NOLTA, IEICE

Paper

Interval arithmetic with fixed rounding mode

Siegfried M. Rump [1,3a], *Takeshi Ogita* [2b], *Yusuke Morikura* [3], and *Shin'ichi Oishi* [3]

[1] *Institute for Reliable Computing, Hamburg University of Technology, Schwarzenbergstraße 95, Hamburg 21071, Germany*

[2] *Division of Mathematical Sciences, Tokyo Woman's Christian University, 2-6-1 Zempukuji, Suginami-ku, Tokyo 167-8585, Japan*

[3] *Faculty of Science and Engineering, Waseda University, 3-4-1 Okubo, Shinjuku-ku, Tokyo 169–8555, Japan*

[a] *rump@tuhh.de*
[b] *ogita@lab.twcu.ac.jp*

Received December 10, 2015; Revised March 25, 2016; Published July 1, 2016

Abstract: We discuss several methods to simulate interval arithmetic operations using floating-point operations with fixed rounding mode. In particular we present formulas using only rounding to nearest and using only chop rounding (towards zero). The latter was the default and only rounding on GPU (Graphics Processing Unit) and cell processors, which in turn are very fast and therefore attractive in scientific computations.

Key Words: rounding mode, chop rounding, interval arithmetic, predecessor, successor, IEEE 754

1. Notation

There is a growing interest in so-called verification methods, i.e. algorithms with rigorous verification of the correctness of the result. Such algorithms require error estimations of the basic floating-point operations.

A specifically convenient way to do this is using rounding modes as defined in the IEEE 754 floating-point standard [1]. In particular performing an operation in rounding downwards and rounding upwards computes the narrowest interval with floating-point bounds including the correct value of the operation.

However, such an approach requires frequent changing of the rounding mode. The computational cost for that is sometimes not acceptable. Therefore we discuss in this paper various methods to compute rigorous error bounds for the basic floating-point operations with one or without changing of the rounding mode.

We demonstrate the computational cost for the methods by computing the range of certain functions. It is a fundamental property of interval arithmetic that replacing each operation by its corresponding interval operation and evaluating a nonlinear function f with interval argument X, then

 ©IEICE 2016 DOI: 10.1587/nolta.7.362

the result interval $f(X)$ contains the range $\{f(x) : x \in X\}$.

We use the following notation. First, we collect the constants[1] used in this paper:

$\texttt{Eps}$:	The distance from 1 to the next smaller floating-point number $(1 - \texttt{Eps})$
$\texttt{Fmax}$:	The largest positive floating-point number
$\texttt{Fmin}$:	The smallest positive normalized floating-point number
$\texttt{Eta}$:	The smallest positive subnormal floating-point number $(\texttt{Eta} = 2\,\texttt{Eps} \cdot \texttt{Fmin})$

For example, for IEEE 754 binary64 (double precision) [1],

$$\texttt{Eps} = 2^{-53}, \quad \texttt{Fmax} = (1 - 2^{-53})2^{1024}, \quad \texttt{Fmin} = 2^{-1022}, \quad \texttt{Eta} = 2^{-1074}\ .$$

Let $\mathbb{F}$ denote a set of binary floating-point numbers, and denote by $\mathrm{fl}_{\mathrm{near}} : \mathbb{R} \to \mathbb{F} \cup \{\pm\infty\}$ a rounding to nearest satisfying

$$r \in \mathbb{R},\ f = \mathrm{fl}_{\mathrm{near}}(r) \quad \Rightarrow \quad \begin{cases} f \in \mathbb{F} \quad \text{s.t.} \quad |r - f| = \min\{|r - f'| :\ f' \in \mathbb{F}\}, & |r| < \mathcal{H} \\ f = \infty, & r \geq \mathcal{H} \\ f = -\infty, & r \leq -\mathcal{H} \end{cases}, \tag{1}$$

where $\mathcal{H}$ is some huge number[2] related to $\max\{f \in \mathbb{F}\}$. This definition does not yet specify the tie, it can be rounded, for example, to even or away from zero. The directed roundings downwards $\mathrm{fl}_{\mathrm{down}} : \mathbb{R} \to \mathbb{F} \cup \{-\infty\}$ and upwards $\mathrm{fl}_{\mathrm{up}} : \mathbb{R} \to \mathbb{F} \cup \{\infty\}$ are uniquely defined by

$$r \in \mathbb{R}\ :\ \mathrm{fl}_{\mathrm{down}}(r) := \begin{cases} \max\{f \in \mathbb{F} :\ f \leq r\}, & r \geq \min\{f \in \mathbb{F}\}\ (= -\max\{f \in \mathbb{F}\}) \\ -\infty, & \text{otherwise} \end{cases}$$

and

$$r \in \mathbb{R}\ :\ \mathrm{fl}_{\mathrm{up}}(r) := \begin{cases} \min\{f \in \mathbb{F} :\ r \leq f\}, & r \leq \max\{f \in \mathbb{F}\} \\ \infty, & \text{otherwise} \end{cases},$$

respectively. Moreover, we consider chopping $\mathrm{fl}_{\mathrm{chop}} : \mathbb{R} \to \mathbb{F}$ (also known as truncation or rounding towards zero), which is also uniquely defined by

$$r \in \mathbb{R},\ f = \mathrm{fl}_{\mathrm{chop}}(r) \quad \Rightarrow \quad r \cdot f \geq 0 \quad \text{and} \quad |f| = \max\{f' \in \mathbb{F} :\ |f'| \leq r\}. \tag{2}$$

For $r \geq 0$ we have $\mathrm{fl}_{\mathrm{chop}}(r) = \mathrm{fl}_{\mathrm{down}}(r)$, and $\mathrm{fl}_{\mathrm{chop}}(r) = \mathrm{fl}_{\mathrm{up}}(r)$ for $r \leq 0$. Note that the rounded result in chopping cannot overflow.

Let $\mathrm{fl}(\cdot)$ denote any one of $\mathrm{fl}_{\mathrm{near}}(\cdot)$, $\mathrm{fl}_{\mathrm{up}}(\cdot)$, $\mathrm{fl}_{\mathrm{down}}(\cdot)$ and $\mathrm{fl}_{\mathrm{chop}}(\cdot)$. For operations $\circ \in \{+, -, \cdot, /\}$ we assume the corresponding floating-point operations to satisfy

$$a, b \in \mathbb{F},\ \mathrm{fl}(a \circ b) = (a \circ b) \cdot (1 + \varepsilon) + \delta \quad \text{with} \quad |\varepsilon| \leq \mathbf{u},\ |\delta| \leq \underline{\mathbf{u}}, \tag{3}$$

where $\mathbf{u}$ denotes the relative rounding error unit, and $\underline{\mathbf{u}}$ the underflow unit. The size of $\mathbf{u}$ depends on the rounding mode. Moreover, the size of $\underline{\mathbf{u}}$ depends on the rounding mode and whether gradual underflow is allowed or not. For example, for IEEE 754 binary64 we have

$$\mathbf{u} = 2^{-53}\ (= \texttt{Eps}) \quad \text{and} \quad \underline{\mathbf{u}} = 2^{-1075}$$

in rounding to nearest with gradual underflow. In any case, $\varepsilon \cdot \delta = 0$ in (3). More precisely,

$$|\mathrm{fl}(a \circ b)| \geq \mathbf{u}^{-1}\underline{\mathbf{u}} \ \Rightarrow \ \delta = 0 \qquad \text{and} \qquad |\mathrm{fl}(a \circ b)| < \mathbf{u}^{-1}\underline{\mathbf{u}} \ \Rightarrow \ \varepsilon = 0\ .$$

Moreover, $\mathrm{fl}(\cdots)$ means a result where each operation inside the parentheses is executed as the corresponding floating-point operation.

[1] Note that `Eps` differs from the Matlab constant `eps`, which is defined as the distance from 1 to the next larger floating-point number, i.e. $\texttt{eps} = 2^{-52}$ for IEEE 754 binary64. Moreover, `realmax` and `realmin` are available on Matlab as `Fmax` and `Fmin`, respectively.

[2] For IEEE 754 binary64, $\mathcal{H} := (1 - \frac{1}{2}\texttt{Eps})\texttt{Fmax}$.

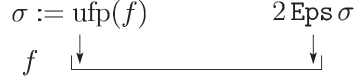

Fig. 1. Normalized floating-point number: unit in the first place and unit in the last place.

We define the predecessor and successor of a floating-point number a by pred : $\mathbb{F} \to \mathbb{F} \cup \{-\infty\}$ and succ : $\mathbb{F} \to \mathbb{F} \cup \{\infty\}$ with

$$\text{pred}(a) := \begin{cases} \max\{f \in \mathbb{F} : f < a\}, & f \neq -\texttt{Fmax} \\ -\infty, & \text{otherwise} \end{cases}$$

and

$$\text{succ}(a) := \begin{cases} \min\{f \in \mathbb{F} : a < f\}, & f \neq \texttt{Fmax} \\ \infty, & \text{otherwise} \end{cases} .$$

For analysis we use the ufp-concept (unit in the first place, see Fig. 1), which was introduced in [4]. In contrast to the "unit in the last place", the ufp-concept is independent of a floating-point format. The function ufp : $\mathbb{R} \to \mathbb{R}$ is defined by

$$0 \neq r \in \mathbb{R} \quad \Rightarrow \quad \text{ufp}(r) := 2^{\lfloor \log_2 |r| \rfloor}$$

and ufp(0) := 0. It follows

$$0 \neq r \in \mathbb{R} : \qquad \text{ufp}(r) \leq |r| < 2\,\text{ufp}(r) \ , \tag{4}$$

$$f \in \mathbb{F} : \qquad f \in 2\,\texttt{Eps} \cdot \text{ufp}(f)\mathbb{Z} \ , \tag{5}$$

$$\texttt{Fmin} \leq f \in \mathbb{F} : \qquad \text{succ}(f) = f + 2\,\texttt{Eps} \cdot \text{ufp}(f) \ , \tag{6}$$

$$\texttt{Fmin} \leq f \in \mathbb{F} : \qquad f + 4\,\texttt{Eps} \cdot \text{ufp}(f) \leq \text{succ}(\text{succ}(f)) \ , \tag{7}$$

where the first three are taken from [4], and the last follows by

$$f + 4\,\texttt{Eps} \cdot \text{ufp}(f) \leq \text{succ}(f) + 2\,\texttt{Eps} \cdot \text{ufp}(\text{succ}(f)) = \text{succ}(\text{succ}(f)) \ .$$

2. Interval bounds using rounding downwards and upwards

As has been mentioned, for $a, b \in \mathbb{F}$, the simplest way to compute bounds $X = [\texttt{cinf}, \texttt{csup}]$ for $a \circ b$ and $\circ \in \{+, -, \cdot, /\}$ is using the following Algorithm 1.

Algorithm 1 Bounds for $a \circ b$ using directed roundings.

function $X = \texttt{BoundUpDown}(a, b)$
  $\texttt{cinf} = \text{fl}_{\text{down}}(a \circ b)$
  $\texttt{csup} = \text{fl}_{\text{up}}(a \circ b)$
  $X = [\texttt{cinf}, \texttt{csup}]$

The computed bounds are best possible. In fact, we have the nice mathematical property

$$\texttt{cinf} = \texttt{csup} \ \Leftrightarrow \ a \circ b \in \mathbb{F} \ .$$

But two switches of the rounding mode are necessary, and if the computation should proceed in rounding to nearest even three. It is known that it is better to use $\mathbb{F} = -\mathbb{F}$ and $-\text{fl}(r) = \text{fl}(-r)$ for $r \in \mathbb{R}$. Then the algorithm for multiplication, for example, is as follows.

Algorithm 2 Bounds for $a \cdot b$ using rounding upwards.

function $X = \texttt{BoundMultUp}(a, b)$
  $\texttt{cinf} = \text{fl}_{\text{up}}(-(a \cdot (-b)))$
  $\texttt{csup} = \text{fl}_{\text{up}}(a \cdot b)$
  $X = [\texttt{cinf}, \texttt{csup}]$

Note that the results of Algorithms 1 and 2 for $a \cdot b$ are identical, i.e. narrowest possible. However, Algorithm 2 needs only one switching of the rounding mode. Corresponding algorithms for the other basic operations follow, but not for the square root.

Similar algorithms using rounding downwards are easily derived. However, in verified computations it seems advantageous to use the rounding upwards. For example, an upper bound for the 2-norm of a vector $x \in \mathbb{F}^n$ is computed by

Algorithm 3 Upper bound of Euclidian norm of a vector using rounding upwards.

$$
\begin{array}{l}
\textbf{function} \;\; \alpha = \texttt{Norm2Up}(x) \\
\quad s = 0 \\
\quad \textbf{for } i = 1 : n \\
\quad\quad s = \mathrm{fl}_{\mathrm{up}}(s + x_i \cdot x_i) \\
\quad \textbf{end for} \\
\quad \alpha = \mathrm{fl}_{\mathrm{up}}(\sqrt{s})
\end{array}
$$

Partly due to the lack of an upper bound for square root this is more involved in rounding downwards. Moreover, we need extra negations of the x_i.

However, still one switching of the rounding mode is necessary. Next we are interested in computing bounds without any switching of the rounding mode.

3. Interval bounds using rounding to nearest

The most obvious choice if the rounding mode shall not be altered seems rounding to nearest. An advantage of computational error bounds in rounding to nearest is that the rounding mode does not have to be changed at all, also not after finishing rigorous computations and returning to ordinary floating-point operations. Various interval libraries utilize the rigorous error bounds (3) provided by the IEEE 754 floating-point standard.

For example, it is easy to see [2] that $c = \mathrm{fl}_{\mathrm{near}}(a \circ b)$ implies

$$
\texttt{cinf} := \mathrm{fl}_{\mathrm{near}}((c - (2\,\texttt{Eps}) \cdot |c|) - \texttt{Eta}) \le a \circ b \le \mathrm{fl}_{\mathrm{near}}((c + (2\,\texttt{Eps}) \cdot |c|) + \texttt{Eta}) =: \texttt{csup}
$$

for $\circ \in \{+, -, \cdot, /\}$. This is used in Kearfott's interval arithmetic library INTLIB [2].

Obviously, the width of the interval $[\texttt{cinf}, \texttt{csup}]$ depends on whether c is in the left half or the right half of $[2^k, 2^{k+1}]$ for $2^k \le |c| < 2^{k+1}$, namely, the width of the result interval is 2 ulps or 4 ulps, respectively. This drawback can be overcome as follows.

If for $r \in \mathbb{R}$ the only available information is $c = \mathrm{fl}_{\mathrm{near}}(r)$, then we only know that r is bounded by the predecessor and the successor of c, so that the narrowest interval enclosing r is $[\mathrm{pred}(c), \mathrm{succ}(c)]$. In [3] we presented two algorithms to compute an enclosing interval for $a \circ b$. The first is as follows, using the constant $\phi_1 := \texttt{Eps}(1 + 2\,\texttt{Eps}) = \mathrm{succ}(\texttt{Eps})$:

Algorithm 4 (Rump et al. [3]) Enclosing interval of $a \circ b$ for $\circ \in \{+, -, \cdot, /\}$ using rounding to nearest.

$$
\begin{array}{ll}
\textbf{function} \;\; X = \texttt{BoundNear1}(a, b) & \\
\quad c = \mathrm{fl}_{\mathrm{near}}(a \circ b) & \\
\quad e = \mathrm{fl}_{\mathrm{near}}(\phi_1 \cdot |c| + \texttt{Eta}) & \texttt{\%}\ \phi_1 := \texttt{Eps}(1 + 2\,\texttt{Eps}) \\
\quad \texttt{cinf} = \mathrm{fl}_{\mathrm{near}}(c - e) & \\
\quad \texttt{csup} = \mathrm{fl}_{\mathrm{near}}(c + e) & \\
\quad X = [\texttt{cinf}, \texttt{csup}] &
\end{array}
$$

In [3] the following is proved.

Theorem 1 (Rump et al. [3]) For all finite $c \in \mathbb{F}$ with $|c| \notin [1, 4] \cdot \texttt{Fmin}$ the quantities `cinf` and `csup` computed by Algorithm 4 satisfy

$$
\texttt{cinf} = \mathrm{pred}(c) \qquad \text{and} \qquad \texttt{csup} = \mathrm{succ}(c). \tag{8}
$$

Remark 1 In IEEE 754 binary64 the excluded range is $[1,4] \cdot \texttt{Fmin} = [2^{-1022}, 2^{-1020}]$. In this range Algorithm 4 returns $\texttt{cinf} = \text{pred}(\text{pred}(c))$ and $\texttt{csup} = \text{succ}(\text{succ}(c))$, i.e., it is one bit off.

Remark 2 The assertions remain true for any tie-breaking rule in rounding to nearest.

This algorithm is branch-free and fast because only ordinary floating-point operations are used, and the rounding mode need not to be changed. The result interval $[\texttt{cinf}, \texttt{csup}]$ is 2-ulps wide, i.e. best possible with only the information $c = \text{fl}_{\text{near}}(a \circ b)$ at hand, except a tiny range near underflow.

On today's processors one observes that operations with quantities in the underflow range involved are significantly slower than ordinary floating-point operations[3]. This may slow down Algorithm 4 significantly because the smallest positive subnormal floating-point number `Eta` has always to be used in the computation of e.

The idea of the second algorithm is to scale the input off the underflow range if necessary. This solves two problems. First, an operation with quantities in the underflow range occurs only if the input c is in the underflow range. Second, the result interval is always 1 ulp wide without exception, and moreover the case distinction allows to use simpler formulas. However, the algorithm contains a branch, although almost always the first branch will be taken. The same constant $\phi_1 := \texttt{Eps}(1 + 2\,\texttt{Eps}) = \text{succ}(\texttt{Eps})$ as before is used.

Algorithm 5 (Rumip et al. [3]) Enclosing interval of $a \circ b$ for $\circ \in \{+, -, \cdot, /\}$ using rounding to nearest.

function $X = \texttt{BoundNear2}(a, b)$
  $c = \text{fl}_{\text{near}}(a \circ b)$
  if $|c| \geq \texttt{Eps}^{-1}\,\texttt{Fmin}$ **then**
    $e = \text{fl}_{\text{near}}(\phi_1 \cdot |c|)$ % $\phi_1 := \texttt{Eps}(1 + 2\,\texttt{Eps})$
    $\texttt{cinf} = \text{fl}_{\text{near}}(c - e)$
    $\texttt{csup} = \text{fl}_{\text{near}}(c + e)$
  elseif $|c| < 2\,\texttt{Fmin}$
    $\texttt{cinf} = \text{fl}_{\text{near}}(c - \texttt{Eta})$
    $\texttt{csup} = \text{fl}_{\text{near}}(c + \texttt{Eta})$
  else
    $C = \text{fl}_{\text{near}}(\texttt{Eps}^{-1} \cdot c)$ % scaled c
    $e = \text{fl}_{\text{near}}(\phi_1 \cdot |C|)$
    $\texttt{cinf} = \text{fl}_{\text{near}}(\texttt{Eps} \cdot (C - e))$
    $\texttt{csup} = \text{fl}_{\text{near}}(\texttt{Eps} \cdot (C + e))$
  end if
  $X = [\texttt{cinf}, \texttt{csup}]$

Then for all $c \in \mathbb{F}$ the quantities `cinf` and `csup` computed by Algorithm 5 satisfy

$$\texttt{cinf} = \text{pred}(c) \quad \text{and} \quad \texttt{csup} = \text{succ}(c)\ ,$$

hence the result interval $X = [\texttt{cinf}, \texttt{csup}]$ is always 2 ulps wide. With only the information $c = \text{fl}_{\text{near}}(a \circ b)$ at hand, this is best possible. However, we may ask whether 1-ulp wide intervals can be computed without switching the rounding mode. This is indeed possible using chop rounding.

4. Interval bounds using chopping

The chop mode has some interesting properties. For example, there is no overflow because a very large result in magnitude is rounded towards zero, i.e. to the largest magnitude positive or negative floating-point number. Moreover, the implementation is very simple because no care is necessary for some rounding bit: One just computes the leading bits, as many as given by the floating-point format, and discards the rest (see Fig. 2 for addition). This is one reason why both GPU and cell processors used chop rounding by default[4].

[3]On the other hand, it does not happen on today's GPU environments.
[4]Today's GPU processors use rounding to nearest by default.

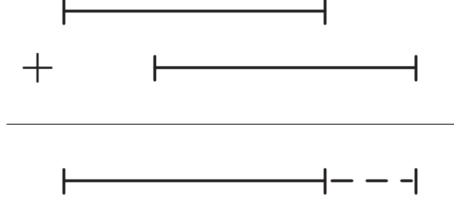

Fig. 2. Addition in chop mode.

Concerning interval bounds for $a \circ b$, we use $\mathbb{F} = -\mathbb{F}$ and have for $c := \mathrm{fl}_{\mathrm{chop}}(a \circ b)$,

$$\begin{array}{llll} a \circ b & \in & [c, \mathrm{succ}(c)] & \text{if } c > 0 \\ a \circ b & \in & [\mathrm{pred}(c), c] = [-\mathrm{succ}(-c), c] & \text{if } c < 0. \end{array}$$

So an enclosing interval for $a \circ b$ is known a priori without case distinction if $c \neq 0$. The only exception is $c = 0$, in which case the interval $[-\mathtt{Eta}, \mathtt{Eta}]$ is always correct but 2-bits wide; alternatively the sign of $a \circ b$ can be determined somehow. Thus for $c \neq 0$ it suffices to compute $\mathrm{succ}(c)$ for positive $c \in \mathbb{F}$ to obtain a 1-ulp interval for $a \circ b$.

In IEEE standard 754 [1] there is $\pm\infty$, so the successor of `Fmax` can be defined to be $+\infty$. However, the chop rounding avoids overflow. That means there is no way to produce ∞ as a result of a floating-point operation except the divide-by-zero. Hence always special care is necessary to check for $c = \mathtt{Fmax}$ or for overflow.

On cell processors [7, 9] the computation of the successor of a positive floating-point number is in particular simple because there is no gradual underflow. That means it suffices to compute the successor of a positive normalized floating-point number.

Theorem 2 Let $c \in \mathbb{F}$ be given and define

$$\mathtt{csup} = \mathrm{fl}_{\mathrm{chop}}(\phi_2 \cdot c) \qquad \text{for} \qquad \phi_2 := 1 + 2\,\mathtt{Eps} = \mathrm{succ}(1). \tag{9}$$

Then for all normalized positive $c \neq \mathtt{Fmax}$ we have $\mathtt{csup} = \mathrm{succ}(c)$.

Proof. By assumption we have $c \geq \mathtt{Fmin}$, and therefore $\mathrm{succ}(c) = c + 2\,\mathtt{Eps} \cdot \mathrm{ufp}(c)$ by (6). Hence $\mathrm{succ}(c) \leq \phi_2 c$, and $\mathrm{succ}(c) \in \mathbb{F}$ and chopping implies $\mathrm{succ}(c) \leq \mathtt{csup}$. Furthermore, chopping, (4) and (7) give

$$\mathrm{succ}(c) \leq \mathtt{csup} \leq (1 + 2\,\mathtt{Eps})c < c + 4\,\mathtt{Eps} \cdot \mathrm{ufp}(c) \leq \mathrm{succ}(\mathrm{succ}(c)) \ ,$$

and the result follows. □

To compute the successor of a floating-point number in IEEE 754 arithmetic including gradual underflow is a little more involved. One way is to check for underflow as follows [8], using the constant ϕ_2 as in Theorem 2:

Algorithm 6 Computation of the successor of $0 \leq c \in \mathbb{F} \backslash \mathtt{Fmax}$ in chop rounding.

> **function** $\mathtt{csup} = \mathtt{SuccChop1}(c)$
> $\quad \mathtt{csup} = \max\big(\mathrm{fl}_{\mathrm{chop}}(\phi_2 \cdot c), \mathrm{fl}_{\mathrm{chop}}(c + \mathtt{Eta})\big)$ % $\phi_2 := 1 + 2\,\mathtt{Eps}$

The correctness follows by Theorem 2. However, as mentioned before, the operation involving `Eta` may slow down the algorithm significantly. This is the more annoying as the maximum in Algorithm 6 will almost always be the number $\mathrm{fl}_{\mathrm{chop}}(\phi_2 \cdot c)$. An alternative along the lines of Algorithm 5 may be the following.

Algorithm 7 Computation of the successor of $0 \leq c \in \mathbb{F} \backslash \mathtt{Fmax}$ in chop rounding.

> **function** $\mathtt{csup} = \mathtt{SuccChop2}(c)$
> $\quad$ **if** $c \geq \mathtt{Fmin}$ **then**
> $\quad\quad \mathtt{csup} = \mathrm{fl}_{\mathrm{chop}}(\phi_2 \cdot c)$ % $\phi_2 := 1 + 2\,\mathtt{Eps}$
> $\quad$ **else**
> $\quad\quad \mathtt{csup} = \mathrm{fl}_{\mathrm{chop}}(c + \mathtt{Eta})$
> $\quad$ **endif**

This variant uses quantities in the underflow range only if necessary; however, it contains a branch. A third variant computes the successor without branch, however the underflow unit Eta is again involved. It is as follows.

Algorithm 8 Computation of the successor of $0 \le c \in \mathbb{F} \backslash \texttt{Fmax}$ in chop rounding.

$$\textbf{function}\ \ \texttt{csup} = \texttt{SuccChop3}(c)$$
$$\texttt{csup} = \text{fl}_{\text{chop}}(c + (2 \cdot (\texttt{Eps} \cdot c) + \texttt{Eta}))$$

Theorem 3 Let $c \in \mathbb{F}$ be given, and suppose csup is computed by Algorithm 8. Then for all $c \ge 0$, $c \ne \texttt{Fmax}$, we have

$$\texttt{csup} = \text{succ}(c).$$

PROOF. Depict the intermediate steps as follows.

$$\begin{aligned} x &= \text{fl}_{\text{chop}}(\texttt{Eps} \cdot c) \\ y &= \text{fl}_{\text{chop}}(2 \cdot x) \\ e &= \text{fl}_{\text{chop}}(y + \texttt{Eta}) \\ \texttt{csup} &= \text{fl}_{\text{chop}}(c + e) \end{aligned}$$

We distinguish two main cases. First, suppose c is near the underflow range, more precisely $0 \le c < 2\,\texttt{Fmin}$. Then $\texttt{Eps}\, c < \texttt{Eta}$ and $x = y = 0$, $e = \texttt{Eta}$ and $\texttt{csup} = c + \texttt{Eta} = \text{succ}(c)$.

Second, assume $c \ge 2\,\texttt{Fmin}$. Then $\text{succ}(c) = c + 2\,\texttt{Eps} \cdot \text{ufp}(c)$ by (6). As a first sub-case assume $c \ge \texttt{Eps}^{-1}\,\texttt{Fmin}$. Then (5) implies

$$c \in 2\,\texttt{Eps} \cdot \text{ufp}(c)\mathbb{Z} \subseteq 2\,\texttt{Fmin}\,\mathbb{Z},$$

so that $x = \texttt{Eps}\, c$ and $y = 2\,\texttt{Eps}\, c$. The assumption $c \ge 2\,\texttt{Fmin}$ implies $\text{succ}(y) \ge y + 2\,\texttt{Eta}$, so that the chop rounding and (4) imply $e = 2\,\texttt{Eps}\, c < 4\,\texttt{Eps} \cdot \text{ufp}(c)$. With (7) we obtain

$$\text{succ}(c) = c + 2\,\texttt{Eps} \cdot \text{ufp}(c) \le c + 2\,\texttt{Eps} \cdot c = c + e < c + 4\,\texttt{Eps} \cdot \text{ufp}(c) \le \text{succ}(\text{succ}(c))\ .$$

It remains the second sub-case $2\,\texttt{Fmin} \le c < \texttt{Eps}^{-1}\,\texttt{Fmin}$. Define $m \in \mathbb{N}$ and $\kappa \in \mathbb{R}$ by

$$c = 2(m + \kappa)\texttt{Fmin} \qquad \text{with}\ \ 1 \le m \le \frac{1}{2}\texttt{Eps}^{-1} - 1\ \ \text{and}\ 0 \le \kappa < 1\ .$$

Note that $m \in \mathbb{F}$. Furthermore, let $k \in \mathbb{N}$ such that

$$2^k \le m \le 2^{k+1} - 1\ .$$

Then

$$\text{ufp}(c) = 2^{k+1}\,\texttt{Fmin} \qquad \text{and} \qquad \text{succ}(c) = c + 2^{k+1}\,\texttt{Eta}\ .$$

It follows $x = \text{fl}_{\text{chop}}(\texttt{Eps} \cdot c) = \text{fl}_{\text{chop}}((m + \kappa) \cdot \texttt{Eta}) = m\,\texttt{Eta}$ and $y = 2m\,\texttt{Eta}$. Furthermore, $2m + 1 < \texttt{Eps}^{-1}$, so that $y + \texttt{Eta} < 2\,\texttt{Fmin}$ and $y + \texttt{Eta} \in \mathbb{F}$, which means $e = (2m + 1)\,\texttt{Eta} < 2^{k+2}\,\texttt{Eta}$. Hence (7) gives

$$\text{succ}(c) = c + 2^{k+1}\,\texttt{Eta} \le c + 2m\,\texttt{Eta} < c + e < c + 2^{k+2}\,\texttt{Eta} = c + 4\,\texttt{Eps} \cdot \text{ufp}(c) \le \text{succ}(\text{succ}(c))\ ,$$

and the chop rounding implies $\texttt{csup} = \text{succ}(c)$ and finishes the proof. □

Formula (9) in Theorem 2 computes the successor of a *normalized* floating-point number without exception over the whole nonnegative range. It requires 1 flop. Algorithm 8 does the same including subnormal floating-point numbers, but requires 4 flops. If one accepts a small range near underflow where, as before, $\text{succ}(\text{succ}(c))$ is computed rather $\text{succ}(c)$, then 2 flops suffice as follows.

Algorithm 9 Computation of the successor of $0 \le c \in \mathbb{F} \backslash \texttt{Fmax}$ in chop rounding.

$$\textbf{function}\ \ \texttt{csup} = \texttt{SuccChop4}(c)$$
$$\texttt{csup} = \text{fl}_{\text{chop}}(\phi_2 \cdot c + \texttt{Eta}) \qquad \%\ \phi_2 := 1 + 2\,\texttt{Eps}.$$

Table I. The methods for calculating the successor in chop rounding with considering the possible gradual underflow.

Method	flops	branches	maximum	always using Eta	always 1 ulp
Algorithm 6	2	–	1	yes	yes
Algorithm 7	1	1	–	no	yes
Algorithm 8	4	–	–	yes	yes
Algorithm 9	2	–	–	yes	no

Theorem 4 Let $c \in \mathbb{F}$ be given, and suppose $\texttt{csup}$ is computed by Algorithm 9. Then $\texttt{csup} = \mathrm{succ}(c)$ for all $c \geq 0$, $c \neq \texttt{Fmax}$, except $c \in E := [1, \mathrm{pred}(\mathrm{pred}(2))] \cdot \texttt{Fmin}$.

PROOF. For $0 \leq c < \texttt{Fmin}$ we have $\mathrm{succ}(c) = c + \texttt{Eta}$ and $2\,\texttt{Eps}\,c < \texttt{Eta}$, and therefore $\mathrm{fl}_{\mathrm{chop}}(\phi_2 \cdot c) = c$. Hence $\texttt{csup} = \mathrm{succ}(c)$ in that case.

For $c \in E$ we know $\mathrm{fl}_{\mathrm{chop}}(\phi_2 \cdot c) = \mathrm{succ}(c)$ by Theorem 2. So $\mathrm{succ}(c) \leq \mathrm{pred}(2)\texttt{Fmin}$ implies $\mathrm{succ}(\mathrm{succ}(c)) = \mathrm{succ}(c) + \texttt{Eta} = \texttt{csup}$ in that case. Finally, assume $c \geq \mathrm{pred}(2)\texttt{Fmin}$. Then

$$x := \mathrm{fl}_{\mathrm{chop}}(\phi_2 \cdot c) = \mathrm{succ}(c) \geq 2\,\texttt{Fmin}$$

again by Theorem 2, and $\mathrm{succ}(x) \geq x + 2\,\texttt{Eta}$. The result follows. □

The methods for computing the successor in chop rounding with considering the possible gradual underflow are summarized as in Table I. As mentioned before, using the smallest positive subnormal floating-point number Eta may slow down the computational speed significantly. We will see it from the numerical results in the next section.

As mentioned before, chopping implies a particular difficulty if the result $c = \mathrm{fl}_{\mathrm{chop}}(a \circ b)$ of a floating-point operation is zero. Unless further checking on the operands a, b is performed, it is not a priori clear whether the true result $a \circ b$ is negative, truly zero or positive. This suggests the following algorithm to compute interval bounds for a floating-point operation $a \circ b$.

Algorithm 10 Narrow enclosing interval of $a \circ b$ for $\circ \in \{+, -, \cdot, /\}$ using chopping.

function $X = \texttt{BoundChop}(a, b)$
  $c = \mathrm{fl}_{\mathrm{chop}}(a \circ b)$
  if $c > 0$ **then**
    if $c \neq \texttt{Fmax}$ **then**
      $X = [c, \mathrm{succ}(c)]$
    else
      $X = [\texttt{Fmax}, \infty]$
    end if
  elseif $c < 0$
    if $c \neq -\texttt{Fmax}$ **then**
      $X = [-\mathrm{succ}(-c), c]$
    else
      $X = [-\infty, -\texttt{Fmax}]$
    end if
  else
    $X = [-\texttt{Eta}, \texttt{Eta}]$
  end if

Of course, Algorithm 10 is easily adapted to the square root. All presented possibilities for $\mathrm{succ}(\cdot)$ may be used in Algorithm 10. Then we have the following result.

Theorem 5 Assume $a, b \in \mathbb{F}$ and $|\mathrm{fl}_{\mathrm{chop}}(a \circ b)| \neq \texttt{Fmax}$. Then the following are true:

i) If $\mathrm{succ}(\cdot)$ is computed by Algorithm 8, then the result interval X is always 1-bit wide, i.e. best possible unless $a \circ b \in \mathbb{F}$.

Table II. Testing environments.

	Processor	Compiler, compiler options
I)	CPU: Intel Core i7 (1.7 GHz)	Intel C++ Compiler version 15.0.3
	L2: 256 KB, L3: 4MB	`icc -fast -fp-model strict`
II)	GPU: Tesla K40 (745 MHz)	NVCC: CUDA compilation tools, version 6.5.12
	(single thread use)	`nvcc`

ii) If there is no gradual underflow and $\mathrm{succ}(\cdot)$ is computed by (9), then the result interval is always 1-bit wide, i.e. best possible unless $a \circ b \in \mathbb{F}$.

Remark 3 To avoid the presence of subnormal numbers, one may replace `Eta` by `Fmin` in the algorithms, though it may widen the result intervals in the case where c is near or within underflow range, i.e. $|c| < \mathtt{Eps}^{-1}\,\mathtt{Fmin}$.

As a conclusion, we have several possibilities to compute narrow, often best possible bounds for $a \circ b$ without switching the rounding mode. This may be particularly useful on processors such as GPU and cell processors. Next we compare the approaches in a practical application, namely the computation of the range of functions.

5. Computation of the range of functions

We present numerical results by applying the different approaches of interval arithmetic presented in this paper to the evaluation of some common test functions for optimization methods.

We compare the following implementations of interval arithmetic:

Method	
`UpDown`	a traditional approach, similar to Algorithm 1 (`BoundUpDown`)
`Up`	similar to Algorithm 2 (`BoundMultUp`)
`Near1`	similar to Algorithm 4 (`BoundNear1`)
`Near2`	similar to Algorithm 5 (`BoundNear2`)
`Chop1`	similar to Algorithm 10 (`BoundChop`) with Algorithm 6 (`SuccChop1`)
`Chop2`	similar to Algorithm 10 (`BoundChop`) with Algorithm 7 (`SuccChop2`)
`Chop3`	similar to Algorithm 10 (`BoundChop`) with Algorithm 8 (`SuccChop3`)
`Chop4`	similar to Algorithm 10 (`BoundChop`) with Algorithm 9 (`SuccChop4`)

All algorithms are tested in two different environments, namely Core i7 (CPU) and Tesla K40 (GPU), see Table II. All the floating-point computations are performed in IEEE 754 binary64 (double precision). We carefully choose compiler options to achieve best possible results, see Table II. Following are remarks on the CPU environment:

- We should be careful to make the computations conform to IEEE 754 arithmetic with validating the change of rounding mode, for example, by setting suitable compiler options.
- The change of rounding mode slows down the computational speed significantly.
- For the operation $c = \mathrm{fl}(a \circ b)$ for $a, b \in \mathbb{F}$, $\circ \in \{\cdot, /\}$, if c is a subnormal number ($0 \neq |c| < \mathtt{Fmin}$), it requires much more computing time than an ordinary case where c is a normalized number.

On the other hand, following are remarks on the GPU environment:

- The change of rounding mode does not affect the computational speed, since every rounding mode can explicitly be specified with the arithmetic itself, for example, a floating-point addition $\mathrm{fl}(a + b)$ for $a, b \in \mathbb{F}$ in rounding to nearest, rounding upwards, rounding downwards, and rounding towards zero can be executed by the instructions `_dadd_rn`(a, b), `_dadd_ru`(a, b), `_dadd_rd`(a, b), and `_dadd_rz`(a, b), respectively.
- No slowdown in the speed of floating-point arithmetic occurs in the presence of subnormal numbers.

Table III. Measured computing times (sec) for computing the successor on the CPU environment, 10^6 elements, 1000 iterations.

Case	`SuccChop1`	`SuccChop2`	`SuccChop3`	`SuccChop4`
Case 1: $c_i \approx 1$	1.35	1.47	1.36	1.35
Case 2: $c_i \approx 2^{-1000}$	1.34	1.46	25.98	1.34
Case 3: $c_i \approx 2^{-1030}$	13.34	1.42	13.29	13.15

Table IV. Measured computing times (sec) for computing the successor on the GPU environment, 10^6 elements, 1000 iterations.

Case	`SuccChop1`	`SuccChop2`	`SuccChop3`	`SuccChop4`
Case 1: $c_i \approx 1$	0.34	0.33	0.34	0.31
Case 2: $c_i \approx 2^{-1000}$	0.34	0.33	0.34	0.31
Case 3: $c_i \approx 2^{-1030}$	0.34	0.33	0.34	0.31

Before evaluating the range of functions, we compare Algorithms 6–9 (`SuccChop1`–`SuccChop4`) for computing the successor of a floating-point number as summarized in Table I. We generate pseudo-random numbers c_i, $1 \le i \le 10^6$, with three specified ranges:

Case 1. $c_i \approx 1$ for all i (normalized floating-point numbers)

Case 2. $c_i \approx 2^{-1000}$ for all i (normalized floating-point numbers, but near underflow range)

Case 3. $c_i \approx 2^{-1030}$ for all i (subnormal floating-point numbers)

The results are displayed in Tables III and IV.

From Table III, we can observe the significant slowdown in the methods except the method `SuccChop2` for near underflow range (Cases 2 and 3) on the CPU environment, namely, they are about 10 times slower than the method `SuccChop2` in some cases. On the GPU environment, there are not so much differences among the methods. Since there is no branch nor maximum in the method `SuccChop4`, it is slightly faster than the others. In any case, the method `SuccChop1` is faster than or as fast as `SuccChop3` on both environments. Therefore, we omit the results for `SuccChop3` in the rest of this section.

Now we start to evaluate the range of functions. First, we evaluate the following multivariate Shekel function [5]:

$$f(x) = -\sum_{j=1}^{m} \left[\sum_{i=1}^{n} (x_i - a_{ji})^2 + c_j \right]^{-1}, \quad x \in \mathbb{R}^n.$$

This function is frequently used to define the so-called STU (Standard Unit Time) in order to compare the performance of optimization methods. We choose some standard setting for the Shekel function, that is $n = 10$ and $m = 4$ with the following parameters:

$$A = \begin{pmatrix} 4 & 1 & 8 & 6 & 3 & 2 & 5 & 8 & 6 & 7 \\ 4 & 1 & 8 & 6 & 7 & 9 & 3 & 1 & 2 & 3.6 \\ 4 & 1 & 8 & 6 & 3 & 2 & 5 & 8 & 6 & 7 \\ 4 & 1 & 8 & 6 & 7 & 9 & 3 & 1 & 2 & 3.6 \end{pmatrix}$$

and

$$c = \frac{1}{10}(1, 2, 2, 4, 4, 6, 3, 7, 5, 5).$$

We use the standard search domain as $0 \le x_i \le 10$ for $1 \le i \le 4$. Then the global minimum of f is $f(x^*) = -10.536283726219\cdots$ at $x^* = (4, 4, 4, 4)$. Note that we take into account the rounding errors when constructing A and c, which are represented as interval data in the experiments.

We first compute the verified range of $f(x)$ with an input interval vector $[x_i] = [0, 10]$, $1 \le i \le 4$. The result is displayed in Table V. We also measure the diameter of the result interval for the global minimum of $f(x)$ with an input interval vector $[x_i] = [\mathrm{pred}(4), \mathrm{succ}(4)]$, $1 \le i \le 4$. Computing times in both cases are displayed in Tables VII and VIII.

Table V. Verified range of the Shekel function $f(x)$ with an input interval vector $[x_i] = [0, 10]$, $1 \le i \le 4$.

Method	Verified range of $f(x)$	Diameter
`UpDown`, `Up`	$[-35.4285714285714\underline{38}, -0.050098570050846\underline{22}]$	$35.3784728585206\underline{0}$
`Near1`, `Near2`	$[-35.4285714285714\underline{81}, -0.050098570050846\underline{14}]$	$35.3784728585206\underline{4}$
`Chop1`, `Chop2`, `Chop4`	$[-35.4285714285714\underline{60}, -0.050098570050846\underline{19}]$	$35.3784728585206\underline{2}$

Table VI. Inclusion of the global minimum of the Shekel function $f(x)$ with an input interval vector $x_i = [\mathrm{pred}(4), \mathrm{succ}(4)]$, $1 \le i \le 4$.

Method	Diameter
`UpDown`, `Up`	2.31×10^{-14}
`Near1`, `Near2`	4.27×10^{-14}
`Chop1`, `Chop2`, `Chop4`	2.67×10^{-14}

Table VII. Measured computing times (sec) for the Shekel function $f(x)$ for an input interval vector on the CPU environment, 10^6 iterations.

Case	`UpDown`	`Up`	`Near1`	`Near2`	`Chop1`	`Chop2`	`Chop4`
$[x_i] = [0, 10]$	31.91	0.13	1.85	0.37	1.85	0.41	1.84
$[x_i] = [\mathrm{pred}(4), \mathrm{succ}(4)]$	34.69	0.13	0.59	0.41	0.49	0.32	0.46

Table VIII. Measured computing times (sec) for the Shekel function $f(x)$ for an input interval vector on the GPU environment, 10^5 iterations.

Case	`UpDown`	`Up`	`Near1`	`Near2`	`Chop1`	`Chop2`	`Chop4`
$[x_i] = [0, 10]$	3.00	2.96	3.27	4.56	5.92	6.17	5.46
$[x_i] = [\mathrm{pred}(4), \mathrm{succ}(4)]$	2.75	2.76	2.96	4.29	4.96	5.17	4.62

Table IX. Verified range of the Rosenbrock function $g(x)$ with an input interval vector $[x_i] = [\mathrm{pred}(1), \mathrm{succ}(1)]$, $1 \le i \le n$, $n = 1000$.

Method	Verified range of $g(x)$
`UpDown`, `Up`	$[0, 6.0386020395\underline{69747} \times 10^{-26}]$
`Near1`, `Near2`	$[0, 6.0386020395\underline{70732} \times 10^{-26}]$
`Chop1`, `Chop2`, `Chop4`	$[0, 6.0386020395\underline{70731} \times 10^{-26}]$

Table X. Measured computing times (sec) for the Rosenbrock function $g(x)$ for an input interval vector $[x_i] = [\mathrm{pred}(1), \mathrm{succ}(1)]$, $1 \le i \le n$, $n = 1000$ on the CPU environment, 10^5 iterations.

`UpDown`	`Up`	`Near1`	`Near2`	`Chop1`	`Chop2`	`Chop4`
198.93	0.63	2.16	1.82	2.05	1.94	1.92

Table XI. Measured computing times (sec) for the Rosenbrock function $g(x)$ for an input interval vector $[x_i] = [\mathrm{pred}(1), \mathrm{succ}(1)]$, $1 \le i \le n$, $n = 1000$ on the GPU environment, 10^3 iterations.

`UpDown`	`Up`	`Near1`	`Near2`	`Chop1`	`Chop2`	`Chop4`
0.63	0.64	0.89	1.62	2.09	2.19	1.80

The next test function is the extended Rosenbrock function

$$g(x) = \sum_{i=1}^{n-1} \left[(1 - x_i)^2 + 100(x_{i+1} - x_i^2)^2 \right], \quad x \in \mathbb{R}^n,$$

which is a non-convex function also widely used for optimization methods [6]. The trivial global minimum of g is $g(x^*) = 0$ at $x^* = (1, \ldots, 1)$.

We measure the verified range of $g(x)$ with an input interval vector $[x_i] = [\mathrm{pred}(1), \mathrm{succ}(1)]$, $1 \le i \le n$ for $n = 1000$, which should contain the global minimum. The result is displayed in Table IX. Moreover, computing times are displayed in Tables X and XI

From the results, we can confirm the following facts:

- As by the theory, the methods `Up` and `UpDown` always give the narrowest results, and the methods `Chop1`, `Chop2` and `Chop4` do the second narrowest ones, though there are not so much differences among all the methods in terms of the width of result intervals.
- All the methods that can be performed with fixed rounding mode are much faster than the traditional approach `UpDown` on the CPU environment, especially the method `Up` is considerably faster than the others, while there are not so much differences among all the methods in terms of computational speed on the GPU environment.

It turns out that interval arithmetic with fixed rounding mode is useful in terms of computational speed, if the change of rounding mode slows down it. Since the quality of the result with every rounding mode is comparable, we can adapt the implementation of interval arithmetic to the computational environment for any purpose as long as it supports correct rounding.

Acknowledgments

Part of this work was done during a stay of the first author as visiting professor at the university of Paris VI. My dearest thanks go to Jean-Marie Chesneaux and his team for their kind hospitality in Paris. In particular many thanks to Stef Graillat for detailed and fruitful discussions, and for looking at earlier versions of this paper. The second author likes to express his sincere thanks to Katsuhisa Ozaki and Naoya Yamanaka for their stimulating discussions. This research was partially supported by CREST, Japan Science and Technology Agency (JST).

References

[1] *IEEE Std 754-2008 (Revision of IEEE Std 754-1985): IEEE Standard for Floating Point Arithmetic*, IEEE, New York, 2008.

[2] R.B. Kearfott, M. Dawande, and C. Hu, "INTLIB: A portable Fortran-77 interval standard function library," *ACM Trans. Math. Software*, vol. 20, pp. 447–459, 1994.

[3] S.M. Rump, P. Zimmermann, S. Boldo, and G. Melquiond, "Computing predecessor and successor in rounding to nearest," *BIT Numerical Mathematics*, vol. 49, no. 2, pp. 419–431, 2009.

[4] S.M. Rump, T. Ogita, and S. Oishi, "Accurate floating-point summation Part I: Faithful rounding," *SIAM J. Sci. Comput.*, vol. 31, no. 1, pp. 189–224, 2008.

[5] J. Shekel, "Test functions for multimodal search techniques," *Fifth Annual Princeton Conference on Information Science and Systems*, 1971.

[6] Y.-W. Shang and Y.-H. Qiu, "A note on the extended Rosenbrock function," *Evol. Comput.*, vol. 14, no. 1, pp. 119–126, 2006.

[7] C. Jacobi, H.-J. Oh, K.D. Tran, S.R. Cottier, B.W. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, and N. Yano, "The vector floating-point unit in a synergistic processor element of a Cell processor," *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH '05)*, pp. 59–67, IEEE Computer Society, Washington, DC, USA, 2005.

[8] S. Graillat, private communication.

[9] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 589–604, 2005.