Proceedings of ALGORITMY 2002
Conference on Scientific Computing, pp. 153–161

COPING WITH NUMERICAL PROBLEMS IN IMPLEMENTATION OF 3D DELAUNAY TRIANGULATION [*]

PAVEL MAUR[†] AND IVANA KOLINGEROVÁ[‡]

Abstract. In this paper we present our experience with the solution how to make the implementation of 3D Delaunay triangulation more stable. A brief overview of possible methods is given. The proper solution is found in geometric predicates. Tests of two different implementations of them are provided and results of incorporating one of them into our implementation of 3D Delaunay triangulator are presented. We also offer small improvement of this method.

Key words. 3D Delaunay triangulation, numerical stability

1. Introduction.

Delaunay triangulation in 2D an 3D space is one of fundamental structures in computational geometry. Applications of Delaunay triangulation are mainly in mesh generation methods, e.g., for FEM computation, contouring, object reconstruction, terrain modelling, etc. Many algorithms based on different approaches were developed for Delaunay triangulation.

Usually the theoretical algorithms are developed under the assumption that there is infinite precision arithmetic and data is without singularity. If such algorithms are implemented, we can be disappointed with their behaviour. Particularly, the algorithms for DT often assume that input points are in general position, i.e. no four or more points are coplanar and no five or more points are cospherical. The assumption of simplicity is not always fulfilled by real data and the behaviour of the program is ambiguous in such cases. Moreover, only arithmetic with limited precision is available on real computers. Computation in integer arithmetic can cause overflow, floating-point arithmetic can produce round off errors. Thus an incorrect decision can be produced by the implementation and can even lead to its failure.

We implemented the algorithm for 3D DT based on incremental approach with the use of flips [8, 9] in standard floating-point arithmetic. Our implementation worked fine on data with randomly generated points with uniform distribution up to 200,000 points. But when the points were more “organized”, our implementation sometimes failed. The worst case—in the sense of stability—appeared, when the points were distributed regularly, i.e., they lied on a uniform grid (in the corners of regularly distributed cubes). Then the implementation was not able to construct the triangulation even for small number of points.

We were searching for a method how to make our implementation more stable. In this paper the basic methods of numerically robust computation, which exist for the Delauany triangulation problem, are mentioned. Those methods are based either on integer arithmetic (with 32 bit-length, which is most common on current computers) or on floating-point arithmetic according to IEEE 754 standard [6]. Most attention will be given to the two methods developed by [3] and [10] and to the results of their testing. We also present our modification of the latter method.

[*]This work was supported by the Ministry of Education of Czech Republic—project MSM 235200005.

[†]University of West Bohemia, Pilsen, Czech republic, (maur@kiv.zcu.cz).

[‡]University of West Bohemia, Pilsen, Czech republic, (kolinger@kiv.zcu.cz).

2. Problems and Solutions. The problems of our implementation were found in numerical computing. Particularly in existence of incorrect decisions, which were caused by the round off errors of usual arithmetic.

The first problem was in detection whether a point lies inside, outside or on a face of a tetrahedron. The plane of the face is given by the three vertices of the tetrahedron face. We computed all coefficients of plane equations in advance and stored them in data structures. The reason for this was in our desire to speed up the repetitive tests of mutual position of a point and a plane. When this test produces a bad answer, the point-in-tetrahedron test fails, too. It causes failure of the whole program.

The second problem was in the test whether the point lies inside, on or outside the sphere. The sphere was given by four points, which were the points of particular tetrahedron. At first we computed the centre of the sphere as an intersection of three planes, then the radius was computed. The position of the fifth point was derived from its distance from the centre. The failure of this test does not cause the crash of the program, but non-Delaunay tetrahedra are generated in the triangulation.

The third problem was in detection of different configurations of tetrahedra, which share a common face. Originally it was done by computing the intersection of the shared triangle with the line given by the two vertices of tetrahedra lying on the opposite side of the shared triangle. The problems came, when the point of intersection lied on the edge of the triangle. Because of inexact arithmetic the intersection could be taken as inside or outside the triangle, which could lead to generation of tetrahedron with zero volume or to making wrong topological changes. It led to the failure of the program.

The reason why the program fails especially on the grid data sets is that there are a lot of singular cases in these data sets. The singular case means situation when four or more points lie on a plane or when five or more points lie on a sphere. In data sets with another point distribution such singular cases occur very rarely.

In our implementation we used the so-called ε-arithmetic. It means the result is not taken as it is, but with a small ε-tolerance around the computed value. We found that this method could not sufficiently solve numerical problems that we had, because sometimes it still produced incorrect decisions in the mentioned tests. Its results are "exact" only with some probability. Thus we started to look for really exact solution. In the available literature we have found several solutions, which are based on both types of arithmetic—integer and floating-point. Following lines give a short overview of them, more information from this area can be found e.g. in [5], chap. 35.

LEDA and CGAL are examples of huge packages, which provide operations with integer and floating-point data types with arbitrary precision. Floating-point filters are provided as well. The exact data types are composed from basic ones so the precision of the result is limited only with available memory. But exact arithmetic is not able to solve all of the problems—there are also problems caused by singularities in the input data. Well known solution based on the exact integer arithmetic is so-called SoS—Simulation of Simplicity described in [2]. It is designed to remove the degenerated cases from the input data by symbolical perturbations in arbitrary dimension. The algorithm, which uses SoS, is run on perturbed points, but the result is presented as the solution of the original point set. The result has to be used carefully, some kind of post-processing is needed sometimes. Other methods are based on analysis of the input set of points and geometric predicates [1], on restriction of input data [12] and on the use of topological constraints [7].

For the exact decisions in triangulation algorithms it is possible to use the approaches described above. But if we do not want to make any restriction on input points in coordinate values, in their distribution or position—shortly said if we want to keep the input data as they are—there are two more solutions, which offer exact evaluation of geometric predicates both for integer and floating-point arithmetic. Both of them are described below, because both of them were candidates for solving problems of our numerically unstable implementation.

3. Exact Evaluation of Geometric Predicates. Geometric predicates provide results of various geometrical tests (e.g., mutual position of a point and a line). They are computed as a value of determinant of a particular matrix. They can be generalized into an arbitrary dimension. For the purpose of a triangulation algorithm in 3D space, two following predicates are of main interest:

- the test of a mutual position of a point d and a plane given by three oriented points a, b, c, so called *orient*3*d* predicate; this predicate is given by the following matrix of dimension 4 or after some algebraic transformations by the matrix of dimension 3

$$\begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix} = \begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z \\ b_x - d_x & b_y - d_y & b_z - d_z \\ c_x - d_x & c_y - d_y & c_z - d_z \end{vmatrix}. \tag{1}$$

  The value of the predicate is positive if the point d lies below the plane given by a, b, c in counterclockwise order. Predicate is negative, if d lies above the plane, and zero if all points lie on the plane.
- the test of a mutual position of a point e and a sphere given by four oriented points a, b, c, d, so called *insphere* predicate; given by matrices

$$\begin{vmatrix} a_x & a_y & a_z & a_x^2 + a_y^2 + a_z^2 & 1 \\ b_x & b_y & b_z & b_x^2 + b_y^2 + b_z^2 & 1 \\ c_x & c_y & c_z & c_x^2 + c_y^2 + c_z^2 & 1 \\ d_x & d_y & d_z & d_x^2 + d_y^2 + d_z^2 & 1 \\ e_x & e_y & e_z & e_x^2 + e_y^2 + e_z^2 & 1 \end{vmatrix} = \tag{2}$$

$$= \begin{vmatrix} a_x - e_x & a_y - e_y & a_z - e_z & (a_x - e_x)^2 + (a_y - e_y)^2 + (a_z - e_z)^2 \\ b_x - e_x & b_y - e_y & b_z - e_z & (b_x - e_x)^2 + (b_y - e_y)^2 + (b_z - e_z)^2 \\ c_x - e_x & c_y - e_y & c_z - e_z & (c_x - e_x)^2 + (c_y - e_y)^2 + (c_z - e_z)^2 \\ d_x - e_x & d_y - e_y & d_z - e_z & (d_x - e_x)^2 + (d_y - e_y)^2 + (d_z - e_z)^2 \end{vmatrix}.$$

  The value of the predicate is positive if the point e lies inside the sphere given by a, b, c, d, which are ordered to be positive oriented. Predicate is negative, if e lies outside the sphere, and zero if all points lie on the common sphere.

To make a correct geometric decision, these determinants have to be evaluated exactly. It is possible because the evaluation leads to an expression with finite number of terms. More, the methods for predicate evaluation usually use some speed-up techniques. It means the result is computed exactly only so far to give us a correct result. The result is then not exact in the numerical sense, but is always correct in the geometrical sense. Two such methods of predicate evaluation are briefly presented. The first one is based on integer arithmetic, the second one uses floating-point.

3.1. LN Generator. The first method was presented in [3]. The program called LN generator was created, which serves for the purpose of symbolical writing of an arbitrary predicate in the LN language. This symbolical input is then transformed into a C++ routine for exact computation of the predicate enriched with the so-called floating-point filter. Such filter is good for checking whether the result is exact enough in usual arithmetic or if it is necessary to employ the exact arithmetic procedure.

The implementation of exact computation is done with the vector of numbers, which is long enough to store the exact result. Because of speed, 53-bit mantissa of 64-bit floating-point number is used as the basic data type of the vector instead of 32-bit integer. The overhead of this method is in checking the overflows between the adjacent members of the vector and in ensuring the carries between them—this process is called normalization. The size of input coordinates is limited by 53-bit integer. It is recommended to use smaller number of input bits (23 bits), because the higher bit-lengths on input cause the normalization is needed more often and the exact computation is slower.

The computation is firstly done by usual arithmetic. Then the floating-point filter is used to filter out the results, which are exact enough. The filter is based on computation of an error of the result in advance. The static error bound is derived from the maximal bit-length needed for particular expression, which arises from following basic rules

$$\begin{aligned} maxbitlen(a \pm b) &= 1 + max(maxbitlen(a), maxbitlen(b)),\\ maxbitlen(a \times b) &= maxbitlen(a) + maxbitlen(b), \end{aligned}$$

where a, b are integer numbers. Then the static upper bound *maxerr* is computed as follows. If $maxbitlen(x) \leq 53$, then $maxerr(x) = 0$, otherwise

$$\begin{aligned} maxerr(a \pm b) &= maxerr(a) + maxerr(b) + 2^{maxbitlen(a+b)-53},\\ maxerr(a \times b) &= maxerr(a)maxbitlen(b) + maxerr(b)maxbitlen(a) +\\ &\quad + 2^{maxbitlen(a\times b)-53}. \end{aligned}$$

The great advantage of the static upper bound of the error is that it can be computed statically before the program is run. In the runtime the program only compares the absolute value of the determinant with the value of *maxerr* and if the determinant is greater than *maxerr*, the result is taken as exact enough. In the other case, the process of the exact computation starts.

3.2. Adaptive Floating-Point Geometric Predicates. The previous approach can be also called *multiple-digit* format. In this section a method based on *multiple-term* format, which was described in [10], is presented. This technique uses all 64 bits of floating-point double precision numbers. The advantage of this approach can be seen for example in adding two numbers of very different magnitudes—if we want to evaluate the expression $2^{300} + 2^{-300}$, only two words in memory are enough to store the result. The multiple-digit approach has to use at least 601 bits of memory, because all intermediate zero values have to be stored.

The computation in this method is based on a nonoverlapping expansion of the number. Or let us say the numbers are stored as a vector of floating-point values, where the gaps in the vector are allowed in the places, where only zero bits are stored. It can be imagined as a sparse vector. The values of two floating-point numbers x, y are not overlapping if the least significant nonzero bit of x is more significant than the most significant nonzero bit of y, or vice versa. For example the binary values 1100 and -10.1 are not overlapping, whereas 101 and 10 overlap.

The elementary functions, which transform the addition, subtraction and multiplication of two floating-point numbers into the nonoverlapping expansion of the result (that are also two floating-point numbers) and complex functions working with the nonoverlapping values in vector as well are presented in the [10]. Thanks to nonoverlapping expansion the result of evaluating the determinant can be computed adaptively. The adaptivity is based on the dynamic computation of error bounds for expression, which is produced by the evaluation of determinant. It can be imagined as several-stages floating-point filter.

4. Testing. To make our implementation of 3D Delaunay triangulation numerically stable, we decided to incorporate the solution based on exact geometric predicates for the following reasons: we needed to make robust only the critical points in our program which are geometric predicates. And, also, because of time reasons and because of our orientation towards computer graphics, not numerical mathematics, we wanted as simple solution as possible. The solution based on exact predicates is easy to adapt for our implementation. It can be done quickly and also only small change of our source code is necessary. To decide between Fortune's predicates (multiple-digit) and Shewchuk's predicates (multiple term), we performed several tests.

The first part of the tests was made on artificial data sets with different distribution of points. Particularly we used points with uniform and gaussian distribution, points distributed in clusters, on the surface of the sphere and points on regular grid. The number of points in each data set was 50,000, 100,000, 200,000 and 240,000. The results were taken as an average of results from five different data sets. As the second type of input data we used 17 real objects, particularly the points on surface of the real models. There were used objects such as chair, dino, teapot, CTHead and others [11, 4], the number of their vertices was up to 277,000.

To provide the same input data for all tested implementations, all input data was transformed onto integer numbers by the simple scale. The reason for this was that the Fortune's predicates work only with integers. This correction of the range of the data did not affect their data type. Each tested function used the data type suited for it.

For the testing, a simple algorithm was used. In the following example the processing of input points is shown. The *orient3d* predicate is used, which needs only four input points. If we used *insphere* predicate to test, five input points are needed.

```
for (i = 0; i < (the number of input points - 3); i++) {
  result = orient3d(i,i+1,i+2,i+3);
  process the result;
}
```

Thanks to Mr. Fortune and Mr. Shewchuk we obtained the source codes of their predicates implementations. In the case of Fortune's predicates two versions of them were provided—for 31 bit-length and for 53 bit-length. We had to modify slightly all of the source codes to achieve the same conditions in our tests. At the end we obtained five functions to be tested. Exact Fortune's predicates for two different bit-lengths, named as *Fort31* and *Fort53* in the following text. Then exact Shewchuk's predicates named as *Shew*. And, finally, pure inexact computation of determinant called *det*. We also separated our own test-functions from our inexact implementation to use them in the tests. These functions are referred as *orig*.

All mentioned functions will be called as "tested functions" in following text. All experiments were executed on Pentium 3, 450 MHz equipped with 1GB of memory under Windows 2000 operating system. Microsoft Visual C++ was used as the compiler.

TABLE 1
Percentage of bad decisions of orient3d and insphere tests on grid data.

orient3d

points	Fort31	Fort53	Shew	det	orig
50,000	0	0	0	0.0012	0.0024
100,000	0	0	0	0.0002	0.0008
200,000	0	0	0	0.0002	0.0003
240,000	0	0	0	0.0004	0.0006

insphere

points	Fort31	Fort53	Shew	det	orig
50,000	0	0	0	0	0.0060
100,000	0	0	0	0	0.0032
200,000	0	0	0	0.0001	0.0019
240,000	0	0	0	0	0.0014

4.1. Test of Precision. To test the precision of tested functions we used the following approach: in each step of the testing cycle we obtained the results of all tested functions and compared their results. As the correct ones, the results of exact predicates were taken. Differences are depicted in Table 1.

It was surprising that on artificial data sets only the points distributed on uniform grid caused incorrect results. On the other data sets *det* and *orig* functions gave the correct results. This held for *orient3d* tests as well as for *insphere* tests.

The result of precision tests on real data sets is as follows. Results of exact predicates were the same for all input data as it was expected. For the *orient3d* tests the percentage of bad decisions made by the *det* was only 0.0002, the percentage of *orig* was worse: 0.0026. For the *insphere* predicate the percentage of bad decisions made by the *det* was 0.0878, the percentage of *orig* was surprisingly high: 10.69.

4.2. Test of Speed. To test the speed of all tested functions we simply measured the time of each function needed for processing the whole input data set. To show the results of these tests, we present here only the following representative selection.

In Fig. 1 you can see the time needed for executing all tested functions on several artificial data sets. It is quite strange that the times are nearly equal for all data sets. We expected that execution on sphere or grid data would be significantly slower. The reason for the speed similarity is probably in the random placement of the points in the data sets. If the points were placed in a "more organized" way, e.g. the close points were also "near" in the input file, there would be probably more differences among the different distribution of points.

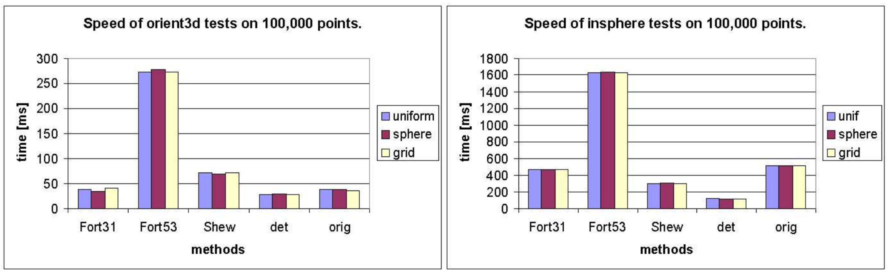

FIG. 1. *Orient3d and insphere tests on artificial data sets.*

The situation is quite different for real data. As it can be seen from Table 2, *det*

Table 2
Time of orient3d tests spent on real data.

orient3d

object	points	Fort31	Fort53	Shew	det	orig
teapot	80,203	94	219	141	15	31
CTHead	230,328	172	641	1094	62	78
14spheres	277,228	296	765	2780	78	109

insphere

object	points	Fort31	Fort53	Shew	det	orig
teapot	80,203	375	1312	719	94	469
CTHead	230,328	1094	3750	5047	281	1750
14spheres	277,228	1313	4516	14500	328	2860

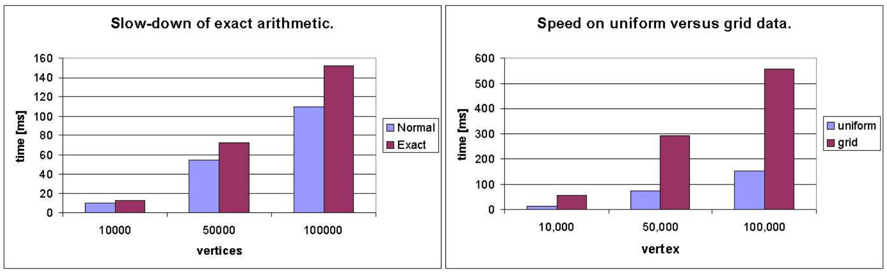

Fig. 2. *Slow-down of new implementation.*

and *orig* functions are always faster ones, because they are inexact. The situation in the field of exact predicates is not as clear as before. Sometimes the Fortune's predicates are faster than Shewchuk's one sometimes the situation is reversed. In general, Fortune's predicates are slower in cases, when the exact arithmetic is needed. It is due to the fact that the floating-point filter marks more cases as inexact than necessary. On the other side, Shewchuk's predicates use exact computation in less cases, but there is more time spent on making the decision, whether the exact computation is needed or not. This decision is made faster in Fortune's predicates, as the error bound is computed in advance.

5. Practical Use. From the results of our testing none of the methods for exact evaluation of predicates is the clear winner. The decision, which one of them to select, has to be supported by other arguments specific to particular implementation. The use of floating-point is advantage for our Delaunay triangulator, as our implementation is based on this arithmetic. The second advantage of floating-point predicates is no restriction of input data. The disadvantage is, of course, the test made on real data are slow. However we think that the price for worse speed is reasonable. Thus we decided to use Shewchuk's predicates.

Now several words and graphs about time proportions. The speed of our implementation without and with the exact predicates is depicted in Fig. 2 in the left part. The graphs are made for uniform data sets and we can see the slow-down about 30%. The situation is more drastic for grid data. The right part of Fig. 2 shows a time comparison of exact predicates on uniform and grid data sets of the same amount of points. The slow-down is more significant now, it is nearly 400%.

Because we used exact predicates in implementation of 3D Delaunay triangulator, we were curious, how many non-Delauany tetrahedra were produced by old unexact

TABLE 3
The time ratio of improved to original predicates. Bit-length of input data is 40.

object	orient2d	orient3d	incircle	insphere
teapot	0.5161	1.128	1.0683	1.0208
CTHead	0.8297	0.9694	0.9880	0.9811
14spheres	0.8623	0.9996	1.1063	1.0033

implementation. The measurement on uniform data sets showed us that only about 0.1% of tetrahedra were wrong.

Because the error bound in Shewchuk's predicates is computed in runtime, which slows down the evaluation, we decided to use a similar filter as in Fortune's predicates. This filter would be placed before the procedure of adaptive computation to make a first step of filtering. But for arbitrary floating-point numbers it is impossible to give any general restriction on the values like in Fortune's filter. Thus the static error bound has to be dependent on the range of the input data. In our explanation we restrict ourselves only on integer numbers. But the main idea works for any kind of input—even for fixed point and floating-point numbers. The only thing we have to know is the range, i.e. the maximal bit-length of the data values. The algorithm for getting the error bound is as follows:

```
get maxbitlength for x, y, z coordinates from input data set;
compute maxbitlength for predicate expression;
if (maxbitlength <= 53) {
  errbound = 0.0;
}
else {
  compute maximal value of expression from maxbitlength;
  compute errbound;
}
```

The computation of maxbitlength is based on formulae mentioned in section 3.1. If the maxbitlength is less or equal to 53, the whole expression can be stored exactly in the memory, thus no error bound is needed. Else the error bound is computed according to the rules derived in [10].

The static error bound is then used as the entering point for the original Shewchuk's predicates. We tested the mentioned method on all predicates provided in [10], i.e. *orient2d*, *orient3d*, *incircle* and *insphere*. Our improved predicates took only about 50% of time of original predicates on artificial data. But this speed up was achieved mainly due to the fact that exact arithmetic was not needed too often—we were able to filter out such cases quickly. On the real data, where the necessity of exact computation is greater, the results of our predicates are similar to the original ones, as you can see in Table 3. This is clear, because in such cases, actually original predicates are used plus the time needed for two comparisons of floating-point filter.

6. Conclusion. In this paper we presented possible solutions how to make the implementation of 3D Delaunay triangulation more stable. We found such a solution in geometric predicates. We made some tests on two different implementations of them and compared their results. We incorporated the one of the proposed solutions into our 3D Delaunay triangulator. As it was expected the implementation became really stable. We were also dealing with idea to enrich the predicates with the floating-point filter. However, from our experiments it is clear, that such enrichment is good only for cases, where the exact arithmetic is needed only rarely.

REFERENCES

[1] O. Devillers and F. P. Preparata, *Further Results on Arithmetic Filters for Geometric Predicates*, INRIA, 1998.

[2] H. Edelsbrunner and E. P. Mücke, *Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms*, ACM Transactions on Graphics, 9(1), 1990, pp. 66–104.

[3] S. Fortune and C. J. Van Wyk, *Static Analysis Yields Efficient Exact Integer Arithmetic for computational Geometry*, ACM Transactions on Graphics 15(3), 1996, 223–248.

[4] Georgia Institute of Technology. Large geometric models archive. http://www.cc.gatech.edu/projects/large_models

[5] *Handbook of Discrete and Computational Geometry*, Edited by J. E. Goodman and J. O'Rourke. CRC Press, 1997.

[6] S. Hollasch, *IEEE Standard 754 Floating Point Numbers*, 1998. http://research.microsoft.com/~hollasch/cgindex/coding/ieeefloat.html

[7] P. M. Hubbard, *Improving Accuracy in a Robust Algorithm for Three-Dimensional Voronoi Diagrams*, The Journal of Graphics Tools, 1(1), 1996, pp. 33–47.

[8] B. Joe, *Construction of three-dimensional Delaunay triangulations using local transformations*, Computer Aided Geometric Design, Vol. 8, 1991, pp. 123–142.

[9] P. Maur and I. Kolingerová, *Post-optimization of Delaunay Tetrahedrization*, In Proceedings of Spring Conference on Computer Graphics, Budmerice, Slovak Republic, 2001, ISBN 80 223 1606-7, pp.76–85.

[10] J. R. Shewchuk, *Robust Adaptive Floating-Point Geometric Predicates*, Proceedings of the Twelfth Annual Symposium on Computational Geometry, ACM, 1996.

[11] Stanford Computer Graphics Laboratory. http://graphics.stanford.edu/data/3Dscanrep

[12] K. Sugihara, *A Simple Method for Avoiding Numerical Errors and Degeneracy in Voronoi Diagram Construction*, IEICE Trans. Fundamentals, vol. E75-A, No. 4, 1992.