Why is Floating-Point Computation so Hard to Debug when it Goes Wrong ?

Prof. W. Kahan
Math. and Computer Science Depts.
Univ. of Calif. @ Berkeley

Read this side first.

It presents a counter-intuitive example of floating-point computation gone utterly wrong.

“ Numerical Instability ” is too often misattributed to one or more of these three “ causes ” :

- Catastrophic Cancellation
- Gargantuan Intermediate Results
- Accumulations of Vast Hordes of Rounding Errors

More often, however, computations that go wrong do so for reasons similar to what causes the following example to go wrong. This example avoids the foregoing “ causes ” because it has ...

- No adds nor subtracts, so no cancellation.
- No divisions nor transcendental functions, so no gargantuan intermediate results.
- Only 257 algebraic operations, so no hordes of rounding errors.

None the less, this example manages to compute the simple function $h(x) := |x|$ wrongly on *every* computer whose floating-point (or other approximate) arithmetic carries less precision than about 39 sig. dec. Here it is:

```
Real Function  $h(x)$ :
        Real     $x$,  $y$,  $w$ ;
        Integer  $k$ ;
                $y := |x|$ ;
                For  $k$ = 1 to 128  do   $y := \sqrt{y}$  ;
                $w := y$ ;
                For  $k$ = 1 to 128  do   $w := w^2$ ;
                Return   $h := w$  .
```

In the absence of rounding error, this program’s final values of y and w would be respectively

$$y = |x|^{1/2^{128}} \quad \text{and} \quad w = y^{2^{128}} = |x| \;.$$

But something goes very wrong.

Run the program above on your own computer or calculator and plot $h(x)$ over $0 \le x \le 2$, say, to see it go utterly wrong. Then think about it before you read the explanation overleaf.

Read the Other Side First !

If your computer fails to get $h(0) = 0$ and $h(\pm 1) = 1$, something is seriously wrong with its arithmetic or with your program. Otherwise, diverse computers with different floating-point hardware get diverse results for $h(x)$, and none get anywhere near $h(x) = |x|$ for all x.

Computers and calculators whose square root and multiplication are correctly rounded get

$$h(x) = 0 \quad \text{for all} \quad |x| < 1, \quad \text{and} \quad h(x) = 1 \quad \text{for all} \quad |x| \ge 1 .$$

Computers, like Intel-based PCs, that can round $\sqrt{y}$ first to extra precision before rounding it again to the precision of y when it is assigned back to y, can get $h(x) = 1$ for all nonzero x.

Some calculators get an *ERROR* indication because $h(x)$ Overflows for all $|x| > 1$.

What goes wrong?

Let's try a crude Error-Analysis first: The final value computed for y is obscured by roundoff:

$$y = |x|^{1/2^{128}} \cdot (1 + \varepsilon)$$

in which $|\varepsilon|$ is some tiny quantity usually smaller than $5 \cdot 10^{-10}$ on a ten-digit calculator, or $2^{-53} \approx 1.11 \cdot 10^{-16}$ on a typical workstation computing in Double-Precision. Consequently, even if w's final value could be computed from y exactly, its value would be not $|x|$ but instead

$$|x| \cdot (1 + \varepsilon)^{2^{128}} \approx |x| \cdot \exp(\varepsilon \cdot 2^{128}) \quad .$$

If ε can run between $\pm 2^{-53}$ then $\exp(\varepsilon \cdot 2^{128})$ can run between $\exp(\pm 2^{75})$, which spans a range far beyond the Over/Underflow thresholds of floating-point hardware. Consequently our first try at a crude error-analysis leaves the computed value of $h(x) = w$ completely uncertain.

Let's try a more realistic analysis: Assume $x \ne 0$. Then $|x|$ lies between the Over/Underflow thresholds so its logarithm is bounded: $|\log(|x|)| < 12000$ on all commercially significant hardware. Then, because $\log(y) \approx \log(|x|)/2^{128} \approx \log(|x|)/3.4 \cdot 10^{38}$ is so tiny, the final computed value of y must differ from 1 by at most one unit in its last digit. When $y = 1$ exactly then finally $h(x) = w = 1$ too. When $y = 1.000\cdots001$ then the machine's square root is not rounded correctly ($\sqrt{1.000\cdots001}$ should have rounded to 1), and then $h(x)$ Overflows. When y is the floating-point number next less than 1, namely $0.999\cdots999$ in decimal arithmetic, then $h(x)$ Underflows to 0.0 ; this should happen for all nonzero $|x| < 1$ if square root is correctly rounded (once) because then $\sqrt{0.999\cdots999}$ rounds back to $0.999\cdots999$.

Thus, although the computed values of $h(x)$ may be completely wrong, they are completely explainable. Roundoff is amplified extravagantly because the problem's given data lie too near a *singularity*, namely that possessed by $\exp(2^N)$ at $N = \infty$, which is approached too closely when $N = 128$. In general, a computation that goes astray gets pushed there by a nearby singularity.

The questions raised here are discussed further in the posting at `<www.cs.berkeley.edu/~wkahan/Mindless.pdf>`.