FUNDAMENTALS OF NUMERICAL COMPUTING

L.F. Shampine
R.C. Allen, Jr.
S. Pruess

FUNDAMENTALS OF NUMERICAL COMPUTING

L. F. Shampine
Southern Methodist University

R. C. Allen, Jr.
Sandia National Laboratories

S. Pruess
Colorado School of Mines

JOHN WILEY & SONS, INC.
New York Chichester Brisbane Toronto Singapore

Acquisitions Editor	*Barbara Holland*
Editorial Assistant	*Cindy Rhoads*
Marketing Manager	*Cathy Faduska*
Senior Production Manager	*Lucille Buonocore*
Senior Production Editor	*Nancy Prinz*
Manufacturing Manager	*Mark Cirillo*
Cover Designer	*Steve Jenkins*

This book was set in Times Roman, and was printed and bound by R.R. Donnelley & Sons Company, Crawfordsville. The cover was printed by The Lehigh Press, Inc.

Recognizing the importance of preserving what has been written, it is a policy of John Wiley & Sons, Inc. to have books of enduring value published in the United States printed on acid-free paper, and we exert our best efforts to that end.

The paper in this book was manufactured by a mill whose forest management programs include sustained yield harvesting of its timberlands. Sustained yield harvesting principles ensure that the number of trees cut each year does not exceed the amount of *new growth.*

Library of Congress Cataloging-in-Publication Data:
Shampine, Lawrence
Fundamentals of numerical computing / Richard Allen, Steve Pruess, Lawrence Shampine.
p. cm.
Includes bibliographical reference and index.
ISBN 0-471-16363-5 (cloth : alk. paper)
1. Numerical analysis—Data processing. I. Pruess, Steven. II. Shampine, Lawrence F. III. Title.
QA297.A52 1997
519.4'0285'51—dc20 96-22074
CIP

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

FUNDAMENTALS OF NUMERICAL COMPUTING

PRELIMINARIES

The purpose of this book is to develop the understanding of basic numerical methods and their implementations as software that are necessary for solving fundamental mathematical problems by numerical means. It is designed for the person who wants to do numerical computing. Through the examples and exercises, the reader studies the behavior of solutions of the mathematical problem along with an algorithm for solving the problem. Experience and understanding of the algorithm are gained through hand computation and practice solving problems with a computer implementation. It is essential that the reader understand how the codes provided work, precisely what they do, and what their limitations are. The codes provided are powerful, yet simple enough for pedagogical use. The reader is exposed to the art of numerical computing as well as the science.

The book is intended for a one-semester course, requiring only calculus and a modest acquaintance with FORTRAN, C, C++, or MATLAB. These constraints of background and time have important implications: the book focuses on the problems that are most common in practice and accessible with the background assumed. By concentrating on one effective algorithm for each basic task, it is possible to develop the fundamental theory in a brief, elementary way. There are ample exercises, and codes are provided to reduce the time otherwise required for programming and debugging. The intended audience includes engineers, scientists, and anyone else interested in scientific programming. The level is upper-division undergraduate to beginning graduate and there is adequate material for a one semester to two quarter course.

Numerical analysis blends mathematics, programming, and a considerable amount of art. We provide programs with the book that illustrate this. They are more than mere implementations in a particular language of the algorithms presented, but they are not production-grade software. To appreciate the subject fully, it will be necessary to study the codes provided and gain experience solving problems first with these programs and then with production-grade software.

Many exercises are provided in varying degrees of difficulty. Some are designed to get the reader to think about the text material and to test understanding, while others are purely computational in nature. Problem sets may involve hand calculation, algebraic derivations, straightforward computer solution, or more sophisticated computing exercises.

The algorithms that we study and implement in the book are designed to avoid severe roundoff errors (arising from the finite number of digits available on computers and calculators), estimate truncation errors (arising from mathematical approximations), and give some indication of the sensitivity of the problem to errors in the data. In Chapter 1 we give some basic definitions of errors arising in computations and study roundoff errors through some simple but illuminating computations. Chapter 2 deals with one of the most frequently occurring problems in scientific computation, the solution of linear systems of equations. In Chapter 3 we deal with the problem of

interpolation, one of the most fundamental and widely used tools in numerical computation. In Chapter 4 we study methods for finding solutions to nonlinear equations. Numerical integration is taken up in Chapter 5 and the numerical solution of ordinary differential equations is examined in Chapter 6. Each chapter contains a case study that illustrates how to combine analysis with computation for the topic of that chapter.

Before taking up the various mathematical problems and procedures for solving them numerically, we need to discuss briefly programming languages and acquisition of software.

PROGRAMMING LANGUAGES

The FORTRAN language was developed specifically for numerical computation and has evolved continuously to adapt it better to the task. Accordingly, of the widely used programming languages, it is the most natural for the programs of this book. The C language was developed later for rather different purposes, but it can be used for numerical computation.

At present FORTRAN 77 is very widely available and codes conforming to the ANSI standard for the language are highly portable, meaning that they can be moved to another hardware/software configuration with very little change. We have chosen to provide codes in FORTRAN 77 mainly because the newer Fortran 90 is not in wide use at this time. A Fortran 90 compiler will process correctly our FORTRAN 77 programs (with at most trivial changes), but if we were to write the programs so as to exploit fully the new capabilities of the language, a number of the programs would be structured in a fundamentally different way. The situation with C is similar, but in our experience programs written in C have not proven to be nearly as portable as programs written in standard FORTRAN 77. As with FORTRAN, the C language has evolved into C++, and as with Fortran 90 compared to FORTRAN 77, exploiting fully the additional capabilities of C++ (in particular, object oriented programming) would lead to programs that are completely different from those in C. We have opted for a middle ground in our C++ implementations.

In the last decade several computing environments have been developed. Popular ones familiar to us are MATLAB [1] and *Mathematica* [2]. MATLAB is very much in keeping with this book, for it is devoted to the solution of mathematical problems by numerical means. It integrates the formation of a mathematical model, its numerical solution, and graphical display of results into a very convenient package. Many of the tasks we study are implemented as a single command in the MATLAB language. As MATLAB has evolved, it has added symbolic capabilities. *Mathematica* is a similar environment, but it approaches mathematical problems from the other direction. Originally it was primarily devoted to solving mathematical problems by symbolic means, but as it has evolved, it has added significant numerical capabilities. In the book we refer to the numerical methods implemented in these widely used packages, as well as others, but we mention the packages here because they are programming languages in their own right. It is quite possible to implement the algorithms of the text in these languages. Indeed, this is attractive because the environments deal gracefully with a number of issues that are annoying in general computing using languages like FORTRAN or C.

At present we provide programs written in FORTRAN 77, C, C++, and MATLAB that have a high degree of portability. Quite possibly in the future the programs will be made available in other environments (e.g., Fortran 90 or *Mathematica*.)

SOFTWARE

In this section we describe how to obtain the source code for the programs that accompany the book and how to obtain production-grade software. It is assumed that the reader has available a browser for the World Wide Web, although some of the software is available by ftp or gopher.

The programs that accompany this book are currently available by means of anonymous ftp (log in as *anonymous* or as *ftp*) at

ftp.wiley.com

in subdirectories of public/college/math/sapcodes for the various languages discussed in the preceding section.

The best single source of software is the Guide to Available Mathematical Software (GAMS) developed by the National Institute of Standards and Technology (NIST). It is an on-line cross-index of mathematical software and a virtual software repository. Much of the high-quality software is free. For example, GAMS provides a link to netlib, a large collection of public-domain mathematical software. Most of the programs in netlib are written in FORTRAN, although some are in C. A number of the packages found in netlib are state-of-the-art software that are cited in this book. The internet address is

http://gams.nist.gov

for GAMS.

A useful source of microcomputer software and pointers to other sources of software is the Mathematics Archives at

http://archives.math.utk.edu:80/

It is worth remarking that one item listed there is an "Index of resources for numerical computation in C or C++."

There are a number of commercial packages that can be located by means of GAMS. We are experienced with the NAG and IMSL libraries, which are large collections of high-quality mathematical software found in most computing centers. The computing environments MATLAB and *Mathematica* mentioned in the preceding section can also be located through GAMS.

REFERENCES

1. C. Moler, J. Little, S. Bangert, and S. Kleiman, *ProMatlab User's Guide, MathWorks*, Sherborn, Mass., 1987. email: info@mathworks.com

2. S. Wolfram, *Mathematica*, Addison-Wesley, Redwood City, Calif., 1991. email: info@wri.com

ACKNOWLEDGMENTS

The authors are indebted to many individuals who have contributed to the production of this book. Professors Bernard Bialecki and Michael Hosea have been especially sharp-eyed at catching errors in the latest versions. We thank the people at Wiley, Barbara Holland, Cindy Rhoads, and Nancy Prinz, for their contributions. David Richards at the University of Illinois played a critical role in getting the LaTeXmacros functioning for us, and quickly and accurately fixing other LaTeXproblems. We also acknowledge the work of James Otto in checking all solutions and examples, and Hong-sung Jin who generated most of the figures. Last, but certainly not least, we are indeed grateful to the many students, too numerous to mention, who have made valuable suggestions to us over the years.

CONTENTS

CHAPTER 1

ERRORS AND FLOATING POINT ARITHMETIC

Errors in mathematical computation have several sources. One is the modeling that led to the mathematical problem, for example, assuming no wind resistance in studying projectile motion or ignoring finite limits of resources in population and economic growth models. Such errors are not the concern of this book, although it must be kept in mind that the numerical solution of a mathematical problem can be no more meaningful than the underlying model. Another source of error is the measurement of data for the problem. A third source is a kind of mathematical error called discretization or truncation error. It arises from mathematical approximations such as estimating an integral by a sum or a tangent line by a secant line. Still another source of error is the error that arises from the finite number of digits available in the computers and calculators used for the computations. It is called roundoff error. In this book we study the design and implementation of algorithms that aim to avoid severe roundoff errors, estimate truncation errors, and give some indication of the sensitivity of the problem to errors in the data. This chapter is devoted to some fundamental definitions and a study of roundoff by means of simple but illuminating computations.

1.1 BASIC CONCEPTS

How well a quantity is approximated is measured in two ways:

$$\textit{absolute error} = \text{true value} - \text{approximate value}$$

$$\textit{relative error} = \frac{\text{true value} - \text{approximate value}}{\text{true value}}.$$

Relative error is not defined if the true value is zero. In the arithmetic of computers, relative error is the more natural concept, but absolute error may be preferable when studying quantities that are close to zero.

A mathematical problem with input (data) x and output (answer) $y = F(x)$ is said to be *well-conditioned* if "small" changes in x lead to "small" changes in y. If the

changes in y are "large," the problem is said to be *ill-conditioned.* Whether a problem is well- or ill-conditioned can depend on how the changes are measured. A concept related to conditioning is stability. It is concerned with the sensitivity of an algorithm for solving a problem with respect to small changes in the data, as opposed to the sensitivity of the problem itself. Roundoff errors are almost inevitable, so the reliability of answers computed by an algorithm depends on whether small roundoff errors might seriously affect the results. An algorithm is *stable* if "small" changes in the input lead to "small" changes in the output. If the changes in the output are "large," the algorithm is *unstable.*

To gain some insight about condition, let us consider a differentiable function $F(x)$ and suppose that its argument, the input, is changed from x to $x+\varepsilon x$. This is a relative change of ε in the input data. According to Theorem 4 of the appendix, the change induces an absolute change in the output value $F(x)$ of

$$F(x) - F(x+\varepsilon x) \approx -\varepsilon x F'(x).$$

The relative change is

$$\frac{F(x) - F(x+\varepsilon x)}{F(x)} \approx -\varepsilon x \frac{F'(x)}{F(x)}.$$

Example 1.1. If, for example, $F(x) = e^x$, the absolute change in the value of the exponential function due to a change εx in its argument x is approximately $-\varepsilon x e^x$, and the relative change is about $-\varepsilon x$. When x is large, the conditioning of the evaluation of this function with respect to a small relative change in the argument depends strongly on whether the change is measured in an absolute or relative sense. ■

Example 1.2. If $F(x) = \cos x$, then near $x = \pi/2$ the absolute error due to perturbing x to $x+\varepsilon x$ is approximately $-\varepsilon x(-\sin x) \approx \pi\varepsilon/2$. The relative error at $x = \pi/2$ is not defined since $\cos(\pi/2) = 0$. However, the accurate values

$$\cos(1.57079) = 0.63267949 \times 10^{-5}$$
$$\cos(1.57078) = 1.63267949 \times 10^{-5}$$

show how a very small change in the argument near $\pi/2$ can lead to a significant (63%) change in the value of the function. In contrast, evaluation of the cosine function is well-conditioned near $x = 0$ (see Exercise 1.4). ■

Example 1.3. A common application of integration by parts in calculus courses is the evaluation of families of integrals by recursion. As an example, consider

$$E_n = \int_0^1 x^n e^{x-1}\, dx \text{ for } n = 1,2,\ldots.$$

From this definition it is easy to see that

$$E_1 > E_2 > \cdots > E_{n-1} > E_n > \cdots > 0.$$

To obtain a recursion, integrate by parts to get

$$\begin{aligned} E_n &= x^n e^{x-1}\Big|_0^1 - \int_0^1 n x^{n-1} e^{x-1}\,dx \\ &= 1 - nE_{n-1}. \end{aligned}$$

The first member of the family is

$$E_1 = 1 - \int_0^1 e^{x-1}\,dx = e^{-1},$$

and from it we can easily compute any E_n. If this is done in single precision on a PC or workstation (IEEE standard arithmetic), it is found that

$$\begin{array}{rcll} E_1 &=& 0.367879 & \\ E_2 &=& 0.264241 & \\ &\vdots & & \\ E_{10} &=& 0.0506744 & \\ E_{11} &=& 0.442581 & \text{(the exact } E_n \text{ decrease!)} \\ E_{12} &=& -4.31097 & \text{(the exact } E_n \text{ are positive!)} \\ &\vdots & & \\ E_{20} &=& -0.222605 \times 10^{11} & \text{(the exact } E_n \text{ are between 0 and 1!)} \end{array}$$

This is an example of an unstable algorithm. A little analysis helps us understand what is happening. Suppose we had started with $\widehat{E}_1 = E_1 + \delta$ and made no arithmetic errors when evaluating the recurrence. Then

$$\begin{aligned} \widehat{E}_2 &= 1 - 2\widehat{E}_1 = 1 - 2E_1 - 2\delta = E_2 - 2\delta \\ \widehat{E}_3 &= 1 - 3\widehat{E}_2 = 1 - 3E_2 + 6\delta = E_3 + 3!\delta \\ &\vdots \\ \widehat{E}_n &= E_n \pm n!\delta. \end{aligned}$$

A small change in the first value E_1 grows very rapidly in the later E_n. The effect is worse in a relative sense because the desired quantities E_n decrease as n increases.

For this example there is a way to get a stable algorithm. If we could find an approximation $\widehat{E}_N$ to E_N for some N, we could evaluate the recursion in reverse order,

$$E_{n-1} = \frac{1 - E_n}{n}, \qquad n = N, N-1, \ldots, 2,$$

to approximate $E_{N-1}, E_{N-2}, \ldots, E_1$. Studying the stability of this recursion as before, if $\widehat{E}_N = E_N + \varepsilon$, then

$$\widehat{E}_{N-1} = \frac{1 - \widehat{E}_N}{N} = \frac{1 - E_N}{N} - \frac{\varepsilon}{N} = E_{N-1} - \frac{\varepsilon}{N}$$

$$\widehat{E}_{N-2} = E_{N-2} + \frac{\varepsilon}{N(N-1)}$$

$$\vdots$$

$$\widehat{E}_1 = E_1 \pm \frac{\varepsilon}{N!}.$$

The recursion is so cheap and the error damps out so quickly that we can start with a poor approximation $\widehat{E}_N$ for some large N and get accurate answers inexpensively for the E_n that really interest us. Notice that recurring in this direction, the E_n increase, making the relative errors damp out even faster. The inequality

$$0 < E_n < \int_0^1 x^n \, dx = \frac{1}{n+1}$$

shows how to easily get an approximation to E_n with an error that we can bound. For example, if we take $N = 20$, the crude approximation $\widehat{E}_{20} = 0$ has an absolute error less than 1/21 in magnitude. The magnitude of the absolute error in $\widehat{E}_{19}$ is then less than $1/(20 \times 21) = 0.0024, \ldots$, and that in $\widehat{E}_{15}$ is less than 4×10^{-8}. The approximations to $E_{14}, \ldots, E_1$ will be even more accurate.

A stable recurrence like the second algorithm is the standard way to evaluate certain mathematical functions. It can be especially convenient for a series expansion in the functions. For example, evaluation of an expansion in Bessel functions of the first kind,

$$f(x) = \sum_{n=0}^{\infty} a_n J_n(x),$$

requires the evaluation of $J_n(x)$ for many n. Using recurrence on the index n, this is accomplished very inexpensively. ■

Any real number $y \neq 0$ can be written in scientific notation as

$$y = \pm .d_1 d_2 \cdots d_s d_{s+1} \cdots \times 10^e. \tag{1.1}$$

Here there are an infinite number of digits d_i. Each d_i takes on one of the values $0, 1, \ldots, 9$ and we assume the number y is *normalized* so that $d_1 > 0$. The portion $.d_1 d_2 \ldots$ is called the *fraction* or *mantissa* or *significand*; it has the meaning

$$d_1 \times 10^{-1} + d_2 \times 10^{-2} + \cdots + d_s \times 10^{-s} + \cdots.$$

There is an ambiguity in this representation; for example, we must agree that

$$0.24000000 \cdots$$

is the same as

$$0.23999999 \cdots.$$

The quantity e in (1.1) is called the *exponent*; it is a signed integer.

Nearly all numerical computations on a digital computer are done in floating point arithmetic. This is a number system that uses a finite number of digits to approximate the real number system used for exact computation. A system with s digits and base 10 has all of its numbers of the form

$$y = \pm .d_1 d_2 \cdots d_s \times 10^e. \tag{1.2}$$

Again, for nonzero numbers each d_i is one of the digits $0, 1, \ldots, 9$ and $d_1 > 0$ for a normalized number. The exponent e also has only a finite number of digits; we assume the range

$$m \leq e \leq M.$$

The number zero is special; it is written as

$$0.0 \cdots 0 \times 10^m.$$

Example 1.4. If $s = 1$, $m = -1$, and $M = 1$, then the set of floating point numbers is

$$\begin{array}{llll} +0.1 \times 10^{-1}, & +0.2 \times 10^{-1}, & \ldots, & +0.9 \times 10^{-1} \\ +0.1 \times 10^{0}, & +0.2 \times 10^{0}, & \ldots, & +0.9 \times 10^{0} \\ +0.1 \times 10^{1}, & +0.2 \times 10^{1}, & \ldots, & +0.9 \times 10^{1}, \end{array}$$

together with the negative of each of these numbers and 0.0×10^{-1} for zero. There are only 55 numbers in this floating point number system. In floating point arithmetic the numbers are not equally spaced. This is illustrated in Figure 1.1, which is discussed after we consider number bases other than decimal. ■

Because there are only finitely many floating point numbers to represent the real number system, each floating point number must represent many real numbers. When the exponent e in (1.1) is bigger than M, it is not possible to represent y at all. If in the course of some computations a result arises that would need an exponent $e > M$, the computation is said to have *overflowed*. Typical operating systems will terminate the run on overflow. The situation is less clear when $e < m$, because such a y might reasonably be approximated by zero. If such a number arises during a computation, the computation is said to have *underflowed*. In scientific computation it is usually appropriate to set the result to zero and continue. Some operating systems will terminate the run on underflow and others will set the result to zero and continue. Those that continue may report the number of underflows at the end of the run. If the response of the operating system is not to your liking, it is usually possible to change the response by means of a system routine.

Overflows and underflows are not unusual in scientific computation. For example, $\exp(y)$ will overflow for $y > 0$ that are only moderately large, and $\exp(-y)$ will underflow. Our concern should be to prevent going out of range *unnecessarily*.

FORTRAN and C provide for integer arithmetic in addition to floating point arithmetic. Provided that the range of integers allowed is not exceeded, integer arithmetic is exact. It is necessary to beware of overflow because the typical operating system does *not* report an integer overflow; the computation continues with a number that is not related to the correct value in an obvious way.

Both FORTRAN and C provide for two precisions, that is, two arithmetics with different numbers of digits s, called single and double precision. The languages deal with mixing the various modes of arithmetic in a sensible way, but the unwary can get into trouble. This is more likely in FORTRAN than C because by default, constants in C are double precision numbers. In FORTRAN the type of a constant is taken from the

way it is written. Thus, an expression like (3/4)*5. in FORTRAN and in C means that the integer 3 is to be divided by the integer 4 and the result converted to a floating point number for multiplication by the floating point number 5. Here the integer division 3/4 results in 0, which might not be what was intended. It is surprising how often users ruin the accuracy of a calculation by providing an inaccurate value for a basic constant like π. Some constants of this kind may be predefined to full accuracy in a compiler or a library, but it should be possible to use intrinsic functions to compute accurately constants like $\pi = \text{acos}(-1.0)$.

Evaluation of an asymptotic expansion for the special function $\text{Ei}(x)$, called the exponential integral, involves computing terms of the form $n!/x^n$. To contrast computations in integer and floating point arithmetic, we computed terms of this form for a range of n and $x = 25$ using both integer and double precision functions for the factorial. Working in C on a PC using IEEE arithmetic, it was found that the results agreed through $n = 7$, but for larger n the results computed with integer arithmetic were useless—the result for $n = 8$ was *negative*! The integer overflows that are responsible for these erroneous results are truly dangerous because there was no indication from the system that the answers might not be reliable.

Example 1.5. In Chapter 4 we study the use of bisection to find a number z such that $f(z) = 0$, that is, we compute a root of $f(x)$. Fundamental to this procedure is the question, Do $f(a)$ and $f(b)$ have opposite signs? If they do, a continuous function $f(x)$ has a root z between a and b. Many books on programming provide illustrative programs that test for $f(a)f(b) < 0$. However, when $f(a)$ and $f(b)$ are sufficiently small, the product underflows and its sign cannot be determined. This is likely to happen because we are interested in a and b that tend to z, causing $f(a)$ and $f(b)$ to tend to zero. It is easy enough to code the test so as to avoid the difficulty; it is just necessary to realize that the floating point number system does not behave quite like the real number system in this test. ■

As we shall see in Chapter 4, finding roots of functions is a context in which underflow is quite common. This is easy to understand because the aim is to find a z that makes $f(z)$ as small as possible.

Example 1.6. Determinants. In Chapter 2 we discuss the solution of a system of linear equations. As a by-product of the algorithm and code presented there, the determinant of a system of n equations can be computed as the product of a set of numbers returned:

$$\det = y_1 y_2 \cdots y_n.$$

Unfortunately, this expression is prone to unnecessary under- and overflows. If, for example, $M = 100$ and $y_1 = 10^{50}$, $y_2 = 10^{60}$, $y_3 = 10^{-30}$, all the numbers are in range and so is the determinant 10^{80}. However, if we form $(y_1 \times y_2) \times y_3$, the partial product $y_1 \times y_2$ overflows. Note that $y_1 \times (y_2 \times y_3)$ *can* be formed. This illustrates the fact that floating point numbers do not always satisfy the associative law of multiplication that is true of real numbers.

The more fundamental issue is that because $\det(cA) = c^n \det(A)$, the determinant is extremely sensitive to the scale of the matrix A when the number of equations n is large. A software remedy used in LINPACK [4] in effect extends the range of exponents available. Another possibility is to use logarithms and exponentials:

$$\ln|\det| = \sum_{i=1}^{n} \ln|y_i|$$

$$|\det| = \exp(\ln|\det|).$$

If this leads to an overflow, it is because the answer cannot be represented in the floating point number system. ■

Example 1.7. Magnitude. When computing the magnitude of a complex number $z = x + iy$,

$$|z| = \sqrt{x^2 + y^2},$$

there is a difficulty when either x or y is large. Suppose that $|x| \geq |y|$. If $|x|$ is sufficiently large, x^2 will overflow and we are not able to compute $|z|$ even when it is a valid floating point number. If the computation is reformulated as

$$|z| = |x|\sqrt{1 + (y/x)^2},$$

the difficulty is avoided. Notice that underflow could occur when $|y| \ll |x|$. This is harmless and setting the ratio y/x to zero results in a computed $|z|$ that has a small relative error.

The evaluation of the Euclidean norm of a vector $\mathbf{v} = (v_1, v_2, \ldots, v_n)$,

$$||v||_2 = \left(\sum_{i=1}^{n} v_i^2 \right)^{0.5},$$

involves exactly the same kind of computations. Some writers of mathematical software have preferred to work with the maximum norm

$$||v||_\infty = \max_{1 \leq i \leq n} |v_i|,$$

because it avoids the unnecessary overflows and underflows that are possible with a straightforward evaluation of the Euclidean norm. ■

If a real number y has an exponent in the allowed range, there are two standard ways to approximate it by a floating point number $fl(y)$. If all digits after the first s in (1.1) are dropped, the result is known as a *chopped* or *truncated* representation. A floating point number that is usually closer to y can be found by adding $5 \times 10^{-(s+1)}$ to the fraction in (1.1) and then chopping. This is called *rounding*.

Example 1.8. If $m = -99$, $M = 99$, $s = 5$, and $\pi = 3.1415926\cdots$, then in chopped arithmetic

$$fl(\pi) = 0.31415 \times 10^1$$