Applications of fast and accurate summation in computational geometry

Stef Graillat (graillat@univ-perp.fr)
Équipe DALI, Laboratoire LP2A, Université de Perpignan Via Domitia, 52, avenue Paul Alduy, F-66860 Perpignan Cedex, France

Nicolas Louvet (nicolas.louvet@univ-perp.fr)
Équipe DALI, Laboratoire LP2A, Université de Perpignan Via Domitia, 52, avenue Paul Alduy, F-66860 Perpignan Cedex, France

January 14, 2006

Abstract. The aim of the paper is to provide fast and accurate algorithms for computing determinants and robust geometric predicates used in computational geometry. We use a recent algorithm given by Ogita, Rump and Oishi (Ogita et al., 2005) that computes accurately the sum of n floating point numbers with a valid error bound. We use this error bound to provide an adaptive algorithm that computes determinants up to a given relative accuracy. We apply this algorithm to computed robust geometric predicates and more particularly the 3D orientation predicate.

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

AMS Subject Classification: 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 developed to accurately perform it in finite precision arithmetic. A good survey of these algorithms is presented in chapter 4 of the book (Higham, 2002).

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 (Avnaim et al., 1997; Brönnimann and Yvinec, 2000; Krishnan et al., 2001), but the proposals take integer or rational inputs of small precision. These methods are not usable for floating point inputs.

In such cases, a possible way to improve the accuracy is to increase the working precision. For this purpose, some multiprecision libraries have been developed and used in computational geometry. One can divide those libraries into three categories.

- Arbitrary precision library using a *multiple-digit* format where a number is expressed as a sequence of digits coupled with a single exponent. Examples of this format are Bailey's MPFUN (Bailey, 1995), Brent's MP (Brent, 1978) or MPFR (MPFR, 2005).
- 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 (Priest, 1992) and Shewchuk (Shewchuk, 1997).
- 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* (Bailey, 2001) (double-double numbers are represented as an unevaluated sum of a leading double and a trailing double) and *quad-double* (Hida et al., 2001) (quad-double numbers are represented as an unevaluated sum of four IEEE doubles).

Shewchuk (Shewchuk, 1997) uses 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 :

1. fast algorithms for arbitrary precision arithmetic, and
2. adaptive implementation; the running time depends on the degree of uncertainty of the result.

Recently, Demmel and Hida presented algorithms using a wider accumulator (Demmel and Hida, 2004). 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 performances significantly and restrict application to specific computer architectures or compilers.

In this paper, we use recent algorithms from Ogita, Rump and Oishi (Ogita et al., 2005) to provide fast and accurate algorithms that compute determinants of matrices and geometric predicates. The advantages of these algorithms is that they use one working precision still being adaptive. These algorithms stop when the computed result has the wanted relative error. Otherwise, the computation continue to increase the accuracy of the result using the same working precision. Contrary to Demmel and Hida (Demmel and Hida, 2004), we do not need extra-precise floating point format and contrary to Shewchuk (Shewchuk, 1997), we do not need renormalizations that slow down the running time performances.

The paper is organized as follows. In Section 2, we present basic notations used in the rest of the paper and in particular for floating point arithmetic. In Section 3, we recall the so-called error-free transformations introduced by Ogita, Rump and Oishi (Ogita et al., 2005). In Section 4, we present the summation algorithm of Ogita, Rump and Oishi presented in (Ogita et al., 2005). They designed accurate and fast algorithms to compute the sum of floating point number. Section 3 and Section 4 borrow heavily from Ogita, Rump and Oishi (Ogita et al., 2005). In Section 5, we present applications of those algorithms 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 (IEEE Computer Society, 1985). We do not address issues of overflow and underflow. The set of floating point numbers is denoted by $\mathbf{F}$, and the relative rounding error by $\texttt{eps}$. For IEEE 754 double precision we have $\texttt{eps} = 2^{-53}$.

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 (Higham, 2002)

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

This implies that

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

One can notice that $a \circ b \in \mathbb{R}$ and $\mathrm{fl}(a \circ b) \in \mathbf{F}$ but in general we do not have $a \circ b \in \mathbf{F}$. 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 (Dekker, 1971)):

$$\begin{array}{lll} x = \mathrm{fl}(a \pm b) & \Rightarrow\ a \pm b = x + y & \text{with } y \in \mathbf{F} \\ x = \mathrm{fl}(a \cdot b) & \Rightarrow\ a \cdot b = x + y & \text{with } y \in \mathbf{F} \end{array} \quad (2)$$

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 (Higham, 2002) by

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

3. Error-free transformations

Fortunately, the quantities x and y in (2) can be computed exactly in floating point arithmetic. For the algorithms, we use Matlab-like notations. For addition, we can use the following algorithm by Knuth (Knuth, 1998, Thm B. p.236).

ALGORITHM 1 (Knuth (Knuth, 1998)). *Error-free transformation of the sum of two floating point numbers.*

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

Another algorithm to compute an error-free transformation is the following algorithm from Dekker (Dekker, 1971). The drawback of this algorithm is that we have $x + y = a + b$ if $|a| \geq |b|$.

ALGORITHM 2 (Dekker (Dekker, 1971)). *Error-free transformation of the sum of two floating point numbers.*

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

For the error-free transformation of a product, we first need to split the input argument into two parts. Let p be given by $\texttt{eps} = 2^{-p}$ and define $s = \lceil p/2 \rceil$. For example, if the working precision is IEEE 754 double precision, then $p = 53$ and $s = 27$. The following algorithm by Dekker (Dekker, 1971) splits a floating point number $a \in \mathbf{F}$ into two parts x and y such that

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

ALGORITHM 3 (Dekker (Dekker, 1971)). *Error-free split of a floating point number into two parts.*

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

With this function, an algorithm from Veltkamp (see (Dekker, 1971)) enables to compute an error-free transformation for the product of two floating point numbers. This algorithm returns two floating point numbers x and y such that

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

ALGORITHM 4 (Veltkamp (Dekker, 1971)). *Error-free transformation of the product of two floating point numbers.*

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

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

THEOREM 1 (Ogita, Rump and Oishi (Ogita et al., 2005, Thm. 3.4)). *Let $a, b \in \mathbf{F}$ and let $x, y \in \mathbf{F}$ such that $[x, y] = \mathtt{TwoSum}(a, b)$ (Algorithm 1). Then,*

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

Let $a, b \in \mathbf{F}$ and let $x, y \in \mathbf{F}$ such that $[x, y] = \mathtt{TwoProduct}(a, b)$ (Algorithm 4). Then,

$$a \cdot b = x + y, \quad x = \mathrm{fl}(a \cdot b), \quad |y| \le \mathtt{eps}|x|, \quad |y| \le \mathtt{eps}|a \cdot b|. \tag{4}$$

4. Summation

Let floating point numbers $p_i \in \mathbf{F}$, $1 \le i \le n$, be given. The aim of this section is to present algorithms from Ogita, Rump and Oishi (Ogita et al., 2005) that compute a good approximation of the sum $s = \sum p_i$. In (Ogita et al., 2005), they cascade `TwoSum` Algorithm and sum up the errors to improve the result of the ordinary floating point summation $\mathrm{fl}(\sum p_i)$. We present hereafter their cascading algorithm. It is based on the following error-free transformation `VecSum` that transforms the vector p into a new vector with identical sum but with a condition number improved by a factor `eps`.

ALGORITHM 5 (Ogita, Rump and Oishi (Ogita et al., 2005, Algo. 4.3)). *Error-free vector transformation for summation.*

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

We describe now the cascaded summation algorithm.

ALGORITHM 6 (Ogita, Rump and Oishi (Ogita et al., 2005, Algo. 4.4)). *Cascaded summation.*

```
function $\mathtt{res} = \mathtt{Sum2}(p)$
  $p = \mathtt{VecSum}(p)$
  $\mathtt{res} = \mathrm{fl}\left(\left(\sum_{i=1}^{n-1} p_i\right) + p_n\right)$
```

The algorithm `Sum2` satisfies the following error bound, which means that the computed result `res` is as accurate as if the sum was computed in twice working precision.

PROPOSITION 1 (Ogita, Rump and Oishi (Ogita et al., 2005, Prop. 4.5)). *Suppose Algorithm 4.4 (`Sum2`) is applied to floating point numbers $p_i \in \mathbf{F}, 1 \le i \le n$, set $s := \sum p_i \in \mathbb{R}$ and $S := \sum |p_i|$ and suppose $n\mathtt{eps} < 1$. Then,*

$$|\mathtt{res} - s| \le \mathtt{eps}|s| + \gamma_{n-1}^2 S. \tag{5}$$

The error bound (5) for the result `res` of Algorithm 6 is not computable since it involves the exact value s of the sum. The following theorem (Ogita et al., 2005, Cor. 4.7) computes a valid error bound in floating point in round to nearest, which is also less pessimistic than (5).

PROPOSITION 2 (Ogita, Rump and Oishi (Ogita et al., 2005, Cor. 4.7)). *Let floating point numbers $p_i \in \mathbf{F}, 1 \le i \le n$, be given. Append the statements*

```
if $2n\mathtt{eps} \ge 1$, error(“dimension too large”), end
$\beta = (2n\mathtt{eps}/(1-2n\mathtt{eps})) \cdot \left(\sum_{i=1}^{n-1} |p_i|\right)$
$\mathtt{err} = \mathtt{eps}|\mathtt{res}| + (\beta + (2\mathtt{eps}^2|\mathtt{res}|))$
```

to Algorithm 4.4 (`Sum2`*). If the error message is not triggered,* `err` *satisfies*

$$\mathtt{res} - \mathtt{err} \le \sum p_i \le \mathtt{res} + \mathtt{err}.$$

It may be interesting to cascade the error-free transformation. The algorithm is as follows.

ALGORITHM 7 (Ogita, Rump and Oishi (Ogita et al., 2005, Algo. 4.8)). *Summation as in K-fold precision by $(K-1)$-fold error-free transformation.*

```
function res = SumK($p, K$)
  for $k = 1 : K-1$
    $p$ = VecSum($p$)
  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 7 which means that the computed result `res` is as accurate as if the sum was computed in K-fold working precision.

PROPOSITION 3 (Ogita, Rump and Oishi (Ogita et al., 2005, Prop. 4.10)). *Let floating point numbers $p_i \in \mathbf{F}, 1 \le i \le n$, be given and assume $4n\mathtt{eps} \le 1$. Then, the result* `res` *of Algorithm 4.8 (*`SumK`*) satisfies for $K \ge 3$*

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

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

The following proposition gives a valid error bound in pure floating point in round to nearest for the classical summation algorithm. This bound is more pessimistic than the one given by Proposition 2 but does not need the computation of the error in each operation.

PROPOSITION 4. *Let floating point numbers $p_i \in \mathbf{F}$, $1 \le i \le n$, be given and assume that $2n\mathtt{eps} \le 1$. Then, the result $\mathtt{res} = \mathrm{fl}(\sum p_i)$ satisfies*

$$|\mathtt{res} - s| \le \mathrm{fl}(\gamma_{2n} \sum |p_i|).$$

Proof. It is shown in (Higham, 2002, Lem. 8.4) that

$$|\mathtt{res} - s| \le \gamma_{n-1} \sum |p_i|.$$

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

$$\gamma_m \le (1 - \mathtt{eps})^{-1} \,\mathrm{fl}(\gamma_m).$$

Moreover, a simple induction leads to

$$\sum |p_i| \le (1 + \mathtt{eps})^{n-1} \,\mathrm{fl}(\sum |p_i|).$$

It follows that

$$\begin{aligned}
|\mathtt{res} - s| &\le \gamma_{n-1} \sum |p_i| \\
&\le \gamma_{n-1}(1 + \mathtt{eps})^{n-1} \,\mathrm{fl}(\sum |p_i|) \\
&\le \gamma_{2n-2} \,\mathrm{fl}(\sum |p_i|) \;\; \text{since} \;\; (1 + \mathtt{eps})\gamma_n \le \gamma_{n+1} \\
&\le (1 - \mathtt{eps})^2 \gamma_{2n} \,\mathrm{fl}(\sum |p_i|) \;\; \text{since} \;\; \gamma_n \le (1 - \mathtt{eps})\gamma_{n+1} \\
&\le (1 - \mathtt{eps}) \,\mathrm{fl}(\gamma_{2n}) \,\mathrm{fl}(\sum |p_i|) \\
&\le \mathrm{fl}(\gamma_{2n} \sum |p_i|).
\end{aligned}$$

This concludes the proof.

5. Applications in robust computational geometry

5.1. Accurate computation of determinant

The literature for the computation of accurate determinant is quite important (see for example (Clarkson, 1992; Brönnimann and Yvinec, 1997; Daumas and Finot, 1999) and the references therein). They often use multiprecision arithmetic to compute accurately the determinant. Here, we present how to compute the determinant of a floating-point matrix up to a given relative error ε using only one working precision (here the IEEE 754 double precision). Let us study the case of a 2×2 determinant

$$\mathrm{det2} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} = a \cdot d - b \cdot c.$$

Using `TwoProduct`, we can write $[x, y] = \mathtt{TwoProduct}(a, d)$ and $[z, t] = \mathtt{TwoProduct}(b, c)$. Then we have $\mathrm{det2} = x + y + z + t$. We transformed the computation of the 2×2 determinant into the computation of a sum of four floating point numbers. We can then apply the accurate summation Algorithm 6 to compute this sum.

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

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

using the following algorithm `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 8. *Error-free transformation of the product of three floating point numbers.*

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

For the three floating point numbers $a, b, c \in \mathbf{F}$, `ThreeProduct` transforms the product abc into the unevaluated sum $x + y + z + t$ of four floating point numbers. Indeed, we have

$$a \cdot b \cdot c = (p + e) \cdot c = p \cdot c + e \cdot c = x + y + z + t.$$

This can be used to compute for example the area of a planar triangle or the volume of a tetrahedron as shown in (Nievergelt, 2004).

The following algorithm computes the determinant of a matrix up to a given 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 with `VecSum` and an associated error bound. If this error bound is less than the desired relative error ε then we stop. Otherwise, we continue the computation since `VecSum` improves the accuracy of the computed sum.

ALGORITHM 9. *Algorithm to compute the determinant up to a relative error ε.*

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

5.2. Robust geometric predicates

An application requiring guaranteed accuracy is the computation of geometric predicates. Some algorithms like Delaunay triangulation and mesh generation need self-consistent geometric tests.

Shewchuk (Shewchuk, 1997) gives a survey of the issues involved and presents an adaptive (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 (Shewchuk, 1997; Avnaim et al., 1997; Brönnimann and Yvinec, 2000; Krishnan et al., 2001; Demmel and Hida, 2004). Consider for example the predicate ORIENT3D which determines whether a point D is to the left or right of the oriented plane defined by the points A, B and C. The result of the predicate depends on the sign of the determinant

$$\begin{aligned}\textsc{Orient3D}(a,b,c,d) &= \operatorname{sign}\begin{pmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{pmatrix} \qquad (6)\\ &= \operatorname{sign}\begin{pmatrix} a_x-d_x & a_y-d_y & a_z-d_z \\ b_x-d_x & b_y-d_y & b_z-d_z \\ c_x-d_x & c_y-d_y & c_z-d_z \end{pmatrix}. \qquad (7)\end{aligned}$$

The computed result is of the same sign that the exact result if the relative error is less than 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 (Demmel and Hida, 2004) provide another method to certify the sign of a small determinant. They used an accurate summation algorithm using large accumulators. Here, we use the same kind of techniques as in the previous section. First, we compute the determinant by using equation (7). We compute a rigorous error bound for the error. If the relative error is less than one we obtained the sign of the determinant.

Otherwise, the determinant is expanded into a sum of 96 floating point numbers: indeed, the determinant can be expressed as a sum of 24 monomials of the form $\pm a_i \cdot b_j \cdot c_k$, and each of these monomials is here transformed into a sum of 4 floating point numbers with `ThreeProduct` (in our implementation, we use a slightly optimized version of this process avoiding some redundant computation). We can then apply summation Algorithm 7 until the relative error is less than one. This can be done using the error bound of Proposition 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 this vector (here $n = 94$).

In next Algorithm 10, the function `det` computes the determinant using equation (7). The function `ErrDet` computes a rigorous error bound for the computation of `det` using similar techniques than ones of Proposition 4.

ALGORITHM 10. *Algorithm to compute the predicate* ORIENT3D*(a,b,c,d).*

```
function sign = Orient3D($A$)
  res = det($A$)
  err = ErrDet($A$)
  if (err $\le$ |res|)
    return sign = sign(res)
  p = DetVector($A$)
```

```
*repeat*
    $p = \texttt{VecSum}(p)$
    $\mathrm{res} = p_n$
    $\beta = (2n\texttt{eps}/(1-2n\texttt{eps}))\cdot\left(\sum_{i=1}^{n-1}|p_i|\right)$
    $\mathrm{err} = \texttt{eps}|\mathrm{res}| + (\beta + (2\texttt{eps}^2|\mathrm{res}|))$
*until* $(\mathrm{err} \le |\mathrm{res}|)$
$\texttt{sign} = \mathrm{sign}(\mathrm{res})$
```

5.3. Condition number for determinants

It is classic in perturbation analysis to introduce a *condition number* that measures the sensitivity of the solution to perturbation in the data. It gives informations about the difficulty to solve the problem accurately.

In this subsection, we define a *normwise condition number* for the problem of computing the determinant of a given matrix by

$$\kappa(A) := \lim_{\varepsilon\to 0}\sup\left\{\frac{|\det(A+H)-\det(A)|}{\varepsilon|\det(A)|} : \|H\|_F \le \varepsilon\|A\|_F\right\}.$$

We use the Frobenius norm defined by

$$\|A\|_F = \left(\sum_{i=1}^{n}\sum_{j=1}^{n}|a_{i,j}|^2\right)^{1/2} = \mathrm{trace}(A^*A)^{1/2}.$$

The choice of this norm rather than the 2-norm is motivated by the fact that we can give a computable formula for the condition number.

THEOREM 2. *Let $A \in \mathbb{R}^{n\times n}$ a real matrix. Then*

$$\kappa(A) = \frac{\|A\|_F\|\operatorname{adj}(A)\|_F}{|\det(A)|},$$

where $\operatorname{adj}(A) := ((-1)^{i+j}\det(A_{ij}))$ *is the adjugate matrix with A_{ij} denoting the submatrix of A obtained by deleting row i and column j.*

Proof. The function $\det : \mathbb{R}^{n\times n} \to \mathbb{R}, A \mapsto \det(A)$ is Fréchet differentiable since it is a polynomial of the coefficients of the matrix. Let us denote (E_{ij}) the canonical basis of $\mathbb{R}^{n\times n}$. Let $A = (a_{ij})_{1\le i,j\le n}$ a matrix. We want to compute $\frac{\partial\det}{\partial a_{ij}}(A)$. Let $C \in \mathbb{R}^{n\times n}$ be the adjugate matrix of A. The n-linearity of the determinant implies that, for all (i,j) and all $t \in \mathbb{R}$,

$$\det(A + tE_{ij}) = \det(A) + tC_{ij}$$

and so

$$\frac{\partial\det}{\partial a_{ij}}(A) = \lim_{t\to 0}\frac{\det(A+tE_{ij})-\det(A)}{t} = C_{ij}.$$

Hence, if $H = (h_{ij}) \in \mathbb{R}^{n\times n}$, we have

$$D\det(A)(H) = \sum_{i,j} h_{ij}\frac{\partial \det}{\partial a_{ij}}(A) = \sum_{i,j} h_{ij}C_{ij} = \operatorname{trace}(C^T H).$$

Here D denotes the differential operator. It follows from the definition of the condition number that

$$\kappa(A) = \frac{\|A\|_F\|D\det(A)\|}{|\det(A)|}.$$

By definition

$$\|D\det(A)\| = \sup_{\|M\|_F=1} |D\det(A)(M)| = \sup_{\|M\|_F=1} |\operatorname{trace}(C^T M)|.$$

Since $\langle A, B\rangle = \operatorname{trace}(A^T B)$ is a scalar product on $\mathbb{R}^{n\times n}$, it follows easily from the Cauchy-Schwarz inequality that

$$\|D\det(A)\| = \|C\|_F.$$

As a consequence, $\kappa(A) = \|A\|_F\|\operatorname{adj}(A)\|_F/|\det(A)|$. If A is invertible, we have $C^T = \det(A)A^{-1}$ and so $\kappa(A) = \|A\|_F\|A^{-1}\|_F$.

We can also define a *componentwise condition number* for the determinant by

$$\operatorname{cond}(A) := \lim_{\varepsilon\to 0}\sup\left\{\frac{|\det(A+H)-\det(A)|}{\varepsilon|\det(A)|} : |H| \le \varepsilon|A|\right\},$$

where absolute value and comparison have to be understood componentwise. A standard computation yields

$$\operatorname{cond}(A) = \frac{n\operatorname{per}(|A|)}{\det(A)},$$

where $\operatorname{per}(A)$ is the permanent of the matrix A defined by

$$\operatorname{per}(A) = \sum_{\sigma\in\mathfrak{S}_n}\prod a_{i,\sigma(i)}.$$

This condition number makes it possible to appreciate the difficulty of computing a determinant. We will use it in the next section to show the efficiency of our algorithm for ill-conditioned determinants.

One can find another condition number for determinant in (Higham, 1996, Pb 13.15).

6. Experimental results

We compare the following three algorithms.

1. Approximate. Straightforward double precision implementation which is not robust.

Table I. Comparison of the 3 algorithms. The ratio is the running time ratio compared to the ADAPTIVE algorithm. All times are measured in μs.

Algorithm	Points			
	Random		Nearly coplanar	
	Time	Ratio	Time	Ratio
APPROXIMATE	0.023	0.34	0.023	0.0042
ADAPTIVE	0.066	1	5.4	1
ORIENT3DEXACT	0.066	1	2.4	0.44

2. ADAPTIVE. The adaptive multiprecision arithmetic of Shewchuk (Shewchuk, 1997). It works by successively getting a better estimate and stopping when accuracy is guaranteed. It is considered as the current state-of-the-art algorithm.
3. ORIENT3DEXACT. Our algorithm which is adaptive as well.

Table I shows the performance of the three algorithms. The ratio is the running time ratio compared to the ADAPTIVE algorithm. All the timing were done in C on a Intel Pentium IV 3 Ghz using GNU C compiler (`gcc-3.4.1`). Every timing is a mean of 1000 tests. For random points, the probability to be nearly coplanar is very small. Thus, the three algorithms share the same timing. Indeed, the three algorithms use the same method, that is computing with (7), to compute the determinant. Our algorithm ORIENT3DEXACT and Shewchuk's algorithm ADAPTIVE compute also a valid error bound which explain that there is a different timing for these two algorithms.

For nearly coplanar set of points, our algorithm is over twice as fast as Shewchuk's algorithm ADAPTIVE. This may be explained by the fact that Shewchuk needs renormalisation of the expansion which is not the case for our algorithm. For our choice of nearly coplanar points, the componentwise condition number varies from about 10^{10} to 10^{50}. The nearly coplanar points represent a very small part of the input in practical applications. Nevertheless, they generate the most time-consuming cases. This is why it is important to be able to deal with such ill-conditioned cases in an efficient way.

7. Conclusion

We provide a fast and robust algorithm to compute the ORIENT3D predicate. Our algorithm is as fast as Shewchuk's algorithm for random sample of points and is twice as fast as Shewchuk's algorithm for nearly coplanar set of points which are the difficult cases to compute. We have tested only the ORIENT3D geometric predicate. Of course, the same techniques can be easily applied to the other predicates like INCIRCLE and INSPHERE for example (see (Shewchuk, 1997)). 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

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

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

Bailey, D. H.: 2001, 'A Fortran-90 double-double library'. Available at URL = `http://crd.lbl.gov/~dhbailey/mpdist/index.html`.

Brent, R. P.: 1978, 'A Fortran Multiple-Precision Arithmetic Package'. *ACM Trans. Math. Softw.* **4**(1), 57–70.

Brönnimann, H. and M. Yvinec: 1997, 'Efficient exact evaluation of signs of determinants'. In: *SCG '97: Proceedings of the thirteenth annual symposium on Computational geometry.* New York, NY, USA, pp. 166–173, ACM Press.

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

Clarkson, K. L.: 1992, '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* .

Daumas, M. and C. Finot: 1999, 'Division of floating point expansions with an application to the computation of a determinant.'. *J. UCS* **5**(6), 323–338.

Dekker, T. J.: 1971, 'A floating-point technique for extending the available precision'. *Numer. Math.* **18**, 224–242.

Demmel, J. and Y. Hida: 2004, 'Fast and accurate floating point summation with application to computational geometry'. *Numer. Algorithms* **37**(1-4), 101–112.

Hida, Y., X. S. Li, and D. H. Bailey: 2001, 'Algorithms for Quad-Double Precision Floating Point Arithmetic'. In: *Proc. 15th IEEE Symposium on Computer Arithmetic.* pp. 155–162, IEEE Computer Society Press, Los Alamitos, CA, USA.

Higham, N. J.: 1996, *Accuracy and stability of numerical algorithms.* Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM).

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

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

Knuth, D. E.: 1998, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms.* Reading, MA, USA: Addison-Wesley, third edition.

Krishnan, S., M. Foskey, T. Culver, J. Keyser, and D. Manocha: 2001, '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.* New York, NY, USA, pp. 274–283, ACM Press.

MPFR: 2005, 'MPFR, the Multiprecision Precision Floating Point Reliable library'. Available at URL = `http://www.mpfr.org`.

Nievergelt, Y.: 2004, 'Analysis and applications of Priest's distillation'. *ACM Trans. Math. Softw.* **30**(4), 402–433.

Ogita, T., S. M. Rump, and S. Oishi: 2005, 'Accurate Sum And Dot Product'. *SIAM J. Sci. Comput.* **26**(6), 1955–1988.

Priest, D. M.: 1992, 'On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations'. Ph.D. thesis, Mathematics Department, University of California, Berkeley, CA, USA. `ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z`.

Shewchuk, J. R.: 1997, 'Adaptive precision floating-point arithmetic and fast robust geometric predicates'. *Discrete Comput. Geom.* **18**(3), 305–363.