# A Computation with Almost No Significance

Prof. W. Kahan
Mathematics Dept., and
Elect. Eng. & Computer Science Dept.
University of California
Berkeley CA 94720

## Abstract.

An amusing little program computes Z = 2.0 correctly, despite roundoff, only on computers that round products and quotients in the way specified by IEEE Standard 754 for Binary Floating-Point Arithmetic. On every other commercially significant computer the program computes the same wrong result Z = 1.0 . What makes the program act this way are properties of rounded multiplication and division unobvious enough to justify writing this note to explain them. No other reason for the program's existence is known.

## The Program.

All variables except j have the same floating-point type, be it *Single*, *Double* or *Extended* Precision. Variable j has type *Integer*. The only input is a variable W , which can take any value between 1,000 and 8,000,000 ; the larger values are best avoided on slower computers since the the program's running time is proportional to W . The only output is a variable Z which would have the value 2.0 if no rounding errors occurred and is computed correctly on almost all computers nowadays. But Z is miscomputed as 1.0 on just those machines that round floating-point multiplication and division in ways that do not conform to IEEE Standard 754 (1985). Here is the program, annotated:

```
Display " Enter a value between 1000 and 8000000 for W : ";
Input W ;
If W < 1000 or W > 8000000 then
      { Display " You seem to lack interest in this game.";
        Stop } ;
One := 1.0 ; Two := One + One ; H := One/Two ;          ... = 1/2
Three := One + Two ; R := Two/Three ;                   ... ≈ 2/3
E := (((R – H) – H) + (R – H)) + (R – H) ;              ... = 3*( Roundoff in R )
If E ≠ 0.0 then C := One/(E*E) else       …, since radix 3 is not used nowadays, …
      { Display " Please stop your compiler from over-optimizing E away!";
        Stop } ;
... Now C is a huge number, normally like (1/Roundoff)^2 .
```

```
S := One ;  Y := One ;                  ...  later  Y := 3, 5, 7, 9, ...
While  Y < W  do
  { D := Three ;                        ...  later  D = 1 + 2^j .
    for  j = 1 to 15  do
      { Q := Y/D ;                      ...  Q = Y/D  rounded.
        X := Q*D ;                      ...  X = Y + two rounding errors.
        E := (X – Y)*C ;                ...  E = (rounding error in  X)*C .
        S := E*E + S ;                  ...  S = 1 + ∑ E^2 .
       D := D – One + D } ;
    Y := Y + Two } ;                    ...;  now  S = 1 + ∑(roundoffs*C)^2 .
Z := One + One/S ;                      ...  If  all roundoffs = 0  then  Z = 2 .
Display  " Z = ", Z ;
Stop.
```

## What Happens?

If the program is run on a computer whose multiplication and division are rounded in conformity with IEEE standard 754 (1985), the final value displayed is Z = 2.0 , as would be expected if no rounding errors occurred. On all other computers the incorrect value Z = 1.0 is displayed. That correct 2.0 is harder to explain than this incorrect 1.0 , so this is where we shall begin.

Consider first a programmable calculator that rounds every arithmetic operation to ten significant decimal digits. It will first compute R = 0.6666666667 instead of 2/3 , and then it will find E = 3R – 2 = 0.0000000001 exactly and C = 1.0E20 . In the innermost loop the calculator will first compute not Q = 1/3 but Q = 0.3333333333 , and then X = Q*3 = 0.9999999999 exactly so E = (1.0E–10)*1.0E20 = 1.0E10 instead of 0 . This causes S to be increased and rounded to 1.0E40 , and subsequent passes around the loop increase S beyond that. Finally Z rounds to 1.0 , as predicted. The same final result Z = 1.0 is computed on almost every other decimal calculator regardless of how many significant decimals it carries and whether it rounds or chops. The exceptional calculators are *Casio* models that do not compute 0.333...333*3 = 0.999...999 correctly but instead round it cosmetically to 1.000...00 because that displays as a small integer 1 ; only for very large values Y ( and W ) can such a machine produce nonzero values for E and hence Z = 1.0 .

Next consider IBM mainframes descended from the IBM /370 , a family of machines that were once ubiquitous. These machines perform *hexadecimal* floating-point arithmetic with products and quotients that are *chopped* to fit the floating-point format in use, which may be *Single* Precision with 6 sig. hex. digits, *Double* Precision with 14, or *Extended* (Quadruple) Precision with 28. These computers get $0.555...555_H$ ( in hexadecimal ) instead of 1/3 for Q , and then $0.FFF...FFF_H$ instead of 1 for X , so E is nonzero and finally Z = 1.0 instead of 2.0 . This miscomputation occurs also on all computers that chop products and quotients instead of rounding them, although some such computers ( CRAYs) introduce unnecessary extra rounding errors when X – Y is computed. On all computers that chop, Q := Y/D is chopped to something actually smaller than Y/D if it is not exact, and then X := Q*D is chopped to something smaller than Y , so E := (X – Y)*C turns out to be a fairly big negative number and finally Z = 1.0 .

Next consider the DEC VAX™ , with its four binary floating-point formats:
(F) Single Precision rounded to t = 24 sig.bits.
(G) Double Precision rounded to t = 53 sig.bits.
(D) Double Precision rounded to t = 56 sig.bits.
(H) Extended Precision rounded to t = 113 sig.bits.
Arithmetic on this machine is comparatively well-behaved, yet it computes Z = 1.0 , instead of the correct 2.0 , in a way that depends upon whether the number t of sig.bits carried is even or odd. Let us consider these cases separately.

When t is even, the value computed for R is not 2/3 but $0.1010...1011_2$ ( in binary ), and then $E = 2^{-t}$ and $C = 2^{2t}$. When Y = 13 and D = 3 , the value computed for Q is not 13/3 but $100.01010...1011_2$ , and $Q*3 = 1101.00000...0001_2$ rounds up to $X = 1101.00000...001_2$ instead of $1101._2 = 13$ . This produces $E = 2^{t+4}$ and finally Z = 1.0 instead of 2.0 .

When t is odd, the value computed for R is not 2/3 but $0.1010...101_2$ , and then $E = -2^{-t}$ and $C = 2^{2t}$ . When Y = 7 and D = 3 , the value computed for Q is not 7/3 but $10.0101...011_2$ , and $Q*3 = 111.0000...001_2$ rounds up to $X = 111.0000...01_2$ instead of $111._2 = 7$ . This produces $E = 2^{t+3}$ and finally Z = 1.0 instead of 2.0 .

On a VAX , all the cases that produce nonzero values for E do so when a value $xxxx.00...001_2$ rounds up to $X = xxxx.00...01_2$ instead of down to $Y = xxxx._2$ . This happens because a VAX rounds all such "halfway cases" away from zero. Later we shall see that rounding these cases to "nearest even", as is required to conform to IEEE standard 754, would keep X = Y .

Thus we conclude that Z will finally be computed incorrectly as 1.0 instead of 2.0 on all the machines in the following classes:
-- Those with radix larger than 3 : For Q := 1/3 they compute a value Q slightly different from 1/3 , after which X = Q*3 exactly, so $X \neq 1$ and hence $E = (X - 1)*C \neq 0$ .
-- Those that chop products and quotients: Whenever Q := Y/D is inexact it is too small, and then X := Q*D < Y too.
-- Binary machines that round halfway cases away from zero, as does a VAX, or round them differently than specified by IEEE standard 754.

## Why IEEE Standard 754 gets Z = 2.0 .

IEEE 754 specifies binary floating-point arithmetic with t = 24 sig.bits for *Single* Precision, t = 53 for *Double*, and any t > 79 for D*ouble-Extended* Precision. For the program above the only necessary constraints upon t are that $t - 1 \geq \log_2 W \geq \log_2 Y$ and $t > 15 \geq j$ . The IEEE standard also specifies a *Rounding Mode* to be supplied by default ( in lieu of an explicit request for something else ). The default mode rounds to nearest, and breaks ties in halfway cases by rounding to "nearest even"; this will be explained later when we need it although it is explained also in items cited in the Reading List below.

Henceforth that default rounding mode is taken for granted.

Now the crucial insight is the observation that the two operations

Q := Y/D ;

X := Q*D ;

*always* produce X = Y *exactly,* despite rounding errors in both Q and X , provided Y and D are floating-point variables with positive integer values subject to the following constraints:

Y is not too enormous; in fact, $Y \le 2^{t-1}$ .

D is a sum of two powers of 2 ; *i.e.*, $D = 2^j + 2^k$ .

The constraint upon D may seem peculiar, but the necessity for some such constraint can be demonstrated as follows.

The first several integers D that are sums of two powers of 2 are

1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 16, 17, 18, 20, 24, 32, 33, 34, 36, 40, ... .

The first positive integer not in this sequence is 7 . Let us try D := 7 and, for IEEE 754 Single Precision with t = 24 try Y := 31 in the two operations above. Since $31/7 = 100.011011011..._2$ in binary, Q := Y/D rounds to $Q = 100.011011011011011011011_2$ . Then

$X := Q*D = 11110.111111111111111111101_2$ rounds down to

$X = 11110.111111111111111111_2 < Y = 31$ .

For IEEE 754 Double with t = 53 , try Y := 29 to get X > Y . When D := 11 try Y := 13 and Y := 15 to get $X \ne Y$ . The first such instances for D := 31 are Y := 497 and Y := 249 . For D := 511 they are somewhat big; Y := 32705 and Y := 130817 . X = Y rather often. Only by suitably constraining D can we keep X = Y for *all* integers Y that are not too enormous.

Next we explain why the foregoing peculiar constraints do keep X = Y so long as its floating-point value's last sig.bit is zero. To do so we shall standardize integers Y and D by multiplying them by powers of 2 so chosen that afterwards $D = 1 + 2^j$ and Y is an even integer in the range $2^{t-1} \le Y \le 2^t - 2$ . Multiplications by powers of 2 introduce no rounding errors, so they have no effect upon whether the subsequent operations Q := Y/D and X := Q*D will produce X = Y . And since no rounding error would occur if D = 2 , we assume henceforth that j > 0 .

Now the explanation breaks into two cases called "Low" and "High" in accordance with the way Y/D compares with $2^{t-1-j}$ .

**Low Case:** $2^{t-j-2} < 2^{t-1}/(1 + 2^j) \le Y/D < 2^{t-j-1}$ .

In this case the quantity $2^{j+1}Y/D$ lies in the interval $2^{t-1} < 2^{j+1}Y/D < 2^t$ , so it must round to its nearest integer with a rounding error strictly smaller in magnitude than 1/2 ;

$$2^{j+1}Q = (2^{j+1}Y/D) \text{ rounded} = 2^{j+1}Y/D + r/D \quad \text{with} \quad |r/D| < 1/2 .$$

In fact, r is a remainder, a signed integer strictly between –D/2 and D/2 , so $-2^{j-1} \le r \le 2^{j-1}$ . Now, $Q*D = Y + r/2^{j+1} = Y \pm (\text{ at most } 1/4 )$ , and this rounds to the nearest integer since $2^{t-1} \le Y \le 2^t - 2$ . Therefore Q*D rounds to X = Y exactly in this Low Case. Note that Q*D cannot fall halfway between two integers, but a halfway case can arise when $Y = 2^{t-1}$ and Q*D = Y – 1/4 ; fortunately that halfway case rounds up to Y because its last sig.bit is zero.

**High Case:** $2^{t-j-1} \le Y/D \le (2^t - 2)/(1 + 2^j) < 2^{t-j}$ .

In this case the quantity $2^jY/D$ lies in the interval $2^{t-1} \le 2^jY/D < 2^t$ , so it must round to its nearest integer with a rounding error strictly smaller in magnitude than 1/2 ;

$$2^jQ = (2^jY/D) \text{ rounded} = 2^jY/D + r/D \quad \text{with} \quad |r/D| < 1/2 .$$

Again, r is a remainder, a signed integer strictly between –D/2 and D/2 , so $-2^{j-1} \le r \le 2^{j-1}$ . Now, $Q*D = Y + r/2^j =$ Y ± ( at most 1/2) , and this rounds to its nearest integer since $2^{t-1} < Y \le 2^t - 2$ . The nearest integer is unambiguously Y unless Q*D is a half-integer, in which event IEEE 754 will round it to the nearest even integer, which turns out to be Y again. Therefore Q*D rounds to X = Y exactly in this High Case too. End of explanation.

The High Case is the one that can fail on a DEC VAX when a half-integer Q*D is rounded up to X = Y+1 . And it could fail for IEEE 754 if Y were too big, say an odd integer between $2^{t-1}$ and $2^t$ , whereupon Q*D rounded to the nearest even integer would produce $X \ne Y$ .

## Roundoff is Accidental, Not Random but Ragged.

The phenomenon that has just been explained must be very special to ensure that the two rounding errors committed during the two operations Q := Y/D and X := Q*D will *always* neatly cancel and leave X = Y . Were Y and D independent random floating-point numbers we should expect $X \ne Y$ about 11% of the time. And then those operations' rounding errors are not random; they are still correlated so strongly that, despite $X \ne Y$ , the further computation of G := X/D *always* produces G = Q *exactly* on every computer that rounds correctly ( rather than chops ) to keep the error no worse than half a unit in the last place, regardless of radix and the treatment of halfway cases, provided ( as is almost universally the case ) the arithmetic carries a constant number ( at least two ) of significant digits. The proof that G = Q despite three roundings is more complicated than the explanation above, so it has been relegated to the Appendix.

## Conclusions.

One might wish that the two operations Q := Y/D and X := Q*D would *always* yield X = Y *exactly,* but that is too much to ask of any computer's floating-point. IEEE 754 ensures that X = Y whenever Y is any integer no bigger than $2^{23}$ = 8,388,608 and D is any integer drawn from the interesting sequence

1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 16, 17, 18, 20, 24, 32, 33, 34, 36, 40, ... .

No other commercially significant floating-point arithmetic does this. Whether this phenomenon has any commercial significance remains to be seen. Perhaps it is no more than another small piece of evidence supporting the claim that the main benefit derived from IEEE 754 is …

**Program Importability:** Almost any application of floating-point arithmetic, programmed in a higher-level language and designed to work on a few different families of computer arithmetics in existence before IEEE Standard 754, will work at least about as well on a machine conforming to IEEE 754 as on any other nonconforming computer with similar capabilities ( memory, speed, word-size and compilers ).

The coincidences $X = Y$ and $G = Q$ are instances of underappreciated phenomena that cast a pall over procedures often used to assess the accuracy of numerical software. The software under test is offered test data chosen because the ideal result, that would be computed absent rounding errors, is known. Then the difference between the computed and ideal results provides a sample of the software's (in)accuracy. A realistic assessment can require many such samples because roundoff is accidental, not random but ragged. And small integer test data or results are best avoided since they often incur atypical roundings and coincidences like those viewed above.

## Reading List:

ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic, published by the Inst. of Electrical and Electronic Engineers, Inc., 345 E. 47th St., New York NY 10017 (item SH10116).

W. J. Cody *et al*. "A Proposed Radix- and Word-length-independent Standard for Floating-Point Arithmetic" in *IEEE MICRO* vol. 4 no. 4 (August, 1984) pp. 86-100. ... Easier to read.

*Apple Numerics Manual*, 2nd ed. (1988), Addison-Wesley, Mass. ... Describes the most conscientious available implementation of IEEE 754 on early μ680x0-based Macintoshes.

Harold G. Diamond "Stability of Rounded Off Inverses Under Iteration" in *Mathematics of Computation* **32** (1978) pp. 227-232 ... Slightly weaker inferences from much more general hypotheses, and another reading list.

## APPENDIX:

Version dated 16 Nov. 1988

Proved hereunder is that correctly rounding the three operations

$$Q := Y/D ; \quad X := Q*D ; \quad G := X/D ;$$

in floating-point arithmetic always yields $G = Q$ regardless of whether $X \neq Y$. The coincidence resembles Harry Diamond's Theorem but is a stronger inference from a special hypothesis that lacks his strict convexity.

Arithmetic is assumed *Correctly Rounded* to t sig.digits of an integer *Radix* $Ç \geq 2$, which keeps the rounding error no bigger than one half a unit in the last sig.digit; the proof remains valid regardless of whether halfway cases are rounded away from zero or to nearest even. Q and X are assumed not to over/underflow; and for simplicity's sake we assume that $t > 1$. Otherwise Y and D are arbitrary floating-point numbers of the same precision as the arithmetic. Then we use the abbreviation $B := Ç^{t-1}$, so that the floating-point numbers between B and ÇB consist of the integers B, B+1, B+2, ..., ÇB – 2, ÇB – 1, ÇB . The next floating-point number after ÇB is ÇB + Ç ; the one before B is B – 1/Ç . We also use repeatedly the fact that multiplications and divisions by B and by any other integer power of Ç are exact absent over/underflow.

When D is a power of Ç , or when $D = Y$ , no roundoff occurs to prevent $G = Q$ . Therefore we can disregard these cases henceforth when we scale the data Y and D to integers in the ranges

$$B \leq Y \leq ÇB - 1 \quad \text{and} \quad B + 1 \leq D \leq ÇB - 1 .$$

Doing so restricts $Y/D$ to the range

$$1/Ç < B/(ÇB - 1) \le Y/D \le (ÇB - 1)/(B+1) < Ç .$$

This range will be broken into two cases: LOW , when $Y/D < 1$ , and HIGH , when $Y/D > 1$ . Later both cases will be subdivided further.

**LOW Case:** $B \le Y < Y + 1 \le D \le ÇB - 1$ .
In this case $B/(ÇB - 1) \le Y/D \le (D - 1)/D \le 1 - 1/(ÇB - 1)$ ; now multiply by $ÇB$ to put $ÇBY/D$ into a range where it must round to the nearest integer;

$$B + B/(ÇB - 1) \le ÇBY/D \le ÇB - 1 - 1/(ÇB - 1) .$$

Then $ÇBY/D$ rounds to the nearest integer $ÇBQ$ between $B$ and $ÇB - 1$ inclusive. This $ÇBQ$ is a quotient whose signed remainder is $r := ÇBY - ÇBQD$ , and $|r| \le D/2$ .

If $Q = 1/Ç$ then both $X = QD$ and $G = X/D = Q$ exactly, so we need consider further only the case $B + 1 \le ÇBQ = ÇBY/D - r/D$ . Then

$$QD = Y - r/(ÇB) \quad \text{and} \quad |r/(ÇB)| \le D/(2ÇB) \le (ÇB - 1)/(2ÇB) < 1/2 .$$

Therefore $QD$ rounds to $X = Y$ and then $G = Q$ exactly, except perhaps if $QD < B - 1/(2Ç)$ .

In this sub-case when $B - 1/(2Ç) > QD = Y - r/(ÇB) > B - 1/2$ , still $ÇQD$ rounds to an integer $ÇX = ÇQD + f$ with $|f| \le 1/2$ ; and then $ÇBX/D = ÇBQ + fB/D$ differs from the integer $ÇBQ$ by $|fB/D| \le (1/2)B/(B + 1) < 1/2$ , so $ÇBX/D$ rounds to $ÇBG = ÇBQ$ . Therefore $G = Q$ exactly again, finishing off the LOW Case.

**HIGH Case:** $B + 1 \le D < D + 1 \le Y \le ÇB - 1$ .
In this case $1 + 1/(ÇB - 2) \le (D+1)/D \le Y/D \le (ÇB - 1)/(B + 1)$ , so we multiply by $B$ to put $BY/D$ into a range where it must round to the nearest integer;

$$B + B/(ÇB - 2) \le BY/D \le ÇB - Ç - 1 + (Ç + 1)/(B +1) .$$

Then $BY/D$ rounds to the nearest integer $BQ$ between $B$ and $ÇB - Ç$ inclusive. This $BQ$ is a quotient whose signed remainder is $r = BQD - BY$ , and $|r| \le D/2$ .

If $Q = 1$ then both $X = QD$ and $G = X/D = Q$ exactly, so we need consider further only the case $B + 1 \le BQ = BY/D + r/D \le ÇB - Ç$ . Then

$$B+2+1/B \le QD = Y + r/B \le Y + D/(2B) \le ÇB - 1 + (ÇB - 2)/(2B) < ÇB - 1 + Ç/2 ;$$

therefore $QD$ rounds to the nearest integer except perhaps in the rare case that $QD > ÇB + 1/2$ , which we shall deal with later.

In the likely sub-case when $QD$ rounds to the nearest integer $X$ , the difference $f := QD - X$ satisfies $|f| \le 1/2$ ; and then $BX/D = BQ - fB/D$ with $|fB/D| \le (1/2)B/(B + 1) < 1/2$ , so $BX/D$ rounds to $BG = BQ$ which is the nearest integer. This is why $G = Q$ exactly in this sub-case.

In the rare sub-case that $ÇB + 1/2 < QD < ÇB - 1 + Ç/2$ , which cannot arise unless $Ç \ge 4$ , the rounded value of $QD$ is $X = ÇB$ . Now

$G = X/D$ rounded $\ge Y/D$ rounded $= Q$ on the one hand, and
$G = ÇB/D$ rounded $\le (QD)/D$ rounded $= Q$ on the other.

Therefore $G = Q$ exactly again, and the HIGH Case is finished.

End of proof.

### Testing Correctly Rounded Multiplication and Division.

Whether the coincidence just proved has any worthwhile application is not known. At first sight it seems to supply a simple way to test whether computers round floating-point multiplication and division correctly. The test program would generate a large number of pairs Y and D , perhaps at random, and for each pair test whether the three operations

Q := Y/D ; X := Q*D ; G := X/D ;

yielded G = Q exactly. A failure would signify that one or both of multiplication and division is not correctly rounded.

Unfortunately this test can succeed, finding G = Q every time, even if multiplication and/or division is merely *almost* correctly rounded in so far as its error exceeds one half in the last sig.digit by extremely little extremely rarely. Therefore such a test is valuable only as a quick way to expose arithmetic that is very incorrectly rounded. More refined tests have been supplied in some of the author's reports:

– *Checking Whether Floating-point Division is Correctly Rounded* (April 1987)

– *To Test Whether Binary Floating-Point Multiplication is Correctly Rounded* (July 1988)

(These have been superseded by papers produced by Michael Parks in the late 1990s.)

A *Floating Point Validation* (*FPV*) package of software that tests all the arithmetic operations ( +, –, *, / and $\sqrt{\ }$ ) can be purchased for under $1000 from the *Numerical Algorithms Group*, 1101 31st Street, Suite 100, Downers Grove IL 60515-1263 .

## Acknowledgment:

Some of this work was supported by grants to Prof. Beresford N. Parlett from the U. S. Office of Naval Research under contract N00014-85-K-0180 .