Brief Contributions

Accurate Floating-Point Product and Exponentiation

Stef Graillat

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 integer power of a floating-point number.

Index Terms—Accurate product, exponentiation, finite precision, floating-point arithmetic, faithful rounding, error-free transformations.

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, the computed result is equal to the exact result if the latter is a floating-point number and otherwise is one of the two adjacent floating-point numbers of the exact result.

This paper was motivated by papers [1], [2], [3], [4], and [5], where similar approaches are used to compute summation, dot product, polynomial evaluation, and integer power.

The applications of our algorithms are multiple. One of the examples frequently used in Sterbenz's book [6] is the computation of the product of some floating-point numbers. Our algorithms can be used, for instance, 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

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

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 integer power of a floating-point number.

• *The author is with the Laboratoire LIP6, Département Calcul Scientifique, Université Pierre et Marie Curie (Paris 6), 4 place Jussieu, F-75252 Paris Cedex 05, France. E-mail: stef.graillat@lip6.fr.*

Manuscript received 22 July 2007; revised 25 Aug. 2008; accepted 10 Sept. 2008; published online 5 Dec. 2008.
Recommended for acceptance by P. Kornerup, P. Montuschi, J.-M. Muller, and E. Schwarz.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2007-07-0354.
Digital Object Identifier no. 10.1109/TC.2008.215.

The rest of this 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. We also present an accurate algorithm using a double-double library and we compare it with our compensated algorithm. 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 this paper, we assume to work with a floating-point arithmetic adhering to IEEE 754 floating-point standard in rounding to nearest [7]. We assume that no overflow nor underflow occur. The set of floating-point numbers is denoted by $\mathbb{F}$, the relative rounding error by $\mathbf{eps}$. For IEEE 754 double precision, we have $\mathbf{eps} = 2^{-53}$ and for single precision $\mathbf{eps} = 2^{-24}$.

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]

$$\begin{aligned} \mathrm{fl}(a \circ b) &= (a \circ b)(1 + \varepsilon_1) \\ &= (a \circ b)/(1 + \varepsilon_2) \text{ for } \circ = \{+, -, \cdot, /\} \text{ and} \\ &\qquad |\varepsilon_\nu| \le \mathbf{eps} \text{ for } \nu = 1, 2. \end{aligned} \tag{1}$$

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 we usually 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, [9]):

$$\begin{aligned} x = \mathrm{fl}(a \pm b) &\Rightarrow a \pm b = x + y \quad \text{with } y \in \mathbb{F}, \\ x = \mathrm{fl}(a \cdot b) &\Rightarrow a \cdot b = x + y \quad \text{with } y \in \mathbb{F}. \end{aligned} \tag{2}$$

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

Fortunately, the quantities x and y in (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 1 (Knuth [10]*):* Error-free transformation of the sum of two floating-point numbers

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

For the error-free transformation of a product, we first need to split the input argument into two parts. Let p be given by $\mathbf{eps} = 2^{-p}$ 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 [9] splits a floating-point number $a \in \mathbb{F}$ into two parts x and y such that

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

Algorithm 2 (Dekker [9]*):* Error-free split of a floating-point number into two parts

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

With this function, an algorithm from Veltkamp (see [9]) makes it possible 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

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

Algorithm 3 (Veltkamp [9]*):* Error-free transformation of the product of two floating-point numbers

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

The following theorem summarizes the properties of algorithm TwoProduct.

Theorem 1 (Ogita et al. [1]). *Let $a, b \in \mathbb{F}$ and let $x, y \in \mathbb{F}$ such that $[x, y] = \texttt{TwoProduct}(a, b)$ (Algorithm 3). Then,*

$$a \cdot b = x + y, \quad x = \text{fl}(a \cdot b), \quad |y| \le \texttt{eps}|x|, \quad |y| \le \texttt{eps}|a \cdot b|. \tag{3}$$

The algorithm TwoProduct *requires 17 flops.*

The TwoProduct algorithm can be rewritten in a very simple way if a Fused-Multiply-and-Add (FMA) operator is available on the targeted architecture [11]. Some computers have an 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 and as the new Cell processor [12]. 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

$$\begin{aligned}\texttt{FMA}(a, b, c) &= (a \cdot b + c)(1 + \varepsilon_1) \\ &= (a \cdot b + c)/(1 + \varepsilon_2) \text{ with } |\varepsilon_\nu| \le \texttt{eps for } \nu = 1, 2.\end{aligned}$$

Due to the FMA, the TwoProduct algorithm can be rewritten as follows, which costs only two flops:

Algorithm 4 (Ogita et al. [1]*):* 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 Section 3.1, we recall the classic method and we give a theoretical error bound as well as a validated computable error bound. In Section 3.2, we present our new algorithm based on a compensated scheme together with a theoretical error bound. In Section 3.3, we give sufficient conditions on the number of floating-point numbers so as to get a faithfully rounded result. In Section 3.4, we give a validated computable error bound for our new algorithm. In Section 3.5, we present the classic recursive algorithm for computing the product of floating-point numbers but using internally a double-double library. This makes it possible to achieve the same accuracy as the compensated algorithm. Finally, in Section 3.6, we provide some numerical experiments showing the performance of our compensated 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 5: 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 analyze its accuracy.

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

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

When using γ_n, we implicitly assume that $n\texttt{eps} \le 1$. A forward error bound is

$$|a_1 a_2 \cdots a_n - \texttt{res}| = |a_1 a_2 \cdots a_n - \text{fl}(a_1 a_2 \cdots a_n)| \le \gamma_{n-1}|a_1 a_2 \cdots a_n|. \tag{4}$$

Indeed, by induction,

$$\texttt{res} = \text{fl}(a_1 a_2 \cdots a_n) = a_1 a_2 \cdots a_n (1 + \varepsilon_2)(1 + \varepsilon_3) \cdots (1 + \varepsilon_n), \tag{5}$$

with $|\varepsilon_i| \le \texttt{eps}$ for $i = 2 : n$. It follows from Lemma 3.1 in [8, p. 63] that $(1 + \varepsilon_2)(1 + \varepsilon_3), \ldots, (1 + \varepsilon_n) = 1 + \theta_{n-1}$, where $|\theta_{n-1}| \le \gamma_{n-1}$. We also have that $(1 + \theta_k)(1 + \theta_j) = (1 + \theta_{k+j})$.

It is shown in [13] that for $a \in \mathbb{F}$, we have

$$\begin{aligned}(1 + \texttt{eps})^n &\le \frac{1}{(1 - \texttt{eps})^n} \\ &\le \frac{1}{1 - n\texttt{eps}} \quad \text{and} \\ \frac{|a|}{1 - n\texttt{eps}} &\le \text{fl}\left(\frac{|a|}{1 - (n+1)\texttt{eps}}\right).\end{aligned} \tag{6}$$

We also have

$$(1 + \varepsilon_2)(1 + \varepsilon_3) \cdots (1 + \varepsilon_n)\texttt{res} = a_1 a_2 \cdots a_n,$$

and then it follows that

$$|a_1 a_2 \cdots a_n - \texttt{res}| \le \gamma_{n-1}|\texttt{res}|.$$

If $m\texttt{eps} \le 1$ for $m \in \mathbb{N}$, $\text{fl}(m\texttt{eps}) = m\texttt{eps}$ and $\text{fl}(1 - m\texttt{eps}) = 1 - m\texttt{eps}$. Therefore,

$$\gamma_m \le (1 + \texttt{eps})\text{fl}(\gamma_m). \tag{7}$$

Hence,

$$|a_1a_2\cdots a_n - \mathtt{res}| \le (1+\mathtt{eps})\mathrm{fl}(\gamma_{n-1})|\mathtt{res}|,$$

and so

$$|a_1a_2\cdots a_n - \mathtt{res}| \le \mathrm{fl}\left(\frac{\gamma_{n-1}|\mathtt{res}|}{1-2\mathtt{eps}}\right).$$

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 by Ogita et al. [1]. This technique using error-free transformations have been widely used to provide some new accurate algorithms in floating-point arithmetic (see [1] and [2] for accurate sum and dot product and [3] and [4] for polynomial evaluation). It was also recently used in [5] to accurately compute the integer power of a floating-point number. We generalize the work of Kornerup et al. [5] to the product of several floating-point numbers.

Algorithm 6: Product evaluation with a compensated scheme

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

This algorithm requires $19n - 18$ flops if we use `TwoProduct`. It only requires $3n - 2$ flops if we use `TwoProductFMA` instead of `TwoProduct` (if, of course, an `FMA` is available) and $e_i = \mathtt{FMA}(e_{i-1}, a_i, \pi_i)$ instead of $e_i = \mathrm{fl}(e_{i-1}a_i + \pi_i)$.

Algorithm 7: Product evaluation with a compensated scheme with `TwoProductFMA` and `FMA`

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

We will provide an error analysis only for Algorithm `CompProd`. The error analysis for `CompProdFMA` is very similar to the one of `CompProd` with little changes having to be done to take into account the operation $e_i = \mathtt{FMA}(e_{i-1}, a_i, \pi_i)$. This changes nearly nothing so it is straightforward to modify the analysis to deal with it.

For error analysis, we note that

$$p_n = \mathrm{fl}(a_1a_2\cdots a_n) \quad \text{and} \quad e_n = \mathrm{fl}\left(\sum_{i=2}^{n} \pi_i a_{i+1}\cdots a_n\right).$$

We also have

$$p = a_1a_2\cdots a_n = \mathrm{fl}(a_1a_2\cdots a_n) + \sum_{i=2}^{n} \pi_i a_{i+1}\cdots a_n = p_n + e, \qquad (8)$$

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

Before proving the main theorem, we will need a bound on the individual error of the multiplication π_i as a function of the original data a_i.

Suppose floating-point numbers $\pi_i \in \mathbb{F}$, $2 \le i \le n$, are computed by the following algorithm.

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

Then,

$$|\pi_i| \le \mathtt{eps}(1+\gamma_{i-1})|a_1\cdots a_i| \quad \text{for } i = 2:n. \qquad (9)$$

Indeed, from (1), it follows that

$$|\pi_i| \le \mathtt{eps}|p_i|.$$

Moreover, $p_i = \mathrm{fl}(a_1\ldots a_i)$ so that from (4),

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

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

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

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

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

Then,

$$\left|e_n - \sum_{i=2}^{n} \pi_i a_{i+1}\cdots a_n\right| \le \gamma_{n-1}\gamma_{2n}|a_1a_2\cdots a_n|.$$

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]

$$\begin{aligned}\mathrm{fl}(x_1 + x_2 + \cdots + x_n) = {} & x_1(1+\theta_{n-1}) + x_2(1+\theta_{n-1}) \\ & + x_3(1+\theta_{n-2}) + \cdots + x_n(1+\theta_1).\end{aligned}$$

This implies that

$$\begin{aligned}e_n &= \mathrm{fl}\left(\sum_{i=2}^{n}(\pi_i a_{i+1}\cdots a_n)\right) \\ &= \mathrm{fl}(\pi_2 a_3\cdots a_n)(1+\theta_{n-2}) \\ &\quad + \mathrm{fl}(\pi_3 a_4\cdots a_n)(1+\theta_{n-2}) + \cdots + \mathrm{fl}(\pi_n)(1+\theta_1).\end{aligned}$$

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

$$\begin{aligned}e_n = {} & \pi_2 a_3\cdots a_n(1+\theta_{n-2})(1+\theta_{n-1}) \\ & + \pi_3 a_4\cdots a_n(1+\theta_{n-3})(1+\theta_{n-1}) + \cdots + \pi_n(1+\theta_1).\end{aligned}$$

A straightforward computation yields

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

From (9), we have $|\pi_i| \le \texttt{eps}(1+\gamma_{i-1})|a_1 \cdots a_i|$, and hence

$$\left| e_n - \sum_{i=2}^{n} \pi_i a_{i+1} \cdots a_n \right| \le (n-1)\texttt{eps}(1+\gamma_{n-1})\gamma_{2n-3}|a_1 a_2 \cdots a_n|.$$

Since $\texttt{eps}(1+\gamma_{n-1}) = \gamma_{n-1}/(n-1)$ 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 2. *Suppose Algorithm 6 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,*

$$|\texttt{res} - p| \le \texttt{eps}|p| + \gamma_n\gamma_{2n}|p|. \tag{10}$$

Proof. The fact that $\texttt{res} = \text{fl}(p_n + e_n)$ implies that $\texttt{res} = (1+\varepsilon)(p_n + e_n)$ with $|\varepsilon| \le \texttt{eps}$. So, it follows that

$$\begin{aligned}
|\texttt{res} - p| &= |\text{fl}(p_n + e_n) - p| = |(1+\varepsilon)(p_n + e_n - p) + \varepsilon p| \\
&= \left| (1+\varepsilon)\left(p_n + \sum_{i=2}^{n} \pi_i a_{i+1} \cdots a_n - p \right) \right. \\
&\quad \left. + (1+\varepsilon)\left(e_n - \sum_{i=2}^{n} \pi_i a_{i+1} \cdots a_n \right) + \varepsilon p \right| \\
&= \left| (1+\varepsilon)(e_n - \sum_{i=2}^{n} \pi_i a_{i+1} \cdots a_n) + \varepsilon p \right| \quad \text{by (8)} \\
&\le \texttt{eps}|p| + (1+\texttt{eps}) \left| e_n - \sum_{i=2}^{n} \pi_i a_{i+1} \cdots a_n \right| \\
&\le \texttt{eps}|p| + (1+\texttt{eps})\gamma_{n-1}\gamma_{2n}|a_1 a_2 \cdots a_n|.
\end{aligned}$$

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

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

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

A standard computation yields

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

Corollary 1. *Suppose Algorithm 6 is applied to floating-point number $a_i \in \mathbb{F}$, $1 \le i \le n$, and set $p = \prod_{i=1}^{n} a_i \ne 0$. Then,*

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

In fact, if $p \ne 0$, we can rewrite (10) in the following form:

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

Since $\gamma_n\gamma_{2n} \approx 2n^2\texttt{eps}^2$, for n not too large, it follows that $\gamma_n\gamma_{2n}$ is negligible compared to eps. As a consequence, the relative error $|\texttt{res} - p|/|p|$ is of the order of eps, that is to say, the result has nearly full accuracy. We will show this in Section 3.3.

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

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

Definition 1. *A floating-point number $f \in \mathbb{F}$ is called a* faithful rounding *of a real number $r \in \mathbb{R}$ if*

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

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

Faithful rounding means that the computed result is equal to the exact result if the latter is a floating-point number and otherwise is one of the two adjacent floating-point numbers of the exact result.

Lemma 2 (Rump et al. [2, Lemma 2.4]). *Let $r, \delta \in \mathbb{R}$ and $\tilde{r} := \text{fl}(r)$. Suppose that $2|\delta| < \texttt{eps}|\tilde{r}|$. Then, $\tilde{r} \in \square(r+\delta)$, that means $\tilde{r}$ is a faithful rounding of $r+\delta$.*

Let res be the result of CompProd. Then, we have $p = p_n + e$ and $\texttt{res} = \text{fl}(p_n + e_n)$ 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.

With the previous notations, if $2|e - e_n| < \texttt{eps}|\texttt{res}|$, then res is a faithful rounding of p. Since we have $|e - e_n| \le \gamma_n\gamma_{2n}|p|$ and $(1-\texttt{eps})|p| - \gamma_n\gamma_{2n}|p| \le |\texttt{res}|$, a sufficient condition to ensure a faithful rounding is

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

that is

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

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

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

which is equivalent to

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

and then to

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

Lemma 3. *If $n < \frac{\sqrt{1-\texttt{eps}}}{\sqrt{2}\sqrt{2+\texttt{eps}}+2\sqrt{(1-\texttt{eps})\texttt{eps}}}\texttt{eps}^{-1/2}$, then res is a faithful rounding of p.*

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

We can propose a weaker form of Lemma 3 but with a nicer constant if we suppose, for instance, that $\texttt{eps} \le 2^{-7}$. This is not a strong assumption since in general $\texttt{eps} = 2^{-53}$ or $\texttt{eps} = 2^{-24}$. We can easily show that if $\texttt{eps} \le 2^{-7}$ and $n < (4/9)\texttt{eps}^{-1/2}$, then res is a faithful rounding of p.

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

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

Since $|e_n-e|\le\gamma_{n-1}\gamma_{2n}|p|$ and $|p|\le(1+\mathtt{eps})^{n-1}\mathrm{fl}(|a_1a_2\cdots a_n|)$, we obtain

$$\begin{aligned}|\mathtt{res}-p| &\le \mathtt{eps}|\mathtt{res}|+\gamma_{n-1}\gamma_{2n}|p|\\ &\le \mathtt{eps}|\mathtt{res}|+\gamma_{n-1}\gamma_{2n}(1+\mathtt{eps})^{n-1}\mathrm{fl}(|a_1a_2\cdots a_n|).\end{aligned}$$

Using (6) and (7), we obtain

$$\begin{aligned}|\mathtt{res}-p| &\le \mathrm{fl}(\mathtt{eps}|\mathtt{res}|)+(1+\mathtt{eps})^n\ \mathrm{fl}(\gamma_n)\ \mathrm{fl}(\gamma_{2n})\ \mathrm{fl}(|a_1a_2\cdots a_n|)\\ &\le \mathrm{fl}(\mathtt{eps}|\mathtt{res}|)+(1+\mathtt{eps})^{n+2}\ \mathrm{fl}(\gamma_n\gamma_{2n}|a_1a_2\cdots a_n|)\\ &\le \mathrm{fl}(\mathtt{eps}|\mathtt{res}|)+\mathrm{fl}\left(\frac{\gamma_n\gamma_{2n}|a_1a_2\cdots a_n|}{1-(n+3)\mathtt{eps}}\right)\\ &\le (1+\mathtt{eps})\ \mathrm{fl}\left(\mathtt{eps}|\mathtt{res}|+\frac{\gamma_n\gamma_{2n}|a_1a_2\cdots a_n|}{1-(n+3)\mathtt{eps}}\right)\\ &\le \mathrm{fl}\left(\left(\mathtt{eps}|\mathtt{res}|+\frac{\gamma_n\gamma_{2n}|a_1a_2\cdots a_n|}{1-(n+3)\mathtt{eps}}\right)/(1-2\mathtt{eps})\right).\end{aligned}$$

We can summarize this as follows:

Lemma 4. *Suppose Algorithm 6 is applied to floating-point numbers $a_i\in\mathbb{F}$, $1\le i\le n$ and set $p=\prod_{i=1}^n a_i$. Then, the absolute forward error affecting the product is bounded according to*

$$|\mathtt{res}-p|\le \mathrm{fl}\left(\left(\mathtt{eps}|\mathtt{res}|+\frac{\gamma_n\gamma_{2n}|a_1a_2\cdots a_n|}{1-(n+3)\mathtt{eps}}\right)/(1-2\mathtt{eps})\right). \quad (11)$$

We have shown that

$$|e_n-e|\le \mathrm{fl}\left(\frac{\gamma_n\gamma_{2n}|a_1a_2\cdots a_n|}{1-(n+3)\mathtt{eps}}\right).$$

Lemma 3 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

$$\mathrm{fl}\left(2\frac{\gamma_n\gamma_{2n}|a_1a_2\cdots a_n|}{1-(n+3)\mathtt{eps}}\right)<\mathrm{fl}(\mathtt{eps}|\mathtt{res}|),$$

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

3.5 Double-Double Library

Compensated methods are a possible way to improve the accuracy. Another possibility is to increase the working precision. For this purpose, one can use the Bailey's *double-double* [14]: double-double numbers are represented as an unevaluated sum of a leading double and a trailing double. More precisely, a double-double number a is the pair (a_h,a_l) of floating-point numbers with $a=a_h+a_l$ and $|a_l|\le\mathtt{eps}|a_h|$.

In the sequel, we present two algorithms to compute the product of two double-double or a double times a double-double. Those algorithms are taken from [15].

Algorithm 8: Multiplication of two double-double numbers

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

Algorithm 9: Multiplication of double-double number by a double number

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

The following result gives the accuracy of the product of two double-double numbers.

Theorem 3 (Lauter [15, Thm. 4.7]). *Let a_h+a_l and b_h+b_l be the double-double arguments of Algorithm 8. 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 ε is bounded as follows: $|\varepsilon|\le 16\mathtt{eps}^2$. Furthermore, we have $|r_l|\le\mathtt{eps}|r_h|$.

Results for Algorithm 9 are very similar with $a=a_h$ and $a_l=0$.

We can now propose an algorithm that computes the product of floating-point numbers using internally double-double numbers.

Algorithm 10: Product evaluation with a double-double library

function $\mathtt{res}=\mathtt{DDProd}(a)$
$[h,1]=[a_1,0]$
for $i=2:n$
$[h,1]=\mathtt{prod\_dd\_d}(a_i,h,l)$
end
$\mathtt{res}=\mathrm{fl}(h+l)$

This algorithm requires $25n-24$ flops.
For the sequel, let us denote $\varphi=\mathtt{eps}(1-\mathtt{eps})$ and

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

Let us now study the accuracy of the result of Algorithm 10.

Theorem 4. *The two values h and l returned by Algorithm 10 applied to floating-point number $a_i\in\mathbb{F}$, $1\le i\le n$, satisfy*

$$h+l=p(1+\varepsilon)$$

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

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

Furthermore, if $\mathtt{res}=\mathrm{fl}(h+l)$, then we have

$$|\mathtt{res}-p|\le\mathtt{eps}|p|+\bar{\gamma}_n|p|.$$

Proof. It comes from the fact that by induction one can show that the approximation of $a_1a_2\cdots a_k$ is of the form $a_1a_2\cdots a_k(1+\varepsilon_k)$ with $(1-16\mathtt{eps}^2)^{k-1}\le 1+\varepsilon_k\le(1+16\mathtt{eps}^2)^{k-1}$.

We also have

$$\begin{aligned}|\mathtt{res}-p| &= |\mathrm{fl}(h+l)-p| = |\mathrm{fl}(h+l)-(h+l)+(h+l)-p|\\ &\le \mathtt{eps}|p|+|(h+l)-p|\\ &\le \mathtt{eps}|p|+|\varepsilon|\cdot|p|.\end{aligned}$$

It follows that $1+\varepsilon\le(1+16\mathtt{eps}^2)^{n-1}\le 1+\bar{\gamma}_n$ [8, p. 63]. As a consequence,

$$|\mathtt{res}-p|\le\mathtt{eps}|p|+\bar{\gamma}_n|p|.$$

□

We can now ask when the result is faithfully rounded. Since $h+l=p(1+\varepsilon)$, then it holds that $p=h+l-\varepsilon p$. Furthermore, it also holds that $\mathtt{res}=\mathrm{fl}(h+l)$. From Lemma 2, if we prove that $2\varepsilon|p|<\mathtt{eps}|\mathtt{res}|$, then res is a faithful rounding of p. From the definition of res, it follows that $(1-\mathtt{eps})|h+l|\le|\mathtt{res}|$ and so

TABLE 1
Measured Computing Times with Prod Normalized to 1.0

n	Prod	CompProd	DDProd
100	1.0	3.5	4.3
500	1.0	3.9	6.2
1000	1.0	5.0	8.6
10000	1.0	6.1	10.3
100000	1.0	5.3	9.0

$(1-\mathtt{eps})(1+\varepsilon)|p| \le |\mathtt{res}|$. A sufficient condition to ensure a faithful rounding is then

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

which is equivalent to

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

Since $\varepsilon \le \bar{\gamma}_n$, a sufficient condition is

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

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 obtain a faithfully rounded result.

3.6 Numerical Experiments

We have performed some numerical experiments to test our new algorithm on a laptop with a Pentium M processor at 1.73 GHz. We used gcc version 4.0.2. We compared Prod, CompProd, and DDProd in terms of measured computing time.

The theoretical ratio for CompProd is 19. From Table 1, we see that the measured computing time ratio is better than the theoretical one. The result can be surprising, but as shown in [16], the compensated algorithms are generally faster than the theoretical performances. This is especially due to a better instruction-level parallelism.

The theoretical ratio between CompProd and DDProd is approximately $25/19 \approx 1.3$. As you can see in Table 1, the measured ratio is about 1.7 when n is sufficiently large. This is in part due to the renormalization step of double-double that is needed to ensure that $[h,l]$ satisfies $|l| \le \mathtt{eps}|h|$. This renormalization step breaks instruction-level parallelism.

As we have seen, CompProd is faster than DDProd while the results share the same accuracy (faithful rounding). In fact, DDProd is more accurate than CompProd in the sense that DDProd gives a faithful rounding for $n \lesssim \mathtt{eps}^{-1}$ whereas CompProd gives a faithful rounding for $n \lesssim \mathtt{eps}^{-1/2}$. Nevertheless, it is quite rare to need to compute the product of more that 10^7 double precision floating-point numbers. As a consequence, it seems that CompProd is a fast and accurate algorithm to compute the product of floating-point numbers.

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 $\mathcal{O}(\log n)$).

The natural method to compute x^n is to apply algorithm CompProd for $a_i \in \mathbb{F}$ with $a_i = x$ for $1 \le i \le n$. As a consequence, if $n < 2^{25}$ then the result is faithfully rounded. This algorithm is also similar to the one in [5]. It is also the same as the Compensated Horner scheme [3] applied to the polynomial $p(x) = x^n$. Results concerning faithful polynomial evaluation can be found in [4].

A logarithmic algorithm was introduced in [5] 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 number. 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 11: Power evaluation with double-double

```
function res = CompLogPower(x, n)        % n = (n_t n_{t-1} ... n_1 n_0)_2
  [h, l] = [1, 0]
  for i = t : -1 : 0
    [h, l] = prod_dd_dd(h, l, h, l)
    if n_i = 1
      [h, l] = prod_dd_d(x, h, l)
    end
  end
res = fl(h + l)
```

Theorem 5. *The two values h and l returned by Algorithm 11 satisfy*

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

with

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

Proof. The proof is very similar to the one in [5, 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 $(1-16\mathtt{eps}^2)^{k-1} \le 1+\varepsilon_k \le (1+16\mathtt{eps}^2)^{k-1}$. □

We still use the notations $\varphi = \mathtt{eps}(1-\mathtt{eps})$ and

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

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

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

which is equivalent to

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

Since $\varepsilon \le \bar{\gamma}_n$, a sufficient condition is

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

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 obtain a faithfully rounded result.

We have compared our algorithm `CompLogPower` with the right-to-left binary exponentiation algorithm in [5]. Our algorithm is just 1.07 times faster, in average, than the right-to-left binary exponentiation algorithm. The fact to replace a multiplication of two double-double numbers by a multiplication of a double number by a double-double number is not sufficient to obtain a good speedup.

5 Conclusion

In this paper, we have provided an accurate algorithm for computing the product of floating-point numbers. We gave some sufficient conditions to obtain a faithfully rounded result as well as validated error bounds. We compared this algorithm with the classic recursive one using internally a double-double library. We have shown that our algorithm was faster while sharing the same accuracy. We applied our compensated algorithm to compute exponentiation of floating-point numbers. We improved this algorithm by using a double-double library.

Acknowledgments

The author would like to thank the anonymous referees for their valuable comments and suggestions.

References

[1] T. Ogita, S.M. Rump, and S. Oishi, "Accurate Sum and Dot Product," *SIAM J. Scientific Computing*, vol. 26, no. 6, pp. 1955-1988, 2005.
[2] S.M. Rump, T. Ogita, and S. Oishi, "Accurate Floating-Point Summation. Part I: Faithful Rounding," *SIAM J. Scientific Computing*, vol. 31, no. 1, Oct. 2008.
[3] Research Report 04, S. Graillat, N. Louvet, and P. Langlois, "Compensated Horner Scheme," Équipe de recherche DALI, Laboratoire LP2A, Université de Perpignan Via Domitia, France, July 2005.
[4] P. Langlois and N. Louvet, "How to Ensure a Faithful Polynomial Evaluation with the Compensated Horner Algorithm," *Proc. 18th IEEE Symp. Computer Arithmetic (ARITH '07)*, pp. 141-149, 2007.
[5] P. Kornerup, V. Lefevre, and J.-M. Muller, *Computing Integer Powers in Floating-Point Arithmetic*, arXiv:0705.4369v1 [cs.NA], 2007.
[6] P.H. Sterbenz, *Floating-Point Computation*. Prentice-Hall, 1974.
[7] *IEEE Standard for Binary Floating-Point Arithmetic*, vol. 22, no. 2, ANSI/IEEE Standard 754-1985, New York, IEEE, 1985, reprinted in SIGPLAN Notices, pp. 9-25, 1987.
[8] N.J. Higham, *Accuracy and Stability of Numerical Algorithms*, second ed. SIAM, 2002.
[9] T.J. Dekker, "A Floating-Point Technique for Extending the Available Precision," *Numerical Math.*, vol. 18, pp. 224-242, 1971.
[10] D.E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, third ed. Addison-Wesley, 1998.
[11] Y. Nievergelt, "Scalar Fused Multiply-Add Instructions Produce Floating-Point Matrix Arithmetic Provably Accurate to the Penultimate Digit," *ACM Trans. Math. Software*, vol. 29, no. 1, pp. 27-48, 2003.
[12] C. Jacobi, H.-J. Oh, K.D. Tran, S.R. Cottier, B.W. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, and N. Yano, "The Vector Floating-Point Unit in a Synergistic Processor Element of a Cell Processor," *Proc. 17th IEEE Symp. Computer Arithmetic (ARITH '05)* pp. 59-67, 2005.
[13] T. Ogita, S.M. Rump, and S. Oishi, "Verified Solution of Linear Systems without Directed Rounding," Technical Report 2005-04, Advanced Research Inst. of Science and Eng., Waseda Univ., 2005.
[14] D.H. Bailey, *A Fortran-90 Double-Double Library*, http://crd.lbl.gov/dhbailey/mpdist/index.html, 2001.
[15] C.Q. Lauter, *Basic Building Blocks for a Triple-Double Intermediate Format*, Research Report RR-5702, INRIA, Sept. 2005.
[16] P. Langlois and N. Louvet, *More Instruction Level Parallelism Explains the Actual Efficiency of Compensated Algorithms*, hal-00165020, version 1, 2007.