Multiplications of floating point expansions

Marc Daumas
CNRS - Lab LIP - UMR 8512 - ENS de Lyon - INRIA
46, allée d'Italie - 69634 Lyon Cedex 07 - FRANCE
Marc.Daumas@ENS-Lyon.Fr

Abstract

In modern computers, the floating point unit is the part of the processor delivering the highest computing power and getting most attention from the design team. Performance of any multiple precision application will be dramatically enhanced by adequate use of floating point expansions. We present in this work three multiplication algorithms faster and more integrated than the stepwise algorithm proposed earlier. We have tested these new algorithms on an application that computes the determinant of a matrix. In the absence of overflow or underflow, the process is error free and possibly more efficient than its integer based counterpart.

1. Introduction

Some groups have developed multiple precision packages based on the error-free integer arithmetic to compute some very precise quantities on a computer. Some of the most successful packages available today for a fast accurate scientific computation are GMP (Gnu Multiple Precision), Brent's MP [3] and Bailey's package[1] [1].

Expansions were introduced by Priest [12] based on some earlier compound operations defined to reduce the rounding error of a long expression [11, 10] or to compute an accurate rounding [2]. Actually, Dekker was the first in the past to propose this technique but he restricted himself to doubling the precision available to the user [8].

Priest's expansions adapt to the working radix and the precision of the rounding attained by the floating point unit[2] [9]. IEEE standard commercial floating point units lead to faster algorithms more tightly connected to the machines. In the following, we assume that the floating point unit is IEEE compatible. This assumption provides us with a powerful set of axiomatic properties on floating point operations.

Shewchuk was the first one to present a working library available on the net with an actual application to computational geometry. His library tests if a point is in a circle defined by three other points. It uses adaptive multiple precision arithmetic based only on IEEE standard floating point operations [13, 14]. Although the library relies on expansions, the inputs are limited by choice to common floating point numbers.

In this work, we will focus on the multiplication of expansions with the application of an exact $n \times n$ determinant for n between 3 and 8. We will first present former work, definitions and properties for the floating point numbers and for the expansions in section 2. Section 3 presents three multiplication algorithms all based on the same normalization operation. The running times of our applications are presented in the section 4 for some different values of the inputs. This presentation ends with concluding remarks in the last section.

2. Definitions and properties

2.1. Floating point numbers

A floating point number is stored as a finite length fraction and a bounded exponent. Its value is given below.

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

With this notation a weight can be associated to each bit of the mantissa. Subsequently we call $\alpha(x)$ the weight of the most significant non-zero bit of x (*ie.* $\alpha(x) = 2^{\text{exponent}}$) and $\omega(x)$ the weight of its least significant non-zero bit. Formally, $\alpha(x)$ and $\omega(x)$ can be defined as follows where the notation extends to zero and both infinity quantities defined in the standard such that $\alpha(0) = \omega(0) = 0$ and $\alpha(\pm\infty) = \omega(\pm\infty) = \infty$.

$$\begin{aligned} \alpha(x) &= \inf\left\{w \in \mathbb{N}^* \middle| w = 2^k; k \in \mathbb{Z}; 2w > |x|\right\} \\ \omega(x) &= \inf\left\{w \in \mathbb{N}^* \middle| w = 2^k; k \in \mathbb{Z}; wx \in \mathbb{Z}\right\} \end{aligned}$$

[1] This package uses floating point numbers to store exactly integers in the range $[-2^{2^2}, 2^{2^2}]$.

[2] Priest refined and used the notions of faithful rounding, correct rounding, proper truncation and correct chopping. The reader should refer to his original work to explore the different possible algorithms.

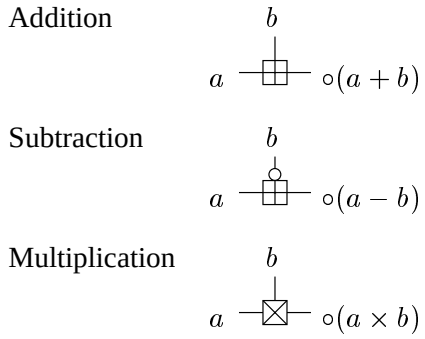


Figure 1. Standard floating point operations

Given a finite precision data type, we can define the **ulp** function that stands for the weight of one unit in the last place of x and that is the weight of the last bit of the mantissa of x. Any non zero normalized bounded floating point number x satisfies $\omega(x) \geq \mathrm{ulp}(x) = \alpha(x)\,\mathrm{ulp}(1)$. We will later use these equations and the fact that α and ulp are monotonous functions. The last equality does not hold for denormalized numbers, yet treating these numbers is beyond our goal in this work.

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.

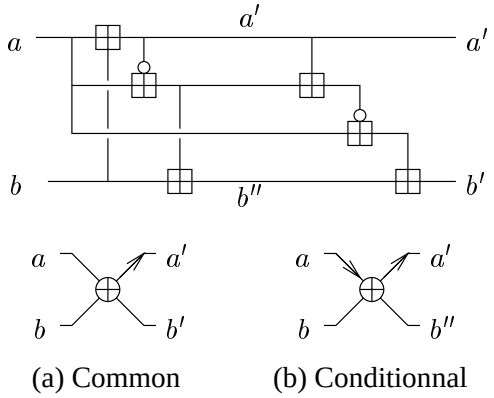
For example, if $a \oplus b$ is the machine floating point addition and $\circ(x)$ is the rounded to nearest value of x for any x, than $a \oplus b = \circ(a + b)$. In cases of tie, the rounded value is the one that has an even mantissa: the mantissa's last digit is a 0. Figure 1 presents the symbol of the three standard floating point operators used in this work.

2.2. Exact sum

Dekker first, then Knuth proposed two exact sum operators. The common exact sum of Figure 2-(a) involves 4 additions and 2 subtractions on an IEEE compliant computer. The result is a pair (a', b') such that $a' = \circ(a + b)$ and $a' + b' = a + b$. The following properties on a' and b' can be easily checked. We will see later that they are sufficient to prove the correct behavior of the new exact multiplication operators.

$$\begin{aligned} \mathrm{ulp}(a') &\leq 2\max(\mathrm{ulp}(a), \mathrm{ulp}(b)) \\ \omega(b') &\geq \min(\omega(a), \omega(b)) \\ \alpha(b') &\leq \min\left(\frac{\mathrm{ulp}(a')}{2}, \frac{\omega(a')}{4}\right) \end{aligned}$$

It has been proved that one can get the correct pair (a', b') by an early exit of the exact sum provided $|b| \leq |a|$.



(a) Common (b) Conditionnal

Figure 2. Exact sum

The conditional sum of Figure 2-(b) returns the exact couple $(a', b'') = (a', b')$ with only 2 additions and a subtraction. We present a better sufficient condition for $b' = b''$ below. The proof of the correctness of the conditional sum is not modified by this new condition and we do not repeat the proof here.

$$\mathrm{ulp}(b) \leq \omega(a)$$

2.3. Expansions

Priest and Shewchuk presented a set of algorithms dealing with expansions where an expansion is defined as a sequence $\mathbf{x} = (x_{n-1}, \cdots, x_1, x_0)$ of floating point numbers. Any component of an expansion may be equal to zero, but the subsequence of the non-zero components x_i must be non overlapping and ordered by magnitude. In a sequence of non zero components, the non overlapping condition is equivalent to $\omega(x_i) > \alpha(x_{i-1})$ for any $i \in \{1 \cdots n-1\}$. The value of the expansion is the exact, not rounded, sum of its components.

Example 1 *Any rational number that can be written as the product $\mathbf{m} \times 2^e$ can be expressed as an expansion as long as $\mathbf{m}$ and e are not too large. We cut the radix 2 representation of $\mathbf{m}$ in slices m_i of length 53 bits. Let p_i be the trailing position of the ith slice, the integer number read in each slice is converted to a floating point value and multiplied by 2^{e+p_i}.*

To store the following number

$$1\,101\,110\,001\,111\,101\,101\,000\,000\,011 \times 2^5$$

a toy four bit word integer based multiple precision system will store seven components plus an exponent.

$$(1101; 1100; 0111; 1101; 1010; 0000; 0011; 5)$$

The above procedure will produce the following ordered set of five floating point numbers in a toy floating point system with 5 fraction bits.

$$\{1.10111 \times 2^{32}; 1.11 \times 2^{23}; 1.1011 \times 2^{20}; 2^{14}; 1.1 \times 2^{6}\}$$

A non-adjacent expansion is an expansion $\mathbf{x} = x_{n-1} + \cdots + x_1 + x_0$ where the non-zero components x_i are also non-adjacent. For a sequence of non zero components we have $\omega(x_i) > 2\alpha(x_{i-1})$ for any $i \in \{1 \cdots n-1\}$: there is at least one bit set to 0 between the two nearest non-zero signed bits of two consecutive components.

Example 2 *Another way to obtain an expansion for a rational $\mathbf{x} = \mathbf{m} \times 2^e$ is to operate by iterated roundings. Let $\square$ be any IEEE standard rounding mode, we define the sequence (x_i, y_i) by the following induction*

$$\left\{ \begin{array}{rcl} y_0 & = & \mathbf{x} \\ x_i & = & \square(y_i) \\ y_{i+1} & = & y_i - x_i \end{array} \right.$$

This sequence is ultimately equal to $(0, 0)$. If (x_n, y_n) is the last non-zero pair, the sequence $(x_0, x_1, \ldots, x_n)$ is an expansion of $\mathbf{x}$. Moreover, if the rounding mode is set to the nearest (even) value, the expansion is non adjacent. In the same condition as example 1 this procedure produces the set

$$\{1.10111 \times 2^{32}; 1.11111 \times 2^{23}; -1.1 \times 2^{15}; 1.1 \times 2^{6}\}$$

2.4. Distillation

The distillation algorithm turns a set of n floating point numbers $\{a_0, a_1, \cdots a_{n-1}\}$ into an expansion. The complexity of this recursive algorithm can be computed by induction. Shewchuk's routine needs $\frac{9n(n-1)}{2} - 3(n-1)$ floating point additions. Priest's first algorithm was actually faster in using a linear addition algorithm with a total of $6n \log n - 9n + 9$ floating point additions and $n \log n - n + 1$ comparisons. Results of complexity on the operations in Shewchuk's 1997 paper [14] are presented in details in [6].

2.5. Multiplication

All existing algorithms compute the product of two expansions $\mathbf{x}$ and $\mathbf{y}$ of length n and m by generating the $n \times m$ componentwise partial products $x_i \times y_j$. They divide each component of each operand into half-words using the split operator of Figure 3 and they compute the exact componentwise partial products $x_i \times y_j$ from these values as presented in the Figure 4.

If an adequate data structure is created for this step of the multiplication, the split operation is done only once for each operand and the complexity of the actual generation of the partial product is $(n + m)$ split operations followed by $n \times m$ exact products. In terms of floating point additions (A) and floating point multiplications (M), it turns to mean:

$$\begin{array}{rcl} A & = & 3 \times (n+m) + 4nm \\ & = & 4nm + 3n + 3m \\ M & = & 5nm + n + m \end{array}$$

For the second step of the multiplication algorithms, Shewchuk proposed to build a distillation tree of the partial products of the first expansion each time scaled by a different component of the second expansion $\mathbf{x} \times y_j$. This algorithm would require $12nm \log m - 9m + 9$ additions, $nm \log m - m + 1$ comparisons plus the necessary floating point operations to generate the componentwise partial products (see [6] for details).

As the preceding loop unfolds, the algorithm computes all the partial products before it starts the distillation and it stores a total number of $2nm$ components. No report of such implementation was published. To avoid such storage needs, Priest decided to use a suboptimal algorithm that accumulates the partial products $\mathbf{x} \times y_j$ as they are generated by the scaling of the first operand. The complexity of these steps is $6nm^2 - 6nm - 9m + 9$ additions, $2nm^2 - 2nm - m + 1$ comparisons. This simple algorithm can be implemented quickly from Shewchuk's library.

Asymptotic complexity is not a good criteria with double precision since the length of both operands is bounded

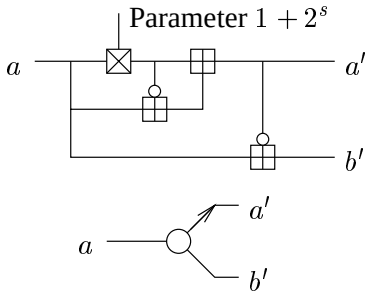


Figure 3. Split a word into half-words

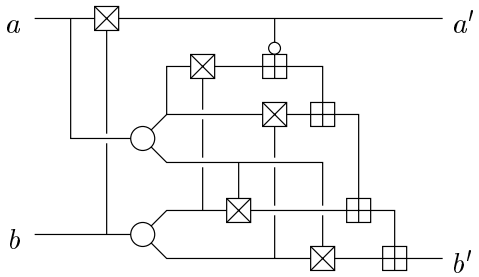


Figure 4. Exact product

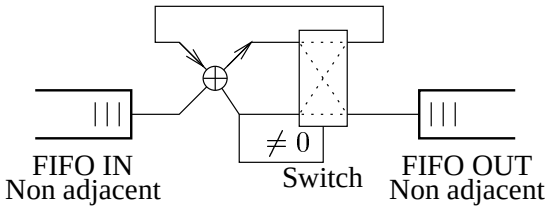


Figure 5. Optimize an expansion (MSD first)

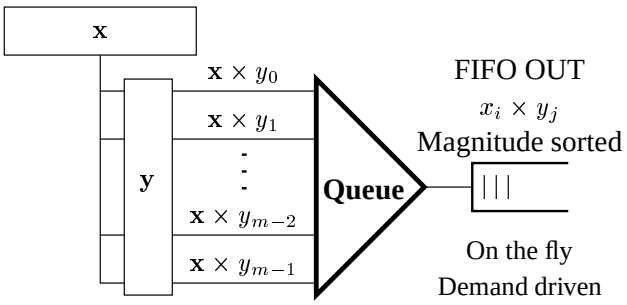


Figure 6. Generation of partial products (LSD first)

by $39 = \left\lceil \frac{2^{11}}{53} \right\rceil$ as long as the compression algorithm is used correctly. We have break this bound allowing up to $512 = \frac{2^{15}}{64}$ components by using the 80 bit double extended precision numbers implemented in hardware on personal computers.

2.6. Optimization

Arithmetic operators on expansions produce non optimal results. It is therefore very important to be able to compress an expansion to reduce its memory occupation. Moreover, the number of elementary floating point operations necessary to compute an exact arithmetic operation on expansions depends on the length of its inputs.

The renormalization algorithm provided by Priest works in two passes, the first pass builds a non overlapping representation of the input and the second pass builds an optimized expansion. Although the second pass seems to work in a quadratic number of operations in regard to the length of the input, Priest proved from numerical properties on the quantities produced that the second pass is limited to $2n$ exact additions.

In our approach, we separate the least significant digit first pass from the most significant digit first one. The first pass is implemented by our new normalization operator presented in the following section. It is slightly more powerful than Priest's one[3]. The second pass is implemented by the very simple operator presented in Figures 5. The switch cell exchanges its two inputs if the control value is non zero. The compression algorithm provided by Shewchuk is weaker since it works only on a non overlapping expansion. Its second pass only involves n exact additions.

3. Multiplication algorithms

To multiply two expansions $\mathbf{x}$ and $\mathbf{y}$, we generate all the terms $x_i \times y_j$ when needed. They are arranged into a priority queue to reduce the complexity in space of this operation (section 3.1). The magnitude sorted list is turned into a non adjacent expansion as it is produced by the priority queue (section 3.2). The expansion is the result of our operator. It can be reduced using a compression operation like optimize (section 2.6).

[3]The reader is invited to refer to Priest's original paper for the condition necessary to a correct behavior of his compression routine.

3.1. Generation of partial products

Since both expansions $\mathbf{x}$ and $\mathbf{y}$ are sorted by magnitude, this problem is equivalent to the well known problem of sorting $\mathbf{X} + \mathbf{Y}$ where $\mathbf{X}$ and $\mathbf{Y}$ are vectors [15].

Whereas some networks are known to solve the $\mathbf{X} + \mathbf{Y}$ sorting problem with a linear complexity in the number of components of the resulting vector we are interested in memory efficient algorithms. Priority queue sorting as proposed in Figure 6 presents both advantages of a good complexity $\Theta(nm \log m)$ identical for the integer operations and for the floating point operations and of a restricted memory occupation of $\Theta(n)$ with on the fly generation of the partial products. The total number of integer and floating point operations, though not necessarily optimal is equivalent to the best algorithm known for the $\mathbf{X} + \mathbf{Y}$ sorting problem.

3.2. Normalization

Given a set of numbers $\{a_0, a_1, \cdots a_{n-1}\}$, if we want to construct an expansion that represents its sum (distillation), we use the normalization algorithm on the set once it is sorted. Iterated least significant digit first, the sum process presented in Figure 7 constructs an expansion.

For a correct behavior of the normalization process, we can use the sum operator defined in Figure 8. The properties required on the inputs at each stage are defined below where n counts the number of iterations. Steps where $in = 0$ are not presented here. The proof is presented in the appendix.

$$\begin{aligned} |a + b| &\leq 2n|in| \\ \alpha(b) &\leq \min(\tfrac{\mathrm{ulp}(a)}{2}, \tfrac{\omega(a)}{4}) \end{aligned}$$

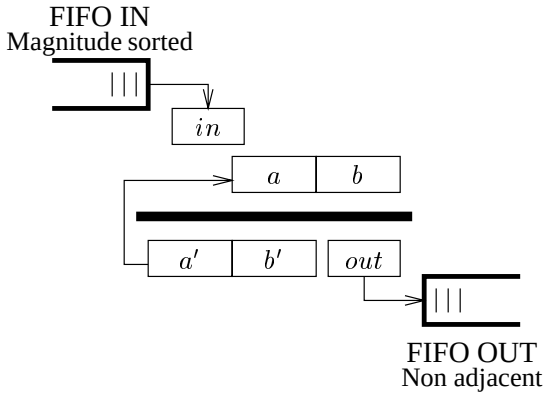


Figure 7. Normalization (LSD first)

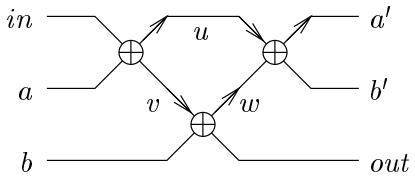


Figure 8. Sum operator for the normalization

The worst case number of additions is $12(n-2)+6$ for this algorithm. The zero-eliminating tests are not counted in this figure as it was for all the algorithms presented earlier. If the set is not already sorted, we can use any sorting algorithm. We get a worst case total of $n \log n - n + 1$ comparisons to sort the set by magnitude using for example a merge sort [5].

3.3. Complexity and generalization

If N is the number of terms computed by the generation of the partial products, the normalization and the compression of the generated list require slightly less than $15N$ floating point additions. Table 1 sums up the complexity of the multiplication of two expansions.

We can change the operator by changing the partial product generator. The algorithm is slightly modified to accommodate a 2×2 determinant of expansions as presented in Figure 9. We choose n and m such that the lengths of the

Table 1. Worst case complexity

Generate partial products	Add	$4nm + 3n + 3m$
	Mult	$5nm + n + m$
Maintain priority queue	Comp	$2nm \log m$
Normalize and compress	Add	$15nm$

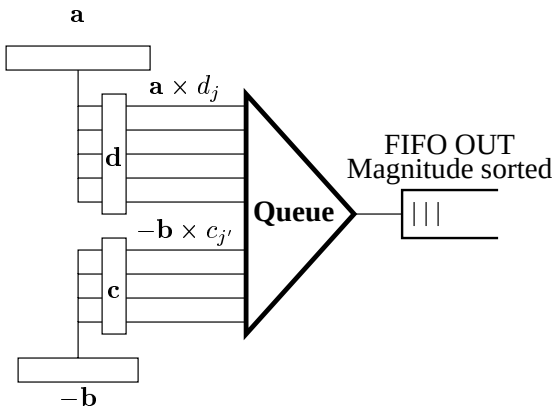


Figure 9. Determinant 2×2

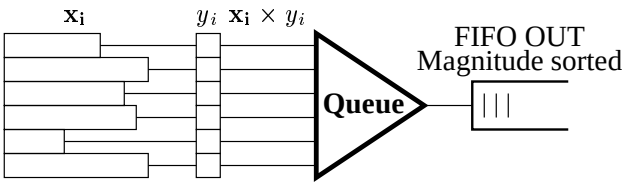


Figure 10. Mixed dot product

first operands $\mathbf{a}$ and $\mathbf{b}$ are bounded by n and the lengths of the second operands $\mathbf{c}$ and $\mathbf{d}$ are bounded by m. One has to double the figures of Table 1 to obtain worst case complexity of a 2×2 determinant.

The dot product of a vector of m expansions by a vector of scalar floating point numbers $\sum_i \mathbf{x_i} \times y_i = \sum_i \sum_j x_{i,j} \times y_i$ is presented in Figure 10. We define n_i to be the length of the expansion $\mathbf{x_i}$ and $n = \max n_i$ the length of the longest expansion. The Table 1 still represents the complexity of the mixed dot-product.

4. Experimentation on the determinants

The main domain of application of our study is in computing determinants of size between 3 and 8. An introduction to the need for an efficient robust evaluation of the sign of determinants is available in [4]. We have implemented three routines: a division-free Gaussian elimination; a purely recursive columnwise development; and a mixed dynamic programming-recursive columnwise development that stores all the necessary 3×3 sub-factors and computes recursively the necessary $k \times k$ sub-determinants.

The division-free Gaussian elimination and the mixed dynamic-recursive procedure present similar performances for an acceptable memory occupancy. With input coefficients ranging in magnitude between 10^{15} and 10^{-15}, some

Table 2. Multiplication of expansions[4]

Priest's	$7nm^2 - 7nm - 10m + 10$
Shewchuk's	$14nm \log m - 10m + 10$
Ours	$2nm \log m + 24nm + 4n + 4m$

Table 3. Distillation

Priest's	$7n \log n - 10n + 10$
Shewchuk's	$\frac{9n^2}{2} - \frac{15n}{2} + 3$
Ours	$n \log n + 11n - 17$

parts of the Gaussian elimination may quickly overflow or underflow as n reaches 10. Since no division is performed, the intermediate quantities are algebraically much more complex than the actual determinant of the matrix. New implementations include division of expansion to implement Bareiss method [7]. The purely recursive procedure is much slower than the two other ones and development of this procedure has been abandoned.

In our first draft with double precision arithmetic, experimentations showed that if compression is performed periodically on the computed expansions, it is very difficult to obtain an expansion more than ten word long for the intermediate quantities and for the determinant. Yet our multiplication reduced the running time from 20% to 50% compared to the naive algorithm suggested by Priest. Since the difference should increase as we turned to 80 bit double extended precision we abandoned development of this simple routine.

We have tested our algorithms on an Intel Pentium Pro and on a Cyrix 6x86 (see Figure 11). Both machines where running Linux with GNU `gcc` compiler version 2.7.2. This version of the compiler does not use Pentium's extended instruction set. Although the result of Gaussian elimination on floating point numbers is almost never correct, we used it for the reference time. The ratios of Figure 11 are the price for the correct result. Ratios of Gaussian elimination on GMP is presented as a comparison with a high speed freely available product containing some machine specific assembly code.

Purely random matrices are built from random generation of all the coefficients. In matrices of rank $n - 1$, the last line is a random linear combination of the $n - 1$ previous lines. These matrices are built to be difficult cases for GMP.

5. Conclusion

We have presented the first set of algorithms to compute multiplications on expansions. The worst case analysis of the algorithms presented in Table 2 shows a significant improvement compared to Priest's feasible routine. Incidently, we are able to propose a simple distillation process with good complexity as shown Table 3.

[4]We count all the floating point operations but the ones necessary to compute the componentwise exact partial products.

For our experimentation on expansions *vs*. common multiple precision packages, we have used two very different processors. The Intel Pentium Pro is a top of the shelf high speed computer meant to be integrated in high speed computers. Cyrix 6x86 is an aging processor. It was known to reach impressive performances on integer operations. Comparatively, the Cyrix 6x86 has a poor floating point unit. A gap has been filled between Figures 11-a and 11-b compare to Figures 11-c and 11-d. It shows that the trade-offs implemented by processor manufacturers will influence the performances of floating point expansions compared to multiple precision packages like GMP. Odds are that new generations of computers will continue in the actual trend of a powerful floating point unit.

As a side result of this work, we have shown that algorithms on floating point numbers can be proved exact using a limited number of well isolated axioms. This is a first step in integrating all these techniques in an automatic proof-checker.

References

[1] D. H. Bailey. Algorithm 719, multiprecision translation and execution of fortran programs. *ACM Transactions on Mathematical Software*, 19(3):288–319, 1993.

[2] G. Bohlender. Floating point computation of functions with maximum accuracy. *IEEE Transactions on Computers*, 26(7):621–632, 1977.

[3] R. P. Brent. Algorithm 524. MP, a fortran multiple precision arithmetic package. *ACM Transactions on Mathemetical Software*, 4(1):71–81, 1978.

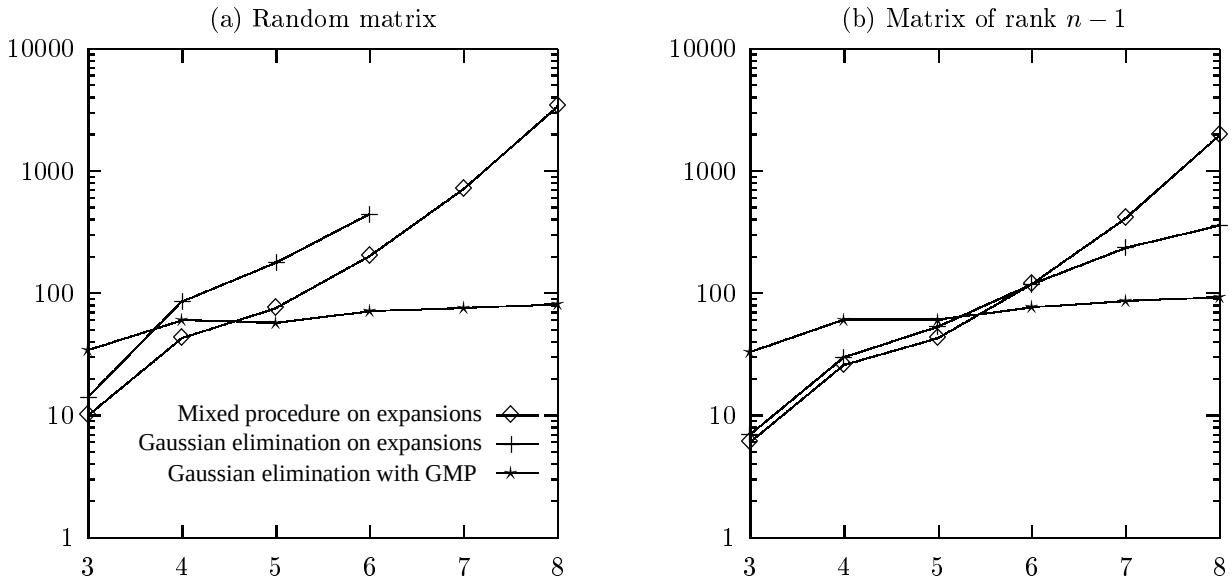
[4] H. Brönnimann and M. Yvinec. Efficient exact evaluation of signs of determinants. In *13th Annual ACM Symposium on Computational Geometry*, Nice, France, 1997.

[5] T. H. Cormen, C. E. Leiserson, and R. R. Rivest. *Intoduction to algorithms*. McGraw Hill, 1991.

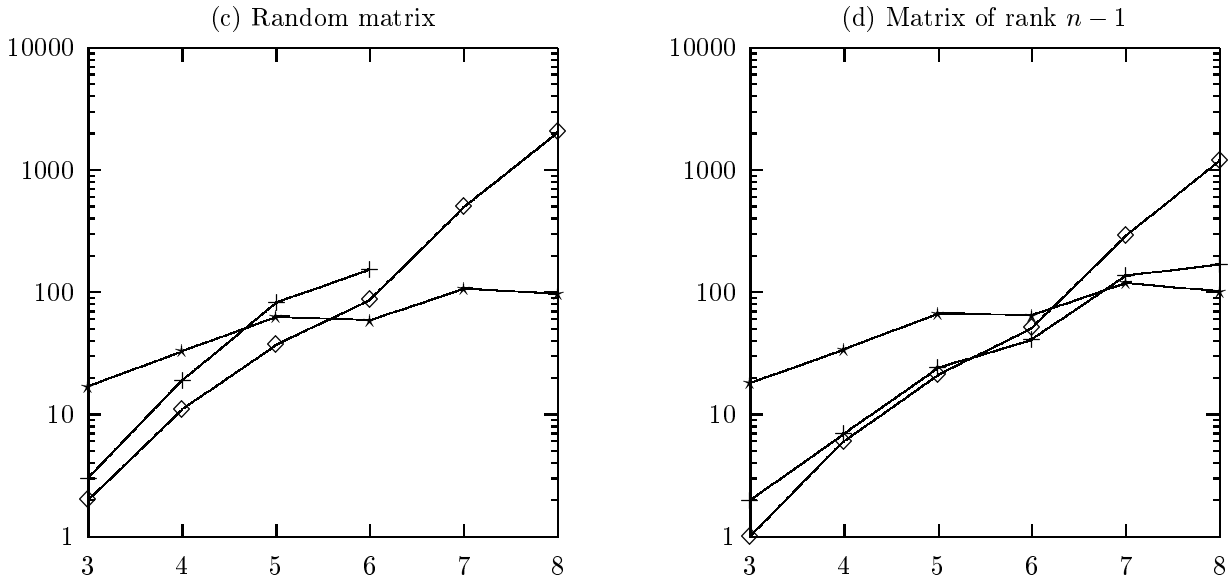
[6] M. Daumas. Multiplications of floating point expansions. Research report 98-39, Laboratoire de l'Informatique du Parallélisme, Lyon, France, 1998.

[7] M. Daumas and C. Finot. Division of floating point expansions. In *IMACS-GAMMM International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, pages 45–46, Budapest, Hungaria, 1998.

[8] T. J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.



Cyrix 6x86 processor



Intel Pentium Pro processor

Ratio to Gaussian elimination on floating point numbers
Common legend on Figure (a)

Figure 11. Running time to compute an exact determinant

[9] D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–47, 1991.
[10] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*. Addison-Wesley, 1981.
[11] M. Pichat. Correction d'une somme en arithmétique à virgule flottante. *Numerische Mathematik*, 19:400–406, 1972.
[12] 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.
[13] 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.
[14] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. In *Discrete and Computational Geometry*, volume 18, pages 305–363, 1997.
[15] W. Steiger and I. Streinu. A pseudo-algorithmic separation of line from pseudo-lines. In *Proceedings of the Sixth Canadian Conference on Computational Geometry*, Saskatoon, Saskatchewan, 1994.

A. Correct behavior of the normalization

If $b = 0$, the whole process is easily proved correct and $out = 0$. Furthermore the following equations are valid.

$$\begin{aligned} |a' + b'| &= |a + in| \\ &\leq |a| + |in| \\ &\leq 2(n+1)|in| \end{aligned}$$

We are now interested in the case where both $a \neq 0$ and $b \neq 0$. We will prove an intermediate result that $4\,\mathrm{ulp}(a) \leq \alpha(in)$ as long as $n \leq \frac{1-\mathrm{ulp}}{16\,\mathrm{ulp}}$ or simpler $n \leq \frac{\mathrm{ulp}^{-1}}{32}$.

$$\begin{aligned} |a| &= |a + b - b| \\ &\leq |a + b| + |b| \\ &\leq 2n|in| + \mathrm{ulp}(a) \\ \alpha(a) &\leq 4n\alpha(in) + \mathrm{ulp}\,\alpha(a) \\ &\leq \tfrac{4n}{1-\mathrm{ulp}}\alpha(in) \\ \mathrm{ulp}(a) &\leq \tfrac{4n\,\mathrm{ulp}}{1-\mathrm{ulp}}\alpha(in) \end{aligned}$$

Since the first addition is unconditional, u and v are correctly defined. The second addition is correct if $v = 0$ or $\mathrm{ulp}(b) \leq \omega(v)$. We prove for later use even that $\omega(v) \geq 8\,\mathrm{ulp}(b)$.

$$\begin{aligned} \omega(v) &\geq \min(\omega(in), \omega(a)) \\ &\geq \min(\mathrm{ulp}\,\alpha(in), \mathrm{ulp}(a)) \\ &\geq \min(4\,\mathrm{ulp}\,\mathrm{ulp}(a), \mathrm{ulp}(a)) \\ &\geq 4\,\mathrm{ulp}\,\mathrm{ulp}(a) \\ &\geq 8\,\mathrm{ulp}\,\alpha(b) \end{aligned}$$

Therefore w and out are correctly defined. For the last addition, we have to prove that either $u = 0$ or $\mathrm{ulp}(w) \leq \omega(u)$. Here again we prove a tighter bound since $\omega(u) \geq 32\,\mathrm{ulp}(w)$ by studying each case of the inequality $\mathrm{ulp}(w) \leq 2\max(\mathrm{ulp}(v), \mathrm{ulp}(b))$.

$$\begin{aligned} \mathrm{ulp}(v) &= \mathrm{ulp}\,\alpha(v) \\ &\leq \mathrm{ulp}\,\omega(u) \\ \omega(u) &\geq 4\alpha(v) \\ &\geq 4\omega(v) \\ &\geq 32\,\mathrm{ulp}(b) \end{aligned}$$

An interesting property for the correct behavior of the transformation is that $\alpha(out) \leq \min(\frac{\mathrm{ulp}(in)}{4}, \frac{\omega(b')}{4})$ to ensure that the list of the out values produced by the operator will be non adjacent and therefore it is an expansion. We prove the first part of the inequality from a case study on $\alpha(out) \leq \frac{\mathrm{ulp}(w)}{2} \leq \max(\mathrm{ulp}(b), \mathrm{ulp}(v))$.

$$\begin{aligned} \mathrm{ulp}(b) &= \mathrm{ulp}\,\alpha(b) \\ &\leq \tfrac{\mathrm{ulp}}{2}\,\mathrm{ulp}(a) \\ &\leq \tfrac{\mathrm{ulp}}{8}\alpha(in) \\ &\leq \tfrac{\mathrm{ulp}(in)}{8} \\ \mathrm{ulp}(v) &= \mathrm{ulp}\,\alpha(v) \\ &\leq \tfrac{\mathrm{ulp}}{2}\,\mathrm{ulp}(u) \\ &\leq \mathrm{ulp}\,\max(\mathrm{ulp}(a), \mathrm{ulp}(in)) \\ &\leq \mathrm{ulp}\,\max(\tfrac{\alpha(in)}{4}, \mathrm{ulp}(in)) \\ &\leq \tfrac{\mathrm{ulp}(in)}{4} \end{aligned}$$

The second part of the inequality is proved as follows since $\omega(u) \geq 32\,\mathrm{ulp}(w) \geq 16\alpha(out)$.

$$\begin{aligned} \omega(b') &\geq \min(\omega(u), \omega(w)) \\ &\geq \min(16\alpha(out), 4\alpha(out)) \end{aligned}$$

The second induction hypothesis on a' and b' is always easy to verify since both a' and b' are produced by the last addition. We only have to prove that $|a'+b'| \leq 2(n+1)|in|$ as follows.

$$\begin{aligned} |a' + b'| &= |a + b + in - out| \\ &\leq |a + b| + |in| + |out| \\ &\leq 2(n+1)|in| \end{aligned}$$