The Occasional Futility of Higher-Precision Arithmetic

Prof. W. Kahan
Elect. Eng. & Computer Science
University of California at Berkeley

Abstract:
There is a function G(x) which should take the value 1.0 for all real numbers x ; but a straightforward program to compute G(x) gets 0.0 instead for almost all of G(15), G(16), G(17), ..., G(9999) no matter how high the relative accuracy to which every arithmetic operation is carried out provided it is rounded or chopped to the nearest rational number representable with a previously chosen finite number of digits, as happens in most floating-point arithmetics.

Background:
When numerical computation loses accuracy to roundoff, the usual remedy is recomputation carrying higher precision. The reward for every additional digit carried is normally one more correct digit in final results. Occasionally improvement comes slower, another correct digit for every two or more additional digits carried, as when we compute unknowingly what turns out to be a multiple root. (Why *unknowingly* ? Because if we knew the root's multiplicity, we would prefer to solve instead a derivative equation with the same root but at lower multiplicity, thereby enhancing speed as well as accuracy.) In any event, accuracy is expected to improve as precision increases. Is that expectation always fulfilled?

There can be no guarantees. Increasing arithmetic's precision throughout a program without keeping a correct relationship among different precisions in different places can break a program that was previously cast iron. Increasing the precision of arithmetic without also conserving other important properties, for instance monotonicity and sign-symmetry, can also break a program that was previously cast iron. A numerically unstable program may seem to " dislike " certain data for which it produces inaccurate results though the desired results would have been unexceptionable; then increasing precision may well provide satisfactory results at data previously " disliked " without curing the instability. In that case, increasing precision diminishes the risk of suffering from the instability without eliminating it.

Alas, when one part of a program generates intermediate results too close to data that a subsequent part " dislikes," increasing precision may do no good at all. That situation is illustrated here by an example, the following short program:

```
Real Variables  x, y, z ;
Real Function   F(z) := if z = 0 then 1 else  (e^z - 1)/z ;
Real Function   Q(y) := |y - √(y²+1)| - 1/(y + √(y²+1)) ;
Real Function   G(x) := F(Q(x)²) ;
 For  integer n = 15 to 9999  do  Print( n, G(n) ).
```

Results from this Program:
If no rounding error occurred during the program's execution, it would print $G(n) = 1$ for all $n > 0$. That outcome is unlikely.

Transcribe the foregoing program into your favorite programming language and run it on your favorite computer using approximate arithmetic as precise as you like; it could be floating-point built into the hardware or simulated in software, or it could be rational arithmetic of limited accuracy chosen in advance. The likeliest result is $G(n) = 0$ for all 9985 values of n.

There may be exceptions. For instance, the HP-28S and other Hewlett-Packard programmable calculators that carry 12 sig. dec. compute $G(2) = G(42) = 1$ correctly but otherwise $G = 0$. Binary floating-point rounded to 24 sig. bits delivers 1 for $G(1)$, $G(7)$ and $G(2048)$ but otherwise 0 . Huge values of $G(n)$ might be generated by a peculiar arithmetic to be described later. Also described later are results from high-precision arithmetics that come with some automated algebra systems.

How does the program (not) work?
Our example concerns the computation of a value of a function $F(z)$ that can be rewritten

$$\begin{aligned} F(z) &= \int_0^1 e^{z\tau}\,d\tau \\ &= 1 + z/2 + z^2/6 + z^3/24 + z^4/120 + \ldots \end{aligned}$$

to reveal that it is really an entire analytic function with no singularity at any finite z . But the formula used to compute $F(z)$ in the program above is numerically unstable when $|z|$ is very small. For instance, when $|z|$ is so small that e^z must round to 1 then the computed value of $F(z)$ must be 0 instead of nearly 1 . (A simple program that computes $F(z)$ accurately practically everywhere in its domain will be presented later.)

In the absence of roundoff the function $Q(y)$ would reduce to 0 for every real y . Instead, $\sqrt{}(y^2+1)$ rounds to something a little different, say $\sqrt{}(y^2+1) + \eta y$ where ηy is comparable to a rounding error in y . After that y is subtracted exactly on most machines, and on all when y is a small integer; and subsequent rounding errors amount to a quantity ζ/y comparable to a rounding error in $1/y$. Hence, the computed value of $Q(n)$ must be $\sqrt{}(n^2+1) + \eta n - n - 1/(\sqrt{}(n^2+1) + n) - \zeta/n = \eta n - \zeta/n$. This value vanishes only when $\eta = \zeta/n^2$ is exceptionally small, smaller by a factor like $1/n^2$ than might reasonably be expected. We should be mildly surprised were $Q(n)$ to vanish once for any integer n between 15 and 9999 ; experience with a wide variety of floating-point arithmetics and with the approximate rational arithmetic capabilities of DERIVE™ on an IBM PC confirms this expectation. Therefore $Q(n)$ is almost never 0 but is instead roughly ηn , a quantity comparable to a rounding error in n .

Unless the function exp is computed to far higher relative accuracy than the $\sqrt{}$ function, the closest available rational approximation for the computed value of

$$\exp(Q(n)^2) \doteqdot \exp((\eta n)^2) = 1 + (\eta n)^2 + (\eta n)^4/2 + \ldots$$

is clearly 1 . That is why $G(n) = F(Q(n)^2)$ is almost always computed as 0 incorrectly instead of 1 correctly.

Something else might happen if $\exp(Q(n)^2)$ were approximated not by 1 but by the next available rational approximation after 1 ; call it $1+\delta$. Then the computed value of $G(n)$ could be huge, roughly $\delta/(\eta n)^2$. This peculiar event cannot occur if addition is chopped or rounded-to-nearest, which covers practically all computers' hardware floating-point arithmetics. Known exceptions are the IIT 2C87 and 3C87 floating-point coprocessor chips at their widest (10-byte) precision.

Numerically stable computation of F(z) :
The function $F(z)$ turns up in financial calculations. That is reason enough to facilitate its computation, as do some floating-point coprocessor chips like Motorola's 68881/2 and Intel's i80387 , and as do some run-time *math.* libraries like those in 4.3 BSD Berkeley UNIX™ and the Standard Apple™ Numerical Environment. They all provide a function like

$$\text{expm1}(z) := e^z - 1 = z + z^2/2 + z^3/6 + z^4/24 + \ldots$$

correct to full working relative accuracy no matter how small $|z|$ may be, thus eliminating instability from the obvious formula

$$F(z) := \text{if } z = 0 \text{ then } 1 \text{ else } \text{expm1}(z)/z .$$

Without expm1, a more devious program is needed:

```
Function F(z) :
   y := exp(z) ;  ...   rounded to working precision.
   if  y underflowed  then  return  F := -1/z
      else if  y = 1  then  return  F := 1
                      else  return  F := (y-1)/ln(y) .
```

Provided exp, ln, subtraction and division is each performed accurately to within a unit or two in its last place delivered, this program has been *proved* to deliver F accurately to within a few units in its last place too. If z is too big then exp(z) will overflow, but that lies beyond this note's scope. The point is that replacing the first program for F by this last one cures its numerical instability (despite violating folk-wisdom by an exact comparison of one floating-point number, y , with another, 1); and then computation yields $G(x) = 1$ for all $x > 0$.

Computerized Algebra Systems
To-day these systems offer the easiest way to exercize arithmetic of arbitrarily high precision, but their arithmetics behave in strange ways understood by only their creators, if anyone. Three such systems were invoked to compute $G(n)$, namely
DERIVE 2.5 from the Soft Warehouse Inc., Honolulu, HI,
Maple V from Waterloo Maple Software, Waterloo, Ont., Canada,
Mathematica 1.2 from Wolfram Research Inc., Champaign, IL.

The best results came from DERIVE. Although it could not prove that $G(n) = 1$ at all $n > 0$ (because it could not deduce that $|y - (n^2+1)| = \sqrt{(n^2+1)} - n$ at all $n > 0$), DERIVE did Simplify $G(1), G(2), \ldots, G(9999)$ to 1 correctly in exact arithmetic for both the first and last versions of F . In its approXimate modes DERIVE obtained 0 using the first F , 1 using the last F , for all of $G(1), G(2), \ldots, G(9999)$ for all Precisions tried, with one exception: $G(25)$ was computed as 1 at 60 digits of precision.

DERIVE tells us this about its approXimate arithmetic:

" In **approximate** mode, irrational numbers and large rational numbers simplify to the simplest rational number that approximates the original number accurate to the current precision."

The word " simplest " here appears to mean " with the shortest continued fraction expansion," but it is still ambiguous.

Maple V got the most consistent results, all wrong at first. In all arithmetics, exact and approximate, and for both versions of F , Maple V produced 0 for all of G(1), G(2), ..., G(9999) . Trouble seemed to stem from its interpretation of the conditional

if z = 0 then 1 else (exp(z) - 1)/z fi .

The test z = 0 was tried before z was simplified, so quotient (exp(z) - 1)/z was returned though z simplified to 0 later. Then the numerator vanished before the unsimplified denominator z was recognized as 0 , so Maple called the quotient 0 without checking first for a zero divisor! The same thing happened to

if y = 1 then 1 else (y-1)/ln(y) fi ;

consequently both programs for F yielded the same results. When approximate arithmetic was used instead of exact, some kind of schizophrenia allowed the divisor to vanish occasionally despite that the equality predicate predicted that it wouldn't, causing a Division-by-Zero message for some precisions but not others.

To get around Maple's foibles, Mike B. Monagan's delayed *'If'* in the Maple Share library's Spline package was adapted to our needs. This got the predicates z = 0 and y = 1 tested after instead of before evaluation. However we replaced " y = 1 " by " y-1 = 0 " because Maple thinks the first is *false* but the second *true* when y is 1.000000000 . Then quotients were put between apostrophes to delay their evaluation until after tests. And because the delayed *If* rebuffed floating-point evaluation by *evalf* , " G(convert(n, float)) " had to replace " G(n) ". At last, G(1), G(2), ... all simplified to 1 in exact arithmetic with all versions of F ; and G(1.0), G(2,0), ... yielded 0.0 with the first version of F , 1.0 for the last, in floating-point arithmetic at all precisions tried. These are the expected results, though perhaps right for the wrong reasons since so much of Maple's kernel's behavior is undocumented. Do its architects think it behaves so completely logically as needs no explanation?

Mathematica got the most perplexing results. In exact arithmetic Mathematica could not simplify G(1), G(2), ... to 1 because it could not simplify Abs[Sqrt[1+n^2] - n] to Sqrt[1+n^2] - n for each n = 1, 2, ... in turn; for instance, Abs[Sqrt[2] - 1] could not shake off its Abs. The results obtained in approximate arithmetic of various precisions depended at first sight upon the definition used for F . With the first unstable definition the results G(1), G(2), ... were all 0. for Precision below 17 sig. dec. but *Indeterminate* for higher precisions except possibly for a first few of G(1), G(2), ... which came out as 1. . With the last stable definition of F , the results G were all 1. .

But the reported Accuracy of each result was inexplicable. To make a long story short, the Accuracy reported by Mathematica for the numerical value N[F[1/10^19], k] for each precision k (no. of sig. dec. carried) is tabulated below, first for the

first unstable F and next for the last stable version:

Alleged Accuracy[N[F[1/10^19], k]] in dec. digits

	k:	16	17	18	19	20	21	22	23	24	25	...
1st F :		**323**	**17**	**18**	**19**	1	2	3	4	5	6	...
Last F :		16	17	18	19	20	21	3	4	5	6	...

R. J. Fateman reports that Mathematica 2.0 gets a somewhat better middle line: **323 -3 -2 -1** 1 2 3 4 5 6 ... .

These results perplex us in several ways. First, the accuracies in bold face pertain to numerical values N[F[...], k] displayed as 0. , quite wrong. All other values N[F[...], k] displayed as 1. , which is close enough. Mathematica 1.2 reported high accuracy for all incorrect results, Mathematica 2.0 only for the first of them. Both versions of Mathematica reported a drop in accuracy for the last version of F when precision increased to 22 sig. dec. carried. Why a drop? Why is the last F accurate to so many fewer digits than were carried despite error-analyses that prove hardly any accuracy can be lost?

Since Mathematica's inner workings are proprietarily secret, we have to speculate. Mathematica's floating-point arithmetic appears to match the underlying hardware's floating-point for precision below 17 sig. dec. For higher precisions, a kind of Significance Arithmetic is employed to perform automatically a form of error-analysis; the arithmetic discards all digits past the last digit regarded as reliable in any particular result.

A scheme like this was advocated about thirty years ago by Nick Metropolis and Bob Ashenhurst, but later abandoned when its misbehavior became apparent. First, discarding digits believed to be erroneous merely urged errors that would have grown linearly with time to grow exponentially instead. Second, unless it is implemented entirely pessimistically, Significance Arithmetic is quite often too optimistic about error, as in the table's bold-face entries above. Third, it ruins algorithms that cancel out correlated errors, as does the last version of F . Fourth, a far better scheme is Interval Arithmetic, which never under-estimates error and yet is usually less pessimistic.

Conclusion:
Numerical software that computes a continuously differentiable function of its input data is often presumed to depend at least continuously upon its rounding errors, so that one could imagine taking a " limit as roundoff tends to zero " to justify the presumption that increasing arithmetic's precision will increase accuracy at the end. But roundoff is discrete, not continuous; and the limiting process must not be applied indiscriminately lest it fail, as it does for the example offered herein.

The data-dependent test-and-branch in that example may be thought by some readers to cause the failure; however other more complex test-free examples confound the "limit" process by "converging" very convincingly but prematurely to an incorrect "limit."

Other readers may place their faith in a scheme that carries some indicator of accuracy with every intermediate result, as do both Significance Arithmetic and Interval Arithmetic. But when such schemes are used naively they cry " Wolf! " too often; either they destroy the accuracy of results they are supposed to guard, or they allege inaccuracy in results that could be proved quite accurate, but only in some other way.

The only way (if one exists) to guarantee numerical accuracy is to submit a candidate program to error-analysis, and to modify it as necessary to achieve stability, before resorting to more precise arithmetic. The analysis can be automated to some degree, and is often far easier with Interval Arithmetic than without.

In any event, the candidate program must be read and *understood* rather than merely *executed*. Moreover, to be understandable, its approximate arithmetic must have other mathematical properties besides just " enough precision "; more about that another day.

Acknowledgment:
I am grateful for advice from Prof. B. N. Parlett, assistance from Prof. R. J. Fateman, and support from the U. S. Office of Naval Research (N00014-90-J-1372) and the National Science Foundation (CCR-8812843).