Robust Construction of 3-D Conforming Delaunay Meshes Using Arbitrary-Precision Arithmetic

Konstantin Bogomolov

Institute of Mathematical Modeling, Moscow kbogomolov@mail.ru

An algorithm for the construction of 3-D conforming Delaunay tetrahedralizations is presented. The boundary of the meshed domain is contained within Voronoï cells of the boundary vertices of the resulting mesh. The algorithm is explained heuristically. It has been implemented. The problem of numerical precision is shown to be a major obstacle to robust implementation of the algorithm. The Automatic Arbitrary-Precision Arithmetic Library is introduced to solve this problem. The resulting program is intended to be applicable to any mathematically correct input. It has performed successfully on a number of test cases, including a known difficult case for tetrahedral meshing. It is available on the Internet. The Arithmetic Library may be useful for resolving numerical precision problems in any application, and as a base for experimenting with new meshing strategies.

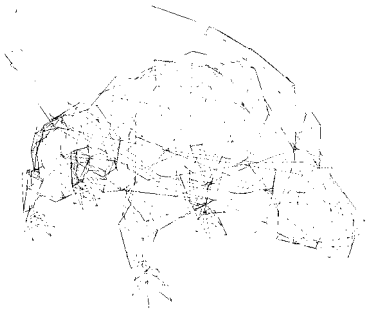

Fig. 1. *Bullito* model

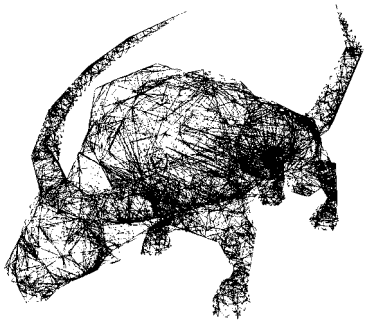

Fig. 2. Delaunay mesh of *Bullito*

1 Introduction

The Delaunay tetrahedralization and its dual, the Voronoï diagram, are attractive approaches to three-dimensional mesh generation due to several

useful properties of these objects. The Voronoï diagram, in particular, seems well suited for application to 3-D numerical simulation problems because it divides the space into domains naturally "belonging" to each of the points of the input set.

The difficulty of Delaunay / Voronoï methods lies in obtaining a mesh that fits into the specified boundary. The basic Delaunay tetrahedralization covers the convex hull of the set of input points. If the domain to be meshed is not convex, there is no guarantee that all of its faces and edges will appear in the tetrahedralization.

An overview of the known approaches to this problem is contained in the paper by Jonathan Richard Shewchuk [1]. These include the conforming Delaunay, the "almost Delaunay" and the constrained Delaunay approaches. The conforming Delaunay approach is justifiably criticized for adding more points than the other approaches and for its difficulty in controlling the quality of the mesh. However, of the three approaches, only this one generates a truly Delaunay mesh.

The conforming Delaunay tetrahedralization is a normal Delaunay tetrahedralization of the set of input points, together with some additional points that are introduced, typically, onto the domain boundary. It must contain all of the domain's edges and faces as unions of its own edges and faces. The task of a conforming Delaunay tetrahedralization algorithm is to determine the positions of additional points needed to recover the given domain boundary.

Fig. 3. Voronoï cells of *Bullito*

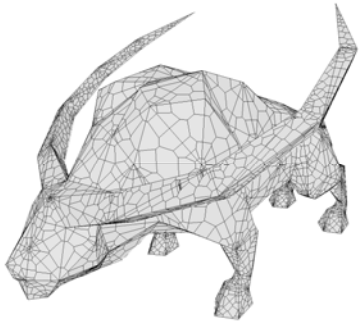

Fig. 4. Voronoï cells at the boundary

It seems that the Delaunay property would be important for the user of a Delaunay algorithm, and not merely due to the quality of the Delaunay tetrahedra (especially considering that quality Delaunay meshing can be actually difficult)! The other good property of Delaunay meshes is their duality to the Voronoï diagrams. Perhaps the user of a Delaunay meshing program will be more interested in the Voronoï cells than in the tetrahedral mesh, - and in that case, the conforming Delaunay approach should be the

most straightforward one, as it immediately provides the covering of the domain with the Voronoï cells in the usual metric.

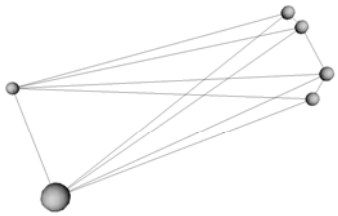

Fig. 5. A domain with sharp angles

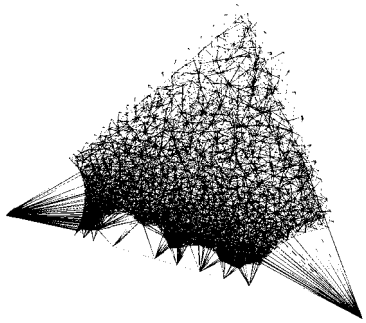

Fig. 6. Delaunay mesh

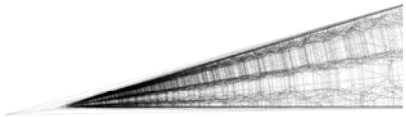

Fig. 7. Voronoï cells

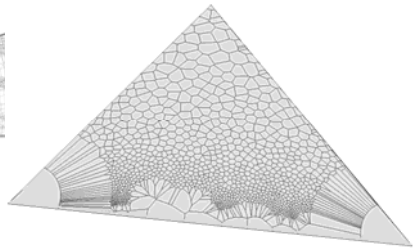

Fig. 8. Voronoï cells on the boundary

My algorithm and program strive to achieve a stricter conformation. I want the Voronoï cells of the points belonging to the domain boundary to contain within them this entire boundary. They will constitute the outer layer of the domain's Voronoï decomposition, and it will be possible to trim them with the boundary, thus achieving a boundary-fitting Voronoï mesh (figures 3, 4). The cells of the interior points will all be contained strictly inside the domain, without any of them touching the boundary or sticking outside. The internal two-sided boundaries (which can also be present in the domain) will also be fully covered by their Voronoï cells. This construction (also known as the empty smallest circumsphere property for the boundary triangles) is a known way to make Delaunay meshing simpler, at the price of adding more points. However, it may perhaps be also useful to the potential user of the Voronoï mesh: for example, it could simplify the specification of boundary conditions (or of the special processing near the boundary) in 3-D numerical simulation problems. Besides, it helps to make the pictures look nice!

Despite the conceptual simplicity of the conforming Delaunay tetrahedralizations, it seems that the first proof of their existence for arbitrary domains was published only in 2000, by Michael Murphy, David M. Mount and Carl W. Gable [2]. Algorithms for constructing them have also been published by Cohen-Steiner, de Verdière and Yvinec [3], Cheng and Poon [4], Pav and Walkington [5], and Cheng, Dey, Ramos and Ray [6]. These papers all present proofs of the algorithms' correctness. Cohen-Steiner, de Verdière and Yvinec also report an implementation of their algorithm that requires knowledge of the domain's local feature size.

In this article, a simple but unproven conforming Delaunay algorithm that does not require additional information about the input domain is presented. It has been successfully implemented. I tried to develop a black-box code that works correctly with arbitrary, mathematically correct input. The solution to numerical precision problems I have implemented may be of interest by itself. The code, including the Library, is available for download at `http://kbogomolov.informatics.ru`.

I would like to thank Professor Vladimir Tishkin for many useful discussions, and the reviewers of this article, whose comments were extremely helpful for improving this publication.

The work was supported by ISTC grant 1820. The *Bullito* model is courtesy of Primal Software, Moscow.

2 The Problem

The input for the algorithm is a Piecewise-Linear Complex (defined, for example, in the article [3]). There is an additional requirement that some of the faces of the PLC are marked as "boundary": these faces should divide the space into the interior ("the domain") and the exterior. All other faces should belong to the interior; they play the role of 2-sided internal boundaries. If we need to mesh an arbitrary PLC, it is possible convert it to the required form by enclosing it inside a polyhedron and marking all of the polyhedron's faces as boundary. This discrimination between the interior and the boundary faces is useful, in practice, for avoiding the unnecessary refinement caused by the interactions of the tetrahedra outside the domain.

All of the points added into the Delaunay tetrahedralization (the vertices of the domain, the interior points and the points added during the conformation process) will be called throughout this paper "points" or "vertices". The vertices of the original input domain will also sometimes be called "corners". The PLC will sometimes be called "domain" or "boundary".

It is additionally required of the algorithm that the faces of the PLC must be fully contained inside the Voronoï cells of their points. To illustrate this requirement, a conformed Delaunay triangle on a boundary face may be considered. Its presence does mean that the cells of its three vertices share a common Voronoï edge somewhere along the line that is orthogonal to the face and passes through the triangle's circumcenter. It does not guarantee, however, that this common edge intersects the boundary; thus, some part of the boundary may be outside the outer layer of cells, and some interior cells may be sticking outside the domain. So, in order for this condition to be fulfilled, it is required that all of the tetrahedra adjacent to the boundary have their circumcenters on the same side of the boundary as themselves. In 2-D, this requirement would translate into triangles near border having acute angles opposite the border. In 3-D, I will sometimes call such tets *sufficiently acute*. Since the tetrahedra outside the domain are removed once the mesh is constructed, they are not required to be sufficiently acute. Boundary Voronoï cells are trimmed by the boundary for output, without requiring access to the deleted exterior tets.
So, the entire problem the algorithm and program must solve is this: build a conforming Delaunay tetrahedralization of the given PLC such that all of the interior tetrahedra adjacent to the input faces have their circumcenters on the same side of the face as themselves.

3 The Algorithm

3.1 Delaunay Tetrahedralization

The first step in solving the stated problem is to construct an unconstrained Delaunay tetrahedralization of the set of input points. I use the simple incremental point addition (Bowyer-Watson) algorithm, described, for example, in the book [7]. The search for the first hit tetrahedron is performed using a spatial binary tree, with each node subdividing the parent's region in half, and the axis of subdivision changing with each new level.
The most important numerical operation in this algorithm is the check whether the added point is inside a tet's circumsphere. For the program to be robust, it is essential for this check to always give the correct result. The computations performed are as follows:
Let $\mathbf{p}$ be the added point and let the tet's vertices be $\mathbf{0}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$ (we translate the origin into the first vertex). The vertices should be ordered so that $\mathbf{b} \cdot \mathbf{c} \times \mathbf{d} > 0$. Then, the point shall be strictly inside the tet's circumsphere if and only if

$$\begin{vmatrix} |\mathbf{b}|^2 & \mathbf{b}_x & \mathbf{b}_y & \mathbf{b}_z \\ |\mathbf{c}|^2 & \mathbf{c}_x & \mathbf{c}_y & \mathbf{c}_z \\ |\mathbf{d}|^2 & \mathbf{d}_x & \mathbf{d}_y & \mathbf{d}_z \\ |\mathbf{p}|^2 & \mathbf{p}_x & \mathbf{p}_y & \mathbf{p}_z \end{vmatrix} > 0. \tag{1}$$

So, in order to implement this check robustly, we must be able to determine the sign of an expression that depends on the coordinates of the input vertices and contains the operations of addition, subtraction and multiplication. A method for doing this is described in Section 4.

3.2 Conforming Tetrahedralization

Once the unconstrained Delaunay mesh is constructed, we must add points on the boundary in order to recover it and to make the adjacent tets in the interior sufficiently acute.

The following ideas provide the base from which the algorithm for this operation has been constructed. However, I do not have a formal proof of the algorithm's termination. The algorithm is listed in the pseudocode below.

The condition of the boundary Voronoï cells containing entire boundary means that the vertices owning these cells must be packed closely enough on the boundary, so that the cells of other vertices could not "pierce" their layer. In the terms of Delaunay mesh, for any triangle on the boundary to be present in the tetrahedralization, it is sufficient that its smallest circumscribed sphere does not contain any vertex inside it. This condition (*empty smallest circumsphere*) is also a necessary and sufficient one for the adjacent tets to be sufficiently acute. Therefore, if we add points on a boundary face so densely that its 2-D Delaunay triangles have empty smallest circumspheres, we will achieve the desired conforming tetrahedralization. But the radius of the smallest circumsphere is the circumradius of the Delaunay triangle; therefore, we can use a 2-D Delaunay refinement algorithm, such as Ruppert's [8], to minimize it!

Of course, the difficulty lies in refining the faces so that the refinement of one face does not interfere with the refinement of others. If all faces were disjoint, there would be no problem here: we would simply refine the triangulation on each face until the circumcircles of all triangles were small enough, e.g. smaller than the minimal distance between the faces. However, input faces usually have common edges and vertices, and interference between them may happen, especially between the faces at an acute dihedral angle to each other.

In 2-D, there is a similar problem. If we recover domain boundary, for example, by splitting each non-recovered segment at its midpoint, we may get infinite looping near domain corners with sharp angles. A solution is to split the segments adjacent to corners by putting points at some fixed distances from the corner – e.g. powers of 2. This way, points on the edges near corners will arrange themselves into "wheels" of some radius, and there will be no interference between edges near each corner even if they are at a sharp angle to each other. This notion of powers of two distances appears in a number of places: Ruppert's concentric shell splitting [8], Shewchuk's work [9], Pav's thesis [10], etc.

The solution in 3-D, it seems, could be similar; however, we need to construct such protecting structures both at the corners and the edges of the boundary, since the adjacent faces may meet at both.

For each corner, we can simply put points at a fixed distance from it on all incident edges and faces. It is possible to show that if these points are added on the faces densely enough, fans of triangles will appear near each corner with each triangle having empty smallest circumsphere, and, therefore, sufficiently acute adjacent tets. In order to avoid interference between faces through a common edge, the first and the last face point of each fan is situated so that it makes a fixed angle with the edge, for example, one the sine of which is a power of 2.

It remains to protect the common edges of the faces, away from the corners. Suppose we have already recovered one face, and are now processing a face that is adjacent to it via a small dihedral angle. Then let us copy some of the points from the already recovered face on the current face by *rotating them around the common edge*. If we do that at least with the first layer of points neighboring the edge, we will get identical triangulations on both faces near this common edge, and will, therefore, avoid interference and achieve conformation.

This description is, of course, incomplete, because I did not specify exactly which points are transferred between the faces and under which conditions. The precise formulation is contained in the algorithm listed below, but I am not sure it is a correct one, i.e. that it guarantees the algorithm to always terminate. That is the obstacle to proving the algorithm.

However, it seems that a provably correct algorithm can be constructed from the same basic ideas: 1) each face can be refined independently of all others until the circumcircles of its 2-D Delaunay triangles are small enough (while obeying the radii of the corners by refining the arcs around them); 2) interference between the faces through the common edges can be neutralized by transferring the points between faces through rotations

around edges. Note that when we add points to a face after transferring, the circumradii of the face's triangles *do not increase*.

Here is the algorithm I have implemented. **Bold** letters indicate a definition of a "routine", *italics* indicate a call to a routine. The entry point is *recovery of the domain*. The unconstrained mesh is supposed to be constructed at this point.

Recovery of the domain:
    compute radii of the corners;
    recover edges;
    recover faces.

Computation of the corner radii:
    for each corner of the domain:
        find its nearest neighbor point in the tetrahedralization;
        determine d = distance to it;
        determine the radius $R = 2^q$ such that $0.2d \le 2^q \le 0.4d$ (q is an integer)

Recovery of the edges:
    while any of the segments into which the domain edges are already subdivided *requires subdivision*:
        add a point on such *segment*;
        recalculate corner radii after adding a new point.

Check whether a segment of an edge requires subdivision:
    if it is absent from the tetrahedralization, then
        requires.
    Otherwise:
        for each of the PLC faces incident to the edge:
            find the triangle of the 2-D Delaunay triangulation on this face that is incident to the segment;
            if its angle opposite the segment is not acute, then the subdivision is required.
    Otherwise (if the segment is present and all adjacent 2-D Delaunay triangles have acute angles), the subdivision is not required.

Addition of a point on a segment:
    if both ends of the segment are domain corners, or if both aren't:
        put a point approximately in the middle;
    otherwise (if only one end is a corner):
        if the segment's length is larger than 3 radii of the corner,
            put a point approximately in the middle;
        otherwise
            put a point at the radius distance from the corner.
    (See Section 4.3 for details on adding a point approximately in the middle.)

Recalculation of corner radii after adding a new point:
for each of the neighbors of the newly added point:
if the neighbor is a corner of the PLC:
check if its distance to the new point is less than or equal to its radius;
if it is, reduce the radius by halving it repeatedly until it becomes strictly less than the distance to the added point.

Recovery of the faces:
repeat for all faces of the PLC until all are fully recovered:
Let *F* be the current face that needs recovery.
Repeat while *F* is not recovered:
recover all edges of the face;
recover triangles of the face (adding one point);
mark *F* as "once recovered"

Recovery of the face's edges:
while any of the segments into which the face's edges are already subdivided *requires subdivision*:
add a point on such *segment*;
recalculate corner radii after adding a new point.

(Note that this is similar to the *Recovery of the edges* function, but works only for one face.)

Recovery of triangles of the face (adding one point):
Find one of the following on the PLC face *F*:
a triangle that is present in the 2-D Delaunay triangulation of *F* but not in the current tetrahedralization,
or
a triangle on *F* that is present in the tetrahedralization but has adjacent tets the circumcenters of which are on a wrong side of *F* (check two tets for interior faces and one tet for boundary ones).
Add point on the found triangle.

Addition of point on triangle:
temporarily (so that it can be removed later) add a point into the tetrahedralization at the circumcenter of the triangle. Let this point be *P*.
For each point *Q* to which *P* becomes a neighbor:
if *Q* is a corner of the current face,
if *P* is closer to *Q* than the radius of *Q*,
cancel the addition of *P*;
add point P' on the arc around corner Q;

recalculate corner radii after adding a new point
return from function.
else, if Q belongs to an edge of the current face:
gather candidates for transferring by point Q.
Look among all candidates for transferring for the suitable one. If found, transfer and add it instead of P, otherwise confirm the addition of P.
Recalculate corner radii after adding a new point.

Addition of point on the arc around domain corner:
Let P be the point we were trying to add on the face F that has violated the radius of the F's corner Q.
Find the sector between the neighbors of Q belonging to F which was hit by P. Add P' on the arc of Q's radius in this sector. If this sector is an extreme one near F's border, put the point at a fixed angle from the border (the sine of which is a power of 2); if the sector is strictly inside the face, add P' approximately in the middle of it (see Section 4.3 about adding point approximately in the middle).

Gathering candidates for transferring:
Let Q be a point lying on the domain edge E that is incident to the face F.
For each neighbor vertex N of Q:
if it belongs to a PLC face $G \neq F$:
if G is incident to F via the edge E:
if G is marked as "once recovered":
select N as a candidate.

Looking for the suitable candidate for transferring:
Let P be the point we were trying to add on the face F and C be the total set of candidates for transferring gathered after adding P.
For each element of C, calculate the coordinates of the result of its transfer onto F and the distance of the result from P.
Choose the candidate the result of which gets nearest to P while obeying the following requirements: 1) it is strictly inside the circumcircle of the triangle for the refining of which we were adding P; 2) it does not violate the radii of the corners of F; 3) it is strictly inside F.
If none of the candidates obey all of these rules, do not select any.

In this algorithm, we initially subdivide the domain edges until 1) they all become connected, and 2) all triangles near edges in 2-D triangulations of the domain faces have acute angles opposite the edge. This guarantees that

any point added on a face at the circumcenter of a 2-D Delaunay triangle will be strictly inside this face.
The refinement of faces is performed by adding points at the circumcenters of those triangles that are not present in the tetrahedralization, or the adjacent tets of which are not sufficiently acute. If the added point is close to a corner or an edge of the face, we may, instead of adding this point, refine the fan of the corner, or transfer a point around the edge from an already recovered face. We add points on faces one by one; before adding each point we ensure that all triangles near the border of the face are acute.
Currently, the algorithm tries to arrange the points so that the first layer of a face's vertices near an edge is identical for all faces sharing this edge. The transferring of points around the edge, and the setting of fan vertices at fixed angles from the edge, both serve this purpose.
After the boundary is recovered, the program marks the tets outside the domain by passing over them with an advancing front, starting from domain boundary. This allows to check easily whether a point with given coordinates belongs to the domain's interior: simply locate the tet to which it belongs (using the spatial tree) and see if it is an interior one. It is possible to perform addition of points at the circumcenters of the interior tets. Because Voronoï cells of the boundary points contain the entire boundary, we may be sure that all of the interior tets have their circumcenters strictly inside the domain.

4 Numerical Precision

4.1 Numerical Precision and Robustness

The presented algorithm is very sensitive to numerical precision errors in several places: the main point-in-tet-circumsphere check, distance checks when updating corner radii, point-inside-polygon check, etc. For the code to be robust, these checks must be implemented so as to give the correct result in the maximum number of cases, desirably for all acceptable inputs. There exist methods for performing such checks robustly and efficiently when the input is integer or machine-precision floating-point [9][11][12]. Unfortunately, in my algorithm, the points added during the conformation stage may require more precision than a floating-point variable may contain, and may even be irrational. For example, the computation of a triangle's circumcenter coordinates involves division, and may, therefore, result in an infinite binary fraction. Some vertices that are introduced during the conformation process may have such coordinates, and the geometry

checks, ideally, should be able to cope with them. Even worse, rotating a point around an edge, or putting a point on a line at a fixed distance from a given point, may introduce square roots, and, therefore, irrational coordinates for the geometric predicates to handle.

I do not see any simple way to construct a robust conforming Delaunay algorithm that avoids placing points with such bad coordinates. At least, no limited-precision arithmetic seems to be sufficient as an easy solution to this problem. Such a limited-precision arithmetic (no matter fixed- or floating-point) will constrain the input points onto some grid. It is always possible to imagine a face or an edge the vertices of which belong to this grid, but none of the interior points do. If this face or edge needs to be recovered, it will not be possible to place a point exactly on it.

If we use a limited-precision arithmetic, the point will be placed a little away from the plane of the face. In this case, there is a possibility that a Delaunay tet will be constructed with one triangle on the face and the 4th vertex at the new point (for example, if the face is on external boundary and the point a little way inside the domain; in this case, even though the tet will have big circumradius, the circumscribing sphere may be almost entirely outside the domain, where there may be no points to block it).

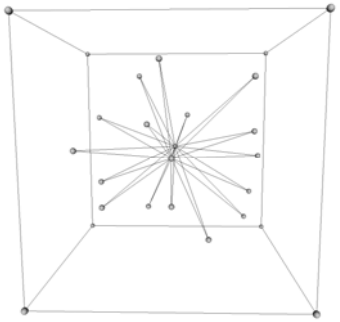

Fig. 9. PLC similar to hard case in [1]

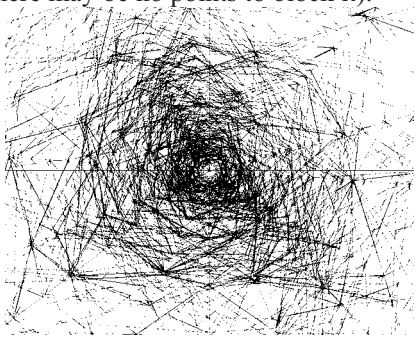

Fig. 10. Delaunay mesh

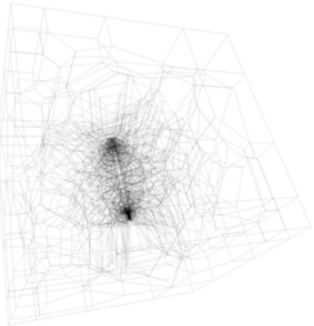

Fig. 11. Voronoï cells

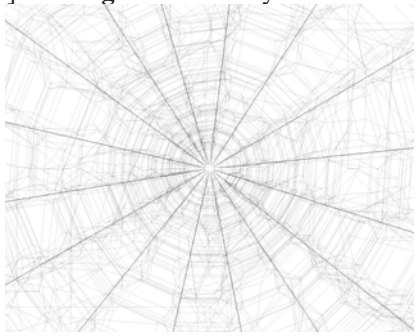

Fig. 12. Voronoï cells, another view

It is, perhaps, possible to handle such situations, for example, with some sliver removal technique; however, that would complicate things as the sliver removal might interfere with the conformation. Most importantly, placing the new points not exactly where they should go would make the code non-robust (on extreme inputs) by design. It would be difficult to track whether a given anomaly during the execution of the program was caused by a fault in the algorithm, an implementation bug or by the rounding of coordinates.

Because of all these difficulties with limited-precision arithmetic, I have implemented the presented algorithm using arbitrary-precision arithmetic. I believe the benefits of this approach to be these:

1. It allows to empirically verify the unproven algorithm. The arithmetic library may serve as a test base for experimenting with new strategies in meshing or other fields.
2. The code is almost totally robust by design (almost, because there remains one adjustable parameter for choosing speed or robustness in one extreme case).
3. The library may also be useful in other applications where a high level of robustness is sought for; it would probably help to save much work (compared to algorithm-specific modification) to achieve the required level of robustness. Also, the library may help reduce constraints on the input data (e.g., no limit on how different the size of the domain and the size of small features are), and possibly reduce the number of tweakable "epsilon" parameters.

A method for robust computation of *staged geometric predicates* has been recently presented by Shewchuk [11] and Nanevski, Blelloch and Harper [12]. Unlike the approach presented here, this method makes use of the hardware-supported floating-point numbers, by exploiting the properties of their roundoff errors. This allows to achieve a significantly better speed of computation than with the approach presented here; however, the following presents justifications for developing my method.

First, the staged geometric predicates operate on points with floating-point coordinates. This means that the expression the sign of which is evaluated by the predicate is fixed: the predicate is either hand-coded or generated from a given expression by a compiler. My approach, however, allows the coordinates of input points to be expressions themselves; it puts no limit on the level of nesting of operations. The expression the sign of which is being determined can be constructed during the execution of the program.

Second, the staged geometric predicates that operate with floating-point variables require that the exceptional conditions of overflow and underflow do not occur during the computation [12]. If they do occur, it is sug-

gested that the computation should be rerun in another, slower form of exact arithmetic. The presented library should be suitable exactly for that!

4.2 The Automatic Arbitrary-Precision Arithmetic Library

The following notes describe the implemented arbitrary-precision strategy for avoiding numerical error problems.

1. Use arbitrary-precision arithmetic for all computations.
2. Keep all results that can be reused (e.g., vertex coordinates) in memory, in the form of a graph of expressions. Each expression in this graph is linked to its arguments, which may be other expressions or constants. Each expression also stores its value in the numerical form, exactly or with some precision.
3. Make the arithmetic library able to resume computations, increasing the precision of any expression starting from the previously reached precision.
4. Make the library automatically choose the precision needed for the intermediate results, in order to get the requested precision for the final result.

The strategy has been implemented in the Automatic Arbitrary-Precision Arithmetic Library. It can compute, in accordance with the formulated rules, arbitrary expressions containing the 4 arithmetic operations, square roots, and finite-bit-length constants.

The input vertices of the mesh may be usual floating-point constants. They are converted and stored as the Library's finite-bit-length constants. The coordinates of vertices added during the conformation are expressions that depend on these input constants, and, possibly, on some previously constructed expressions. Many different expressions may point to the same expression as an argument, and, if the value of this argument expression has been computed to some precision, all of the depending expressions will be able to use it. The entire graph of expressions for vertex coordinates remains in memory while the mesh is being constructed. This is necessary since the addition of new vertices may require to increase the precision of already existing vertices. The expressions for performing geometry checks, such as the point-in-circumsphere check, are usually not reused, and, therefore, exist in memory only for the duration of the check. This applies only to the "top" part of the expression graph for a given check, e.g. to the multiplications and additions that comprise the determinant (1). The argument expressions (vertex coordinates) are those stored in the main, permanent, graph.

To perform a geometric check, the sign of some expression needs to be evaluated, for example, the sign of the determinant (1) for the point-in-circumsphere check. The program constructs the graph for this determinant and calls the Refine function for the top expression in the graph. The Refine function increases the precision of an expression by the given amount of bits. It first determines what precision the arguments must have in order to achieve the requested precision in the result. If the arguments are not constant, and their current precision is not sufficient, Refine is called for them, recursively, in order to achieve the required precision.

So, in order to know the sign of the determinant, we call Refine for its expression, repeatedly, increasing its precision by some amount of bits each time, until it becomes sufficient, i.e., until we get to a non-zero bit.

A special case is when the determinant is zero, and, because of divisions or roots, its numerical evaluation results in an infinite sequence of zero bits. In this case, we cannot reliably determine whether it is really zero, or just a very small positive or negative number, the leading "1" bit of which we have not yet reached. Currently, when the program reaches the 300^{th} bit after the decimal point without encountering any 1s, it accepts the number as zero; this is the only situation when the program's logic is not fully robust. This constant (-300) can be changed if necessary. It may also be possible to implement an analytical transformation of any expression showing such behavior that eliminates all divisions and roots and checks if it is in fact zero. My current implementation of this operation, however, has complexity proportional to 2^q, where q is the number of roots in the expression, and was disabled for performance reasons.

The Library is implemented with C++ classes. The basic class, `CPrecise`, stores a number (as an arbitrary-length array of bits). The finite-bit-length constants are stored as objects of this class. Expressions are implemented as classes inherited from `CPrecise`. In addition to the bit array (that can be expanded, if necessary, to accommodate the increasing precision), they store the pointers to their arguments, as well as the information about current precision. This precision information has the form of integer w such that

$$l_{cur} \le l_{ex} \le l_{cur} + 2^w, \tag{2}$$

where l_{cur} is the approximate absolute value currently stored in the bit array, and l_{ex} is absolute value of the exact number this expression represents. I call this w value the *error bit*.

When Refine function is called, for example, for an addition object, it checks the error bits of the arguments to see what precision we can reach with the current state of the arguments. E.g., if we are adding two positive

numbers, one having error bit w_1, and another w_2, we might truncate both arguments to the bit $w_{max} = \max(w_1, w_2)$, add them, and set the error bit of the result to $w_{max} + 2$, which can be shown to be sufficient for the result to conform to the inequality (2). From this, we can determine how much the arguments should be refined for the result to have the precision requested by the caller. Similar considerations are used with all other operations. The logic is contained in the Refine functions for all operations and is completely transparent to the user. The user only specifies by how many bits the end result should be refined; the Library refines all intermediate values as necessary, choosing their precision automatically.

The Library also handles a special case of numbers that are so close to zero that their signs are not yet known. For such an "around zero" number, the error bit is specified so that the number is inside the $[-2^w, 2^w]$ segment. Refine functions for all operations perform special handling if any of the arguments is an "around zero" number. When evaluating the sign of determinant, we actually call the Refine function repeatedly until the determinant's expression becomes a regular, non-"around zero" number.

The implementation of all 5 operations and the handling of all special cases resulted in a quite big and complex code. The Library has built-in self-verification (for debug builds only), that, when turned on, verifies all results and error bounds with simple bitwise checks. At the time of writing there are no known bugs; however, ***the Library should not be used in any application the reliability of which has critical effect in the real world***.

4.3 Performance and Memory Use

If only a fixed-precision (or the common floating-point) arithmetic was used, each vertex of the mesh would store its coordinates in a few bytes, e.g. 8 per coordinate. With the Library, the coordinates of vertices are the nodes in the expression graph, which also contains all of the intermediate values. Each vertex, however, adds only a fixed number of expressions to the graph (exactly how many, depends on what kind of vertex it is: one added at a circumcenter, or on an edge, or on a circle around corner, etc.). Therefore, the number of nodes in the graph grows linearly with the number of points added during the conformation.

Each node, however, stores an arbitrary-length array of bits. In order to understand how the presented approach will scale with increasing problem size, we need to consider the impact of this increase on the precision with which the intermediate results shall have to be computed. It is very important that this precision does not grow too fast, as the algorithms for arbitrary-precision arithmetic may have bad asymptotic properties in relation

to requested precision (e.g. $O(N^2)$ for the current implementation of multiplication).

When the sign of an expression needs to be determined, this expression is refined until either its leading "1" bit, or the 300^{th} bit after the decimal point, is reached. This means that we'll need to perform only a fixed amount of work for the top expression in the graph. The total amount of work, however, will depend on how many bits will need to be computed in the intermediate expressions.

For addition and subtraction, in order to refine the result down to a given bit, each of the arguments must be refined down to that bit and by 2 bits more. For multiplication, the arguments must each have the number of significant bits roughly equal to the required amount of significant bits in the result. For the reciprocal ($1/x$), the argument must have, roughly, the same amount of significant bits as the result. The argument of the square root must have about 2 times more significant bits than the result.

This means that the amount of bits needed to be calculated for intermediate expressions may grow with the increasing depth of dependence in the graph. However, the fastest, really prohibitive, growth happens only with square roots; luckily, the algorithm does not require any nesting of square roots (they appear only in unit edge vectors and normals of the input PLC, and when putting point at an angle with a power of 2 sine).

Note that in the algorithm pseudocode it is said that a point is sometimes added approximately in the middle of an edge or an arc. This is done in order to reduce the nested dependence between points. Suppose, for example, that an edge of the PLC has the points **a** and **b** at the ends, and the points $\mathbf{p}_1$ and $\mathbf{p}_2$ already added on it. Suppose also that we need to subdivide the edge further by adding a point between $\mathbf{p}_1$ and $\mathbf{p}_2$. If we put it exactly at midpoint, i.e. at $0.5\ (\mathbf{p}_1 + \mathbf{p}_2)$, we will introduce a dependence of the new point on $\mathbf{p}_1$ and $\mathbf{p}_2$ (that, in their own turn, probably depend already on the edge ends **a** and **b**). If the point can be placed not precisely in the middle between $\mathbf{p}_1$ and $\mathbf{p}_2$, we can compute its coordinates as $\mathbf{a} + t\ (\mathbf{b} - \mathbf{a})$ with some finite-precision constant t, and thus avoid one level of dependence. Similar considerations are used when subdividing protecting arcs around PLC vertices (instead of making the new point dependent on its neighbors on the arc, we add the point at an angle with finite-precision sine or cosine from edge).

The level of dependence for other operations can grow with increasing problem size. This happens in 2 cases: (a) the circumcenter of a triangle depends on its 3 corner vertices; (b) a vertex transferred from another face by rotation around edge depends on the original vertex. It seems that the growth in the level of nesting of operations shouldn't be too fast in prac-

tice. Note that, if the level of nesting of circumcenters is increased by 1, the amount of points that can be placed on the face is, roughly, tripled. Also, perhaps, the (a) dependence can be totally avoided by placing points not exactly at the circumcenters (but exactly on the face!), making them dependent only on the PLC corners. The (b) dependence, it seems, cannot be completely avoided, but it is unlikely to be common that a vertex jumps consecutively around many different edges.

These considerations show that the presented approach is likely to scale well with increasing problem size in practice. That, however, has not been verified, as the current implementation suffers from an extremely inefficient ($O(N^2)$-like) search for unconformed triangles and edges (avoiding it poses no theoretical problems, but has not yet been implemented).

References

1. Jonathan Richard Shewchuk (2002) Constrained Delaunay Tetrahedralizations and Provably Good Boundary Recovery. In: Proceedings of the 11th International Meshing Roundtable 193-204
2. Michael Murphy, David M. Mount, Carl W. Gable (2000) A Point-Placement Strategy for Conforming Delaunay Tetrahedralization. In: Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms 67-74
3. David Cohen-Steiner, Éric Colin de Verdière, Mariette Yvinec (2002) Conforming Delaunay Triangulations in 3D. In: Proceedings of the 18th ACM Symposium on Computational Geometry 199-208
4. Siu-Wing Cheng, Sheung-Hung Poon (2003) Graded Conforming Delaunay Tetrahedralization with Bounded Radius-Edge Ratio. In: Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms 295-304
5. Steven E. Pav, Noel J. Walkington (2004) Robust Three Dimensional Delaunay Refinement. In: Proceedings of the 13th International Meshing Roundtable
6. Siu-Wing Cheng, Tamal K. Dey, Edgar A. Ramos, Tathagata Ray (2004) Quality Meshing for Polyhedra with Small Angles. In: Proceedings of the 20th Annual ACM Symposium on Computational Geometry 290-299
7. Timothy J. Baker (1999) Delaunay-Voronoï Methods. In: Joe F. Thompson, Bharat K. Soni, Nigel P. Weatherill (eds) Handbook of Grid Generation 16. CRC Press, Boca Raton London New York Washington, D.C.
8 Jim Ruppert (1995) A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation. In: J. Algorithms 18,3:548-585
9 Jonathan Richard Shewchuk (1997) Delaunay Refinement Mesh Generation. PhD Thesis, Carnegie Mellon University

10 Steven Elliot Pav (2003) Delaunay Refinement Algorithms. PhD Thesis, Carnegie Mellon University
11 Jonathan Richard Shewchuk (1996) Robust Adaptive Floating-Point Geometric Predicates. In: Proceedings of the 12th Annual Symposium on Computational Geometry
12 Aleksandar Nanevski, Guy Blelloch, Robert Harper (2003) Automatic Generation of Staged Geometric Predicates. In: Higher-Order and Symbolic Computation, vol. 16, issue 4, 379-400