Chapter 25
Software Issues in Floating Point Arithmetic

The first American Venus probe was lost due to a program fault caused by the inadvertent substitution of a statement of the form
\[ \text{DO 3 I = 1.3 for one of the form DO 3 I = 1, 3.} \]
— JIM HORNING, Note on Program Reliability\textsuperscript{22} (1979)

Numerical subroutines should deliver results that satisfy simple, useful mathematical laws whenever possible.

No method of solving a computational problem is really available to a user until it is completely described in an algebraic computing language and made completely reliable. Before that, there are indeterminate aspects in any algorithm.
— GEORGE E. FORSYTHE, Today’s Computational Methods of Linear Algebra (1967)

The extended precision calculation of \( \pi \) has substantial application as a test of the “global integrity” of a supercomputer . . . Such calculations . . . are apparently now used routinely to check supercomputers before they leave the factory. A large-scale calculation of \( \pi \) is entirely unforgiving; it soaks into all parts of the machine and a single bit awry leaves detectable consequences.
— J. M. BORWEIN, P. B. BORWEIN, and D. H. BAILEY, Ramanujan, Modular Equations, and Approximations to \( \pi \) or How to Compute One Billion Digits of \( \pi \) (1989)

\textsuperscript{22} Quoted, with further details, in Tropp [1021, 1984].
In this chapter we discuss some miscellaneous aspects of floating point arithmetic that have an impact on software development.

25.1. Exploiting IEEE Arithmetic

IEEE standard 754 and 854 arithmetic can be exploited in several ways in numerical software, provided that proper access to the arithmetic’s features is available in the programming environment. Unfortunately, although most commercially significant floating point processors at least nearly conform to the IEEE standards, language standards and compilers generally provide poor support (exceptions include the Standard Apple Numerics Environment (SANE) [233, 1988], Apple’s PowerPC numerics environment [234, 1994], and Sun’s SPARCstation compilers [969, 1992], [970, 1992]). We give four examples to show the benefits of IEEE arithmetic on software.

Suppose we wish to evaluate the dual of the vector \( p \)-norm \( \left( 1 \leq p \leq \infty \right) \), that is, the \( q \)-norm, where \( p^{-1} + q^{-1} = 1 \). In MATLAB notation we simply write \( \text{norm}(x, 1/(1-1/p)) \), and the extreme cases \( p = 1 \) and \( \infty \) correctly yield \( q = 1/(1-1/p) = \infty \) and 1 in IEEE arithmetic. (Note that the formula \( q = p/(p - 1) \) would not work, because \( \infty/\infty \) evaluates to a NaN.)

The following code finds the biggest element in an array \( a(1:n) \), and elegantly solves the problem of choosing a “sufficiently negative number” with which to initialize the variable max:

```plaintext
max = -inf
for j = 1:n
    if a(j) > max
        max = a(j)
    end
end
```

Any unknown or missing elements of the array \( a \) could be assigned NaNs (\( a(j) = \text{NaN} \)) and the code would (presumably) still work, since a NaN compares as unordered with everything.

Consider the following continued fraction [630, 1981]:

\[
r(x) = 7 - \frac{3}{x - 2 - \frac{1}{x - 7 + \frac{10}{x - 2 - \frac{2}{x - 3}}}}\]

which is plotted over the range \([0,4]\) in Figure 25.1. Another way to write the
rational function $r(x)$ is in the form

$$r(x) = \frac{((7x - 101)x + 540)x - 1204)x + 958}{(((x - 14)x + 72)x - 151)x + 112},$$

in which the polynomials in the numerator and denominator are written in the form in which they would be evaluated by Homer’s rule. Examining these two representations of $r$, we see that the continued fraction requires less arithmetic operations to evaluate (assuming it is evaluated in the obvious “bottom up” fashion) but it incurs division by zero at the points $x = 1:4$, even though $r$ is well behaved at these points. However, in IEEE arithmetic $r$ evaluates correctly at these points because of the rules of infinity arithmetic. For $x = 10^{77}$, the rational form suffers overflow, while the continued fraction evaluates correctly to 7.0; indeed, in IEEE arithmetic the continued fraction is immune to overflow. Figure 25.2 shows the relative errors made in evaluating $r$ in double precision on an equally spaced grid of points on the range $[0,4]$ (many of the errors for the continued fraction are zero, hence the gaps in the error curve); clearly, the continued fraction produces the more accurate function values. The conclusion is that in IEEE arithmetic the continued fraction representation is the preferred one for evaluating $r$.

The exception handling provided by IEEE arithmetic can be used to simplify and speed up certain types of software, provided that the programming environment properly supports these facilities. Recall that the exceptional
operations underflow, overflow, divide by zero, invalid operation, and inexact
deliver a result, set a flag, and continue. The flags (one for each type of excep-
tion) are “sticky”, remaining set until explicitly cleared, and implementa-
tions of the IEEE standard are required to provide a means to read and write the
flags. Unfortunately, compiler writers have been slow to make exception han-
dling, and some of IEEE arithmetic’s other unusual features, available to the
programmer. Kahan has commented that “the fault lies partly with the IEEE
standard, which neglected to spell out standard names for the exceptions, their
flags, or even $\infty$.”

As an example of how exception handling can be exploited, consider the
use of the LAPACK norm estimator (Algorithm 14.4) to estimate $||A^{-1}||_1$. The
algorithm requires the solution of linear systems of the form $Ax = b$ and
$A^Ty = c$, which is typically done with the aid of an LU factorization $PA =
LU$. Solution of the resulting triangular systems is prone to both overflow and
division by zero. In LAPACK, triangular systems are solved in this context not
with the level-2 BLAS routine $xTRSV$ but with a routine $xLATRS$ that contains
elaborate tests and scalings to avoid overflow and division by zero [18, 1991].
In an IEEE arithmetic environment a simpler and potentially faster approach
is possible: solve the triangular systems with $xTRSV$ and after computing each
solution check the flags to see whether any exceptions occurred. Provided that
some tests are added to take into account overflow due solely to $||A||_1$, being

Figure 25.2. Error in evaluating rational function $r$. Solid line: continued fraction,
dotted line: usual rational form.
tiny, the occurrence of an exception allows the conclusion that $A$ is singular to working precision; see Demmel and Li [298, 1994] for the details.

For the condition estimator that uses exception handling to be uniformly faster than the code with scaling it is necessary that arithmetic with NaNs, infinities, and subnormal numbers be performed at the same speed as conventional arithmetic. While this requirement is often satisfied, there are machines for which arithmetic on NaNs, infinities, and subnormal numbers is between one and three orders of magnitude slower than conventional arithmetic [298, 1994, Table 2]. Demmel and Li compared the LAPACK condition estimation routines $\text{xxxCON}$ with modified versions containing exception handling and found the latter to be up to 6 times faster and up to 13 times slower, depending on the machine and the matrix.

Appendices to the IEEE standards 754 and 854 recommend 10 auxiliary functions and predicates to support portability of programs across different IEEE systems. These include nextafter($x,y$), which returns the next floating point number to $x$ in the direction of $y$, and scalb($x,n$), which returns $x \times \beta^n$ without explicitly computing $\beta^n$, where $n$ is an integer and $\beta$ the base. Not all implementations of IEEE arithmetic support these functions. Portable C versions of six of the functions are presented by Cody and Coonen [226, 1993].

### 25.2. Subtleties of Floating Point Arithmetic

The difference $x - y$ of two machine numbers can evaluate to zero even when $x \neq y$, because of underflow. This makes a test such as

```plaintext
if x \neq y
  f = f / (x - y)
end
```

unreliable. However, in a system that supports gradual underflow (such as IEEE arithmetic) $x - y$ always evaluates as nonzero when $x \neq y$, as is easy to verify. On several models of Cray computer (Cray 1, 2, X-MP, Y-MP, and C90) this code could fail for another reason: they compute $f/(x - y)$ as $f \times (1/(x - y))$ and $1/(x - y)$ could overflow.

It is a general principle that one should not test two floating point numbers for equality, but rather use a test such as if $|x - y| \leq \text{tol}$ (there are exceptions, such as Algorithm 2 in §1.14.1). Of course, skill may be required to choose an appropriate value for tol. A test of this form would correctly avoid the division in the example above when underflow occurs.
Table 25.1. Results from Cholesky factorization.

<table>
<thead>
<tr>
<th>Bits</th>
<th>u</th>
<th>Computer</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 le-29</td>
<td>Cray 2</td>
<td></td>
<td>.447440341</td>
</tr>
<tr>
<td>64 le-17</td>
<td>IBM 3090</td>
<td></td>
<td>.447440344</td>
</tr>
<tr>
<td>64 le-16</td>
<td>Convex 220</td>
<td></td>
<td>.447440339</td>
</tr>
<tr>
<td>64 le-16</td>
<td>IRIS</td>
<td></td>
<td>.447440339</td>
</tr>
<tr>
<td>64 le-15</td>
<td>Cray 2</td>
<td></td>
<td>.447440303</td>
</tr>
<tr>
<td>64 le-15</td>
<td>Cray Y-MP</td>
<td></td>
<td>.447436106</td>
</tr>
</tbody>
</table>

25.3. Cray Peculiarities

Carter [188, 1991] describes how a Cray computer at the NASA Ames Laboratories produced puzzling results that were eventually traced to properties of its floating point arithmetic. Carter used Cholesky factorization on a Cray Y-MP to solve a sparse symmetric positive definite system of order 16146 resulting from a finite element model of the National Aerospace Plane. The results obtained for several computers are shown in Table 25.1, where “Displacement” denotes the largest component of the solution and incorrect digits are set in italics and underlined. Since the coefficient matrix has a condition number of order 10^{11}, the errors in the displacements are consistent with the error bounds for Cholesky factorization.

Given that both the Cray 2 and the Cray Y-MP lack guard digits, it is not surprising that they give a less accurate answer than the other machines with a similar unit roundoff. What is surprising is that, even though both machines use 64-bit binary arithmetic with a 48-bit mantissa, the result from the Cray Y-MP is much less accurate than that from the Cray 2. The explanation (diagnosed over the telephone to Carter by Kahn, as he scratched on the back of an envelope) lies in the fact that the Cray 2 implementation of subtraction without a guard digit produces more nearly unbiased results (average error zero), while the Cray Y-MP implementation produces biased results, causing \( f(x - y) \) to be too big if \( x > y > 0 \). Since the inner loop of the Cholesky algorithm contains an operation of the form \( a_{ii} = a_{ii} - a_{ij}^2 \), the errors in the diagonal elements reinforce as the factorization proceeds on the Cray Y-MP, producing a Cholesky factor with a diagonal that is systematically too large. Similarly, since Carter’s matrix has off-diagonal elements that are mostly negative or zero, the Cray Y-MP produces a Cholesky factor with off-diagonal elements that are systematically too large and negative. For the large value of \( n \) used by Carter, these two effects in concert are large enough to cause the loss of a further two digits in the answer.

An inexpensive way to improve the stability of the computed solution is to
25.4 Compilers

Some of the pitfalls a compiler writer should avoid when implementing floating-point arithmetic are discussed by Farnum [364, 1988]. The theme of his paper is that programmers must be able to predict the floating point operations that will be performed when their codes are compiled; this may require, for example, that the compiler writer forgoes certain “optimizations”. In particular, compilers should not change groupings specified by parentheses. For example, the two expressions

\[(1.0E-30 + 1.0E+30) - 1.0E+30\]
\[1.0E-30 + (1.0E+30 - 1.0E+30)\]

will produce different answers on many machines. Farnum explains that

Compiler texts and classes rarely address the peculiar problems of floating-point computation, and research literature on the topic is generally confined to journals read by numerical analysts, not compiler writers. Many production-quality compilers that are excellent in other respects make basic mistakes in their compilation of floating-point, resulting in programs that produce patently absurd results, or worse, reasonable but inaccurate results.

25.5. Determining Properties of Floating Point Arithmetic

Clever algorithms have been devised that attempt to reveal the properties and parameters of a floating point arithmetic. The first algorithms were published by Malcolm [724, 1972] (see Problem 25.3). These have been improved and extended by other authors. Kahan’s paranoia code carries out detailed investigations of a computer’s floating point arithmetic; there are Basic, C, Modula, Pascal, and Fortran versions, all available from netlib. In addition to computing the arithmetic parameters, paranoia tries to determine how well the arithmetic has been implemented (so it can be regarded as a program to test a floating point arithmetic—see the next section). Karpinski [645, 1985] gives an introduction to paranoia for the layman, but the best documentation for the code is the output it produces.

Cody has a Fortran routine macher for determining 13 parameters associated with a floating point arithmetic system [222, 1988] (an earlier version was published in the book by Cody and Waite [228, 1980]). Cody notes some
strange behaviour of compilers and says that he was unable to make his code run correctly on the Cyber 205. A routine based on machar is given in Numerical Recipes [842, 1992, §20.1].

LAPACK contains a routine xLAMCH for determining machine parameters. Because of the difficulty of handling the existing variety of machine arithmetics it is over 850 lines long (including comments and the subprograms it calls). Programs such as machar, paranoia, and xLAMCH are difficult to write; for example, xLAMCH tries to determine the overflow threshold without invoking overflow. The Fortran version of paranoia handles overflow by requiring the user to restart the program, after which checkpointing information previously written to a file is read to determine how to continue.

Fortran 90 contains environmental inquiry functions, which for REAL arguments return the precision (PRECISION\(^{23}\)), exponent range (RANGE), machine epsilon (EPSILON), largest number (HUGE), and so on, corresponding to that argument [749, 1990]. The values of these parameters are chosen by the implementor to best fit a model of the machine arithmetic due to Brown [150, 1981] (see §25.7.4). Fortran 90 also contains functions for manipulating floating point numbers: for example, to set or return the exponent or fractional part (EXPONENT, FRACTION, SET_EXPONENT) and to return the spacing of the numbers having the same exponent as the argument (SPACING).

25.6. Testing a Floating Point Arithmetic

How can we test whether a particular implementation of floating point arithmetic is correct? It is impractical to test a floating point operation with all possible arguments because there are so many of them—about \(10^{19}\) in IEEE double precision arithmetic, for example! Special techniques are therefore needed that test with a limited number of arguments.

A package FPV [784, 1986] from NAG Ltd. attempts to verify experimentally that a floating point arithmetic has been correctly implemented according to its specification. FPV must be supplied by the user with the arithmetic parameters \(b, t, e_{\text{min}}, e_{\text{max}}\), and with the rounding rule; it then attempts to verify that the arithmetic conforms to these parameters by probing for errors. The tests cover the basic arithmetic operations (including square root) but not the elementary functions. FPV, which is available in both Fortran 77 and Pascal versions, adopts an approach used in an earlier program FPTST [906, 1981], [907, 1984] by Schryer of AT&T Bell Laboratories: it tests arithmetic operations with a limited set of operands that are regarded as being most likely to reveal errors. This approach is based on the premise that errors are most likely to occur as “edge effects”, induced by operands close to a discontinuity or boundary of the floating point number system (such as a power of

\(^{23}\) Intrinsic function names are shown in parentheses.
the base $\beta$).

FPV and FPTST have both revealed implementation errors in floating point arithmetics on commercial computers. Errors detected include multiplication and negation producing unnormalized results, $x \times y$ differing from $(-x) \times (-y)$, and the product of two numbers greater than 1 being less than 1! Wichmann [1078, 1992] suggests that it was probably revelations such as these that led the UK Ministry of Defence to issue an interim standard prohibiting the use of floating point arithmetic in safety-critical systems.

A Fortran package ELEFUNT by Cody and Waite contains programs to test the elementary functions [221, 1982], [228, 1980]; the package is available from netlib. It checks identities such as $\cos(x) = \cos(x/3)(4 \cos^2(x/3) - 1)$, taking care to choose arguments $x$ for which the errors in evaluating the identities are negligible. A package CELEFUNT serves the same purpose for the complex elementary functions [224, 1993]. Tang [988, 1990] develops table-driven techniques for testing the functions $\exp$ and $\log$.

### 25.7. Portability

Software is portable if it can be made run on different systems with just a few straightforward changes (ideally, we would like to have to make no changes, but this level of portability is often impossible to achieve). Sometimes the word “transportable” is used to describe software that requires certain well-defined changes to run on different machines. For example, Demmel, Dongarra, and Kahan [290, 1992] describe LAPACK as “a transportable way to achieve high efficiency on diverse modern machines”, noting that to achieve high efficiency the BLAS need to be optimized for each machine. A good example of a portable collection of Fortran codes is LINPACK. It contains no machine-dependent constants and uses the PFORT subset of Fortran 77; it uses the level-1 BLAS, so, ideally, optimized BLAS would be used on each machine.

#### 25.7.1. Arithmetic Parameters

Differences between floating point arithmetics cause portability problems. First, what is meant by REAL and DOUBLE PRECISION in Fortran varies greatly between machines, as shown by the following table:

<table>
<thead>
<tr>
<th></th>
<th>$u$ (REAL)</th>
<th>$u$ (DOUBLE PRECISION)</th>
<th>$u$ (EXTENDED)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>$\approx 10^{-7}$</td>
<td>$\approx 10^{-16}$</td>
<td>$\approx 10^{-31}$</td>
</tr>
<tr>
<td>Cray</td>
<td>$\approx 10^{-15}$</td>
<td>$\approx 10^{-28}$</td>
<td>$\approx 10^{-31}$</td>
</tr>
</tbody>
</table>

Second, for a given level of precision, $u$, the various arithmetic parameters such as base, unit roundoff, largest and smallest machine numbers, and the
type of rounding, can all vary. Some possible ways for a code to acquire these parameters are as follows.

(1) The parameters are evaluated and embedded in the program in PARAMETER and DATA statements. This is conceptually the simplest approach, but it is not portable.

(2) Functions are provided that return the machine parameters. Bell Laboratories’ PORT library [405, 1978] has three functions:

\[ \text{R1MACH}(k) \quad 1 \leq k \leq 5 \quad \text{real parameters}, \]
\[ \text{D1MACH}(k) \quad 1 \leq k \leq 5 \quad \text{double precision parameters}, \]
\[ \text{I1MACH}(k) \quad 1 \leq k \leq 16 \quad \text{integer parameters}. \]

\text{R1MACH} returns the underflow and overflow thresholds, the smallest and largest relative spacing \((\beta^{-t}, \beta^{t'})\) respectively, and \(\log_{10} \beta\), where \(\beta\) is the base and \(t\) the number of digits in the mantissa. \text{I1MACH} returns standard input, output and error units and further floating point information, such as \(\beta, t\), and the minimum and maximum exponents \(e_{\min}\) and \(e_{\max}\). These functions do not carry out any computation; they contain DATA statements with the parameters for most common machines in comment lines, and the relevant statements have to be uncommented for a particular machine. This approach is more sensible for a library than the previous one, because only these three routines have to be edited, instead of every routine in the library.

The NAG Library takes a similar approach to PORT. Each of the 18 routines in its X02 chapter returns a different machine constant. For example, \text{X02AJF} returns the unit roundoff and \text{X02ALF} returns the largest positive floating point number. These values are determined once and for all when the NAG library is implemented on a particular platform using a routine similar to paranoia and machar, and they are hard coded into the Chapter X02 routines.

(3) The information is computed at run-time, using algorithms such as those described in §25.5.

25.7.2. 2 × 2 Problems in LAPACK

LAPACK contains 10 or so auxiliary routines for solving linear equation and eigenvalue problems of dimension 2. For example, \text{SLAS2} computes the singular values of a 2 × 2 triangular matrix, \text{SLAEV2} computes the eigenvalues and eigenvectors of a 2 × 2 symmetric matrix, and \text{SLASY2} solves a Sylvester equation \(AX - XB = C\) where \(A\) and \(B\) have order 1 or 2. These routines need to be reliable and robust because they are called repeatedly by higher-level solvers. Because LAPACK makes minimal assumptions about the underlying floating point arithmetic, the 2 × 2 routines are nontrivial to write and are surprisingly long: the counts of executable statements are 39 for \text{SLAS2}, 61 for
25.7 Portability

SLAEV2, and 207 for SLASY2. If the availability of extended precision arithmetic (possibly simulated using the working precision) or IEEE arithmetic can be assumed, the codes can be simplified significantly. Complicated and less efficient code for these 2 x 2 problem solvers is a price to be paid for portability across a wide range of floating point arithmetics.

25.7.3. Numerical Constants

How should numerical constants be specified, e.g., for a Gauss quadrature rule or a Runge–Kutta method? Some compilers limit the number of significant digits that can be specified in a DATA statement or an assignment statement. One possibility is to use rational approximations, but again the number of digits required to cater for all precision may be too large for some compilers. Another scheme is to store a carefully designed table of leading and trailing parts of constants. A constant such as $\pi$ is best computed via a statement such as $\pi = 4.0 \times \text{atan}(1.0)$, if you trust your language’s mathematics library.

25.7.4. Models of Floating Point Arithmetic

The model (2.4) of floating point arithmetic contains only one parameter, the unit roundoff. Other more detailed models of floating point arithmetic have been proposed, with the intention of promoting the development of portable mathematical software. A program developed under a model and proved correct for that model has the attraction that it necessarily performs correctly on any machine for which the model is valid.

The first detailed model was proposed by van Wijngaarden [1046, 1966]; it contained 32 axioms and was unsuccessful because it was “mathematically intimidating” and did not cover the CDC 600, an important high-performance computer of that time [635, 1991]. A more recent model is that of Brown [150, 1981]. Brown’s model contains four parameters: the base $\beta$, the precision $t$, and the minimum and maximum exponents $e_{\min}$ and $e_{\max}$ together with a number of axioms describing the behaviour of the floating point arithmetic. Aberrant arithmetics are accommodated in the model by penalizing their parameters (for example, reducing $t$ by 1 from its actual machine value if there is no guard digit). Brown builds a substantial theory on the model and gives an example program to compute the 2-norm of a vector, which is accompanied by a correctness proof for the model.

Brown’s model is intended to cater to diverse computer arithmetics. In fact, “any behaviour permitted by the axioms of the model is actually exhibited by at least one commercially important computer” [150, 1981, p. 457]. This broadness is a weakness. It means that there are important properties shared by many (but not all) machines that the model cannot reflect, such as the exactness of $fl(x - y)$ described in Theorem 2.4. As Kahan [630,
1981] notes, “Programming for the [IEEE] standard is like programming for one of a family of well-known machines, whereas programming for a model is like programming for a horde of obscure and ill-understood machines all at once.” Although Brown’s model was used in the Ada language to provide a detailed specification of floating point arithmetic, the model is still somewhat controversial.

Wichmann [1077, 1989] gives a formal specification of floating point arithmetic in the VDM specification language based on a modification of Brown’s model.

The most recent model is the Language Independent Arithmetic (LIA-1) [702, 1993]. The LIA-1 specifies floating point arithmetic far more tightly than Brown’s model. It, too, is controversial; an explanation of flaws in an earlier version (known then as the Language Compatible Arithmetic Standard) was published by Kahan [635, 1991].

For a more detailed critique of models of floating point arithmetic see Priest [844, 1992].

25.8. Avoiding Underflow and Overflow

A classic example showing how care is needed to avoid underflow and overflow is the evaluation of a vector 2-norm, $\|x\|_2 = \sum_i (|x_i|^2)^{1/2}$. For about half of all machine numbers $x$, $x^2$ either underflows or overflows! Overflow is avoided by the following algorithm:

$$\begin{align*}
t &= \|x\|_\infty \\
 s &= 0 \\
 &\text{for } i = 1:n \\
 &\quad s = s + (x(i)/t)^2 \\
 &\text{end} \\
\|x\|_2 &= t\sqrt{s}
\end{align*}$$

The trouble with this algorithm is that it requires $n$ divisions and two passes over the data (the first to evaluate $\|x\|_\infty$), so it is slow. (It also involves more rounding errors than the unscaled evaluation, which could be obviated by scaling by a power of the machine base.) Blue [128, 1978] develops a one-pass algorithm that avoids both overflow and underflow and requires between $O$ and $n$ divisions, depending on the data, and he gives an error analysis to show that the algorithm produces an accurate answer. The idea behind the algorithm is simple. The sum of squares is collected in three accumulators, one each for small, medium, and large numbers. After the summation, various logical tests are used to decide how to combine the accumulators to obtain the final answer.

The original, portable implementation of the level-1 BLAS routine $xNRM2$ (listed in [307, 1979]) was written by C. L. Lawson in 1978 and, according to
Lawson, Hanson, Kincaid, and Krogh [694, 1979], makes use of Blue’s ideas. However, \texttt{xNRM2} is not a direct coding of Blue’s algorithm and is extremely difficult to understand—a classic piece of “Fortrans spaghetti!” Nevertheless, the routine clearly works and is reliable, because it has been very widely used without any reported problems. Lawson’s version of \texttt{xNRM2} has now been superseded in the LAPACK distribution by a concise and elegant version by S. J. Hammarling, which implements a one-pass algorithm; see Problem 25.5.

A special case of the vector 2-norm is the Pythagorean sum \(\sqrt{a^2 + b^2}\), which occurs in many numerical computations. One way to compute it is by a beautiful cubically convergent, square-root-free iteration devised by Moler and Morrison [773, 1983; see Problem 25.6. LAPACK has a routine \texttt{xLAPY2} that computes \(\sqrt{x_1^2 + x_2^2}\) and another routine \texttt{xLAPY3} that computes \(\sqrt{x_1^2 + x_2^2 + x_3^2}\); both routines avoid overflow by using the algorithm listed at the start of this section with \(n = 2\) or 3.

Pythagorean sums arise in computing the 1-norm of a complex vector:

\[
\|x\|_1 = \sum_{i=1}^{n} \sqrt{(\text{Re } x_i)^2 + (\text{Im } x_i)^2}, \quad x \in \mathbb{C}^n.
\]

The level-1 BLAS routine \texttt{xCASUM} does not compute the 1-norm, but the more easily computed quantity

\[
\sum_{i=1}^{n} (|\text{Re } x_i| + |\text{Im } x_i|).
\]

The reason given by the BLAS developers is that it was assumed that users would expect \texttt{xCASUM} to compute a less expensive measure of size than the 2-norm [694, 1979]. This reasoning is sound, but many users have been confused not to receive the true 1-norm. See Problem 6.16 for more on this pseudo 1-norm.

Another example where underflow and overflow can create problems is in complex division. If we use the formula

\[
\frac{a + ib}{c + id} = \frac{ac + bd}{c^2 + d^2} + i \frac{bc - ad}{c^2 + d^2},
\]

then overflow will occur whenever \(\text{cord}\) exceeds the square root of the overflow threshold, even if the quotient \((a + ib)/(c + id)\) does not overflow. Certain Cray and NEC machines implement complex division in this way [290, 1992]; on these machines the exponent range is effectively halved from the point of view of the writer of robust software. Smith [929, 1962] suggests a simple way to avoid overflow: if \(|c| \geq |d|\) use the following formula, obtained by multiplying the numerators and denominators by \(c^{-1}\),

\[
\frac{a + ib}{c + id} = \frac{a + b(dc^{-1})}{c + d(dc^{-1})} + i \frac{b - a(dc^{-1})}{c + d(dc^{-1})},
\]

(25.1)
and if $|d| \geq |c|$ use the analogous formula involving $d^{-1}$. Stewart [947, 1985] points out that underflow is still possible in these formulae, and suggests a more complicated algorithm that avoids both underflow and overflow.

Demmel [280, 1984] discusses in detail the effects of underflow on numerical software and analyses different ways of implementing underflows, the main two of which are flush to zero and gradual underflow (as used in IEEE standard arithmetic). Cody [222, 1988] makes the following observations:

The variety in the treatment of underflow is wondrous. There exist machines in which the exponent wraps around and underflow results in a large floating-point number, perhaps with an accompanying sign change. There are also machines on which underflow is replaced by the smallest in magnitude nonzero number, machines on which small positive numbers vanish when added to themselves, and machines on which such small numbers vanish when multiplied by 1.0. Finally, on IEEE-style machines, underflow may be graceful.

25.9. Multiple Precision Arithmetic

If we wish to work at a precision beyond that provided by a Fortran compiler’s DOUBLE PRECISION (or the quadruple precision supported as an extension by some compilers) there are two basic approaches. The first is to implement higher precision arithmetic using arrays of double precision variables, where the unevaluated sum of the elements of an array represents a high precision variable. Arithmetic can be implemented in this system using techniques such as those of Dekker [275, 1971] (cf. compensated summation, described in §4.3), as refined by Linnainmaa [706, 1981] and Priest [843, 1991], [844, 1992]. These techniques require a “well-behaved” floating point arithmetic, and are not easy to apply.

The second approach is to use arrays to represent numbers in “standard” floating point form with a large base and a long mantissa spread across the elements of an array. All the software described in this section is of the second type.

The first major piece of multiple precision Fortran software was Brent’s package [145, 1978], [146, 1978]. This is a portable collection of Fortran 66 subroutines with wide capabilities, including the evaluation of special functions. Brent’s package represents multiple precision numbers as arrays of integers and operates on them with integer arithmetic. An interface between Brent’s package and the Fortran precompiled Augment [254, 1979] is described by Brent, Hooper, and Yohe [147, 1980]; it makes Brent’s package much easier to use.
Recently, Bailey has written a multiple precision arithmetic package MP-FUN [44, 1993], which consists of about 10,000 lines of Fortran 77 code in 87 subprograms. In MPFUN, a multiprecision (MP) number is a vector of single precision floating point numbers; it represents a number in base $2^{24}$ (for IEEE arithmetic). Complex multiprecision numbers are also supported.

MPFUN routines are available to perform the basic arithmetic operations, evaluate $n$th roots and transcendental functions, compare MP numbers, produce a random MP number, solve polynomial equations, and perform other operations. For many of these routines both simple and advanced algorithms are provided; the advanced algorithms are intended for computations at precision levels above 1000 digits. One advanced algorithm is a fast Fourier transform technique for evaluating the convolution involved in a multiplication. Another is used for division: $x/y$ is evaluated as $x \times (1/y)$, where $1/y$ is evaluated by Newton’s method (see Problem 2.26), which involves only multiplications. An interesting aspect of Newton’s method in this context is that it can be implemented with a precision level that doubles at each stage, since the iteration damps out any errors. Another interesting feature of a variable precision environment is that numerical instability can be tackled simply by increasing the precision. As an example, MPFUN does complex multiplication using the 3M method

$$(a + ib)(c + id) = ac - bd + i[(a + b)(c + d) - ac - bd],$$

which uses three real multiplications instead of the usual four. As we saw in §22.2.4, the 3M method produces a computed imaginary part that can have large relative error, but this instability is not a serious obstacle to its use in MPFUN.

Bailey provides a translator that takes Fortran 77 source containing directives in comments that specify the precision level and which variables are to be treated as multiprecision, and produces a Fortran 77 program containing the appropriate multiprecision subroutine calls. He also provides a Fortran 90 version of the package that employs derived data types and operator extensions [45, 1994]. This Fortran 90 version is a powerful and easy to use tool for doing high precision numerical computations.

Bailey’s own use of the packages includes high precision computation of constants such as $\pi$ [42, 1988] and Euler’s constant $\gamma$. A practical application to a fluid flow problem is described by Bailey, Krasny, and Pelz [46, 1993]. They found that converting an existing code to use MPFUN routines increased execution times by a factor of about 400 at 56 decimal digit precision, and they comment that this magnitude of increase is fairly typical for multiple precision computation.

Another recent Fortran 77 program for multiple precision arithmetic is the FM package of Smith [926, 1991]. It is functionally quite similar to Bailey’s MPFUN. No translator is supplied for converting from standard Fortran 77
source code to code that invokes FM, but Smith notes that the general purpose precompiled Augment [254, 1979] can be used for this purpose.

Fortran routines for high precision computation can also be found in Numerical Recipes [842, 1992, §20.6], and high precision numerical computation is supported by many symbolic manipulation packages, including Maple [199, 1991] and Mathematical [1109, 1991].

A Pascal-like programming language called Turing [579, 1988] developed at the University of Toronto in the 1980s is the basis of an extension called Numerical Turing, developed by Hull and his co-workers [591, 1985]. Numerical Turing includes decimal floating point arithmetic whose precision can be dynamically varied throughout the course of a program, a feature argued for in [590, 1982] and implemented in hardware in [231, 1983].

An extension to the level-2 BLAS (see Appendix D) is proposed in [313, 1988, App. B], comprising routines having the same specifications as those in the standard BLAS but which calculate in extended precision.

### 25.10. Patriot Missile Software Problem

A report from the United States General Accounting Office begins “On February 25, 1991, a Patriot missile defense system operating at Dhahran, Saudi Arabia, during Operation Desert Storm failed to track and intercept an incoming Scud. This Scud subsequently hit an Army barracks, killing 28 Americans” [1036, 1992]. The report finds that the failure to track the Scud missile was caused by a precision problem in the software.

The computer used to control the Patriot missile is based on a 1970s design and uses 24-bit arithmetic. The Patriot system tracks its targets by measuring the time it takes for radar pulses to bounce back from them. Time is recorded by the system clock in tenths of a second, but is stored as an integer. To enable tracking calculations the time is converted to a 24-bit floating point number. Rounding errors in the time conversions cause shifts in the system’s “range gate”, which is used to track the target.

On February 11, 1991 the Patriot Project Office received field data identifying a 20% shift in the Patriot system’s range gate after the system had been running continuously for 8 hours. This data implied that after 20 consecutive hours of use the system would fail to track and intercept a Scud. Modified software that compensated for the inaccurate time calculations was released on February 16 by army officials. On February 25, Alpha Battery, which was protecting the Dhahran Air Base, had been in continuous operation for over 100 hours. The inaccurate time calculations caused the range gate to shift so much that the system could not track the incoming Scud. On February 26, the next day, the modified software arrived in Dhahran. Table 25.2, taken from [1036, 1992], shows clearly how, with increasing time of operation, the
Patriot lost track of its target. Note that the numbers in Table 25.2 are consistent with a relative error of $2^{-20}$ in the computer’s representation of 0.1, this constant being used to convert from the system clock’s tenths of a second to seconds ($2^{-20}$ is the relative error introduced by chopping 0.1 to 23 bits after the binary point).

### 25.11. Notes and References

The issues discussed in this chapter are rarely treated in textbooks. One book that is worth consulting is Miller’s *Engineering of Numerical Software* [759, 1984], which is concerned with the designing of reliable numerical software and presents practical examples.

Kahan’s paper [627, 1972] is not, as the title suggests, a survey of error analysis, but treats in detail some specific problems, including solution of a quadratic equation and summation, giving error analysis, programs illustrating foibles of Fortran compilers, and ingenious workarounds.

The collection of papers improving *Floating-Point Programming* [1061, 1990] contains, in addition to an introduction to floating point arithmetic and Brown’s model, descriptions of recent work on embedding interval arithmetic with a super-accurate inner product in Pascal and ADA (see §24.4).

There are several algorithms for evaluating a continued fraction; see, for example, Blanch [123, 1964] and Press, Teukolsky, Vetterling, and Flannery [842, 1992, §5.2]. Rounding error analysis for the “bottom up” algorithm is given by Jones and Thron [617, 1974].

A standard set of names and definitions of machine parameters was proposed by the IFIP Working Group on Numerical Software (WG 2.5) [383,
1978], though these do not appear to have been widely adopted.

The use of high precision computations to investigate the Riemann hypothesis and other mathematical problems and conjectures is described by Varga [1052, 1990], who has made use of Brent’s package [145, 1978], [146, 1978].

**Problems**

25.1. (a) The following MATLAB code attempts to compute

\[
\mu := \min\{\text{floating point } x : flipt(1 + x) > 1\},
\]

under the assumption that the base \( \beta = 2 \).

```matlab
function x = mu()

x = 1;
xp1 = x + 1;
while xp1 > 1
    x = x/2;
    xp1 = x + 1;
end
x = 2*x;
```

On my workstation, running this code gives

```matlab
>> mu

ans =
2.2204e-016
```

Try this code, or a translation of it into another language, on any machines available to you. Are the answers what you expected?

(b) Under what circumstances might the code in (a) fail to compute \( \mu \)? (Hint: consider optimizing compilers.)

(c) What is the value of \( \mu \) in terms of the arithmetic parameters \( \beta \) and \( t \)? Note: the answer depends subtly on how rounding is performed, and in particular on whether double rounding is in effect; see Higham [550, 1991] and Moler [771, 1990]. On a workstation with an Intel 486DX chip (in which double rounding does occur), the following behaviour is observed in MATLAB:

```matlab
>> format hex; format compact % Hexadecimal format.

>> x = 2^(-53) + 2^(-64) + 2^(-105) ; y = [1+x 1 x]

y =
3ff000000000001 3ff000000000000 3ca002000000001
```
\[ \begin{align*} \gg x &= 2^{(-53)} + 2^{(-64)} + 2^{(-106)}; \quad y \sim [1+x \ 1 \ x] \\
Y &= \begin{bmatrix} 3ff0000000000000 & 3ff0000000000000 & 3ca0020000000000 \end{bmatrix} \end{align*} \]

**25.2.** Show that Smith’s formulae (25.1) can be derived by applying Gaussian elimination with partial pivoting to the 2 x 2 linear system obtained from \((c + id)(x + iy) = a + ib\).

**25.3.** The following MATLAB function implements an algorithm of Malcolm [724, 1972] for determining the floating point arithmetic parameters \(\beta\) and \(t\).

```matlab
function [beta, t] = param(x)
% a and b are floating point variables.

a = 1;
while (a+1) - a == 1
    a = 2*a;
end
b=2;
while (a+b) == a
    b = 2*b;
end
beta = - a;
t = 1;
a = beta;
while (a+1) - a == 1
    t = t+1;
    a = a*beta;
end
```

Run this code, or a translation into another language, on any machines available to you. Explain how the code works. (Hint: consider the integers that can be represented exactly as floating point numbers.) Under what circumstances might it fail to produce the desired results?

**25.4.** Hough [585, 1989] formed a 512 x 512 matrix using the following Fortran random number generator:

```fortran
subroutine matgen (a, lda, n, b, norms)

REAL a(lda,1), b(1), norms

init = 1325
norms = 0.0

do 30 j = 1,n
    do 20 i = 1,n
        \ldots
    end
30 continue
```

```
init = mod(3125*init,65536)
a(i, j) = (init - 32768 .0)/16384.0
norms = max(a(i, j), norms)
20 continue
30 continue

He then solved a linear system using this matrix, on a workstation that uses IEEE single precision arithmetic. He found that the program took an inordinate amount of time to run and tracked the problem to underflows. On the system in question underflows are trapped and the correct result recomputed very slowly in software. Hough picks up the story:

I started printing out the pivots in the program. They started out as normal numbers like 1 or –10, the suddenly dropped to about 1e-7, then later to 1e-14, and then:

\[
\begin{align*}
\text{k 82 pivot } & -1.8666e-20 & \text{k 98 pivot } & 1.22101e-21 \\
\text{k 83 pivot } & -2.96595e-14 & \text{k 99 pivot } & -7.12407e-22 \\
\text{k 84 pivot } & 2.46156e-14 & \text{k 100 pivot } & -1.75579e-21 \\
\text{k 85 pivot } & 2.40541e-14 & \text{k 101 pivot } & 3.13343e-21 \\
\text{k 86 pivot } & -4.99053e-14 & \text{k 102 pivot } & -6.99946e-22 \\
\text{k 87 pivot } & 1.7579e-14 & \text{k 103 pivot } & 3.82048e-22 \\
\text{k 88 pivot } & 1.69295e-14 & \text{k 104 pivot } & 8.05538e-22 \\
\text{k 89 pivot } & -1.56396e-14 & \text{k 105 pivot } & -1.18164e-21 \\
\text{k 90 pivot } & 1.37869e-14 & \text{k 106 pivot } & -6.349e-22 \\
\text{k 91 pivot } & -3.10221e-14 & \text{k 107 pivot } & -2.48245e-21 \\
\text{k 92 pivot } & 2.35206e-14 & \text{k 108 pivot } & -8.89452e-22 \\
\text{k 93 pivot } & 1.32175e-14 & \text{k 109 pivot } & -8.23235e-22 \\
\text{k 94 pivot } & -7.77593e-15 & \text{k 110 pivot } & 4.40549e-21 \\
\text{k 95 pivot } & 1.34815e-14 & \text{k 111 pivot } & 1.12387e-21 \\
\text{k 96 pivot } & -1.02589e-21 & \text{k 112 pivot } & -4.78853e-22 \\
\text{k 97 pivot } & 4.27131e-22 & \text{k 113 pivot } & 4.38739e-22 \\
& & \text{k 114 pivot } & 7.3868e-28 \\
\end{align*}
\]

SIGFPE 8: numerical exception, CHK, or TRAP
stopped at daxpy+0x18c: movl a40(0xe10),a30

Explain this behaviour.

25.5. The version of the level-1 BLAS routine xNRM2 that is distributed with LAPACK implements the following algorithm for computing \( \| x \|_2, x \in \mathbb{R}^n \):

\[
\begin{align*}
t &= 0 \\
s &= 1 \\
\text{for } i &= 1:n
\end{align*}
\]
Prove that the algorithm works and describe its properties.

25.6. (Moler and Morrison [773, 1983], Dubrulle [323, 1983]) This MATLAB M-file computes the Pythagorean sum $\sqrt{a^2 + b^2}$ using an algorithm of Moler and Morrison.

function p = pythag(a, b)

%PYTHAG Pythagorean sum.

P = max(abs(a), abs(b));
q = min(abs(a), abs(b));
while 1
    r = (q/P)^2;
    if r+4 == 4, return, end
    s = r/(4+ r);
    p = p+2*s*p;
    q = s*q;
    fprintf('p = %19.15e, q = %19.15e
', p, q)
end

The algorithmic immune to underflow and overflow (unless the result overflows), is very accurate, and converges in at most three iterations, assuming the unit roundoff $u \geq 10^{-20}$.

Example:

p = pythag(3,4); (p-5)/5

p = 4.986301369863014e+000, q = 3.698630136986301e-001
p = 4.999999999999999e+000, q = 5.080526329415358e-004
p = 5.000000000000000e+000, q = 1.311372652397091e-012
ans = 1.7764e-016

The purpose of this problem is to show that pythag is Halley’s method applied to a certain equation. Halley’s method for solving $f(x) = 0$ is the iteration

$$x_{n+1} = x_n - \frac{f_n/f'_n}{1 - \frac{1}{2}f'_n/(f_n)^2}.$$
where \( f_n, f'_n \) and \( f''_n \) denote the values of \( f \) and its first two derivatives at \( x_n \).

(a) For given \( x_0 \) and \( y_0 \) such that \( 0 < y_0 \leq x_0 \), the Pythagorean sum \( p = \sqrt{x_0^2 + y_0^2} \) is a root of the equation \( f(x) = x^2 - p^2 = 0 \). Show that Halley's method applied to this equation gives

\[
x_{n+1} = x_n \left( 1 + \frac{2p^2 - x_n^2}{p^2 + 3x_n^2} \right).
\]

Show that if \( 0 \leq x_n \leq p = \sqrt{x_0^2 + y_0^2} \) then \( x_n \leq x_{n+1} \leq p \) for all \( n \). Deduce that \( y_n := \sqrt{p^2 - x_n^2} \) is defined and that

\[
x_{n+1} = x_n \left( 1 + \frac{y_n^2}{4x_n^2 + y_n^2} \right),
\]

\[
y_{n+1} = \frac{y_n^3}{4x_n^2 + y_n^2}.
\]

Confirm that pythag implements a scaled version of these equations.

(b) Show that

\[
p - x_{n+1} = \frac{(p - x_n)^3}{p^2 + 3x_n^2},
\]

which displays the cubic convergence of the iteration.

25.7. (Incertis [604, 1985]) (a) Given a skew-symmetric matrix \( Y \in \mathbb{R}^{n \times n} \) with \( \|Y\|_2 \leq 1 \), show that there is a real, symmetric \( X \) satisfying \( X^2 = I + Y^2 \) such that \( X + Y \) is orthogonal.

(b) Consider the following iteration, for \( P_0, Q_0 \in \mathbb{R}^{n \times n} \) with \( \|P_0\|_2 \geq \|Q_0\|_2 \), which generalizes to matrices the "pythag" iteration in Problem 25.6.

\[
\begin{align*}
R_k &= (Q_k P_k^{-1})^2 \\
S_k &= R_k(4I + R_k)^{-1} \\
P_{k+1} &= P_k + 2S_k P_k \\
Q_{k+1} &= S_k Q_k
\end{align*}
\]

Show that this iteration can be used to compute \( X \) in part (a).

(c) Investigate the convergence of the iteration in (b) for general \( P_0 \) and \( Q_0 \).

25.8. Why might we prefer the expression \( \sqrt{|x|}/\sqrt{3} \) to \( \sqrt{|x|}/3 \) in an algorithm intended to be robust in floating point arithmetic?