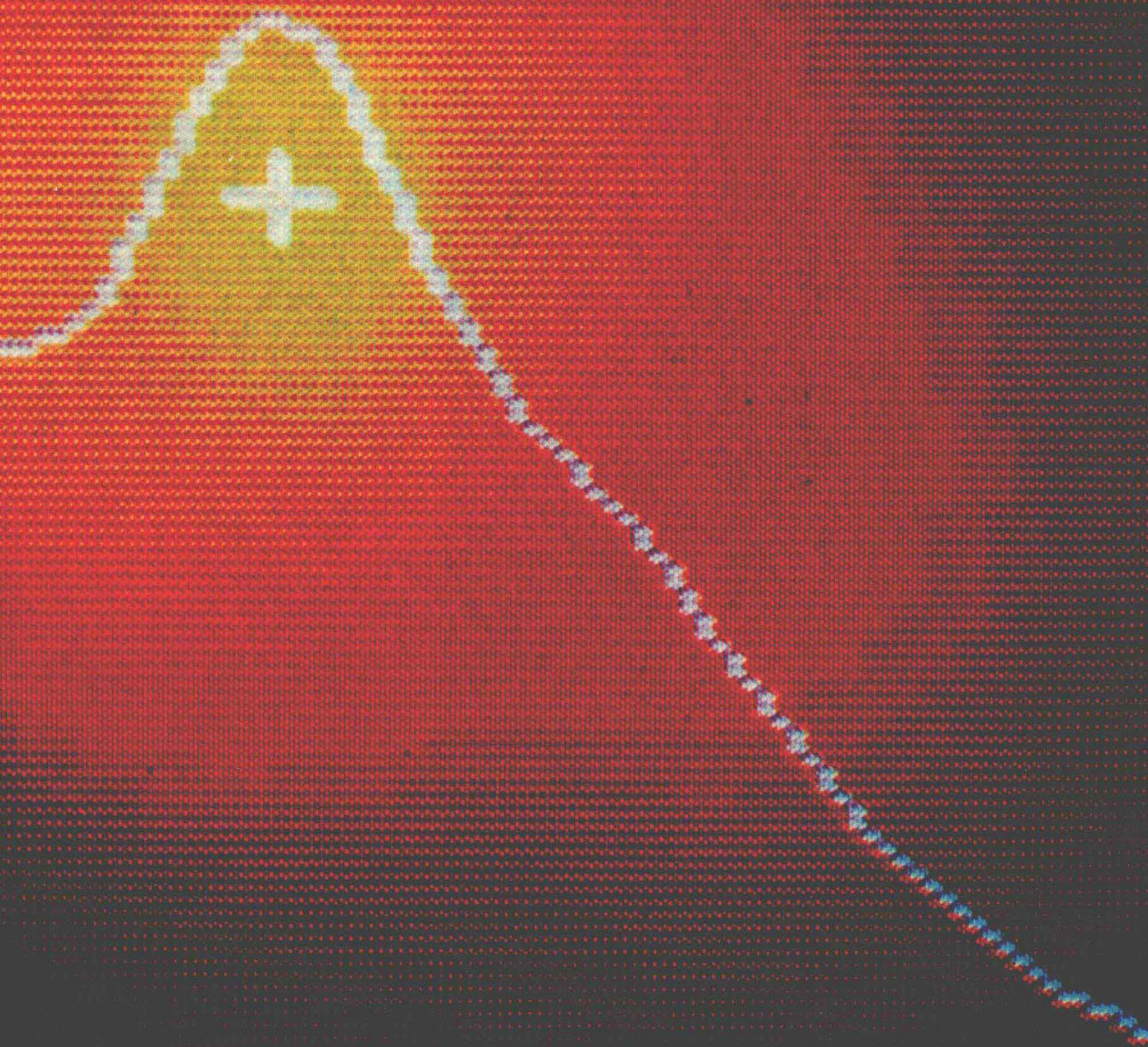
# NUMERICAL METHODS

## FOR ENGINEERS AND COMPUTER SCIENTISTS

PAUL F. HULTQUIST



# NUMERICAL METHODS

## FOR ENGINEERS AND COMPUTER SCIENTISTS

PAUL F. HULTQUIST

*University of Colorado at Denver*


**The Benjamin/Cummings Publishing Company, Inc.**
Menlo Park, California • Reading, Massachusetts
Don Mills, Ontario • Wokingham, U.K. • Amsterdam • Sydney
Singapore • Tokyo • Madrid • Bogota • Santiago • San Juan

Sponsoring Editor: Sally Elliott
Production Editor: Wendy Calmenson, The Book Company
Copyeditor: Linda Thompson
Cover Designer: Wendy Calmenson, The Book Company
Cover Photo: N. Foster, © The Image Bank
Technical Artists: Reese and Deborah Thornton, Folium
Composition: Graphic Typesetting Service

The basic text of this book was designed using the Modular Design System, as developed by Wendy Earl and Design Office Bruce Kortebein.



**Library of Congress Cataloging-in-Publication Data**

Hultquist, Paul F.
Numerical methods for engineers and computer scientists / Paul F. Hultquist.
p. cm.
Includes index.
ISBN 0-8053-4652-X
1. Numerical analysis—Data processing. 2. Engineering mathematics—Data processing. I. Title.
QA297.H85 1988
519.4—dc19

ISBN: 0-8053-4652-X
BCDEFGHIJ-DO-898

**The Benjamin/Cummings Publishing Company, Inc.**
2727 Sand Hill Road
Menlo Park, California 94025

# Preface

## Purpose

As science and engineering become more and more sophisticated, there is an increasing need for practitioners to develop skills in using the computer. The necessary skills go beyond the ability to program and include the ability to formulate problems correctly and to solve problems requiring substantial amounts of computation. Because there is now good numerical software available, it is no longer efficient for persons needing to use the computer to write their own numerical procedures. Rather, they should be knowledgeable about the commonly used numerical algorithms and why, where, and when these algorithms succeed or fail. The underlying philosophy of this book is to try to create understanding about the algorithms and not to belabor the fine points of the construction of them. In a true sense it is a book on numerical methods rather than one on numerical analysis. Nonetheless, it can be used in a mathematics department to introduce mathematics majors to the computational mathematics field with the notion that they may be motivated to study the subject later in a more rigorous way. The target audiences are junior and senior students in the sciences and in engineering who need to learn about numerical methods in a one-semester course and professionals who need to upgrade their computer problem-solving skills.

## Features

The official language chosen is PASCAL. It is widely taught and it is a well-structured and strongly typed language that encourages good programming. There are PASCAL procedures in the book for all the major topics, and these are available

on floppy disks for individual student use on personal computers. For those who prefer FORTRAN we encourage the use of good program libraries, notably IMSL. An IMSL subroutine is suggested for each major topic, and examples are shown for the use of some of these subroutines.

Problem sets are given in each chapter. One of the features of the book is the inclusion of *project problems*. These are real-life kinds of problems, many of which are adapted from work that the author has performed in industry or as a consultant. The author generally has required the solution of five to seven of these as a major part of the one-semester course. The materials required to be handed in included a program listing, printed (or graphical) output, and a brief report. The instructor should pay careful attention to such matters as physical units, the correctness of the results, the accuracy, style, spelling, and grammar of the report, and the organization of the output. If the course in which the book is to be used is a junior or senior engineering course, then it should have a design emphasis with respect to the software. The student should be encouraged to do a professional job on all of the project reports.

## Prerequisites

Prerequisites for the course include mathematics through analytic geometry and calculus, an introduction to ordinary differential equations, and an introduction to linear algebra. The last two courses are combined into a single course in many schools, which is a minimum satisfactory preparation. The standard physics and chemistry courses, of course, are necessary background for more advanced work in some field of science or engineering; lack of advanced background in mechanics, electricity, or heat and thermodynamics may make it difficult to solve some of the project problems without help from an instructor.

## Acknowledgments

Finally, I wish to thank the many persons who have helped me in the creation of this book. Paul Saylor of the University of Illinois, David S. Scott of the University of Texas and now of Intel, and Stanley L. Steinberg of the University of New Mexico read the first draft of the manuscript and were most helpful in suggesting substantial improvements. James Hummel of the University of Maryland, Donald Grace of Oklahoma State University, and Thomas Foley of Arizona State University read the second draft and were very helpful in suggesting changes and improvements. Alan Apt, who encouraged me to submit a book proposal, Craig Bartholomew, who patiently guided me through two revisions of the manuscript, and Sally Elliott of Benjamin/Cummings deserve special thanks, as well as Wendy Calmenson of The Book Company, who handled the book production.

Craig Schons, a former student, helped significantly by writing several of the programs as an independent study project. Colleagues in my department deserve thanks for the encouragement they gave me, in particular Bill Murray, who was chairman at the time I started on the project. My family, Juanita, Fred, and Ann, encouraged and supported me through the long days and nights of writing and rewriting. Also, I shall be forever grateful to a host of unnamed friends without whose support I would not have been able to finish this book.

Lakewood, Colorado
August 1, 1987

# Contents

# 1

# Numbers and Their Representations

Because numbers are the lifeblood of scientific and engineering computing, it is necessary for us to understand how they are represented in the computer and how they are treated in numerical computations. Central to that understanding is the whole matter of number systems.

## 1.1 Number Bases

In the system of number representation with which we are familiar (called a *positional system*), each digit's value is determined by its position with respect to the decimal point. For example, the number 675 means 6 times 100 plus 7 times 10 plus 5, so that the digits carry different values depending upon their position within the number. Contrast this with the Roman numeral system, where a symbol has a value more or less independent of its position. However, in the case where a smaller numeral precedes a larger one, the smaller numeral has a negative value:

$$\text{IV} = -1 + 5 = 4$$

$$\text{VI} = 5 + 1 = 6$$

It is extremely difficult to do arithmetic in the Roman numeral system, and one of its few uses in recent times is in representing dates, such as on the cornerstones of public buildings.

The Arabic system of numerals (symbols borrowed from the Sanskrit) with positional representation leads to relatively easy rules of arithmetic. *Algorithms* (prescriptions for carrying out some task that can be described in systematic terms) for doing arithmetic in the decimal system have been known for many centuries.

The idea of other number bases than 10, once thought mainly to be a curiosity, has become an important one in the last generation. John von Neumann suggested during World War II that the binary (base 2) number system was the natural one for computers, because it allowed the use of enormously simplified electronic circuitry of high reliability.* Binary systems are now almost universally used in computing, although as we shall see later, groups of three or four binary digits can be grouped to form octal (base 8) or hexadecimal (base 16) digits for convenience.

Every positional number system has the same features:

1. A base $b$, such as 2 (binary), 8 (octal), 10 (decimal), 16 (hexadecimal).
2. A set of $b$ symbols, such as {0, 1}, {0, 1, 2, 3, 4, 5, 6, 7}, {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, or {0, 1, . . . , 9, $A$, $B$, $C$, $D$, $E$, $F$}. (Notice in the hexadecimal system, $A = 10$, $B = 11$, $C = 12$, and so on.)
3. Positional representation:

   $$a_n a_{n-1} \cdots a_1 a_0 . a_{-1} a_{-2} \cdots$$

   where the $a$'s are members of the set of symbols and where $a_k$ carries as its value $a_k b^k$.

Thus $225.3 = 2 \times 10^2 + 2 \times 10^1 + 5 \times 10^0 + 3 \times 10^{-1}$. Likewise, the binary number

$$101.11 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$$

which is 5.75 in decimal.

In order to distinguish 101.11 (binary) from 101.11 (decimal) we often write

$$101.11_2 = 5.75_{10}$$

or

$$(101.11)_2 = (5.75)_{10}$$

### 1.1.1 Conversion Among Bases

The previous discussion gives a hint of how conversions between number bases can be done. Basically, there are four kinds of conversions needed. A general conversion of a number with an integer part and a fraction requires the use of

*There is evidence that John Atanasoff independently came to the same conclusion before World War II. Because he was not successful in creating a complete working computer that attracted wide attention, his name did not become attached to the idea.

two of the four methods. The four methods depend upon whether the conversion is made from the decimal system to some other system, or vice versa, and on whether the number being converted is a fraction or an integer. These methods will be illustrated using octal and decimal systems, but the principles are the same for conversions among any two number systems.

**Converting Octal Integers to Decimal Integers** This method is easily understood by means of an example.

$$257_8 = 2 \times 8^2 + 5 \times 8 + 7 = 128 + 40 + 7 = 175_{10}$$

The conversion is more efficiently done in *nested* form:

$$(2 \times 8 + 5) \times 8 + 7 = 21 \times 8 + 7 = 168 + 7 = 175_{10}$$

**Converting Octal Fractions to Decimal Fractions**

$$(0.257)_8 = 2 \times 8^{-1} + 5 \times 8^{-2} + 7 \times 8^{-3} = 0.341796875_{10}$$

This is also done more efficiently in nested form:

$$\left(\left(\tfrac{7}{8} + 5\right)\Big/8 + 2\right)\Big/8 = (5.875/8 + 2)/8 = 2.734375/8 = 0.341796875_{10}$$

**Converting Decimal Integers to Octal Integers** The algorithm is easy to understand if we recognize that, for example,

$$375_{10} = a_0 + a_1 \cdot 8 + a_2 \cdot 8^2 + \cdots$$

We now divide both sides by 8:

$$46 + \tfrac{7}{8} = \frac{a_0}{8} + a_1 + a_2 \cdot 8 + \cdots$$

The integer parts must equal the integer parts and the fractions must equal the fractions, which allows us to conclude that $a_0 = 7$ and that

$$46_{10} = a_1 + a_2 \cdot 8 + a_3 \cdot 8^2 + \cdots$$

Division by 8 again yields

$$5 + \tfrac{6}{8} = \frac{a_1}{8} + a_2 + a_3 \cdot 8 + \cdots$$

Hence $a_1 = 6$ and

$$5 = a_2 + a_3 \cdot 8 + a_4 \cdot 8^2 + \cdots$$

Again, dividing by 8, we have

$$0 + \frac{5}{8} = \frac{a_2}{8} + a_3 + a_4 \cdot 8 + \cdots$$

which allows us to write $a_2 = 5$ and

$$0 = a_3 + a_4 \cdot 8 + \cdots$$

This can be true if and only if $a_3 = a_4 = \cdots = 0$. Thus

$$375_{10} = (a_2 a_1 a_0)_8 = 567_8$$

In a shortcut form, this method is as follows:

$$\begin{array}{r} 46 \\ 8\overline{)375} \\ \underline{32} \\ 55 \\ \underline{48} \\ 7 = a_0 \end{array} \qquad \begin{array}{r} 5 = a_2 \quad \text{(because } 5 < 8\text{)} \\ 8\overline{)46} \\ \underline{40} \\ 6 = a_1 \end{array}$$

**Converting Decimal Fractions to Octal Fractions** The method here is analogous to that of the last method, except that we use multiplication. For example, let us convert decimal 0.35 to octal:

$$0.35_{10} = a_{-1} \cdot 8^{-1} + a_{-2} \cdot 8^{-2} + a_{-3} \cdot 8^{-3} + \cdots$$

Multiply by 8:

$$2.80 = a_{-1} + a_{-2} \cdot 8^{-1} + a_{-3} \cdot 8^{-2} + \cdots$$

Because the whole number part on the left equals the whole number $a_{-1}$, we can subtract $a_{-1} = 2$ from both sides and multiply again:

$$6.40 = a_{-2} + a_{-3} \cdot 8^{-1} + a_{-4} \cdot 8^{-2} + \cdots$$

Now $a_{-2} = 6$. Subtract this from both sides and continue:

$$3.20 = a_{-3} + a_{-4} \cdot 8^{-1} + a_{-5} \cdot 8^{-2} + \cdots$$

Thus $a_{-3} = 3$. If we continue, we find that $a_{-4} = 1$, $a_{-5} = 4$, and so on, so that

$$0.35_{10} = 0.2631463146\cdots_8$$

This process does not terminate as in earlier processes. Here we have an example of one of the problems we face in numerical computation: numbers that are represented by terminating decimals in one number base system may not be exactly represented in another base system because we have only finite-length registers in the computer.

## 1.1.2 Decimal, Binary, Octal, and Hexadecimal

Because humans have four fingers and a thumb on each hand, we are stuck with the decimal system, at least for now. Those of us who deal with computing machines are also stuck with at least three other common bases: binary, octal, and hexadecimal. That is not quite as bad as it seems because binary, octal, and

hexadecimal are in a sense all interchangeable. Suppose we write a large binary number such as $10101110101101_2 = 11181_{10}$ and group the digits by 3s (adding 1 zero to the leftmost group), starting from the right end:

010_101_110_101_101

Now write each triplet in decimal (which has the same set of digits as octal for 0 through 7):

2_5_6_5_5

This is simply the octal number 25655.

If we group the binary digits in 10101110101101 by fours (adding 2 zeros to the leftmost group) and replace each group by hexadecimal digits, we have

$$0010\_1011\_1010\_1101 = 2BAD_{16}$$

where $A = 10 = 1010_2$, $B = 11 = 1011_2$, and $D = 13 = 1101_2$.

If we write a binary integer in general as

$$\sum_{k=0}^{n} a_k 2^k = a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \cdots + a_1 \cdot 2 + a_0$$

we see that successive 3-bit (binary integer) groups (counted from the right-hand end of a number) are of the form

$$a_{3m+2} \cdot 2^{3m+2} + a_{3m+1} \cdot 2^{3m+1} + a_{3m} \cdot 2^{3m}$$

where $m = 0$ for the rightmost 3 bits, $m = 1$ for the 3 bits to the left of those, and so on. We can factor $2^{3m} = 8^m$ out of each of the terms to obtain

$$(a_{3m+2} \cdot 2^2 + a_{3m+1} \cdot 2 + a_{3m}) \cdot 8^m$$

The quantity in parentheses is simply an octal digit, which is multiplied by the appropriate power of 8. Thus

$$\sum_{k=0}^{n} a_k 2^k = \sum_{j=0}^{\lceil n/3 \rceil} b_j 8^j$$

where $\lceil n/3 \rceil$ means the least integer equal to or greater than $n/3$ and where the $b_j$ are the octal digits obtained by grouping the binary digits by 3s.

A similar kind of proof can be done for the hexadecimal case using groups of 4 bits instead of groups of 3 bits.

All computer systems provide input-output routines that convert internal binary to decimal for printing or for cathode-ray tube (CRT) monitor display. However, for many system functions it is easier to print or display octal or hex digits rather than decimal. The conversions from binary to octal are simple. For certain purposes the individual bits may be significant and may represent states (on/off) of equipment or system functions. The use of byte-oriented machines (1 byte = 8 bits) has tended to increase the importance of hexadecimal over octal. A byte holds exactly 2 hex digits, whereas octal digits will fit only those systems having word lengths that are multiples of 3 bits.

## 1.2 Number Representation

Computers are capable of representing numbers in several different ways. Integers are stored in one particular form, but numbers possessing fractional parts are stored in a different form. Although the actual form of storage is binary, for clarity we will use decimal in some of the discussions. The principles in each case are the same.

### 1.2.1 Integer Representation

The previous material dealt mostly with positive numbers. However, we need to be able to deal with negative-number representations in the computer as well. In the case of integers, there are three different methods of representing negative numbers: sign-magnitude, 1's complement, and 2's complement. The purpose of using complemented (negative) numbers is that they can be added in a normal adder, which eliminates the need for a subtractor. Complementation is simple to do in the hardware, and the ability to complement is necessary for some of the computer's other logic operations. Sign-magnitude computers also use complementation but at the actual time of addition or subtraction.

In most machines the most-significant bit of the computer word is reserved for the sign. Consequently, in the simplest case of sign-magnitude form, the decimal number $-11$ is in 1-byte form:

```
1  0  0  0  1  0  1  1
↑
└── Sign bit
```

In 1's complement, all the bits are simply complemented (i.e., each 1 bit is converted to 0, and each 0 bit is converted to 1), or it can be considered that the magnitude of the number is subtracted from a word filled with 1 bits:

$$11111111 - 1011 = 11110100$$

Note that the most-significant bit is a 1 to designate a negative number. Unused bit positions at the left end are filled with 1s in what is sometimes called *sign extension*. In the case of 2's complementation, the number is subtracted from all 1s plus 1:

$$\begin{aligned} 100000000 &= 11111111 + 1 \\ -1011 & \\ \hline 11110101 &= -11_{10} \end{aligned}$$

In this case it is clear that the 2's complement is 1 plus the 1's complement. This is very little harder to implement in hardware than the 1's complement.

We will show a few examples of what happens in the hardware when complemented numbers are used.