THE ENGINEERING OF NUMERICAL SOFTWARE

WEBB MILLER

THE ENGINEERING OF NUMERICAL SOFTWARE

WEBB MILLER

The University of Arizona

PRENTICE-HALL, INC.
Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging in Publication Data

Miller, Webb.
The engineering of numerical software.

Bibliography: p.
Includes index.
1. Numerical analysis—Computer programs. I. Title.
QA297.M527 1984 519.5 84-9830
ISBN 0-13-279043-2

Editorial / production supervision: Nancy Milnamow
Cover design:
Manufacturing buyer: Gordon Osbourne

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

0-13-279043-2

Prentice-Hall International, Inc., *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Whitehall Books Limited, *Wellington, New Zealand*

PREFACE

A number of important and interesting concepts about numerical software began to emerge in the 1970s: concepts that are independent of traditional numerical analysis and possess intrinsic merit and appeal. At the same time, experience gathered by publicly funded research projects and by the development of commercial software libraries raised understanding of the broad issues of numerical software production to the point that those concepts can be woven into a body of knowledge appropriate for systematic study.

This book is my attempt to make these advancements readily accessible. Whereas books such as *Computer Solution of Mathematical Problems* by George Forsythe, Michael Malcolm, and Cleve Moler (Prentice-Hall, Englewood Cliffs, N.J., 1977) teach the *use* of numerical software, much as introductory programming courses teach students to use a compiler without learning how to build one, my goal is to present principles for *writing* numerical software. The ideal textbook about the production of numerical software remains to be written, but I hope that I have verified its worth and feasibility and hastened its arrival.

The book is primarily intended for use as the text in a one-term course for students of computer science, engineering, science, and mathematics. It can also be used for self-study or for supplementary reading in a more traditional course on numerical computation. The material is accessible to readers with one year's programming experience, a minimum of training in mathematics, and the general level of academic maturity that can be expected of a senior undergraduate or first-year graduate student.

I believe that computer scientists will find this material useful, understandable, and even enjoyable. Educated guesses attribute as much as 50% of all software costs to numerical software, and it is appropriate that a study of computer science include the issues particular to that area.

Scientists, engineers, and mathematicians can also profit from exposure to this material. In some situations an awareness of the tremendous effort required to produce numerical software will lead them to use packaged software. In cases where software is not available and is worth the cost of development, a thorough understanding of the ingredients that go into successful software for several computational problems provides insight into what must be done for the problem at hand.

The topics covered, and their order of presentation, were chosen to proceed systematically across a spectrum of numerical software. There is, however, some flexibility in the order that the material is studied: Chapters 3–5 can be taken in any order once Chapters 1 and 2 have been read, while Chapter 6 can immediately follow Chapter 1.

The programming assignments and exercises are integral parts of the text. I recommend that the reader write all the programs and work all the exercises, except for material that is explicitly labeled as optional.

FORTRAN 77 programs are sprinkled throughout the book, so a reading knowledge of the language will be needed. Experience indicates that time will be saved in the long run if the reader invests whatever effort is needed to write the assigned programs in FORTRAN.

I would like to thank the faculty and students of the Department of Computer Science at the University of Arizona for providing the ideal environment for writing this book. Many insightful suggestions were offered by Tim Budd, Helen Deluga, Lee Henderson, James W. Johnson, and Titus Purdin.

Alan George of the University of Waterloo read the entire manuscript and recommended several improvements. My special thanks go to Jim Cody, Jack Dongarra, and James Lyness of the Argonne National Laboratory. Through their writings and direct comments they have contributed immeasurably to this book.

Webb Miller

CONTENTS

1

INTRODUCTION

The purpose of this chapter is to explain what the book is all about. We begin, in Section 1.1, by defining some key concepts concerning the process of developing numerical software.

Sections 1.2 and 1.3 illustrate these concepts in the contexts of solving a quadratic equation and checking the solvability of a nonlinear equation. In addition to clarifying our definitions, these examples provide an opportunity for us to point out reasons why the material covered in this book is important. In particular, Section 1.3 illustrates the difficulties associated with fixed-precision arithmetic and motivates our devoting an entire chapter, Chapter 2, to the subject.

Using the vocabulary developed in Sections 1.1–1.3, Section 1.4 outlines the common structure of Chapters 3–6 and explains the systematic differences among them.

1.1 HIGHLIGHTS OF THE SOFTWARE ENGINEERING PROCESS

It is useful to think of the process of producing numerical software as divided into four activities: (1) determining specifications, (2) designing and analyzing algorithms, (3) implementing software, and (4) testing and measuring programs. Of course, the production of an actual program often does not follow an orderly progression through these phases. (For instance, we will see that algorithm analysis may have to precede formulation of a specification.) In addition, we will classify programming mistakes into three loose categories: (1) blunders, (2) numerical oversights, and (3) portability oversights. In spite of

being vague, overlapping, and incomplete, this classification facilitates later discussions.

The meanings that we will attach to the terms used in these two categorizations and some related concepts will be explained in this section. In addition, we will describe a class of computational experiments that will be used to investigate the process of locating simple programming mistakes. The reader is urged to consider the definitions carefully but should keep in mind that our definitions are not standard.

Four Phases of Software Production. By a *specification* we mean a precise and complete statement of the intended relationship between a program's input and its computed output. One of the surprising things about numerical software is that for many computational problems it is not possible to determine an appropriate specification, in which case the software must be developed with only an imprecise idea of what it will actually do. Even when specifications exist, they may be difficult to determine. (See Exercise 1.)

An *algorithm* is an outline of the sequence of arithmetic operations that the software is to perform on "unexceptional" data. Much of the activity of designing algorithms is conceptual in nature. Often there are mathematical theorems to be understood and algebraic formulas to be manipulated. The process of algorithm design may include a theoretical analysis of the algorithm's resilience to the use of fixed-precision arithmetic.

The goal of the *implementation* phase of numerical software development is to produce a running computer program. Doing so involves determining the calling sequences, the handling of exceptional cases, the use of storage, etc.

For discussing the software development process, let us distinguish three purposes for running programs on a computer: testing, measuring, and experimenting. (We will not attempt to classify uses of the finished software.) A *test* is conducted on a specification and an associated computer program and is performed by running the program on one or more sets of input data and checking for conformance of the input and output to the specification. On the other hand, we will say that a program is being *measured* if either (1) we have no specification for input/output behavior or (2) we are checking some aspect of program performance other than input/output behavior (for instance, we might be determining efficiency). The act of running a program for the stated purpose of investigating some aspects of the software engineering process (rather than to develop a particular program) is an *experiment*. We will require that an experiment involve an explicit *hypothesis* that makes an assertion about the program development process, just as testing requires a specification.

For example, suppose we were writing sorting programs, that is, programs to satisfy the specification of arranging an array of numbers into ascending order. Running a program to see if it correctly sorts its input would be considered testing. Seeing which of two programs runs faster on a particular

set of data would be classified as measuring. We might also formulate the hypothesis that our approach to testing will catch over 90% of all potential programming bugs and then conduct an experiment by running a collection of bug-ridden sorting programs through our testing process and seeing how many are caught.

Three Kinds of Software Mistakes. A program *error* is a specification-program pair with the property that under certain conditions of input and execution environment the program's output does not meet the specification. Thus the term will not be applicable to a program having no specification. (We will also speak of the "error" in a number, meaning the difference between an approximate number and the true value, but it will be easy to distinguish these two uses.) A *mistake* is a program construct that should be changed because it is contrary to the programmer's intent. Thus an error need not be a mistake (since it might only signal that the specification is not appropriate), and a mistake need not be an error (since it might affect efficiency instead of input/output behavior).

We will classify mistakes according to how they are understood. A *blunder* can be explained in terms of an idealized model of computation, e.g., one which assumes that arithmetic is exact and that FORTRAN programs are executed directly by the computer hardware. For instance, we classify the accident of replacing a + by −, or of writing INTGER instead of INTEGER, as a blunder. To understand an *oversight*, on the other hand, requires a grasp of certain details of actual computers, for instance, specifics about arithmetic units or FORTRAN compilers. In particular, a *numerical oversight* is a mistake that hinges on differences between ideal arithmetic and arithmetic as performed by computers. A *portability oversight* is a mistake because of differences, in hardware or software, among computer systems. See Exercises 3 and 4 for examples of portability oversights.

Experiments about Typographical Mistakes. The following changes to a program statement are defined to be *typographical changes:*

1. Replace the value of a binary arithmetic operation by one of the operands
2. Replace one of the arithmetic operators +, −, *, /, or ** by another
3. Replace one of the relational operators $=$, $\neq$, $<$, $\leq$, $>$, $\geq$ by another
4. Add 1 to a constant or subtract 1 from a constant
5. Replace one occurrence of a scalar (i.e., nonsubscripted variable) by another scalar that appears in the program

For instance, the following statements can be derived from the FORTRAN statement

```
IF (X+Y .GT. 1.0) Z=0.0
```

by making a typographical change of the indicated type:

Statement	Type of Change
`IF (X .GT. 1.0) Z=0.0`	1
`IF (X-Y .GT. 1.0) Z=0.0`	2
`IF (X+Y .LT. 1.0) Z=0.0`	3
`IF (X+Y .GT. 2.0) Z=0.0`	4
`IF (X+Y .GT. 1.0) X=0.0`	5

A *mutant* of a given program is another program that can be derived by making a single typographical change to a statement of the original program. A *mutation experiment* is conducted using the following objects:

1. A collection of one or more programs for a certain computational problem
2. A collection of one or more sets of data for that problem
3. Rules for deciding if a computed solution for that problem is *acceptable*

The mutation experiment consists of (1) generating all mutants of the given procedures, (2) executing every mutant on every one of the given sets of data, and (3) determining which (or how many) mutants "survive." A mutant is said to *survive* the experiment if for each of the given sets of data it (a) performs a valid computation (does not divide by zero, generate an out-of-bounds subscript, reference an undefined variable, etc.) and (b) produces an acceptable answer. In brief, the survival of a mutant indicates that the given collection of test data sets is inadequate to distinguish the mutant from the original program.

The software system described in "Mutation Analysis: Ideas, Examples, Problems and Prospects" by T. Budd (in *Software Testing*, edited by B. Chandrasakeran and S. Radicchi, North-Holland, Amsterdam, 1981) almost completely automates the process of conducting mutation experiments on FORTRAN programs. One of the few tasks that must be performed manually is determining which of the surviving mutants are equivalent to the original program, i.e., which of the typographical changes are not mistakes. (For instance, changing an occurrence of $\geq$ to $>$ might not affect the program's quality.) Usually this task is quite simple.

The mutation analysis tool used in experiments described throughout the book was originally developed as an aid for a particular approach to systematic

generation of test data in a production environment. The survival of a mutant would be taken as indicating a weakness in the test cases that should be remedied before the program could be considered well tested. Because of the cost (in terms of computer resources) of using this tool, it has not been widely applied in that context. Here we are using the mutation tool as an experimental apparatus to explore both the effectiveness of various test data generation methods and the difficulty of adequately testing various kinds of programs for various kinds of errors. In this laboratory setting the tool has proven to be invaluable.

EXERCISES

1. Consider the following FORTRAN 77 *random number generator*:

```
FUNCTION RAN()
SAVE K
DATA K / 100001 /
K = MOD(K*125,2796203)
RAN = REAL(K) / 2796203.0
RETURN
END
```

[*MOD(K * 125,2796203)* is the remainder after division of *K * 125* by *2796203*.]

(a) Show that any value returned by *RAN* lies between 0 and 1, given certain assumptions about the number of bits in a FORTRAN integer.

(b) Does the statement that *RAN* returns a value between 0 and 1 constitute a specification in the sense discussed in this section? Explain.

(c) Give an informal, yet plausible, condition that the sequence of numbers generated by

```
1  PRINT *, RAN()
   GO TO 1
```

should meet to exhibit "randomness." Execute *RAN* to see whether it meets this condition.

(d) Can you give a specification (that is, a precise and complete statement of requirements) that a random number generator should satisfy? What test would determine whether or not *RAN* is correct in this sense? See *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, by Donald E. Knuth (Addison-Wesley, Reading, Mass., 1981) for a discussion of the subtleties of random number generators.

(e) Suppose that a mistake is made when typing *RAN*, causing 100001 to be replaced by 10001. Would that necessarily be considered an error in the sense used here?

2. **(a)** Let P be a program containing p binary arithmetic operations, q occurrences of relational operators, r occurrences of constants, s occurrences of scalars, and t distinct scalars. Show that P has $6p + 5q + 2r + (t - 1)s$ mutants.
   (b) Show that the following program has 185 mutants.

$$d \leftarrow b^2 - 4c$$
$$\text{if } d \geq 0$$
$$e \leftarrow \sqrt{d}$$
$$r \leftarrow \frac{-b + e}{2}$$
$$s \leftarrow \frac{-b - e}{2}$$
$$\text{else}$$
$$p \leftarrow \frac{-b}{2}$$
$$q \leftarrow \frac{\sqrt{-d}}{2}$$

3. (Optional) The 1966 FORTRAN standard did not include the *SAVE* statement, and FORTRAN compilers differed in the ways they handled *DATA* statements. Specifically, in some environments a variable in a *DATA* statement was reinitialized every time the procedure was entered; in other environments the variable's value at a procedure invocation was its value at termination of the preceding invocation. Discuss the repercussions for the procedure *RAN* of Exercise 1 with the *SAVE* statement removed.
4. (Optional) FORTRAN programs are collections of external subprograms that share data through procedure arguments or *COMMON* (that is, global) data areas. Some FORTRAN compilers handle unsubscripted variables with "call by reference" (i.e., by passing the addresses of procedure arguments), while others use "copy-in, copy-out" (i.e., upon entry to a subprogram, an initial value for the argument is copied into a temporary location to be used throughout execution of the subprogram; upon return, the final contents of the temporary are copied back into the actual argument). This difference among FORTRAN compilers results in two situations where a program's output depends on the compiler's implementation. One case occurs when a variable appears more than once as an actual argument in a procedure reference, as in *CALL THUD(X, X)*. The other case is when an actual argument passed to a subprogram is in a *COMMON* area to which the called subprogram has access. Give programs that illustrate each of these cases. In each case, specify a set of data and the differing outputs.

1.2 AN EXAMPLE OF A TEST AND TWO EXPERIMENTS

In this section we will illustrate the notions of *specification, algorithm, implementation, test, hypothesis, error, numerical oversight, experiment*, and *mutation*

experiment as they arise during development of programs to solve quadratic equations of the form

$$x^2 + bx + c = 0$$

In addition, the following general points about numerical software are illustrated by this example:

1. It may require some effort to determine an appropriate specification.
2. It may be difficult in practice to decide whether the output from a test with a particular set of input data has adhered to a given specification.

A computer can represent numbers to only a limited precision and in general cannot represent the exact solutions of $x^2 + bx + c = 0$ even in cases where b and c are representable. With this limitation in mind, we might propose Specification 1.1a.

Specification 1.1a
The computed solutions of $x^2 + bx + c = 0$ should agree to within the computer's precision with the true solutions.

However, Specification 1.1a is not particularly useful because, as it turns out, no program can satisfy it (unless extra precision is used in the computation). It is more helpful to work with the following specification, which is not the sort of assertion that one is likely to think of right away:

Specification 1.1b
The computed solutions of $x^2 + bx + c = 0$ should agree to within the computer's precision with the true solutions of $x^2 + bx + C = 0$, where C is some number that agrees with c to within the computer's precision.

Contemplate Specification 1.1b awhile and work Exercise 1. We think that you will come to agree with point 1 above.

The mathematical theory that underlies our quadratic equation procedures consists of the following fact:

Theorem 1.1
If b and c are real numbers, then the equation $x^2 + bx + c = 0$ is satisfied by exactly two (possibly identical) numbers, namely

$$\frac{-b \pm e}{2}$$

where $e = \sqrt{d}$ and $d = b^2 - 4c$. In particular, if $b^2 - 4c < 0$, then the two solutions are the complex numbers

$$p \pm i \times q,$$

where $p = -b/2$, $q = \sqrt{-d}/2$, and $i^2 = -1$.

This fact immediately suggests Algorithm 1.1a:

Algorithm 1.1a

$d \leftarrow b^2 - 4c$

if $d \geq 0$

$e \leftarrow \sqrt{d}$

$r_1 \leftarrow \dfrac{-b + e}{2}$

$r_2 \leftarrow \dfrac{-b - e}{2}$

else

$p \leftarrow \dfrac{-b}{2}$

$q \leftarrow \dfrac{\sqrt{-d}}{2}$

It turns out that Algorithm 1.1a is not as accurate as it should be (see Exercise 1), which illustrates the fact that straightforward embodiment of mathematically correct formulas often results in a poor computer program.

The following variant of Algorithm 1.1a is *correct*, in the sense that it satisfies Specification 1.1b. The modified procedure computes one solution, call it r_1, by a formula that can be guaranteed to be accurate. It follows from basic algebra that the other solution is c/r_1. This approach is followed in Algorithm 1.1b.

Implementing Algorithm 1.1b requires, among other things, that a decision be made about how the calling program is to be notified when the roots are complex. The quadratic equation solver might always return two complex numbers, perhaps having imaginary parts equal to zero, or it might always return two real numbers, p and q, and an indication of whether the roots are p and q or are the complex conjugate pair $p \pm i \times q$.

To *test* whether an implementation of Algorithm 1.1b satisfies Specification 1.1b, we might generate, by some means, a sequence (b_1, c_1), $(b_2, c_2), \ldots, (b_k, c_k)$ of sets of test data. For each set we could apply the quadratic equation solver and check the specification. But how is this check to be made? In other words, given a set b, c of data and computed solutions r_1 and r_2, how can you tell whether Specification 1.1b is satisfied? Not only is it hard to see how this check might be performed in exact arithmetic, but the check is itself a numerical computation that is subject to the same implementation annoyances and susceptibility to arithmetic inaccuracies that plague algorithms. Even designing a test for Specification 1.1a, which is appreciably

Algorithm 1.1b

$d \leftarrow b^2 - 4c$

if $d \geq 0$

$e \leftarrow \sqrt{d}$

if $b \geq 0$

$r_1 \leftarrow \dfrac{-b - e}{2}$

else

$r_1 \leftarrow \dfrac{-b + e}{2}$

$r_2 \leftarrow \dfrac{c}{r_1}$

else

$p \leftarrow \dfrac{-b}{2}$

$q \leftarrow \dfrac{\sqrt{-d}}{2}$

easier to check than is Specification 1.1b, requires some thought (see Exercise 2). Here, as is often the case, the intellectual effort required to produce a reliable program to check that the computed solution meets the specification may well equal or exceed that required to understand and implement the algorithm.

Experience indicates that even extensive testing can fail to find program mistakes caused by use of inexact arithmetic. This observation might lead us to formulate the following experimental hypothesis:

Hypothesis 1.1a
Errors resulting from numerical oversights in procedures to solve quadratic equations are hard to uncover through testing.

To conduct an *experiment* to investigate Hypothesis 1.1a we might see how many sets of test data are needed to expose some known *numerical oversights*. For instance, we might use any of the following *errors*:

1. Specification 1.1a and an implementation of Algorithm 1.1a,
2. Specification 1.1a and an implementation of Algorithm 1.1b, or
3. Specification 1.1b and an implementation of Algorithm 1.1a.

(Remember, we are using the term *error* in a special way; an error consists of a specification and a nonconforming program.) Such an experiment differs from testing because, among other things, it requires the use of known program errors, so the specifications being investigated need not be satisfiable and the quadratic equation solvers being executed need not be candidates for general use.

Our experiment might repeat the following process for each of the three errors: Apply the given algorithm to each of 100 "random" sets (b_1, c_1), $\ldots, (b_{100}, c_{100})$ of data and check whether the computed results satisfy the given specification. If, say, only 1% of the sets of data reveal the error (in the sense that the corresponding computed solutions do not meet the specification), then the experiment could be interpreted as supporting Hypothesis 1.1a. (As it turns out, the problem of solving a quadratic equation is easy enough that a simple theoretical analysis determines which sets b, c of data can expose these errors. See Example 1 of Section 2.4 for such a theoretical corroboration of Hypothesis 1.1a.)

Many potential programming mistakes in quadratic equation procedures, especially mistakes that are not related to the use of inexact arithmetic, can be detected by even the most naive approach to program development. The following specific claim can be subjected to experimentation.

Hypothesis 1.1b
To expose almost any typographical mistake in a quadratic equation solver, it is sufficient to apply the procedure to one quadratic with real roots and one quadratic with complex roots.

A *mutation experiment* to investigate Hypothesis 1.1b might involve (1) a FORTRAN implementation of Algorithm 1.1a, (2) the quadratics $x^2 - 5x + 6$ and $x^2 + 4x + 5$, and (3) the acceptance criterion that the errors in the two computed values not exceed 0.0001. This experiment will determine that 8 of the 185 typographical changes will not be exposed by either set of data. (See Exercise 2 of the previous section and Exercise 3, following.) At that point one has such options as claiming that the experiment supports Hypothesis 1.1b, reformulating or discarding Hypothesis 1.1b, designing a mutation experiment with a different pair of quadratics (when you solve Exercise 3, you will see why using small integers for test data is not a good idea), etc.

EXERCISES

1. Imagine a hypothetical computer that carries six digits of precision. Thus, if the computer attempts to add 1.23456 and 0.111111, the best it can do is to produce 1.34567 since it cannot represent 1.345671. Numbers are represented in "scientific notation," e.g., $10^6 \times 0.123456$ or $-10^{-3} \times 0.111111$, so the restriction to six digits of precision does not limit the size of the numbers that can be computed.