A Computer Program with Almost No Significance

Prof. W. Kahan
Elect. Eng. & Computer Science Dept.
Univ. of Calif.
Berkeley CA 94720

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. The variable j is an integer. The only input is the variable A , which can take any value between 1,000 and 8,000,000 ; but since the program's running time is proportional to A , larger values are best avoided on slower computers. The only output is the variable Z , which would have the value 2.0 if no rounding errors occurred; otherwise Z will be miscomputed as 1.0 on all computers except those (now in the majority) whose arithmetics conform to IEEE Standard 754 (1985) . Here is the program, plus annotations:

```
Display " Enter a value between 1000 and 8000000 for A : ";
Input A ;
If A < 1000 or A > 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
U := (((R-H)-H) + (R-H)) + (R-H) ; ...   = 3*( Roundoff in R )
If U = 0.0 then C := 1.0E36 else C := One/(U*U) ;
...  Now C is a huge number, normally like  (1/Roundoff)²

S := One ;  I := One ;         ...;  later  I = 1, 3, 5, 7, ...
While I < A do
   { D := Three ;                       ...;  later  D = 1 + 2^j
     for j = 1 to 15 do
         { Q := I/D ;                   ...   Q = I/D  rounded
           X := Q*D ;                   ...   X = I + roundoff
           E := (X-I)*C ;               ...   E = roundoff*C
           S := E*E + S ;               ...   S = 1 + ∑ E²
           D := D - One + D } ;
     I := I + Two } ;      ...;  now  S  =  1 + ∑(roundoff*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, 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. This incorrect 1.0 is easier to explain than the correct 2.0, 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 U = 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 certain 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 I (and A) can such a machine produce nonzero values for E and hence Z = 1.0.

Now consider the IBM /370, a family of machines that are used very widely. 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. The same kind of thing happens on all computers that chop products and quotients instead of rounding them, although some of those computers introduce an unnecessary extra rounding error when X−I is computed; on all computers that chop, Q := I/D is chopped to something actually smaller than I/D if it is not exact, and then X := Q*D is chopped to something smaller than I, so E := (X−I)*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 significant bits carried is even or odd. Let us consider those cases separately.

When t is even, the value computed for R is not 2/3 but $0.1010...1011_2$ (in binary), and then $U = 2^{-t}$ and $C = 2^{2t}$. When I = 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 $U = -2^{-t}$ and $C = 2^{2t}$. When $I = 7$ 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 $I = xxxx._2$. 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 $X = I$.

Thus we conclude that Z will finally be computed incorrectly as 1.0 instead of 2.0 on all the following classes of machines:
-- Those with radix greater 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 $Q := I/D$ is inexact it is too small, and then $X := Q*D < I$ 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 Double-Extended Precision. For the program above the only necessary constraints upon t are that $t-1 \geq \log_2 A > \log_2 I$ 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 we take that default rounding mode for granted.

Now the crucial insight is the observation that the two operations

```
        Q := I/D ;
        X := Q*D ;
```

always produce $X = I$ exactly, despite rounding errors in both Q and X , provided I and D are floating-point variables with positive integer values subject to the following constraints:
   I is not too enormous; in fact, $I \leq 2^{t-1}$.
   D is a sum of two powers of 2 ; i. e., $D = 2^j + 2^k$.
The constraint upon D may seem peculiar, but its necessity can be demonstrated as follows.

The first few 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, ...
The first integer not in this sequence is 7 . Let us try $D = 7$ and, for IEEE 754 Single Precision with $t = 24$, try $I = 31$

in the two operations above. Since $31/7 = 100.011011011\ldots_2$ in binary, $Q := I/D$ rounds to $Q = 100.0110110110110110110110_2$. Then $X := Q*D = 11110.11111111111111111111101_2$ rounds down to $X = 11110.11111111111111111111_2 < I = 31$. For IEEE 754 Double Precision with $t = 53$, try $I = 29$ and get $X > I$. When $D = 11$ the trial values $I = 13$ and $I = 15$ cause $X \neq I$. If we wish to keep $X = I$ for all integers I that are not too enormous, some constraint upon D must be accepted.

If we accept the two constraints upon I and D, our next task is to demonstrate why they imply $X = I$ exactly. For this purpose we shall standardize the integers I and D by multiplying them by powers of 2 so chosen that afterwards $D = 1 + 2^j$ and I is an even integer in the range $2^{t-1} \le I \le 2^t-2$. Multiplications by powers of 2 introduce no rounding errors, so they have no effect upon whether the subsequent operations $Q := I/D$ and $X := Q*D$ will produce $X = I$. And since no rounding error would occur if $D = 2$, we assume henceforth that $j > 0$.

Now the demonstration breaks into two cases called Low and High according to the way I/D compares with 2^{t-j-1}.

Low Case: $2^{t-j-2} < 2^{t-1}/(1+2^j) \le I/D < 2^{t-j-1}$.
In this case the quantity $2^{j+1}I/D$ lies in the interval

$$2^{t-1} < 2^{j+1}I/D < 2^t ,$$

so it must round to the nearest integer with a rounding error strictly smaller in magnitude than 1/2 ;

$$2^{j+1}Q = (2^{j+1}I/D) \text{ rounded } = 2^{j+1}I/D + r/D$$

with $|r/D| < 1/2$. In fact, r is a remainder, an integer strictly between $-D/2$ and $D/2$, so $-2^{j-1} \le r \le 2^{j-1}$. Now

$$Q*D = I + 2^{-j-1}r = I \pm (\text{ at most } 1/4) ,$$

and this rounds to the nearest integer since $2^{t-1} \le I \le 2^t-2$. Therefore $Q*D$ rounds to $X = I$ exactly in this case. Note that $Q*D$ cannot fall halfway between two integers, but a halfway case can arise when $I = 2^{t-1}$ and $Q*D = I - 1/4$; fortunately both IEEE 754 and the DEC VAX round that halfway case up to I.

High Case: $2^{t-j-1} \le I/D \le (2^t-2)/(1+2^j) < 2^{t-j}$.
In this case the quantity $2^jI/D$ lies in the interval

$$2^{t-1} \le 2^jI/D < 2^t ,$$

so it must round to the nearest integer with a rounding error strictly smaller in magnitude than 1/2 ;

$$2^jQ = (2^jI/D) \text{ rounded } = 2^jI/D + r/D$$

with $|r/D| < 1/2$. Again, r is a remainder, an integer strictly between $-D/2$ and $D/2$, so $-2^{j-1} \le r \le 2^{j-1}$. Now

$$Q*D = I + 2^{-j}r = I \pm (\text{ at most } 1/2) ,$$

and this rounds to the nearest integer since $2^{t-1} < I \le 2^t-2$. The nearest integer is unambiguously I 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 I again. Therefore $Q*D$ rounds to $X = I$ exactly in this case too. End of demonstration.

The High Case is the one that can fail on a DEC VAX when a half-integer $Q*D$ is rounded up to $X = I+1$. And it could fail for IEEE 754 if I were too enormous, so much so that it were an odd integer between 2^{t-1} and 2^t, in which case rounding $Q*D$ to the nearest even integer would yield $X \neq I$.

Roundoff is Not Random.
The situation we have just finished studying must be very special to ensure that the two rounding errors committed during the two operations Q := I/D and X := Q*D will neatly cancel and leave X = I . Normally, in the absence of constraints upon I and D , we must expect X ≠ I from time to time. However, the behavior of those rounding errors is not random; they are still correlated strongly enough that, even if X ≠ I , 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 universally the case) the arithmetic carries a constant number (at least two) of significant figures. The proof that G = Q resembles the one above but is more complicated; see the Appendix.

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

1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 16, 17, 18, 20, ...

This is better than every other commercially significant floating-point arithmetic can do; but 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 this:

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 computers 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).

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 and most widely available implementation of IEEE 754.

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

APPENDIX: We prove here that correctly rounding the operations

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

in floating-point arithmetic always yields $G = Q$.

Arithmetic is assumed *correctly rounded* to t sig. digits of radix β , 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 up or to nearest even. X is assumed not to over/underflow; and for simplicity's sake we also assume that $t > 1$. Then we use the abbreviation $B = \beta^{t-1}$, so that the floating-point numbers between B and βB consist of the integers $B, B+1, B+2, \ldots, \beta B-2, \beta B-1, \beta B$. The next floating-point number after βB is $\beta B+\beta$; the one before B is $B-1/\beta$. We also use repeatedly the fact that multiplication and division by B or by any other integer power of β are exact.

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

$$B \le I \le \beta B-1 \quad \text{and} \quad B+1 \le D \le \beta B-1 \;.$$

Then I/D is restricted to the range

$$1/\beta \;<\; B/(\beta B-1) \;\le\; I/D \;\le\; (\beta B-1)/(B+1) \;<\; \beta \;;$$

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

LOW Case: $B \le I < I+1 \le D < \beta B-1$.
In this case $B/(\beta B-1) \le I/D \le (D-1)/D \le (\beta B-2)/(\beta B-1)$; now multiply by βB to put $\beta BI/D$ into a range where it must round to the nearest integer;

$$B + B/(\beta B-1) \le \beta BI/D \le \beta B-1 - 1/(\beta B-1) \;.$$

Then $\beta BI/D$ rounds to the nearest integer βBQ between B and $\beta B-1$ inclusive. This βBQ is a quotient whose remainder is

$$r = \beta BI - \beta BQD \;, \quad \text{and} \quad |r| \le D/2 \;.$$

If $Q = 1/\beta$ then both $X = QD$ and $G = X/D = Q$ exactly, so we need consider only the case $B+1 \le \beta BQ = \beta BI/D - r/D$. Then $QD = I - r/(\beta B)$ and $|r/(\beta B)| \le D/(2\beta B) \le (\beta B-1)/(2\beta B) < 1/2$. Therefore, unless $B - 1/(2\beta) > QD$, QD rounds to $X = I$ and then $G = Q$ exactly.

In the case when $B - 1/(2\beta) > QD = I - r/(\beta B) \ge B - 1/2$, still βQD rounds to an integer $\beta X = \beta QD + f$ with $|f| \le 1/2$; and then $\beta BX/D = \beta BQ + fB/D$ differs from the integer βBQ by $|fB/D| \le (1/2)B/(B+1) < 1/2$, so $\beta BX/D$ rounds to $\beta BG = \beta BQ$. Therefore $G = Q$ exactly again, finishing off the LOW Case.

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

$$B + B/(\beta B-2) \le BI/D \le \beta B-\beta-1 + (\beta+1)/(B+1) \;.$$

Then BI/D rounds to the nearest integer BQ between B and $\beta B-\beta$ inclusive. This BQ is a quotient whose remainder is

$$r = BQD - BI \;, \quad \text{and} \quad |r| \le D/2 \;.$$

If $Q = 1$ then both $X = QD$ and $G = X/D = Q$ exactly, so we need consider only the case $B+1 \le BQ = BI/D + r/D \le \beta B-\beta$. Then $B+2+1/B \le QD = I + r/B \le I + D/(2B) \le \beta B-1 + (\beta B-2)/2B < \beta B-1+\beta/2$; therefore QD rounds to the nearest integer except in the rare case that $\beta B + 1/2 < QD$, which we shall deal with later.

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. That is why $G = Q$ in this case.

In the rare case that $\beta B + 1/2 < QD < \beta B - 1 + \beta/2$, which cannot arise unless $\beta \ge 4$, the rounded value of QD is $X = \beta B$. Now
$G = X/D$ rounded $\ge$ I/D rounded $= Q$ on the one hand, and
$G = \beta B/D$ 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 phenomenon just proved has any worthwhile application is not known. At first sight it seems to supply a simple way to test whether computers round multiplication and division correctly in floating-point. The test program would generate a large number of pairs I and D , perhaps at random, and for each pair test whether the three operations

Q := I/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 are 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)

A *Floating Point Validation* (FPV) package of software that tests all the arithmetic operations (+, −, *, / and √) 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 from the U. S. Office of Naval Research under contract N00014-85-K-0180 .