Accelerating Floating-Point Division When the Divisor is Known in Advance

Jean-Michel Muller

Laboratoire de l'Informatique du Parallélisme

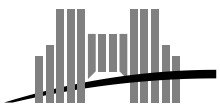
École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 5668

Accelerating Floating-Point Division When the Divisor is Known in Advance

Jean-Michel Muller

Août 2002

Research Report N° 2002-30



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr

Accelerating Floating-Point Division When the Divisor is Known in Advance

Jean-Michel Muller

Août 2002

Abstract

We present techniques for accelerating the floating-point computation of x/y when y is known before x. The goal is to get exactly the same result as with usual division with rounding to nearest. These techniques can be used by compilers to make some numerical programs run faster, without any loss in terms of accuracy.

Keywords: Computer arithmetic, Floating-point arithmetic, Division, Compilation optimization.

Résumé

Nous présentons des mthodes permettant d'accélérer les divisions virgule flottante de la forme x/y lorsque y est connu avant x. Le but est d'obtenir exactement le même résultat que par la division usuelle avec arrondi au plus près. Ces méthodes peuvent être utilisées à la compilation pour accélérer l'exécution de programmes numériques sans nuire à la précision des calculs.

Mots-clés: Arithmétique des ordinateurs, division virgule flottante, optimisation la compilation.

1 Introduction

In this paper, we deal with floating-point divisions of the form x/y for which y is known before x, either at compile-time (i.e., y is a constant. In such a case, much pre-computation can be performed), or at run-time. Our goal is to get the result more quickly than by just performing a division, yet with the same accuracy: we wish to get a correctly rounded value, as required by the IEEE 754 Standard for floating-point arithmetic [1, 5]. In this paper, we focus on rounding to nearest only.

Divisions by constants are a clear application of what we are planning to do. There are other applications, for instance when several divisions by the same y are performed. Consider for instance Gaussian elimination:

```
for j=1 to n-1 do
  if a[j,j] = 0 then stop
    else
      for i = j+1 to n do
         c[i,j] = a[i,j] / a[j,j]
            for k = j+1 to n do a[i,k] = a[i,k] - c[i,j]*a[j,k]
            end for
         b[i] = b[i] - l[i,j]*b[j]
      end for
end for
```

Most programmers will replace the divisions `a[i,j] / a[j,j]` by multiplications by `p = 1 / a[j,j]` (computed in the `for j...` loop). The major drawback is a loss of accuracy. A second possible drawback is that in some rare cases, the replacement may lead to "artificial" over / underflows (that is, the final result of a calculation would have been representable, whereas a partial subcalculation over / underflows). Our goal is to get the same result as if actual divisions were performed, without the delay penalty they would involve. Presentation of conventional division methods can be found in [4, 9, 13].

To make this paper easier to read, we have put the proofs in appendix.

2 Definitions and notations

Define $\mathbb{F}_n$ as the set of exponent-unbounded, n-bit mantissa, binary floating-point numbers (with $n \geq 1$), that is:

$$\mathbb{F}_n = \left\{ M \times 2^E, 2^{n-1} \leq M \leq 2^n - 1, M, E \in \mathbb{Z} \right\} \cup \{0\}$$

The **mantissa** of a nonzero element $M \times 2^E$ of $\mathbb{F}_n$ is the number $m(x) = M/2^{n-1}$. $\mathbb{F}_n$ is not the set of the available floating-point numbers on an existing system. It is an "ideal" system, with no overflows or underflows. We will show results in $\mathbb{F}_n$. These results will remain true in actual systems that implement the IEEE 754 standard, provided that no overflows or underflows do occur.

The result of an arithmetic operation whose input values belong to $\mathbb{F}_n$ may not belong to $\mathbb{F}_n$ (in general it does not). Hence that result must be *rounded*. The standard defines 4 different rounding modes:

- rounding towards $+\infty$, or upwards: $\circ_u(x)$ is the smallest element of $\mathbb{F}_n$ that is greater than or equal to x;
- rounding towards $-\infty$, or downwards: $\circ_d(x)$ is the largest element of $\mathbb{F}_n$ that is less than or equal to x;
- rounding towards 0: $\circ_z(x)$ is equal to $\circ_u(x)$ if $x < 0$, and to $\circ_d(x)$ otherwise;
- rounding to the nearest even: $\circ_n(x)$ is the element of $\mathbb{F}_n$ that is closest to x. If x is exactly halfway between two elements of $\mathbb{F}_n$, $\circ_n(x)$ is the one for which M is an even number.

The IEEE 754 standard requires that the user should be able to choose one rounding mode among these ones, called the **active rounding mode**. After that, when performing one of the 4 arithmetic operations, or when computing square roots, the obtained result should be equal to the rounding of the exact result.

For $a \in \mathbb{F}_n$, we define a^+ as its **successor** in $\mathbb{F}_n$, that is, $a^+ = \min\{b \in \mathbb{F}_n, b > a\}$, and $ulp(a)$ as $|a|^+ - |a|$. If a is not an element of $\mathbb{F}_n$, we define $\mathrm{ulp}(a)$ as $\circ_u(a) - \circ_d(a)$. The name ulp is an acronym for **unit in the last place**. When $x \in \mathbb{F}_n$, $ulp(x)$ is the "weight" of the last mantissa bit of x. We also define a^- as the **predecessor of** a.

We call a **breakpoint** a value z where the rounding changes, that is, if t_1 and t_2 are real numbers satisfying $t_1 < z < t_2$ and $\circ_t$ is the rounding mode, then $\circ_t(t_1) < \circ_t(t_2)$. For "directed" rounding modes (i.e., towards $+\infty$, $-\infty$ or 0), the breakpoints are the floating-point numbers. For rounding to the nearest mode, they are the exact middle of two consecutive floating-point numbers.

3 Division

3.1 Preliminary results

We will frequently use the following, straightforward, property.

Property 1 *Let $y \in \mathbb{F}_n$. There exists q such that $1/y$ belongs to $\mathbb{F}_q$ if and only if y is a power of 2.*

The following very elementary property will help to simplify some proofs.

Property 2 *Let $x, y \in \mathbb{F}_n$. If $x \neq y$, then the distance between x/y and 1 is at least 2^{-n}.*

The next straightforward result gives a lower bound on the distance between a breakpoint (in round-to-nearest mode) and the quotient of two floating-point numbers.

Property 3 *If $x, y \in \mathbb{F}_n$, $1 \leq x, y < 2$, then the distance between x/y and the middle of two consecutive floating-point numbers is lower-bounded by*

- $\dfrac{1}{y \times 2^{2n-1}} > \dfrac{1}{2^{2n}}$ *if* $x \geq y$;
- $\dfrac{1}{y \times 2^{2n}} > \dfrac{1}{2^{2n+1}}$ *otherwise.*

Moreover, if the last mantissa bit of y is a zero, then the lower bounds become twice these ones.

3.2 The naive method

As said in the introduction, we have to evaluate x/y, and y is known before x (either earlier at run-time, or at compile-time). An obvious solution consists in pre-computing $z = 1/y$ (or more precisely z rounded-to-nearest, that is, $z_h = \circ_n(1/y)$), and then to multiply x by z_h. We will refer to this method as "the naive method". Unfortunately, this method does not necessarily give the correctly-rounded expected result. Before trying to give better solutions, let us focus on the properties of that one. We assume round-to-nearest mode.

3.2.1 Maximum error of the naive solution

Property 4 *The naive solution returns a result that is at most at distance:*

- 1.5 *ulps from the exact result if* $x < y$;
- 1 *ulp from the exact result if* $x \geq y$

More precisely, if $x < y$ and $1 \leq x, y < 2$, the following property holds. This property will allow us to analyze the behavior of another algorithm (Algorithm 1).

Table 1: Largest error (in ulps) of the naive solution for small values of n.

n	Largest error
6	1.246
7	1.312
8	1.344
9	1.416
10	1.419
11	1.429

Property 5 *If $x < y$ and $1 \leq x, y < 2$, then the naive solution returns a result q that satisfies:*

- *either q is within 1 ulp from x/y;*
- *or x/y is at least at a distance*

$$\frac{2^{-2n+1}}{y} + 2^{-2n+1} - \frac{2^{-3n+2}}{y}$$

from a breakpoint of the round-to-nearest mode.

It is worth noticing that there are values x and y for which the naive solution leads to an error quite close to 1.5 ulps. Table 1, computed by exhaustive searching, gives the largest error (in ulps) of the naive solution for small values of n.

An open question is to find a fast algorithm that builds, for a given n, the values x and y for which the maximum error is committed.

3.2.2 Probability of getting a correctly rounded result using the naive solution

For the first few values of n (up to $n = 13$), we have computed, through exhaustive testing, the proportion of couples (x, y) for which the naive method gives a incorrectly rounded result. These results are given in Table 2. The proportion seems to converge, as n grows, to a constant value that is around 27%. More precisely,

Conjecture 1 *Assuming a uniform distribution of the mantissas of floating-point numbers, rounding to nearest, and n bits of mantissa, the probability, for given values x and y that the naive method return an incorrect result (that is, a result different from $\circ_n(x/y)$) goes to $13/48 = 0.2708\cdots$ as n goes to $+\infty$.*

It is worth being noticed that this conjecture is an "half-conjecture" only, since we have a rough sketch of a proof, given in the Appendix.

The figures given in Table 2 and our conjecture tend to show that for any n, the naive method gives a proportion of not correctly rounded results around 27%, which is by far too large to be neglected.

3.2.3 Values of y for which the naive method always work

Depending on n, there are a very few values of y (including, of course, the powers of 2) for which the naive method always work (i.e., for all values of x). These values for n less than 13 are given in Table 3. Unfortunately, we have not succeeded in being able to predict them, nor to compute them much faster than by exhaustive testing (which does not allow to tackle with the most interesting values of n, namely 24 and 53).

3.3 Division with one multiplication and two fused MACs

On some modern processors (such as the PowerPC, the IBM RISCSystem / 6000 [12] and IA64-based architectures [2, 11]), a fused-multiply accumulate instruction (fused-MAC) is available. This makes

Table 2: Actual probability of an incorrect result for small values of n.

n	probability
5	$0.2578\cdots$
6	$0.2773\cdots$
7	$0.2434\cdots$
8	$0.2562\cdots$
9	$0.2644\cdots$
10	$0.2708\cdots$
11	$0.2737\cdots$
12	$0.2697\cdots$
13	$0.2717\cdots$

Table 3: The n-bit numbers y between 1 and 2 for which, for any n-bit number x, $\circ_n(x \times \circ_n(1/y))$ equals $\circ_n(x/y)$.

n					
3	1				
4	1				
5	1	$\frac{19}{16}$			
6	1				
7	1	$\frac{105}{64}$	$\frac{117}{64}$		
8	1	$\frac{151}{128}$	$\frac{163}{128}$	$\frac{183}{128}$	$\frac{217}{128}$
9	1	$\frac{307}{256}$			
10	1				
11	1	$\frac{1705}{1024}$	$\frac{1971}{1024}$		
12	1				
13	1	$\frac{4411}{4096}$	$\frac{4551}{4096}$	$\frac{4915}{4096}$	$\frac{7735}{4096}$

it possible to evaluate an expression of the form $ax + b$ with one final (correct) rounding only. That is, we compute $\circ_t(ax + b)$, where $\circ_t$ is the active rounding mode.

Let us now investigate how can such an instruction be used to solve our problem. One can use the following result, due to Cornea, Golliver and Markstein [3], that was designed in order to get a correctly rounded result from an approximation to a quotient obtained using Newton-Raphson or Goldschmidt iterations.

Theorem 1 (Cornea, Golliver and Markstein, 1999 [3]) *Assume, $x, y \in \mathbb{F}_n$. If z_h is within $1/2$ ulp of $1/y$ and $q \in \mathbb{F}_n$, $q \approx x/y$ within 1 ulp of x/y then one application of*

$$\begin{cases} r &= \circ_n(x - qy) \\ q' &= \circ_n(q + rz_h) \end{cases}$$

yields $q' = \circ_n(x/y)$.

One would like to use Theorem 1 to get a correctly rounded result from an initial value q obtained by the naive method, that is, by computing $\circ_n(xz_h)$, where $z_h = \circ_n(1/y)$. Unfortunately, q will not always be within one ulp from x/y (see Property 4), so Theorem 1 cannot be directly applied. One could get a better initial approximation to x/y by performing one step of Newton-Raphson iteration from q. But this turns out to be useless: such an iteration step is not necessary, as shown by the following result. It is worth noticing that this result could also be used to save a few Mac instructions at the end of some algorithms for "usual" division (such as the one given in [3], page 100).

Theorem 2 (Division with one multiplication and two Macs) *If $n \geq 4$ then Algorithm 1, given below, always returns the correctly rounded (to nearest) quotient $\circ_n(x/y)$.*

Algorithm 1 (Division with one multiplication and two Macs)

- *in advance, evaluate $z_h = \circ_n(1/y)$;*
- *as soon as x is known, compute $q = \circ_n(x \times z_h)$;*
- *compute $r = \circ_n(x - qy)$;*
- *compute $q' = \circ_n(q + rz_h)$.*

This method requires 1 multiplication before x is known, and 3 consecutive (and dependent) MACs once x is known.

In the following sections, we try to design a faster algorithm. Unfortunately, it does not work for all possible values of y, so a preliminary testing turns out to be necessary.

3.4 Division with one multiplication and one fused MAC

Using the method presented in Section 3.3, we could compute x/y using one multiplication and two MACs, once x is known. Let us show that in many cases, one multiplication and one MAC (once x is known) do suffice. To do this, we need a double-word approximation to $1/y$. Let us first see how can such an approximation be computed.

3.4.1 Preliminary result: Getting a double-word approximation to $1/y$.

Kahan [7] explains that the fused MAC allows to compute remainders exactly. Let us show how it works. Let $x, y, q \in \mathbb{F}_n$, such that

$$q \in \{\circ_d(x/y), \circ_u(x/y)\}.$$

Without loss of generality, we assume $1 \leq x, y < 2$.

Property 6 *$r = x - qy$ is computed exactly with a fused MAC. That is, $\circ_n(x - qy) = x - qy$.*

The division methods we are now going to examine will require a double-word approximation to $1/y$, that is, two floating-point values z_h and z_ℓ such that $z_h = \circ_n(1/y)$ and $z_\ell = \circ_n(1/y - z_h)$. The only reasonably fast algorithm we have found for getting these values requires the availability of a fused MAC. This means that without a fused MAC, the following methods can be used only if y is a constant, or if it is known *much* before x.

Using Property 6, let us now show z_h and z_ℓ can be computed using a fused MAC.

Property 7 *Assume $y \in \mathbb{F}_n$, $y \neq 0$. The following sequence of 3 operations computes z_h and z_ℓ such that $z_h = \circ_n(1/y)$ and $z_\ell = \circ_n(1/y - z_h)$.*

- $z_h = \circ_n(1/y)$;
- $\rho = \circ_n(1 - y z_h)$;
- $z_\ell = \circ_n(\rho/y)$;

3.4.2 The algorithm

We assume that from y, we have computed $z = 1/y$, $z_h = \circ_n(z)$ and $z_\ell = \circ_n(z - z_h)$ (for instance using Property 7). We suggest the following 2-step method:

Algorithm 2 (Division with one multiplication and one MAC) *Compute:*

- $q_1 = \circ_n(x z_\ell)$;
- $q_2 = \circ_n(x z_h + q_1)$.

Theorem 3 *Algorithm 2 gives a correct result (that is, $q_2 = \circ_n(x/y)$), as soon as at least one of the following conditions is satisfied:*

1. *the last mantissa bit of y is a zero;*
2. *n is less than or equal to 8;*
3. *$n = 9$ and $y \notin \{469, 485\}$;*
4. *$n = 10$ and $y \notin \{795, 837, 849, 967, 999, 1015\}$;*
5. *for some reason, we know in advance that the mantissa of x will be larger than that of y;*
6. *Algorithm 3, given below, returns **true** when the input value is the integer $Y = y \times 2^{n-1-e_y}$, where e_y is the exponent of y (that is, Y is the mantissa of y, interpreted as an integer).*

Algorithm 3 (Tries to find solutions to Eqn. (5).) *We give the algorithm as a Maple program. If it returns "true" then Algorithm 2 returns a correctly rounded result. It requires the availability of $2n + 1$-bit integer arithmetic.*

```
TestY := proc(Y,n)
 local Pminus, Qminus, Xminus, OK, Pplus, Qplus, Xplus;
 Pminus := (1/Y) mod 2^(n+1)
 # requires computation of a modular inverse
 Qminus := (Pminus-1) / 2;
 Xminus := ((2 * Qminus + 1) * Y - 1) / 2^(n+1);
 if (Qminus >= 2^(n-1)) and (Xminus >= 2^(n-1))
          then OK := false else OK := true end if;
 Pplus := (-1/Y) mod 2^(n+1);
 Qplus := (Pplus-1) / 2;
 Xplus := ((2 * Qplus + 1) * Y + 1) / 2^(n+1);
 if (Qplus >= 2^(n-1)) and (Xplus >= 2^(n-1))
      then OK := false end if;
 print(OK)
end proc;
```

Translation of Algorithm 3 into a C or Fortran program can be done without real difficulty. Computing a modular reciprocal modulo a power of two requires a few arithmetic operations only, using the extended Euclidean GCD algorithm [8].

Let us discuss the consequences of Theorem 3.

- condition "the last mantissa bit of y is a zero" is very easily checked on most systems. Hence, that condition can be used for accelerating divisions when y is known at run-time, soon enough[1] before x. Assuming that the last bits of the mantissas of the floating-point numbers appearing in computations are 0 or 1 with probability $1/2$, that condition allows to accelerate half divisions;
- Our experimental testings up to $n = 24$ show that condition "Algorithm 3 returns **true**" allows to accelerate around 39% of the remaining cases (i.e., the cases for which the last bit of y is a one). And yet, it requires much more computation: it is definitely interesting if y is known at compile-time, and might be interesting if y is known at run-time much before x, or if many divisions by the same y are performed.

Conclusion

We have suggested several ways of accelerating a division of the form x/y, where y is known before x. Our methods could be used in optimizing compilers, to make some numerical programs run faster, without any loss in terms of accuracy. Algorithm 1 always works and do not require much pre-computation (so it can be used even if y is known a few tens of cycles only before x). Algorithm 2 is faster, and yet it requires much pre-computation (for computing z_h and z_ℓ, and making sure that the algorithm works) so it is more suited for division by a constant, or if y is known much before x – in some cases a compiler may make sure that this occurs, without delay penalty.

References

[1] American National Standards Institute and Institute of Electrical and Electronic Engineers. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985*, New York, 1985.

[2] M. Cornea-Hasegan and Bob Norin. IA-64 floating-point operations and the ieee standard for binary floating-point arithmetic. *Intel Technology Journal*, Q4, 1999.

[3] Marius A. Cornea-Hasegan, Roger A. Golliver, and Peter Markstein. Correctness proofs outline for newton-raphson based floating-point divide and square root algorithms. In Koren and Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 96–105, Los Alamitos, CA, April 1999. IEEE Computer Society Press.

[4] M. D. Ercegovac and T. Lang. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, Boston, 1994.

[5] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, March 1991.

[6] Cristina Iordache and David W. Matula. On infinitely precise rounding for division, square root, reciprocal and square root reciprocal. In Koren and Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 233–240, Los Amlamitos, CA, April 1999. IEEE Computer Society Press.

[7] W. Kahan. Lecture notes on the status of IEEE-754. Postscript file accessible electronically through the Internet at the address http://http.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps, 1996.

[1] The order of magnitude behind this "soon enough" highly depends on the architecture and operating system.

[8] D. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, Reading, MA, 1973.

[9] I. Koren. *Computer arithmetic algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[10] T. Lang and J.M. Muller. Bound on run of zeros and ones for algebraic functions. In Burgess and Ciminiera, editors, *Proc. of the 15th IEEE Symposium on Computer Arithmetic (Arith-15)*. IEEE Computer Society Press, 2001.

[11] P. Markstein. *Ia-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. ISBN: 0130183482.

[12] P. W. Markstein. Computation of elementary functions on the IBM risc system / 6000 processor. *IBM Journal of Research and Development*, 34(1):111–119, January 1990.

[13] S. F. Oberman and M. J. Flynn. Division algorithms and implementations. *IEEE Transactions on Computers (to appear)*, 1997.

Appendix: proof of the properties and theorems

Proof of Property 1. It obviously suffices to assume that $1 \leq y < 2$. Define $z = 1/y$ and assume $z \in \mathbb{F}_q$. This gives:

- $Y = 2^{n-1}y$ is an integer;
- $Z = 2^q z$ is an integer (since $1/2 < z \leq 1$).

YZ is equal to 2^{n+q-1}. Therefore, in the prime number decomposition of Y and Z, 2 is the only prime number that can appear. □

Proof of Property 2. The proof is elementary by considering that, for a given y, the values closest to 1 are obtained for $x = y^+$ or y^-. This minimal distance is actually attained, for $x = 2 - 2^{-n+1}$ and $y = 2$. □

Proof of Property 3. The numbers $X = x \times 2^{n-1}$ and $Y = y \times 2^{n-1}$ are integers. A breakpoint (for the round-to-nearest mode) has the form $P/2^{n+1}$ for $x < y$ and $P/2^n$ for $x \geq y$, where P is odd. So the distance between x/y and a breakpoint has the form:

- if $x < y$:
$$\frac{2^{n+1}X - PY}{Y2^{n+1}}$$
- if $x \geq y$:
$$\frac{2^{n}X - PY}{Y2^{n+1}}.$$

The numerators of these fractions cannot be zero (otherwise, since P is odd, Y would be a multiple of 2^n, which is impossible). Hence the absolute value of these numerators is at least one. Moreover, if Y is even, the numerators are even numbers, so their absolute value is at least 2. □

Proof of Property 4. Let $x, y \in \mathbb{F}_n$. Without loss of generality, we can assume that x and y belong to $[1, 2)$. Since the cases $x, y = 1$ or 2 are straightforward, we assume that x and y belong to $(1, 2)$. Since $z_h = \circ_n(z)$ and $z \in (1/2, 1)$, we have,

$$\left|\frac{1}{y} - z_h\right| < 2^{-n-1}.$$

Therefore,

$$\left|\frac{x}{y} - xz_h\right| < 2^{-n}. \tag{1}$$

From Property 2 and (1), we cannot have $x/y > 1$ and $xz_h < 1$ or the converse. So xz and xz_h belong to the same "binade" (i.e., $\mathrm{ulp}(xz_h) = \mathrm{ulp}(xz)$).

Now, there are two possible cases:

- if $x \geq y$, then
$$|xz_h - \circ_n(xz_h)| \leq 2^{-n}$$
so
$$\left|\frac{x}{y} - \circ_n(xz_h)\right| < 2^{-n+1} = \mathrm{ulp}(x/y).$$

- if $x < y$, then

$$|xz_h - \circ_n(xz_h)| \leq 2^{-n-1}$$

so

$$\left|\frac{x}{y} - \circ_n(xz_h)\right| < 3 \times 2^{-n-1} = 1.5 \times \text{ ulp}(x/y).$$

□

Proof of Property 5. The proof is very similar to that of property 4. We just use the tighter bounds:

- $|1/y - z_h| < 2^{-n-1} - 2^{-2n}/y$ (this comes from Property 3: $1/y$ is at a distance at least $2^{-2n}/y$ from a breakpoint);
- $x \leq 2 - 2^{-n+2}$ (this comes from $x < y < 2$, which implies $x \leq (2^-)^-$).

Combining these bounds gives

$$\left|\frac{x}{y} - xz_h\right| \leq 2^{-n} - \frac{2^{-2n+1}}{y} - 2^{-2n+1} + \frac{2^{-3n+2}}{y}.$$

The final bound ℓ_{min} is obtained by adding the $1/2$ ulp bound on $|xz_h - \circ n(xz_h)|$:

$$\left|\frac{x}{y} - \circ_n(xz_h)\right| \leq \ell_{min} = 3 \times 2^{-n-1} - \frac{2^{-2n+1}}{y} - 2^{-2n+1} + \frac{2^{-3n+2}}{y}.$$

Now, if $\circ_n(xz_h)$ is not within 1 ulp from x/y, it means that x/y is at a distance at least $1/2$ ulp from the breakpoints that are immediately above or below $q = \circ_n(xz_h)$. And since the breakpoints that are immediately above $\circ_n(xz_h)^+$ or below $\circ_n(xz_h)^-$ are at a distance 1.5 ulps $= 3 \times 2^{-n-1}$ from $\circ_n(xz_h)$, x/y is at least at a distance $3 \times 2^{-n-1} - \ell_{min}$ from these breakpoints. This is illustrated in Figure 1. □

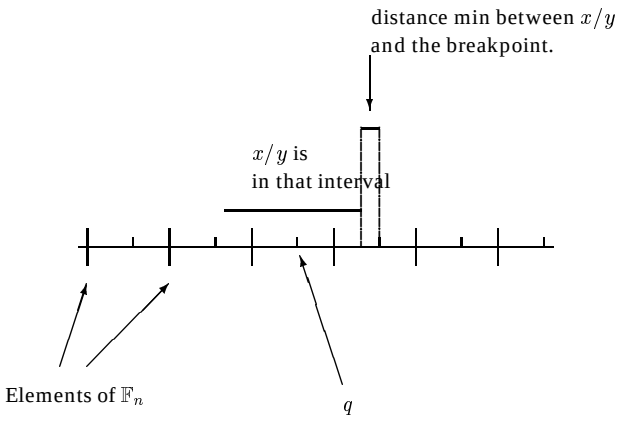


Figure 1: Illustration of the Proof of Property 5.

Sketch of a proof for conjecture 1.

Define $z = 1/y = z_h + z_\rho$, where $z_h = \circ_n(z)$, with $1 < y < 2$. When $n \to \infty$, the maximum value of $|z_\rho|$ is asymptotically equal to $1/2\text{ulp}(z)$, and its average value is asymptotically equal to $1/4\text{ulp}(z) = 2^{-n-2}$. Hence, for $1 < x < 2$, we can write:

$$xz = xz_h + \epsilon$$

where the average value of $|\epsilon|$ is

- $\frac{y+1}{2} \times 2^{-n-2} = (y+1)2^{-n-3}$ for $x < y$;
- $\frac{2+y}{2} \times 2^{-n-2} = (2+y)2^{-n-3}$ for $x > y$.

(to get these figures, we multiply the average value of ϵ by the average value of x, which is $\frac{y+1}{2}$ for $1 < x < y$ and $\frac{2+y}{2}$ for $y < x < 2$).

The "breakpoints" of the rounding mode, that is, the values where the rounding changes[2], are regularly spaced, at distance 2^{-n} for $x < y$, and 2^{-n+1} for $x > y$. Therefore, the probability that $\circ_n(xz) \neq \circ_n(xz_h)$ is the probability that there should be a breakpoint between these values. That probability is:

- $(y+1)2^{-n-3}/2^{-n} = \frac{y+1}{8}$ for $x < y$;
- $(2+y)2^{-n-3}/2^{-n+1} = \frac{y+2}{16}$ for $x > y$.

Therefore, for a given y, the probability that the naive method should give a result different from $\circ_n(x/y)$ is

$$\frac{(y+1)(y-1)}{8} + \frac{(y+2)(y-2)}{16} = \frac{y^2}{16} + \frac{1}{8}.$$

Therefore, assuming now that y is variable, the probability that the naive method give an incorrectly rounded result is

$$\int_1^2 \left(\frac{y^2}{16} + \frac{1}{8}\right) dy = \frac{13}{48} \approx 0.27.$$

□

Proof of Theorem 2. We assume $1 \leq x, y < 2$. First, let us notice that if $x \geq y$, then (from Property 4), q is within one ulp from x/y, therefore Theorem 1 applies, hence $q' = \circ_n(x/y)$. Let us now focus on the case $x < y$. Define

$$\begin{cases} \epsilon_1 &= \dfrac{x}{y} - q \\ \epsilon_2 &= \dfrac{1}{y} - z_h \end{cases}$$

From Property 4 and the definition of rounding to nearest, we have,

$$\begin{cases} |\epsilon_1| &< 3 \times 2^{-n-1} \\ |\epsilon_2| &< 2^{-n-1} \end{cases}$$

The number $\rho = x - qy = \epsilon_1 y$ is less than 3×2^{-n} and is a multiple of 2^{-2n+1}. This shows that it can be represented exactly with $n+1$ bits of mantissa. Hence, the difference between that number and $r = \circ_n(x - qy)$ (i.e., ρ rounded to n bits of mantissa) is zero or $\pm 2^{-2n+1}$. Therefore,

$$r = \epsilon_1 y + \epsilon_3, \text{ with } \epsilon_3 \in \{0, \pm 2^{-2n+1}\}.$$

Let us now compute $q + rz_h$. We have

$$\begin{aligned} q + rz_h &= \left(\frac{x}{y} - \epsilon_1\right) + (\epsilon_1 y + \epsilon_3)\left(\frac{1}{y} - \epsilon_2\right) \\ &= \frac{x}{y} + \frac{\epsilon_3}{y} - \epsilon_1\epsilon_2 y - \epsilon_2\epsilon_3 \end{aligned}$$

[2] Since we assume rounding to nearest mode, the breakpoints are the exact middles of two consecutive machine numbers.

Hence,

$$\left|\frac{x}{y} - (q + rz_h)\right| \leq \frac{2^{-2n+1}}{y} + 3 \times 2^{-2n-2}y + 2^{-3n}$$

Define $\epsilon = 2^{-2n+1}/y + 3 \times 2^{-2n-2}y + 2^{-3n}$. Now, From Property 5:

- either q was at a distance less than one ulp from x/y (but in such a case, $q' = \circ_n(x/y)$ from Theorem 1);
- or q is at a distance larger than one ulp from x/y. In such a case, x/y is at least at a distance

$$\delta = \frac{2^{-2n+1}}{y} + 2^{-2n+1} + \frac{2^{-3n+2}}{y}.$$

A straightforward calculation shows that, if $n \geq 4$, then $\epsilon < \delta$. It makes it possible to deduce that there is no breakpoint between x/y and $q + rz_h$. Hence $\circ_n(q + rz_h) = \circ_n(x/y)$, q.e.d.

□

Proof of property 6. Define

$$K = \begin{cases} n+1 & \text{if} \quad q \leq 1 \\ n & \text{if} \quad q > 1 \end{cases}$$

It suffices to notice that r is a multiple of 2^{-n-K+2} that is less than $2^{-K+1}y$. This suffices to show that $r \in \mathbb{F}_n$. Hence, it is computed exactly. □

Proof of Property 7. From Property 6, ρ is computed exactly. Therefore it is exactly equal to $1 - yz_h$. Hence, ρ/y is equal to $1/y - z_h$. Hence, z_ℓ is equal to $\circ_n(1/y - z_h)$. □

Proof of Theorem 3. The cases $n \leq 8$ and $n = 9, 10$ have been processed through exhaustive searching. Let us deal with the other cases. Without loss of generality, we can assume $x \in (1,2)$ and $y \in (1,2)$ (the cases $x = 1$ and $y = 1$ are straightforward). This gives $z \in (1/2, 1)$. Hence, $z_h \in [1/2, 1]$. The case $z_h = 1$ is impossible (from $y > 1$ and $y \in \mathbb{F}_n$ we deduce $y \geq 1 + 2^{-n+1}$, therefore $1/y \leq 1 - 2^{-n+1} + 2^{-2n+2} < 1 - 2^{-n} \in \mathbb{F}_n$, therefore $\circ_n(1/y) \leq 1 - 2^{-n}$). From that, we deduce that the binary representation of z_h has the form $0.z_h^1 z_h^2 z_h^3 \cdots z_h^n$. Since z_h is obtained by rounding z to the nearest, we have:

$$|z - z_h| \leq \frac{1}{2}\text{ulp}(z) = 2^{-n-1}$$

Moreover, Property 1 shows that the case $|z - z_h| = 2^{-n-1}$ is impossible. Therefore

$$|z - z_h| < 2^{-n-1}.$$

From this, we deduce: $|z_\ell| = |\circ_n(z - z_h)| \leq 2^{-n-1}$. Again, the case $|z_\ell| = 2^{-n-1}$ is impossible. This would imply:

$$\left|z - \left(z_h + 2^{-n-1}\right)\right| < 2^{-2n-1}$$

or

$$\left|z - \left(z_h - 2^{-n-1}\right)\right| < 2^{-2n-1}$$

which would contradict the fact that the binary representation of the reciprocal of an n-bit number cannot contain more than $n-1$ consecutive zeros or ones [6, 10]. Therefore:

$$|z_\ell| < 2^{-n-1}.$$

Thus, from the definition of z_ℓ:

$$|(z - z_h) - z\ell| < 2^{-2n-2},$$

thus,

$$|x(z - z_h) - xz\ell| < 2^{-2n-1},$$

thus,

$$\begin{aligned} |x(z - z_h) - \circ_n(xz\ell)| &< 2^{-2n-1} + \frac{1}{2}\text{ulp}(xz_\ell) \\ &< 2^{-2n}. \end{aligned}$$

Therefore,

$$|xz - \circ_n[xz_h + \circ_n(xz_\ell)]| < 2^{-2n} + \frac{1}{2}\text{ulp}\left(xz_h + \circ_n(xz_\ell)\right) \tag{2}$$

Hence, if for a given y there does not exist any x such that $x/y = xz$ is at a distance less than 2^{-2n} from the middle of two consecutive floating-point numbers, then $\circ_n[xz_h + \circ_n(xz_\ell)]$ will always be equal to $\circ_n(xz)$, i.e., Algorithm 2 will give a correct result. So, let us now try to find values of y for which that property holds. To do that, let us try to characterize all possible values x and y for which x/y is at a distance less than 2^{-2n} from the middle of two consecutive floating-point numbers.

- If $x \geq y$. Let $q = \circ_n(x/y)$, and define integers X, Y and Q as

  $$\begin{cases} X &= x \times 2^{n-1} \\ Y &= y \times 2^{n-1} \\ Q &= q \times 2^{n-1} \end{cases}$$

  If we have

  $$\frac{x}{y} = q + 2^{-n} + \epsilon, \text{ with } |\epsilon| < 2^{-2n},$$

  then

  $$2^n X = 2QY + Y + 2^n \epsilon Y, \text{ with } |\epsilon| < 2^{-2n}. \tag{3}$$

  This is impossible:

  - Equation (3) implies that $R = 2^n \epsilon Y$ should be an integer.
  - The bounds $Y < 2^n$ and $\epsilon < 2^{-2n}$ imply $|R| < 1$;
  - Property 1 implies $R \neq 0$.

- If $x < y$. Let $q = \circ_n(x/y)$, and define integers X, Y and Q as

  $$\begin{cases} X &= x \times 2^{n-1} \\ Y &= y \times 2^{n-1} \\ Q &= q \times 2^{n} \end{cases}$$

  If we have

  $$\frac{x}{y} = q + 2^{-n-1} + \epsilon, \text{ with } |\epsilon| < 2^{-2n},$$

  then

  $$2^{n+1} X = 2QY + Y + 2^{n+1} \epsilon Y, \text{ with } |\epsilon| < 2^{-2n}. \tag{4}$$

  But:

  - Equation (4) implies that $R' = 2^{n+1} \epsilon Y$ should be an integer.
  - The bounds $Y < 2^n$ and $\epsilon < 2^{-2n}$ imply $|R'| < 2$;
  - Property 1 implies $R' \neq 0$.

Hence, the only possibility is $R' = \pm 1$. Therefore, to find values y for which for any x Algorithm 2 gives a correct result, we have to examine the possible integer solutions to

$$\begin{cases} 2^{n+1}X = 2QY + Y \pm 1 \\ 2^{n-1} \leq X \leq 2^n - 1 \\ 2^{n-1} \leq Y \leq 2^n - 1 \\ 2^{n-1} \leq Q \leq 2^n - 1 \end{cases} \tag{5}$$

There are no solutions to (5) for which Y is even. This shows that if the last mantissa bit of y is a zero, then Algorithm 2 always returns a correctly rounded result.

Now, if $2^{n+1}X = (2Q+1)Y \pm 1$, this means that $2Q + 1 = (\pm 1/Y) \bmod 2^{n+1}$, hence the last condition of the theorem.

□