Journal of Universal Computer Science, vol. 5, no. 6 (1999), 323-338
submitted: 26/4/99, accepted: 16/6/99, appeared: 28/6/99 © *Springer Pub. Co.*

Division of Floating Point Expansions with an Application to the Computation of a Determinant

Marc Daumas, Claire Finot
Laboratoire de l'Informatique du Parallelisme · Lyon · France
UMR 5668 · CNRS · ENS Lyon · INRIA
{Marc.Daumas,Claire.Finot}@ens-lyon.fr

Abstract: Floating point expansion is a technique for implementing multiple precision using a processor's floating point unit instead of its integer unit. Research on this subject has arised recently from the observation that the floating point unit becomes a more and more efficient part of modern computers. Many simple arithmetic operators and some very useful geometric operators have already been presented on expansions. Yet previous work included only a very simple division algorithm. We present in this work a new algorithm that allows us to extend the set of geometric operators with Bareiss' determinant on a matrix of size between 3 and 10. Running times with different determinant algorithms on different machines are compared with GMP, a very common multi-precision package.

Key Words: Exact arithmetic, multiple precision, expansion, division, computational geometry, floating point, library.
Category: B.2, G.4.

1 Introduction

Thanks to the 754 and 854 IEEE standards [17, 18, 3, 7], both the operations on floating point numbers and the behavior of the floating point unit in cases of exceptions are completely specified. Since MFLOPs figures are highly publicized by processor manufacturers, the standard floating point operators are also the most developed and the most powerful arithmetic operators on common processors [9, 8, 16].

Expansions were introduced by Priest in 1991 [13] as a multiple precision tool on floating point numbers based on earlier work by Dekker [6] and Knuth [10]. He proposed algorithms on expansions including the addition, the multiplication and the division. Priest's algorithms were capable to adapt to the machine. They are correct for different working radices and for the different levels of precision achieved on rounding by some specific floating point unit.

In 1996 and 1997 Shewchuk implemented some enhanced algorithms restricted to the case of an IEEE standard floating point rounding [14, 15]. His library is available on the net[1] with an application to computational geometry. It includes the addition, the subtraction and the scaling of an expansion by a floating point number. One of us proposed in [4, 5] the multiplication of two expansions with a survey on former work on expansions in [4].

The section 2 of this article presents definitions and properties of floating point numbers and floating point expansions. We have isolated a new lemma that is used in this paper. We study in section 3 a new division operator with

[1] URL: `http://www.cs.cmn.edu/~jrs`.

a look to Priest's prior division algorithm. Our modified algorithm will enable us to implement on-line most significant digit first adaptive computing. Section 4 proves the exact halting of both algorithms: when the result of the division exactly fits in an expansion, both algorithms find it. Section 5 outlines an application of expansions to compute the exact determinant of a small matrix (size 3 to 10) as it may be used in computational geometry. We present three different algorithms and significant running times on two different IEEE machines. This paper ends with concluding remarks and directions for further developments.

2 Definitions and properties

2.1 Floating point numbers

A **floating point number** is stored in binary as a finite length fraction and a bounded exponent as presented in figure 1. It reads

$$x = (-1)^{\text{sign}} \cdot (1.\text{fraction}) \cdot 2^{\text{exponent}}$$

With this notation, a weight can be associated to each bit of the mantissa. Later on, we call $\alpha(x)$ the weight of the most significant non-zero bit of x and $\omega(x)$ the weight of its least significant non-zero bit. For a normalized number x, $\alpha(x) = 2^{\text{exponent}}$. We define the **ulp** function that means "Unit in the Last Place" as the weight of the last bit of the mantissa of x. For any non zero floating point number, we also define $m(x) = \frac{x}{\alpha(x)}$; $m(x)$ is the mantissa of x and $|m(x)| \in [1, 2 - ulp]$.

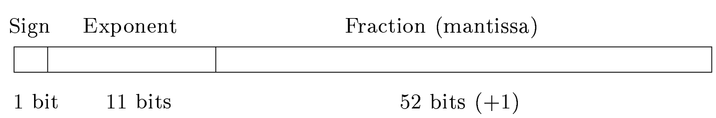


Figure 1: Representation of floating point numbers in IEEE standard double precision

Properties : *Let a and b be two floating point numbers. We know that:*

$$\alpha(a) \leq |a| < 2\alpha(a) \tag{1}$$
$$\alpha(a)\alpha(b) \leq \alpha(ab) \leq 2\alpha(a)\alpha(b) \tag{2}$$

$$|b| \cdot \alpha(a) < 2|a| \cdot \alpha(b) \tag{3}$$

Proof of property (1) :

Since $|a| = \alpha(a)|m(a)|$ and $|m(a)| \in [1, 2)$ then $\alpha(a) \leq |a| < 2\alpha(a)$. 2

Proof of property (2) :

We expand $\alpha(ab) = \alpha(\alpha(a)m(a) \cdot \alpha(b)m(b)) = \alpha(a)\alpha(b)\alpha(m(a)m(b))$. Since $|m(a)m(b)| \in [1, 4)$ we know that $\alpha(m(a)m(b)) \in \{1, 2\}$ and we find $\alpha(a)\alpha(b) \leq \alpha(ab) \leq 2\alpha(a)\alpha(b)$. 2

Proof of property (3) :

Since both $|m(a)|$ and $|m(b)|$ are in $[1, 2)$, $|m(a)| < 2|m(b)|$. Following the definition of $|m(a)|$ and $|m(b)|$, we have $|b| \cdot \alpha(a) < 2|a| \cdot \alpha(b)$. 2

The IEEE standard describes four rounding modes but the rounding to the nearest floating point number is the rounding mode used by default in most computers. The result of any implemented operation, namely the addition, the multiplication, the division and the square root extraction, is the rounded result of the exact mathematical operation. For example, if $\circ(x)$ is the rounded to the nearest value of x for any x, and the machine floating point addition of a and b is $a \oplus b$, then $a \oplus b = \circ(a + b)$.

As proposed by Dekker [6] and Knuth [10], an exact two-sum operator can be constructed from standard floating point operators. The result of the addition is a pair (a', b') such that $a' = \circ(a+b)$ and $a'+b' = a+b$. An exact multiplication is also available. It computes a pair (a', b') such that $a' = \circ(a \cdot b)$ and $a'+b' = a \cdot b$. It was proved that b' always fits in a common floating point number. The exact sum needs 4 floating point additions and 2 floating point subtractions. In particular cases, a fast exact sum only requires 2 floating point additions and one floating point subtraction. These operators are surveyed in [4, 5] with a new improved condition to apply the fast exact sum.

Definition 1 *Let* $\mathbf{A}$ *be a set of floating point numbers with its sum* $A = \sum_{a \in \mathbf{A}} a$. *We say that* $\widehat{A}$ *is a* **fair most significant component** *of* A *if either:*

- $\widehat{A}$ *equals to the rounded to the nearest value of* A
- $\widehat{A}$ *equals to the rounded to the nearest value of* A_1 *with* $|A - A_1| \leq ulp(ulp(\widehat{A}))$

Properties : *Let* $\mathbf{A}$ *be a set of floating point numbers with its sum* $A = \sum_{a \in \mathbf{A}} a$, *and* $\widehat{A}$ *a fair most significant component. The following properties are satisfied:*

$$\left|A - \widehat{A}\right| \leq ulp\left(\frac{1}{2} + ulp\right)\alpha(\widehat{A}) \tag{4}$$

$$\frac{\left|\widehat{A}\right|}{2} \leq |A| \leq 2\left|\widehat{A}\right| \tag{5}$$

$$\frac{1}{2}\alpha(\widehat{A}) \leq \alpha(A) \tag{6}$$

$$\alpha(A) \leq \alpha(\widehat{A}) \tag{7}$$

Proof of property (4)

Thanks to definition 1, we know that $\widehat{A} = \circ(A_1)$ and $|A - A_1| \le ulp(ulp(\widehat{A}))$.

$$\begin{aligned}\left|A - \widehat{A}\right| &\le |A - A_1| + \left|A_1 - \widehat{A}\right| \\ &\le ulp(ulp(\widehat{A})) + \frac{1}{2} ulp(\widehat{A}) \\ &\le ulp(\alpha(\widehat{A}).ulp) + \frac{1}{2}\alpha(\widehat{A}) \cdot ulp \\ &\le \alpha(\widehat{A}) \cdot ulp^2 + \frac{1}{2}\alpha(\widehat{A}) \cdot ulp \\ &\le ulp\left(\frac{1}{2} + ulp\right)\alpha(\widehat{A})\end{aligned}$$

2

Property (5) is a strict corollary of property (4) and its proof is omitted here.

Proof of property (6)

Thanks to property 4, we know that,

$$\begin{aligned}\left|A - \widehat{A}\right| &\le ulp\left(\frac{1}{2} + ulp\right)\alpha(\widehat{A}) \\ &\le ulp\left(\frac{1}{2} + ulp\right)|\widehat{A}|\end{aligned} \tag{8}$$

So there exists $\epsilon \in [-1, 1]$ such that:

$$A - \widehat{A} = \epsilon \cdot ulp\left(\frac{1}{2} + ulp\right)\widehat{A}$$

Then

$$\begin{aligned}A &= \left(\epsilon \cdot ulp\left(\frac{1}{2} + ulp\right) + 1\right)\widehat{A} \\ \alpha(A) &= \alpha\left(\left[\epsilon \cdot ulp\left(\frac{1}{2} + ulp\right) + 1\right]\widehat{A}\right) \\ \alpha(A) &\ge \alpha\left(\epsilon \cdot ulp\left(\frac{1}{2} + ulp\right) + 1\right)\alpha\left(\widehat{A}\right)\end{aligned}$$

As

$$\left|\epsilon \cdot ulp\left(\frac{1}{2} + ulp\right) + 1\right| \ge \frac{1}{2}$$

Then

$$\alpha(A) \ge \frac{1}{2}\alpha(\widehat{A})$$

2

Proof of property (7)

As we did in the proof of property (6), we can find ϵ in $[-1,1]$ as follows:

$$\begin{aligned} |A - A_1| &\le ulp(ulp(\widehat{A})) \le ulp^2 \cdot \alpha(\widehat{A}) \le 2ulp^2 \cdot \alpha(A) \\ A - A_1 &= 2\epsilon \cdot ulp^2 \cdot \alpha(A) \\ A_1 &= \alpha(A)(m(A) + 2\epsilon \cdot ulp^2) \end{aligned}$$

It follows that:

$$\begin{aligned} \widehat{A} &= \circ(\alpha(A)(m(A) + 2\epsilon \cdot ulp^2)) \\ &= \alpha(A) \circ (m(A) + 2\epsilon \cdot ulp^2) \\ \left|\widehat{A}\right| &\ge \alpha(A) \circ (1 - 2ulp^2) \\ &\ge \alpha(A) \end{aligned}$$

Since $\circ(1 - 2ulp^2) = 1$. It follows that $\alpha(\widehat{A}) \ge \alpha(A)$. 2

The six properties presented above and the following lemma are very general results on standard floating point numbers as well as the lemmas presented in [4, 5]. They will probably be applied in the future to other situations. We have started building an adapted *corpus* of sensible results on floating point operations.

Lemma 1 *Let* $\mathbf{A}$ *be a set of floating point numbers with its sum* $A = \sum_{a \in \mathbf{A}} a$ *such that* $\widehat{A}$ *is a fair most significant component. The following property is satisfied for any floating point number* b *and any set of floating point numbers* $\mathbf{C}$ *such that* $C = b.A$*—with* $C = \sum_{c \in \mathbf{C}} c$ *and* $\widehat{C}$ *is a fair most significant component of* C*.*

$$\alpha(\widehat{C}) \le 2\alpha(b) \cdot \alpha(\widehat{A})$$

Proof :

In the most general case, $\widehat{C} = \circ(C_1)$ with $|C - C_1| \le ulp(ulp(\widehat{C}))$

$$\widehat{C} = \circ(C_1) = \circ(bA + e)$$

with

$$\begin{aligned} |e| &\le ulp^2 \cdot \alpha(\widehat{C}) \\ &\le 2ulp^2 \cdot \alpha(C) \\ &\le 2ulp^2 \cdot \alpha(bA) \\ &\le 4ulp^2 \cdot \alpha(b)\alpha(A) \end{aligned}$$

Let $e = \epsilon \cdot ulp^2 \cdot \alpha(b)\alpha(A)$, with $\epsilon \in [-4,4]$. We know that:

$$\begin{aligned} \widehat{C} &= \circ(\alpha(b)\alpha(A)m(A)m(b) + \epsilon \cdot ulp^2 \cdot \alpha(b)\alpha(A)) \\ &= \alpha(b)\alpha(A) \circ (m(A)m(b) + \epsilon \cdot ulp^2) \end{aligned}$$

As $|m(b)| \leq 2 - ulp$ and $|m(A)| \leq 2$,

$$|m(A)m(b) + \epsilon \cdot ulp^2| \leq 2(2 - ulp) + 4ulp^2 \qquad \text{and}$$
$$\left|\circ(m(A)m(b) + \epsilon \cdot ulp^2)\right| \leq 2(2 - ulp)$$

It follows that:

$$\begin{aligned}\alpha(\widehat{C}) &\leq \alpha(\alpha(b)\alpha(A) \cdot 2(2 - ulp)) \\ &\leq \alpha(b)\alpha(A) \cdot \alpha(2(2 - ulp)) \\ &\leq 2\alpha(b)\alpha(A) \\ &\leq 2\alpha(b)\alpha(\widehat{A})\end{aligned}$$

2

2.2 Expansions

An **expansion** is the representation of a large floating point quantity, $\mathbf{x}$, by an n-tuple of machine floating point numbers $(x_n \cdots x_2, x_1)$ called the **components** (see figure 2). The **length** of the expansion is the number of its components.

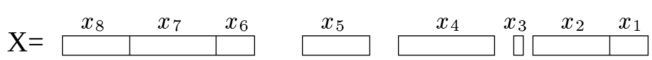


Figure 2: Representation of an expansion

The components must be sorted by magnitude and two components cannot have significant bits with the same weight as illustrated figure 3. We say that the components are **non-overlapping**. For any value of i, two non zero components x_{i+1} and x_i satisfy $\omega(x_{i+1}) > \alpha(x_i)$.

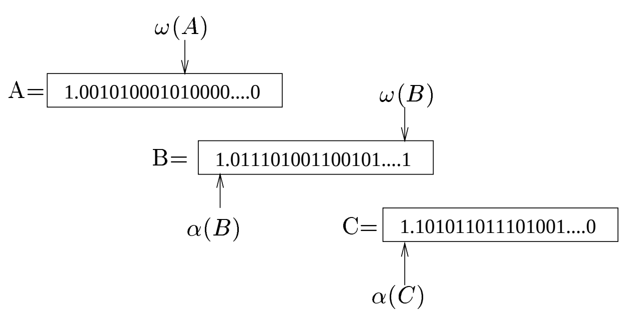


Figure 3: Floating point numbers A and B do not overlap whereas B and C do

For any mathematical operation, **x** is equal to the exact sum of all its components. The addition and the multiplication on expansions are implemented with exact operations. No information is lost. For example, the sum $2^{200} + 1$ is represented by two components 2^{200} and 1. We do not store all the zero bits between the two components.

Priest proposed a non-restoring algorithm for the division of expansions [11, 12]. It initially stores the value of the dividend in the remainder R_0. At iteration i, a new approximate **quotient digit** q_i is guessed from a fair most significant component $\widehat{D}$ of the divisor D and a fair most significant component $\widehat{R}_i$ of the remainder R_i: $q_i = \circ(\widehat{R}_i/\widehat{D})$. The remainder is replaced by $R_{i+1} = R_i - q_i \cdot D$ as presented figure 4.

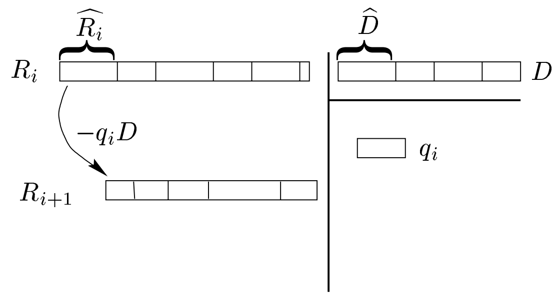


Figure 4: Iteration i of Priest's algorithm: $R_{i+1} = R_i - q_i \cdot D$.

Since R_i is at least divided by a constant factor at each iteration, the algorithm is converging. The quotient may be truncated to a given precision to obtain an approximate expansion with a bounded number of components.

$$\frac{R_0}{D} = \sum_{i=0}^{n-1} q_i + \frac{R_n}{D}$$

3 Modified algorithm

Our algorithm only estimates the most significant digit $\widehat{R}_i$ of the remainder. The less significant components of the dividend are stored unchanged with all the components of the terms $-q_i \cdot D$ that have not been reduced so far. A priority queue extracts the most significant components among this set. These numbers are accumulated until two non zero non overlapping digits $\widehat{R}_i$ and $\widetilde{R}_i$ can be estimated. This process guarantees that $\widehat{R}_i$ is a faithful approximation of R_i. Figure 5 outlines the relative position of the different quantities summed by our algorithm.

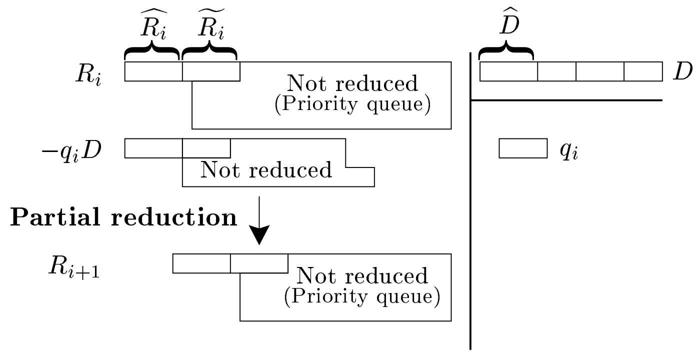


Figure 5: Iteration i of our modified algorithm: estimate $\widehat{R}_{i+1}$ and $\widetilde{R}_{i+1}$.

3.1 Convergence

The new remainder R_{i+1} is the sum of three terms (see equation 9). The quantity $(R_i - \widehat{R}_i)$ represents all the digits of R_i but the most significant one, the second term is a single floating point number and the remaining product is represented as the sum of 2-number expansions sorted by magnitude or as 2 expansions of length $(n-1)$ as n is the length of D.

$$R_{i+1} = (R_i - \widehat{R}_i) + (\widehat{R}_i - q_i \widehat{D}) - q_i(D - \widehat{D}) \tag{9}$$

Our algorithm is converging since the sequence of $|R_i|$ is quickly decreasing as presented below.

$$|R_{i+1}| \leq \left|R_i - \widehat{R_i}\right| + \left|\widehat{R}_i - q_i \widehat{D}\right| + \left|\frac{\widehat{R_i}}{\widehat{D}}\left(\widehat{D} - D\right)\right| \tag{10}$$

We present an upper bound of each term of inequation (10):

- At iteration i, the priority queue contains at most $(2i+1)$ expansions since each $q_i(D - \widehat{D})$ adds two expansions to the queue that contains only R_0 at initialization: one for the upper parts of the individual products $(q_i \times D_j)_H$ and one for the the lower parts $(q_i \times D_j)_L$ for $j = 2 \cdots n$.

$$\begin{aligned} |R_i - \widehat{R_i}| &\leq (2i+1)ulp(\widehat{R_i}) \\ &\leq (2i+1)\alpha(\widehat{R_i})ulp \\ &\leq 2(2i+1)|R_i|ulp \end{aligned}$$

– $\left|\widehat{R}_i - q_i\widehat{D}\right|$ can be reduced from the definition of q_i using as above the properties (1),(6) plus now property (5).

$$\begin{aligned}
\left|\widehat{R}_i - q_i\widehat{D}\right| &= \left|\widehat{D}\left(\frac{\widehat{R_i}}{\widehat{D}} - \circ\left(\frac{\widehat{R_i}}{\widehat{D}}\right)\right)\right| \\
&\leq \left|\widehat{D}\right| \frac{1}{2} \cdot ulp\left(\circ\left(\frac{\widehat{R_i}}{\widehat{D}}\right)\right) \\
&\leq \frac{1}{2}\left|\widehat{D}\right| \cdot 2 \left|\frac{\widehat{R_i}}{\widehat{D}}\right| ulp \\
&\leq 2\left|R_i\right| ulp
\end{aligned}$$

– The last term of inequation (10) is bounded by a function of R_i.

$$\begin{aligned}
\left|\frac{\widehat{R_i}}{\widehat{D}}(\widehat{D} - D)\right| &\leq \left|\frac{\widehat{R_i}}{\widehat{D}}\right| \cdot ulp(\widehat{D}) \\
&\leq \left|\frac{\widehat{R_i}}{\widehat{D}}\right| \alpha(\widehat{D}) \cdot ulp \\
&\leq \left|\frac{\widehat{R_i}}{\widehat{D}}\right| \left|\widehat{D}\right| \cdot ulp \\
&\leq 2\left|R_i\right| \cdot ulp \qquad (11)
\end{aligned}$$

Thus, we can write:

$$|R_{i+1}| \leq (4i+6)ulp \cdot |R_i|$$

Consequently, the algorithm is always converging with a bound as large as $i \leq \frac{ulp^{-1}}{8}$, that leaves plenty of space for an expansion.

3.2 Comparison of complexities

We consider that evaluating the number of floating point operations (additions, multiplications, divisions and comparisons) is sufficient to compare our algorithm with Priest's one. Let n be the length of the divisor, l_i be the length of the remainder at iteration i and m be the number of iterations. The length of the quotient is subsequently directly equal to m.

Both algorithms need n multiplications and $3n$ additions to split the components of the divisor and m multiplications and $3m$ additions to split the components of the quotient digits as they are produced.

In Priest's algorithm, each iteration is broken downto 4 steps as presented in next table: compute a new digit of the quotient; compute the quantity that should be withdrawn from the remainder; compute the new remainder and finally compress it.

Step	1	2	3	4
Addition		$4nm$	$\sum_i(l_i+2n)\times 3$	$\sum_i(l_i+2n)\times 9$
Comparison			$\sum_i(l_i+2n)\times 1$	
Multiplication		$5nm$		
Division	m			

In our algorithm, we can evaluate directly the number of operations from the number of quotient digits produced as presented below. Each quotient digit generates a given number of tasks. Steps 1 and 2 are unchanged except that step 1 also includes a test for exact halting (see section 4). Steps 3 and 4 are replaced by the elimination of the terms of the queue: manage the priority queue and consume the topmost digit if time is correct.

Step	1	2	3	4
Addition	$2m$	$4nm$		$m+2m(n-1)\times 9$
Comparison	m		$2m(n-1)\times\log m$	$m+2m(n-1)$
Multiplication	m	$5nm$		
Division	m			

To bound the complexity of Priest's algorithm, we have to evaluate l_i. In the worst case, we may only know that $l_i \leq l_{i-1}+2n$ leading to $l_i \leq 2in+l_0$. We can however assume that we have a situation close to the one experienced with multiple precision arithmetic based on integers (later on called integer-like algorithm). In this case, we assume that $l_i \leq m-i+n+\lambda$ where λ is a constant.

Both algorithms require $5nm+m+n$ multiplications and m divisions. Our algorithm requires m additional multiplications to preserve exact halting as presented in the next section. The total number of floating point additions and comparisons for each algorithm is given below. Managing the priority queue is quite expensive. We have a better asymptotic complexity than Priest's algorithm in general, but Priest's algorithm is faster on simple cases where the enhanced control given by the priority queue is not necessary.

Algorithm	Additions	Comparisons
Priest's	$12nm^2+16nm+3n+3m$	$2nm+nm(m-1)$
Integer-like	$6m^2+40nm+m(12\lambda-3)+3n$	$3nm+\lambda m+\frac{m(m-1)}{2}$
Modified	$23mn-2m+3n$	$2m(n-1)\log m+2mn$

4 Exact halting

When the quotient may be represented as a floating point expansion, both algorithms find it. They halt after a finite number of steps with the exact quotient. We show that the rounding errors introduced in the approximate guesses of the quotient digits are not sufficient to create a infinite sequence of approximate quotient digits that would converge to the quotient but fail to attain it exactly.

Related questions are entirely new in the literature on floating point operations. Interestingly, we were forced to modify our algorithm to obtain exact halting. This led us to a different proof for Priest's algorithm and for our algorithm.

4.1 Priest's algorithm

We focus our interest on the step k where the algorithm would finish if it finds the correct floating point quotient digit. The difference between the exact result and the sum of the quotient digits we have computed so far, is a single floating point number, $q = R_k/D$. Let q_k be the value computed by the algorithm and q'_k the exact (not rounded) quotient of two fair most significant components of R_k and D: $q'_k = \widehat{R_k}/\widehat{D}$. Here we assume, as Shewchuk conjectured in [15], that the compress algorithm always yield a fair most significant component of an expansion.

$$
\begin{aligned}
|q - q'_k| &= \frac{1}{\widehat{D}} \left| q\widehat{D} - q'_k\widehat{D} \right| \\
&\leq \frac{1}{\widehat{D}} \left(\left| q\widehat{D} - qD \right| + \left| R_k - \widehat{R_k} \right| \right) \\
&\leq \frac{1}{\widehat{D}} \left(\left| q.ulp(\frac{1}{2} + ulp)\alpha(\widehat{D}) \right| + \left| ulp(\frac{1}{2} + ulp)\alpha(\widehat{R_k}) \right| \right) \quad \text{Using property 4} \\
&\leq \frac{1}{\widehat{D}}.ulp(\frac{1}{2} + ulp) \left(2\alpha(q) \cdot \alpha(\widehat{D}) + \alpha(q) \cdot \alpha(\widehat{D}) \right) \quad \text{Using lemma 1} \\
&\leq \frac{1}{\widehat{D}} 4ulp(\frac{1}{2} + ulp) \cdot \alpha(q) \cdot \alpha(\widehat{D}) \\
&\leq \alpha(q) \cdot 4ulp(\frac{1}{2} + ulp) \\
&< \frac{5}{2} \cdot ulp \cdot \alpha(q)
\end{aligned}
$$

As shown on figure 6—where crosses are representable floating point values on the real axis—the value of $|q - q_k|$ equals to 0, 1 or 2 ulps. We have 2 different cases:

- $|q - q_k| = 0$. The algorithm has found the exact result.
- $|q - q_k| = 2^n$. The exact result will be found at the next step.

4.2 Modified division

Since it only estimates the most significant component of the remainder, our algorithm may produce an infinite string of approximated quotient digits. We have inserted a fix in the main loop to avoid this situation.

At each iteration of our algorithm (as well as in Priest's one), the quotient digit has to be split into two half words to compute the components of $-q_i \cdot D$. In our algorithm, if the quotient digit is very close to its higher half word, we consider that the lower half word is not relevant. For example, an estimated quotient digit of 1.01000000000...0001, 1.00111111111...1100 or 1.01000000000...0110, will be replaced by 1.01000000000...0000.

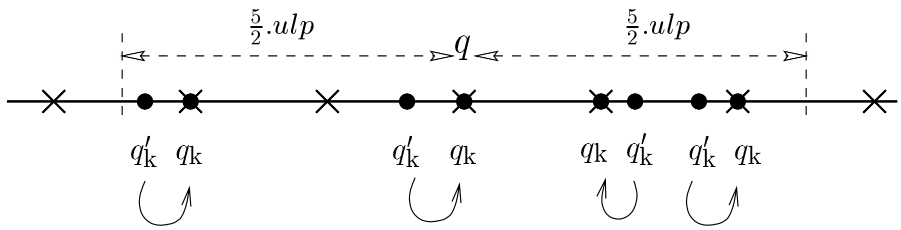


Figure 6: Difference between q and q_k

If the lower part of the digit were correctly estimated, this operation only lengthens the algorithm of one iteration, but it solves our problem. As expected, this event happens rarely in actual applications.

5 Application

As a benchmark, we used expansions to compute the determinant of a small matrix (size 3 to 10). Evaluating the extended sign of a determinant (zero, positive or negative) is a crucial issue in computational geometry. The algorithm has to be fast and error free.

$$\mathcal{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

Let $\mathcal{A}$ be the $n \times n$ square matrix defined above. Let m_{ij} be the minor associated to a_{ij}. To compute the determinant, we may first apply the **column wise development** on $\mathcal{A}$ as follows. We recursively apply the development on the m_{i1} determinants that need to be computed to finish the computation. Some limited dynamic programming was implemented to reduce the total running time.

$$\begin{aligned} det(\mathcal{A}) &= a_{11}m_{11} - a_{21}m_{21} + a_{31}m_{31} - ... + (-1)^{n+1} \cdot a_{n1}m_{n1} \\ &= \sum_{i=1}^{n}((-1)^{i+1}a_{i1} \cdot m_{i1}) \end{aligned}$$

Gaussian elimination uses operations on the lines of the matrix to compute an upper triangular matrix. The product of the terms on the diagonal is directly connected to the determinant of $\mathcal{A}$. We used limited partial pivoting when necessary. Our second algorithm is described below without divisions.

```
tmp=1;
For k = 1 to n-1 do
    For i = k+1 to n do
        For j = k+1 to n do
            a_ij = a_kk * a_ij - a_ik * a_kj
    tmp = tmp * (sign(a_kk))^(n-k-1)
return tmp * sign(a_nn)
```

The magnitude and the number of words necessary to store the intermediate values in our Gaussian elimination grow up very quickly leading to an exponential growth. **Bareiss' method** involves a division at each step to reduce the size of the intermediate coefficients. The division is exact and the quotient is not approximated. The algorithm becomes:

```
simp = 1;
For k = 1 to n-1 do
    For i = k+1 to n do
        For j = k+1 to n do
            a_ij = (a_kk * a_ij - a_ik * a_kj)/simp
    simp = a_kk
return sign(a_nn)
```

We compared the three different algorithms on our package, on GMP and on regular floating point implementations. The later produced constantly erroneous results as soon as the size of the matrices reached 4. We however used them as a reference. The figures 7 and 8 present a ratio of the running time compared to Gaussian elimination on floating point numbers. This ratio is the price to be paid to obtain the correct result. GMP (GNU Multiple Precision package) is a well-known library that uses assembly code on the integer unit to construct very efficient multiple precision operators.

In our test, we have used two different types of matrices: **random matrices** (fig. 7 and 8(a)) and **non invertible matrices** (fig. 8(b)). The former use 57 bit random integers. The later use ill formed 200 bit integers computed so that the rank of the matrix is $n-1$.

For all the tests, column wise development showed good performances with small matrices. Gaussian elimination was running correctly for medium size matrices. Bareiss' elimination is the best algorithm as soon as the additional cost of the division is amortized by the length of the expansion.

The two different machines involved in the test contain a Cyrix 6x86 for the first one and an Intel Pentium Pro for the second one. The double extended 80 bit format was used to store the components of the expansions.

6 Conclusion

We now have a complete library of arithmetic operations on expansions and we have tested the performances of expansions on the computation of a small determinant. Results are quite good and with chosen matrices, expansions have better results than widely used libraries like GMP. For the third diagram, the matrices were carefully chosen to give a little advantage to the expansions over GMP. We succeeded since expansions are there constantly the best package.

This situation is representative of a user that does not want to learn about the peculiarities of his code. As it stands, determinants on floating point expansions are a good solution for a package that should run on any situation. However, a hand tuned package still represents a faster solution for a user with a good experience in this field.

Results on the 6x86 are not very good for expansions compared to GMP. The Cyrix 6x86 is known to have a weak floating point unit compared to its integer unit. The second diagram shows a visible improvement with exactly the same tests and exactly the same program. The expansion library uses the floating point unit whereas GMP uses the integer unit. As long as the floating point unit continues to speed up in comparison to the integer unit, expansions will show impressive results compared to integer libraries for this kind of applications.

Priest's algorithm needs all the components of the divisor and the dividend before it begins computation. Our algorithm only requires a few of their most significant components before it produces the most significant digit of the quotient. We say that our division may work on-line with the most significant digits of its operands first. Yet, the run time system able to take advantage of this property is still to be created. One of us presents in [1, 2] a prototype of such system. Odds remains that the user will be provided individual basic tools rather than a transparent system.

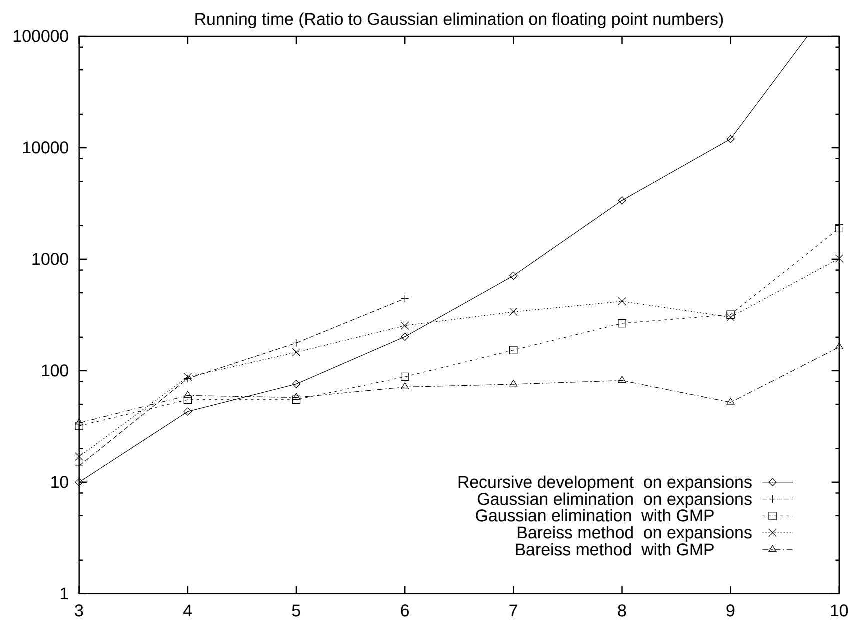
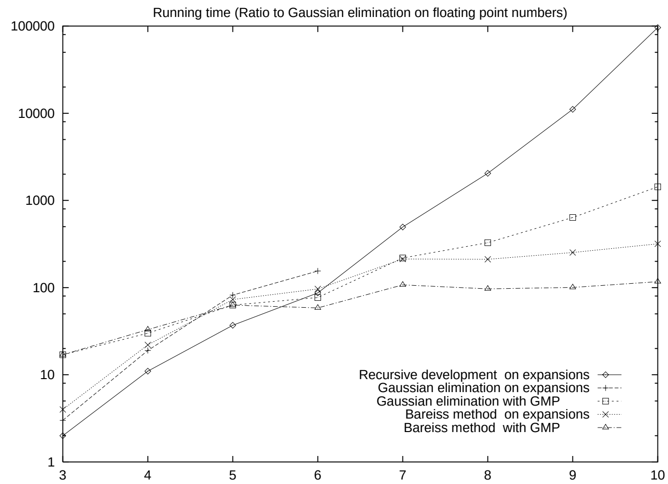
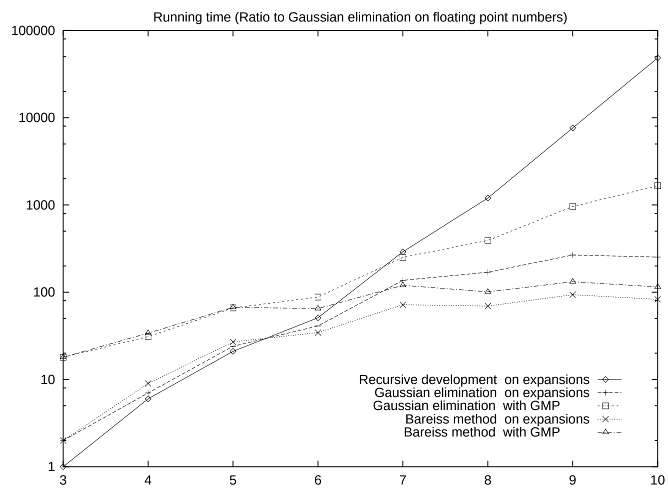


Figure 7: Results on a Cyrix 6x86 with a random matrix.



(a) Random matrix.



(b) Chosen matrix.

Figure 8: Results on an Intel Pentium Pro.

The addition and the multiplication operators implemented in [13, 14, 15, 5] operate least significant digits first. We will in near future transform all these operators to operate on-line. This is the natural step to obtain an adaptive library.

References

[1] D. Berthelot and M. Daumas. A library for real numbers using Cauchy's embedded interval sequences. In *International Conference on Interval Methods and Computer Aided Proofs in Science and Engineering*, pages 23–24, Würzburg, Germany, 1996.
[2] D. Berthelot and M. Daumas. Computing on sequences of embedded intervals. *Reliable Computing*, 3(3):219–227, 1997.
[3] W. J. Cody, R. Karpinski, et al. A proposed radix and word-length independent standard for floating point arithmetic. *IEEE Micro*, 4(4):86–100, 1984.
[4] M. Daumas. Multiplications of floating point expansions. Research report 98-39, Laboratoire de l'Informatique du Parallélisme, Lyon, France, 1998.
[5] M. Daumas. Multiplications of floating point expansions. In *Proceedings of the 14th Symposium on Computer Arithmetic*, Adelaide, Australia, 1999.
[6] T. J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.
[7] D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–47, 1991.
[8] D. Goldberg. *Computer Architecture: A Quantitative Approach*, chapter Computer Arithmetic, pages A1–A77. Morgan Kaufmann, 1996.
[9] J. L. Hennessy and D. A. Patterson. *Computer architecture: A quantitative approach*. Morgan Kaufmann, 1996.
[10] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*. Addison-Wesley, 1981.
[11] I. Koren. *Computer Arithmetic Algorithms*. Prentice Hall, 1993.
[12] J.-M. Muller. *Arithmétique des Ordinateurs*. Masson, 1989.
[13] D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–144, Grenoble, France, 1991. IEEE Computer Society Press.
[14] J. R. Shewchuk. Robust adaptative floating point geometric predicates. In *Proceedings of the 12th Annual ACM Symposium on Computational Geometry*, pages 141–150, Philadelphia, Pensylvania, 1996.
[15] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. In *Discrete and Computational Geometry*, volume 18, pages 305–363, 1997.
[16] B. Shriver and B. Smith. *The Anatomy of a High Performance Microprocessor, A Systems Perspective*. IEEE Computer Society, 1998.
[17] D. Stevenson et al. A proposed standard for binary floating point arithmetic. *IEEE Computer*, 14(3):51–62, 1981.
[18] D. Stevenson et al. An american national standard: IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, 1987.