Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.12785/amis/080607

Numerical Robustness in Geometric Computation: An Expository Summary

Gang Mei[1,2,*], *John C. Tipper*[2] *and Nengxiong Xu*[1]

[1] School of Engineering and Technology, China University of Geosciences (Beijing), 100083, Beijing, China
[2] Institute of Earth and Environmental Science, University of Freiburg, Albertstr.23B, D-79104, Freiburg, Germany

Received: 9 Oct. 2013, Revised: 7 Jan. 2014, Accepted: 8 Jan. 2014
Published online: 1 Nov. 2014

Abstract: This paper attempts to present an expository summary on the numerical non-robustness issues in geometric computation. We try to give the answers to two questions: (1) why numerical non-robustness issues occur in geometric computing, and (2) how to deal with them in practice. We first describe the theoretical causes of the problematic robustness behaviors, and then present several popular and practical solutions to the non-robustness problem. Note that these algorithm-specific or general solutions are only part of the existing efforts to attack the numerical robustness problem, but are quite useful in practical applications. Additionally, geometric examples and sample codes are included for illustrations.

Keywords: Computational geometry, Robustness, Geometric computation, Floating-point arithmetic

1 Introduction

In geometric computations, non-robustness issues always occur due to the numerical and the geometrical nature of involved algorithms; see reference [1] for several classic non-robustness examples. The causes of non-robustness behaviours can be mainly classified into two categories: numeric precision and geometrical degeneracy; thus, the robustness problems can also be divided correspondingly into two types according to the causes, i.e. the *numerical robustness* and the *geometrical robustness* [2].

Numeric precision (numeric stability) problems arise due to the inexact computer arithmetic: when carrying out various computations on computers, the numbers adopted are normally the fixed-precision floating-point numbers or even integers, rather than the expected exact real numbers that have arbitrary precisions in theory. Degeneracy refers to the special cases that are usually not considered during the generic implementations of algorithms and have to be treated in problem-specific ways. Both of the above two kinds of causes lead to incorrect answers.

There are many approaches presented in the literature that focus on the non-robustness problems in geometric computing; see [3–8] for the surveys. Those methods can be divided into two categories: the *arithmetic* and the *geometric* [9].

The arithmetic methods seek to address the problem of non-robustness in geometric computations by dealing with the numerical errors occurring due to fixed-precision arithmetic, which can be realized, for example, by using multi-precision arithmetic [10]. Within such methods, all the arithmetic operations are generally carried out on the algebraic quantities. Noticeably, the use of multi-precision arithmetic methods results in a high memory and running time overhead, as compared to the widely used IEEE-754 floating-point standard.

The geometric methods try to make an assurance that some geometric properties are preserved by the adopted algorithms. For instance, it is needed to guarantee that the output is a planar graph when calculating the Voronoi diagram of a set of points in two dimensions. Some of this kind of methods are the topology oriented approach [11], the consistency and topological approaches [12], the finite resolution geometry [13], and the approximate predicates and fat geometry [14].

Yap [8] gives an excellent survey on techniques for achieving robust geometric computation. Kettner et al. [1] provides graphic evidence of the troubles that arise when employing real arithmetic in geometric algorithms such as convex hull. Controlled perturbation [15] is a new method for implementing robust computation that has been drawn considerable attention.

* Corresponding author e-mail: gang.mei@geologie.uni-freiburg.de; gangmeiphd@gmail.com

Aiming at the problems of when to use the arbitrary-precision arithmetic and how to maintain the reasonable efficiency, Shewchuk [16], and Fortune and van Wyk [17] present excellent studies on the costs of using arbitrary-precision arithmetic while achieving complete robustness for geometric computation. A mathematical description of the arbitrary precision arithmetic is presented in [18].

Several software packages that are capable of robustly implementing geometric computations or dealing with the non-robustness problems in geometric computation have been developed, including CGAL [19], LEDA [20], CORE [21], and predicate.c [16]. Robust geometric computation can be achieved by simply using these packages.

CGAL and LEDA both provide very complete sets of robust geometric computations. LEDA is easier to learn and to work with, but CGAL is more comprehensive and publicly available. The CORE Library provides an API, which supports the Exact Geometric Computation (EGC) principle [22]to implement numerically robust algorithms; with small changes, any C/C++ program can use CORE to readily support four levels of accuracy. The small piece of C++ code, predicate.c, is developed by Shewchuk [16], which includes the robust implementation of four kinds of basic geometric tests (i.e., the 2D and 3D orientation, inCircle, and inSphere predicates).

As mentioned above, a very easily-used package is the CORE library. A newer version of this library, the CORE 2, is reported recently [23]. The common C++ code can be transferred into robust ones by simply using CORE library. However, when supporting the EGC principle by using the CORE library for robust geometric computation in the entire procedure of computing, the speed would be very slow. This is because geometric computations based on multi-precisions are much slower than those based on the IEEE-754 format floating-point numbers.

An effective solution to improve the efficiency when supporting the EGC paradigm is to accept the so-called "Lazy principle" [24], i.e., to carry out the most important geometric computations using multi-precisions arithmetic while performing the less important computations using standard-precision arithmetic. The basic idea behind the "Lazy principle" is that it is only needed to carry out the expensive multi-precisions computations when they are quite important and also need to be very accurate.

The objective of this paper is to (1) give an expository explanation to the origins of numerical non-robustness in geometric computation, and (2) present several practical approaches of dealing with the above problem. The main content will focus on answering two questions: the first is why numerical non-robustness occurs and the second is how to deal with the problematic robustness issues in geometric computation.

This paper is organized as follows. The theoretical causes of numerical non-robustness issues in geometric computation are described in Section 2. Several practical solutions of the numerical non-robustness problems are presented in Section 3. And finally in Section 4, several concluding remarks are given.

2 Theoretical Causes of Non-robustness

The algorithms of geometric computations are designed under an assumption that all computations are performed using exact real arithmetic. However, when implementing these algorithms on computers, inexact limited-precision arithmetic such as floating-point arithmetic is normally used in place of the exact arithmetic. And, floating point arithmetic is by nature inaccurate due to numerical errors; these numerical errors may lead to non-robust behaviours in the geometric computations. This section will describe the theoretical origins of the numerical non-robustness.

2.1 Floating-point Arithmetic

2.1.1 Real Numbers and Machine Numbers

In mathematics, the definition of real numbers is given as: a real number can be thought as a value that represents a quantity along an infinitely long line (i.e., the *number line* or the *real line*). The real numbers include all the rational numbers (such as integers and fractions) and the irrational numbers (such as $\sqrt{3}$ and π).

The set of all real numbers is infinite; however, only part of real numbers can be exactly represented in binary format on computers while carrying out computations. Those real numbers that cannot be directly represented are approximated by the nearest representable numbers. The real numbers that are directly and exactly represented on computers are called *machine representable numbers*.

The formal definition of representable numbers is as follows: Representable numbers are numbers capable of being represented exactly in a given binary number format [27]. The format could be either an integer or floating point numbers.

2.1.2 Fixed- and Floating-point Numbers

Generally, real numbers can be represented on computers with two representations: the *fixed-point numbers* and the *floating-point numbers*.

Fixed-point number representation is a data type for expressing a number that has a fixed number of digits after the radix point. The fixed-point numbers are equally spaced on the number line. Unlike fixed-point numbers, the decimal point of a floating-point number is not "fixed" but "float". The term *floating point* means the fact that the radix point can "float" and be placed in any position with respect to the significant digits of the real number. The floating of the decimal point is represented by expressing numbers in a format similar to scientific notation.

Compared to fixed-point numbers, for floating-point numbers the density of representable numbers spaced on the number line is no longer even: the space distance between representable numbers increases the farther away from zero a number is located.

Floating-point numbers allow a much larger range of values to be represented. The property of having larger range can make floating-point numbers more useful in computing than fixed-point numbers, especially in the case that the number become larger than can be expressed with a fixed-size exponent (this exactly means *overflow*). Similarly, larger range is quite useful when the number becomes smaller than can be represented with a fixed-size exponent (i.e., *underflow*).

It is clear that the data type, floating-point numbers, and its arithmetic do not have the same properties as that of real numbers and the exact arithmetic using real numbers. So far, there are many types of floating-point representations used on computers; but the most widely used floating-point representation format is that defined by the IEEE-754 standard [28].

2.1.3 The IEEE-754 Floating-point Formats

The IEEE-754 floating-point format introduced in 1985 is nowadays the most widely used floating-point standard. A new version was revised in 2008 [28], which specifies two basic and two extended binary floating-point formats. An excellent description of IEEE-754 floating-point numbers and arithmetic can be found in the literature [29].

The IEEE-754 standard floating-point numbers are composed of three basic fields: i.e., the *sign* bit, the *exponent* component, and the *mantissa* field. The sign has only one bit. The exponent base does not need to be stored for that it is implicit. The mantissa consists of the *fraction* and a hidden leading digit. There are 32 and 64 bits for the IEEE-754 single-precision numbers and the double-precision numbers, respectively.

The IEEE-754 single-precision numbers can represent numbers of absolute value in the range of approximately 10^{-38} to 10^{38}, with a precision of 6 to 9 decimal digits. The double-precision format can represent numbers of absolute value between approximately 10^{-308} and 10^{308}, with a precision of 15 to 17 decimal digits.

2.1.4 Round-off Error in Floating-point Arithmetic

Several types of computational errors inherently exist in floating-point arithmetic, including the conversion errors, overflow and underflow errors, round-off errors, digit-cancellation errors, and input errors [2]. These errors may produce incorrect results in both numeric and geometric computations. One of them is caused by the operation of rounding-off; and this kind of error source is the main one of all types of error causes.

Remark The errors in floating-point arithmetic are mainly due to "rounding-off".

When using floating-point numbers, "rounding" needs to be carried out in almost all arithmetic operations. The following are the most common three cases:

(1) For the representable numbers with a quite accurate precision, when taking this type of data into binary calculation on computers, they need to be represented with limited-precision (such as 32-bits or 64-bits) numbers, rather than arbitrary precision numbers, and thus have to be rounded to their corresponding nearest machine representable numbers.
(2) For the numbers that cannot be represented exactly by binary numbers, such as the irrational numbers π and $\sqrt{3}$, they need to be rounded off and represented by their nearest machine numbers. The error will certainly occur between the original numbers and their inexact approximated counterparts.
(3) For the intermediate data obtained during computing, they are expected to be more accurate than the input data, but this is impractical since that the precision for representing numbers is limited and the intermediate data must be also represented with the specific limited-precision.

The first and second cases state that some kinds of decimal numbers cannot be exactly represented in binary format on computers. For instance, in the decimal floating -point representation form, the number 0.1 can be exactly represented using a fixed number of digits; but, when this number is normalized and rounded off to the IEEE-754 single-precision format (24 bits), the representation of 0.1 is a little bigger than 0.1 [2].

The third case states that the intermediate data cannot be represented with precisions larger than that is allowed on computers. In alternative words, the intermediate data which supposed to be more accurate but in fact it cannot be more accurate. Consider multiplying two real numbers that represented with n bits, e.g., c=ab, in theory the result c needs higher precision to be represented than both a and b, but in practice, c must be rounded to n bits and thus will be inaccurate.

The first and the second cases are the situations for the input data during calculations; the third one is the type for the intermediate data. All the above three cases (perhaps include other cases that have not described here) are due to the requirement of representing data on computers.

2.2 Summary on the Causes of Non-robustness

a) The algorithms of various geometric computations are designed under the assumption that all computations are performed in *exact real numbers* and arithmetic.
b) Real numbers are exact but need to be represented on computers with machine representable numbers such as floating-point numbers.
c) Floating-point numbers and arithmetic are by nature inexact due to numeric errors such the round-off error.
d) Using inexact fixed-precision floating point arithmetic for the assumed exact real arithmetic leads to the non-robustness problem in geometric computation.

3 Practical Solutions of Non-robustness

The section will introduce several practical solutions of the numerical non-robustness. These algorithm-specific or general approaches are only part of the existing efforts to attack the numerical robustness problem, but quite useful in practical applications.

3.1 Comparing Floating-point Numbers

In computing, usually there are many situations that need to compare two floating-point numbers. And in geometric computations, comparison of floating-point values is also quite essential; for example, in some geometric predicates, it needs to check whether two vertices are the same, and thus the coordinates of target points have to be compared. The comparison seems to be very easy, but in fact it is not as easy as it may be thought.

In integer arithmetic, it is possible and reasonable to compare two values directly using the following routine `if(a == b) then {do something}`. This is the exact way people usually consider and expect. However, unlike the integers, floating-point numbers by nature are inexact or not accurate, and cannot be directly compared with equality.

Giving an exact real number, and after converting and representing it with a specific format of floating-point representation, e.g., the IEEE-754 double-precision, the represented corresponding floating-point number is not exactly identical to the original number in general cases, but will be slightly bigger or smaller. This is due to the rounding-off error explained in the previous section.

The following C++ code simply demonstrates that the floating-point numbers cannot be compared directly. In order to create values that are mathematical equivalent, the number `b` is directly specified, and the other number `a` is obtained according to `b` via a simple formula.

```
#include <iostream>
using namespace std;

int main(void)
{
    double a = 1.80 * (1.0 + 1.0 / 10.0);
    double b = 1.98;

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    if(a < b)  cout << "a < b" << endl;
    else if(a > b) cout << "a > b" << endl;
    else cout << "a == b" << endl;

    return 0;
}
```

In above code, the floating-point numbers `a` and `b` are expected to be exactly identical. But in fact, they are not. The running results under compiler GCC and VC++ 2010 are both as follows:

```
a = 1.98
b = 1.98
a > b
```

There are several methods of comparing the floating-point numbers; and the most commonly used is perhaps the epsilon approach which is based on a very small number names *epsilon* ε. In the epsilon approach, two floating-point numbers are thought to be almost equal if the absolute value of their difference is less than a given tiny *range* (also called *tolerance* or *error*). A paradigm called "epsilon geometry" which analyses the usage of *epsilon* in geometric computation can be found in [14].

The tolerance is calculated via some formulas based on the epsilon or directly be set as same as the epsilon. According to the methods of how to compute the tolerance / error, there are three types of comparisons in the epsilon approach: the *absolute*, *relative*, and *combinational*.

Note that the comparisons based on epsilon cannot be used as a complete solution of this problem of comparing floating-point numbers, but can be just used as a relatively simple and efficient solution. An excellent description of how to compare floating-point numbers can be available via a web page [30].

3.1.1 Absolute Tolerance Comparison

The comparison with an absolute error is very easy to understand: if the absolute value of the difference of a pair of floating-point numbers is less than (or not greater than) a specific tolerance, then the two floating-point numbers can be roughly thought as being equivalent. The implementation can be as:

```
if(fabs(a-b) <= epsilon)  { do something }
```

This is may be the simplest way to compare floating-point numbers, although it does not work correctly in many cases.

In this type of comparison, the tolerance is directly as the same as the epsilon value. When implementing the comparison in practice, a careful attention must be drawn to the consideration of how to determine the epsilon value properly. In theory, the epsilon can be as small as it can be, and also can be very large; but in practice, it is not. The selection of the epsilon will be introduced later.

3.1.2 Relative Tolerance Comparison

In the comparison with absolute errors, the tolerance is fixed and will never change no matter the numbers being compared is very large or very small; in other words, the numbers for comparing may have dramatic magnitudes, and in this case, using a permanent tolerance with a fixed magnitude seems to be improper.

For the reason mentioned above, the comparison with a relative error which considers the magnitudes of the target numbers is much more reasonable. This type of comparison was first described in the famous book *The Art of Computer Programming* authored by Donald E. Knuth. Several useful relations for comparing floating-point numbers are presented in the subsection 4.2.2 of the volume 2, *Seminumerical Algorithms* [31].

For two floating-point numbers, the absolute value of their difference must be calculated first; this is the same as that of the comparison with absolute error. However, the tolerance is no longer as same as the epsilon, but the multiplication with the minimum or maximum value of the two target numbers. This can be simply represented using the following code:

```
if(fabs(a-b) <= epsilon * min(a, b)) {
   do something
}
if(fabs(a-b) <= epsilon * max(a, b)) {
   do something
}
```

The first one can be thought strict equality, while the second is the approximate equality. Usually, the second one is adopted since that its condition is much easier to be met, and can work better in more cases.

3.1.3 Combinational Tolerance Comparison

Although the comparison with relative errors looks much better than the comparison with absolute errors, it still has drawbacks: if the numbers for comparing is less than 1.0, then the tolerance value `epsilon * max(a, b)` will certainly less than epsilon itself; furthermore, if the value `max(a, b)` is small, then `epsilon * max(a, b)` would be much smaller, and in this case, the comparison cannot work correctly in most cases because its condition is too strict to be met.

For the problem described above, the comparison with absolute errors is considered again: if the value `max(a, b)` < 1.0, then adopt the comparison with absolute error; otherwise, use the comparison with relative errors. Thus, the comparison with combinational errors is in fact the combination of the above two comparisons that are based on the absolute or relative errors. The following code illustrates the comparison with combinational errors.

```
if(fabs(a-b) <= epsilon *
   max(max(a, b), 1.0)) {
   do something
}
```

3.1.4 Considerations for Selecting epsilon

Theoretically, the epsilon can be any number as the users would like to choose. But obviously, a too large epsilon is not meaningful. Also, a too small one is not suggested to be used because it is possibly less than the *machine epsilon*.

It is recommended to determine the epsilon according to the following formula:

$$u \leqslant epsilon \leqslant 2^n * u \tag{1}$$

, in which u is the machine epsilon, and n is a given unsigned exponent ($n \geqslant 0$).

Note that the epsilon cannot be less than the machine epsilon due to the inherent property of machine epsilon. This property comes from its definition and will be explained in the closing later content. The upper bound of epsilon is $2^n * u$, which is a magnification of the machine epsilon. Why not magnify the u directly with n times, like this formula $epsilon \leqslant n * u$? This is because the machine epsilon is actually an index number with a base 2: for IEEE-754 single precision, the machine epsilon is 2^{-23}, and for double precision it is 2^{-52}. Thus, the relation to determine the range of epsilon (Equation.1) for single- and double- precision can be updated as:

$$2^{-23} \leqslant epsilon \leqslant 2^{n-23} \tag{2}$$

$$2^{-52} \leqslant epsilon \leqslant 2^{n-52} \tag{3}$$

In practical applications, the epsilon can be implemented in C++ as follows:

```
unsigned int n =  ;
#define floatEpsilon pow(2.0, n-23)
#define doubleEpsilon pow(2.0, n-52)
```

Machine epsilon An easily-understood definition of machine epsilon is described in [32]. Let the notation fl(x) denote the floating-point machine representable number that corresponds to the real number x, the machine epsilon is the smallest positive machine numbers ε such that

$$\mathrm{fl}(1+\varepsilon) > 1$$

In other word, machine epsilon ε is the distance from 1 to the next larger floating-point number. On computers that support the IEEE-754 floating-point arithmetic, the machine epsilon ε is 2^{-23} for single-precision, and 2^{-52} for double-precision.

Remark When comparing floating-point numbers, it is preferable that the relative test should be less-equal ($\leqslant$) rather than less-than ($<$). The relative test fails when both compared numbers are exactly zero. Thus, the following three checking procedures are not recommended.

```
// Absolute Tolerance Comparison
if(fabs(a-b) < epsilon)

// Relative Tolerance Comparison
if(fabs(a-b) < epsilon * max(a,b))

// Combinational Tolerance Comparison
if(fabs(a-b) < epsilon * max(max(a,b),1.0))
```

3.2 Several Geometric Predicates

Many geometric algorithms in computational geometry make decisions based on predicates (i.e., geometric tests). These predicates may produce incorrect answers because of the round-off errors in floating-point arithmetic. Robust implementations of the geometric predicates are essential for complex algorithms.

3.2.1 Coincidence

The coincidence predicate is to test whether various basic geometric primitives, e.g., points, segments, or polygons, coincide (or are identical to each other). The simplest and most commonly used coincidence predicate is to check whether two points are the same. In many algorithms of geometric computation, coincident points are not allowed: for example, the adjacent vertices of a polygon cannot be coincident when triangulating the simple polygons; and similar case appears when calculating the convex hull of a set of points.

In applications, there are two quite simple methods to compare a pair of points (p and q): the first is to calculate the distance between p and q, and if the distance is less than or equal to a very small positive value – epsilon, then p and q can be thought to be approximately coincident. This simple method can be represented by the following formula:

$$dist = \sqrt{(px-qx)^2+(py-qy)^2+(pz-qz)^2} \leqslant epsilon \tag{4}$$

Another simple approach is to directly compare the corresponding coordinates of two points. In this method, coordinates of points are tested separately by comparing the absolute value of the coordinates' difference with a specified tolerance – epsilon:

$$\begin{aligned} |px-qx| &\leqslant epsilon \\ |py-qy| &\leqslant epsilon \\ |pz-qz| &\leqslant epsilon \end{aligned} \tag{5}$$

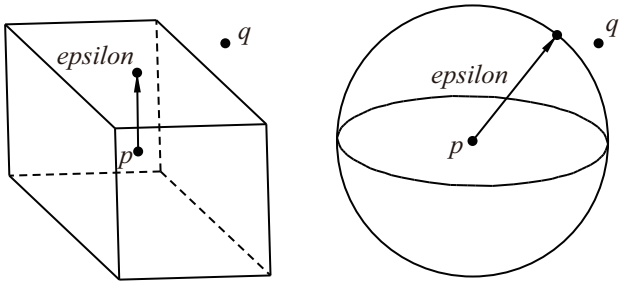


Fig. 1: Coincidence predicate for two points. Left: by difference. Right: by distance

Assuming one of the points, for example p, to be the centre point of a cube or a sphere (shown in Figure.1), the predicate of determining whether p and q coincide can be considered to check whether q locates inside of the cube or sphere. Three regions can be divided in Cartesian space by the cube and sphere, as shown in Figure.2.

Region 1: This region is exactly the space occupied by the sphere with radius = epsilon. Points that locate in this region can be considered to be *definitely coincident*.

Region 2: This region is the subtraction of the cube with the sphere. Points that locate outside the sphere but inside the cube can be considered to be *approximately coincident*.

Region 3: This region is the space that has not been occupied by the cube. Points that locate in this region can be considered to be *NOT coincident*.

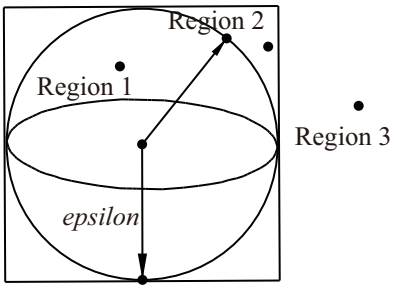


Fig. 2: Space regions divided by a cube and a sphere

Theoretically, the cases of point q locating in Region 1 or Region 3 usually appear since the volume of the first and the third regions are larger than that of the second region. This means that points locate inside the cube but outside the sphere is not so often as that in Region 1 or 3.

Obviously, the coincidence predicate represented by the Equation.5 has a little looser condition than that represented by the Equation.4. Points that are tested to be coincident according to the Equation.4 must be also coincident when using the Equation.5, but the opposite situation of above statement perhaps is no longer true. For example, set the coordinates of point q based on p:

$$\begin{aligned} qx &= px + epsilon \\ qy &= py + epsilon \\ qz &= pz + epsilon \end{aligned}$$

Then,

$$dist = \sqrt{epsilon^2 + epsilon^2 + epsilon^2} > epsilon.$$

Therefore, in this case, p and q are not coincident according to the Equation.4, but would be coincident according to Equation.5. This example and conclusion seem to be meaningless. Since that the points p and q are actually not coincident, and the predicate represented by the Equation.4 gives the correct answer. However, in

practice, those two points, p and q, are usually considered to be the same in order to avoid geometric degeneracy in the further computations, although the points p and q are exactly distinct.

This is because p and q are too close, and thus are difficult to be distinguished. Probably during a step of geometric computation, they are exactly tested to be NOT coincident according to a very strict condition such as that in the Equation.4, but in later geometric computations, very close points may cause geometric degeneracy; and thus this case should be avoided at the beginning of the entire procedure of geometric computation.

Remark In order to avoid the potential geometric degeneracy, very close, exactly distinct points should be considered to be coincident, and then must be merged to have the same and shared coordinates.

As a summary, the approach of testing whether two points coincide is recommended to use the way expressed in the Equation.5. And the effective C++ code is listed in the following Listing.1.

```
#include <cmath>
#include <limits>
#include <algorithm>

#define eps
numeric_limits<double>::epsilon()

bool isEqual(double a, double b)
{
    if(fabs(a-b) <=  eps*max(max(a,b),1.0))
        return true;
    else return false;
}

bool isSame(double p[3], double q[3])
{
    return isEqual(p[0], q[0], eps) &&
           isEqual(p[1], q[1], eps) &&
           isEqual(p[2], q[2], eps) ;
}
```

Listing 1: Coincidence predicate for points

3.2.2 Orientation

In 2D, the orientation predicate is to test whether a point locates in the left or right side of a line, or on the line (Figure.3(a)); normally, the directed line is defined by two points. In 3D, this predicate is extended to determine the position of a point with relative to an orientated plane defined by three points (see Figure.3(b)), i.e., to find out whether a point is above, on, or below a plane.

The orientation predicate is commonly used and quite useful since many important geometric algorithms, e.g., finding convex hulls, Delaunay triangulating, and polygon / polyhedron intersecting, have to call this predicate.

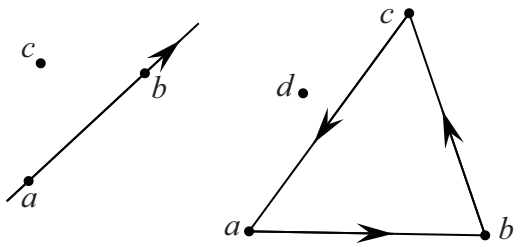
The orientation predicates (denoted as the procedures *orient*2*D* and *orient*3*D*) can be mathematically interpreted by the following formulas:

$$orient2D(a,b,c) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} \tag{6}$$

$$orient3D(a,b,c,d) = \begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix} \tag{7}$$

Determination of the position of point c with respect to the directed line ab can be directly obtained according to the calculation result of the determinant in Equation.6. However, there is no need to calculate the magnitude of the determinant but only to know the sign: if return a positive value, it means the point c locates in the left side of the directed line ab, while zero and the negative indicate c is exactly on and in the right side of ab, respectively.

Similar to the predicate *orient*2*D*, the extended version – *orient*3*D*, can give the position of point d with relative to the orientated plane defined by a, b and c according to the sign of determinant in Equation.7. The positive, negative, and zero values indicate the point d lie above, below and on the plane, respectively.



(a) 2D (b) 3D

Fig. 3: The orientation predicates in 2D and 3D

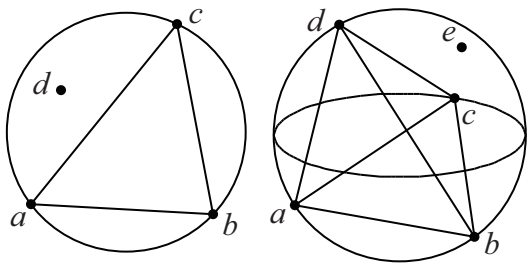
The idea behind the orientation predicates is easy to understand. However, these predicates may give incorrect answers in geometric computation when adopting floating -point arithmetic, and lead the entire geometric algorithm crashing or producing strange results. The above problem certainly needs to be avoided.

Shewchuk [16, 33] has developed quite robust and fast four geometric predicates, i.e., the orientation (described here) and the inCircle (described later) tests in 2D and 3D, based on adaptive precision floating-point arithmetic. The C implementation of above four predicates is available at http://www.cs.cmu.edu/~quake/robust.html.

3.2.3 InCircle / InSphere

Given three non-linear points, a circle can be defined by them while all points are required to locate on the circle. In other words, the circle is the circumcircle of a triangle whose vertices are the given three points. Similarly, a sphere can be formed based on four non-planar points of which no pair of points coincides.

The inCircle predicate is to test whether a point is inside, outside or on the above mentioned circle (shown in Figure.4(a)), while the inSphere predicate, which can be thought as the extended version of the inCircle predicate, is to determine whether a point locate inside the sphere defined by four points, as shown in Figure.4(b).



(a) inCircle (b) inSphere

Fig. 4: The inCircle and inSphere predicates

Also, the inCircle / inSphere predicate (denoted as the procedure *inCircle* and *inSphere*) can be fully interpreted by mathematical equations. Relative position of points with respect to the circle or sphere can be determined according to sign: the positive, negative, and zero values indicate the point d or e lie inside, on, and outside the circle or sphere, respectively.

$$inCircle(a,b,c,d) = \begin{vmatrix} a_x & a_y & a_x^2+a_y^2 & 1 \\ b_x & b_y & b_x^2+b_y^2 & 1 \\ c_x & c_y & c_x^2+c_y^2 & 1 \\ d_x & d_y & d_x^2+d_y^2 & 1 \end{vmatrix} \quad (8)$$

$$inSphere(a,b,c,d,e) = \begin{vmatrix} a_x & a_y & a_z & a_x^2+a_y^2+a_z^2 & 1 \\ b_x & b_y & b_z & b_x^2+b_y^2+b_z^2 & 1 \\ c_x & c_y & c_z & c_x^2+c_y^2+c_z^2 & 1 \\ d_x & d_y & d_z & d_x^2+d_y^2+d_z^2 & 1 \\ e_x & e_y & e_z & e_x^2+e_y^2+e_z^2 & 1 \end{vmatrix} \quad (9)$$

Similar to the two orientation predicates described in previous section, the inCircle and inSphere predicates may also give incorrect answers due to numeric errors such as the rounding-off error in floating-point arithmetic. For these predicates, Shewchuk [16, 33] also developed the fast and robust implementations based on adaptive precision floating-point arithmetic.

3.3 Exact Geometric Computation (EGC)

Since about 1990, an approach named Exact Geometric Computation (EGC) has emerged [22], and soon become one of the most successful options for achieving robust geometric computation. Major geometric computation libraries such as CGAL [19, 25] and LEDA [20, 26] are designed on the basis of supporting the EGC principle.

3.3.1 What is EGC and why EGC ?

The term "Exact Geometric Computation (EGC)" firstly advocated by Yap [22] is the preferred name for the general method of "exact computation", which identifies the goal of determining geometric relations exactly [8]. In [34], the prescription of the EGC approach is stated as: *it is to ensure that all branches for a computation path are correct*. Several key techniques of the EGC approach were introduced in Li's thesis [35]. Li also presents a survey on the recent developments in EGC research in three key aspects: constructive zero bounds, approximate expression evaluation and numerical filters [36].

Geometric computation not only involves numerical computation but also includes combinatorial computation. This can be represented with the following formulation:

Geometric = Numeric + Combinatorial.

The *numeric* part is exactly the numerical formats (rational, irrational, fixed- or arbitrary- precision floating point numbers) to represent various geometric primitives, e.g., the parameters of a line or plane, the coordinates of a point. The *combinatorial* (sometimes also called *discrete* or *topological*) part characterizes the discrete relations between geometric objects, e.g., whether a point is inside a polyhedron, the incident triangular elements of a node in a triangulated mesh.

Geometric algorithms focus on the combinatorial part by determining the discrete relations among geometric objects based on various geometric predicates such as orientation or intersection tests. Non-robustness issues appear because of incorrect determinations that due to numerical errors in the computation.

As introduced above, the prescription of the EGC is to *ensure that all branches for a computation path are correct* [34]. If all geometric predicates can be evaluated correctly, then the correctness of the combinatorial part of geometric algorithms will be guaranteed, and finally the robustness of the algorithms can be ensured.

Under the EGC paradigm, the input is assumed to be numerically exact. Exact geometry cannot be guaranteed if the input is not exact. For the inexact input, it can be processed by either "cleaning-up" or "formulating" [8]. Calculating based on the actual or assumed valid and exact inputs, the EGC approach can effectively ensure the correctness of the combinatorial part of various geometric algorithms.

EGC numbers. The term EGC numbers are accepted to refer to any number type that used to support the EGC calculations [36]. The arithmetic operations based on EGC numbers can ensure any specified precision. Users of EGC libraries can simply develop their robust implementations via building the codes on these EGC numbers.

The EGC numbers in CORE Library are named `Expr` which derives from the package Real/Expr [37]. The EGC numbers in LEDA [20] are `LEDA_real` [38, 39], while in CGAL [19], rational EGC numbers are provided, but it also adopts either the numbers from the CORE Library or `LEDA_real`.

To sum up, the EGC approach can be applied directly to many geometric problems without requiring special treatments specific to individual algorithms. EGC is a general framework to solve the numerical non-exactness problems in geometric computing. The drawback of the EGC approach is that the running cost on exact arithmetic under the EGC paradigm is much higher than that of the standard floating-point arithmetic.

3.3.2 How to support EGC ?

As described above, implementing geometric algorithms under the EGC paradigm is so far the best option to solve the robustness problem in geometric computation. By supporting the EGC principle, computational geometers can concentrate on developing good algorithms, and do not need to worry about non-robustness issues caused by numerical error. However, the geometric degeneracy, e.g., collinear three points, still exist under the EGC paradigm, and need to be treated specifically.

How to support the EGC principle? The easiest way to take advantage of existing package or library such as CGAL [19], LEDA [20] or CORE [21]. A short summary about the EGC mechanism in above libraries can be found in [39]; and the so-called *lazy* principle specially developed in CGAL is described in [24].

CGAL and LEDA are large libraries for almost all issues involved in geometric computation, which are both designed according to the EGC principle. Users can implement their robust geometric applications within the framework of CGAL or LEDA. The CORE library is an API developed to convert conventional C/C++ codes to robust ones by supporting the EGC principle.

The following example presents a test to demonstrate the effectiveness of the EGC principle. The centre of gravity of a triangle can be easily calculated according the coordinates of three vertices. The three results calculated according to different sequence of vertices, i.e., *abc*, *bca* or *cab*, are definitely the same in theory. However, this is not true in practice because of the numerical errors in floating-point arithmetic. The expected unique centre of gravity of a triangle is in fact a set of random points that locate closely around the expected position, as shown in Figure.5.

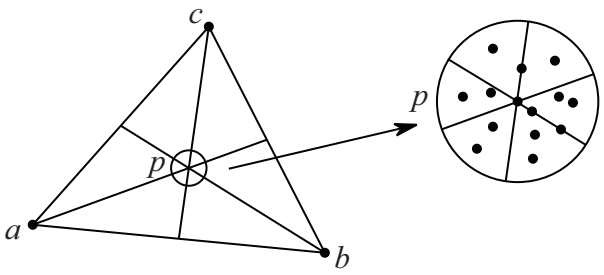


Fig. 5: Centre of gravity of a triangle

In the code Listing.2, centres calculated according to different sequences of vertices are tested whether they are coincident: without supporting the EGC, about 4300 tests fail when calculating 10000 times; in contrast, all tests are correct when supporting the EGC. This test illustrates the effectiveness of the EGC principle.

4 Concluding Remarks

The objective of this paper is to present an expository summary on the numerical non-robustness behaviours in geometric computation. Focusing on two questions (i.e., why numerical robustness problems arise and how to deal with them), we have given an expository explanation to the theoretical origins of numerical non-robustness, and presented several practical algorithm-specific or general approaches of dealing with those numerical robustness problems. Several geometric examples and sample codes are included for illustrations.

It is clearly stated that the numerical non-robustness issues arise mainly due to the round-off errors in the fixed precisions floating-point arithmetic. The numerical errors occur due to replacing the assumed exact real arithmetic with adopted inexact floating-point arithmetic, and thus lead most problematic robustness behaviours in geometric computations.

A general solution, the Exact Geometric Computation (EGC) approach, can very successfully deal with the issue of non-robustness in geometric computations. The EGC approach guarantees that all the geometric predicates are determined correctly thereby ensuring the correctness of the computed combinatorial structure and hence the robustness of the algorithm.

Although the EGC approach can be used to handle the numerical non-robustness issues effectively, an obvious drawback is its inefficiency. Another potential problem is the effectiveness of the EGC approach in the case that parallel geometric computations are performed on GPUs under heterogeneous models such as CUDA or OpenCL. More efforts besides the *Lazy* principle need to be made in order to improve the efficiency and effectiveness of the EGC approach.

```
#ifndef CORE_LEVEL
#   define CORE_LEVEL 3
//  define CORE_LEVEL 1
#endif
#include "CORE/CORE.h"

bool isSame(double p[2], double q[2])
{
    double dist = sqrt(pow(p[0]-q[0], 2.0)
        + pow(p[1]-q[1], 2.0));
    if(dist < 1.0e-15)  return true;
    else return false;
}

double center(double a, double b, double c)
{
    double t = (a + b) / 2.0;
    return (c - t) / 3.0 + t;
}

int main(void)
{
    int i, j, Correct = 0;
    double a[2], b[2], c[2]; // Vertices
    double pa[2], pb[2], pc[2]; // Centers

    // Seed for rand()
    srand((unsigned)time(NULL));

    for(i = 0; i < 10000; i++) {
        for(j = 0; j < 2; j++) {
            a[j] = rand();
            b[j] = rand();
            c[j] = rand();
        }

        for(j = 0; j < 2; j++) {
            pa[j] = center(a[j],b[j],c[j]);
            pb[j] = center(b[j],c[j],a[j]);
            pc[j] = center(c[j],a[j],b[j]);
        }

        if(isSame(pa, pb) &&
           isSame(pb, pc) &&
           isSame(pc, pa) ){
            // Number of correct tests
            Correct++;
        }
    }

    std::cout <<"Correct = " << Correct;

    return 0;
}
```

Listing 2: A sample code of supporting EGC

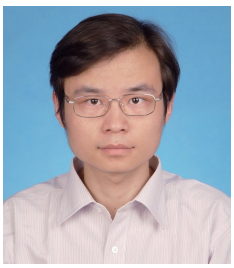
Acknowledgement

The first author would like to acknowledge the PhD scholarship provided by the China Scholarship Council (CSC) to support his studying in Germany. The authors are grateful to the anonymous referee for careful checking and helpful comments that improved this paper.

References

[1] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap, Computational Geometry, **40**, 61–78 (2008).
[2] C. Ericson, Real–time Collision Detection, Morgan Kaufmann, (2004).
[3] C. M. Hoffmann, J. E. Hopcroft, and M. S. Karasick, Towards implementing robust geometric computations, in Proceedings of the fourth annual symposium on Computational geometry, 106–117 (1988).
[4] S. Fortune, Robustness issues in geometric algorithms, Applied Computational Geometry Towards Geometric Engineering, 9–14 (1996).
[5] F. J. Santisteve, Robust Geometric Computation (RGC): State of the Art, Universitat Politcnica de Catalunya. Departament de Llenguatges i Sistemes Informtics, (1999).
[6] S. Schirra, in: J.-R. Sack, J. Urrutia (Eds.), Handbook of Computational Geometry, North-Holland, Amsterdam, Chapter 14, 597–632, (2000).
[7] C. M. Hoffmann, Robustness in Geometric Computations, Journal of Computing and Information Science in Engineering, **1**, 143–155 (2001).
[8] C. K.Yap, in: J. E. Goodman, J. O'Rourke (Eds.), Handbook of Discrete and Computational Geometry, Chapman & Hall/CRC, Boca Raton, FL, Chapter 41, 927–952 (2004).
[9] C. K. Yap and V. Sharma, Robust Geometric Computation, in Encyclopedia of Algorithms, ed: Springer, 788–790 (2008).
[10] P. Gowland and D. Lester, A survey of exact arithmetic implementations, in Computability and Complexity in Analysis, ed: Springer, 30–47 (2001).
[11] K. Sugihara, M. Iri, H. Inagaki, and T. Imai, Topology-oriented implementationan approach to robust geometric algorithms, Algorithmica, **27**, 5–20 (2000).
[12] S. Fortune, Stable maintenance of point set triangulations in two dimensions, in Foundations of Computer Science, 1989., 30th Annual Symposium on, 494–499 (1989).
[13] D. H. Greene and F. F. Yao, Finite-resolution computational geometry, in Foundations of Computer Science, 1986., 27th Annual Symposium on, 143–152 (1986).
[14] D. Salesin, J. Stolfi, and L. Guibas, Epsilon geometry: building robust algorithms from imprecise computations, in Proceedings of the fifth annual symposium on Computational geometry, 208–217 (1989).
[15] K. Mehlhorn, R. Osbild, and M. Sagraloff, Reliable and efficient computational geometry via controlled perturbation, in Automata, Languages and Programming, ed: Springer, 299–310 (2006).
[16] J. R. Shewchuk, Adaptive precision floating-point arithmetic and fast robust geometric predicates, Discrete & Computational Geometry, **18**, 305–363 (1997).

[17] S. Fortune and C. J. V. Wyk, Efficient exact arithmetic for computational geometry, presented at the Proceedings of the ninth annual symposium on Computational geometry, San Diego, California, USA, (1993).
[18] V. Mnissier-Morain, Arbitrary precision real arithmetic: design and algorithms, Journal of Logic and Algebraic Programming, **64**, 13–39 (2005).
[19] CGAL, Computational Geometry Algorithms Library, http://www.cgal.org, (2013).
[20] LEDA, Library for Efficient Data Types and Algorithms, http://www.mpi-inf.mpg.de/LEDA, (2013).
[21] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap, in: Proceedings of the fifteenth annual symposium on Computational geometry, ACM, 351–359 (1999).
[22] C. K. Yap, Computational Geometry, **7**, 3–23 (1997). Invited talk in the proceedings of the 5th Canadian Conference on Computational Geometry, University of Waterloo, August 5-9, (1993).
[23] J. Yu, C. Yap, Z. Du, S. Pion, and H. Brnnimann, The Design of Core 2: A Library for Exact Numeric Computation in Geometry and Algebra, in Mathematical Software ICMS 2010, K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, Eds., ed: Springer Berlin Heidelberg, **6327**, 121–141 (2010).
[24] S. Pion and A. Fabri, A generic lazy evaluation scheme for exact geometric computations, Science of Computer Programming, **76**, 307–323 (2011).
[25] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr, On the Design of CGAL, the Computational Geometry Algorithms Library, Technical Report RR-3407, INRIA, (1998).
[26] K. Mehlhorn and S. Näher, LEDA: A Platform for Combinatorial and Geometric Computing, Cambridge University Press, (1999).
[27] D. Kirk and W. W. Hwu, Programming Massively Parallel Processors: A Hands–On Approach, Morgan Kaufmann, (2010).
[28] IEEE, IEEE Std 754-2008, 1–58 (2008).
[29] D. Goldberg, ACM Computing Surveys, **23**, 5–48 (1991).
[30] http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm, (2013).
[31] D. E. Knuth, The Art of Computer Programming, Volume 2 (3rd ed.): Seminumerical Algorithms, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, (1997).
[32] E. Cheney and D. Kincaid, Numerical Mathematics and Computing, Brooks/Cole Publishing Company, (2012).
[33] J. Shewchuk, in: Proceedings of the twelfth annual symposium on Computational geometry, ACM, 141–150 (1996).
[34] K. Mehlhorn and C. Yap, Robust Geometric Computation, http://cs.nyu.edu/yap/book/egc/, (2004).
[35] C. Li, Exact Geometric Computation: Theory and Applications, Ph.D. thesis, New York University, (2001).
[36] C. Li, S. Pion, and C. K. Yap, Journal of Logic and Algebraic Programming, **64**, 85–111 (2005).
[37] J. Yu, Exact Arithmetic Solid Modeling, Ph.D. thesis, Purdue University, Department of Computer Sciences, (1992).
[38] C. Burnikel, J. Könemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig, in: Proceedings of the eleventh annual symposium on Computational geometry, ACM, 418–419 (1995).
[39] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra, in: Proceedings of the fifteenth annual symposium on Computational geometry, ACM, 341–350 (1999).



Gang Mei is expected to receive his Ph.D degree in 2014 from the University of Freiburg in Germany. He has obtained both bachelor and master degrees from China University of Geosciences. His main research interests are in the areas of numerical simulation and computational modeling, including mesh generation, GPU-computing (CUDA), computational geometry, geological modeling. He has published several research articles in journals and scientific conferences.

John Tipper is Professor of Historical Geology, Paleontology and Sedimentology at the University of Freiburg. His research interests are principally in theoretical and quantitative aspects of stratigraphy, and in the modelling of geological systems.

Nengxiong Xu is a full Professor at China University of Geosciences in Beijing. He obtained his Ph.D degree in geotechnical engineering from the China University of Mining and Technology in June, 2002. His main research interests are in the fields of rock structure and mechanics, geotechnical engineering, 3D geological modeling, and numerical simulation.