10. COMPUTER ROUNDING ERRORS: APPLICATIONS

In chapter 9 we saw how limited memory lead inevitability to rounding errors whenever real numbers are stored. That analysis also resulted in our establishing a bound for the absolute and relative errors incurred as a result of that storage. In this chapter we look at how those simple relationships used to store a single number can be used to estimate the error associated with complex computations involving multiple operations. We begin by looking at how 'computer arithmetic' differs from what we might expect.

10.1. COMPUTER ARITHMETIC [Jac62], [San91], [Kor93]

Computer round off error coupled with the use of normalized numbers can lead to some interesting variances with the rules of ordinary arithmetic. We distinguish between integer and floating-point computations.

10.1.1 Integer Arithmetic

As we have seen, integers, provided they do not exceed the computer's word size, are stored without error. Furthermore, integer arithmetic is exact *if the result is also an integer*. Fractional parts are removed by chopping so that the usual arithmetic rule: (a+b)/c = a/c + b/c does not hold. Integer arithmetic has the advantage in that it requires far fewer operations to perform and is therefore much faster than floating-point arithmetic. It is obviously of limited value for general computation.

10.1.2. Floating Point Arithmetic [Ove01], [Gol91]

We will assume that floating-point arithmetic is done exactly, and that the only source of error is in the storage of the result. This assumption has already been justified for coprocessor arithmetic on the basis of the relative precision of the computed (80 bits) and stored result (64 bits). Without a coprocessor however, arithmetic is software dependent so that the assumption of exact arithmetic is no longer valid. Example 10.1 will demonstrate this difference.

We now briefly examine how round off error can influence the accuracy of certain arithmetic operations.

10.1.3. Addition/Subtraction

Addition and subtraction operation are especially susceptible to a potential loss of significant digits due to the requirement that exponents must match before aligning radix points.

To see this, suppose we wish to add the two real numbers, x = 8521.97 and y = 1.09977, using a base 10 computer with a 4-bit normalized mantissa, i.e., p=4 and chopped arithmetic. The procedure for addition (and subtraction) implements the following steps:

1. Normalize both numbers:
   $x = 0.852197 \times 10^4$; $y = 0.109977 \times 10^1$
2. Chop the mantissas to 4 digits:
   $f_x = 0.852197 \longrightarrow 0.8521$; $f_y = 0.109977 \longrightarrow 0.1099$
3. Adjust the exponent of the smaller number (y) to match the exponent of the larger number (x) with the rule: $f_{y'} = f_y B^{(e_y - e_x)}$, where y' is the adjusted mantissa. For B = 10, we have, $e_{y'} = e_x = 4$, $f_{y'} = 0.1099 \times 10^{(1-4)} = 0.0001099$.
4. Add: $x + y = 0.8521 \times 10^4 + 0.0001099 \times 10^4 = 0.8522099 \times 10^4$
5. Chop to 4 digits: $x + y = 0.8522 \times 10^4$
6. Re-normalize if necessary.

We observe that if $e_x - e_y \geq p$ then y will be too small to contribute to the sum in which case $x + y \rightarrow x$. In this example, if e_y had been 0 instead of 1, y would have made no contribution at all to the sum. As it is, we see from a comparison of the result with the exact sum, 8523.06977, that normalization coupled with the necessity for matched exponents has led to an absolute error of 1.06977 corresponding to a loss of most of the contribution from y.

This example illustrates how the larger number dictates the radix shift in a smaller number prior to addition or subtraction with the consequence that the smaller number may be shifted to oblivion.

10.1.4. Multiplication/Division

In multiplication and division, exponents are added or subtracted and mantissas are multiplied or divided. In neither case is an exponent match required before aligning the radix so these operations are inherently less susceptible to significant digit loss.

10.1.5. Failure of Arithmetic Rules

Given how limited precision can affect the accuracy with which floating-point numbers are stored, it will come as no surprise that the usual rules of algebra, like the associative and distributive laws, which hold for real numbers, can sometimes fail for computer numbers.

10.1.6. Error Bounds on Extended Floating-Point Arithmetic

We have just seen how limited precision can lead to limited accuracy in the storage of a single real number. *Assuming rounded arithmetic*, we now extend our analysis to include compound operations.

From Table 9.7, we know that the relative round off error is bounded by u, that is:

$$\frac{|x - FL(x)|}{|x|} \le u \tag{10.1}$$

We now define the quantity δ, with the relationship:

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

where δ may be positive, negative or zero*, and is obviously the *actual* relative error, sans absolute value signs, of representing x by FL(x). In general, we do not know the value of δ, but we can establish its bound by taking absolute values of both sides of the previous expression, and then

* For chopped arithmetic, δ £ 0.

comparing the result to Eq. (10.1), to establish that $|\delta| \leq u$, that is δ is bounded by u. From the definition of δ, we have:

$$FL(x) = x(1+\delta) \tag{10.2}$$

which says that the computer number representation of x can be viewed as the value of x plus a perturbation term, $x\delta$, where $|\delta|$ is bounded by u. If one rearranges Eq (10.2) to read, RE=$|\delta|$, it is easy to appreciate just how conservative the use of the bound $|\delta| \leq u = 2^{-p}$ is, since, as we have said, δ can be either sign or even zero.

Equation (10.2) is limited to computing FL(x) for the storage of a *single* real number x. However, if we make a couple of reasonable assumptions, we can extend its use to include the situation in which x is the *result* of one of the basic arithmetic operations, +, -, * and ÷ between any two *computer numbers* y and z. This in turn allows us to analyze complex computations as a sequential series of these individual operations. These assumptions include:

- Computer arithmetic is exact. This assumption has already been justified since the arithmetic registers are 80 bits wide, so the only error, apart from underflow or overflow, is storing the computed result using 64 bits.

- The numbers y and z are computer numbers and thus belong to $F(B, p, E_{max}, E_{min})$

Actually, apart from the round off error incurred with the initial input, all numbers called from memory *prior* to any arithmetic operation must belong to $F(B, p, E_{max}, E_{min})$. The error associated with the initial input should become increasingly less important the more complex the computation.

In the context of these assumptions, if we view x in Eq (10.2) as the numerical result of one of the four basic arithmetic operations, +, -, $*, \div$, which we represent by the symbol '$\odot$', between two computer numbers, y and z, we can write,

$$FL(y\odot z) = (y\odot z)(1+\delta)\,, \qquad (10.3)$$

where y and z are assumed to belong to F(B, p, E_{max}, E_{min}), but the result of the operation, $y\odot z$, may not. Note however, that while $y\odot z$ may not be belong to F(B, p, E_{max}, E_{min}), FL($y\odot z$) will.

The four basic arithmetic operations then become:

Table 10.1. Basic Arithmetic Operations

$FL(y*z) = (y*z)(1+\delta)$
$FL(y/z) = (y/z)(1+\delta)$
$FL(y+z) = (y+z)(1+\delta)$
$FL(y-z) = (y-z)(1+\delta)$

where each of these expressions refer to a *single* arithmetic operation between two computer numbers.

COMMENT

The relationships in Table 10.1 apply to the computer representation of $y\odot z$ assuming that y and z are computer numbers, but the result of the '$\odot$' operation may not be. But if y and z are not computer numbers to begin with, then the result of the operation $y\odot z$ is not Eq (10.3), but is, instead:

$FL(y\odot z) = FL[FL(y) \odot FL(z)]$, for the initial storage of y and z

$= FL[y(1+\delta_1) \odot z(1+\delta_2)]$, using Eq (10.2)

$= [y(1+\delta_1) \odot z(1+\delta_2)](1+\delta_3)$ using Eq (10.3);

and where again, the δ's are bounded by u.

The absolute error bound in the former case where y and z are computer numbers and where Eq (10.3) applies, is: AE ≤ $|y \odot z|$ u, whereas in the case where y and z are not computer numbers, the bound depends on the specific operation ⊙.

For example, if ⊙ represents multiplication, we have, ignoring all non-linear terms in u:

AE ≤ |y∗z| u , if: y,z Î F(b, p, E_{max}, E_{min})
AE ≤ |y∗z|3u , if: y,z Ï F(b, p, E_{max}, E_{min})

which can be a significant difference.

10.2. PROPAGATION OF FLOATING-POINT ERROR [Mor83], [K&C02], [For70], [M&H78], [FMM77]

We now apply these relations to a more complex calculation consisting of a sequence of operations. It is important to stress that the order in which "FL" is applied must be the same as the order in which the computer does its arithmetic.

We will assume, as is typically the case that computation proceeds left to right, quantities in brackets first, multiplication and division before addition and subtraction etc. Exponentiation is not included since it is not an elementary arithmetic operation.

10.2.1. Addition/Subtraction

Consider the simple sum $x_1+x_2+x_3$ where it is assumed that x_1, x_2 and x_3 are machine numbers, i.e. they have been stored and recalled from memory. The order of operation will be from left to right, that is: $(x_1+x_2)+x_3$. Applying the addition rule from Table 10.1, we have:

$$
\begin{aligned}
FL(x_1+x_2+x_3) &= FL((x_1+x_2)+x_3) \\
&= FL(FL(x_1+x_2)+x_3) && \text{inner ( ) first} \\
&= FL\{(x_1+x_2)(1+\delta_1)+x_3\} && \text{using Table 10.1 once} \\
&= \{(x_1+x_2)(1+\delta_1)+x_3\}(1+\delta_2) && \text{and again, to give,}
\end{aligned}
$$

$$= (x_1 + x_2 + x_3) + (x_1 + x_2)(\delta_1 + \delta_2 + \delta_1\delta_2) + x_3\delta_2 \ .$$

The absolute error is then:

$$\mathrm{AE} = \left|(x_1 + x_2 + x_3) - \mathrm{FL}(x_1 + x_2 + x_3)\right| = \left|(x_1 + x_2)\left[\delta_1 + \delta_2 + \delta_1\delta_2\right] + x_3\,\delta_2\right|$$

$$\le \left|x_1 + x_2\right|\left|\delta_1 + \delta_2 + \delta_1\delta_2\right| + \left|x_3\,\delta_2\right| \qquad \text{using the triangle inequality}^{*}$$

$$\le \left|x_1 + x_2\right|\left[\left|\delta_1\right| + \left|\delta_2\right| + \left|\delta_1\right|\left|\delta_2\right|\right] + \left|x_3\right|\left|\delta_2\right| \qquad \text{and again, and then}$$

$$\le \left|x_1 + x_2\right|\left(2u + u^2\right) + \left|x_3\right|u \quad \text{replacing } \delta \text{ by its bound, i.e., } |\delta| \le u$$

We now drop u^2, since $u^2 << u$, apply the triangle inequality again to the $|x_1+x_2|$ term, add and subtract the quantity $|x_3|u$, and rearrange the result to read:

$$\mathrm{AE} \le \left(\left|x_1\right| + \left|x_2\right| + \left|x_3\right|\right) 2u - \left|x_3\right|u$$

$$\le \left(\left|x_1\right| + \left|x_2\right| + \left|x_3\right|\right) 2u$$

and the relative error:

$$\mathrm{RE} \le \frac{\left(\left|x_1\right| + \left|x_2\right| + \left|x_3\right|\right)2u}{\left|x_1 + x_2 + x_3\right|} \qquad (10.4)$$

Clearly a lot of work for such a simple problem. We now apply Eq (10.4) to addition and subtraction. For the case in which x_1,x_2 and x_3 are all positive, the ratio of absolute values cancel (using the triangle inequality again) in Eq (10.4), with the conclusion that the relative error is bounded by 2u.

Of greater interest however, is what happens if one of the three numbers, say x_3, happens to be negative, and also happens to be of approximately the same magnitude as the sum of the other two, i.e., $|x_3| \approx x_1+x_2$. In this case, the ratio of the numerator to the denominator in Eq(10.4), and therefore the relative error, can be very large, in fact, virtually unbounded. This effect is often called *catastrophic* or *subtractive* cancellation, and should be avoided at all costs, because even a single

[*] The triangle inequality reads, $|x+y| \le |x|+|y|$

instance can severely compromise the accuracy of an otherwise well designed calculation. We will return to this topic when we discuss error reduction strategies.

10.2.2. Multiplication/Division

Using an analysis similar to our previous one, the reader may wish to show that the relative error bound for both of the compound operations, $x_1x_2x_3$ and x_1x_2 / x_3 is given by RE $\leq$ 2u.

Before leaving this topic, we will analyze one additional compound procedure that illustrates just how unwieldy this method of analysis can be.

EXAMPLE 10.1. Round off in Extended Addition and the Value of the Coprocessor [McC02]

While ε is the smallest number that can be added to 1, which results in a sum differing from 1, it is not the smallest number that can be added to 0, to produce a result different from 0. That number is $X_{\min}$ (Eq 9.12), and represents the first positive (or negative) computer number greater than 0. The interval between 0 and $X_{\min}$ is just the underflow gap.

To illustrate this point, we add the number 1 x 10^{-16}, which is less than ε ($= 2^{-53}$) to 0, one million times, and which, assuming no round off error, ought to total to 1 x 10^{-10} exactly.

We do this computation in ANSI/IEEE double precision with, and without, a math coprocessor. The result is not exactly 1 x 10^{-10}, but is instead:

$$1.00000000026118000 \times 10^{-10} \text{ (no coprocessor)}$$
$$1.00000000002310420 \times 10^{-10} \text{ (coprocessor)}$$

where all of the non-zero digits after the 1 correspond to the total accumulated round off error.

Note that the coprocessor, which implements the IEEE/ANSI Standard, reduces the accumulated error by more than an order of magnitude.

To estimate the theoretical error bound, we begin by writing the summation as the sequential process:

$$\sum_{i=1}^{n} x_i = \left(\cdots\left(\left(x_1 + x_2\right) + x_3\right) + \cdots + x_n\right) .$$

Our goal is to compute the machine representation of this sum, namely:

$$\begin{aligned} \mathrm{FL}\sum_{i=1}^{n} x_i &= \mathrm{FL}\left(\cdots \mathrm{FL}\left(\mathrm{FL}\left(x_1 + x_2\right) + x_3\right) + \cdots + x_n\right) \\ &= \left[x_1\left(1+\delta_1\right)\left(1+\delta_2\right) + \cdots + \left(1+\delta_n\right)\right] + \left[x_2\left(1+\delta_2\right)\left(1+\delta_3\right) + \cdots + \left(1+\delta_n\right)\right] + \cdots + x_n\left(1+\delta_n\right) \\ &= x_1 \prod_{j=1}^{n}(1+\delta_j) + x_2 \prod_{j=2}^{n}(1+\delta_j) + \cdots + x_n(1+\delta_n) \qquad (10.5) \end{aligned}$$

where $\delta\,\square \equiv 0$ has been introduced for notational convenience.

Because the δ_j are all bounded by the unit round off error: $|\delta_j| \le u$, it follows that (the validity of this expression follows by inspection):

$$\prod_{j=1}^{k}\left(1+\delta_j\right) \le \left(1+u\right)^k .$$

Term by term substitution into Eq (10.5) gives:

$$\begin{aligned} \mathrm{FL}\sum_{i=1}^{n} x_i &\le x_1\left(1+u\right)^n + x_2\left(1+u\right)^{n-1} + \cdots + x_n\left(1+u\right) \\ &= \sum_{i=1}^{n} x_i\left(1+u\right)^{n-i+1} \approx \sum_{i=1}^{n} x_i\left[1+\left(n-i+1\right)u\right] \end{aligned}$$

where the last line follows from the binomial series[*].

This last expression can then be expanded to read:

$$\mathrm{FL}\sum_{i=1}^{n} x_i \le \sum_{i=1}^{n} x_i + \left[n\sum_{i=1}^{n} x_i - \sum_{i=1}^{n} i x_i + \sum_{i=1}^{n} x_i\right]u$$

Now all of the x_i are the same in the example we are analyzing which leads to the following simplifications,

---

[*] $(1+x)^m = 1 + mx + \frac{m(m-1)}{2!}x^2 + \cdots + \frac{m(m-1)\cdots(m-n+1)}{n!}x^n + \cdots \approx 1 + mx \qquad for\; 0 < x << 1$

Strictly speaking, we should take into account the fact that, while powers of u are very small relative to u itself, the coefficients in the binomial expansion will become very large. To account for the contribution of these higher order terms, one can show that $(1+x)^m$ is bounded by 1+kmx, i.e., $(1+ x)^m \le 1+ kmx$, where k is a number slightly greater than 1, typically 1.01-1.06. For our purposes here, this factor can be ignored.

$$\sum_{i=1}^{n} x_i = nx$$

$$\sum_{i=1}^{n} i x_i = x\sum_{i=1}^{n} i = x\left[\frac{n(n+1)}{2}\right].$$

We then have, after substitution of these expressions:

$$FL\sum_{i=1}^{n} x_i \le nx + \left[n^2 x - \frac{x}{2}\left(n^2 + n\right) + nx\right]u \cong nx + \frac{n^2 xu}{2} \qquad \text{for } n^2 >> n, \qquad (10.6)$$

so our final bound on the absolute error reads:

$$AE = \left|nx - FL\sum_{i=1}^{n} x_i\right| \le \frac{n^2 xu}{2}.$$

Taking $u = \varepsilon$, we calculate a bound of $1/2 * (10^6)^2 * 10^{-16} * 1.1 \text{ x } 10^{-16} = 0.55 \text{ x } 10^{-20}$. This may be compared with an actual error of $0.23 \text{ x } 10^{-20}$ for the coprocessor case. Note however that the bound underestimates the error if a coprocessor is not present, a fact, fortunately, that is of historical interest only. This emphasizes the point made earlier that the accuracy of floating-point arithmetic done on a PC *without* a math coprocessor is software dependent with the result that the assumption we made in constructing Table (10.1), namely that all floating-point arithmetic is done exactly, is invalid.

One final example illustrates the importance of the math coprocessor in reducing both the computation time and round off error.

The following code fragment, which makes liberal use of transcendental functions, was run in double precision, with, and without a math coprocessor.

Code Fragment (10.1) Coprocessor Test

```
Sum = 0
For n = 1 to 10000
  x = (Tan(Atn(Exp(Log(Sqr(n*n))))))/n
  Sum = Sum + x
Next n
```

Assuming no arithmetic or round off error, the value of x would be exactly 1 for each value of N so the final value of SUM should be 10000 exactly. The results are listed in Table (10.2).

The difference between the two results is partly due to the fact that the coprocessor, with its extended precision registers does its basic arithmetic operations more accurately, but even more important is the fact that the transcendental library functions built into the coprocessor results in much greater accuracy and speed.

Table (10.2) Effect of a Math Coprocessor on Speed and Accuracy

COPROC.	REL. TIME(sec.)	SUM
NO	*11.5*	*10000.0000096474000*
YES	1.0	10000.0000000000130

Incidentally, it is interesting to note that the error value in the sum (i.e., Sum -10000), is not necessarily the maximum error because the cumulative round off error does not increase uniformly with increasing N. Figure (10.1) illustrates this fact. Here we have plotted 10,000 values corresponding to the difference (N-SUM)$*10^{13}$ vs. N (x axis) for the coprocessor calculation. If there were neither computation nor round off error, this difference would be zero for all values of N.

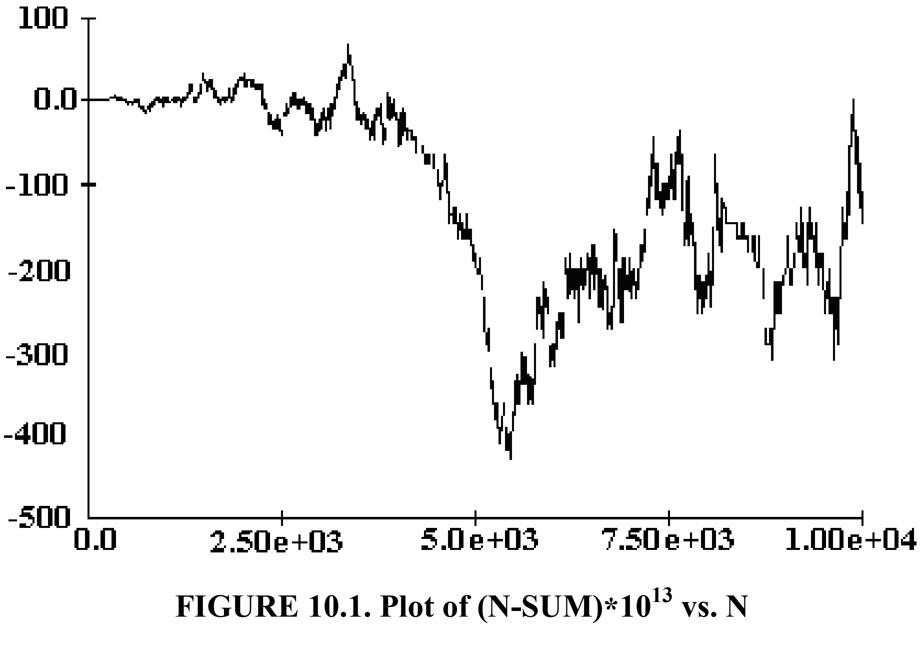

FIGURE 10.1. Plot of (N-SUM)$*10^{13}$ vs. N

This pseudo-random walk behavior reflects the fact that the actual individual errors δ_i can take on any value within the constraints of their bound, u. Establishing a theoretical error bound for the algorithm in Code Fragment (10.1) would be difficult at best.

10.3. ERROR REDUCTION STRATEGIES

As we have just seen that a detailed error analysis of a complex computation is itself complicated and time consuming - in fact, prohibitively so for most real computations. The need for such an analysis can be reduced significantly however by designing algorithms that respect the potential for round off error problems before they occur.

What follows are a few rules and examples that are worth keeping in mind when writing code. Obviously, for programs involving relatively few computations, or when computational errors are negligible compared with experimental error, these suggestions will be unimportant. Quite the opposite may be true for certain kinds of extended computations, especially if the problem is already ill conditioned. And obviously, double precision floating point math should be a given.

RULE 1: MINIMIZE THE NUMBER OF ARITHMETIC OPERATIONS

This recommendation follows from the observation that, since each arithmetic operation can potentially contribute to the total round off error, then it makes sense to try and make complex procedures as efficient as possible.

EXAMPLE (10.2) Evaluation of a Polynomial

Suppose you wished to compute the following polynomial for a specific value of n and a range of x values: $p(x) = a_0 + a_1x + \cdots + a_nx^n$.

You could simply code the polynomial as written, but if you care about efficiency, and you should, then coding as written is the least efficient method possible. A few moments with pencil and paper should convince you that the total number of multiplications and additions required are n(n+1)/2, and n respectively.

Alternatively, Horner's rule, which actually dates back to Isaac Newton, rewrites the polynomial in the following form thereby reducing the total number of operations to 2n for multiplication and addition combined.

$$p(x)=\left(\left(\cdots\left(\left(a_n x+a_{n-1}\right)x+a_{n-2}\right)x+\cdots+a_2\right)x+a_1\right)x+a_0$$

Another advantage of Horner's rule is that it is actually easier to code than the polynomial itself, as the following code fragment shows:

Code Fragment (10.2) Coding a Polynomial

```
Poly = a(n)
FOR i = n-1 TO 0 STEP -1
    Poly=Poly*x+a(i)
NEXT i
```

where the a(i) are the coefficients of the polynomial.

Historically, Horner's rule was related to the need to facilitate pencil and paper calculations, nevertheless it is the usual method for the computer evaluation of polynomials of any size.

RULE 2: AVOID ADDING AND SUBTRACTING NUMBERS OF DISPARATE MAGNITUDES

We have already seen how, when two numbers of widely differing magnitudes are added or subtracted, there can be a loss of significant figures due to the necessity for matching exponents by de-normalizing the smaller number. When faced with multiple additions and subtractions, try to arrange the computations to ensure that the numbers are of roughly the same magnitude in order to preserve significant digits.

EXAMPLE (10.3) Summation of a Constant

To illustrate the problem, suppose we were to successively sum the number 1.2, 50,000 times, which, without round off error, would result in an exact value of 60,000. Using the following loop with *single* precision, which better illustrates the problem, we have:

```
FOR k=1 TO 50000
    Sum = Sum + 1.2
NEXT k
```

we obtain instead: Sum = 59973.49.

The problem of course, is that as the sum builds, the difference between the partial sum, and the number 1.2, causes an increasing loss of significant digits.

The recommended procedure is to partition the sum into smaller groups of equal size, which can then be added together. For example, if we divide the summation into 10 groups of 5000 summations each, and then add the results, we obtain the improvement: Sum = 60001.92.

Thus when doing extended summation with numbers of approximately equal magnitude, use partial sums of about equal magnitude, and then add the partial sums.

EXAMPLE (10.4) Series Summation

Suppose you had occasion to sum the following series,

$$\sum_{n=0}^{\infty} \frac{1}{(2n+1)^2} = 1 + \frac{1}{3^2} + \frac{1}{5^2} + \cdots = \frac{\pi^2}{8} \approx 1.233700...$$

Coding the sum using a 'forward summing' loop, from n = 0 to a sufficient upper index, gave the *single* precision result, SUM = 1.233**596** .

The computed value is stable to the last digit stated so disagreement between the sum and the correct value is due to round off error, and not early truncation of the summation process.

As in the previous example, we are adding increasingly smaller numbers to a partial sum that achieved most of its magnitude early in the summation process. The solution to the problem is to 'back sum' the series starting with large values of n and working backwards to n = 0. Summing from the same *upper* index as for the “forward sum”, to n = 0, gives SUM = 1.233698, for a factor of 40 decrease in the relative error.

We may quantify this result by referring to Eq (10.5) which shows that the first term, x_1, is multiplied by all n error terms, the next term x_2, by n-1 error terms and so on until the last term is multiplied by only the single error term $(1+\delta_n)$.

Now x_1 is simply the first term, whether we forward or back sum, and since it is multiplied by the largest number of error terms then it usually makes sense to ensure that x_1 is the smallest value in the series, namely, the first back-summed term.

As a rule then, one should carry out extended summations in the direction of *increasing* magnitude rather than the other way round. It should also be pointed out that the need for 'back summing' is dictated by the nature of the particular series being summed and the acceptable error. In other words, back summation may not be necessary, but the method is straight-forward if it is.

The next example illustrates an interesting phenomenon, sometimes termed “smearing”, where the relative magnitudes of terms in a series can really get out of hand.

EXAMPLE 10.5. Alternating Series Summation

The Taylor series for exp (-x) is given by: $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots$

and is valid for all values of x.

Suppose we use this series to compute a double precision approximation to exp(-20) by forward addition of the terms until the individual contributions are less than machine epsilon. This computation gave the following results,

SERIES SUM $\approx 6.14756 \times 10^{-9}$,
CORRECT VALUE = $2.06115... \times 10^{-9}$

To see what went wrong, we list a selection of the 82 terms that contribute to the final sum. We distinguish between the even and odd values of the summation index, j, to differentiate between positive and negative terms.

Table 10.3. Terms in Expansion of exp(-20)

ODD j	TERM	EVEN j	TERM
1	-20	2	200
3	-1333.3	4	6666.6
...		...	
9	-1.4109 E+06	10	2.8218 E+06
...		...	
21	-4.1047 E+07	22	3.7315 E+07
...		...	
43	-1455.9	44	661.79
...		...	
71	-2.7763 E-10	72	7.7119 E-11
...		...	
81	-4.1707 E-16	82	1.0172 E-16

In forward summing the series, we see that x^n builds rapidly with increasing n. This results in large terms of alternating sign whose sum is still large. Even though the n! in the denominator will eventually take over and drive the ratio to zero, nevertheless these large numbers ultimately determine the number of significant digits in the final result.

We can roughly estimate the overall error in the summation by assuming the bound is determined by the largest term in the sum. This largest term corresponds to j = 20 where the contribution is 4.3 x 10^7, so roughly:

$$\begin{aligned} \text{Absolute Error} &\le |\text{max term}|\,\varepsilon \\ &= 4.3 \times 10^7 \times 1.1 \times 10^{-16} \\ &= 4.7 \times 10^{-9} \end{aligned}$$

which compares favorably with an actual absolute error of about 4 x 10^{-9}.

This so called 'smearing' effect will occur in any situation in which the individual terms in the sum are large relative to the value of the sum itself. Backward summation makes no sense and actually makes matters worse so this technique is not an answer.

The standard solution to the problem is to compute $[\exp(-1)]^{20}$. This avoids the problem since the series for exp(-1) shows little of the smearing behavior that exp(-20) did. Alternatively, one can compute 1/exp(20) because e^x is well behaved for positive values of x. These conclusions can be confirmed with a simple calculator.

RULE 3: AVOID SUBTRACTING NUMBERS OF NEARLY EQUAL MAGNITUDE [For70], [K&C02]

We have already discussed the fact that the subtraction of two nearly equal numbers, termed subtractive cancellation, is a dangerous proposition because the relative error can increase without bound. One obvious solution is to use double or extended precision, but there are situations where even that remedy cannot save a flawed algorithm. In the following examples we place the emphasis on a reformulation of the problem rather than just relying on more computing horsepower.

EXAMPLE 10.6. Solution of a Quadratic Equation

A classic example of subtractive cancellation arises in the solution of the quadratic equation: $ax^2 + bx + c = 0$, using the familiar quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} .$$

If the product 4ac is small relative to b^2, then the magnitude of the discriminate and b will be nearly the same, that is:

$$\sqrt{b^2 - 4ac} \approx b .$$

In this situation, subtractive cancellation will almost certainly lead to a significant error. For example, the coefficients a=1, b=1, and $c=10^{-6}$ gave the roots, on using the quadratic formula in single precision:

$$x_1 = -0.999999 ;\; x_2 = -1.013279 \times 10^{-6} .$$

While x_1 is correct, x_2 is too large by more than 1%.

A better way to formulate the solution is to rationalize the numerator of the quadratic formula in order to ensure that b and $\sqrt{b^2 - 4ac}$ will add rather than subtract. This is done by multiplying numerator and denominator by the quantity

$$-b - \mathrm{sign}(b)\sqrt{b^2 - 4ac}$$

The second root can then computed from the un-rationalized form of the quadratic formula. The results of this process may be summarized as follows:

$$x_1 = 2c/\gamma \ , \qquad x_2 = \gamma/2a$$

where:

$$\gamma \equiv -\left[b + \mathrm{sign}(b)\sqrt{b^2 - 4ac} \right].$$

Using these relationships for the previous example gave the same correct value for x_1, but changed the value of x_2 to -1.000001×10^{-6}, which is now correct to 7 significant digits.

EXAMPLE 10.7. Series Expansion

Other potential subtractive cancellation problems can arise if one attempts to compute quantities like x - sin x, or $e^x - 1$ for small values of x.

In the first case, subtractive cancellation is a potential hazard for small x since then $\sin(x) \cong x$. The solution to the problem follows from the observation that the leading term in the Taylor series expansion of sin(x) is x, i.e.:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

By computing the difference, x-sin x, the x cancels, leaving a power series starting with the term $x^3/3!$ This resulting series is much less susceptible to subtractive cancellation.

In the second case, $e^x \cong 1$, for small x, so the difference $1-e^x$ is bound to cause problems. Again, the solution is to expand e^x as a Taylor series in which case the 1 again cancels leaving a better-behaved series.

Such examples are not at all uncommon; indeed one has only to peruse the common trigonometric formulae to find a rich selection of potential headaches. The important thing for the programmer is to be aware of the potential for loss of significance and then look for alternative methods for formulating the problem.

10.3.1. Miscellaneous Examples

Round off error can also influence the results of routine programming statements like the following.

For example, the statement,

```
IF X = Y THEN ...
```

will almost certainly be a source of error if either or both X or Y are floating-point numbers because it is unlikely that they will ever be found exactly equal. As a result, the action called for will never be implemented. Instead, it makes more sense to write

IF ABS(X-Y)< δ THEN

where δ is some defined limit of precision, e.g., $1\text{x}10^{-6}$, machine epsilon etc.

Another related problem concerns loop indices in statements like

```
FOR J=1 TO N
```

where N is the result of one or more floating-point operations. Loop indices should be integers, but because N is a computed number, round off error will most certainly dictate otherwise. This difference can result in a loop being executed the wrong number of times.

We turn now to a different kind of problem, one where the algorithm has a built in trap and where the solution to the problem calls for modification of the algorithm itself.

10.3.2. Numerical Instability [FMM77]

To this point, we have focused on the cumulative effect of multiple round off errors in complex computations. We now look at how even a single, normally inconsequential error, can sometimes propagate through a calculation causing a final error of disastrous proportions. Algorithms with this property are called "unstable".

To illustrate how this can happen, we turn to a more detailed study of Example (9.2).

Integrals of the type: $I_n = \int_0^t x^n e^{x-1} dx$ are common to the physical sciences. Suppose we have need for a table of I_n values as a function of the integer n, for a fixed value of t. Taking t =1 for purposes of illustration, we can do a by-parts integration to establish the expression:

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

from which follows the recurrence relation:

$$I_n = 1 - nI_{n-1} \qquad \text{for } n=2,3,... \tag{10.7}$$

This seems to be a very useful expression because all we need is one value of *I*, say I_1, and we can compute I_n for all values of $n > 1$. This is, on the surface at least, a very attractive alternative to the work of performing a numerical quadrature for each value of *n*. Even I_1 is easy to obtain since a single integration yields the result $I_1 = e^{-1} = 0.367879...$.

Values of I were then computed from Eq (10.7) for successive values of n using double precision arithmetic. These results are tabulated in column 2, labeled "Unstable Algorithm", of Table (10.4). For comparison purposes, the "correct values" of the integral are listed in column 3. These were determined by Romberg integration to a consistency of eight digits.

We note that Eq (10.7) is giving poor results by the time $n \approx 15$, and by the time *n*=20, agreement is non-existent. An analysis of this discrepancy is simple if we make the seemingly naive assumption that the only important error is that of storing the original number e^{-1}. We ignore all additional error associated with the computing and storing of successive results.

We begin the analysis by computing the absolute error E_1 associated with the storage of I_1 ($= e^{-1}$) - see Table (10.4):

$$E_1 \equiv \left|I_1 - FL(I_1)\right| \le \left|I_1\right|\varepsilon = e^{-1}\varepsilon$$

If we assume no round off error beyond the storage of I_1, we can write for the recurrence relation, Eq (10.7):

$$FL(I_n) = FL(1 - nI_{n-1}) = 1 - nFL(I_{n-1}) \qquad (10.8)$$

where $FL(I_{n-1})$ is the floating-point representation of I_{n-1}. The absolute error in I_n is then,

$$\begin{aligned} E_n \equiv \left|I_n - FL(I_n)\right| &= \left|1 - nI_{n-1} - \left[1 - nFL(I_{n-1})\right]\right| \\ &= n\left|FL(I_{n-1}) - I_{n-1}\right| \\ &= nE_{n-1} \qquad n = 2,3,\cdots, \end{aligned}$$

where we have used Equations (10.7) and (10.8) in the expression for E_n. We now have a recurrence relation for the absolute error, E_n.

Starting with n = 2, and writing a few successive terms, E_2, E_3, etc., in terms of E_1, we quickly establish the following relationship between E_n and E_1:

$$E_n = n!E_1 \le n!e^{-1}\varepsilon \qquad n=2,3,\cdots$$

where we have used the bound, $|e^{-1}|\varepsilon$, for the absolute error E_1.

This tells us that the error in the storage of e^{-1} is successively propagated through each step of the calculation with a magnification factor of n! A computational nightmare! So just how valid is our assumption that the whole of the error in I_n can be explained by the error in the initial storage of exp(-1)? Taking $\varepsilon = 1.1 \times 10^{-16}$ as usual, we can calculate the predicted absolute error E_n for each value of n. These results are tabulated in column 6, labeled Pred. Error, of Table (10.4), and should be compared with the Actual Error in Col. 5. The Actual Error is just the difference between the values in columns 2 and 3. As unlikely as it may seem, the error bound for the storage of e^{-1} does indeed account for most of the actual error.

The remedy to this problem is deceptively simple. An unstable recurrence relation for increasing values of n may be perfectly stable if the relation is written for decreasing n. Equation (10.7) then becomes:

$$I_{n-1} = \frac{1 - I_n}{n} \qquad n = \cdots, 3, 2$$

Thus, for some starting value, I_n, we can compute I_{n-1}, I_{n-2}, ... The obvious questions are, 'how accurately', and if better than the unstable algorithm, why? A “starting value” is derived by noting that, because

$$e^{-1} \leq e^{x-1} \leq 1 \text{ for } 0 \leq x \leq 1:$$

$$I_n = \int_0^1 x^n e^{x-1} dx \leq \int_0^1 x^n dx = \frac{1}{n+1} .$$

This result not only gives us an initial estimate of I_n but also tells us that as $n \to \infty$, the integral $I_n \to 0$, which is a conclusion that certainly contrasts with the trend for the unstable algorithm in Col. 2, Table 10.4.

To test this proposed solution to the instability problem, suppose we take $n = 60$, so that I_{60} (initial guess) = 1/61, and then compute I_n for the same set of n values as used previously. The computed values of I_{n-1} are then listed in Col. 4 of Table 10.4 under the heading “Stable Algorithm”. It is seen that the value of the computed integral, and the correct values in Col. 3, are in complete agreement to the number of figures stated. The new algorithm is clearly very stable.

Table 10.4. Stable Vs Unstable Algorithm

n	UNSTABLE ALGORITHM	CORRECT VALUE	STABLE ALGORITHM	ACTUAL ERROR	PREDICTED ERROR
5	14.5533 E-2	14.5533 E-2	14.5533 E-2	<1E-8	5E-16
10	8.38771 E-2	8.38771 E-2	8.38771 E-2	<1E-8	2E-10
15	5.90338 E-2	5.90175 E-2	5.90175 E-2	2E-5	5E-5
20	-3.01924 E+1	4.55449 E-2	4.55449 E-2	3E1	10E1
30	-3.29676 E+15	3.12797 E-2	3.12797 E-2	3E15	10E15
40	-1.01408 E+31	2.38227 E-2	2.38227 E-2	1E31	3E31
50	-3.78009 E+47	1.92378 E-2	1.92378 E-2	4E47	10E47

Why has simply rewriting the recurrence relation changed a hopelessly unstable algorithm to an extremely stable one? In the unstable algorithm, we used a very precise *initial value* $I_1 = e^{-1}$, and then watched how roundoff error in the initial storage of I_1 lead to a total corruption of subsequent values of I. By contrast, with the stable algorithm we started with an approximate initial value for I_n, yet obtained excellent subsequent values for I.

The analysis of the stable algorithm is similar to that done previously, but with one major difference. For the unstable algorithm we looked at propagated roundoff error in I_1 whereas here we ignore round off error entirely, and instead concentrate on the effect of the initial approximation, I_n. It is easy to show that the error in the (n-1)th value E_{n-1} is related to the error in the starting value E_n by the expression:

$$E_{n-1} = \frac{1}{n} E_n \, .$$

Again we can derive a recurrence relation linking the absolute errors. Writing out a few terms establishes the error E_k in terms of E_n, namely,

$$E_k = \frac{1}{n(n-1)(n-2)\cdots(k+1)} E_n = \frac{k(k-1)(k-2)\cdots 1}{n(n-1)(n-2)\cdots(k+1)k(k-1)\cdots 1} E_n$$

or:
$$E_k = \frac{k!}{n!} E_n \, .$$

Here we see that the absolute error, E_n, in the initial estimate of I_n, is damped by the k!/n! term. Since n > k, the larger n is relative to k, the smaller the absolute error in the kth computed integral.

This has been a nice example of how a simple analysis can sometimes help guide one's approach to implementing an algorithm.

Recurrence equations are not at all uncommon; in fact they appear frequently in the numerical solution of differential equations. The rule here is to be wary of their predilection for treachery.

10.3.3. Analysis of a Simple Function Using Multiple Precision Arithmetic [Lec], [McC02], [Rum88], [Ral65], [Bailey], [Kre], [Hof97]

Here we look at a problem that seems, at least on the surface, to be a typical function that we might want to evaluate using a computer. In fact, this function was designed to illustrate an ominous fact of computing, and that is this – you really never know if your results are right because rounding errors can mutilate *any* fixed precision floating point computation.

Consider the following function, whose value we want at x = 77617 and y = 33096:

$$f(x,y) = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$$

Straight forward substitution using single and double precision gave:

f = 6.33825 x 10^{29} single precision

f = 1.17260394005317 double precision

All products were done using successive multiplications, to obviate the use of built-in library functions, that is, instead of writing x^n, each product was coded as x*x*…*x.

The fact that single and double precision disagree is not surprising. To confirm the double precision result, Leclerc [Lec] recomputed the function using extended precision (128 bit, 35 digits), and *found exact agreement with the double precision result – a finding that should certainly make one confident about the calculation*. However, what is surprising as well as disconcerting is the fact that *not even the sign is correct for either the double or extended precision computation!* In other words, extended precision, good to 35 digits, was found to be no better than ordinary double precision when dealing with this function.

The correct result, according to Leclerc [Lec] who used variable precision interval arithmetic, was trapped in the following 40 digit interval:

f = – 0.827396059946821368141165095479816292005

f = – 0.827396059946821368141165095479816291986

Using Mathematica, [McC02], the author confirmed this interval by computing the following exact (also to 40 digits) result:

f = – 0.8273960599468213681411650954798162919990

which, according to Mathematica, is equivalent to the fraction -54767/66192.

The reader might find it interesting to rationalize what has happened here. The following table gives the value of each term in the equation for f(x,y), to 40 digits of precision.

Table 10.5. Analysis Of: $f(x,y) = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$

Expression	**Exact Value**
(33375/100) * y^6	0.0000004386057508463931619307038310400000 x 10^36
x * x	0.0000000000000000000000000000060243986890000 x 10^36
11 * x^2 * y^2	0.00000000000000000072586759116001040064 0000 x 10^36
-y^6	-0.0000000013141745343712154664590376960000 x 10^36
-121 * y^4 - 2	-0.0000000000000000145173518207904485378 0000 x 10^36
(11 * x^2y^2 - y^6 – 121* y^4 - 2)	-0.0000000013141746069579745583624830100000 x 10^36
x^2 * (11 * x^2y^2 - y^6 – 121* y^4 - 2)	-7.917111779274712207494296632228773890 0000 x 10^36
(55/10) * y^8	7.917111340668961361101134701524942848 0000 x 10^36
(33375/100) * y^6 + x^2 * (11 * x^2y^2 - y^6 – 121 * y^4 - 2) + (55/10) * y^8	-2.0000000000000000000000000000000000000000
x/(2 * y)	1.172603940053178631858834904520183708001
(33375/100) * y^6 + x^2 * (11* x^2y^2 - y^6 - 121* y^4 - 2) + (55/10) * y^8 + x/(2 * y)	-0.8273960599468213681411650954798162919990

It is worth pointing out that the double precision (and extended precision for that matter) result, i.e., 1.17260…, is exactly equal to the just the *last term* in the function, namely, x/(2y), which means that all of the other terms in the function were rounded to zero. Mathematica confirms this result, provided we use the program's facility for exact arithmetic. That is how the numbers in the previous table were obtained. In addition, simply changing the order in which the terms in the function are evaluated (again in double precision) will change the final result due to how rounding is done. The reader might find this an instructive exercise in his or her programming language of choice.

Admittedly this problem was cleverly designed to demonstrate the nature of rounding errors, but it is not beyond the realm of possibility that a very unlucky coincidence could result in such a ill behaved function. The point here is that you can never really know if your results are correct and further, as we have seen agreement between double and extended precision is no guarantee of anything. This emphasizes the second point stated by Leclerc [Lec]: "By simply observing floating point results at increasing precision (single, double, extended), *no* indication of the seriousness of round off error may be given." The importance of this remark for scientific computation cannot be overstated.

CONCLUDING REMARKS [Bailey]

While it is true that computer round off error is unlikely to be a problem in the vast majority of routine computations, it is nevertheless, important to be aware of the potential for problems. However, it should also be obvious that a detailed floating point error analysis will be prohibitive for anything other than the most trivial of computational problems and that severely limits its usefulness. So what choices do we have? At the very least one should employ the simple expedients discussed above as standard programming practices. Even then, there is no assurance that your computations will be correct.

Probably one of the more successful general approaches to handling floating point rounding errors is the use of interval arithmetic, which, provided the problem is not ill conditioned, is capable of giving an acceptable bound for results. The strength of interval arithmetic is that it lends itself to software implementation which makes the analysis automatic. Such implementations are available in FORTRAN and C as well as for symbolic packages like Maple, Mathematica and MATLAB.

Arbitrary precision FORTRAN and C libraries are also available where the precision is dictated by software, not hardware, so speed is the tradeoff. Still, if the problem is sufficiently ill conditioned, the computed solution will be meaningless. In any case, interval arithmetic and arbitrary precision arithmetic are solutions well worth investigating when the problem is complicated and the error bounds must be controlled.