When double rounding is odd

Sylvie Boldo, Guillaume Melquiond

When double rounding is odd

Sylvie Boldo and **Guillaume Melquiond**

Laboratoire de l'Informatique du Parallélisme
UMR 5668 CNRS, ENS Lyon, INRIA, UCBL
46 allée d'Italie, 69 364 Lyon Cedex 07, France
E-mail: {Sylvie.Boldo, Guillaume.Melquiond}@ens-lyon.fr

Abstract - Many general purpose processors (including Intel's) may not always produce the correctly rounded result of a floating-point operation due to double rounding. Instead of rounding the value to the working precision, the value is first rounded in an intermediate extended precision and then rounded in the working precision; this often means a loss of accuracy. We suggest the use of rounding to odd as the first rounding in order to regain this accuracy: we prove that the double rounding then gives the correct rounding to the nearest value. To increase the trust on this result, 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.**

I. INTRODUCTION

Floating-point users expect their results 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 [STE 81], [STE 87] for atomic operations. This standard 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 II, 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. Indeed, the processor pipeline has to be flushed in order to set the new precision. If the precision had to be modified before each operation, it would defeat the whole purpose of the pipeline. Computationally intensive programs would then get excessively slow.

As the precision is not explicitly set correspondingly to the destination format, 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 happened. The idea is to prevent the first rounding to approach the tie-breaking value between the two possible floating-point results.

This algorithm only deals with two consecutive roundings. But we have also tried to extend this property for situations where an arithmetic operation is executed between the two roundings. It has led us to the other algorithm we present. This new algorithm allows us to correctly compute the rounding of a sum of floating-point numbers under mild assumptions.

II. DOUBLE ROUNDING

A. Floating-point definitions

Our formal proofs are based on the floating-point formalization of M. Daumas, L. Rideau and L. Théry and on the corresponding library by L. Théry and one of the authors [DAU 01], [BOL 04a] in Coq [BER 04]. 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 a pair 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 referred to; they will simply be denoted as floating-point numbers.

All the IEEE-754 rounding modes are also defined in the Coq library, especially the default rounding: the rounding to nearest even, denoted by $\circ$. We have $f = \circ(x)$ if f is the floating-point number closest to x; when x is half way between two consecutive floating-point numbers, the one with an even mantissa 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.

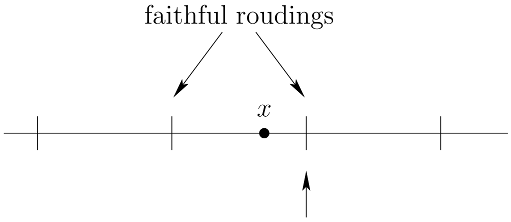

Fig. 1. Faithful roundings.

B. Double rounding accuracy

As explained before, 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 nearest 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 midpoint of two consecutive 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 nearest, 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.

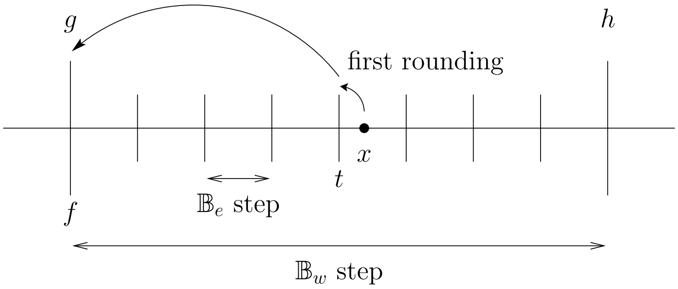

Fig. 2. Bad case for double rounding.

C. 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.

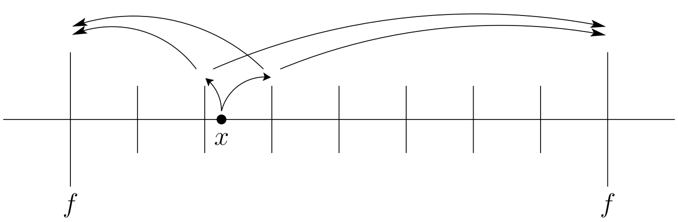

Fig. 3. Double roundings are faithful.

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 is a very powerful result as faithfulness is the best result we can expect as soon as we at least consider two roundings to nearest. And this result can be applied to any two successive IEEE-754 rounding modes (to zero, toward $+\infty\ldots$).

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

III. ALGORITHM

As seen in previous sections, two roundings to nearest induce a bigger round-off error than one single rounding to nearest 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 nearest.

A. 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 nearest odd (similar to the default rounding: rounding to the nearest even).

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

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

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

The first proofs we formally checked guarantee that this operator is a rounding mode as defined in our formalization [DAU 01]. We then added 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.
- 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 $-f = \square_{\text{odd}}(-x)$.

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

B. 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 nearest 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.

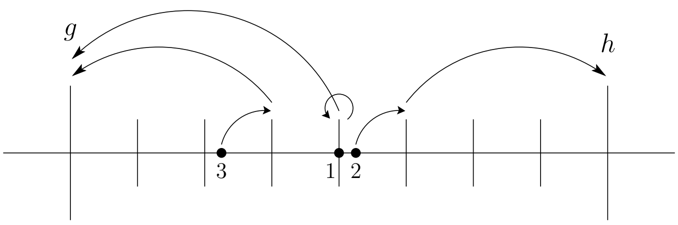

Fig. 4. Different cases of Algorithm 1.

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 $\mathbb{B}_e$.

C. Proof

The pen and paper proof is a bit technical but does seem easy (see Figure 4). But it does not consider the special cases: especially the ones where v was a power of two, and subsequently where v was the smallest normal float. And we must look into all these special cases in order to be sure that the algorithm can always be applied, even when Underflow occurs. We formally proved this result using the Coq proof assistant in order to be sure not to forget any case or hypothesis and not to make mistakes in the numerous computations. The formal proof follows the path described below. There are many splittings into subcases that 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 nearest 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.

IV. APPLICATIONS

A. Rounding to odd is easy

Algorithm 1 has an interest as rounding to odd 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 [GOL 91] 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.

B. 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 nearest operators from the precision $p_3 + 2$ to smaller precisions p_1, p_2 and p_3 as shown in Figure 5. The correctness of Algorithm 1 ensures that the final result in precision p_i will be the correct even nearest rounding of the exact value.

Let us take the DSP processor TMS320C3x as an example [Tex04]. This processor provides floating-point operators. These operators are not IEEE-754 compliant, but the floating-point format is not that different

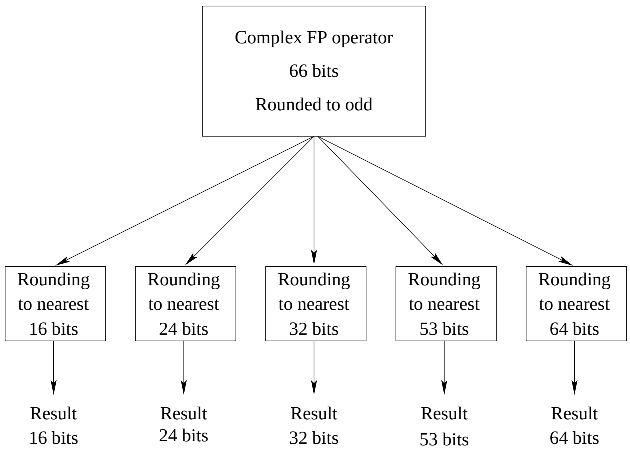

Fig. 5. Multi-precision operator.

from single precision: an exponent stored on 8 bits and a mantissa on 24 bits. Our result is still valid when two's complement is used as the set of represented values is quite the same [BOL 04b].

Source operands of the floating-point multiplication are in this format. The multiplier first produces a 50-bit mantissa, and then dispose of the extra bits so that the mantissa fits on 32 bits. Indeed, the results of the operators are stored in an extended precision format: 32-bit mantissa and 8-bit exponent. The user can then explicitly round this result to single precision, before using it in subsequent operations.

Now, instead of simply discarding the extra bits when producing the extended precision result, the multiplier could round the mantissa to odd. The result would be as precise, the hardware cost would be negligible, and the explicit rounding to simple precision could then be made to guarantee that the final result is correctly rounded.

C. Constants

Rounding to odd can also be used to ensure that a single set of constants can be used for various floating-point formats. For example, to get a constant 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 rounded to odd in double-extended precision and let the processor correctly round it to the required precision. The constant will then be available and correct for both formats.

V. Odd Addition

As another property of the rounding to odd, let us see how it can be used to get the correctly rounded result of the sum of floating-point numbers. we have n floats $f_1, \dots f_n$, sorted by value and we would like to obtain

$$s = \circ\left(\sum_{i=1}^{n} f_i\right).$$

This problem is not new: adding several floating-point numbers with good accuracy is an important problem of scientific computing [HIG 96]. Recently, Demmel and Hida proposed a simple algorithm that yields almost correct summation results [DEM 03]. We present an algorithm that gives the correctly rounded result by using rounding to odd and under mild assumptions. More precisely, we will add the numbers from the smallest to the biggest one (in magnitude) and we use the following algorithm:

Algorithm 2 Correct addition algorithm when $\forall i,\ |f_i| \geq 2|g_{i-1}|$.

Sort the f_i by magnitude
$g_1 = f_1$
For i from 2 to n,
    $g_i = \square_{\text{odd}}^{p+k}(g_{i-1} + f_i)$
$s = \circ^p(g_n)$

Thanks to the correctness of Algorithm 1, we just have to prove $g_n = \square_{\text{odd}}^{p+k}\left(\sum_{i=1}^{n} f_i\right)$.

This property is trivially true for $n = 1$ since f_1 is a floating-point number of precision p. In order to use the induction principle, we just have to guarantee that

$$\square_{\text{odd}}^{p+k}(g_{i-1} + f_i) = \square_{\text{odd}}^{p+k}\left(\sum_{j=1}^{i} f_j\right).$$

Here, let us denote $\square = \square_{\text{odd}}^{p+k}$. Let us consider $x \in \mathbb{R}$ and a floating-point f. It is enough to prove that

$$\square(x + f) = \square(\square(x) + f).$$

Let us assume that $|f| \geq 2|x|$ and let us note $t = \square(x) + f$.

First, if $x = \square(x)$, then the result is trivial. We can now assume that $x \neq \square(x)$, therefore $\square(x)$ is odd.

- If $\square(t)$ is even, then $t = \square(t)$. This case is impossible: as $|f| \geq 2|x|$, we have that $|f| \geq 2|\square(x)|$. Therefore, $t = \square(x) + f$ is representable implies that $\square(x)$

is even as f has p bits and is twice bigger than $\square(x)$. But $\square(x)$ cannot be even by the previous assumption.

• Let us assume that $\square(t)$ is odd. This means that $\square(t)$ cannot be a power of the radix. Then with the current hypotheses on $\square(t)$, if we prove the inequality $|x+f-\square(t)| < \mathrm{ulp}(\square(t))$, it implies the wanted result: that $\square(x+f) = \square(\square(x)+f)$.

And $|x+f-\square(t)| \leq |x+f-t|+|t-\square(t)| = |x-\square(x)|+|t-\square(t)|$. We know that $|x-\square(x)| < \mathrm{ulp}(\square(x))$. Let us look at $|t-\square(t)|$. We of course know that $|t-\square(t)| < \mathrm{ulp}(\square(t))$, but we also know that $t-\square(t)$ can exactly be represented by an unbounded float with the exponent $e = \min(e_{\square(x)}, e_f, e_{\square(t)})$.

Therefore $|t-\square(t)| \leq \mathrm{ulp}(\square(t))-2^e$ and $|x+f-\square(t)| < \mathrm{ulp}(\square(t)) + \mathrm{ulp}(\square(x)) - 2^e$. Due to the definition of e, we have only left to prove that $e_{\square(x)} \leq e_f$ and that $e_{\square(x)} \leq e_{\square(t)}$. We have assumed that $|f| \geq 2|x|$, so we have that $|f| \geq 2|\square(x)|$ and so $e_{\square(x)} \leq e_f$. Moreover $|t| = |\square(x)+f| \geq |f| - |\square(x)| \geq |\square(x)|$, so $e_{\square(x)} \leq e_{\square(t)}$ that ends the proof.

Under the assumption that $\forall i,\ |f_i| \geq 2|g_{i-1}|$, we have a linear algorithm to exactly add any set of floating-point numbers. This hypothesis is not easy to check: it can be replaced by $\forall i,\ |f_i| \geq 3|f_{i-1}|$.

The proofs of this section are not yet formally proved. The authors do believe in their correctness and in their soon formal translation. Note that, as for Algorithm 1, there will be an hypothesis linking the exponent ranges of the working and extended precision. This is needed to guarantee that $|f| \geq 2|x|$ implies $|f| \geq 2|\square_{\mathrm{odd}}^{p+k}(x)|$, for f in working precision.

VI. CONCLUSION

The first 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.

The second algorithm is also very simple: it simply adds floating-point number in an extended precision with rounding to odd. We give some hypotheses enough to ensure that the final number once rounded to the working precision is the correctly rounded value of the sum of these numbers. Although such a method works well with the addition, it cannot be extended to the fused multiply-and-add operation: $\mathrm{fma}(a,x,y) = \circ(a \cdot x + y)$. As a consequence, it cannot be directly applied to obtain a correctly rounded result of Horner's polynomial evaluation.

The two algorithms are even more general than what we presented here. They are not limited to a final result rounded to nearest even, they can also be applied to any other realistic rounding (meaning that the result of a computation is uniquely defined by the exact value of the real result and does not depend on the machine state). In particular, they handle all the IEEE-754 standard roundings and the new rounding to the nearest away from zero defined by the revision of this standard.

REFERENCES

[BER 04] Bertot Y., Casteran P., *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions*, Texts in Theoretical Computer Science, Springer Verlag, 2004.

[BOL 04a] Boldo S., Preuves formelles en arithmétiques à virgule flottante, PhD thesis, Ecole Normale Supérieure de Lyon, November 2004.

[BOL 04b] Boldo S., Daumas M., *Properties of two's complement floating point notations*, *International Journal on Software Tools for Technology Transfer*, vol. 5, n2-3, p. 237-246, 2004.

[DAU 01] Daumas M., Rideau L., Théry L., A generic library of floating-point numbers and its application to exact computing, *14th International Conference on Theorem Proving in Higher Order Logics*, Edinburgh, Scotland, p. 169-184, 2001.

[DEM 03] Demmel J. W., Hida Y., Fast and accurate floating point summation with applications to computational geometry, *Proceedings of the 10th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN 2002)*, January 2003.

[GOL 91] Goldberg D., *What every computer scientist should know about floating point arithmetic*, *ACM Computing Surveys*, vol. 23, n1, p. 5-47, 1991.

[HIG 96] Higham N. J., *Accuracy and stability of numerical algorithms*, SIAM, 1996.

[STE 81] Stevenson D. et al., *A proposed standard for binary floating point arithmetic*, *IEEE Computer*, vol. 14, n3, p. 51-62, 1981.

[STE 87] Stevenson D. et al., *An American national standard: IEEE standard for binary floating point arithmetic*, *ACM SIGPLAN Notices*, vol. 22, n2, p. 9-25, 1987.

[Tex04] Texas Instruments, TMS320C3x — User's guide, 2004.