Exact 2D Convex Hull for Floating-point Data *

Katsuhisa Ozaki[1),3)], Takeshi Ogita[2),3)] and Shin'ichi Oishi[1),3)]
1) Faculty of Science and Engineering, Waseda University.
2) Department of Mathematical Sciences, Tokyo Woman's Christian University
3) JST, CREST.
k_ozaki@aoni.waseda.jp

Abstract. This paper is concerned with robustness problems in computational geometry, especially, a convex hull for a set of points on two-dimensional space. Floating point arithmetic is preferred to be used for calculating a convex hull in terms of computational performance. However, it may output a meaningless result due to accumulation of rounding errors. In this paper, it is discussed how to output the exact convex hull for floating-point data. Even if overflow or underflow in the floating-point computations is occurred, our method can output the exact convex hull. At the end of this paper, numerical examples are presented to show the efficiency of the proposed algorithm.

Keywords: Computational Geometry, Robust Algorithms

1. Introduction

In this paper, we discuss how to develop an algorithm which outputs the exact convex hull for a given set of points on two-dimensional space. We assume that all coordinates are expressed by floating-point numbers defined by IEEE 754-2008 (IEEE 754-2008, 2008). Let $\mathbb{F}$ be the set of floating-point numbers. To generate the convex hull, two-dimensional orientation problems must be solved many times in several algorithms, for example, Incremental Algorithm, Gift-Wrapping Algorithm and Graham's Algorithm and so on (see (O'Rourke, 2001)). Here is an explanation of the two-dimensional orientation problem: Suppose that there are an oriented line and a point on wo-dimensional space. The oriented line passes from $A(a_x, a_y)$ to $B(b_x, b_y)$. The point is denoted by $C(c_x, c_y)$ (see Fig. 1). The aim is to clarify which the point is left or right to the oriented line, or on the line. This problem can be boiled down to a sign of a 3-by-3 matrix determinant as follows:

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

If floating-point arithmetic is used to evaluate (1), then an incorrect sign may be obtained due to accumulation of rounding errors. It is especially worried that the point is very close to the oriented line. Once the incorrect sign is output, algorithms for computational geometry have prospects of outputting an inexact result. In such a case, taking a convex hull for a set of points for examples,

4th International Workshop on Reliable Engineering Computing (REC 2010)
Edited by Michael Beer, Rafi L. Muhanna and Robert L. Mullen
Copyright © 2010 Professional Activities Centre, National University of Singapore.
ISBN: 978-981-08-5118-7. *Published by* Research Publishing Services.
doi:10.3850/978-981-08-5118-7_028

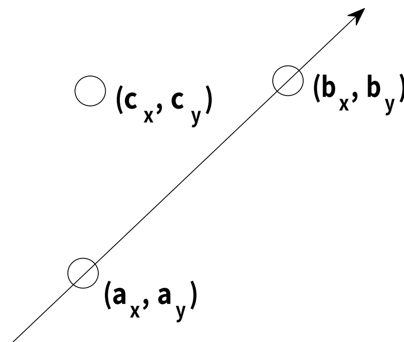

Figure 1. 2D orientation problem.

the result does not become a convex, or some points are not enclosed by the result. Plentiful topics of robustness problems in computational geometry are introduced in (Kettnear et al., 2008).

Many discussions for this robustness problem have been held. If we use multi-precision arithmetic, it may be possible to obtain the correct sign of the determinant (1). However, we cannot know in advance how the problem is ill-conditioned. Therefore, we can not determine how many mantissa bits the algorithm requires. There is a further problem. Although the problem is not ill-conditioned, multi-precision arithmetic takes much more computing time than pure floating-point arithmetic. On the contrary, it is rare for such ill-conditioned problems to appear in practice. Namely, many problems can be solved exactly by pure floating-point arithmetic. Essential points for these problems are as follows:

- Many problems are exactly solved by floating-point arithmetic, however, an incorrect result may be output.

- The costs for using multi-precision arithmetic are expensive. In addition, we can not know in advance how many bits the algorithm requires in order to output the correct result.

- In any cases, algorithms uses an incorrect sign, a result of geometric problems may be meaningless.

To overcome these problem, 'floating-point filter' (Shewchuk, 1997; Brönnimann et al., 2001; Pan and Yu, 1999) is first applied. The filter quickly checks whether a sufficient condition for the correctness of the sign of the determinant is satisfied or not. If the filter cannot guarantee the sign, we apply more robust algorithms which always output the correct sign with higher computational costs. The computing time depends on a difficulty of the problem.

Our aim is to obtain the exact convex hull as fast as possible. If the orientation problem is solved correctly as if we use rational arithmetic (symbolic computation), the algorithm always outputs the correct convex hull. Actually, algorithms in (Shewchuk, 1997; Brönnimann and Yvinec, 2000; Ozaki

et al., 2008) can guarantee the correct sign. These are so-called 'verification methods' for the two-dimensional orientation problem. By using these methods, the algorithms for the convex hull can always output the correct result. In this paper, we first remark that a sort of filters in (Shewchuk, 1997) can work even if overflow or underflow occurs. Next, we apply them for a convex hull for a set of points on two-dimensional spaces. Finally, we present numerical examples to illustrate the efficiency of the proposed method. We use MATLAB like notation (MATLAB, 2005) for readability.

2. Notation and Filter for Two-dimensional Orientation Problem

In this section, we introduce the filter for the 2D orientation problem. Let $fl(\cdots)$ denote that an expression in the parenthesis is evaluated by pure floating-point arithmetic. Let $\mathbf{u}$ [1], $\underline{\mathbf{u}}$ [2] and $\underline{\mathbf{u}}_N$ be the unit roundoff, the underflow unit and the smallest positive normalized floating-point number, respectively. The determinant (1) can be expanded as follows:

$$\begin{aligned} \det(G) &= (a_x - c_x)(b_y - c_y) - (b_x - c_x)(a_y - c_y) && (2)\\ &= a_x(b_y - c_y) + b_x(c_y - a_y) + c_x(a_y - b_y) && (3)\\ &= a_x b_y + a_y c_x + b_x c_y - a_x c_y - a_y b_x - b_y c_x && (4) \end{aligned}$$

We prefer to use (2) in terms of the number of operations. Let **res** be a result of floating-point computation of (2), i.e.

$$\mathbf{res} = fl((a_x - c_x)(b_y - c_y) - (b_x - c_x)(a_y - c_y)). \quad (5)$$

We present an algorithm computing (5):

ALGORITHM 1. *Let $A = (a_x, a_y)$, $B = (b_x, b_y)$ and $C = (c_x, c_y)$ on two-dimensional space. Suppose that an oriented line passes from A to B. The following algorithm outputs a sign of the determinant (1) by floating-point arithmetic. If the result is positive (negative), the point C is left (right) to the oriented line from A to B. However, the correctness of the result is not guaranteed.*

```
function res = AppOrient2D(a_x, a_y, b_x, b_y, c_x, c_y)
    l = (a_x - c_x)(b_y - c_y);
    r = (b_x - c_x)(a_y - c_y);
    res = l - r;
    return res
end
```

We introduce a filter for the form (5) in (Shewchuk, 1997). Let $\mathbf{err}_1$ be an a priori error bound of **res** such that

$$\mathbf{err}_1 = fl((3\mathbf{u} + 16\mathbf{u}^2)(|(a_x - c_x)(b_y - c_y)| + |(b_x - c_x)(a_y - c_y)|)).$$

If $|\mathbf{res}| > \mathbf{err}_1$ is satisfied, then $\mathrm{sign}(\det(G)) = \mathrm{sign}(\mathbf{res})$. We present an algorithm using this filter.

[1] For single precision, $\mathbf{u} = 2^{-24}$. For double precision, $\mathbf{u} = 2^{-53}$
[2] For single precision, $\underline{\mathbf{u}} = 2^{-149}$. For double precision, $\underline{\mathbf{u}} = 2^{-1074}$

Table I. Floating-point exceptions

x	$\circ$	y	z	x	$\circ$	y	z	x	$\circ$	y	z
Inf	+	Inf	Inf	-Inf	+	-Inf	-Inf	Inf	+	-Inf	NaN
Inf	-	Inf	NaN	-Inf	-	-Inf	NaN	Inf	-	-Inf	Inf
Inf	*	Inf	Inf	-Inf	*	-Inf	Inf	Inf	*	-Inf	-Inf
Inf	+	0	Inf	Inf	+	p	Inf	Inf	+	-p	Inf
Inf	-	0	-Inf	Inf	-	p	Inf	Inf	-	-p	Inf
Inf	*	0	NaN	Inf	*	p	Inf	Inf	*	-p	-Inf

ALGORITHM 2. *Let $A = (a_x, a_y)$, $B = (b_x, b_y)$ and $C = (c_x, c_y)$ on two-dimensional space. Suppose that an oriented line passes from A to B. If the result* **d** *is positive (negative), the point C is left (right) to the oriented line from A to B. If the result is zero, the problem is not solved by this filter.*

```
function = Filter1(a_x, a_y, b_x, b_y, c_x, c_y)
    l = (a_x − c_x)(b_y − c_y);
    r = (b_x − c_x)(a_y − c_y);
    d = l − r;
    err_1 = (3u + 16u^2)(|l| + |r|);
    if |d| > err_1
        return d;
    end
    d = 0;
    return d
end
```

We find Algorithm 2 works correctly, also in the presence of overflow.

THEOREM 1. *Even if overflow occurs in Algorithm 2, the algorithm outputs zero.*

proof
First, we introduce floating-point exceptions. We compute $z := x \circ y$ for $\circ \in \{+, -, *\}$. Let p be a positive floating-point number. Tabel I shows the result of $x \circ y$.

We assume that overflow occurs at least one of l, r, $l - r$ and $|l| + |r|$. From Table I, $\mathbf{d}$ becomes one of Inf, -Inf, NaN. Also the error bound $\mathbf{err}_1$ always becomes Inf or NaN. For the comparisons, the followings are satisfied:

$$\text{Inf} > \text{Inf} : \text{false}, \quad -\text{Inf} > \text{Inf} : \text{false}, \quad \text{NaN} > \text{Inf} : \text{false}$$
$$\text{Inf} > \text{NaN} : \text{false}, \quad -\text{Inf} > \text{NaN} : \text{false}, \quad \text{NaN} > \text{NaN} : \text{false}$$

Thus, the algorithm outputs zero in such cases. It indicates that robust algorithms are required to obtain the correct result. Therefore, even if overflow occurs in the evaluation in Algorithm 2,

Algorithm 2 outputs a suitable result. q.e.d

Next we discuss underflow in Algorithm 2. The computation involves two products, so that an a priori error bound becomes

$$\mathit{fl}((3\mathbf{u}+16\mathbf{u}^2)(|(a_x-c_x)(b_y-c_y)|+|(b_x-c_x)(a_y-c_y)|))+2\underline{\mathbf{u}}.$$

We need to compute the above error bound by floating-point arithmetic. After a little consideration, we have

$$\mathbf{err}_2 := \mathit{fl}((3\mathbf{u}+25\mathbf{u}^2)(|(a_x-c_x)(b_y-c_y)|+|(b_x-c_x)(a_y-c_y)|)+3\underline{\mathbf{u}}).$$

If $|\mathbf{res}| > \mathbf{err}_2$ is satisfied, then $\mathrm{sign}(\mathbf{res})$ is correct even if underflow occurs. The following is a filter which also works in the presence of overflow and underflow.

ALGORITHM 3. *Let $A = (a_x, a_y)$, $B = (b_x, b_y)$ and $C = (c_x, c_y)$ on two-dimensional space. Suppose that an oriented line passes from A to B. If the result $\mathbf{d}$ is positive (negative), the point C is left (right) to the oriented line from A to B even if underflow occurs. If the result is zero, the problem is not solved by this filter.*

```
function d = Filter2(a_x, a_y, b_x, b_y, c_x, c_y)
      l = (a_x − c_x)(b_y − c_y);
      r = (b_x − c_x)(a_y − c_y);
      d = l − r;
      err_2 = (3u + 25u^2)(|l| + |r|) + 3u̲;
      if |d| > err_2
         return d;
      end
      d = 0;
      return d;
end
```

There is a remark about Algorithm 3. The constant $\underline{\mathbf{u}}$ is a subnormal [3] floating-point number so that the evaluation in Algorithm 3 may work very slow. To overcome this problem, we use $\underline{\mathbf{u}}_N$ instead of $\underline{\mathbf{u}}$. By using $\underline{\mathbf{u}}_N$, $\mathbf{err}_2$ is bounded by

$$\mathbf{err}_2 \le \mathit{fl}((3\mathbf{u}+25\mathbf{u}^2)(|(a_x-c_x)(b_y-c_y)|+|(b_x-c_x)(a_y-c_y)|)+\underline{\mathbf{u}}_N) =: \mathbf{err}_3. \tag{6}$$

We present the following algorithm by using the error bound (6).

ALGORITHM 4. *Let $A = (a_x, a_y)$, $B = (b_x, b_y)$ and $C = (c_x, c_y)$ on two-dimensional space. Suppose that an oriented line passes from A to B. If the result $\mathbf{d}$ is positive (negative), the point C is left (right) to the oriented line from A to B. If the result is zero, the problem is not solved by*

[3] This is also called 'denormalized' floating-point number.

this filter.

```
function d = Filter3(a_x, a_y, b_x, b_y, c_x, c_y)
    l = (a_x − c_x)(b_y − c_y);
    r = (b_x − c_x)(a_y − c_y);
    d = l − r;
    err_3 = (3u + 25u^2)(|l| + |r|) + u_N;
    if |d| > err_3
        return d;
    end
    d = 0;
    return d;
end
```

REMARK 1. *Remark that Algorithm 4 cannot guarantee the sign of the determinant if* $\underline{\mathbf{u}} \lesssim |\mathbf{d}| \leq \underline{\mathbf{u}}_N$*, while Algorithm 3 may guarantee. Also, when algorithms treat Inf, -Inf or NaN, the performance of the computation may become very slow on several environments.*

We introduce an another filter in (Melquiond and Pion, 2007). The filter in (Melquiond and Pion, 2007) first checks possibility for overflow and underflow. It involves at least two branches. Next, the algorithm checks whether heavy cancellation occurs or not (see Figure 2 to understand the flow of the algorithm). The filter can work even if overflow or underflow occurs.

However, it is rare for overflow or underflow to occur in practice. The costs for checking the existence of overflow and underflow are not cheap. Moreover, even if underflow occurs in the filter, it may not be ill-conditional problem. If overflow or heavy cancellation occur, Algorithm 4 can judge them by only one criterion ($\mathbf{d} > \mathbf{err}_3$). We need not to implement the branches for overflow and underflow. In addition, even if underflow occurs and the problem is well-conditioned, Algorithm 4 outputs the correct result. Thus, these are some differences in the flow among algorithms [4].

When Algorithm 4 outputs zero, then we next apply more robust algorithms. The following is an algorithm outputting a correct result.

ALGORITHM 5. *Let* $A = (a_x, a_y)$*,* $B = (b_x, b_y)$ *and* $C = (c_x, c_y)$ *on two-dimensional space. Suppose that an oriented line passes from A to B. The following algorithm outputs the exact sign of the determinant (1).*

```
function d = Verified1(a_x, a_y, b_x, b_y, c_x, c_y)
    d = Filter3(a_x, a_y, b_x, b_y, c_x, c_y);
    if d ≠ 0
        return d;
    end
    d = Robust(a_x, a_y, b_x, b_y, c_x, c_y);
    return d;
end
```

[4] However, it is known that the algorithm in (Melquiond and Pion, 2007) also works fast as a filter. They showed its efficiency throughout some examples in 3D-Delaunay Triangulation by using 'In Sphere Problem'.

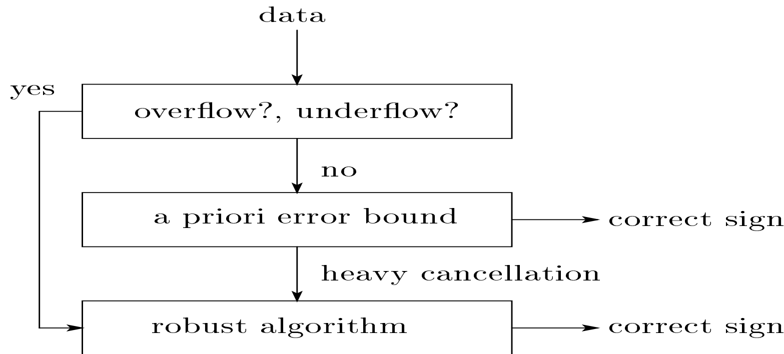

Figure 2. Flow of the algorithm in the paper by G. Melquiond and S. Pion.

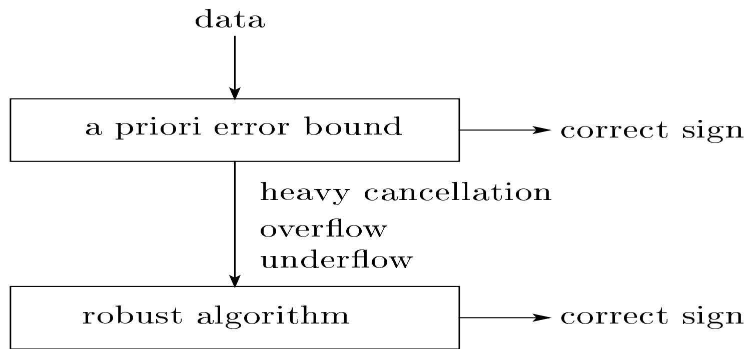

Figure 3. Flow of our algorithm.

In Algorithm 5, Robust$(a_x, a_y, b_x, b_y, c_x, c_y)$ returns the correct sign of the determinant (1) with relatively expensive computational costs. We can apply algorithms in (Shewchuk, 1997; Brönnimann and Yvinec, 2000; Ozaki et al., 2008) into Robust$(a_x, a_y, b_x, b_y, c_x, c_y)$.

3. Algorithm for Convex Hull and its Verification

In this section, we introduce Gift-Wrapping Algorithm which obtains a convex hull for n-points. When the number of vertices of the result is h, it is known that complexity of Gift-Wrapping Algorithm is $\mathcal{O}(nh)$. Suppose that n-points denote $P_i(x_i, y_i)$ for $1 \leq i \leq n$. The following is Gift-Wrapping Algorithm by using Algorithm 1.

ALGORITHM 6. *Let $P_i = (x_i, y_i)$ for $1 \le i \le n$ with $x_i, y_i \in \mathbb{F}$. The following is Gift-Wrapping Algorithm using Algorithm 1:*

Find the highest point (the largest y coordinate).
Let i_1 be the index of the highest point and set $i := i_1$.
repeat
  for *each j ($\neq i$)* do
    Find k satisfying $\mathrm{AppOrient2D}(x_i, y_i, x_j, y_j, x_k, y_k) \le 0$.
    If *we find such k twice or more,*
    then *we select the farthest one from (x_i, y_i).*
  Output (P_i, P_k) as a hull edge.
  $i := k$;
until $i = i_1$;

Next we denote Gift-Wrapping Algorithm using Algorithm 5.

ALGORITHM 7. *Let $P_i = (x_i, y_i)$ for $1 \le i \le n$ with $x_i, y_i \in \mathbb{F}$. The following is Gift-Wrapping Algorithm using Algorithm 5:*

Find the highest point (the largest y coordinate).
Let i_1 be be the index of the highest point and set $i := i_1$.
repeat
  for *each j ($\neq i$)* do
    Find k satisfying $\mathrm{Verified1}(x_i, y_i, x_j, y_j, x_k, y_k) \le 0$.
    If *we find such k twice or more,*
    then *we select the farthest one from (x_i, y_i).*
  Output (P_i, P_k) as a hull edge.
  $i := k$;
until $i = i_1$;

Here, we discuss acceleration of Algorithm 7. To do this, we modify and adapt Algorithm 4 for Algorithm 7. Algorithm 4 computes $\mathbf{err}_3$ for each orientation problem, respectively. However, if we compute a global error constant in advance, each computation for $\mathbf{err}_3$ can be omitted. Now we derive the global error constant. Let m_x and m_y be

$$m_x = 2^{\lceil \log_2 \max_{1 \le i \le n}(|x_i|) \rceil}, \quad m_y = 2^{\lceil \log_2 \max_{1 \le i \le n}(|y_i|) \rceil}. \tag{7}$$

Then, (6) is bounded by using (7) as follows:

$$\begin{aligned} \mathbf{err}_3 &\le fl((3\mathbf{u} + 25\mathbf{u}^2)(|(x_i - x_k)(y_j - y_k) + (y_i - y_k)(x_j - x_k)|) + \underline{\mathbf{u}}_N) \\ &\le fl((3\mathbf{u} + 25\mathbf{u}^2)(|(|x_i| + |x_k|)(|y_j| + |y_k|)| + (|y_i| + |y_k|)(|x_j| + |x_k|)) + \underline{\mathbf{u}}_N) \\ &\le fl((3\mathbf{u} + 25\mathbf{u}^2)((2m_x \cdot 2m_y) + (2m_y \cdot 2m_x)) + \underline{\mathbf{u}}_N) \\ &\le fl(8(3\mathbf{u} + 25\mathbf{u}^2) \cdot m_x \cdot m_y + \underline{\mathbf{u}}_N) =: \mathbf{err}_4 \end{aligned} \tag{8}$$

Here $\mathbf{err}_4$ does not depend on combinations of i, j and k so that $\mathbf{err}_4$ can globally be used. We present an algorithm for the 2D orientation problem using the global error constant.

ALGORITHM 8. *Let $A = (a_x, a_y)$, $B = (b_x, b_y)$ and $C = (c_x, c_y)$ on two-dimensional space. Suppose that an oriented line passes from A to B. Let $\mathbf{err}_4$ be the computed result of (8). If the result $\mathbf{d}$ is positive (negative), the point C is left (right) to the oriented line from A to B. Even if the result is zero, the problem is correctly solved.*

function $\mathbf{d} = \text{Verified2}(a_x, a_y, b_x, b_y, c_x, c_y, \mathbf{err}_4)$
  $l = (a_x - c_x)(b_y - c_y)$;
  $r = (b_x - c_x)(a_y - c_y)$;
  $\mathbf{d} = l - r$;
  if $|\mathbf{d}| > \mathbf{err}_4$ *% quick check*
    return $\mathbf{d}$;
  end
  $\mathbf{err}_3 = (3\mathbf{u} + 25\mathbf{u}^2)(|l| + |r|) + \underline{\mathbf{u}}_N$;
  if $|\mathbf{d}| > \mathbf{err}_3$
    return $\mathbf{d}$;
  end
  $\mathbf{d} = \text{Robust}(a_x, a_y, b_x, b_y, c_x, c_y)$;
  return $\mathbf{d}$;
end

Finally, we present Gift-Wrapping Algorithm by using Algorithm 8.

ALGORITHM 9. *Let $P_i = (x_i, y_i)$ for $1 \le i \le n$ with $x_i, y_i \in \mathbb{F}$. This is Gift-Wrapping Algorithm using Algorithm 8:*

Find the highest point (the largest y coordinate).
Let i_1 be the index of the highest point and set $i := i_1$.
$\alpha = \max_{1 \le i \le n} |x_i|$.
$\beta = \max_{1 \le i \le n} |y_i|$.
$m_x = 2\hat{\ } \text{ceil}(\log 2(|\alpha|))$;
$m_y = 2\hat{\ } \text{ceil}(\log 2(|\beta|))$;
$\mathbf{err}_4 = 8(3\mathbf{u} + 25\mathbf{u}^2) m_x m_y + \underline{\mathbf{u}}_N$;
repeat
  for *each j ($\neq i$)* do
    Find k satisfying $\text{Verified2}(x_i, y_i, x_j, y_j, x_k, y_k, \mathbf{err}_4) \le 0$.
    If *we find such k twice or more,*
    then *we select the farthest one from (x_i, y_i).*
  end
  Output (P_i, P_k) as a hull edge.
  $i := k$;
until $i = i_1$;

Table II. Ratios of elapsed times with various ϕ (Intel Core 2 Duo 1.2 GHz).

$\phi \setminus n$	Visual C++		Intel C++	
	M2/M1	M3/M1	M2/M1	M3/M1
1	1.20	1.13	1.27	1.22
5	1.22	1.15	1.30	1.22
10	1.20	1.14	1.33	1.29
15	1.24	1.16	1.33	1.26
20	1.22	1.15	1.30	1.26

4. Numerical Experiments

In this section, we compare computational performance of Algorithms 6, 7 and 9:

- M1: Algorithm 6
- M2: Algorithm 7
- M3: Algorithm 9

We generate n-points $(P_i(x_i, y_i),\ 1 \le i \le n)$ as follows:

$$\begin{aligned} x_i &= (\text{rand}() - 0.5). * \exp(\phi * \text{randn}()) \\ y_i &= (\text{rand}() - 0.5). * \exp(\phi * \text{randn}()) \end{aligned}$$

Here, all coordinates are expressed by double precision floating-point numbers. The function randn() returns a scalar value containing pseudo-random values drawn from the standard normal distribution. The function rand() returns a scalar value containing pseudo-random values drawn from the standard uniform distribution on the open interval $(0, 1)$. The function $\exp(X)$ returns the exponential for X. If ϕ is large, there is a big difference in the order of the magnitude among the elements of x and y. The codes for each method are compiled by Visual C++ 2005, Intel C++ Compiler 9.1 and GCC 4.1.3. These examples are performed on Intel Cure 2 Duo 1.2 GHz (Table II) and Xeon 2.5 GHz (Table III). We set n as 10^7 for Table II and 10^8 for Table III.

In Tables II and III, the ratios of computing time, (M2 / M1) and (M3 / M1), are shown (the average of 10 examples). From Tables II and III, the average of the ratios is from 1.09 to 1.33 so that additional costs for verified computations are not expensive. Algorithm 9 is always faster than Algorithm 7 in these tests.

Table III. Ratios of elapsed times with various ϕ on Intel Xeon 2.5 GHz.

$\phi \setminus n$	Visual C++		Intel C++		GCC	
	M2 / M1	M3 / M1	M2 / M1	M3 / M1	M2 / M1	M3 / M1
1	1.24	1.12	1.16	1.09	1.33	1.29
5	1.25	1.14	1.16	1.09	1.33	1.27
10	1.25	1.14	1.16	1.09	1.33	1.28
15	1.24	1.14	1.16	1.09	1.33	1.28
20	1.25	1.14	1.16	1.09	1.33	1.28

5. Conclusion

In this paper, we first discuss the property of the filter. We can modify and prove that the filter correctly works even if overflow or underflow occurs. Also, we apply it into the convex hull for a set of points on two-dimensional space. We also develop a new filter using the global error. It is confirmed from numerical examples that Algorithm 9 works faster than Algorithm 7.

References

IEEE Standard for Floating-Point Arithmetic , Std 754–2008, 2008.

Guillaume Melquiond and Sylvain Pion. Formally certified floating-point filters for homogenous geometric predicates. *Theoretical Informatics and Applications, Special issue on Real Numbers*, 41:57–69, 2007.

H. Brönnimann, C. Burnikel and S. Pion. Interval Arithmetic Yields Efficient Dynamic Filters for Computational Geometry. *Discrete Applied Mathematics*, 109:25–47, 2001.

H. Brönnimann and M. Yvinec. Efficient Exact Evaluation of Signs of Determinants. *Algorithmica*, 27:21–56, 2000.

L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, C. Yap. Classroom Examples of Robustness Problems in Geometric Computations. *Computational Geometry*, 40, 61–78, 2008.

J. O'Rourke. Computational geometry in C, Second Edition. *Cambridge University Press*, 2001.

V. Y. Pan and Y. Yu. Certified Computation of the Sign of a Matrix Determinant. *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms, 715-724*, ACM Press, New York, and SIAM Publications, Philadelphia, 1999.

K. Ozaki, T. Ogita, S. M. Rump, S. Oishi. Adaptive and Efficient Algorithm for 2D Orientation Problem. *Japan Journal of Industrial and Applied Mathematics*, to appear.

J. R. Shewchuk. Adaptive Precision Floating-point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18, 305–363, 1997.

MATLAB Programming version 7, the Mathworks..