



**HAL**  
open science

## How do you compute the midpoint of an interval?

Frédéric Goualard

► **To cite this version:**

Frédéric Goualard. How do you compute the midpoint of an interval?. ACM Transactions on Mathematical Software, 2014, 40 (2), 10.1145/2493882 . hal-00576641v1

**HAL Id: hal-00576641**

**<https://hal.science/hal-00576641v1>**

Submitted on 15 Mar 2011 (v1), last revised 17 Apr 2014 (v2)

**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.

# How do you compute the midpoint of an interval?

Frédéric Goualard

Université de Nantes  
Nantes Atlantique Université  
CNRS, LINA, UMR 6241  
`Frederic.Goualard@univ-nantes.fr`

**Abstract.** The algorithm that computes the midpoint of an interval with floating-point bounds requires some careful devising to correctly handle all possible inputs. We review several implementations from prominent C/C++ interval arithmetic packages and analyze their potential failure to deliver correct results. We then highlight two implementations that avoid common pitfalls. The results presented are also relevant to non-interval arithmetic computation such as the implementation of bisection methods. Enough background on IEEE 754 floating-point arithmetic is provided for this paper to serve as a practical introduction to the analysis of floating-point computation.

## 1 Introduction

In his 1966 report [3] “*How do you solve a quadratic equation?*”, Forsythe considers the seemingly simple problem of reliably solving a quadratic equation on a computer using floating-point arithmetic. Forsythe’s goal is both to warn a large audience off of unstable classical textbook formulae as well as get them acquainted with the characteristics of pre-IEEE 754 standard floating-point arithmetic [11], a dual objective shared by his later paper “*Pitfalls in Computation, or Why a Math Book isn’t Enough*” [4].

Following Forsythe’s track, we consider here the problem of computing a good approximation to the midpoint between two floating-point numbers. We strive to provide both a reliable algorithm for midpoint computation and an introduction to floating-point computation according to the IEEE 754 standard.

Given a non-empty interval  $I = [a, b]$ , how hard can it be to compute its midpoint  $m(I)$ ? Over reals, the following simple formula does the job:

$$m([a, b]) = \frac{a + b}{2} \tag{1}$$

For  $a$  and  $b$  two floating-point numbers from a set  $\mathbb{F}$  (See Section 2), the sum  $a + b$  may not be a floating-point number itself, and we therefore have to take

care of rounding it correctly to ensure that our floating-point implementation of  $\mathbf{m}(\mathbf{I})$  does not violate the two following properties:

$$\mathbf{m}(\mathbf{I}) \in \mathbf{I} \tag{2}$$

$$\forall v \in \mathbb{F}: \quad |c - v| \geq |c - \mathbf{m}(\mathbf{I})|, \quad \text{with } c = \frac{a + b}{2}, c \in \mathbb{R} \tag{3}$$

Equation (2) and Eq. (3) both state reasonable facts, *viz.*, that the floating-point midpoint of an interval  $\mathbf{I}$  should belong to  $\mathbf{I}$ , and that it should be the floating-point number closest to the real midpoint of  $\mathbf{I}$ <sup>1</sup>. For non-empty intervals with finite bounds, Eq. (3) trivially entails Eq. (2). However, some implementations may choose to only ensure the latter and a relaxation of the former.

The midpoint operator is a staple of interval arithmetic [16,1,9] libraries. It is also intensively used in many numerical methods such as root-finding algorithms with bisection. It is, therefore, paramount that its floating-point implementation at least verifies Eq. (2). Accuracy, as stipulated by Eq. (3), is also desirable. Nevertheless, we will see in Section 3 that most formulae implemented in popular C/C++ interval libraries may not ensure even the containment requirement for some inputs. In Section 3, we analyze the various formulae both theoretically and practically; contrary to most expositions, we consider the impact of both overflow and underflow on the accuracy and correction of the formulae.

The error analysis conducted in this paper requires slightly more than a simple working knowledge of floating-point arithmetic as defined by the IEEE 754 standard. As a consequence, the basic facts on floating-point arithmetic required in Section 3 are presented in Section 2 for the sake of self-contentedness.

It turns out that the study of the correct implementation of a floating-point midpoint operator may serve as a nice introduction to many important aspects of floating-point computation at large: the formulae studied are simple enough for their analysis to be easily understandable, while the set of problems raised is sufficiently broad in scope to be of general interest. We then hope that this paper will be valuable as both a technical presentation of reliable, accurate, and fast methods to compute a midpoint as well as an introduction to error analysis of floating-point formulae.

## 2 Floating-point Arithmetic in a Nutshell

According to the IEEE 754 standard [11], a floating-point number  $\varphi$  is represented by a sign bit  $s$ , a significand  $m$  (where  $m$  is a bit chain of the form “0. $f$ ” or “1. $f$ ”, with  $f$  the *fractional part*) and an integral exponent  $E$ :

$$\varphi = (-1)^s \times m \times 2^E \tag{4}$$

The IEEE 754 standard defines several formats varying in the number of bits  $l(f)$  and  $l(E)$  allotted to the representation of  $f$  and  $E$ , the most promi-

<sup>1</sup> Note that if  $c$  is right between two floating-point numbers, there exist two possible values for  $\mathbf{m}(\mathbf{I})$  that verify Eq. (3).

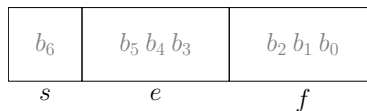
ment ones being *single precision*— $(l(E), l(f)) = (8, 23)$ —and *double precision*— $(l(E), l(f)) = (11, 52)$ . We will also use for pedagogical purposes an *ad hoc* IEEE 754 standard-compliant *tiny* precision format— $(l(E), l(f)) = (3, 3)$ .

Wherever possible, the significand must be of the form “1.*f*” since it is the form that stores the largest number of significant figures for a given size of *m*:

$$\begin{aligned}\varphi &= \mathbf{0.01101} \times 2^0 \\ &= \mathbf{0.1101} \times 2^{-1} \\ &= \mathbf{1.101} \times 2^{-2}\end{aligned}$$

Floating-point numbers with such a significand are called *normal numbers*. Such prevalence is given to normal numbers that the leading “1” is left implicit in the representation of an IEEE 754 number and only the fractional part *f* is stored (See Fig. 1).

The exponent *E* is a signed integer stored as a biased exponent  $e = E + \text{bias}$ , with  $\text{bias} = 2^{l(E)-1} - 1$ . The biased exponent *e* is a positive integer that ranges from  $e_{\min} = 0$  to  $e_{\max} = 2^{l(E)} - 1$ . However, for the representation of normal numbers, *E* only ranges from  $E_{\min} = (e_{\min} - \text{bias}) + 1$  to  $E_{\max} = (e_{\max} - \text{bias}) - 1$  because the smallest and largest values are reserved for special purposes (see below). As an example of what precedes, the bias for the *tiny* format is equal to 3, *e* ranges from 0 to 7, and *E* ranges from  $-2$  to  $+3$ .



**Fig. 1.** Binary representation as a seven bits chain of a *tiny* floating-point number.

Evidently, it is not possible to represent 0 as a normal number. Additionally, consider the number  $\varphi$ :

$$\varphi = 0.00011 \times 2^0$$

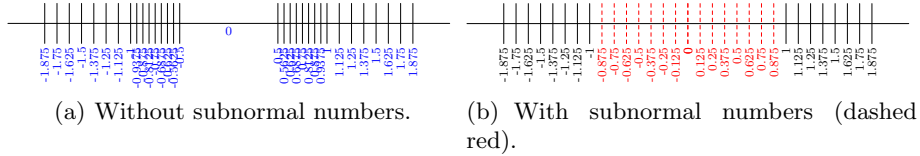
To store it as a *tiny* floating-point normal number requires shifting the leftmost “1” of the fractional part to the left of the radix point:

$$\varphi = 1.100 \times 2^{-4}$$

However, doing so requires an exponent smaller than  $E_{\min}$ . It is nevertheless possible to represent  $\varphi$ , provided we accept to store it with a “0” to the left of the radix point:

$$\varphi = 0.011 \times 2^{-2}$$

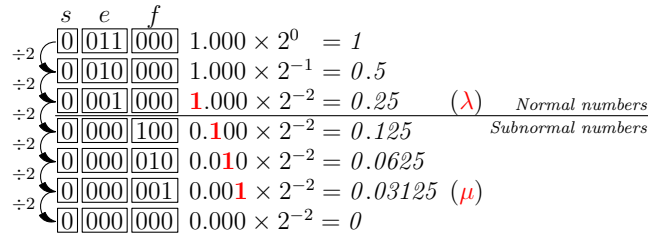
Numbers with a significand of the form “0.*f*” are called *subnormal numbers*. Their introduction is necessary in order to represent 0 and to avoid the large gap that would otherwise occur around 0 (Compare Fig. 2(a) and Fig. 2(b)).



**Fig. 2.** The *tiny* floating-point format with and without subnormals (focus on 0)

To signal that a number is a subnormal number, the biased exponent  $e$  stored is set to the reserved value 0, even though the unbiased exponent  $E$  is  $E_{\min}$  (otherwise, it would not be possible to distinguish between normal and subnormal numbers whose unbiased exponent are  $E_{\min}$ ). Exceptional steps must be taken to handle subnormal numbers correctly since their leading bit is “0”, not “1”. This has far reaching consequences in terms of performances, as we will see at the end of Section 3.

Fig. 3 shows the repeated division by two of a *tiny* number across the normal/subnormal divide. As seen in that figure, dividing an IEEE 754 floating-point number by two is an error-free operation only if the result is not a subnormal number. Otherwise, the rightmost bit of the fractional part may be shifted out and lost. The values  $\lambda$  and  $\mu$  are, respectively, the *smallest positive normal* and the *smallest positive subnormal* floating-point numbers.



**Fig. 3.** Repeated division by two from 1.0 to 0.0 in the *tiny* format

The condition whereby an operation leads to the production of a subnormal number is called *underflow*. On the other side of the spectrum, an operation may produce a number that is too large to be represented in the floating-point format on hand, leading to an *overflow* condition. In that case, the number is replaced by an infinity of the right sign, which is a special floating-point number whose biased exponent  $e$  is set to  $e_{\max}$  and whose fractional part is set to zero. In the rest of this paper, we note  $\mathbb{F}$  the set of normal and subnormal floating-point numbers and  $\overline{\mathbb{F}} = \mathbb{F} \cup \{-\infty, +\infty\}$  its affine extension, the size of the underlying

format (*tiny* or *double* precision, mainly) being unambiguously drawn from the context.

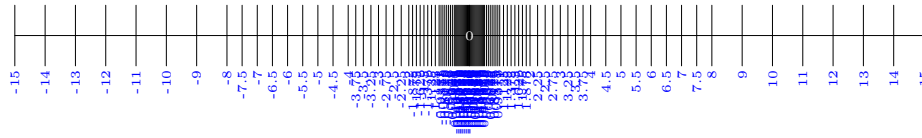
To ensure non-stop computation even in face of a meaningless operation, the IEEE 754 standard defines the outcome of all operations, the undefined ones generating an NaN (*Not a Number*), which is a floating-point number whose biased exponent is set to  $e_{\max}$  and whose fractional part is any value different from zero. We will have, *e.g.*:

$$\sqrt{-1} = \text{NaN}, \quad \infty - \infty = \text{NaN}$$

The NaNs are supposed to be unordered, and any test in which they take part is false<sup>2</sup>. As a consequence, the right way to test whether  $a$  is greater than  $b$  is to check whether  $\neg(a \leq b)$  is true since that form returns a correct result even if either  $a$  or  $b$  is an NaN.

To sum up, the interpretation of a chain bit representing a floating-point number depends on the value of  $e$  and  $f$  as follows:

$$\begin{cases} e = 0, & f = 0: & \varphi = (-1)^s \times 0.0 \\ e = 0, & f \neq 0: & \varphi = (-1)^s \times (0.f) \times 2^{1-\text{bias}} \\ 0 < e < e_{\max} & : & \varphi = (-1)^s \times (1.f) \times 2^{e-\text{bias}} \\ e = e_{\max} & f = 0: & \varphi = (-1)^s \times \infty \\ e = e_{\max} & f \neq 0: & \varphi = \text{NaN} \end{cases}$$



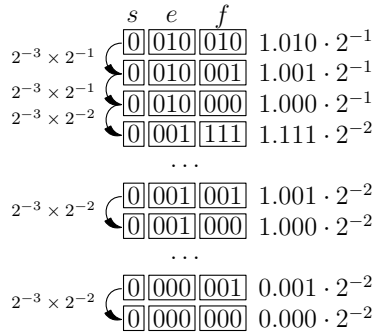
**Fig. 4.** IEEE 754 *tiny* floating-point (normal and subnormal) numbers.

Figure 4 presents the repartition of all *tiny* numbers from  $\mathbb{F}$  on the real line. The farther from 0, the larger the gap from a floating-point number to the next. More specifically, as Fig. 5 shows, the difference between two consecutive floating-point numbers doubles every  $2^{l(e)}$  numbers in the normal range, while it is constant throughout the whole subnormal range.

Consider the binary number  $\rho = 1.0011$ . It cannot be represented as a *tiny* floating-point number since its fractionnal part has four bits and the *tiny* format has room for only three. It therefore has to be rounded to the floating-point number  $\text{fl}\langle\rho\rangle$  according to one of four rounding directions<sup>3</sup> (see Fig. 6):

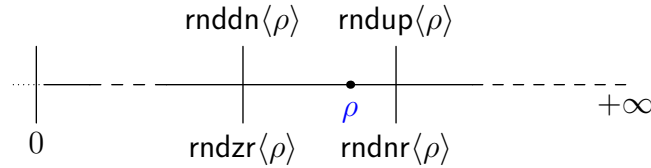
<sup>2</sup> Surprisingly enough to the unsuspecting, even the equality test turns false for an NaN, so much so that the statement  $x \neq x$  is an easy way to check whether  $x$  is one.

<sup>3</sup> The actual direction chosen may depend on the settings of the Floating-Point Unit at the time, or alternatively, on the machine instruction used, for some architecture.



**Fig. 5.** Gap between consecutive floating-point numbers in the *tiny* format.

- Rounding towards 0:  $\text{fl}\langle\rho\rangle = \text{rndzr}\langle\rho\rangle$ ;
- Rounding to nearest-even:  $\text{fl}\langle\rho\rangle = \text{rndnr}\langle\rho\rangle$ ;
- Rounding towards  $-\infty$ :  $\text{fl}\langle\rho\rangle = \text{rnddn}\langle\rho\rangle$ ;
- Rounding towards  $+\infty$ :  $\text{fl}\langle\rho\rangle = \text{rndup}\langle\rho\rangle$ .



**Fig. 6.** Rounding a real number according to the IEEE 754 standard

Note the use of angles “ $\langle \rangle$ ” instead of more conventional parentheses for the rounding operators. They are used to express the fact that each individual value and/or operation composing the expression in-between is individually rounded according to the leading operator. For example:

$$\text{fl}\langle\rho_1 + \rho_2\rangle \equiv \text{fl}(\text{fl}\langle\rho_1\rangle + \text{fl}\langle\rho_2\rangle), \quad \forall(\rho_1, \rho_2) \in \mathbb{R}^2$$

When rounding to the nearest, if  $\rho$  is equidistant from two consecutive floating-point numbers, it is rounded to the one whose rightmost bit of the fractional part is zero (the “even” number).

Rounding is order-preserving (*monotonicity of rounding*):

$$\forall(\rho_1, \rho_2) \in \mathbb{R}^2: \rho_1 \leq \rho_2 \implies \text{fl}\langle\rho_1\rangle \leq \text{fl}\langle\rho_2\rangle$$

for the same instantiation of  $\text{fl}\langle \rangle$  in both occurrences. The IEEE 754 standard also mandates that real constants and the result of some operations (addition, subtraction, multiplication, and division, among others) be *correctly rounded*:

depending on the current rounding direction, the floating-point number used to represent a real value must be one of the two floating-point numbers surrounding it. Disregarding overflow—for which error analysis is not very useful—, a simple error model may be derived from this property: let  $\rho$  be a positive real number<sup>4</sup>,  $\varphi_l$  the largest floating-point number smaller or equal to  $\rho$  and  $\varphi_r$  the smallest floating-point number greater or equal to  $\rho$ . Let also  $\text{rndnr}\langle\rho\rangle$  be the correctly rounded representation to the nearest-even of  $\rho$  by a floating-point number ( $\text{rndnr}\langle\rho\rangle$  is either  $\varphi_l$  or  $\varphi_r$ ). Obviously, we have:

$$|\text{rndnr}\langle\rho\rangle - \rho| \leq \frac{\varphi_r - \varphi_l}{2} \quad (5)$$

We may consider two cases depending on whether  $\varphi_l$  is normal or not:

**Case  $\varphi_l$  normal.** The real  $\rho$  may be expressed as  $m_\rho \times 2^E$ , with  $1 \leq m_\rho < 2$ . Then,  $\varphi_l$  can be put into the form  $m_{\varphi_l} \times 2^E$ , with  $1 \leq m_{\varphi_l} < 2$ . If we call  $\varepsilon_M$  the *machine epsilon* corresponding to the value  $2^{-l(f)}$  of the last bit in the fractional part<sup>5</sup> of a floating-point number, we have  $\varphi_r = \varphi_l + \varepsilon_M \times 2^E$ . From Eq. (5), we get:

$$|\text{rndnr}\langle\rho\rangle - \rho| \leq \frac{\varepsilon_M}{2} \times 2^E$$

We have (for  $\rho \neq 0$ ):

$$\frac{|\text{rndnr}\langle\rho\rangle - \rho|}{\rho} \leq \frac{(\varepsilon_M/2) \times 2^E}{m_\rho \times 2^E}$$

We know  $1 \leq m_\rho < 2$ . Then:

$$|\text{rndnr}\langle\rho\rangle - \rho| \leq \frac{\varepsilon_M}{2} \rho$$

or, alternatively:

$$\text{rndnr}\langle\rho\rangle = \rho(1 + \delta), \quad \text{with } |\delta| \leq \frac{\varepsilon_M}{2} \quad (6)$$

**Case  $\varphi_l$  subnormal.** The distance between any subnormal number to the next floating-point number is constant and equal to  $\mu = \varepsilon_M \times 2^{E_{\min}}$ . From Eq. (5), we then have:

$$|\text{rndnr}\langle\rho\rangle - \rho| \leq \frac{\mu}{2}$$

which may be expressed as:

$$\text{rndnr}\langle\rho\rangle = \rho + \eta, \quad \text{with } |\eta| \leq \frac{\mu}{2} \quad (7)$$

<sup>4</sup> The negative case may be handled analogously.

<sup>5</sup> Alternatively,  $\varepsilon_M$  may be defined as the difference between 1.0 and the next larger floating-point number.



We may unify Eq. (6) and Eq. (7) as:

$$\text{rndnr}\langle\rho\rangle = \rho(1 + \delta) + \eta, \quad \text{with} \quad \begin{cases} \delta\eta = 0, \\ |\delta| \leq \frac{\varepsilon_M}{2} \\ |\eta| \leq \frac{\mu}{2} \end{cases} \quad (8)$$

where either one of  $\delta$  and  $\eta$  has to be null ( $\delta = 0$  in case of underflow, and  $\eta = 0$  otherwise). This error model is valid for all correctly rounded operations and then:

$$\forall(\varphi_1, \varphi_2) \in \mathbb{F}^2, \forall \top \in \{+, -, \times, \div\}: \\ \text{rndnr}\langle\varphi_1 \top \varphi_2\rangle = (\varphi_1 \top \varphi_2)(1 + \delta) + \eta, \quad \text{with} \quad \begin{cases} \delta\eta = 0, \\ |\delta| \leq \frac{\varepsilon_M}{2} \\ |\eta| \leq \frac{\mu}{2} \end{cases} \quad (9)$$

For the other rounding directions, the same error model can be used, except that the bounds on  $\delta$  and  $\eta$  are now larger:

$$|\delta| < \varepsilon_M \quad \text{and} \quad |\eta| < \mu$$

In case of overflow, a real may be represented either by  $\pm\infty$  or by  $\pm\mathbf{realmax}$  depending on the rounding direction used, where  $\mathbf{realmax}$  is the largest finite positive floating-point number.

**Table 1.** Characteristics of floating-point formats

Format	$l(E)$	$l(f)$	$E_{\min}$	$E_{\max}$	$\varepsilon_M^\dagger$	$\lambda^*$	$\mu^\diamond$
<i>tiny</i>	3	3	-2	+3	$2^{-3}$	$2^{-2}$	$2^{-5}$
<i>single</i>	8	23	-126	+127	$2^{-23}$	$2^{-126}$	$2^{-149}$
<i>double</i>	11	52	-1022	+1023	$2^{-52}$	$2^{-1022}$	$2^{-1074}$

†) *machine epsilon*, equal to  $2^{-l(f)}$

\*) *realmin* (smallest positive normal number), equal to  $2^{E_{\min}}$

◇) *subrealmin* (smallest positive subnormal number), equal to  $\varepsilon_M \lambda$

Table 1 summarizes the values of the various constants encountered for the *tiny*, *single*, and *double* precision floating-point formats.

The error model presented in Eq. (8) and Eq. (9) is pessimistic by nature as it only gives an upper bound of the error. In some instances, it is possible to refine it to more accurate bounds. For example, we have for any finite floating-point number  $\varphi$ :

$$\text{rndnr}\left\langle\frac{\varphi}{2}\right\rangle = \frac{\varphi}{2} + \eta \quad \text{with} \quad \eta \in \left\{-\frac{\mu}{2}, 0, \frac{\mu}{2}\right\} \quad (10)$$

The same holds when rounding towards zero; for upward and downward rounding, the possible values for  $\eta$  are reduced to, respectively,  $\{0, \mu/2\}$  and  $\{-\mu/2, 0\}$ .

The proof of Eq. (10) is easy: in the absence of underflow, the division by 2 is error-free since it amounts to decrementing the biased exponent by one; in case of underflow, the division by 2 is realized by shifting the fractional part one bit to the right (See Fig. 3), which can only introduce an error of  $\pm 2^{-(l(f)+1)} \times 2^{E_{\min}} = \pm \frac{\epsilon_M}{2} \times 2^{E_{\min}} = \pm \frac{\mu}{2}$  if the discarded bit is a “1” and no error if the bit is a “0”.

We may also enjoy an error-free addition/subtraction for certain operands, as stated by the following three theorems, which will be used in the next section. Beforehand, note that we may consider for all practical purposes that  $\text{fl}\langle\varphi_1 + \varphi_2\rangle = \text{fl}\langle\varphi_1 - (-\varphi_2)\rangle$  and  $\text{fl}\langle\varphi_1 - \varphi_2\rangle = \text{fl}\langle\varphi_1 + (-\varphi_2)\rangle$  since taking the opposite of a floating-point number only involves flipping its sign bit, which is error-free.

**Theorem 1 (Hauser’s theorem 3.4.1 [8]).** *If  $\varphi_1$  and  $\varphi_2$  are floating-point numbers, and if  $\text{fl}\langle\varphi_1 + \varphi_2\rangle$  underflows, then  $\text{fl}\langle\varphi_1 + \varphi_2\rangle = \varphi_1 + \varphi_2$ .*

In effect, Theorem 1 means that  $\eta$  is always equal to 0 in Eq. (9) when “+” is the addition or the subtraction.

**Theorem 2 (Hauser’s theorem 3.4.1a [8]).** *If  $\varphi_1$  and  $\varphi_2$  are floating-point numbers such that  $\text{fl}\langle\varphi_1 + \varphi_2\rangle < 2\lambda$ , then  $\text{fl}\langle\varphi_1 + \varphi_2\rangle = \varphi_1 + \varphi_2$ .*

**Theorem 3 (Hauser’s theorem 3.4.2 [8]<sup>6</sup>).** *If  $\varphi_1$  and  $\varphi_2$  are floating-point numbers such that  $1/2 \leq \varphi_1/\varphi_2 \leq 2$ , then  $\text{fl}\langle\varphi_1 + \varphi_2\rangle = \varphi_1 + \varphi_2$ .*

Many other results to narrow the error bounds for some expressions may be found in the writings of Hauser [8], Goldberg [6] or Higham [10], to name a few, but the ones presented here will be enough for our purpose.

Hauser’s theorems give sufficient conditions to have error-free additions and subtractions. On the other hand, these operations may generate large errors through *absorption* and *cancellation*.

*Absorption.* Consider the two *tiny* floating-point numbers  $a = 1.000 \times 2^3$  and  $b = 1.000 \times 2^{-1}$ . In order to add them, we have first to align their exponents. This is done by scaling the one with the smallest exponent:

$$\begin{array}{r} a \quad 1.000 \times 2^3 \\ b \quad +0.000\mathbf{1} \times 2^3 \\ \hline \quad 1.000\mathbf{1} \times 2^3 \end{array}$$

The addition is performed with one extra bit for the significand of all values involved, the *guard bit*, which is necessary to ensure the validity of the error model of Eq. (9). However, the result loses this guard bit when stored in the floating-point format on hand; if we round to the nearest-even, we then obtain  $a + b = 1.000 \times 2^3 = a$ . The *absolute error* is:

$$|\text{rndnr}\langle a + b \rangle - (a + b)| = 2^{-1}$$

---

<sup>6</sup> The better known Sterbenz’s theorem [22] states the same result in a restricted form where underflows are forbidden [10].

while the *relative error* is:

$$\frac{|\text{rndnr}\langle a + b \rangle - (a + b)|}{|a + b|} = \frac{2^{-1}}{2^3 + 2^{-1}} = \frac{1}{17}$$

As expected by the error model in Eq. (9), the relative error is bounded by  $\varepsilon_M/2 = 2^{-4} = 1/16$ .

Absorption in addition will occur whenever the difference in magnitude between the two operands leads to a shifting of all significant bits of one of the operands to the right of the rightmost bit of its fractional part.

*Cancellation.* Consider two nearly equal *tiny* numbers  $a = 1.001 \times 2^3$  and  $b = 1.000 \times 2^3$  and suppose they are the rounded approximations of earlier computation, respectively  $a' = 1.00011 \times 2^3$  and  $b' = 1.00010 \times 2^3$ . If we perform the subtraction  $a - b$ , we get:

$$\begin{array}{r} a \quad 1.001 \times 2^3 \\ b \quad -1.000 \times 2^3 \\ \hline \quad 0.001 \times 2^3 \end{array}$$

Even though the subtraction performed is error-free, we get the result “1” while  $a' - b' = 2^{-2}$ . All correct figures from  $a$  and  $b$  cancel each other out and only the wrong figures remain and are magnified in the process—the more so for large  $a$  and  $b$ . As a rule of thumb, we should therefore avoid subtracting quantities spoiled by previous errors.

### 3 The Floating-Point Midpoint Operator

As said at the beginning of Section 1, a naive floating-point implementation of Eq. (1) must consider the rounding of  $\rho = a + b$ , for  $a$  and  $b$  two floating-point numbers<sup>7</sup>. A floating-point implementation of Eq. (1) may fatally fail in two different ways:

**Pitfall 1.** For the interval  $[-\infty, +\infty]$ ,  $a + b$  returns a *not-a-number* (NaN), as defined by the IEEE 754 standard for the addition involving two infinities of differing signs;

**Pitfall 2.** If  $a$  and  $b$  have the same sign and their sum overflows,  $a + b$  will be an infinity, which divided by two will still give an infinity<sup>8</sup>.

The MPFI 1.4 [20] interval arithmetic library implements the midpoint operator using Eq. (1) rounded to the nearest:

$$m([a, b]) = \text{rndnr}\left\langle \frac{a + b}{2} \right\rangle \quad (\text{MPFI 1.4})$$

<sup>7</sup> In the rest of this paper, we consider the double precision format, even though all results are applicable to other IEEE 754 formats as well.

<sup>8</sup> See below for a proviso on this, though.

It avoids falling into the second pitfall by using MPFR [5], a package for arbitrary precision floating-point computation. As a consequence,  $a+b$  does not overflow; the addition is simply computed with the required number of bits and then rounded down to the original size. On the other hand, the MPFI implementation is obviously subject to the first problem, and that library will happily return an NaN as midpoint of the interval  $[-\infty, +\infty]$ .

The C++ Boost Interval Arithmetic Library [2] uses a slightly more complex implementation of the midpoint operator, *viz.*:

$$m([a, b]) = \begin{cases} -\mathbf{realmax} & \text{if } a = -\infty; \\ \mathbf{realmax} & \text{if } b = +\infty; \\ \mathbf{rndnr}\left\langle \frac{a+b}{2} \right\rangle & \text{otherwise.} \end{cases} \quad (\text{Boost rev84})$$

The cases are not mutually exclusive; however, only one rule is used (the first one that applies, in the order given), noting that the actual implementation adds a zeroth case for empty interval inputs, for which it returns an NaN.

The error analysis for the third rule of Eq. (Boost rev84) is fairly simple with the model of Eq. (9):

$$\mathbf{rndnr}\left\langle \frac{a+b}{2} \right\rangle = \frac{(a+b)(1+\delta_1) + \eta_1}{2}(1+\delta_2) + \eta_2$$

Thanks to Theorem 1, we have  $\eta_1 = 0$ . We also have  $\delta_2 = 0$ , since a division by two is error-free in the absence of underflow. Hence the simplified expression:

$$\mathbf{rndnr}\left\langle \frac{a+b}{2} \right\rangle = \frac{(a+b)}{2}(1+\delta_1) + \eta_2 \quad (11)$$

with  $|\delta_1| \leq \varepsilon_M/2$  and  $\eta_2 \in \{-\frac{\mu}{2}, 0, \frac{\mu}{2}\}$  (See Eq. (10)). In the absence of overflow/underflow, we then have a correctly rounded result:

$$\mathbf{rndnr}\left\langle \frac{a+b}{2} \right\rangle = \frac{a+b}{2}(1+\delta_1)$$

which is the best we can expect.

Thanks to the first rule, the Boost library will not return an NaN for  $m([-\infty, +\infty])$  but  $-\mathbf{realmax}$ . This result does not violate Eq. (2), and Eq. (3) is not relevant in the context of intervals with an infinite radius. It might, however, appear odd to the unsuspecting user to discover that the midpoint of a non-empty symmetric interval<sup>9</sup> may not be 0.

That the Boost library may, in some instances, not fall prey to Pitfall 2 may come as a surprise; in fact, the occurrence of overflows depends on minute details on the actual compilation of the expression  $\mathbf{rndnr}\langle(a+b)/2\rangle$ : for example, on a computer with an ix86-based processor offering at least the SSE2 instructions set [12, Chap. 11], there are essentially two ways to compute with double precision floating-point numbers:

<sup>9</sup> A non-empty interval  $[a, b]$  with floating-point bounds is symmetric if  $a = -b$ .

1. With the FPU instructions set and 80-bits registers stack [12, Chap. 8]: computation is performed internally with an extended precision format having 64 bits of significand and 15 bits of exponent. It is still possible to restrict the size of the significand to 53 bits in order to emulate double precision<sup>10</sup> but this feature is rendered almost useless by the fact that it is not possible to restrict the size of the exponent;
2. With the SSE2 instructions set and the SSE2 registers: computation is performed exclusively in double precision.

If a program that uses the Boost library is compiled without any particular option on an ix86-based processor, the expression  $\text{rndnr}\langle(a + b)/2\rangle$  will usually be computed as follows:

1. The double precision variables  $a$  and  $b$  will be loaded into the FPU floating-point registers and promoted to the 80 bits extended format;
2. The expression  $\text{rndnr}\langle(a + b)/2\rangle$  will be computed entirely with extended precision in the FPU registers;
3. The result of the expression will be stored back into a double precision variable with a cast.

In this situation, the overflow cannot happen since  $\text{rndnr}\langle a + b \rangle$  is always representable by a finite number in extended precision; once divided by 2, the result is again representable in double precision.

Now, suppose the program is compiled for an ix86 processor with a compiler option that instructs it to emit code to store back floating-point values into double precision variables after each operation<sup>11</sup> (This behavior has an obvious adverse effect on performances. It is useful, however, to ensure repeatability of the computation across different platforms, some of which may not offer extended precision registers of the same size, if at all). In that case, the expression  $\text{rndnr}\langle\frac{a+b}{2}\rangle$  will be computed as follows:

1. The double precision variables  $a$  and  $b$  will be loaded into the FPU floating-point registers and promoted to the 80 bits extended format;
2. The expression  $\text{rndnr}\langle a + b \rangle$  will be computed with extended precision in the FPU registers;
3. The value of  $\text{rndnr}\langle a + b \rangle$  will be stored back as a double precision floating-point number, at which point an overflow may occur;
4. The value of  $\text{rndnr}\langle a + b \rangle$  will be loaded into an extended register, promoted to 80 bits, and then divided by 2;
5. The result of the division will be stored in a double precision variable with a cast.

In this situation, the Boost implementation of the midpoint operator may fail to comply with its requirements, and will return an infinite value as the midpoint

<sup>10</sup> Since version 4.3.0, The GNU C Compiler gcc offers the option `-mpc64` to do that easily.

<sup>11</sup> This is the `-ffloat-store` option of gcc, or the `-mp` option for the Intel C++ compiler.

of any interval with finite bounds whose absolute value of the sum is greater than `realmax`. The same problem would arise on architectures that do not rely internally on extended precision floating-point registers.

A different way to achieve the same failure on an ix86 processor relies on the possibility for modern compilers to transparently—or upon explicit request by the programmer—make use of the SIMD facilities offered by the SSE2 instructions set: some floating-point expressions may be computed in double precision SSE2 registers instead of the FPU extended precision registers. Once again, `rndnr⟨a + b⟩` may overflow, as it will be computed in double precision only.<sup>12</sup>

To avoid the overflow problem that affects the Boost implementation, Intlab V5.5 [21] uses the following formula:

$$\begin{aligned} \mathbf{m}([a, b]) = & \\ & \begin{cases} 0 & \text{if } a = -\infty \text{ and } b = +\infty; \\ \mathbf{rndup}\langle \frac{a}{2} + \frac{b}{2} \rangle & \text{otherwise.} \end{cases} \end{aligned} \quad (\text{Intlab V5.5})$$

The first rule protects explicitly against the creation of an NaN for the interval  $[-\infty, +\infty]$ , while the second rule avoids an overflow by first dividing the operands by two before adding the results, instead of the other way round.

The error analysis is as follows:

$$\mathbf{rndup}\langle \frac{a}{2} + \frac{b}{2} \rangle = \left[ \left( \frac{a}{2}(1 + \delta_1) + \eta_1 \right) + \left( \frac{b}{2}(1 + \delta_2) + \eta_2 \right) \right] (1 + \delta_3) + \eta_3$$

Using Theorem 1 and Eq. (10), we may simplify the expression as:

$$\mathbf{rndup}\langle \frac{a}{2} + \frac{b}{2} \rangle = \left( \frac{a+b}{2} + \eta_1 + \eta_2 \right) (1 + \delta_3) \quad (12)$$

with  $\eta_1 \in \{0, \mu/2\}$ ,  $\eta_2 \in \{0, \mu/2\}$ , and  $0 \leq \delta_3 < \varepsilon_M$ , since results are rounded upward now. In the absence of underflow/overflow, Eq. (12) simplifies to:

$$\mathbf{rndup}\langle \frac{a}{2} + \frac{b}{2} \rangle = \frac{a+b}{2} (1 + \delta_3)$$

meaning that the result is correctly rounded. Note that, in practice, the quality of this formula is not as good as Boost's since the bound on  $\delta_3$  in Eq. (12) is twice the one on  $\delta_1$  in Eq. (11).

This algorithm has a peculiar behavior in that the midpoint of an interval with one infinite bound (but not both) is this infinite (for example:  $\mathbf{m}([-\infty, 3]) = -\infty$  and  $\mathbf{m}([3, +\infty]) = +\infty$ ). In addition, a symmetric interval with finite bounds

<sup>12</sup> This transparent use of SIMD registers instead of FPU ones leads to another problem: the instructions that set the rounding direction for the FPU and SSE2 instructions are separate; therefore, the rounding direction set by the programmer for some FPU computation will not apply if, unbeknownst to him, the compiler chooses to use SSE2 instructions instead.

may have a midpoint different from 0: the midpoint of the interval  $[-\mu, \mu]$  is computed as  $\mu$  because the expressions  $\text{rndup}\langle 0.5 \times -\mu \rangle$  and  $\text{rndup}\langle 0.5 \times \mu \rangle$  are rounded, respectively, to 0 and  $\mu$ . For the interval  $[-\mu, \mu]$ , this has the unfortunate effect of violating Eq. (3) since 0 is a representable floating-point number closer to the midpoint of the interval.

Additionally and more importantly, Intlab V5.5 may also violate Eq. (2). Consider the reduction of Eq. (12) for subnormal inputs:

$$\text{rndup}\left\langle \frac{a}{2} + \frac{b}{2} \right\rangle = \frac{a+b}{2} + \eta_1 + \eta_2$$

We have  $\eta_1 + \eta_2 \in \{0, \mu/2, \mu\}$ , which means that in some cases, the result may not be correctly rounded. Take for example, the interval  $[\mu, \mu]$ . Since we have:

$$\text{rndup}\langle 0.5 \times \mu \rangle = \mu,$$

it follows:

$$\text{m}([\mu, \mu]) = 2\mu \notin [\mu, \mu]$$

BIAS 2.0.8 [14] follows a different path and computes the midpoint as follows:

$$\text{m}([a, b]) = \text{rndup}\left\langle a + \frac{b-a}{2} \right\rangle \quad (\text{BIAS 2.0.8})$$

Despite using a subtraction in place of an addition, this algorithm has the same flaws as a naive implementation of midpoint: if  $b - a$  overflows (say, because  $a$  and  $b$  are huge, with differing signs), the result will be infinite; additionally,  $\text{m}([-\infty, +\infty])$  is a NaN because  $b - a = +\infty - -\infty = +\infty$  and  $a + (b - a) = -\infty + \infty = \text{NaN}$ .

The error analysis is also interesting:

$$\text{rndup}\left\langle a + \frac{b-a}{2} \right\rangle = \left[ a + \frac{(b-a)(1+\delta_1) + \eta_1(1+\delta_2) + \eta_2}{2} \right] (1+\delta_3) + \eta_3$$

which reduces to:

$$\text{rndup}\left\langle a + \frac{b-a}{2} \right\rangle = \left( a + \frac{(b-a)}{2}(1+\delta_1) + \eta_2 \right) (1+\delta_3) \quad (13)$$

with  $\eta_2 \in \{0, \mu/2\}$ ,  $0 \leq \delta_1 < \varepsilon_M$  and  $0 \leq \delta_3 < \varepsilon_M$ . In the absence of overflow/underflow, we get:

$$\text{rndup}\left\langle a + \frac{b-a}{2} \right\rangle = \frac{a+b}{2} + \left( \delta_1 \frac{b-a}{2} + \delta_3 \frac{a+b}{2} + \delta_1 \delta_3 \frac{b-a}{2} \right)$$

In theory, BIAS's formula is then even worse than Intlab V5.5's in the absence of underflow. On the other hand, the reduction of Eq. (13) for the case of subnormal inputs shows that it is not affected by Intlab V5.5's problem:

$$\text{rndup}\left\langle a + \frac{b-a}{2} \right\rangle = \frac{a+b}{2} + \eta_2$$

For subnormal inputs, BIAS computes a correctly rounded result.

CXSC 2.4.0 [7] offers two functions to compute the midpoint: a “fast” one,  $\text{Mid}(\mathbf{I})$ , and an accurate one,  $\text{mid}(\mathbf{I})$ . The fast one computes the midpoint by a formula akin to both BIAS’s and Intlab’s, except for the rounding, which is now downward:

$$\text{m}([a, b]) = \text{rnddn}\left\langle a + \left(\frac{b}{2} - \frac{a}{2}\right) \right\rangle \quad (\text{CXSC 2.4.0})$$

That formula is still subject to the NaN problem whenever  $[a, b] = [-\infty, +\infty]$ . The accurate version uses extended precision, and then, contrary to the fast version, it always returns the correctly rounded value for the midpoint, except for the interval  $[-\infty, +\infty]$ . For that interval, CXSC aborts abruptly with an exception.

The error analysis for the “fast” midpoint operator of CXSC gives:

$$\begin{aligned} \text{rnddn}\left\langle a + \left(\frac{b}{2} - \frac{a}{2}\right) \right\rangle = \\ \left[ a + \left( \left(\frac{b}{2}(1 + \delta_1) + \eta_1\right) - \left(\frac{a}{2}(1 + \delta_2) + \eta_2\right) \right) (1 + \delta_3) + \eta_3 \right] (1 + \delta_4) + \eta_4, \end{aligned}$$

which simplifies to:

$$\text{rnddn}\left\langle a + \left(\frac{b}{2} - \frac{a}{2}\right) \right\rangle = \left[ a + \left( \frac{b-a}{2} + \eta_1 - \eta_2 \right) (1 + \delta_3) \right] (1 + \delta_4) \quad (14)$$

with  $\eta_1 \in \{-\mu/2, 0\}$ ,  $\eta_2 \in \{-\mu/2, 0\}$ ,  $-\varepsilon_M < \delta_3 \leq 0$ , and  $-\varepsilon_M < \delta_4 \leq 0$ .

In the absence of overflow/underflow, CXSC’s formula is as bad as BIAS’s:

$$\text{rnddn}\left\langle a + \left(\frac{b}{2} - \frac{a}{2}\right) \right\rangle = \frac{a+b}{2} + \left( \delta_3 \frac{b-a}{2} + \delta_4 \frac{a+b}{2} + \delta_3 \delta_4 \frac{b-a}{2} \right)$$

For subnormal inputs, the halving of  $a$  and  $b$  may introduce some error, and the subtraction  $b/2 - a/2$  may lead to some cancellation for  $a \approx b$ . We have:

$$\text{rnddn}\left\langle a + \left(\frac{b}{2} - \frac{a}{2}\right) \right\rangle = \frac{a+b}{2} + \eta_1 - \eta_2$$

with  $\eta_1 - \eta_2 \in \{-\mu/2, 0, \mu/2\}$ .

Fi.lib++ 2.0 [15] uses almost the same formula as Intlab V5.5 with additional rules to avoid some of the pitfalls previously encountered:

$$\text{m}([a, b]) = \begin{cases} a & \text{if } a = b \\ 0 & \text{if } a = -b \\ \text{rndnr}\left\langle \frac{a}{2} + \frac{b}{2} \right\rangle & \text{otherwise} \end{cases} \quad (\text{Fi.lib++ 2.0})$$



The error analysis for `Fi.lib++`'s formula leads to the same expression as for Intlab V5.5, except for the bounds on the errors:

$$\text{rndnr}\left\langle\frac{a}{2} + \frac{b}{2}\right\rangle = \left(\frac{a+b}{2} + \eta_1 + \eta_2\right)(1 + \delta_3) \quad (15)$$

with  $\eta_1 \in \{-\mu/2, 0, \mu/2\}$ ,  $\eta_2 \in \{-\mu/2, 0, \mu/2\}$ , and  $|\delta_3| \leq \varepsilon_M/2$ .

As for Intlab V5.5, the result is correctly rounded whenever no overflow/underflow occurs. On the other hand, for subnormal inputs, we still get:

$$\text{rndnr}\left\langle\frac{a}{2} + \frac{b}{2}\right\rangle = \frac{a+b}{2} + \eta_1 + \eta_2$$

with  $\eta_1 + \eta_2 \in \{-\mu, -\mu/2, 0, \mu/2, \mu\}$ , which means that the computed midpoint may be one of the floating-point numbers that surround the correctly rounded midpoint. For a degenerate interval reduced to one point, the first rule protects against computing a midpoint outside the interval, as was the case with Intlab V.5.5; Obviously, the formula ensures that the midpoint of an interval with a width greater or equal to  $2\mu$  is included in it. For an interval whose bounds  $a$  and  $b$  are consecutive subnormal floating-point numbers (hence, with a width precisely equal to  $\mu$ ), it suffices to notice that exactly one of them has a rightmost bit of the fractional part equal to “1” and the other one has the corresponding bit equal to “0”. Consequently,  $\eta_1 + \eta_2 \in \{-\mu/2, 0, \mu/2\}$  and the inclusion is, once again, ensured.

The second rule in Eq. ([Fi.lib++ 2.0](#)) is more general than the first one of Intlab V5.5: it protects against the computation of a NaN for the interval  $[-\infty, +\infty]$  as well as ensuring that symmetric intervals in general have 0 as their midpoint (another “flaw” of the Intlab V5.5 formula). The expression in the third rule cannot overflow for finite  $a$  and  $b$ , which makes the whole formula immune to the overflow problem. On the other hand, the midpoint of any interval with an infinite bound (but not both) is infinite (*e.g.*:  $\mathfrak{m}([3, +\infty]) = +\infty$ ). This result does not violate Eq. ([2](#)) and Eq. ([3](#)); it may nevertheless lead to serious trouble if the returned value is used as a bound of an interval, say in a bisection algorithm.

The recent Version 6 of the Intlab package has a different algorithm from Version 5.5 to compute the midpoint of an interval:

$$\mathfrak{m}([a, b]) = \begin{cases} 0 & \text{if } a = -\infty \text{ and } b = +\infty; \\ b & \text{if } a = -\infty; \\ a & \text{if } b = +\infty; \\ \text{rndnr}\left\langle a + \left(\frac{b}{2} - \frac{a}{2}\right) \right\rangle & \text{otherwise;} \end{cases} \quad (\text{Intlab V6})$$

Once again, these rules are not mutually exclusive, but the first that applies is executed to the exclusion of the others. The first rule explicitly avoids the NaN problem for the interval  $[-\infty, +\infty]$ ; the second and third rules ensure that the

midpoint of an interval with one infinite bound (and one only) is finite and equal to the other—finite—bound. This is one possible choice among many to resolve the problem and it would be interesting to investigate which choice is usually the best.

Apart from the rounding direction, the formula in the fourth rule is the same as CXSC's formula, and consequently, an error analysis along the same lines may be made:

$$\text{rndnr}\left\langle a + \left(\frac{b}{2} - \frac{a}{2}\right) \right\rangle = \left[ a + \left( \left(\frac{b}{2} + \eta_1\right) - \left(\frac{a}{2} + \eta_2\right) \right) (1 + \delta_3) \right] (1 + \delta_4) \quad (16)$$

with  $\eta_1 \in \{-\mu/2, 0, \mu/2\}$ ,  $\eta_2 \in \{-\mu/2, 0, \mu/2\}$ ,  $|\delta_3| \leq \varepsilon_M/2$ , and  $|\delta_4| \leq \varepsilon_M/2$ .

For subnormal inputs, it may violate Eq. (3), as evident from the error formula:

$$\text{rndnr}\left\langle a + \left(\frac{b}{2} - \frac{a}{2}\right) \right\rangle = \frac{a+b}{2} + \eta_1 - \eta_2$$

with  $\eta_1 - \eta_2 \in \{-\mu, -\mu/2, 0, \mu/2, \mu\}$ . For example, with Intlab V6's formula, we have:

$$\text{m}([- \mu, \mu]) = -\mu$$

On the other hand, for finite  $a$  and  $b$ , the fourth rule is immune to overflow and it ensures that Eq. (2) is verified, as proven by Prop. 1.

**Proposition 1.** *Given  $a$  and  $b$  two finite floating-point numbers, with  $a \leq b$ , we have:*

$$a \leq \text{rndnr}\left\langle a + \left(\frac{b}{2} - \frac{a}{2}\right) \right\rangle \leq b$$

*Proof.* Since  $a \leq b$ , we have  $\text{rndnr}\langle a/2 \rangle \leq \text{rndnr}\langle b/2 \rangle$ , thanks to rounding monotonicity. This is true even if an underflow occurs. As a consequence,  $\text{rndnr}\langle b/2 - a/2 \rangle \geq 0$ , and then  $\text{rndnr}\langle a + (b/2 - a/2) \rangle \geq a$ , again by monotonicity of rounding. Note that, for finite  $a$  and  $b$ ,  $\text{rndnr}\langle b/2 - a/2 \rangle$  cannot overflow.

Let us now prove that  $\text{rndnr}\langle a + (b/2 - a/2) \rangle \leq b$ . We first prove that  $\text{rndnr}\langle b/2 - a/2 \rangle \leq b - a$ , restricting the discussion to the case  $a \neq b$  since the proof is trivial otherwise. We have:

$$\text{rndnr}\left\langle \frac{b}{2} - \frac{a}{2} \right\rangle = \left[ \left(\frac{b}{2} + \eta_1\right) - \left(\frac{a}{2} + \eta_2\right) \right] (1 + \delta_3) \quad (17)$$

Using Eq. (17), let us now find the conditions such that  $\text{rndnr}\langle b/2 - a/2 \rangle > b - a$ :

$$\begin{aligned} \text{rndnr}\left\langle \frac{b}{2} - \frac{a}{2} \right\rangle > b - a &\iff (b - a) \frac{1 + \delta_3}{2} + (\eta_1 - \eta_2)(1 + \delta_3) > b - a \\ &\iff b - a < 2(\eta_1 - \eta_2) \frac{1 + \delta_3}{1 - \delta_3} \end{aligned}$$

We have  $|\delta_3| \leq \varepsilon_M/2$ ,  $|\eta_1| \leq \mu/2$ , and  $|\eta_2| \leq \mu/2$ . As soon as  $l(f) > 1$ ,  $(1 + \delta_3)/(1 - \delta_3)$  is strictly smaller than 2. Hence,  $b - a$  has to be less than  $4\mu$  for  $\text{rndnr}\langle b/2 - a/2 \rangle$  to be greater than  $b - a$ .

We then deduce that  $\text{rndnr}\langle b/2 - a/2 \rangle$  is safely strictly less than  $\lambda$  for any floating-point format of practical use (just use  $b - a < 4\mu$  in Eq. (17)), that is  $\text{rndnr}\langle b/2 - a/2 \rangle$  underflows. In that case,  $\delta_3 = 0$ , thanks again to Theorem 1, and then we have:

$$\begin{aligned} \text{rndnr}\left\langle \frac{b}{2} - \frac{a}{2} \right\rangle > b - a &\iff \frac{b - a}{2} + \eta_1 - \eta_2 > b - a \\ &\iff b - a < 2(\eta_1 - \eta_2) \end{aligned}$$

With the known bounds on  $\eta_1$  and  $\eta_2$ , we now deduce that  $b - a$  must be strictly less than  $2\mu$  for  $\text{rndnr}\langle b/2 - a/2 \rangle$  to be greater than  $b - a$ . But  $b - a$  must be a strictly positive floating-point number (we have  $b > a$ , and  $b - a < \lambda$  implies that Theorem 1 applies). Consequently,  $b - a$  must be equal to  $\mu$ , which means that  $b$  and  $a$  are two consecutive floating-point numbers. As a result, either  $a$  or  $b$  has an “even” binary significand (rightmost bit equal to 0), which means that either  $\eta_1 = 0$  or  $\eta_2 = 0$ . We may then refine our condition:

$$\begin{aligned} \text{rndnr}\left\langle \frac{b}{2} - \frac{a}{2} \right\rangle > b - a &\iff \frac{b - a}{2} + \eta > b - a \quad \text{with } |\eta| \leq \mu/2 \\ &\iff b - a < \mu \end{aligned}$$

This condition is not possible for  $a \neq b$  since there is no positive floating-point number smaller than  $\mu$ . Consequently:

$$\text{rndnr}\left\langle \frac{b}{2} - \frac{a}{2} \right\rangle \leq b - a \quad \forall (a, b) \in \mathbb{F}^2$$

We then have:

$$a + \text{rndnr}\left\langle \frac{b}{2} - \frac{a}{2} \right\rangle \leq b$$

And, by monotonicity of rounding:

$$\text{rndnr}\left\langle a + \left(\frac{b}{2} - \frac{a}{2}\right) \right\rangle \leq b$$

□

The formulae investigated so far all have defects: some are prone to overflows (e.g.,  $(a + b)/2$ ), some may violate the containment requirement (e.g.,  $a/2 + b/2$ , if used without precautions), and others may compute a value that is markedly different from the center of the interval, thereby violating Eq. (3).

The true impact of the violation of Eq. (3) should be carefully assessed. But for now, we will content ourselves with investigating whether it is possible to design a formula that computes a correctly rounded floating-point approximation of the midpoint of an interval, without exhibiting the flaws previously identified.

We should rule out a formula based on the expression  $(a + b)/2$ , as this would require to detect a possible overflow in the addition. On the other hand, it is

easy to amend `Fi.lib++`'s algorithm to avoid returning infinities when a bound is not finite:

$$m([a, b]) = \begin{cases} a & \text{if } a = b; \\ 0 & \text{if } a = -b; \\ ? & \text{if } a = -\infty; \\ ? & \text{if } b = +\infty; \\ \text{rndnr}\langle \frac{a}{2} + \frac{b}{2} \rangle & \text{otherwise.} \end{cases} \quad (18)$$

In the absence of underflow, this algorithm computes a correctly rounded approximation of the midpoint, since then, dividing a floating-point number by 2 is error-free. The question marks for Cases 3 and 4 are here to remind us that we may promote any finite value belonging to the interval  $[a, b]$  as the midpoint without violating Eq. (3).

This algorithm fits the bill, except that, having so many cases, it is not very efficient. However, it is possible to devise a correct algorithm with less cases by taking advantage of Hauser's theorems:

$$m([a, b]) = \begin{cases} 0 & \text{if } a = -b; \\ ? & \text{if } a = -\infty; \\ ? & \text{if } b = +\infty; \\ \text{rndnr}\langle (a - \frac{a}{2}) + \frac{b}{2} \rangle & \text{otherwise.} \end{cases} \quad (\text{H-method})$$

As before, only the first rule whose guard is true is considered. Case 4 rewrites the expression " $a + (b/2 - a/2)$ " from `Intlab V6` as " $(a - a/2) + b/2$ ", a small algebraic manipulation indeed, but one that can have a significant impact on the accuracy of the whole expression. In effect, the error analysis gives:

$$\text{rndnr}\langle (a - \frac{a}{2}) + \frac{b}{2} \rangle = \left[ \left( a - \left( \frac{a}{2}(1 + \delta_1) + \eta_1 \right) \right) (1 + \delta_2) + \eta_2 + \left( \frac{b}{2}(1 + \delta_3) + \eta_3 \right) \right] (1 + \delta_4) + \eta_4 \quad (19)$$

This complicated formula may be simplified thanks to the following lemma:

**Lemma 1.** *For any finite floating-point number  $a \in \mathbb{F}$ , we have:*

$$\text{rndnr}\langle a - \frac{a}{2} \rangle = a - \left( \frac{a}{2} + \eta \right), \quad \text{with } \eta \in \left\{ -\frac{\mu}{2}, 0, \frac{\mu}{2} \right\}$$

*Proof.* There are basically two cases to consider: let us first suppose no underflow occurs when halving  $a$ . Then, we have  $\text{rndnr}\langle a/2 \rangle = a/2$  since halving is error-free in the absence of underflow. We may then use Hauser's Theorem 3 to state that  $\text{rndnr}\langle a - a/2 \rangle = a - a/2$ .

Now, suppose an underflow occurs when halving  $a$ ; then,  $\text{rndnr}\langle a/2 \rangle = a/2 + \eta$ . This event is only possible if  $a/2$  is strictly smaller than  $\lambda$  (by definition), and

this can happen only if  $a$  is strictly smaller than  $2\lambda$ . But then,  $\text{rndnr}\langle a - (a/2 + \eta) \rangle$  is also strictly smaller than  $2\lambda$ , and we may then use Theorem 2 to state that  $\text{rndnr}\langle a - a/2 \rangle = a - (a/2 + \eta)$ .  $\square$

Using Lemma 1 and the property on halving given by Eq. (10), we are able to simplify Eq. (19) to:

$$\text{rndnr}\left\langle \left( a - \frac{a}{2} \right) + \frac{b}{2} \right\rangle = \left( \frac{a+b}{2} + \eta_3 - \eta_1 \right) (1 + \delta_4) \quad (20)$$

with  $\eta_1 \in \{-\mu/2, 0, \mu/2\}$ ,  $\eta_3 \in \{-\mu/2, 0, \mu/2\}$ ,  $|\delta_4| \leq \varepsilon_M/2$ . It is then straightforward to show that containment is always ensured: no overflow can occur in computing  $(a - a/2) + b/2$  and, in the absence of underflow, a correctly rounded midpoint is computed; additionally, in case of underflow, the error is bounded by  $\mu$ .

Implementation evidences presented below show that the cost of the extra subtraction in Eq. (H-method) is compensated by having one test less than in Eq. (18) while both have similar error bounds (Compare Eq. (20) and Eq. (15)). At least, this is true for computers on which branching is more expensive than floating-point arithmetic (most desktop computers nowadays, it appears). For those computers that do not verify this statement, we should retain the formula of Eq. (18). We therefore continue to consider both formulae in the rest of the paper.

How should we instantiate the question marks in Eq. (H-method) and Eq. (18)? Arnold Neumaier [17] argues for returning the other, finite, bound of the interval, his justification being that this is a sensible choice when using a midpoint in the context of centered forms [19]. On the other hand, that choice is not so useful in the context of a dichotomic exploration algorithm (Neumaier's stance on the subject is that we should use a different, specialized operator for this kind of algorithm).

For exploration algorithms, a simple choice seems to return  $\pm\text{realmax}$  for semi-unbounded intervals, obtaining the two formulae:

$$\text{m}([a, b]) = \begin{cases} a & \text{if } a = b; \\ 0 & \text{if } a = -b; \\ -\text{realmax} & \text{if } a = -\infty; \\ +\text{realmax} & \text{if } b = +\infty; \\ \text{rndnr}\langle \frac{a}{2} + \frac{b}{2} \rangle & \text{otherwise.} \end{cases} \quad \text{m}([a, b]) = \begin{cases} 0 & \text{if } a = -b; \\ -\text{realmax} & \text{if } a = -\infty; \\ +\text{realmax} & \text{if } b = +\infty; \\ \text{rndnr}\langle (a - \frac{a}{2}) + \frac{b}{2} \rangle & \text{otherwise.} \end{cases}$$

Obviously, any finite value from the interval could be used too. A direction for future researches should be to identify the best choice for the case of semi-unbounded intervals, depending on the main uses of the midpoint operator, *viz.*:

- Centered forms [19];

- Newton steps [16];
- Dichotomic exploration;
- Preconditioning of matrices [13].

### From Theory to Practice

So far, we have only theoretically analyzed the various formulae to compute the midpoint of an interval  $[a, b]$ . Table 2 synthesizes the error formulae in three cases:

- GC.** The general case, where an underflow may or may not occur at any step of the computation (for the experiments, overflows are allowed in the General Case as well, though the formulae in Table 2 are then no longer relevant);
- No OF/UF.** The case where no overflow nor any underflow occurs at any step of the computation;
- UF.** The case where all quantities manipulated (counting the initial inputs) are subnormal numbers.

According to Table 2, we should be able to separate the various methods into four classes for the case where no underflow nor any overflow occurs:

- Boost, Filib++ and the H-method, which deliver a correctly rounded result to the *unit roundoff* (half the epsilon of the machine);
- Intlab V5.5, which delivers a correctly rounded result to the epsilon of the machine;
- Intlab V6, which delivers a result to the unit roundoff;
- Bias and CXSC, which deliver a result to the epsilon of the machine.

This is indeed verified by the experiments, as shown in Table 3, where the first column “Disc” indicates the number of floating-point numbers between the correctly rounded midpoint (as computed by MPFR [5]) and the midpoint computed by the method tested<sup>13</sup>. The last row “Fail” in the table indicates the number of times a midpoint computed was not included in the interval input. The table reports the results for the computation of the midpoint of ten millions non-empty random intervals.

The comparison of all formulae when subnormal numbers are produced for each operation shows less drastic differences (See Table 4): the best we can compute in that case is  $(a + b)/2 + \eta$ , with  $|\eta| \in \{0, \mu/2\}$ , which is exactly what is computed by Boost. BIAS exhibits the same error bound but the computed result may differ by one unit in the last place from Boost’s, due to the upward rounding. Intlab V5.5 shows a discrepancy of one unit in the last place almost half of the time: the error is  $\eta_1 + \eta_2$  with  $\eta_1$  and  $\eta_2$  both positive or null, which means that they can never cancel each other out. On the other hand, CXSC offers a good accuracy because the error term is  $\eta_1 - \eta_2$  with both  $\eta_1$  and  $\eta_2$

<sup>13</sup> If the real midpoint is equidistant from two floating-point numbers, we report the shortest distance between the computed midpoint and both numbers to avoid the arbitrary bias introduced by the “rounding to even” rule.

**Table 2.** Synthesis of error bounds

Method	GC	Error	
		No OF/UF	UF
<b>Boost</b> rndnr $\langle(a+b)/2\rangle$	$\frac{a+b}{2}(1+\delta_1)+\eta_2$ $ \eta_2  \in \{0, \mu/2\},  \delta_1  \leq \varepsilon_M/2$	$\frac{a+b}{2}(1+\delta_1)$	$\frac{a+b}{2}+\eta_2$
<b>Intlab V5.5</b> rndup $\langle a/2+b/2\rangle$	$\left(\frac{a+b}{2}+\eta_1+\eta_2\right)(1+\delta_3)$ $\eta_1 \in \{0, \mu/2\}, \eta_2 \in \{0, \mu/2\}, 0 \leq \delta_3 < \varepsilon_M$	$\frac{a+b}{2}(1+\delta_3)$	$\frac{a+b}{2}+\eta_1+\eta_2$
<b>BIAS</b> rndup $\langle a+(b-a)/2\rangle$	$\left(\frac{a+b}{2}+\delta_1\frac{b-a}{2}+\eta_2\right)(1+\delta_3)$ $\eta_2 \in \{0, \mu/2\}, 0 \leq \delta_1 < \varepsilon_M, 0 \leq \delta_3 < \varepsilon_M$	$\left(\delta_1\frac{b-a}{2}+\delta_3\frac{a+b}{2}+\delta_1\delta_3\frac{b-a}{2}\right)$	$\frac{a+b}{2}+\eta_2$
<b>CXSC</b> rnddn $\langle a+(b/2-a/2)\rangle$	$\left[\begin{array}{l} a+ \\ \left(\frac{b-a}{2}+\eta_1-\eta_2\right)(1+\delta_3) \end{array}\right](1+\delta_4)$ $\eta_1 \in \{-\mu/2, 0\}, \eta_2 \in \{-\mu/2, 0\}, -\varepsilon_M < \delta_3 \leq 0$	$\left(\frac{a+b}{2}+\delta_3\frac{b-a}{2}+\delta_4\frac{a+b}{2}+\delta_3\delta_4\frac{b-a}{2}\right)$	$\frac{a+b}{2}+\eta_1-\eta_2$
<b>Filib++</b> rndnr $\langle a/2+b/2\rangle$	$\left(\frac{a+b}{2}+\eta_1+\eta_2\right)(1+\delta_3)$ $ \eta_1  \in \{0, \mu/2\},  \eta_2  \in \{0, \mu/2\},  \delta_3  \leq \varepsilon_M/2$	$\frac{a+b}{2}(1+\delta_3)$	$\frac{a+b}{2}+\eta_1+\eta_2$
<b>Intlab V6</b> rndnr $\langle a+(b/2-a/2)\rangle$	$\left[\begin{array}{l} a+ \\ \left(\frac{b-a}{2}+\eta_1-\eta_2\right)(1+\delta_3) \end{array}\right](1+\delta_4)$ $ \eta_1  \in \{0, \mu/2\},  \eta_2  \in \{0, \mu/2\},  \delta_3  \leq \varepsilon_M/2,  \delta_4  \leq \varepsilon_M/2$	$\left(\delta_3\frac{b-a}{2}+\delta_4\frac{a+b}{2}+\delta_3\delta_4\frac{b-a}{2}\right)$	$\frac{a+b}{2}+\eta_1-\eta_2$
<b>H-method</b> rndnr $\langle(a-a/2)+b/2\rangle$	$\left(\frac{a+b}{2}+\eta_3-\eta_1\right)(1+\delta_4)$ $ \eta_1  \in \{0, \mu/2\},  \eta_3  \in \{0, \mu/2\},  \delta_4  \leq \varepsilon_M/2$	$\frac{a+b}{2}(1+\delta_4)$	$\eta_3-\eta_1$

**Table 3.** Comparison of the quality of the midpoint() implementations — No OF/UF

Disc.	Boost	Intlab V5.5	BIAS	CXSC	Filib++	Intlab V6	H-method
0	10000000	5012760	2512450	2512118	10000000	9867139	10000000
1		4987240	7483333	7483658		130507	
2			2478	2442		1178	
3			460	467			
4			585	569		603	
6			103	139			
8			287	291		294	
12			18	34			
16			126	125		127	
24			7	3			
32			73	74		74	
48			1	2			
64			36	36		36	
96			1				
128			25	25		25	
256			8	8		8	
512			4	4		4	
1024			4	4		4	
16384			1	1		1	
Fail	0	0	0	0	0	0	0



being of the same sign. Lastly, Filib++, Intlab V6 and the H-method are of similar accuracy, with Intlab V6 and the H-method being rigorously equivalent, as Table 2 suggests already.

**Table 4.** Comparison of the quality of the midpoint() implementations — UF

Disc.	Boost	Intlab V5.5	BIAS	CXSC	Filib++	Intlab V6	H-method
0	10000000	5001983	7502568	7500491	6251593	6250468	6250468
1		4998017	2497432	2499509	3748407	3749532	3749532
Fail	0	0	0	0	0	0	0

If we allow both underflows and overflows (See Table 5), we now have five classes:

- Boost, which always returns a correctly rounded result, except when overflows make it fail to ensure inclusion of the midpoint;
- The H-method and Filib++, which never fail to meet the requirement of inclusion while never departing of more than one unit in the last place from the correctly rounded result;
- Intlab V5.5, which delivers half of the time a result departing from the correctly rounded result by one ulp. Though it is not apparent from Table 5, recall that it may fail to ensure containment in some rare cases (See page 14);
- Intlab V6, whose results may significantly depart from the correctly rounded one, though it always meets the inclusion requirement;
- BIAS and CXSC, whose results may significantly depart from the correctly rounded one, and which may not meet the inclusion requirement.

Note that discrepancies for intervals with at least one infinite bound are not reported in Table 5.

From these tables, it seems clear that the most robust and accurate methods to choose from are the modified version of Filib++’s shown in Eq. (18) and the H-method (Eq. (H-method)). Table 6 also compares the various methods performance-wise. All the methods have been implemented in C++ in the same environment. Times are given in seconds for the computation of the midpoint of fifty millions non-empty random intervals, with the same three categories (GC, No OF/UF, UF) as in Table 2.

Note, en passant, the difference in time between the first two categories and the last one: subnormal numbers have to be handled as exceptional values, often in software rather than in hardware, which slows the computation down dramatically.

As hinted at previously, the H-method is competitive with the modified Filib++ method, even though it requires one subtraction more, as it uses one test less. Since they offer the same accuracy for both the normal and subnormal

**Table 5.** Comparison of the quality of the midpoint() implementations — GC

Disc.	Boost	Intlab V5.5	BIAS	CXSC	Filib++	Intlab V6	H-method
0	9990382	5007214	2510168	2509920	9990380	9857852	9990381
1		4983176	7475969	7476173	10	130087	9
2			2490	2488		1226	
3			428	468			
4			601	612		632	
6			105	116			
8			277	275		283	
12			31	20			
16			160	161		162	
24			6	8			
32			68	70		70	
48			2	1			
64			46	46		46	
96			2				
128			15	15		15	
256			6	6		6	
512			5	5		5	
1024			3	3		3	
8192			2	2		2	
32768			1	1		1	
$> 10^{15}$	8		5				
Fail	8	0	4847	4842	0	0	0

**Table 6.** Performances (in seconds) of different midpoint() implementations.

Boost	intlab V5.5	BIAS	CXSC	Filib++	Intlab V6	H-method	Eq. (18)*
GC							
9.95	9.01	9.48	8.85	8.85	10.32	<b>9.84</b>	<b>10.21</b>
No OF/UF							
10.03	8.86	8.66	8.70	9.09	10.09	<b>9.69</b>	<b>10.13</b>
UF							
150.71	150.90	146.84	161.49	198.42	198.26	<b>222.92</b>	<b>224.33</b>

\* filib++ formula with tests added to avoid returning an infinite value.

range, we advocate the use of the H-method as the standard implementation for the midpoint of an interval. Only in those environments where a floating-point subtraction incurs a cost significantly higher than a branching should the modified Filib++ method be used.

To be fair, the Boost method should not be completely discarded since it is the one that offers the best accuracy. However, it should only be used for those problems where an overflow is sure never to occur.

## 4 Quis Custodiet Ipsos Custodes?

**Table 7.** Midpoint formulae shortcomings.

Package	NaN	Inf	NC	SYM	IR
Boost rev84 (P. 11)		✓		✓	
Intlab V5.5 (P. 13)		✓	✓	✓	✓
BIAS 2.0.8 (P. 14)	✓	✓			✓
CXSC 2.4.0 (Mid) (P. 15)	✓	✓			✓
Fi.lib++ 2.0 (P. 15)		✓			✓
Intlab V6 (P. 16)				✓	✓
H-method (P. 19)				✓	✓

NaN. Returning an NaN for a non-empty interval (e.g., for  $m([-\infty, +\infty])$ ).

Inf. Returning an infinite midpoint (e.g., in case of overflow in the addition  $a + b$ , or if a bound is infinite).

NC. Failing to meet the containment requirement (e.g.,  $m([a, b]) \notin [a, b]$  in case of underflow).

SYM. Violation of Eq. (3): not returning 0 for a symmetric interval.

IR. Violation of Eq. (3): returning a value different from the correctly rounded center of the interval in the absence of overflow.

Table 7 summarizes the potential shortcomings of the midpoint implementation for the packages we have considered. Constraint (3) is violated by almost all packages. Pending further investigation, SYM and IR may be considered benign errors in that they should not compromise the reliability of algorithms based on packages that do exhibit them. All other problems are serious flaws that may put reliability in jeopardy. At present, the only library that avoids them is Intlab V6. This is a testimony to the difficulty of implementing one hundred percent reliable interval software, even for seemingly so simple an operator as the midpoint operator. This is troublesome and calls for formally proved interval arithmetic packages—similar to what is done by the INRIA Proval team for floating-point operators [18]—if we really aspire to promote interval computation as proof.

## Acknowledgements

The origin of the work presented can be traced back to some email exchanges with Prof. Arnold Neumaier; Prof. Siegfried Rump read a preliminary version of this paper and gave very valuable advices and feedback, pointing out in particular an error in the first version of the proof of Prop. 1. Dr. Alexandre Goldsztejn made insightful comments on one of the last drafts of the paper. All remaining errors are, naturally, my own.

## References

1. Alefeld, G.E., Herzberger, J.: Introduction to interval computations. Academic Press (1983)
2. Brönnimann, H., Melquiond, G., Pion, S.: The design of the Boost interval arithmetic library. *Theoretical Computer Science* **351**(1), 111–118 (2006)
3. Forsythe, G.E.: How do you solve a quadratic equation? Technical Report CS40, Computer Science Department, Stanford University (1966)
4. Forsythe, G.E.: Pitfalls in computation, or why a math book isn't enough. *The American Mathematical Monthly* **77**(9), 931–956 (1970)
5. Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.* **33**(2), 13 (2007)
6. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys* **23**(1), 5–48 (1991)
7. Hammer, R., Ratz, D., Kulisch, U., Hocks, M.: C++ Toolbox for Verified Scientific Computing I: Basic Numerical Problems. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1997)
8. Hauser, J.R.: Handling floating-point exceptions in numeric programs. *ACM Transactions on Programming Languages and Systems* **18**(2), 139–174 (1996)
9. Hayes, B.: A lucid interval. *American Scientist* **91**(6), 484–488 (2003)
10. Higham, N.J.: Accuracy and Stability of Numerical Algorithms, second edn. Society for Industrial and Applied Mathematics (2002)
11. IEEE: IEEE standard for binary floating-point arithmetic. Tech. Rep. IEEE Std 754-1985, Institute of Electrical and Electronics Engineers (1985). Reaffirmed 1990
12. Intel: Intel 64 and IA-32 architectures software developer's manual: Vol. 1, basic architecture. Manual 253665-025US, Intel Corporation (2007)
13. Kearfott, R.B., Hu, C., III, M.N.: A review of preconditioners for the interval Gauss-Seidel method. *Interval Computations* **1**, 59–85 (1991)
14. Knüppel, O.: PROFIL/BIAS—a fast interval library. *Computing* **53**, 277–287 (1994)
15. Lerch, M., Tischler, G., Gudenberg, J.W.V., Hofschuster, W., Krämer, W.: Filib++, a fast interval library supporting containment computations. *ACM Trans. Math. Softw.* **32**, 299–324 (2006)
16. Moore, R.E.: Interval Analysis. Prentice-Hall, Englewood Cliff, NJ (1966)
17. Neumaier, A.: Private communication (2010)
18. The ProVal project: Proofs of programs. Web site at <http://proval.lri.fr/>
19. Ratschek, H., Rokne, J.: Computer Methods for the Range of Functions. Mathematics and its applications. Ellis Horwood Ltd. (1984)

20. Rouillier, F., Revol, N.: Motivations for an arbitrary precision interval arithmetic and the mpfi library. *Reliable Computing* **11**, 275–290 (2005). DOI 10.1007/s11155-005-6891-y
21. Rump, S.: INTLAB - INTerval LABoratory. In: T. Csendes (ed.) *Developments in Reliable Computing*, pp. 77–104. Kluwer Academic Publishers, Dordrecht (1999). <http://www.ti3.tu-harburg.de/rump/>
22. Sterbenz, P.: *Floating-point Computation*. Prentice Hall (1974)