On the Implementation of a Delaunay-based 3-dimensional Mesh Generator

K.J. van der Kolk and N.P. van der Meijs

Delft University of Technology, EEMCS, Circuits and Systems Group,
Mekelweg 4, NL-2628 CD Delft
{keesjan,nick}@cas.et.tudelft.nl

1 Introduction

A typical problem in engineering is to find a numerical solution to a partial differential equation (or a coupled set thereof), given a number of boundary conditions, and the usual approach of solving the problem starts by discretizing the domain into elementary volumes. In this paper, we focus on mesh generation suitable for the solution of field problems in arbitrary VLSI structures. We assume that the problem cannot be easily reduced to a lower dimensionality by exploiting symmetry or regularity, so that the problem-domain is intrinsically three-dimensional. Also, we assume that the selected numerical technique (e.g., the finite element method) requires a three-dimensional discretization (as opposed to a surface-discretization).
Surveys on mesh generation are given in [2] and [5]. The mesh generator described in this paper is based on techniques from the Delaunay-based mesh generation literature. The main benefit of these techniques is that the quality of the resulting meshes can be *guaranteed*, and, equally important, that the meshes are still small enough to be *practically useful*. An additional advantage is that computation of the mesh is efficient in practice. In general, mesh computation is much faster than solving the subsequent numerical problems. An example mesh generated by our implementation is shown in Figure 1.
Although the principles of Delaunay-based mesh generation are well-known, implementing a mesh generator of this kind is a real challenge. Besides the fact that coding the topological manipulation of the three-dimensional basic elements (tetrahedra in our case) is tedious, one must make sure that the algorithm is robust against floating point errors. Furthermore, the Delaunay-based mesh generation theory allows fast construction of the mesh using only local operations, and it is not trivial to exploit this fact.

2 Delaunay Refinement

Our mesh-generator follows the traditional approach of Delaunay-based mesh refinement. Limitation of space prohibits us to give a full account of the method and its theoretical underpinnings, hence we are obliged to refer the reader to the literature. See, e.g., [5], or [10] for especially good expositions.
For later reference, the global Delaunay refinement algorithm is depicted in Figure 2. The terms SPLIT_1, SPLIT_2, and SPLIT_3 denote the operations of subsegment, subfacet and tetrahedron splitting, respectively. Given a tetrahedron t, $\gamma(t)$ denotes its circumradius-to-shortest-edge ratio. Further, the constant B reflects the "minimum tetrahedron quality," and should be selected greater than 2.

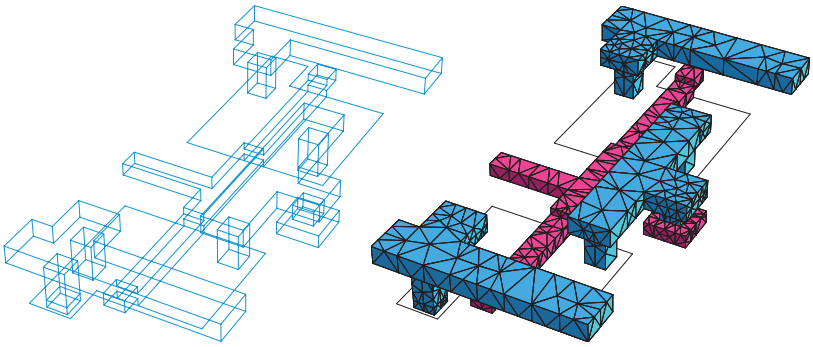

Fig. 1: Example PLC and corresponding mesh. The structure is contained in a bounding box (not shown) and the exterior of the structure is meshed as well in this case.

The input of the algorithm is restricted to a piecewise linear complex (PLC) with angles greater than or equal to $\pi/2$ (this holds both for inter-edge angles and dihedral angles). In our case of VLSI structures, this forms no real limitation. In cases where smaller input-angles are unavoidable, a remedy is to clip-off the offending angles, thereby introducing a slight modification of the original geometry, but one could even mesh the clipped domains separately, in some cases.

3 Delaunay Tetrahedrization

An essential data-structure maintained by the mesh-generator is the Delaunay tetrahedrization (abbreviated DT; we use the same abbreviation for the Delaunay *triangulation*, but add the prefix '2d' or '3d' when confusion may occur) [4]. The DT is constructed incrementally: we start with a basic DT (in fact a single tetrahedron), and as new points are added to the mesh, the representation of the DT is updated. Mathematical tools described in Section 4 guarantee that the DT is uniquely defined.
For incremental point-insertion, we use the Bowyer-Watson method [3, 11]. In essence, given a point p to be inserted into the DT, the method computes the so called Bowyer-Watson polyhedron, which is the union of tetrahedra which have p in their circumsphere (it can be shown that this polyhedron can be computed in $\mathcal{O}(t)$, where t is the number of tetrahedra in the polyhedron; a simple breadth-first search can be used). Then, the polyhedron is emptied (its constituent tetrahedra are removed from the mesh), and new tetrahedra are formed between p and the triangular faces of the polyhedron. It is theoretically guaranteed that the resulting complex is the DT.
Note that the initial DT, consisting exclusively of the points in the input PLC, can be constructed more efficiently by using a sweepline algorithm, as mentioned in [10].

```
while True:
    (STEP 1) if some subsegment s is encroached:
        SPLIT_1 s

    else (STEP 2) if some subfacet u is encroached:
        let c be the circumcenter of u

        if c encroaches upon a subsegment s:
            SPLIT_1 s
        else:
            SPLIT_2 u

    else (STEP 3) if there exists a tetrahedron t with γ(t) > B:
        let c be the circumcenter of t

        if c encroaches upon some subsegment s:
            SPLIT_1 s
        else if c encroaches upon some subfacet u:
            SPLIT_2 u
        else:
            SPLIT_3 t
    else:
        break
```

Fig. 2: Pseudo-code for the three-dimensional mesh-refinement algorithm.

4 Geometric Predicates

The mesh generator depends on geometric predicates for making elementary decisions. Essentially, geometric predicates form the connection between topological information (how elements are connected together) and geometric information (where elements are physically located). Reducing the number of geometrical predicates to a minimum creates the best opportunities for making our algorithm robust w.r.t. degeneracies and round-off error.
We require only two different predicates. One predicate, ORIENT3D, determines the relative orientation of four points in 3d space. Another predicate, INSPHERE, determines, given four points in 3d space, whether a fifth points lies inside or outside the circumscribing sphere of those four points. Both predicates can be computed by evaluating the sign of a determinant. See [10] for precise definitions of these predicates. Note that the algorithm does not utilize two-dimensional equivalents of the predicates (see also Section 8).
In order to uniquely define the Delaunay tetrahedrization, it is necessary and sufficient that any degeneracies are expelled from the geometric predicates. This means that the corresponding determinants must be either positive or negative, but never exactly zero.
We implement non-degeneracy by using the Simulation of Simplicity (SoS) method devised by Edelsbrunner and Mücke [6]. However, it would be highly inefficient to use the SoS method directly for all geometric predicates that need to be evaluated, since the method relies on exact arithmetic. Therefore, we compute first, for every predicate we encounter, the sign of the determinant without the symbolic perturbation exerted by SoS. Only if a degeneracy is found, we resort to SoS.
For the computation of regular signs of determinants, we rely on Shewchuk's adaptive floating point predicate library [9]. For SoS, we have implemented a module capable of performing the necessary symbolic manipulations, where the handling of exact floating-point arithmetic is delegated to the GNU multiprecision library (`libgmp` [1]). In order to reduce the probability of degeneracies, we (physically) perturb the inserted points slightly. The result is that in practice, the SoS-module is used only in a small fraction of the cases, and thus its efficiency

is marginal. Most of the optimizations mentioned in [6] can be ignored (at least in our case) since they do not result in any appreciable advantage.

5 Elementary Data-structures

A fair amount of research was needed to come up with the actual data-structures to be used in the mesh-generator. Of course, having proper data-structures is of paramount importance to the efficiency of the algorithm. Here we give a overview of the types of object that the mesh generator handles. This overview is a necessary aid in understanding the algorithm and in seeing why it is efficient.
For every point inserted in the mesh, a node-object is allocated, which contains the physical x, y, z-coordinates of the point. The address of the node is used as the perturbation-index for the SoS method (see Section 4). All other objects (subsegments, subfacets, tetrahedra, etc.) refer to a point by a pointer to the corresponding node-object. Every subsegment-object records an (arbitrarily large) set of subfacet-objects to which it is attached (its 'wings'). Every subfacet records a set of (at most two) abutting tetrahedra. Each subfacet contains three pointers to neighboring subfacets (a pointer can be null in case a neighbor does not exist). Similarly, every tetrahedron contains four pointers to its neighbors.
As a local optimization, a subfacet stores the orientation of each neighboring subfacet, by storing the neighbor's edge-number. Similarly, a tetrahedron stores the orientation of each neighboring tetrahedron, by storing its face-number and by storing the orientation of that face. Note however, that this information can be easily obtained by a local topological investigation based on node-comparisons.

6 Detached Elements

In the final mesh, every subfacet has two abutting tetrahedra associated with it (assuming the subfacet is not at the boundary of the mesh). However, during mesh refinement, a subfacet may of course be detached from its two potentially abutting tetrahedra.
We keep subfacets attached to tetrahedra as much as possible. Thus, when inserting a point into the mesh (modifying the three-dimensional DT, and/or two-dimensional DTs), we examine every face of every *modified* tetrahedron, and see if it matches some subfacet which is in a detached state. If we find such a subfacet, we simply attach it to the tetrahedron. The table used for attaching subfacets and tetrahedra is implemented as a hash-table. Indexing is done using the set of the three corresponding node pointers.
A similar mechanism is used to attach subsegments to subfacets. In our implementation, we also explicitly attach subsegments to tetrahedra (thus requiring an extra table).

7 Encroached Elements

Throughout the refinement process, certain subsegments and subfacets may become encroached [8, 10] and they remain encroached until the corresponding subsegment or subfacet is split (note that a subfacet may also disappear from the mesh due to some other splitting).
Subfacets which are (possibly) encroached are kept in a list. Any time a subfacet may have become encroached (this happens during point-insertion), we add it to the list. It turns out that the Bowyer-Watson insertion-polyhedron (see Section 3) used during point-insertion contains exactly those subfacets which may have become encroached by the point to be inserted, thus the amount of elements to be inserted into the encroachment-list is limited to the neighborhood of the given point.
The actual encroachment-test is delayed until the elements are fetched from the encroachment-list: whenever we need to select an arbitrary encroached subfacet, we pick one from the

encroachment-list, and we check whether the subfacet still exists, and whether it is indeed encroached; encroachment is easily detected by examining the apices of the two tetrahedra abutting the subfacet. It remains to be said that for subsegments, the approach is similar.

8 Representation of Facets

For every facet of the PLC, we need to keep a separate (2d) Delaunay triangulation (all nodes contained in the facet are inserted into both the corresponding 2d DT and the global 3d DT, but not vice versa). Each 2d DT is actually implemented using three-dimensional predicates (we define an auxiliary node at some distance from the plane of the facet, and use this node to augment the set of parameters of our three-dimensional predicates).
Most operations on 2d DT's are straighforward modifications of the corresponding operations on 3d DT's. In 2d DT's, however, we remove subfacets at the exterior of the facet, for efficiency. The boundaries of a facet thus form the constraining edges in a constrained Delaunay triangulation (CDT), see, e.g., [5]. Using the fact that these edges form a set of contiguous boundaries, one can show that (in 2d) the Bowyer-Watson insertion scheme is still applicable and retains its efficiency (with only some trivial modifications).

9 Zone Bookkeeping

For each tetrahedron, we record whether it is inside, or outside the part of the domain to be meshed, and this information is updated dynamically. Whenever it is unknown in what part of the domain a tetrahedron resides, its zone is marked as 'unknown'.
If, when querying the zone of a tetrahedron τ, its zone is not 'unknown', we can be sure that we have the correct zone. Otherwise, we perform a straight walk through the mesh towards some point at infinity, and we count the number of walls crossed. When we run into the boundary of the mesh, or if we encounter some tetrahedron with a known zone, we can reconstruct the zone information of τ. After a query, in order to optimize future queries, we record the zone information with τ, and also recursively with its neightbors.
Note from Figure 2 that the concept of the zone in which a tetrahedron lies is properly defined only at STEP 3, since then all subfacets are unencroached and guaranteed to be part of the mesh (in this case no tetrahedron can pierce through a wall of the domain). Fortunately, we need the zone-information exclusively in this step: when splitting skinny tetrahedra, we need to consider just the tetrahedra which are on the 'inside' of the domain, as refinement of the exterior would be wasteful.
Note that in the context of physical VLSI layout, the domain is often bounded by a large cuboid; in that case, determining the zone of a tetrahedron is quite simple. However, when analyzing a design in parts, one can easily end up with different types of geometry.

10 Point-location

We frequently need to determine the tetrahedron in which a given point lies. For example, when splitting a tetrahedron τ_1, we need to insert a node into the tetrahedron τ_2 containing the circumcenter of τ_1. Finding a tetrahedron containing a given point, or 'point-location', is implemented using a linear walk (see, e.g., [7]).
A linear walk takes time proportional to the number of tetrahedra visited, and thus the main trick is to always make sure that the initial tetrahedron is topologically close to the final tetrahedron. In the case of splitting a skinny tetrahedron, an obvious candidate for the initial tetrahedron is of course the skinny one, and fortunately, in practice, with this choice one observes that the amount of intermediate tetrahedra visited is generally below a small constant.

There are other situations in which point-location is required. For example, when splitting a subfacet f, we need to find the tetrahedron containing the circumcenter of f (because we need to insert a node into it). Now since the subfacet is not guaranteed to be attached to a tetrahedron (in fact, it is encroached so it is likely not attached), we have no simple initial tetrahedron. To remedy this, we record with each detached subfacet a tetrahedron which is in its proximity (basically, the tetrahedron to which it was attached most recently) so that we can use this tetrahedron as our starting point. Of course, once a subfacet is detached, it can 'drift' away from this tetrahedron (due to insertions elsewhere), but practice shows that point-location still approximately takes constant time. Intuitively, the effect of drifting is only small, since all operations on the mesh are designed to be as localized as possible for additional efficiency reasons.
We also need point-location in two dimensions, i.e., we need the ability to find the subfacet in which a given point lies. The operation is similar to that described above, except that in 2d, the elements at the exterior of the domain are not present. Thus, a linear walk may prematurely halt at the border of the domain. In such cases, we perform an additional breadth-first search from the subfacet at which the linear walk ended. This search is expected to be relatively inexpensive (practice confirms this) since we already assume that the starting subfacet is close to the target subfacet.

11 Conclusion

Although limitation of space precluded an in-depth treatment, we have described the most important ingredients of an efficient three-dimensional mesh generator. Our implementation works and produces around 15.000 tetrahedra per second on a 2.4GHz Intel-based workstation. The reader may obtain a copy of the implementation by contacting the authors.

References

1. The GNU multi-precision arithmetic library, `http://www.swox.com/gmp`.
2. Bern and Eppstein. Mesh generation and optimal triangulation. In *Computing in Euclidean Geometry, Edited by Ding-Zhu Du and Frank Hwang, World Scientific, Lecture Notes Series on Computing – Vol. 1*. 1992.
3. A. Bowyer. Computing Dirichlet tessellations. *Computer J.*, 24:162–166, 1981.
4. B. Delaunay. Sur la sphère vide. *Bull. Acad. Sci. USSR(VII)*, pages 793–800, 1934. Classe Sci. Mat. Nat.
5. H. Edelsbrunner. *Geometry and Topology for Mesh Generation.* Cambridge University Press, 2001.
6. H. Edelsbrunner and E. P. Mücke. Simulation of simplicity, a technique to cope with degenerate cases in geometric computations. *ACM Trans. Graphics*, 9:66–104, 1990.
7. E. P. Mücke, I. Saias, and B. Zhu. Fast randomized point location without preprocessing in two- and three-dimensional delaunay triangulations. In *Proceedings of the 11th Annual Symposium on Computational Geometry*, pages 274–283, 1996.
8. J. Ruppert. A new and simple algorithm for quality 2-dimensional mesh generation. In *Proc. 4th ACM-SIAM Symp. on Disc. Algorithms*, pages 83–92, 1993.
9. Jonathan Richard Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18(3):305–363, October 1997.
10. Jonathan Richard Shewchuk. *Delaunay refinement mesh generation*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.
11. D. F. Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *Computer J.*, 24:167–171, 1981.