NUMERICAL METHODS

A SOFTWARE APPROACH

R. L. Johnston

NUMERICAL METHODS
A SOFTWARE APPROACH

R. L. Johnston
University of Toronto

John Wiley & Sons
New York • Chichester • Brisbane • Toronto • Singapore

Library of Congress Cataloging in Publication Data:

Johnston, Robert L.
Numerical methods.

Includes index.
1. Numerical analysis—Computer programs.
I. Title.
QA297.J64 519.4'028'5425 81-12974
ISBN 0-471-09397-1 AACR2

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

PREFACE

This book is intended to serve as a text for an introductory course in numerical methods. It evolved from a set of notes developed for such a course taught to science and engineering students at the University of Toronto.

The purpose of a numerical methods course is to acquaint the student with up-to-date techniques for carrying out scientific computations on an electronic computer. A recent and important tool in this regard is mathematical software—pre-programmed, reliable computer subroutines for solving mathematical problems. As the title implies, this book is oriented toward learning how to use this tool effectively, that is, how to select the most appropriate routine available for solving a particular problem at hand and how to interpret the results that are returned by it. This approach involves more than the usual discussion of numerical methods plus a simple citing of the various software routines that are currently available. In order to be an effective user of a subroutine, one must be aware of its capabilities and limitations and this implies at least an intuitive understanding of how the underlying algorithm is designed and implemented. Hence, while the list of topics covered in the book is more or less standard for a numerical methods text, the treatment is different from most texts in that it emphasizes the software aspects. The aim is to provide an understanding, at the intuitive level, of how and why subroutines work in order to help the reader gain the maximum benefit from them as a computational tool.

The mathematical background assumed is two years of college mathematics including calculus, basic linear algebra, and an introduction to differential equations. Also, the reader should be familiar with programming in a high-level language such as Fortran. In addition, it is assumed that, in order to do the computational exercises, the reader has access to a general-purpose mathematical software package on the local computing system.

In this regard, a package such as TEAPACK (see Appendix), that is designed to facillitate experimentation with algorithms, would be very useful.

This particular package was developed, at Toronto, for use in our numerical methods courses, and it has been made available for general distribution.

I am indebted to a number of people for encouragement and assistance during preparation of the manuscript. They are: Uri Ascher, Cliff Addison, Steve Cook, Julio Diaz, Wayne Enright, Graeme Fairweather, Ian Gladwell, Ken Jackson, Steve Ho-Tai, Tom Hull, Pat Keast, Rudi Mathon, Richard Pancer, David Sayers, Pavol Sermer, Bruce Simpson, and Jim Varah. In addition, I wish to thank the many students in my courses who suffered through preliminary versions of the manuscript and, through their criticisms, helped me to improve it.

Robert L. Johnston

CONTENTS

CHAPTER 1
INTRODUCTION

This book deals with the solution of mathematical problems using an electronic computer. There are two aspects to the subject. One is the development and analysis of viable computer methods for solving the various types of problems that arise. Such methods are called *numerical methods* and their study is a field called *numerical analysis*. It is a highly specialized field requiring a rather sophisticated mathematical background. The second aspect is the use of these methods in the course of carrying out scientific computations. The typical "user" of numerical methods is a non-expert (in numerical analysis) who simply wants to apply the product of the numerical analyst as a reliable tool to assist in the pursuit of his or her own field of study. Usually, such a person has no interest in learning all of the intricacies of a method as long as it solves the problem at hand. However, since numerical methods are not infallible, a "black-box" approach to using them can be dangerous. In order to avoid difficulties, a user should acquire a certain level of expertise. For instance, it is desirable to know whether or not a particular method will indeed compute a solution to a given problem. Also, whenever there is a choice of methods, it is useful to be able to choose the most efficient one available. In other words, one should be an *intelligent*, rather than a naive, user of numerical methods. The purpose of this book is to help the reader become an intelligent user.

In order to select a numerical method for solving a particular problem, a user should (1) know what methods are available, (2) how they work, and (3) have an appreciation of their relative advantages and disadvantages. However, instead of the detailed knowledge that a numerical analyst must have, it is sufficient to have an intuitive understanding of the basic principles involved. This is the level at which the material in this book is presented. The mathematical background assumed is that normally acquired in the first two years of university mathematics—differential and integral calculus, elementary linear algebra and an introduction to differential equations—that is, exposure to the various types of mathematical problems treated here.

In recent years, numerical analysts have produced a new product, namely, *mathematical software*—packages of computer subroutines for carrying out the basic computations of science and engineering. A list of

some well-known and widely available packages is given in the Appendix at the end of the book. Most computer installations have mathematical software package(s) available in their libraries so that subroutines can easily be called by a user's program. These routines are based on up-to-date numerical methods and their implementations are designed to be as efficient as possible for the particular machine on which they reside. Hence, mathematical software is a computational tool that can be of great benefit to the scientific programmer. Now, it might seem that the availability of preprogrammed, state-of-the-art subroutines would allow a programmer to adopt the black box approach in using them. However, this is not the case. A perusal of the index of any software package will reveal that it contains several subroutines for each type of problem. Hence, one is faced with the question of choosing the most appropriate routine for solving a given problem. In addition, subroutines are not "fail safe," that is, sometimes they may fail to compute a solution. Very often a routine will recognize such a situation itself and return the information. This essentially eliminates the problem of detecting failure but one must still be able to understand what caused the difficulty so that an appropriate remedy can be taken. In other words, one must have an adequate knowledge of the algorithms implemented in a mathematical software package in order to use it intelligently. As its title implies, this book adopts a "software approach" in that it is intended to help the reader become an intelligent user of mathematical software.

In each of Chapters 2 to 6, we discuss the numerical solution of a specific type of mathematical problem. The list of topics is the usual one for a course entitled "Numerical Methods for Scientists" or some variation thereof. By and large, each chapter follows a similar format. We begin with a discussion of the methods that are normally used by software routines for solving the type of problem under consideration. Following this, a discussion of some aspects of typical subroutines is given. Specifically, we consider calling sequences, some ideas concerning design and implementation, and how to interpret the information returned by a routine. The exercises at the end of each chapter are designed, for the most part, to encourage the reader to investigate the properties of whatever software packages are available on the local computer system. This is in keeping with the stated goal of learning about mathematical software in order to prepare the reader for applying it to solve problems in his or her own field of interest.

This chapter deals with some basic ideas about scientific computing. In the next section, we develop the concepts of computer arithmetic and illustrate some of the pitfalls to be avoided. Then, in Section 1.2, we discuss the process of producing mathematical software.

1.1. COMPUTER ARITHMETIC AND ERROR CONTROL

In the course of carrying out a mathematical computation, one has to deal with the problem of errors. There are three ways in which errors can enter a calculation. First, they may be present at the outset in the original data (*inherent* error). Second, they may occur as the result of replacing an infinite process by a finite one (*truncation* error). A typical example is the representation of a function by the first few terms of its Taylor series expansion. The third source of error arises from the finite precision of the numbers that can be represented in a computer (*round-off* error). The latter is a topic which is discussed in Section 1.1.2. Each of these types of error is unavoidable in a calculation. Hence, the "problem of errors" is not one of preventing their occurrence. Instead, it is one of controlling their size in order to obtain a final result that is as accurate as possible. This process is called *error control.* It is concerned with the propagation of errors throughout a computation. For example, we want to be sure that the error that results from performing an arithmetic operation on two numbers, which are themselves in error, is within tolerable limits. In addition, the propagation, or cumulative effects, of this error in subsequent calculations should also be kept under control. These questions are discussed in this section. We remark that there is also a fourth source of error—one caused by doing an arithmetic operation incorrectly (a blunder). However, we view this type of error as avoidable, that is, it need not occur at all, and will not consider it further.

A modern computer is capable of performing arithmetic operations at very high speeds. As a consequence, large scale computations, which are intractable by desk calculation, can be handled routinely. However, while a computer greatly facilities the job of carrying out mathematical calculations, it also introduces a new form of problem with respect to error control. This is due to the fact that intermediate results are not normally seen by the user. Such results are useful because they provide indications of possible large error buildup as the calculation proceeds. In desk computation, all intermediate results are in front of the problem solver. Consequently, error buildup is relatively easy to detect. On the other hand, a computer programmer must be able to detect or anticipate any possible large errors without seeing the warning signals. The examples in this section illustrate some of the ways that this can occur. Before considering them, however, we discuss the source of round-off errors.

The mathematician, in devising a method for solving a problem, assumes that all calculations will be done within the system R of real numbers. This assumption greatly simplifies the mathematical analysis of problems. However, when it comes to actually computing a solution, we must do

without the real number system. This is because it is infinite and any set of numbers that is representable on a computer is necessarily finite. Actually, R is infinite in two senses. First, it is infinite in range, that is, it contains arbitrarily large numbers (of both signs). On the other hand, a computer number system can, at best, represent only those real numbers within a given finite interval. Second, it is infinitely dense, that is, the interval between any two real numbers contains infinitely many real numbers. The absence of this property in a computer's number system is the source of round-off error. In order to be more precise, we need to discuss the type of (finite) number systems used in computers.

1.1.1. Computer Number Systems

In a computer memory, each number is stored in a location that consists of a sign ($\pm$) plus a fixed number of digits. One question that confronts the designer of the machine is how to use these digits to represent numbers. One approach is to assign a fixed number of them for the fractional part. This is called a *fixed-point* number system. It can be characterized by three parameters:

β—the number base.
t—the number of digits.
f—the number of digits in the fractional part.

We denote such a system by $P(\beta, t, f)$. As an example, we consider $P(10, 4, 1)$. It contains the 19,999 evenly spaced numbers -999.9, $-999.8, \ldots, 999.8, 999.9$. This set is uniformly dense in $[-1000, 1000]$. As a consequence, any real number x in this interval can be represented by an element $fix(x) \in P(10, 4, 1)$ with an *absolute error* $x - fix(x)$ of, at most, 0.05 in magnitude. For example, if $x = 865.54$, then $fix(x) = 865.5$ and the absolute error is 0.04. However, assuming $x \neq 0$, it is preferable, instead, to look at the *relative error* $(x - fix(x))/x$. In this respect, the set $P(10, 4, 1)$ gives an uneven representation of R. For example, the relative error in the representation of 865.54 is $0.04/865.54 \doteq 0.00005$, or 0.005%. On the other hand, if $x = 0.86554$, then $fix(x) = 000.9$ and the relative error is 4%! Hence, the *relative density* of $P(10, 4, 1)$ is not uniform in $[-1000, 1000]$. This weakness is shared by all fixed-point number systems.

Most computers use a *floating-point* number system, denoted by $F(\beta, t, L, U)$. The four parameters are:

β —the number base.
t —the precision.
L, U—the exponent range.

Any nonzero number $x \in F$ has the form

$$x = \pm\left(\frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t}\right) \times \beta^e$$

written as

$$x = \pm .d_1 d_2 \ldots d_t \times \beta^e$$

where the digits $d_1, \ldots, d_t$ in the *fractional part* satisfy

$$1 \le d_1 < \beta$$

$$0 \le d_s < \beta \qquad 2 \le s \le t$$

and the *exponent* e is such that

$$L \le e \le U$$

Also, the number 0 belongs to F. Its representation is

$$0 = +.00 \ldots 0 \times \beta^L$$

As implied, the advantage of a floating point number system is that, within its range of values, the relative density in R is uniform. As an example, consider the system $F(10, 4, -2, 3)$. Its range of values is the two intervals $\pm[.001, 999.9]$ plus 0. Referring to the previous example, the representation of 865.54 is $.8655 \times 10^3$ and, for 0.86554, it is $.8655 \times 10^0$. In each case, the relative error is the same, namely, 0.005%.

To illustrate the comparison between fixed and floating point systems, we display the 33-number sets $P(2, 4, 2)$ and $F(2, 3, -1, 2)$[1] in Figure 1.1.

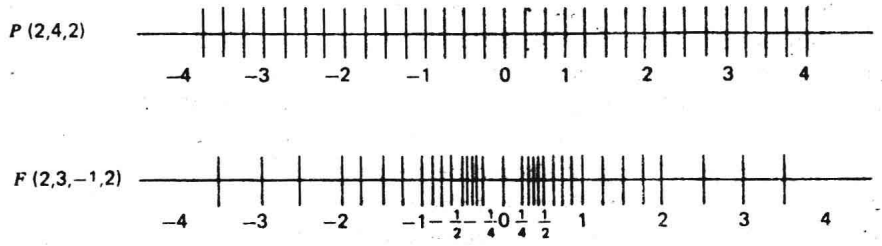

FIGURE 1.1

[1]The representation of the set $F(2, 3, -1, 2)$ is reproduced from [16] with permission.

The difference between absolute and relative density is readily apparent. In effect, the choice between the two types of number systems is one of choosing between absolute and relative error for the purpose of assessing accuracy. Relative error is a measure of the number of correct digits in a number and, as our examples indicate, it is clearly preferable. We remark that this comparison holds in many other contexts as well. As we will see, subroutines usually use relative error as a basis for checking accuracy.

1.1.2. Round-off Errors

Now we consider the differences between performing calculations in F as opposed to R. The source of the differences lies in the fact that F is not closed under the arithmetic operations of addition and multiplication (and the complementary operations of subtraction and division as well). That is, the sum or product of two numbers in F is not necessarily an element of F also. Hence, to stay within the set, we must replace the "true" result of an operation by an element of F and, in the process, incur some error.

There are two ways in which an arithmetic result can lie outside F. First of all, the exponent e in the result may lie outside the range $L \le e \le U$. For example, consider the system $F(2, 3, -1, 2)$ illustrated in Figure 1.1. The product

$$.100 \times 2^2 \times .110 \times 2^2 = .110 \times 2^3 \qquad (2 \times 3 = 6)$$

is not in F because the exponent 3 is too large. This situation is called *overflow*. Similarly, we can have *underflow* when the exponent is too small, as is the case with the product

$$.100 \times 2^0 \times .110 \times 2^{-1} = .110 \times 2^{-2} \qquad (\tfrac{1}{2} \times \tfrac{3}{8} = \tfrac{3}{16})$$

The actual result of trying to represent a number that lies outside the admissible range is highly software and hardware dependent, with little consistency between different computer systems. We will not attempt to define an acceptable criterion here. However, we remark that the occurrence of either overflow or underflow must be considered as an abnormal event in a calculation and the programmer should determine what caused it. Very often, it is an indication of some trouble with the overall algorithm being used, in which case it should be redesigned. Sometimes, however, the difficulty arises from the fact that the range of numbers that has occurred in the computation overlaps one boundary of the admissible range. In this case, a rescaling of the problem will remedy the situation. We will not pursue this topic further. In what follows, we assume that the exponent of a number is within the admissible range.

The second way of obtaining a result outside F is when the fractional

part has more than t digits. Consider again the system $F(2, 3, -1, 2)$. The result of the addition

$$.110 \times 2^0 + .111 \times 2^0 = .1101 \times 2^1 \qquad (\tfrac{3}{4} + \tfrac{7}{8} = \tfrac{13}{8})$$

is not in our set because four (binary) digits are required to represent the fractional part. Similarly, the product

$$.111 \times 2^0 \times .110 \times 2^0 = .10101 \times 2^0 \qquad (\tfrac{7}{8} \times \tfrac{3}{4} = \tfrac{21}{32})$$

is not in F. We remark that while this situation does not always arise with addition, it almost invariably does with multiplication. To define a result that can be represented in the machine, we select a nearby element of F. There are two methods for doing this. Suppose that the actual result of an operation is $.d_1 \ldots d_t d_{t+1} \ldots d_u \times \beta^e$. (Recall our assumption that $L \le e \le U$.) Then the two methods are:

1. *Chopping*, whereby the digits beyond d_t are simply dropped.
2. *Rounding*, whereby the fractional part is taken to be the first t digits of $d_1 d_2 \ldots d_t d_{t+1} + \frac{1}{2}\beta$.

For example, the number in $F(2, 3, -1, 2)$ corresponding to $.1101 \times 2^1$ is $.110 \times 2^1$ by chopping, and $.111 \times 2^1$ by rounding. For $.10101 \times 2^0$, it is $.101 \times 2^0$ by either method. Briefly, the relative merits of the two methods are that chopping is less expensive whereas rounding produces better accuracy. Both methods are in common usage on present day computers.

No matter which method—chopping or rounding—is used to obtain a result in F, there is some error created in the process. We call this *round-off* error (even when chopping is done). More precisely, let $fl(x)$ denote the machine representation of a real number x (whose exponent is within range). Then round-off error is the difference $x - fl(x)$. For $x \neq 0$, we define the *relative round-off* error $\delta(x)$ in $fl(x)$ by

$$\delta(x) = \frac{x - fl(x)}{x}$$

It can be shown [16, p. 88–9] that

$$(1.1) \qquad |\delta(x)| \le \text{EPS} = \begin{cases} \beta^{1-t} & \text{for chopping} \\ \frac{1}{2}\beta^{1-t} & \text{for rounding} \end{cases}$$

Consider, for example, the system $F(10, 4, -50, 50)$ with chopping and suppose $x = 12.467$. Then $fl(x) = .1246 \times 10^2$ and

$$\delta(x) = \frac{0.007}{12.467} \doteq 0.00056 < \text{EPS} = 10^{-3} = 0.001$$

For the same system with rounding, we have $fl(x) = .1247 \times 10^2$ and

$$\delta(x) = \frac{0.003}{12.467} \doteq 0.00024 < \text{EPS} = \frac{1}{2}\,10^{-3} = 0.0005$$

The parameter EPS in (1.1) plays an important role in computation in a floating-point number system. It is commonly referred to as *machine epsilon* and is defined to be the smallest positive machine number such that

$$fl(1 + \text{EPS}) > 1$$

that is, the machine representation of the sum $1 + \text{EPS}$ is different from 1. For example, machine epsilon for $F(10, 4, -50, 50)$ with chopping is $10^{-3} = 0.001$ since

$$fl(1 + .001) = .1001 \times 10^{1} > 1$$

and this is not true for any smaller positive number within the system. Similarly, machine epsilon for $F(10, 4, -50, 50)$ with rounding is 0.0005. Machine epsilon is an indicator of the attainable accuracy in a floating-point number system. For this reason, it is often used in subroutines to determine if the maximum possible accuracy has been achieved. We discuss this further in Section 1.2.

1.1.3. Control of Round-off Error

So far, we have only considered the round-off error incurred by representing the result of a single arithmetic operation whereas, in the course of carrying out a computer calculation, a very large number of arithmetic operations is performed. Therefore, we must be concerned with the question of how these errors propagate and affect the final result. One of the tasks of the numerical analyst is to provide an answer by performing a "round-off error analysis." This is a highly technical process that will not be pursued here. Instead, we adopt the more pragmatic approach of trying to minimize the error created in each operation with the view that this provides less error to be propagated, making the final result as accurate as possible.

There are several ways in which round-off error in each operation or set of operations can be minimized. They fall into three categories: hardware features, software features and careful programming. We discuss one example of each, using the system $F(10, 4, -50, 50)$, with chopping, to illustrate them.

1. *Hardware feature.* Suppose that we want to subtract 0.5678 from 12.34. Before subtracting, the machine representations of the numbers must be adjusted in order to align the decimal points. In the process, some

of the least significant digits of the smaller number will be lost. The provision of a *guard digit*—an extra digit in the fractional part of a number—in the arithmetic unit of a computer can prevent undue loss of accuracy in such situations. To illustrate, we have

No guard digit	*With a guard digit*
$.1234 \times 10^2$	$.12340 \times 10^2$
$.0056 \times 10^2$	$.00567 \times 10^2$
$.1178 \times 10^2$	$.11773 \times 10^2$

The result with the guard digit is closer to the exact result 11.7722. The slash through the 3 indicates that it is chopped when the result is stored. At first glance, it may seem unimportant to quibble about a difference of only one in the last digit of a number. However, in large-scale computations involving millions of arithmetic operations, there is a potential for round-off error to accumulate significantly. Consequently, it is important to ensure that the result of each individual operation is as accurate as possible. For this reason, the provision of a guard digit in the arithmetic unit is generally regarded as essential in a computer which is designed for scientific computation.

2. *Software feature.* An expression that appears frequently in scientific calculations is of the form

(1.2) $\quad a + b \cdot c$

This combination of operations is often referred to as a *floating-point operation* or, briefly, a *flop*. It arises, for instance, in the solution of problems in linear algebra (see Chapter 2). Due to the precedence of operations, the multiplication is performed first. This produces a "double-length" result, that is, either $2t - 1$ or $2t$ digits long. Normally, this would be chopped to t digits before doing the addition. However, better accuracy can be assured if the addition is done before chopping. For example, let $a = 0.1462$, $b = 12.34$, and $c = 0.5678$. Then, assuming a guard digit, we have

	Single length	*Double length*
$b \cdot c$	$.70060 \times 10^2$	$.7006652 \times 10^2$
$+a$	$.00146 \times 10^2$	$.00146 \quad \times 10^2$
	$.70206 \times 10^2$	$.7021\not{2} \quad \times 10^2$

Since flops occur so often in scientific calculations, many compilers are designed to recognize them within an arithmetic statement and assemble the appropriate machine code to carry out the addition using the double