Équipe de Recherche DALI

Laboratoire LP2A, EA 3679
Université de Perpignan

Applications of fast and accurate summation in computational geometry

Stef Graillat

16 mai 2005

Research Report N° RR2005-03

Applications of fast and accurate summation in computational geometry

Stef Graillat

16 mai 2005

Abstract

In this paper, we present a recent algorithm given by Ogita, Rump and Oishi [39] for accurately computing the sum of n floating point numbers. They also give a computational error bound for the computed result. We apply this algorithm in computing determinant and more particularly in computing robust geometric predicates used in computational geometry. We improve existing results that use either a multiprecision libraries or extended large accumulators.

Keywords: accurate summation, finite precision, floating-point arithmetic, computational geometry, robust geometry predicate, determinant

Résumé

Dans ce papier, nous présentons un algorithme récent de Ogita, Rump et Oishi [39] pour sommer de façon précise n nombres flottants. Dans leur papier, Ogita, Rump et Oishi fournissent aussi une borne calculable de l'erreur commise par rapport au résultat exact. Nous appliquons cet algorithme pour calculer des déterminants de matrices et en particulier pour calculer des prédicats géométriques utilisés en géométrie algorithmique. Nous améliorons les algorithmes existants qui utilisent soit la multiprécision soit des accumulateurs très larges.

Mots-clés: sommation précise, précision fine, arithmétique flottante, géométrie algorithmique prédicat géometrique, déterminant

Applications of fast and accurate summation in computational geometry

Stef Graillat[*]

May 16, 2005

Abstract

In this paper, we present a recent algorithm given by Ogita, Rump and Oishi [39] for accurately computing the sum of n floating point numbers. They also give a computational error bound for the computed result. We apply this algorithm in computing determinant and more particularly in computing robust geometric predicates used in computational geometry. We improve existing results that use either a multiprecision libraries or extended large accumulators.

Key words: accurate summation, finite precision, floating-point arithmetic, computational geometry, robust geometry predicate, determinant

AMS Subject Classifications: 15-04, 65G99, 65G50, 65-04

1 Introduction

Floating point summation is one of the most basic operation in scientific computing, and many algorithms have been developped [3, 13, 17, 18, 25, 26, 31, 27, 28, 32, 33, 34, 36, 35, 39, 40, 41, 42, 43, 44, 46, 47]. A good survey of these algorithms is presented in chapter 4 of the book [22] and in the article [21].

Algorithms that make decisions based on geometric test such as determining which side of a line a point falls on, often fail due to roundoff error. A solution to answer these problems is to use software implementations of exact arithmetic often at great expense. Improving the speed of correct geometric computation has received recent attention [4, 11, 30], but the proposals tale integer or rational inputs of small precision. These methods do not seem to be usable if it is important to use ordinary floating point inputs.

A possible way to improve the accuracy is to increase the working precision. For this purpose, some multiprecision libraries have been developed. One can divide those libraries into three parts :

- Arbitrary precision library using a *multiple-digit* format where a number is expressed as a sequence of digits coupled with a single exponent. Examples of this format are Bailey's MPFUN [5, 6], Brent's MP [9], MPFR [2] or GMP [1].

- Extended fixed precision library using the *multiple-component* format but with a limited number of components. Examples of this format are Bailey's *double-double* [7], Brigg's

[*]Laboratoire LP2A, Université de Perpignan, 52, avenue Paul Alduy, F-66860 Perpignan Cedex, France (graillat@univ-perp.fr,http://gala.univ-perp.fr/~graillat).

doubledouble [10] (double-double numbers are represented as an unevaluated sum of a leading double and a trailing double) and *quad-double* [20] (quad-double numbers are represented as an unevaluated sum of four IEEE doubles).

- Arbitrary precision library using a *multiple-component* format where a number is expressed as unevaluated sums of ordinary floating point words. Examples of this format are Priest [42, 43] and Shewchuk [45, 46].

Shewchuk [45, 46] used an arbitrary precision library to obtain fast C implementation of four geometric predicates, the 2D and 3D orientation and incircle tests. The inputs are single or double precision floating point numbers. The speed of these algorithms is due to two features. First, they employ fast algorithms for arbitrary precision arithmetic and second they are adaptive; the running time depends on the degree of uncertainty of the result.

Recently, Demmel and Hida presented algorithms using a wider accumulator [17, 18]. Floating point numbers p_i, $1 \leq i \leq n$, given in working precision with f bits in the mantissa are added in an extra-precise accumulator with F bits, $F > f$. Some algorithms are presented with and without sorting the input data. The authors give a detailed analysis of the accuracy of the computed result depending on f,F and the number of summands.

Those algorithms bear one or more of the following disadvantages:

- sorting of input data is necessary, either
  - by absolute value or,
  - by exponent,
- besides working precision, some extra (higher) precision is necessary,
- access to mantissa and/or exponent is necessary.

Each of those properties can slow down the performance significantly and restrict application fo specific computer architectures or compilers.

In this paper, we use recent algorithms from Ogita, Rump and Oishi [39] to provide fast and accurate algorithms to compute determinants of matrices and geometric predicates. The advantages of these algorithms is that they use one working precision and are adaptive. If the computed result has the wanted relative error, we stop. Otherwise, we continue the computation still in the same working precision. Contrary to Demmel and Hida [18], we do not need extra-precise floating point format and contrary to Shewchuk [45, 46], we do not need renormalisation that slow down the performance.

The rest of the paper is organized as follows. In Section 2, we present basic notation we will use in the rest of the paper and in particular on floating point arithmetic. In Section 3, we recall the so-called error-free transformation introduced by Ogita, Rump and Oishi [39]. In Section 4, we present the summation algorithm of Ogita, Rump and Oishi presented in [39]. They designed accurate and fast algorithm to compute the sum of floating point number. Section 3 and Section 4 borrow heavily from Ogita, Rump and Oishi [39]. In Section 5, we present applications of those algorithm for computing determinant of matrices and robust geometric predicates used in computational geometry.

2 Notations

Throughout the paper, we assume a floating point arithmetic adhering to IEEE 754 floating point standard [23]. We do not address issues of overflow and underflow [16, 19]. The set of floating point numbers is denoted by $\mathbb{F}$, and the relative rounding error by $\mathtt{eps}$. For IEEE 754 double precision we have $\mathtt{eps} = 2^{-53}$.

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 [22]

$$\mathrm{fl}(a \circ b) = (a \circ b)(1 + \varepsilon) \text{ for } \circ = \{+, -, \cdot, /\} \text{ and } |\varepsilon| \le \mathtt{eps}.$$

This implies that

$$|a \circ b - \mathrm{fl}(a \circ b)| \le \mathtt{eps}|a \circ b| \text{ and } |a \circ b - \mathrm{fl}(a \circ b)| \le \mathtt{eps}|\,\mathrm{fl}(a \circ b)| \text{ for } \circ = \{+, -, \cdot, /\}. \tag{2.1}$$

One can notice that $a \circ b \in \mathbb{R}$ and $\mathrm{fl}(a \circ b) \in \mathbb{F}$ but in general we do not have $a \circ b \in \mathbb{F}$. It is known that for the basic operations $+, -, \cdot$, the approximation error of a floating point operation is still a floating point number (see for example [15]):

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

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

We use standard notation for error estimations. The quantities γ_n are defined as usual [22] by

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

3 Error-free transformations

Fortunately, the quantities x and y in (2.2) can be computed exactly in floating point arithmetic. For the algorithm, we use Matlab-like [24] notation. For addition, we can use the following algorithm by Knuth [29, Thm B. p.236].

Algorithm 3.1 (Knuth [29]). Error-free transformation of the sum of two floating point numbers.

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

Another algorithm to compute an error-free transformation is the following algorithm from Dekker [15]. The drawback of this algorithm is that we have $x + y = a + b$ if $|a| \ge |b|$.

Algorithm 3.2 (Dekker [15]). Error-free transformation of the sum of two floating point numbers.

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

For the error-free transformation of a product, we first need to split the input argument into two parts. Let p be given by $\mathtt{eps} = 2^{-p}$ 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 [15] splits a floating point number $a \in \mathbb{F}$ into two parts x and x such that

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

Algorithm 3.3 (Dekker [15]). Error-free split of a floating point number into two part.

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

With this function, an algorithm from Veltkamp (see [15]) enables to compute an error-free transformation for the product of two floating point numbers. This algorithm returns two floating point numbers x and y such that

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

Algorithm 3.4 (Veltkamp [15]). 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 algorithms `TwoSum` and `TwoProduct`.

Theorem 3.1 (Ogita, Rump and Oishi [39, Thm. 3.4]). *Let $a, b \in \mathbb{F}$ and let $x, y \in \mathbb{F}$ such that $[x, y] = \texttt{TwoSum}(a, b)$ (Algorithm 3.1). Then, also in the presence of underflow,*

$$a + b = x + y, \quad x = \text{fl}(a + b), \quad |y| \le \texttt{eps}|x|, \quad |y| \le \texttt{eps}|a + b|. \tag{3.3}$$

Let $a, b \in \mathbb{F}$ and let $x, y \in \mathbb{F}$ such that $[x, y] = \texttt{TwoProduct}(a, b)$ (Algorithm 3.4). Then, if no underflow occurs,

$$a \cdot b = x + y, \quad x = \text{fl}(a \cdot b), \quad |y| \le \texttt{eps}|x|, \quad |y| \le \texttt{eps}|a \cdot b|, \tag{3.4}$$

and, in the presence of underflow,

$$a \cdot b = x + y + 5\eta, \quad x = \text{fl}(a \cdot b), \quad |y| \le \texttt{eps}|x| + 5\texttt{eta}, \\ |y| \le \texttt{eps}|a \cdot b| + 5\texttt{eta} \textit{ with } |\eta| \le \texttt{eta}. \tag{3.5}$$

The `TwoProduct` algorithm can be re-written in a very simple way if a Fused-Multiply-and-Add (`FMA`) operation is available on the targeted architecture [37, 8]. This means that for $a, b, c \in \mathbb{F}$, the result of $\text{FMA}(a, b, c)$ is the nearest floating point number of $a \cdot b + c \in \mathbb{R}$.

Algorithm 3.5 (Ogita, Rump and Oishi [39, Algo. 3.5]). Error-free transformation of the product of two floating point numbers using a `FMA`.

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

If we suppose that we have $\text{ADD3}(a, b, c)$ which compute the nearest floating point number of the exact sum $a + b + c \in \mathbb{R}$ for $a, b, c \in \mathbb{F}$ then the algorithm `TwoSum` can be replaced by the following one.

Algorithm 3.6 (Ogita, Rump and Oishi [39, Algo. 3.6]). Error-free transformation of the sum of two floating point numbers using `ADD3`.

```
function $[x, y] = \texttt{TwoSumADD3}(a, b)$
  $x = a + b$
  $y = \texttt{ADD3}(a, b, -x)$
```

An error-free algorithm for the `FMA` has been recently given by Boldo and Muller [8]. The approximation cannot be represented by a single floating-point number anymore. It is now the sum of two floating point numbers.

Algorithm 3.7 (Boldo and Muller [8]). Error-free transformation of the FMA of three floating point numbers.

```
function $[x, y, z] = \texttt{ErrFMA}(a, b, c)$
  $x = \mathrm{fl}(a \cdot b + c)$
  $[u_1, u_2] = \texttt{TwoProduct}(a, b)$
  $[\alpha_1, \alpha_2] = \texttt{TwoSum}(b, u_2)$
  $[\beta_1, \beta_2] = \texttt{TwoSum}(u_1, \alpha_1)$
  $\gamma = \mathrm{fl}(\mathrm{fl}(\beta_1 - r_1) + \beta_2)$
  $[y, z] = \texttt{TwoSum}(\gamma, \alpha_2)$
```

4 Summation

Let floating point numbers $p_i \in \mathbb{F}$, $1 \le i \le n$, be given. The aim of this section is to present algorithms for Ogita, Rump and Oishi [39] that compute a good approximation of the sum $s = \sum p_i$. In [39], they cascade `TwoSum` Algorithm and sum up the errors to improve the result of the ordinary floating point sum $\mathrm{fl}(\sum p_i)$. Their cascading algorithm summing up errors terms is as follows.

Algorithm 4.1 (Ogita, Rump and Oishi [39, Algo. 4.1]). Cascaded summation.

```
function $\texttt{res} = \texttt{Sum2s}(p)$
  $\pi_1 = p_1;\ \sigma_1 = 0;$
  for $i = 2 : n$
    $[\pi_i; q_i] = \texttt{TwoSum}(\pi_{i-1}, p_i)$
    $\sigma_i = \mathrm{fl}(\sigma_{i-1} + q_i)$
  $\texttt{res} = \mathrm{fl}(\pi_n + \sigma_n)$
```

This is a compensated summation [22]. One can rewrite Algorithm 4.1 in order to overwrite the vector entries. This can be done as follows.

Algorithm 4.2 (Ogita, Rump and Oishi [39, Algo. 4.3]). Error-free vector transformation for summation.

```
function $p = \texttt{VecSum}(p)$
  for $i = 2 : n$
    $[p_i, p_{i-1}] = \texttt{TwoSum}(p_i, p_{i-1})$
```

Using ideas of Algorithm 4.2, Algorithm 4.1 can be written in the following equivalent way.

Algorithm 4.3 (Ogita, Rump and Oishi [39, Algo. 4.4]). Cascaded summation equivalent to Algorithm 4.1.

```
function $\texttt{res} = \texttt{Sum2}(p)$
  for $i = 2 : n$
    $[p_i; p_{i-1}] = \texttt{TwoSum}(p_i, p_{i-1})$
  $\texttt{res} = \mathrm{fl}\left(\left(\sum_{i=1}^{n-1} p_i\right) + p_n\right)$
```

Proposition 4.1 (Ogita, Rump and Oishi [39, Prop. 4.5]). *Suppose Algorithm 4.4 (*Sum2*) is applied to floating point numbers $p_i \in \mathbb{F}, 1 \leq i \leq n$, set $s := \sum p_i \in \mathbb{R}$ and $S := \sum |p_i|$ and suppose $n\texttt{eps} < 1$. Then, also in the presence of underflow,*

$$|\texttt{res} - s| \leq \texttt{eps}|s| + \gamma_{n-1}^2 S. \tag{4.6}$$

The error bound (4.6) for the result res of Algorithm 4.1 is not computable since it involves the exact value s of the sum. The following theorem [39, Cor. 4.7] compute a valid error bound in floating point in round to nearest, which is also less pessimistic.

Proposition 4.2 (Ogita, Rump and Oishi [39, Cor. 4.7]). *Let floating point numbers $p_i \in \mathbb{F}, 1 \leq i \leq n$, be given. Append the statements*

```
if $2n\texttt{eps} \geq 1$, $\texttt{error}('\text{dimension too large}')$, end
$\beta = (2n\texttt{eps}/(1 - 2n\texttt{eps})) \cdot \left(\sum_{i=1}^{n-1} |p_i|\right)$
$\texttt{err} = \texttt{eps}|\texttt{res}| + (\beta + (2\texttt{eps}^2|\texttt{res}| + 3\texttt{eta}))$
```

*(to be executed in working precision) to Algorithm 4.4 (*Sum2*). If the error message is not triggered,* err *satisfies*

$$\texttt{res} - \texttt{err} \leq \sum p_i \leq \texttt{res} + \texttt{err}.$$

A computation shows that the algorithm Sum2 transform a vector p_i begin ill-conditioned with respect to summation into a new vector with identical sum but with a condition number improved by a factor eps. This is why it is interesting to cascade the error-free transformation. The algorithm is as follows.

Algorithm 4.4 (Ogita, Rump and Oishi [39, Algo. 4.8]). Summatin as in K-fold precision by $(K-1)$-fold error-free transformation.

```
function $\texttt{res} = \texttt{SumK}(p, K)$
  for $k = 1 : K - 1$
    $p = \texttt{VecSum}(p)$
  $\texttt{res} = \mathrm{fl}\left(\left(\sum_{i=1}^{n-1} p_i\right) + p_n\right)$
```

The following theorem gives an estimate error for Algorithm 4.4.

Proposition 4.3 (Ogita, Rump and Oishi [39, Prop. 4.10]). *Let floating point numbers $p_i \in F, 1 \leq i \leq n$, be given and assume $4n\texttt{eps} \leq 1$. Then, also in the presence of underflow, the result* res *of Algorithm 4.8 (*SumK*) satisfiees for $K \geq 3$*

$$|\texttt{res} - s| \leq (\texttt{eps} + 3\gamma_{n-1}^2)|s| + \gamma_{2n-2}^K S,$$

where $s := \sum p_i$ and $S := \sum |p_i|$.

5 Applications in robust computational geometry

5.1 Accurate computation of determinant

The literature for the computation of accurate determinant is quite large (see for example [13, 12, 14] and the references therein). They often use multiprecision arithmetic to compute accurately the determinant. Here, we present an algorithm to compute the determinant of a floating-point matrix using only one working precision (IEEE 754 double precision) to compute the result up to a given relative error ε. Let us study the case of a 2×2 determinant

$$\mathrm{det2} = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc.$$

Using `TwoProduct`, we can write $[x, y] = \texttt{TwoProduct}(a, d)$ and $[z, t] = \texttt{TwoProduct}(b, c)$. Then we have $\mathrm{det2} = x + y + z + t$. We have transformed the problem of computing the determinant into a problem of computing a sum accurately. We can then apply the accurate summation algorithm 4.1. to compute this sum.

We can do the same thing for example with a 3×3 determinant

$$\mathrm{det3} = \begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{vmatrix} = \sum_{\sigma \in \mathfrak{S}_3} \mathrm{signature}(\sigma) a_{1,\sigma(1)} \cdot a_{2,\sigma(2)} \cdot a_{3,\sigma(3)},$$

using `ThreeProduct` to transform $a_{1,\sigma(1)} \cdot a_{2,\sigma(2)} \cdot a_{3,\sigma(3)}$ into a sum of four floating point numbers..

Algorithm 5.1. Error-free transformation of the product of three floating point numbers.

```
function $[x, y, z, t] = \texttt{ThreeProduct}(a, b, c)$
  $[p, e] = \texttt{TwoProduct}(a, b)$
  $[x, y] = \texttt{TwoProduct}(p, c)$
  $[z, t] = \texttt{TwoProduct}(e, c)$
```

This can be used to compute for example the area of a planar triangle or the volume of a tetrahedron as shown in [38].

The following algorithm computes the determinant of a matrix up to a give relative error. We suppose we have a function `DetVector` that transforms the computation of the determinant into a summation like mentioned above. We then compute the sum and compute an error bound. If this error bound is less than the desired relative error ε then we stop. Otherwise, we continue the computation.

Algorithm 5.2. Algorithm to compute the determinant up to a relative error ε.

```
function resdet $= \texttt{det}(A, \varepsilon)$
  $p = \texttt{DetVector}(A)$
  $p = \texttt{VecSum}(p)$
  $\texttt{res} = p_n$
  $\beta = (2n\texttt{eps}/(1 - 2n\texttt{eps})) \cdot \left(\sum_{i=1}^{n-1} |p_i|\right)$
  $\texttt{err} = \texttt{eps}|\texttt{res}| + (\beta + (2\texttt{eps}^2|\texttt{res}|))$
  while $(\texttt{err} > \varepsilon|\texttt{res}|)$
    $p = \texttt{VecSum}(p)$
    $\mathrm{res} = p_n$
    $\beta = (2n\texttt{eps}/(1 - 2n\texttt{eps})) \cdot \left(\sum_{i=1}^{n-1} |p_i|\right)$
    $\texttt{err} = \texttt{eps}|\texttt{res}| + (\beta + (2\texttt{eps}^2|\texttt{res}|))$
  $\texttt{resdet} = p_n$
```

5.2 Robust geometric predicates

An application requiring guaranteed accuracy is the computation of geometric predicates. Some algorithm like Delaunay triangulation and mesh generation need self-consistent geometric tests. Shewchuck [45, 46] gives a survey of the issues involved and presents an adaptative (and arbitrary) precision floating point arithmetic to solve this problem. Most of these geometric predicates require the computation of the sign of a small determinant. Recent work on this issue are [45, 46, 4, 11, 30, 18]. Consider for example the predicate ORIENT3D which determines whether a point D is to the left or right if the oriented plane defined by the points A, B and C. The result of the predicate depend on the sign of the determinant

$$\textsc{Orient3D}(a,b,c,d) = \text{sign}\begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix} = \text{sign}\begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z \\ b_x - d_x & b_y - d_y & b_z - d_z \\ c_x - d_x & c_y - d_y & c_z - d_z \end{vmatrix}.$$

The computed result is of the same sign that the exact result if the relative error is less that one. As a consequence, it is sufficient to compute an approximate value of the determinant only with a relative error less than one. Recently, Demmel and Hida [17, 18] provide another method to certified the sign of a small determinant. They used an accurate sommation algorithm using large accumulators. Here, we use algorithms developped in [39] with accurate computation of an error bound. Indeed, the determinant can be evaluated as a sum of 24 monomials of the form $\pm a_i b_j c_k$. Each monomial can be expressed as the sum of four numbers with the algorithm `ThreeProduct`. It follows that the determinant can be expressed as a sum of $4 \times 24 = 96$ terms. We can then apply a summation algorithm until the relative error is less than one. This can be done using the error bound of Proposition 4.2. In the following algorithm, we suppose we have a function `DetVector` that transforms the matrix of the determinant into a vector whose the sum is the determinant (using `ThreeProduct`). We denote by n the length of the vector (here $n = 94$).

Algorithm 5.3. Algorithm to compute the predicate ORIENT3D(a,b,c,d).

```
function sign = $\texttt{Orient3D}(A)$
  $p = \texttt{DetVector}(A)$
  $p = \texttt{VecSum}(p)$
  $\texttt{res} = p_n$
  $\beta = (2n\texttt{eps}/(1 - 2n\texttt{eps})) \cdot \left(\sum_{i=1}^{n-1} |p_i|\right)$
  $\texttt{err} = \texttt{eps}|\texttt{res}| + (\beta + (2\texttt{eps}^2|\texttt{res}|))$
  while ($\texttt{err} > |\texttt{res}|$)
    $p = \texttt{VecSum}(p)$
    $\text{res} = p_n$
    $\beta = (2n\texttt{eps}/(1 - 2n\texttt{eps})) \cdot \left(\sum_{i=1}^{n-1} |p_i|\right)$
    $\texttt{err} = \texttt{eps}|\texttt{res}| + (\beta + (2\texttt{eps}^2|\texttt{res}|))$
  $\texttt{sign} = \text{sign}(p_n)$
```

6 Conclusion

We have tested one geometric predicate that is ORIENT3D. Of course, the same techniques can be easily applied to compute other predicates like INCIRCLE and INSPHERE for example (see [45, 46]. The potential of our algorithms is in providing a fast and simple way to extend slightly the precision of critical variable in numerical algorithms. The techniques used here are simple enough to be coded directly in numerical algorithms, avoiding function call overhead and conversion costs.

References

[1] *GMP, the GNU Multi-Precision library.* Available at URL = `http://www.swox.com/gmp/`.

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

[3] I. J. Anderson. A distillation algorithm for floating-point summation. *SIAM J. Sci. Comput.*, 20(5):1797–1806 (electronic), 1999.

[4] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. P. Preparata, and M. Yvinec. Evaluating signs of determinants using single-precision arithmetic. *Algorithmica*, 17(2):111–132, 1997.

[5] David H. Bailey. Algorithm 719; multiprecision translation and execution of fortran programs. *ACM Trans. Math. Softw.*, 19(3):288–319, 1993.

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

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

[8] Sylvie Boldo and Jean-Michel Muller. Some functions computable with a fused-mac. In *Proceedings of the 17th Symposium on Computer Arithmetic*, Cape Cod, USA, 2005.

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

[10] Keith Briggs. *Doubledouble floating point arithmetic*, 1998. Available at URL = `http://members.lycos.co.uk/keithmbriggs/doubledouble.html`.

[11] H. Brönnimann and M. Yvinec. Efficient exact evaluation of signs of determinants. *Algorithmica*, 27(1):21–56, 2000.

[12] Hervé Brönnimann and Mariette Yvinec. Efficient exact evaluation of signs of determinants. In *SCG '97: Proceedings of the thirteenth annual symposium on Computational geometry*, pages 166–173, New York, NY, USA, 1997. ACM Press.

[13] Kenneth L. Clarkson. Safe and effective determinant evaluation. In *33rd annual symposium on Foundations of computer science (FOCS). Proceedings, Pittsburgh, PA, USA, October 24-27, 1992. Washington, DC: IEEE Computer Society Press, 387-395* .

[14] Marc Daumas and Claire Finot. Division of floating point expansions with an application to the computation of a determinant. *J. UCS*, 5(6):323–338, 1999.

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

[16] James Demmel. Underflow and the reliability of numerical software. *SIAM J. Sci. Statist. Comput.*, 5(4):887–919, 1984.

[17] James Demmel and Yozo Hida. Accurate and efficient floating point summation. *SIAM J. Sci. Comput.*, 25(4):1214–1248 (electronic), 2003.

[18] James Demmel and Yozo Hida. Fast and accurate floating point summation with application to computational geometry. *Numer. Algorithms*, 37(1-4):101–112, 2004.

[19] John R. Hauser. Handling floating-point exceptions in numeric programs. *ACM Trans. Program. Lang. Syst.*, 18(2):139–174, 1996.

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

[21] Nicholas J. Higham. The accuracy of floating point summation. *SIAM J. Sci. Comput.*, 14(4):783–799, 1993.

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

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

[24] The MathWorks Inc. *MATLAB User's Guide*, 2000.

[25] M. Jankowski, A. Smoktunowicz, and H. Woźniakowski. A note on floating-point summation of very many terms. *J. Information Processing and Cybernetics-EIK*, 19(9):435–440, 1983.

[26] M. Jankowski and H. Woźniakowski. The accurate solution of certain continuous problems using only single precision arithmetic. *BIT*, 25:635–651, 1985.

[27] W. Kahan. Further remarks on reducing truncation errors. *J. Assoc. Comput. Mach.*, 8(1):40, 1965.

[28] W. Kahan. A survey of error analysis. In *Proc. IFIP Congress, Ljubljana*, Information Processing 71, pages 1214–1239, Amsterdam, The Netherlands, 1972. North-Holland.

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

[30] Shankar Krishnan, Mark Foskey, Tim Culver, John Keyser, and Dinesh Manocha. Precise: efficient multiprecision evaluation of algebraic roots and predicates for reliable geometric computation. In *SCG '01: Proceedings of the seventeenth annual symposium on Computational geometry*, pages 274–283, New York, NY, USA, 2001. ACM Press.

[31] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J. Yoo. Design, implementation and testing of extended and mixed precision blas. *ACM Trans. Math. Softw.*, 28(2):152–205, 2002.

[32] Seppo Linnainmaa. Analysis of some known methods of improving the accuracy of floating-point sums. *BIT*, 14:167–202, 1974.

[33] Seppo Linnainmaa. Software for doubled-precision floating-point computations. *ACM Trans. Math. Software*, 7(3):272–283, 1981.

[34] Michael A. Malcolm. On accurate floating-point summation. *Comm. ACM*, 14(11):731–736, 1971.

[35] Ole Møller. Note on quasi double-precision. *BIT*, 5:251–255, 1965.

[36] Ole Møller. Quasi double-precision in floating point addition. *BIT*, 5:37–50, 1965.

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

[38] Yves Nievergelt. Analysis and applications of priest's distillation. *ACM Trans. Math. Softw.*, 30(4):402–433, 2004.

[39] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM J. Sci. Comput.*, 2005. to appear.

[40] M. Pichat. Correction d'une somme en arithmétique à virgule flottante. *Numer. Math.*, 19:400–406, 1972.

[41] Michèle Pichat. *Contributions à l'etude des erreurs d'arrondi en arithmétique à virgule flottante*. PhD thesis, Université Scientifique et Médicale de Grenoble, Grenoble, France, 1976.

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

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

[44] D. R. Ross. Reducing truncation errors using cascading accumulators. *J. Assoc. Comput. Mach.*, 8(1):32–33, 1965.

[45] Johnathan Richard Shewchuk. Robust adaptive floating-point geometric predicates. In *SCG '96: Proceedings of the twelfth annual symposium on Computational geometry*, pages 141–150, New York, NY, USA, 1996. ACM Press.

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

[47] Jack M. Wolfe. Reducing truncation errors by programming. *Comm. ACM*, 7(6):355–356, 1964.