Japan J. Indust. Appl. Math., 26 (2009), 215–231 Area ⟨2⟩

Adaptive and Efficient Algorithm for 2D Orientation Problem

Katsuhisa OZAKI*, Takeshi OGITA†, Siegfried M. RUMP‡ and Shin'ichi OISHI§

* *Research Institute for Science and Engineering, Waseda University*
E-mail: k_ozaki@aoni.waseda.jp
† *Department of Mathematical Sciences, Tokyo Woman's Christian University*
‡ *Institute for Reliable Computing, Hamburg University of Technology*
§ *Department of Applied Mathematics*
Faculty of Science and Engineering, Waseda University

Received April 28, 2008
Revised November 17, 2008

This paper is concerned with a robust geometric predicate for the 2D orientation problem. Recently, a fast and accurate floating-point summation algorithm is investigated by Rump, Ogita and Oishi, which provably outputs a result faithfully rounded from the exact value of the summation of floating-point numbers. We optimize their algorithm for applying it to the 2D orientation problem which requires only a correct sign of a determinant of a 3×3 matrix. Numerical results illustrate that our algorithm works fairly faster than the state-of-the-art algorithm in various cases.

Key words: robust geometric predicate, 2D orientation problem, floating-point arithmetic, accurate algorithm

1. Introduction

In this paper, we are concerned with geometric predicates using floating-point arithmetic. In particular, we focus our mind on a 2D orientation problem "Orient2D" (cf. [8]), which determines whether a point c lies on, to the left of, or to the right of the oriented line defined by two points a and b. Let $\mathbb{F}$ be a set of floating-point numbers. Let $a = (a_x, a_y)$, $b = (b_x, b_y)$ and $c = (c_x, c_y)$. The answer of Orient2D can be predicated by the sign of the determinant

$$\mathrm{sign}(\det(G)), \quad G := \begin{pmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{pmatrix}. \tag{1.1}$$

Throughout the paper, we assume that $a, b, c \in \mathbb{F}^2$.

When calculating (1.1) by pure floating-point arithmetic, say, double precision defined by IEEE 754 standard [1] as working precision, then we sometimes obtain incorrect results due to the accumulation of roundoff errors. To avoid this, one may increase the working precision, e.g., quadruple or multiple precision. The computational speed is, however, significantly slower than that of double precision, because they are often realized by software simulation. Moreover, even if using such an approach, the result is still not guaranteed because the problem can be very ill-conditioned.

Shewchuk [8] has developed his adaptive algorithms for solving several computational geometry problems including Orient2D by his clever use of floating-point arithmetic, which aim to do only as much work as necessary to guarantee a correct result. He showed that his algorithms are in many cases much faster than the others based on traditional arbitrary precision libraries. Therefore, his algorithms can be viewed as the state-of-the-art algorithms. He also showed some applications utilizing his robust algorithms and their efficiencies [9].

Recently, the latter three of the authors (Rump, Ogita and Oishi) have investigated new accurate summation algorithms [6, 7]. Their algorithms are very fast in terms of measured computing time since the algorithms use only usual floating-point arithmetic (without using extra higher precision). Moreover, the algorithms guarantee the *result accuracy* so that it is suitable to apply them to the geometric predicates.

We propose in this paper an adaptive and efficient algorithm of solving Orient2D by adapting the Rump–Ogita–Oishi's summation algorithm to it. Results of numerical experiments are presented showing that our algorithm is fairly faster than Shewchuk's in many cases.

2. Transformation of determinant

In this section, we review an *error-free transformation* of the determinant (1.1) into a sum of a vector by floating-point arithmetic without loss of information. The technique has been used by Shewchuk [8], Demmel–Hida [3] and others.

Our notation is as follows. Let $\mathrm{fl}(\,\cdot\,)$ be a result of floating-point computations with the default rounding mode (rounding-to-nearest, tie to even). To simplify our discussions, we assume that double precision floating-point arithmetic defined by IEEE standard 754 [1] is used as working precision for all computations. Let $\mathbf{u}$ be the unit roundoff (especially, $\mathbf{u} = 2^{-53}$ in double precision). In this paper, we assume for simplicity that no overflow nor underflow occurs, although it is not so difficult theoretically to take such cases into account (see Remark 1). Then it holds for $\circ \in \{+, -, *, /\}$ that

$$\mathrm{fl}(a \circ b) = (a \circ b)(1 + \delta), \quad |\delta| \le \mathbf{u}. \tag{2.1}$$

We express algorithms in Matlab-style [10].

We first introduce so-called error-free transformations of floating-point arithmetic [5]. For $a, b \in \mathbb{F}$, there is a well-known algorithm due to Knuth [5] which transforms a sum $a + b$ into $x + y$ with $x, y \in \mathbb{F}$, $x = \mathrm{fl}(a + b)$ and $|y| \le \mathbf{u}|x|$:

Algorithm 1. *Error-free transformation of a sum of two floating-point numbers.*

---

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

Similarly, a subtraction of two floating-point numbers can be transformed without rounding errors as well. We denote the algorithm as $[x, y] = \texttt{TwoDiff}(a, b)$, which satisfies $a - b = x + y$ with $x, y \in \mathbb{F}$ and $|y| \leq \mathbf{u}|x|$.

For $a, b \in \mathbb{F}$, there is a well-known algorithm due to G.W. Veltkamp [2] which transforms the product $a \cdot b$ into $x + y$ with $x, y \in \mathbb{F}$, $x = \mathrm{fl}(a \cdot b)$ and $|y| \leq \mathbf{u}|x|$:

ALGORITHM 2. *Error-free transformation of a product of two floating-point numbers.*

---

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

It relies on the following splitting algorithm by Dekker [2] which splits a 53-bits floating-point number into two 26-bits parts:

ALGORITHM 3. *Error-free splitting of a floating-point number into two parts.*

---

function $[x, y] = \texttt{Split}(a)$
  $c = \mathrm{fl}(\texttt{factor} \cdot a)$ % $\texttt{factor} = 2^{27} + 1$
  $x = \mathrm{fl}(c - (c - a))$
  $y = \mathrm{fl}(a - x)$

Such error-free transformation algorithms are very useful for accurate computations by floating-point arithmetic. See [5, 6, 7] for detail.

We now present a way of transforming the determinant in (1.1) into a summation of 16 terms. From (1.1), we have

$$\det(G) = (a_x - c_x)(b_y - c_y) - (a_y - c_y)(b_x - c_x). \tag{2.2}$$

Then, we apply `TwoDiff` for each subtraction in (2.2) as follows:

$$\begin{cases} [t_1, e_1] = \texttt{TwoDiff}(a_x, c_x), & [t_2, e_2] = \texttt{TwoDiff}(b_y, c_y), \\ [t_3, e_3] = \texttt{TwoDiff}(a_y, c_y), & [t_4, e_4] = \texttt{TwoDiff}(b_x, c_x). \end{cases} \tag{2.3}$$

This implies

$$\begin{aligned} \det(G) &= (t_1 + e_1)(t_2 + e_2) - (t_3 + e_3)(t_4 + e_4) \\ &= t_1 t_2 + t_1 e_2 + t_2 e_1 + e_1 e_2 - t_3 t_4 - t_3 e_4 - t_4 e_3 - e_3 e_4. \end{aligned} \tag{2.4}$$

Next, we apply `TwoProduct` to each product in (2.4) as follows:

$$\begin{cases} [p_1, p_3] = \texttt{TwoProduct}(t_1, t_2), & [p_2, p_4] = \texttt{TwoProduct}(-t_3, t_4), \\ [p_5, p_9] = \texttt{TwoProduct}(t_1, e_2), & [p_6, p_{10}] = \texttt{TwoProduct}(t_2, e_1), \\ [p_7, p_{11}] = \texttt{TwoProduct}(-t_3, e_4), & [p_8, p_{12}] = \texttt{TwoProduct}(-t_4, e_3), \\ [p_{13}, p_{15}] = \texttt{TwoProduct}(e_1, e_2), & [p_{14}, p_{16}] = \texttt{TwoProduct}(-e_3, e_4). \end{cases} \tag{2.5}$$

The numbering of p_i is concerned with the distribution of floating-point numbers. Finally, we obtain a vector $p \in \mathbb{F}^{16}$ such that $\det(G) = \sum_{i=1}^{16} p_i$. When the problem is ill-conditioned, it is difficult to obtain the correct result by pure floating-point arithmetic. Thus, any robust summation algorithm [3, 5, 6, 7] can be used for calculating an accurate determinant.

Remark 1. If underflow occurs in `TwoProduct`, it holds that [5]

$$|a \cdot b - (x + y)| \leq 5\underline{\mathbf{u}},$$

where $\underline{\mathbf{u}}$ denotes the underflow unit ($\underline{\mathbf{u}} = 2^{-1074}$ in IEEE 754 double precision). Taking this into account for setting a stopping criterion in our algorithm in Section 4, our algorithm can also guarantee the sign of $\det(G)$, also in the presence of underflow, except the case where $\det(G) = 0$. If $\det(G) = 0$, G must be scaled to avoid underflow if necessary.

Remark 2. One may consider to transform the determinant as

$$\begin{aligned} \det(G) &= (a_x - c_x)(b_y - c_y) - (a_y - c_y)(b_x - c_x) \\ &= a_x b_y - a_x c_y - c_x b_y - a_y b_x + a_y c_x + c_y b_x \end{aligned} \tag{2.6}$$

and then apply `TwoProduct` to each product. In this case, we obtain a summation of only 12 terms, which is less than 16 terms in (2.4). However, it is difficult to derive an adaptive and efficient algorithm based on (2.6). We can not obtain much information about a distribution of floating-point numbers comparing to (4.6) in Section 4.

Remark 3. One may also consider to transform the determinant as

$$\det(G) = a_x(b_y - c_y) - b_x(a_y - c_y) + c_x(a_y - b_y).$$

Then a similar discussions in this paper can be done to develop an another adaptive algorithm, which may be more efficient than ours in the extremely ill-conditioned case. However, it is rare to see extremely ill-conditioned problem. Moreover, it is also difficult to investigate fast and adaptive algorithm corresponding to Method B proposed in this paper. Therefore, we do not treat it in this paper.

3. Accurate summation algorithm

Recently, a fast and accurate summation algorithm `AccSum` [6] using floating-point arithmetic has been developed by Rump, Ogita and Oishi. The main part of the algorithm is also useful for predicating the sign of a summation.

We briefly explain the main part of `AccSum`. Let p be a floating-point n-vector. Let $M_1 := \lceil \log_2(n+2) \rceil$ and $M_2 := \lceil \log_2 \max_i |p_i| \rceil$. The algorithm `AccSum` defines a constant σ_1 as

$$\sigma_1 = 2^{M_1+M_2}. \tag{3.1}$$

Then the following algorithm realizes an error-free transformation of a summation [6].

ALGORITHM 4 (Rump–Ogita–Oishi [6]). *For $p \in \mathbb{F}^n$,* `ExtractSum` *transforms $\sum_{i=1}^n p_i$ into $\tau + \sum_{i=1}^n p'_i$ without rounding errors, i.e., $\sum_{i=1}^n p_i = \tau + \sum_{i=1}^n p'_i$.*

---

function $[p', \tau] =$ `ExtractSum`(p, σ)
  $q = \mathrm{fl}((\sigma + p) - \sigma)$; % $q_i = \mathrm{fl}((\sigma + p_i) - \sigma)$
  $\tau = \mathrm{fl}\big(\sum_{i=1}^n q_i\big)$; % $\tau = \mathrm{fl}\big(\sum_{i=1}^n q_i\big) = \sum_{i=1}^n q_i$
  $p' = \mathrm{fl}(p - q)$; % $p'_i = \mathrm{fl}(p_i - q_i) = p_i - q_i$

In the first loop of `AccSum`, $[p^{(1)}, \tau_1] =$ `ExtractSum`(p, σ_1) is executed. If a stopping criterion[1] for $t_1 := \tau_1$ is satisfied, then the algorithm finishes after some minor computations. Otherwise, we continue by setting σ_2 as

$$\sigma_2 = \sigma_1 \cdot \phi, \tag{3.2}$$

where $\phi := 2^{M_1}\mathbf{u}$, and execute $[p^{(2)}, \tau_2] =$ `ExtractSum`$(p^{(1)}, \sigma_2)$. Again, we check the stopping criterion for $t_2 := t_1 + \tau_2$. Generalizing it, σ_k and t_k are defined as

$$\sigma_k = \sigma_{k-1} \cdot \phi, \quad t_k := t_{k-1} + \tau_k, \quad k \geq 2.$$

We repeat such a procedure with σ_k until the stopping criterion for $t^{(k)}$ is satisfied, i.e., the required accuracy is guaranteed. Fig. 3.1 illustrates the behavior of this algorithm. In Algorithm 4, it is proved that there is no rounding error in $\tau = \mathrm{fl}\big(\sum_{i=1}^n q_i\big)$. Therefore, we can obtain an accurate result even if problems are ill-conditioned. See [6, 7] for details.

In the next section, we will develop a new method of predicating the sign of the determinant in (1.1) based on the algorithm `AccSum`.

[1] The stopping criterion is different for the purpose. See [6, 7].

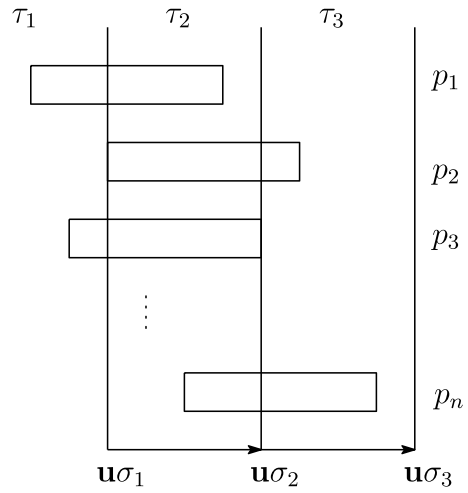

Fig. 3.1. Behavior of `AccSum`.

4. New adaptive method for sign determination

As mentioned in Section 2, the determinant $\det(G)$ in (1.1) can be transformed into the summation of the vector $p \in \mathbb{F}^{16}$. When using robust summation algorithms which guarantee the result accuracy (for example, `AccSum` [6] or `AccSign` [7]), we can obtain the correct sign of $\det(G)$. However, it is not efficient to apply such a robust algorithm regardless of the difficulty of the problems, because the sign of $\det(G)$ can frequently be guaranteed by a pure floating-point arithmetic for (2.2) and its a priori error analysis (see Subsection 4.1). Therefore, it is favorable that an algorithm adaptively works as much as necessary to guarantee the sign of the determinant according to the difficulty of the problem. This type of algorithm is called an *adaptive algorithm*.

For this purpose, we will prepare four verification methods for the sign of the determinant, i.e., Methods A, B, C and D. Combining them, we develop an adaptive algorithm which guarantees the sign of the determinant by testing Methods A, B, C and D in order. The proposed algorithm will be designed to work as follows: When a problem is well-conditioned, Method A can verify the sign of the determinant with less costs compared to more robust methods. Otherwise, we proceed to Method B. If Method B cannot verify it either, we next proceed to Method C, and so forth. Method D is a fallback algorithm. For example, even if a problem is so ill-conditioned that Methods A, B nor C cannot verify the sign of the determinant, Method D can definitely, which means Method D is never-failing since Method D keeps working until the correct sign of the determinant is obtained. Namely, the computational cost of the proposed adaptive algorithm depends on the difficulty of the problems.

Note that we do not generate all the 16 elements of p in advance but only necessary elements of p in each verification method.

4.1. Method A

First, we introduce the fastest known verification method which is the first one in Shewchuk's adaptive algorithm. Our adaptive algorithm also adopts it as the first one and call it "Method A" in this paper. Method A computes (2.2) by pure floating-point arithmetic

$$f_{\mathrm{A}} := \mathrm{fl}((a_x - c_x)(b_y - c_y) - (a_y - c_y)(b_x - c_x)). \tag{4.1}$$

Let e_{A} be defined by

$$e_{\mathrm{A}} := \mathrm{fl}((|(a_x - c_x)(b_y - c_y)| + |(a_y - c_y)(b_x - c_x)|). \tag{4.2}$$

If an inequality

$$|f_{\mathrm{A}}| > \mathrm{fl}((3\mathbf{u} + 16\mathbf{u}^2)e_{\mathrm{A}}) =: err_{\mathrm{A}} \tag{4.3}$$

is satisfied, then $\mathrm{sign}(f_{\mathrm{A}}) = \mathrm{sign}(\det(G))$. The right-hand side of (4.3) is an a priori error bound of f_{A}. Computing (4.1) requires 7 flops.[2] To calculate e_{A}, we can reuse the intermediate results in the computation of (4.1). If taking the absolute value of a floating-point number is counted as 1 flop, then Method A requires 12 flops and 1 branch. If (4.3) is not satisfied, then we proceed to the next method.

4.2. Method B

Next, we develop our Method B which is almost as robust as Shewchuk's second method [8, Figure 21] but works faster.

In Method B, we treat only the 4 elements $p_{1:4}$ to speed up the method. Of course, it is possible to apply and repeat Algorithm 4 (`ExtractSum`) for $p_{1:4}$ straightforwardly. However, to speed up further, we remark on the data structure among the 16 elements of p.

When we apply $[x, y] = \mathtt{TwoProduct}(a, b)$ or $[x, y] = \mathtt{TwoDiff}(a, b)$ for $a, b \in \mathbb{F}$, it holds that

$$|y| \le \mathbf{u}|x|. \tag{4.4}$$

From (2.3) and (2.5), we obtain

$$|e_1| \le \mathbf{u}|t_1|, \quad |e_2| \le \mathbf{u}|t_2|, \quad |p_9| \le \mathbf{u}|\mathrm{fl}(t_1 e_2)|, \quad |p_{15}| \le \mathbf{u}|\mathrm{fl}(e_1 e_2)|. \tag{4.5}$$

From the fact $p_1 = \mathrm{fl}(t_1 t_2)$, (4.5) and the monotonicity of floating-point arithmetic, it holds that

$$|p_9| \le \mathbf{u}^2|p_1|, \quad |p_{15}| \le \mathbf{u}^3|p_1|.$$

[2]Flops denotes the number of floating-point operations [11]. Please remark that it does not mean "floating-point operations per seconds."

Since the similar discussions can be applied for the other elements of p, we have the following inequalities:

$$\begin{cases} \max\limits_{3\le i\le 8}|p_i| \le \mathbf{u}\cdot\max(|p_1|,|p_2|), \\ \max\limits_{9\le i\le 14}|p_i| \le \mathbf{u}^2\cdot\max(|p_1|,|p_2|), \\ \max(|p_{15}|,|p_{16}|) \le \mathbf{u}^3\cdot\max(|p_1|,|p_2|). \end{cases} \tag{4.6}$$

Moreover, as in (2.5), we obtain the partial vector $p_{1:4}$ as

$$[p_1,p_3] = \texttt{TwoProduct}(t_1,t_2), \quad [p_2,p_4] = \texttt{TwoProduct}(-t_3,t_4). \tag{4.7}$$

From (4.4) and (4.7), we have

$$|p_3| \le \mathbf{u}|p_1|, \quad |p_4| \le \mathbf{u}|p_2|. \tag{4.8}$$

Therefore, it can be seen that there are big differences in the order of magnitude among p_i, $1 \le i \le 16$.

We show that we can omit to apply `ExtractSum` to p_1 and p_2. This is due to the following theorem by Sterbenz [4] for the subtraction of two floating-point numbers of the same sign:

Theorem 4.1 (Sterbenz [4]). *For $x, y \in \mathbb{F}$ $(x, y \neq 0)$, assume that $1/2 \le x/y \le 2$. Then $\mathrm{fl}(x-y) = x-y$.*

From (2.3), (2.5) and (4.1), we can see that $f_{\mathrm{A}} = \mathrm{fl}(p_1+p_2)$. Suppose Method A failed, which means the criterion (4.3) is not satisfied. Then $|f_{\mathrm{A}}| \le \mathrm{fl}((3\mathbf{u}+16\mathbf{u}^2)e_{\mathrm{A}})$. A little computation yields $1/2 \le p_2/p_1 \le 2$, so that Theorem 4.1 implies

$$f_{\mathrm{A}} = \mathrm{fl}(p_1+p_2) = p_1+p_2 \quad \text{(if Method A failed)}.$$

From this, we have

$$\sum_{i=1}^{16} p_i = f_{\mathrm{A}} + \sum_{i=3}^{16} p_i. \tag{4.9}$$

Now, we apply `ExtractSum` to the right-hand side of (4.9). From (4.1) and (4.6), it holds that

$$\max\Big(|f_{\mathrm{A}}|, \max_{3\le i\le 16}(|p_i|)\Big) \le err_{\mathrm{A}}.$$

Considering the data structure of f_{A} and p_i, $3 \le i \le 16$ from (2.3) and (4.6), we can see that it is sufficient to regard $n = 12$ for setting σ_k, $k \ge 1$. Therefore, we can safely set $M_1 = \lceil \log_2(12+2) \rceil = 4$, $M_2 = \lceil \log_2 err_{\mathrm{A}} \rceil$ and $\sigma_1 = 2^{M_1+M_2}$. By using this σ_1, we apply $[p'_{3:4}, \alpha_0] = \texttt{ExtractSum}(p_{3:4}, \sigma_1)$. Let $\alpha_1 := \mathrm{fl}(\alpha_0 + f_{\mathrm{A}})$, then $\alpha_1 = \alpha_0 + f_{\mathrm{A}}$ (see Fig. 4.1).

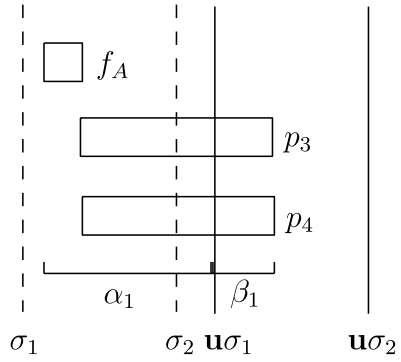

Fig. 4.1. The relation among f_{A}, p_3 and p_4.

Next we construct the criterion of Method B. Here, Method B transforms $\sum_{i=1}^{16} p_i$ as

$$\sum_{i=1}^{16} p_i = \alpha_1 + p_3' + p_4' + \sum_{i=5}^{16} p_i.$$

Let $g := p_3' + p_4' + \sum_{i=5}^{16} p_i$. If $|\alpha_1| > |g|$ is satisfied, then $\mathrm{sign}(\alpha_1) = \mathrm{sign}(\det(G))$. Let $e' = (1+\mathbf{u})e_{\mathrm{A}}$. From (4.2) and (2.5), it holds that

$$\sum_{i=5}^{8}|p_i| \le 2\mathbf{u}e', \quad \sum_{i=9}^{14}|p_i| \le 3\mathbf{u}^2e', \quad \sum_{i=15}^{16}|p_i| \le \mathbf{u}^3e'. \tag{4.10}$$

From [6, Lemma 3.2], it holds that

$$|p_i'| \le \mathbf{u}\sigma_1 \quad \text{for } i = 3, 4, \tag{4.11}$$

which implies $|p_3'| + |p_4'| \le 2\mathbf{u}\sigma_1$. From this and (4.10), we have

$$\begin{aligned} |g| &\le (2\mathbf{u} + 3\mathbf{u}^2 + \mathbf{u}^3)e' + 2\mathbf{u}\sigma_1 = (2\mathbf{u} + 3\mathbf{u}^2 + \mathbf{u}^3)(1+\mathbf{u})e_{\mathrm{A}} + 2\mathbf{u}\sigma_1 \\ &\le (2\mathbf{u} + 5\mathbf{u}^2 + 4\mathbf{u}^3 + \mathbf{u}^4)(e_{\mathrm{A}} + \sigma_1). \end{aligned} \tag{4.12}$$

Considering an a priori error bound of floating-point computations for (4.12), we have

$$\begin{aligned} |g| &\le (2\mathbf{u} + 5\mathbf{u}^2 + 4\mathbf{u}^3 + \mathbf{u}^2)(1+\mathbf{u}) \cdot \mathrm{fl}(e_{\mathrm{A}} + \sigma_1) \\ &= (1+\mathbf{u})^{-1}(1+\mathbf{u})^2(2\mathbf{u} + 5\mathbf{u}^2 + 4\mathbf{u}^3 + \mathbf{u}^4) \cdot \mathrm{fl}(e_{\mathrm{A}} + \sigma_1) \\ &= (1+\mathbf{u})^{-1}(2\mathbf{u} + 9\mathbf{u}^2 + 16\mathbf{u}^3 + 14\mathbf{u}^4 + 6\mathbf{u}^5 + \mathbf{u}^6) \cdot \mathrm{fl}(e_{\mathrm{A}} + \sigma_1) \\ &< (1+\mathbf{u})^{-1}2\mathbf{u}(1+6\mathbf{u}) \cdot \mathrm{fl}(e_{\mathrm{A}} + \sigma_1). \end{aligned}$$

Note that $2\mathbf{u}(1+6\mathbf{u}) = \mathrm{fl}(2\mathbf{u}(1+6\mathbf{u}))$. Therefore, we finally have

$$|g| \le \mathrm{fl}((2\mathbf{u}+12\mathbf{u}^2)(e_{\mathrm{A}}+\sigma_1)) =: err_{\mathrm{B}}. \tag{4.13}$$

Note that we can also bound σ_1 as $|\sigma_1| \le 64\mathbf{u}e_{\mathrm{A}}$ by a little computation but does not use this bound to avoid the overestimation. Thus, the criterion for verification of the sign of the determinant becomes

$$|\alpha_1| > err_{\mathrm{B}}. \tag{4.14}$$

This criterion is almost the same as that of Shewchuk's second method. Summarizing the above-mentioned discussions, we here present our Method B.

Algorithm 5 (Proposed Method B). *For* $f_{\mathrm{A}}, p_3, p_4 \in \mathbb{F}$, `Method_B` *transforms* $f_{\mathrm{A}}+p_3+p_4$ *into* $\alpha_1+p_3'+p_4'$ *without rounding errors. If* $flag = 1$, *then* $\mathrm{sign}(\alpha_1) = \mathrm{sign}(G)$. *Otherwise Method B failed to verify the sign of* $\det(G)$.

---

function $[p'_{3:4}, \alpha_1, flag] =$ `Method_B`$(f_{\mathrm{A}}, p_{3:4})$
  $flag = 0$;
  $q = \mathrm{fl}\,(\sigma_1 + p_{3:4}) - \sigma_1$; % `ExtractSum`$(p_{3:4}, \sigma_1)$
  $p'_{3:4} = \mathrm{fl}(p_{3:4} - q)$;
  $\alpha_1 = \mathrm{fl}((q_1+q_2)+f_{\mathrm{A}})$; % $\alpha_1 = q_1+q_2+f_{\mathrm{A}}$
  if $|\alpha_1| > err_{\mathrm{B}}$ % err_{B} *is defined in* (4.13)
    $flag = 1$;
  end

Here, we remark on the flops count needed for Method B. To use `ExtractSum` for $p_{1:4}$ straightforwardly, it has to be used 3 times from the relation among $p_{1:4}$ (see Fig. 4.1), which requires 48 flops. Our Method B requires only 8 flops and its stopping criterion is almost the same as that of Shewchuk's Method B. Moreover, Shewchuk's algorithm uses `TwoSum` 4 times in his Method B and it requires 24 flops. Therefore, it is expected that our Method B is almost as robust as Shewchuk's Method B, and works faster.

4.3. Method C

At the beginning of Method C, we discuss treatments of p_3' and p_4' after Method B. Once p_3 and p_4 are split by the extraction in Method B, the parts p_3' and p_4' need no splitting again (see Fig. 4.1). Therefore, we can omit to apply `ExtractSum` to p_3' and p_4'. Let $\beta_1 := \mathrm{fl}(p_3'+p_4')$, then $\beta_1 = p_3'+p_4'$ since

$$p_3', p_4' \in \mathbf{u}\sigma_2\mathbb{Z} \quad \text{and} \quad |p_3'| + |p_4'| < \sigma_2.$$

Here, it holds that

$$\sum_{i=1}^{16} p_i = \alpha_1 + \beta_1 + \sum_{i=5}^{16} p_i. \tag{4.15}$$

In Method C, we generate $p_{5:8}$ by using `TwoDiff` in (2.3). Let $d := \alpha_1 + \beta_1 + \sum_{i=5}^{8} p_i$, then $d = \sum_{i=1}^{8} p_i$. We compute a floating-point approximation of d by

$$\tilde{d} := \mathrm{fl}(((\alpha_1 + \beta_1) + (p_5 + p_6)) + (p_7 + p_8))$$

and consider its error bound. A standard error analysis for floating-point arithmetic [4] yields

$$\tilde{d} = [\{(\alpha_1 + \beta_1)(1 + \delta_1) + (p_5 + p_6)(1 + \delta_2)\}(1 + \delta_4) + (p_7 + p_8)(1 + \delta_3)](1 + \delta_5),$$

where $|\delta_i| \le \mathbf{u}$. From this, we can obtain an upper bound of $|d - \tilde{d}|$ as

$$|d - \tilde{d}| \le (3\mathbf{u} + 3\mathbf{u}^2 + \mathbf{u}^3)|\alpha_1 + \beta_1| + (3\mathbf{u} + 3\mathbf{u}^2 + \mathbf{u}^3)\sum_{i=5}^{8}|p_i|. \tag{4.16}$$

From (2.5), (4.6) and the definition of e_{A}, we have

$$\mathrm{fl}\left(\sum_{i=5}^{8}|p_i|\right) \le 2\mathbf{u}e_{\mathrm{A}}. \tag{4.17}$$

Inserting (4.17) into (4.16) yields

$$|d - \tilde{d}| \le (3\mathbf{u} + 3\mathbf{u}^2 + \mathbf{u}^3)|\alpha_1 + \beta_1| + 2\mathbf{u}(3\mathbf{u} + 3\mathbf{u}^2 + \mathbf{u}^3)e_{\mathrm{A}} =: f_1. \tag{4.18}$$

Here, we have to take $p_{9:16}$ into account for the verification. From (2.5) and the definition of e_{A}, we have

$$\sum_{i=9}^{16}|p_i| \le 3\mathbf{u}^2 e_{\mathrm{A}} + \mathbf{u}^3 e_{\mathrm{A}} =: f_2. \tag{4.19}$$

From (4.18) and (4.19), it holds that

$$\begin{aligned}\left|\sum_{i=1}^{16} p_i - \tilde{d}\right| &\le \left|\sum_{i=1}^{16} p_i - d\right| + |d - \tilde{d}| = \left|\sum_{i=9}^{16} p_i\right| + |d - \tilde{d}| \\ &\le f_1 + f_2 = c_1|\alpha_1 + \beta_1| + c_2 e_{\mathrm{A}},\end{aligned}$$

where $c_1 := 3\mathbf{u} + 3\mathbf{u}^2 + \mathbf{u}^3$ and $c_2 := 9\mathbf{u}^2 + 7\mathbf{u}^3 + \mathbf{u}^4$. To compute the error bound by floating-point arithmetic, we have

$$\begin{aligned}\left|\sum_{i=1}^{16} p_i - \tilde{d}\right| &\le (1 + \mathbf{u})c_1\,\mathrm{fl}(|\alpha_1 + \beta_1|) + c_2 e_{\mathrm{A}} \\ &= (1 + \mathbf{u})^{-2}\{(1 + \mathbf{u})^3 c_1\,\mathrm{fl}(|\alpha_1 + \beta_1|) + (1 + \mathbf{u}^2)c_2 e_{\mathrm{A}}\}.\end{aligned}$$

Here, a little computations yield

$$(1+\mathbf{u})^3 c_1 < \mathrm{fl}(3\mathbf{u}+14\mathbf{u}^2),$$
$$(1+\mathbf{u})^2 c_2 < \mathrm{fl}(9\mathbf{u}^2+24\mathbf{u}^3).$$

Finally, we have

$$\left|\sum_{i=1}^{16} p_i - \tilde{d}\right| \le \mathrm{fl}((3\mathbf{u}+14\mathbf{u}^2)|\alpha_1+\beta_1| + (9\mathbf{u}^2+24\mathbf{u}^3)e_{\mathrm{A}}) =: err_{\mathrm{C}}. \tag{4.20}$$

Thus, if $|\tilde{d}| > err_{\mathrm{C}}$ is satisfied, then $\mathrm{sign}(\tilde{d}) = \mathrm{sign}(\det(G))$ is guaranteed.

4.4. Method D

If neither Method A, B nor C can guarantee the sign of $\det(G)$, the fallback method, Method D, is used.

To reduce the computational cost as much as possible, we aim to construct Method D by generating the data of p adaptively. For the purpose, we consider the data structure of p again. Let $p^{(1)} := p$. After Method C, the elements $p^{(1)}_{1:4}$ have already been computed. From (4.6) and the definition of σ_1, the following equality holds by the same discussion in Subsection 4.2:

$$\mathrm{fl}\big(\big(\sigma_1 + p^{(1)}_{9:16}\big) - \sigma_1\big) = 0. \tag{4.21}$$

Therefore, we first apply `ExtractSum` with σ_1 to only $p^{(1)}_{5:8}$, i.e., we generate $p^{(1)}_{5:8}$ and execute

$$\big[p^{(2)}_{5:8}, \alpha_2\big] = \texttt{ExtractSum}\big(p^{(1)}_{5:8}, \sigma_1\big).$$

Let $t_1 := \tau_1 = \mathrm{fl}(\alpha_1+\alpha_2)$, where α_1 has been computed in Method B, then $t_1 = \alpha_1+\alpha_2$ since we use the same principle of `AccSum` in [6]. We proceed to the next `ExtractSum` with σ_2. Let $p^{(2)}_{9:16} := p^{(1)}_{9:16}$. Since the same discussion as above can be applied for σ_2, it holds that

$$\mathrm{fl}\big(\big(\sigma_2 + p^{(2)}_{15:16}\big) - \sigma_2\big) = 0. \tag{4.22}$$

Therefore, we generate only $p^{(2)}_{9:14}$ and apply `ExtractSum` to $p^{(2)}_{5:14}$ as

$$\big[p^{(3)}_{5:14}, \beta_2\big] = \texttt{ExtractSum}\big(p^{(2)}_{5:14}, \sigma_2\big).$$

Let $\tau_2 := \beta_1+\beta_2$, where β_1 has been computed in Method C. Let $t_2 := \mathrm{fl}(t_1+\tau_2)$, then $t_2 = t_1+\tau_2$. If the stopping criterion for t_2, which is mentioned later, is not satisfied, then we finally generate $p^{(3)}_{15:16}$ and apply `ExtractSum` iteratively to $p^{(k)}_{5:16}$ as

$$\big[p^{(k+1)}_{5:16}, \tau_k\big] = \texttt{ExtractSum}\big(p^{(k)}_{5:16}, \sigma_k\big), \quad k \ge 3$$

until the stopping criterion for $t_k := t_{k-1}+\tau_k$ is satisfied.

Here, we define the stopping criterion for Method D. From [6, Lemma 3.2], it holds that

$$\max_{5\le i\le 16}\left|p_i^{(k+1)}\right| \le \mathbf{u}\sigma_k, \quad k \ge 2. \tag{4.23}$$

Since $t_k = \sum_{i=1}^{k} \tau_i$, it holds that

$$\det(G) = \sum_{i=1}^{16} p_i = t_k + \sum_{i=5}^{16} p_i^{(k+1)}, \quad k \ge 2.$$

From (4.23), the following inequality holds:

$$\left|\sum_{i=5}^{16} p^{(k+1)}\right| \le \sum_{i=5}^{16} |p^{(k+1)}| \le 12\mathbf{u}\sigma_k = \mathrm{fl}(12\mathbf{u}\sigma_k), \quad k \ge 2.$$

Therefore, if $|t_k| > \mathrm{fl}(12\mathbf{u}\sigma_k)$ is satisfied, then $\det(G) = \det(t_k)$ is guaranteed.

Here, we remark on an acceleration of the algorithm. If $\alpha_1 + \beta_1 = 0$, then it holds from (4.15) that

$$\sum_{i=1}^{16} p_i = \sum_{i=5}^{16} p_i.$$

Therefore, we can reset σ_1 from $p_{5:8}$ as

$$\sigma_1' = 2^{M_1 + M_2'},$$

where $M_2' := \lceil \log_2 \max|p_{5:8}| \rceil$. Even if replacing σ_1 by σ_1', the discussions in this section is also true. Since there is a possibility that both $|p_3|$ and $|p_4|$ are much larger than $\max|p_{5:8}|$ (see Fig. 4.2), it may significantly speed up the method. The following algorithm can guarantee the sign of the determinant:

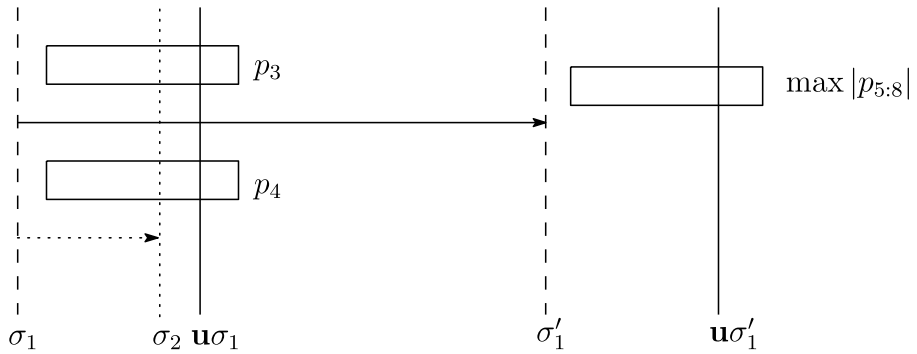

Fig. 4.2. The big difference in order of magnitude.

Algorithm 6 (Proposed Method D). *After obtaining* α_1 *and* β_1 *in Methods A, B and C, this method can definitely verify the sign of* $\det(G)$.

---

function $s = \texttt{Method\_D}(\alpha_1, \beta_1, p_{5:8})$
  $\left[p^{(2)}_{5:8}, \alpha_2\right] = \texttt{ExtractSum}(p_{5:8}, \sigma_1)$
  $t_1 = \mathrm{fl}(\alpha_1 + \alpha_2)$;
  (Generate $p_{9:14}$ *and set* $p^{(2)}_{9:14} := p_{9:14}$.*)*
  $\left[p^{(3)}_{5:14}, \beta_2\right] = \texttt{ExtractSum}\left(p^{(2)}_{5:14}, \sigma_2\right)$
  $\tau_2 = \mathrm{fl}(\beta_1 + \beta_2)$;
  $t_2 = \mathrm{fl}(t_1 + \tau_2)$;
  if $|t_2| > \mathrm{fl}(12\mathbf{u}\sigma_2)$
    $s = \texttt{sign}(t_2)$;
    return;
  end
  (Generate $p_{15:16}$ *and set* $p^{(3)}_{15:16} := p_{15:16}$.*)*
  $k = 2$;
  repeat
    $k = k + 1$;
    $\left[p^{(k+1)}_{5:16}, \tau_k\right] = \texttt{ExtractSum}\left(p^{(k)}_{5:16}, \sigma_k\right)$
    $t_k = \mathrm{fl}(t_{k-1} + \tau_k)$;
  until $|t_k| > \mathrm{fl}(12\mathbf{u}\sigma_k)$
  $s = \texttt{sign}(t_k)$;

4.5. Method D′: a special case

There is an alternative method which can be applied for a special case where the exponent range of p_i for $1 \le i \le 16$ is sufficiently narrow. Suppose $p_{13}, p_{14} \neq 0$. If

$$\mathbf{u}\sigma_4 < \frac{1}{2}\mathbf{u}^2 \min(|p_{13}|, |p_{14}|) = \frac{1}{2}\mathbf{u}^2 \min(\mathrm{fl}(|e_1 e_2|), \mathrm{fl}(|e_3 e_4|)) \tag{4.24}$$

is satisfied, then it can easily be proved that once splitting $p^{(k)}_i$ with σ_k, $p^{(j)}_i$ for $j \ge k+1$, no splittings with σ_j are needed again (see Fig. 4.3).

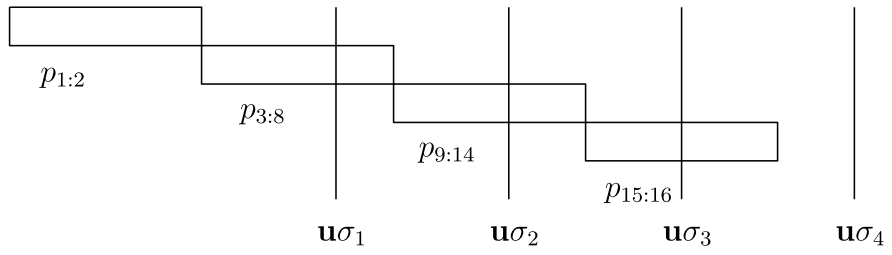

Fig. 4.3. Narrow distribution of $p_i, 1 \le i \le 16$.

Hence in this case, we can construct a more efficient algorithm than Method D. The following algorithm can be applied if the special condition (4.24) is satisfied.

ALGORITHM 7 (Proposed Method D′).

---

function $s = \texttt{Method\_D}'(\alpha_1, \beta_1, p_{5:8})$
$\left[p^{(2)}_{5:8}, \alpha_2\right] = \texttt{ExtractSum}\left(p_{5:8}, \sigma_1\right)$;
$t_1 = \mathrm{fl}(\alpha_1 + \alpha_2)$;
(Generate $p_{9:14}$ *and set* $p^{(2)}_{9:14} := p_{9:14}$*.)*
$\left[p^{(3)}_{9:14}, \beta_2'\right] = \texttt{ExtractSum}\left(p^{(2)}_{9:14}, \sigma_2\right)$
$\beta_2 = \mathrm{fl}\left(\beta_2' + \sum_{i=5}^{8} p_i^{(2)}\right)$;
$\tau_2 = \mathrm{fl}(\beta_1 + \beta_2)$;
$t_2 = \mathrm{fl}(t_1 + \tau_2)$;
if $|t_2| > \mathrm{fl}(12\mathbf{u}\sigma_2)$
$s = \texttt{sign}(t_2)$;
return;
end
(Generate $p_{15:16}$ *and set* $p^{(3)}_{15:16} := p_{15:16}$*.)*
$\left[p^{(4)}_{15:16}, \tau_3'\right] = \texttt{ExtractSum}\left(p^{(3)}_{15:16}, \sigma_3\right)$;
$\tau_3 = \mathrm{fl}\left(\left(\tau_3' + \sum_{i=9}^{14} p_i^{(3)}\right)\right)$;
$t_3 = \mathrm{fl}(t_2 + \tau_3)$;
$t_4 = \mathrm{fl}\left(t_3 + \left(p^{(4)}_{15} + p^{(4)}_{16}\right)\right)$;
$s = \texttt{sign}(t_4)$;

If we can use Method D′ instead of Method D, it can be expected that the verification of the sign of $\det(G)$ becomes fairly faster.

5. Numerical examples

We present some numerical results showing the effectiveness of our adaptive algorithm proposed so far. We use two different computer environments; One is a PC with Intel Pentium 4 (3.3 GHz) CPU and Intel C compiler (ICC) version 9.0 with compile options `-O3 -aW`. The other is a PC with AMD Athlon 64 (2.2 GHz) CPU and GNU C Compiler (GCC) version 4.1.1 with compile options `-O3 -march=athlon64 -funroll-loops`.

The following three methods are implemented on both computer environments:

- R1: `Orient2d_fast` by Shewchuk [8] (exact).
- A1: `Orient2d_adapt` by Shewchuk [8] (adaptive).
- A2: Our proposed adaptive algorithm (adaptive).

Here, "exact" means that the method adopts the most robust algorithm as the first step, which is not an adaptive algorithm, and "adaptive" means that their own Methods A, B, C and D are adaptively applied in order. Therefore, computing

times for both A1 and A2 strongly depend on the difficulty of problems. The problems can be grouped into the following four cases:

- P_{A}: Problems where Method A can verify the result.
- P_{B}: Problems where Method A fails to verify and Method B successes.
- P_{C}: Problems where Methods A and B fail to verify and Method C successes.
- P_{D}: Problems where Methods A, B and C fail to verify and Method D successes.

For each case, we generate a million sets of the coordinates a, b and c whose elements are pseudo-random (floating-point) numbers.

In Tables 5.1 and 5.2, we display the sum of the elapsed time (sec) for each case on the Pentium 4 and the Athlon 64 architectures, respectively.

Table 5.1. Computing time (sec) in various problems on Pentium 4 and ICC.

Algorithm	P_{A}	P_{B}	P_{C}	P_{D}
R1 (`Orient2d_fast`)	0.679	0.679	0.680	0.681
A1 (`Orient2d_adapt`)	0.033	0.253	0.291	1.050
A2 (our adaptive algorithm)	0.033	0.190	0.244	0.659

Table 5.2. Computing time (sec) in various problems on Athlon 64 and GCC.

Algorithm	P_{A}	P_{B}	P_{C}	P_{D}
R1 (`Orient2d_fast`)	0.308	0.336	0.296	0.316
A1 (`Orient2d_adapt`)	0.008	0.080	0.100	0.384
A2 (our adaptive algorithm)	0.008	0.052	0.080	0.280

From Tables 5.1 and 5.2, we can confirm that A1 is as fast as A2 for the problems P_{A} because both A1 and A2 finish at Method A by Shewchuk. For the problems P_{B}, A2 is faster than A1 due to the difference of flops counts. In A1 and A2, almost the same computations are done in their own Method C, so that the advantage of our Method B against Shewchuk's one is preserved. As a result, A2 becomes faster than A1 also for the problems P_{C}. For the ill-conditioned problems P_{D}, our method A2 works fastest, which means even faster than R1.

We can not guarantee that our method always works faster than the others; The computational speed of our method strongly depends on the property of the input data (the number of zero/nonzero elements, distribution of the data) and computer environment (CPU, compiler and its compile options), etc. Nevertheless, we can confirm that our proposed method works efficiently in many numerical examples on some computer environments.

Acknowledgments. This research was partially supported by CREST program, Japan Science and Technology Agency (JST).

References

[1] ANSI/IEEE, IEEE Standard for Binary Floating Point Arithmetic, Std 754–1985 edition. IEEE, New York, 1985.
[2] T.J. Dekker, A floating-point technique for extending the available precision. Numer. Math., **18** (1971), 224–242.
[3] J. Demmel and Y. Hida, Fast and accurate floating point summation with application to computational geometry. Numerical Algorithms, **37** (2004), 101–112.
[4] N.J. Higham, Accuracy and Stability of Numerical Algorithms, second edition. SIAM Publications, Philadelphia, 2002.
[5] T. Ogita, S.M. Rump and S. Oishi, Accurate sum and dot product. SIAM J. Sci. Comput., **26** (2005), 1955–1988.
[6] S.M. Rump, T. Ogita and S. Oishi, Accurate floating-point summation part I: Faithful rounding. SIAM J. Sci. Comput., **31** (2008), 189–224.
[7] S.M. Rump, T. Ogita and S. Oishi, Accurate floating-point summation part II: Sign, K-fold faithful and rounding to nearest. SIAM J. Sci. Comput., **31** (2008), 1269–1302.
[8] J.R. Shewchuk, Adaptive precision floating-point arithmetic and fast robust geometric predicates. Discrete & Computational Geometry, **18** (1997), 305–363.
[9] J.R. Shewchuk, Delaunay refinement algorithms for triangular mesh generation. Computational Geometry: Theory and Applications, **22** (2002), 21–74.
[10] MATLAB Programming version 7, the Mathworks.
[11] G.H. Golub and C.F. Van Loan, Matrix Computations, third edition. The Johns Hopkins University Press, 1996.