# Accurate Floating Point Product and Exponentiation

Stef Graillat

# Accurate Floating Point Product and Exponentiation

**Stef Graillat***

July 23, 2007

**Abstract**

Several different techniques and softwares intend to improve the accuracy of results computed in a fixed finite precision. Here we focus on a method to improve the accuracy of the product of floating point numbers. We show that the computed result is as accurate as if computed in twice the working precision. The algorithm is simple since it only requires addition, subtraction and multiplication of floating point numbers in the same working precision as the given data. Such an algorithm can be useful for example to compute the determinant of a triangular matrix and to evaluate a polynomial when represented by the root product form. It can also be used to compute the power of a floating point number.

**Key words:** accurate product, exponentiation, finite precision, floating point arithmetic, faithful rounding, error-free transformations

**AMS Subject Classifications:** 65-04, 65G20, 65G50

## 1 Introduction

In this paper, we present fast and accurate algorithms to compute the product of floating point numbers. Our aim is to increase the accuracy at a fixed precision. We show that the results have the same error estimates as if computed in twice the working precision and then rounded to working precision. Then we address the problem on how to compute a faithfully rounded result, that is to say one of the two adjacent floating point numbers of the exact result.

This paper was motived by papers [15, 18, 6, 12] and [11] where similar approaches are used to compute summation, dot product, polynomial evaluation and power.

The applications of our algorithms are multiple. One of the examples frequently used in Sterbenz's book [20] is the computation of the product of some floating point numbers. Our algorithms can be used to compute the determinant of a triangle matrix

$$T = \begin{bmatrix} t_{11} & t_{12} & \cdots & t_{1n} \\ & t_{22} & & t_{2n} \\ & & \ddots & \vdots \\ & & & t_{nn} \end{bmatrix}.$$

Indeed, the determinant of $T$ is

$$\det(T) = \prod_{i=1}^{n} t_{ii}.$$

*Laboratoire LIP6, Département Calcul Scientifique, Université Pierre et Marie Curie (Paris 6), 4 place Jussieu, F-75252, Paris cedex 05, France (stef.graillat@lip6.fr,http://www-pequan.lip6.fr/~graillat).

Another application is for evaluating a polynomial when represented by the root product form $p(x) = a_n \prod_{i=1}^{n}(x - x_i)$. We can also apply our algorithms to compute the power of a floating point number.

The rest of the paper is organized as follows. In Section 2, we recall notations and auxiliary results that will be needed in the sequel. We present the floating point arithmetic and the so-called error-free transformations. In Section 3, we present a classic algorithm to compute the product of floating point numbers. We give an error estimate as well as a validated error bound. We also present a new compensated algorithm together with an error estimate and a validated error bound. We show that under mild assumptions, our algorithm gives a faithfully rounded result. In Section 4, we apply our algorithm to compute the power of a floating point number. We propose two different algorithms: one with our compensated algorithm, the other one with the use of a double-double library.

# 2 Notation and auxiliary results

## 2.1 Floating point arithmetic

Throughout the paper, we assume to work with a floating point arithmetic adhering to IEEE 754 floating point standard in rounding to nearest [9]. We assume that no overflow nor underflow occurs. The set of floating point numbers is denoted by $\mathbb{F}$, the relative rounding error by $\texttt{eps}$. For IEEE 754 double precision, we have $\texttt{eps} = 2^{-53}$ and for single precision $\texttt{eps} = 2^{-24}$.

We denote by $\mathrm{fl}(\cdot)$ the result of a floating point computation, where all operations inside parentheses are done in floating point working precision. Floating point operations in IEEE 754 satisfy [8]

$$\mathrm{fl}(a \circ b) = (a \circ b)(1 + \varepsilon_1) = (a \circ b)/(1 + \varepsilon_2) \text{ for } \circ = \{+, -, \cdot, /\} \text{ and } |\varepsilon_\nu| \leq \texttt{eps}.$$

This implies that

$$|a \circ b - \mathrm{fl}(a \circ b)| \leq \texttt{eps}|a \circ b| \text{ and } |a \circ b - \mathrm{fl}(a \circ b)| \leq \texttt{eps}\,|\mathrm{fl}(a \circ b)| \text{ for } \circ = \{+, -, \cdot, /\} \quad (2.1)$$

## 2.2 Error-free transformations

One can notice that $a \circ b \in \mathbb{R}$ and $\mathrm{fl}(a \circ b) \in \mathbb{F}$ but in general we do not have $a \circ b \in \mathbb{F}$. It is known that for the basic operations $+, -, \cdot$, the approximation error of a floating point operation is still a floating point number (see for example [5]):

$$\begin{array}{lcll} x = \mathrm{fl}(a \pm b) & \Rightarrow & a \pm b = x + y & \text{with } y \in \mathbb{F}, \\ x = \mathrm{fl}(a \cdot b) & \Rightarrow & a \cdot b = x + y & \text{with } y \in \mathbb{F}. \end{array} \quad (2.2)$$

These are *error-free* transformations of the pair $(a, b)$ into the pair $(x, y)$.

Fortunately, the quantities $x$ and $y$ in (2.2) can be computed exactly in floating point arithmetic. For the algorithms, we use Matlab-like notations. For addition, we can use the following algorithm by Knuth [10, Thm B. p.236].

**Algorithm 2.1 (Knuth [10]).** Error-free transformation of the sum of two floating point numbers

function $[x, y] = \texttt{TwoSum}(a, b)$
$x = \mathrm{fl}(a + b)$
$z = \mathrm{fl}(x - a)$
$y = \mathrm{fl}((a - (x - z)) + (b - z))$

Another algorithm to compute an error-free transformation is the following algorithm from Dekker [5]. The drawback of this algorithm is that we have $x+y = a+b$ provided that $|a| \geq |b|$. Generally, on modern computers, a comparison followed by a branching and 3 operations costs more than 6 operations. As a consequence, `TwoSum` is generally more efficient than `FastTwoSum`.

**Algorithm 2.2 (Dekker [5]).** Error-free transformation of the sum of two floating point numbers.

function $[x, y] = \texttt{FastTwoSum}(a, b)$
  $x = \mathrm{fl}(a + b)$
  $y = \mathrm{fl}((a - x) + b)$

For the error-free transformation of a product, we first need to split the input argument into two parts. Let $p$ be given by $\texttt{eps} = 2^{-p}$ and define $s = \lceil p/2 \rceil$. For example, if the working precision is IEEE 754 double precision, then $p = 53$ and $s = 27$. The following algorithm by Dekker [5] splits a floating point number $a \in \mathbb{F}$ into two parts $x$ and $y$ such that

$$a = x + y \quad \text{and} \quad x \text{ and } y \text{ nonoverlapping with } |y| \leq |x|.$$

**Algorithm 2.3 (Dekker [5]).** Error-free split of a floating point number into two parts

function $[x, y] = \texttt{Split}(a, b)$
  $\texttt{factor} = \mathrm{fl}(2^s + 1)$
  $c = \mathrm{fl}(\texttt{factor} \cdot a)$
  $x = \mathrm{fl}(c - (c - a))$
  $y = \mathrm{fl}(a - x)$

With this function, an algorithm from Veltkamp (see [5]) enables to compute an error-free transformation for the product of two floating point numbers. This algorithm returns two floating point numbers $x$ and $y$ such that

$$a \cdot b = x + y \quad \text{with } x = \mathrm{fl}(a \cdot b).$$

**Algorithm 2.4 (Veltkamp [5]).** Error-free transformation of the product of two floating point numbers

function $[x, y] = \texttt{TwoProduct}(a, b)$
  $x = \mathrm{fl}(a \cdot b)$
  $[a_1, a_2] = \texttt{Split}(a)$
  $[b_1, b_2] = \texttt{Split}(b)$
  $y = \mathrm{fl}(a_2 \cdot b_2 - (((x - a_1 \cdot b_1) - a_2 \cdot b_1) - a_1 \cdot b_2))$

The following theorem summarizes the properties of algorithms `TwoSum` and `TwoProduct`.

**Theorem 2.1 (Ogita, Rump and Oishi [15]).** *Let $a, b \in \mathbb{F}$ and let $x, y \in \mathbb{F}$ such that $[x, y] = \texttt{TwoSum}(a, b)$ (Algorithm 2.1). Then,*

$$a + b = x + y, \quad x = \mathrm{fl}(a + b), \quad |y| \leq \texttt{eps}|x|, \quad |y| \leq \texttt{eps}|a + b|. \tag{2.3}$$

*The algorithm* `TwoSum` *requires* 6 *flops.*
*Let $a, b \in \mathbb{F}$ and let $x, y \in \mathbb{F}$ such that $[x, y] = \texttt{TwoProduct}(a, b)$ (Algorithm 2.4). Then,*

$$a \cdot b = x + y, \quad x = \mathrm{fl}(a \cdot b), \quad |y| \leq \texttt{eps}|x|, \quad |y| \leq \texttt{eps}|a \cdot b|. \tag{2.4}$$

*The algorithm* `TwoProduct` *requires* 17 *flops.*

The `TwoProduct` algorithm can be re-written in a very simple way if a Fused-Multiply-and-Add (`FMA`) operator is available on the targeted architecture [14]. Some computers have a *Fused-Multiply-and-Add* (`FMA`) operation that enables a floating point multiplication followed by an addition to be performed as a single floating point operation. The Intel IA-64 architecture, implemented in the Intel Itanium processor, has an `FMA` instruction as well as the IBM RS/6000 and the PowerPC before it. On the Itanium processor, the `FMA` instruction enables a multiplication and an addition to be performed in the same number of cycles than one multiplication or one addition. As a result, it seems to be advantageous for speed as well as for accuracy.

Theoretically, this means that for $a, b, c \in \mathbb{F}$, the result of $\texttt{FMA}(a, b, c)$ is the nearest floating point number of $a \cdot b + c \in \mathbb{R}$. The `FMA` satisfies

$$\texttt{FMA}(a, b, c) = (a \cdot b + c)(1 + \varepsilon_1) = (a \cdot b + c)/(1 + \varepsilon_2) \text{ with } |\varepsilon_\nu| \leq \texttt{eps}.$$

Thanks to the `FMA`, the `TwoProduct` algorithm can be re-written as follows which costs only 2 flops.

**Algorithm 2.5 (Ogita, Rump and Oishi [15]).** Error-free transformation of the product of two floating point numbers using an `FMA`.

function $[x, y] = \texttt{TwoProductFMA}(a, b)$
  $x = a \cdot b$
  $y = \texttt{FMA}(a, b, -x)$

# 3 Accurate floating point product

In this section, we present a new accurate algorithm to compute the product of floating point numbers. In Subsection 3.1, we recall the classic method and we give a theoretical error bound as well as a validated computable error bound. In Subsection 3.2, we present our new algorithm based on a compensated scheme together with a theoretical error bound. In Subsection 3.3, we give sufficient conditions on the number of floating point numbers so as to get a faithfully rounded result. Finally, in Subsection 3.4, we give a validated computable error bound for our new algorithm.

## 3.1 Classic method

The classic method for evaluating a product of $n$ numbers $a = (a_1, a_2, \ldots, a_n)$

$$p = \prod_{i=1}^{n} a_i$$

is the following algorithm.

**Algorithm 3.1.** Product evaluation

function $\texttt{res} = \texttt{Prod}(a)$
  $p_1 = a_1$
  for $i = 2 : n$
    $p_i = \text{fl}(p_{i-1} \cdot a_i)$
  end
  $\texttt{res} = p_n$

This algorithm requires $n-1$ flops. Let us now analyse its accuracy.

We will use standard notations and standard results for the following error estimations (see [8]). The quantities $\gamma_n$ are defined as usual [8] by

$$\gamma_n := \frac{n\mathtt{eps}}{1-n\mathtt{eps}} \quad \text{for } n \in \mathbb{N}.$$

When using $\gamma_n$, we implicitly assume that $n\mathtt{eps} \leq 1$. A forward error bound is

$$|a_1 a_2 \cdots a_n - \mathtt{res}| = |a_1 a_2 \cdots a_n - \mathrm{fl}(a_1 a_2 \cdots a_n)| \leq \gamma_{n-1} |a_1 a_2 \cdots a_n|. \tag{3.5}$$

Indeed, by induction,

$$\mathtt{res} = \mathrm{fl}(a_1 a_2 \cdots a_n) = a_1 a_2 \cdots a_n (1+\varepsilon_2)(1+\varepsilon_3)\cdots(1+\varepsilon_n), \tag{3.6}$$

with $\varepsilon_i \leq \mathtt{eps}$ for $i = 2:n$. It follows from Lemma 3.1 of [8, p.63] that $(1+\varepsilon_2)(1+\varepsilon_3)\cdots(1+\varepsilon_n) = 1+\theta_n$ where $|\theta_{n-1}| \leq \gamma_{n-1}$.

A convenient device for keeping track of power of $1+\varepsilon$ term is describe in [8, p.68]. The relative error counter $\langle k \rangle$ denotes the product

$$\langle k \rangle = \prod_{i=1}^{k} (1+\varepsilon_i), \quad |\varepsilon_i| \leq \mathtt{eps}.$$

A useful rule for the counter is $\langle j \rangle \langle k \rangle = \langle j+k \rangle$. Using this notation, Equation (3.6) can be written $\mathtt{res} = \mathrm{fl}(a_1 a_2 \cdots a_n) = a_1 a_2 \cdots a_n \langle n-1 \rangle$.

It is shown in [16] that for $a \in \mathbb{F}$, we have

$$\begin{aligned} (1+\mathtt{eps})^n &\leq \frac{1}{(1-\mathtt{eps})^n} \leq \frac{1}{1-n\mathtt{eps}}, \\ \frac{|a|}{1-n\mathtt{eps}} &\leq \mathrm{fl}\left(\frac{|a|}{1-(n+1)\mathtt{eps}}\right). \end{aligned} \tag{3.7}$$

From Equation (3.6), it follows that

$$|a_1 a_2 \cdots a_n - \mathtt{res}| \leq (1+\mathtt{eps})^{n-1} \gamma_{n-1} |\mathtt{res}|.$$

If $m\mathtt{eps} \leq 1$ for $m \in \mathbb{N}$, $\mathrm{fl}(m\mathtt{eps}) = m\mathtt{eps}$ and $\mathrm{fl}(1-m\mathtt{eps}) = 1-m\mathtt{eps}$. Therefore,

$$\gamma_m \leq (1+\mathtt{eps})\,\mathrm{fl}(\gamma_m). \tag{3.8}$$

Hence,

$$\begin{aligned} |a_1 a_2 \cdots a_n - \mathtt{res}| &\leq (1+\mathtt{eps})^n\,\mathrm{fl}(\gamma_{n-1})|\mathtt{res}| \\ &\leq (1+\mathtt{eps})^{n+1}\,\mathrm{fl}(\gamma_{n-1}|\mathtt{res}|), \end{aligned}$$

and so

$$|a_1 a_2 \cdots a_n - \mathtt{res}| \leq \mathrm{fl}\left(\frac{\gamma_{n-1}|\mathtt{res}|}{1-(n+2)\mathtt{eps}}\right).$$

The previous inequality gives us a validated error bound that can be computed in pure floating point arithmetic in rounding to nearest.

### 3.2 Compensated method

We present hereafter a compensated scheme to evaluate the product of floating point numbers, i.e. the error of individual multiplication is somehow corrected. The technique used here is based on the paper [15].

**Algorithm 3.2.** Product evaluation with a compensated scheme

function `res` = `CompProd`$(a)$
  $p_1 = a_1$
  $e_1 = 0$
  for $i = 2 : n$
    $[p_i, \pi_i] =$ `TwoProduct`$(p_{i-1}, a_i)$
    $e_i = \mathrm{fl}(e_{i-1}a_i + \pi_i)$
  end
  `res` $= \mathrm{fl}(p_n + e_n)$

This algorithm requires $19n - 18$ flops if we use `TwoProduct`. It only requires $4n - 3$ flops if we use `TwoProductFMA` instead of `TwoProduct` (if, of course, an `FMA` is available).

**Algorithm 3.3.** Product evaluation with a compensated scheme with `TwoProductFMA`

function `res` = `CompProdFMA1`$(a)$
  $p_1 = a_1$
  $e_1 = 0$
  for $i = 2 : n$
    $[p_i, \pi_i] =$ `TwoProductFMA`$(p_{i-1}, a_i)$
    $e_i = \mathrm{fl}(e_{i-1}a_i + \pi_i)$
  end
  `res` $= \mathrm{fl}(p_n + e_n)$

It finally costs $3n - 2$ flops if we also use $e_i =$ `FMA`$(e_{i-1}, a_i, \pi_i)$ instead of $e_i = \mathrm{fl}(e_{i-1}a_i + \pi_i)$.

**Algorithm 3.4.** Product evaluation with a compensated scheme with `TwoProductFMA` and `FMA`

function `res` = `CompProdFMA2`$(a)$
  $p_1 = a_1$
  $e_1 = 0$
  for $i = 2 : n$
    $[p_i, \pi_i] =$ `TwoProductFMA`$(p_{i-1}, a_i)$
    $e_i =$ `FMA`$(e_{i-1}, a_i, \pi_i)$
  end
  `res` $= \mathrm{fl}(p_n + e_n)$

We will provide an error analysis only for Algorithm `CompProd`. The error analysis for `CompProdFMA1` is the same as `CompProd` since they share the sames operations. For the error analysis of `CompProdFMA2`, little changes must be done to take into account the operation $e_i =$ `FMA`$(e_{i-1}, a_i, \pi_i)$. This changes nearly nothing so it is straightforward to modify the analysis to deal with it.

For error analysis, we note that

$$p_n = \mathrm{fl}(a_1 a_2 \cdots a_n) \quad \text{and} \quad e_n = \mathrm{fl}\left(\sum_{i=2}^{n} \pi_i a_{i+1} \cdots a_n\right).$$

We also have

$$p = a_1 a_2 \dots a_n = \mathrm{fl}(a_1 a_2 \dots a_n) + \sum_{i=2}^{n} \pi_i a_{i+1} \cdots a_n = p_n + e, \tag{3.9}$$

where $e = \sum_{i=2}^{n} \pi_i a_{i+1} \cdots a_n$.

Before proving the main theorem, we will need two technical lemmas. The next lemma makes it possible to obtain a bound on the individual error of the multiplication namely $\pi_i$ in function of the inital data $a_i$.

**Lemma 3.1.** *Suppose floating point numbers $\pi_i \in \mathbb{F}$, $2 \leq i \leq n$ are computed by the following algorithm*

$p_1 = a_1$
for $i = 2 : n$
  $[p_i, \pi_i] = \texttt{TwoProduct}(p_{i-1}, a_i)$
end

*Then,*

$$|\pi_i| \leq \texttt{eps}(1 + \gamma_{i-1})|a_1 \cdots a_i| \quad \textit{for } i = 2 : n.$$

*Proof.* From Equation (2.1), it follows that

$$|\pi_i| \leq \texttt{eps}|p_i|.$$

Moreover, $p_i = \mathrm{fl}(a_1 \cdots a_i)$ so that from (3.5),

$$|p_i| \leq (1 + \gamma_{i-1})|a_1 \cdots a_i|.$$

Hence, $|\pi_i| \leq \texttt{eps}(1 + \gamma_{i-1})|a_1 \cdots a_i|$. □

The following lemma enables us to bound the rounding errors during the computation of the error during the full product.

**Lemma 3.2.** *Suppose floating point numbers $e_i \in \mathbb{F}$, $1 \leq i \leq n$ are computed by the following algorithm*

$e_1 = 0$
for $i = 2 : n$
  $[p_i, \pi_i] = \texttt{TwoProduct}(p_{i-1}, a_i)$
  $e_i = \mathrm{fl}(e_{i-1} a_i + \pi_i)$
end

*Then,*

$$|e_n - \sum_{i=2}^{n} \pi_i a_{i+1} \cdots a_n| \leq \gamma_{n-1}\gamma_{2n}|a_1 a_2 \cdots a_n|.$$

*Proof.* First, one notices that $e_n = \mathrm{fl}(\sum_{i=2}^{n}(\pi_i a_{i+1} \cdots a_n))$. We will use the error counters described above. For $n$ floating point numbers $x_i$, it is easy to see that [8, chap.4]

$$\mathrm{fl}(x_1 + x_2 + \cdots + x_n) = x_1\langle n-1\rangle + x_2\langle n-1\rangle + x_3\langle n-2\rangle + \cdots + x_n\langle 1\rangle.$$

This implies that

$$e_n = \mathrm{fl}(\sum_{i=2}^{n}(\pi_i a_{i+1} \cdots a_n)) = \mathrm{fl}(\pi_2 a_3 \cdots a_n)\langle n-2\rangle + \mathrm{fl}(\pi_3 a_4 \cdots a_n)\langle n-2\rangle + \cdots + \mathrm{fl}(\pi_n)\langle 1\rangle.$$

Furthermore, we have shown before that $\mathrm{fl}(a_1a_2\cdots a_n) = a_1a_2\cdots a_n\langle n-1\rangle$. Consequently,

$$e_n = \pi_2 a_3\cdots a_n\langle n-2\rangle\langle n-1\rangle + \pi_3 a_4\cdots a_n\langle n-3\rangle\langle n-1\rangle + \cdots + \pi_n\langle 1\rangle.$$

A straightforward computation yields

$$|e_n - \sum_{i=2}^{n} \pi_i a_{i+1}\cdots a_n| \le \gamma_{2n-3}\sum_{i=2}^{n}|\pi_i a_{i+1}\cdots a_n|.$$

From Lemma 3.1, we have $|\pi_i| \le \mathtt{eps}(1+\gamma_{i-1})|a_1\cdots a_i|$ and hence

$$|e_n - \sum_{i=2}^{n} \pi_i a_{i+1}\cdots a_n| \le (n-1)\mathtt{eps}(1+\gamma_{n-1})\gamma_{2n-3}|a_1a_2\cdots a_n|.$$

Since $\mathtt{eps}(1+\gamma_{n-1}) = \gamma_{n-1}/(n-1)$ and $\gamma_{2n-3} \le \gamma_{2n}$, we obtain the desired result. □

One may notice that the computation of $e_n$ is similar to the Horner scheme. One could have directly applied a result on the error of the Horner scheme [8, Eq.(5.3),p.95].

We can finally state the main theorem.

**Theorem 3.3.** *Suppose Algorithm 3.2 is applied to floating point number $a_i \in \mathbb{F}$, $1 \le i \le n$, and set $p = \prod_{i=1}^{n} a_i$. Then,*

$$|\mathtt{res} - p| \le \mathtt{eps}|p| + \gamma_n\gamma_{2n}|p|.$$

*Proof.* The fact that $\mathtt{res} = \mathrm{fl}(p_n + e_n)$ implies that $\mathtt{res} = (1+\varepsilon)(p_n+e_n)$ with $\varepsilon \le \mathtt{eps}$. So it follows

$$\begin{aligned}
|\mathtt{res} - p| &= |\mathrm{fl}(p_n+e_n) - p| = |(1+\varepsilon)(p_n + e_n - p) + \varepsilon p| \\
&= |(1+\varepsilon)(p_n + \sum_{i=2}^{n}\pi_i a_{i+1}\cdots a_n - p) + (1+\varepsilon)(e_n - \sum_{i=2}^{n}\pi_i a_{i+1}\cdots a_n) + \varepsilon p| \\
&= |(1+\varepsilon)(e_n - \sum_{i=2}^{n}\pi_i a_{i+1}\cdots a_n) + \varepsilon p| \quad \text{by (3.9)} \\
&\le \mathtt{eps}|p| + (1+\mathtt{eps})|e_n - \sum_{i=2}^{n}\pi_i a_{i+1}\cdots a_n| \\
&\le \mathtt{eps}|p| + (1+\mathtt{eps})\gamma_{n-1}\gamma_{2n}|a_1a_2\cdots a_n|.
\end{aligned}$$

Since $(1+\mathtt{eps})\gamma_{n-1} \le \gamma_n$, it follows that $|\mathtt{res} - p| \le \mathtt{eps}|p| + \gamma_n\gamma_{2n}|p|$. □

It may be interesting to study the condition number of the product evaluation. Ones defines

$$\mathrm{cond}(a) = \lim_{\varepsilon\to 0}\sup\left\{\frac{|(a_1+\Delta a_1)(a_2+\Delta a_2)\cdots(a_n+\Delta a_n) - a_1a_2\cdots a_n|}{\varepsilon|a_1a_2\cdots a_n|} : |\Delta a_i| \le \varepsilon|a_i|\right\}.$$

A standard computation yields

$$\mathrm{cond}(a) = n.$$

**Corollary 3.4.** *Suppose Algorithm 3.2 is applied to floating point number $a_i \in \mathbb{F}$, $1 \le i \le n$, and set $p = \prod_{i=1}^{n} a_i$. Then,*

$$\frac{|\mathtt{res} - p|}{|p|} \le \mathtt{eps} + \frac{\gamma_n\gamma_{2n}}{n}\mathrm{cond}(a).$$

### 3.3 Faithful rounding

We define the floating point predecessor and successor of a real number $r$ satisfying $\min\{f : f \in \mathbb{R}\} < r < \max\{f : f \in \mathbb{F}\}$ by

$$\mathrm{pred}(r) := \max\{f \in \mathbb{F} : f < r\} \quad \text{and} \quad \mathrm{succ}(r) := \min\{f \in \mathbb{F} : r < f\}.$$

**Definition 3.1.** A floating point number $f \in \mathbb{F}$ is called a *faithful rounding* of a real number $r \in \mathbb{R}$ if

$$\mathrm{pred}(f) < r < \mathrm{succ}(f).$$

We denote this by $f \in \square(r)$. For $r \in \mathbb{F}$, this implies that $f = r$.

A faithful rounding is then one of the two adjacent floating point numbers of the exact result.

**Lemma 3.5 (Rump, Ogita and Oishi [18, lem. 2.5]).** *Let $r, \delta \in \mathbb{R}$ and $\widetilde{r} := \mathrm{fl}(r)$. Suppose that $2|\delta| < \mathtt{eps}|\widetilde{r}|$. Then $\widetilde{r} \in \square(r + \delta)$, that means $\widetilde{r}$ is a faithful rounding of $r + \delta$.*

Let `res` be the result of `CompProd`. Then we have $p = p_n + e$ and $\mathtt{res} = \mathrm{fl}(p_n + e_n)$ with $e = \sum_{i=2}^{n} \pi_i a_{i+1} \cdots a_n$. It follows that $p = (p_n + e_n) + (e - e_n)$. This leads to the following lemma which gives a criterion to ensure that the result of `CompProd` is faithfully rounded.

**Lemma 3.6.** *With the previous notations, if $2|e - e_n| < \mathtt{eps}|\mathtt{res}|$ then `res` is a faithful rounding of $p$.*

Since we have $|e - e_n| \leq \gamma_n\gamma_{2n}|p|$ and $(1 - \mathtt{eps})|p| - \gamma_n\gamma_{2n}|p| \leq |\mathtt{res}|$, a sufficient condition to ensure a faithful rounding is

$$2\gamma_n\gamma_{2n}|p| < \mathtt{eps}((1 - \mathtt{eps})|p| - \gamma_n\gamma_{2n}|p|)$$

that is

$$\gamma_n\gamma_{2n} < \frac{1 - \mathtt{eps}}{2 + \mathtt{eps}}\mathtt{eps}.$$

Since $\gamma_n\gamma_{2n} \leq 2(n\mathtt{eps})^2/(1 - 2n\mathtt{eps})^2$, a sufficient condition is

$$2\frac{(n\mathtt{eps})^2}{(1 - 2n\mathtt{eps})^2} < \frac{1 - \mathtt{eps}}{2 + \mathtt{eps}}\mathtt{eps}$$

which is equivalent to

$$\frac{n\mathtt{eps}}{1 - 2n\mathtt{eps}} < \sqrt{\frac{(1 - \mathtt{eps})\mathtt{eps}}{2(2 + \mathtt{eps})}}$$

and then to

$$n < \frac{\sqrt{1 - \mathtt{eps}}}{\sqrt{2}\sqrt{2 + \mathtt{eps}} + 2\sqrt{(1 - \mathtt{eps})\mathtt{eps}}}\mathtt{eps}^{-1/2}.$$

We have just shown that if $n < \alpha\mathtt{eps}^{-1/2}$ where $\alpha \approx 1/2$ then the result is faithfully rounded. More precisely, in double precision where $\mathtt{eps} = 2^{-53}$, if $n < 2^{25} \approx 5 \cdot 10^7$, we get a faithfully rounded result.

### 3.4 Validated error bound

We present here how to compute a valid error bound in pure floating point arithmetic in rounding to nearest. It holds that

$$\begin{aligned} |\mathtt{res} - p| &= |\,\mathrm{fl}(p_n + e_n) - p| = |\,\mathrm{fl}(p_n + e_n) - (p_n + e_n) + (p_n + e_n) - p| \\ &\leq \mathtt{eps}|\mathtt{res}| + |p_n + e_n - p| \\ &\leq \mathtt{eps}|\mathtt{res}| + |e_n - e|. \end{aligned}$$

Since $|e_n - e| \leq \gamma_{n-1}\gamma_{2n}|p|$ and $|p| \leq (1+\mathtt{eps})^{n-1}\,\mathrm{fl}(|a_1 a_2 \cdots a_n|)$ we obtain

$$\begin{aligned} |\mathtt{res} - p| &\leq \mathtt{eps}|\mathtt{res}| + \gamma_{n-1}\gamma_{2n}|p| \\ &\leq \mathtt{eps}|\mathtt{res}| + \gamma_{n-1}\gamma_{2n}(1+\mathtt{eps})^{n-1}\,\mathrm{fl}(|a_1 a_2 \cdots a_n|). \end{aligned}$$

Using (3.7) and (3.8), we get

$$\begin{aligned} |\mathtt{res} - p| &\leq \mathrm{fl}(\mathtt{eps}|\mathtt{res}|) + (1+\mathtt{eps})^{n}\,\mathrm{fl}(\gamma_n)\,\mathrm{fl}(\gamma_{2n})\,\mathrm{fl}(|a_1 a_2 \cdots a_n|) \\ &\leq \mathrm{fl}(\mathtt{eps}|\mathtt{res}|) + (1+\mathtt{eps})^{n+2}\,\mathrm{fl}(\gamma_n\gamma_{2n}|a_1 a_2 \cdots a_n|) \\ &\leq \mathrm{fl}(\mathtt{eps}|\mathtt{res}|) + \mathrm{fl}\left(\frac{\gamma_n\gamma_{2n}|a_1 a_2 \cdots a_n|}{1-(n+3)\mathtt{eps}}\right) \\ &\leq (1+\mathtt{eps})\,\mathrm{fl}\left(\mathtt{eps}|\mathtt{res}| + \frac{\gamma_n\gamma_{2n}|a_1 a_2 \cdots a_n|}{1-(n+3)\mathtt{eps}}\right) \\ &\leq \mathrm{fl}\left(\left(\mathtt{eps}|\mathtt{res}| + \frac{\gamma_n\gamma_{2n}|a_1 a_2 \cdots a_n|}{1-(n+3)\mathtt{eps}}\right) / \,(1-2\mathtt{eps})\right). \end{aligned}$$

We can summarize this as follows.

**Lemma 3.7.** *Suppose Algorithm 3.2 is applied to floating point numbers $a_i \in \mathbb{F}$, $1 \leq i \leq n$ and set $p = \prod_{i=1}^{n} a_i$. Then, the absolute forward error affecting the product is bounded according to*

$$|\mathtt{res} - p| \leq \mathrm{fl}\left(\left(\mathtt{eps}|\mathtt{res}| + \frac{\gamma_n\gamma_{2n}|a_1 a_2 \cdots a_n|}{1-(n+3)\mathtt{eps}}\right) / \,(1-2\mathtt{eps})\right).$$

### 3.5 Validated error bound and faithful rounding

In the previous subsection, we have shown that

$$|e_n - e| \leq \mathrm{fl}\left(\frac{\gamma_n\gamma_{2n}|a_1 a_2 \cdots a_n|}{1-(n+3)\mathtt{eps}}\right). \tag{3.10}$$

Lemma 3.6 tells us that if $2|e - e_n| < \mathtt{eps}|\mathtt{res}|$ then `res` is a faithful rounding of $p$ (where `res` is the result of `CompProd`).

As a consequence, if

$$\mathrm{fl}\left(2\frac{\gamma_n\gamma_{2n}|a_1 a_2 \cdots a_n|}{1-(n+3)\mathtt{eps}}\right) < \mathrm{fl}(\mathtt{eps}|\mathtt{res}|)$$

then we got a faitfully rounded result. This makes it possible to check *a posteriori* if the result is faithfully rounded.

## 4 Exponentiation

In this section, we study two exponentiation algorithms (for computing $x^n$ with $x \in \mathbb{F}$ and $n \in \mathbb{N}$). The first one is linear (in $\mathcal{O}(n)$) whereas the second one is logarithmic (in $\mathcal{O}(\log n)$).

### 4.1 A linear algorithm

The natural method to compute $x^n$ is to apply algorithm `CompProd` for $a_i \in \mathbb{F}$ with $a_i = x$ for $1 \leq i \leq n$. This leads to the following algorithm.

**Algorithm 4.1.** Power evaluation with a compensated scheme

function `res` = `CompLinPower`$(x, n)$
  $p_1 = x$
  $e_1 = 0$
  for $i = 2 : n$
    $[p_i, \pi_i] =$ `TwoProduct`$(p_{i-1}, x)$
    $e_i = \mathrm{fl}(e_{i-1}x + \pi_i)$
  end
  `res` $= \mathrm{fl}(p_n + e_n)$

All that has been done concerning `CompProd` is still valid with `CompLinPower`. For example, if $n < 2^{25}$ then the result is faithfully rounded.

This algorithm is similar to the one of [11]. It is also the same as the Compensated Horner scheme [6] applied to the polynomial $p(x) = x^n$. Results concerning faithful polynomial evaluation can be found in [12].

### 4.2 A double-double library

Compensated methods are a possible way to improve the accuracy. Another possibility is to increase the working precision. For this purpose, some multiprecision libraries have been developed. One can divide those libraries into three categories.

- Arbitrary precision library using a *multiple-digit* format where a number is expressed as a sequence of digits coupled with a single exponent. Examples of this format are Bailey's MPFUN [2], Brent's MP [4] or MPFR [1].
- Arbitrary precision library using a *multiple-component* format where a number is expressed as unevaluated sums of ordinary floating point words. Examples of this format are Priest [17] and Shewchuk [19].
- Extended fixed precision library using the *multiple-component* format but with a limited number of components. Examples of this format are Bailey's *double-double* [3] (double-double numbers are represented as an unevaluated sum of a leading double and a trailing double) and *quad-double* [7] (quad-double numbers are represented as an unevaluated sum of four IEEE doubles).

In the sequel, we present two algorithms to compute product of two double-double or a double times a double-double. Those algorithms are taken from [13].

**Algorithm 4.2.** Multiplication of two double-double numbers

function $[r_h, r_l] =$ `prod_dd_dd`$(a_h, a_l, b_h, b_l)$
  $[t_1, t_2] =$ `TwoProduct`$(a_h, b_h)$
  $t_3 = \mathrm{fl}(((a_h \cdot b_l) + (a_l \cdot b_h)) + t_2)$
  $[r_h, r_l] =$ `TwoProduct`$(t_1, t_3)$

**Algorithm 4.3.** Multiplication of double-double number by a double number

function $[r_h, r_l] = \texttt{prod\_dd\_d}(a, b_h, b_l)$
  $[t_1, t_2] = \texttt{TwoProduct}(a, b_h)$
  $t_3 = \mathrm{fl}((a \cdot b_l) + t_2)$
  $[r_h, r_l] = \texttt{TwoProduct}(t_1, t_3)$

**Theorem 4.1 (Lauter [13, thm. 4.7]).** *Let be $a_h + a_l$ and $b_h + b_l$ the double-double arguments of Algorithm 4.2. Then the returned values $r_h$ and $r_l$ satisfy*

$$r_h + r_l = ((a_h + a_l) \cdot (b_h + b_l))(1 + \varepsilon)$$

*where $\varepsilon$ is bounded as follows : $|\varepsilon| \leq 16\texttt{eps}^2$. Furthermore, we have $|r_l| \leq \texttt{eps}|r_h|$.*

Results for Algorithm 4.3 are very similar with $a = a_h$ and $a_l = 0$.

### 4.3 A logarithmic algorithm

A logarithmic algorithm was introduced in [11] using the classic right-to-left binary exponentiation algorithm and the double-double library. Hereafter, we propose a variant of this algorithm with the left-to-right binary exponentiation algorithm together with the double-double library. Contrary to the right-to-left binary exponentiation algorithm which needs two multiplications of two double-double numbers, the left-to-right binary exponentiation algorithm only needs a multiplication of two double-double numbers and a multiplication of a double number by a double-double numbers. Moreover, the multiplication of a double-double by a double-double is actually a square so that it can be a little bit optimized.

**Algorithm 4.4.** Power evaluation with a compensated scheme

function $\texttt{res} = \texttt{CompLogPower}(x, n)$ % $n = (n_t n_{t-1} \cdots n_1 n_0)_2$
  $[h, l] = [1, 0]$
  for $i = t : -1 : 0$
    $[h, l] = \texttt{prod\_dd\_dd}(h, l, h, l)$
    if $n_i = 1$
      $[h, l] = \texttt{prod\_dd\_d}(x, h, l)$
    end
  end
  $\texttt{res} = \mathrm{fl}(h + l)$

**Theorem 4.2.** *The two values $h$ and $l$ returned by Algorithm 4.4 satisfy*

$$h + l = x^n(1 + \varepsilon)$$

*with*

$$(1 - 16\texttt{eps}^2)^{n-1} \leq 1 + \varepsilon \leq (1 + 16\texttt{eps}^2)^{n-1}.$$

*Proof.* The proof is very similar to the one of [11, Thm 4]. It comes from the fact that by induction one can show that the approximation of $x^k$ is of the form $x^k(1 + \varepsilon_k)$ with $(1 - 16\texttt{eps}^2)^{k-1} \leq 1 + \varepsilon_k \leq (1 + 16\texttt{eps}^2)^{k-1}$. □

Let us denote $\varphi = \texttt{eps}(1 - \texttt{eps})$ and

$$\overline{\gamma}_n = \frac{16n\texttt{eps}^2}{1 - 16n\texttt{eps}^2}.$$

It follows that $1+\varepsilon \leq (1+16\mathtt{eps}^2)^{n-1} \leq 1+\overline{\gamma}_n$ [8, p.63]. Since $h+l = x^n(1+\varepsilon)$ then it holds $x^n = h+l-\varepsilon x^n$. Furthermore, it also holds $\mathtt{res} = \mathrm{fl}(h+l)$. From Lemma 3.5, if we prove that $2\varepsilon|x^n| < \mathtt{eps}|\mathtt{res}|$ then $\mathtt{res}$ is a faithful rounding of $x^n$. From the definition of $\mathtt{res}$, it follows that $(1-\mathtt{eps})|h+l| \leq |\mathtt{res}|$ and so $(1-\mathtt{eps})(1+\varepsilon)|x^n| \leq |\mathtt{res}|$. A sufficient condition to ensure a faithful rounding is then

$$2\varepsilon|x^n| < \mathtt{eps}(1-\mathtt{eps})(1+\varepsilon)|x^n| \quad \text{and so} \quad 2\varepsilon < \mathtt{eps}(1-\mathtt{eps})(1+\varepsilon),$$

which is equivalent to

$$\varepsilon < \frac{\varphi}{2-\varphi}.$$

Since $\varepsilon \leq \overline{\gamma}_n$, a sufficient condition is

$$\overline{\gamma}_n < \frac{\varphi}{2-\varphi},$$

which is equivalent to

$$n < \frac{\varphi}{16\mathtt{eps}^2} = \frac{(1-\mathtt{eps})\mathtt{eps}^{-1}}{16}.$$

For example, in double precision where $\mathtt{eps} = 2^{-53}$, if $n < 2^{49} \approx 5 \cdot 10^{14}$, then we get a faithfully rounded result.

## 5 Conclusion

In this paper, we provided an accurate algorithm for computing product of floating point numbers. We gave some sufficient conditions to obtain a faithfully rounded result as well as validated error bounds. We applied this algorithm to compute exponentiation of floating point numbers. We improved this algorithm by using a double-double library.

## References


[1] *MPFR, the Multiprecision Precision Floating Point Reliable library.* Available at URL = http://www.mpfr.org.

[2] David H. Bailey. A Fortran 90-based multiprecision system. *ACM Trans. Math. Softw.*, 21(4):379–387, 1995.

[3] David H. Bailey. *A Fortran-90 double-double library*, 2001. Available at URL = http://crd.lbl.gov/~dhbailey/mpdist/index.html.

[4] Richard P. Brent. A Fortran multiple-precision arithmetic package. *ACM Trans. Math. Softw.*, 4(1):57–70, 1978.

[5] T. J. Dekker. A floating-point technique for extending the available precision. *Numer. Math.*, 18:224–242, 1971.

[6] Stef Graillat, Nicolas Louvet, and Philippe Langlois. Compensated Horner scheme. Research Report 04, Équipe de recherche DALI, Laboratoire LP2A, Université de Perpignan Via Domitia, France, 52 avenue Paul Alduy, 66860 Perpignan cedex, France, July 2005.

[7] Yozo Hida, Xiaoye S. Li, and David H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *Proc. 15th IEEE Symposium on Computer Arithmetic*, pages 155–162. IEEE Computer Society Press, Los Alamitos, CA, USA, 2001.

[8] Nicholas J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, second edition, 2002.

[9] *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. Institute of Electrical and Electronics Engineers, New York, 1985. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987.

[10] Donald E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Addison-Wesley, Reading, MA, USA, third edition, 1998.

[11] Peter Kornerup, Vincent Lefevre, and Jean-Michel Muller. Computing integer powers in floating-point arithmetic. arXiv:0705.4369v1 [cs.NA].

[12] Philippe Langlois and Nicolas Louvet. How to ensure a faithful polynomial evaluation with the compensated horner algorithm. In *Proceedings of the 18th IEEE Symposium on Computer Arithmetic (ARITH '07), Montpellier, France*, pages 141–149. IEEE Computer Society, Los Alamitos, CA, USA, 2007.

[13] Christoph Quirin Lauter. Basic building blocks for a triple-double intermediate format. Research Report RR-5702, INRIA, September 2005.

[14] Yves Nievergelt. Scalar fused multiply-add instructions produce floating-point matrix arithmetic provably accurate to the penultimate digit. *ACM Trans. Math. Software*, 29(1):27–48, 2003.

[15] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM J. Sci. Comput.*, 26(6):1955–1988, 2005.

[16] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Verified solution of linear systems without directed rounding. Technical Report No. 2005-04, Advanced Research Institute for Science and Engineering, Waseda University, 2005.

[17] Douglas M. Priest. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, Mathematics Department, University of California, Berkeley, CA, USA, November 1992. `ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z`.

[18] Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi. Accurate floating-point summation. Technical Report 05.12, Faculty for Information and Communication Sciences, Hamburg University of Technology, nov 2005.

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

[20] Pat H. Sterbenz. *Floating-point computation*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1974. Prentice-Hall Series in Automatic Computation.