as accessibility. The user generally (but not always) wishes to be required to supply only those parameters that are natural to the statement of the problem. Long calling sequences generally discourage use of a subroutine. Even the order of the arguments can affect ease of use, so this must be considered when designing the calling sequence.

2.8. Reasonable Diagnostics. Another part of the user interface is the design of the diagnostic messages produced in case of abnormal conditions. For example, good mathematical software will not produce the diagnostic "negative argument in square root function" when the real problem is that a matrix that was supposedly positive definite has a negative eigenvalue.

2.9. Transportability. A program is transportable (or simply portable) if it can be moved from one computer system to another with a minimum of change. A hand-coded

*The concepts of correct and wrong are necessarily "fuzzy" in this context.

subroutine is clearly not transportable. A routine written in ANSI Standard Fortran is quite likely to be transportable, especially if calculations that depend on machine characteristics (word size, maximum floating-point exponent, etc.) are avoided or carefully isolated and identified. The designer of mathematical software should avoid the use of exotic features of a particular compiler whenever practical.

2.10. Common Sense. Rice [14] defines common sense to be "the application of one (or more) of a large number of simple rules to an appropriate bit or two from a large body of information." Common sense is required, for example, in the stopping criteria for an iterative process. It is extremely difficult to implement. (See [14] for further discussion of this concept.)

3. Tradeoffs: Software Engineering

It is obvious that many of the design criteria discussed in the previous section are incompatible, i.e.,:

(1) It is usually impossible to simultaneously minimize the speed and size of a subprogram.

(2) Short calling sequences (ease of use) may not provide the user with adequate control over the solution of his problem (flexibility).

(3) A hand-coded subroutine generally runs faster (efficiency) than a Fortran version (portability).

(4) Flexibility demands that library routines contain no input/output commands, which inhibits the provision of adequate diagnostics. (One possible solution to this problem is to provide an output error flag which, at the user's option, can be passed on to a separate subroutine that prints the error messages.)

Because of this conflict of goals, certain tradeoffs must be made. It is here that the construction of mathematical software becomes an art (or perhaps an engineering discipline) rather than a science.

The individual designing a piece of mathematical software should be aware of the tradeoffs (engineering decisions) he has made in the process. These decisions should be compatible with the intended use of the routine, and they should be based on such considerations as the following:

(1) Is the software to be part of an automatic numerical analysis system, a general-purpose subroutine library, or a production code?

(2) How often will it be used? (one-shot job vs production code)

(3) Who will use it? (That is, how much work should be devoted to the user interface?)

With the list of design criteria from section 2 in mind, the mathematical software designer frequently finds ways to improve some features of his program without seriously degrading the others:

(1) It is usually possible to increase transportability without seriously affecting the efficiency of a program. For example, a fast, hand-coded dot product routine (DOT) is used in a Fortran matrix manipulation routine. If one simply inserts comment cards containing a mathematically equivalent Fortran DO-loop immediately after the call to DOT, the Fortran routine has been rendered transportable without sacrificing efficiency.

(2) Consider whether the computer time saved is worth the extra people time when resolving the conflict between ease of use and efficiency.

(3) It is usually easy to find ways to improve the efficiency of a well-commented logically organized (i.e., easily modifiable) program.

(4) It is generally easier to consider the user interface in the initial design stages than to attempt to modify it after the completed routine has been debugged.

Fritsch and Hausman [7, p. 10] state that the programmer generally attempts to attain some optimum combination of the following goals:

(1) Maximize accuracy.
(2) Minimize execution time.
(3) Minimize size of object code.
(4) Maximize range of parameters to be allowed (i.e., domain of applicability).
(5) Maximize comprehensibility of source program (modifiability).
(6) Maximize convenience of usage.

The programmer should be aware of the combination he has chosen to optimize, why he has made this decision, and how it has been implemented.

One final note: <u>Good documentation does not conflict with any of the other nine design criteria</u>. Any piece of good mathematical software must be well documented, and this documentation should contain information on the author's choice of optimum design criteria.

CHAPTER 3. PROBLEMS OF MATHEMATICAL SOFTWARE DISTRIBUTION AT LLL

We discuss here the current mathematical subroutine library situation at LLL and propose a remedy. But first, let us consider how the current situation came about.

1. Background

The following factors have all contributed to the current library status.

1.1. Variety of Hardware. LLL has traditionally supported a wide variety of computing equipment from different manufacturers. Five years ago we had the following large computers: IBM 7094, IBM 7030 (STRETCH), Remington-Rand LARC, CDC 3600, CDC 6600. Such an environment is clearly not conducive to the development of a central, standardized subroutine library.

1.2. Variety of Systems. To make matters even more complex, LLL computer users avail themselves of a wide variety of software systems (usually with incompatible compiler/library structures). For example, on the CDC 6600's we currently have three Fortran compilers (MONITOR-LRLTRAN, CHIP, ORDER-CHAT), each with its own loader and its own supporting library.

1.3. Other Languages. While LLL is basically a Fortran laboratory, there is a growing number of users who program in assembly language (using one of several incompatible assemblers), COBOL, or APL. Incompatible languages generally require separate subroutine libraries.

1.4. Incompatible Requirements. Finally, the LLL computer facility has to service two distinct groups: users with short jobs and those with production codes that grind away for hours at a time. It is difficult, if not impossible, to meet the needs of such a wide variety of computer users with a single program library.

2. The Current Situation

At present there are two types of program libraries available to LLL computer users.

2.1. System Libraries. These are the libraries that exist to provide I/O support and other services to the users of specific systems.

a. Examples:

(1) CLIB (for CHIP)
(2) LRLLIB (for MONITOR)
(3) ORDERLIB (for CHAT/ORDER)

b. Properties:

(1) Reside permanently on disk (public file) for each worker computer.

(2) Easy access — the user need only call the routine(s) he wishes to use.

(3) Must be constantly used to justify the disk space.

(4) Contain very few mathematical subroutines. (Generally, only the elementary functions, a uniform random-number generator, and MLR. CLIB also contains routines to compute one- and two-dimensional integrals and an ODE solver.)

(5) Documentation generally very scanty.

2.2. The "Use at Your Own Risk" Library. This is what is left of the CIC (Computer Information Center) Library. Properties:

(1) Card decks and write-ups available at the TID Main Library.

(2) No quality control — anybody who wishes can donate a routine.

(3) Nobody is responsible for the integrity of the library.

(4) Contains most of the mathematical software that is generally available to LLL computer users.

We feel that the current library situation is intolerable for a scientific computing facility of the magnitude of LLL's. In the following section we propose one possible solution: a NAG-sponsored mathematical subroutine library.

3. Proposal for an LLL Mathematical Subroutine Library

3.1. Criteria for Design of the Library.

(1) Reliable.

(2) Easy to use — machine accessible. (Because of the premium on permanent disk space, this will probably require use of the Photostore.)

(3) As system-independent as possible.

(4) Easy to update.((3) and (4) together imply a Fortran source library.)

(5) Consultants available.

3.2. Minimal Requirements for Inclusion of a Routine in the Library.

(1) Has been documented in the computing literature or as a CIC or UCID Computer Documentation Report. (The submitter must provide us with a copy of the documentation.)

(2) Has undergone extensive testing at LLL or has been certified elsewhere and has successfully run a few test problems locally. (The submitter must provide evidence that the certification has been done. The EISPACK routines are in the second category.)

(3) Has a NAG sponsor. (That is, somebody in NAG* has agreed to be the Group's consultant on the routine. If a routine was developed by someone else at LLL, the sponsor will generally be in close communication with the author.)

*It seems more realistic, in view of limited manpower in NAG, to establish a Laboratory-wide board of sponsors for the mathematical subroutine library.

3.3. Problems.

(1) What sort of a user's manual should be provided? How extensive should the write-up be? How distributed?
(2) How can we obtain statistics on library usage?
(3) How can we obtain a "hostile referee" for a locally developed routine? (Quite frequently, the subroutine developer is the only person at the Laboratory with the expertise needed for an evaluation of the program.)

3.4. Other Considerations.

(1) How much manpower will be required? Who will pay for it?
(2) What is the optimal level of testing for the routines in such a library?
(3) What about the use of a commercially available library?

CHAPTER 4. EVALUATION OF MATHEMATICAL SOFTWARE

1. Introduction

In Chapter 2 we discussed the following ten attributes of good mathematical software:

(1) Good documentation
(2) Reliability
(3) Error control
(4) Efficiency
(5) Flexibility
(6) Modifiability
(7) Ease of use
(8) Reasonable diagnostics
(9) Transportability
(10) Common sense

Now we wish to discuss evaluation: How does a given piece of mathematical software rate on these ten design criteria?

The major sources on the general philosophy of software evaluation are Kuki, et al. [13], Kuki and Cody [12], and Cody [2]. Rice [16, pp. 33-37] also discusses various aspects of the subject. Other references may be found in the "Bibliography on Subroutine Certification" in Appendix B.

1.1. Motivation. Why is the evaluation of mathematical software important? We quote from Kuki and Cody [12] on this subject:

> The computing public is faced with an ever-growing arsenal of programs prepared and distributed in divers

ways. Without adequate controls on the quality of those programs, it is not feasible for individual users to distinguish the good from the bad among them. It is often simpler to build a new program than to search for one in the existing arsenal. For a collection of programs to be useful, users must have reliable information on the programs' efficiency, areas of applicability, areas of comparative strength, etc. Good documentation by the author is supposed to contain this information. When it does, verification of the documentation is all that is needed. When, as is too often the case, such information is inadequate or missing from the documentation, certification can only be effective if it provides the missing information through a general evaluation of the program.

Furthermore, the results of evaluation projects must be accessible to the users of mathematical software if this need is to be met.

1.2. Definitions. The preceding passage introduces several terms which should be defined more precisely.

Testing will be left as an undefined term. Several types of testing will be distinguished later.

Verification refers to the validation of claims made by the author of a program in his documentation, by analytical means or by actual machine testing.

Evaluation goes beyond verification to express a value judgement as to how well the above design criteria have been met and, possibly, how the program compared with other programs to solve the same problem.

Certification is frequently used as a synonym for evaluation. With Cody [2], however, we prefer to reserve the term "certified software" for the case where some guarantee of the quality of the program is implied, as in the case of the subroutines certified by the NATS project [1]. We shall be concerned here with evaluation, not certification.

2. Aspects of Program Evaluation

Kuki and Cody [12] set forth four aspects of mathematical subroutine evaluation:

(1) Quality of the program
(2) Quality of the documentation
(3) Design of tests
(4) Documentation of the evaluation

2.1. Quality of the Program. When evaluating the quality of the program, one should distinguish among limitations inherent in the problem to be solved (such as ill-conditioned polynomial root-finding or matrix inversion problems), limitations of an algorithm, and limitations in its implementation (a good linear system solver can be ruined by failure to accumulate a crucial inner product in double precision). While inherent limitations are generally not considered to be the responsibility of a mathematical subroutine, they must still be taken into account by the evaluator. Kuki and Cody [12] recommend that a program's accuracy be interpreted relative to the inherent indeterminacy of the problem. They cite as an example of poor software design a polynomial root-finder that "liberally employs extra precision arithmetic to insure machine accuracy of multiple root solutions based on the assumption of exactness of the given coefficients," an assumption that is usually false.

The SHARE program evaluation guidelines [13] suggest that reviewers consider the following "performance attributes" when evaluating the quality of a program:

(1) Reliability
(2) Accuracy
(3) Timing
(4) Size (of the program)
(5) Features: "Consider the range of admissible input values, extra entry points, features such as error analysis, by-products, and so on."
(6) Design: ease of use, quality of output

We shall have more to say about this later.

2.2. Quality of the Documentation. Documentation has been discussed at length in a previous lecture. (See Section 2.1 of Chapter 2.) As we have already mentioned, much of the evaluation process is aimed at verification of the documentation. The documentation must also be evaluated with regard to clarity, conciseness, and completeness (the "three C's" of program documentation). Good documentation should contain accurate information on all of the above performance attributes, including an indication of any limitations. "The documentation should state clearly the extent and the condition of tests conducted by the author." [12] Furthermore, it should be written in such a way that the potential user can easily determine whether the program is applicable to the solution of his problem.

2.3. Design of Tests. We concur with the following material quoted from Kuki and Cody [12]:

> We believe certification of any significance should involve a fair amount of actual machine tests unless it is obvious from the author's own documentation that the program is not worthy of such attention. Even when the documentation contains the results of careful and thorough tests made by the author, a reasonable amount of machine tests should be done to verify them or supplement them. Although the amount of testing is left to the judgement of the reviewer, an effort should be made

> to cover the entire range of applicability of the program.
>
> The primary characteristics of good tests are rigor and significance. Tests must be conducted in the most rigorous manner consistent with the requirements of the certification. Tests which are significant for one program may not be significant for another very similar program. For example, an examination of the Wronskian values for a Bessel function routine can be significant only if the Wronskian is not used to generate the function values in the first place.
>
> Choice of proper criteria for the measurement of accuracy is a delicate problem. Here a great deal of care must be taken to choose error criteria which must reveal the performance characteristics of the program. ...Ultimately, it is the user's application which determines the preference on error criteria. Users will want to have error figures presented in the form that can be most effectively incorporated into an error estimate of the total computation. ...
>
> ...(For many subroutines) the heart of the test plan should be what we call critical range tests; that is, tests designed to probe particularly sensitive areas of the program such as regions where underflow or overflow of intermediate results may occur, or argument ranges bracketing the boundaries between methods of computation. The choice of a significant set of critical range tests is indeed a task challenging the ingenuity of the reviewer.

Timing tests also provide useful information, especially when comparing rival programs. The results may be reported in terms of actual machine running times, operation counts, or number of function evaluations (for a routine requiring a function as one of its inputs).

The theory and practice for testing of function subroutines is "in a most satisfactory state of affairs. The most widely exploited technique for accuracy evaluation is a Monte Carlo approach based on forward error anlaysis." [2] See Cody [3] for a general description of this process. Several specific evaluations of this type are cited in Appendix B. The design of tests in other areas is still pretty much up to the ingenuity of the evaluator. Such tests frequently involve the running of a large collection of "standard" test problems.

2.4. Documentation of the Evaluation. The final phase of any evaluation project is the preparation of a report. Please refer to Appendix A for the SHARE guidelines as to the purpose and desired contents of such a report. We find it difficult to improve on this material. The evaluator must completely and reliably document his work. The report must contain substantiating evidence for any criticism levied against the original program.

3. Levels of Testing

We digress for a moment to distinguish several levels of mathematical subroutine testing.

3.1. Plausibility Testing. The most superficial level of testing consists of running two or three sample problems and comparing the results with those obtained elsewhere. This type of testing may be appropriate in two cases:

(1) A program is imported from another installation — The purpose of the test is to detect any installation-dependent differences that prevent proper functioning of the program in its new environment.
(2) A library subroutine — The purpose of the test is to determine whether the library has been clobbered. (The subroutine had previously undergone much more extensive testing before it was put on the library.)

3.2. Program Checkout/Debugging. This is the minimal amount of testing that must be performed before a program can be considered debugged. As stated by Fritsch and Hausman [7, p. 15], at the very least "enough test cases should be performed to insure that all possible paths of the program are checked." Where possible, such results should be carefully checked with hand computations or with built-in tables of correct answers.

3.3. Performance Evaluation. Performance evaluation is a complete, but objective, testing process designed to assess program quality as discussed earlier in section 2. The result of a performance evaluation is "a set of statistics and parameters describing the performance of a given program on a standard set of problems." [2] This will be discussed in more detail in section 4.

3.4. Complete Program Evaluation. A complete program evaluation goes beyond performance evaluation to consider the quality of such subjective attributes as programming standards, ease of use, flexibility, and program documentation.

3.5. Comparison Testing. Up to this point we have been considering the evaluation of a single piece of mathematical software. Comparison testing involves performing a complete program evaluation on two or more routines that purport to solve the same problem, in order to compare their domains of applicability and their ratings on the various design criteria. The purpose of comparison testing may be to find one optimum algorithm for a given problem area (as in the Bell Laboratories Library One project [17]) or to merely indicate the areas of comparative strength and weakness among a collection of rival programs.

T. E. Hull [10] has attempted to establish a rigorous definition for the term "best method" in this context. While he applied this to the numerical integration of ordinary differential equations, the same ideas carry over to other problem areas. We assume that two rigorously defined sets are given:

P = a class of problems. (Usually, a set of inputs to a subroutine, including error tolerances. Perhaps it is useful to think of P as a sample space.)

M = a class of methods. (Usually, a collection of subroutines.)

On P and M we define a function G such that, for $m \in M$, $G(m,P)$ is a measure of the goodness of the method m, relative to the set of problems P. We now quote Hull's definition [10]:

> If G is real-valued, and if it is desirable to have small values of G, we can now give a precise meaning to the statement that one method is better than another. Method m is better than method m', relative to the class of problems P, according to the criterion G, provided $G(m,P) < G(m',P)$. And the definition of "best" now takes the following precise form. Method m is a best method, from the class of methods M, relative to the class of problems P, according to the criterion G, if $G(m,P) \leq G(m',P)$ for any $m' \in M$.

For further discussion, with examples, see [10].

While this definition is unlikely to be of much use in practice, it does serve to point out that one cannot claim that a program is "optimum" unless he clearly specifies the class of problems being considered, the criterion of goodness, and the class of methods over which the optimization is being performed.

4. Complete Program Evaluation

Cody [2] describes an ideal evaluation effort for an item of mathematical software as providing answers to five questions:

(1) What are the numerical properties (accuracy, etc.) and speed of the program?

(2) Precisely what problems does the program solve, i.e., what is the domain of the program?

(3) Is the coding correct?

(4) Is the program easy to use?

(5) Is the documentation appropriate?

The first two of these Cody identifies with the term "performance evaluation" (cf., Section 3.3, above), while the entire process is called "complete program evaluation" (cf., Section 3.4, above).

We have already discussed documentation and ease of use, here and in chapter 2. We shall consider the other three questions in reverse order.

4.1. Correct Coding. An affirmative answer to question 3, above, implies that the routine is free of both programming errors, or "bugs," and design errors. The latter is extremely difficult to check for. Cody [2] states, "The only common denominator appears to be the association of design errors with error detection or the lack of it. A square root routine which does not do something special when it encounters a negative argument clearly has a design error. Similarly, a routine for the solution of ordinary differential equations has a design error if it objects (or malfunctions) when the independent variable is stepped in a negative direction. The difference between these errors is essentially the degree of foresight required of the evaluator to check for them."

One should also consider good programming practices when evaluating the coding of a particular routine. A well-commented, logically organized Fortran program is clearly easier to modify, easier to evaluate, and simply more aesthetic than one that is not. See the CACM Algorithms Policy [6] for one group's thinking in this area.

4.2. Domain of Applicability. At the present state of the art it is only possible to obtain a very fuzzy picture of the domain of applicability of a given subroutine. This is not so difficult in the case of a function of one or two independent variables, but it becomes increasingly more difficult as the dimensionality of the input data increases. In cases where a function is one of the inputs, about the best we can do is to test the program on increasingly ill-conditioned input and report where, and in what manner, it fails. At present we generally are unable to determine precisely what properties of a problem are the most important for determining the difficulty of its numerical solution.

4.3. Numerical Properties. We have already touched upon the subject of question 1 in section 2.1. Here we shall be concerned with sources of error and with assessment of error.

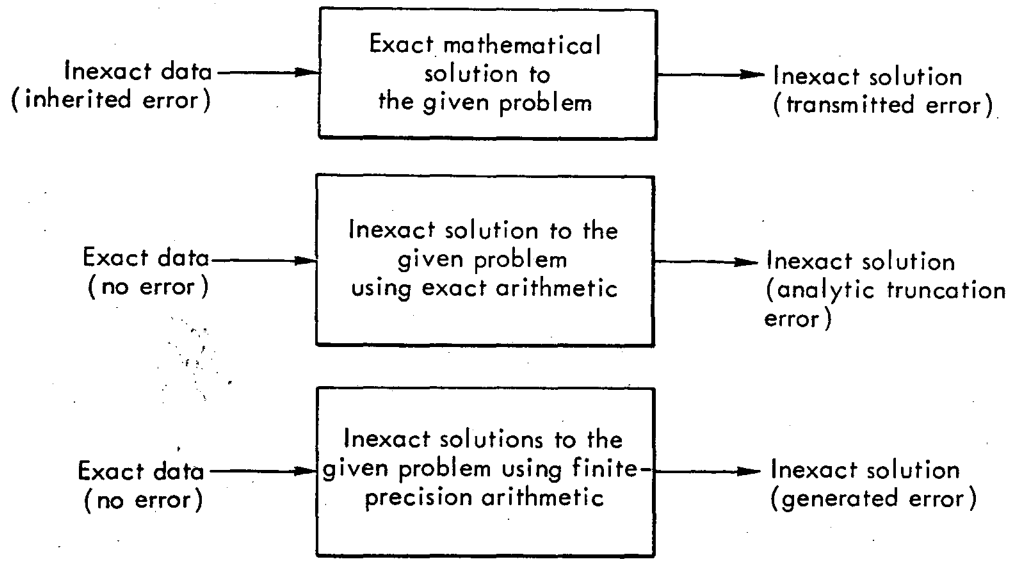
There are three primary sources of error in any numerical result. Transmitted error is the result of error in the original data (inherited error). While a mathematical subroutine is not responsible for transmitted error, the program evaluator must be careful not to contaminate his results with transmitted error due to the conversion of decimal numbers to the internal floating-point format of his machine. Analytic truncation error is the result of replacing an infinite mathematical process (integration, infinite series, etc.) by a finite one. Roundoff error is the result of using finite precision arithmetic. The sum of analytic truncation error and roundoff error is called generated error because it is the error generated in the program on the assumption of exact data.

$$\{\text{total error}\} = \{\text{transmitted error}\} + \{\text{generated error}\}$$

$$\{\text{generated error}\} = \{\text{analytic truncation error}\} + \{\text{roundoff error}\}$$

We see that it generally makes little sense for a program to do a lot of work reducing the generated error far below the level of the transmitted error due to a one-bit inherited error in the data. (See Fig. 4.1.)

To be somewhat more specific, suppose our subroutine is intended to evaluate a differentiable function f, given a value for the independent variable x. We know that $y = f(x)$ implies



Inexact data (inherited error)
Exact mathematical solution to the given problem
Inexact solution (transmitted error)
Exact data (no error)
Inexact solution to the given problem using exact arithmetic
Inexact solution (analytic truncation error)
Exact data (no error)
Inexact solutions to the given problem using finite-precision arithmetic
Inexact solution (generated error)

Fig. 4.1. Three primary sources of error.

$$\frac{dy}{y} = \frac{f'(x)\ dx}{f(x)} = x\ \frac{f'(x)}{f(x)}\ \frac{dx}{x}$$

(assuming $x \neq 0$, $f(x) \neq 0$). If z^* is an approximation to the nonzero quantity z, and the absolute error is $\Delta z = z - z^*$, then the relative error in z^* is

$$\delta z = \frac{\Delta z}{z} \approx \frac{dz}{z}.$$

Thus the above formula can be used to approximate the transmitted error due to an inherited error Δt:

$$\Delta y \approx dy = f'(x)\ dx \approx f'(x)\ \Delta x \qquad (1)$$

or

$$\delta y \approx \frac{dy}{y} = x\ \frac{f'(x)}{f(x)}\ \frac{dx}{x} \approx x\ \frac{f'(x)}{f(x)}\ \delta x \qquad (2)$$

In Eq. (1) the absolute error is multiplied by the factor $f'(x)$; in Eq. (2) the relative error is multiplied by the factor $xf'(x)/f(x)$. If the multiplicative factor is greater than one in absolute error, the transmitted error is greater than the inherited error, and no improvement of the mathematical method can reduce the total error below this level. Example: $f(x) = \tan x$, x near $\pi/2$.

The most obvious approach to accuracy evaluation is based on forward error analysis. That is, the program is checked on problems with known solutions, and the computational results are compared with the exact answers. If one is careful to avoid unintentional inclusion of inherited error, this can provide a direct measure of the generated error.

In certain areas, notably matrix computations and polynomial zero-finding, a backward error analysis is more appropriate. This means that the computed result is interpreted as the exact solution to a perturbation of the original problem, and we measure the difference between the problem posed and the problem solved. One can also use "consistency checks" involving theoretical identities. See Cody [2] and the papers referenced there for further discussion of this subject.

5. Evaluation Projects

We provide here a brief summary of known mathematical software evaluation projects. Refer to the section entitled "Present Technology" in Cody [2] for a more complete survey.

(1) The SHARE Numerical Analysis Project has been engaged in an evaluation of the mathematical subroutines in the SHARE (IBM computer users' group) Library for many years. See [13] and various SSD's for reports on the activities of the project.

(2) While currently inactive, the SIGNUM Subroutine Certification Group attempted to act as a centralized clearing-house for information on evaluation efforts. Reports on their activities can be found in various issues of the SIGNUM Newsletter.

(3) We have already mentioned the Bell Laboratories project [17].

(4) Funded by NSF and supported by Argonne National Laboratory, Stanford University, and University of Texas, the NATS (National Activity to Test Software) project "is a prototype effort to test and disseminate certain collections of routines as certified software." [1] A package of over thirty subroutines, known as EISPACK, to solve the algebraic eigenvalue problem has been tested and is now available to computer users. A collection of special function routines is currently being tested.

(5) Various universities and laboratories (including Argonne National Laboratory, Jet Propulsion Laboratory, and the University of Minnesota) have done performance evaluations on manufacturer-supplied libraries of function subprograms. These are cited in the "Bibliography on Subroutine Certification" (see Appendix B). In this same bibliography appear the results of various comparison tests that have been undertaken as individual efforts.

6. Conclusion

To summarize what has been said here, one must carefully determine his goals before undertaking an evaluation project. With these goals in mind, a plan for testing is designed and carried out — the details of the plan depending on the problem area. Finally, a report describing the testing procedure and the evaluator's interpretation of the results must be prepared.

CHAPTER 5. SUMMARY AND CONCLUSIONS

1. Course Summary

In the final lecture of the course we attempted to summarize what we have learned about the construction of mathematical software. First, we recalled certain basic definitions (see Chapter 2, Section 1):

(1) Algorithm
(2) Implementation
(3) Mathematical software — An item of mathematical software consists of a computer program that implements a mathematical algorithm, together with its documentation.
(4) Basic algorithm
(5) Subalgorithm
(6) Polyalgorithm
(7) Automatic numerical analysis
(8) Driver
(9) Domain of applicability

Next, we reviewed the ten design criteria (see Chapter 2, Section 2):

(1) Good documentation
(2) Reliability (accuracy; robustness)
(3) Error control
(4) Efficiency (speed; size)
(5) Flexibility
(6) Modifiability (modularity)
(7) Ease of use (accessibility) } user interface
(8) Reasonable diagnostics }
(9) Transportability
(10) Common sense

We noted that the construction of good mathematical software requires skills from three traditionally separate disciplines: mathematics, programming, and engineering. (See Fig. 5.1.) At present, mathematical software construction is an art, rather than a science.

Finally, we reviewed the eleven examples of mathematical software considered during the course, which are listed in Table 5.1, and attempted to evaluate them in terms of the ten design criteria. The results of these impromptu evaluations, based upon material presented in class, are given in Table 5.2. Example 10 was omitted because it doesn't really fit into this evaluation scheme. It can be seen that there is a good deal of variability in the quality of published mathematical software. (Note that thorough

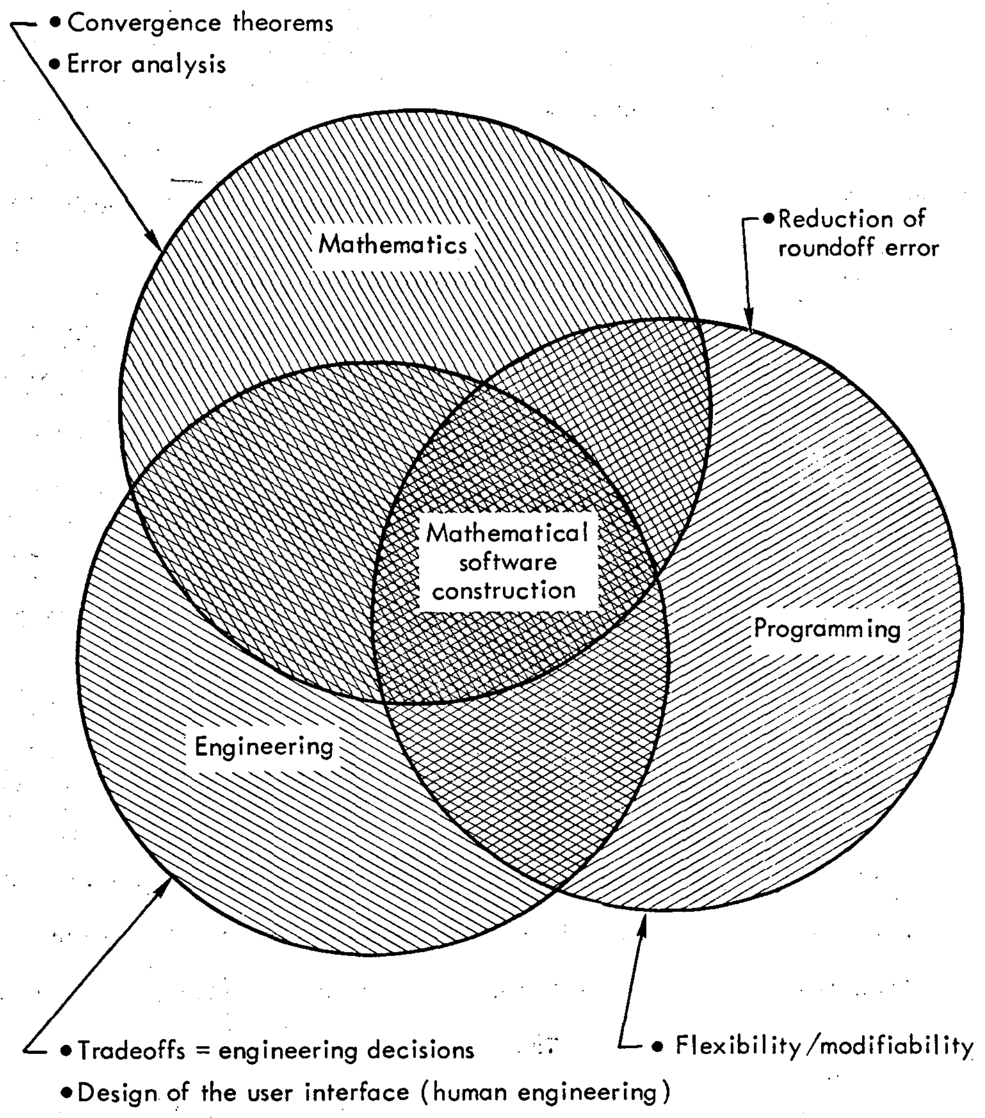

• Convergence theorems
• Error analysis
Mathematics
• Reduction of roundoff error
Mathematical software construction
Programming
Engineering
• Tradeoffs = engineering decisions
• Design of the user interface (human engineering)
• Flexibility/modifiability

Fig. 5.1. Mathematical software construction involves mathematical, programming, and engineering skills.

evaluations of these routines, according to the philosophy of Chapter 4, would require much more extensive study and testing.) The number of question marks after "Common Sense" probably indicates either that this design criterion was not adequately understood by the participants or that it is relevant only to polyalgorithms.

2. Conclusions

We feel that the course was successful in exposing the participants to several examples of mathematical software and introducing them to the basic concepts of construction and evaluation of such software. In retrospect, however, we believe that the students would have gotten more concrete value from the course if more time were spent on the presentation of general concepts, such as structured programming and documentation principles. The term project could then be the actual construction of a piece of good mathematical software, or the detailed evaluation of an existing program, by the entire class. That is, we recommend that the subject be treated as a participation course, rather than a reading course.

Table 5.1. Examples of mathematical software.

Example No.	Presented by	Description
1	von Holdt	Software for the Elementary Functions.
2	von Holdt	Input/Output Conversion.
3	Hindmarsh	STIFF: An Ordinary Differential Equations Package (Gear).
4	Boldtad	Solution of Linear Systems (Forsythe-Moler).
5	Johnston	Nonlinear Least Squares Curve Fitting (Bevington).
6	Dickinson	EISPACK: Eigenvalues and Eigenvectors of Matrices.
7	Pexton	Padé Table Computation (Longman).
8	Suyehiro	HEMP: A Large Applications Code, Using Finite Difference Methods.
9	Davies	MORSE: A Moderate-Sized Monte Carlo Applications Code.
10	Hage	Table Loop-Up Methods and the RANK Code.
11	Hausman	Four Subroutines for One-Dimensional Function Minimization.

Table 5.2. Tentative evaluations for some examples of mathematical software.

The example numbers are keyed to Table 5.1.

The following symbols are used in this table:

E	= Excellent
G	= Good
A	= Adequate
P	= Poor
B	= Bad
-	= Not applicable
?	= Not enough information available

	Example No.									
Criterion	1	2	3	4	5	6	7	8	9	11
Documentation	P	B	E	A	G	E	G	G	G	E
Reliability	E	E	E	E	?	E	?	G	?	E
Error control	E	G	E	E	?	E	?	A	?	E
Efficiency	E	E	G[a]	G	A	G	P[b]	A	A	?[c]
Flexibility	B	B	E	E	A	E	A	A	E	G
Modifiability	P	B	G	G	A	E	A	?	?	?
Ease of use	E	A	A	A	G	G	?	?	G	G
Diagnostics	-	?	G	G	?	G	?	A	?	G
Transportability	B	B	E	E	E	E	G	P	G	E
Common sense	-	?	A	?	?	?	?	?	?	?

[a]Applies to implementation of Gear's methods in the package. The implementation of Adam's methods in the package, as available at the time of the course, would have to be rated as P.

[b]Applies to the original code, not to Pexton's modifications.

[c]Depends on which subroutine is used and what function is to be minimized.

REFERENCES

1. J. M. Boyle, W. J. Cody, W. R. Cowell, B. S. Garbow, Y. Ikebe, C. B. Moler, and B. T. Smith, NATS, A Collaborative Effort to Certify and Disseminate Mathematical Software (Preprint). (Presented at the ACM National Conference, August 1972.)
2. W. J. Cody, The Evaluation of Mathematical Software (Preprint). (Presented at the ACM SIGPLAN Symposium on Computer Program Test Methods, June 1972.)
3. W. J. Cody, "Performance Testing of Function Subroutines," AFIPS Conf. Proc. 34 (1969 SJCC), 759-763.
4. S. D. Conte, Elementary Numerical Analysis: An Algorithmic Approach (McGraw-Hill, New York, 1965).
5. C. de Boor, "On Writing an Automatic Integration Algorithm," in the text [15], pp. 201-209.
6. L. D. Fosdick, "Algorithms Policy, Revised October 1971," Comm. ACM 14, 10 (Oct. 1971), 676.
7. F. N. Fritsch and R. F. Hausman, Jr., On the Documentation of Computer Programs, Lawrence Livermore Laboratory, Rept. UCID-30043, (March 1972).
8. C. W. Gear, "Experience and Problems with the Software for the Automatic Solution of Ordinary Differential Equations," in the text [15], pp. 211-227.
9. P. Henrici, Elements of Numerical Analysis (Wiley, New York, 1964).
10. T. E. Hull, "A Search for Optimum Methods for the Numerical Integration of Ordinary Differential Equations," SIAM Review 9, 4 (Oct. 1967), 647-654.
11. D. E. Knuth, "Fundamental Algorithms," Vol. 1 of The Art of Computer Programming, (Addison-Wesley, Reading, Mass., 1968).
12. H. Kuki and W. J. Cody, General Aspects of Program Certification for Numerical Subroutines, unpublished manuscript (ca. 1968).
13. H. Kuki, E. Hansen, J. M. Ortega, J. C. Butcher, and D. G. Anderson, Evaluation Guidelines, SHARE Numerical Analysis Project, SSD 150, Part II (1966), Item C4304.
14. J. R. Rice, "On the Construction of Polyalgorithms for Automatic Numerical Analysis," in Interactive Systems for Experimental Applied Mathematics, M. Klerer and J. Reinfelds, Eds. (Academic Press, New York, 1968), pp. 301-313.
15. J. R. Rice, Ed., Mathematical Software (Academic Press, New York, 1971).
16. J. R. Rice, "The Challenge for Mathematical Software," in the text [15], pp. 27-41.
17. J. F. Traub, "High Quality Portable Numerical Mathematics Software," in the text [15], pp. 131-139.
18. L. A. Zadeh, "Fuzzy Sets," Information and Control 8 (June 1965), 338-353.
19. L. A. Zadeh, "Fuzzy Algorithms," Information and Control 12, (Feb. 1968), 94-102.

APPENDIX A.
MATERIAL FROM THE SHARE EVALUATION GUIDELINES

The following material is quoted directly from Ref. [13]:

The three functions of the review write-up are (1) verification of claims made for the program, (2) clarification and supplementation of ambiguous or missing information, and (3) evaluation of the merits of the program.

A4.1 <u>Balanced appraisal</u>

Since a program which fails in one respect may still be very valuable in other respects, it is desired that the reviewer will look for merits as well as demerits. Merits are often relative to the requirements of a user's application.

A4.2 <u>Constructive criticism</u>

We believe in sharp criticism where due. On the other hand, if the fault found by the reviewer is correctable, it is desired that he will indicate the remedy in his review write-up. Similarly, if the program documentation is ambiguous or illogical, it is desired that the reviewer communicate with the author of the program and, if possible, clarify the confusion in his review write-up. This is better than simply stating that the documentation lacks clarity; and it is particularly helpful when a program's performance is good.

A4.3 <u>Comparison with other programs</u>

We wish to find out how the performance of a program compares with other SHARE (or non-SHARE) programs. More often than not, two similar programs do not show 100% overlap in purpose. A less general program is expected to be more efficient in carrying out its limited task. Also comparison of two programs often results in trade-offs such as size versus speed, speed versus accuracy etc. When distributing programs to reviewers for evaluation, the Manager of the Evaluation Project will attempt to assign as a set those SHARE programs which ought to be compared with each other. We also encourage comparisons with non-SHARE programs, and if such comparisons should favor a non-SHARE program, effort should be made to induce the author of the program to submit the same to SHARE. At any rate, mention should be made of the existence of such a program and how it can be obtained. Indeed, it has happened that for the sole purpose of conducting a thorough review of a SHARE Bessel function program, a reviewer wrote his own single precision and double precision Bessel function programs which turned out to be definitely preferable to the SHARE program being reviewed. Often the program to be reviewed happens to be the only one of the kind available. Though as such it is a useful addition to the SHARE library, even in this case, the reviewer will allow for future additions to the library, and he should design his test plan in a manner that would act as a challenge for better programs, should the existing program fall short of the current state of the art.

A4.4 Coverage and style of review write-up

The review write-up should be self-contained, written in a clear, concise, and easily digestible style. It is suggested that it consists of five parts: (1) a brief qualitative description of the overall lay-out of the comprehensive test plan, (2) a detailed description of the actual testing procedure, (3) a summary of the program's tested performance, (4) explanation, clarifications, complimentary remarks, or any criticism on items listed on the section A1 and A2, a description of difficulties in getting the program to run, etc., (5) a summarizing opinion by the reviewer including comparison with other programs of similar purpose.

APPENDIX B.
BIBLIOGRAPHY ON SUBROUTINE CERTIFICATION

The SIGNUM Subroutine Certification Committee is collecting a certification bibliography. Their list as of the middle of 1970 has been published in the SIGNUM Newsletter (see item 1, below).

The following, arranged in the same manner as the SIGNUM bibliography, provides additional references in this area. No claim of completeness is made for this list. In fact, it specifically excludes relevent papers in our text.

GENERAL PHILOSOPHY

1. SIGNUM Subroutine Certification Committee, "Certification Bibliography," SIGNUM Newsletter 4, 3 (Oct. 1969), 16-18; continued in SIGNUM Newsletter 5, 2 (Aug. 1970), 14-15.
2. R. L. Ashenhurst, "Evaluation and Certification Project at Argonne," SIGNUM Newsletter 4, 3 (Oct. 1969), 14.
3. J. M. Boyle, et al., NATS, A Collaborative Effort to Certify and Disseminate Mathematical Software (preprint); presented at the 1972 ACM National Conference.
4. W. J. Cody, "Performance Testing of Function Subroutines," Proc. Spring Joint Computer Conf. (1969), 759-763.
5. W. J. Cody, The Evaluation of Mathematical Software (preprint) (April 1972); presented at the Symposium on Computer Program Test Methods, June 1972.
6. F. N. Fritsch and R. F. Hausman, Jr., On the Documentation of Computer Programs, Report UCID-30043 (March 1972), Lawrence Livermore Laboratory.
7. W. M. Gentleman, "More on Publishing Programs," SICNUM Newsletter 3, 3 (Oct. 1968).

8. R. W. Hamming, Introduction to Applied Numerical Analysis (McGraw-Hill, New York, 1971); Chapter 15 (pp. 325-328) discusses the "Design of a Library."

9. O. G. Johnson, "IMSL's Ideas on Subroutine Library Problems," SIGNUM Newsletter 6, 3 (Nov. 1971), 10-12.

10. F. T. Krogh, "A Plea for Tolerance in the Evaluation of Numerical Methods and Mathematical Software," SIGNUM Newsletter 6, 3 (Nov. 1971), 7-8.

11. E. W. Ng, Mathematical Software Testing Activities at the Jet Propulsion Laboratory (preprint) (March 1972); presented at the Symposium on Computer Program Test Methods, June 1972.

12. K. A. Redish and W. Ward, "Environment Enquiries for Numerical Analysis," SIGNUM Newsletter 6, 1 (Jan 1971), 10-15.

13. G. M. Truszynski, Computer Program Documentation Guideline, Report NHB 2411.1 (July 1971), National Aeronautics and Space Administration.

14. K. H. Usow and L. D. Fosdick, "Guidelines for Evaluating an Algorithm for Publication," SIGNUM Newsletter 4, 3 (Oct. 1969), 19-20.

15. B. F. W. Witte, "Publication of Comprehensive Fortran Programs," SICNUM Newsletter 3, 1 (April 1968).

ARITHMETIC, ELEMENTARY FUNCTIONS, AND NUMBER THEORY

16. M. W. Cox, "UNIVAC Claims Super Accurate Fortran Math Library," SIGNUM Newsletter 6, 3 (Nov. 1971), 9.

17. K. E. Hillstrom, Performance Statistics for the Fortran IV (H) and PL/I (Version 5) Libraries in IBM OS/360 Release 18, Report ANL-7666 (August 1970), Argonne National Laboratory.

18. A. C. R. Newbery and A. P. Leigh, "Consistency Tests for Elementary Functions," Proc. Fall Joint Computer Conf. (1971), 419-422.

POLYNOMIALS AND SPECIAL FUNCTIONS

19. J. D. Lawrence, Comparison of Polynomial Root Finding Methods, CIC Note C2.2-A (Jan. 1966), Lawrence Livermore Laboratory.

20. E. W. Ng, "Certification of Algorithm 385, Exponential Integral Ei(x)," Comm. ACM 13, 7 (July 1970), 449.

QUADRATURE, DIFFERENTIAL AND INTEGRAL EQUATIONS

21. P. C. Crane and P. A. Fox, "A Comparative Study of Computer Programs for Integrating Differential Equations," Numerical Mathematics Computer Programs, Library One, vol. 2, issue 2 (Feb. 6, 1969), Bell Telephone Laboratories.

22. T. E. Hull, "A Search for Optimum Methods for the Numerical Integration of Ordinary Differential Equations," SIAM Rev. 9, 4 (Oct. 1967), 647-654.

23. K. E. Fitzgerald, "Error Estimates for the Solution of Linear Algebraic Systems," J. Res. Nat. Bur. Stds., Ser. B. Math. Sci. 74B, 4 (Oct.-Dec 1970), 251-310.

24. F. W. Luttman, Matrix Inversion Tests, CIC Report F1.2-004 (Sept. 1965), Lawrence Livermore Laboratory.