View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by INRIA a CCSD electronic archive server

Computing a correct and tight rounding error bound using rounding-to-nearest

Sylvie Boldo

To cite this version:

Sylvie Boldo. Computing a correct and tight rounding error bound using rounding-to-nearest. 9th International Workshop on Numerical Software Verification, Jul 2016, Toronto, Canada. hal-01377152

HAL Id: hal-01377152
https://hal.inria.fr/hal-01377152

Submitted on 6 Oct 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Computing a correct and tight rounding error bound using rounding-to-nearest

Sylvie Boldo*

Inria, Université Paris-Saclay, F-91893 Palaiseau
LRI, CNRS & Univ. Paris-Sud, F-91405 Orsay
Email: sylvie.boldo@inria.fr

Abstract. When a floating-point computation is done, it is most of the time incorrect. The rounding error can be bounded by folklore formulas, such as $\varepsilon|x|$ or $\varepsilon|\circ(x)|$. This gets more complicated when underflow is taken into account as an absolute term must be considered. Now, let us compute this error bound in practice. A common method is to use a directed rounding in order to be sure to get an over-approximation of this error bound. This article describes an algorithm that computes a correct bound using only rounding to nearest, therefore without requiring a costly change of the rounding mode. This is formally proved using the Coq formal proof assistant to increase the trust in this algorithm.

1 Introduction

Floating-point (FP) arithmetic is the way computers deal with computations on real numbers. It is clearly defined [5,6], but it is not perfect. In particular, FP numbers have a finite precision, therefore even a single computation may be incorrect when the result does not fit in a FP number. This error is called a rounding error, and a part of the computer arithmetic literature tries to improve or bound these errors on some given algorithms.

The most common model for bounding this rounding error is the standard model [4] where

$$\circ(x) = x(1+\delta) \qquad \text{with} \quad |\delta| \leq u$$

with $\circ$ being the rounding to nearest and u being the machine epsilon, therefore 2^{-p} when p is the number of bits of the FP mantissa.

This model has two main drawbacks. The first one is that it does not take underflow into account. For example in `binary64`, if you round 3×2^{-1075}, you get the FP number 2×2^{-1074}, which gives a huge relative error of 33 % compared to the input, but a small absolute error.

The second drawback is when this error bound needs to be computed. That is the case for example when considering midpoint-radius interval arithmetic [7]. Indeed, computing $\circ\,(2^{-p} \times \circ(x))$ may not be an overestimation of the error

* This work was supported by the FastRelax (ANR-14-CE25-0018-01) project of the French National Agency for Research (ANR).

bound. For example, let us consider $x = \frac{2^{-1022}+2^{-1074}}{2}$. Then $\circ(x) = 2^{-1023}$ and the error is 2^{-1075}. But $\circ\left(2^{-p} \times \circ(x)\right) = \circ\left(2^{-53} \times 2^{-1023}\right) = \circ\left(2^{-1076}\right) = 0$, which is *not* an overestimation of 2^{-1075}.

This article aims at providing a correct algorithm that computes a bound on the error of a FP operation. As we also want this algorithm to be fast, we wish to avoid changing the rounding mode, as it breaks the pipeline. We will therefore only consider rounding to nearest, both for the operation considered, and for our algorithm. Moreover, we want to prevent tests as they also break the pipeline.

More than a pen-and-paper proof, this work gives a high guarantee of its correctness and gives precise hypotheses on the needed precision and underflow threshold. We will rely on the Coq proof assistant. From the formal methods point of view, we will base our proof on the Flocq library [2]. Flocq is a formalization in Coq that offers a multi-radix and multi-precision formalization for various floating- and fixed-point formats (including FP with or without gradual underflow) with a comprehensive library of theorems. Its usability and practicality have been established against test-cases [1]. The corresponding Coq file is named `Error_bound_fp.v` and is available in the `example` directory of Flocq[1], available in the current git version, and in the next released versions > 2.5.1.

Notations We denote by $\circ$ the rounding to nearest, ties to even in radix 2. We denote by $\circ$[expr] the rounding of the expression into brackets, where all the operations are considered to be rounded operations. For example, $\circ[3 \times x + y]$ denotes $\circ(\circ(3 \times x) + y)$. The smallest subnormal number is denoted by 2^{E_i} and the number of bits of the mantissa is $p > 0$. For basic knowledge about FP arithmetic (roundings, subnormal, error bounds), we refer the reader to [3,6].

2 Theorem

Theorem 1. *Let x be a real number. Assume that $E_i \leq -p$. Then*

$$|\circ(x) - x| \leq \circ\left[2^{-p} \times |\circ(x)| + 2^{E_i}\right].$$

The assumption is very light: $E_i \leq -p$ only means that 2^{-p} is in the format. It holds in all IEEE formats: for example in `binary64`, $p = 53$ and $E_i = -1074$ and in `binary32`, $p = 24$ and $E_i = -149$.

[1] `http://flocq.gforge.inria.fr/`.

Proof The first step is to prove that 2^{-p} and 2^{E_i} are in the format, therefore not rounded. This is trivial as long as $E_i \leq -p$.

Then, the error bound we are used to is $|\circ(x) - x| \leq 2^{-p} \times |\circ(x)| + 2^{E_i-1}$ or $\max\left(2^{-p} \times |\circ(x)|, 2^{E_i-1}\right)$. The first formula looks like our the theorem, rounding excepted. The other difference is the 2^{E_i} instead of 2^{E_i-1}. Let us prove that the roundings do not endanger the result.

Let $t = \circ\left[2^{-p} \times |\circ(x)| + 2^{E_i}\right]$. Then, we split the proof into 4 cases, depending on the value of $|x|$.

1. Assume $x = 0$. Then $\circ(x) = 0$, and $|\circ(x) - x| = 0$, while $t = \circ\left(0 + 2^{E_i}\right) = 2^{E_i}$. So the result holds.
2. Assume $0 < |x| < 2^{E_i+p}$. Then x has exponent E_i and $|\circ(x) - x| \leq \frac{1}{2}\text{ulp}(x) = 2^{E_i-1}$. Moreover, $t = \circ\left[2^{-p} \times |\circ(x)| + 2^{E_i}\right] \geq \circ\left(2^{E_i}\right) = 2^{E_i}$. So the result holds.
3. Assume $2^{E_i+2p-1} \leq |x|$. Then $2^{-p} \times |\circ(x)| \geq 2^{E_i+p-1}$ and is normal. Therefore, the multiplication by 2^{-p} is correct and does not create any rounding error. Then $|\circ(x) - x| \leq 2^{-p} \times |\circ(x)| = \circ\left[2^{-p} \times |\circ(x)|\right] \leq \circ\left[2^{-p} \times |\circ(x)| + 2^{E_i}\right]$ by monotony of the rounding.
4. Assume $2^{E_i+p} \leq |x| < 2^{E_i+2p-1}$. This is the most complex case, as all roundings may go wrong. First $|\circ(x) - x| \leq 2^{-p} \times |\circ(x)|$. Let y be $2^{-p} \times |\circ(x)|$. Then $y < 2^{E_i+p-1}$ and is therefore in the subnormal range. As $|\circ(x) - x| \leq y$, what is left to prove is that $y \leq \circ\left(\circ(y) + 2^{E_i}\right)$. As y is small, $\circ(y)$ is a positive FP number in the subnormal range, therefore $\circ(y) + 2^{E_i}$ is also in the FP format and $\circ\left(\circ(y) + 2^{E_i}\right) = \circ(y) + 2^{E_i}$. What is left to prove is then $y \leq \circ(y) + 2^{E_i}$. Finally, as y is in the subnormal range, $|\circ(y) - y| \leq 2^{E_i-1}$. So $y \leq \circ(y) + 2^{E_i-1}$ and the result holds.

□

This pen-and-paper proof exactly corresponds to the Coq proof (as it was written from it). The itemized cases corresponding to the interval values of x become several lemmas in the Coq proof for the sake of readability.

3 Tightness of the bound

The next question is how tight is the proved bound. In particular, is it much coarser than the usual bound? The answer is no and a graphical explanation is given in Figure 3.

When $|x|$ is small, then the optimal bound is 2^{E_i-1}. As this bound is not a FP number, our algorithm returns the best possible bound, that is to say 2^{E_i}. When $|x|$ is big enough, meaning greater than 2^{E_i+2p} (that is 2^{-968} in `binary64`), we have the optimal bound, meaning $2^{-p} \times |\circ(x)|$. In this case, the 2^{E_i} is indeed negligible and is neglected as we use rounding to nearest (and not rounding towards $+\infty$). In the middle, meaning between 2^{E_i+p-1} and 2^{E_i+2p}, we have a slight overestimation compared to the usual bound. Note that this overestimation is bounded by 2^{E_i}.

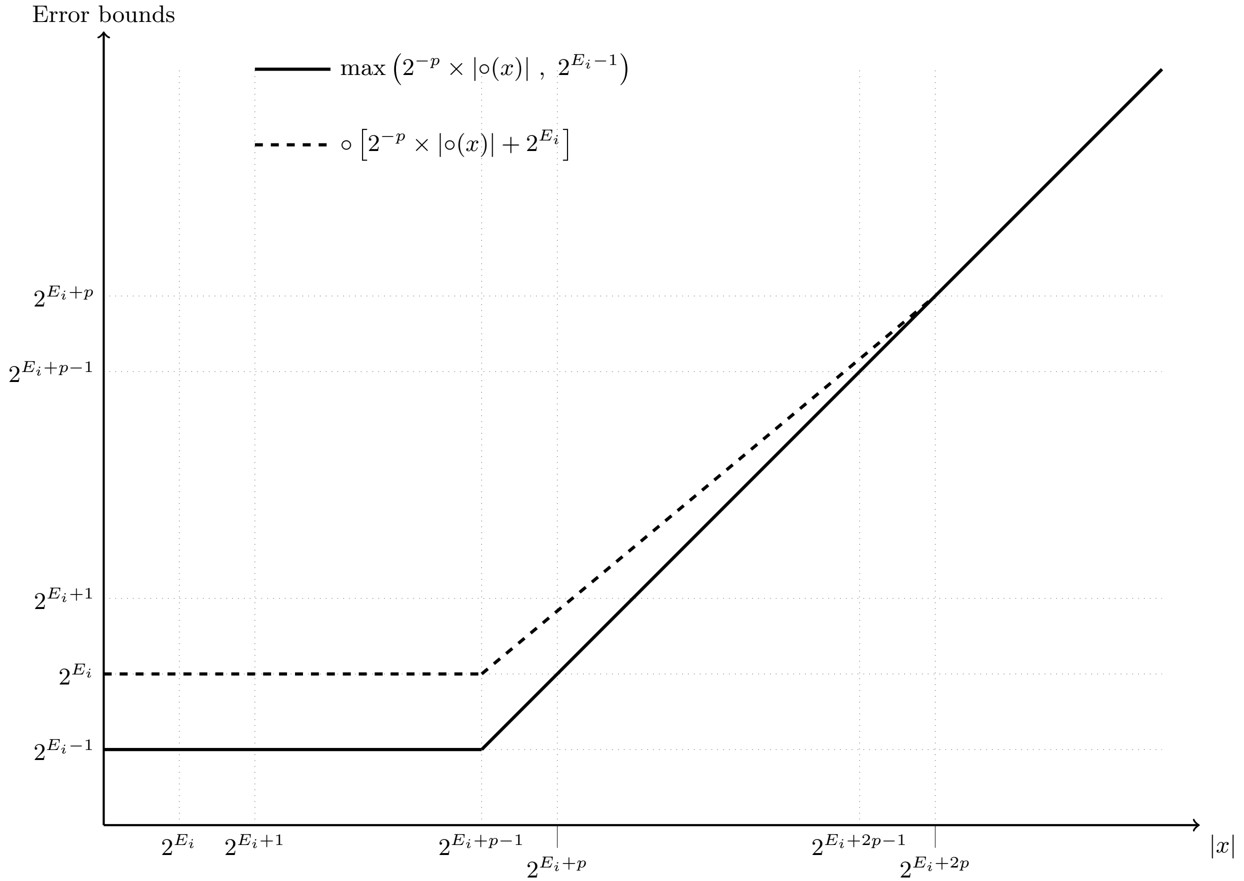

Fig. 1. Drawings comparing the error bounds of $\circ(x)$

4 Conclusion

We have formally proven that the following algorithm using only the rounding-to nearest mode:

```
fabs(x)*0x1.p-53+0x1.p-1074
```

gives a correct tight bound on the rounding error of x in rounding-to-nearest `binary64`. Note that overflow cannot happen here as all values involved are strictly smaller than the input x.

As for efficiency, random tests have shown it is quite efficient, as it involves neither tests, nor rounding change (2 flops plus the memory accesses). Nevertheless, on some architectures, subnormal numbers are trapped and handled in software, and are therefore much slower than normal FP operations. In this case, computing `x*0x1.p-53+0x1.p-1022` might be a better idea. Indeed, the previ-

ous theorem implies that $\circ\left[2^{-p} \times |\circ(x)| + 2^{E_i+p-1}\right]$ is also an overestimation of the rounding error. And as 2^{E_i+p-1} is the smallest normal number, this algorithm does not involve any subnormal number if x is not one and will be faster in most cases, at the price of a worse bound. A use of the processor max function may also prevent the use of operation on subnormal numbers, which is known to be quite costly.

A perspective is to be able to compute an error bound on a given rounding, using only this given rounding. The formula will probably need to be modified, for example suppressing the addition when rounding towards $+\infty$, negating twice the values when using rounding towards $-\infty$. But this needs to be worked out in depth and formally proved to get correct algorithms.

A harder perspective is to deal with radix 10. Then the multiplication should not be by 2^{-p}, but by 5×10^{-p}, and this multiplication is not always exact with big numbers, as was the case here with radix 2. The provided algorithm does therefore not hold in radix 10.

References

1. Sylvie Boldo. *Deductive Formal Verification: How To Make Your Floating-Point Programs Behave.* Thèse d'habilitation, Université Paris-Sud, October 2014.
2. Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In Elisardo Antelo, David Hough, and Paolo Ienne, editors, *20th IEEE Symposium on Computer Arithmetic*, pages 243–252, Tübingen, Germany, 2011.
3. David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.
4. Nicholas J. Higham. *Accuracy and stability of numerical algorithms.* SIAM, 2002. Second edition.
5. Microprocessor Standards Committee. IEEE Standard for Floating-Point Arithmetic. *IEEE Std. 754-2008*, pages 1–58, August 2008.
6. Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic.* Birkhäuser, 2010.
7. Siegfried M. Rump. Fast and parallel interval arithmetic. *BIT Numerical Mathematics*, 39(3):534–554, 1999.