19 Errors and Error Control

"Sometimes it is useful to know how large your zero is."
ANONYMOUS

Machine arithmetic is inexact in two ways. First, many numbers of interest, such as $\sqrt{2}$ or π, do not have exact representations. Second, floating-point operations, even when performed on exactly representable numbers, may lead to errors in the results. It is essential for arithmetic designers and serious computer users to understand the nature and extent of such errors, as well as how they can lead to results that are counterintuitive and, occasionally, totally invalid. Chapter topics include:

19.1 SOURCES OF COMPUTATIONAL ERRORS

Integer arithmetic is exact and all integer results can be trusted to be correct as long as overflow does not occur (assuming that the hardware was designed and built correctly and has not since failed; flaw- and fault-induced errors are dealt with in Chapter 27). Floating-point arithmetic, on the other hand, only approximates exact computations with real numbers. There are two sources of errors: (1) representation errors and (2) arithmetic errors.

Representation errors occur because many real numbers do not have exact machine representations. Examples include 1/3, $\sqrt{2}$, and π. Arithmetic errors, on the other hand,

occur because some results are inherently inexact or need more bits for exact representation than are available. For example, a given exact operand may not have a finitely representable square root, and multiplication produces a double-width result that must be rounded to single-width format.

Thus, familiarity with representation and arithmetic errors, as well as their propagation and accumulation in the course of computations, is important for the design of arithmetic algorithms and their realizations in hardware, firmware, or software. Example 19.1 illustrates the effect of representation and computation errors in floating-point arithmetic.

■ **EXAMPLE 19.1** Consider the decimal computation $1/99 - 1/100$, using a decimal floating-point format with a four-digit significand in [1, 10) and a single-digit signed exponent. Given that both 99 and 100 have exact representations in the given format, the floating-point divider will compute 1/99 and 1/100 accurately to within the machine precision:

$$\begin{aligned} x = 1/99 &\approx 1.010 \times 10^{-2} \qquad \text{error} \approx 10^{-6} \text{ or } 0.01\% \\ y = 1/100 &= 1.000 \times 10^{-2} \qquad \text{error} = 0 \end{aligned}$$

The precise result is 1/9900, with its floating-point representation 1.010×10^{-4} containing an approximate error of 10^{-8} or 0.01%. However, the floating-point subtraction $z = x -_{\text{fp}} y$ yields the result

$$z = 1.010 \times 10^{-2} - 1.000 \times 10^{-2} = 1.000 \times 10^{-4}$$

which has a much larger error of around 10^{-6} or 1%.

A floating-point number representation system may be characterized by a radix r (which we assume to be the same as the exponent base b), a precision p in terms of radix-r digits, and an approximation or "rounding" scheme A. We symbolize such a floating-point system as

$$\text{FLP}(r, p, \text{A})$$

where A $\in$ {chop, round, rtne, chop(g), …}; "rtne" stands for "round to nearest even" and chop(g) for a chopping method with g guard digits kept in all intermediate steps. Rounding schemes were discussed in Section 17.5.

Let $x = r^e s$ be an unsigned real number, normalized such that $1/r \leq s < 1$, and x_{fp} be its representation in FLP(r, p, A). Then

$$x_{\text{fp}} = r^e s_{\text{fp}} = (1 + \eta)x$$

where

$$\eta = \frac{x_{\text{fp}} - x}{x} = \frac{s_{\text{fp}} - s}{s}$$

is the relative representation error. One can establish bounds on the value of η:

$$\begin{array}{lll} \text{A} = \text{chop} & -ulp < s_{\text{fp}} - s \leq 0 & r \times ulp < \eta \leq 0 \\ \text{A} = \text{round} & -ulp/2 < s_{\text{fp}} - s \leq ulp/2 & |\eta| \leq r \times ulp/2 \end{array}$$

where $ulp = r^{-p}$. We note that the worst-case relative representation error increases linearly with r; the larger the value of r, the larger the worst-case relative error η and the greater its variations. As an example, for FLP($r = 16, p = 6$, chop), we have $|\eta| \leq 16^{-5} = 2^{-20}$. Such a floating-point system uses a 24-bit fractional significand. To achieve the same bound for $|\eta|$ in FLP($r = 2, p$, chop), we need $p = 21$.

Arithmetic in FLP(r, p, A) assumes that an infinite-precision result is obtained and then chopped, rounded, …, to the available precision. Some real machines approximate this process by keeping $g > 0$ guard digits, thus doing arithmetic in FLP(r, p, chop(g)). In either case, the result of a floating-point arithmetic operation is obtained with a relative error that is bounded by some constant η, which depends on the parameters r and p and the approximation scheme A. Consider multiplication, division, addition, and subtraction of the positive operands

$$x_{\text{fp}} = (1 + \sigma)x \qquad \text{and} \qquad y_{\text{fp}} = (1 + \tau)y$$

with relative representation errors σ and τ, respectively, in FLP(r, p, A). Note that the relative errors σ and τ can be positive or negative.

For the multiplication operation $x \times y$, we can write

$$\begin{aligned} x_{\text{fp}} \times_{\text{fp}} y_{\text{fp}} &= (1 + \eta)x_{\text{fp}}y_{\text{fp}} = (1 + \eta)(1 + \sigma)(1 + \tau)xy \\ &= (1 + \eta + \sigma + \tau + \eta\sigma + \eta\tau + \sigma\tau + \eta\sigma\tau)xy \\ &\approx (1 + \eta + \sigma + \tau)xy \end{aligned}$$

where the last expression is obtained by ignoring second- and third-order error terms. We see that in multiplication, relative errors add up in the worst case.

Similarly, for the division operation x/y, we have

$$\begin{aligned} x_{\text{fp}} /_{\text{fp}} y_{\text{fp}} &= \frac{(1 + \eta)x_{\text{fp}}}{y_{\text{fp}}} = \frac{(1 + \eta)(1 + \sigma)x}{(1 + \tau)y} \\ &= (1 + \eta)(1 + \sigma)(1 - \tau)(1 + \tau^2)(1 + \tau^4)(\cdots)\,\frac{x}{y} \\ &\approx (1 + \eta + \sigma - \tau)\,\frac{x}{y} \end{aligned}$$

So, relative errors add up in division just as they do in multiplication. Note that the negative sign of τ in the last expression above is of no consequence, given that each of the three relative error terms can be positive or negative.

Now, let's consider the addition operation $x + y$:

$$x_{\text{fp}} +_{\text{fp}} y_{\text{fp}} = (1+\eta)(x_{\text{fp}} + y_{\text{fp}}) = (1+\eta)(x + \sigma x + y + \tau y)$$
$$= \left[(1+\eta)\left(1 + \frac{\sigma x + \tau y}{x+y}\right)\right](x+y)$$

Since $|\sigma x + \tau y| \leq \max(|\sigma|, |\tau|)(x+y)$, the magnitude of the worst-case relative error in the computed sum is upper-bounded by $|\eta| + \max(|\sigma|, |\tau|)$.

Finally, for the subtraction operation $x - y$, we have

$$x_{\text{fp}} -_{\text{fp}} y_{\text{fp}} = (1+\eta)(x_{\text{fp}} - y_{\text{fp}}) = (1+\eta)(x + \sigma x - y - \tau y)$$
$$= \left[(1+\eta)\left(1 + \frac{\sigma x - \tau y}{x-y}\right)\right](x-y)$$

Unfortunately, $(\sigma x - \tau y)/(x-y)$ can be very large if x and y are both large but $x - y$ is relatively small (recall that τ can be negative). The arithmetic error η is also unbounded for subtraction without guard digits, as we will see shortly. Thus, unlike the three preceding operations, no bound can be placed on the relative error when numbers with like signs are being subtracted (or numbers with different signs are added). This situation is known as cancellation or loss of significance.

The part of the problem that is due to η being large can be fixed by using guard digits, as suggested by the following result.

THEOREM 19.1 In FLP$(r, p, \text{chop}(g))$ with $g \geq 1$ and $-x < y < 0 < x$, we have

$$x +_{\text{fp}} y = (1+\eta)(x+y) \qquad \text{with} \qquad -r^{-p+1} < \eta < r^{-p-g+2}$$

Proof: The left-hand side of the inequality is just the worst-case effect of chopping that makes the result smaller than the exact value. The only way that the result can be larger than $x + y$ is if we right-shift y by more than g digits, thus losing some of its digits and, hence, subtracting a smaller magnitude from x. The maximum absolute error in this case is less than r^{p+g}. The right-hand side follows by noting that $x + y$ is greater than $1/r^2$: x is in $[1/r, 1)$ and the shifted y has a magnitude of at most $1/r^2$, given that it has been shifted by at least two digits.

COROLLARY: In FLP$(r, p, \text{chop}(1))$

$$x +_{\text{fp}} y = (1+\eta)(x+y) \qquad \text{with} \qquad |\eta| < r^{-p+1}$$

So, a single guard digit is sufficient to make the relative arithmetic error in floating-point addition or subtraction comparable to the representation error with truncation.

■ **EXAMPLE 19.2** Consider a decimal floating-point number system ($r = 10$) with $p = 6$ and no guard digit. The exact operands x and y are shown below along with their floating-point representations in the given system:

$$x = -0.100\,000\,000 \times 10^3 \qquad x_{\text{fp}} = -.100\,000 \times 10^3$$
$$y = -0.999\,999\,456 \times 10^2 \qquad y_{\text{fp}} = -.999\,999 \times 10^2$$

Then, $x + y = 0.544 \times 10^{-4}$ and $x_{\text{fp}} + y_{\text{fp}} = 10^{-4}$, but

$$x_{\text{fp}} +_{\text{fp}}\ y_{\text{fp}} = .100\,000 \times 10^3 -_{\text{fp}}\ .099\,999 \times 10^3 = .100\,000 \times 10^{-2}$$

The relative error of the result is thus $[10^{-3} - (0.544 \times 10^{-4})]/(0.544 \times 10^{-4}) \approx 17.38$; that is, the result is 1638% larger than the correct sum! With 1 guard digit, we get

$$x_{\text{fp}} +_{\text{fp}}\ y_{\text{fp}} = .100\,000\,0 \times 10^3 -_{\text{fp}}\ .099\,999\,9 \times 10^3 = .100\,000 \times 10^{-3}$$

The result still has a large relative error of 80.5% compared with the exact sum $x + y$; but the error is 0% with respect to the correct sum of x_{fp} and y_{fp} (i.e., what we were given to work with).

19.2 INVALIDATED LAWS OF ALGEBRA

Many laws of algebra do not hold for floating-point arithmetic (some don't even hold approximately). Such areas of inapplicability can be a source of confusion and incompatibility. For example, take the associative law of addition

$$a + (b + c) = (a + b) + c$$

If the associative law of addition does not hold, as we will see shortly, then an optimizing compiler that changes the order of operations in an attempt to reduce the delays resulting from data dependencies may inadvertently change the result of the computation.

The following example shows that the associative law of addition does not hold for floating-point computations, even in an approximate sense:

$$a = 0.123\,41 \times 10^5 \qquad b = -0.123\,40 \times 10^5 \qquad c = 0.143\,21 \times 10^1$$

$$\begin{aligned} a +_{\text{fp}}\ (b +_{\text{fp}}\ c) &= (0.123\,41 \times 10^5) +_{\text{fp}}\ \left[(-0.123\,40 \times 10^5) +_{\text{fp}}\ (0.143\,21 \times 10^1)\right] \\ &= (0.123\,41 \times 10^5) -_{\text{fp}}\ (0.123\,39 \times 10^5) = 0.200\,00 \times 10^1 \end{aligned}$$

$$\begin{aligned} (a +_{\text{fp}}\ b) +_{\text{fp}}\ c &= \left[(0.123\,41 \times 10^5) -_{\text{fp}}\ (0.123\,40 \times 10^5)\right] +_{\text{fp}}\ (0.143\,21 \times 10^1) \\ &= (0.100\,00 \times 10^1) +_{\text{fp}}\ (0.143\,21 \times 10^1) = 0.243\,21 \times 10^1 \end{aligned}$$

The two results $0.200\ 00 \times 10^1$ and $0.243\ 21 \times 10^1$ differ by about 20%. So the associative law of addition does not hold.

One way of dealing with the preceding problem is to use unnormalized arithmetic. With unnormalized arithmetic, intermediate results are kept in their original form (except as needed to avoid overflow). So normalizing left shifts are not performed. Let us redo the two computations using unnormalized arithmetic:

$$\begin{aligned} a +_{\text{fp}} (b +_{\text{fp}} c) &= (0.123\ 41 \times 10^5) +_{\text{fp}} \left[(-0.123\ 40 \times 10^5) +_{\text{fp}} (0.143\ 21 \times 10^1)\right] \\ &= (0.123\ 41 \times 10^5) -_{\text{fp}} (0.123\ 39 \times 10^5) = 0.000\ 02 \times 10^5 \\ (a +_{\text{fp}} b) +_{\text{fp}} c &= \left[(0.123\ 41 \times 10^5) -_{\text{fp}} (0.123\ 40 \times 10^5)\right] +_{\text{fp}} (0.143\ 21 \times 10^1) \\ &= (0.000\ 01 \times 10^5) +_{\text{fp}} (0.143\ 21 \times 10^1) = 0.000\ 02 \times 10^5 \end{aligned}$$

Not only are the two results the same but they carry with them a kind of warning about the extent of potential error in the result. In other words, here we know that our result is correct to only one significant digit, whereas the earlier result $(0.243\ 21 \times 10^1)$ conveys five digits of accuracy without actually possessing it. Of course the results will not be identical in all cases (i.e., the associative law still does not hold), but the user is warned about potential loss of significance.

The preceding example, with normalized arithmetic and two guard digits, becomes

$$\begin{aligned} a +_{\text{fp}} (b +_{\text{fp}} c) &= (0.123\ 41 \times 10^5) +_{\text{fp}} \left[(-0.123\ 40 \times 10^5) +_{\text{fp}} (0.143\ 21 \times 10^1)\right] \\ &= (0.123\ 41 \times 10^5) -_{\text{fp}} (0.123\ 385\ 7 \times 10^5) = 0.243\ 00 \times 10^1 \\ (a +_{\text{fp}} b) +_{\text{fp}} c &= \left[(0.123\ 41 \times 10^5) -_{\text{fp}} 0.123\ 40 \times 10^5)\right] +_{\text{fp}} (0.143\ 21 \times 10^1) \\ &= (0.100\ 00 \times 10^1) +_{\text{fp}} (0.143\ 21 \times 10^1) = 0.243\ 21 \times 10^1 \end{aligned}$$

The difference has now been reduced to about 0.1%; the error is much better but still too high to be acceptable in practice.

Using more guard digits will improve the situation but the associative law of addition still cannot be assumed to hold in floating-point arithmetic. Here are some other laws of algebra that do not hold in floating-point arithmetic:

Associative law of multiplication	$a \times (b \times c) = (a \times b) \times c$
Cancellation law (for $a > 0$)	$a \times b = a \times c$ implies $b = c$
Distributive law	$a \times (b + c) = (a \times b) + (a \times c)$
Multiplication canceling division	$a \times (b/a) = b$

Before the IEEE 754-1985 floating-point standard became available and widely adopted, the preceding problem was exacerbated by different ranges and precisions in the floating-point representation formats of various computers. Now, with standard representation, one of the sources of difficulties has been removed, but the fundamental problems persist.

Because laws of algebra do not hold for floating-point computations, it is desirable to determine, if possible, which of several algebraically equivalent computations yields

the most accurate result. Even though no general procedure exists for selecting the best alternative, numerous empirical and theoretical results have been developed over the years that help us in organizing or rearranging the computation steps to improve the accuracy of the results. We present three examples that are indicative of the methods used. Additional examples can be found in the problems at the end of the chapter.

■ **EXAMPLE 19.3** The formula $x = -b \pm d$, with $d = \sqrt{b^2 - c}$, yields the two roots of the quadratic equation $x^2 + 2bx + c = 0$. The formula can be rewritten as $x = -c/(b \pm d)$. When $b^2 >> c$, the value of d is close to $|b|$. Thus, if $b > 0$, the first formula results in cancellation or loss of significance in computing the first root $(-b + d)$, whereas no such cancellation occurs with the second formula. The second root $(-b - d)$, however, is more accurately computed based on the first formula. The roles of the two formulas are reversed for $b < 0$.

■ **EXAMPLE 19.4** The area of a triangle with sides of length a, b, and c is given by the formula $A = \sqrt{s(s-a)(s-b)(s-c)}$, where $s = (a+b+c)/2$. For ease of discussion, let $a \geq b \geq c$. When the triangle is very flat, such that $a \approx b + c$, we have $s \approx a$ and the term $s - a$ in the preceding formula causes precision loss. The following version of the formula returns accurate results, even for flat triangles:

$$A = \frac{1}{4}\sqrt{(a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))}$$

W. Kahan offers a thorough discussion of this problem [Kaha00].

■ **EXAMPLE 19.5** Consider $f(x) = (1 - \cos x)/x^2$, which has a value in the range $0 \leq f(x) < 1/2$ for all $x \neq 0$. For $x = 1.2 \times 10^{-5}$, the value of $\cos x$, rounded to 10 significant digits, is 0.999 999 999 9, yielding $f(x) = (1 - 0.999\,999\,999\,9)/(1.44 \times 10^{-10}) = 0.694\,444\,444\,4$, which is clearly a wrong answer. The source of the problem is magnification of the small error in $\cos x$ when its difference with 1 is divided by a very small number. Cancellation in this example can be avoided by rewriting our function as $f(x) = [\sin(x/2)/(x/2)]^2/2$. Using the latter formula yields $f(1.2 \times 10^{-5}) = 0.500\,000\,000\,0$, which is correct to 10 significant digits.

19.3 WORST-CASE ERROR ACCUMULATION

In a sequence of computations, arithmetic or round-off errors may accumulate. The larger the number of cascaded computation steps (that depend on results from earlier steps), the greater the chance for, and the magnitude of, accumulated errors. With rounding, errors of opposite signs tend to cancel each other out in the long run, thus leading to smaller average error in the final result. Yet one cannot count on such cancellations.

For example, in computing the inner product

$$z = \sum_{i=0}^{1023} x^{(i)} y^{(i)}$$

if each multiply-add step introduces an absolute error of $ulp/2 + ulp/2 = ulp$, the total absolute error will be 1024 *ulp* in the worst case. This is equivalent to losing 10 bits of precision. If we perform the inner-product computation by means of a fused-multiply-add operation, the upper bound on absolute error per iteration is reduced from *ulp* to $ulp/2$, which is only slightly better (losing 9 bits of precision instead of 10). As for the relative error, the situation may be worse. This is because in computing the sum of signed values, cancellations, or loss of precision, can occur in one or more intermediate steps.

The kind of worst-case analysis carried out for the preceding example is very rough, and its results are expressed in terms of the number of *significant digits* in the computation results. When cascading of computations leads to the worst-case accumulation of an absolute error of *m ulp*, the effect is equivalent to losing $\log_2 m$ bits of precision.

For our inner-product example, if we begin with 24 bits of precision, say, the result is only guaranteed to have $24 - 10 = 14$ significant digits (15, if we fused multiply-add). For more complicated computations, the worth of such a worst-case estimate decreases. At the extreme, the analysis might indicate that the result has no significant digit remaining.

An obvious cure for our inner-product example is to keep the double-width products in their entirety and add them to compute a double-width result, which is then rounded to single-width at the very last step. Now, the multiplications do not introduce any round-off error and each addition introduces a worst-case absolute error of $ulp^2/2$. Thus, the total error is bounded by $1024 \times ulp^2/2$ (or $n \times ulp^2/2$ when n product terms are involved). Therefore, provided overflow is not a problem, a highly accurate result is obtained. In fact, if n is smaller than $r^p = 1/ulp$, the result can be guaranteed accurate to within *ulp* (error of $n \times ulp^2/2 < ulp/2$ as described previously, plus $ulp/2$ for the final rounding). This is as good as one would get with infinitely precise computation and final truncation.

The preceding discussion explains the need for performing the intermediate computations with a higher precision than is required in the final result. Carrying more precision in intermediate results is in fact very common in practice; even inexpensive calculators use several "guard digits" to protect against serious error accumulation (see Section 1.2). As mentioned in Section 17.2, IEEE 754-2008 defines extended formats associated with single- and double-precision numbers for precisely this reason. Virtually all digital signal processors, which are essentially microprocessor chips designed with the goal of efficiently performing the computations commonly required in signal processing applications, have the built-in capability to compute inner products with very high precision.

Clearly, reducing the number of cascaded arithmetic operations counteracts the effect of error accumulation. So, using computationally more efficient algorithms has the double benefit of reducing both execution time and accumulated errors. However, in some cases, simplifying the arithmetic leads to problems elsewhere. A good example is found in numerical computations whose inherent accuracy is a function of a step size or grid resolution (numerical integration is a case in point). Since a smaller step size or finer grid leads to more computation steps, and thus greater accumulation of round-off errors,

there may be an optimal choice that yields the best result with regard to the worst-case total error.

Since summation of a large number of terms is a frequent cause of error accumulation in software floating-point computations, Kahan's summation algorithm or formula is worth mentioning here. To compute $s = \sum_{i=0}^{n-1} x^{(i)}$, proceed as follows (justifying this algorithm is left as an exercise):

```
s ← x^(0)
c ← 0                          {c is a correction term}
for i = 1 to n − 1 do
   y ← x^(i) − c               {subtract correction term}
   z ← s + y
   c ← (z − s) − y             {find next correction term}
   s ← z
endfor
```

We will see, at the end of Section 20.3, that similar techniques can be applied to achieve greater precision in computations, using numbers represented as pairs of floating-point values.

19.4 ERROR DISTRIBUTION AND EXPECTED ERRORS

Analyzing worst-case errors and their accumulation (as was done in Section 19.3) is an overly pessimistic approach, but it is necessary if guarantees are to be provided for the precision of the results. From a practical standpoint, however, the distribution of errors and their expected values may be more important. In this section, we review some results concerning average representation errors with chopping and rounding.

Denoting the magnitude of the worst-case or maximum relative representation error by MRRE, we recall that in Section 19.1 we established

$$\text{MRRE}(\text{FLP}(r, p, \text{chop})) = r^{-p+1}$$

$$\text{MRRE}(\text{FLP}(r, p, \text{round})) = \frac{r^{-p+1}}{2}$$

In the analysis of the magnitude of average relative representation error (ARRE), we limit our attention to positive significands and begin by defining

$$\text{ARRE}(\text{FLP}(r, p, \text{A})) = \int_{1/r}^{1} \frac{|x_{\text{fp}} - x|}{x} \frac{dx}{x \ln r}$$

where "ln" stands for the natural logarithm ($\log_e$) and $|x_{\text{fp}} - x|/x$ is the magnitude of the relative representation error for x. Multiplying this relative error by the probability density function $1/(x \ln r)$ is a consequence of the logarithmic law for the distribution of normalized significands [Tsao74]. Recall that a density function must be integrated to obtain the cumulative distribution function, prob($\varepsilon \leq z$), and that the area underneath it is 1.

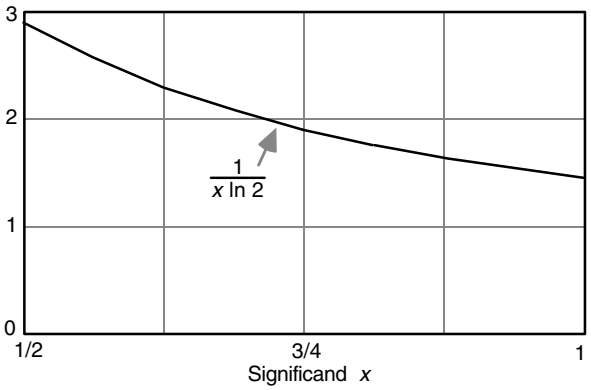


Figure 19.1 Probability density function for the distribution of normalized significands in FLP($r = 2, p,$ A).

Figure 19.1 plots the probability density function $1/(x \ln r)$ for $r = 2$. The density function $1/(x \ln r)$ essentially tells us that the probability of having a significand value in the range $[x, x + dx]$ is $dx/(x \ln r)$, thus leading to the integral above for ARRE. Note that smaller significand values are more probable than larger values.

For a first-cut approximate analysis, we can take $|x_{\text{fp}} - x|$ to be equal to $r^{-p}/2$ for FLP(r, p, chop) and $r^{-p}/4$ for FLP(r, p, round): that is, half of the respective maximum absolute errors. Then the definite integral defining ARRE can be evaluated to yield the expected errors in the two cases

$$\text{ARRE(FLP}(r, p, \text{chop})) \approx \int_{1/r}^{1} \frac{r^{-p}}{2x} \frac{dx}{x \ln r} = \frac{(r-1)r^{-p}}{2 \ln r}$$

$$\text{ARRE(FLP}(r, p, \text{round})) \approx \frac{(r-1)r^{-p}}{4 \ln r}$$

More detailed analyses can be carried out to derive probability density functions for the relative error $|x_{\text{fp}} - x|/x$ with various rounding schemes, which are then integrated to provide accurate estimates for the expected errors.

One such study [Tsao74] has yielded the following probability density functions for the relative error ε being equal to z with chopping and rounding:

$$\text{pdf}_{\text{chop}}(z) = \begin{cases} \dfrac{r^{p-1}(r-1)}{\ln r} & \text{for } 0 \le z < r^{-p} \\ \dfrac{1/z - r^{p-1}}{\ln r} & \text{for } r^{-p} \le z < r^{-p+1} \end{cases}$$

$$\text{pdf}_{\text{round}}(z) = \begin{cases} \dfrac{r^{p-1}(r-1)}{\ln r} & \text{for } |z| \le \dfrac{r^{-p}}{2} \\ \dfrac{1/(2z) - r^{p-1}}{\ln r} & \text{for } \dfrac{r^{-p}}{2} \le |z| < \dfrac{r^{-p+1}}{2} \end{cases}$$

Note the uniform distribution of the relative error at the low end and the reciprocal distribution for larger values of the relative error z. From the preceding probability density functions, the expected error can be easily derived:

$$\text{ARRE(FLP}(r, p, \text{chop})) = \int_0^{r^{-p+1}} [\text{pdf}_{\text{chop}}(z)]z\, dz = \frac{(r-1)r^{-p}}{2 \ln r}$$

$$\text{ARRE(FLP}(r, p, \text{round})) = \int_{-r^{p+1}/2}^{r^{-p+1}/2} [\text{pdf}_{\text{round}}(z)]z\, dz = \frac{(r-1)r^{-p}}{4 \ln r} \left(1 + \frac{1}{r}\right)$$

We thus see that the more rigorous analysis yields the same result as the approximate analysis in the case of chopping and a somewhat larger average error for rounding. In particular, for $r = 2$, the expected error of rounding is 3/4 (not 1/2, as the worst-case values and the approximate analysis indicate) that of chopping. These analytical observations are in good agreement with experimental results.

19.5 FORWARD ERROR ANALYSIS

Consider the simple computation $y = ax + b$ and its floating-point version

$$y_{\text{fp}} = (a_{\text{fp}} \times_{\text{fp}} x_{\text{fp}}) +_{\text{fp}} b_{\text{fp}}$$

Assuming that $y_{\text{fp}} = (1 + \eta)y$ and given the relative errors in the input operands a_{fp}, b_{fp}, and x_{fp} can we establish any useful bound on the magnitude of the relative error η in the computation result? The answer is that we cannot establish a bound on η in general, but we may be able to do it with specific constraints on the input operand ranges. The reason for the impossibility of error-bounding in general is that if the two numbers $a_{\text{fp}} \times_{\text{fp}} x_{\text{fp}}$ and b_{fp} are comparable in magnitude but different in sign, loss of significance may occur in the final addition, making the result quite sensitive to even small errors in the inputs. Example 19.2 of Section 19.1 illustrates this point.

Estimating or bounding η, the relative error in the computation result, is known as "forward error analysis": that is, finding out how far y_{fp} can be from $ax + b$, or at least from $a_{\text{fp}}x_{\text{fp}} + b_{\text{fp}}$, in the worst case. In the remainder of this section, we briefly review four methods for forward error analysis.

Automatic error analysis

For an arithmetic-intensive computation whose accuracy is suspect, one might run selected test cases with higher precision and observe the differences between the new, more precise, results and the original ones. If the computation under study is single precision, for example, one might use double-precision arithmetic, or execute on a multiprecision software package in lieu of double precision. If test cases are selected carefully and the differences resulting from automatic error analysis turn out to be insignificant, the computation is probably safe, although nothing can be guaranteed.

For an interesting example showing that the statement "using greater precision reduces computation errors or at least exposes them by producing different results" is a myth, see Problem 19.26.

Significance arithmetic

Roughly speaking, *significance arithmetic* is the same as unnormalized floating-point arithmetic, although there are some fine distinctions [Ashe59], [Metr63]. By not normalizing the intermediate computation results, except as needed to correct a significand spill, we at least get a warning when precision is lost. For example, the result of the unnormalized decimal addition

$$(.1234 \times 10^5) +_{\text{fp}} (.0000 \times 10^{10}) = .0000 \times 10^{10}$$

tells us that precision has been lost. Had we normalized the second intermediate result to true zero, we would have arrived at the misleading answer $.1234 \times 10^5$. The former answer gives us a much better feel for the potential errors.

Note that if 0.0000×10^{10} is a rounded intermediate decimal result, its infinitely precise version can be any value in $[-0.5 \times 10^6, 0.5 \times 10^6]$. Thus, the true magnitude of the second operand can be several times larger than that of the first operand. Normalization would hide this information.

Noisy-mode computation

In noisy-mode computation, (pseudo)random digits, rather than 0s, are inserted during left shifts that are performed for normalization of floating-point results. Noisy-mode computation can be either performed with special hardware support or programmed; in the latter case, significant software overhead is involved.

If several runs of the computation in noisy mode produce comparable results, loss of significance is probably not serious enough to cause problems. This is true because in various runs, different digits will be inserted during each normalization postshift. Getting comparable results from these runs is an indication that the computation is more or less insensitive to the random digits, and thus to the original digits that were lost as a result of cancellation or alignment right shifts.

Interval arithmetic

One can represent real values by intervals: an interval $[x_{\text{lo}}, x_{\text{hi}}]$ representing the real value x means that $x_{\text{lo}} \leq x \leq x_{\text{hi}}$. So, x_{lo} and x_{hi} are lower and upper bounds on the true value of x. To find $z = x/y$, say, we compute

$$[z_{\text{lo}}, z_{\text{hi}}] = [x_{\text{lo}} /_{\nabla\text{fp}} y_{\text{hi}}, x_{\text{hi}} /_{\Delta\text{fp}} y_{\text{lo}}] \qquad \text{assuming } x_{\text{lo}}, x_{\text{hi}}, y_{\text{lo}}, y_{\text{hi}} > 0$$

with downward-directed rounding used in the first division ($/_{\nabla\text{fp}}$), and upward-directed rounding in the second one ($/_{\Delta\text{fp}}$), to ensure that the interval $[z_{\text{lo}}, z_{\text{hi}}]$ truly bounds the value of z.

Interval arithmetic [Alef83], [Moor09] is one the earliest methods for the automatic tracking of computational errors. It is quite intuitive, efficient, and theoretically appealing. Unfortunately, however, the intervals obtained in the course of long computations tend to widen until, after many steps, they become so wide as to be virtually worthless. Note that the span, $z_{hi} - z_{lo}$, of an interval is an indicator of the precision in the final result. So, an interval such as $[.8365 \times 10^{-3}, .2093 \times 10^{-2}]$ tells us little about the correct result.

It is sometimes possible to reformulate a computation to make the resulting output intervals narrower. Multiple computations also may help. If, using two different computation schemes (e.g., different formulas, as in Examples 19.3–19.5 at the end of Section 19.2) we find the intervals containing the result to be $[u_{lo}, u_{hi}]$ and $[v_{lo}, v_{hi}]$, we can use the potentially narrower interval

$$[w_{lo}, w_{hi}] = [max(u_{lo}, v_{lo}), min(u_{hi}, v_{hi})]$$

for continuing the computation or for output. We revisit interval arithmetic in Section 20.5 in connection with certifiable arithmetic computations.

19.6 BACKWARD ERROR ANALYSIS

In the absence of a general formula to bound the relative error $\eta = (y_{fp} - y)/y$ of the computation $y_{fp} = (a_{fp} \times_{fp} x_{fp}) +_{fp} b_{fp}$, alternative methods of error analysis may be sought. Backward error analysis replaces the original question:

> How much does the result y_{fp} deviate from the correct result y?

with another question:

> What changes in the inputs would produce the same deviation in the result?

In other words, if the exact identity $y_{fp} = a_{alt}x_{alt} + b_{alt}$ holds for alternate input parameter values a_{alt}, b_{alt}, and x_{alt}, we want to find out how far a_{alt}, b_{alt}, and x_{alt} can be from a_{fp}, b_{fp}, and x_{fp}. Thus, computation errors are, in effect, converted to or compared with additional input errors.

We can easily accomplish this goal for our example computation $y = (a \times x) + b$:

$$\begin{aligned}
y_{fp} &= (a_{fp} \times_{fp} x_{fp}) +_{fp} b_{fp} \\
&= (1+\mu)\big[(a_{fp} \times_{fp} x_{fp}) + b_{fp}\big] \qquad \text{with } |\mu| < r^{-p+1} = r \times ulp \\
&= (1+\mu)\big[(1+\nu)a_{fp}x_{fp} + b_{fp}\big] \qquad \text{with } |\nu| < r^{-p+1} = r \times ulp \\
&= (1+\mu)a_{fp}\,(1+\nu)x_{fp} + (1+\mu)b_{fp} \\
&= (1+\mu)(1+\sigma)a\,(1+\nu)(1+\delta)x + (1+\mu)(1+\gamma)b \\
&\approx (1+\sigma+\mu)a\,(1+\delta+\nu)x + (1+\gamma+\mu)b
\end{aligned}$$

So the approximate solution of the original problem is viewed as the exact solution of a problem close to the original one (i.e., with each input having an additional relative error of μ or ν). According to the preceding analysis, we can assure the user that the effect of arithmetic errors on the result y_{fp} is no more severe than that of $r \times ulp$ additional error

in each of the inputs a, b, and x. If the inputs are not precise to this level anyway, then arithmetic errors should not be a concern.

More generally, we do the computation $y_{\text{fp}} = f_{\text{fp}}(x_{\text{fp}}^{(1)}, x_{\text{fp}}^{(2)}, \ldots, x_{\text{fp}}^{(n)})$, where the subscripts "fp" indicate approximate operands and computation. Instead of trying to characterize the difference between y (the exact result) and y_{fp} (the result obtained), we try to characterize the difference between $x_{\text{fp}}^{(i)}$ and $x_{\text{alt}}^{(i)}$ such that the identity $y_{\text{fp}} = f(x_{\text{alt}}^{(1)}, x_{\text{alt}}^{(2)}, \ldots, x_{\text{alt}}^{(n)})$ holds exactly, with f being the exact computation. When it is applicable, this method is very powerful and useful.

PROBLEMS

19.1 Representation errors

In Section 19.1, MRRE was related to *ulp* using the assumption $1/r \leq s < 1$. Repeat the analysis, this time assuming $1 \leq s < r$ (as in IEEE 754-2008). Explain your results.

19.2 Variations in rounding

a. Show that in FLP(r, p, A) with even r, choosing round-to-nearest-even for $r/2$ odd, and round-to-nearest-odd for $r/2$ even, can reduce the errors. *Hint:* Successively round the decimal fraction 4.4445, each time removing one digit [Knut81].

b. What about FLP(r, p, A) with an odd radix r?

19.3 Addition errors with guard digits

a. Is the error derived in Example 19.1 consistent with Theorem 19.1?

b. Redo the computation of Example 19.2 with two guard digits.

c. Is it beneficial to have more than one guard digit as far as the worst-case error in floating-point addition is concerned?

19.4 Errors with guard digits

a. Show that in FLP(r, p, chop) with no guard digit, the relative error in addition or subtraction of exactly represented numbers can be as large as $r - 1$.

b. Show that if $x - y$ is computed with one guard digit and $y/2 \leq x \leq 2y$, the result is exact.

c. Modify Example 19.2 such that the relative arithmetic error is as close as possible to the bound given in the corollary to Theorem 19.1.

19.5 Optimal exponent base in a floating-point system

Consider two floating-point systems FLP($r = 2^a$, p, A) and FLP($r = 2^b$, q, A) with comparable ranges, and the same total number w of bits.

a. Derive a relationship between a, b, p, and q. *Hint:* Assume that x and y bits are used for the exponent parts and use the identity $x + ap = y + bq = w - 1$.

b. Using the relationship of part a, show that FLP($r = 2$, p, A) provides the lowest worst-case relative representation error among all floating-point systems with comparable ranges and power-of-2 radices.

19.6 Laws of algebra

In Section 19.2, examples were given to show that the associative law of addition may be violated in floating-point arithmetic. Provide examples that show the violation of the other laws of algebra listed in Section 19.2.

19.7 Laws of algebra for inequalities

a. Show that with floating-point arithmetic, if $a < b$, then $a +_{\text{fp}} c \leq b +_{\text{fp}} c$ holds for all c; that is, adding the same value to both sides of a strict inequality cannot affect its direction but may change the strict "$<$" relationship to "$\leq$."

b. Show that if $a < b$ and $c < d$, then $a +_{\text{fp}} c \leq b +_{\text{fp}} d$.

c. Show that if $c > 0$ and $a < b$, then $a \times_{\text{fp}} c \leq b \times_{\text{fp}} c$.

19.8 Equivalent computations

Evaluating expressions of the form $(1 + g)^n$, where $g << 1$, is quite common in financial calculations. For example, g might be the daily interest rate ($0.06/365 \approx 0.000\,164\,383\,6$ with a 6% annual rate) for a savings account that compounds interest daily. In calculating $1 +_{\text{fp}} g$, many bits of g are lost as a result of the alignment right shift. This error is then amplified when the result is raised to a large power n. The preceding expression can be rewritten as $e^{n \ln(1+g)}$. Even if an accurate natural logarithm function LN is available such that LN(x) is within *ulp*/2 of $\ln x$, our problem is still not quite solved since LN($1 +_{\text{fp}} g$) may not be close to $\ln(1 + g)$. Show that, for $g << 1$, computing $\ln(1 + g)$ as g when $1 +_{\text{fp}} g = g$ and as $[g \times_{\text{fp}} \text{LN}(1 +_{\text{fp}} g)]/_{\text{fp}} [(1 +_{\text{fp}} g) -_{\text{fp}} 1]$ when $1 +_{\text{fp}} g \neq 1$ provides good relative error.

19.9 Equivalent computations

Assume that x and y are numbers in FLP(r, p, chop(g)), $g \geq 1$.

a. Show that the midpoint of the interval $[x, y]$, obtained from $(x +_{\text{fp}} y)/_{\text{fp}} 2$ may not be within the interval but that $x +_{\text{fp}} ((y -_{\text{fp}} x)/_{\text{fp}} 2)$ always is.

b. Show that the relative error in the floating-point calculation $(x \times_{\text{fp}} x) -_{\text{fp}} (y \times_{\text{fp}} y)$ can be quite large but that $(x -_{\text{fp}} y) \times_{\text{fp}} (x +_{\text{fp}} y)$ yields good relative error.

c. Assume that the library program SQRT has good relative error. Show that calculating $1 -_{\text{fp}} \text{SQRT}(1 -_{\text{fp}} x)$ may lead to bad worst-case relative error but that $x/_{\text{fp}}[1 + \text{SQRT}(1 -_{\text{fp}} x)]$ is safe.

19.10 Errors in radix conversion

a. Show that when an IEEE 754-2008 binary single-precision number is converted to the closest eight-digit decimal number, the original binary number may not be uniquely recoverable from the resulting decimal version.

b. Would nine decimal digits be adequate to remedy the problem stated in part a? Fully justify your answer.

19.11 Kahan's summation algorithm

a. Apply Kahan's summation algorithm, presented at the end of Section 19.3, to the example computations in Section 19.2 showing that the associative law of addition does not hold in floating-point arithmetic. Explain the results obtained.

b. Provide an intuitive justification for the use of the correction term c in Kahan's summation algorithm.

19.12 Distribution of significand values

a. Verify that Fig. 19.1 does in fact represent a probability density function.

b. Find the average value of a normalized binary significand x based on Fig. 19.1 and comment on the result.

19.13 Error distribution and expected errors

a. Verify that $\text{pdf}_{\text{chop}}(z)$ and $\text{pdf}_{\text{round}}(z)$, introduced near the end of Section 19.4, do in fact represent probability density functions.

b. Verify that the probability density functions of part a lead to the ARRE values derived near the end of Section 19.4.

c. Provide an intuitive explanation for the expected error in rounding being somewhat more than half that of truncation.

19.14 Noisy-mode computation

Perform the computation $(a +_{\text{fp}} b) +_{\text{fp}} c$, where $a = .123\ 41 \times 10^5, b = -.123\ 40 \times 10^5$, and $c = .143\ 21 \times 10^1$ four times in noisy mode, using pseudo-random digits during normalization left shifts. Compare and discuss the results.

19.15 Interval arithmetic

You are given the decimal floating-point numbers $x = .100 \times 10^0$ and $y = .555 \times 10^{-1}$.

a. Use interval arithmetic to compute the mean of x and y via the arithmetic expression $(x +_{\text{fp}} y)/_{\text{fp}} 2$.

b. Repeat part a, this time using the arithmetic expression $x +_{\text{fp}} [(y -_{\text{fp}} x)/_{\text{fp}} 2]$.

c. Combine the results of parts a and b into a more precise resulting interval. Discuss the result.

d. Repeat parts a, b, and c with the equivalent computations $(x \times_{\text{fp}} x) -_{\text{fp}} (y \times_{\text{fp}} y)$ and $(x -_{\text{fp}} y) \times_{\text{fp}} (x +_{\text{fp}} y)$.

e. Repeat parts a, b, and c with the equivalent computations $1 -_{\text{fp}} \text{SQRT}(1 -_{\text{fp}} x)$ and $x/_{\text{fp}} [1 + \text{SQRT}(1 -_{\text{fp}} x)]$, assuming that the library program SQRT provides precisely rounded results.

19.16 Backward error analysis

An $(n - 1)$th-degree polynomial in x, with the coefficient of the ith-degree term denoted as $c^{(i)}$, is evaluated with at least one guard digit by using Horner's rule (i.e., n computation steps, each involving a floating-point multiplication by

x followed by a floating-point addition). Using backward error analysis, show that this procedure, has allowed us to compute a polynomial with coefficients $(1 + \eta^{(i)})c^{(i)}$, and find a bound for $\eta^{(i)}$. Then, show that if $c^{(i)} \geq 0$ for all i and $x > 0$, a useful bound can be placed on the relative error of the final result.

19.17 Computational errors

a. Armed with what you have learned from this chapter, reexamine the sources of computation errors in Problem 1.1 of Chapter 1. Describe your findings using the terminology introduced in this chapter.
b. Repeat part a for Problem 1.2.
c. Repeat part a for Problem 1.3.
d. Repeat part a for Problem 1.28.

19.18 Errors in incomplete multipliers

Unsigned i-bit and j-bit bit-normalized binary fractions f and g ($j \leq i$ and $0.5 \leq f, g < 1$) are multiplied by means of an $i \times j$ multiplier that produces a k-bit result ($k < i + j$).

a. What can you say about the maximum absolute and relative errors in the resulting k-bit product? Assume that all $i + j$ bits of the product are produced and the result is then truncated to k bits.
b. What should the value of k be if the relative error due to incomplete multiplication is to be no larger than that of either input operand?
c. How would you answer the question of part b if the multiplier did not produce all $i + j$ product bits but rather ignored all bits beyond position $-k$ in the partial product bit-matrix at the outset?

19.19 Associative law of addition

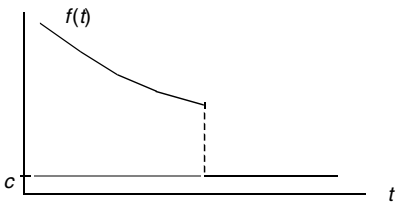
Show that in FLP(r, p, rtna), we have $x +_{\text{fp}} (y +_{\text{fp}} z) = (1 + \rho)[(x +_{\text{fp}} y) +_{\text{fp}} z]$, where $|\rho| \leq r^{-(p-1)}$, provided that the three operands x, y, and z have like signs (i.e., the associative law of addition holds approximately in this case). Note that a looser bound $|\rho| \leq 2r^{-(p-1)}$ is relatively easy to prove by relating each of the two parenthesized expressions above to $x + y + z$, but this is not what is required here.

19.20 Backward error analysis

Suppose we compute x^4 by squaring x and again squaring the result. Each squaring operation is done via a floating-point multiplication. Show that the computed result is $z = [(1 + \rho)x]^4$ and establish a bound on the magnitude of ρ.

19.21 Errors in floating-point arithmetic

Within a program, the function $f(t) = c + e^{at} \times e^{-bt}$ is evaluated by two calls to a built-in exponential function, a floating-point multiplication, and a floating-point addition. Even though the function $f(t)$ is continuous, the program results yield the following plot for $f(t)$. Explain the observed discontinuity and suggest how the "bug" might be fixed.



19.22 Algorithms for statistical calculations

Given a sample $x^{(0)}, x^{(1)}, \ldots, x^{(n-1)}$ of size n, its mean and standard deviation are defined as $\mu = \Sigma x^{(i)}/n$ and $\sigma = [\Sigma (x^{(i)} - \mu)^2/(n-1)]^{1/2}$. Study the derivation of σ from the viewpoint of computation errors when using floating-point arithmetic [Chan79].

19.23 Microsoft Excel 2007 flaw

According to news stories published in the last week of September 2007, the then newest version of Microsoft Excel spreadsheet program contained a flaw that led to incorrect values in rare cases. For example, when multiplying 77.1 by 850, 10.2 by 6425, or 20.4 by 3212.5, the number 100 000 was displayed instead of the correct result 65 535. Similar errors were observed for calculations that produced results close to 65 536. Study this problem using Internet sources and discuss, in one single-spaced typed page, the nature of the flaw, why it went undetected, exactly what caused the errors, and how Microsoft dealt with the problem.

19.24 Computing the value of *e*

The value of e is the limit of $E = (1 + 1/n)^n$, when n goes to infinity. If we compute E with larger and larger values of n, we obtain the value of e with better and better accuracy. However, if we use too large a value of n in floating-point arithmetic, $1 + 1/n$ evaluates to 1, leading to very poor accuracy. Derive the optimal value of n so that E provides the best approximation to e.

19.25 Avoiding catastrophic cancellation

Each of the following functions, when evaluated near the point indicated based on expression supplied, leads to catastrophic cancellation. In each case, derive an equivalent expression for the function that does not lead to a large error.

a. $f(x) = e^x - \sin x - \cos x$, near $x = 0$
b. $g(x) = \ln x - \ln(1/x)$, near $x = 1$
c. $h(x) = \ln x - 1$, near $x = e$

19.26 More precision doesn't always help

The following example, courtesy of W. Kahan, is offered to dispel the myth that if you repeat a computation with more and more precision and keep getting the same result, then your result must be correct. Let $f(z) = (e^z - 1)/z$, with $f(0) = 1, g(x) = |x - (x^2 + 1)^{1/2}| - 1/(x + (x^2 + 1)^{1/2})$, and $h(x) = f(g(x)^2)$.

Compute $h(x)$ for $x = 15.0, 16.0, 17.0, \ldots, 9999.0$. Repeat the computation with more precision and discuss the results.

REFERENCES AND FURTHER READINGS

[Alef83] Alefeld, G., and J. Herzberger, *An Introduction to Interval Computations*, Academic Press, 1983.

[Ashe59] Ashenhurst, R. L., and N. Metropolis, "Unnormalized Floating-Point Arithmetic," *J. ACM*, Vol. 6, pp. 415–428, 1959.

[Chan79] Chan, T. F., and J. G. Lewis, "Computing Standard Deviations: Accuracy," *Communications of the ACM*, Vol. 22, No. 9, pp. 526–531, 1979.

[Cody73] Cody, W. J., "Static and Dynamic Numerical Characteristics of Floating-Point Arithmetic," *IEEE Trans. Computers*, Vol. 22, No. 6, pp. 598–601, 1973.

[Gold91] Goldberg, D., "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, Vol. 23, No. 1, pp. 5–48, 1991.

[High02] Higham, N. J., *Accuracy and Stability of Numerical Algorithms*, 2nd ed., SIAM, 2002.

[Kaha00] Kahan, W., "Miscalculating Area and Angles of a Needle-like Triangle," available at http://www.cs.berkeley.edu/~wkahan/Triangle.pdf, 22 pp., March 24, 2000.

[Kaha04] Kahan, W., "Applications of IEEE 754's Rounding Modes," August 4, 2004. http://754r.ucbtest.org/roundingmode.txt

[Kaha04a] Kahan, W., "How Futile Are Mindless Assessments of Roundoff in Floating-Point Computation," November 1, 2004. http://www.cs.berkeley.edu/~wkahan/Mindless.pdf

[Knut81] Knuth, D. E., *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 2nd ed., Addison-Wesley, 1981.

[Kuck77] Kuck, D. J., D. S. Parker, and A. H. Sameh, "Analysis of Rounding Methods in Floating-Point Arithmetic," *IEEE Trans. Computers*, Vol. 26, No. 7, pp. 643–650, 1977.

[Kuli02] Kulisch, U. W., *Advanced Arithmetic for the Digital Computer*, Springer, 2002.

[McKe67] McKeenan, W. M., "Representation Error for Real Numbers in Binary Computer Arithmetic," *IEEE Trans. Computers*, Vol. 16, pp. 682–683, 1967.

[Metr63] Metropolis, N., and R. L. Ashenhurst, "Basic Operations in an Unnormalized Arithmetic System," *IEEE Trans. Electronic Computers*, Vol. 12, pp. 896–904, 1963.

[Moor09] Moore, R. E., R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*, SIAM, 2009.

[Ogit05] Ogita, T., S. M. Rump, and S. Oishi, "Accurate Sums and Dot Product," *SIAM J. Scientific Computing*, Vol. 26, No. 6, pp. 1955–1988, 2005.

[Over01] Overton, M. L., *Numerical Computing with IEEE Floating Point Arithmetic*, SIAM, 2001.

[Rump05] Rump, S. M., T. Ogita, and S. Oishi, "Accurate Floating-Point Summation," Technical Report 05.12, Information and Communication Science, Hamburg University of Technology, 2005.

[Ster74] Sterbenz, P. H., *Floating-Point Computation*, Prentice-Hall, 1974.

[Tsao74] Tsao, N., "On the Distribution of Significant Digits and Roundoff Errors," *Commun. ACM*, Vol. 17, No. 5, pp. 269–271, 1974.