Nr.: FIN-09-2011

Summing Expansions Exactly and Efficiently

Marc Mörig, Silvio Weging

Arbeitsgruppe Algorithmische Geometrie

Nr.: FIN-09-2011

Summing Expansions Exactly and Efficiently

Marc Mörig, Silvio Weging

Arbeitsgruppe Algorithmische Geometrie

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Marc Mörig
Postfach 4120
39016 Magdeburg
E-Mail: marc.moerig@ovgu.de

http://www.cs.uni-magdeburg.de/Technical_reports.html
Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 09.11.2011

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

Summing Expansions Exactly and Efficiently

Marc Mörig Silvio Weging

Department of Simulation and Graphics,
Otto-von-Guericke University Magdeburg, Germany,
marc.moerig@ovgu.de

November 9, 2011

Abstract

A floating-point expansion is a sequence of fixed precision floating-point numbers $a_1, \ldots, a_n$, representing the sum $A = \sum_{i=1}^{n} a_i$. We address the problem of computing A exactly and efficiently as an arbitrary precision floating-point number. In an expansion the subsequence of nonzero summands is ordered by magnitude and summands are pairwise nonoverlapping: the most significant nonzero bit of the smaller summand is less significant than the least significant nonzero bit of the larger summand. This property allows to compute A more efficiently than simply summing up $a_1, \ldots, a_n$ with arbitrary precision arithmetic.

1 Introduction

Sums of floating-point numbers can be used to efficiently perform accurate numerical computations that require slightly more precision than hardware floating-point arithmetic offers [9, Chapter 14]. A sequence of summands $a_1, \ldots, a_n$ is used to represent the sum $A = \sum_{i=1}^{n} a_i$. Examples include double-double or more generally multi word arithmetic, where the number of summands is fixed [2, 3], but also arithmetic with a variable number of summands [11, 12, 17].
We address the problem of converting sums exactly into an arbitrary precision floating-point number. As input we accept sums where the subsequence of nonzero summands is ordered by magnitude. Stray zero summands are explicitly allowed. Furthermore nonzero summands must be pairwise nonoverlapping: the most significant nonzero bit of the smaller summand is less significant than the least significant nonzero bit of the larger summand. Sums with these properties are called expansions and were introduced by Shewchuk [17]. Shewchuks work is based on work by Priest [11, 12] who uses a different, more restrictive type of sum. We will see, that these and other types of sums that are used in the literature are expansions too and hence can be handled our algorithms.
Interestingly, the problem of converting an expansion into an arbitrary precision floating-point number seems to have not been considered previously. After all there is a trivial solution: convert each summand individually, then compute the sum exactly using arbitrary precision arithmetic. This is, however, a lot of unnecessary overhead. Since the summands are nonoverlapping, all the bits of the sum are essentially known. The only obstruction is that summands may have different signs. Our approach therefore first turns an expansion into a monotone expansion, where all summands have the same sign. This algorithm uses floating-point arithmetic only and is independent of the target arbitrary precision type. Then we convert the monotone expansion by copying the mantissae of the summands to the correct position in the mantissa of an arbitrary precision floating-point number. As target floating-point numbers we use both MPFR [8] and `leda::bigfloat` [5], but other number types could be used as well.
The conversion problem came to our attention through work on our number type `Real_algebraic` [6, 7]. There, we attempt to avoid or postpone expensive operations by computing exactly with expansions. Ultimately a conversion from expansion to arbitrary precision floating-point number may be necessary. Experiments suggest that this conversion is a major bottleneck in our approach and improvement seemed possible. An implementation of our conversion methods is available on the `Real_algebraic` website [13] and will be part of `Real_algebraic` soon.
The paper is organized as follows. In Section 2 we subsume basic facts about floating-point numbers that are required later. In Section 3 we present a new algorithm that turns an expansion into a monotone expansion and prove its correctness. In Section 4 we give an algorithm that converts a monotone expansion into an MPFR floating-point number. We also give two algorithms that convert general expansions, one by transforming into a monotone expansion and another one by splitting the expansion into two monotone expansion. Finally in Section 5 we compare the efficiency of our new approaches to the trivial approach.

2 Floating-Point Numbers: Notation and Basic Facts

Throughout this paper we use binary floating-point numbers, i.e., real numbers that have a finite binary representation. We start with two functions that let us control the most significant and least significant nonzero bit in such a representation. Let $x \in \mathbb{R}$, then we define the *most significant bit* of x as

$$\operatorname{msb}(0) = 0, \qquad \operatorname{msb}(x) = 2^{\lfloor \log_2 |x| \rfloor} \text{ for } x \neq 0. \tag{1}$$

This quantity was originally introduced as ufp or *unit in the first place* by Rump et al. [15] to allow better analysis of floating-point algorithms.[1]

[1]We use the name msb here as it better matches the complimentary name lsb. The name ulp or unit in the last place that would match ufp already has a meaning different from lsb.

We can view numbers with a finite binary representation as elements of a ring $\sigma\mathbb{Z}$, where σ is a power of two. If $x \in \mathbb{R}$ has a finite binary representation, then we define the *least significant bit* as

$$\operatorname{lsb}(0) = 0, \qquad \operatorname{lsb}(x) = \max\{\sigma \mid x \in \sigma\mathbb{Z},\ \sigma = 2^k,\ k \in \mathbb{Z}\} \ \text{ for } x \neq 0. \tag{2}$$

Using msb and lsb we can now define the set of floating-point numbers.

Definition 1. *Let $\eta, \varepsilon_{\mathtt{m}}, \tau$ be powers of two such that $\eta < 1, \varepsilon_{\mathtt{m}} < \tau$ and let $\mathbb{F} = \mathbb{F}(\eta, \varepsilon_{\mathtt{m}}, \tau)$ be the set of floating-point numbers. $\mathbb{F}$ contains 0. Let $0 \neq x \in \mathbb{R}$ have a finite binary representation, then $x \in \mathbb{F}$ if and only if*

$$\eta \leq \operatorname{lsb}(x), \qquad \operatorname{msb}(x) \leq \tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\operatorname{lsb}(x), \qquad \operatorname{msb}(x) \leq \tau. \tag{3}$$

Let furthermore $\overline{\mathbb{F}} = \mathbb{F} \cup \{\pm\infty\}$.

Hence $x \neq 0$ is a floating-point number if its least significant bit is at least η, its most significant bit is at most τ and the binary representation of x requires at most $p = \log_2(\varepsilon_{\mathtt{m}}^{-1})$ consecutive bits. p is called the *precision* of the floating-point set. Most floating-point numbers f have p bits, but if $\operatorname{msb}(f)$ is smaller than $\frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$, less than p bits are available. Floating-point numbers with $\operatorname{msb}(f) < \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$ are called *denormalized*. The largest number in $\mathbb{F}$ is $2\tau(1-\varepsilon_{\mathtt{m}})$ with most significant bit τ and p consecutive nonzero bits. Note that $\mathbb{F}$ is symmetric, i.e., $\mathbb{F} = -\mathbb{F}$.
The binary floating-point formats defined by the IEEE 754-2008 standard [4] can be obtained from Definition 1 for certain values of $\eta, \varepsilon_{\mathtt{m}}, \tau$. Of special interest to us is the `binary64` format with $\eta = 2^{-1074}$, $\varepsilon_{\mathtt{m}} = 2^{-53}$ and $\tau = 2^{1023}$ that is in widespread use and that we also use in our implementation.
A floating-point number $f \neq 0$ can be decomposed into *sign* $s \in \{-1, 1\}$, *mantissa* $m \in \{0,1\}^p$ and *exponent* $e \in \mathbb{Z}$ such that

$$f = s \cdot m \cdot 2^e. \tag{4}$$

The decomposition is usually made unique by some normalization condition on m. We can for example require that the leading bit in m is nonzero and assign a value of one to it, i.e., m is interpreted as number in $[1, 2)$. Another variant is to assign a value of one to the last bit of m, i.e., m is interpreted as integer. In this case m and e are not necessarily unique. We will state the normalizing conditions whenever we refer to mantissa or exponent of a floating-point number.
After having fixed the set of numbers we need to discuss basic arithmetic operations over $\mathbb{F}$. Since we only have a finite set of numbers, rounding is inevitable. Arithmetic is performed as if first the exact result is computed and then rounded to a floating-point number. We only consider rounding to a nearest floating-point number here.
Since there are only finitely many floating-point numbers, rounding is essential to floating-point arithmetic. Let $\mathbf{fl} : \mathbb{R} \to \overline{\mathbb{F}}$ be a function that rounds $x \in \mathbb{R}$ to a nearest floating-point number. This fixes $\mathbf{fl}$ except for $|x| > 2\tau(1-\varepsilon_{\mathtt{m}})$ and when x falls exactly halfway between two floating-point numbers. Thus, let $\mathbb{F}' = \mathbb{F}(\eta, \varepsilon_{\mathtt{m}}, \tau')$ be a floating-point set with $\tau' > \max\{\tau, \operatorname{msb}(x)\}$ and f' be a floating-point number in $\mathbb{F}'$ closest to x. If $f' \in \mathbb{F}$ then $\mathbf{fl}(x) = f'$, otherwise $\mathbf{fl}(x) = \pm\infty$. If x is exactly halfway between two floating-point numbers in $\mathbb{F}'$, then select any of these numbers, however such that $\mathbf{fl}(-x) = -\mathbf{fl}(x)$ is preserved for all x. *Overflow* is said to occur, when x is rounded to a different number in the modified floating-point set $\mathbb{F}'$ than in $\mathbb{F}$. This coincides with rounding to $\pm\infty$ for our rounding function $\mathbf{fl}$. For $x, y \in \mathbb{R}$, $\mathbf{fl}$ satisfies:

$$x \leq y \quad \Rightarrow \quad \mathbf{fl}(x) \leq \mathbf{fl}(y) \tag{5}$$

$$\mathbf{fl}(x) < \mathbf{fl}(y) \quad \Rightarrow \quad x < y \tag{6}$$

Equation (5) follows directly from rounding to the nearest floating-point number and Equation (6) is its contraposition. A nice consequence of Equation (5) (with $x \in \mathbb{F}$) is, that we never jump over a floating-point number in the process of rounding. The two IEEE 754-2008 rounding direction attributes *roundTiesToEven*, which is the default rounding attribute, and *roundTiesToAway* can both serve as rounding function $\mathbf{fl}$.

We will now show a refined variant of the standard error estimate for floating-point numbers which is due to Rump et al. [15]. As tools but also for later use, we introduce floating-point predecessor and successor for any real number.

Definition 2. *Let* $x \in \mathbb{R}$, *then* $\operatorname{pred}, \operatorname{succ} : \mathbb{R} \to \overline{\mathbb{F}}$ *with*

$$\operatorname{pred}(x) = \max\{f \in \overline{\mathbb{F}} \mid f < x\} \qquad \operatorname{succ}(x) = \min\{f \in \overline{\mathbb{F}} \mid x < f\} \tag{7}$$

give the predecessor *and* successor *of* x.

For floating-point numbers we can give predecessor and successor explicitly [14, 15]. Let $0 \le f \in \mathbb{F}$. If $f \le \frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta$ then

$$\operatorname{pred}(f) = f - \eta, \qquad \operatorname{succ}(f) = f + \eta. \tag{8}$$

If $\frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta < f < 2\tau(1-\varepsilon_{\mathrm{m}})$, then

$$\begin{aligned} f \neq \operatorname{msb}(f) \quad &\Rightarrow \quad \operatorname{pred}(f) = f - 2\varepsilon_{\mathrm{m}}\operatorname{msb}(f), \quad \operatorname{succ}(f) = f + 2\varepsilon_{\mathrm{m}}\operatorname{msb}(f), \\ f = \operatorname{msb}(f) \quad &\Rightarrow \quad \operatorname{pred}(f) = f - \varepsilon_{\mathrm{m}}\operatorname{msb}(f), \quad \operatorname{succ}(f) = f + 2\varepsilon_{\mathrm{m}}\operatorname{msb}(f). \end{aligned} \tag{9}$$

Finally, if $f = 2\tau(1-\varepsilon_{\mathrm{m}})$, then

$$\operatorname{pred}(f) = f - 2\varepsilon_{\mathrm{m}}\operatorname{msb}(f), \qquad \operatorname{succ}(f) = +\infty. \tag{10}$$

For $f < 0$ analogous equalities hold. In floating-point arithmetic, basic operations are performed as if the mathematically correct result is computed and then rounded to a floating-point number. That is, for $a, b \in \mathbb{F}$, $\circ \in \{+,-,\cdot,/\}$ and $a \circ b \in \mathbb{R}$, the value $\mathbf{fl}(a \circ b)$ is computed. We often abbreviate $\mathbf{fl}(a \circ b)$ with $a \circledcirc b$. The following theorem bounds the error of these operations.

Theorem 3. [15]
Let $x \in \mathbb{R}$ *and* $f \in \mathbb{F}$ *with* $\mathbf{fl}(x) = f$. *Then*

$$|f| \le \tfrac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta \quad \Rightarrow \quad f = x + \mu \quad \textit{with } |\mu| \le \tfrac{1}{2}\eta \tag{11}$$

$$|f| \ge \tfrac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta \quad \Rightarrow \quad f = x + \delta \quad \textit{with } |\delta| \le \varepsilon_{\mathrm{m}}\operatorname{msb}(f) \tag{12}$$

Proof. Since both $\mathbb{F}$ and $\mathbf{fl}$ are symmetric, we only need to consider the case $f \ge 0$. Note that for $|f| = \frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta$ both estimates are equal: $\varepsilon_{\mathrm{m}}\operatorname{msb}(f) = \frac{1}{2}\eta$. First assume $x \in \mathbb{F}$, then $|x - f| = 0$ and there is nothing to show. For $x \notin \mathbb{F}$, let $f' = \operatorname{pred}(x) \in \mathbb{F}$. In case $\operatorname{succ}(f') \notin \mathbb{F}$, let $\operatorname{succ}(f') = f' + 2\varepsilon_{\mathrm{m}}\operatorname{msb}(f')$, which would be the successor of f' in some $\mathbb{F}'$ with larger τ'. Now either $f = f'$ or $f = \operatorname{succ}(f')$, whichever is closer. Therefore

$$|x - f| \le \tfrac{1}{2}(\operatorname{succ}(f') - f'). \tag{13}$$

When $f \le \frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta$, then also $f' \le \frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta$ and by Equation 8

$$|x - f| \le \tfrac{1}{2}(f' + \eta - f') = \tfrac{1}{2}\eta. \tag{14}$$

If $f > \frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta$, then $f' \ge \frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta$. If $f' = \frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta$, then again

$$|x - f| \le \tfrac{1}{2}\eta = \varepsilon_{\mathrm{m}}\operatorname{msb}(f') \le \varepsilon_{\mathrm{m}}\operatorname{msb}(f). \tag{15}$$

Finally if $f' > \frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta$, then by Equation 9

$$|x - f| \le \tfrac{1}{2}(f' + 2\varepsilon_{\mathrm{m}}\operatorname{msb}(f') - f') = \varepsilon_{\mathrm{m}}\operatorname{msb}(f') \le \varepsilon_{\mathrm{m}}\operatorname{msb}(f), \tag{16}$$

which concludes the proof. □

Next, we turn to addition and subtraction, the main issue of this paper. By symmetry of $\mathbb{F}$ and **fl** we can conclude that

$$a \oplus b = b \oplus a = b \ominus -a = -(a \ominus b) \tag{17}$$

and so forth, so all properties of $\oplus$ also hold for $\ominus$ as long as no condition on the signs of a and b is imposed. For the same reasons we can retreat to a special case that is symmetric to the remaining cases in many proofs.

To control the lower bits arising in computations we view floating-point numbers as part of the ring $\sigma\mathbb{Z}$, where σ is always a power of two. A statement such as $f \in \sigma\mathbb{Z}$ tells us that $f = 0$ or $\sigma \leq \operatorname{lsb}(f)$. Some useful properties are:

$$\begin{aligned} \mathbb{F} &\quad\subset\quad \eta\mathbb{Z} && (18)\\ \sigma_1 \geq \sigma_2 &\quad\Rightarrow\quad \sigma_1\mathbb{Z} \subseteq \sigma_2\mathbb{Z} && (19)\\ a, b \in \sigma\mathbb{Z} &\quad\Rightarrow\quad a + b \in \sigma\mathbb{Z} && (20)\\ a, b \in \sigma\mathbb{Z} \cap \mathbb{F},\ \ u \oplus v \in \mathbb{F} &\quad\Rightarrow\quad u \oplus v \in \sigma\mathbb{Z} && (21) \end{aligned}$$

Equation (18), Equation (19) and Equation (20) are simply ring properties. We can derive Equation (21) from Equation (19) and Equation (20) because in the process of rounding $u + v$ to $u \oplus v$, trailing bits are removed, i.e., either $u + v = 0$ or $\eta \leq \sigma \leq \operatorname{lsb}(u + v) \leq \operatorname{lsb}(u \oplus v)$. Equation (21) allows us to keep track of the least significant bit of floating-point numbers. To demonstrate its usefulness and for later use we now show that addition and subtraction are exact in case the result falls in the range of denormalized numbers.

Lemma 4. *Let $a, b \in \mathbb{F}$ with $\operatorname{msb}(a + b) \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$. Then $a + b \in \mathbb{F}$ and hence $a \oplus b = a + b$.*

Proof. We know $a, b \in \mathbb{F}$ and hence from Equation (18) and Equation (20) that $a + b \in \eta\mathbb{Z}$. If $a + b = 0$ the claim holds. Otherwise we have $\eta \leq \operatorname{lsb}(a + b)$ and

$$\operatorname{msb}(a + b) \;\leq\; \tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta \;\leq\; \tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\operatorname{lsb}(a + b). \tag{22}$$

Furthermore $\operatorname{msb}(a + b) < \tau$ and hence $a + b \in \mathbb{F}$. □

Lemma 4 has some nice consequences. It implies for example that no nonzero number is ever rounded to zero in an addition or subtraction. Since we never jump over a floating-point number when rounding it follows for $a, b \in \mathbb{F}$ that

$$\operatorname{sign}(a \oplus b) = \operatorname{sign}(a + b). \tag{23}$$

Furthermore we get an improved version of Theorem 3. If $a, b, a \oplus b \in \mathbb{F}$ then

$$a \oplus b = a + b + \delta \qquad \text{with } |\delta| \leq \varepsilon_{\mathtt{m}} \operatorname{msb}(a \oplus b). \tag{24}$$

The name *error-free transformation* has been given to small algorithms that transform expressions of floating-point numbers into mathematically equivalent expressions [10]. We will now demonstrate one error-free transformation called FASTTWOSUM that goes back to Dekker [2]. FASTTWOSUM allows to recover the exact roundoff error of a floating-point addition. It is used frequently in algorithms that drive for exactness or increased accuracy and is a major tool in our algorithm in Section 3 too. Our proof follows Shewchuk [17], but using the machinery developed above where appropriate. First we show that the roundoff error in an addition is itself a floating-point number.

Lemma 5. *Let $a, b, a \oplus b \in \mathbb{F}$ and $\delta = a \oplus b - (a + b)$. Then $\delta \in \mathbb{F}$.*

Proof. For $\delta = 0$ there is nothing to show, so assume $\delta \neq 0$. We first assume $|a| \leq |b|$ and claim $|\delta| \leq |a| \leq |b|$. δ is the distance from $a + b$ to the closest floating-point number, hence it is at most the distance to any floating-point number, e.g., b.

$$|\delta| \leq |a + b - b| = |a| \leq |b|. \tag{25}$$

As an immediate consequence we get $\mathrm{msb}(\delta) \leq \mathrm{msb}(a) \leq \tau$. We now drop the assumption $|a| \leq |b|$. Instead assume $a \neq 0$ and let $\sigma = \mathrm{lsb}(a)$. Assume further that $b \in \sigma\mathbb{Z}$ (if not switch roles of a and b). Then we know $a, b, a+b, a \oplus b, \delta \in \sigma\mathbb{Z}$ by Equations (19) – (21). Thus it follows $\eta \leq \sigma \leq \mathrm{lsb}(\delta)$. Finally

$$\mathrm{msb}(\delta) \leq \mathrm{msb}(a) \leq \tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\,\mathrm{lsb}(a) \leq \tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\,\mathrm{lsb}(\delta) \tag{26}$$

and hence $\delta \in \mathbb{F}$. □

The other tool we need is Sterbenz Lemma. It gives a sufficient condition when the subtraction of two floating-point numbers with the same sign is free from rounding error.

Lemma 6 (Sterbenz). [18]
Let $a, b \in \mathbb{F}$ with $\frac{1}{2} \leq \frac{a}{b} \leq 2$. Then $a - b \in \mathbb{F}$ and consequently $a \ominus b = a - b$.

Proof. Note that a and b have the same sign. It suffices to show the claim for $0 < \frac{1}{2}a \leq b \leq a$ and $a - b \neq 0$, the other cases being clear or symmetric. By Equations (19) and (20) we have $\eta \leq \min\{\mathrm{lsb}(a), \mathrm{lsb}(b)\} \leq \mathrm{lsb}(a-b)$. Furthermore $a - b \leq a - \frac{1}{2}a = \frac{1}{2}a \leq b \leq a$ and thus

$$\mathrm{msb}(a-b) \leq \min\{\mathrm{msb}(a), \mathrm{msb}(b)\} \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1} \min\{\mathrm{lsb}(a), \mathrm{lsb}(b)\} \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\,\mathrm{lsb}(a-b). \tag{27}$$

Finally $\mathrm{msb}(a-b) \leq \mathrm{msb}(a) \leq \tau$ and therefore $a - b \in \mathbb{F}$. □

Theorem 7 (FASTTWOSUM). [2]
Let $a, b \in \mathbb{F}$ with $|a| \geq |b|$ and compute

$$x = a \oplus b, \qquad q = x \ominus a, \qquad y = b \ominus q. \tag{28}$$

Assume $x \in \mathbb{F}$. Then $a + b = x + y$ and $|y| \leq \varepsilon_{\mathtt{m}}\,\mathrm{msb}(x)$.

Proof. We only consider the case $a \geq 0$, the other case being symmetric. If $\frac{1}{2}a \leq -b$, then by Sterbenz Lemma $x = a \oplus b = -b \ominus a = a + b$. Then $q = b$ and $y = 0$. If $\frac{1}{2}a \geq -b$, then $\frac{1}{2}a = a - \frac{1}{2}a \leq a + b \leq 2a$ and hence by Equation (5) we have $\frac{1}{2}a \leq x \leq 2a$. Then again by Sterbenz Lemma $q = x \ominus a = x - a$. Therefore $y = b \ominus q = \mathbf{fl}(a + b - x)$. By Lemma 5 we have $a + b - x \in \mathbb{F}$ and therefore $y = a + b - x$. The bound on $|y|$ follows from Equation (24). □

2.1 Expansions

Sums of floating-point numbers can be used for accurate computations that require more precision than available from the floating-point arithmetic directly. Typically some normalization conditions are imposed on a sum and sums fulfilling these conditions are called expansions. We use the following definition by Shewchuk.

Definition 8. *(nonoverlapping, expansion)* [17]
Let $a, b \in \mathbb{F}$ with $|a| \leq |b|$. Then a and b are nonoverlapping *if there exist integers r, s such that $b = r2^s$ and $|a| < 2^s$. Otherwise a and b* overlap. *A sequence $f_1, f_2, \ldots, f_n \in \mathbb{F}$ is called an* expansion, *if the elements are pairwise nonoverlapping and they are ordered by increasing magnitude, except that any of the f_i may be zero.*

An expansion $f_1, f_2, \ldots, f_n$ represents the value $F = \sum_{i=1}^{n} f_i$, for this reason we call the elements also summands. When translating the nonoverlapping condition into our notation, it becomes clear why it is called nonoverlapping.

Lemma 9. *Let $a, b \in \mathbb{F}$ with $0 < |a| \leq |b|$. Then a and b are nonoverlapping if and only if $\mathrm{msb}(a) < \mathrm{lsb}(b)$.*

Proof. Let a, b be nonoverlapping, i.e., let $b = r2^s$ and $|a| < 2^s$. Then $\operatorname{msb}(a) < 2^s \leq \operatorname{lsb}(b)$. Now let $\operatorname{msb}(a) < \operatorname{lsb}(b)$ and let $2^s = \sigma = 2\operatorname{msb}(a)$. Then $b \in \sigma\mathbb{Z}$, i.e., $r = b/\sigma \in \mathbb{Z}$ and $|a| < 2^s$. $\square$

Expansions can be added, subtracted and multiplied exactly, they form an effective ring.[2] The addition algorithm, however, takes quadratic time. Shewchuk improves upon this for a more restrictive type of expansion, called strongly nonoverlapping expansion.

Definition 10. *(nonadjacent, strongly nonoverlapping)* [17]
Let $a, b \in \mathbb{F}$ with $|a| \leq |b|$ be nonoverlapping. Then a and b are adjacent*, if $2a$ overlaps b. Otherwise a and b are* nonadjacent*. An expansion is called nonadjacent, if its summands are pairwise nonadjacent.*
An expansion is strongly nonoverlapping *if each summand is adjacent to at most one other summand and each summand that is adjacent to some other summand is a power of two.*

Hence, in a strongly nonoverlapping expansion, adjacent summands come in pairs and both summands are a power of two. The definition of strongly nonoverlapping expansions may seem a bit complicated, but they admit a linear time addition algorithm. Furthermore, the multiplication algorithm for expansions also maintains the strongly nonoverlapping property. Hence strongly nonoverlapping expansion also form an effective ring. We use strongly nonoverlapping expansions in our number type `Real_algebraic` [7].
In an expansion, summands may carry only a few nonzero bits, in which case the expansion may have an unnecessary large number of summands. Shewchuk presents a compression algorithm to put more bits into each summand and reduce the number of summands, however without an explicit performance guarantee. Shewchuks work is based on work by Priest [11, 12] who considers a different notion of nonoverlappingness, we call *maximal nonoverlapping*.

Definition 11. *(maximal nonoverlapping)*
Let $a, b \in \mathbb{F}$ with $|a| \leq |b|$. Then a and b are maximal nonoverlapping *if $\operatorname{msb}(a) \leq \varepsilon_{\mathrm{m}} \operatorname{msb}(b)$. A sequence $f_1, f_2, \ldots, f_n \in \mathbb{F}$ is maximal nonoverlapping if the elements are pairwise maximal nonoverlapping and they are ordered by increasing magnitude, except that any of the f_i may be zero.*

Each maximal nonoverlapping sequence is an expansion since for $b \neq 0$

$$\operatorname{msb}(a) \leq \varepsilon_{\mathrm{m}} \operatorname{msb}(b) \leq \varepsilon_{\mathrm{m}} \tfrac{1}{2} \varepsilon_{\mathrm{m}}^{-1} \operatorname{lsb}(b) < \operatorname{lsb}(b) \tag{29}$$

and Lemma 9. Hence we can talk of maximal nonoverlapping expansions. In a maximal nonoverlapping expansion, the summands are optimally spaced. Priest presents arithmetic operations maintaining maximal nonoverlapping expansions without stray zero summands, particularly exact ring operations. An important substep is a renormalization procedure that restores the maximal nonoverlapping property.
Rump et al. [16] present an algorithm that transforms any sum of floating-point numbers into a maximal nonoverlapping expansion, either exactly, or approximately, with a relative error that depends on the number of output summands only. The quad double arithmetic of Hida et al. [3] maintains maximal nonoverlapping expansions too. Here the number of summands is fixed to four.
Our conversion algorithms accept general expansions as input, hence output from any of the mentioned algorithms may be used. An important step is our algorithm MONOTONIZE in Section 3, converting an expansion into a monotone expansion. This algorithm is quite similar to the renormalization procedure of Priest or the compression algorithm of Shewchuk.

Definition 12. *(monotone expansion)*
An expansion is called monotone *if all summands are positive or all summands are negative or it consists of one zero summand only.*

A monotone expansion is truly monotone in that the summands are strictly ordered, increasing if the summands are positive, decreasing otherwise. A monotone expansions $f_1, f_2, \ldots, f_n$ is very useful because

[2]Ignoring, of course, the problem of overflow and underflow.

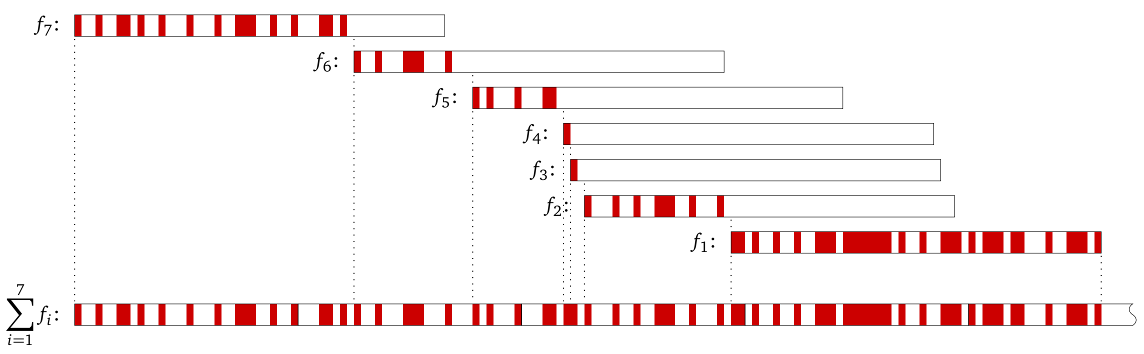

Figure 1: Copying the mantissae of a monotone expansion onto the mantissa of an arbitrary precision floating-point number.

it contains each nonzero bit of a binary representation of $F = \sum_{i=1}^{n} f_i$ explicitly. The bits only have to be copied to convert F into an arbitrary precision floating-point number, see Figure 1. Monotone, maximal nonoverlapping expansions are unique, for $x \in \mathbb{R}$ there exists at most one monotone, maximal nonoverlapping expansion $f_1, \dots, f_n$ with $x = \sum_{i=1}^{n} f_i$.

Lemma 13. *Let $e_1, \dots, e_m$ and $f_1, \dots, f_n$ be two monotone and maximal nonoverlapping expansions with $\sum_{i=1}^{m} e_i = \sum_{j=1}^{n} f_j$. Then $n = m$ and $e_i = f_i$ for $1 \le i \le n$.*

Proof. Clearly, e_m and f_n have the same sign. We can assume the sign is positive, the other case is symmetric. Let $s = \sum_{i=1}^{n} e_i$. It is $s \ge e_m$ and hence $\operatorname{msb}(s) \ge \operatorname{msb}(e_m)$. Furthermore

$$
\begin{aligned}
s &= \sum_{i=1}^{m} e_i && (30)\\
&\le \sum_{i=1}^{m} 2 \operatorname{msb}(e_i)(1 - \varepsilon_{\mathtt{m}}) && (31)\\
&\le \sum_{i=1}^{m} 2 \operatorname{msb}(e_m)\varepsilon_{\mathtt{m}}^{m-i}(1 - \varepsilon_{\mathtt{m}}) && (32)\\
&= 2 \operatorname{msb}(e_m) \left[\sum_{i=0}^{m-1} \varepsilon_{\mathtt{m}}^{i} - \sum_{i=1}^{m} \varepsilon_{\mathtt{m}}^{i} \right] && (33)\\
&= 2 \operatorname{msb}(e_m)(1 - \varepsilon_{\mathtt{m}}^{m}) && (34)
\end{aligned}
$$

and therefore $\operatorname{msb}(s) \le \operatorname{msb}(e_m)$. The same reasoning applies to f_n and therefore $\operatorname{msb}(e_m) = \operatorname{msb}(f_n)$. Let $\sigma = 2\varepsilon_{\mathtt{m}} \operatorname{msb}(f_n)$, then $e_m - f_n \in \sigma\mathbb{Z}$. Assume $e_m > f_n$, then

$$
\sigma \le e_m - f_n = \sum_{j=1}^{n-1} f_j - \sum_{i=1}^{m-1} e_i \le \sum_{j=1}^{n-1} f_j < 2 \operatorname{msb}(f_{n-1}) \le 2\varepsilon_{\mathtt{m}} \operatorname{msb}(f_n), \tag{35}
$$

which is a contradiction. The case $e_m < f_n$ follows analogously. Therefore $e_m = f_n$. The claim follows by induction. □

2.2 Hardware Representation of `binary64`

Our implementation uses the C++ type `double` for floating-point arithmetic. On contemporary architectures, the `double` type is an IEEE 754-2008 number in `binary64` format. For conversion to arbitrary precision floating-point numbers we need direct access to the sign, the exponent and the mantissa of a `double`. The IEEE 754-2008 standard fixes the storage format, which enables platform independent access. Code similar to the one below is present in other projects, for example the MPFR library. Using the `union` keyword, we interpret a `double` d as 64 bit unsigned integer l. The code depends on `unsigned long long` having 64 bits and having the same endianness as `double`.

⟨*direct access to double representation*⟩≡
```
union ieee_binary64{
  double d;
  unsigned long long l;

  ⟨access individual double parts⟩
};
```

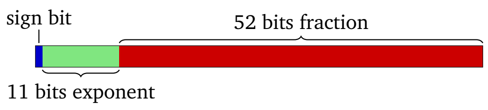

Figure 2: The IEEE 754-2008 `binary64` format.

The placement of the individual data is shown in Figure 2. A `double` stores one bit for the sign, 11 bits for the exponent and 52 bits for a part of the mantissa called fraction. The following methods directly return this data as integers. Denote these integers by s_b, e_b and f respectively.

⟨*access individual double parts*⟩≡
```
inline long sign_b(){
  return (l & 0x8000000000000000ULL) » 63;
}

inline long exponent_b(){
  return (l & 0x7FF0000000000000ULL) » 52;
}

inline unsigned long long fraction(){
  return (l & 0x000FFFFFFFFFFFFFULL);
}
```
A triple $(s_b, e_b, f) \in \{0,1\} \times \{0,\dots,2^{11}-1\} \times \{0,\dots,2^{52}-1\}$ represents the following `double` d. If $e_b = 0$ then d is zero or denormalized nonzero:

$$d = (-1)^{s_b} \cdot f \cdot 2^{-1074} \tag{36}$$

If $0 < e_b < 2^{11} - 1$ then d is a normalized number:

$$d = (-1)^{s_b} \cdot (2^{52} + f) \cdot 2^{e_b - 1075} \tag{37}$$

If $e_b = 2^{11} - 1$ then $d \notin \mathbb{R}$:

$$f = 0 \quad \Rightarrow \quad d = (-1)^{s_b} \cdot \infty \tag{38}$$
$$f \neq 0 \quad \Rightarrow \quad d = \mathtt{nan} \tag{39}$$

Access to these raw numbers is however not sufficient for our needs. Assuming $d \notin \{0, \mathtt{nan}, \pm\infty\}$ we need integral sign s, mantissa m and exponent e such that

$$d = s \cdot m \cdot 2^e. \tag{40}$$

For efficiency reasons, we do not check whether $d \in \{0, \mathtt{nan}, \pm\infty\}$. For these cases the following code will not return a correct or desired result. Omitting these cases, allows to perform all computations branch free. The difficulty that remains, is to handle normalized and denormalized numbers uniformly.

By casting e_b to bool and back we map nonzero e_b to one and zero e_b to zero and can then compute the remaining quantities with simple arithmetic.

⟨*access individual double parts*⟩+≡
```
  inline bool normalized(){
    return static_cast<bool>(l & 0x7FF0000000000000ULL);
  }
```
The following functions return s, e, m satisfying Equation (40).

⟨*access individual double parts*⟩+≡
```
  inline long sign(){
    return 1-2*sign_b();
  }

  inline long exponent(){
    const long norm = static_cast<long>(normalized());
    return exponent_b() - 1074 - norm ;
  }

  inline unsigned long long mantissa(){
    const unsigned long long norm =
      static_cast<unsigned long long>(normalized());
    return ( (norm « 52) + fraction() );
  }
```

3 Converting Expansions to Monotone Expansion

We are now ready to present our algorithm MONOTONIZE to convert an expansion into a monotone maximal nonoverlapping expansion. MONOTONIZE is similar to the first stage of Shewchuks compression algorithm. In Shewchuks compression, starting with the most significant summand, the summands are added to a running total x, using FASTTWOSUM. Once the error term y is nonzero, x is stored away and y becomes the new running total x'. At this point, x and x' are maximal nonoverlapping. Since more summands are added to x', it may however later overlap x. The first stage continues until all summands have been processed. Then the second stage of compression removes any overlap. After both stages, stored summands may have different signs.

The difference of MONOTONIZE is, that it handles the case of a roundoff error with wrong sign differently. We describe MONOTONIZE first for an expansion $e_1, e_2, \ldots, e_n$ with $e_n > 0$, the case $e_n < 0$ is handled analogously. Since the summands are nonoverlapping, it is $E = \sum_{i=1}^{n} e_i > 0$ too. A flowchart for MONOTONIZE for the case $e_n > 0$ is given in Figure 3.

We compute the bits of E starting with the more significant bits and summands. Using FASTTWOSUM we add the summands to a running total x until the error term y becomes nonzero. Adding a less significant, nonoverlapping summand e_i to x is much like incrementing or decrementing x. No bit of x more significant that $\operatorname{lsb}(x)$ will be altered. Hence, once $y \neq 0$, the remaining summands $e_i, e_{i-1}, \ldots$ can not alter the bits in x but only those in y.

If y is positive, this means x now stores the leading bits of E. We store x into an output summand f_j and continue with y as new running total. If y is negative, then x is too large. We can however store $\operatorname{pred}(x)$ as output summand, since the error term $w = x - \operatorname{pred}(x)$ is positive and protects the bits in $\operatorname{pred}(x)$ from being changed. Note that w is a floating-point number. With FASTTWOSUM(w, y) we compute a new running total and error term and continue as before.

Let $f_1, \ldots, f_k$ be the sequence of output summands. Several properties must be shown. First of course, that $\sum_{i=1}^{n} e_i = \sum_{j=1}^{k} f_j$ and also that $f_j > 0$ for $1 \leq j \leq k$. The most interesting claim is however that the f_j are maximal nonoverlapping. This reduces to the claim that, when $y \neq 0$ occurs, x does not overlap already stored summands. We show below that if overlap occurs, then in at most one bit, x is exactly this

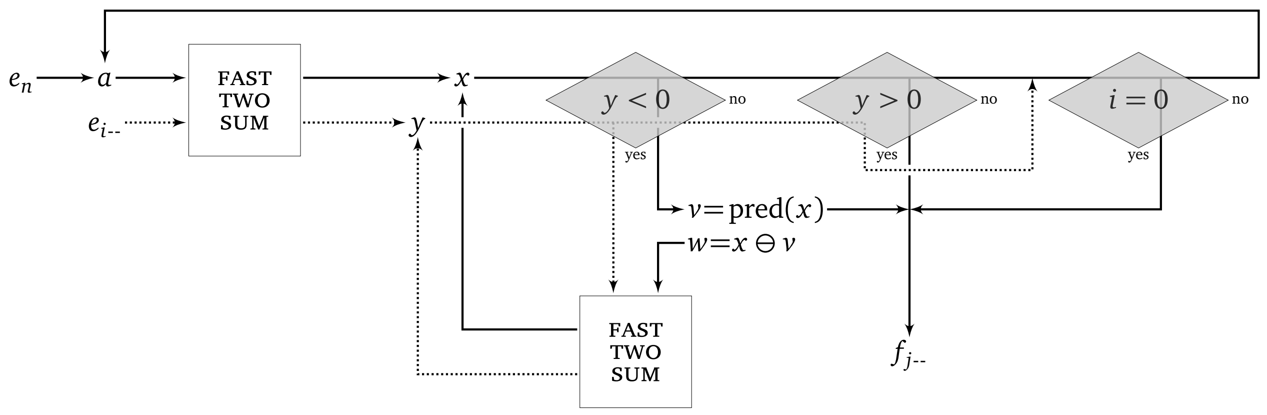

Figure 3: Flowchart for MONOTONIZE, applied to an expansion $e_1, e_2, \ldots, e_n$ with $e_n > 0$. The output is a monotone, maximal nonoverlapping expansion $f_1, f_2, \ldots, f_k$.

bit, and y is negative. Hence, by passing on to $\mathrm{pred}(x)$, the potential overlap is removed. We formally prove these claims in Lemma 17 below. To this end we need to analyze the operations in MONOTONIZE more closely. Our first lemma corresponds to the FASTTWOSUM operation on the left in Figure 3.

Lemma 14. *Let $a, e_i \in \mathbb{F}$ be nonoverlapping with $a > |e_i|$. Furthermore, let $x, y \in \mathbb{F}$ with $x = a \oplus e_i$ and $x + y = a + e_i$. Then $x > 0$ and either*

$$\mathrm{msb}(x) \le \mathrm{msb}(a) \qquad \text{or} \qquad x = 2\,\mathrm{msb}(a),\ y < 0. \tag{41}$$

Proof. Since $a + e_i > 0$ also $x > 0$ by Equation (23). Since a and e_i are nonoverlapping, it is $2\,\mathrm{msb}(a) > a + e_i$ and since rounding never skips over a floating-point number also $2\,\mathrm{msb}(a) \ge x$. (If $2\,\mathrm{msb}(a) \notin \mathbb{F}$, then actually $2\,\mathrm{msb}(a) > x$, since $x \in \mathbb{F}$.) If $2\,\mathrm{msb}(a) > x$, then $\mathrm{msb}(a) \ge \mathrm{msb}(x)$. Otherwise $2\,\mathrm{msb}(a) = x$, in this case $x > a + e_i$ and hence $y < 0$. □

Assuming a does not overlap the last stored summand f_j, we see that x overlaps f_j in at most one bit and in this case x is a power of two and $y < 0$. Both x and y may also be recomputed by the other FASTTWOSUM operation at the bottom of Figure 3. We show a similar result for this FASTTWOSUM operation in Lemma 16. But first we collect some properties of v and $w = x \ominus v$ that are computed prior to it.

Lemma 15. *Let $x, y \in \mathbb{F}$ with $x = x \oplus y$ and $x > 0, y < 0$. Then $x > \frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta$. Let furthermore $v = \mathrm{pred}(x)$ and $w = x \ominus v$. Then $v > 0$, $\mathrm{msb}(w) = w = x - v$ and $w > -y$.*

Proof. Since y is the rounding error in $x \oplus y$, x and y are maximal nonoverlapping. Assume $x \le \frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta$, then $\mathrm{msb}(x+y) \le \frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta$ and hence $x \oplus y = x + y$ by Lemma 4. This is a contradiction to $y \ne 0$. Since $x > \frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta$, also $v \ge \frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta > 0$. Since x is the successor of v, it is $w = x - v = 2\varepsilon_{\mathrm{m}}\,\mathrm{msb}(v) \in \mathbb{F}$ by Equation (9). We have $v < x + y < x$ and since we round to nearest, it follows $w \ge -2y > -y$. □

Note that w and v are not maximal nonoverlapping but overlap in one bit. We next look what happens when we compute a new running total x and error term y from w and the old error term y'.

Lemma 16. *Let $w, y' \in \mathbb{F}$ with $\mathrm{msb}(w) = w > -y' > 0$. Furthermore let $x, y \in \mathbb{F}$ with $x = w \oplus y'$ and $x + y = w + y'$. Then $x > 0$ and either*

$$\mathrm{msb}(x) < \mathrm{msb}(w) \qquad \text{or} \qquad x = w,\ y = y'. \tag{42}$$

Proof. Since $w + y' > 0$ also $x > 0$ by Equation (23). It is $\mathrm{msb}(w) > w + y'$, therefore $\mathrm{msb}(w) \ge x$, since $\mathrm{msb}(w) \in \mathbb{F}$. If $\mathrm{msb}(w) > x$, then $\mathrm{msb}(w) > \mathrm{msb}(x)$. If otherwise $w = x$, then $y = y'$. □

Thus, either x and the last output summand v are maximally nonoverlapping, or we are again in the situation where x is a power of two, the overlap is exactly one bit and $y < 0$. We are now ready to combine these results into our main technical lemma summarizing properties of MONOTONIZE for $e_n > 0$.

Lemma 17. *Call* MONOTONIZE *with an expansion $e_1, e_2, \dots, e_n$ with $e_n > 0$ and assume that no overflow occurs. Let the output be the sequence of summands $f_1, f_2, \dots, f_k$. After any of the two* FASTTWOSUM *operations, we have a sequence of summands*

$$s = e_1,\ e_2,\ \dots,\ e_i,\ y,\ x,\ f_j,\ f_{j+1},\ \dots,\ f_k. \tag{43}$$

Then

$$y + x + \sum_{l=j}^{k} f_l = \sum_{l=i+1}^{n} e_l \tag{$=_\Sigma$}$$

and

$$\begin{aligned} x, f_l &> 0 && \text{for } j \le l \le k && (\bigstar)\\ \operatorname{msb}(f_{l-1}) &\le \varepsilon_{\mathrm{m}} \operatorname{msb}(f_l) && \text{for } j < l \le k && (\blacktriangle)\\ \operatorname{msb}(y) &\le \varepsilon_{\mathrm{m}} \operatorname{msb}(x). &&&& (\triangle) \end{aligned}$$

Furthermore, either

$$\operatorname{msb}(x) \le \varepsilon_{\mathrm{m}} \operatorname{msb}(f_j) \qquad \text{or} \qquad x = 2\varepsilon_{\mathrm{m}} \operatorname{msb}(f_j),\ y < 0. \tag{✠}$$

Thus, MONOTONIZE iteratively transforms the sequence of summands into a new sequence with the same sum. Along the way, the subsequence $e_1, \dots, e_i, y, x$ stays nonoverlapping, while the subsequence $y, x, f_j, \dots, f_k$ is maximal nonoverlapping, with the only exception, that x may overlap the last bit of f_j. Furthermore the subsequence $x, f_j, \dots, f_k$ is monotone.

Proof. First we note that ($\triangle$) follows from Theorem 7 by $\operatorname{msb}(y) \le |y| \le \varepsilon_{\mathrm{m}} \operatorname{msb}(x)$, since x and y are always the results of a FASTTWOSUM operation. It remains however to be shown that Theorem 7 can be applied, i.e., that the input numbers to FASTTWOSUM were ordered by absolute value.
For ($=_\Sigma$), ($\bigstar$), ($\blacktriangle$) and (✠) we proceed by induction. The base case occurs at the start of MONOTONIZE. The sequence

$$s' = e_1, \dots, e_n \qquad \text{is turned into} \qquad s = e_1, \dots, e_{n-2}, y, x \tag{44}$$

by

$$(x, y) = \text{FASTTWOSUM}(e_n, e_{n-1}). \tag{45}$$

The summands $e_n > 0$ and e_{n-1} are nonoverlapping and hence we can apply Theorem 7 and Lemma 14. Equations ($=_\Sigma$) and ($\bigstar$) follow directly and for ($\blacktriangle$) and (✠) there is nothing to show.

For the induction step, let y' be the error term in the sequence, before the FASTTWOSUM operation to be considered. We distinguish three cases, $y' < 0$, $y' > 0$ and $y' = 0$, corresponding to different computation paths in Figure 3. In case $y' < 0$, the sequence

$$s' = \dots, e_i, y', x', f_{j+1}, \dots \qquad \text{is turned into} \qquad s = \dots, e_i, y, x, f_j, f_{j+1}, \dots \tag{46}$$

by

$$f_j = \operatorname{pred}(x') \tag{47}$$
$$w = x' \ominus \operatorname{pred}(x') \tag{48}$$
$$(x, y) = \text{FASTTWOSUM}(w, y'). \tag{49}$$

By induction hypothesis, for s' we get,

$$y' + x' + \sum_{l=j+1}^{k} f_l = \sum_{l=i+1}^{n} e_l \tag{$=_\Sigma'$}$$

$$\begin{aligned} x', f_l, &> 0 && \text{for } j+1 \le l \le k && (\bigstar') \\ \operatorname{msb}(f_{l-1}) &\le \varepsilon_{\mathrm{m}} \operatorname{msb}(f_l) && \text{for } j+1 < l \le k && (\blacktriangle') \\ \operatorname{msb}(y') &\le \varepsilon_{\mathrm{m}} \operatorname{msb}(x'). && && (\triangle') \end{aligned}$$

Furthermore by induction hypothesis, either

$$\operatorname{msb}(x') \le \varepsilon_{\mathrm{m}} \operatorname{msb}(f_{j+1}) \qquad \text{or} \qquad x' = 2\varepsilon_{\mathrm{m}} \operatorname{msb}(f_{j+1}),\ y' < 0. \tag{✠'}$$

We can apply Lemma 15 to x' and y'. Hence $f_j > 0$, which together with ($\bigstar'$) shows ($\bigstar$). Furthermore $w = x' - \operatorname{pred}(x')$ and $\operatorname{msb}(w) = w > -y'$. Thus we can apply Theorem 7 and Lemma 16 to w and y'. It follows that

$$y' + x' = y' + w + f_j = y + x + f_j \tag{50}$$

and together with ($=_\Sigma'$) follows ($=_\Sigma$). Since $f_j < x'$ we can deduct from (✠') that $\operatorname{msb}(f_j) \le \varepsilon_{\mathrm{m}} \operatorname{msb}(f_{j+1})$ which together with ($\blacktriangle'$) yields ($\blacktriangle$). Since w is the distance from f_j to the next larger floating-point number, we have $w = 2\varepsilon_{\mathrm{m}} \operatorname{msb}(f_j)$ by Equation (9). Following Lemma 16 we either have $\operatorname{msb}(x) < \operatorname{msb}(w) = w = 2\varepsilon_{\mathrm{m}} \operatorname{msb}(f_j)$, i.e., $\operatorname{msb}(x) \le \varepsilon_{\mathrm{m}} \operatorname{msb}(f_j)$ or $x = w = 2\varepsilon_{\mathrm{m}} \operatorname{msb}(f_j)$, $y = y' < 0$. This shows (✠) and concludes this case.

In case $y' \ge 0$, the FASTTWOSUM operation on the left in Figure 3 is used. We next show that e_i does not overlap a and hence Theorem 7 and Lemma 14 can be applied. Let $\sigma = 2^k, k \in \mathbb{Z}$ be maximal such that $e_{i+1}, \dots, e_n \in \sigma\mathbb{Z}$. We need to show that no nonzero bit smaller than σ is created before e_i is used. MONOTONIZE manipulates summands with addition and subtraction, which are safe by Equation (21) and by computing the predecessor $v = \operatorname{pred}(x)$. While v may have smaller nonzero bits than x, the error term y contains already smaller bits, i.e., $\operatorname{lsb}(v) \ge \operatorname{lsb}(y)$ and no nonzero bit smaller than already present is created. Hence $a \in \sigma\mathbb{Z}$ and a does not overlap e_i.

We return to the induction. In case $y' > 0$, the sequence

$$s' = \dots, e_i, e_{i+1}, y', x', f_{j+1}, \dots \qquad \text{is turned into} \qquad s = \dots, e_i, y, x, f_j, f_{j+1}, \dots \tag{51}$$

by

$$f_j = x' \tag{52}$$

$$(x, y) = \text{FASTTWOSUM}(y', e_{i+1}). \tag{53}$$

By induction hypothesis, for s' we get,

$$y' + x' + \sum_{l=j+1}^{k} f_l = \sum_{l=i+2}^{n} e_l \tag{$=_\Sigma'$}$$

$$\begin{aligned} x', f_l &> 0 && \text{for } j+1 \le l \le k && (\bigstar') \\ \operatorname{msb}(f_{l-1}) &\le \varepsilon_{\mathrm{m}} \operatorname{msb}(f_l) && \text{for } j+1 < l \le k && (\blacktriangle') \\ \operatorname{msb}(x') &\le \varepsilon_{\mathrm{m}} \operatorname{msb}(f_{j+1}) && \text{since } y' > 0 && (✠') \\ \operatorname{msb}(y') &\le \varepsilon_{\mathrm{m}} \operatorname{msb}(x'). && && (\triangle') \end{aligned}$$

From Theorem 7 follows that

$$e_{i+1} + y' + x' = y + x + f_j \tag{54}$$

and together with $(=_\Sigma')$ follows $(=_\Sigma)$. By Lemma 14, $x > 0$ therefore together with $(\bigstar')$ we have $(\bigstar)$. Furthermore $(\blacktriangle)$ follows from $(\blacktriangle')$ and $(✠')$. Again by Lemma 14 (with $a = y'$), either $\operatorname{msb}(x) \le \operatorname{msb}(y')$ or $x = 2\operatorname{msb}(y')$, $y < 0$ and by $(\triangle')$ $\operatorname{msb}(y') \le \varepsilon_{\mathrm{m}} \operatorname{msb}(f_j)$. If $x = 2\operatorname{msb}(y')$ and $\operatorname{msb}(y') = \varepsilon_{\mathrm{m}} \operatorname{msb}(f_j)$ then $x = 2\varepsilon_{\mathrm{m}} \operatorname{msb}(f_j)$, $y < 0$ otherwise $\operatorname{msb}(x) \le \varepsilon_{\mathrm{m}} \operatorname{msb}(f_j)$, i.e., $(✠)$ holds.

In case $y' = 0$, the sequence

$$s' = \ldots, e_i, e_{i+1}, y' = 0, x', f_j, \ldots \qquad \text{is turned into} \qquad s = \ldots, e_i, y, x, f_j, \ldots \tag{55}$$

by

$$(x, y) = \textsc{FastTwoSum}(x', e_{i+1}). \tag{56}$$

By induction hypothesis, for s' we get,

$$x' + \sum_{l=j}^{k} f_l = \sum_{l=i+2}^{n} e_l \tag*{$(=_\Sigma')$}$$

$$\begin{aligned} x', f_l &> 0 && \text{for } j \le l \le k && (\bigstar') \\ \operatorname{msb}(f_{l-1}) &\le \varepsilon_{\mathrm{m}} \operatorname{msb}(f_l) && \text{for } j < l \le k && (\blacktriangle') \\ \operatorname{msb}(x') &\le \varepsilon_{\mathrm{m}} \operatorname{msb}(f_j) && \text{since } y' = 0. && (✠') \end{aligned}$$

For $(\blacktriangle)$ there nothing to show, it is identical to $(\blacktriangle')$. From Theorem 7 follows that

$$e_{i+1} + x' = y + x \tag{57}$$

and together with $(=_\Sigma')$ we have $(=_\Sigma)$. By Lemma 14, $x > 0$, and together with $(\bigstar')$ we have $(\bigstar)$. Again by Lemma 14 (with $a = x'$), either $\operatorname{msb}(x) \le \operatorname{msb}(x')$ or $x = 2\operatorname{msb}(x')$, $y < 0$ and by $(✠')$ $\operatorname{msb}(x') \le \varepsilon_{\mathrm{m}} \operatorname{msb}(f_j)$. If $x = 2\operatorname{msb}(x')$ and $\operatorname{msb}(x') = \varepsilon_{\mathrm{m}} \operatorname{msb}(f_j)$ then $x = 2\varepsilon_{\mathrm{m}} \operatorname{msb}(f_j)$, $y < 0$ otherwise $\operatorname{msb}(x) \le \varepsilon_{\mathrm{m}} \operatorname{msb}(f_j)$, i.e., $(✠)$ holds. ☐

We still need to extend MONOTONIZE to expansions $e_1, \ldots, e_n$ with $e_n < 0$ and $e_n = 0$. The first case is handled analogously to the case $e_n > 0$ and the second case is handled by skipping over summands until a nonzero summand is found. If only zero summands are present, a single zero summand is returned. Below an implementation of MONOTONIZE in C++ and using floating-point numbers in the `binary64` format is given. It is necessary to provide a buffer h, large enough for the output expansion. By Theorem 21 below, for `binary64` we have $k \le 40$. The computation of $\operatorname{pred}(x)$ (and $\operatorname{succ}(x)$ in case $e_n < 0$) is done by the function call `nexttozero(x)`. In Section 3.1 we present two alternative implementations of `nexttozero()` that may be used.

Algorithm 18. *Let $e_0, e_1, \ldots, e_{n-1}$ be an expansion. Then* `de_monotonize()` *computes an expansion with k summands, stored in $h_{l-k}, h_{l-k+1}, \ldots, h_{l-1}$ and returns k.*

⟨*monotonize an expansion*⟩≡
```
inline int de_monotonize(const int n, const double *const e,
                         const int l, double *const h){

  ⟨check summands are finite⟩

  int j=l-1;

  int i=n-1;
  while(i>0 && e[i] == 0.0) i-;
```

```
    double a = e[i-];
    const double s = a > 0.0 ? 1.0 : -1.0;

    while(i >= 0){
      double x,y;
      de_fast_two_sum(a,e[i-],x,y);

      while(s*y < 0.0){
        const double v = nexttozero(x);
        const double w = x - v;
        h[j-] = v;
        de_fast_two_sum(w,y,x,y);
      }

      if(s*y > 0.0){
        h[j-] = x;
        a = y;
      }else{
        a = x;
      }
    }

    assert(s*a > 0.0 || j==l-1);
    h[j] = a;
    assert(j >= 0);

    return l-j;
  }
```

We next give a criterion, when no overflow occurs in MONOTONIZE and all computations are therefore correct. Note that the number $\tau(2-\varepsilon_\mathrm{m})$ is the smallest number that may be rounded to $+\infty$, it is halfway between $2\tau(1-\varepsilon_\mathrm{m})$ and $\mathrm{succ}(2\tau(1-\varepsilon_\mathrm{m})) = 2\tau$, where the successor is taken in some floating-point set $\mathbb{F}'$ with $\tau' > \tau$.

Lemma 19. *Call* MONOTONIZE *with an expansion $e_1, e_2, \dots, e_n$ with $\sum_{i=1}^n |e_i| < \tau(2-\varepsilon_\mathrm{m})$, then no overflow occurs.*

Proof. We only have to show $a \oplus b \in \mathbb{F}$ for all a, b that we call FASTTWOSUM(a, b) for, cf. Theorem 7. Assume $e_n > 0$. At the start of MONOTONIZE we add $e_n, e_{n-1}, e_{n-2}, \dots$ to our running total x until the exact sum is not a floating-point number for first time. Let l be the index where this happens, then

$$x = \mathbf{fl}\left(\sum_{i=l}^{n} e_i\right). \tag{58}$$

Since

$$\left|\sum_{i=l}^{n} e_i\right| \le \sum_{i=1}^{n} |e_i| < \tau(2-\varepsilon_\mathrm{m}) \tag{59}$$

we round to a real number, i.e., $x \in \mathbb{F}$. We store either x or $\mathrm{pred}(x)$ as first output summand f_k and therefore $f_k \in \mathbb{F}$. All further computations involve numbers smaller than f_k by Lemma 17 and hence no overflow can occur. □

This criterion is useful because it allows us to show that for strongly nonoverlapping expansions MONOTONIZE will always work correctly.

Lemma 20. *Let $e_1, e_2, \dots, e_n$, be a strongly nonoverlapping expansion, then $\sum_{i=1}^n |e_i| < \tau(2-\varepsilon_\mathrm{m})$.*

Proof. The sequence $|e_1|, |e_2|, \ldots |e_n|$ is a strongly nonoverlapping expansion too. Consider the binary representation of $E = \sum_{i=1}^{n} |e_i|$. We have $\operatorname{msb}(E) \le \tau$ and E contains a zero bit at least every $p+1$ bits. Hence $E < \tau(2 - \varepsilon_{\mathrm{m}})$. □

Finally we can state our main result.

Theorem 21. *Given an expansion $e_1, e_2, \ldots, e_n$ with $\sum_{i=1}^{n} |e_i| < \tau(2-\varepsilon_{\mathrm{m}})$,* MONOTONIZE *computes a monotone, maximal nonoverlapping expansion $f_1, f_2, \ldots, f_k$ with $\sum_{i=1}^{n} e_i = \sum_{i=1}^{k} f_i$ and $k \le \lceil \log_2(2\tau/\eta)/p \rceil$ summands.* MONOTONIZE *takes $O(n+k)$ steps.*

The bound on k is tight, running MONOTONIZE with $-\eta, \tau$ as input expansion will create an output expansion with $\lceil \log_2(2\tau/\eta)/p \rceil$ summands. This example furthermore shows that $k > n$ is possible.

Proof. MONOTONIZE first skips zero summands. If there are only zero summands, a single zero is returned which is a monotone, maximal nonoverlapping expansion. Otherwise it continues computing with a leading nonzero summand and Lemma 17 can be applied. After the last FASTTWOSUM operation we have $y \ge 0$ since otherwise MONOTONIZE would perform further FASTTWOSUM operations. Therefore at this point we either have a sequence

$$y = 0,\ x,\ f_2,\ f_3,\ \ldots,\ f_k \qquad \text{with} \qquad \operatorname{msb}(x) \le \varepsilon_{\mathrm{m}} \operatorname{msb}(f_2) \tag{60}$$

and the computation ends with $f_1 = x$ or we have a sequence

$$y > 0,\ x,\ f_3,\ f_4,\ \ldots,\ f_k \qquad \text{with} \qquad \operatorname{msb}(x) \le \varepsilon_{\mathrm{m}} \operatorname{msb}(f_3) \tag{61}$$

and the computation ends with $f_2 = x$, $f_1 = y$. Therefore by Lemma 17

$$\sum_{i=1}^{k} f_i = \sum_{i=1}^{n} e_i \tag{62}$$

and

$$\operatorname{msb}(f_{l-1}) \le \varepsilon_{\mathrm{m}} \operatorname{msb}(f_l) \qquad \text{for } 1 < l \le k \tag{63}$$

and all summands have the same sign. Output summands are optimally spaced, i.e., their leading bits are at least p bits apart. Hence there can be at most $\lceil \log_2(2\tau/\eta)/p \rceil$ summands. Each iteration consumes an input summand or creates an output summand, therefore the number of iterations is bounded by $O(n+k)$. □

3.1 Computing the Next Floating-Point Number Towards Zero

Let $x \in \mathbb{F}$. For our implementation of MONOTONIZE, we need a subroutine `nexttozero(x)` that computes $\operatorname{pred}(x)$ if $x > 0$ and $\operatorname{succ}(x)$ if $x < 0$. By Lemma 15 $|x| > \frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta$, i.e., x is normalized in this case. We present two alternative implementations of `nexttozero()`. The first method is a slight modification of an algorithm by Rump et al. [14] to simultaneously compute predecessor and successor of a floating-point number. We removed the part handling denormalized numbers and we replaced $|c|$ by c in the computation of e, to always compute the next number in direction towards zero instead of the predecessor. This gives the desired result by symmetry of $\mathbb{F}$ and $\mathbf{fl}$.

Algorithm 22. *Let $c \in \mathbb{F}$ with $|c| > \frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta$. If $c > 0$, then* `nexttozero(c)` *returns* $\operatorname{pred}(c)$*, otherwise* $\operatorname{succ}(c)$ *is returned.*

⟨*next to zero*⟩≡
```
inline double nexttozero(const double c){

  const double eps = std::numeric_limits<double>::epsilon();
  const double u = 0.5*eps;
```

```
    const double invu = 9007199254740992.0;
    assert(invu == ldexp(1.0,53));
    const double phi = u*(1+eps);
    assert(phi == std::ldexp(1.0,-53)+ldexp(1.0,-105));
    const double nearsubn = invu*std::numeric_limits<double>::min();

    double cinf;
    if(std::fabs(c) > nearsubn){
      double e = phi*c;
      cinf = c - e;
    }else{
      double C = invu*c;
      double e = phi*C;
      cinf = (C-e)*u;
    }

    assert(cinf == nextafter(c,0));
    return cinf;
  }
```

Since our modifications are minimal, we refer to [14] for a proof of correctness. The second method computes predecessor or successor using bit access. The representation of IEEE 754-2008 numbers is such that the floating-point number next to x in direction towards zero is simply the next smaller number when interpreting x as unsigned integer, see Section 2.2.

Algorithm 23. *Let* $0 \neq x \in \mathbb{F}$*. If* $x > 0$ *then* `nexttozero(x)` *returns* $\mathrm{pred}(x)$*, if* $x < 0$*, then* $\mathrm{succ}(x)$ *is returned.*

⟨*basic next to zero*⟩≡
```
  inline double nexttozero(const double x){
    ieee_binary64 X;
    X.d = x;
    X.l-;
    assert(X.d == nextafter(x,0));
    return X.d;
  }
```

In preliminary experiments, both methods showed comparable performance when used in MONOTONIZE. Using the C99 `nextafter()` function was slightly but noticeably slower. The first method is certainly more elegant since it uses basic floating-point operations only, while the other approach relies on moving a floating-point number into an integer register and bit manipulation. Nevertheless, it avoids branches and may be faster inside an algorithm that is also branch free.

4 Converting Expansions to MPFR

In this section we present algorithms and code to convert expansions into arbitrary precision floating-point numbers. While our algorithms convert to the MPFR number type, they can easily be carried over to other arbitrary precision floating-point types. For example in Appendix A we provide the same functionality for the `leda::bigfloat` number type.
We want to set an MPFR number x to some real value represented by a monotone expansion. An MPFR number consists of the following four fields (excerpt from `mpfr.h`).

⟨*mpfr.h*⟩≡
```
  typedef struct {
    mpfr_prec_t  _mpfr_prec;
    mpfr_sign_t  _mpfr_sign;
    mpfr_exp_t   _mpfr_exp;
    mp_limb_t   *_mpfr_d;
  } __mpfr_struct;
```

For $0 \neq x \in R$ the fields have the following semantics (again from `mpfr.h`).

⟨*mpfr.h*⟩+≡

```
  /*
     The represented number is
        _sign*(_d[k-1]/B+_d[k-2]/B^2+...+_d[0]/B^k)*2^_exp
     where k=ceil(_mp_prec/GMP_NUMB_BITS) and B=2^GMP_NUMB_BITS.

     For the msb (most significant bit) normalized representation, we must have
        _d[k-1]>=B/2, unless the number is singular.

     We must also have the last k*GMP_NUMB_BITS-_prec bits set to zero.
  */
```

Thus, `_mpfr_prec` stores the precision or number of bits of the mantissa. The mantissa is stored in `_mpfr_d`, in pieces of `GMP_NUMB_BITS` consecutive bits each. Such a piece is called a limb and limbs at a higher index store the more significant bits of the mantissa. The most significant bit in the most significant limb must be nonzero and has value $\frac{1}{2}$. Hence, the mantissa is interpreted as a number in $[0.5, 1)$. The sign is stored in `_mpfr_sign` and the exponent in `_mpfr_exp`.

MPFR can also represent some special values. The cases $x \in \{0, \mathtt{nan}, \pm\infty\}$ correspond to `_mpfr_exp` being near the smallest number representable by `mpfr_exp_t`. To set a number to zero, we use an MPFR function call and we never set a number to `nan` or $\pm\infty$. Exponents arising in expansions are always in the safe range of `mpfr_exp_t`.

To set an MPFR number to some $0 \neq x \in \mathbb{R}$ we first set the precision by calling `mpfr_set_prec()`. This will allocate the array `_mpfr_d` with sufficient entries. Then we write `_mpfr_sign`, `_mpfr_exp` and the entries of `_mpfr_d`. Thus we only rely on our knowledge of the representation of nonzero, real numbers in MPFR.

4.1 Expansions of Length One

For warmup we discuss converting a single floating-point number d into an MPFR number. Similar code is available in MPFR, but we can omit the special cases $d \in \{\mathtt{nan}, \pm\infty\}$ and the possibility to reduce the precision by rounding. Our code can be inlined into our other functions.

Algorithm 24. *Let $d \in \mathbb{F}$, then* `mpfr_set_double()` *computes an* MPFR *number* `rop` *with* `rop` $= d$.

⟨*convert a double to mpfr*⟩≡

```
  inline void mpfr_set_double(mpfr_t rop,const double d){
    assert(ra_isfinite(d));
    ⟨double to mpfr⟩
  }
```

First we handle the case $d = 0$ using an MPFR function call. If $d \neq 0$ we set the precision of `rop` to 53 bits, allocating the mantissa.

⟨*double to mpfr*⟩≡

```
  if(d == 0.0){
    mpfr_set_ui(rop,0,GMP_RNDN);
    return;
  }
  mpfr_set_prec(rop,53);
```

Remember that `ieee_binary64` computes integral sign s, exponent e and mantissa m such that

$$d = s \cdot m \cdot 2^e. \tag{64}$$

Hence, if d is normalized, the most significant nonzero bit in the mantissa has value 2^{52}. The leading bit of an MPFR mantissa must be nonzero and has value $\frac{1}{2}$. Therefore, we have to adjust the exponent by 53. In case d is denormalized, we normalize it by multiplying with 2^{53}, to ensure that the most significant bit is nonzero. In this case, the necessary exponent corrections cancel.

⟨*double to mpfr*⟩+≡
```
ieee_binary64 X;
X.d = d;
rop->_mpfr_sign = X.sign();

if(X.normalized()){
  rop->_mpfr_exp  = X.exponent()+53;
}else{
  double p = 9007199254740992.0;
  assert(p == ldexp(1.0,53));
  X.d *= p;
  assert(X.normalized());
  rop->_mpfr_exp  = X.exponent(); //-53+53;
}
```
Since d in `X` is now normalized, the most significant nonzero bit in the mantissa is the 53rd bit. Hence, in case of 64 bit limbs we have to shift 11 bits to the left. For 32 bit limbs, the mantissa overlaps two limbs and different shifting is needed, cf. m' in Figure 4.

⟨*double to mpfr*⟩+≡
```
#if GMP_NUMB_BITS==32
  rop->_mpfr_d[1] = (X.mantissa() >> 21);
  rop->_mpfr_d[0] = (X.mantissa() << 11);
#else //GMP_NUMB_BITS==64
  rop->_mpfr_d[0] = (X.mantissa() << 11);
#endif
```

4.2 Monotone Expansions

Recall that in a monotone expansion all summands have the same sign. No zero summands are allowed unless there is only one summand. Since the summands are also nonoverlapping, converting into an arbitrary precision floating-point number becomes as simple as aligning the mantissae of the summands by exponent and copying them, as already illustrated in Figure 1. This is done in the following algorithm.

Algorithm 25. *Given a monotone expansion* $e_0, e_1, \ldots, e_{n-1}$, `mpfr_set_monotone_expansion()` *computes an* MPFR *number* `rop` *with* `rop` $= \sum_{i=0}^{n-1} e_i$.

⟨*convert a monotone expansion to mpfr*⟩≡
```
inline void mpfr_set_monotone_expansion(mpfr_t rop,
                                        const int n ,const double *const e){
  ⟨check summands are finite⟩
  ⟨handle less than two summands⟩
  ⟨check summands are nonzero⟩
  ⟨set up precision, sign and exponent⟩
  ⟨clear mantissa and write first summand⟩
  ⟨write remaining summands⟩
}
```

Here we have some debug code to check that all input summands represent real numbers and are nonzero.

⟨*check summands are finite*⟩≡
```
#ifndef NDEBUG
  for(int i=0;i<n;++i) assert(ra_isfinite(e[i]));
#endif
```

⟨*check summands are nonzero*⟩≡
```
#ifndef NDEBUG
  for(int i=0;i<n;++i) assert(e[i] != 0.0);
#endif
```

First we handle the case of less than two summands. If the expansion has at least two summands, all summands are nonzero and we can use `ieee_binary64` safely.

⟨*handle less than two summands*⟩≡
```
if(n == 0){
  mpfr_set_ui(rop,0,GMP_RNDN);
  return;
}else if(n == 1){
  mpfr_set_double(rop,e[0]);
  return;
}
```

The sign and exponent of `rop` are determined by the leading summand of the expansion and are computed as above in Section 4.1. The exponent `maxexp` of `rop` can be interpreted as pointing directly in front of the most significant nonzero bit of the expansion. The exponent `minexp` of the last summand points somewhere behind the least significant nonzero bit, so it is sufficient to set the precision to $\texttt{maxexp} - \texttt{minexp}$. In case the leading summand is not normalized, this value may be smaller than 53 and even as low as 1. Therefore we adjust `minexp` to fulfill the minimal precision requirements of MPFR. Setting the precision allocates a mantissa with $k = \lceil(\texttt{maxexp} - \texttt{minexp})/\texttt{GMP\_NUMB\_BITS}\rceil$ limbs. We use `maxexp` later to align the remaining summands to the mantissa of `rop`. Figure 4 shows the alignment of the first and later summands on the mantissa of `rop` for the case of 32 bit limbs.

⟨*set up precision, sign and exponent*⟩≡
```
ieee_binary64 X;
X.d = e[0];
mp_exp_t minexp = X.exponent();

X.d = e[n-1];
mp_exp_t maxexp = X.exponent()+53;

if(!X.normalized()){
  const double d = 9007199254740992.0;
  assert(d == ldexp(1.0,53));
  X.d *= d;
  assert(X.normalized());
  maxexp = X.exponent(); //+53-53;
  minexp = std::min(minexp,maxexp-MPFR_PREC_MIN);
}

const mpfr_prec_t prec = maxexp-minexp;
assert(prec >= MPFR_PREC_MIN);
mpfr_set_prec(rop,prec);
assert(rop->_mpfr_prec == prec);

const int k = (prec+GMP_NUMB_BITS-1)/GMP_NUMB_BITS;
assert( (k-1)*GMP_NUMB_BITS < prec && prec <= k*GMP_NUMB_BITS );

rop->_mpfr_sign = X.sign();
rop->_mpfr_exp  = maxexp;
mp_limb_t *const mant = rop->_mpfr_d;
```

Writing the mantissa of the leading summand is again done as in Section 4.1. The lower limbs of `rop` may be overlapped by multiple mantissae of remaining summands. Furthermore, mantissae may overlap

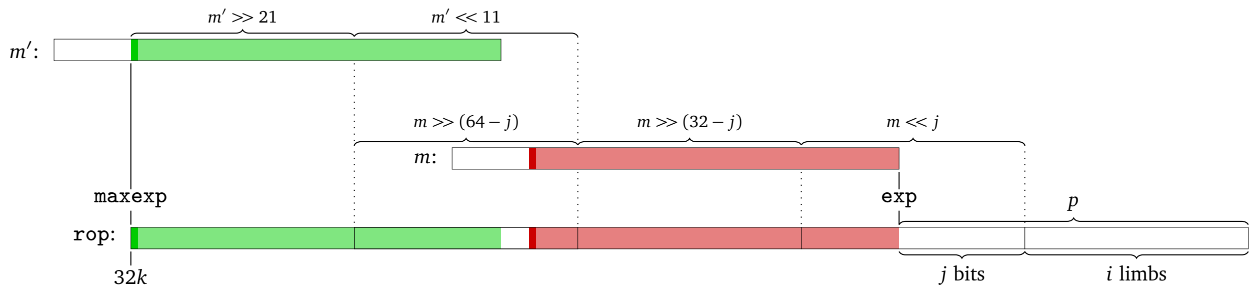

Figure 4: Aligning the mantissa of the leading summand m' and any other mantissa m to a 32 bit limb arbitrary precision mantissa.

each other, however in all but one mantissa the overlapping bits are zero, cf. again Figure 1. We zero the lower limbs of `rop` so we can write the remaining summands by bitwise `or`.

⟨*clear mantissa and write first summand*⟩≡
```
#if GMP_NUMB_BITS==32
  mant[k-1] = (X.mantissa() » 21);
  mant[k-2] = (X.mantissa() « 11);
  for(int i=k-3;i>=0;-i) mant[i] = 0;
#else //GMP_NUMB_BITS==64
  mant[k-1] = (X.mantissa() « 11);
  for(int i=k-2;i>=0;-i) mant[i] = 0;
#endif
```

We iterate over the remaining summands and copy each mantissa. This loop is completely free of branches, including computations inside `ieee_binary64` member functions.

⟨*write remaining summands*⟩≡
```
    int i = n-2;
    while(i>=0){
      const double wrt = e[i-];
      ⟨write summand⟩
    }
```

For any of the remaining summands we compute the position p of the last bit of its mantissa m in the mantissa of `rop`, cf. Figure 4. Then we compute the index i of the limb this bit belongs to and the number of bits j it has to be shifted to the left.

⟨*write summand*⟩≡
```
  {
    X.d = wrt;
    mpfr_exp_t exp = X.exponent();
    unsigned long long m = X.mantissa();

    const int p = GMP_NUMB_BITS*k - maxexp + exp;
    assert(p >= 0);

    const int i = p/GMP_NUMB_BITS;
    const int j = p%GMP_NUMB_BITS;
    assert(0 <= i && i < k);

  #if GMP_NUMB_BITS==32
    ⟨write mantissa to 32 bit limbs⟩
  #else //GMP_NUMB_BITS==64
    ⟨write mantissa to 64 bit limbs⟩
  #endif
  }
```

For 32 bit limbs, the mantissa m may overlap up to three limbs. While index i always points to an existing limb, the limbs at position $i+1$ and $i+2$ may not exist, i.e., we are out of array bounds. In this case, however, the part m_1 of m overlapping this nonexisting limb is zero. Instead of branching whether we are outside bounds, we use m_1 to recalculate the index. If $m_1 = 0$ then $i_1 = 0$ and otherwise $i_1 = i+1$. Since we write the mantissa using bitwise `or`, no harm is done in the first case and the mantissa is written correctly in the other case.

⟨*write mantissa to 32 bit limbs*⟩≡
```
  const unsigned long long m1 = (m  >> (32 - j));
  const unsigned long long m2 = (m1 >> 32);

  const int i1 = (i+1) * static_cast<bool>(m1);
  const int i2 = (i+2) * static_cast<bool>(m2);

  assert(i+1 < k || m1 == 0);
  assert(i+2 < k || m2 == 0);

  mant[i]  |= (m << j);
  mant[i1] |= m1;
  mant[i2] |= m2;
```
For 64 bit limbs the mantissa may overlap up to two limbs. We use the same trick as above to avoid branching when out of array bounds. But there is another issue. Shifting m by 64 bits will quite unintuitively leave m unchanged instead of setting $m = 0$. Since j may be zero, we halve $64 - j$ and perform the right shift in two steps.

⟨*write mantissa to 64 bit limbs*⟩≡
```
  const int j1 = (64 - j) >> 1;
  const int j2 = (64 - j) - j1;

  const unsigned long long m1 = ((m >> j1) >> j2);
  const int i1 = static_cast<bool>(m1)*(i+1);

  assert(i+1 < k || m1 == 0);

  mant[i]  |= (m << j);
  mant[i1] |= m1;
```

4.3 General Expansions

Using the algorithms from above we present two options to convert a general expansion into an MPFR number. The first option is to split the expansion into two monotone expansions, convert them separately and then perform an exact addition of two MPFR numbers. Note that there are no floating-point operations involved, hence there is no danger of overflow.

Algorithm 26. *Given an expansion* $e_0, e_1, \ldots, e_{n-1}$*, the function* `mpfr_set_expansion_split()` *computes an* MPFR *number* `rop` *with* `rop` $= \sum_{i=0}^{n-1} e_i$.

⟨*convert an expansion by splitting to mpfr*⟩≡
```
  void mpfr_set_expansion_split(mpfr_t rop,const int n ,const double *const e){

    ⟨check summands are finite⟩

    double pos[n];
    double neg[n];

    int plen=0;
    int nlen=0;
    for(int i=0;i<n;i++) {
      if(e[i]>0.0)      pos[plen++]=e[i];
      else if(e[i]<0.0) neg[nlen++]=e[i];
    }

    if(plen*nlen!=0){

      mpfr_t x1,x2;
      mpfr_init(x1);
      mpfr_init(x2);

      mpfr_set_monotone_expansion(x1,plen,pos);
      mpfr_set_monotone_expansion(x2,nlen,neg);

      mp_exp_t e1 = mpfr_get_exp(x1);
      mp_exp_t p1 = mpfr_get_prec(x1);

      mp_exp_t e2 = mpfr_get_exp(x2);
      mp_exp_t p2 = mpfr_get_prec(x2);

      mp_prec_t p = std::max(e1,e2) - std::min(e1-p1,e2-p2);

      mpfr_set_prec(rop,p);

  #ifndef NDEBUG
      int round =
  #endif
      mpfr_add(rop,x1,x2,GMP_RNDN);
      assert(!round);

      mpfr_clear(x1);
      mpfr_clear(x2);

    }else if(plen!=0){
      mpfr_set_monotone_expansion(rop,plen,pos);
    }else if(nlen!=0){
      mpfr_set_monotone_expansion(rop,nlen,neg);
    }else{
      mpfr_set_ui(rop,0,GMP_RNDN);
    }
  }
```

The exact addition of x_1 and x_2 in `mpfr_set_expansion_split()` is justified by the following lemma. For the proof we reuse the functions msb and lsb that were defined for arbitrary floating-point numbers.

Lemma 27. *Let $x_i = m_i \cdot 2^{e_i},\ i = 1,2$ be two positive* MPFR *numbers, where the precision of m_i is p_i. For the difference $x = x_1 - x_2$ to be exact, it suffices to set the precision of x to*

$$p = \max\{e_1, e_2\} - \min\{e_1 - p_1, e_2 - p_2\}. \tag{65}$$

Proof. It suffices to set

$$p = \log_2(\mathrm{msb}(x)) - \log_2(\mathrm{lsb}(x)). \tag{66}$$

MPFR normalizes numbers such that $m_i \in [0.5, 1)$, so we have

$$\begin{aligned} \log_2(\mathrm{msb}(x_i)) &= e_i \\ \log_2(\mathrm{lsb}(x_i)) &= e_i - p_i \quad \text{for } i = 1,2. \end{aligned} \tag{67}$$

Furthermore $x \leq \max\{x_1, x_2\}$, so

$$\log_2(\text{msb}(x)) \;\leq\; \max\{e_1, e_2\} \tag{68}$$

and

$$\log_2(\text{lsb}(x)) \;\geq\; \log_2(\min_{i=1,2} \text{lsb}(x_i)) \;=\; \min\{e_1 - p_1, e_2 - p_2\}. \tag{69}$$

□

The second option is, to first convert an expansion into an equivalent monotone expansion and then into an MPFR number. By Theorem 21, the expansion generated by MONOTONIZE has at most $\lceil \log_2(2\tau/\eta)/p \rceil$ summands which yields $\lceil (1024 + 1074)/53 \rceil = 40$ for `binary64`.

Algorithm 28. *Given an expansion* $e_0, e_1, \ldots, e_{n-1}$ *with* $\sum_{i=0}^{n-1} |e_i| < \tau(2 - \varepsilon_\mathrm{m})$*, e.g., a strongly nonoverlapping expansion, the function* `mpfr_set_expansion_monotonize()` *computes an* MPFR *number* `rop` *with* `rop` $= \sum_{i=0}^{n-1} e_i$.

⟨*convert an expansion by monotonizing to mpfr*⟩≡

```
void mpfr_set_expansion_monotonize(mpfr_t rop, const int n, const double *const e){

  ⟨check summands are finite⟩
  ⟨handle less than two summands⟩

  const int l = 40;
  double h[l];

  const int k = de_monotonize(n,e,l,h);
  double *const f= h+l-k;

  mpfr_set_monotone_expansion(rop,k,f);
}
```

5 Experiments

We compare the running time of our new conversion approaches to two direct conversion approaches that compute the sum exactly using MPFR functionality. In both direct approaches we add summands in decreasing magnitude to a total s using the `mpfr_add_d()` function. For the addition to be exact, s has to have sufficient precision. The first variant *direct 1* increases the precision in every step, if necessary, while the second variant *direct 2* sets the precision to a sufficient value at the beginning. The latter method turned out to be more efficient. Using the MPFR function `mpfr_sum()`, to compute s in one step is even slower. Our two conversion approaches are called *split* if the first step is to split the expansion and *monotonize* if the first steps consists of using MONOTONIZE.
As input data we use randomly generated expansions. Although expansions could be generated artificially summand by summand, we create them by evaluating a polynomial expression using Shewchuks arithmetic operations. This way, test expansions are more likely to have a structure that actually occurs in applications. As expression D we compute a 4×4 determinant of 4×4 determinants of randomly generated numbers. D has a polynomial degree of $d = 16$. We compute input numbers using the `rand48()` family of functions. We chose a floating-point number $x \in [0, 1]$, a sign $s \in \{-1, +1\}$ and an exponent $e \in \{-17, \ldots, 17\}$ uniformly at random and use $s \cdot x \cdot 2^e$ as input number. All input numbers can be uniformly scaled to integers with $p = 35 + 53$ bit precision. Hence, D can be represented with approximately $dp \approx 1400$ bits. We observed however that D was in fact representable with approximately 1000 bits on average only.
The result of the evaluation are strongly nonoverlapping expansions with about 220 summands on average. If we additionally compress the sums, we get nonadjacent expansions with about 20 summands on

average. In fact, we chose d and the range for e above to generate expansions with about 20 summands after compression. In a compressed expansion, each summand carries about 52 bits of information. Therefore, before compressing, each summand carries only about 5 bits of information on average. Thus, uncompressed and compressed expansions are input sets with quite different characteristics. This is interesting insofar MONOTONIZE compresses as a side effect and hence may reduce the number of summands significantly before the actual conversion. We consider expansions with $n = 1, 2, \dots, 64$ summands in the uncompressed case and $n = 1, 2, \dots, 20$ in the compressed case. To generate expansion with fewer than 220 (respectively 20) summands we simply ignore the leading summands.

We run tests for an MPFR limb size of both 64 bit and 32 bit. For 64 bit limbs we use MPFR 3.0.1 (with GMP 5.0.1) as provided by our operating system, for 32 bit limbs we compiled MPFR 3.0.1 (with GMP 5.0.2) on our own. This was done using g++'s -m32 flag, since MPFR does not support 32 bit limbs in a 64 bit environment. All our code was compiled with g++ 4.6.1 and $-O3$. Experiments were run on an Intel Core i5 CPU with 3.33 Ghz. To get measurable running times, we generate 2000 expansions as described above, and measure the total time for converting all expansions a 1000 times. The results are shown in Figure 5. The graphs do not show running time but the speedup $t_{\text{direct }1}/t_x$ of each method x with respect to the faster direct approach direct 1 on a logarithmic scale. This improves the display of differences for the important range with very few summands, where the actual running times are very small.

Both, monotonize and split clearly outperform the direct approaches, and monotonize is uniformly the fastest method. As expected, monotonize achieves greater speedup for uncompressed than compressed expansions. There is a small dip in the speedup achieved by monotonize for uncompressed expansions, starting at approximately 12 summands. Since these summands carry only about 5 bits of information,

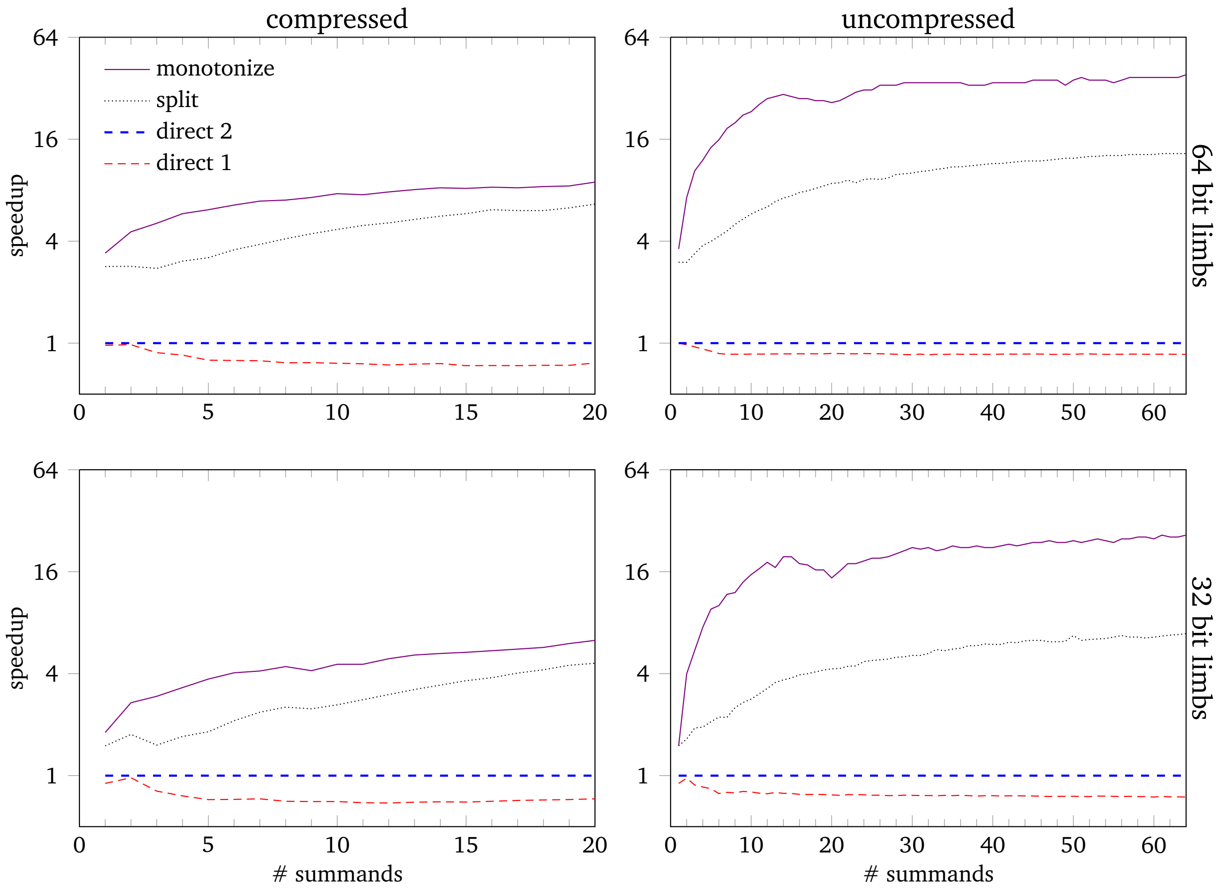

Figure 5: Experimental results for converting expansions to MPFR. Plottings show the speedup for each conversion method, relative to the trivial conversion - - - direct 2 on a logarithmic scale.

this is the range where the number of output summand of MONOTONIZE jumps from one to two. While in general the graphs look similar for 32 bit and 64 bit limbs, the advantage of our new methods is even larger for 64 bit limbs.
Note that even in the case of one summand, our new approaches are better. In this case both direct approaches simply call `mpfr_set_d()`, while our new approaches call `mpfr_set_double()` from Section 4.1. The differences between monotonize and split on the one hand and direct 1 and direct 2 on the other hand originate from noise in the measurement process, since the actual running times are very small if only one summand is converted.

6 Conclusion

We have presented two new methods to convert expansions to arbitrary precision floating-point numbers. The faster method, based on our new algorithm MONOTONIZE works for almost all expansions, in particular for strongly nonoverlapping expansions that are relevant in practice. Only expansions with a value that is very close to the overflow range, i.e., almost not representable as expansion can not be converted. For these cases the other conversion method may be used, which still gives a significant improvement over the trivial approach to conversion.
Our methods are so fast, that they may help to speed up the exact computation of arbitrary sums, not only expansions, too. Efficient algorithms to compute a sum of floating-point numbers exactly as expansion have been given by Shewchuk [17] or Rump et al. [16].
Considering our own application, we have yet to integrate the new conversion methods into our package `Real_algebraic` and examine the resulting speedup.

References

[1] Christoph Burnikel and Jochen Könemann. High-precision floating point numbers in LEDA. Research Report MPI-I-96-1-002, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, January 1996.

[2] Theodorus Jozef Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[3] Yozo Hida, Xiaoye S. Li, and David H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH '01)*, pages 155–162, Washington, DC, USA, 2001. IEEE Computer Society.

[4] *IEEE Standard 754-2008: IEEE Standard for Floating-Point Arithmetic*. 2008. Revision of ANSI/IEEE Standard 754-1985.

[5] LEDA: Library of Efficient Data Structures and Algorithms. `http://www.algorithmic-solutions.com/`.

[6] Marc Mörig. Deferring dag construction by storing sums of floats speeds-up exact decision computations based on expression dags. In *3rd International Congress on Mathematical Software (ICMS 2010)*, volume 6327 of *LNCS*, pages 109–120, September 2010.

[7] Marc Mörig, Ivo Rössling, and Stefan Schirra. On the design and implementation of a generic number type for real algebraic number computations based on expression dags. *Mathematics in Computer Science*, 4(4):539–556, 2010.

[8] MPFR: A multiple precision floating-point library. `http://www.mpfr.org/`.

[9] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[10] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

[11] Douglas M. Priest. Algorithms for arbitrary precision floating point arithmetic. In *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 132–143, New York, June 1991. IEEE Computer Society Press.

[12] Douglas M. Priest. *On properties of floating point arithmetics: numerical stability and the cost of accurate computations*. PhD thesis, University of California at Berkeley, 1992.

[13] RealAlgebraic: A number type for exact geometric computation. `http://wwwisg.cs.uni-magdeburg.de/ag/RealAlgebraic/`.

[14] Siegfried Rump, Paul Zimmermann, Sylvie Boldo, and Guillaume Melquiond. Computing predecessor and successor in rounding to nearest. *BIT Numerical Mathematics*, 49:419–431, 2009.

[15] Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi. Accurate floating-point summation part I: Faithful rounding. *SIAM J. Scientific Computing*, 31(1):189–224, 2008.

[16] Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi. Accurate floating-point summation part II: Sign, k-fold faithful and rounding to nearest. *SIAM J. Scientific Computing*, 31(2):1269–1302, 2008.

[17] Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete and Computational Geometry*, 18(3):305–363, 1997.

[18] Pat H. Sterbenz. *Floating-Point Computation*. Prentice-Hall, 1974.

A Converting Expansions to `leda::bigfloat`

Similar to Section 4 we present algorithms and code to convert expansions to `leda::bigfloat`.

A.1 Monotone Expansions

We have two alternative functions to convert a monotone expansion into a `leda::bigfloat`. The first one is based on the LEDA interface, since we can not access the internal structure of a `leda::bigfloat` without some hackery. Instead we extract sign and exponent as integers and the mantissa as an array of limbs first and then create a `leda::bigfloat` using low level constructors.

Algorithm 29.
Given a monotone expansion $e_0, e_1, \ldots, e_{n-1}$*, then* `monotone_expansion_to_leda_bigfloat()` *computes a* `leda::bigfloat rop` *with* `rop` $= \sum_{i=0}^{n-1} e_i$.

```
⟨convert a monotone expansion to leda::bigfloat⟩≡
  inline void
  monotone_expansion_to_leda_bigfloat(leda::bigfloat& rop,
                                      const int n, const double *const e){

    ⟨check summands are finite⟩
    const int crossover = 4;
    ⟨handle expansions shorter than crossover⟩
    ⟨check summands are nonzero⟩
    ⟨leda set up precision and sign⟩
    ⟨leda set up mantissa and write first summand⟩
    ⟨leda write all summands⟩
    ⟨create leda::bigfloat from extracted data⟩
  }
```

If we stick to the LEDA interface, our conversion method is inefficient for expansions with only a few summands. Hence we convert short expansions directly. The crossover point of 4 summands was determined experimentally.

⟨handle expansions shorter than crossover⟩≡

```
  if(n==0){
    rop = 0;
    return;
  }else if(n < crossover){
    rop = e[n-1];

    for(int i=n-2;i>=0;-i){
      leda::bigfloat y = e[i];
      rop = add(rop,y,53,leda::EXACT);
    }
    return;
  }
```

Then we compute the necessary precision. `maxexp` points somewhere before the most significant bit and `minexp` somewhere behind the least significant bit of the expansion. LEDA considers the mantissa to be an integer, hence we use `minexp` as exponent for the final result and there is no need to normalize in any way.

⟨leda set up precision and sign⟩≡

```
  ieee_binary64 X;
  X.d = e[n-1];
  long maxexp = X.exponent()+53;

  X.d = e[0];
  long minexp = X.exponent();

  const long prec = maxexp-minexp;
  assert(prec >= 2);

  const int k = (prec+LEDA_NUMB_BITS-1)/LEDA_NUMB_BITS;
  assert( (k-1)*LEDA_NUMB_BITS < prec && prec <= k*LEDA_NUMB_BITS );

  int sign = X.sign();
```

We create a vector for the mantissa and copy the mantissa of the least significant summand such that its last bit (zero or nonzero), corresponding to `minexp`, is the last bit of the vector.

⟨leda set up mantissa and write first summand⟩≡

```
  leda::digit mant[k];

  #if LEDA_NUMB_BITS==32
    mant[0] = X.mantissa();
    mant[1] = (X.mantissa()»32);
    for(int i=2;i<k;i++) mant[i] = 0;
  #else //LEDA_NUMB_BITS==64
    mant[0] = X.mantissa();
    for(int i=1;i<k;i++) mant[i] = 0;
  #endif
```

Then we convert the remaining summands.

⟨*leda write all summands*⟩≡
```
  int i = 1;
  do{
    const double wrt = e[i++];
    ⟨leda write summand⟩
  }while(i < n);
```
We align the summand using `minexp`. Copying the mantissa is done in code chunks from Section 4.

⟨*leda write summand*⟩≡
```
  {
    X.d = wrt;
    long exp = X.exponent();
    unsigned long long m = X.mantissa();

    const int p = exp-minexp;
    assert(p >= 0);

    const int i = p/LEDA_NUMB_BITS;
    const int j = p%LEDA_NUMB_BITS;
    assert(0 <= i && i < k);

  #if LEDA_NUMB_BITS==32
    ⟨write mantissa to 32 bit limbs⟩
  #else //LEDA_NUMB_BITS==64
    ⟨write mantissa to 64 bit limbs⟩
  #endif
  }
```
Then we create a `leda::integer` for the mantissa and a `leda::bigfloat` from there. This has the disadvantage that the mantissa must be copied to the `leda::integer` and will probably be copied again when creating the `leda::bigfloat`. For this reason, this conversion method is inefficient for short expansions.

⟨*create leda::bigfloat from extracted data*⟩≡
```
  leda::integer mantissa(k,mant,sign);
  rop = leda::bigfloat(mantissa,minexp);
```
We now give an alternative, that writes all data directly into the `leda::bigfloat` data-structures. The code below works by

⟨*include leda::bigfloat public*⟩≡
```
  #define private public
  #include <LEDA/numbers/bigfloat.h>
  #undef private
```
you have been warned. LEDA is not an open source library, so we have little insight into the internals. What we know, we learned from a very outdated technical report [1] on one of the first versions of `leda::bigfloat`, from access to source code for LEDA 4.4 and from the header files of a current LEDA version.

Algorithm 30.
Given a monotone expansion $e_0, e_1, \ldots, e_{n-1}$, then `monotone_expansion_to_leda_bigfloat()` *computes a* `leda::bigfloat rop` *with* `rop` $= \sum_{i=0}^{n-1} e_i$.

⟨*convert a monotone expansion to leda::bigfloat faster*⟩≡
```
  inline void
  monotone_expansion_to_leda_bigfloat(leda::bigfloat& rop,
                                      const int n ,const double *const e){

    ⟨check summands are finite⟩
    ⟨handle zero or one summand⟩
    ⟨check summands are nonzero⟩

    ⟨set up precision leda internal⟩
    ⟨set up bigfloat representation⟩
    ⟨copy all summands to mantissa⟩
    ⟨remove trailing zero limbs⟩
    ⟨set bigfloat to representation⟩
  }
```

We handle expansions with less than two summands first and separately.

⟨handle zero or one summand⟩≡

```
if(n == 0){
  rop = 0;
  return;
}else if(n == 1){
  rop = e[0];
  return;
}
```

`leda::bigfloat` has some interesting normalizing conditions. Stored are an integer mantissa m and exponent e representing $m \cdot 2^e$. Let $b =$ `LEDA_NUMB_BITS` be the number of bits in a limb, then e must be divisible by b. The trailing $\log_2(b)$ bits of e are therefore always zero and not stored! Furthermore the least significant and most significant limb in m must be nonzero.
From the leading summand, we first compute `maxexp` to point directly in front of the most significant nonzero bit of the expansion. This ensures later that the most significant limb is not zero. Then we compute `minexp`, to point somewhere behind the least significant nonzero bit of the expansion. We decrease `minexp` further to be divisible by b. Integer division rounds towards zero, so we have to make sure `minexp` is positive at the point of division. Then we can compute the necessary precision and number of limbs k.

⟨set up precision leda internal⟩≡

```
ieee_binary64 X;
X.d = e[n-1];
long maxexp = X.exponent()+53;

if(!X.normalized()){
  const double d = 9007199254740992.0;
  assert(d == ldexp(1.0,53));
  X.d *= d;
  assert(X.normalized());
  maxexp = X.exponent(); //+53-53;
}

X.d = e[0];
long minexp = X.exponent();

minexp += 100*LEDA_NUMB_BITS;
minexp = LEDA_NUMB_BITS*(minexp/LEDA_NUMB_BITS);
minexp -= 100*LEDA_NUMB_BITS;

const long prec = maxexp-minexp;
assert(prec >= 2);

const long k = (prec+LEDA_NUMB_BITS-1)/LEDA_NUMB_BITS;
assert( (k-1)*LEDA_NUMB_BITS < prec && prec <= k*LEDA_NUMB_BITS );
```

`leda::bigfloat` is reference counted. We create a representation for k limbs and set/check some variables. The less obvious ones are: `count` – the reference counter, `size` – the number of available limbs, `exp_ptr` – pointer to a representation for the exponent if it is not representable as `long` and `used` – the number of limbs in use.

⟨*set up bigfloat representation*⟩≡
```
  leda::bigfloat_rep* rep = leda::bigfloat::new_rep(k);
  assert(rep->count == 1);
  assert(rep->size == k);
  assert(rep->exp_ptr == 0);

  rep->used = k;
  rep->bitlength=prec;
  rep->sign = X.sign();
  rep->exponent = minexp/LEDA_NUMB_BITS;
```
Then we copy the mantissae of the summands. We need to align the least significant summand like all other summands, since `minexp` is not necessarily its exponent.

⟨*copy all summands to mantissa*⟩≡
```
  leda::digit *const mant = rep->vec;
  for(int i=0;i<k;i++) mant[i] = 0;

  for(int i=0;i<n;i++){
    const double wrt = e[i];
    ⟨leda write summand⟩
  }
```
After copying the mantissae, there may be trailing zero limbs, however at most $\lfloor(52+31)/32\rfloor = 2$ for 32 bit limbs (52 zero bits in the least significant summand + 31 zero bits for decreasing `minexp`) and $\lfloor(52+63)/64\rfloor = 1$ for 64 bit limbs.

⟨*remove trailing zero limbs*⟩≡
```
  if(mant[0] == 0){
    int z=0;
    while(mant[++z]==0);
    rep->vec+=z;
    rep->exponent+=z;
    rep->used-=z;
    rep->bitlength -= z*LEDA_NUMB_BITS;
  }
```
Then we assign the new representation to `rop`. `rop` may currently point to some other representation, so we have to clear it fist.

⟨*set bigfloat to representation*⟩≡
```
  rop.clear();
  rop.PTR = rep;
  rop.special = leda::bigfloat::NOT_VAL;
```

A.2 General Expansions

The code here is really just a copy from Section 4.3. `leda::bigfloat` does however have an exact addition that we can use in `expansion_to_leda_bigfloat_split()`. Again we handle short expansions with fewer than 8 summands directly if we are bound to the LEDA interface.

Algorithm 31.
Given an expansion $e_0, e_1, \ldots, e_{n-1}$*, then the function* `expansion_to_leda_bigfloat_split()` *computes a* `leda::bigfloat rop` *with* `rop` $= \sum_{i=0}^{n-1} e_i$.

⟨*convert an expansion by splitting to leda::bigfloat*⟩≡
```
  void expansion_to_leda_bigfloat_split(leda::bigfloat& rop,
                                        const int n ,const double *const e){
    ⟨check summands are finite⟩
  #ifdef USE_DOCUMENTED_LEDA_INTERFACE
    const int crossover = 8;
    ⟨handle expansions shorter than crossover⟩
  #else
    ⟨handle zero or one summand⟩
  #endif

    double pos[n];
    double neg[n];

    int plen=0;
    int nlen=0;

    for(int i=0;i<n;i++) {
      if(e[i]>0.0)      pos[plen++]=e[i];
      else if(e[i]<0.0) neg[nlen++]=e[i];
    }

    if(plen*nlen!=0){

      leda::bigfloat x;

      monotone_expansion_to_leda_bigfloat(rop,plen,pos);
      monotone_expansion_to_leda_bigfloat(x,nlen,neg);

      rop = leda::add(rop,x,53,leda::EXACT);

    }else if(plen!=0){
      monotone_expansion_to_leda_bigfloat(rop,plen,pos);
    }else if(nlen!=0){
      monotone_expansion_to_leda_bigfloat(rop,nlen,neg);
    }else{
      rop = 0;
    }
  }
```

Algorithm 32. *Given an expansion* $e_0, e_1, \ldots, e_{n-1}$ *with* $\sum_{i=0}^{n-1} |e_i| < \tau(2-\varepsilon_\mathrm{m})$*, e.g., a strongly nonoverlapping expansion, then* `expansion_to_leda_bigfloat_monotonize()` *computes a* `leda::bigfloat` `rop` *with* `rop` $= \sum_{i=0}^{n-1} e_i$.

⟨*convert an expansion by monotonizing to leda::bigfloat*⟩≡
```
  void expansion_to_leda_bigfloat_monotonize(leda::bigfloat& rop,
                                             const int n ,const double *const e){
    ⟨check summands are finite⟩
    ⟨handle zero or one summand⟩

    const int l = 40;
    double g[l];

    const int k = de_monotonize(n,e,l,g);
    double *const f = g+l-k;

    monotone_expansion_to_leda_bigfloat(rop,k,f);
  }
```

A.3 Experiments

For experiments with LEDA, we use LEDA 6.3 that comes precompiled for both 32 bit and 64 bit environments. Here, `leda::bigfloat` has a limb size of 32 and 64 bits in the corresponding LEDA versions. For the direct conversion *direct 1*, we convert each summand individually. Then we add them up one by one and by decreasing magnitude, using LEDA's exact addition. In *direct 2* we add by increasing magnitude. Otherwise, the experimental setup is the same as in Section 5. We have each of the new conversion approaches *split* and *monotonize* in two version, one that adheres to the LEDA interface and one that bypasses the interface and writes directly to `leda::bigfloat` internals. Results are shown in Figure 6 and are similar to the results obtained for conversion to MPFR.
If we rely on the LEDA interface, an improvement is made with monotonize for more than 3 summands and with split for more than 7 summands only. Originally, monotonize and split were even slower below these thresholds, but the final versions use a direct conversion approach in this case. For monotonize, one can nicely see the ranges where number of output summands of MONOTONIZE increases from one to two and from two to three for the case of uncompressed input expansions.
The conversion methods that avoid the LEDA interface and write data directly to the `leda::bigfloat` internals are a much better choice. They improve upon the direct approaches starting with two summands and by a much wider margin. Conversion by monotonize is uniformly the best choice in this case.

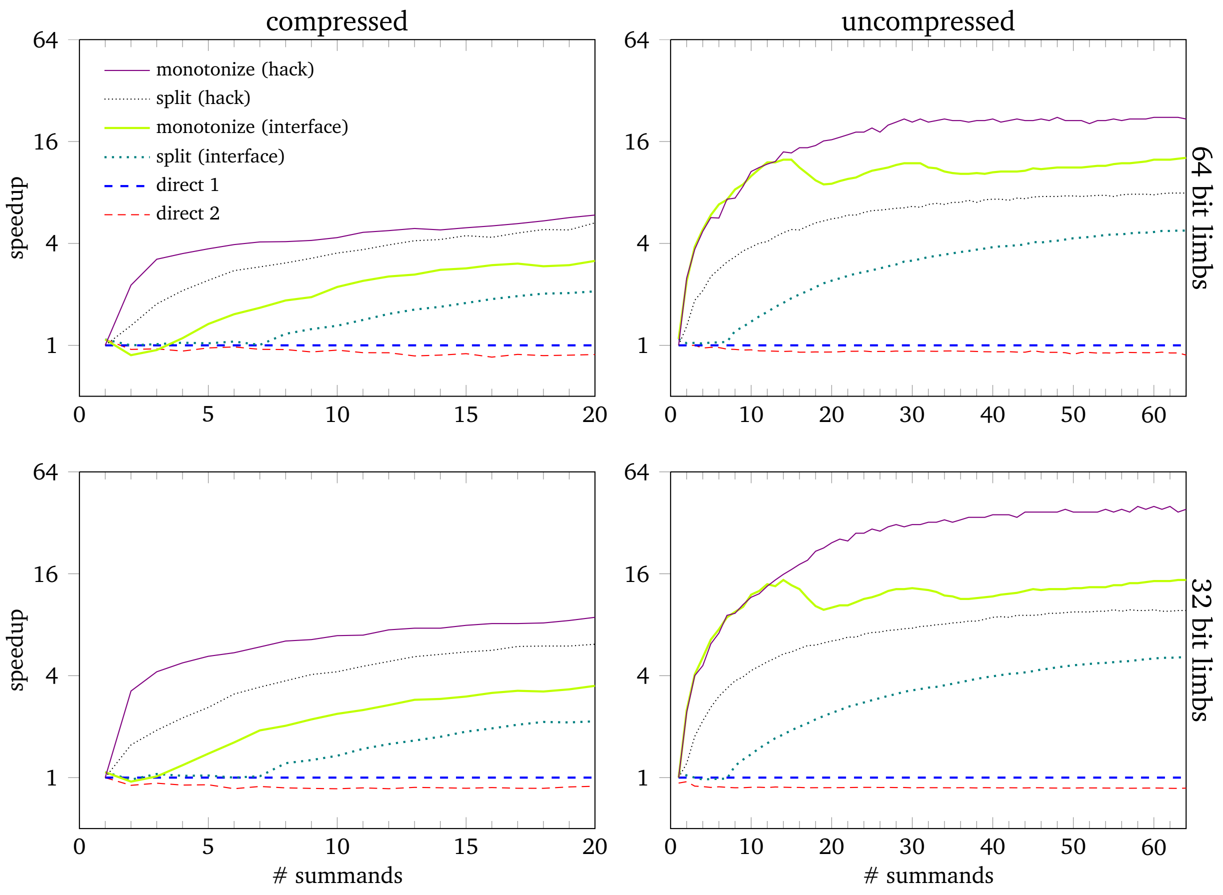

Figure 6: Experimental results for converting expansions to `leda::bigfloat`. Plottings show the speedup for each conversion method, relative to the trivial conversion - - - direct 1 on a logarithmic scale.