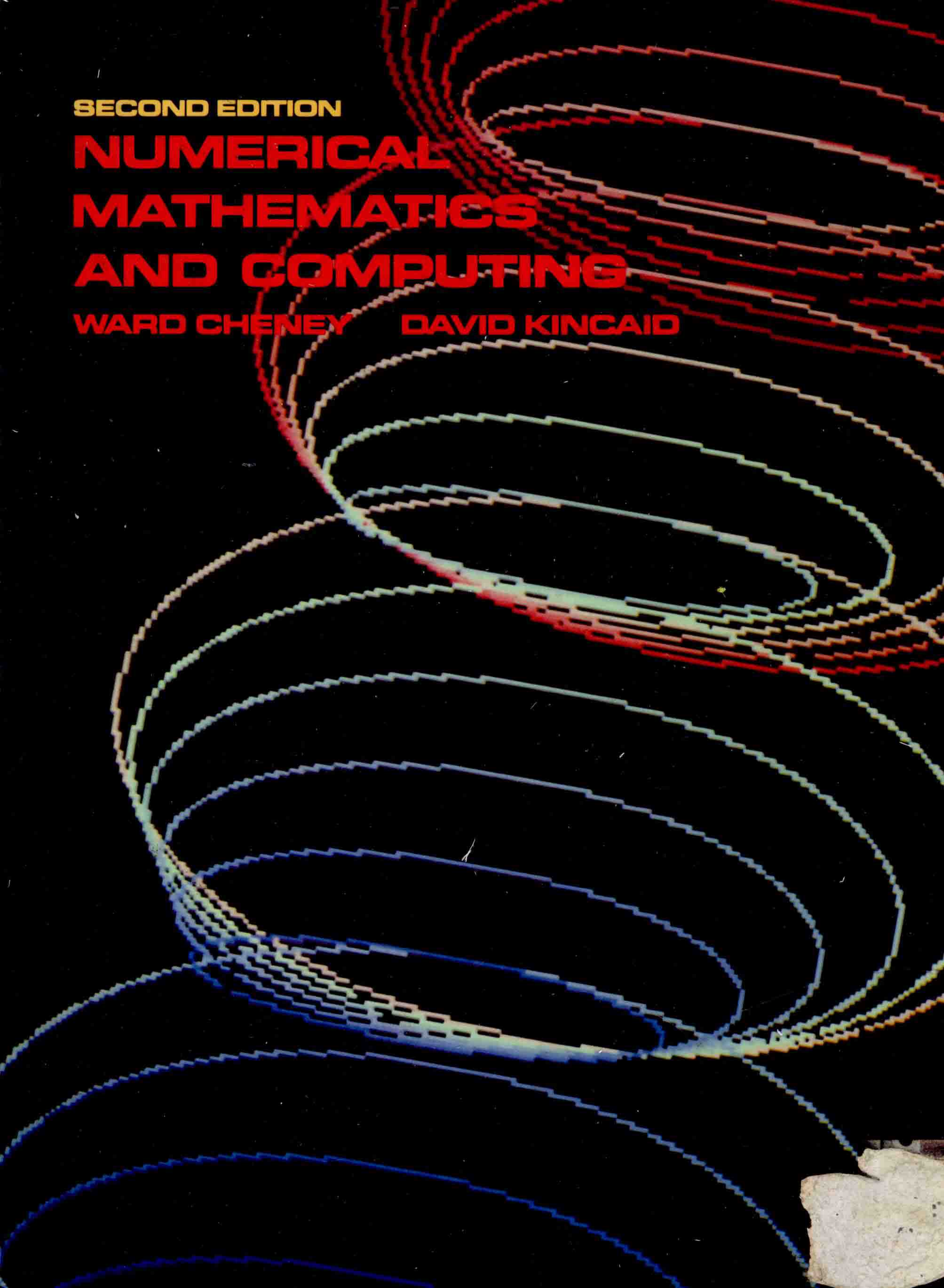
SECOND EDITION

NUMERICAL MATHEMATICS AND COMPUTING

WARD CHENEY DAVID KINCAID



NUMERICAL MATHEMATICS AND COMPUTING

Second Edition

WARD CHENEY
DAVID KINCAID
The University of Texas at Austin

Brooks / Cole Publishing Company
Pacific Grove, California

Brooks/Cole Publishing Company
A Division of Wadsworth, Inc.

Printed in the United States of America

10 9 8 7 6 5

Library of Congress Cataloging in Publication Data

Cheney, E. W. (Elliott Ward), [date]
Numerical mathematics and computing.
Bibliography: p.
Includes index.
1. Numerical analysis—Data processing. 2. FORTRAN
(Computer program language) I. Kincaid, David (David Ronald)
II. Title.
QA297.C426 1985 519.4 84-27420

ISBN 0-534-04356-9

Sponsoring Editor: *Craig Barth*
Production Editor: *Candyce Cameron*
Production Associate: *Dorothy Bell*
Editorial Assistant: *Eileen Galligan*
Manuscript Editors: *Paul Monsour, Charles Hibbard*
Interior and Cover Design: *Vernon T. Boes*
Cover Photo: *Digital Image Corporation*
Art Coordinator: *Judith Macdonald*
Interior Illustration: *Lori Heckelman*
Typesetting: *Science Typographers, Long Island, New York*
Cover Printing: *Phoenix Color Corporation, Long Island City, New York*
Printing and Binding: *The Maple-Vail Book Manufacturing Group, York, Pennsylvania*

To the memory of ROBERT TODD GREGORY (1920–1984)

We wish to dedicate this book to the memory of Professor Robert Todd Gregory, who made significant contributions to the development of modern numerical analysis. He was our friend and colleague for many years.

PREFACE

In preparing the second edition of this book, we have adhered to the basic objective of the first edition—namely, to acquaint students of science and engineering with the potentialities of the modern computer for solving the numerical problems that will arise in their professions. A secondary objective is to give students an opportunity to hone their skills in programming and in problem solving. A third objective is to help students arrive at an understanding of the important subject of *errors* that inevitably accompany scientific computing, and to arm them with methods for detecting, predicting, and controlling these errors.

Since the book is to be accessible to students who are not necessarily very advanced in their formal study of mathematics, we have tried to achieve an elementary style of presentation, including numerous examples and fragments of computer code for illustrative purposes. Believing that most students at this level need a *survey* of the subject of numerical mathematics, we have presented a wide diversity of topics, including some rather advanced ones that play an important role in current scientific computing.

Features in the Second Edition. Features that are new to this second edition are as follows:

- Many more examples of worked problems have been included, especially in the earlier chapters where presumably they are most helpful.
- The computer programs in the book have been revised to illustrate the structured style available in Fortran 77 now favored by many programmers.
- New topics, such as Gaussian quadrature, adaptive Simpson's integration, and discrete orthogonal polynomials, have been added.
- Many problem sets throughout the book have been revised and reorganized in an attempt to group the problems according to subject matter and difficulty.
- Important topics formerly developed in the problems have been incorporated into the text instead.
- The chapters have been reordered for logical and pedagogical reasons.
- The discussion of floating-point number representation has been revised and simplified.
- Richardson extrapolation has been given greater prominence and emphasis.
- More material on the **LU** factorization has been added in the chapter on solving linear systems of equations.

- A new appendix on the concepts and notation of basic linear algebra has been included.
- The appendix on Fortran has been expanded and now emphasizes Fortran 77.
- Many more answers and hints to problems have been supplied.

Suggestions for Use. *Numerical Mathematics and Computing, Second Edition* can be used in a variety of ways, depending on the emphasis the instructor prefers. Problems have been supplied in abundance to enhance the book's versatility. They have been divided into two categories: "Problems" and "Computer Problems." In the first category there are more than 800 exercises in analysis that require pencil, paper, and possibly a calculator. In the second category, there are approximately 450 problems that involve writing a program and testing it on a computer. The Fortran 77 code displayed in this text is available on a floppy disk from the publisher. Also available is a Solution Manual for instructors adopting the book.Readers can often follow a model or example in the text to assist them in working out exercises, but in other cases they must proceed on their own from a mathematical description given in the text or in the problems. In most of the computer problems there is something to be learned beyond simply writing code—a *moral*, if you like. Some computing problems are designed to give experience in using preprogrammed or "canned" library codes. Computer supprograms from general-purpose software libraries (available on many computer systems) can be used with this text. Two such products are the IMSL Library and the NAG Library.

Our own recommendations for courses based on this text are as follows:

- A one-term course carefully covering Chapters 1 through 8 (possibly omitting starred sections) and then a selection of material from the remaining chapters as time permits.
- A one-term survey rapidly covering all chapters in the text, omitting sections which are starred.
- A full-year course carefully covering all chapters and all starred sections.

Acknowledgments. In preparing the second edition, we have been able to profit from advice and suggestions kindly offered by a large number of colleagues, students, and users of the first edition. It is our pleasure to thank them and others who helped with the task of preparing the new edition. Katherine Mueller typed our revisions in the manuscript with great care and attention to detail. Tran Phien checked many of the exercises and helped eliminate errors. Valuable comments and suggestions were made by our colleagues at The University of Texas at Austin: David Young, Thomas Oppe, Arnold Miller, Jonathan Barzilai, Alfred Borm, Cecilia Jea, and Murray Cantor. In particular, David Young has been very generous with suggestions for improving the accuracy and clarity of the exposition. The appendix on Fortran was read by Joyce Brennan, John Respess, and Carole Kincaid, all of whom offered useful suggestions. The revised manuscript was read by several reviewers who provided detailed critiques. These were José G. Arguello of Texas A&M University; Betty J. Barr of the University of Houston; Sidney Birnbaum of California Polytechnic University; Charles R. Deeter of

Texas Christian University; Joseph M. Egar of Cleveland State University; Jack Goebel of Montana College of Mineral Science and Technology; Donald W. Grace of Oklahoma State University; the late Robert Todd Gregory of the University of Tennessee; and John H. Mathews of California State University at Fullerton. Many individuals took the trouble to write us with suggestions and criticisms. We thank J. P. Chandler, Richard Cleary, Charles Groetsch, Henry L. Loeb, George Minty, and Ramon Moore for their letters. The staff of Brooks/Cole Publishing Company, especially Vernon Boes, Judith Macdonald, and Dorothy Bell, has been most understanding and patient in bringing this book to fruition. In particular, we thank Candyce Cameron and Craig Barth for their efforts on behalf of this project.

We would appreciate any comments, questions, criticisms, or corrections that readers may take the trouble of communicating to us.

Ward Cheney
David Kincaid

CONTENTS

*Throughout this text, some sections are indicated with asterisks. These sections may be skipped at the instructor's discretion.

1 INTRODUCTION

The Taylor series for $\ln(1 + x)$ gives us

$$\ln 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \frac{1}{7} - \frac{1}{8} + \cdots$$

Adding together the eight terms shown, we obtain 0.63452, which is a poor approximation to $\ln 2 \approx 0.69315$. However, the Taylor series for $\ln[(1 + x)/(1 - x)]$ gives us

$$\ln 2 = 2\left(3^{-1} + \frac{3^{-3}}{3} + \frac{3^{-5}}{5} + \frac{3^{-7}}{7} + \cdots\right)$$

Now adding only the four terms shown, we obtain 0.69313.

This chapter contains a discussion of Taylor's theorem, one of the important results from calculus and a ubiquitous feature in much of numerical analysis, and a review of Fortran in the form of hints and suggestions for good programming.

The objective of this text is to help the reader to understand some of the many methods of solving scientific problems on a modern computer. We intentionally limit ourselves to the typical problems that arise in science, engineering, and technology. Thus we do not touch upon problems of accounting, modeling in the social sciences, information retrieval, artificial intelligence, and so on.

Usually our treatment of problems will not begin at the source, for that would take us far afield into such areas as physics, engineering, and chemistry. Instead we consider problems after they have been cast into certain standard mathematical forms. The reader is asked, therefore, to accept on faith the assertion that the chosen topics are indeed important ones in scientific computing.

In order to survey a large number of topics, some must be covered briefly and therefore compressed somewhat to make them fit comfortably into this book's format. Obviously our treatment of some topics must be superficial. But it is hoped that the reader will acquire a good bird's-eye view of the subject as a whole and therefore be better prepared for a further, deeper study of numerical analysis.

For each principal topic, we have listed good current sources for further information. In any realistic computing situation, considerable thought should be given to the choice of method to be employed. Although most procedures presented here are useful and important, they may or may not be the optimum choice for a particular problem. In choosing among available methods for solving a problem, the analyst or programmer should consult recent references.

Becoming familiar with basic numerical procedures without realizing their limitations would be foolhardy. Numerical computations are almost invariably contaminated by errors, and it is important to understand the source, propagation, and magnitude of these errors. While we cannot help but be impressed by the speed and accuracy of the modern computer, we should temper our admiration with generous measures of skepticism. As a wise observer commented upon the computer age: "Never in the history of mankind has it been possible to produce so many wrong answers so quickly." Thus one of our goals is to help the reader arrive at this state of skepticism, armed with methods for estimating and controlling errors.

1.1 PROGRAMMING SUGGESTIONS

The reader is expected to be familiar with the rudiments of programming. The programming language adopted here is Fortran, the one most commonly used in scientific applications. Many other languages are suitable for the computing problems presented, but no attempt will be made to include them in the discussion. A summary of Fortran is given in Appendix A, and a table of standard Fortran functions is printed on the inside back cover. Furthermore, the programs scattered throughout the book illustrate many features of Fortran and can serve as models.

Programming Advice

Since the programming of numerical schemes is essential to understanding them, we offer here a few words of advice on programming. Strive to write programs carefully and correctly. Before beginning the Fortran coding, write out in complete detail the mathematical algorithm to be used. When writing the code, use a style that is easy to read and understand without its having to be run on a computer. Check the code thoroughly for errors and omissions before beginning to edit on a terminal. Every extra minute spent in checking the code is worth at least 10 minutes at the computer terminal, spent submitting the run, waiting for output, discovering an error, correcting the error, resubmitting the run, and so on ad nauseam.

If the code can be written so that it can handle a more general situation, then in many cases it will be worth the extra effort to do so. A program written for only a particular set of numbers must be completely rewritten for another set. For example, only a few additional Fortran statements are required to write a program with an arbitrary step size compared with a program in which the step size is fixed numerically. However, too much generality can make a simple programming task difficult!

Build a large program in steps by writing and testing a series of subroutines or function subprograms. Try to keep subprograms reasonably small, less than a page whenever possible, to make reading and debugging easier.

After writing a subprogram, check it by tracing through the code with pencil and paper on a typical yet simple example. Checking boundary cases, such as the values of the first and second iteration in a loop and the processing of the first and last element in a data structure, will often disclose embarrassing errors. These same sample cases can be used as the first set of test cases on the computer.

Print out enough intermediate results to assist in debugging and understanding the program's operation. Fancy output formats are not necessary, but some simple labeling of the output is recommended. Never count lines of data; instead terminate the input with a flag or end-of-file check. Always echo-print the input data.

It is often helpful to assign meaningful names to the variables. For example, `KOUNT`, `XMAX`, and `NEW` may have greater mnemonic value than simply `K`, `X`, and `N`. On the other hand, the chances of making a typing or spelling error are slightly increased. There is perennial confusion between the characters `O` (letter "oh") and `Ø` (number zero) and between `I` (letter "eye") and `1` (number one). One way to decrease the chance of error is to use these symbols sparingly, especially in variable names. In writing code, a slash through a zero (`Ø`) is often used to distinguish it from the letter `O`.

Some comments within a routine are helpful in recalling at some later time what the code does. Extensive comments are not necessary, but a preface to each subprogram explaining usage and parameters is recommended. Inserting blank comment lines can greatly improve the readability of code. In this book, comments are not included in the code because explanatory material is present in abundance near each routine.

Use data structures that are natural to the problem at hand. If the problem adapts more easily to a three-dimensional array than to several one-dimensional arrays, then a three-dimensional array should be used.

Do not sacrifice the clarity of a program in an effort to make the code run faster. Clarity of code is preferable to "optimized code" when the two criteria conflict.

In preference to one you might write yourself, a *preprogrammed* routine from a program library should be used when applicable. Such routines can be expected to be state-of-the-art software, well tested, and, of course, completely debugged. The programmer should use elementary intrinsic subprograms, such as `SIN` and `SQRT`, when possible.

`DATA` statements should be used in preference to executable statements in setting constants. For example, use the statement

```
      DATA PI/3.1415926535898/
```

instead of

```
      PI=3.1415926535898
```

In programs where maximum efficiency is not important, mathematical constants such as π and e can be more easily and safely entered by use of some standard Fortran functions. Thus, the statements

```
PI=4.0*ATAN(1.0)
 E=EXP(1.0)
```

should give π and e to full machine precision. Since it is easy to mistype long sequences of digits in such constants, the use of Fortran functions is recommended for the programming problems in this book. In setting a constant such as 10^9, one should use `1.0E9`, which is assigned at compilation time, rather than `10.**9`, which involves a computation.

In the usual mode of representing numbers in a computer, one word of storage is used for each number. This mode of representation is called **single precision**. In calculations requiring greater precision, it is possible to allot two or more words of storage to each number. The mode of representation in which two words are used is called **double precision**. In order to be of use, arithmetic operations and function evaluations must be available in double precision.

Here is a simple code to illustrate programming in double precision. It computes π:

```
DOUBLE PRECISION PI
PI=4.0D0*DATAN(1.0D0)
PRINT *,PI
STOP
END
```

In brief: (1) double-precision variables must be declared as such; (2) double-precision constants use the letter `D` in the exponent; (3) a double-precision library function is used; and (4) format-free printing is utilized. Mixed-mode arithmetic between single and double precision is allowed in Fortran. For example, the second line of code could be written using the "generic" library function `ATAN`. We recommend not using the mixed-mode features of Fortran 77; errors can be avoided by being explicit. In a format statement, the letter `D` should be used instead of the letter `E` in the output field. The reader should use the preceding remarks only as an informal introduction to double-precision computing.

Two words of memory are also used to store the real and imaginary parts of a complex number. Complex variables and arrays must be explicitly declared as being of complex type. Expressions involving variables and constants of complex type are evaluated according to the normal rules of complex arithmetic. Functions `CMPLX`, `REAL`, and `AIMAG` should be used to convert between real and complex types. For example,

$$z = (x + iy)(3 - 7i)$$

is computed by

```
COMPLEX Z
Z=CMPLX(X,Y)*CMPLX(3.0,-7.0)
```

There is no complex double-precision data type in Standard Fortran 77.

There are many predefined mathematical functions available in Fortran. The main ones are listed in the table on the inside back cover of this book. In Fortran 77, many intrinsic functions accept arguments of more than one type and return a result whose type may vary depending on the type of the argument used. Such functions are called **generic functions**, for they represent an entire family of related functions. For example, the functions `ABS`, `SQRT`, `LOG`, `EXP`, `SIN`, and `COS` can be used with real, double-precision, or complex arguments, and the results will have the same type as the argument. Many other functions can handle arguments of real and double-precision type only, while others allow complex, integer, real, and double precision, such as `ABS`. For the novice programmer, we recommend being conservative and explicitly using the Fortran function of proper type. For example, use `IABS` for integer absolute value, `DSIN` for double-precision sine, `CSQRT` for complex square root, and so on.

One can avoid the necessity for type declaration statements by choosing variable names appropriately. Variables whose names begin with `I`, `J`, `K`, `L`, `M`, or `N` are automatically interpreted as integer variables. All others are real unless declared otherwise.

In general, one should avoid "mixed expressions." These are formulas in which variables of different type appear. If the floating-point form of an integer variable is needed, use the Fortran function `REAL`. For example, the statement

```
X=1.0/REAL(N)
```

gives x the value $1/n$. The statement

```
X=1.0/N
```

is a mixed expression *to be avoided*. Compilers may treat complicated mixed expressions in an unexpected way and it is better not to take chances.

When values of a function at arbitrary points are needed in a program, several ways of coding are available. For example, if values of the function $f(x) = e^x - \sin x + x^2$ are needed, a simple approach is to use the Fortran statement

```
Y=EXP(X)-SIN(X)+X*X
```

at appropriate places within the program. Equivalently, a statement function at the beginning of the program

```
F(X)=EXP(X)-SIN(X)+X*X
```

can be used with `Y=F(2.5)` or whatever value of `X` is desired. A function subprogram

```
FUNCTION F(X)
F=EXP(X)-SIN(X)+X*X
RETURN
END
```

would accomplish the same objective. If the function subprogram `F` is passed as a parameter in another subprogram, then the statement `EXTERNAL F` is needed before the first executable statement in each routine that contains such a call. Which implementation is best? It depends on the situation. The Fortran statement is simple and safe. The statement function and function subprogram can be utilized to avoid duplicating code. In using program library routines, the user may be required to furnish a subprogram in order to communicate function values to the library routine. A separate subprogram is the best way to avoid difficulties that inadvertently occur when someone must insert code into another's routine.

Although the mathematical description of an algorithm may indicate that a sequence of values is computed, thus seeming to imply the need for an array, it is often possible to avoid arrays. (This is especially true if only the final value of a sequence is required.) For example, the theoretical description of Newton's method (Chapter 3) reads

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

but the Fortran code can be written simply as

```
X=X-F(X)/FP(X)
```

Such a statement automatically effects the replacement of the value of the "old" x with the "new" numerical value of $x - f(x)/f'(x)$. Here `F` and `FP` are statement functions or function subprograms for f and f', respectively.

Some care should be exercised in writing Fortran statements that involve exponents. The general function x^y is computed on many machines as $\exp(y \ln x)$. Sometimes this is unnecessarily complicated and may contribute to round-off errors. For example, `X**5` is preferable to `X**5.0` for this reason. Similarly, `SQRT(X)` is preferable to `X**0.5`.

Never put unnecessary statements within `DO` loops. Move expressions and variables that do not change within a `DO` loop outside the loop. Also, indenting `DO` loops can add to the readability of code. Using a `CONTINUE` statement as the terminator of a `DO` loop allows a program to be easily altered and may contribute to good programming style.

Do not write code for repetitive computations when a `DO` loop can be utilized, since unforeseen problems (e.g., programming errors and round-off errors) could cause programmed loops to become infinite. Similarly, in a repetition algorithm always limit the number of permissible steps by the use of a `DO` loop in order to minimize the chance of endless cycling. For example, in Newton's method we might write

```
 2 PRINT *,X
   X=X-F(X)/FP(X)
   IF(ABS(F(X)) .GE. 0.5E-8) GO TO 2
   PRINT*,X
```

If the function involves some erratic behavior, there is danger here in not limiting the number of repetitions of this loop. It is better to use a **DO** loop.

```
   DO 2 N=1,10
     PRINT *,X
     X=X-F(X)/FP(X)
     IF(ABS(F(X)) .LT. 0.5E-8) GO TO 3
 2 CONTINUE
 3 PRINT *,X
```

There is rarely any need for a calculation such as `J=(-1)**N`, since there are better ways of obtaining the same results. For example, in a **DO** loop we can write `J=1` before the loop and `J=-J` inside the loop.

Arrays in Fortran, whether one-, two-, or three-dimensional, are stored in consecutive words of memory. Since the compiler maps two and three subscript expression values into a single subscript value that is used as a position index to determine the location of elements in storage, the use of two- and three-dimensional arrays can be considered a notational convenience for the user. For example, if `IX`, `JX`, and `KX` are the dimensions of an array `X` in a dimension statement, and if `I`, `J`, and `K` are subscript expressions that appear when the array is used, then the location of the element in storage is given by the following table:

Dimension of Array	*Element of Array*	*Index of Element of Array*
`A(IA)`	`A(I)`	`I`
`B(IB,JB)`	`B(I,J)`	`I+IB*(J-1)`
`C(IC,JC,KC)`	`C(I,J,K)`	`I+IC*(J-1)+IC*JC*(K-1)`

Since any advantage in using only one-dimensional arrays and performing complicated subscript calculations is slight, this matter is best left to the compiler.

When using **DO** loops, write the code so that fetches are made from *adjacent* words in memory. To illustrate, the following loops are correctly written so that the arrays are processed down the columns:

```
   DO 2 J=1,500
   DO 2 I=1,500
     X(I,J)=Y(I,J)*Z(I,J)
 2 CONTINUE
```

For some programs, this detail may be only a secondary concern. However, some computers have immediate access only to a portion or "page" of memory at a time; in this case, it is advantageous to process arrays down the columns.