Line Segment Intersection Testing

Yong-Kang Zhu, Jun-Hai Yong, Guo-Qin Zheng

Line Segment Intersection Testing

Y.-K. Zhu, **J.-H. Yong**, and **G.-Q. Zheng**, Beijing

Received March 26, 2004; revised November 29, 2004
Published online: May 18, 2005

Abstract

A method for accurately determining whether two given line segments intersect is presented. This method uses the standard floating-point arithmetic that conforms to IEEE 754 standard. If three or four ending points of the two given line segments are on a same vertical or horizontal line, the intersection testing result is obtained directly. Otherwise, the ending points and their connections are mapped onto a 3×3 grid, and the intersection testing falls into one of the five testing classes. The intersection testing method is based on our method for floating-point dot product summation, whose error bound is $1ulp$. Our method does not have the limitation in the method of Gavrilova and Rokne (2000) that the product of two floating-point numbers is calculated by a twice higher precision floating-point arithmetic than that of the multipliers. Furthermore, this method requires less than one-fifth of the running time used by the method of Gavrilova and Rokne (2000), and our new method for calculating the sign of a sum of n floating-point numbers requires less than one-fifteenth of the running time used by ESSA.

AMS Subject Classifications: 65G50, 65B10, 68U05.

Keywords: Intersection testing for line segments, floating-point arithmetic, rounding error, dot product summation.

1. Introduction

The problem of whether two line segments intersect in the plane in Euclidean geometry is one of the most fundamental issues in areas such as geometric modelling, computer graphics, GIS (Geographic Information System) and computational geometry [6], [21]. A lot of researches on this problem focus on the accuracy of the testing results by computers. Ramshaw [17] finds that the errors in floating-point arithmetic may make two lines intersect more than once. Exact arithmetic methods are able to make each step of computations exact and always produce correct results. However, they require much more execution time than the inaccuracy floating-point arithmetic methods, which are frequently used in a lot of softwares. Thus, one of the research directions on this problem is to improve the efficiency of the exact arithmetic. Milenkovic [13] presents a technique, named double precision geometry, which could replace exact arithmetic with rounded arithmetic in computing intersections of a set of lines or line segments. Fortune and Wyk [4] give a method to reduce the cost of exact integer arithmetic with a floating-point filter and interval analysis [14].

Later, they [5] provide another method, named static-analysis techniques, to reduce the cost of exact integer arithmetic.

Another research direction is to investigate how to reduce the error when using fixed-length floating-point arithmetic. Priest [15] presents algorithms to compensate errors in the floating-point operations and applies them to computing the intersection of a line and a line segment. This method is reliable, but in some cases the time cost is very expensive. Recently, Gavrilova and Rokne [6] give a method that converts the problem of line segment intersection testing into calculating signs of several dot product summations. If those signs could be computed exactly, then the testing result is reliable. In their method, signs are obtained by ESSA (Exact Sign of Sum Algorithm) [18] method, which can exactly calculate the sign of a sum of n floating-point numbers. When the coordinates of the ending points are given with single precision floating-point numbers, their method uses a double precision floating-point number to exactly calculate the product of two single precision numbers. However, the double precision is the highest precision floating-point arithmetic available in typical computer languages. In other words, when coordinates are represented with double floating-point numbers, a new floating-point type, which has twice higher precision than the double floating-point type, has to be created to compute the products.

Our method falls into the second research direction. A line segment that degenerates into a point is not considered in this paper. If two line segments share a common ending point, or more than two ending points are on the same vertical or horizontal line, then the intersection testing is processed separately. Otherwise, the ending points of two give line segments are mapped onto a 3×3 grid, which falls into one of the five testing classes. The intersection testing results of the first two testing classes can be obtained immediately, and in other testing classes the intersection testing is converted into calculating floating-point dot product summations. Our method for dot product summation uses the floating-point arithmetic which conforms to IEEE 754 standard [2]. The addition and multiplication operations in our dot product summation algorithm are exactly performed by computing the exact error for either operation. Then, the final result of dot product summation is obtained by our floating-point summation method.

The paper is organized as follows. In Sect. 2, our method for dot product summation is presented with error analysis. The method for deciding whether two line segments intersect is described in Sect. 3. Some comparisons are made in Sect. 4. Conclusions are drawn in Sect. 5.

2. Dot Product Summation

Our method for line segment intersection testing is based on dot product summation. The method for dot product summation is introduced in this section. It calculates

$$s = \sum_{i=1}^{n} a_i \cdot b_i, \quad i = 1, 2, ..., n, \tag{1}$$

where each a_i or b_i is a floating-point number. There are two fundamental operations, addition and multiplication, in Eq. (1). In floating-point addition, when exponents

of two summands are not equal, the number, whose exponent is smaller, will right shift the mantissa to make its exponent increase until two exponents are equal, and then add them. If the bits discarded by right shifting have at least one significant digit, the error occurs. In floating-point multiplication, some significant digits of the product may be discarded because of the length limit of the floating-point number. In order to make the result as accurate as possible, we must guarantee the accuracy of these two basic operations.

In this paper, only fixed-length floating-point arithmetic is considered. And we use β to represent the base (sometimes called the radix), use t to represent the precision, and use the $fl(a \circ b)$ (where $\circ \in \{+, -, \cdot, /\}$) to represent the floating-point arithmetic. So a way of expressing the floating-point number d is

$$d = \pm 0.d_1 d_2 \ldots d_t \times \beta^e, \tag{2}$$

where each d_i is an integer and satisfies $0 \leq d_i \leq \beta - 1$, and $d_1 \neq 0$ for normalized numbers. The algorithms are often implemented on a machine using single (named $float$, $t = 24$, $\beta = 2$) or double (named $double$, $t = 53$, $\beta = 2$) precision floating-point arithmetic, which conforms to IEEE 754 standard [2], so only the case that $\beta = 2$ and $t \geq 3$ is considered in this paper. Additionally, floating-point overflow and underflow are not taken into account.

Knuth [10] proves that when $\beta = 2$ the error of floating-point addition $fl(a + b)$ ($|a| \geq |b|$), can be exactly obtained by $e = fl(fl(a - s) + b)$, where $s = fl(a + b)$. According to his conclusion, we obtain the exact addition algorithm as follows:

Algorithm 1: Exact addition.
Input: two floating-point numbers, a and b ($\beta = 2$).
Output: the sum s and the error e, where $s + e = a + b$.

1. $s = fl(a + b)$;
2. If ($EXP(a) < EXP(b)$), then $e = fl(fl(b - s) + a)$;
3. Otherwise $e = fl(fl(a - s) + b)$;
4. END.

In Algorithm 1, $EXP(x)$ represents the exponent of the floating-point number x. And, we have

$$EXP(s) \geq EXP(e) + t, \tag{3}$$

according to the algorithm.

As the addition of two floating-point numbers can be represented by a sum and an error, the multiplication between two floating-point numbers also has such a property, which means that the exact value of the product of two floating-point numbers can be represented by a sum of two floating-point numbers [12]. Priest [15] introduces a compound representation for floating-point numbers, called expansions, to reduce the rounding error and compute an accurate error. But this method is designed for

arbitrary floating-point arithmetic, which makes the algorithm complicated. Our method only considers the case $\beta = 2$, and is based on the following theorems.

Theorem 1: *Assume that $t \geq 3$. When t is even, the exact value of the product of two floating-point numbers can be expressed by a sum of four partial products:*

$$\begin{aligned} a \cdot b &= (a_1 + a_2) \cdot (b_1 + b_2) \\ &= a_1 \cdot b_1 + a_1 \cdot b_2 + a_2 \cdot b_1 + a_2 \cdot b_2 \\ &= fl(a_1 \cdot b_1) + fl(a_1 \cdot b_2) + fl(a_2 \cdot b_1) + fl(a_2 \cdot b_2), \end{aligned}$$

where each of the mantissas of a and b is evenly divided into two parts: a_1 has the first $\frac{t}{2}$ digits of a, $a_2 = a - a_1$; and b_1 has the first $\frac{t}{2}$ digits of b, $b_2 = b - b_1$.

When t is odd, the exact value of the product of two floating-point numbers can be expressed by a sum of five partial products:

$$\begin{aligned} a \cdot b &= (a_1 + a_2) \cdot (b_1 + b_2) \\ &= a_1 \cdot b_1 + a_1 \cdot b_2 + a_2 \cdot b_1 + a_2 \cdot b_2 \\ &= a_1 \cdot b_1 + a_1 \cdot b_2 + a_2 \cdot b_1 + (a_3 + a_4) \cdot b_2 \\ &= fl(a_1 \cdot b_1) + fl(a_1 \cdot b_2) + fl(a_2 \cdot b_1) + fl(a_3 \cdot b_2) + fl(a_4 \cdot b_2), \end{aligned}$$

where a_1 has the first $\frac{t-1}{2}$ digits of a, $a_2 = a - a_1$; and b_1 has the first $\frac{t-1}{2}$ digits of b, $b_2 = b - b_1$. What is more, the mantissa of a_2 is divided into two parts again: a_3 has the first $\frac{t-1}{2}$ digits of a_2; and $a_4 = a_2 - a_3$.

Proof: When t is even, since a_1, a_2, b_1, and b_2 all have at most $\frac{t}{2}$ significant digits, the product of every two of them has at most $\frac{t}{2} + \frac{t}{2} = t$ significant digits. Thus, we have $a_1 \cdot b_1 = fl(a_1 \cdot b_1)$, $a_1 \cdot b_2 = fl(a_1 \cdot b_2)$, $a_2 \cdot b_1 = fl(a_2 \cdot b_1)$, and $a_2 \cdot b_2 = fl(a_2 \cdot b_2)$.

When t is odd, the mantissa cannot be divided evenly. For example, according to IEEE754 [2], the double precision floating-point type, *double*, has $t = 53$. In such a case, the mantissa is divided into two components, $\frac{t-1}{2}$ and $\frac{t+1}{2}$. Thus, the product of the components, both having $\frac{t+1}{2}$ bits of significant digits, may have $\frac{t+1}{2} + \frac{t+1}{2} = t+1$ significant digits, which means the last significant digit will be discarded. So, a_2 is divided again into a_3 and a_4, where a_3 has the first $\frac{t-1}{2}$ digits of a_2, and $a_4 = a_2 - a_3$. Clearly, a_4 has at most 1 significant digit. Because $a_1 \cdot b_1$ has at most $\frac{t-1}{2} + \frac{t-1}{2} = t-1$ significant digits, $a_2 \cdot b_1$ and $a_1 \cdot b_2$ have at most $\frac{t-1}{2} + \frac{t+1}{2} = t$ significant digits, $a_3 \cdot b_2$ has at most $\frac{t-1}{2} + \frac{t+1}{2} = t$ significant digits, and $a_4 \cdot b_2$ has at most $1 + \frac{t+1}{2} \leq t$ (when $t \geq 3$) significant digits, we have $a_1 \cdot b_1 = fl(a_1 \cdot b_1)$, $a_1 \cdot b_2 = fl(a_1 \cdot b_2)$, $a_2 \cdot b_1 = fl(a_2 \cdot b_1)$, $a_3 \cdot b_2 = fl(a_3 \cdot b_2)$, and $a_4 \cdot b_2 = fl(a_4 \cdot b_2)$. □

According to Theorem 1, the floating-point multiplication between two numbers is converted into a summation of four or five floating-point numbers, when $t \geq 3$. Then, with Algorithm 1, we obtain the exact multiplication algorithm.

Algorithm 2: Exact multiplication.
Input: two floating-point numbers, a and b.
Output: the product p and the error q, where $a \cdot b = p + q$.

1. Let $a_1 = a$, and clear the last $\lceil t/2 \rceil$ digits of the mantissa of a_1. Namely, set them zero;
2. Let $b_1 = b$, and clear the last $\lceil t/2 \rceil$ digits of the mantissa of b_1;
3. $a_2 = fl(a - a_1), \quad b_2 = fl(b - b_1)$;
4. $p_1 = fl(a_1 \cdot b_1), \quad p_2 = fl(a_1 \cdot b_2), \quad p_3 = fl(a_2 \cdot b_1), \quad p_4 = fl(a_2 \cdot b_2), \quad p_5 = 0.0$;
5. If t is odd, and both the $\frac{t+1}{2}$-th digit of a_2's mantissa and that of b_2's are 1, then
   (a) Let $a_3 = a_2$, and clear the $\frac{t+1}{2}$-th digit of a_3;
   (b) $a_4 = fl(a_2 - a_3)$;
   (c) $p_4 = fl(a_3 \cdot b_2), \quad p_5 = fl(a_4 \cdot b_2)$;
   (d) Call Algorithm 1 to add p_4, p_5, and return the sum using p_4 and the error using p_5;
6. Call Algorithm 1 to add p_2, p_3, and return the sum using p_2 and the error using p_3;
7. Call Algorithm 1 to add p_3, p_4, and return the sum using p_3 and the error using p_4;
8. Call Algorithm 1 to add p_2, p_3, and return the sum using p_2 and the error using p_3;
9. Call Algorithm 1 to add p_1, p_2, and return the sum using p and the error using p_2;
10. $q = fl(fl(p_2 + p_3) + fl(p_4 + p_5))$;
11. END.

Each variable in Algorithm 2 has the same type as a or b, and each step before Step 10 does not discard any significant digit. Thus the accuracy of Algorithm 2 depends on Step 10, namely whether the sum of p_2, p_3, p_4 and p_5 can be represented by only one floating-point number, q.

Theorem 2: *The outputs of Algorithm 2, p and q, satisfy $a \cdot b = p + q$.*

Proof: Without loss of generality, we assume $a > 0$ and $b > 0$. When t is even, the condition in Step 5 cannot be satisfied, namely $p_5 = 0$. According to Theorem 1, we have $p_1 = fl(a_1 \cdot b_1) = a_1 \cdot b_1$, $p_2 = fl(a_1 \cdot b_2) = a_1 \cdot b_2$, $p_3 = fl(a_2 \cdot b_1) = a_2 \cdot b_1$, and $p_4 = fl(a_2 \cdot b_2) = a_2 \cdot b_2$ after Step 4. If both p_2 and p_3 are zero, then we have $q = 0$, whence the theorem is proved. Otherwise, after Steps 6 and 7, we have $EXP(p_3) \geq EXP(p_4) + t$ according to Inequation (3). If a or b or both are not normalized numbers, then $a_1 \cdot b_1$ may equal to 0. So if $a_1 \cdot b_1 = 0$, then after Step 8, we have $p_1 = 0$, and $EXP(p_2) \geq EXP(p_3) + t$ and $EXP(p_3) \geq EXP(p_4) + t$. Otherwise if $a_1 \cdot b_1 \neq 0$, then after Step 8, so we obtain $EXP(p_1) \geq EXP(p_2)$, and $EXP(p_2) \geq EXP(p_3) + t$ and $EXP(p_3) \geq EXP(p_4) + t$. Thus, after Step 9 the product of a and b can be obtained with p. Since the exact value of the product of

two floating-point numbers can be represented by a sum of two floating-point numbers, $p_2 + p_3 + p_4 + p_5$ can be represented by one floating-point number. Therefore, no significant digit is discarded in the algorithm, which means the algorithm is exact when t is even.

When t is odd and the condition in Step 5 is satisfied, $a_2 \cdot b_2$ may be unequal to $fl(a_2 \cdot b_2)$. Here, the condition that "both the $\frac{t+1}{2}$-th digit of a_2's mantissa and that of b_2's are 1" in Step 5 is to exclude the case that the digit which may be discarded by $fl(a_2 \cdot b_2)$ is 0. For the result is always exact when a zero is discarded. It is possible that two numbers both having $\frac{t+1}{2}$ significant digits have a t-digit product, namely no significant digit is discarded. So even when the condition in Step 5 is satisfied, we also have $fl(a_2 \cdot b_2) = a_2 \cdot b_2$, which means the division of a_2 is redundant. But this redundancy does not produce any error. After Step 5(d) is done, we have $EXP(p_4) \geq EXP(p_5) + t$. Similar to the case that t is even, we have that the product of a and b can be obtained with p, and the sum of p_2, p_3, p_4 and p_5 can be represented by one floating-point number. Therefore, Algorithm 2 provides an exact multiplication. □

So, Eq. (1) can be calculated by

$$\sum_{i=1}^{n} a_i \cdot b_i = \sum_{i=1}^{n} p_i + \sum_{i=1}^{n} q_i, \tag{4}$$

where p_i and q_i are obtained by multiplying a_i and b_i using Algorithm 2. Thus, the problem is converted to floating-point summation, which is to sum up all the p_i and q_i together to obtain the result.

Many people [1], [3], [7], [9], [11], [16], [23] have devoted to studying floating-point summation. The method [3] is based a special assumption that it exists several extra precise floating-point accumulators, which is higher than the register used to save summands. However, in practice, there is no particular accumulator in computer when we use the highest precision type of floating-point number. Other methods can reduce the rounding error in some extent, and can perform well in some special cases, but most of them have an error bound that depends on n, where n is the number of summands. The method [23] has presented an algorithm, which iteratively distills the summands without discarding any significant digit until the partial sums cannot change the whole sum. The error bound of it is $1ulp$ (unit in last place [8]), independent of n, which has been proved in that paper. So we can use it here for summing all the p_i and q_i in Eq. (4). However, the error of product, q_i, is small relative to p_i, and can be put aside while the algorithm runs in the first loop of distillation. The complete algorithm for floating-point dot product summation is given as follows:

Algorithm 3: Dot product summation.
Input: $2n$ floating-point numbers, $a_1, a_2, ..., a_n$ and $b_1, b_2, ..., b_n$.
Output: The result s, i.e. the floating-point value of $\sum_{i=1}^{n} a_i \cdot b_i$.

1. Initialize three empty sets, $\mathcal{P}$, $\mathcal{N}$, and $\mathcal{Q}$, where $\mathcal{P}$ records positive numbers, $\mathcal{N}$ records negative numbers, and $\mathcal{Q}$ records q_i in Eq. (4);
2. For $i = 1$ to n
   (a) Call Algorithm 2 to multiple a_i and b_i, and return the product using p_i and the error using q_i;
   (b) if $p_i > 0$, then put p_i into $\mathcal{P}$. Otherwise if $p_i < 0$, then put p_i into $\mathcal{N}$;
   (c) if $q_i \neq 0$, then put q_i into $\mathcal{Q}$;
3. Let $s = 0$, $e_1 = 0$, $e_2 = 0$, and $flagFirst = 1$;
4. If $e_1 \neq 0$, then put e_1 into the end of $\mathcal{P}$ or $\mathcal{N}$ depending on its sign;
5. If $e_2 \neq 0$, then put e_2 into the end of $\mathcal{P}$ or $\mathcal{N}$ depending on its sign;
6. Let $n_1 =$ the size of $\mathcal{P}$, and $s_+ = 0$;
7. For $i = 1$ to n_1
   (a) Remove the head number of $\mathcal{P}$, say a, and let $b = s_+$;
   (b) Call Algorithm 1 to add a and b, and return the sum using s_+ and the error using e. If $e \neq 0$, then redistribute e to the end of $\mathcal{P}$ or $\mathcal{N}$ depending on its sign;
8. Call Algorithm 1 to add s and s_+, and return the sum using s and the error using e_1;
9. Let $n_2 =$ the size of $\mathcal{N}$, and $s_- = 0$;
10. For $i = 1$ to n_2
    (a) Remove the head number of $\mathcal{N}$, say a, and let $b = s_-$;
    (b) Call Algorithm 1 to add a and b, and return the sum using s_- and the error using e. If $e \neq 0$, then redistribute e to the end of $\mathcal{P}$ or $\mathcal{N}$ depending on its sign;
11. Call Algorithm 1 to add s and s_-, and return the sum using s and the error using e_2;
12. If $flagFirst \neq 0$, then
    (a) Put all the numbers in $\mathcal{Q}$ into $\mathcal{P}$ or $\mathcal{N}$ according to their signs, and then let $flagFirst = 0$;
    (b) Go to Step 4;
13. $\hat{s} = 2^{\max\left((EXP(s_+), EXP(s_-)\right)-t} \cdot \max(n_+, n_-)$;
14. If $fl(s + \hat{s}) \neq s$, then go to Step 4;
15. $s = fl\big(s + fl(e_1 + e_2)\big)$;
16. END.

Before Step 15, Algorithm 3 does not discard any significant digit since the addition and multiplication are exact, and all the errors are saved in sets $\mathcal{P}$ and $\mathcal{N}$. So the error of the whole algorithm, say e_0, is produced only at Step 15. After Step 15 has been performed, e_0 is composed of the sum of remaining numbers in the sets $\mathcal{P}$ and $\mathcal{N}$, and the error occurred in Step 15. According to the Inequation (3), the value of $\hat{s}$ in Step 13 is the upper bound of the sum of the remaining numbers in the sets $\mathcal{P}$ and $\mathcal{N}$. When the condition in Step 14 has been satisfied, the value of $\hat{s}$

is too small to change the whole sum s. Thus, we have $|s - \hat{s}| \leq 0.5ulp$. Because $EXP(s) \geq EXP(e_1) + t$ and $EXP(s) \geq EXP(e_2) + t$, we have the error bound of $fl(s + fl(e_1 + e_2))$ is also $0.5ulp$. Therefore, the whole error bound of Algorithm 3 is $e_0 = 0.5 + 0.5 = 1(ulp)$.

Based on the floating-point dot product summation algorithm above, we gives our method for line segment intersection testing in the next section.

3. Intersection Testing

A clear definition of the line segment intersection test is: two line segments $\overline{\mathbf{a}_1\mathbf{a}_2}$ and $\overline{\mathbf{b}_1\mathbf{b}_2}$ in $\mathbb{R}^2$ are given by ending points: $\mathbf{a}_1(a_{1x}, a_{1y})$, $\mathbf{a}_2(a_{2x}, a_{2y})$, $\mathbf{b}_1(b_{1x}, b_{1y})$, and $\mathbf{b}_2(b_{2x}, b_{2y})$, then the test is to determine whether $\overline{\mathbf{a}_1\mathbf{a}_2}$ and $\overline{\mathbf{b}_1\mathbf{b}_2}$ intersect. Three testing results are: two line segments intersect at an ending point; intersect but not at an ending point; and do not intersect. Above all, we assume that every line segment does not degenerate into a point, and two given line segments do not share the same ending point.

When any two ending points of two given line segments are not on the same vertical or horizontal line, each ending point can be mapped onto a 3×3 grid according to the new grid coordinates (i, j), where i means the x-coordinate of this ending point is the i-th smallest number of the x-coordinates of all ending points, j means the y-coordinate of this ending point is the j-th smallest number of the y-coordinates of all ending points, and the grid coordinates are defined as Fig. 1. Thus, there is only one point on each vertical line or horizontal line in a grid. Then, we redefine these four ending points as $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$, and $\mathbf{p}_4$ by the ascending order of their x-coordinates. For example, four ending points are $\mathbf{a}_1$ (a_{1x}, a_{1y}), $\mathbf{a}_2$ (a_{2x}, a_{2y}), $\mathbf{b}_1$ (b_{1x}, b_{1y}), and $\mathbf{b}_2$ (b_{2x}, b_{2y}). If $b_{2x} < a_{1x} < a_{2x} < b_{1x}$ and $a_{2y} < b_{2y} < a_{1y} < b_{1y}$, then the grid coordinates of $\mathbf{a}_1$ is (2,3), that of $\mathbf{a}_2$ is (3,1), that of $\mathbf{b}_1$ is (4,4), and that of $\mathbf{b}_2$ is (1,2), according to the ordering of their coordinates. And $\mathbf{a}_1$, $\mathbf{a}_2$, $\mathbf{b}_1$, and $\mathbf{b}_2$ are redefined as $\mathbf{p}_2$, $\mathbf{p}_3$, $\mathbf{p}_4$, and $\mathbf{p}_1$ respectively. This example can be represented as Fig. 2(5). Another example is if $b_{2x} < a_{1x} < a_{2x} < b_{1x}$ and $a_{2y} < a_{1y} < b_{2y} < b_{1y}$, then it corresponds to Fig. 2(6). Thus, all the cases of sorting results of four ending points can be represented by $4! = 24$ grids as shown in Fig. 2, where the first six grids are marked with the tags ($\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$, and $\mathbf{p}_4$) of ending points and others are omitted.

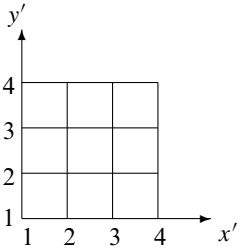

Fig. 1. Grid coordinates system

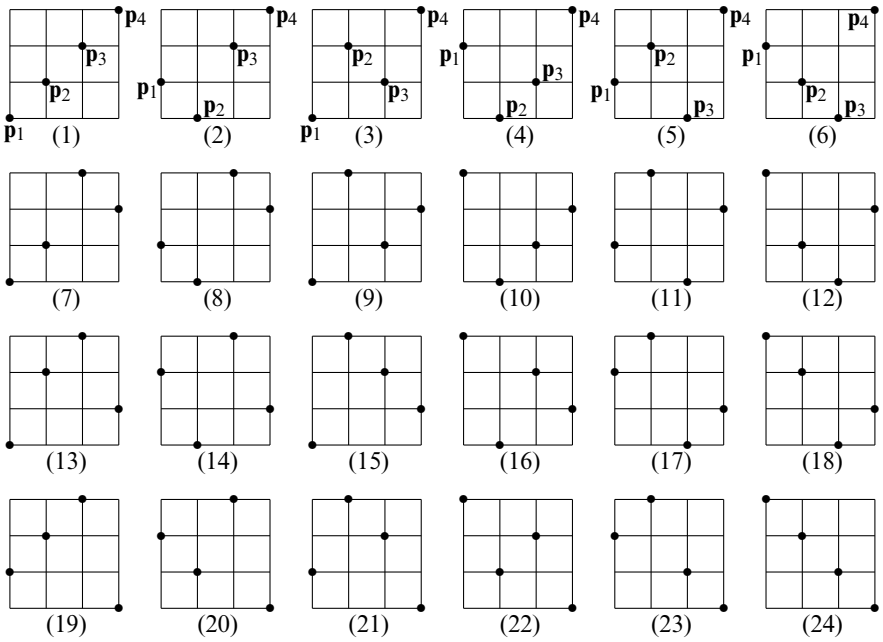

Fig. 2. Mapping four ending points on a 3×3 grid

In Fig. 2, each grid corresponds to a unique 4-digit number, $n_1n_2n_3n_4$, which makes points $\mathbf{p}_{n_1}$, $\mathbf{p}_{n_2}$, $\mathbf{p}_{n_3}$ and $\mathbf{p}_{n_4}$ be arranged in the ascending order of their y-coordinates. For example, the y-coordinate ordering (ascending) of the grid in Fig. 2(5) is $\mathbf{p}_3$, $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_4$. So, Fig. 2(5) corresponds to the number, 3124. Similarly, Fig. 2(6) corresponds to 3214. Thus, each grid in Fig. 2 can be represented by a unique case number n_c, ranging from 1 to 24, which can be obtained from $n_1n_2n_3n_4$ by the function:

$$C(n_1n_2n_3n_4) = 3! \times w_4 + 2! \times w_3 + w_2 + 1,$$

where w_i is the number of digits that are smaller than i on the right side of digit i in $n_1n_2n_3n_4$, for all the $i = 2, 3, 4$. For example, when calculating $C(3124)$, we have $w_4 = 0$, $w_3 = 2$, and $w_2 = 0$, so $C(3124) = 6 \times 0 + 2 \times 2 + 0 + 1 = 5$. Similarly, $C(3214) = 6 \times 0 + 2 \times 2 + 1 + 1 = 6$. All the grids in Fig. 2 are arranged as that the case number n_c of each grid is just equal to its serial number of the subfigure.

Since every two ending points form a line segments, each grid case in Fig. 2 corresponds to three subcases according to the connections between ending points. Each subcase is represented by the subcase number n_s, where n_s means that one line segment connects points $\mathbf{p}_1$ and $\mathbf{p}_{n_s+1}$, and another line segment connects the other two ending points. So, $n_s = 1$, 2, and 3 correspond to these three subcases. Therefore, when any two ending points does not on the same vertical or horizontal line, two line segments in the plane can be entirely mapped into $24 \times 3 = 72$ subcases, and each subcase can be represented by a unique pair, (n_c, n_s), where $n_c = 1, 2, \ldots, 24$ and $n_s = 1, 2, 3$.

Our method for intersection testing is to analyze the position relation between each ending point and the other line segment. If two ending points of either line segment are on the different sides of the other line segment, or one ending point of a line segment is just on the other line segment, then these two segments intersect. Otherwise, they do not intersect each other. The position value of a point to a line segment is used to represent their position relationship, which is defined as follows.

Definition 1: *The line passing through the line segment that connects points* $\mathbf{a}(x_1, y_1)$ *and* $\mathbf{b}(x_2, y_2)$ *is* $f(x, y) = 0$*, where* $f(x, y) =$

$$\begin{cases} yx_2 - yx_1 - xy_2 + xy_1 + x_1y_2 - x_2y_1, & \begin{cases} \text{when } x_1 < x_2, \\ \text{or when } x_1 = x_2,\ y_1 > y_2. \end{cases} \\ yx_1 - yx_2 - xy_1 + xy_2 + x_2y_1 - x_1y_2, & \begin{cases} \text{when } x_1 > x_2, \\ \text{or when } x_1 = x_2,\ y_1 < y_2. \end{cases} \end{cases} \tag{5}$$

Definition 2: *The position value of a point* $\mathbf{p}(x_p, y_p)$ *to a line segment connecting points* $\mathbf{a}(x_1, y_1)$ *and* $\mathbf{b}(x_2, y_2)$ *is defined as*

$$PV(\mathbf{p}, \mathbf{a}, \mathbf{b}) = \begin{cases} 1, & \text{when } f(x_p, y_p) > 0; \\ -1, & \text{when } f(x_p, y_p) < 0; \\ 0, & \text{when } f(x_p, y_p) = 0 \text{ and } \mathbf{p}(x_p, y_p) \text{ is NOT on } \overline{\mathbf{ab}}; \\ 2, & \text{when } f(x_p, y_p) = 0 \text{ and } \mathbf{p}(x_p, y_p) \text{ is on } \overline{\mathbf{ab}}; \end{cases} \tag{6}$$

where $f(x, y)$ *is the function of the line passing through* $\overline{\mathbf{ab}}$*, according to Definition 1.*

For example, if the given point is on the line segment, then its position value is 2. If the given point is only on the line passing through the line segment but not on the segment, then the position value is 0. Otherwise, the position value of a point to a line segment can be clearly illustrated with Fig. 3.

We find that calculating the position value needs to compute the function f, which is a dot product summation. Thus, with Algorithm 3, we obtain the algorithm for calculating the position value of a point to a line segment.

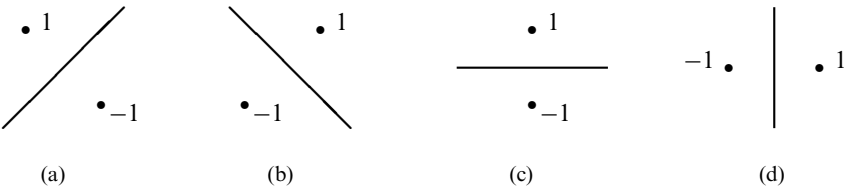

Fig. 3. Position values given by Definition 2

Algorithm 4: Calculating the position value of a given point to a given line segment.
Input: The given point, (g_x, g_y), and the given line segment connecting points (x_1, y_1), (x_2, y_2).
Output: The position value, p_v, as Definition 2.

1. Calculate $f = f(g_x, g_y)$ using Algorithm 3;
2. If $f > 0$, then $p_v = 1$;
3. If $f < 0$, then $p_v = -1$;
4. If $f = 0$ and $x_1 \neq x_2$, then
   (a) If $x_1 < g_x < x_2$ or $x_1 > g_x > x_2$, then $p_v = 2$;
   (b) Otherwise, $p_v = 0$;
5. If $f = 0$ and $x_1 = x_2$, then
   (a) If $y_1 < g_y < y_2$ or $y_1 > g_y > y_2$, then $p_v = 2$;
   (b) Otherwise, $p_v = 0$;
6. END.

All 72 subcases can be classified into five testing classes as follows. In the first two testing classes, intersection testing result can be obtained directly from the grid without further computations. Other three classes need calculating the position values of some points to get the result.

The first testing class is that two line segments do not intersect according to the grid, since two ending points of one line segment are on the same side of the other line segment. Figure 4a is an example of this testing class. We obtain that $\mathbf{p}_2$ and $\mathbf{p}_3$ are both above the segment $\overline{\mathbf{p}_1\mathbf{p}_4}$, which guarantees two given line segments do not intersect.

The second testing class is that two line segments intersect, as shown in Fig. 4b. From this figure, $\mathbf{p}_1$ and $\mathbf{p}_4$ are on the different sides of segment $\overline{\mathbf{p}_2\mathbf{p}_3}$, and $\mathbf{p}_2$ and $\mathbf{p}_3$ are also on the different sides of segment $\overline{\mathbf{p}_1\mathbf{p}_4}$. Thus, segments $\overline{\mathbf{p}_1\mathbf{p}_4}$ and $\overline{\mathbf{p}_2\mathbf{p}_3}$ have an intersection point which is not an ending point.

The third testing class is that the position values of three ending points can be obtained directly from the grid, and obtaining that of the other ending point needs to call Algorithm 4. For example, we can immediately obtain from the Fig. 4(c) that $\mathbf{p}_2$ and $\mathbf{p}_4$ are on the different sides of $\overline{\mathbf{p}_1\mathbf{p}_3}$, and the position value of $\mathbf{p}_1$ to $\overline{\mathbf{p}_2\mathbf{p}_4}$ is -1.

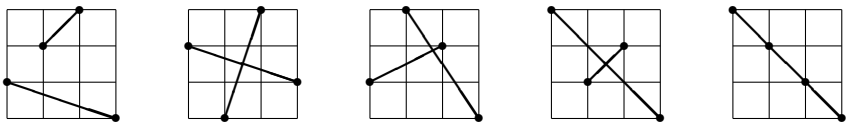

Fig. 4. Examples of the five testing classes: (**a**) Class 1, subcase (19,3); (**b**) Class 2, subcase (14,3); (**c**) Class 3, subcase (21,2); (**d**) Class 4, subcase (22,3); (**e**) Class 5, subcase (24,2)

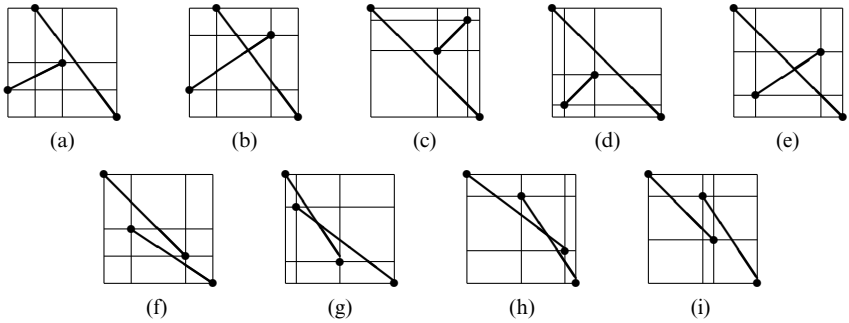

Fig. 5. Some examples of the latter three testing classes: (**a**, **b**) Class 3, Subcase (21,2); (**c**, **d**, **e**) Class 4, Subcase (22,3); (**f**, **g**, **h**, **i**) Class 5, Subcase (24,2)

However, the position value of $\mathbf{p}_3$ to $\overline{\mathbf{p}_2\mathbf{p}_4}$ may be -1 (see Fig. 5a), or 1 (see Fig. 5b). Therefore, Algorithm 4 should be called once to calculate this position value, i.e., only one dot product summation is needed.

The fourth testing class is that Algorithm 4 is needed to compute the position values of two ending points. As shown in Fig. 4d, $\mathbf{p}_1$ and $\mathbf{p}_4$ are on the different sides of $\overline{\mathbf{p}_2\mathbf{p}_3}$. So $\overline{\mathbf{p}_1\mathbf{p}_4}$ intersects $\overline{\mathbf{p}_2\mathbf{p}_3}$ only when the position value of $\mathbf{p}_2$ to $\overline{\mathbf{p}_1\mathbf{p}_4}$ is -1, and the position value of $\mathbf{p}_3$ to $\overline{\mathbf{p}_1\mathbf{p}_4}$ is 1, as shown in Fig. 5e. But it is possible that $\mathbf{p}_2$ and $\mathbf{p}_3$ are be on the same side of $\overline{\mathbf{p}_1\mathbf{p}_4}$: the position values of $\mathbf{p}_2$ and $\mathbf{p}_3$ to $\overline{\mathbf{p}_1\mathbf{p}_4}$ are both 1 (as Fig. 5c), or both -1 (as Fig. 5d). Thus, Algorithm 4 should be called twice to calculate the position values of $\mathbf{p}_2$ and $\mathbf{p}_3$, i.e., at most two dot product summations are needed.

The last testing class is illustrated by Fig. 4e, where it seems Algorithm 4 should be called four times to compute the position values of all the ending points, since none of them can be directly obtained from the grid. Four possible grids, corresponding to the grid Fig. 4e, are shown in Fig. 5f–i. In fact, only two position values of ending points are necessarily calculated, namely two dot product summations are needed (discussed in detail later).

Each subcase of the 72 has been analyzed and been mapped into one of those testing classes, as listed in Table 1. The testing class flag of the subcase (n_c, n_s) is just the n_s-th number of the item at the i-th row and j-th column of Table 1, where $i = \lceil \frac{n_c}{6} \rceil$ and $j = ((n_c - 1) \textbf{ mod } 6) + 1$. If the testing class flag of a subcase is x, then this subcase falls into the x-th testing class mentioned above. For example, the subcase (19,3) means the subcase whose case number is 19 and subcase number is 3. We obtain the item {1,3,1} at the fourth row and first column of Table 1. Thus, the class flag of this subcase is 1, the third number of the item {1,3,1}, which means this subcase falls into the first testing class mentioned above, i.e. two given line segments

Table 1. Testing class flag matrix

{1,5,4}	{1,3,3}	{1,1,4}	{1,3,1}	{1,1,3}	{1,3,1}
{1,3,3}	{1,1,2}	{1,1,3}	{1,3,1}	{1,1,2}	{1,3,1}
{1,3,1}	{1,1,2}	{1,3,1}	{1,1,3}	{1,1,2}	{1,3,3}
{1,3,1}	{1,1,3}	{1,3,1}	{1,1,4}	{1,3,3}	{1,5,4}

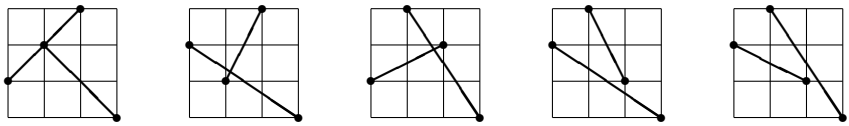

Fig. 6. Examples for subcases whose the testing class flag is 3: (**a**) subcase (19,2); (**b**) subcase (20,3); (**c**) subcase (21,2); (**d**) subcase (23,3); (**e**) subcase (23,2)

do not intersect. From the table, we find that 46 subcases (whose testing class flags are 1 or 2) can produce results directly, 20 subcases (whose testing class flags are 3) need one dot product summation, and only 6 subcases (whose testing class flags are 4 or 5) require at most two dot product summations.

Results of the first two testing classes can be obtained immediately, while the other three need further calculations which are as follows. When the testing class flag of a subcase is 3, Algorithm 4 should be called once to compute the position value of one ending point. Here, we should know the position value is calculated with which point, $\mathbf{p}_x$, to which line segment, $\overline{\mathbf{p}_y\mathbf{p}_z}$, and should record the corresponding position value for intersecting or not intersecting. Our method is to record $\mathbf{p}_x$ and $\overline{\mathbf{p}_y\mathbf{p}_z}$ as $\{x, y, z\}$, if and only if the position value of $\mathbf{p}_x$ to $\overline{\mathbf{p}_y\mathbf{p}_z}$ is 1 can deduce that two line segments intersect, and to record them as $\{-x, y, z\}$, if and only if the position value of $\mathbf{p}_x$ to $\overline{\mathbf{p}_y\mathbf{p}_z}$ is -1 can deduce that two line segments intersect. For example, in Fig. 6a, $\overline{\mathbf{p}_1\mathbf{p}_3}$ intersects $\overline{\mathbf{p}_2\mathbf{p}_4}$ if and only if the position value of $\mathbf{p}_2$ to $\overline{\mathbf{p}_1\mathbf{p}_3}$ is 1. Thus, we record {2,1,3} for this subcase. Similarly, we record {−2,1,4} for Fig. 6b, {3,2,4} for Fig. 6c, {−3,1,4} for Fig. 6d, and {3,2,4} for Fig. 6e.

Table 2 gives all the records in a matrix for subcases whose testing class flag is 3, where each cell of Table 2 has three records. For the testing class flag of a subcase (in Table 1) is not 3, the corresponding record (in Table 2) is marked with "—". Similar to the method of finding the testing class flag of a subcase in Table 1, the record of a subcase (n_c, n_s) whose testing class flag is 3 is just the n_s-th record of the item at the i-th row and j-th column of Table 2, where $i = \lceil \frac{n_c}{6} \rceil$ and $j = ((n_c - 1) \textbf{ mod } 6) + 1$. For example, the record for the subcase (23,3) is the third record of the cell at the fourth row and fifth column of Table 2, which is {−3, 1, 4}.

Table 2. Records of subcases whose testing class flag is 3

—	— {−3,2,4} {3,1,4}	—	— {−3,2,4} —	— — {2,1,4}	— {−2,1,3} —
— {2,1,3} {−2,1,4}	—	— — {−3,1,4}	— {−3,2,4} —	—	— {−2,1,3} —
— {2,1,3} —	—	— {3,2,4} —	— — {3,1,4}	—	— {−2,1,3} {2,1,4}
— {2,1,3} —	— — {−2,1,4}	— {3,2,4} —	—	— {3,2,4} {−3,1,4}	—

From Table 1, we find there are four subcases, (1,3), (3,3), (22,3), and (24,3), fall into the fourth testing class. After analyzing these four subcases, we obtain that $\overline{\mathbf{p}_1\mathbf{p}_4}$ intersects $\overline{\mathbf{p}_2\mathbf{p}_3}$ if and only if $\mathbf{p}_2$ and $\mathbf{p}_3$ are on the different sides of $\overline{\mathbf{p}_1\mathbf{p}_4}$. Moreover, if it is subcase (3,3) and the position value of $\mathbf{p}_2$ to $\overline{\mathbf{p}_1\mathbf{p}_4}$ is -1, or it is subcase (22,3) and the position value of $\mathbf{p}_2$ to $\overline{\mathbf{p}_1\mathbf{p}_4}$ is 1, then $\overline{\mathbf{p}_1\mathbf{p}_4}$ and $\overline{\mathbf{p}_2\mathbf{p}_3}$ do not intersect. These two judgments make the algorithm produce the result that two segments do not intersect without computing the position value of $\mathbf{p}_3$ when the condition is satisfied. Otherwise, calculating the position value of $\mathbf{p}_3$ is needed. Please see Algorithm 5, Step 8 for details.

When the testing class flag of a subcase is 5, every ending point does not have a constant position value. It seems that we should compute the position values of all ending points. But only 2 subcases , subcases (1,2) and (24,2), fall into this class, and each subcase only has four intersection appearances. Figure 5f–i gives all these intersection appearances for subcase (24,2), where the cases that two line segments intersect just at an ending point are omitted. We find $\overline{\mathbf{p}_1\mathbf{p}_3}$ intersects $\overline{\mathbf{p}_2\mathbf{p}_4}$ if and only if the position value of $\mathbf{p}_2$ to $\overline{\mathbf{p}_1\mathbf{p}_3}$ and that of $\mathbf{p}_3$ to $\overline{\mathbf{p}_2\mathbf{p}_4}$ are both 1 or -1. The same conclusion can be obtained in subcase (1,2). Therefore, Algorithm 4 is needed only twice, which means two dot product summations are required in this class. Please see Algorithm 5, Step 9 for details.

However, above analysis is based on the assumption that any two ending points are not on the same vertical or horizontal line. If three or four ending points are on the same vertical or horizontal line, then there exists only two possible results. The first one is two line segments intersect at one or two ending points. The other one is two line segments do not intersect. In these cases, we can obtain the result easily by comparing the coordinates of the ending points. When four ending points are on a horizontal line, if $\min(a_{1x}, a_{2x}) > \max(b_{1x}, b_{2x})$ or $\min(b_{1x}, b_{2x}) > \max(a_{1x}, a_{2x})$ is satisfied, then $\overline{\mathbf{a}_1\mathbf{a}_2}$ and $\overline{\mathbf{b}_1\mathbf{b}_2}$ do not intersect. Otherwise, they overlap, namely intersect at ending points. Similarly, we compare the y-coordinates of ending points when they are on a vertical line. When only three ending points are on the same vertical or horizontal line, two of them must be two ending points of a give line segment. Without loss, we assume only $\mathbf{a}_1$, $\mathbf{a}_2$, and $\mathbf{b}_1$ are on a horizontal line. We have $\overline{\mathbf{a}_1\mathbf{a}_2}$ intersects $\overline{\mathbf{b}_1\mathbf{b}_2}$ at $\mathbf{b}_1$ if and only if $a_{1x} < b_{1x} < a_{2x}$ or $a_{1x} > b_{1x} > a_{2x}$.

When at most two ending points are on the same vertical or horizontal line, the method above is robust as well. For our method is to map two give line segments onto a 3×3 grid, and each ending point on the grid can be moved vertically and horizontally between two neighboring borderlines. The testing class flag matrix, Table 1, is just constructed under this consideration. If two x-coordinates or y-coordinates are same, then the order of these two equal values is arbitrary. For example, if $b_{2y} < a_{1y} < a_{2y} = b_{1y}$, then the ascending ordering of these four y-coordinates can be $b_{2y}, a_{1y}, a_{2y}, b_{1y}$, or $b_{2y}, a_{1y}, b_{1y}, a_{2y}$. So the y'-coordinates (in grid coordinates system as shown in Fig. 1) of points $\mathbf{a}_2$ and $\mathbf{b}_1$ are 3 and 4, or 4 and 3, and either is right. Thus, when two ending points are on the same vertical or horizontal line, these two line segments can be mapped into at least 2 different subcases of the 72, but each subcase can generate the right result. Figure 7a shows $\mathbf{p}_2$ and $\mathbf{p}_3$ are on the

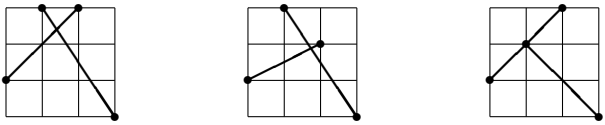

Fig. 7. Two ending points are on the same vertical or horizontal line

same horizontal line. This case falls into subcase (21,2) as Fig. 7b, or subcase (19,2) as Fig. 7c, since the y-coordinate ascending ordering of four ending points could be $\mathbf{p}_4\mathbf{p}_1\mathbf{p}_3\mathbf{p}_2$ (Fig. 7b) or $\mathbf{p}_4\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3$ (Fig. 7c) due to the same y-coordinate of $\mathbf{p}_2$ and $\mathbf{p}_3$. In these two subcases, the program needs to calculate the position value of the $\mathbf{p}_3$ or $\mathbf{p}_2$ to yield the result. However, this would not produce any error.

Then, with the analysis above, we give our algorithm for determining whether two line segments intersect as follows. And we use $PV(\mathbf{p}_x, \mathbf{p}_y, \mathbf{p}_z)$ to represent the position value of point $\mathbf{p}_x$ to segment $\overline{\mathbf{p}_y\mathbf{p}_z}$, calculated by Algorithm 4.

Algorithm 5: Determining whether two line segments intersect.
Input: two given line segments, $\overline{\mathbf{a}_1\mathbf{a}_2}$ and $\overline{\mathbf{b}_1\mathbf{b}_2}$, where $\mathbf{a}_1 = (a_{1x}, a_{1y})$, $\mathbf{a}_2 = (a_{2x}, a_{2y})$, $\mathbf{b}_1 = (b_{1x}, b_{1y})$, and $\mathbf{b}_2 = (b_{2x}, b_{2y})$.
Output: the value v_f, which is

$$\begin{cases} 0, & \text{if } \overline{\mathbf{a}_1\mathbf{a}_2} \text{ and } \overline{\mathbf{b}_1\mathbf{b}_2} \text{ do not intersect,} \\ 1, & \text{if they intersect but NOT at an ending point,} \\ 2, & \text{if they intersect just at ending points.} \end{cases}$$

1. If $\overline{\mathbf{a}_1\mathbf{a}_2}$ and $\overline{\mathbf{b}_1\mathbf{b}_2}$ share the same ending point, then let $v_f = 2$, and go to END;
2. If three or four ending points are on the same vertical or horizontal line, then compute the result by comparing the coordinates of the ending points as mentioned above, and go to END;
3. According to $a_{1x}, a_{1y}, a_{2x}, a_{2y}, b_{1x}, b_{1y}, b_{2x}, b_{2y}$, obtain the case number, n_c, and the subcase number, n_s;
4. According to (n_c, n_s), look up the testing class flag matrix (Table 1) to obtain the testing class flag, $flag$, of the subcase (n_c, n_s);
5. If $flag = 1$, then $v_f = 0$;
6. If $flag = 2$, then $v_f = 1$;
7. If $flag = 3$, then
   (a) According to (n_c, n_s), obtain the record, $\{x, y, z\}$, from Table 2, and let $v_1 = PV(\mathbf{p}_{|-x|}, \mathbf{p}_y, \mathbf{p}_z)$;
   (b) If $v_1 = 2$, then $v_f = 2$;
   (c) If $v_1 = 1$ and $x > 0$, or $v_1 = -1$ and $x < 0$, then $v_f = 1$;
   (d) Otherwise, $v_f = 0$;
8. If $flag = 4$, then
   (a) $v_1 = PV(\mathbf{p}_2, \mathbf{p}_1, \mathbf{p}_4)$;
   (b) If $v_1 = 2$, then $v_f = 2$;

(c) If $v_1 = 1$ and $n_c = 22$, or $v_1 = -1$ and $n_c = 3$, then $v_f = 0$;
(d) Otherwise,

(1) $v_2 = PV(\mathbf{p}_3, \mathbf{p}_1, \mathbf{p}_4)$;
(2) If $v_2 = 2$, then $v_f = 2$;
(3) If $v_1 = 1$ and $v_2 = -1$, or $v_1 = -1$ and $v_2 = 1$, then $v_f = 1$;
(4) Otherwise, $v_f = 0$;

9. If $flag = 5$, then

(a) $v_1 = PV(\mathbf{p}_2, \mathbf{p}_1, \mathbf{p}_3)$; $v_2 = PV(\mathbf{p}_3, \mathbf{p}_2, \mathbf{p}_4)$;
(b) If $v_1 = 2$ or $v_2 = 2$, then $v_f = 2$;
(c) If $v_1 = 1$ and $v_2 = 1$, or $v_1 = -1$ and $v_2 = -1$, then $v_f = 1$;
(d) Otherwise, $v_f = 0$;

10. END.

4. Examples and Results

Some examples are provided in this section. Figure 8 gives a model for line segment intersection testing. Three line segments $\overline{\mathbf{ab}}$, $\overline{\mathbf{bc}}$ and $\overline{\mathbf{de}}$ are constructed as follows to test the algorithms. Segments $\overline{\mathbf{ab}}$ and $\overline{\mathbf{bc}}$ share a common ending point **b**. The segment $\overline{\mathbf{de}}$ passes through the ending point **b**, and satisfies $\|\overline{\mathbf{be}}\|_2 = \|\overline{\mathbf{bd}}\|_2$. A vertical line passes through the point **b**, and points $\mathbf{d}'$ and $\mathbf{e}'$ are respectively two projection points of **d** and **e** on this vertical line, then we have $\|\overline{\mathbf{dd}'}\|_2 = \|\overline{\mathbf{ee}'}\|_2$ and $\|\overline{\mathbf{be}'}\|_2 = \|\overline{\mathbf{bd}'}\|_2$ (see Fig. 8). Algorithms are called to test whether $\overline{\mathbf{de}}$ intersects $\overline{\mathbf{ab}}$, and whether $\overline{\mathbf{de}}$ intersects $\overline{\mathbf{bc}}$. The coordinates of the ending points are given with IEEE double precision floating-point numbers. Here we gives two examples according to the testing model as shown in Fig. 8. In the rest of this paper, the method found in [6] is represented by "M_1", and our method Algorithm 5 is represented by "M_2".

M_1 requires that a twice higher precision floating-point arithmetic than the precision of the multipliers exists for exactly calculating the product of two floating-point numbers. However, in C/C++ language there does not exist a twice higher precision floating-point arithmetic than the double precision. Therefore, in M_1 the product of

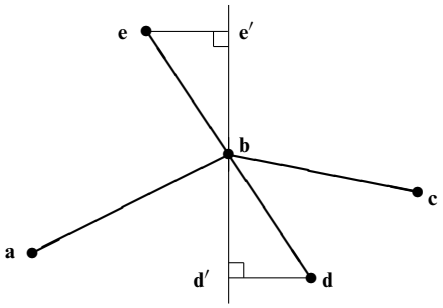

Fig. 8. A model for line segment intersection testing

two double precision floating-point numbers only can be computed by the double precision floating-point arithmetic. These two examples are provided as follows.

The positions of points, **a**, **b**, **c**, **d** and **e**, are given before the test and represented by IEEE754 *double* ($\beta = 2$, $t = 53$) floating-point numbers. The first example corresponding to the testing model above is **a**$(-2, -2)$, **b**$(1.625, 0)$, **c**$(4, 4)$, **d**$(1.125, -2^{52} - 1)$, and **e**$(2.125, 2^{52} + 1)$ (see Fig. 9a), where the coordinates of each point can be exactly represented by floating-point numbers, that is to say, each real number can be exactly converted into a computer floating-point number. The testing result is that M_2 tells $\overline{\mathbf{de}}$ intersects both $\overline{\mathbf{ab}}$ and $\overline{\mathbf{bc}}$ just at the point **b**. But, the result figured out by M_1 [6] is $\overline{\mathbf{de}}$ intersects $\overline{\mathbf{ab}}$ (not at **b**), and $\overline{\mathbf{de}}$ does not intersect $\overline{\mathbf{bc}}$.

Another example corresponding to the testing model is when $\mathbf{a} = (-1.1, -1.1)$, $\mathbf{b} = (1.1, 1.1)$, $\mathbf{c} = (2.2, -1.1)$, $\mathbf{d} = (1.09, -2.1)$, and $\mathbf{e} = (1.11, 4.3)$ (see Fig. 8b). Here, each real number cannot be expressed by IEEE 754 *double* exactly. The *double* value of each decimal number, which is just a truncation of its exact binary floating-point value, is listed in the right of Table 3. With these numbers, the result obtained by M_1 is $\overline{\mathbf{de}}$ intersects $\overline{\mathbf{ab}}$ (not at **b**), and $\overline{\mathbf{de}}$ does not intersect $\overline{\mathbf{bc}}$. On the contrary, M_2 tells $\overline{\mathbf{de}}$ intersects $\overline{\mathbf{bc}}$ (not at **b**), and $\overline{\mathbf{de}}$ does not intersect $\overline{\mathbf{ab}}$. In order to check which result is right, the numbers listed in the right of Table 3 are converted into big binary integers, and then they can be represented by Java's "class BigInteger", which can calculate the result of intersection exactly. The result obtained by this Java program is the same as the result of M_2.

The time complexity of above methods are compared as well. All the algorithms are implemented with Microsoft Visual C++ 6.0 on Pentium IV 1.7 GHz processor,

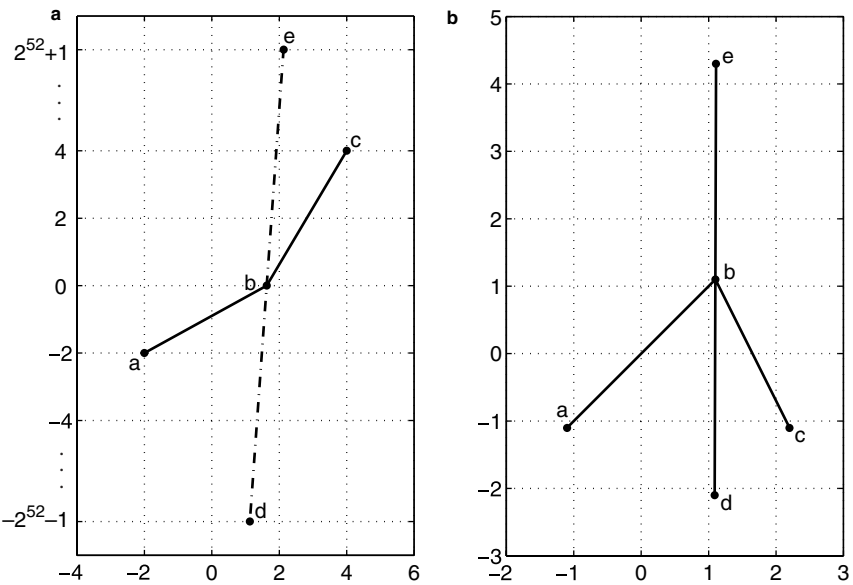

Fig. 9. Two examples where M_1 produces wrong results

Table 3. Decimal numbers represented by IEEE 754 *double*

Decimal	IEEE 754 *double*, $\beta = 2, t = 53$
1.09	$0.10001011100001010001111010111000010100011110101110001 \times 2^1$
1.1	$0.10001100110011001100110011001100110011001100110011001 \times 2^1$
1.11	$0.10001110000101000111101011100001010001111010111000011 \times 2^1$
2.1	$0.10000110011001100110011001100110011001100110011001100 \times 2^2$
2.2	$0.10001100110011001100110011001100110011001100110011001 \times 2^2$
4.3	$0.10001001100110011001100110011001100110011001100110011 \times 2^3$

and the software environment is Microsoft Windows 2000 Professional. The type of floating-point numbers used in algorithms is *double* ($\beta = 2, t = 53$), which conforms to IEEE 754 standard [2].

The floating-point summation method used in M_1 is ESSA [18], which can exactly calculate the sign of a finite sum. This algorithm has been applied in line segment intersection testing [6], and more applications of ESSA could be found in [19], [20], [22]. Algorithm 3 can be used in all the applications where ESSA is used, after a simplification. For the main task of Algorithm 3 is to add all the summands in the sets $\mathcal{P}$, $\mathcal{N}$ and $\mathcal{Q}$. If only to sum up n floating-point numbers, the only change we should make is to directly put n summands into the sets $\mathcal{P}$ or $\mathcal{N}$ according to their signs (see Step 2 in Algorithm 3), and delete Step 12 of Algorithm 3. After we get the whole sum whose error is less than $1ulp$, the sign of it is obtained. Here, we use "M_3" to represent the method that simplified from Algorithm 3 as described above.

Table 4 gives the running time values of M_3 and ESSA for calculating the sign of a sum, using ill-conditioned data and various data sizes, where ill-conditioned data is the data which have so heavy a cancellation that the sum is very closed to zero. And the running time curves corresponding to Table 4 are shown in Fig. 10. We find that M_3 requires about one-fifteenth of the running time used by ESSA, where line least square approximation is used. Moreover, M_3 can figure out not only the sign but also the value of the sum.

Another comparison made here is among methods M_1, M_2 and M_4 for line segment intersection testing, where M_4 is the method simplified from M_2. The difference

Table 4. Running time of calculating the sign of a sum by ESSA and M_3

Size of data	Running time (in seconds)	
	M_3	ESSA
100,000	0.028	0.190
500,000	0.147	1.724
1,000,000	0.294	3.343
1,500,000	0.439	5.778
2,000,000	0.586	7.890
2,500,000	0.739	10.889
3,000,000	0.880	13.573

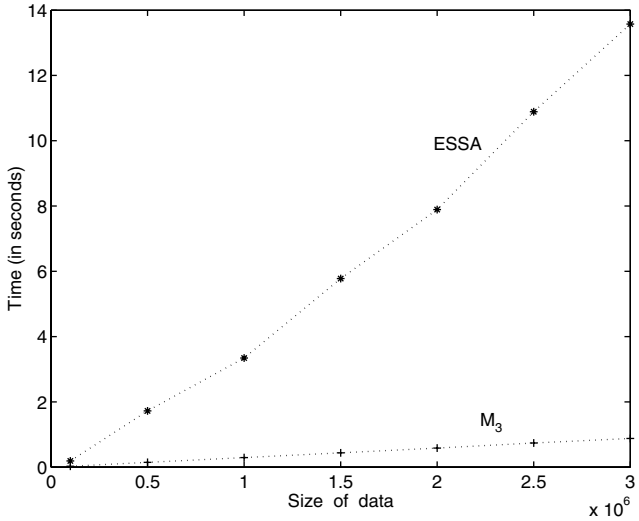

Fig. 10. Running time of calculating the sign of a sum by ESSA and M_3

between M_2 and M_4 is that M_4 directly uses M_3 to obtain the result of dot product summation, i.e., the error of floating-point multiplication is neglected. So, both M_1 and M_4 only guarantee the accuracy of floating-point summation in calculating dot product summations. In each test, eight coordinates of two line segments are randomly selected from an array, which is composed of 100 floating-point numbers ranges from 0.00 to 0.99, where the interval is 0.01. After 3,000,000 random tests, we find that M_2 always gets a correct result, however M_1 makes 1551 times of mistakes, and M_4 makes 581 times of mistakes. Running time values and corresponding curves are shown in Table 5 and Fig. 11. We find that M_2 requires less than one-fifth of the running time used by M_1, and M_4 needs no more than one-tenth of that by M_1, where line least square approximation is used. Furthermore, the worst case of all algorithms (as Fig. 5g) is tested. The result is that M_2 demands less than one-third of the running time used by M_1, and M_4 needs no more than one-seventh of that by M_1. This is because M_1 needs five dot product summations in this case, but M_2 and M_4 need only two.

Table 5. Running time of line segment intersection testing by M_1, M_2 and M_4

Testing	Running time (in seconds)		
times	M_1	M_2	M_4
100,000	1.902	0.320	0.180
500,000	9.503	1.632	0.911
1,000,000	18.997	3.254	1.832
1,500,000	28.480	4.887	2.743
2,000,000	37.994	6.529	3.655
2,500,000	47.528	8.141	4.576
3,000,000	57.011	9.774	5.497

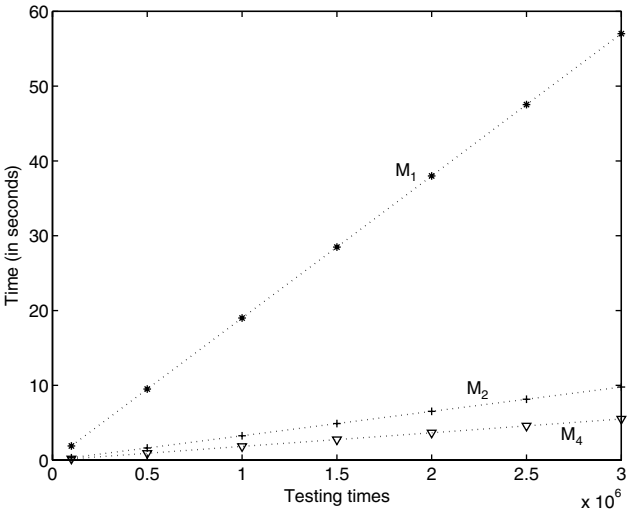

Fig. 11. Running time of line segment intersection testing by M_1, M_2 and M_4

5. Conclusions

In this paper, we present a new method, M_2, for line segment intersection testing. A line segment degenerates into a point is not considered. If two line segments share a common ending point, the result is returned right away. If three or four ending points are on the same vertical or horizontal line, we directly compare the coordinates of the ending points to obtain the intersection result without floating-point computations. Otherwise, the positions of four ending points and their connections are mapped onto a 3×3 grid, which falls into one of the five testing classes. In the first two classes, the result can be obtained immediately without further calculation. In the other three classes, the intersection testing is converted into calculating the position values of several points, which can be computed by our floating-point dot product summation algorithm, Algorithm 3. The floating-point arithmetic used in our method conforms to IEEE 754 standard. Algorithm 2 can exactly calculate the product of two floating-point numbers without the limitation that a twice higher precision floating-point arithmetic (with respect to the precision of multipliers) is needed. We have proved the error bound of Algorithm 3 is $1ulp$. The running time of M_2 is less than one-fifth of that of M_1. Furthermore, our dot product algorithm (Algorithm 3) can be simplified into a method M_3 for exactly calculating the sign of a sum of n floating-point numbers. M_3 requires less than one-fifteenth of the running time used by ESSA method [18].

Acknowledgements

The authors thank Prof. Jon Rokne for providing the source code of ESSA. The authors also thank Prof. Lyle Ramshaw and Dr. Huai-Ping Yang for providing the documents of two references in this paper. The authors appreciate the comments and suggestions of the anonymous reviewers. The second author was supported by a Foundation for the Author of National Excellent Doctoral Dissertation of PR China (200342). The research was supported by the National Science Foundation of China (60403047) and the Chinese 973 Program (2004CB719400).

References

[1] Anderson, I. J.: A distillation algorithm for floating-point summation. SIAM J. Sci. Comput. *20(5)*, 1797–1806 (1999).
[2] ANSI/IEEE (New York): IEEE Standard for Binary Floating-Point Arithmetic, Standard 754–1985, 1985.
[3] Demmel, J., Hida, Y.: Accurate and efficient floating point summation. SIAM J. Sci. Comput. *25(4)*, 1214–1248 (2003).
[4] Fortune, S., Wyk, C.: Efficient exact arithmetic for computational geometry. In: Proc. 9th Annual ACM Symp. on Computational Geometry, pp. 163–172 (1993).
[5] Fortune, S., Wyk, C.: Static analysis yields efficient exact integer arithmetic for computational geometry. ACM Trans. Graph. *15*(3), 223–248 (1996).
[6] Gavrilova, M., Rokne, J. G.: Reliable line segment intersection testing. Comput. Aided Des. *32*, 737–745 (2000).
[7] Higham, N. J.: The accuracy of floating point summation. SIAM J. Sci. Comput. *14(4)*, 783–799 (1993).
[8] Higham, N. J.: Accuracy and stability of numerical algorithms, 2nd edn. Philadelphia: SIAM 2002.
[9] Kahan, W.: Further remarks on reducing truncation error. Commun. ACM *8(1)*, 40 (1965).
[10] Knuth, D. E.: The art of computer programming, 3rd ed. vol 2: Seminumerical algorithms. Addison-Wesley 1998.
[11] Linz, P.: Accurate floating-point summation. Commun. ACM *13(6)*, 361–362 (1970).
[12] Masotti, G.: Floating-point numbers with error estimates. Comput. Aided Des. *25(9)*, 524–538 (1993).
[13] Milenkovic, V.: Double precision geometry: a general technique for calculating line and segment intersections using rounded arithmetic. In: Proc. 30th Annual Symp. on the Foundations of Computer Science, pp. 500–506 (1989).
[14] Moore, R. E.: Interval analysis. Englewood Cliffs, NJ: Prentice-Hall 1966.
[15] Priest, D. M.: Algorithms for arbitrary precision floating point arithmetic. In: Proc. 10th IEEE Symp. on Computer Arithmetic (Kornerup, P., Matula, D. W., eds.), pp. 132–143. Los Alamitos, CA: IEEE Computer Society Press 1991.
[16] Priest, D. M.: On properties of floating point arithmetics: numerical stability and the cost of accurate computations. PhD thesis, Mathematics Department, University of California, Berkeley, CA, 1992.
[17] Ramshaw, L.: CSL notebook entry: The braiding of floating point lines. unpublished note. Xerox PARC, October 1982.
[18] Ratschek, H., Rokne, J.: Exact computation of the sign of a finite sum. Appl. Math. Comput. *99*, 99–127 (1999).
[19] Ratschek, H., Rokne, J.: Exact and optimal convex hulls in 2D. Int. J. Comput. Geom. Appl. *10*, 109–129 (2000).
[20] Ratschek, H., Rokne, J.: How trivial are reliable box-plane intersections. Comput. Aided Des. *32*, 643–648 (2000).
[21] Ratschek, H., Rokne, J.: Geometric computations with interval and new robust methods: with applications in computer graphics, Gis and computational geometry. Chichester: Horwood Publishing 2003.
[22] Ratschek, H., Rokne, J., Leriger, M.: Robustness in GIS algorithm implementation with application to line simplification. Int. J. Geogr. Inf. Sci. *15*, 707–720 (2001).
[23] Zhu, Y.-K.: Yong, J.-H., Zheng, G.-Q.: A new distillation algorithm for floating-point summation. SIAM J. Sci. Comput. (2005) (accepted).

Yong-Kang Zhu
School of Software
Tsinghua University
Beijing
100084, China
e-mail: zhuyk@tsinghua.org.cn

Jun-Hai Yong
School of Software
Tsinghua University
Beijing
100084, China
e-mail: yongjh@tsinghua.edu.cn

Guo-Qin Zheng
School of Software
Tsinghua University
Beijing
100084, China
e-mail: gqzheng@tsinghua.edu.cn