Arithmetic algorithms for extended precision using floating-point expansions

Mioara Joldes, Olivier Marty, Jean-Michel Muller, Valentina Popescu

Arithmetic algorithms for extended precision using floating-point expansions

Mioara Joldes[1], Olivier Marty[2], Jean-Michel Muller[3] and Valentina Popescu[4]

[1]CNRS, LAAS Laboratory, 7 Avenue du Colonel Roche, 31031 Toulouse, France
[2]ENS Cahan, 61 Avenue du Président Wilson, 94230 Cachan, France
[3]CNRS, Inria Grenoble Rhône-Alpes / LIP Laboratoire de l'Informatique du Parallélisme - ARIC, ENS Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France
[4]Inria Grenoble Rhône-Alpes / LIP Laboratoire de l'Informatique du Parallélisme - ARIC, ENS Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France

January 30, 2015

Abstract

Many numerical problems require a higher computing precision than the one offered by standard floating-point (FP) formats. One common way of extending the precision is to represent numbers in a *multiple component* format. By using the so-called *floating-point expansions*, real numbers are represented as the unevaluated sum of standard machine precision FP numbers. This representation offers the simplicity of using directly available, hardware implemented and highly optimized FP operations and is used by multiple-precision libraries such as Bailey's QD or the analogue Graphics Processing Units (GPU) tuned version, GQD. In this article we revisit algorithms for adding and multiplying FP expansions, then we introduce and prove new algorithms for normalizing, dividing and square rooting of FP expansions. The new method used for computing the reciprocal $\boldsymbol{a^{-1}}$ and the square root $\boldsymbol{\sqrt{a}}$ of a FP expansion $\boldsymbol{a}$ is based on an adapted Newton-Raphson iteration where the intermediate calculations are done using "truncated" operations (additions, multiplications) involving FP expansions. We give here a thorough error analysis showing that it allows very accurate computations. More precisely, after $\boldsymbol{q \geq 0}$ iterations, the computed FP expansion $\boldsymbol{x = x_0 + \ldots + x_{2^q-1}}$ satisfies, for the reciprocal algorithm, the relative error bound: $\left|\frac{x-a^{-1}}{a^{-1}}\right| \leq \mathbf{2^{-2^q(p-3)-1}}$ and, respectively, for the square root one: $\left|x - \frac{1}{\sqrt{a}}\right| \leq \frac{2^{-2^q(p-3)-1}}{\sqrt{a}}$, where $\boldsymbol{p > 2}$ is the precision of the FP representation used ($\boldsymbol{p = 24}$ for single precision and $\boldsymbol{p = 53}$ for double precision).

1 Introduction

Many numerical problems in dynamical systems or planetary orbit dynamics, such as the long-term stability of the solar system [12], finding sinks in the Henon Map [10], iterating the Lorenz attractor [1], etc., require higher precisions than

the standard double precision (now called *binary64* [7]). Quad or higher precision is rarely implemented in hardware, and the most common solution is to use software emulated higher precision libraries, also called arbitrary precision libraries. There are mainly two ways of representing numbers in higher precision. The first one is the *multiple-digit representation*: numbers are represented with a sequence of possibly high-radix digits coupled with a single exponent. An example is the representation used in GNU MPFR [5], an open-source C library, which, besides arbitrary precision, also provides correct rounding for each atomic operation (basic operations and functions). The second way is the *multiple-term representation* in which a number is expressed as the unevaluated sum of several standard floating-point (FP) numbers. This sum is usually called a *FP expansion*. Bailey's library QD [6] uses this approach and supports double-double (DD) and quad-double (QD) computations, i.e. numbers are represented as the unevaluated sum of 2 or 4 standard double-precision FP numbers. It is known [13], however, that the DD and QD formats and the operations implemented in that library are not compliant with the IEEE 754-2008 standard, and do not provide correctly rounded operations. However, this multiple-term representation offers the simplicity of using directly available and highly optimized hardware implemented FP operations. This makes most multiple-term algorithms straightforwardly portable to highly parallel architectures, such as GPUs. In consequence, there is a demand for algorithms for arithmetic operations with FP expansions, that are sufficiently simple yet efficient, and for which effective error bounds and thorough proofs are given. Several algorithms already exist for addition and multiplication [13, 6, Thm. 44, Chap. 14].

In this article we mainly focus on division (and hence, *reciprocal*) and square root, which are less studied in literature. For these algorithms we also provide a thorough error analysis and effective error bounds. There are two classes of algorithms for performing division and square root: the so-called *digit-recurrence algorithms* [4], that generalize the paper-and-pencil method, and the algorithms based on the Newton-Raphson iteration [20, 2]. While the algorithms suggested so far for dividing expansions belong to the former class, here we will be interested in studying the possible use of the latter class: since its very fast, quadratic convergence is appealing when high precision is at stake.

Another contribution of this article is a new method for the renormalization of FP expansions. This operation ensures certain precision related requirements and is an improrant basic brick in most computations with FP expansions. Our renormalization procedure takes advantage of the computer's pipeline, so it is fast in practice. For the sake of completeness, we also briefly present a variant of addition and multiplication algorithms which we implemented, and for which we intend on providing a full error analysis in a future related article.

A preliminary version of our work concerning only the case of division was recently presented in [9].

The outline of the paper is the following: in Section 2 we recall some basic notions about FP expansions and the algorithms used for handling them. Then, in Section 3 we give the new renormalization algorithm along with the proof of correctness. In Section 4 we present methods for computing the division, including existing algorithms based on long classical division on expansions (Sec. 4.1) and the Newton based method (Sec. 4.2), followed by the correctness proof, the error analysis and the complexity analysis. After that, in Section 5 we give a similar method for computing the square root of an expansion along with the

complexity analysis of the algorithm. Finally, in Section 6 we assess the performance of our algorithms – in terms of number of FP operations and proven accuracy bounds –.

2 Floating-point expansions

A normal binary precision-p floating-point (FP) number is a number of the form

$$x = M_x \cdot 2^{e_x - p + 1},$$

with $2^{p-1} \leq |M_x| \leq 2^p - 1$ and M_x is an integer. The integer e_x is called the *exponent* of x, and $M_x \cdot 2^{-p+1}$ is called the *significand* of x. We denote accordingly to Goldberg's definition: $\mathrm{ulp}(x) = 2^{e_x - p + 1}$ [13, Chap. 2] (ulp is an acronym for *unit in the last place*). Another useful concept is that of *unit in the last significant place*: $\mathrm{uls}(x) = \mathrm{ulp}(x) \cdot 2^{z_x}$, where z_x is the number of trailing zeros at the end of M_x.

In order to ensure the uniqueness of the representation we need to set the first bit of the significand to 1 and adjust the exponent according to that. This is called a *normalized* representation. This is not possible if x is less than $2^{e_{min}}$, where e_{min} is the smallest allowed exponent. Such numbers are called *subnormal*, the first bit of the significand is 0 and the exponent is the minimum representable one. The IEEE 754-2008 standard specifies that an *underflow* exception is raised every time subnormal numbers occur.

A natural extension of the notion of DD or QD is the notion of *floating-point expansion*. The arithmetic on FP expansions was first developed by Priest [16], and in a slightly different way by Shewchuk [19]. If, starting from a set of FP inputs, we only perform exact operations, then the values we obtain are always equal to finite sums of FP numbers. Such finite sums are called expansions. Hence, a natural idea is to try to manipulate such expansions, for performing calculations that are either exact, either approximate, yet very accurate.

Definition 2.1. A FP expansion u with n terms is the unevaluated sum of n FP numbers u_0, u_1, ..., u_{n-1}, in which all nonzero terms are ordered by magnitude (i.e., $u_i \neq 0 \Rightarrow |u_i| \geq |u_{i+1}|$). Each u_i is called a component of u.

It can be easily seen that the notion of expansion is redundant since a nonzero number always has more than one representation as a FP expansion. To make the concept useful in practice and easy to manipulate, we must introduce a constraint on the components: the u_i's cannot "overlap". The notion of overlapping varies depending on the authors. We give here two very different definitions, using the above-introduced notation.

Definition 2.2. (Nonoverlapping FP numbers) Assuming x and y are normal numbers with representations $M_x \cdot 2^{e_x - p + 1}$ and $M_y \cdot 2^{e_y - p + 1}$ (with $2^{p-1} \leq |M_x|, |M_y| \leq 2^p - 1$), they are *$\mathcal{P}$-nonoverlapping* (that is, nonoverlapping according to Priest's definition [17]) if $|e_y - e_x| \geq p$.

Definition 2.3. An expansion is *$\mathcal{P}$-nonoverlapping* (that is, nonoverlapping according to Priest's definition [17]) if all of its components are mutually *$\mathcal{P}$-nonoverlapping*.

A visual representation of Definition 2.3 can be seen in Fig. 1 (a).

Shewchuk [19] weakens this into *nonzero-overlapping* sequences as shown in Fig. 1 (b):

Definition 2.4. An expansion u_0, u_1, ..., u_{n-1} is $\mathcal{S}$*-nonoverlapping* (that is, nonoverlapping according to Shewchuk's definition [19]) if for all $0 < i < n$, we have $e_{u_{i-1}} - e_{u_i} \geq p - z_{u_{i-1}}$.

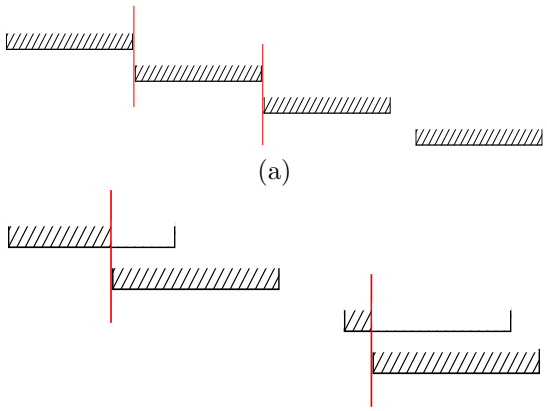

Figure 1: Nonoverlapping sequence by (a) Priest's scheme and (b) Shewchuk's scheme.

In general, a $\mathcal{P}$*-nonoverlapping* expansion carries more information than an $\mathcal{S}$*-nonoverlapping* one with the same number of components. In the worst case, in radix 2, an $\mathcal{S}$*-nonoverlapping* expansion with 53 components may not contain more information than one double-precision FP number; it suffices to put one bit of information into every component.

When Priest first started developing the FP expansion arithmetic, he considered that all the computations were done in faithful FP arithmetic (see [17]), since round-to-nearest rounding mode was not so common. More recently, a slightly stronger sense of nonoverlapping was introduced in 2001 by Hida, Li and Bailey [6]:

Definition 2.5. An expansion u_0, u_1, ..., u_{n-1} is $\mathcal{B}$*-nonoverlapping* (that is, nonoverlapping according to Bailey's definition [6]) if for all $0 < i < n$, we have $|u_i| \leq \frac{1}{2}\,\mathrm{ulp}(u_{i-1})$.

Remark 2.6. Note that for $\mathcal{P}$*-nonoverlapping* expansions we have $|u_i| \leq \frac{2^p-1}{2^p}\,\mathrm{ulp}(u_{i-1})$ and for $\mathcal{S}$*-nonoverlapping* expansions $|u_i| \leq \frac{2^p-1}{2^p}\,\mathrm{uls}(u_{i-1})$.

Even though we presented here three different types of nonoverlapping, in what follows we will focus only on the $\mathcal{P}$ and $\mathcal{B}$*-nonoverlapping* expansions, since in general they provide more precision per given number of terms of a FP expansion.

2.1 Error free transforms

The majority of algorithms performing arithmetic operations on expansions are based on the so-called *error-free transforms* (EFT) (such as the algorithms

2Sum, Fast2Sum, Dekker's product and 2MultFMA presented for instance in [13]), that make it possible to compute both the result and the rounding error of a FP addition or multiplication. This implies that in general, each such *error-free transform* applied to two FP numbers, still returns two FP numbers. Although these algorithms use native precision operations only, they keep track of all accumulated rounding errors, ensuring that no information is lost.

We present here the two algorithms that we use as basic bricks for our work. The algorithm *2Sum* (Algorithm 1) computes the exact sum of two FP numbers a and b and return the result under the form $s+e$, where s is the result rounded to nearest and e is the rounding error. This algorithm requires only 6 native precision FP operations (*flops*), which it was proven to be optimal in [11], if we have no information on the ordering of a and b.

Algorithm 1 2Sum (a, b).

$s \leftarrow \mathrm{RN}(a + b)$
// RN stands for performing the operation in rounding to nearest mode.
$t \leftarrow \mathrm{RN}(s - b)$
$e \leftarrow \mathrm{RN}(\mathrm{RN}(a - t) + \mathrm{RN}(b - \mathrm{RN}(s - t)))$
return (s, e) such that $s = \mathrm{RN}(a + b)$ and $s + e = a + b$

There exists a similar algorithm, that performs the same addition using only 3 native precision FP operations. This one is called *Fast2Sum* [13] and it requires the exponent of a to be larger than or equal to that of b. This condition might be difficult to check, but of course, if $|a| \geq |b|$, it will be satisfied.

For multiplying two FP numbers there exist two algorithms: Dekker's product and 2MultFMA. These algorithms compute the product of two FP numbers a and b and returns the exact result as p, the result rounded to nearest plus e, the rounding error. The first one requires 17 *flops* to achieve the final result. The most expensive part of the algorithm is the computation of the error $e = a \cdot b - p$, but if a *fused-multiply-add* (FMA [13]) instruction, that takes only one *flop*, is available, this value is easily computed. This gives the algorithm 2MultFMA (Algorithm 2) that takes only 2 *flops*. This algorithm works providing that the product $a \cdot b$ does not overflow and $e_a + e_b \geq e_{min} + p - 1$, where e_a and e_b are the exponents of a and b and e_{min} is the minimum representable exponent. If the second condition is not satisfied, the product may not be representable as the exact sum of two FP numbers (e would be below the underflow threshold).

Algorithm 2 2MultFMA (a, b).

$p \leftarrow \mathrm{RN}(a \cdot b)$
// RN stands for performing the operation in rounding to nearest mode.
$e \leftarrow \mathtt{fma}(a, b, -p)$
return (p, e) such that $p = \mathrm{RN}(a \cdot b)$ and $p + e = a \cdot b$

Correctness proofs of these two algorithms can be found in [13, Chap. 4].

These EFT can be extended to work on several inputs by chaining, resulting in the so-called distillation algorithms [18]. From these we make use of the

VecSum algorithm [19, 15], presented in Fig. 7 and Algoithm 3, which is simply a chain of *2Sum* that performs an EFT on n FP numbers.

Algorithm 3 VecSum $(x_0, \ldots x_{n-1})$.

Input: $x_0, \ldots, x_{n-1}$ FP numbers.
Output: $e_0 + \ldots + e_{n-1} = x_0 + \ldots + x_{n-1}$.

$s_{n-1} \leftarrow x_{n-1}$
for $i \leftarrow n-2$ to 0 **do**
  $(s_i, e_{i+1}) \leftarrow 2Sum(x_i, s_{i+1})$
end for
$e_0 \leftarrow s_0$
return $e_0, \ldots, e_{n-1}$

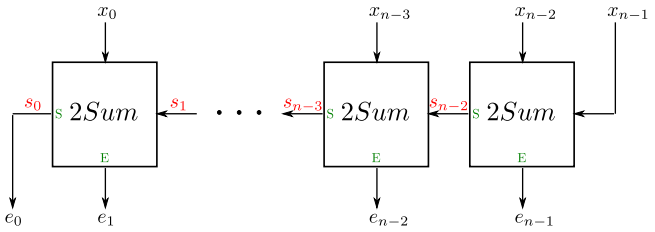

Figure 2: VecSum with n terms. Each *2Sum* box performs Algorithm 1, the sum is outputted to the left and the error downwards.

2.2 Addition and multiplication algorithms for expansions

In general, an algorithm that performs addition of two expansions a and b with n and m terms, respectively, will return a FP expansion with at most $n + m$ terms. Similarly, for multiplication, the product is a FP expansion with at most $2nm$ terms [16]. So-called normalization algorithms are used to render the result nonoverlapping, and this also implies a potential reduction in the number of terms.

Many variants of algorithms that compute the sum and the product of two FP expansions have been presented in the literature [16, 19, 6, 18]. In this article we only briefly present the algorithms that we used in our actual computer implementation. These are generalizations of Bailey's algorithms for DD and QD [6] and were first presented in [10]. Although we have implemented fully customized versions, we give here only the *"k input - k output"* variants of our algorithms.

Algorithm 4 Algorithm of addition of FP expansions with k terms.

Input: FP expansions $a = a_0 + \ldots + a_{k-1}$; $b = b_0 + \ldots + b_{k-1}$.
Output: FP expansion $r = r_0 + \ldots + r_{k-1}$.

1: $(r_0, e_0) \leftarrow \mathit{2Sum}(a_0, b_0)$
2: **for** $n \leftarrow 1$ to $k-1$ **do**
3: $\quad (s_n, e_n) \leftarrow \mathit{2Sum}(a_n, b_n)$
4: $\quad r_n, e_0, \ldots, e_{n-1} \leftarrow \mathit{VecSum}(s_n, e_0, \ldots, e_{n-1})$
5: **end for**
6: $r_k \leftarrow 0$
7: **for** $i \leftarrow 0$ to $k-1$ **do**
8: $\quad r_k \leftarrow r_k + e_i$
9: **end for**
10: $r[0 : k-1] \leftarrow \mathit{Renormalize}(r[0 : k])$
11: **return** FP expansion $r = r_0 + \ldots + r_{k-1}$.

The addition algorithm presented in Algorithm 4 and Fig. 3 is based on a chain of *2Sum* EFTs. Given two FP expansions a and b, each with k terms, it produces the k most significant FP components of the sum $r = a + b$. In the renormalization step (see line 10 of Algorithm 4), we use an extra error correction term, so we perform our "error free transformation scheme" $k+1$ times. The last term is computed using simple round-to-nearest FP addition because the resulted errors are not accounted for anymore. At step $n = 0$ we compute the exact sum $a_0 + b_0 = r_0 + e_0$. Since $|e_0| \leq \frac{1}{2}\,\mathrm{ulp}(r_0)$, we use the following intuition: let $\varepsilon = \frac{1}{2}\,\mathrm{ulp}(r_0)$, then roughly speaking, if r_0 is of order of $\mathcal{O}(\Lambda)$, then e_0 is of order $\mathcal{O}(\varepsilon\Lambda)$. At each step $n = 1, \ldots, k$ we compute the exact result of $a_n + b_n = s_n + e_n$, where s_n and e_n are of order $\mathcal{O}(\varepsilon^n\Lambda)$ and $\mathcal{O}(\varepsilon^{n+1}\Lambda)$, respectively. From previous steps we have already obtained n error terms of order $\mathcal{O}(\varepsilon^n\Lambda)$ that we add together with s_n to obtain the term r_n of order $\mathcal{O}(\varepsilon^n\Lambda)$ before the renormalization step. This addition is done using *VecSum* (Algorithm 3). The $(k+1)$-th component r_k is obtained by a simple summation of the previously obtained terms of order $\mathcal{O}(\varepsilon^k\Lambda)$.

Note that, in this setting, subtraction is simpler than for the multiple digit case, and can be performed simply by negating the FP terms in b.

In Fig. 4 and Algorithm 5 we present the multiplication algorithm. As in the case of addition, we consider two FP expansions a and b, each with k terms and we compute the k most significant FP components of the product $r = a \cdot b$. We use the same type of intuition, so for the product $(p, e) = \mathit{2ProdFMA}(a_i, b_j)$, p is of order $\mathcal{O}(\varepsilon^n\Lambda)$ and e of order $\mathcal{O}(\varepsilon^{n+1}\Lambda)$, where $n = i + j$, and we consider only the terms for which $0 \leq n \leq k$. This implies that for each n we have $n+1$ products to compute (see line 4 of Algorithm 5). Next, we need to add all terms of the same order of magnitude. By induction, it can be easily shown that beside the $n+1$ products, we also have n^2 terms resulting from the previous iteration. This addition is performed using *VecSum* (Algorithm 3) to obtain r_n in line 6. The remaining terms are concatenated with the errors from the $n+1$ products, and the entire $e_0, \ldots, e_{(n+1)^2-1}$ array is used in the next iteration. The $(k+1)$-st component r_k is obtained by simple summation of all remaining errors with the simple products of order $\mathcal{O}(\varepsilon^k\Lambda)$. *2ProdFMA* is not needed in the last step

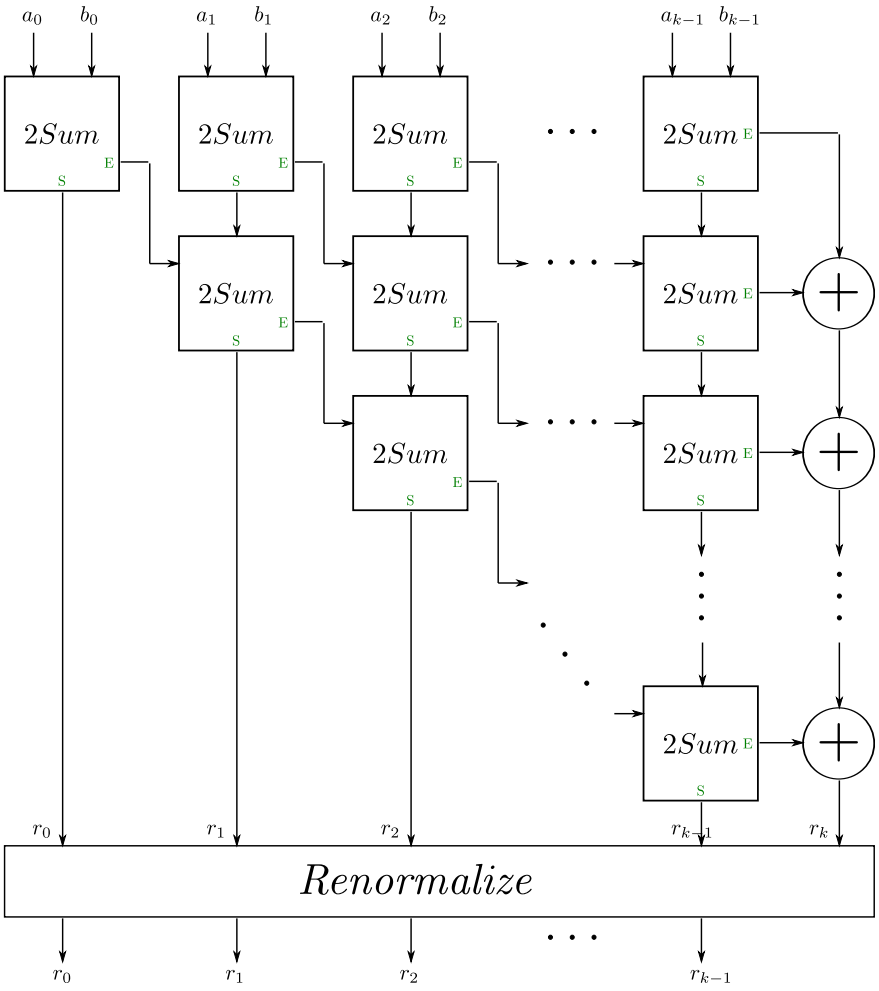

Figure 3: Addition of FP expansions with k terms. Each *2Sum* box performs Algorithm 1, the sum is outputted downwards and the error to the right; the circled + sign represents standard round-to-nearest FP addition.

Algorithm 5 Algorithm of multiplication of FP expansions with k terms.

Input: FP expansions $a = a_0 + \ldots + a_{k-1}$; $b = b_0 + \ldots + b_{k-1}$.
Output: FP expansion $r = r_0 + \ldots + r_{k-1}$.

1: $(r_0, e_0) \leftarrow$ *2ProdFMA*(a_0, b_0)
2: **for** $n \leftarrow 1$ to $k-1$ **do**
3: **for** $i \leftarrow 0$ to n **do**
4: $(p_i, \hat{e}_i) \leftarrow$ *2ProdFMA*(a_i, b_{n-i})
5: **end for**
6: $r_n, e[0 : n^2 + n - 1] \leftarrow$ *VecSum*$(p[0 : n], e[0 : n^2 - 1])$
7: $e[0 : (n+1)^2 - 1] \leftarrow e[0 : n^2 + n - 1], \hat{e}[0 : n]$
8: **end for**
9: **for** $i \leftarrow 1$ to $k-1$ **do**
10: $r_k \leftarrow r_k + a_i \cdot b_{k-i}$
11: **end for**
12: **for** $i \leftarrow 0$ to $k^2 - 1$ **do**
13: $r_k \leftarrow r_k + e_i$
14: **end for**
15: $r[0 : k-1] \leftarrow$ *Renormalize*$(r[0 : k])$
16: **return** FP expansion $r = r_0 + \ldots + r_{k-1}$.

since the errors are not reused. For the addition and multiplication algorithms presented in this section, we intend to provide an effective error analysis in a future work. An important step for this goal is to provide a thorough proof for the renormalization, which is used at the end of each of these two algorithms. So, in what follows we focus on our new algorithm for renormalization of FP expansions.

3 Renormalization algorithm for expansions

While several renormalization algorithms have been proposed in literature, Priest [16] algorithm seems to be the only one provided with a complete proof of correctness. But that algorithm has many conditional branches, which make it slow in practice, and has a worst case FP operation count of: $R(n) = 20(n-1)$, for an input FP expansion with n-terms.

In an attempt to overcome the branching problem we developed a new algorithm (see Algorithm 6), for which we provide a full correctness proof.

First, we need to define the concept of FP numbers that overlap by at most d digits.

Definition 3.1. Consider an array of FP numbers: $x_0, x_1, \ldots, x_{n-1}$. According to Priest's [16] definition, they overlap by at most d digits ($0 \leq d < p$) if and only if $\forall i, 0 \leq i \leq n-2, \exists k_i, \delta_i$ such that:

$$2^{k_i} \leq |x_i| < 2^{k_i+1}, \tag{1}$$

$$2^{k_i-\delta_i} \leq |x_{i+1}| \leq 2^{k_i-\delta_i+1}, \tag{2}$$

$$\delta_i \geq p - d, \tag{3}$$

$$\delta_i + \delta_{i+1} \geq p - z_{i-1}, \tag{4}$$

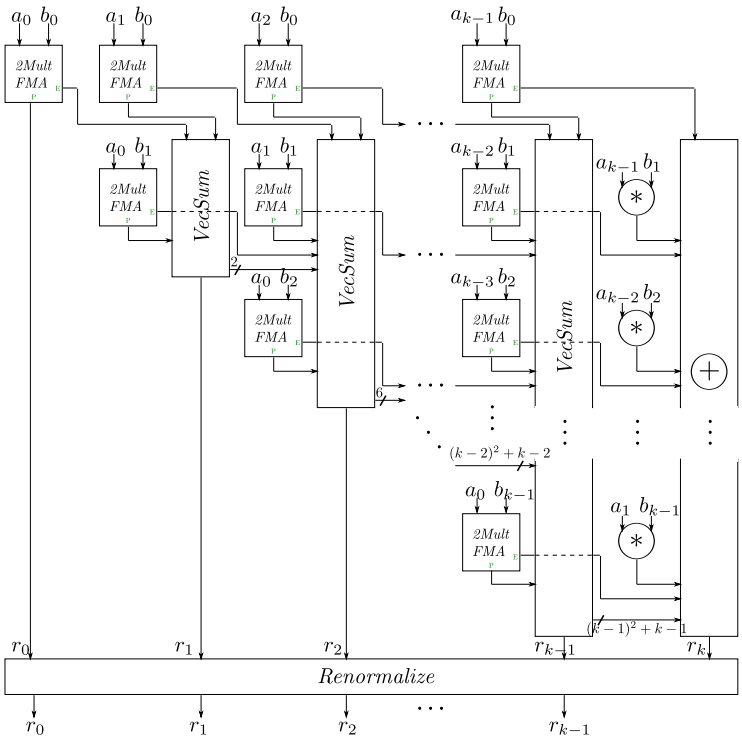

Figure 4: Multiplication of FP expansions with k terms. Each *2MultFMA* box performs Algorithm 2, the product is outputted downwards and the error to the right; the *VecSum* box performs Algorithm 3, in which the most significant component of the sum is outputted downwards; the circled $+$ and $*$ signs represent standard round-to-nearest FP addition and multiplication.

where z_{i-1} is the number of trailing zeros at the end of x_{i-1} and for $i = 0$, $z_{-1} := 0$.

Proposition 3.2. *Let $x_0, x_1, \ldots, x_{n-1}$ be an array of FP numbers which overlap by at most d digits ($0 \leq d < p$). The following properties hold:*

$$|x_{j+1}| < 2^d \operatorname{ulp}(x_j), \tag{5}$$

$$\operatorname{ulp}(x_{j+1}) \leq 2^{d-p} \operatorname{ulp}(x_j), \tag{6}$$

$$|x_{j+2} + x_{j+1}| \leq (2^d + 2^{2d-p}) \operatorname{ulp}(x_j). \tag{7}$$

Proof. We have $\operatorname{ulp}(x_j) = 2^{k_j - p + 1}$ and from (3) we get $|x_{j+1}| < 2^{k_j - \delta_j + 1} < 2^{p - \delta_j} \operatorname{ulp}(x_j) < 2^d \operatorname{ulp}(x_j)$. This proves that property (5) holds for all $0 \leq j < n - 1$.

By applying (3) we get $\operatorname{ulp}(x_{j+1}) = 2^{k_j - \delta_j - p + 1} \leq 2^{d-p} \operatorname{ulp}(x_j)$, which proves that (6) holds for all $0 \leq j < n - 1$.

We have $|x_{j+1}| \leq 2^d \operatorname{ulp}(x_j)$ and $|x_{j+2}| \leq 2^d \operatorname{ulp}(x_{j+1}) \leq 2^{2d-p} \operatorname{ulp}(x_j)$ from which (7) immediately follows. □

The renormalization algorithm (see Algorithm 6 and Fig. 5) is based on different layers of chained *2Sum*. For the sake of simplicity, these are grouped in simpler layers based on *VecSum*. We will prove that our algorithm returns a $\mathcal{P}$*-nonoverlapping* sequence.

Proposition 3.3. *Consider an array $x_0, x_1, \ldots, x_{n-1}$, consisting of FP numbers that overlap by at most $d \leq p-2$ digits and let m an input parameter, with $1 \leq m \leq n-1$. Provided that no underflow / overflow occurs during the calculations, the renormalization Algorithm 6 returns a "truncation" to m terms of a* $\mathcal{P}$-nonoverlapping *FP expansion $f = f_0 + \ldots + f_{n-1}$ such that $x_0 + \ldots + x_{n-1} = f$.*

Algorithm 6 Renormalization algorithm

Input: FP expansion $x = x_0 + \ldots + x_{n-1}$ consisting of FP numbers that overlap by at most d digits, with $d \leq p - 2$; m length of output FP expansion.

Output: FP expansion $f = f_0 + \ldots + f_{m-1}$ with $f_{i+1} \leq (\frac{1}{2} + 2^{-p+2} + 2^{-p}) \operatorname{ulp}(f_i)$, for all $0 \leq i < m - 1$.

1: $e[0 : n-1] \leftarrow VecSum(x[0 : n-1])$
2: $f^{(0)}[0 : m] \leftarrow VecSumErrBranch(e[0 : n-1], m+1)$
3: **for** $i \leftarrow 0$ to $m - 2$ **do**
4: $\quad f^{(i+1)}[i : m] \leftarrow VecSumErr(f^{(i)}[i : m])$
5: **end for**
6: **return** FP expansion $f = f_0^{(1)} + \ldots + f_{m-2}^{(m-1)} + f_{m-1}^{(m-1)}$.

In order to prove this proposition, in what follows, we prove first several intermediate properties. The notations used in the proof ($s_i, e_i, \varepsilon_i, f_i, \rho_i$ and g_i) are defined on the schematic drawings of the algorithms discussed. We also make the important remark that at each step we prove that all the *2Sum* blocks can be replaced by *Fast2Sum* ones, but for simplicity of the proof we chose to present first the *2Sum* version.

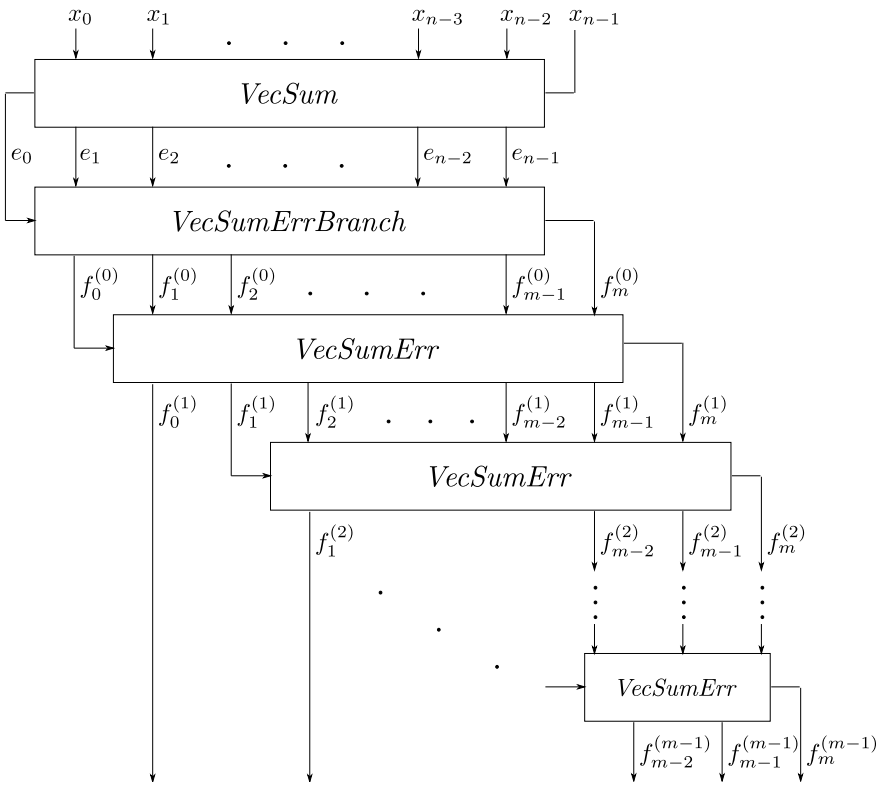

Figure 5: Renormalization of FP expansions with n terms. The *VecSum* box performs Algorithm 3, the *VecSum-ErrBranch* box, Algorithm 7 and the *VecSumErr* box, Algorithm 8.

First level (line 1, Algorithm 6)

It consists in applying Algorithm 3, *VecSum* (see also Fig. 7) on the input array, from where we obtain the array $e = (e_0, e_1, \ldots, e_{n-1})$.

Proposition 3.4. *After applying the* VecSum *algorithm, the output array* $e = (e_0, e_1, \ldots, e_{n-1})$ *is* $\mathcal{S}$-nonoverlapping *and may contain interleaving zeros.*

Proof. Observe first that since $s_i = \mathrm{RN}(x_i + s_{i+1})$, s_i is closer to $x_i + s_{i+1}$ than x_i. Hence $|(x_i + s_{i+1}) - s_i| \leq |(x_i + s_{i+1}) - x_i|$, and so $|e_{i+1}| \leq |s_{i+1}|$.

Similarly, s_i is closer to $x_i + s_{i+1}$ than s_{i+1}, so $|e_{i+1}| \leq |x_i|$. From (5) we get:

$$\begin{aligned} |x_{j+1}| \quad + \quad & |x_{j+2}| + \cdots \leq \\ \leq \quad & [2^d + 2^{2d-p} + 2^{3d-2p} + 2^{4d-3p} + \ldots] \operatorname{ulp}(x_j) \\ \leq \quad & 2^d \frac{2^p}{2^p - 1} \operatorname{ulp}(x_j). \end{aligned} \tag{8}$$

We know that $s_{j+1} = \mathrm{RN}(x_{j+1} + \mathrm{RN}(\cdots + x_{n-1}))$ and by using a property given by Jeannerod and Rump in [8] we get:

$$\begin{aligned} |s_{j+1} \quad - \quad & (x_{j+1} + \ldots + x_{n-1})| \\ \leq \quad & (n - j - 2) \cdot 2^{-p} \cdot (|x_{j+1}| + \cdots + |x_{n-1}|) \,. \end{aligned} \tag{9}$$

From (8) and (9) we have:

$$|s_{j+1}| \leq 2^d \frac{2^p}{2^p - 1} (1 + (n - j - 2)2^{-p}) \operatorname{ulp}(x_j).$$

It is easily seen that

$$2^d \frac{2^p}{2^p - 1} (1 + (n - j - 2)2^{-p}) \leq 2^{p-1}, \tag{10}$$

is satisfied for $p \geq 4$ and $n \leq 16$, for $p \geq 5$ and $n \leq 32$ and so on. This includes all practical cases, when $d \leq p - 2$, so that $\operatorname{ulp}(s_{j+1}) < \operatorname{ulp}(x_j)$.

Therefore x_j and s_{j+1} are multiples of $\operatorname{ulp}(s_{j+1})$, thus $x_j + s_{j+1}$ is multiple of $\operatorname{ulp}(s_{j+1})$, hence $\mathrm{RN}(x_j + s_{j+1})$ is multiple of $\operatorname{ulp}(s_{j+1})$ and $|e_{j+1}| = |x_j + s_{j+1} - \mathrm{RN}(x_j + s_{j+1})|$ is multiple of $\operatorname{ulp}(s_{j+1})$.

Also, by definition of *2Sum*, we have $|e_{j+2}| \leq \frac{1}{2} \operatorname{ulp}(s_{j+1})$. Now, we are able to compare $|e_{j+1}|$ and $|e_{j+2}|$. Since $|e_{j+1}|$ is a multiple of $\operatorname{ulp}(s_{j+1})$, either $e_{j+1} = 0$ or e_{j+1} is larger than $2\,|e_{j+2}|$ and multiple of 2^k, such that $2^k > |e_{j+2}|$. This implies that the array $e = (e_0, e_1, \ldots, e_{n-1})$ is $\mathcal{S}$-*nonoverlapping* and may have interleaving zeros. □

Remark 3.5. Since we have $|s_{j+1}| \leq 2^{p-1} \operatorname{ulp}(x_j)$, for $d \leq p - 2$ and $p \geq 4$ for n up to 16 and also $\operatorname{ulp}(x_j) \leq 2^{1-p} |x_j|$ we can deduce that $|s_{j+1}| \leq |x_j|$. Hence, at this level we can use instead of *2Sum* basic blocks the *Fast2Sum* ones.

Second level (line 2, Algorithm 6)

It is applied on the array e obtained previously. This is also a chain of *2Sum*, but instead of starting from the least significant, we start from the most significant

component. Also, instead of propagating the sum we propagate the error. If however, the error after a *2Sum* block is zero, then we propagate the sum (this is shown in Figure 6). In what follows we will refer to this algorithm by *VecSumErrBranch* (see Algorithm 7). The following property holds:

Proposition 3.6. *Let an input array $e = (e_0, \ldots, e_{n-1})$ of S-non-overlapping terms and $1 \leq m \leq n$ the required number of output terms. After appling* VecSumErrBranch*, the output array of $f = (f_0, \ldots, f_{m-1})$, with $0 \leq m \leq n-1$ satisfies $|f_{i+1}| \leq \text{ulp}(f_i)$ for all $0 \leq i < m-1$.*

Proof. The case when the array e contains at most 2 elements is trivial. Consider now at least 3 elements. By definition of 2Sum, we have $|\varepsilon_1| \leq \frac{1}{2}\,\text{ulp}(f_0)$ and by definition of $\mathcal{S}$*-nonoverlapping*,

$$\begin{aligned} e_0 &= E_0 \cdot 2^{k_0} \text{ with } |e_1| < 2^{k_0}, \\ e_1 &= E_1 \cdot 2^{k_1} \text{ with } |e_2| < 2^{k_1}. \end{aligned}$$

Hence, f_0 and ε_1 are both multiples of 2^{k_1}. Two possible cases occur:

(i) $\varepsilon_1 = 0$. If we choose to propagate directly $\varepsilon_1 = 0$, then $f_1 = e_2$ and $\varepsilon_2 = 0$. This implies by induction that $f_i = e_{i+1}, \forall i \geq 1$. So, directly propagating the error poses a problem, since the whole remaining chain of 2Sum is executed without any change. So, as shown in Algorithm 7, line 11, when $\varepsilon_{i+1} = 0$ we propagate the sum f_j.

(ii) $\varepsilon_1 \neq 0$. Then $|e_2| < |\varepsilon_1|$ and $|\varepsilon_1 + e_2| < 2\,|\varepsilon_1|$, from where we get $|f_1| = |\text{RN}(\varepsilon_1 + e_2)| \leq 2\,|\varepsilon_1| \leq \text{ulp}(f_0)$.

We prove by induction the following statement: at step $i > 0$ of the loop in Algorithm 7, both f_{j-1} and ε_i are multiples of 2^{k_i} with $|e_{i+1}| < 2^{k_i}$. We proved above that $i = 1$ holds. Suppose now it holds for i and prove for $i+1$. Since f_{j-1} and ε_i are multiples of 2^{k_i} with $|e_{i+1}| < 2^{k_i}$ and $e_{i+1} = E_{i+1} \cdot 2^{k_{i+1}}$ with $|e_{i+2}| < 2^{k_{i+1}}$ (by definition of $\mathcal{S}$*-nonoverlapping*), it follows that both f_j and ε_{i+1} are multiples of $2^{k_{i+1}}$ (by definition of *2Sum*).

Finally, we prove the relation between f_j and f_{j-1}. If $\varepsilon_{i+1} = 0$, we propagate f_j, i.e. $\varepsilon_{i+1} = f_j$. Otherwise $|e_{i+1}| < |\varepsilon_i|$, so $|e_{i+1} + \varepsilon_i| < 2\,|\varepsilon_i|$ and finally $|f_j| = |\text{RN}(e_{i+1} + \varepsilon_i)| \leq 2\,|\varepsilon_i| \leq \text{ulp}(f_{j-1})$. □

Remark 3.7. For this algorithm we can also use the *Fast2Sum* algorithm instead of *2Sum*. We already showed that either $|e_{i+1}| < |\varepsilon_i|$, or $\varepsilon_i = 0$, in which case we replace $\varepsilon_i = f_{j-1}$, which is a multiple of 2^{k_i} with $|e_{i+1}| < 2^{k_i}$.

Third level and further (lines 3-5, Algorithm 6)

On the previously obtained array we apply a similar chain of *2Sum*, starting from the most significant component and propagating the error. The advantage in these subsequent levels is that no conditional branching is needed anymore (see Algorithm 8).

We prove the following property:

Proposition 3.8. *After applying Algorithm 8,* VecSumErr *on an input array $f = (f_0, \ldots, f_m)$, with $|f_{i+1}| \leq \text{ulp}(f_i)$, for all $0 \leq i \leq m-1$, the output array $g = (g_0, \ldots, g_m)$ satisfies $|g_1| \leq \left(\frac{1}{2} + 2^{-p+2}\right)\text{ulp}(g_0)$ and $|g_{i+1}| \leq \text{ulp}(g_i)$, for $0 < i \leq m-1$.*

Algorithm 7 Second level of the renormalization algorithm - *VecSumErrBranch*

Input: $\mathcal{S}$-*nonoverlapping* FP expansion $e = e_0 + \ldots + e_{n-1}$; m length of the output expansion.

Output: FP expansion $f = f_0 + \ldots + f_{m-1}$ with $f_{j+1} \leq \text{ulp}(f_j)$, $0 \leq j < m-1$.

$j \leftarrow 0$
$\varepsilon_0 = e_0$
for $i \leftarrow 0$ to $n-2$ **do**
  $(f_j, \varepsilon_{i+1}) \leftarrow$ *2Sum*(ε_i, e_{i+1})
  if $\varepsilon_{i+1} \neq 0$ **then**
    if $j \geq m-1$ **then**
      return FP expansion $f = f_0 + \ldots + f_{m-1}$. //enough output terms
    end if
    $j \leftarrow j+1$
  else
    $\varepsilon_{i+1} \leftarrow f_j$
  end if
end for
if $\varepsilon_{n-1} \neq 0$ **and** $j < m$ **then**
  $f_j \leftarrow \varepsilon_{n-1}$
end if
return FP expansion $f = f_0 + \ldots + f_{m-1}$.

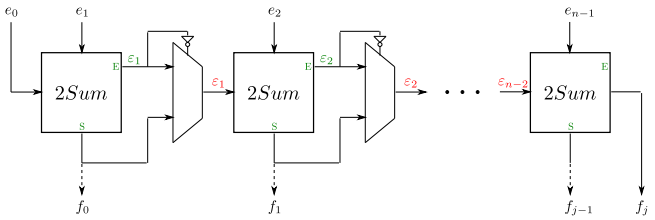

Figure 6: *VecSumErrBranch* with n terms. Each *2Sum* box performs Algorithm 1, the sum is outputted downwards and the error to the right. If the error is zero, the sum is propagated to the right, otherwise the error is propagated and the sum is outputted.

Algorithm 8 Third level of the renormalization algorithm - *VecSumErr*

Input: FP expansion $f = f_0 + \ldots + f_m$ with $|f_{i+1}| \leq \text{ulp}(f_i)$, for all $0 \leq i \leq m-1$.

Output: FP expansion $g = g_0 + \ldots + g_m$ with $|g_1| \leq \left(\frac{1}{2} + 2^{-p+2}\right) \text{ulp}(g_0)$ and $|g_{i+1}| \leq \text{ulp}(g_i)$, for $0 < i \leq m-1$.

1: $\rho_0 = f_0$
2: **for** $i \leftarrow 0$ to $m-1$ **do**
3: $\quad (g_i, \rho_{i+1}) \leftarrow \mathit{2Sum}(\rho_i, f_{i+1})$
4: **end for**
5: $g_m \leftarrow \varepsilon_m$
6: **return** FP expansion $g = g_0 + \ldots + g_m$.

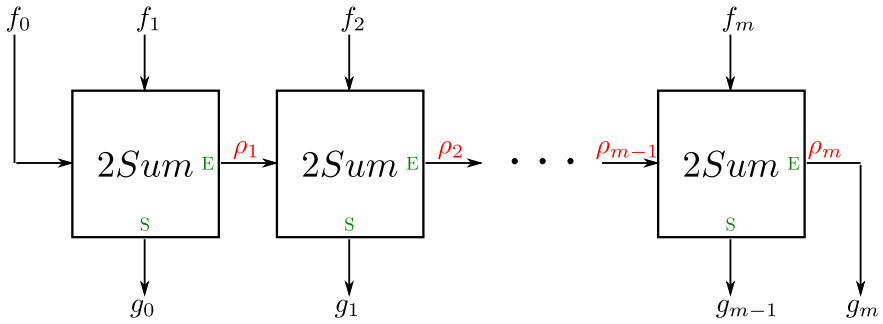

Figure 7: *VecSumErr* with $m+1$ terms. Each *2Sum* box performs Algorithm 1, the sum is outputted downwards and the error to the right.

Proof. Since $|f_1| \leq \mathrm{ulp}(f_0)$ and $g_0 = \mathrm{RN}(f_0 + f_1)$ we have:

$$\frac{1}{2}\,\mathrm{ulp}(f_0) \leq \mathrm{ulp}(g_0) \leq 2\,\mathrm{ulp}(f_0),$$

and

$$|\rho_1| \leq \frac{1}{2}\,\mathrm{ulp}(g_0).$$

We also have:

$$\begin{aligned} |f_1| &\leq \mathrm{ulp}(f_0), \text{ which implies } \mathrm{ulp}(f_1) \leq 2^{-p+1}\,\mathrm{ulp}(f_0), \\ |f_2| &\leq \mathrm{ulp}(f_1) \leq 2^{-p+2}\,\mathrm{ulp}(g_0). \end{aligned}$$

Hence:

$$|\rho_1 + f_2| \leq \left(\frac{1}{2} + 2^{-p+2}\right)\mathrm{ulp}(g_0).$$

Since $\left(\frac{1}{2} + 2^{-p+2}\right)\mathrm{ulp}(g_0)$ is a FP number, we also have:

$$|g_1| = |\mathrm{RN}(\rho_1 + f_2)| \leq \left(\frac{1}{2} + 2^{-p+2}\right)\mathrm{ulp}(g_0).$$

This bound is very close to $\frac{1}{2}\,\mathrm{ulp}(g_0)$ and it seems that in most practical cases, one actually has $|g_1| \leq \frac{1}{2}\,\mathrm{ulp}(g_0)$. This implies that g_0 and g_1 are "almost" $\mathcal{B}$-*nonoverlapping* and a simple computation shows that they are $\mathcal{P}$-*nonoverlapping* as soon as $p \geq 3$, which occurs in all practical cases.

As we iterate further, we get:

$$\begin{aligned} |\rho_{i+1}| &\leq \frac{1}{2}\,\mathrm{ulp}(g_i), \\ |f_{i+1}| &\leq \mathrm{ulp}(f_i), \text{ which implies } \mathrm{ulp}(f_{i+1}) \leq 2^{-p+1}\,\mathrm{ulp}(f_i). \end{aligned}$$

We know that ρ_i is multiple of $\mathrm{ulp}(f_i)$ and from this we can derive two cases: *(i)* $\rho_i = 0$, and as a consequence $\forall j \geq i, g_j = f_{j+1}$ and $g_m = 0$. In the second case *(ii)* we get:

$$\begin{aligned} |f_{i+1}| \leq |\rho_i| &\leq \frac{1}{2}\,\mathrm{ulp}(g_{i-1}), \\ |f_{i+1} + \rho_i| &\leq \mathrm{ulp}(g_{i-1}), \\ |g_i| = |\mathrm{RN}(f_{i+1} + \rho_i)| &\leq \mathrm{ulp}(g_{i-1}). \end{aligned}$$

□

Remark 3.9. For Algorithm 8 we can also use the faster algorithm, $Fast2Sum(\rho_i, f_{i+1})$, because we either have $\rho_i = 0$ or $|f_{i+1}| \leq |\rho_i|$.

The above proposition clearly shows that while we obtain a *nonoverlapping* condition for the first two elements of the resulting array g, for the others we don't prove to strengthen the existing bound $|g_{i+1}| \leq \mathrm{ulp}(g_i)$. There is an advantage however: if zeros appear in the summation process, they are pushed at the end; we don't use any branching. This suggest to continue applying a subsequent level of the same algorithm on the remaining elements, say $g_1, \ldots, g_m$. This is the idea of applying $m-1$ levels of *VecSumErr* in lines 3-5, Algorithm 6. We are now able to prove Prop. 3.3.

Proof. (of Prop. 3.3) Consider $m \geq 2$, otherwise the output reduces to only one term. Then, the loop in lines 3-5, Algorithm 6 is executed at least once. From Prop. 3.4, 3.6 and 3.8 we deduce that $\left|f_1^{(1)}\right| \leq \left(\frac{1}{2} + 2^{-p+2}\right) \operatorname{ulp}(f_0^{(1)})$ and $\left|f_{i+1}^{(1)}\right| \leq \operatorname{ulp}(f_i^{(1)})$, for $i > 0$. If $m = 2$ then $f_0^{(1)}, f_1^{(1)}$ are outputted and the proposition is proven. Otherwise, $f_0^{(1)}$ is kept unchanged and another *VecSumErr* is applied to remaining $f_1^{(1)}, \ldots, f_m^{(1)}$. We have:

$$\begin{aligned}
\left|f_1^{(1)}\right| &\leq \left(\frac{1}{2} + 2^{-p+2}\right) \operatorname{ulp}(f_0^{(1)}), \\
\left|f_2^{(1)}\right| &\leq \operatorname{ulp}(f_1^{(1)}) \leq 2^{-p+1} \left(\frac{1}{2} + 2^{-p+2}\right) \operatorname{ulp}(f_0^{(1)}), \\
&\leq 2^{-p+1} \operatorname{ulp}(f_0^{(1)}).
\end{aligned}$$

Hence,

$$\begin{aligned}
\left|f_1^{(2)}\right| &= \left|\mathrm{RN}(f_1^{(1)} + f_2^{(1)})\right|, \\
&\leq \left(\frac{1}{2} + 2^{-p+2} + 2^{-p+1}\right) \operatorname{ulp}(f_0^{(1)}).
\end{aligned}$$

Similarly,

$$\begin{aligned}
\left|f_2^{(2)}\right| &\leq \left(\frac{1}{2} + 2^{-p+2}\right) \operatorname{ulp}(f_1^{(2)}), \\
\left|f_3^{(2)}\right| &\leq \operatorname{ulp}(f_2^{(2)}) \leq 2^{-p+1} \left(\frac{1}{2} + 2^{-p+2}\right) \operatorname{ulp}(f_1^{(2)}), \\
&\leq 2^{-p+1} \operatorname{ulp}(f_1^{(2)}).
\end{aligned}$$

So, $f_0^{(1)}, f_1^{(2)}, f_2^{(2)}$ are *nonoverlapping*. It easily follows by induction that after $m-1$ loop iterations the output $f_0^{(1)}, \ldots, f_{m-2}^{(m-1)}, f_{m-1}^{(m-1)}$ is a $\mathcal{P}$*-nonoverlapping* expansion. Finally, when all $n-1$ terms are considered, after at most $n-1$ loop iterations we have: $x_0 + \ldots + x_{n-1} = f_0^{(1)} + \ldots + f_{n-2}^{(n-1)} + f_{n-1}^{(n-1)}$. □

Remark 3.10. In the worst case, Algorithm 6 performs $n-1$ *Fast2Sum* calls in the first level and $n-2$ *Fast2Sum* calls plus $n-1$ comparisons in the second one. During the following $m-1$ levels we perform $m-i$ *Fast2Sum* calls, with $0 \leq i < m-2$. This accounts for a total of $R_{new}(n,m) = 7n + \frac{3}{2}m^2 + \frac{3}{2}m - 13$ FP operations.

In table 1 we give some effective values of the worst case FP operation count for Priest's renormalization algorithm [16] and Algorithm 6. It can be seen that for $n \leq 7$ our algorithm performs better or the same. Even though from values of $n > 7$ Algorithm 6 performs worse in terms of operation count than Priest's one, in practice, the last $m-1$ levels will take advantage of the computers pipeline, because we do not need branching conditions anymore, which makes it faster in practice.

In what follows we denote by *AddRoundE*$(x[0:n-1], y[0:m-1], k)$, an algorithm for expansions addition, which given two ($\mathcal{P}-$ or $\mathcal{B}-$) nonoverlapping expansions, returns the k most significant terms of the exact normalized ($\mathcal{P}-$ or

Table 1: FP operation count for Algorithm 6 vs. Priest's renormalization algorithm [16]. We consider that both algorithms compute $n-1$ terms in the output expansion.

q	2	4	7	8	10	12	16
Alg. 6	10	45	120	151	222	305	507
Priest's alg. [16]	20	60	120	140	180	220	300

$\mathcal{B}-$) nonoverlapping sum. If no request is made on the number of terms to be returned, then we denote simply by $AddE(x[0:n-1], y[0:m-1])$. Similarly, we denote by *MulRoundE*, *MulE*, *SubRoundE*, *SubE*, *DivRoundE*, *RenormalizeE* algorithms for multiplication, subtraction, division and normalization.

4 Reciprocal algorithm

4.1 Algorithms using classical long division on expansions

In reference [16], division is done using the classical long division algorithm (a variation of the algorithm we use for paper-and-pencil calculations), which is recalled in Algorithm 9.

Algorithm 9 Priest's [16] division algorithm. We denote by $f[0:\ldots]$ and expansion f whose number of terms is not known in advance.

Input: FP expansion $a = a_0 + \ldots + a_{n-1}$; $b = b_0 + \ldots + b_{m-1}$; length of output quotient FP expansion d.

Output: FP expansion $q = q_0 + \ldots$ with at most d terms s.t. $\left|\frac{q-a/b}{a/b}\right| < 2^{1-\lfloor (p-4)d/p \rfloor}$.

1: $q_0 = \mathrm{RN}(a_0/b_0)$
2: $r^{(0)}[0:n-1] \leftarrow a[0:n-1]$
3: **for** $i \leftarrow 1$ to $d-1$ **do**
4: $f[0:\ldots] \leftarrow MulE(q_{i-1}, b[0:m-1])$
5: $r^{(i)}[0:\ldots] \leftarrow RenormalizeE(SubE(r^{(i-1)}[0:\ldots], f[0:\ldots]))$
6: $q_i = \mathrm{RN}(r_0^{(i)}/b_0)$
7: **end for**
8: $q[0:\ldots] \leftarrow RenormalizeE(q[0:d-1])$
9: **return** FP expansion $q = q_0 + \ldots$.

Bailey's division algorithm [6] is very similar. For instance, let $a = a_0 + a_1 + a_2 + a_3$ and $b = b_0 + b_1 + b_2 + b_3$ be QD numbers. First, one approximates the quotient $q_0 = a_0/b_0$, then compute the remainder $r = a - q_0 b$ in quad-double. The next correction term is $q_1 = r_0/b_0$. Subsequent terms q_i are obtained by continuing this process. Note that at each step when computing r full quad-double multiplication and subtraction must be performed since most of the bits will be canceled out when computing q_3 and q_4, in Bailey's algorithm. A renormalization step is performed only at the end, on $q_0 + q_1 + q_2 + ...$ in order to ensure non-overlapping. No error bound is given in [6].

Note that in Algorithm 9 [16] a renormalization step is performed after each computation of $r = r - q_i b$. An error bound is given in [16]:

Proposition 4.1. *[16] Consider two $\mathcal{P}$-*nonoverlapping *expansions: $a = a_0 + \ldots + a_{n-1}$ and $b = b_0 + \ldots + b_{m-1}$, Priest division algorithm [16] computes a quotient expansion $q = q_0 + \ldots + q_{d-1}$ s.t.*

$$\left| \frac{q - a/b}{a/b} \right| < 2^{1-\lfloor (p-4)d/p \rfloor}. \tag{11}$$

In Daumas and Finot's paper [3], Priest's division algorithm is improved by using only estimates of the most significant component of the remainder r_0 and storing the less significant components of the remainder and the terms $-q_i b$ unchanged in a set that is managed with a priority queue. While the asymptotic complexity of this algorithm is better, in practical simple cases Priest's algorithm is faster due to the control overhead of the priority queue [3]. The error bound obtained with Daumas' algorithm is (using the same notations as above):

$$\left| \frac{q - a/b}{a/b} \right| < 2^{-d(p-1)} \prod_{i=0}^{d-1} (4i + 6). \tag{12}$$

4.2 Reciprocal of expansions with an adapted Newton-Raphson iteration

The classical Newton-Raphson iteration for computing reciprocals is briefly recalled in what follows [20, 2, 13, Chap. 2]. It is based on the Newton iteration for computing the roots of a given function f, which is:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{13}$$

When x_0 is close to a root α, $f'(\alpha) \neq 0$, the iteration converges quadratically. For computing $1/a$ we look for roots of the function $f(x) = 1/x - a$ which implies using the iteration

$$x_{n+1} = x_n(2 - a x_n). \tag{14}$$

The iteration converges to $1/a$ for all $x_0 \in (0, 2/a)$. However, taking any point in $(0, 2/a)$ as the starting point x_0 would be a poor choice. A much better choice is to choose x_0 equal to a FP number very close to $1/a$. This only requires one FP division. The quadratic convergence of (14) is deduced from $x_{n+1} - \frac{1}{a} = -a(x_n - \frac{1}{a})^2$. This iteration is *self-correcting* because minor errors, like rounding errors, do not modify the limit value.

While iteration (14) is well known, in Algorithm 10 we use an adaptation for computing reciprocals of FP expansions, with truncated operations involving FP expansions. Our algorithm works with both *$\mathcal{B}$- and $\mathcal{P}$-nonoverlapping* FP expansions. For the sake of clarity we consider first the case of *$\mathcal{B}$-nonoverlapping* FP expansions, and then make the necessary adjustments for *$\mathcal{P}$-nonoverlapping* expansions in Proposition 4.4.

Algorithm 10 Truncated Newton iteration based algorithm for reciprocal of an FP expansion.

Input: FP expansion $a = a_0 + \ldots + a_{2^k-1}$; length of output FP expansion 2^q.

Output: FP expansion $x = x_0 + \ldots + x_{2^q-1}$ s.t. $\left|x - \frac{1}{a}\right| \leq \frac{2^{-2^q(p-3)-1}}{|a|}$.

1: $x_0 = \mathrm{RN}(1/a_0)$
2: **for** $i \leftarrow 0$ to $q-1$ **do**
3: $\hat{v}[0 : 2^{i+1}-1] \leftarrow \mathit{MulRoundE}(x[0 : 2^i-1],\, a[0 : 2^{i+1}-1],\, 2^{i+1})$
4: $\hat{w}[0 : 2^{i+1}-1] \leftarrow \mathit{SubRoundE}(2,\, \hat{v}[0 : 2^{i+1}-1],\, 2^{i+1})$
5: $x[0 : 2^{i+1}-1] \leftarrow \mathit{MulRoundE}(x[0 : 2^i-1],\, \hat{w}[0 : 2^{i+1}-1],\, 2^{i+1})$
6: **end for**
7: **return** FP expansion $x = x_0 + \ldots + x_{2^q-1}$.

4.3 Error analysis of Algorithm 10

In the following, let $a = a_0 + \ldots + a_{2^k-1}$ be a *$\mathcal{B}$-nonoverlapping* FP expansion with 2^k terms and $q \geq 0$. We will prove that our algorithm returns an approximation $x = x_0 + \ldots + x_{2^q-1}$ of $\frac{1}{a}$, in the form of a *$\mathcal{B}$-nonoverlapping* FP expansion with 2^q terms, such that

$$\left|x - \frac{1}{a}\right| \leq \frac{2^{-2^q(p-3)-1}}{|a|}. \tag{15}$$

We will first prove the following proposition:

Proposition 4.2. *Consider a* $\mathcal{B}$-nonoverlapping *expansion* $u = u_0+u_1+\ldots+u_k$ *with* $k > 0$ *normal binary FP terms of precision* p. *Denote* $u^{(i)} = u_0 + u_1 + \cdots + u_i, i \geq 0$, *i.e. "a truncation" of* u *to* $i+1$ *terms. The following inequalities hold for* $0 \leq i \leq k$*:*

$$|u_i| \leq 2^{-ip} |u_0|, \tag{16}$$

$$\left|u - u^{(i)}\right| \leq 2^{-ip} |u| \frac{\eta}{1-\eta}, \tag{17}$$

$$\left(1 - 2^{-ip}\frac{\eta}{1-\eta}\right)|u| \leq \left|u^{(i)}\right| \leq \left(1 + 2^{-ip}\frac{\eta}{1-\eta}\right)|u|, \tag{18}$$

$$\left|\frac{1}{u} - \frac{1}{u_0}\right| \leq \frac{1}{|u|}\eta, \tag{19}$$

where

$$\eta = \sum_{j=0}^{\infty} 2^{(-j-1)p} = \frac{2^{-p}}{1-2^{-p}}.$$

Proof. By definition of a *$\mathcal{B}$-nonoverlapping* expansion and since for any normal binary FP number u_i, $\mathrm{ulp}(u_i) \leq 2^{-p+1}|u_i|$ we have $|u_i| \leq \frac{1}{2}\mathrm{ulp}(u_{i-1}) \leq 2^{-p}|u_{i-1}|$ and (16) follows by induction.

Consequently we have $|u-u_0| = |u_1+u_2+\cdots+u_k| \leq 2^{-p}|u_0| + 2^{-2p}|u_0| + \cdots + 2^{-kp}|u_0| \leq |u_0|\,\eta$. One easily observes that u and u_0 have the same sign. One possible proof is by noticing that $1-\eta > 0$ and $-|u_0|\,\eta \leq u-u_0 \leq |u_0|\,\eta$. Suppose $u_0 > 0$, then $-u_0\eta \leq u-u_0 \leq u_0\eta$, and hence $u_0(1-\eta) \leq u \leq u_0(1+\eta)$ which implies $u > 0$. The case $u_0 < 0$ is similar. It follows that

$$\frac{|u|}{1+\eta} \leq |u_0| \leq \frac{|u|}{1-\eta}. \tag{20}$$

For (17) we use (20) together with:

$$\left|u-u^{(i)}\right| \leq \sum_{j=0}^{\infty} 2^{(-i-j-1)p}\,|u_0| \leq 2^{-ip}\eta\,|u_0|\,,$$

and (18) is a simple consequence of (17). Similarly, (19) follows from $\left|\frac{1}{u}-\frac{1}{u_0}\right| = \frac{1}{|u|}\left|\frac{u_0-u}{u_0}\right| \leq \frac{1}{|u|}\eta$. □

Proposition 4.3. *Provided that no underflow (subnormal numbers appear) / overflow occurs during the calculations, Algorithm 10 is correct when run with* $\mathcal{B}$-nonoverlapping *expansions.*

Proof. The input of the algorithm is $a = a_0 + a_1 + \cdots + a_{2^k-1}$ a non-overlapping FP expansion in which every term a_i is a normal binary FP number of precision p. Let $f_i = 2^{i+1}-1$ and $a^{(f_i)} = a_0 + a_1 + \ldots + a_{f_i}$ i.e. "a truncation" of a to f_i+1 terms, with $0 \leq i$.

For computing $1/a$ we use Newton iteration: $x_0 = \mathrm{RN}(1/a_0)$, $x_{i+1} = x_i(2 - a^{(f_i)}x_i)), i \geq 0$ by truncating each operation involving FP expansions in the following way:

- let $v_i := (a^{(f_i)} \cdot x_i)$ be the exact product represented as a non-overlapping FP expansion on $2^{2(i+1)}$ terms, we compute $\hat{v}_i := v_i^{(2^i)}$ i.e. v_i "truncated to" 2^{i+1} terms;
- let $w_i := 2 - \hat{v}_i$ be the exact result of the subtraction represented as a non-overlapping FP expansion on $2^{i+1}+1$ terms, we compute $\hat{w}_i := w_i^{(2^i)}$ i.e. v_i "truncated to" 2^{i+1} terms;
- let $\tau_i := x_i \cdot \hat{w}_i$ be the exact product represented as a non-overlapping FP expansion on $2 \cdot 2^i(2^{i+1})$ terms, we compute $x_{i+1} := \tau_i^{(2^{i+1}-1)}$ i.e. τ_i "truncated to" 2^{i+1} terms.

Let us first prove a simple upper bound for the approximation error in x_0:

$$\varepsilon_0 = \left|x_0 - \frac{1}{a}\right| \leq \frac{2\eta}{|a|}. \tag{21}$$

Since $x_0 = \mathrm{RN}(1/a_0)$, then $\left|x_0 - \frac{1}{a_0}\right| \leq 2^{-p}\left|\frac{1}{a_0}\right|$, so $\left|x_0 - \frac{1}{a}\right| \leq 2^{-p}\left|\frac{1}{a_0}\right| + \left|\frac{1}{a} - \frac{1}{a_0}\right| \leq \frac{(1+\eta)2^{-p}+\eta}{|a|} \leq \frac{2\eta}{|a|}$ (from (20)).

Let us deduce an upper bound for the approximation error in x at step $i+1$, $\varepsilon_{i+1} = \left|x_{i+1} - \frac{1}{a}\right|$. For this, we will use a chain of triangular inequalites that

make the transition from our "truncated" Newton error, to the "untruncated" one. Let $\gamma_i = 2^{-(2^{i+1}-1)p}\frac{\eta}{1-\eta}$. We have from Proposition 4.2, eq. (17):

$$|x_{i+1} - \tau_i| \leq \gamma_i |x_i \cdot \hat{w}_i|, \tag{22}$$

$$|w_i - \hat{w}_i| \leq \gamma_i |w_i| \leq \gamma_i |2 - \hat{v}_i|, \tag{23}$$

$$|v_i - \hat{v}_i| \leq \gamma_i \left|a^{(f_i)} \cdot x_i\right|, \tag{24}$$

$$\left|a - a^{(f_i)}\right| \leq \gamma_i |a|. \tag{25}$$

From (22) we have:

$$\begin{aligned}
\varepsilon_{i+1} &\leq |x_{i+1} - \tau_i| + \left|\tau_i - \frac{1}{a}\right| \\
&\leq \gamma_i |x_i \cdot \hat{w}_i| + \left|x_i \cdot \hat{w}_i - \frac{1}{a}\right| \\
&\leq \gamma_i |x_i(w_i - \hat{w}_i)| + \gamma_i |x_i w_i| + \left|x_i \cdot \hat{w}_i - \frac{1}{a}\right| \\
&\leq (1+\gamma_i) |x_i| |w_i - \hat{w}_i| + \gamma_i |x_i w_i| \\
&+ \left|x_i \cdot w_i - \frac{1}{a}\right|.
\end{aligned}$$

Using (23) and (24):

$$\begin{aligned}
\varepsilon_{i+1} &\leq \left|x_i \cdot w_i - \frac{1}{a}\right| + ((1+\gamma_i)\gamma_i + \gamma_i) |x_i w_i| \\
&\leq \left|x_i \cdot (2 - v_i) - \frac{1}{a}\right| + |x_i| \cdot |(v_i - \hat{v}_i)| \\
&+ (\gamma_i(1+\gamma_i) + \gamma_i) |x_i| (|(2 - v_i)| + |v_i - \hat{v}_i|) \\
&\leq \left|x_i \cdot (2 - a^{(f_i)} \cdot x_i) - \frac{1}{a}\right| \\
&+ \gamma_i(1+\gamma_i)^2 \left|x_i^2\right| \left|a^{(f_i)}\right| \\
&+ (\gamma_i(1+\gamma_i) + \gamma_i) \left|x_i(2 - a^{(f_i)} \cdot x_i)\right|.
\end{aligned}$$

By (25), we have:

$$\left|x_i \cdot (2 - a^{(f_i)} \cdot x_i) - \frac{1}{a}\right| \leq |a| \left|x_i - \frac{1}{a}\right|^2 + \gamma_i |x_i|^2 |a|,$$

$$|x_i|^2 \left|a^{(f_i)}\right| \leq (1+\gamma_i) |x_i|^2 |a|,$$

and

$$\left|x_i \cdot (2 - a^{(f_i)} \cdot x_i)\right| \leq |a| \left|x_i - \frac{1}{a}\right|^2 + \gamma_i |x_i|^2 |a| + \frac{1}{|a|}.$$

Hence we have:

$$\begin{aligned}\varepsilon_{i+1} &\leq (1+\gamma_i)^2 |a| \left| x_i - \frac{1}{a} \right|^2 \\ &+ \gamma_i(1+\gamma_i)^2(2+\gamma_i) \left| x_i^2 \right| |a| \\ &+ \gamma_i(2+\gamma_i)\frac{1}{|a|}. \end{aligned} \tag{26}$$

We now prove by induction that for all $i \geq 0$:

$$\varepsilon_i = \left| x_i - \frac{1}{a} \right| \leq \frac{2^{-2^i(p-3)-1}}{|a|}. \tag{27}$$

For $i = 0$, this holds from (21) and the fact that $\eta = \frac{1}{2^p-1} \leq 2^{-p+1}$. For the induction step, we have from (26):

$$\begin{aligned}\varepsilon_{i+1} &\leq (1+\gamma_i)^2 |a| |\varepsilon_i|^2 \\ &+ \gamma_i(1+\gamma_i)^2(2+\gamma_i) (1 \pm \varepsilon_i |a|)^2 \frac{1}{|a|} \\ &+ \gamma_i(2+\gamma_i)\frac{1}{|a|}, \end{aligned} \tag{28}$$

which implies

$$\begin{aligned}\frac{|a|\,\varepsilon_{i+1}}{2^{-2^{i+1}(p-3)}} &\leq \frac{(1+\gamma_i)^2}{4} + \frac{(1+2^{-p+2})(2+\gamma_i)}{64} \\ &\cdot \left(1 + (1+\gamma_i)^2 \left(1 + 2^{-2^i(p-3)-1}\right)^2\right) \\ &\leq \frac{1}{2}. \end{aligned} \tag{29}$$

This completes our proof. □

Proposition 4.4. *Algorithm 10 is correct when run with* $\mathcal{P}$-nonoverlapping *expansions.*

Proof. It is easy to see that the previous analysis holds, provided that we use Remark 2.6. This mainly implies the following changes $\eta' = \frac{2}{2^p-3}$, $\gamma_i' = \left(\frac{2}{2^p-1}\right)^{2^{i+1}-1} \frac{\eta'}{1-\eta'}$. With this change it is easy to verify that equations (21)–(28) hold as soon as $p > 2$. Note that for the induction step at $i = 1$, a tighter bound is needed for $\varepsilon_0' \leq \frac{2^{-p}(1+\eta')+\eta'}{|a|} \leq \frac{2\eta'}{|a|}\frac{3-2^{-p}}{4}$, but the rest of the proof is identical, safe for some tedious computations. □

4.4 Complexity analysis for reciprocal

As presented before, our algorithm has the feature of using "truncated" expansions, while some variants of *AddRoundE* and *MulRoundE* need to compute the result fully and only then truncate. This is the case of Priest's addition and

multiplication, which are not tuned for obtaining "truncated" expansions on the fly –and thus penalize our algorithm–. On the other hand, our algorithms, presented in Sec. 2.2, take into account only the significant terms of the input expansions in order to compute the result. Even though these algorithms have not been proven to work properly yet, we obtained promising results in practice, so we will perform the complexity analysis based on them and we intend on providing full proofs in the near future.

We present here the operation count of our algorithms, by taking 6 FP operations for *2Sum* (Algorithm 1) [13], 3 for *Fast2Sum* [13] and 2 for *2MultFMA* (Algorithm 2) [13]. For the sake of simplicity we will consider that the input expansions have the same number of terms and that the output is smaller or as big as the inputs.

– The renormalization (Algorithm 6) of an overlapping expansion x with n terms, requires $(2n-3)+\sum_{i=0}^{m-2} m-i$ *Fast2Sum* calls and $n-1$ comparisons. This accounts for $R_{new}(n,m) = 7n + \frac{3}{2}m^2 + \frac{3}{2}m - 13$ FP operations.

– The addition (Algorithm 4) of two $\mathcal{P}$-*nonoverlapping* expansions requires $\sum_{i=1}^{k} i$ *2Sum* calls, $k-1$ simple FP additions and a renormalization $R_{new}(k+1,k)$. This accounts for $A(k) = \frac{9}{2}k^2 + \frac{25}{2}k - 7$ FP operations.

–> The special case of adding a FP expansion to a single FP value accounts for only $A1(k) = \frac{3}{2}k^2 + \frac{29}{2}k - 6$.

– The multiplication (Algorithm 5) of two $\mathcal{P}$-*nonoverlapping* expansions requires $\sum_{i=1}^{k} i$ *2MultFMA* calls, $\sum_{i=1}^{k-1}(n^2+n)$ *2Sum* calls, $k-1$ FP multiplications, followed by k^2+k-2 FP additions and a renormalization $R_{new}(k+1,k)$ in the end. This accounts for $M(k) = 2k^3 + \frac{7}{2}k^2 + \frac{19}{2}k - 9$ FP operations.

–> The special case of multiplying a FP expansion to a single FP value accounts for only $M1(k) = \frac{9}{2}k^2 + \frac{17}{2}k - 7$.

– Using these algorithms for addition and multiplication of FP expansions, Priest's division (Algorithm 9) requires d divisions and $(d-1)(M1(k)) + \sum_{i=0}^{d-1} A(k+2k(i-1)) + R_{new}(d,d)$ function calls in the worst case. This accounts for $D(d,k) = 6d^3k^2 - 18d^2k^2 + \frac{25}{2}d^2k + \frac{3}{2}d^2 + 21dk^2 - \frac{33}{2}dk - \frac{9}{2}d - \frac{9}{2}k^2 - \frac{17}{2}k - 6$ FP operations.

Proposition 4.5. *Using for addition and multiplication of FP expansions the algorithms presented in Sec. 2.2, Algorithm 10 requires* $\frac{32}{7}8^q + \frac{34}{3}4^q + 67\cdot 2^q - 24q - \frac{1720}{21}$ *FP operations.*

Proof. We observe that during the ith iteration the following operations with expansions are performed: 2 multiplications $M(2^{i+1})$; one addition $A1(2^{i+1})$. Since q iterations are done, the total number of FP operations is: $\frac{32}{7}8^q + \frac{34}{3}4^q + 67\cdot 2^q - 24q - \frac{1720}{21}$ FP operations.

□

Remark 4.6. Division is simply performed with Algorithm 10 followed by a multiplication $M(2^q)$ where the numerator expansion has 2^q terms.

5 Square root algorithms

The families of algorithms most commonly used are exactly the same as for division, although, in the case of FP expansions the digit-recurrence algorithm that generalizes the paper-and-pencil technique is typically a little more complicated than for division. This is the reason why a software implementation would be tedious. Moreover, Newton-Raphson based algorithms offer the advantage of assuring a quadratic convergence.

5.1 Square root of expansions with an adapted Newton-Raphson iteration

Starting from the general Newton-Raphson iteration (13), we can compute the square root in two different ways. We can look for the zeros of the function $f(x) = x^2 - a$ that leads to the so called "Heron iteration":

$$x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n}). \tag{30}$$

One can easily show that if $x_0 > 0$, then x_n goes to $\sqrt{a}$. This iteration needs a division at each step, which counts as a major drawback.

To avoid performing a division at each step we can look for the positive root of the function $f(x) = \frac{1}{x^2} - a$. From here we get the iteration

$$x_{n+1} = \frac{1}{2}x_n(3 - ax_n^2). \tag{31}$$

This iteration converges to $\frac{1}{\sqrt{a}}$, provided that $x_0 \in (0, \sqrt{3}/\sqrt{a})$. The result can be multiplied by a in order to get an approximation of $\sqrt{a}$. To obtain fast, quadratic, convergence, the first point x_0 must be a close approximation to $\frac{1}{\sqrt{a}}$. In this case we still need to perform a division (by 2), but this one is much simpler. Since, dividing a FP number by 2 can be done by multiplying it with 0.5, this being an exact operation, we can compute the division of a FP expansion by 2 by simply multiplying each of the terms by 0.5, separately.

As in the case of the reciprocal (Sec. 4.2), in our algorithm (Algorithm 11) we use an adaption of iteration (31), using the same truncated algorithms presented above.

The error analysis for this algorithm follows the same principle as the one for the reciprocal algorithm. The detailed proof is given in Appendix A. The goal is to show that the relative error decreases after every loop of the algorithm, by taking into account the truncations performed after each operation. The strategy is to make the exact Newton iteration term and bound appear. We show by induction that by the end of the ith iteration of the loop, $\varepsilon_i = \left| x^{(2^i)} - \frac{1}{\sqrt{a}} \right| \leq \frac{2^{-2^i(p-3)-1}}{\sqrt{a}}$.

In his library, QD, Bailey also uses the Newton iteration for the square root computation. Although he uses the same function as we do, he uses the iteration under the form: $x_{i+1} = x_i + \frac{1}{2}x_i(1 - ax_i^2)$, which from a mathematical point of view is the same, but it requires a different implementation. Even though Bailey does not provide an error analysis for his algorithm, we managed to prove that the error bound is preserved when using this iteration (see Appendix A for the detailed proof).

Algorithm 11 Truncated "division-free" Newton iteration (31) based algorithm for reciprocal of the square root of an FP expansion. By "division-free" we mean that we do not need a division of FP expansions, we just need a division by 2.

Input: FP expansion $a = a_0 + \ldots + a_{2^k-1}$; length of output FP expansion 2^q.
Output: FP expansion $x = x_0 + \ldots + x_{2^q-1}$ s.t.

$$\left| x - \frac{1}{\sqrt{a}} \right| \leq \frac{2^{-2^q(p-3)-1}}{\sqrt{a}}. \tag{32}$$

1: $x_0 = \mathrm{RN}(1/\sqrt{a_0})$
2: **for** $i \leftarrow 0$ to $q-1$ **do**
3: $\hat{v}[0 : 2^{i+1}-1] \leftarrow \mathit{MulRoundE}(x[0 : 2^i-1],\, a[0 : 2^{i+1}-1],\, 2^{i+1})$
4: $\hat{w}[0 : 2^{i+1}-1] \leftarrow \mathit{MulRoundE}(x[0 : 2^i-1],\, \hat{v}[0 : 2^{i+1}-1],\, 2^{i+1})$
5: $\hat{y}[0 : 2^{i+1}-1] \leftarrow \mathit{SubRoundE}(3,\, \hat{w}[0 : 2^{i+1}-1],\, 2^{i+1})$
6: $\hat{z}[0 : 2^{i+1}-1] \leftarrow \mathit{MulRoundE}(x[0 : 2^i-1],\, \hat{y}[0 : 2^{i+1}-1],\, 2^{i+1})$
7: $x[0 : 2^{i+1}-1] \leftarrow \hat{z}[0 : 2^{i+1}-1] * 0.5$
8: **end for**
9: **return** FP expansion $x = x_0 + \ldots + x_{2^q-1}$.

"Heron iteration" algorithm

The same type of proof as above can be applied for the algorithm using the "Heron iteration" (30) and the same type of truncations. In this case (Algorithm 12) we obtain a slightly larger error bound for both types of *nonoverlapping* FP expansions: $|x - \sqrt{a}| \leq 3\sqrt{a} \cdot 2^{-2^q(p-3)-2}$.

5.2 Complexity analysis for square root

We will perform our operation count based on the addition and multiplication presented in Sec. 2.2, the same as in Section 4.4.

Proposition 5.1. *Using for addition, multiplication and division of FP expansions the algorithms previously presented, Algorithm 11 requires* $\frac{48}{7}8^q + 16 \cdot 4^q + 88 \cdot 2^q - 33q - \frac{769}{7}$ *FP operations.*

Proof. We observe that during the ith iteration the following operations with expansions are performed: 3 multiplications $M(2^{i+1})$, one addition $A1(2^{i+1})$ and one division by 2. Since q iterations are done, the total number of FP operations is: $\frac{48}{7}8^q + 16 \cdot 4^q + 88 \cdot 2^q - 33q - \frac{769}{7}$. □

Remark 5.2. We obtain the square root of an expansion by simply multiplying the result obtained from Algorithm 11 by a, the input expansion; this means an additional $M(2^q)$, where 2^q is the number of terms in a.

Proposition 5.3. *Using for addition, multiplication and division of FP expansions the algorithms previously presented, Algorithm 12 requires* $\frac{368}{49}8^q + \frac{232}{9}4^q + 180 \cdot 2^q - 12q^2 - \frac{2308}{21}q - \frac{93619}{441}$ *FP operations.*

Algorithm 12 Truncated "Heron iteration" (30) based algorithm for square root of an FP expansion.

Input: FP expansion $a = a_0 + \ldots + a_{2^k-1}$; length of output FP expansion 2^q.
Output: FP expansion $x = x_0 + \ldots + x_{2^q-1}$ s.t.

$$\left|x - \sqrt{a}\right| \leq 3\sqrt{a} \cdot 2^{-2^q(p-3)-2}. \tag{33}$$

1: $x_0 = \mathrm{RN}(\sqrt{a_0})$
2: **for** $i \leftarrow 0$ to $q-1$ **do**
3: $\hat{v}[0 : 2^{i+1}-1] \leftarrow \mathit{DivRoundE}(a[0 : 2^{i+1}-1],\, x[0 : 2^i-1],\, 2^{i+1})$
4: $\hat{w}[0 : 2^{i+1}-1] \leftarrow \mathit{AddRoundE}(x[0 : 2^i-1],\, \hat{v}[0 : 2^{i+1}-1],\, 2^{i+1})$
5: $x[0 : 2^{i+1}-1] \leftarrow \hat{w}[0 : 2^{i+1}-1] * 0.5$
6: **end for**
7: **return** FP expansion $x = x_0 + \ldots + x_{2^q-1}$.

Proof. We observe that one addition $A(2^{i+1})$, one division $D(2^{i+1})$ and a division by 2 are performed during each ith iteration. Since q iterations are done, the total number of FP operations is: $\frac{368}{49}8^q + \frac{232}{9}4^q + 180 \cdot 2^q - 12q^2 - \frac{2308}{21}q - \frac{93619}{441}$. □

Based on this values Algorithm 11 performs slightly better than Algorithm 12, plus in the same time, the error bound obtained is tighter.

6 Comparison and Discussion

In Table 2 we show effective values for the bounds provided by our error analysis compared with those of Priest and Daumas for the reciprocal computation. Our algorithm performs better for the same number of terms in the computed quotient, say $d = 2^q$ in equations (11) and (12). Moreover, our algorithm provides a unified error bound with quadratic convergence independent of using underlying $\mathcal{P}$- or $\mathcal{B}$-*nonoverlapping* expansions. In the last column of the same table we give the largest errors that we obtained through direct computation of the reciprocal using our algorithm. The given value represents the obtained value upper rounded to the immediate power of 2. For each table entry we performed 1 million random tests.

The complexity analysis performed with $\mathcal{P}$-*nonoverlapping* expansions shows that our algorithm performs better, for expansions with more than 2 terms, even if no error bound is requested (see Table 3 for some effective values of the worst case FP operation count).

Note that, for instance, to guarantee an error bound of $2^{-d(p-3)-1}$, Priest's algorithm (based on the bound given in Prop 4.1) needs at least $(dp - 3d + 2)p/(p-4)$ terms, which entails a very poor complexity. This implies that Daumas' algorithm might be a good compromise in this case, provided that the priority queue used there can be efficiently implemented.

This and also the performance tests that we ran confirm our hypothesis that for higher precisions the Newton-Raphson iteration is preferable to classical division.

Table 2: Error bounds values for Priest (11) vs. Daumas (12) vs. our analysis (15). β gives the largest obtained errors for Algorithm 10 using the standard FP formats *double* and *single*. *underflow occurs

Prec, iteration	Eq. (11)	Eq. (12)	Eq. (15)	β
$p = 53, q = 0$	2	2^{-49}	2^{-51}	2^{-52}
$p = 53, q = 1$	1	2^{-98}	2^{-101}	2^{-104}
$p = 53, q = 2$	2^{-2}	2^{-195}	2^{-201}	2^{-208}
$p = 53, q = 3$	2^{-6}	2^{-387}	2^{-401}	2^{-416}
$p = 53, q = 4$	2^{-13}	2^{-764}	2^{-801}	2^{-833}
$p = 24, q = 0$	2	2^{-20}	2^{-22}	2^{-23}
$p = 24, q = 1$	1	2^{-40}	2^{-43}	2^{-46}
$p = 24, q = 2$	2^{-2}	2^{-79}	2^{-85}	2^{-92}
$p = 24, q = 3$	2^{-5}	2^{-155}	2^{-169}	$*$
$p = 24, q = 4$	2^{-12}	2^{-300}	2^{-337}	$*$

Table 3: FP operation count for Priest vs. our algorithm; $d = 2^q$ terms are computed in the quotient.

d	2	4	8	16
Alg. 9 [16]	62	3310	138674	5243818
Alg. 10 + Alg. 5	150	825	4763	31751

In the case of the square root, because no error bound is given for the *digit-recurrence* algorithm we can only compare between the errors that we obtain if using the two different types of Newton iteration available for computing the square root. The effective values of the bounds are given in Table 4. You can see that the bound provided for Algorithm 11 is only slightly tighter that the one for Algorithm 12. The same as for the reciprocal, in the last column we present the bounds obtained through direct computation using Algorithm 11.

Table 4: Error bounds values for (44) vs. (33). β gives the largest obtained errors for Algorithm 11 using the standard FP formats *double* and *single*. *underflow occurs

Prec, iteration	Eq. (44)	Eq. (33)	β
$p = 53, q = 0$	2^{-51}	$3 \cdot 2^{-52}$	2^{-52}
$p = 53, q = 1$	2^{-101}	$3 \cdot 2^{-102}$	2^{-103}
$p = 53, q = 2$	2^{-201}	$3 \cdot 2^{-202}$	2^{-206}
$p = 53, q = 3$	2^{-401}	$3 \cdot 2^{-402}$	2^{-412}
$p = 53, q = 4$	2^{-801}	$3 \cdot 2^{-802}$	2^{-823}
$p = 24, q = 0$	2^{-22}	$3 \cdot 2^{-23}$	2^{-23}
$p = 24, q = 1$	2^{-43}	$3 \cdot 2^{-44}$	2^{-45}
$p = 24, q = 2$	2^{-85}	$3 \cdot 2^{-86}$	2^{-90}
$p = 24, q = 3$	2^{-169}	$3 \cdot 2^{-170}$	$*$
$p = 24, q = 4$	2^{-337}	$3 \cdot 2^{-338}$	$*$

In Table 5 we give some effective values of the worst case FP operation count for Algorithm 11 vs Algorithm 12 based on section 5.2.

All the algorithms presented in this article were implemented in the CAM-

Table 5: FP operation count for Algorithm 11 vs. Algorithm 12; 2^q terms are computed in the quotient.

q	1	2	3	4
Alg. 11 + Alg. 5	192	1084	6345	42580
Alg. 12	189	1133	6285	39397

PARY (CudA Multiple Precision ARithmetic librarY) software available at `http://homepages.laas.fr/mmjoldes/campary`. The library is implemented in CUDA – an extension of the C language developed by NVIDIA [14] for their GPUs. The algorithms presented in this article are very suitable for the GPU: all basic operations $(+, -, *, /, \sqrt{})$ conform to the IEEE 754-2008 standard for FP arithmetic for single and double precision; support for the four rounding modes is provided and dynamic rounding mode change is supported without any penalties. The `fma` instruction also is supported for all devices with CUDA Compute Capability at least 2.0.

In the implementation we use templates for both the number of terms in the expansion and the native type for the terms. In other words, we allow static generation of any input-output precision combinations (e.g. add a double-double with a quad-double and store the result on triple-double) and operations with types like single-single, quad-single etc., are supported. All the functions are defined using __host__ __device__ specifiers, which allows for the library to be used on both CPU and GPU.

This article focuses on the theoretical part of the algorithms, but we also performed tests to measure the performance of CAMPARY to the one of QD, respectively, GQD and MPFR. The effective performance values obtained, given in MFlops/s can be consulted at `http://homepages.laas.fr/mmjoldes/campary`.

As a future work we intend to generalize the theoretical analysis of DD and QD addition/multiplication algorithms and thus to be able to provide a full error analysis for these algorithms.

Acknowledgments

The Authors would like to thank Région Rhône-Alpes and ANR FastRelax Project for the grants that supports this activity.

References

[1] Alberto Abad, Roberto Barrio, and Ángeles Dena. Computing periodic orbits with arbitrary precision. *Phys. Rev. E*, 84:016701, Jul 2011.

[2] M. Cornea, R. A. Golliver, and P. Markstein. Correctness proofs outline for Newton–Raphson-based floating-point divide and square root algorithms. In Koren and Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 96–105. IEEE Computer Society Press, Los Alamitos, CA, April 1999.

[3] Marc Daumas and Claire Finot. Division of floating point expansions with an application to the computation of a determinant. *Journal of Universal Computer Science*, 5(6):323–338, jun 1999.

[4] M. D. Ercegovac and T. Lang. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, Boston, MA, 1994.

[5] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Transactions on Mathematical Software*, 33(2), 2007. available at `http://www.mpfr.org/`.

[6] Y. Hida, X. S. Li, and D. H. Bailey. Algorithms for quad-double precision floating-point arithmetic. In N. Burgess and L. Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*, pages 155–162, Vail, CO, June 2001.

[7] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, August 2008. available at `http://ieeexplore.ieee.org/servlet/opac?punumber=4610933`.

[8] Claude-Pierre Jeannerod and Siegfried M. Rump. Improved error bounds for inner products in floating-point arithmetic. *SIAM Journal on Matrix Analysis and Applications*, 34(2):338–344, April 2013.

[9] M. Joldes, J.-M. Muller, and V. Popescu. On the computation of the reciprocal of floating point expansions using an adapted newton-raphson iteration. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, pages 63–67, June 2014.

[10] M. Joldes, V. Popescu, and W.Tucker. Searching for sinks of Hénon map using a multiple-precision GPU arithmetic library. Technical report, LAAS, Nov 2013.

[11] P. Kornerup, V. Lefèvre, N. Louvet, and J.-M. Muller. On the computation of correctly-rounded sums. In *Proceedings of the 19th IEEE Symposium on Computer Arithmetic (ARITH-19)*, Portland, OR, June 2009.

[12] J. Laskar and M. Gastineau. Existence of collisional trajectories of Mercury, Mars and Venus with the Earth. *Nature*, 459(7248):817–819, June 2009.

[13] 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. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[14] NVIDIA. *NVIDIA CUDA Programming Guide 5.5*. 2013.

[15] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

[16] D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic (Arith-10)*, pages 132–144. IEEE Computer Society Press, Los Alamitos, CA, June 1991.

[17] D. M. Priest. *On Properties of Floating-Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, University of California at Berkeley, 1992.

[18] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: Faithful rounding. *SIAM Journal on Scientific Computing*, 31(1):189–224, 2008.

[19] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Computational Geometry*, 18:305–363, 1997.

[20] Tjalling J. Ypma. Historical development of the Newton-Raphson method. *SIAM Rev.*, 37(4):531–551, December 1995.

A Appendix

Square root algorithm convergence proof

This proof follows the same principle as the convergence proof for the reciprocal algorithm, and it uses the same notations. We will show that Algorithm 11, having as an input a FP expansion $a = a_0 + \cdots + a_{2^k-1}$, computes and returns an approximation of $\frac{1}{\sqrt{a}}$ in the form of a FP expansion $x = x_0 + \cdots = x_{2^q-1}$, s.t:

$$\left| x - \frac{1}{\sqrt{a}} \right| \leq \frac{2^{-2^q(p-3)-1}}{\sqrt{a}}.$$

For the sake of clarity we present first the case of $\mathcal{B}$-*nonoverlapping* FP expansion, and then make the necessary adjustments for $\mathcal{P}$-*nonoverlapping* expansion in A.2.

Proposition A.1. *Algorithm 11 is correct when run with $\mathcal{B}$-nonoverlapping expansions.*

Proof. In Proposition 4.2 we gave and proved some properties of the $\mathcal{B}$-*nonoverlapping* FP expansions. To those we are going to add a new one:

$$\left| \frac{1}{\sqrt{u}} - \frac{1}{\sqrt{u_0}} \right| \leq \frac{1}{\sqrt{u}}\eta. \tag{34}$$

It can be seen that $\left| \frac{1}{\sqrt{u}} - \frac{1}{\sqrt{u_0}} \right| = \frac{1}{\sqrt{u}} \left| 1 - \frac{\sqrt{u}}{\sqrt{u_0}} \right|$. By using Proposition 4.2 eq. (20), the fact that u and u_0 have the same sign, and the fact that the square root is an increasing function, we have: $\left| 1 - \frac{\sqrt{u}}{\sqrt{u_0}} \right| \leq 1 - \sqrt{\frac{1}{1+\eta}} \leq \eta$, which proves the property.

We continue by first proving a simple upper bound for the approximation error in x_0:

$$\varepsilon_0 = \left| x_0 - \frac{1}{\sqrt{a}} \right| \leq \frac{1}{\sqrt{a}}\eta(3+\eta). \tag{35}$$

We denote $\alpha = \mathrm{RN}(\sqrt{a_0})$, so we have $x_0 = \mathrm{RN}(1/\alpha)$. We know that $\left|\alpha - \sqrt{a_0}\right| \leq 2^{-p}\sqrt{a_0}$ and $\left|x_0 - \frac{1}{\alpha}\right| \leq 2^{-p}\frac{1}{\alpha}$, so we obtain: $\left| \frac{1}{\alpha} - \frac{1}{\sqrt{a_0}} \right| \leq (\frac{1}{1-2^{-p}} - 1)\frac{1}{\sqrt{a_0}}$. By (34) we have:

$$\begin{aligned}
\left| x_0 - \frac{1}{\sqrt{a_0}} \right| &\leq \left| x_0 - \frac{1}{\alpha} \right| + \left| \frac{1}{\alpha} - \frac{1}{\sqrt{a_0}} \right| \\
&\leq 2^{-p}\frac{1}{\alpha} + \left| \frac{1}{\alpha} - \frac{1}{\sqrt{a_0}} \right| \\
&\leq 2^{-p}\frac{1}{\sqrt{a_0}} + (2^{-p}+1)\left| \frac{1}{\alpha} - \frac{1}{\sqrt{a_0}} \right| \\
&\leq 2\eta\frac{1}{\sqrt{a_0}}.
\end{aligned}$$

From (34) it follows:

$$\begin{aligned}\varepsilon_0 &\leq \left|x_0 - \frac{1}{\sqrt{a_0}}\right| + \left|\frac{1}{\sqrt{a}} - \frac{1}{\sqrt{a_0}}\right| \\ &\leq 2\eta\frac{1}{\sqrt{a_0}} + \eta\frac{1}{\sqrt{a}}.\end{aligned}$$

Because $\frac{\sqrt{a}}{\sqrt{1+\eta}} \leq \sqrt{a_0}$, then $\frac{1}{\sqrt{a_0}} \leq \frac{\sqrt{1+\eta}}{\sqrt{a}} \leq \frac{1+\eta/2}{\sqrt{a}}$.
Now we can conclude that $\varepsilon_0 \leq (2\eta(1+\frac{\eta}{2})+\eta)\frac{1}{\sqrt{a}} \leq \eta(3+\eta)\frac{1}{\sqrt{a}}$.
Before going further, let $E_i = \varepsilon_i\sqrt{a}$, such that:

$$E_0 \leq \eta(3+\eta).$$

Next we will deduce an upper bound for the approximation error in x at step $i+1$, $\varepsilon_{i+1} = \left|x_{i+1} - \frac{1}{\sqrt{a}}\right|$. For this, we use, same as in the case of the reciprocal, a chain of triangular inequalities that make the transition from our "truncated" Newton error, to the "untruncated" one.

For the truncations we use the same type of notations as for the reciprocal error bound proof, so: at each step of the algorithm v_i, w_i, and y_i represent the exact results of the computations and we use the hatted notation ($\hat{v}_i$, $\hat{w}_i$, and $\hat{y}_i$) to represent the results truncated to 2^{i+1} terms. τ_i denotes the exact result of the last two operations: $\frac{1}{2}x_i \cdot \hat{y}_i$, so x_{i+1} is τ_i's truncation to 2^{i+1} terms.

Let $\gamma_i = 2^{-(2^{i+1}-1)p}\frac{\eta}{1-\eta}$, the same as before. We have from Proposition 4.2, eq. (17):

$$|x_{i+1} - \tau_i| \leq \gamma_i |\tau_i| \leq \gamma_i \left|\frac{1}{2}x_i\hat{y}_i\right|, \tag{36}$$

$$|y_i - \hat{y}_i| \leq \gamma_i |y_i| \leq \gamma_i |3 - \hat{w}_i|, \tag{37}$$

$$|w_i - \hat{w}_i| \leq \gamma_i |w_i| \leq \gamma_i |x_i\hat{v}_i|, \tag{38}$$

$$|v_i - \hat{v}_i| \leq \gamma_i |v_i| \leq \gamma_i \left|x_i a^{(f_i)}\right|, \tag{39}$$

$$\left|a - a^{(f_i)}\right| \leq \gamma_i |a|. \tag{40}$$

From (36) we have:

$$\begin{aligned}\varepsilon_{i+1} &\leq |x_{i+1} - \tau_i| + \left|\tau_i - \frac{1}{\sqrt{a}}\right| \\ &\leq \gamma_i \left|\frac{1}{2}x_i\hat{y}_i\right| + \left|\frac{1}{2}x_i\hat{y}_i - \frac{1}{\sqrt{a}}\right|.\end{aligned}$$

Using (37) and (38):

$$\begin{aligned}\varepsilon_{i+1} &\leq \gamma_i(2+\gamma_i)\left|\frac{1}{2}x_i(3-\hat{w}_i)\right| \\ &\quad + \left|\frac{1}{2}x_i(3-\hat{w}_i) - \frac{1}{\sqrt{a}}\right| \\ &\leq \gamma_i(2+\gamma_i)\left|\frac{1}{2}x_i\right|(|3-w_i| + \gamma_i|w_i|) \\ &\quad + \gamma_i\left|\frac{1}{2}x_i w_i\right| + \left|\frac{1}{2}x_i(3-w_i) - \frac{1}{\sqrt{a}}\right|.\end{aligned}$$

By (39) we have:

$$\begin{aligned}
\varepsilon_{i+1} &\leq \gamma_i \left|\frac{1}{2}x_i\right| ((2+\gamma_i)\,|3 - x_i\hat{v}_i| \\
&\quad +(1+\gamma_i(2+\gamma_i))\,|x_i\hat{v}_i|) \\
&\quad + \left|\frac{1}{2}x_i(3 - x_i\hat{v}_i) - \frac{1}{\sqrt{a}}\right| \\
&\leq \gamma_i \left|\frac{1}{2}x_i\right| ((2+\gamma_i)(|3 - x_i v_i| + \gamma_i\,|v_i x_i|) \\
&\quad +(1+\gamma_i(2+\gamma_i))(1+\gamma_i)\,|x_i v_i|) \\
&\quad +\gamma_i \left|\frac{1}{2}x_i^2 v_i\right| + \left|\frac{1}{2}x_i(3 - x_i v_i) - \frac{1}{\sqrt{a}}\right|.
\end{aligned}$$

From (40) we have:

$$\begin{aligned}
\varepsilon_{i+1} &\leq \gamma_i(2+\gamma_i) \left|\frac{1}{2}x_i\right| (\left|3 - x_i^2 a^{(f_i)}\right| \\
&\quad +(\gamma_i+1)^2 \left|x_i^2 a^{(f_i)}\right|) + \left|\frac{1}{2}x_i(3 - x_i^2 a^{(f_i)}) - \frac{1}{\sqrt{a}}\right| \\
&\leq \gamma_i(2+\gamma_i)(\left|\frac{1}{2}x_i(3 - x_i^2 a)\right| + \gamma_i \left|\frac{1}{2}x_i^3 a\right|) \\
&\quad +\gamma_i \left|\frac{1}{2}x_i^3 a\right| + \gamma_i(2+\gamma_i)(\gamma_i+1)^3 \left|\frac{1}{2}x_i^3 a\right| \\
&\quad + \left|\frac{1}{2}x_i(3 - x_i^2 a) - \frac{1}{\sqrt{a}}\right|.
\end{aligned}$$

Hence we have:

$$\begin{aligned}
\varepsilon_{i+1} &\leq (1+\gamma_i)^2 \left|x_{i+1} - \frac{1}{\sqrt{a}}\right| \\
&\quad +\gamma_i((\gamma_i+1)^2 + (\gamma_i+1)^3 \\
&\quad +(\gamma_i+1)^4) \left|\frac{1}{2}x_i^3 a\right| \\
&\quad +\gamma_i(2+\gamma_i)\frac{1}{\sqrt{a}}.
\end{aligned} \tag{41}$$

By using the quadratic convergence of the sequence we can say that:

$$\left|x_{i+1} - \frac{1}{\sqrt{a}}\right| = \frac{1}{2}\sqrt{a}(x_i\sqrt{a} + 2)\left|x_i - \frac{1}{\sqrt{a}}\right|^2. \tag{42}$$

We now prove by induction that for all $i \geq 0$ $\varepsilon_i = \left|x_i - \frac{1}{\sqrt{a}}\right|$ respects the imposed bound.

We know that $|x_i\sqrt{a}| \leq \varepsilon_i\sqrt{a} + 1$ and $\left|x_i^3 a\right| \leq \frac{(\varepsilon_i\sqrt{a}+1)^3}{\sqrt{a}}$ and from (41) we

have:

$$\begin{aligned}\varepsilon_{i+1} &\leq \frac{1}{2}(1+\gamma_i)^2\sqrt{a}(\varepsilon_i\sqrt{a}+3)\varepsilon_i^2 \\ &+\frac{1}{2}\gamma_i((\gamma_i+1)^2+(\gamma_i+1)^3 \\ &+(\gamma_i+1)^4)\frac{(\varepsilon_i\sqrt{a}+1)^3}{\sqrt{a}} \\ &+\gamma_i(2+\gamma_i)\frac{1}{\sqrt{a}}. \end{aligned} \tag{43}$$

Using the notation $E_i = \varepsilon_i\sqrt{a}$ we can transform (43) in an equation independent of a:

$$\begin{aligned}E_{i+1} &\leq \frac{1}{2}(1+\gamma_i)^2(E_i+3)E_i^2 \\ &+\frac{1}{2}\gamma_i((\gamma_i+1)^2+(\gamma_i+1)^3 \\ &+(\gamma_i+1)^4)(E_i+1)^3+\gamma_i(2+\gamma_i)\end{aligned}$$

For the last part of the proof we denote by f a function that writes the previous inequality as: $E_{i+1} \leq f(E_i, i)$. We want to show that $\forall i \in \mathbb{N}, E_i \leq 2^{-2^i(p-3)-1}$ so we will define $\text{ind}(i) = 2^{-2^i(p-3)-1}$.

For $i = 0$ we verify that $E_0 \leq \text{ind}(0)$ for $p \geq 3$.

For $i \geq 1$ by induction:

- for $i = 1$ we can prove by using computer algebra that the inequality is verified if $p \geq 4$, which holds in practice;
- it is easily shown (by using the definition of a decreasing function and computation for example) that the function $i \mapsto \frac{f(\text{ind}(i),i)}{\text{ind}(i+1)}$ is decreasing and it's value in 1 is < 1 for $p \geq 3$. So, $\frac{f(\text{ind}(i),i)}{\text{ind}(i+1)} \leq 1$;
- suppose that $E_i \leq \text{ind}(i)$, we have $\frac{E_{i+1}}{\text{ind}(i+1)} \leq \frac{f(\text{ind}(i),i)}{\text{ind}(i+1)} \leq 1$ which concludes the induction.

At last we find the final inequality with $i = q$. □

Proposition A.2. *Algorithm 11 is correct when run with $\mathcal{P}$-nonoverlapping expansions.*

Proof. The proof is similar, the same error analysis holds with the same parameter changes as in Prop. 4.4. □

Bailey's Iteration Convergence Proof

Similarly to the previous proof, one has from Proposition 4.2, eq. (17):

$$|x_{i+1} - \tau_i| \leq \gamma_i |\tau_i| \leq \gamma_i \left|x_i + \hat{t}_i\right| \leq \gamma_i \left|x_i + \frac{\hat{z}_i}{2}\right|, \tag{45}$$

Algorithm 13 Algorithm for reciprocal of the square root of an FP expansion based on Newton-Raphson interation of the form $x_{i+1} = x_i + \frac{1}{2}x_i(1 - ax_i^2)$, which is used in QD library

Input: FP expansion $a = a_0 + \ldots + a_{2^k-1}$; length of output FP expansion 2^q.
Output: FP expansion $x = x_0 + \ldots + x_{2^q-1}$ s.t.

$$\left| x - \frac{1}{\sqrt{a}} \right| \leq \frac{2^{-2^q(p-3)-1}}{\sqrt{a}}. \tag{44}$$

1: $x_0 = \mathrm{RN}(1/\sqrt{a_0})$
2: **for** $i \leftarrow 0$ to $q-1$ **do**
3: $\hat{v}[0 : 2^{i+1}-1] \leftarrow \mathit{MulRoundE}(x[0 : 2^i - 1],\, a[0 : 2^{i+1}-1],\, 2^{i+1})$
4: $\hat{w}[0 : 2^{i+1}-1] \leftarrow \mathit{MulRoundE}(x[0 : 2^i - 1],\, \hat{v}[0 : 2^{i+1}-1],\, 2^{i+1})$
5: $\hat{y}[0 : 2^{i+1}-1] \leftarrow \mathit{SubRoundE}(1,\, \hat{w}[0 : 2^{i+1}-1],\, 2^{i+1})$
6: $\hat{z}[0 : 2^{i+1}-1] \leftarrow \mathit{MulRoundE}(x[0 : 2^i - 1],\, \hat{y}[0 : 2^{i+1}-1],\, 2^{i+1})$
7: $\hat{t}[0 : 2^{i+1}-1] \leftarrow \hat{z}[0 : 2^{i+1}-1] * 0.5$
8: $x[0 : 2^{i+1}-1] \leftarrow \mathit{AddRoundE}(x[0 : 2^i - 1],\, \hat{t}[0 : 2^{i+1}-1],\, 2^{i+1})$
9: **end for**
10: **return** FP expansion $x = x_0 + \ldots + x_{2^q-1}$.

$$|z_i - \hat{z}_i| \leq \gamma_i |z_i| \leq \gamma_i |x_i \hat{y}_i|, \tag{46}$$

$$|y_i - \hat{y}_i| \leq \gamma_i |y_i| \leq \gamma_i |1 - \hat{w}_i|, \tag{47}$$

$$|w_i - \hat{w}_i| \leq \gamma_i |w_i| \leq \gamma_i |x_i \hat{v}_i|, \tag{48}$$

$$|v_i - \hat{v}_i| \leq \gamma_i |v_i| \leq \gamma_i \left| x_i a^{(f_i)} \right|, \tag{49}$$

$$\left| a - a^{(f_i)} \right| \leq \gamma_i |a|. \tag{50}$$

From (45) we have:

$$\begin{aligned} \varepsilon_{i+1} &\leq |x_{i+1} - \tau_i| + \left| \tau_i - \frac{1}{\sqrt{a}} \right| \\ &\leq \gamma_i \left| x_i + \frac{\hat{z}_i}{2} \right| + \left| x_i + \frac{\hat{z}_i}{2} - \frac{1}{\sqrt{a}} \right|. \end{aligned}$$

Using (46) and (47):

$$\begin{aligned} \varepsilon_{i+1} &\leq \gamma_i(1+\gamma_i) \left| \frac{1}{2} x_i \hat{y}_i \right| \\ &\quad + \gamma_i \left| x_i + \frac{x_i \hat{y}_i}{2} \right| \\ &\quad + \left| x_i + \frac{x_i \hat{y}_i}{2} - \frac{1}{\sqrt{a}} \right| \\ &\leq \gamma_i(1+\gamma_i)^2 \left| \frac{1}{2} x_i (1 - \hat{w}_i) \right| \end{aligned}$$

$$
\begin{aligned}
&+\gamma_i \left| x_i + \frac{1}{2} x_i (1 - \hat{w}_i) \right| + \gamma_i^2 \left| \frac{1}{2} x_i (1 - \hat{w}_i) \right| \\
&+ \left| x_i + \frac{1}{2} x_i (1 - \hat{w}_i) - \frac{1}{\sqrt{a}} \right| + \gamma_i \left| \frac{1}{2} x_i (1 - \hat{w}_i) \right| \\
\leq \quad & \gamma_i (1 + \gamma_i)(2 + \gamma_i) \left| \frac{1}{2} x_i (1 - \hat{w}_i) \right| \\
&+\gamma_i \left| x_i + \frac{1}{2} x_i (1 - \hat{w}_i) \right| \\
&+ \left| x_i + \frac{1}{2} x_i (1 - \hat{w}_i) - \frac{1}{\sqrt{a}} \right| .
\end{aligned}
$$

By (48) we have:

$$
\begin{aligned}
\varepsilon_{i+1} \quad \leq \quad & \gamma_i (1 + \gamma_i)(2 + \gamma_i) \left| \frac{1}{2} x_i (1 - x_i \hat{v}_i) \right| \\
&+\gamma_i \left| x_i + \frac{1}{2} x_i (1 - x_i \hat{v}_i) \right| \\
&+ \left| x_i + \frac{1}{2} x_i (1 - x_i \hat{v}_i) - \frac{1}{\sqrt{a}} \right| \\
&+\gamma_i (1 + \gamma_i)^3 \left| \frac{1}{2} x_i^2 \hat{v}_i \right|
\end{aligned}
$$

From (49) we have:

$$
\begin{aligned}
\varepsilon_{i+1} \quad \leq \quad & \gamma_i (1 + \gamma_i)(2 + \gamma_i) \left| \frac{1}{2} x_i (1 - x_i^2 a^{(f_i)}) \right| \\
&+\gamma_i \left| x_i + \frac{1}{2} x_i (1 - x_i^2 a^{(f_i)}) \right| \\
&+ \left| x_i + \frac{1}{2} x_i (1 - x_i^2 a^{(f_i)}) - \frac{1}{\sqrt{a}} \right| \\
&+\gamma_i (1 + \gamma_i)^3 (2 + \gamma_i) \left| \frac{1}{2} x_i^3 a^{(f_i)} \right|
\end{aligned}
$$

From (50) we have:

$$
\begin{aligned}
\varepsilon_{i+1} \quad \leq \quad & \gamma_i (1 + \gamma_i)(2 + \gamma_i) \left| \frac{1}{2} x_i (1 - x_i^2 a) \right| \\
&+\gamma_i \left| x_i + \frac{1}{2} x_i (1 - x_i^2 a) \right| \\
&+ \left| x_i + \frac{1}{2} x_i (1 - x_i^2 a) - \frac{1}{\sqrt{a}} \right| \\
&+\gamma_i (1 + \gamma_i)^3 (3 + 3\gamma_i + \gamma_i^2) \left| \frac{1}{2} x_i^3 a \right|
\end{aligned}
$$

From (42) and similarly to (41), one has:

$$
\varepsilon_{i+1} \quad \leq \quad (1 + \gamma_i) \left| x_{i+1} - \frac{1}{\sqrt{a}} \right|
$$

$$
\begin{aligned}
&+\gamma_i(1+\gamma_i)(2+\gamma_i)\left|\frac{1}{2}x_i(1-x_i^2a)\right| \\
&+\gamma_i(1+\gamma_i)^3(3+3\gamma_i+\gamma_i^2)\left|\frac{1}{2}x_i^3a\right| \\
&+\frac{\gamma_i}{\sqrt{a}}
\end{aligned}
$$

We know that $|x_i\sqrt{a}| \leq \varepsilon_i\sqrt{a}+1$ and $\left|x_i^3a\right| \leq \frac{(\varepsilon_i\sqrt{a}+1)^3}{\sqrt{a}}$ and moreover $\left|\frac{1}{2}x_i(1-x_i^2a)\right| \leq \frac{1}{2}\left(\varepsilon_i+\frac{1}{\sqrt{a}}\right)\left(a\varepsilon_i^2+2\sqrt{a}\varepsilon_i\right)$, so we have:

$$
\begin{aligned}
\varepsilon_{i+1} \quad \leq \quad & \frac{1}{2}(1+\gamma_i)\sqrt{a}(\varepsilon_i\sqrt{a}+3)\varepsilon_i^2 \\
& +\gamma_i(1+\gamma_i)(2+\gamma_i)\frac{\varepsilon_i}{2}\left(\varepsilon_i+\frac{1}{\sqrt{a}}\right)\left(a\varepsilon_i+2\sqrt{a}\right) \\
& +\frac{1}{2}\gamma_i(1+\gamma_i)^3(3+3\gamma_i+\gamma_i^2)\frac{(\varepsilon_i\sqrt{a}+1)^3}{\sqrt{a}} \\
& +\frac{\gamma_i}{\sqrt{a}}. \qquad (51)
\end{aligned}
$$

Using the notation $E_i = \varepsilon_i\sqrt{a}$ we can transform (51) in an equation independent of a:

$$
\begin{aligned}
E_{i+1} \quad \leq \quad & \frac{1}{2}(1+\gamma_i)(E_i+3)E_i^2 \\
& +\gamma_i(1+\gamma_i)(2+\gamma_i)\frac{E_i}{2}\left(E_i+1\right)\left(E_i+2\right) \\
& +\frac{1}{2}\gamma_i(1+\gamma_i)^3(3+3\gamma_i+\gamma_i^2)(E_i+1)^3 \\
& +\gamma_i.
\end{aligned}
$$

For the last part of the proof, we proceed exactly like in the previous proof and are able to find the same bound $ind(i) = 2^{-2^i(p-3)-1}$ by applying exactly the same inductive reasoning.