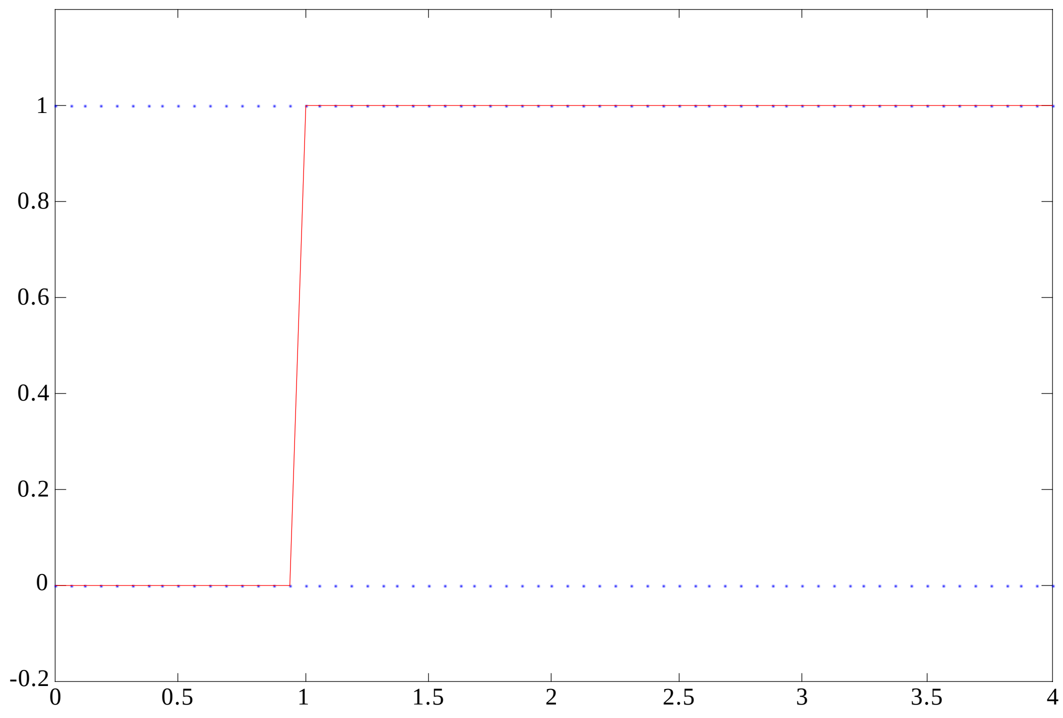
128 Squares of 128 Square Roots

Define a floating-point-valued function F(X) for nonnegative floating-point arguments X thus:

$$Y := \sqrt{\sqrt{\dots\sqrt{\sqrt{\sqrt{X}}}}} \ ; \qquad \dots\ 128 \text{ square roots } \dots$$
$$F := ((\dots((Y^2)^2)^2\dots)^2)^2 \ . \qquad \dots\ 128 \text{ squares.}$$

This computation commits 256 rounding errors, but no cancellations (there are no subtractions) nor divisions by tiny numbers (there are no divisions at all), so a naive expectation is that F(X) should match X within a few hundred units in the last digit carried during the computation. Something else happens; here is a plot of F(X) versus X :

128 squares of 128 sqrts



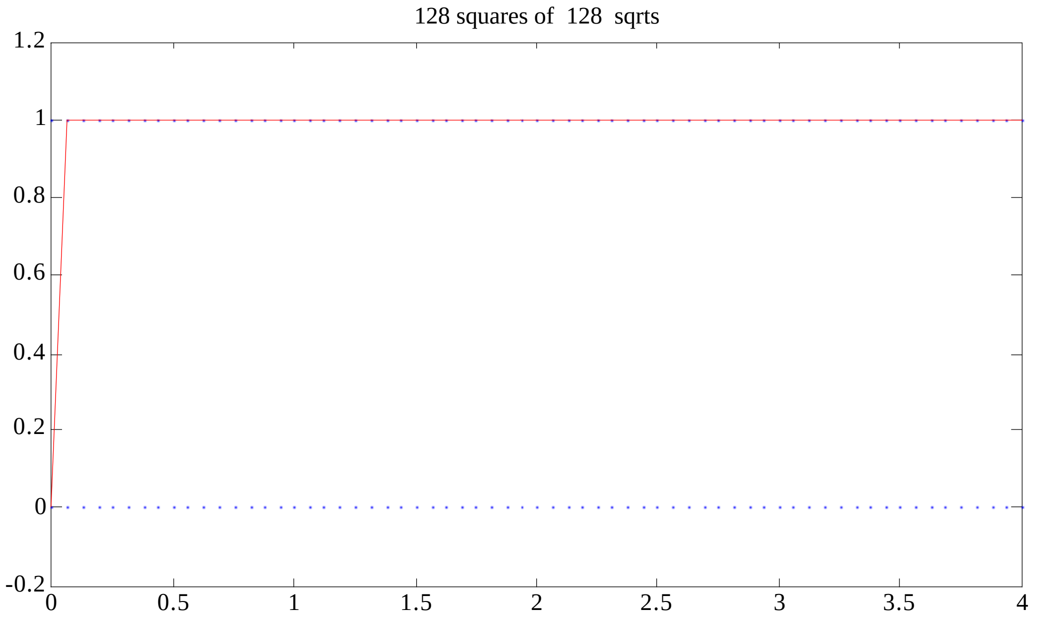
Matlab 3.5 on a Mac Quadra (68040) rounded to DOUBLE

The same thing happens on Sun SPARCs, on recent (for twenty years) hp calculators, on PCs and recent (1995+) Macintoshes using recent versions (5 or later) of MATLAB, and so on.

How can this graph be explained?

Of course F(0.0) = 0.0 and F(1.0) = 1.0 when F commits no rounding errors. But otherwise Y must be a rounded approximation to $X^{2^{-128}}$. Let's suppose that the computer rounds every square root correctly (error smaller than 0.5 in the last digit retained). If X > 1 then Y = 1 exactly; do you see why? And then F = 1 exactly too. But if 0 < X < 1 then Y = 0.999…999 or the the arithmetic's binary floating-point number next less than 1 ; do you see why? And then raising that number Y to the power 2^{128} *Underflows* (why?) to 0.0 , which is returned as F(X) .

However, some computers and calculators do something else; here is their graph of F(X) :



Matlab 3.5 on a Mac Quadra (68040) double-rounded to DOUBLE

Why?

The previous page's analysis leads us to conclude that this graph is produced on computers whose sqrt(x) function does not always return ($\sqrt{x}$ correctly rounded). Instead, if $x = 1 - \mu$ is the floating-point number next less than 1 , namely 0.9999…999 in decimal arithmetic, sqrt(x) returns 1 instead of x on those computers. Actually $\sqrt{x} = 1 - \mu/2 - \mu^2/8 - \dots$ falls so nearly halfway between x and 1 that sqrt(x) can be extremely nearly correctly rounded and yet be rounded wrongly in this case. But this is not what can happen on the Apple Mac Quadra's 68040 nor on PCs' Intel Pentiums and their clones. What can happen on them is far more bizarre.

Intel's floating-point arithmetic registers were designed in the late 1970s to evaluate expressions in a format wider than the 8-byte "double-precision" format currently supported by MATLAB and workstations like Sun SPARCs, Silicon Graphics MIPs, HP's old PA-RISCs and IBM Power-PCs. Registers in Motorola's 680x0 family (used in Apple Macintoshes before about 1993 when John Sculley switched Macs onto Power-PCs, and in Sun's workstations before they switched to their own proprietary SPARCs), behaved very much like Intel's. These extra-wide registers provide 11 extra bits of precision and 4 extra bits of exponent range to help evaluate subexpressions in such a way that the worst effects of exponent over/underflow and roundoff would be confined to assignment operations that round evaluated expressions to one of the narrower 8-byte "double" or 4-byte "float" variables. But designers and implementors of programming languages, with a few exceptions, failed to appreciate the virtues of a third extra- wide floating-point format. (Among the exceptions were Apple's languages for old 680x0- based Macs, Borland's languages for PCs, and C99.) In particular Bill Gates Jr., Microsoft's language expert, disparaged the extra-wide format in 1982 with consequences that persist today in Microsoft's languages for the PC. Sun's Bill Joy did likewise. See "How Java's Floating-Point Hurts Everyone Everywhere" and "Marketing *vs*. Mathematics" on my web page http://www.cs.berkeley.edu/~wkahan/… .

Some versions of MATLAB set bits that control rounding precision in the PC's floating-point registers to mimic SPARCs and other workstations' 8-byte floating-point, thus rounding sqrt(8-byte) once to 53 sig. bits. These versions get the graph with the step at x = 1 . Other versions of MATLAB set those control bits to benefit from the extra-precise arithmetic in a few computations, especially the accumulation of scalar products during multiplication of non-sparse matrices. These versions round sqrt(8-bytes) twice, first to extra-wide precision and then to MATLAB's 8-byte stored variables. These versions produce the graphs without a step at 1 .

Can you see why now?

Early versions (< 4) of MATLAB allowed its user to reset those control bits so as to mimic (or not) on PCs and old Macs the rounded-to-8-byte (or 4-byte) arithmetic of workstations like Sun SPARCs. The graphs on previous pages were produced that way. MATLAB 6 on PCs is again able to mimic workstations' arithmetics to some extent. What cannot now be mimicked so easily is the arithmetic on old supercomputers and some calculators whose square root software rounded sqrt(1.000…001) to 1.000…001 instead of 1.000…000 . The graph produced on these machines steps up to infinity at 1 because squaring 1.000…001 128 times overflows.

What's the Point?

So an unnecessarily complicated computation of $F(X) = X$ malfunctions because of roundoff. Why should we care? Because most computations deemed "numerically unstable" malfunction in a similar way. Examples include differential equation solvers and eigensystem solvers.

Suppose a floating-point program $F(X)$ is intended to compute a function $\mathit{f}(x)$. The program $F(X)$ you see is not the program you get. Instead you get a function $f(x, r)$ in which r is a column of rounding errors, one for every arithmetic operation in $F(X)$ susceptible to roundoff. Of course, r is unknown but tiny; and if $F(X)$ is algebraically correct then $f(x, o) = \mathit{f}(x)$. Consequently, in most cases, $f(x, r) = \mathit{f}(x) + (\partial f/\partial r)_{r=0}\cdot r + O(r)^2$. Here $\partial f/\partial r$ is the Jacobian matrix of first partial derivatives of $f(x, r)$ with respect to variables in r . If $\partial f/\partial r$ is not huge, the execution of program $F(X)$ will produce $f(x, r)$ with an error $f(x, r) - \mathit{f}(x) \approx (\partial f/\partial r)\cdot r$ that is negligible because every elemement of r is so tiny. Otherwise, when the error $f(x, r) - \mathit{f}(x)$ is intolerably big, it must be so because some elements of $\partial f/\partial r$ are gargantuan.

How can $\partial f/\partial r$ become gargantuan? Only if x is close, in some sense, to a *Singularity* of $f(x, r)$ where $\partial f/\partial r$ would become infinite. This singularity of f need not be a singularity of f , but rather an artifact of the formula chosen for F . In our example, $F(x) = (x^{2^{-N}})^{2^N}$ for $N = 128$ and $\mathit{f}(x) = x = F(x)$; but, if all but one crucial rounding error are ignored, $f(x, r) = \left(e^r \cdot x^{2^{-N}}\right)^{2^N}$ and $\partial f/\partial r = 2^N\cdot f(x, r)$. Therefore $f(x, r) - \mathit{f}(x) \approx 2^N\cdot r\cdot \mathit{f}(x)$, and when $N = 128$ we find that the relative error in $f(x, r)$ is a rounding error r (perhaps not so big as $2^{-53} \approx 10^{-16}$) amplified by $2^{128} \approx 10^{38}$. The singularity occurs when the parameter $N = 128$ (which appears in F and f but not f) is replaced by $N = +\infty$. This replacement seems drastic at first; actually it is a consequence of a singularity so strong that its effect is felt when N is big but not very big.

In general, singularities whose nearness amplifies roundoff intolerably will be unobvious. If they were always obvious, error-analysts would be mostly unemployed. Such is not the case.