NOLTA, IEICE

Invited Paper

Robust geometric computation based on the principle of independence

Kokichi Sugihara [1 a)]

[1] *Meiji Institute for Advanced Study of Mathematical Sciences, Meiji University*
1-1-1 Higashimita, Tamaku, Kawasaki 214-8571, Japan

[a)] *kokichis@isc.meiji.ac.jp*

Abstract: We present an approach to robust geometric algorithms, which we call the principle of independence. In this approach, we distinguish between independent judgments and dependent judgments, and use numerical computation only for independent judgments. The result of judgments is always consistent and hence algorithms behave stably even in the presence of large numerical errors. The basic idea of this principle is described with three examples.

Key Words: robustness, geometric algorithm, principle of independence, topology-based approach, topological consistency

1. Introduction

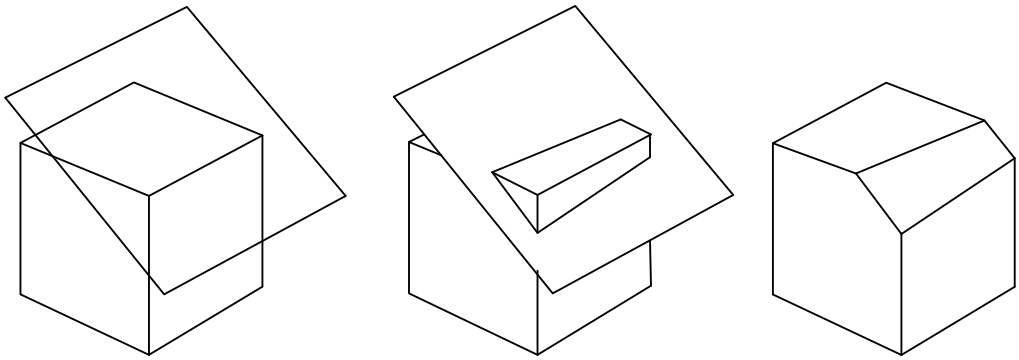
Naively implemented geometric software is usually fragile in the sense that it often fails because of inconsistencies caused by numerical errors. In many computations in other fields, the quality of the result is continuous in the errors arising in the course of computation; larger errors simply increase the distance between the result and the true answer. However, the situation is quite different in geometric computation.

Geometric computation usually contains geometric tests for deciding topological structures; for example, tests of whether a point is in a circle or whether two line segments intersect [1, 2]. The answers to these tests are either "YES" or "NO". Numerical errors often give incorrect answers, giving "YES" when it should be "NO" and giving "NO" when it should be "YES". We may therefore encounter a situation that can never happen in Euclidean space, and consequently the program fails. Even a very small error can make software fail. Therefore, making geometric algorithms robust against numerical errors is one of the most fundamental problems for practical applications of the theory of computational geometry.

To overcome this difficulty, many approaches have been proposed. One class of approach is based on exact computation achieved by multiple-precision arithmetic. Geometric problems are usually represented by finite-precision data, and hence any geometric test can be judged correctly if we use very high but still finite precision and thus can achieve robustness. This fact was recognized more than 20 years ago [3–5], and this approach was taken by many practical programs such as LEDA [6], CGAL [7] and Yap's exact computation platform [8].

However, exact computation requires high precision and hence is time consuming. Acceleration methods are therefore also employed; a typical method is floating-point filtering, in which geometric

[Received: June 9, 2010. Revised: July 20, 2010. Published: January 1, 2011.]
[DOI: 10.1587/nolta.2.32]
Nonlinear Theory and Its Applications, IEICE, vol. 2, no. 1, pp. 32−42



(a) (b) (c)

Fig. 1. Cutting a polyhedron by a plane.

tests are first done by floating-point arithmetic together with error analysis and switched to exact computation only when the results are not reliable [9, 10].

Another problem in exact computation is degeneracy. In exact computation, we can recognize degenerate situations, and hence we should incorporate exception processing in our programs for all possible cases of degeneracy. To overcome this difficulty we can use a symbolic perturbation technique, by which all degenerate cases can be eliminated automatically, and hence, general-case software can handle degenerate inputs [11, 12].

Another class of approach is based on floating-point computation. This class is subdivided into three subclasses: the three-value logic approach, the floating-point exact computation approach, and the topology-based approach.

In the three-value logic approach, the answer to every geometric test is classified into either "YES", "NO" or "UNKNOWN" according to an error analysis that accompanies the test. If the answer is "YES" or "NO", we follow the procedures given by theoretical algorithms; otherwise, we do something else, but this is case-by-case and the software design is very complicated and difficult [13–15].

In floating-point exact computation, exact answers for geometric tests are obtained by floating-point computation only [16, 17]. Hence, the programs obtained by this approach are portable in the sense that they can be used in any computational environment as long as floating-point arithmetic is available.

The topology-based approach [18–20], on the other hand, does not aim at exact answers to geometric tests; instead, we aim at guaranteeing topological consistency of the answers in the sense that even if the answers are incorrect because of numerical errors, they do not contradict each other.

In this approach, we try to divide the set of all judgments into independent ones and dependent ones, and use numerical computation only for the independent judgments; for the other judgments, we adopt the logical consequences obtained from the judgments of the former group.

We review the basic idea of the topology-based approach, and show three examples of geometric problems to which the topology-based approach can be effectively applied.

2. Nonrobustness of geometric algorithms

Geometric algorithms, even though their correctness is theoretically proved, do not necessarily work well in actual computers if they are implemented naively. This is because numerical errors arise in actual computation and hence the algorithms generate inconsistencies that never happen in theory. Let us see from examples how serious these inconsistencies can be.

Example 1 (Planar cut of a convex polyhedron)

Suppose that we are given a convex polyhedron and a plane, and we want to cut the polyhedron by the plane, removing one part, as shown in Fig. 1, where (a) shows a polyhedron and a plane, (b) shows how they are related, and (c) shows the result of cutting. This operation is theoretically simple. We must first classify the vertices into those above the cut plane and those below; second, locate the points of intersection between the cut plane and the edges such that the two terminal vertices are

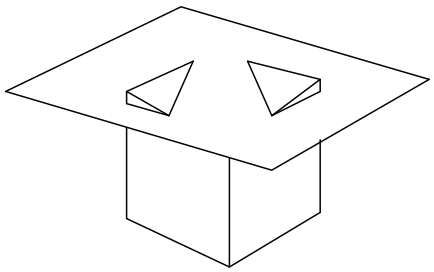


Fig. 2. Inconsistency generated by misclassification of vertices.

classified as mutually opposite sides of the cut plane; and finally, connect those points of intersection to generate the new face.

Suppose that the cut plane is very close and almost parallel to a face of the polyhedron. Then, the classification of the vertices may not always be done correctly because of numerical errors; it can happen that, as shown in Fig. 2, two mutually diagonal vertices are judged as above the cut plane and the other two as below the cut plane. This is inconsistent, because the cut plane and the face have two lines of intersection, which can never happen in Euclidean geometry.

Once that kind of misclassification arises, the algorithm fails because such a situation is not considered.

Example 2 (Construction of a Voronoi diagram)

Let $S = \{P_1, P_2, \ldots, P_n\}$ be a set of n points in the plane. For two points P and Q we denote the Euclidean distance between P and Q by $d(\mathrm{P}, \mathrm{Q})$. We define:

$$R(S; P_i) = \cap_{j \neq i} \{P \in \mathbf{R}^2 \mid d(P, P_i) < d(P, P_j)\}. \tag{1}$$

$R(S; P_i)$ is the region formed by the points P that are closer to P_i than to any other points in S. The plane is partitioned into $R(S; P_1), R(S; P_2), \cdots, R(S; P_n)$ and their boundaries. We call this partition the *Voronoi diagram* generated by S. The elements of S are called generators, and $R(S; P_i)$ is called the *Voronoi region* of P_i. Edges shared by the boundaries of two Voronoi regions are called *Voronoi edges*, and points shared by the boundaries of three or more regions are called *Voronoi points*. The Voronoi diagram is a fundamental data structure in computational geometry and has many applications [21].

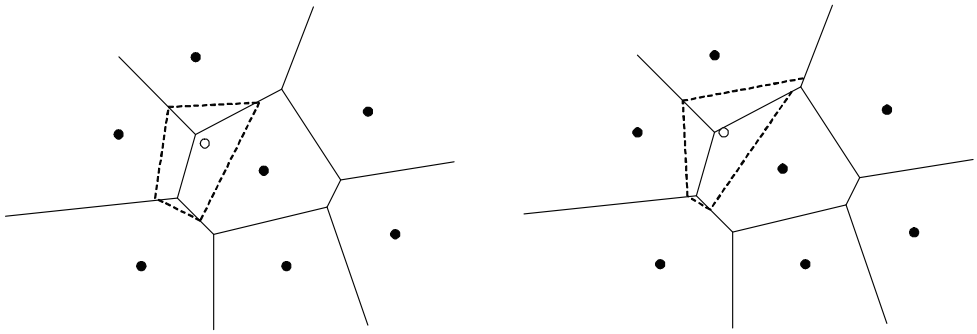
A typical algorithm for constructing a Voronoi diagram is an incremental method. In this method, we start with a simple Voronoi diagram for a few generators (for example, the perpendicular bisector of two generators), and modify it step-by-step by adding new generators one by one.

Figure 3(a) shows a modification of the diagram for addition of a new generator. Suppose that the Voronoi diagram for the filled circles has already been constructed as shown by solid lines in this figure, and that we want to add a new generator represented by the empty circle. The computation of the Voronoi region of the new generator is done in the following way. First, we find the old generator having a Voronoi region that contains the new generator. Then we draw the perpendicular bisector between the old and the new generators. The bisector crosses the boundary of the Voronoi region to enter the neighbor region. We next draw the perpendicular bisector between the new generator and the old generator of the neighbor region. Similarly, we draw the perpendicular bisectors between the new generator and the old generators around it. We eventually reach the start point and thus have a cyclic sequence of bisectors, as shown by the broken lines in Fig. 3(a), which is the boundary of the new generator.

This is the theoretical behavior of the algorithm. If numerical errors arise, the intersection between the bisector and Voronoi edges might be detected incorrectly, and as shown in Fig. 3(b), the bisectors may not form a closed cycle. This is an inconsistency that can never happen in theory and hence the algorithm fails.

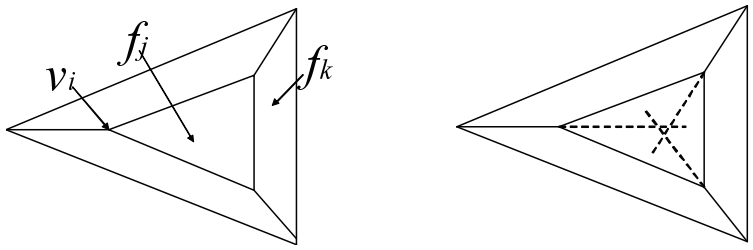
Example 3 (Interpretation of a line drawing as a polyhedron)

Suppose that we are given a line drawing, as shown in Fig. 4(a), and that we want to reconstruct



(a) (b)

Fig. 3. Incremental construction of a Voronoi diagram.



(a) (b)

Fig. 4. Interpretation of a line drawing.

the polyhedron represented by the line drawing. Assume that the line drawing is fixed in the xy plane, and it is generated by projecting a polyhedron orthographically (i.e., the direction of the projection is parallel to the z axis).

Moreover, assume that the viewpoint is at infinity in the positive direction of the z axis.

Suppose that there are n vertices in the line drawings. Let the coordinates of the ith vertex v_i be (x_i, y_i, z_i). Because the line drawing is given in the xy plane, x_i and y_i are known constants; z_i is an unknown variable.

Suppose that there are m faces drawn in the line drawings. Let the plane containing the jth face f_j be represented by:

$$a_j x + b_j y + z + c_j = 0. \tag{2}$$

The three parameters a_j, b_j, c_j are all unknown.

Suppose that v_i is on f_j. Then, we can substitute the coordinates of v_i into the equation of f_j, and we have:

$$a_j x_i + b_j y_i + z_i + c_j = 0, \tag{3}$$

which is linear in unknowns. Collecting all such equations for pairs of vertices and the faces containing them, we have a system of linear equations. Let us denote it as:

$$Aw = 0, \tag{4}$$

where w is a vector of unknown variables

$$w = (z_1, \ldots, z_n, a_1, b_1, c_1, \ldots, a_m, b_m, c_m)$$

and A is a constant matrix.

The line drawing also contains information about relative heights of vertices and faces. As shown in Fig. 4(a), suppose that the vertex v_i is on face f_j, and that face f_k is adjacent to f_j along a convex edge from the other side of v_i. Then, when f_k is extended, it passes between v_i and the viewpoint, which can be represented by the inequality:

$$a_k x_i + b_k y_i + z_i + c_k < 0. \tag{5}$$

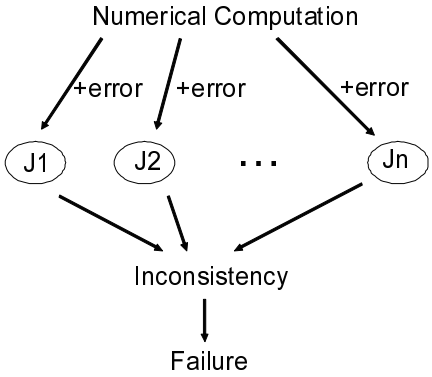


Fig. 5. Relations between errors and failures.

Gathering all similar relative height constraints, we obtain a system of inequalities, which we denote:

$$Bw > 0, \tag{6}$$

where B is a constant matrix.

Then, we can prove that the line drawing represents a polyhedron if and only if the system of linear equations (4) and linear inequalities (6) admits solutions [22]. However, this theorem cannot be used in practical purpose, because it is too sensitive to numerical errors. This can be understood by the following example.

The line drawing in Fig. 4(a) is usually understood as a truncated triangular pyramid seen from above. However, the system of (4) and (6) for this line drawing does not have solutions. Indeed, if this is a truncated pyramid, the three side faces, when extended, should meet at a common apex point, but the apex does not exist, because the three broken lines in Fig. 4(b) do not have a common point of intersection.

In this section, we have seen three examples of geometric algorithms that are fragile; even small numerical errors easily destroy the correctness of the algorithms. In geometric computation, there is a great gap between a theoretically correct algorithm and practically valid software. Filling this gap is a very important task if we want to apply the theory of computational geometry to practical problems.

3. Principle of independence

In this section, we propose a general principle to overcome the nonrobustness of geometric algorithms. For this purpose, first let us review how numerical errors cause algorithms to fail. Figure 5 shows the relation between numerical errors and failures of algorithms.

Geometric algorithms contain many geometric judgments, i.e., procedures that have results that are either "YES" or "NO". For example, we must judge whether a vertex is above the cut plane, or whether a point is to the left of a perpendicular bisector.

Let $J_1, J_2, \ldots, J_n$ be all the judgments in a geometric algorithm. As shown in Fig. 5, each judgment is performed using numerical computation: we compute some value and the result is"YES" if the value is positive and "NO" otherwise.

Because numerical computations contain errors, the results of judgments are sometimes incorrect. Collections of such judgments sometimes contain inconsistencies, which in turn cause the algorithm to fail. This is the causal relation between numerical errors and algorithm failure. We cannot avoid numerical errors, but we want to avoid algorithm failure. This is our goal.

A basic idea to achieve this goal is depicted in Fig. 6. The key point in this figure is that we distinguish between independent judgments and dependent judgments.

Judgments are sometimes mutually dependent in the sense that the results of some judgments determine the results of other judgments as logical consequences.

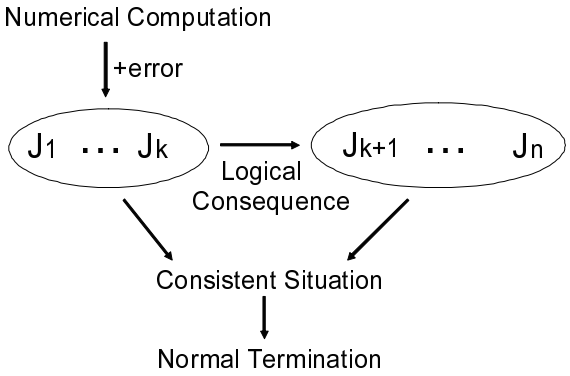


Fig. 6. Independency principle.

For example, let leftturn(p_i, p_j, p_k) denote the judgment having the value "YES" if we turn to the left at point p_j when we move from point p_i through p_j to p_k. Then leftturn(p_1, p_2, p_3) and leftturn(p_2, p_3, p_1) are not independent because if the former is "YES", the latter must also be "YES".

Suppose that, as shown in Fig. 6, we can divide the set of all judgments into two groups; one is a maximal set of mutually independent judgments $\{J_1, J_2, \ldots, J_k\}$, and the other is the set of remaining judgments $\{J_{k+1}, \ldots, J_n\}$. Because the former judgments are mutually independent, any assignment of "YES" and "NO" to these judgments is consistent. Moreover, once assignments of "YES" and "NO" are made to the former set of judgments, the values of the judgments in the latter group are automatically determined as logical consequences.

We carry out numerical computation to obtain the results of the mutually independent judgments, while we adopt the logical consequences to obtain the results of the remaining judgments. Then, the results of the judgments are always consistent.

Numerical errors may cause misjudgments, and hence the results may be different from the true situation, but consistency is still guaranteed. Therefore, the algorithm never fails; it carries the task to the end and gives some output.

Thus, we can achieve a robust implementation of the algorithm. The resulting software is "super-robust" in the sense that, no matter how large the numerical errors are that arise, inconsistency never occurs and the algorithm always terminates "normally". This is our basic idea to overcome nonrobustness of geometric algorithms. The key idea is to discriminate mutually independent judgments, and so we call this idea the "principle of independence".

4. Individual implementations of the principle of independence

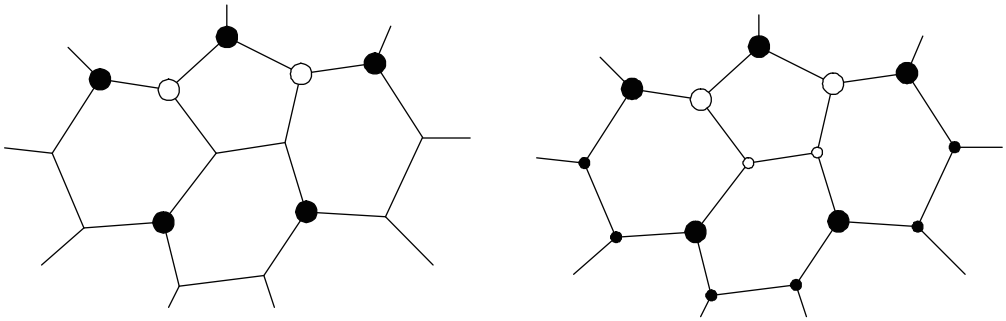
It is not straightforward in general to apply the principle of independence to individual geometric algorithms, because constructing a maximal set of independent judgments is not trivial. In this section, we show how the principle of independence can be used to construct robust algorithms for the three example problems described in Section 2.

4.1 Cutting a convex polyhedron by a plane

Suppose that a convex polyhedron P and a plane Π are given, and we want to cut P by Π, cutting off only one part. The main judgments for this procedure are to decide on which side of Π each vertex lies. These judgments are not necessarily independent.

Let f be a face of the polyhedron P, and $(v_1, v_2, \ldots, v_k)$ be a cyclic sequence of vertices on the boundary of f. Because P is convex, f is a convex polygon. If v_1 and v_3 are judged as being above Π and v_2 as being below Π, the other vertices $v_4, \ldots, v_k$ all should be above Π, because otherwise f and Π have two or more lines of intersection, as shown in Fig. 2. On the other hand, if v_1 is judged as being above Π and v_2 and v_3 as being below Π, the other vertices can be either above Π or below Π. Hence, the independence/dependence relations depend on which values, "YES" or "NO", are assigned to a subset of the judgments.

Let J be the set of all judgments in an algorithm, and J_1 and J_2 be disjoint subsets of J. Suppose



(a) (b)

Fig. 7. Independent and dependent judgments on the above–below tests.

that we have already assigned "YES" to J_1 and "NO" to J_2, while values are not yet assigned to $J - (J_1 \cup J_2)$. We call this situation *partial assignment* (J_1, J_2), abbreviated as $\mathrm{PA}(J_1, J_2)$. For judgment $\alpha \in J - (J_1 \cup J_2)$, we say that α is *independent* under $\mathrm{PA}(J_1, J_2)$ if the value of α is not logically determined from this partial assignment, and *dependent* if the value of α is uniquely determined as the logical consequence of this partial assignment. To implement the principle of independence, some method is required to decide whether or not judgment $\alpha \in J - (J_1 \cup J_2)$ is independent under $\mathrm{PA}(J_1, J_2)$.

For this purpose, we should use purely topological properties that should be satisfied in the procedure of the algorithm.

The polyhedron P contains the graph structure composed of the vertices of P and the edges of P. We call the graph the *vertex–edge graph* of P, and denote it by $G(P)$. Suppose that we cut P by the plane Π. Assume that Π does not contain any vertex of P. Let us call edges that intersect Π *cut edges*. The following property holds.

Property 1 Let P be a convex polyhedron and Π be a plane. If Π intersects P, the subgraph obtained from $G(P)$ by deleting the cut edges consists of exactly two connected components.

This property corresponds to the fact that if we cut a convex polyhedron by a plane, the resulting object consists of exactly two connected components. Although this might seem almost trivial, it is useful for our purpose. Note that Property 1 is stated in purely topological terms, and it does not refer to any numerical values. Therefore, we can always judge correctly whether or not Property 1 is satisfied, without worrying about numerical errors. We can therefore use this property to check whether or not a judgment is independent.

Let V be the set of vertices of P, and $V' \subset V$. We denote by $G(V')$ the subgraph induced by V'.

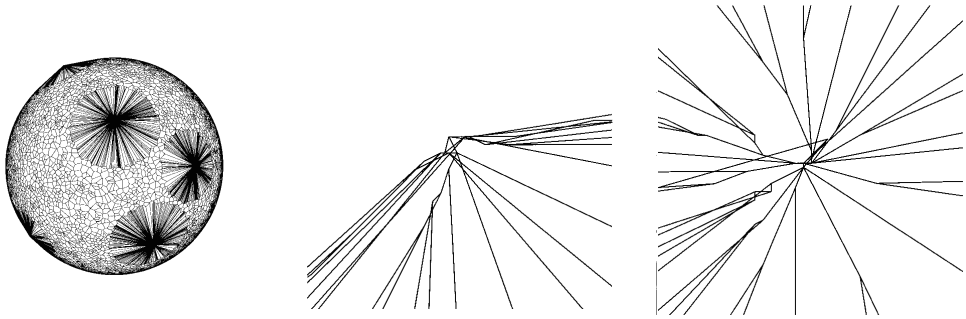
Let us consider an example given in Fig. 7, where the solid lines show part of the vertex–edge graph of P. Suppose that the vertices represented by large white circles in Fig. 7(a) are judged as being above Π, and those by large black circles are judged as being below Π. Then, it is the logical consequence that the vertices with small white circles in Fig. 7(b), should be above Π, and the vertices with small black circles should be below Π; otherwise black circles or white circles constitute a disconnected graph. Thus, the judgments on the small-circle vertices are not independent of the judgments on the large-circle vertices.

This observation can be extended to a general case in the following manner.

Suppose that we have already classified vertices in $V_1 \subset V$ as being above Π, and vertices in $V_2 \subset V - V_1$ as being below Π, and also suppose that the subgraphs induced by V_1 and V_2 are both connected.

First, suppose that the subgraph $G(V - V_2)$ consists of two or more connected components. Then, the components that do not include V_1 should be judged as being below Π; the vertices in those components are not independent.

Next, suppose that the subgraph $G(V - V_1)$ consists of two or more connected components. Then, the components that do not include V_2 should be judged as above Π. Of the remaining vertices,



(a) (b) (c)

Fig. 8. Behavior of a robust program for cutting a polyhedron by planes.

only one is independent in the sense that the assignment of the values of the judgment cannot be determined as the logical consequence of the judgment already done [18].

Figure 8 shows an example of the behavior of a program implemented with this strategy. In this figure, (a) shows the result of cutting the poyhedron by many cut planes that pass through common points. All the numerical computations were done in single-precision floating-point arithmetic. Hence, the cut planes do not pass through an exact common point, but pass near a common point; indeed, two hundred cut planes pass through each common point. Hence, as shown in (b) and (c), which are magnified diagrams around two of the common points, the result of cutting contains complicated microstructure, which makes computation unstable in general. However, our software could compute stably. Indeed, no matter how many cut planes are used, the software will never fail.

4.2 Construction of a Voronoi diagram

As we saw in Fig. 3(a), to update the Voronoi diagram, we must find a closed sequence of perpendicular bisectors between the new generator and the old generators. From a topological point of view, this procedure can be described in the following way.

The Voronoi points and the Voronoi edges can be considered as a graph embedded in the plane. The addition of a new generator corresponds to changing this graph by replacing a substructure of the graph by a cycle associated with the boundary of the Voronoi region of the new generator. In this context, we have the following property.

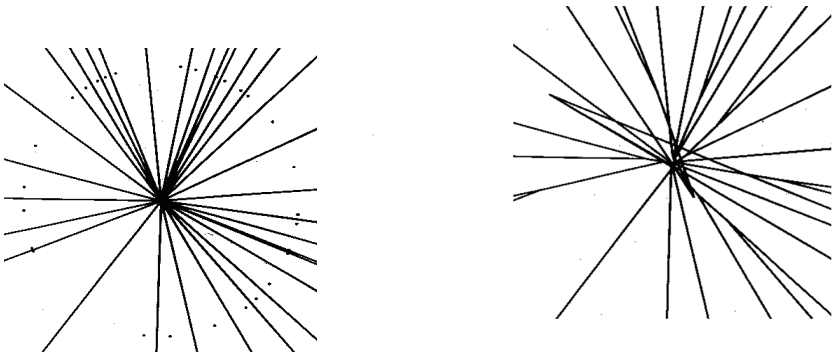
Property 2 The substructure of the Voronoi diagram to be removed in the addition of a new generator should be a tree in a graph-theoretic sense.

A tree is a graph that is connected but does not have cycles. Property 2 can be understood in the following way. If the substructure to be removed is not connected, the Voronoi region of the new generator consists of two or more connected components, but that should not occur because each Voronoi region is connected. Hence, the substructure should be connected. If the substructure contains cycles, some old Voronoi regions disappear completely when we remove it, which should not happen because every generator has a nonempty region. Hence, the substructure should be acyclic.

Property 2 is also purely topological, and hence we can judge whether or not Property 2 is satisfied without worrying about numerical errors. Therefore, we can construct a program that places the highest priority on satisfying Property 2, and uses numerical values only when they do not contradict Property 2 [19].

Figure 9 shows an example of the behavior of a program implemented in this way. In this figure, (a) shows the output Voronoi diagram for 20 generators on a common circle. The computation is again single-precision floating-point, and hence the generators are almost but not exactly cocircular. Hence, the Voronoi edges form a complicated microstructure at the center, and make the computations unstable. However, our program could compute this output stably. If we magnify the central portions, we see the microstructure shown in Fig. 9(b).

This microstructure implies that the program made many misjudgments, but did not encounter



(a) (b)

Fig. 9. Degenerate Voronoi diagram computed by the proposed method.

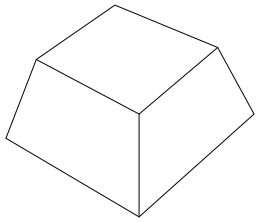


Fig. 10. Insensitive line drawing.

any inconsistency. It carried out the procedure to the end, and gave the output.

4.3 Interpretation of line drawings

As we have seen in Section 2, the mathematical method based on equations (4) and inequalities (6) could not extract a polyhedron from the picture in Fig. 4(a). This is mainly because the system (4) of equations is redundant. It contains more than enough equations, and they are disturbed by numerical errors, generating inconsistency. Therefore, we must remove redundancy from the system (4) of equations. This can be achieved in the following way.

A line drawing is said to be *sensitive to vertex-position errors* (*sensitive*, for short) if slight movements of vertices in the picture plane change the correctness of the line drawing, and *insensitive* otherwise.

The line drawing in Fig. 4 (a) is sensitive, because this picture is correct only when the three edges shared by the two side faces are concurrent. On the other hand, the line drawing in Fig. 10 is insensitive: even if we move the vertices slightly, the resulting picture remains correct.

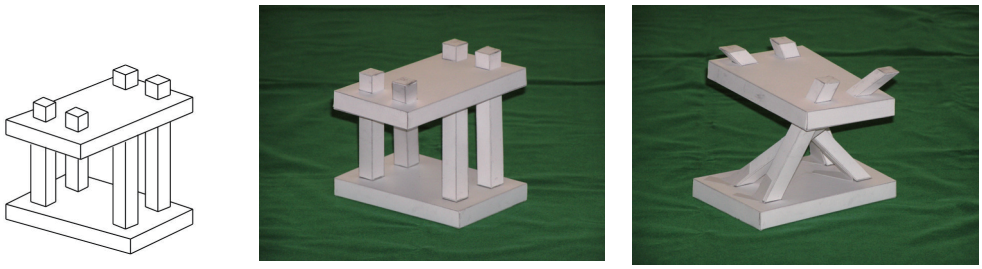
For a finite set X, we denote by $|X|$ the number of elements of X. The following property holds [22].

Property 3 A line drawing is insensitive if and only if:

$$|V| + 3|F| \geq |R| + 4 \tag{7}$$

is satisfied for any subset F of faces such that $|F| \geq 2$, where V is the set of vertices that are on some faces in F, and R is the set of all equations of the form (3) for some face f_j in F. See [22] for the proof.

If the line drawing is sensitive, that intuitively implies that the system (4) of equations is redundant. Therefore, we can judge whether the equations are redundant by Property 3. Moreover, if the equations are redundant, we can find a maximal nonredundant subset of equations, also by Property



(a) (b) (c)

Fig. 11. Impossible object and its realization.

3. Once this is found, we can apply the system of equations (4) and inequalities (6) to judge the realizability of a polyhedron.

By this method, we could find that pictures of impossible objects are not necessarily impossible; some of them are realizable. An example is shown in Fig. 11, where (a) shows a picture of an impossible object, (b) shows a solid having a projection that coincides with (a), and (c) shows the same solid seen from another angle.

5. Concluding remarks

We have presented the principle of independence for designing robust geometric algorithms, and three examples of implementations of this principle. Programs implemented with this principle have many desirable properties.

First, they work stably with any arithmetic precision. This is because we use numerical information only for independent judgments and hence the consistency is guaranteed.

Second, they do not require exception processing for degenerate cases. This is because we assume that every numerical computation contains errors, and hence we cannot recognize degeneracy at all, implying that exception processing for degeneracy is not necessary. Therefore, the programs consists of only general-case processing and consequently are simple.

Third, the output of a program is consistent in a topological sense, and it converges to the true answer as the precision in computation becomes higher.

Fourth, they run quickly because we can use floating-point arithmetic; unlike the exact computation approach, high-precision arithmetic is not necessary.

Thus, the principle of independence has many good properties.

However, the employment of this principle for individual algorithms is not straightforward. This is because some method for distinguishing between independent and dependent judgments is required for each algorithm, which is not trivial. This is the largest drawback of the principle of independence. Hence, our main future work is to augment the set of examples of geometric programs implemented using this principle. The set-theoretic operations for general solids and the construction of generalized Voronoi diagrams are important example algorithms for our future target.

Another problem for future is to evaluate the actual computational speed of software implemented in this principle. In fact we can use floating-point arithmetic instead of high-precision exact arithmetic, but we need additional computation to discriminate between the independent judgments and dependent ones. On the other hand, the exact arithmetic software uses expensive computation basically, but can employ acceleration techniques, which make exact arithmetic fast. Therefore, we need careful study to compare actual speed of the software in this principle.

Acknowledgments

This work is supported by Grants-in-Aid for Scientific Research (B) No. 20360044 and No. 20300098 of the Japanese Society for Promotion of Science, and Meiji Global COE Program on Formation and Development of Mathematical Sciences Based on Modeling and Analysis.

References

[1] F. Preparata and I. Shamos, *Computational Geometry — An Introduction*, Springer, New York, 1985.

[2] M. Berg, O. Cheong, M. Kreveld, and M. Overmars, *Computational Geometry — Algorithms and Applications*, Third Edition, Springer, Heidelberg, 2008.

[3] S. Fortune and C. von Wyk, "Efficient exact arithmetic for computational geometry," *Proc. 9th ACM Annual Symposium on Computational Geometry*, pp. 163–172, 1993.

[4] M. Karasick, D. Lieber, and L.R. Nackman, "Efficient Delaunay triangulation using rational arithmetic," *ACM Trans. Graphics*, vol. 10, pp. 71–91, 1991.

[5] K. Sugihara and M. Iri, "A solid modelling system free from topological inconsistency," *J. Info. Process.*, vol. 12, pp. 380–393, 1989.

[6] K. Mehlhorn and S. Naher, *LEDA — A Platform for Combinatorial and Geometric Computing*, Cambridge University Press, Cambridge, 1999.

[7] *CGAL — Computational Geometry Algorithms Library*, http://www.cgal.org

[8] C.K. Yap, "The exact computational paradigm" in *Computing in Euclidean Geometry*, 2nd edition, eds. D.-Z. Du and F. Hwang, pp. 179–228, World Scientific, Singapore, 1995,

[9] M. Benouamer, D. Michelucci and, B. Peroche, "Error-free boundary evaluation using lazy rational arithmetic — A detailed implementation," *Proc. 2nd Symposium on Solid Modeling and Applications*, pp. 115–126, 1993.

[10] K. Sugihara, "Experimental study on acceleration of an exact-arithmetic geometric algorithm," *Proc. 1997 International Conference on Shape Modeling and Applications*, pp. 160–168, 1997.

[11] H. Edelsbrunner and E.P. Mücke, "Simulation of simplicity — A technique to cope with degenerate cases in geometric algorithms," *Proc. 4th ACM Annual Symposium on Computational Geometry*, pp. 118–133, 1988.

[12] C.K. Yap, "A geometric consistency theorem for a symbolic perturbation scheme," *Proc. 4th Annual ACM Symposium on Computational Geometry*, pp. 134–142, 1988.

[13] L. Guibas, D. Salesin, and J. Stolfi, "Epsilon geometry — Building robust algorithms from imprecise computations," *Proc. 5th ACM Annual Symposium on Computational Geometry*, pp. 208–217, May 1989.

[14] V. Milenkovic and E. Sacks, "A monotonic convolution for Minkowski sums," *Int. J. Comp. Geom. Applic.*, vol. 17, pp. 383–396, 2007.

[15] M. Segal and C.H. Sequin, "Consistent calculations for solid modeling," *Proc. ACM Annual Symposium on Computational Geometry*, pp. 29–35, 1985.

[16] K. Ozaki, T. Ogita, S.M. Rump, and S. Oishi, "Adaptive and efficient algorithm for 2D orientation problem," *Japan J. Indust. Appl. Math.*, vol. 26, pp. 215–231, 2009.

[17] S.M. Rump, "Inversion of extremely ill-conditioned matrices in floating-point," *Japan J. Indust. Appl. Math.*, vol. 26, pp. 419–422, 2009.

[18] K. Sugihara, "A robust and consistent algorithm for intersecting convex polyhedra," *Computer Graphics Forum, EUROGRAPHICS'94*, pp. C-45–C-54, 1994.

[19] K. Sugihara and M. Iri, "A robust topology-oriented incremental algorithm for Voronoi diagrams," *Int. J. Comp. Geom. Applic.*, vol. 4, pp. 179–228, 1994.

[20] M. Held and S. Huber, "Topology-oriented incremental computation of Voronoi diagrams of circular arcs and straight-line segments," *Comp. Aided Des.*, vol. 41, pp. 327–338, 2009.

[21] A. Okabe, B. Boots, K. Sugihara, and S.-N. Chiu, *Spatial Tessellations — Concepts and Applications of Voronoi Diagrams*, Second Edition, John Wiley and sons, Chichester, 2000.

[22] K. Sugihara, *Machine Interpretation of Line Drawings*, MIT Press, Cambridge, 1986.