Software Manual for the Elementary Functions

WILLIAM J. CODY, JR.
WILLIAM WAITE

PRENTICE-HALL SERIES IN COMPUTATIONAL MATHEMATICS
Cleve Moler, Advisor

SOFTWARE MANUAL FOR THE ELEMENTARY FUNCTIONS

WILLIAM J. CODY, JR.
Argonne National Laboratory
and
WILLIAM WAITE
Department of Electrical Engineering
University of Colorado

Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging in Publication Data
Cody, William James
Software manual for the elementary functions.

(Prentice-Hall series in computational mathematics)
Bibliography: p.
1. Functions--Data processing. I. Waite, William McCastline, joint author. II. Title. III. Series.
QA331.C635 1980 519.5 80-14411
ISBN 0-13-822064-6

Prentice-Hall Series in Computational Mathematics
Cleve Moler, Advisor

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Prentice-Hall International, Inc., *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall of Canada, Ltd., *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*

To our two Joannes

PREFACE

This manual is intended to provide guidance towards the preparation and testing of elementary function subroutines for non-vector oriented digital computers. We believe it will be useful to systems programmers, teachers, students of numerical analysis, hobbyists and anyone else concerned or curious about how the elementary functions might be computed. The functions covered are the usual assortment of algebraic, trigonometric, and transcendental functions of real argument, including those required by algebraic languages such as Fortran and Basic. Where programs are included, they are written in Fortran.

Fortran also influences the terminology and notation used: We frequently use the symbol * to mean multiplication, use ** to represent exponentiation, and talk about the SQRT program, for example. But the ideas, algorithms, and programs presented are more widely applicable. Many of the algorithms and accompanying test programs have been implemented in Basic and PL/I by the authors and their students, and some have even been implemented on programmable hand calculators. Nevertheless, the authors make no warranty of any kind with regard to the material in this manual. It is the reader's responsibility to verify that the material is both correct and appropriate for his intended usage.

A magnetic tape containing the Fortran source code for all of the test programs, the random number generators and the environmental inquiry program in this manual are available from either of two sources:

National Energy Software Center
Argonne National Laboratory
9700 South Cass Avenue
Argonne, Illinois 60439
(Phone: 312-972-7250)

International Mathematical and Statistical
Libraries, Inc.
Sixth Floor, GNB Building
7500 Bellaire Boulevard
Houston, Texas 77036
(Phone: 713-772-1927)

Information regarding distribution charges and tape formats can be obtained directly from either source.

Almost the entire text of this manual has been prepared on a computer. While this process has simplified the writing and proofing of our work, it has also introduced limitations on the use of mathematical symbols and notation. Subscripts, for example, had to be inserted by hand and therefore were avoided wherever possible. We apologize for any resulting awkwardness in our presentation.

We owe much to W. Kahan and the late Hirondo Kuki who were especially influential over the years in molding some of the ideas presented here. We are grateful to Argonne National Laboratory for its support of this project and to our colleagues there who contributed so much. We especially thank B. Garbow who patiently read and commented on several versions of the manuscript, R. Clark and W. Tippie who provided special graphics programs for typesetting and flow charting, S. Pieper who developed important enhancements to the text editing program which facilitated the photo typesetting, and G. Pieper who prepared our computer text files for typesetting and otherwise assisted editorially. Finally, we thank the many colleagues and students around the world who tried various versions of our algorithms and provided useful criticism.

CONTENTS

1. INTRODUCTION

The proliferation of mini- and micro-computers has involved ever larger numbers of people in the creation of basic software. Subprograms for the elementary functions are an essential part of such software in a wide variety of applications. These subprograms are often the basic building blocks of an application and as such must be efficient and accurate. The algorithms chosen for them must exploit the hardware of the particular target computer to achieve these goals.

Many pitfalls await a systems programmer who attempts to implement basic function routines using information gleaned from a calculus text. The treatment by Hart and his co-authors [1968] provides some guidance, but its theoretical orientation makes it difficult for the non-mathematician to use, and it ignores many algorithmic details dictated by considerations of computer architecture. This manual is designed as a "cookbook," containing specific "recipes" for the preparation of software for the elementary functions and information on testing procedures. Some material presented here is new, particularly the coefficients used in some of the algorithms and the testing material. Many of the techniques suggested for implementing the algorithms are old enough to be considered folklore by some numerical analysts, but they are gathered together in one place for the first time.

Our intent is to make this work useful to as wide an audience as possible, but especially to systems programmers not familiar with numerical analysis or numerical programming. To this end we have made comments in the implementation notes which our numerically oriented colleagues may find trite, but which our experience indicates the non-numerically oriented will find useful. The technical discussions have also been kept simple, with references suggested for those desiring more detailed discussion.

Chapter 2 of this work discusses the general principles behind the choice of algorithms and the assumptions made about the computer hardware and software environments in which these algorithms are to be implemented. Chapter 3 discusses techniques for testing the accuracy of elementary function subprograms in general, and for testing in the environments described in Chapter 2 in particular. Chapters 4 through 13 contain the recommended algorithms. The discussion for each function includes general comments about the algorithm, a flow chart, detailed implementation notes for several different machine environments, a discussion of performance testing, and a Fortran program to provide minimal testing. Finally, there is a glossary of important concepts and terms, as well as appendices containing certain Fortran subroutines required by the test programs.

Each algorithm has been implemented as a parameterized body of Janus (Haddon and Waite [1978]) text. By setting specific parameters and processing this text with the general-purpose macro processor STAGE2 (Waite [1973]), a Janus program tailored to a particular machine is obtained. We have tested the Janus implementations on a number of machines with various parameter settings to validate the algorithms. While the results are not always as good as the best individual efforts achieved by programming in assembly language and choosing algorithms that exploit specific hardware features of a particular computer, they do demonstrate that these algorithms often come acceptably close.

2. PRELIMINARIES

For expository purposes we will often classify computers according to pertinent arithmetic characteristics. Efficient implementation of our algorithms will then require a detailed knowledge of the particular arithmetic system being used, where the term arithmetic system refers to the combination of the hardware or software for the arithmetic operations and the scheme for representing numbers in the machine. Because there is a great variation in arithmetic systems, we can discuss them only in general terms here, but the important details of a specific arithmetic system can usually be determined from the manual describing the instruction set for the particular machine.

We assume every computer can perform the four basic arithmetic operations for integers because this capability is essential for indexing and addressing. Integer arithmetic is exact with two exceptions. First, there is a largest integer which can be accommodated in the representation scheme, placing a practical bound on the magnitude of integers that can be stored in the machine. If we assume that at most n-digit integers can be stored, then the product of two such integers is too large to be stored as an integer. On many machines the full double-length integer product is developed in the arithmetic registers, with the least significant n digits of the product stored as the result and an *overflow* error indicated if the product exceeds n digits. Overflow is also possible in addition and subtraction provided the integers involved are large enough. Unless overflow occurs, however, the result of adding, subtracting or multiplying two integers is exact and again an integer.

The second exception to exact integer arithmetic is division. The exact result of integer division is an integer plus a proper fraction as a remainder. Similar to multiplication, many machines develop both the integer and the fraction (appropriately rounded or truncated) in the arithmetic registers, but the fraction is ignored in storing the result.

The possibilities for scientific computation using only integer arithmetic are limited, primarily because scientific computation is dominated by non-integer quantities. It is possible but awkward to use integer arithmetic on such numbers. For example, if the computer uses decimal integers but data must be represented to hundredths, then scaling by one hundred gives integer quantities that can be combined by addition and subtraction without error. However, the product of two such numbers contains four places after the decimal point and must be rescaled before being used further. Similarly, unless the dividend is rescaled before division, the quotient will not be an integer.

Integer arithmetic with an implied scale factor is generically called fixed-point arithmetic, although we will reserve that term in a moment to denote a specific scaling. Two extremes of scaling are possible: no scaling, which implies pure integer arithmetic with the "decimal point" at the extreme right of the digits, and "total" scaling with the "decimal point" at the extreme left of the digits. This latter case, in which the stored number represents a proper fraction, a quantity less than one in magnitude, is what we will call *fixed-point* arithmetic (Ralston [1976]). The main differences that distinguish fixed-point arithmetic from integer arithmetic are as follows:

a) every stored number is either zero or a proper fraction;

b) in multiplication, the most significant n digits of the double-length product are retained, not the least significant. Thus overflow is impossible, but *underflow* (a result too small, hence indistinguishable from zero) is possible;

c) the fractional part of the quotient is retained instead of the integer part, and overflow is signaled if the integer part is non-zero. Thus overflow occurs in division unless the divisor is greater in magnitude than the dividend.

Many computers extend the instruction set for integer arithmetic to accommodate fixed-point arithmetic by adding fractional multiply and divide instructions. This procedure is not difficult when double-length integer products and quotients are already available. The add and subtract operations are identical to the integer operations.

Scientific computation is facilitated by representing numbers in a pseudo-scientific notation and using fixed-point arithmetic. Thus a number x is represented as

$$x = \pm f * B^{**}e, \quad 0 \leq f < 1,$$

where f is the fractional part of the number, B is the base for the scaling, and e is the exponent. Such a number is stored in two parts, the fraction and an integer representing the exponent. The base B, which is 10 in normal scientific notation, is usually compatible with the internal number system for the computer. Thus B will ordinarily be 10 on a decimal computer but will be 2, 4, 8, or 16 on a machine that works primarily in binary. In any case, the value of B is understood and not explicitly stored.

Multiplication and division in this system are simple, involving fixed-point multiplication or division of the fractional parts of the numbers, addition or subtraction of the exponents, and appropriate prescaling to avoid overflow in division. Addition and subtraction become complicated, however, requiring appropriate prescaling of the operands to align them and to avoid overflow.

It is only natural that this scheme should be implemented as a separate arithmetic system, called a *floating-point* system, and supported by the instruction set on many computers. The additional instructions may appear as hardware instructions, or they may invoke software subprograms. In either case, the floating-point representation usually differs slightly from the representation just presented. In particular, we can think of a number x as being represented in *normalized* form by

$$x = \pm f * B^{**}e, \quad 1/B \leq f < 1,$$

unless $x = 0$, in which case $f = 0$ and the value of e varies with the implementation. Normalization refers to the lower bound on f, which is now called the *significand*. B is the *radix* of the representation, and e is the exponent. In order to make our algorithms independent of the actual representation of x, we state them in terms of the components f, B, and e. For example, we define functions below which access the f and e components of a floating-point number. These functions hide details of the representation; their values depend only upon the values of x and B and upon the relationship stated above.

The arithmetic performed on floating-point numbers varies from one machine to another in two characteristics that will affect the performance of our algorithms. These are the method of fitting

overlength intermediate results back to working precision, and the availability of guard digits. In an n-digit floating-point arithmetic system, more than n digits are frequently required to represent the true result of an arithmetic operation, but only the n most significant digits can be retained in the machine. In the *chop* mode of rounding, any extra digits in the true result are ignored, while in a *round* mode the retained significand is rounded up or down, depending in various ways upon the magnitude of the digits to be discarded.

Often at an intermediate stage of an arithmetic operation the significand of the result requires a *renormalization shift* of one or more digit positions to the left to compensate for loss of leading digits. If the arithmetic operations generate only the first n digits of the intermediate result, counting possible leading zeros, then there are no extra digits to shift into the low-order positions during renormalization, and we say there are no *guard digits.* However, if extra digits are generated and participate in renormalization shifts, thus protecting the low-order positions, we say there are guard digits. Some arithmetic operations on a machine may have guard digits, while others on the same machine may not. Lack of guard digits for addition/subtraction may cause inaccuracies when subtracting numbers slightly less than an integer power of the radix from numbers slightly larger than that power of the radix, while lack of guard digits for multiplication may mean that $1.0 * x \neq x$.

We can now distinguish several broad classes of machines. We will always assume the existence of some form of floating-point arithmetic. We will also assume that the radix B is either 10 or a small integral power of 2, i.e., B is 2, 4, 8, 10 or 16. Thus we will classify machines as binary ($B = 2$), decimal ($B = 10$), non-decimal ($B \neq 10$), etc. We can also distinguish between fixed-point and floating-point machines by classifying a machine as fixed-point whenever its floating-point operations are extremely slow in comparison to its fixed-point operations. Typically, the floating-point instructions are implemented by software in this case. To be useful, we further assume that a fixed-point machine is a non-decimal machine and that the number of bits in a fixed-point (fractional) number is at least as great as the number of bits in the floating-point significand. In certain explicitly noted cases we will assume that there are more bits in the fixed-point representation than in the floating-point significand (e.g., see the implementation notes for SQRT).

On non-decimal floating-point machines we will assume that, say, b bits are available for the representation of the significand. If $B = 2$, normalization requires that $1/2 \leq f < 1$ unless $f = 0$, and the left-most bit of f is 1. Thus all b bits are potentially significant. If $B = 4$, however, it is possible that $1/4 \leq f < 1/2$ so that the left-most bit of f is 0, in which case only b-1 of the available bits are potentially significant. Similarly, if $B = 16$, there may be as many as three leading zero bits and thus only b-3 potentially significant bits in f. This phenomenon of loss of potentially significant bits in the representation of f for $B = 4$, 8 or 16, which is due entirely to the accidental magnitude of x, is termed *wobbling precision.* As an example of its impact, in hexadecimal arithmetic ($B = 16$) the significand of the constant 2/pi contains no leading zero bits while that for the constant pi/2 contains three leading zero bits. Thus division by 2/pi is potentially one decimal place more accurate than multiplication by pi/2 in this system.

A more subtle form of wobbling precision is present in all floating-point arithmetic systems. Consider two machine numbers $X = (1+eps) * B**(n)$ and $Y = (1-eps) * B**(n)$, where eps is a small positive quantity. The exponent in the floating-point representation of X is one greater than the exponent in the representation of Y, even though X and Y are almost equal. If u represents one unit in the last digit of the significand of X and v represents one unit in the last digit of the significand of Y, then for all practical purposes

```
(u/X) = B*(v/Y),
```

and u is a larger fraction of X than v is of Y. This has implications when we discuss relative error in the next chapter.

The algorithms presented in Chapters 4 through 13 are each accompanied by implementation notes for specific broad classes of machines. These notes describe details of the implementation consistent with the characteristics of the target arithmetic systems. The notes for fixed-point machines, for example, incorporate proper scaling (even of the given coefficients) to avoid fixed-point overflow and to maintain maximum precision in the computed result. Similarly, computations are organized to minimize the effects of wobbling precision on machines where that can be important. We have tried to indicate why some of the suggestions are made, but we have not tried to provide background for every suggestion because to do so would turn our "cookbook" of "recipes" into a ponderous text on practical numerical analysis.