Checking Whether Floating-Point Division is Correctly Rounded

W. Kahan
Elect. Eng. & Computer Sci. Dept.
University of California
Berkeley CA 94720
April 11, 1987

Introduction:

Despite the publication of hardware designs [1] that produce correctly rounded quotients quickly and reasonably economically, some hardware designers continue to use techniques that multiply the numerator by an approximate reciprocal in the hope that it may be cheaper and/or faster. Testing these techniques for accuracy is an interesting challenge because they can produce approximate quotients that are almost always correctly rounded, so aberrant cases may escape detection even though billions of divisions by randomly generated trial arguments may have been scrutinized. On occasion, that kind of testing has tempted someone to claim that his implementation of division rounded correctly when it didn't.

This paper provides practical alternatives to random testing; it shows how to generate trial arguments whose quotients are not so unlikely to be rounded incorrectly by approximate techniques for division. I deplore those techniques as much as I deplore sexual promiscuity among teenagers, but in both cases the greater sin is to deny sinners information that could prevent something worse.

Approximate Floating-Point Division:

The deplorable techniques for approximating a quotient x/y start by approximating the reciprocal $1/y$ by a value ρ obtained from a table-look-up followed by the iteration $\rho := \rho + (1 - y\rho)\rho$. Then a rough quotient $q' := x\rho$ is generated and its residual $r := x - q'y$ is computed very carefully from a double-width product $q'y$; finally $q := q' + r\rho$ is rounded off to produce the desired quotient. Almost. If ρ approximates $1/y$ closely enough, as can be assured by iterating often enough, then q will almost always be correctly rounded at the end. When will q *not* be correctly rounded? The difficult cases arise when x/y takes values like `dddddd.499999ddd` and `dddddd.500000ddd` where the d's are arbitrary decimal digits and the digits to the right of the decimal point are to be rounded off. In such cases, the value of $q' + r\rho$ before rounding can turn out to be respectively `dddddd.500000ddd` and `dddddd.499999ddd` (the latter is the more likely case), thereby rounding off to a quotient q whose last digit is respectively too big or too small. Here is an example, using decimal arithmetic rounded to six sig. dec.:

Let $x := 998586$;
let $y := 0.999307$, so $1/y = 1.000693480582....$
Iterating $\rho := \rho + (1 - y\rho)\rho \rightarrow \rho = 1.00069$. It looks good.
$q' := x\rho$ yields $q' = 999275$; $q'y = 998582.502425$ exactly.
$r := x - q'y = 3.497575$, which rounds to 3.49758.
$q := q' + r\rho = 999278.499993$, which rounds to 999278.
Iterating $q := q + (x - qy)\rho$ rounds again to 999278.
But $x/y = 999278.5000005003...$ should round to 999279.

Having to compute x/y to beyond twelve sig. dec. in order to round it correctly to six makes this example seem extraordinary, but others like it do exist. The trick is to find them.

How Difficult can Correct Rounding be?

Consider floating-point arithmetic carried out to N significant digits of radix β, where $\beta = 2$ for Binary, $\beta = 8$ for Octal, $\beta = 10$ for Decimal, $\beta = 16$ for Hexadecimal arithmetic. Since working with integers will be convenient, we multiply numerator and denominator of a quotient x/y by powers of β, without changing the significant digits, to put them into the range

$$\beta^{N-1} \le X \le \beta^N - 1, \text{ and } \beta^{N-1} \le Y \le \beta^N - 1.$$

To put the correctly rounded quotient Q into the same range, we must scale it by a power of β that depends upon X and Y; so

$$\begin{aligned} &\text{let } j := (Y \le X) = 1 \text{ if } Y \le X, \\ &\qquad\qquad\qquad\;\; = 0 \text{ if } X < Y; \text{ and} \\ &\text{let } Q := \beta^{N-j}X/Y \text{ correctly rounded.} \end{aligned}$$

Now $\beta^{N-1} \le Q \le \beta^N - 1$, and if $\beta = 2$ then $2^{N-1} + 1 - j \le Q \le 2^N - 1$.

The exact quotient is $\beta^{N-j}X/Y = Q \pm (1/2 - \varepsilon)$ for some fraction ε; $0 \le \varepsilon \le 1/2$. We wish to choose X and Y to make ε as tiny as possible.

How tiny can ε be?

To find out, rewrite the foregoing definition of ε thus:

$$\pm 2\varepsilon Y = (2Q \pm 1)Y - 2\beta^{N-j}X = \pm R \text{ respectively} \qquad (*)$$

for some nonnegative integer $R \le Y$. R is a kind of remainder. Now we shall prove a ...
Proposition:

If $\beta > 2$ then R can vanish, and so can ε.
If $\beta = 2$ then $R \ge 1$, so $\varepsilon \ge 1/(2Y)$.

Proof: It depends upon whether $\beta = 2$ or $\beta = 10$ or neither.

Making R and ε vanish is easy when $2 < \beta \ne 10$. Choose a digit η from the set $\{1, 2, ..., \beta/2 - 1\}$ and let $Y := 2\eta\beta^{N-1}$. (Do not choose a power of 2 for η if you wish ρ not to equal $1/Y$ exactly in the deplorable division algorithm above, lest it produce a correctly rounded quotient.) Then choose for X any odd multiple of η strictly between Y and β^N.

Making R and ε vanish when $\beta = 10$ entails a wider range of choices. First choose $j := 0$ or 1. Then choose any multiple of 2^{N+1-j} strictly between 10^{N-1} and 10^N for Y. Let 5^K be the largest power of 5 that divides Y, so $0 \le K < N$. Choose $(2Q \pm 1)$ to be any odd multiple of 5^{N-K-j} lying between limits that depend upon j as follows:

$$\begin{aligned} &\text{if } j = 0 \text{ then } 2 \times 10^{2N-1}/Y \le 2Q \pm 1 < 2 \times 10^N; \\ &\text{if } j = 1 \text{ then } 2 \times 10^{N-1} \qquad < 2Q \pm 1 < 2 \times 10^{2N-1}/Y. \end{aligned}$$

This last choice determines Q and $X = 10^{j-N}(2Q \pm 1)Y/2$, though the choice of Q depends upon whether you like `...ddd.5` to be rounded up or down. For example, when $\beta = 10$ and $N = 6$, choose $j := 0$ and $Y := 2998 \times 2^{N+1-j} = 383744$ arbitrarily; then choose $2Q \pm 1 := 79 \times 5^{N-K-j} = 1234375$ almost arbitrarily, whence $X = 236842$ and $Q \pm 1/2 = 10^N X/Y = 617187.5$. This should round to $Q = 617188$. What would deplorable division deliver? Suppose $1/Y = 2.60590\mathit{393}..._{10} - 6$ is approximated by $\rho = 2.60590_{10} - 6$, which comes as close as it can. Then $10^6 X\rho = 617186.5678$ rounds to $q' := 617187$, whence $r := 10^6 X - q'y = 191872$ exactly, and $q' + r\rho = 617187.499999$ rounds back to q' instead of Q.

When $\beta = 2$ we deduce that R and ε cannot vanish by examining the crucial equation (*) above, which now takes the form

$$\pm 2\varepsilon Y = (2Q \pm 1)Y - 2^{N+1-j}X = \pm R \text{ respectively .}$$

Evidently the integers R and Y have the same number, say K, of trailing zeros in their binary representations; i. e.,

$$R \equiv Y \equiv 2^K \bmod 2^{K+1} \text{ where } 0 \le K < N. \qquad (**)$$

That is why $R \geq 1$. So ends the proof of the *Proposition.*

The *Proposition* shows us that after a floating-point quotient has been computed to *more* than $2N$ sig. digits, it can surely be rounded correctly to N sig. digits. More precisely, if q approximates $\beta^{N-j}X/Y$ with an absolute error smaller than $1/(4Y)$ or with a relative error smaller than $1/(4\beta^{2N})$, then either q matches a half-integer $Q \pm 1/2$ so closely that the *Proposition* implies $\beta^{N-j}X/Y = Q \pm 1/2$ exactly (impossible if $\beta = 2$), or else q rounds to Q correctly. When $\beta = 2$, absolute error smaller than $1/(2Y)$ or relative error smaller than 2^{-2N-1} is good enough.

Finding Hard Cases:

Now we turn to the task of finding scaled floating-point quotients $\beta^{N-j}X/Y$ that differ from a half-integer $Q \pm 1/2$ by about as tiny a positive fraction $\varepsilon = R/(2Y)$ as possible. When $\beta = 2$ these encompass all the *hard cases*, all the trial quotients most likely to be rounded incorrectly by whatever division algorithm is being tested. One measure of the difficulty of rounding is the *absolute* tinyness of ε; another is its *relative* tinyness, i.e.

$$\varepsilon/(Q \pm 1/2) = R/(2\beta^{N-j}X \pm R).$$

The tinier these quantities, the harder is correct rounding. The latter quantity will be tiniest when R is a small integer like 1 or 2 or 3, when $j = 0$, and when X is close to its upper bound $\beta^N - 1$. But, given β and N and R, what reason is there to think j and X can be chosen arbitrarily? In fact they cannot. Instead, so few 4-tuples (j, X, Y, Q) satisfy the crucial equation

$$\pm 2\varepsilon Y = (2Q \pm 1)Y - 2\beta^{N-j}X = \pm R \text{ respectively} \qquad (*)$$

and the constraints $j(j-1) = 0$ and $\beta^{N-1} \leq X, Y, Q < \beta^N - 1$ that we can contemplate computing many 4-tuples systematically.

Finding Hard Cases by Factorization:
Fix β and N for the time being. For $R = 1, 2, 3, \ldots$ in turn, we wish to compute admissible 4-tuples (j, X, Y, Q), preferably with X as large as possible (but less than β^N), that satisfy the crucial equation (*). Rewriting it in the form

$$(2Q \pm 1)Y = 2\beta^{N-j}X \pm R \text{ respectively}$$

provides a motivation for considering factorization as a plausible technique. Having chosen R, for each admissible j and X we need merely factorize each of $2\beta^{N-j}X \pm R$ and determine which factors, if any, constitute admissible values of $2Q \pm 1$ and Y. Of course, factorizing an integer with about $2N$ digits could take a long time. A faster way would be preferable. A faster way does exist to compute 4-tuples whose X, Y and Q lie not too far from their lower and upper bounds β^{N-1} and $\beta^N - 1$. It requires factorizations of rather smaller integers of the form $m\beta^{N-1} \pm R$ with $m = 1, 2, 3, \ldots$; formulas to compute X, Y and Q from those factors are given below. In all six cases, the factors are f and g, and f is an odd factor subject sometimes to further limitations. In particular, if a formula does not produce X, Y and Q that lie between their lower and upper bounds β^{N-1} and $\beta^N - 1$, then it cannot be used with those factors f and g.

Suppose $(2M+1)\beta^N \pm R = fg$, and the odd factor $f \geq 2M+3$. Then
$X := \beta^N + M - g - (f-1)/2; \quad Y := \beta^N - g; \quad Y > X;$
$j := 0; \quad Q := \beta^N - (f \pm 1)/2.$ -(Case 1)

Suppose $(2M+1)\beta^{N-1} \pm R = fg$, and the odd factor is f. Then
$X := \beta^{N-1} + M + g + (f+1)/2; \quad Y := \beta^{N-1} + g; \quad Y < X;$
$j := 1; \quad Q := \beta^{N-1} + (f \mp 1)/2.$ -(Case 2)
If also the odd factor $f \geq 2\beta g - 2M - 1$, then
$X := \beta^N - M + \beta g - (f+1)/2; \quad Y := \beta^{N-1} + g;$
check that $Y < X$ and, if so, continue with
$j := 1; \quad Q := \beta^N - (f \mp 1)/2.$ -(Case 3)
If $\eta := (M + (f+1)/2)/\beta$ is an integer *leg*, then
$X := \beta^{N-1} + g - \eta; \quad Y := \beta^{N-1} + g; \quad Y > X;$
$j := 0; \quad Q := \beta^N - (f \mp 1)/2.$ -(Case 4)

Suppose $2M\beta^{N-1} \pm R = fg$, and the odd factor is f.
If $2M/\beta < f < 2(M+g)/\beta$ then
$X := \beta^N - M - g + f\beta/2; \quad Y := \beta^N - g;$
$j := 1; \quad Q := \beta^{N-1} + (f \pm 1)/2.$ -(Case 5)
If $2(M+g)/\beta \leq f$ and $\eta := (M+g)/\beta - 1/2$ is an integer, then
$X := \beta^{N-1} - \eta + (f-1)/2; \quad Y := \beta^N - g;$
check that $X < Y$ and, if so, continue with
$j := 0; \quad Q := \beta^{N-1} + (f \pm 1)/2.$ -(Case 6)

Each decomposition of a number like $(2M+1)\beta^N \pm R$ into a product of primes produces a number of pairs of factors (f, g) from each of which one or two 4-tuples (j, X, Y, Q) can often be derived, either from cases 1, 5 and 6, or else from cases 2, 3 and 4. Cases 1 and 5 are usually the hardest to round correctly because they usually have the biggest values of X and Y. For example consider six digit decimal arithmetic ($\beta = 10, N = 6$); starting with $R := 1$ and $M := 0$, we obtain prime decompositions of

$$\begin{array}{ll} (2M+1)\beta^{N-1} + R = 100001 = 11 \times 9091; & (2M+1)\beta^{N-1} - R = 99999 = 3^2 \times 41 \times 271; \\ (2M+1)\beta^{N} + R = 1000001 = 101 \times 9901, & (2M+1)\beta^{N} - R = 999999 = 3^3 \times 7 \times 11 \times 13 \times 37, \\ \quad = 2((2M+1)\beta/2)\beta^{N-1} - R. & \quad = 2((2M+1)\beta/2)\beta^{N-1} + R. \end{array}$$

Some of the 115 4-tuples (j, X, Y, Q) derived from these four prime decompositions are displayed in Table 1 below. All 115 of them were fed to a carefully implemented but deplorable division algorithm of the kind described on the first page of this paper. A # sign marks some of the 31 4-tuples that were rounded incorrectly. Were quotients x/y generated at random and fed to the same algorithm, at least about 100000 of them would have to be scrutinized before one incorrectly rounded instance could be expected to appear.

Similar examples for 24-bit binary arithmetic ($\beta = 2, N = 24$) have been derived from the prime decompositions

$$2^{23} - 1 = 47 \times 178481, \quad 2^{23} + 1 = 3 \times 2796203, \quad 2^{24} - 1 = 3^2 \times 5 \times 7 \times 13 \times 17 \times 241, \quad 2^{24} + 1 = 97 \times 257 \times 673.$$

Of 164 4-tuples derived that way, the 10 shown in Table 2 were rounded incorrectly by a deplorable division algorithm so nearly perfect that millions of random test quotients might have been scrutinized without finding one incorrectly rounded.

All computations of 4-tuples except possibly the factorizations can be performed efficiently in an almost obvious way using just (presumably correctly rounded) floating-point addition, subtraction and multiplication in the arithmetic whose division is under test. If the arithmetic's division is unreliable, the factorizations can become difficult, so Table 3 exhibits a few factorizations that I have found helpful. For instance, the factors of $3 \times 2^{27} - 1$ provided the two test cases

$$\begin{aligned} 2^{27} \times 125650617/125650639 &= 134217704.500000004 \text{ and} \\ 2^{27} \times 134088288/134089861 &= 134216153.500000004 \end{aligned}$$

that revealed a flaw in a nearly perfect 27-bit division scheme.

Finding hard cases for each divisor Y :

The foregoing formulas were motivated by a desire to make ε as small as possible *relative* to the quotient Q; this was to be accomplished by seeking 4-tuples (j, X, Y, Q) whose numerator X is as large as possible. If instead we try to make $\varepsilon = R/(2Y)$ as small as possible *absolutely*, our quest will lead to 4-tuples with Y as big as is compatible with a given $R = 1, 2, 3, \ldots$.

Given β and N and a small remainder R and a trial divisor Y between β^{N-1} and $\beta^N - 1$, the crucial equation

$$(2Q \pm 1)Y - 2\beta^{N-j}X = \pm R \text{ respectively} \qquad (*)$$

can be solved for j, Q and X provided Y and R are compatible. If we regard $2Q \pm 1$ as a kind of quotient $\pm R/Y \bmod 2\beta^{N-j}$, we can think of the necessary calculation as a long division carried out from right-to-left instead of left-to-right. To simplify the exposition considerably let us assume that $\beta = 2$ and that R and Y are odd, thereby satisfying the compatibility condition (**) mentioned at the end of the *Proposition*'s proof, namely that

Table 1 : 4-tuples (j, X, Y, Q) with $Q = 10^{6-j}X/Y$ rounded
$\beta = 10, N = 6, R = 1, M = 0$ or 5

$(f \times g = 100001)$

f	g	case	M	j	X	Y	Q	$10^{6-j}X/Y$
1	100001	2	0	1	200002	200001	100000	100000.4999975
11	9091	2	0	1	109097	109091	100005	100005.4999954
9091	11	2	0	1	104557	100011	104545	104545.4999950
		3	0	1	995564	100011	995455	995454.5000050
100001	1	2	0	1	150002	100001	150000	150000.4999950
		3	0	1	950009	100001	950000	949999.5000050

$(f \times g = 99999)$

f	g	case	M	j	X	Y	Q	$10^{6-j}X/Y$
1	99999	2	0	1	200000	199999	100001	100000.5000025
3	33333	2	0	1	133335	133333	100002	100001.5000038
...	...				...	...	...	...
2439	41	2	0	1	101261	100041	101220	101219.5000050
		3	0	1	999190	100041	998780	998780.4999950
...	...				...	...	...	...
99999	1	2	0	1	150001	100001	150000	149999.5000050
		3	0	1	950010	100001	950000	950000.4999950

$(f \times g = 1000001)$

f	g	case	M	j	X	Y	Q	$10^{6-j}X/Y$	
101	9901	1	0	0	990049	990099	999949	999949.4999995	
		5	5	1	990599	990099	100051	100050.5000005	
9901	101	1	0	0	994949	999899	995049	995049.4999995	#
1000001	1	1	0	0	499999	999999	499999	499999.4999995	

$(f \times g = 999999)$

f	g	case	M	j	X	Y	Q	$10^{6-j}X/Y$	
3	333333	1	0	0	666666	666667	999999	999998.5000008	
		5	5	1	666677	666667	100001	100001.4999993	
9	111111	1	0	0	888885	888889	999996	999995.5000006	
		5	5	1	888929	888889	100004	100004.4999994	
27	37037	1	0	0	962950	962963	999987	999986.5000005	
		5	5	1	963093	962963	100013	100013.4999995	
7	142857	1	0	0	857140	857143	999997	999996.5000006	#
		5	5	1	857173	857143	100003	100003.4999994	
...	...				...	...	...	...	
3003	333	1	0	0	998166	999667	998499	998498.5000005	#
9009	111	1	0	0	995385	999889	995496	995495.5000005	#
27027	37	1	0	0	986450	999963	986487	986486.5000005	#
...	...			...	...	...	...		
76923	13	1	0	0	961526	999987	961539	961538.5000005	#
481	2079	1	0	0	997681	997921	999760	999759.5000005	#
1443	693	1	0	0	998586	999307	999279	999278.5000005	#
4329	231	1	0	0	997605	999769	997836	997835.5000005	#
12987	77	1	0	0	993430	999923	993507	993506.5000005	#
3367	297	1	0	0	998020	999703	998317	998316.5000005	#
...	...				...	...	...	...	
111111	9	1	0	0	944436	999991	944445	944444.5000005	
333333	3	1	0	0	833331	999997	833334	833333.5000005	#
999999	1	1	0	0	500000	999999	500001	500000.5000005	

Table 2 : 4-tuples $(0, X, Y, Q)$ with $Q = 2^{24}X/Y$ rounded
$\beta = 2, N = 24, R = -1, j = M = 0$, case 1

$(f * g = 2^{24} - 1 = 16777215)$

f	g	X	Y	Q	$2^{24}X/Y$	
21	798915	15978291	15978301	16777206	16777205.50000003	#
273	61455	16715625	16715761	16777080	16777079.50000003	#
315	53261	16723798	16723955	16777059	16777058.50000003	#
85	197379	16579795	16579837	16777174	16777173.50000003	#
1105	15183	16761481	16762033	16776664	16776663.50000003	#
3315	5061	16770498	16772155	16775559	16775558.50000003	#
2169	7735	16768397	16769481	16776132	16776131.50000003	#
1205	13923	16762691	16763293	16776614	16776613.50000003	#
1687	9945	16766428	16767271	16776373	16776372.50000003	#
3133	5355	16770295	16771861	16775650	16775649.50000003	#

Table 3 : Some Factorizations

$$\begin{array}{rl}
10^5 - 1 = 3^2 \times 41 \times 271, & 10^5 + 1 = 11 \times 9091, \\
10^6 - 1 = 3^3 \times 7 \times 11 \times 13 \times 37, & 10^6 + 1 = 101 \times 9901. \\
10^9 - 1 = 3^4 \times 37 \times 333667, & 10^9 + 1 = 7 \times 11 \times 13 \times 19 \times 52579, \\
10^{10} - 1 = 3^2 \times 11 \times 41 \times 271 \times 9091, & 10^{10} + 1 = 101 \times 3541 \times 27961. \\
10^{11} - 1 = 3^2 \times 21649 \times 513239, & 10^{11} + 1 = 11^2 \times 23 \times 4093 \times 8779. \\
10^{12} - 1 = 3^3 \times 7 \times 11 \times 13 \times 37 \times 101 \times 9901, & 10^{12} + 1 = 73 \times 137 \times 99990001. \\
10^{13} - 1 = 3^2 \times 53 \times 79 \times 265371653, & 10^{13} + 1 = 11 \times 859 \times 1058313049.
\end{array}$$

$$\begin{array}{rl}
2^{23} - 1 = 47 \times 178481, & 2^{23} + 1 = 3 \times 2796203, \\
2^{24} - 1 = 3^2 \times 5 \times 7 \times 13 \times 17 \times 241, & 2^{24} + 1 = 97 \times 257 \times 673. \\
2^{26} - 1 = 3 \times 2731 \times 8191, & 2^{26} + 1 = 5 \times 53 \times 157 \times 1613, \\
2^{27} - 1 = 7 \times 73 \times 262657, & 2^{27} + 1 = 3^4 \times 19 \times 87211. \\
3 \times 2^{27} - 1 = 47 \times 67 \times 127867, & 3 \times 2^{27} - 1 = 5 \times 11 \times 1399 \times 5233. \\
2^{47} - 1 = 2351 \times 4513 \times 13264529, & 2^{47} + 1 = 3 \times 283 \times 165768537521, \\
2^{48} - 1 = 3^2 \times 5 \times 7 \times 13 \times 17 \times 97 \times 241 \times 257 \times 673, & 2^{48} + 1 = 193 \times 65537 \times 22253377. \\
2^{52} - 1 = 3 \times 5 \times 53 \times 157 \times 1613 \times 2731 \times 8191, & 2^{52} + 1 = 17 \times 858001 \times 308761441, \\
2^{53} - 1 = 6361 \times 69431 \times 20394401, & 2^{53} + 1 = 3 \times 107 \times 28059810762433. \\
2^{55} - 1 = 23 \times 31 \times 89 \times 881 \times 3191 \times 201961, & 2^{55} + 1 = 3 \times 11^2 \times 683 \times 2971 \times 48912491, \\
2^{56} - 1 = 3 \times 5 \times 17 \times 29 \times 43 \times 113 \times 127 \times 15790321, & 2^{56} + 1 = 257 \times 5153 \times 54410972897. \\
2^{62} - 1 = 3 \times 715827883 \times 2147483647, & 2^{62} + 1 = 5 \times 5581 \times 8681 \times 49477 \times 384773, \\
2^{63} - 1 = 72 \times 73 \times 127 \times 337 \times 92737 \times 649657, & 2^{63} + 1 = 3^3 \times 19 \times 43 \times 5419 \times 77158673929, \\
2^{64} - 1 = 3 \times 5 \times 17 \times 257 \times 641 \times 65537 \times 6700417, & 2^{64} + 1 = 274177 \times 67280421310721.
\end{array}$$

R and Y have the same number (none) of trailing zeros. We shall also need the *Parity* function

$$\begin{aligned}\pi(n) &:= 1 \quad \text{if } n \text{ is odd,}\\ &:= 0 \quad \text{if } n \text{ is even.}\end{aligned}$$

Starting with $Q_1 := 0$ and $X_1 := (Y - R)/2$, run the recurrence

$$Q_{k+1} := Q_k + 2^{k-1}\pi(X_k); \qquad X_{k+1} := (X_k + \pi(X_k)Y)/2;$$

for $k = 1, 2, ..., N - 1$ in turn to obtain the small solutions Q_N and X_N of $(2Q_N + 1)Y = 2^N X + R$ with $0 < Q_N < 2^{N-1}$ and $0 < X_N < Y$. The crucial equation (*) has one or two solutions:

First try $Q := 2^{N-1} + Q_N$ and $X := Y + X_N$;
if $X < 2^N$ then $j = 1$ and $(2Q + 1)Y = 2^N X + R$; ...(Case A)
else if $\pi(X) = 0$ replace X by $X/2$,
and then $j = 0$ and $(2Q + 1)Y = 2^{N+1}X + R$. ...(Case B)

Next try $Q := 2^N - Q_N$ and $X := 2Y - X_N$;
if $X < 2^N$ then $j = 1$ and $(2Q - 1)Y = 2^N X - R$; ...(Case C)
else if $\pi(X) = 0$ replace X by $X/2$,
and then $j = 0$ and $(2Q - 1)Y = 2^{N+1}X - R$. ...(Case D)

The foregoing recurrences produce 4-tuples (j, X, Y, Q) for each of which Q is the correctly rounded value of the quotient $2^{N-j}X/Y$. Note that these recurrences, including the Parity function π , can be realized using only floating-point addition, subtraction and multiplication operations correctly rounded to N sig. bits, and all the multiplications are exact multiplications by 0.5 or other powers of 2.

Thus for each odd trial divisor Y strictly between 2^{N-1} and 2^N, and for each odd trial remainder R smaller than Y, we obtain one or two 4-tuples (j, X, Y, Q) upon which to test floating- point division. (When R and Y are even with the same number of trailing zeros, the process gets more complicated because more 4-tuples can turn up. The details must be deferred to another occasion.) To test division upon a 4-tuple whose quotient is as hard as possible to round correctly, we have to strike a balance between opposing forces. On the one hand, we should choose R as small as possible, say $R := 1$ or perhaps 3, and Y as big as possible, say $2^N - 1, 2^N - 3, 2^N - 5, ...,$ to keep $\varepsilon = R/(2Y)$ as small as possible. On the other hand, we should try to scatter values Y in an interval, if we can find it, where the division algorithm under test is likely to commit its worst errors.

Whether the deplorable division algorithm discussed on the first page of this paper commits its worst errors when $2^N - Y$ is a small odd integer depends upon details of the implementation. The iteration $\rho := \rho + (1 - \rho Y)\rho$ that produces an approximate reciprocal ρ converges to $1/Y$ from below, so if it is stopped too soon ρ will tend to underestimate $1/Y$. That underestimation's extent will exceed half a unit in the last place of $1/Y$ whenever Y is an odd integer between 2^N and $2^N - 2^{N/2}$ because then $2^{2N-1}/Y$ must have a fractional part bigger than 1/2; underestimation in ρ will be compounded later to produce a final error that our 4- tuples are very likely to expose. On the other hand if iteration is not stopped too soon but instead ρ is computed so carefully that $\rho = (1/Y$ correctly rounded) almost always, then ρ will overestimate $1/Y$ in such a way as will partly compensate for underestimation occurring later in the algorithm; then its final quotient will most likely be correctly rounded too. Therefore, if that deplorable algorithm has been implemented with so much care as to make its flaws hard to find, Y should be chosen rather smaller than $2^N - 2^{N/2}$ to enhance the likelihood that the quotient $2^{N-j}X/Y$ will be rounded incorrectly.

For example, take $N = 24$ and reconsider the nearly perfect but deplorable division algorithm whose flaws were exposed by the 4- tuples derived from factorization and displayed in Table 2. Let $R := 1$ and for Y choose consecutive odd integers starting with $2^N - 1 = 16777215$ and running down through $16777213, 16777211, \ldots$ until a 4-tuple exposes a flaw. The first flaw found was at

$$2^N \times 12237320/16772199 = 12240980.50000003,$$

which the deplorable algorithm rounded down instead of up to $Q = 12240981$. Note that $Y = 16772199 < 2^N - 2^{N/2} = 16773120$, as the previous paragraph leads us to expect. All of the 4-tuples generated from 2508 odd trial divisors between 2^N and 16772198 were rounded correctly. But the next 12500 4-tuples generated from consecutive odd trial divisors $Y < 16772198$ produced over 630 instances (all Case D) incorrectly rounded; this hit rate (over 1/20) far exceeds the yield from purely random tests.

Can Iterative Division be Rounded Correctly?

In software anything is possible, for a price. Two schemes come to mind that obtain correctly rounded quotients x/y from an algorithm like the deplorable one. The first scheme is based upon the *Proposition* proved above; the second is based upon an exact remainder which must be compared with the divisor. Both schemes are expensive. Each will be sketched only very briefly because I believe neither is so cost-effective as a simple hardware divider.

Both schemes require special prescaling to protect division from spurious over/underflows. Therefore, we might as well assume that the quotient q will be an N-digit integer between β^{N-1} and β^N; when correctly rounded, $| q - x/y | \le 1/2$.

The first division scheme carries extra precision for intermediate quantities ρ, q' and r. The iteration $\rho := \rho + (1 - \rho y)\rho$ must be performed carrying ρ to two or three extra digits of precision, and when ρ is in error only in the last of those extra digits then $q' := x\rho$ must be computed to two or three extra digits too. The residual $r := x - q'y$ will require a product with $2N + 2$ or $2N + 3$ digits, but after over half of them cancel only N sig. digits of r need be retained. Finally, $q := q' + r\rho$ will be correct to more than $2N$ digits before the sum is rounded, so the *Proposition* implies that it will be rounded correctly except possibly if x/y differs from q by exactly 1/2. That can happen only if the radix β exceeds 2, and can be detected by observing when $q' + r\rho$ differs from q by too nearly 1/2.

The second scheme resembles what H. Kuki and J. Ascoly did to implement extended-precision (28 hex. digits) division correctly chopped (not rounded) on the IBM 360/85 [2]. After using the deplorable algorithm to compute the integer $q \doteq x/y$ so well that $| q - x/y | < 1$, though q is not necessarily correctly rounded, we must compute the remainder $r := x - qy$

exactly; it requires a double-width product of which at least half the digits will cancel off. Ordinary floating-point subtraction suffices to determine whether $|y| - |r| > |r|$, in which case q is correctly rounded; otherwise the signs of r and y tell whether to increment q or decrement it by 1. When $| q - x/y = 1/2 |$ exactly, something special may have to be done, but that won't happen if $\beta = 2$.

Conclusions:

There is now no excuse for testing division algorithms merely upon quotients x/y with x and y generated independently at random. Such a test is weaker than a test with x and y so correlated as must much enhance the likelihood that the quotient will be rounded incorrectly, if that can happen at all. The procedures herein and programs below achieve that correlation at a tolerable cost.

Does correctly rounded division really matter? The answer to that question would require a longer paper than this one. This paper merely shifts the onus for an answer from those who say "It does matter" to those who say "It doesn't." An ontological principle "*If you can't tell the difference, it can't matter to you.*" is often invoked to excuse slightly incorrectly rounded arithmetic although the validity of that principle would be challenged by a mechanic who has purchased incorrectly marked bolts weaker than what he needs. Regardless of its validity, that principle has now been rendered irrelevant by programs that will very likely expose incorrectly rounded division.

Acknowledgments:

I am indebted to Drs. Brent Baxter and Cleve Moler of INTEL, Oregon, who instigated the work that led to this paper.

References:

[1] George S. Taylor "Radix 16 SRT Dividers with Overlapped Quotient Selection Stages – a 225 Nanosecond Divider for the S-1 Mark IIB" pp. 64-71 of the Proc. 7th IEEE Symposium on Computer Arithmetic, June 4-6, 1985, Univ. of Illinois; IEEE Cat. No. 85CH2146-9, Order no. 632.

[2] Hirondo Kuki and Joseph Ascoly "FORTRAN Extended Precision Library" (1971) *IBM Systems Journal* **10** pp. 39-61 .

Appendix:

Three programs are appended here, all written in BASIC for an IBM PC , to serve as models from which parts may be copied by whoever wishes to test a possibly deplorable division algorithm.

The first program, FACTOR, factorizes any integer that can be represented exactly in the computer as a floating-point number. On the IBM PC, carrying 56 sig. bits in BASIC, consecutive integers up to $2^{56} - 1$ can be factored; larger integers are handled correctly because they are all divisible by some power of 2. On a machine whose radix exceeds 2 the program should be revised to first remove powers of the radix as factors.

The second program, CASES, runs through the formulas in Cases 1 to 6 to deliver 4-tuples (j, X, Y, Q) from an appropriate factorization, and to test whether a possibly deplorable division algorithm handles them correctly.

Ideally, the first two programs should be implemented in fixed-point with integers wider than the significands of the floating-point numbers X, Y, Q that will figure in the tests of the division algorithm. Otherwise several tricks will be required in a few places in those programs to simulate wider precision than is required merely to hold X, Y and Q. But the third program, DVSRS, can be executed entirely in the *binary* floating-point arithmetic whose division is under test after it is substituted for the simulated deplorable division in the program. This program scans any specified sequences of consecutive odd divisors Y and remainders R to generate 4-tuples (j, X, Y, Q) in accordance with cases A to D. Instances of incorrectly rounded division can be recorded in a file named "DEPLORE" at the user's option.

> *Users of these programs become participants in the author's researches, and are asked to report their experiences to him.*

FACTOR

```
10 ' FACTOR :  Integer factorization program for the  IBM-PC ,  in  BASIC .
20 DEFDBL D, C, Q, X :  DEFINT I, K :  DIM C(11) '  Initialization ...
30 FOR I=1 TO 11 :  READ C(I) :  NEXT I :'  This is a table of differences
40 DATA 1, 2, 2, 4, 2, 4, 2, 4, 6, 2, 6 :'  between nonmultiples of  2, 3, 5 .
50 DEF FNT(T$) = (60*VAL(LEFT$(T$,2))+VAL(MID$(T$,4,2)))*60!+VAL(RIGHT$(T$,2))
60 '
70 INPUT "Enter number to be factored  ( or  0  to quit ) : ",X0
80    IF X0=0 THEN PRINT "Goodbye." :  END
90    X=FIX(ABS(X0)) :  T = 0
100   IF X><X0 THEN PRINT "It must be a positive integer." :  GOTO 70
110 '
120 K=0 :  D=2 :  T$=TIME$ :  FOR I=1 TO 3 '  Initialize search for divisors.
130 Q=FIX(X/D) :  IF X=Q*D THEN GOSUB 230 :  GOTO 130 '  Loop to remove powers
140     D = D+C(I) :  NEXT I '                          of  2 ,  3 ,  5 .
150 GOSUB 180 ' ...  to remove powers of primes > 5 .
160 PRINT :  PRINT "Have you any more numbers to factor?" :  GOTO 70
170 '
180 FOR I = 4 TO 11 '  This is the main loop.  It increments a trial divisor  D
190   Q=X/D :  IF FIX(Q)=Q THEN GOSUB 230 :  GOTO 190 '     while  D < sqrt(X)
200   IF Q<D THEN D = X :  GOTO 230 '                        through successive
210   D = D+C(I) :  NEXT I :  GOTO 180 '         nonmultiples of  2, 3 and 5.
220 '
230 T = FNT(TIME$)-FNT(T$) :  K = K+1 '  This subroutine displays results.
240 IF X>1 OR K=1 THEN PRINT "Factor no.";K;" is ";D;".   ","Time =";T;"sec."
250 X = Q :  RETURN '  This subroutine adds negligibly to  TIME$ .
```

CASES:

```
10 ' IBM PC BASIC program to generate difficult quotients from factorizations
20 CLEAR ,,1000 :  DEFDBL A-H,O-Z :  DEFINT I-N
30 KP = 9 :  DIM K(9), P(9), Q(9,9) '  ... at most  KP  distinct prime factors
40 B = 0 :  PRINT "Choose a radix  b = 2, 8, 10 or 16 :  b =";: INPUT B
50 IF B=0 THEN PRINT "Goodbye": END
60 IF (B-2)*(B-8)*(B-10)*(B-16) >< 0 THEN 40 ELSE 80
70 PRINT "Choose integer  N > 3  but  b^N <= 2^56 = 72057594037927936 ."
80 PRINT "Choose the number  N  of sig. digits of radix  b =";B;":    N =";
90 INPUT Z :  N = Z :  IF Z=0 THEN 40 ELSE IF N><Z OR N<4 OR B^N>2^56 THEN 70
100 BN1 = B^(N-1) :  BN = B*BN1 ' ... = B^N  for  N digits of Radix B
110 PRINT "b =";B;",  N =";N;",  b^(N-1) =";BN1;"< X, Y, Q < b^N =";BN
120 BLOG = LOG(B) :  EPS = 3E-24 :  H = .5
130 DEF FNB(Z) = B^(INT(LOG(ABS(Z)+EPS)/BLOG) - N + 1)
140 DEF FNR(Z) = FNB(Z)*FIX(H*SGN(Z) + Z/FNB(Z)) '.. = Z rounded to N sig. dig.
150 PRINT "Choose a small integer  M >= 0  ( or < 0 to quit):  M =";: INPUT Z
160 M = Z :  IF Z<0 THEN 80 ELSE IF M><Z THEN 150
170 PRINT "Choose a small nonzero integer  R :";: INPUT R
180 IF R=0 THEN 150 ELSE IF R><INT(R) THEN 170
190 PRINT "Choose  L = 0 or 1 :   L =";: INPUT Z
200 LLL = Z :  IF Z*(Z-1)><0 THEN 190
210 IF LLL=1 THEN FG = (M+M+1)*BN1 + R ELSE FG = (M+M+1)*BN + R
220 PRINT "M =";M;",  R =";R;",  L =";LLL;",   (2M+1)*b^(N-L) + R = f*g =";FG
230 ' Set up table of factors of  (2M+1)*B^(N-LLL) + R  (or  2M*B^(N-1) + R ) :
240 ' Say  (2M+1)*B^(N-LLL) = (P1^K1)*(P2^K2)*(...)*(Pkp^Kkp) ;  then set
250 '          Q(i,j) = Pj^i  for  j = 1 to kp  and  i = 0 to Kj .
260 PRINT "Presumably you are ready to provide the prime factorization of"
270 PRINT "  ";FG;" = (P(1)^K(1)) * (P(2)^K(2)) *(...)* (P(";
280 PRINT KP;")^K(";KP;")) :"
290 FOR J=1 TO KP :  P(J) = 1 :  K(J) = 0 :  NEXT J
300 PRINT "( Enter a  P(j) = 0  or  K(j) = 0  to quit,  < 0  to go back. )"
310 J = 1 :  WHILE  J <= KP
320   IF J<1 THEN J=1
330   PRINT "1 < prime P(";J;") = ";: INPUT Z :  IF Z><INT(Z) OR Z=1 THEN 330
340   IF Z=0 THEN 380 ELSE IF Z<0 THEN J = J-1 : GOTO 320 ELSE P(J) = Z
350   PRINT "0 < index K(";J;") = ";: INPUT Z :  KJ = Z :  IF  KJ><Z  THEN 350
360   IF KJ=0 THEN P(J) = 1 : GOTO 380 ELSE IF KJ<0 THEN 330 ELSE K(J) = KJ
370   J = J+1 :  WEND
380 NO = 0 :  ND = 0  :  FOR J=1 TO KP :  Q(0,J)=1 :  NEXT J
390 FOR J=1 TO KP : FOR I=1 TO K(J) :  Q(I,J) = P(J)*Q(I-1,J) : NEXT I : NEXT J
400 PRINT FG;" is allegedly the product of the following prime powers:" : G=1
410 FOR J=1 TO KP :  IF P(J)=1 OR K(J)=0 THEN 440
420   G = G*Q(K(J),J)
430   PRINT J;":   ";P(J);: IF K(J)=1 THEN PRINT " " ELSE PRINT "^";K(J)
440   NEXT J
450 IF FG=G THEN PRINT "  And they do match." :  GOTO 520
460   PRINT "  But their product is ";G;" instead."
470   PRINT "Shall we Continue anyway  [C] ,  or Re-enter prime powers  [R] ,"
480   PRINT " or change  M etc.  [M] ,  or just  Quit  [Q] ";:  INPUT Q$
490   IF Q$="C" OR Q$="c" THEN 520
500   IF Q$="R" OR Q$="r" THEN 260
510   IF Q$="M" OR Q$="m" THEN 150 ELSE PRINT "Sorry about that.": STOP
520 FOR I9=0 TO K(9) : FOR I8=0 TO K(8) : FOR I7=0 TO K(7) : FOR I6=0 TO K(6)
530 FOR I5=0 TO K(5) : FOR I4=0 TO K(4) : FOR I3=0 TO K(3) : FOR I2=0 TO K(2)
540 FOR I1=0 TO K(1) :  Z = (((I9*10+I8)*10+I7)*10+I6)*10+I5
550 Z = (((Z*10+I4)*10+I3)*10+I2)*10+I1
560 F = Q(I1,1)*Q(I2,2)*Q(I3,3)*Q(I4,4)*Q(I5,5)*Q(I6,6)*Q(I7,7)*Q(I8,8)*Q(I9,9)
570 G = Q(K(1)-I1,1)*Q(K(2)-I2,2)*Q(K(3)-I3,3)*Q(K(4)-I4,4)*Q(K(5)-I5,5)
```

```
580 G = Q(K(6)-I6,6)*Q(K(7)-I7,7)*Q(K(8)-I8,8)*Q(K(9)-I9,9)*G
590 MO = M
600 PRINT "Combination";Z;":  f =";F;",  g =";G;",  f*g =";F*G
610 IF LLL=1 THEN FOR L=2 TO 4: GOSUB 730: NEXT L: GOTO 680' ... cases 2 to 4
620 ' ... but when  LLL = 0 ,  do cases 1, 5 and 6 : ...
630     L = 1 :  GOSUB 730 :' ... case 1
640     M = (M+M+1)*(B/2)
650     L = 5 :  GOSUB 730 :' ... case 5
660     L = 6 :  GOSUB 730 :' ... case 6
670     M = MO
680 NEXT I1 :  NEXT I2
690 NEXT I3 : NEXT I4 : NEXT I5 : NEXT I6 : NEXT I7 : NEXT I8 : NEXT I9
700 PRINT NO;" examples.  ";ND;" Deplorable Divisions.   The End." :  END
710 '
720 ' Subroutine to compute  X and Y ,  and compare quotients:
730 ON L GOSUB 900, 940, 980, 1020, 1060, 1100 ' ... case L ,  0 < L < 7
740 IF X<BN1 OR X>BN OR Y<BN1 OR Y>BN OR Q<BN1 OR Q>BN THEN RETURN
750 IF  X><INT(X)  THEN RETURN
760 BNJ = BN1*(B-J*(B-1)) :  QQ = BNJ*X/Y
770 ' Possibly deplorable division algorithm:
780 RHO = FNR(1/Y) :  RHO = FNR(RHO + FNR(RHO*FNR(1-FNR(RHO*Y))))
790 Q1 = FNR(RHO*X) :  Q2 = FNR(Q1 + FNR(RHO*FNR(BNJ*X - Y*Q1)))
800 Q3 = FNR(Q2 + FNR(RHO*FNR(BNJ*X - Y*Q2))) ' ... approximates  BNJ*X/Y
810 IF Q3 = Q THEN S$=" " ELSE S$ = "    #" :  ND = ND+1
820 '
830 PRINT "Case";L;":  R =";R;",  M =";M;",  j =";J;"            Q =";Q;S$
840 PRINT "   X =";X;",  Y =";Y;",  (";B;"^";N-J;")*X/Y =";QQ
850 NO = NO+1 :  RETURN
860 '
870 IF RR><R THEN PRINT "Disparity: case";L;"  computes ";RR;" for   R =";R
880 RETURN
890 '
900 RR = F*G - (M+M+1)*BN :  GOSUB 870 '   ... case 1
910 Y = BN - G :  X = Y + M - (F-1)/2 :  J = 0 :  Q = BN - (F+SGN(R))/2
920 RETURN
930 '
940 RR = F*G - (M+M+1)*BN1 :  GOSUB 870 '   ... case 2
950 Y = BN1 + G :  X = Y + M + (F+1)/2 :  J = 1 :  Q = BN1 + (F-SGN(R))/2
960 RETURN
970 '
980 RR = F*G - (M+M+1)*BN1 :  GOSUB 870 '   ... case 3
990 Y = BN1 + G :  X = BN-M+B*G-(F+1)/2 :  J = 1 :  Q = BN - (F-SGN(R))/2
1000 RETURN
1010 '
1020 RR = F*G - (M+M+1)*BN1 :  GOSUB 870 '   ... case 4
1030 Y = BN1 + G :  X = Y - (M+(F+1)/2)/B :  J = 0 :  Q = BN - (F-SGN(R))/2
1040 RETURN
1050 '
1060 RR = F*G - (M+M)*BN1 :  GOSUB 870 '   ... case 5
1070 Y = BN - G :  X = Y - M + F*B/2 :  J = 1 :  Q = BN1 + (F+SGN(R))/2
1080 RETURN
1090 '
1100 RR = F*G - (M+M)*BN1 :  GOSUB 870 '   ... case 6
1110 Y = BN - G :  X = BN1+F/2-(M+G)/B :  J = 0 :  Q = BN1 + (F+SGN(R))/2
1120 RETURN
```

DVSRS:

```
10 ' Test Accuracy of Division  X/Y  on Trial Divisors  Y .   Runs on  IBM PC .
20 CLEAR ,,1000 :  DEFDBL A-H, O-Z :  H = .5 :  TLOG = LOG(2#) :  EPS = 3E-24
30 DEF FNG(Z) = 2^(INT(LOG(ABS(Z)+EPS)/TLOG) - N + 1)
40 DEF FNR(Z) = FNG(Z)*FIX(H*SGN(Z) + Z/FNG(Z)) '... = Z rounded to N sig. bits
50 DEF FNPAR(Z) = Z - 2*INT(Z*H) : ' ... = Parity(z)
60 ON KEY(9) GOSUB 680 : PRINT "Press  [F9]  to stop."
70 KEY(9) STOP ' ... suspends stopping until there is something to report.
80 INPUT "How many sig. bits";N0 : N = INT(ABS(N0))
90 IF N=0 THEN PRINT "Goodbye." :  END
100 IF N><N0 OR N>56 THEN PRINT "... positive integer  < 57 ." :  GOTO 80
110 NC = 0 :  ND = 0 :  N1 = N-1 :  TN1 = 2^N1 :  TN = TN1+TN1 ' ... = 2^N
120 PRINT "Should  Deplorable  cases be written to  DEPLORE  ([Y]es or [N]o)";
130 INPUT Y$
140 IF NOT(Y$="Y" OR Y$="y") THEN 170
150   OPEN "DEPLORE" FOR APPEND AS #1
160   PRINT#1, "N =";N;" sig. bits."
170 PRINT "Choose a range of small odd remainders:"
180 A$ = "an" :  GOSUB 640 :  IF Z=0 THEN 80 ELSE R1 = Z
190 A$ = "another" :  GOSUB 640 :  IF Z=0 THEN 80 ELSE R2 = Z
200 IF R1>R2 THEN SWAP R1,R2 ' ... R1 <= R2
210 PRINT "Choose a range of odd divisors to test with each odd remainder:"
220 YL = TN1 :  IF R2 > TN1 THEN YL = R2+1 ' ... R1 <= R2 < YL < every Y
230 A$ = "a" :  GOSUB 590 :  IF Z=0 THEN PRINT : GOTO 170 ELSE  Y1 = Z
240 A$ = "another" : GOSUB 590 : IF Z=0 THEN PRINT : GOTO 170 ELSE Y2 = Z
250 IF Y1>Y2 THEN DY = -2 ELSE DY = 2
260 R = R1 : WHILE R<=R2 : Y = Y1 : WHILE (Y2-Y)*DY>=0 : GOSUB 330
270   Y = Y+DY : WEND : R = R+2 : WEND
280 PRINT NC;"cases,  ";ND;"deplorable divisions." : PRINT
290 IF Y$="Y" OR Y$="y" THEN PRINT#1, NC;"cases,   ";ND;"deplorable.": PRINT#1,
300 GOTO 170
310 '
320 ' Subroutine to compute  (j, X, Y, Q)  from  Y and R .
330 T = H :  Q0 = 0 :  X0 = Y-R :  FOR I = 1 TO N ' ... run recurrence
340   P = FNPAR(X0) :  Q0 = T*P + Q0 :  X0 = (P*Y+X0)*H :  T = T+T :  NEXT I
350 IF T><TN1 THEN PRINT "DISASTER: T =";T;" >< 2^(N-1) =";TN1: STOP
360 C$ = "A" :  X = X0+Y :  Q = Q0+T :  S = 1 :  J = 1
370 IF X-T<T THEN GOSUB 440 ELSE C$="B": J=0: X=X*H: IF X=INT(X) THEN GOSUB 440
380 C$ = "C" :  X = Y-X0+Y :  Q = T-Q0+T :  S = -1 :  J = 1
390 IF X-T<T THEN GOSUB 440 ELSE C$="D": J=0: X=X*H: IF X=INT(X) THEN GOSUB 440
400 KEY(9) ON ' ...  GOSUBs to  680  if  [F9]  has been pressed.
410 KEY(9) STOP :  RETURN ' ... or stop after  [F9] .
420 '
430 ' Subroutine to compute quotients, compare them, and display results.
440 NC = NC+1 :  TNJ = T + (1-J)*T ' ... = 2^(N-j)
450 QQ = INT(H + TNJ*X/Y) ' ... = (X*2^(N+1-j))/Y rounded if  N < 28
460 IF QQ><Q AND N<28 THEN D$ = "BUG!" :  PRINT "QQ=";QQ: GOSUB 520 : STOP
470 ' Now for a deplorable division algorithm rounded to  N sig. bits:
480   RHO = FNR(1/Y) :  RHO = FNR(RHO + FNR(RHO*FNR(1-FNR(RHO*Y))))
490   Q1 = FNR(RHO*X) :  Q2 = FNR(Q1 + FNR(RHO*FNR(TNJ*X - Y*Q1)))
500   Q3 = FNR(Q2 + FNR(RHO*FNR(TNJ*X - Y*Q2))) ' ... a deplorable quotient?
510 IF Q3 = Q THEN D$ = " " ELSE D$ = " #" :  ND = ND+1
520 PRINT "Y =";Y;",  R =";S*R;",  Case ";C$;":  j =";J;
530 PRINT ",  X =";X;",  Q =";Q;D$
540 IF D$=" " OR NOT(Y$="Y" OR Y$="y") THEN RETURN
550   PRINT#1, "Y =";Y;",  R =";S*R;",  Case ";C$;":  j =";J;
560   PRINT#1, ",  X =";X;",  Q =";Q;D$ :  RETURN
570 '
```

```
580 ' Subroutine to input a divisor and check that it lies in range.
590 PRINT "Choose ";A$;" divisor between ";YL;"and ";TN;":";: INPUT Y0
600 Z = ABS(INT(Y0)) :  IF Z=0 THEN RETURN
610 IF Z><Y0 OR Z<YL OR Z-TN1>TN1 OR FNPAR(Z)=0 THEN 590 ELSE RETURN
620 '
630 ' Subroutine to input a remainder and check that it lies within range.
640 PRINT "Choose ";A$;" odd remainder smaller than";TN-2;":";: INPUT R0
650 Z = ABS(INT(R0)) :  IF Z=0 THEN RETURN
660 IF Z><ABS(R0) OR Z-TN1>TN1-3 OR FNPAR(Z)=0 THEN 640 ELSE RETURN
670 '
680 PRINT NC;"cases,  ";ND;"deplorable."  '... Subroutine for  [F9] .
690 IF Y$="Y" OR Y$="y" THEN PRINT#1, NC;"cases,  ";ND;"deplorable." : CLOSE
700 STOP
```