Invited Paper

The Essentials of verified numerical computations, rounding error analyses, interval arithmetic, and error-free transformations

Katsuhisa Ozaki\textsuperscript{1a)} and Takeshi Ogita\textsuperscript{2}

\textsuperscript{1} Department of Mathematical Sciences, Shibaura Institute of Technology, 307 Fukasaku, Minuma-ku, Saitama-shi, Saitama 337–8570, Japan

\textsuperscript{2} Division of Mathematical Sciences, Tokyo Woman’s Christian University, 2-6-1 Zempukuji, Suginami-ku, Tokyo 167–8585, Japan

\textsuperscript{a)} ozaki@sic.shibaura-it.ac.jp

Received February 15, 2020; Revised March 28, 2020; Published July 1, 2020

Abstract: Floating-point numbers and floating-point arithmetic are widely used in numerical computations. A treatable problem size can quickly become large-scale due to the continual advancement of computational environments. If the number of floating-point operations increases, then problems caused by rounding errors become increasingly critical. In the worst case, an approximate solution obtained by a numerical computation can be inaccurate. Therefore, verified numerical computations are becoming increasingly important. This paper presents a survey of the basics related to verified numerical computations. We focus on floating-point arithmetic, interval arithmetic, rounding error analyses, and error-free transformations of floating-point operations.

Key Words: floating-point arithmetic, interval arithmetic, error-free transformation

1. Introduction

This paper introduces the basics of floating-point numbers, floating-point arithmetic, interval arithmetic, rounding error analyses, and error-free transformations of floating-point numbers. This paper contains Chapters 1 and 2 in [1], which was written in Japanese.

Floating-point arithmetic is performed quickly on modern Central Processing Units (CPUs). The performance of floating-point arithmetic has rapidly increased due to the progress of CPUs following Moore’s law. Hardware-supported binary floating-point numbers, binary32 or binary64, are usually used for numerical computations. This situation has not changed over the last 30 years. Even though the IEEE754-2008 standard [2] defines a floating-point number with 128 bits (binary128), this type of number is not supported on most recent CPUs but, instead has been emulated by software. Accordingly, its performance is very low compared to those of binary32 and binary64. In other words,
the number of floating-point operations has increased, while the precision of floating-point numbers has remained unchanged. This fact makes the problem of rounding errors even more critical.

On today’s computers, the following two types of computations are widely used to solve problems.

Symbolic Computations: We can treat any real numbers using symbolic computations, and the results of such computations are rigorous. However, there are many unsolved problems. In addition, if a given problem is large-scale, symbolic computations cannot be applied to the problem due to low performance.

Numerical Computations: Numerical computations are widely applied to numerical approaches, and using such computations, we can quickly obtain an approximate solution to a problem. However, the accuracy of the approximation is unknown in many cases.

Verification methods [3, 4] work as a compromise between symbolic and numerical computations. Verification methods provide an error bound for an approximate solution using only floating-point arithmetic. Figure 1 shows an image depicting approximations and their error bounds in the complex plane. The exact solutions exist in the circles at the centers \(x_1\) and \(x_2\), and their radii are \(e_1\) and \(e_2\), respectively. Verification methods aim to produce such circles quickly. If \(x_1\) and \(x_2\) are approximate eigenvalues of a matrix and the circles do not overlap, then the two eigenvalues are not multiple. Guaranteeing that there are no multiple eigenvalues is important for large-scale electronic state calculations [5].

This paper is organized as follows. We introduce the basic of floating-point numbers and their arithmetic in Section 2 and the basics of interval arithmetic in Section 3. We summarize rounding error analyses in Section 4. Finally, we introduce error-free transformations for floating-point operations and accurate numerical algorithms for the summation of floating-point numbers in Section 5.

2. Floating-point number and arithmetic

In this section, we introduce floating-point numbers and their arithmetic, which is defined in IEEE 754-2008. IEEE 754-2008 defines the precisions of floating-point numbers such as binary32 (single precision), binary64 (double precision), and binary128 (quadruple precision). The numbers after the word “binary” indicate the number of bits allocated for the representation of the floating-point numbers. There are normal numbers, subnormal numbers, zero, \(\text{Inf}\), \(-\text{Inf}\) in floating-point numbers, and \(\text{NaN}\) (Not a Number)\(^1\). Let \(\mathbb{F}\) be the set of union of normal floating-point numbers, subnormal floating-point numbers, and zero with a specified precision (\(\mathbb{F} \subseteq \mathbb{R}\)). Then, \(\mathbb{F} = -\mathbb{F}\). We define \(\mathbb{F}_* = \mathbb{F} \cup \{\text{Inf}, -\text{Inf}\}\).

2.1 Floating-point number

We explain a format for floating-point numbers and the relative error \(\delta\)

---

\(^1\)binary32 and binary64 were called single precision and double precision in IEEE 754–1985. Subnormal numbers were introduced as denormalized numbers in IEEE 754–1985.
\[ \frac{|b - a|}{|a|} = \delta, \quad a \neq 0 \]

between \( a \in \mathbb{R} \) and its nearest floating-point number \( b \in \mathbb{F} \). A normal or subnormal floating-point number \( a \in \mathbb{F} \) is represented as

\[ a = s \cdot f \cdot 2^e, \quad s = \pm 1, \quad f = \sum_{i=0}^{p-1} d_i \cdot 2^i, \quad d_i \in \{0, 1\}, \quad (1) \]

where \( s \) is the sign, \( f \) is the significand, and \( e \) is the exponent. Let \( p \) be the precision of a floating-point number. In other words, \( p \) is the number of bits in the significand, including the implicit bit. The precision \( p \) depends on the format: for example, \( p = 24 \) for binary32 and \( p = 53 \) for binary64. The range of the exponent \( e \in \mathbb{Z} \) is \( E_{\text{min}} \leq e \leq E_{\text{max}} \), for example, \( (E_{\text{min}}, E_{\text{max}}) = (-126, 127) \) for binary32 and \( (E_{\text{min}}, E_{\text{max}}) = (-1022, 1023) \) for binary64. We omit an explanation for the bias expression of the exponent because it is not related to the rounding error analyses in this paper. It is satisfied that \( d_0 = 1 \) for a normal number, and \( d_0 = 0 \) and \( e = E_{\text{min}} \) for a subnormal number. We also omit an explanation of the representation of 0, \( \text{Inf} \), \( -\text{Inf} \), and \( \text{NaN} \).

We define a function for the Unit in the First Place (ufp). The function \( \text{ufp}(a) \) for \( a \in \mathbb{R} \) is defined as

\[ \text{ufp}(a) := \begin{cases} 2^{|\log_2 |a||} & (a \neq 0) \\ 0 & (a = 0) \end{cases} \quad (2) \]

\( \text{ufp}(a) \) produces \( 2^e \) in the form shown in (1). For example, \( \text{ufp}(3.5) = \text{ufp}(2^1 + 2^0 + 2^{-1}) = 2 \) and \( \text{ufp}(0.625) = \text{ufp}(2^{-1} + 2^{-8}) = 2^{-1} \). From (2), if \( a \neq 0 \), there exists \( k \in \mathbb{Z} \) such that \( \text{ufp}(a) = 2^k \), and

\[ \text{ufp}(a) \leq |a| < 2\text{ufp}(a) \]

is satisfied.

Let \( u = 2^{-p} \) be the unit roundoff, for example, \( u = 2^{-24} \) for binary32 and \( u = 2^{-53} \) for binary64. The unit roundoff plays an important role in rounding error analyses. If we set \( s = 1, d_0 = 1, d_i = 0 \) \((1 \leq i \leq p - 1)\), and \( e = E_{\text{min}} \) in (1), then we obtain the minimum positive number \( F_{\text{min}} \) in normal floating-point numbers \( (F_{\text{min}} = 2^{E_{\text{min}}}) \): in practice, \( F_{\text{min}} = 2^{-126} \) for binary32 and \( F_{\text{min}} = 2^{-1022} \) for binary64.

Let the smallest positive number in \( \mathbb{F} \) be \( S_{\text{min}} \). We obtain \( S_{\text{min}} \) setting \( s = 1, d_0 = 0 \) \((0 \leq i \leq p - 2)\), \( d_{p-1} = 1 \), and \( e = E_{\text{min}} \) in (1), for example, \( S_{\text{min}} = 2^{-149} \) for binary32 and \( S_{\text{min}} = 2^{-1074} \) for binary64. \( S_{\text{min}} = 2u \cdot F_{\text{min}} \) holds true. The maximum floating-point number \( F_{\text{max}} \) is obtained by setting \( s = 1, d_i = 1 \) \((0 \leq i \leq p - 1)\), \( e = E_{\text{max}} \) in (1): in practice, \( F_{\text{max}} = 2^{128}(1 - 2^{-24}) \approx 3.40 \times 10^{38} \) for binary32 and \( F_{\text{max}} = 2^{1024}(1 - 2^{-53}) \approx 1.79 \times 10^{308} \) for binary64. In general, \( F_{\text{max}} = 2^{E_{\text{max}} + 1}(1 - u) \).

The next theorem determines whether a real number is an element of \( \mathbb{F} \). Note that \( r \mathbb{Z} \) for \( r \in \mathbb{R} \) implies a multiple of \( r \).

**Theorem 2.1** a \( \in \mathbb{F} \) if and only if

\[ a \in 2u \cdot \text{ufp}(a) \mathbb{Z}, \quad |a| \leq F_{\text{max}}, \quad a \in S_{\text{min}} \mathbb{Z}. \quad (3) \]

We omit the proof of this theorem: however, the consequence of (3) are as follows:

- The required length of the significand bits to represent \( a \) is sufficient:
- |\( a \)| is not too large: and
- \( a \) is a multiple of \( S_{\text{min}} \).

Here, \( 2u \cdot \text{ufp}(a) \) is known as the Unit in the Last Place (ulp) for \( a \in \mathbb{F} \) (Fig. 2). We can define \( \text{ulp} \) for a real number; however, \( \text{ulp} \) cannot be defined for a real number.

The following theorem gives a sufficient condition for \( a \in \mathbb{F} \).
Theorem 2.2 For $a \in \mathbb{R}$, assume that $|a| \leq F_{\text{max}}$ and $a \in S_{\text{min}} \mathbb{Z}$. If there is $\sigma \in \mathbb{R}$ such that $|a| \leq \sigma = 2^k$, $k \in \mathbb{Z}$, and $a \in u\sigma\mathbb{Z}$, then $a \in \mathbb{F}$.

In numerical computations, we usually use floating-point numbers. Here the point is that floating-point numbers are a subset of real numbers. We define a map $\text{RN} : \mathbb{R} \to \mathbb{F}$, where a given real number is rounded to the nearest floating-point number. That is, $\text{RN}(a)$ produces $b \in \mathbb{F} \cup \{\pm \text{Inf}\}$ from $a \in \mathbb{R}$ such that

$$|a - \text{RN}(a)| \leq u \cdot \text{ufp}(a)$$

is satisfied.

Proof We can find floating-point neighbors $b, c \in \mathbb{F}$, such that $\text{ufp}(a) = \text{ufp}(b)$ and $b \leq a \leq c$. Because $c = b + 2u \cdot \text{ufp}(b) = b + 2u \cdot \text{ufp}(a)$ and $\text{RN}(a)$ is the nearest floating-point number to $a$, the maximum absolute error is half of $2u \cdot \text{ufp}(a)$.

The equality in (4) can be used only when $a$ is the midpoint of floating-point neighbors. In reality, we can check the equality by setting $a = 1 + u$.

The following theorems show the relationships between $a \in \mathbb{R}$ and $\text{RN}(a)$.

Theorem 2.3 (Rump, Ogita, and Oishi [6]) For $a \in \mathbb{R}$ satisfying $F_{\text{min}} \leq |a| < R_{\text{sup}}$, $|a - \text{RN}(a)| \leq u \cdot \text{ufp}(a)$.

Theorem 2.4 (Jeannerod and Rump [7]) For $a \in \mathbb{R}$ satisfying $F_{\text{min}} \leq |a| < R_{\text{sup}}$, $\text{RN}(a) = a(1 + \delta)$, $|\delta| \leq \frac{u}{1 + u}$.

Fig. 2. $\text{ufp}$ and $\text{ulp}$ for a floating-point number.
where $\delta$ for Theorem 2.7

From the assumptions, Theorem 2.8

For

bers.

Floating-point arithmetic produces a floating-point number as a result of the arithmetic of floating-point numbers. The notation $f_1(\cdot)$ indicates that all operations in the parenthesis are evaluated using floating-point arithmetic. For simplicity, $f_1((a + b) + c)$ means $f_1(f_1(a + b) + c)$ for $a, b, c \in \mathbb{F}$. Because $a, b \in \mathbb{F} \Rightarrow a + b \in \mathbb{F}$ is not generally satisfied, we consider the difference between $a + b$ and $f_1(a + b)$. IEEE 754 requires that a result of the four basic operations and the square root is the nearest floating-point number to the exact result.

Next, we introduce the property of the addition and subtraction of subnormal floating-point numbers.

Theorem 2.8 For $x, y \in \mathbb{F}$, if $|f_1(x \pm y)| < 2\text{F}_{\text{min}}$, $f_1(x \pm y) = x \pm y$.

Proof From the assumptions, $x \pm y \in S_{\text{min}} \mathbb{Z}$ and $x \pm y < 2\text{F}_{\text{min}}$: therefore, Theorem 2.2 proves $f_1(x \pm y) = x \pm y$. \hfill $\Box$

Theorem 2.8 says that, if both $x$ and $y$ are subnormal numbers, $f_1(x \pm y) = x \pm y$ holds true. Therefore, we need not care about a rounding error for the underflow case for addition and subtraction, and we can straightforwardly exploit Theorems 2.3–2.5 and obtain the following theorem.
Theorem 2.9 For \(a, b \in \mathbb{F}\),

\[
\text{fl}(a \pm b) = (a \pm b) (1 + \delta_1), \quad |\delta_1| \leq \frac{u}{1 + u} \\
(a \pm b) = \text{fl}(a \pm b) (1 + \delta_2), \quad |\delta_2| \leq u \\
\text{fl}(a \pm b) = a \pm b + \delta_3, \quad |\delta_3| \leq u \cdot \text{ufp}(a \pm b)
\]

are satisfied barring overflow.

Next, we introduce a rounding error analysis for the product of floating-point numbers. This is straightforwardly obtained from Theorems 2.3 and 2.6.

Theorem 2.10 For \(a, b \in \mathbb{F}\), there exist \(\delta\) and \(\eta\) such that

\[
\text{fl}(a \cdot b) = a \cdot b + \delta + \eta, \quad \delta \cdot \eta = 0, \quad |\delta| \leq u \cdot \text{ufp}(a \cdot b), \quad |\eta| \leq \frac{S_{\text{min}}}{2}
\]
barring overflow.

If underflow occurs in \(\text{fl}(a \cdot b)\), we can set \(\delta = 0\) in Theorem 2.10. Otherwise, \(\eta = 0\).

Next, we introduce Sterbenz’s theorem for the subtraction of floating-point numbers.

Theorem 2.11 (Sterbenz [8]) For \(x, y \in \mathbb{F}\) if \(y/2 \leq x \leq 2y\), \(x - y = \text{fl}(x - y)\).

Proof The proof is trivial for \(x = y = 0\). Therefore, we assume \(x \geq y > 0\) without loss of generality. Because \(\text{ufp}(x) \geq \text{ufp}(y), x, y \in 2u \cdot \text{ufp}(y)\mathbb{Z}\). From \(x \leq 2y\), we obtain \(2\text{ufp}(y) > y \geq x - y \in 2u \cdot \text{ufp}(y)\mathbb{Z}\). Therefore, from Theorem 2.2, \(x - y \in \mathbb{F}\) holds true. \(\square\)

Next, we introduce a theorem used later for the rounding error analyses. This is an extension of Theorem 2.9.

Theorem 2.12 (Jeannerod and Rump [7]) For \(a, b \in \mathbb{F}\),

\[
|\text{fl}(a + b) - (a + b)| \leq \min(|a|, |b|, u \cdot \text{ufp}(a + b))
\]
is satisfied barring overflow.

Proof We can use Theorem 2.9, so that we only need to prove \(|\text{fl}(a + b) - (a + b)| \leq \min(|a|, |b|)\). Note that \(\text{fl}(\cdot)\) produces the nearest floating-point number to an exact result. Therefore, for any \(f \in \mathbb{F}\), \(|\text{fl}(a + b) - (a + b)| \leq |f - (a + b)|\) holds true. Setting \(f = a\) or \(f = b\) completes the proof. \(\square\)

Theorem 2.12 produces a better error bound when there is a large difference in the magnitudes of \(|a|\) and \(|b|\).

The following theorem is used in Section 5.

Theorem 2.13 For \(a, b \in \mathbb{F}\), and \(a \neq 0\), \(\text{fl}(a + b) \in u \cdot \text{ufp}(a)\mathbb{Z}\) barring overflow.

Proof The case for \(b = 0\) is trivial. Therefore, we consider \(b \neq 0\). From Theorem 2.2, we obtain \(a \in 2u \cdot \text{ufp}(a)\mathbb{Z} \subset u \cdot \text{ufp}(a)\mathbb{Z}\). If \(ab > 0\), from \(|\text{fl}(a + b)| \geq |a|, \text{fl}(a + b) \in u \cdot \text{ufp}(a)\mathbb{Z}\). Therefore, we assume \(ab < 0\). In addition, \(|a - b| = |b - a|\) and we can exchange \(a\) and \(b\), accordingly, we focus on \(\text{fl}(a - b)\) for \(a > b > 0\) during the rest of the proof.

If \(2\text{ufp}(b) \geq \text{ufp}(a)\), we have \(b \in 2u \cdot \text{ufp}(b)\mathbb{Z} \subset u \cdot \text{ufp}(a)\mathbb{Z}\). Next, we consider the case \(2\text{ufp}(b) < \text{ufp}(a)\). From the assumption, we have

\[
4\text{ufp}(b) \leq \text{ufp}(a), \quad \frac{1}{2} b < \text{ufp}(b)
\]
and these stipulate \(2b < \text{ufp}(a)\). From \(-b > -\frac{\text{ufp}(a)}{2}\), we have

\[
a - b > a - \frac{\text{ufp}(a)}{2} \geq \frac{\text{ufp}(a)}{2}
\]
Therefore, \(\text{fl}(a - b) \geq \frac{\text{ufp}(a)}{2}\), which yields \(\text{fl}(a - b) \in u \cdot \text{ufp}(a)\mathbb{Z}\). \(\square\)
3. Interval arithmetic

Interval arithmetic plays an important role in verified numerical computations. An overview of interval arithmetic can be found in, for example, [9–13].

3.1 Interval

Let $\mathbb{IR}$ be a set of bounded closed intervals on $\mathbb{R}$. Hereafter, an interval refers to an element of $\mathbb{IR}$. We use a bold letter to indicate an interval: e.g., $\mathbf{a}$. Using $\mathbf{a}, \mathbf{\bar{a}} \in \mathbb{R}$, $\mathbf{a} \leq \mathbf{\bar{a}}$, we represent an interval as

$$\mathbf{a} = [\mathbf{a}, \mathbf{\bar{a}}] := \{ x \in \mathbb{R} \mid \mathbf{a} \leq x \leq \mathbf{\bar{a}} \}.$$  

We call this form the infimum-supremum form. For an interval $\mathbf{a}$, mid($\mathbf{a}$) and rad($\mathbf{a}$) indicate the midpoint and the radius of the interval, respectively, and

$$\text{mid}(\mathbf{a}) = \frac{1}{2}(\mathbf{a} + \mathbf{\bar{a}}), \quad \text{rad}(\mathbf{a}) = \frac{1}{2}(\mathbf{\bar{a}} - \mathbf{a}).$$

An interval $\mathbf{a}$ can be represented using a midpoint and a radius. In particular, using $c, r \in \mathbb{R}$, $r \geq 0$, we represent $\mathbf{a}$ as

$$\mathbf{a} = \langle c, r \rangle := \{ x \in \mathbb{R} \mid ||x - c|| \leq r \} = \{ x \in \mathbb{R} \mid c - r \leq x \leq c + r \}.$$  

We call this form the midpoint-radius form. A midpoint-radius form can easily be transformed into an infimum-supremum interval.

To handle an infinite interval, e.g., when the endpoint is $\infty$ or $-\infty$, we define $\mathbb{IR}_\infty := \{ [x, \infty) \mid x \in \mathbb{R} \} \cup \{(\infty, x] \mid x \in \mathbb{R} \} \cup \{(-\infty, \infty)\}$ and let $\mathbb{IR}_* := \mathbb{IR} \cup \mathbb{IR}_\infty$.

If an endpoint of the interval can be $\infty$ or $-\infty$, we can define interval arithmetic for infinite intervals. However, for $\mathbf{a} \in \mathbb{IR}_\infty$, there is no midpoint-radius form except $\mathbf{a} = (-\infty, \infty)$. Exceptionally, if we set the radius of the interval to $\infty$, $\mathbf{a} \subset \langle c, \infty \rangle = (-\infty, \infty)$ for any $c \in \mathbb{R}$.

We define $|\mathbf{a}|$ for $\mathbf{a} \in \mathbb{IR}_*$ as

$$|\mathbf{a}| = \begin{cases} |\min_{a \in \mathbf{a}} \mathbf{a}|, & (\mathbf{a} \in \mathbb{IR}) \\ |\min_{a \in \mathbf{a}} \mathbf{a}|, & (\mathbf{a} \in \mathbb{IR}_\infty) \end{cases}$$  

and define $\text{mag}(\mathbf{a})$ such that

$$\text{mag}(\mathbf{a}) := \sup_{a \in \mathbf{a}} |a| \geq 0.$$  

In particular, for $\mathbf{a} = \langle a_c, a_r \rangle$

$$\text{mag}(\mathbf{a}) = |a_c| + a_r \geq 0$$  

holds true.

For a finite closed set $X$ on $\mathbb{R}$, hull($X$) indicates the smallest enclosure of $X$ such that

$$\text{hull}(X) := \left[ \min_{x \in X} x, \max_{x \in X} x \right] \in \mathbb{IR}.$$  

For an unbounded $X$, hull($X$) $\in \mathbb{IR}_*$.

Up until now, we have discussed real intervals. We have two methods to extend the discussion of real intervals to complex intervals. One method is to use real intervals for the real part and the imaginary part. We call such an interval a complex rectangular interval. The other method is to use a complex disc region. There are merits and demerits to both methods. In this paper, we introduce the latter method.

A complex disc can be represented in a similar way to the midpoint-radius form in the real case. Namely, for $c \in \mathbb{C}$, $r \in \mathbb{R}$, and $r \geq 0$,
For \( (c, r) := \{ z \in \mathbb{C} \mid |z - c| \leq r \} \) represents a disc with a midpoint \( c \) and a radius \( r \) on the complex plane.

Let \( \mathbb{C} \) denote the set of discs on the complex plane. The notation \( \text{mag}(a) \) for \( a \in \mathbb{C} \) is defined in a similar way as \( \mathbb{R} \) in (12). In addition, let \( \mathbb{C}^n \) and \( \mathbb{C}^{m \times n} \) be the set of \( n \)-vectors and \( m \times n \) matrices whose elements are all complex discs. This is a straightforward extension of the real case (13).

### 3.2 Interval arithmetic

#### 3.2.1 Real case

We define four arithmetic operations for \( a, b \in \mathbb{R} \) such that

\[
\begin{align*}
    a \circ b & := \{ a \circ b \in \mathbb{R} \mid a \in a, \ b \in b \}, \quad \circ \in \{ +, -, \times, / \}.
\end{align*}
\]

Except for special cases, \( a \circ b \in \mathbb{R} \) holds true. In practice, for the infimum-supremum intervals \( a = [\alpha, \beta] \) and \( b = [\gamma, \delta] \), we obtain

\[
\begin{align*}
    a + b & = [\alpha + \gamma, \beta + \delta], \\
    a - b & = [\alpha - \delta, \beta - \gamma], \\
    a \times b & = [\min(ab, \alpha \beta, \alpha \delta, \beta \gamma), \max(ab, \alpha \beta, \alpha \delta, \beta \gamma)], \\
    a/b & = a \times [1/\delta, 1/\gamma], (0 \notin b).
\end{align*}
\]

For the midpoint-radius intervals \( a = \langle a_c, a_r \rangle \) and \( b = \langle b_c, b_r \rangle \), we have

\[
\begin{align*}
    a + b & = \langle a_c + b_c, a_r + b_r \rangle, \\
    a - b & = \langle a_c - b_c, a_r + b_r \rangle, \\
    a \times b & = \langle a_c b_c + \delta_1, a_r + \delta_2 \rangle, \\
    \delta_1 & := \text{sgn}(a_c b_c) \min(a_c |b_c|, |a_c| b_r, a_r b_r), \\
    \delta_2 & := \max(a_c (|b_c| + b_r), (a_r + |b_c|) b_r, a_r |b_c| + |a_c| b_r), \\
    a/b & = a \times \langle b_c/d, b_r/d \rangle, \quad d := b_c^2 - b_r^2 \quad (0 \notin b).
\end{align*}
\]

Intervals and interval arithmetic can be extended to matrices and vectors [1, 12].

The function \( \text{sgn}(x) \) for \( x \in \mathbb{R} \) produces the sign of \( x \), such that

\[
\text{sgn}(x) := \begin{cases} 
1, & x > 0 \\
0, & x = 0 \\
-1, & x < 0
\end{cases}
\]

Because the proofs of (18), (19), and (21) are trivial, we introduce part of the proof of (20) in detail. For \( \langle a_c, a_r \rangle \cdot \langle b_c, b_r \rangle = [c, \overline{c}] \) and \( a_c, b_c \geq 0 \), we have \( \overline{c} = (a_c + a_r)(b_c + b_r) = a_c b_c + a_r b_r + a_c b_r + a_r b_c \). Let

\[
\begin{align*}
    \alpha_1 & = (a_c + a_r)(b_c - b_r) = a_c b_c + a_r b_r - a_r b_c - a_r b_r, \\
    \alpha_2 & = (a_c - a_r)(b_c + b_r) = a_c b_c - a_r b_c + a_r b_r - a_r b_r, \\
    \alpha_3 & = (a_c - a_r)(b_c - b_r) = a_c b_c - a_r b_r - a_r b_c + a_r b_r.
\end{align*}
\]

Then, we obtain \( \zeta = \min(\alpha_1, \alpha_2, \alpha_3) \). Letting

\[
\begin{align*}
    \alpha_4 & = a_c b_r - a_r b_c - a_r b_r - a_r b_r
\end{align*}
\]

we have

\[
\begin{align*}
    \alpha_1 & = \alpha_4 + 2a_r b_c, \quad \alpha_2 = \alpha_4 + 2a_r b_r, \quad \alpha_3 = \alpha_4 + 2a_r b_c, \\
    \alpha_4 & = \alpha_4 + 2 \min(a_r b_c, a_c b_r, a_r b_r).
\end{align*}
\]

so that \( \zeta = \alpha_4 + 2 \min(a_r b_c, a_c b_r, a_r b_r) \). Therefore, the center is \( \frac{c + \overline{c}}{2} \), and the radius is \( \frac{\overline{c} - c}{2} \). After several case distinctions such as \( a_c, b_c < 0 \) and \( a_r b_c < 0 \), we can obtain (20).
When we compute a product of intervals using (16) and (20), it is necessary to obtain the maximum or minimum value. To avoid these values, enclosure by a midpoint-radius interval
\[ a \times b \subseteq \langle a_r, b_r \rangle = [a_r, a_r + b_r] \]
is often used. Even though using a midpoint-radius interval causes an overestimation of the interval length compared to (16) and (20), the overestimation of the length is at most 1.5 times [15].

For intervals \( a, b, c \in \mathbb{R} \), the distributive property \( a(b + c) = ab + ac \) is generally not valid. For example, letting \( a = [-1, 1], b = [1, 2], \) and \( c = [-2, 1] \), we have
\[ a(b + c) = a \times [-1, 3] = [-3, 3] \]
and
\[ ab + ac = [-2, 2] + [-2, 2] = [-4, 4]; \]
therefore this example shows that the distributive law is not satisfied. Conversely, subdistributive law
\[ a(b + c) \subseteq ab + ac \] (23)
is satisfied.

For a real function \( f(x) \), we represent the range for \( x \in \mathbb{R} \) as
\[ f(x) := \{ f(x) \in \mathbb{R} \mid x \in x \} \].
Here, if \( f(x) \) is not continuous on \( x \), \( f(x) \in \mathbb{R} \) may not be satisfied and \( f(x) \) is a subset of \( \mathbb{R} \). Such an \( f(x) \) is difficult to obtain in many cases; however, using interval arithmetic, it is often possible to obtain an interval \( y \) such that \( f(x) \subseteq y \in \mathbb{R} \).

We can substitute \( x \) into a given \( f(x) \), and evaluate it straightforwardly using interval arithmetic. Such an evaluation is denoted by \( f_{\mid x} \). If \( f(x) \) is continuous on an interval \( x \),
\[ f(x) \subseteq f_{\mid x}. \]
For example, let
\[ f(x) = x^2 + 3x + 2 \]
and \( f_{\mid x} = x \cdot x + 3x + 2 \). Substituting \( x = [-1, 1] \) into the function and computing the interval arithmetic gives
\[ f(x) \subseteq f_{\mid x} = [-1, 1] \cdot [-1, 1] + 3 \cdot [-1, 1] + 2 = [-2, 6]. \] (24)
If two functions with different forms are mathematically equivalent, we can obtain different results. For example, this is understood with \( f_{\mid x} = (x + 1)(x + 2) \):
\[ f(x) \subseteq f_{\mid x} = ([x + 1 + 1] + [-1, 1] + 2) = [0, 6]. \] (25)

From (24) and (25), and (23), one may misunderstand that we can obtain a narrow interval using a factored form. This is incorrect and can be understood with
\[ x(x - 1)(x + 1), \quad x^3 - x. \]
If we substitute \( x = [-1, 1] \) to the forms, we have
\[ f_{\mid x} = x(x - 1)(x + 1) = [-4, 4], \]
\[ f_{\mid x} = x^3 - x = [-2, 2]; \]
therefore, it is not better to use the factored form. In addition, as an extreme example, we can set \( g(x) = 0 \) and \( g_{\mid x} = x - x \). Substituting \( x = [1, 2] \) into the function, we have
\[ g(x) \subseteq g_{\mid x} = [1, 2] - [1, 2] = [-1, 1] \]
and we cannot obtain \( g_{\mid x} = 0 \). The problem is the dependency on the variable. The radius of the sum (or the subtraction) of the intervals generally increases if we do not care about the dependency of the intervals. This is understood from (18) and (19). To avoid the expansion of an interval, we can apply a mean-value form or affine arithmetic [14].
3.2.2 Case of a disk on a complex plane

We can extend the interval arithmetic to complex numbers. We consider the four arithmetic operations for \( a = \langle a_c, a_r \rangle, b = \langle b_c, b_r \rangle \in \mathbb{C} \) in a similar way as to in the real case. This is done using circular arithmetic. Used for addition and subtraction, interval arithmetic with the midpoint-radius form in the real case, (18) and (19) can be extended to the complex case. For multiplication and division, we consider

\[
a \times b = \langle a_c, a_r \rangle \times \langle b_c, b_r \rangle = \langle a_c b_c (1 + \delta), (|a_c| b_r + a_r |b_c|)(1 + \delta) \rangle,
\]
\[
\delta = a_r b_r / (|a_c| b_c + |a_c| b_r + a_r |b_c|)
\]
\[
a / b = a \times \langle b_c / d, b_r / d \rangle, \quad d := |b_c|^2 - b_r^2 \quad (0 \notin b)
\]

where \( \overline{b_c} \) denotes the complex conjugate of \( b_c \).

It is possible to define \( f_{(\cdot)}(z) \) using circular arithmetic, as in the real case, such that

\[
f(z) \subseteq f_{(\cdot)}(z) \in \mathbb{IC}, \quad z \in \mathbb{IC}.
\]

Here, assuming \( w = f_{(\cdot)}(z) \), we obtain

\[
\max_{z \in z} |f(z)| \leq \text{mag}(w).
\] (26)

3.3 Machine interval arithmetic

We first introduce directed rounding on floating-point arithmetic. \( \text{fl}_\Sigma(\cdot) \) means that the operation in the parentheses is performed with the rounding downward mode (it produces the maximum floating-point number that is less than the exact result). \( \text{fl}_\Delta(\cdot) \) means that the operation in the parentheses is performed with the rounding upward mode (it produces the minimum floating-point number that is less than the exact result). Namely, for \( a, b \in \mathbb{F} \) and \( \circ \in \{ +, -, \times, / \} \), we obtain

\[
\text{fl}_\Sigma(a \circ b) = \begin{cases} 
\max\{x \in \mathbb{F} \mid x \leq a \circ b\} & (a \circ b \geq -\text{F}_{\text{max}}) \\
-\text{Inf} & (a \circ b < -\text{F}_{\text{max}})
\end{cases}
\]
\[
\text{fl}_\Delta(a \circ b) = \begin{cases} 
\min\{x \in \mathbb{F} \mid x \geq a \circ b\} & (a \circ b \leq \text{F}_{\text{max}}) \\
\text{Inf} & (a \circ b > \text{F}_{\text{max}})
\end{cases}
\]

and

\[
\text{fl}_\Sigma(a \circ b) \leq a \circ b \leq \text{fl}_\Delta(a \circ b) \quad (27)
\]

is satisfied. This is specified in the IEEE 754 standard. For example, a function for switching the rounding modes is standardized in C99 (ISO/IEC 9899:1999). Therefore, if a compiler supports a function, we can use it by including \texttt{fenv.h} and using the function

\[
\text{int fesetround(int mode)}.
\]

For the argument \texttt{mode} of the function, we can use the following macro:

| Mode         | Direction of rounding              |
|--------------|-----------------------------------|
| FE_TONEAREST | rounding to the nearest           |
| FE_UPWARD    | rounding upward                    |
| FE_DOWNWARD  | rounding downward                  |
| FE_TOWARDZERO| rounding in the zero direction     |

If real arithmetic is supported, we can rigorously implement interval arithmetic. However, due to the problem of the rounding error, e.g. \( \text{fl}(a + b) = a + b \) is not always satisfied for \( a, b \in \mathbb{F} \), straightforward implementation using floating-point arithmetic is problematic. For example,

\[
[a, \overline{a}] + [b, \overline{b}] := [\text{fl}(a + b), \text{fl}(\overline{a} + \overline{b})]
\]

288
is an incorrect implementation.

Now, we introduce machine intervals and their arithmetic. A machine interval represents an interval using floating-point numbers. In practice, for \(a, \bar{a} \in F\), \(a \leq \bar{a}\), the infimum-supremum machine interval is represented as

\[
[a, \bar{a}] := \{ x \in \mathbb{R} \mid a \leq x \leq \bar{a} \}.
\]

Let \(\mathbb{IF}_{\text{infsup}}\) denote a set of machine intervals with the infimum-supremum form. For \(a_e, a_r \in F\), \(a_r \geq 0\), we represent a machine interval with the midpoint-radius form as

\[
\langle a_e, a_r \rangle := \{ x \in \mathbb{R} \mid a_e - a_r \leq x \leq a_e + a_r \}.
\]

Let \(\mathbb{IF}_{\text{midrad}}\) be a set of machine intervals with the midpoint-radius form, and let

\[
\mathbb{IF} = \mathbb{IF}_{\text{infsup}} \cup \mathbb{IF}_{\text{midrad}}
\]

be the set of all machine intervals.

Note that \(\mathbb{IF}_{\text{infsup}}\) and \(\mathbb{IF}_{\text{midrad}}\) are different. For example, for \(a = [1 - u^2, 1 + u^2] = (1, u^2)\), \(a \notin \mathbb{IF}_{\text{infsup}}\) and \(\langle 1, u^2 \rangle \in \mathbb{IF}_{\text{midrad}}\). In addition, for \(a = [1, 1 + 2u] = (1 + u, u)\), \(a \in \mathbb{IF}_{\text{infsup}}\) and \(a \notin \mathbb{IF}_{\text{midrad}}\).

For infinite intervals, we use the following notation for convenience:

\[
[x, \text{Inf}], x \in F \iff [x, \infty), x \in F,
\]

\[
[-\text{Inf}, x], x \in F \iff (-\infty, x], x \in F,
\]

\[
[-\text{Inf}, \text{Inf}] \iff (-\infty, \infty).
\]

To use infinite intervals, we define the set \(\mathbb{IF}_{\infty}\) such that

\[
\mathbb{IF}_{\infty} := \{ [x, \text{Inf}] \mid x \in F \} \cup \{ [-\text{Inf}, x] \mid x \in F \} \cup \{ [-\text{Inf}, \text{Inf}] \}
\]

and let \(\mathbb{IF}_*\) be

\[
\mathbb{IF}_* := \mathbb{IF} \cup \mathbb{IF}_{\infty}.
\]

This is an extension of \(\mathbb{IF}\).

**Remark 1**

As in the case of a real interval, \(a \in \mathbb{IF}_{\infty}\) cannot be represented in the midpoint-radius form except in the case \(a = (-\infty, \infty)\). Therefore, for \(c \in \mathbb{IF}_*\), we use an enclosure such that \(a \subseteq \langle c, \text{Inf} \rangle = (-\infty, \infty)\). Here, note that \(\langle c, \text{Inf} \rangle = (-\infty, \infty)\) for \(c = \text{Inf}, -\text{Inf}\). If \(c = \text{NaN}\), \(\langle c, \text{Inf} \rangle\) is an invalid interval.

Now, we introduce machine interval arithmetic. Machine interval arithmetic for the intervals \(a, b \in \mathbb{IF}_{\text{infsup}}\) with the infimum-supremum form produces the following enclosures.

\[
a + b \subseteq [f1\sqrt{\langle a + b \rangle}, f1\Delta (\bar{a} + \bar{b})]
\]

(28)

\[
a - b \subseteq [f1\sqrt{\langle a - b \rangle}, f1\Delta (\bar{a} - \bar{b})]
\]

(29)

\[
a \times b \subseteq [f1\sqrt{\langle \min(ab, \bar{a}b, \bar{a}\bar{b}, \bar{a}\bar{b}) \rangle}, f1\Delta (\max(ab, \bar{a}b, \bar{a}\bar{b}, \bar{a}\bar{b}))]
\]

(30)

\[
a/b \subseteq [f1\sqrt{\langle \min(ab, a/b, \bar{a}/b, \bar{a}/b) \rangle}, f1\Delta (\max(ab, a/b, \bar{a}/b, \bar{a}/b))] \quad (0 \notin b)
\]

(31)

For division, it is possible to use (17). However, to reduce the number of rounding errors, we use (31). Therefore, we can obtain the enclosure of the interval arithmetic using only floating-point arithmetic. Note that if \(a\) or \(b\) is an infinite interval, we require exception handling (see Subsection 3.4). Exception handling is only necessary for \(a \times b\). If \(0 \in b\), machine interval arithmetic produces an invalid interval \(a/b = [\text{NaN}, \text{NaN}]\).

We transform the infimum-supremum form \([a, \bar{a}] \in \mathbb{IF}_{\text{infsup}}\) into the midpoint-radius form \(\langle c, r \rangle \in \mathbb{IF}_{\text{midrad}}\). Theoretically, for \([a, \bar{a}]\), the midpoint is \(\frac{a + \bar{a}}{2}\), and the radius is \(\frac{\bar{a} - a}{2}\). However, \(\frac{a + \bar{a}}{2} \notin F\) or \(\frac{\bar{a} - a}{2} \notin F\) is possible. Therefore, producing the enclosure of \([a, \bar{a}]\) using a midpoint-radius interval is possible.
Theorem 3.1 For $a = [a, \overline{a}] \in \mathbb{I}_{\text{infsup}} \cup \mathbb{I}_\infty \setminus \{-\infty, \infty\}$, let

$$c = \text{fl}_\Delta \left( \frac{a + \overline{a}}{2} \right), \quad r = \text{fl}_\Delta (c - \overline{a}).$$

Then, $a \subseteq \langle c, r \rangle \in \mathbb{I}_{\text{midrad}} \cup \{[-\infty, \infty]\}$ holds true.

Proof For the case of $a \in \mathbb{I}_{\text{infsup}}$, we prove $c + r \geq a$ and $c - r \leq a$. Using (27), we have

$$c + r = \text{fl}_\Delta \left( \frac{a + \overline{a}}{2} \right) + \text{fl}_\Delta (c - \overline{a}) \geq \frac{a + \overline{a}}{2} + c - \overline{a} \geq a.$$

The case of $c - r \leq a$ can be similarly proved. If $a \in \{[x, \infty] \mid x \in \mathbb{F}\}$, $c = \infty$ and $r = \infty$. From (1), $a \subseteq \langle \infty, \infty \rangle = (-\infty, \infty)$. The case of $a \in \{[-\infty, x] \mid x \in \mathbb{F}\}$ can be similarly proved. □

For $a = [-\infty, \infty], \ a = \langle c, \infty \rangle = [-\infty, \infty]$ for any $c \in \mathbb{F}_*$. If we use floating-point numbers whose base is not two in Theorem 3.1, $c/2 \in [a, \overline{a}]$ is possible ($[a, \overline{a}] \subseteq \langle c, r \rangle$ is satisfied). Therefore, the midpoint is obtained by

$$c = \text{fl}_\Delta \left( \frac{\overline{a} - a}{2} \right).$$

Then, $c \in [a, \overline{a}]$ is satisfied for floating-point numbers with all base. Oishi proposed the extension of (32) to interval matrices [3]. This extension has contributed to the development of fast algorithms for interval arithmetic [3, 15].

3.4 Exception handling for infinite intervals
Special care is necessary for interval arithmetic for infinite intervals. For addition, subtraction, and division, we can directly use (28), (29), and (31), respectively. However, for multiplication, NaN appears for $0 \times \infty$. This should be avoided.

For $a = [a, \overline{a}], \ b = [\underline{b}, \overline{b}] \in \mathbb{I}_*$, we introduce rules for the machine interval arithmetic of $a \times b$.

- For $\underline{a} = \overline{a} = 0$, if $\underline{b} = -\infty$ or $\overline{b} = \infty$, then $[-\infty, \infty]$. Otherwise, $[0, 0]$.
- For $\underline{b} = \overline{b} = 0$, if $\underline{a} = -\infty$ or $\overline{a} = \infty$, then $[-\infty, \infty]$. Otherwise, $[0, 0]$.
- For other cases, we use (30).

For further topics concerning intervals, see [12, 13].

4. Rounding error analyses
In this section, we introduce rounding error analyses of the sum of floating-point numbers and the dot product of floating-point vectors. The results of the rounding error analyses are very useful for verified numerical computations.

4.1 Rounding error analysis for floating-point summation
For $p \in \mathbb{F}^n$, we compute $\sum_{i=1}^n p_i$ using floating-point arithmetic. In this computation, $n - 1$ rounding errors maximally occur, and we evaluate an absolute error bound. For this purpose, we consider the order of the computations. For example, if we compute the sum of the elements given by an array $a$ in the C language, and we write

```c
for (r = 0.0, i = 0; i < n; i++) r += a[i];
```
and compile it, the computation may not be performed as follows2:

2If we select a proper compile option with no optimization for the accuracy of the floating-point result, then the order of the computations is recursive.
Therefore, it is important to make a rounding error analysis with any computational order. Let $\text{float}(\cdot)$ be a result of floating-point computations, where the computational order inside the parenthesis is arbitrary. For example, $\text{float}(a + b + c)$ is one of the following:

$$\text{fl}((a + b) + c), \quad \text{fl}((a + c) + b), \quad \text{fl}((b + c) + a).$$

If we write $\text{float}(a + b + c) < \alpha$, $\text{fl}(((a + b) + c) < \alpha$, $\text{fl}(((a + c) + b) < \alpha$, $\text{fl}((b + c) + a) < \alpha$ are all satisfied.

To evaluate an expression of a formula, we use a binary tree for the computational order. Figure 3 shows the binary tree of the following evaluation:

$$((p_1 + p_7) + (p_6 + p_2)) + ((p_3 + p_5) + (p_4 + p_8)).$$

First, we give a proof for all the leaves of the binary tree. Next, we give assumptions for the left child and right child of a node, and then we try to give a proof for the node. Then, the proof is completed. $I_1$ contains the set of indices for the computed $p_i$ for the left child. $I_2$ contains the set of indices for the computed $p_i$ for the right child. For example, for the node in Fig. 3, $I_1 = \{1, 7\}$ and $I_2 = \{6, 2\}$.

Let $s = s_1 + s_2$, $\hat{s} = \text{fl}(\hat{s}_1 + \hat{s}_2)$, $S_1 = \sum_{i \in I_1} |p_i|$, $S_2 = \sum_{i \in I_2} |p_i|$, and $S = S_1 + S_2$. $\hat{S}_1$ and $\hat{S}_2$ indicate the computed results of $\sum_{i \in I_1} |p_i|$ and $\sum_{i \in I_2} |p_i|$, respectively. We set $\hat{S} = \text{fl}(\hat{S}_1 + \hat{S}_2)$.

**Theorem 4.1 (Jeannerod and Rump [7, 16])** For $\text{float} \left( \sum_{i=1}^{n} p_i \right)$ for $p \in \mathbb{F}^n$,

$$\left| \text{float} \left( \sum_{i=1}^{n} p_i \right) - \sum_{i=1}^{n} p_i \right| \leq (n - 1)u' \sum_{i=1}^{n} |p_i| \quad (34)$$

$$\leq (n - 1)u \sum_{i=1}^{n} |p_i| \quad (35)$$

is satisfied where $u' := \frac{u}{1 + u}$.

**Proof** First, we prove (34). Because there is no floating-point arithmetic for the leaves of the binary tree, the proof for the leaves is trivial. For the left child ($j = 1$) and right child ($j = 2$), assume that
Theorem 4.2 (Rump [18]) is satisfied. Let $\delta := f_1(\hat{s}_1 + \hat{s}_2) - (\hat{s}_1 + \hat{s}_2)$. We have
\[
|\hat{s} - s| = |\hat{s} - (\hat{s}_1 + \hat{s}_2) + \hat{s}_1 - s_1 + \hat{s}_2 - s_2| \\
\leq |\hat{s} - (\hat{s}_1 + \hat{s}_2)| + |\hat{s}_1 - s_1| + |\hat{s}_2 - s_2| \\
\leq |\delta| + u'(n_1 - 1)S_1 + (n_2 - 1)S_2).
\]

The goal of the proof is to show that
\[
|\hat{s} - s| \leq (n_1 + n_2 - 1)u'(S_1 + S_2) = (n_1 + n_2 - 1)u'S.
\]

Therefore, if $|\delta| \leq u'\pi$ and $\pi = n_2S_1 + n_1S_2$, the proof is complete. Let $e_1 := |\hat{s}_1 - s_1|$ and $e_2 := |\hat{s}_2 - s_2|$.

We consider case distinction. If $S_1 \leq u'S_1$, then $S_2 \leq S_1$. From Theorem 2.12,
\[
|\delta| \leq |\hat{s}_2| \leq |e_2| + |s_2| \leq |e_2| + S_2 \\
\leq u'((n_2 - 1)S_1 + S_2) \leq u'n_2S_1 \leq u'\pi.
\]

We can prove the case of $S_1 \leq u'S_1$ by exchanging the roles of $S_1$ and $S_2$.

Finally, we consider the cases of $u'S_1 < S_2$ and $u'S_2 < S_1$. From Theorem 2.9, we obtain $|\delta| \leq u'|\hat{s}_1 + \hat{s}_2|$. From $|\hat{s}_1 + \hat{s}_2| \leq |e_1| + |s_1| + |e_2| + |s_2| \leq |e_1| + S_1 + |e_2| + S_2$ and
\[
|e_1| \leq (n_1 - 1)u'S_1 \leq (n_1 - 1)S_2, \quad |e_2| \leq (n_2 - 1)u'S_2 \leq (n_2 - 1)S_1,
\]
we have $|\hat{s}_1 + \hat{s}_2| \leq \pi$, and (34) is proved. $u' < u$ immediately leads to (35).

Historically,
\[
\gamma_k := \frac{ku}{1 - ku} \quad (ku < 1)
\]
has been used [17] instead of $(n - 1)u$ in (35). Compared to $\gamma_k$ in (36), $ku$ ($k \in \mathbb{N}$) in Theorem 4.1 is a very beautiful constant, and has no $\mathcal{O}(u^2)$ term. In addition, there is no limit on $n$, compared to $\gamma_k$.

Next, we introduce another variant of the rounding error analysis for the sum [18].

Theorem 4.2 (Rump [18]) For $p \in \mathbb{F}^n$,
\[
\left| \text{float} \left( \sum_{i=1}^{n} p_i \right) - \sum_{i=1}^{n} p_i \right| \leq (n - 1)u \cdot \text{ufp} \left( \text{float} \left( \sum_{i=1}^{n} |p_i| \right) \right)
\]
is satisfied. Note that the computational order of the left and right float(·) in (37) must be the same.

Proof The proof for the leaves is trivial. Assume that (37) is satisfied for the left and right child, such that
\[
\left| \text{float} \left( \sum_{i \in I_j} p_i \right) - \sum_{i \in I_j} p_i \right| \leq (n_j - 1)u \cdot \text{ufp} \left( \text{float} \left( \sum_{i \in I_j} |p_i| \right) \right), \quad j \in \{1, 2\}.
\]

This assumption and Theorem 2.9 yield
\[
|\hat{s} - s| = |\hat{s} - (\hat{s}_1 + \hat{s}_2) + \hat{s}_1 - s_1 + \hat{s}_2 - s_2| \\
\leq |\hat{s} - (\hat{s}_1 + \hat{s}_2)| + |\hat{s}_1 - s_1| + |\hat{s}_2 - s_2| \\
\leq u \cdot \text{ufp}(\hat{s}) + (n_1 - 1)u \cdot \text{ufp}(\hat{S}_1) + (n_2 - 1)u \cdot \text{ufp}(\hat{S}_2) \\
\leq u \cdot \text{ufp}(\hat{S}) + (n_1 + n_2 - 2)u \cdot \text{ufp}(\hat{S}) \\
= (n_1 + n_2 - 1)u \cdot \text{ufp}(\hat{S}).
\]

This completes the proof. \qed
Theorem 4.3 (Jeannerod and Rump [16])
We introduce a rounding error analysis of the dot product of floating-point vectors. The following

4.2 Rounding error analysis for floating-point dot product

$u \cdot y$ is satisfied, because both $u$ and $ufp(\cdot)$ are powers of two. This means that the upper bound of the
error can be obtained using only floating-point arithmetic.

Theorem 4.4 (Rump [18])
We introduce a rounding error analysis of the dot product of floating-point vectors. The following

Proof Let $p_i = x_i y_i$, $\hat{p}_i = fl(x_i y_i)$, and $\hat{s}_n = float\left(\sum_{i=1}^{n} \hat{p}_i\right)$. We consider the rounding errors of $n$
products and $n-1$ sums. We use Theorem 4.1 for the sum and Theorem 2.9 for the products. Then,
we have

\[
|\text{float}(x^T y) - x^T y| = |\hat{s}_n - \sum_{i=1}^{n} p_i| = |\hat{s}_n - \sum_{i=1}^{n} \hat{p}_i + \sum_{i=1}^{n} \hat{p}_i - \sum_{i=1}^{n} p_i|
\leq |\hat{s}_n - \sum_{i=1}^{n} \hat{p}_i| + |\sum_{i=1}^{n} \hat{p}_i - \sum_{i=1}^{n} p_i|
\leq (n-1)u' \cdot \sum_{i=1}^{n} |\hat{p}_i| + u' \sum_{i=1}^{n} |p_i|
\leq (n-1)u' \cdot (1 + u') \sum_{i=1}^{n} |p_i| + u' \sum_{i=1}^{n} |p_i|
\leq nu \sum_{i=1}^{n} |p_i| = nu|x^T||y|.
\]

To consider the underflow, we add a constant with a multiple of $\frac{1}{2}S_{\min}$ to (38). In Theorem 4.3,
underflow may occur in $fl(x_i y_i)$, $i = 1, 2, \ldots, n$, such that from Theorem 2.10,

\[
|\text{float}(x^T y) - x^T y| \leq nu|x^T||y| + \frac{n}{2}S_{\min}.
\]

We introduce other variant of the rounding error analysis for the sum of floating-point numbers [18].

Theorem 4.4 (Rump [18]) For $x, y \in \mathbb{F}_n$, if $(n-1)u \leq 1$ and no underflow occurs, then

\[
|\text{float}(x^T y) - x^T y| \leq (n+2)u \cdot ufp(\text{float}(|x^T||y|)),
\]

where the computational order of the left and right $\text{float}(\cdot)$ in (39) must be the same.

Proof Let $p_i = x_i y_i$, $\hat{p}_i = fl(x_i y_i)$, $\hat{s}_n = float\left(\sum_{i=1}^{n} \hat{p}_i\right)$, and $\hat{S}_n = float\left(\sum_{i=1}^{n} \hat{p}_i\right)$. The compu-
tational order of $\hat{s}_n$ and $\hat{S}_n$ must be the same. From Theorem 4.2 and the assumption of $n$,

\[
|\hat{S}_n - \sum_{i=1}^{n} |\hat{p}_i|| \leq (n-1)u \cdot ufp(\hat{S}_n) \leq ufp(\hat{S}_n)
\]

293
is satisfied. We focus on \( n \) rounding errors in the product and \( n - 1 \) rounding error in the sum. For the sum, we use Theorem 4.2 and have
\[
|\text{float}(x^T y) - x^T y| = |s_n - \sum_{i=1}^n p_i| = |s_n - \sum_{i=1}^n \hat{p}_i + \sum_{i=1}^n \hat{p}_i - \sum_{i=1}^n p_i| \\
\leq |s_n - \sum_{i=1}^n \hat{p}_i| + |\sum_{i=1}^n \hat{p}_i - \sum_{i=1}^n p_i| \\
\leq (n-1)u \cdot \text{ufp} \left( \hat{S}_n \right) + \sum_{i=1}^n |\hat{p}_i - p_i|.
\]
(41)

From Theorem 2.10, we have \( |\hat{p}_i - p_i| \leq u|\hat{p}_i| \). Therefore, from (40), we obtain
\[
\sum_{i=1}^n |\hat{p}_i - p_i| \leq u \sum_{i=1}^n |\hat{p}_i| \leq u \left( \hat{S}_n + \text{ufp} \left( \hat{S}_n \right) \right) \leq 3u \cdot \text{ufp} \left( \hat{S}_n \right)
\]
which is substituted into (41). This completes the proof.

5. Error-free transformation

We introduce error-free transformations of floating-point numbers as defined in IEEE 754. For \( a, b \in \mathbb{F} \), we first explain the methods:

- produce \( x, y \in \mathbb{F} \), such that \( a + b = x + y \); and
- produce \( x, y \in \mathbb{F} \), such that \( a = x + y \).

These methods are very useful for developing accurate algorithms for the sum and dot product. In the subsequent subsections, we focus on the error-free transformations and their application to accurate numerical computations. We use MATLAB-like notation for the representation of the functions.

5.1 Error-free transformation of sum and subtraction

For \( a, b \in \mathbb{F} \), \( \text{fl}(a + b) \) and \( a + b \) are generally not identical. We introduce a numerical algorithm that produces \( x = \text{fl}(a + b) \) and \( y = a + b - x \in \mathbb{F} \). Assuming \( |a| \geq |b| \), Dekker [19] proposed an algorithm to obtain \( x \) and \( y \).

Theorem 5.1 (Dekker [19]) For \( a, b \in \mathbb{F} \), if \( |a| \geq |b| \) and the following function \text{FastTwoSum} is executed,
\[
\begin{align*}
a + b &= x + y, \quad |y| \leq u \cdot \text{ufp}(a + b) \\
\end{align*}
\]
(42)
is satisfied. Here, assume that no overflow occurs in the floating-point arithmetic.

\[
\text{function } [x, y] = \text{FastTwoSum}(a, b) \\
x = \text{fl}(a + b); \\
y = \text{fl}(b - (x - a));
\]

Proof First, we prove \( \text{fl}(x - a) = x - a \) for \( |a| \geq |b| \). Here, we assume that \( a \) is positive without loss of generality. If \( a \) and \( b \) have the same sign, \( \frac{1}{2}a \leq x \leq 2a \) and \( \text{fl}(x - a) = x - a \) from Theorem 2.11. If \( \text{fl}(a + b) = a + b \), the proof is trivial. Now we assume that \( a \) and \( b \) have different signs and that \( \text{fl}(a + b) \neq a + b \). This implies \( \text{fl}(a - |b|) \neq a - |b| \). Therefore, from the contraposition of Theorem 2.11, we have \( a < \frac{1}{2}|b| \) and \( a > 2|b| \). However, the former contradicts the assumption \( |a| \geq |b| \). Then, \( a > 2|b| \) implies \( \text{fl}(a + b) = \text{fl}(a - |b|) \geq \frac{1}{2}a \), and \( \text{fl}(a + b) \leq 2a \) and Theorem 2.11 gives \( \text{fl}(x - a) = x - a \).
Next, we will prove $f_1(b - (x - a)) = b - (x - a)$. If $\frac{1}{2}|a| \geq |b|$, $x = f_1(a + b) = a$, which indicates $f_1(b - (x - a)) = b - (x - a)$. If $\frac{1}{2}|a| < |b|$, from (10),

$$
|f_1(a + b) - (a + b)| \leq u \cdot \text{ufp}(a + b)
\leq u \cdot \text{ufp}(2u^{-1}b + b) = 2\text{ufp}(b)
$$

is satisfied. From $a \in 2u \cdot \text{ufp}(a)Z$, $b \in 2u \cdot \text{ufp}(b)Z$, and $|a| \geq |b|$, we obtain $a \in 2u \cdot \text{ufp}(b)Z$. Then, $x = f_1(a + b) \in 2u \cdot \text{ufp}(b)Z$ and $f_1(b - (x - a)) \in 2u \cdot \text{ufp}(b)Z$. Therefore, (43) and Theorem 2.1 prove $f_1(b - (x - a)) = b - (x - a)$. Replacing $|f_1(a + b) - (a + b)|$ in (43) with $|y|$, we have $|y| \leq u \cdot \text{ufp}(a + b)$. \hfill \Box

If we apply FastTwoSum in binary64 with $a = 1$ and $b = 2^{100}$, we have $x = 2^{100}$ and $y = 0$. Therefore, $a + b = x + y$ is not satisfied. The assumption $|a| \geq |b|$ seems to be important. However, it is shown if $a \in 2u \cdot \text{ufp}(b)Z$, $a + b = x + y$ is satisfied [6].

Knuth developed an algorithm obtaining $x$ and $y$ without the assumption for $a$ and $b$ [20].

**Theorem 5.2 (Knuth [20])** If we apply the following function TwoSum for $a, b \in \mathbb{F}$, (42) is satisfied. Here, we assume that no overflow occurs in floating-point arithmetic.

```plaintext
function [x, y] = TwoSum(a, b)
    x = f_1(a + b);
    t = f_1(x - a);
    y = f_1((a - (x - t)) + (b - t));
end
```

If overflow does not occur, the results of $[x, y] = \text{TwoSum}(a, b)$ are identical to the following results.

```plaintext
if |a| >= |b|
    [x, y] = \text{FastTwoSum}(a, b);
else
    [x, y] = \text{FastTwoSum}(b, a);
end
```

Note that the algorithm in Theorem 5.2 may be faster than the above algorithm due to there being no branch in the program.

### 5.2 A product of two floating-point numbers

We assume that no underflow occurs in floating-point arithmetic in this subsection. We introduce an algorithm for an error-free transformation of a product of two floating-point numbers.

**Theorem 5.3 (Dekker [19])** For $a, b \in \mathbb{F}$, if we execute the following function TwoProd, $a \cdot b = x + y$ holds true. Assume that no underflow occurs in the evaluations of the products.

```plaintext
function [x, y] = TwoProd(a, b)
    [a_h, a_l] = Split(a);
    [b_h, b_l] = Split(b);
    x = f_1(a \cdot b);
    y = f_1(a_h \cdot b_l - ((x - a_h \cdot b_h) - a_l \cdot b_h - a_h \cdot b_l));
end
```

The details of the function Split used in TwoProd($a, b$) are as follows.

**Theorem 5.4 (Dekker [19])** For $a, b \in \mathbb{F}$, the following function Split produces $x$ and $y$, such that $a = x + y$. 
function \[ x, y = \text{Split}(a) \]
\[
c = \text{fl}(\text{factor} \cdot a); \quad \% \text{factor} = 2^{[(\log_2 u)/2]} + 1
\]
\[
x = \text{fl}(c - (c - a));
\]
\[
y = \text{fl}(a - x);
\]
end

The floating-point evaluation of \( a \cdot b + c \) for \( a, b, c \in \mathbb{F} \) produces, at most, two rounding errors. Fused Multiply-Add (FMA) produces the nearest floating-point number to \( a \cdot b + c \). This function can be used for recent CPUs and general-purpose computing of graphics processing units. FMA is included in the IEEE 754–2008 standard. Let \( \text{FMA}(a, b, c) \) be a function computing \( a \cdot b + c \) using FMA.

If FMA can be used, we can obtain the same results as in TwoProd more simply.

**Theorem 5.5** For \( a, b \in \mathbb{F} \), the following function TwoProdFMA produces \( x \) and \( y \), such that \( a \cdot b = x + y \).

\[
\text{function } [x, y] = \text{TwoProdFMA}(a, b)
\]
\[
x = \text{fl}(a \cdot b);
\]
\[
y = \text{FMA}(a, b, -x);
\]
end

5.3 Accurate algorithms for the sum and dot product

We introduce accurate numerical algorithms for the sum and dot product based on error-free transformations.

For \( p \in \mathbb{F}^n \), we introduce an algorithm [21] that produces \( p' \in \mathbb{F}^n \) such that \( \sum_{i=1}^{n} p_i = \sum_{i=1}^{n} p'_i \) (\( p' \in \mathbb{F}^n \)). Here \( p \) and \( p' \) are equivalent in terms of the sum of the elements.

**Theorem 5.6** (Ogita, Oishi, and Rump [21]) For \( p \in \mathbb{F}^n \), we execute the following function VecSum. Then,

\[
\sum_{i=1}^{n} p_i = \sum_{i=1}^{n} p'_i, \quad \sum_{i=1}^{n-1} |p'_i| \leq \gamma_{n-1} \sum_{i=1}^{n} |p_i|
\]
is satisfied.

\[
\text{function } p' = \text{VecSum}(p)
\]
\[
[\beta_1, p'_1] = \text{TwoSum}(p_1, p_2);
\]
\[
\text{for } i = 2 : n - 1
\]
\[
[\beta_i, p'_i] = \text{TwoSum}(\beta_{i-1}, p_{i+1});
\]
end
\[
p'_{n} = \beta_{n-1};
\]
end

**Proof** Theorem 5.2 immediately results in \( \sum_{i=1}^{n} p_i = \sum_{i=1}^{n} p'_i \). From Theorem 5.2, we have \( |p'_i| \leq u \cdot \text{ufp}(\beta_i), i = 1, 2, \ldots, n \), such that we obtain

\[
\sum_{i=1}^{n-1} |p'_i| \leq \sum_{i=1}^{n-1} u \cdot \text{ufp}(\beta_i) \leq \sum_{i=1}^{n-1} u|\beta_i| \leq (n - 1)u \max_k(|\beta_k|).
\]

\( \text{fl} \left( \sum_{i=1}^{k} p_i \right) \) means \( \text{fl}(\cdots \text{fl}(\text{fl}(p_1 + p_2) + p_3) + \cdots) \), i.e., floating-point summation with a recursive order. \( \beta_{k-1} = \text{fl} \left( \sum_{i=1}^{k} p_i \right) \) and Theorem 4.1 lead to

\[
|\beta_{k-1}| \leq (1 + (k - 1)u) \sum_{i=1}^{k} |p_i|.
\]
and we obtain

\[(n - 1)u \max_k(\beta_k) \leq (n - 1)u(1 + (k - 1)u) \sum_{i=1}^{k} |p_i| \leq \gamma_{n-1} \sum_{i=1}^{n} |p_i|.
\]

This completes the proof. \(\square\)

Because \(p'_n = f1\left(\sum_{i=1}^{n} p_i\right)\) using the function VecSum, we can avoid any loss of information due to rounding errors by \(p'_i\) \((1 \leq i \leq n - 1)\). Using \(p'_i\) \((1 \leq i \leq n - 1)\), we can develop accurate numerical algorithms.

Next, for \(p \in \mathbb{F}^n\) and \(K \in \mathbb{N}\) \((K \geq 2)\), we introduce the following accurate numerical algorithms \cite{21}.

**Theorem 5.7 (Ogita, Rump, and Oishi \cite{21})** For \(p \in \mathbb{F}^n\), if the following function is executed with \(K = 2\), then,

\[
|c - \sum_{i=1}^{n} p_i| \leq u \sum_{i=1}^{n} |p_i| + \gamma_{n-1}^{2} \sum_{i=1}^{n} |p_i|
\]

is satisfied. Moreover, for \(K \geq 3\), if \(4(n - 1)u \leq 1\),

\[
|c - \sum_{i=1}^{n} p_i| \leq (u + 3\gamma_{n-1}^{2}) \sum_{i=1}^{n} |p_i| + \gamma_{2(n-2)}^{K} \sum_{i=1}^{n} |p_i|
\]

holds true.

**Proof** Due to the long proof of (45), we only explain the proof of (44). See \cite{21} for the proof of (45). Let

\[
[\beta_1, \beta'_1] = \text{TwoSum}(p_1, p_2), \quad [\beta_i, \beta'_i] = \text{TwoSum}(\beta_{i-1}, p_{i+1}), \quad 2 \leq i \leq n - 1.
\]

Note that by putting \(p'_n = \beta_{n-1} - 1, p^{(1)} = p'\). Let \(\tau = f1\left(\sum_{i=1}^{n-1} p'_i\right)\). From Theorem 4.1,

\[
|\delta_1| \leq (n - 2)u \sum_{i=1}^{n-1} |p'_i|, \quad \delta_1 := \tau - \sum_{i=1}^{n-1} p'_i
\]

is satisfied. From Theorem 2.9,

\[
|\delta_2| \leq u|\beta_{n-1} + \tau|, \quad \delta_2 := f1(\beta_{n-1} + \tau) - (\beta_{n-1} + \tau)
\]

holds true. From \(c = f1(\beta_{n-1} + \tau)\), we have

\[
|c - \sum_{i=1}^{n} p_i| = f1(\beta_{n-1} + \tau) - \sum_{i=1}^{n} p_i \leq |\beta_{n-1} + \tau + \delta_2 - \sum_{i=1}^{n} p_i|
\]

\[
= |\beta_{n-1} + \sum_{i=1}^{n-1} p'_i + \delta_1 + \delta_2 - \sum_{i=1}^{n} p_i| \leq |\delta_1| + |\delta_2|.
\]

Because \(|\delta_2| \leq u|\beta_{n-1} + \tau| = u|\beta_{n-1} + \sum_{i=1}^{n-1} p'_i + \delta_1| \leq u|\sum_{i=1}^{n} p_i| + u|\delta_1|\), (46) and Theorem 5.6 yield

\[
297
\]
\[|\delta_1| + |\delta_2| \leq u \sum_{i=1}^{n} p_i + (1 + u)|\delta_1| \]
\[\leq u \sum_{i=1}^{n} p_i + (1 + u)(n - 2)u \sum_{i=1}^{n-1} |p_i'| \]
\[\leq u \sum_{i=1}^{n} p_i + \gamma_{n-1}^2 \sum_{i=1}^{n} |p_i|. \]

Compared to Theorem 4.1, the constant of \( \sum_{i=1}^{n} |p_i| \) in (44) is \( O(u^2) \). In addition, the constant of \( \sum_{i=1}^{n} |p_i| \) in (45) is \( O(u^K) \). These two points explain why we can expect an accurate computed result.

We introduce the following accurate numerical algorithm [21] for \( x^T y \) for \( x, y \in \mathbb{F}^n \).

**Theorem 5.8 (Ogita, Rump, and Oishi [21])** For \( x, y \in \mathbb{F}^n \), if we execute the following function \( \text{Dot2} \), then,
\[|c - x^T y| \leq u|x^T y| + \gamma_2^2|x^T||y|\]

is satisfied barring underflow.

function \( c = \text{Dot2}(x, y) \)
\[ [p, s] = \text{TwoProd}(x_1, y_1); \]
for \( i = 2 : n \)
\[ [h, r] = \text{TwoProd}(x_i, y_i); \]
\[ [p, q] = \text{TwoSum}(p, h); \]
\[ s = fl(s + (r + q)); \]
end
\[ c = p + s; \]
end

This algorithm produces a computed result for \( x^T y \) as if we were using double the working precision, and the result is rounded to the working precision.

**Theorem 5.9 (Ogita-Rump-Oishi [21])** For \( x, y \in \mathbb{F}^n \), if we execute the following function \( \text{DotK} \), then, for \( K \geq 3 \),
\[|c - x^T y| \leq (u + 2\gamma_{4n-2}^2)|x^T y| + \gamma_K^2|x^T||y|\]

is satisfied barring underflow.

function \( c = \text{DotK}(x, y, K) \)
\[ [p, r_1] = \text{TwoProd}(x_1, y_1); \]
for \( i = 2 : n \)
\[ [h, r_i] = \text{TwoProd}(x_i, y_i); \]
\[ [p, r_{n+i-1}] = \text{TwoSum}(p, h); \]
end
\[ r_{2n} = p; \]
\[ c = \text{SumK}(r, K - 1); \]
end

We omit the proofs of these theorems (see [21] for details). It is possible to extend the algorithm \( \text{DotK} \) to produce an approximation and its error bound.
5.4 Accuracy guaranteed result
We introduced the accurate algorithm DotK for the dot product. Even if DotK is applied, the result may still be inaccurate for ill-conditioned cases. We now introduce an accurate algorithm that always produces a faithfully rounded result.

For \( a \in \mathbb{F} \) and a power of two \( \sigma \in \mathbb{F} \) \( (\sigma \geq |a|) \), the following algorithm transforms \( a \) into an unevaluated sum of two floating-point numbers [6].

**Theorem 5.10 (Rump, Ogita, and Oishi [22])** For \( a, \sigma \in \mathbb{F} \) \( (\sigma = 2^k \geq |a|, k \in \mathbb{Z}) \), if we execute the following function ExtractScalar, then

\[
a = x + y, \quad x \in u\sigma\mathbb{Z}, \quad |y| \leq u\sigma
\]

is satisfied barring overflow.

```plaintext
function \([x, y] = \text{ExtractScalar}(a, \sigma)\)
\[x = \text{fl}\left((a + \sigma) - \sigma\right);\]
\[y = \text{fl}\left(a - x\right);\]
end
```

**Proof** For Theorem 5.1, let \( a = \sigma \) and \( b = a \). This immediately leads to \( a = x + y \) and \( |y| \leq u\sigma \). In addition, from Theorem 2.13, \( \text{fl}(\sigma + a) \in u\sigma\mathbb{Z} \), which yields \( x \in u\sigma\mathbb{Z} \).

For Theorem 5.10,\n
\[
\sigma \geq 2^M a, \quad M \in \mathbb{N} \Rightarrow |x| \leq 2^{-M}\sigma \quad (47)
\]

is also satisfied (we omit the proof).

Next, we introduce an algorithm that is the basis for accurate numerical algorithms in [6].

**Theorem 5.11 (Rump, Ogita, and Oishi [6])** For \( p \in \mathbb{F}^n \), assume that \( \sigma = 2^k \in \mathbb{F}, \quad k \in \mathbb{Z} \) satisfies

\[
\sigma \geq 2^{\lceil \log_2 n \rceil}, 2^{\max_{1 \leq i \leq n} |p_i|}.
\]

If we execute the following function ExtractVector, then,

\[
\sum_{i=1}^{n} p_i = \tau_n + \sum_{i=1}^{n} p'_i \quad (48)
\]

is satisfied barring overflow.

```plaintext
function \([\tau_n, p'] = \text{ExtractVector}(p, \sigma)\)
\[\tau_0 = 0;\]
for \( i = 1 : n \)
\[ [q_i, p'_i] = \text{ExtractScalar}(p_i, \sigma);\]
\[ \tau_i = \text{fl}(\tau_{i-1} + q_i);\]
end
end
```

**Proof** From Theorem 5.10, \( p_i = q_i + p'_i \). Therefore, if \( \tau_i \) is computed without a rounding error, (48) is satisfied. Therefore, we prove \( \tau_i = \text{fl}(\tau_{i-1} + q_i) = \tau_{i-1} + q_i, \quad i = 1, 2, \ldots, n \).

For all \( 1 \leq i \leq n \), Theorem 5.10 and (47) lead to

\[
q_i \in u\sigma\mathbb{Z}, \quad |q_i| \leq M^{-1}\sigma, \quad M := 2^{\lceil \log_2 n \rceil}
\]

and

\[
\tau_{i-1} + q_i \in u\sigma\mathbb{Z}, \quad |\tau_{i-1} + q_i| \leq |\tau_{i-1}| + |q_i| \leq i \cdot M^{-1}\sigma \leq \sigma
\]

is proved recursively. From Theorem 2.2, \( \tau_{i-1} + q_i \in \mathbb{F} \), which gives \( \tau_i = \tau_{i-1} + q_i \).
Theorem 5.12 (Rump, Ogita, and Oishi [6]) For $p \in \mathbb{F}^n$, if we execute the following function \textit{Transform}, then,
\[ \sum_{i=1}^{n} p_i = \tau_1 + \tau_2 + \sum_{i=1}^{n} q_i, \quad \tau_1, \tau_2 \in \mathbb{F} \] is satisfied barring overflow.

\begin{verbatim}
function \[ \tau_1, \tau_2, q, \sigma \] = Transform(p)
q = p; \mu = max_{1 \leq i \leq n} |p_i|;
if \mu == 0, \tau_1 = \tau_2 = \sigma = 0; return; end
M = NextPowerTwo(n + 2); \sigma' = M \cdot NextPowerTwo(\mu);
t' = 0;
repeat
    t = t'; \sigma = \sigma';
    \[ \tau, q \] = ExtractVector(\sigma, q);
    t' = fl(t + \tau);
    if t' = 0, \[ \tau_1, \tau_2, q, \sigma \] = Transform(q); return; end
    \sigma' = fl((M \cdot u) \cdot \sigma);
until \|t'\| \geq fl((M \cdot (M \cdot u)) \cdot \sigma) or \sigma \leq \frac{1}{2}u^{-1}S_{\min}
[\tau_1, \tau_2] = FastTwoSum(t, \tau);
end
\end{verbatim}

NextPowerTwo(a) for $0 \neq a \in \mathbb{F}$ in the above algorithm produces $2^{\lceil \log_2 |a| \rceil}$, i.e., the minimum power of two that is greater than $|a|$. Note that the MATLAB function \textit{nextpow2}(a) produces $\lceil \log_2 |a| \rceil$. If $\tau_1 \neq 0, |\tau_1| \gg |\tau_2| + \sum_{i=1}^{n} |q_i|$. Therefore, cancellation does not occur in $s = \tau_1 + \tau_2 + \sum_{i=1}^{n} q_i$. This implies that $ufp(\tau_1) \approx ufp(s)$.

Finally, we introduce an accurate summation algorithm [6] that produces an accuracy guaranteed computed result.

Theorem 5.13 (Rump, Ogita, and Oishi [6]) For $p \in \mathbb{F}^n$, if we execute the following function \textit{AccSum}, then,
\[ \sum_{i=1}^{n} p_i \in \mathbb{F} \implies \sum_{i=1}^{n} p_i = c \]
\[ \sum_{i=1}^{n} p_i \notin \mathbb{F} \implies \sum_{i=1}^{n} p_i - c \leq 2u \cdot ufp(c) \] is satisfied barring overflow.

\begin{verbatim}
function c = AccSum(p)
[\tau_1, \tau_2, p'] = Transform(p);
c = fl(\tau_1 + (\tau_2 + \left(\sum_{i=1}^{n} p'_i\right)));
end
\end{verbatim}

From (49), the computed result of \textit{AccSum} is called faithful rounding. In addition, Rump, Ogita, and Oishi developed an algorithm [22] that produces $c \in \mathbb{F}$ such that
\[ \sum_{i=1}^{n} p_i - c \leq u \cdot ufp(c). \] (50)

We should mention the difference between \textit{SumK} and \textit{AccSum}. In (44) and (45), if $\left| \sum_{i=1}^{n} p_i \right|$ is small but $\sum_{i=1}^{n} |p_i|$ is large, the relative error of the result computed by \textit{SumK} is large. Conversely, the relative
error of the result computed by AccSum does not depend on these factors, which can be seen from (49).

**Conclusion**

In this paper, we summarized important techniques for verified numerical computations: the fundamental of floating-point numbers and floating-point arithmetic, interval arithmetic, rounding error analyses and accurate numerical algorithms for the sum and dot product.

**Acknowledgments**

The authors wish to thank the anonymous reviewer for fruitful comments. We thank Prof. Shin’ichi Oishi and Prof. Siegfried M. Rump for giving us the opportunity for writing this paper. Many results are obtained thanks to the support by JST/CREST.

**References**

[1] S. Oishi, K. Ichihara, M. Kashiwagi, K. Kimura, X. Liu, H. Masai, Y. Morikura, T. Ogita, K. Ozaki, S.M. Rump, K. Sekine, A. Takayasu, and N. Yamanaka, *Principle of Verified Numerical Computations*, Corona publisher, Tokyo, 2018 (in Japanese).

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

[3] S. Oishi, *Verified Numerical Computation*, Corona publisher, Tokyo, 1999 (in Japanese).

[4] S.M. Rump, “Verification methods: Rigorous results using floating-point arithmetic,” *Acta Numerica*, vol. 19, pp. 287–449, 2010.

[5] T. Hoshi, T. Ogita, K. Ozaki, and T. Terao, “An a posteriori verification method for generalized real-symmetric eigenvalue problems in large-scale electronic state calculations,” submitted to publication.

[6] 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.

[7] C.-P. Jeannerod and S.M. Rump, “On relative errors of floating-point operations: Optimal bounds and applications,” *Math. Comp.*, vol. 87, no. 311, pp. 803–819, 2018.

[8] P.H. Sterbenz, *Floating-Point Computation*, Prentice-Hall, Englewood Cliffs, 1974.

[9] T. Sunaga, “Theory of interval algebra and its application to numerical analysis,” *RAAG memoirs*, vol. 2, pp. 29–46, 1958.

[10] R.E. Moore, *Interval Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1966.

[11] G. Alefeld and J. Herzberger, *Introduction to Interval Computation*, Academic Press, New York, 1983.

[12] A. Neumaier, *Interval Methods for Systems of Equations*, Cambridge University Press, Cambridge, 2008.

[13] R.E. Moore, R.B. Kearfott, and M.J. Cloud, *Introduction to Interval Analysis*, Cambridge University Press, Cambridge, 2009.

[14] J.L.D. Comba and J. Stolfi, “Affine arithmetic and its applications to computer graphics,” *Proc. in SIBGRAPI’93 - VI Simposio Brasileiro de Computacao Grafica e Processamento de Imagens* (Recife, Brazil), pp. 9–18, 1993.

[15] S.M. Rump, “Fast and parallel interval arithmetic,” *BIT Numer. Math.*, vol. 39, no. 3, pp. 539–560, 1999.

[16] C.-P. Jeannerod and S.M. Rump, “Improved error bounds for inner products in floating-point arithmetic,” *SIAM J. Matrix Anal. Appl.*, vol. 34, no. 2, pp. 338–344, 2013.

[17] N.J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, 2002.

[18] S.M. Rump, “Error estimation of floating-point summation and dot product,” *BIT Numer. Math.*, vol. 52, no. 1, pp. 201–220, 2012.

[19] T.J. Dekker, “A floating-point technique for extending the available precision,” *Numer. Math.*, vol. 18, no. 3, pp. 224–242, 1971.
[20] D.E. Knuth, *Art of Computer Programming, vol. 2: Seminumerical Algorithms*, Addison-Wesley Professional, Massachusetts, 1997.

[21] T. Ogita, S.M. Rump, and S. Oishi, “Accurate sum and dot product,” *SIAM J. Sci. Comput.*, vol. 26, no. 6, pp. 1955–1988, 2005.

[22] S.M. Rump, T. Ogita, and S. Oishi, “Accurate floating-point summation part II: Sign, $K$-fold faithful and rounding to nearest,” *SIAM J. Sci. Comput.*, vol. 31, no. 2, pp. 1269–1302, 2008.