Laboratoire de l'Informatique du Parallélisme

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

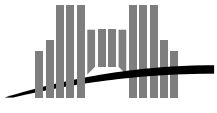
When double rounding is odd

Sylvie Boldo,
Guillaume Melquiond

November 2004

Research Report N° 2004-48

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`



When double rounding is odd

Sylvie Boldo, Guillaume Melquiond

November 2004

Abstract

Double rounding consists in a first rounding in an intermediate extended precision and then a second rounding in the working precision. The natural question is then of the precision and correctness of the final result. Unfortunately, the used double rounding algorithms do not obtain a correct rounding of the initial value. We prove an efficient algorithm for the double rounding to give the correct rounding to the nearest value assuming the first rounding is to odd. As this rounding is unusual and this property is surprising, we formally proved this property using the Coq automatic proof checker.

Keywords: Floating-point, double rounding, formal proof, Coq.

Résumé

Le double arrondi consiste en un premier arrondi dans une précision étendue suivi d'un second arrondi dans la précision de travail. La question naturelle qui se pose est celle de la précision et de la correction du résultat final ainsi obtenu. Malheureusement, les algorithmes de double arrondi utilisés ne produisent pas un arrondi correct de la valeur initiale. Nous décrivons ici un algorithme efficace de double arrondi permettant de fournir l'arrondi correct au plus proche à condition que le premier arrondi soit impair. Comme cet arrondi est inhabituel et que cette propriété est surprenante, nous avons prouvé formellement ce résultat en utilisant l'assistant de preuves Coq.

Mots-clés: Virgule flottante, double arrondi, preuve formelle, Coq.

1 INTRODUCTION

Floating-point users expect their computations to be correctly rounded: this means that the result of a floating-point operation is the same as if it was first computed with an infinite precision and then rounded to the precision of the destination format. This is required by the IEEE-754 standard [5, 6] that is followed by modern general-purpose processors.

But this standard does not require the FPU to directly handle each floating-point format (single, double, double extended): some systems deliver results only to extended destinations. On such a system, the user shall be able to specify that a result be rounded instead to a smaller precision, though it may be stored in the extended format. It happens in practice with some processors as the Intel x86, which computes with 80 bits before rounding to the IEEE double precision (53 bits), or the PowerPC, which provides IEEE single precision by double rounding from IEEE double precision.

Hence, double rounding may occur if the user did not explicitly specify beforehand the destination format. Double rounding consists in a first rounding in an extended precision and then a second rounding in the working precision. As described in Section 2, this rounding may be erroneous: the final result is sometimes different from the correctly rounded result.

It would not be a problem if the compilers were indeed setting correctly the precisions of each floating-point operation for processors that only work in an extended format. To increase efficiency, they do not usually force the rounding to the wanted format since it is a costly operation. Consequently, there is a first rounding to the extended precision corresponding to the floating-point operation itself and a second rounding when the storage in memory is done.

Therefore, double rounding is usually considered as a dangerous feature leading to unexpected inaccuracy. Nevertheless, double rounding is not necessarily a threat: we give a double rounding algorithm that ensures the correctness of the result, meaning that the result is the same as if only one direct rounding happens. The idea is to prevent the first rounding to approach the tie-breaking value between the two possible floating-point results.

2 DOUBLE ROUNDING

2.1 Floating-point definitions

Our formal proofs are based on the floating-point formalization of Théry and on the corresponding library [3, 2] in Coq [1]. Floating-point numbers are represented by pairs (n, e) that stand for $n \times 2^e$. We use both an integral signed mantissa n and an integral signed exponent e for sake of simplicity.

A floating-point format is denoted by $\mathbb{B}$ and is composed by the lowest exponent $-E$ available and the precision p. We do not set an upper bound on the exponent as overflows do not matter here (see below). We define a representable pair (n, e) such that $|n| < 2^p$ and $e \geq -E$. We denote by $\mathbb{F}$ the subset of real numbers represented by these pairs for a given format $\mathbb{B}$. Now only the representable floating-point numbers will be refered to; they will simply be denoted as floating-point numbers.

All the IEEE-754 rounding modes were also defined in the Coq library, especially the default rounding: the even closest rounding, denoted by $\circ$. We have $f = \circ(x)$ if f is the floating-point number closest to x; when x is in the middle of two consecutive floating-point numbers, the even one is chosen.

A rounding mode is defined in the Coq library as a relation between a real number and a floating-point number, and not a function from real values to floats. Indeed, there may be several floats corresponding to the same real value. For a relation, a weaker property than being a rounding mode is being a faithful rounding. A floating-point number f is a faithful rounding of a real x if it is either the rounded up or rounded down of x, as shown on Figure 1. When x is a floating-point number, it is its own and only faithful rounding. Otherwise there always are two faithful roundings when no overflow occurs.

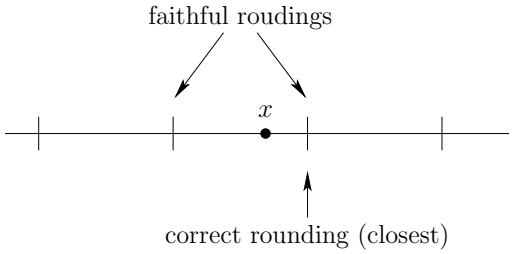


Figure 1: Faithful roundings.

2.2 Double rounding accuracy

In some situations, a floating-point computation may first be done in an extended precision, and later rounded to the working precision. The extended precision is denoted by $\mathbb{B}_e = (p+k, E_e)$ and the working precision is denoted by $\mathbb{B}_w = (p, E_w)$. If the same rounding mode is used for both computations (usually to closest even), it can lead to a less precise result than the result after a single rounding.

For example, see Figure 2. When the real value x is in the neighborhood of the middle of two close floating-point numbers g and h, it may first be rounded in one direction toward this middle t in extended precision, and then rounded in the same direction toward f in working precision. Although the result f is close to x, it is not the closest floating-point number to x, h is. When both roundings are to closest, we formally proved that the distance between the given result f and the real value x may be as much as

$$|f - x| \leq \left(\frac{1}{2} + 2^{-k-1}\right) \mathrm{ulp}(f).$$

When there is only one rounding, the corresponding inequality is $|f - x| \leq \frac{1}{2}\mathrm{ulp}(f)$. This is the expected result for a IEEE-754 compatible implementation.

2.3 Double rounding and faithfulness

Another interesting property of double rounding as defined previously is that it is a faithful rounding. We even have a more generic result.

Let us consider that the relations are not required to be rounding modes but only faithful roundings. We formally certified that the rounded result f of a double faithful rounding is faithful to the real initial value x, as shown in Figure 3. The requirements are $k \geq 0$ and $k \leq E_e - E_w$ (any normal float in the working format is normal in the extended format).

This means that any sequence of successive roundings in decreasing precisions gives a faithful rounding of the initial value.

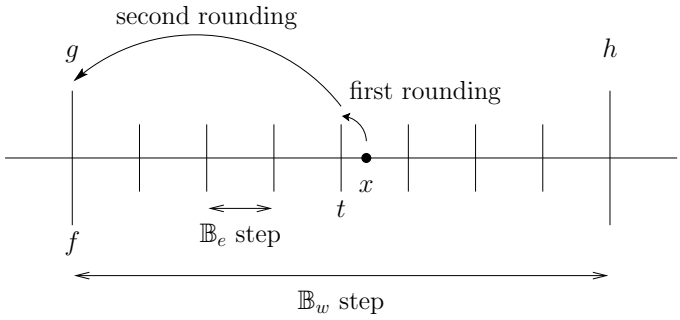


Figure 2: Bad case for double rounding.

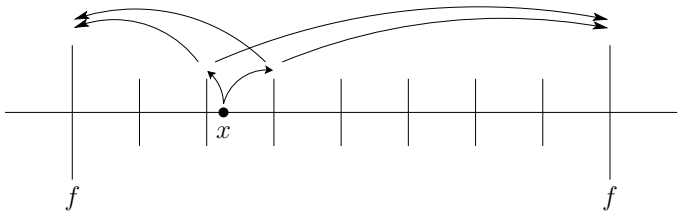


Figure 3: Double roundings are faithful.

3 ALGORITHM

As seen in previous sections, two roundings to closest induce a bigger round-off error than one single rounding to closest and may then lead to unexpected incorrect results. We now present how to choose the roundings for the double rounding to give a correct rounding to closest.

3.1 Odd rounding

This rounding does not belong to the IEEE-754's or even 754R[1]'s rounding modes. Algorithm 1 will justify the definition of this unusual rounding mode. It should not be mixed up with the rounding to the closest odd (similar to the default rounding: rounding to the closest even).

We denote by $\triangle$ the rounding toward $+\infty$ and by $\triangledown$ the rounding toward $-\infty$. The rounding to odd is defined by:

$$\begin{aligned} \square_{\text{odd}}(x) &= x \text{ if } x \in \mathbb{F}, \\ &= \triangle(x) \text{ if } \triangle(x) \text{ is odd}, \\ &= \triangledown(x) \text{ otherwise.} \end{aligned}$$

Note that the result of an odd rounding may be even only in the case where x is a representable even float.

The first proofs were to guarantee that this operator is a rounding mode as defined in our formalization [3] and a few other useful properties. This means that we proved that odd rounding is a rounding mode, this includes the proofs of:

- Each real can be rounded to odd.

[1] See http://www.validlab.com/754R/.

- Any odd rounding is a faithful rounding.
- Odd rounding is monotone.

We also certified that:

- Odd rounding is unique (meaning that it can be expressed as a function).
- Odd rounding is symmetric, meaning that if $f = \square_{\text{odd}}(x)$, then $-f = \square_{\text{odd}}(-x)$.

These properties will be used in the proof of Algorithm 1.

3.2 Correct double rounding algorithm

Algorithm 1 first computes the rounding to odd of the real value x in the extended format (with $p + k$ bits). It then computes the rounding to the closest even of the previous value in the working format (with p bits). We here consider a real value x but an implementation does not need to really handle x: it can represent the abstract exact result of an operation between floating-point numbers.

Algorithm 1 Correct double rounding algorithm.

$$\begin{aligned} t &= \square_{\text{odd}}^{p+k}(x) \\ f &= \circ^{p}(t) \end{aligned}$$

Assuming $p \geq 2$ and $k \geq 2$, and $E_e \geq 2 + E_w$, then

$$f = \circ^{p}(x).$$

Although there is a double rounding, we here guarantee that the computed result is correct. The explanation is in Figure 4 and is as follow.

When x is exactly equal to the middle of two consecutive floating-point numbers g and h (case 1), then t is exactly x and f is the correct rounding of x. Otherwise, when x is slightly different from this mid-point (case 2), then t is different from this mid-point: it is the odd value just greater or just smaller than the mid-point depending on the value of x. The reason is that, as $k \geq 2$, the mid-point is even in the $p+k$ precision, so t cannot be rounded into it if it is not exactly equal to it. This obtained t value will then be correctly rounded to f, which is the closest p-bit float from x. The other cases (case 3) are far away from the mid-point and are easy to handle.

Note that the hypothesis $E_e \geq 2 + E_w$ is not a strong requirement: due to the definition of E, it does not mean that the exponent range (as defined in the IEEE-754 standard) must be greater by 2. As $k \geq 2$, a sufficient condition is: any normal floating-point numbers with respect to $\mathbb{B}_w$ should be normal with respect to $\mathbb{B}_e$.

3.3 Proof

The pen and paper proof is a bit technical but does seem easy (see Figure 4). Unfortunately, the translation into a formal proof was unexpectedly tedious because of some complicated

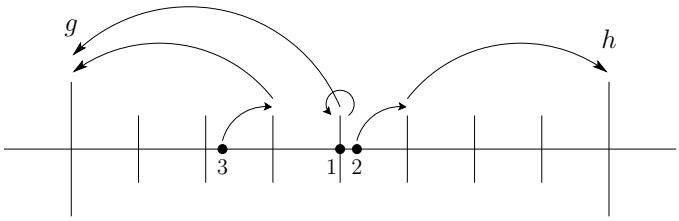


Figure 4: Different cases of Algorithm 1.

degenerate cases. The complex cases were especially the ones where v was a power of two, and subsequently where v was the smallest normal float. Those many splittings into subcases made the final proof rather long: 7 theorems and about one thousand lines of Coq, but we are now sure that every case (normal/subnormal, power of the radix or not) are correctly handled.

The general case, described by the preceding figure, was done in various subcases. The first split was the positive/negative one, due to the fact that odd rounding and even closest rounding are both symmetrical. Then let $v = \circ^p(x)$, we have to prove that $f = v$:

- when v is not a power of two,
  - when $x = t$ (case 1), then $f = \circ^p(t) = \circ^p(x) = v$,
  - otherwise, we know that $|x-v| \leq \frac{1}{2}\text{ulp}(v)$. As odd rounding is monotone and $k \geq 2$, it means that $|t - v| \leq \frac{1}{2}\text{ulp}(v)$. But we cannot have $|t - v| = \frac{1}{2}\text{ulp}(v)$ as it would imply that t is even in $p + k$ precision, which is impossible. So $|t - v| < \frac{1}{2}\text{ulp}(v)$ and $v = \circ^p(t) = f$.
- when v is a power of two,
  - when $x = t$ (case 1), then $f = \circ^p(t) = \circ^p(x) = v$,
  - otherwise, we know that $v - \frac{1}{4}\text{ulp}(v) \leq x \leq v + \frac{1}{2}\text{ulp}(v)$. As odd rounding is monotone and $k \geq 2$, it means that $v - \frac{1}{4}\text{ulp}(v) \leq t \leq v + \frac{1}{2}\text{ulp}(v)$. But we can have neither $v - \frac{1}{4}\text{ulp}(v) = t$, nor $t = v + \frac{1}{2}\text{ulp}(v)$ as it would imply that t is even in $p + k$ precision, which is impossible. So $v - \frac{1}{4}\text{ulp}(v) < t < v + \frac{1}{2}\text{ulp}(v)$ and $v = \circ^p(t) = f$.
- the case where v is the smallest normal floating-point number is handled separately, as $v - \frac{1}{4}\text{ulp}(v)$ should be replaced by $v - \frac{1}{2}\text{ulp}(v)$ in the previous proof subcase.

Even if the used formalization of floating-point numbers does not consider Overflows, this does not restrict the scope of the proof. Indeed, in the proof $+\infty$ can be treated as any other float (even as an even one) and without any problem. We only require that the Overflow threshold for the extended format is not smaller than the working format's.

4 APPLICATIONS

4.1 Rounding to odd is easy

This whole study would have no interest if rounding to odd was impossible to achieve. Fortunately, it is quite easy to implement it in hardware. Rounding to odd the real result x of a

floating-point computation can be done in two steps. First round it to zero into the floating-point number $\mathcal{Z}(x)$ with respect to the IEEE-754 standard. And then perform a logical or between the inexact flag ι (or the sticky bit) of the first step and the last bit of the mantissa. We later found that Goldberg [4] used this algorithm for binary-decimal conversions.

If the mantissa of $\mathcal{Z}(x)$ is already odd, this floating-point number is the rounding to odd of x too; the logical or does not change it. If the floating-point computation is exact, $\mathcal{Z}(x)$ is equal to x and ι is not set; consequently $\square_{\text{odd}}(x) = \mathcal{Z}(x)$ is correct. Otherwise the computation is inexact and the mantissa of $\mathcal{Z}(x)$ is even, but the final mantissa must be odd, hence the logical or with ι. In this last case, this odd float is the correct one, since the first rounding was toward zero.

Computing ι is not a problem *per se*, since the IEEE-754 standard requires this flag to be implemented, and hardware already uses sticky bits for the other rounding modes. Furthermore, the value of ι can directly be reused to flag the odd rounding of x as exact or inexact.

Another way to compute the rounding to odd is the following. We first round x toward zero with $p + k - 1$ bits. We then concatenate the inexact bit of the previous operation at the end of the mantissa in order to get a $p + k$-bit float. The justification is similar to the previous one.

4.2 Multi-precision operators

A possible application of our result is the implementation of multi-precision operators. We assume we want to get the correctly rounded value of an operator at various precisions (namely $p_1 < p_2 < p_3$ for example). It is then enough to get the result with odd rounding on $p_3 + 2$ bits (and a larger or equal exponent range) and some rounding to closest operators from the precision $p_3 + 2$ to smaller precisions p_1, p_2 and p_3. The correctness of Algorithm 1 ensures that the final result in precision p_i will be the correct even closest rounding of the exact value.

The same principle can be applied to the storage of constants: to get C correctly rounded in 24, 53 and 64 bits, it is enough to store it (oddly rounded) with 66 bits. Another example is when a constant may be needed either in single or in double precision by a software. Then, if the processor allows double-extended precision, it is sufficient to store the constant in double-extended precision and let the processor correctly round it to the required precision.

5 CONCLUSION

The algorithm described here is very simple and can be used in many real-life applications. Nevertheless, due to the bad reputation of double rounding, it is difficult to believe that double rounding may lead to a correct result. It is therefore essential to guarantee its validity. We formally proved its correctness with Coq, even in the unusual cases: power of two, subnormal floats, normal/subnormal frontier. All these cases made the formal proof longer and more difficult than one may expect at first sight. It is nevertheless very useful to have formally certified this proof, as the inequality handling was sometimes tricky and as the special cases were numerous and difficult.

This algorithm is even more general than what is presented here. It can also be applied to any realistic rounding to the closest (meaning that the result of a computation is uniquely defined by its operands and does not depend on the machine state). In particular, it handles the new rounding to the closest up defined by the revision of the IEEE-754 standard.

References

[1] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[2] Sylvie Boldo. *Preuves formelles en arithmétiques à virgule flottante*. PhD thesis, École Normale Supérieure de Lyon, November 2004.

[3] Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library of floating-point numbers and its application to exact computing. In *14th International Conference on Theorem Proving in Higher Order Logics*, pages 169–184, Edinburgh, Scotland, 2001.

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

[5] David Stevenson et al. A proposed standard for binary floating point arithmetic. *IEEE Computer*, 14(3):51–62, 1981.

[6] David Stevenson et al. An American national standard: IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, 1987.