Constrained Delaunay Tetrahedrization: A Robust and Practical Approach

LORENZO DIAZZI, UniMoRe, Italy IMATI - CNR, Italy
DANIELE PANOZZO, New York University, USA
AMIR VAXMAN, The University of Edinburgh, UK
MARCO ATTENE, IMATI - CNR, Italy

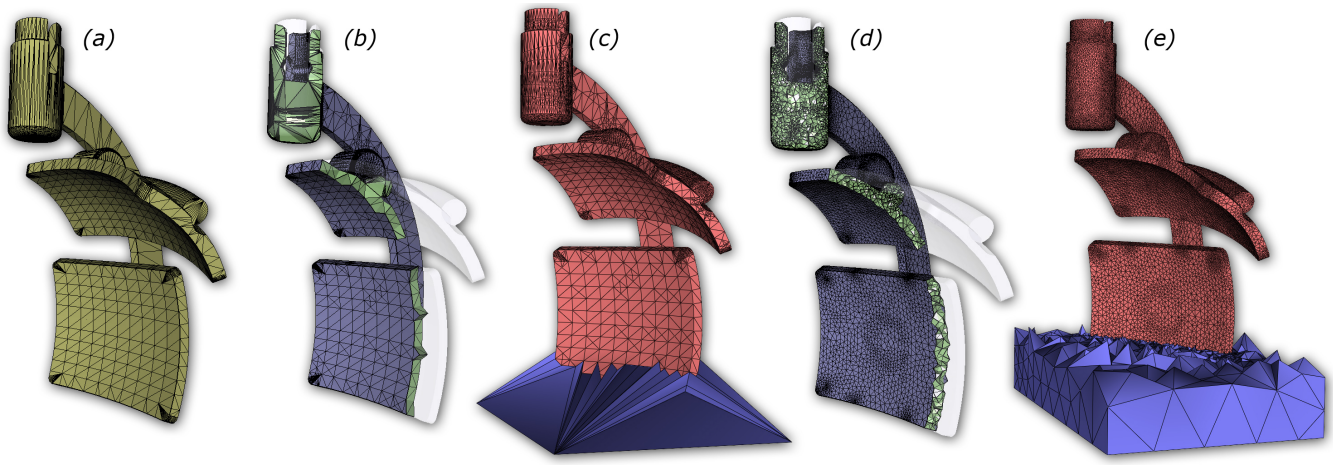


Fig. 1. A valid model in the Thingi10k dataset (a) is enriched with Steiner points and tetrahedrized by our method (b, c). The resulting tetrahedrization is then used as a starting point for a standard mesh optimization process (d, e).

We present a numerically robust algorithm for computing the constrained Delaunay tetrahedrization (CDT) of a piecewise-linear complex, which has a 100% success rate on the 4408 valid models in the Thingi10k dataset.

We build on the underlying theory of the well-known `tetgen` software, but use a floating-point implementation based on indirect geometric predicates to implicitly represent Steiner points: this new approach dramatically simplifies the implementation, removing the need for ad-hoc tolerances in geometric operations. Our approach leads to a robust and parameter-free implementation, with an empirically manageable number of added Steiner points. Furthermore, our algorithm addresses a major gap in `tetgen`'s theory which may lead to algorithmic failure on valid models, even when assuming perfect precision in the calculations.

Our output tetrahedrization conforms with the input geometry without approximations. We can further round our output to floating-point coordinates for downstream applications, which almost always results in valid floating-point meshes unless the input triangulation is very close to being degenerate.

CCS Concepts: • **Computing methodologies** → **Mesh models**; **Mesh geometry models**; *Shape analysis*.

Additional Key Words and Phrases: volume meshing, numeric robustness, representability

ACM Reference Format:
Lorenzo Diazzi, Daniele Panozzo, Amir Vaxman, and Marco Attene. 2023. Constrained Delaunay Tetrahedrization: A Robust and Practical Approach. 1, 1 (September 2023), 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A Constrained Delaunay Tetrahedrization (CDT) is a compact space subdivision widely used for diverse computer graphics and engineering applications, including the solution of partial differential equations [Si 2008], calculation of shape thickness [Cabiddu and Attene 2017], medial axis approximation [Dey and Zhao 2003], ray tracing [Lagae and Dutre 2008], and many other algorithms requiring an explicit discretization of a volume. Preserving the boundary of the input surface in tetrahedral meshing is often critical, especially for solving PDEs on complex geometric domains, as this allows to adhere to the boundary constraints exactly without loss of accuracy. Boundary-*approximating* tetrahedral meshing algorithms (such as TetWild [Hu et al. 2018], Quartet [Labelle and Shewchuk 2007][Bridson and Doran 2014], or CGAL [Fabri and Pion 2009]) need to rely on lossy and potentially not bijective projections to map boundary

Authors' addresses: Lorenzo Diazzi, UniMoRe, Italy and IMATI - CNR, Italy, lorenzo.diazzi@unimore.it; Daniele Panozzo, New York University, USA, panozzo@nyu.edu; Amir Vaxman, The University of Edinburgh, UK, avaxman@inf.ed.ac.uk; Marco Attene, IMATI - CNR, Italy, marco.attene@ge.imati.cnr.it.

XXXX-XXXX/2023/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

conditions, which makes them unsuitable for such applications. Our algorithm by definition produces a boundary-preserving CDT.

Constrained Delaunay Triangulations. Roughly speaking, among all the possible triangulations, the CDT is the one that satisfies the Delaunay criterion *as much as possible* while containing all the input segments (in 2D) and/or facets (in 3D) [Alexa 2020]. In 2D, a collection of non-intersecting and non-degenerate segments always admits a valid CDT. This is not the case in 3D, where the existence of a CDT is no longer guaranteed unless we augment the input Piecewise-Linear Complex (PLC) with a set of additional Steiner points [Murphy et al. 2001; Shewchuk 2002; Si and Gärtner 2005]. Unfortunately, determining the number and the optimal position of these points is still an open problem [Ruppert and Seidel 1992].

Existing Algorithms. Provably-correct algorithms (Sect. 2) exist that first augment a PLC with Steiner points, and then calculate a valid CDT. Most of these algorithms are, however, impractical since they introduce a large number of Steiner points. [Si and Gärtner 2005] made a major theoretical and practical step forward in computing CDTs by introducing a new algorithm which, in practical cases, introduces a small number of Steiner points. The popular software `tetgen` is a floating-point implementation of this algorithm. To the best of our knowledge, `tetgen` is the only publicly-available implementation of an algorithm to compute a CDT in 3D; while well-engineered, it fails to produce a CDT on around 8.5% of the valid models in the Thingi10k dataset. Fig. 1 depicts an example model that makes `tetgen` crash, whereas our method processes it correctly.

Analyzing failure in `tetgen`. While it is intuitive to assume that the said failures in `tetgen` are due to numerical tolerance used in their floating-point implementation, this is not always the case. In Sec. 3.5 we introduce and analyze a theoretical issue that results from a tacit and incorrect assumption in the theory of [Si and Gärtner 2005]. We validated this using an exact-number implementation (CORE library [Burnikel et al. 1996; Karamcheti et al. 1999]).

Contribution. We introduce a novel algorithm for computing the CDT of a valid PLC that is provably correct and can be implemented based on hardware-accelerated floating-point computation, without sacrificing robustness. Our two main contributions are:

(1) A novel algorithm, based on a combination of [Si and Gärtner 2005] and [Shewchuk 2000b], that avoids the tacit assumption that all the local cavities formed to insert PLC faces can be sufficiently expanded (Sec. 3.5). A key property of our algorithm, which is crucial for the second contribution, is that it does not require using irrational coordinates for Steiner points.
(2) An exact and efficient implementation of our algorithm based on indirect floating-point predicates [Attene 2020]. We define a new type of implicit point to represent the Steiner vertices (Sec. 4.3), extend the classical orient3D and inSphere predicates to handle this new point type (Sec. 4.3.1), and show that these new predicates are sufficient to implement all the non-trivial checks required by the combined algorithm (App. B).

To the best of our knowledge, our algorithm is the first that successfully computes the CDT for **all** the 4408 valid PLCs in the Thingi10k dataset. Our implementation combines theoretical correctness, practical robustness, and efficiency thanks to a safe use of floating point arithmetic. Our prototype can process all the said 4408 models in approximately 5 hours on a single CPU core on a standard desktop PC. Our reference implementation is available as an open-source project (Sec. 6) to foster the adoption of our algorithm and support the many applications which use CDT computation as an internal routine.

2 RELATED WORK

Subdividing a volumetric domain with tetrahedra has been extensively studied in the last decades. We refer the reader to [Hu et al. 2020, 2018] for an elaborate literature study, and here focus on the relevant works that mostly target conformity (as much as possible) to valid input boundary surfaces.

2.1 Delaunay meshing

Delaunay tetrahedrization. The Delaunay tetrahedrization (DT) of a point set in space can be calculated using incremental insertion algorithms [Bowyer 1981] [Watson 1981] whose output satisfies the so-called *Delaunay criterion*: the interior of the circumsphere of any tetrahedron does not contain any point of the input point set. Incremental insertion algorithms can be implemented both robustly and efficiently [Fabri and Pion 2009], and modern versions can also be parallelized [Marot et al. 2019]. When the input is a polyhedral surface, one may compute the Delaunay tetrahedrization of its vertices, but the edges and facets are not necessarily represented.

Constrained Delaunay. Given a polygon in 2D, one can calculate its *constrained* Delaunay triangulation [Lee and Lin 1986] [Chew 1989], where both vertices and edges are represented. Unfortunately obtaining a similar result for 3D polyhedra is substantially more difficult because, as previously mentioned, not all polyhedra admit a CDT, and additional Steiner points may be necessary to obtain a useful result. Some pioneering results in this area were obtained in [Joe 1991], where the idea was to first subdivide the polyhedron into convex regions whose DT also conforms with the facets. Then individual DTs can be simply merged, but the initial subdivision turned out to be particularly difficult. [George et al. 1991] and [Weatherill and Hassan 1994] introduced an effective approach which inspired many subsequent works. After having computed the DT of the vertices, this approach first recovers the input segments and then reconstructs the facets in a second phase. This was subsequently adopted in [Guan et al. 2006], where an empirically smaller number of Steiner points was used. [Shewchuk 2002] introduced a provably correct algorithm for computing the CDT by *protecting* acute vertices, which amounts to splitting all edges incident to such vertices. Inspired by this work, [Si and Gärtner 2005] presented an alternative approach that still protects the vertices, but employs a significantly smaller number of Steiner points to do that. The famous `tetgen` software implementing this approach is the state of the art for the calculation of CDTs in 3D. We compare against it in Section 6.

Conforming Delaunay. Constrained tetrahedrizations put in as few Steiner points as possible to make the tetrahedrization feasible. However, the elements are not guaranteed to be Delaunay everywhere. Conversely, conforming tetrahedrizations (e.g., [Cohen-Steiner et al. 2002; Murphy et al. 2001]) generate Delaunay tetrahedrizations that conform to a refinement of the boundary to subfaces and subedges, while introducing potentially many Steiner points, and consequently many tetrahedra. Recently, Alexa [2020] observed that in some cases weights can be assigned to the input vertices so that their weighted Delaunay tetrahedrization contains the input simplices with no need of Steiner points. However, such a set of weights may not exist and, when it does, calculating it is impractically slow.

Delaunay refinement. Given an initial tetrahedrization, one can successively improve it by introducing new vertices at the centers of circumscribing spheres of bad tetrahedra (e.g., [Jamin et al. 2015; Ruppert 1995; Shewchuk 1998]). Such methods are successfully implemented in CGAL Mesh Generation package [Rineau and Yvinec 2007] and in `Tetgen` [Shewchuk and Si 2014] (see Sec. 5.2). While guaranteeing termination, these methods often admit *slivers*, which are tets with nominally good radius-edge-length ratios, but close to degenerate in terms of volume. Approaches to remove slivers include relaxation [Alliez et al. 2005; Du and Wang 2003] and perturbation [Tournois et al. 2009]. We note that a recent method [Alexa 2019] computes *harmonic* triangulations that minimize the Dirichlet energy, rather than Delaunay, noting that these properties are equivalent in 2D. We compare against CGAL in Section 6.

2.2 Non-Delaunay Tetrahedral meshing

Grid based. A class of methods establishes a uniform grid, or an adaptive octree around an object, which is simple to tetrahedrize. To conform to a boundary surface that is not grid aligned, some methods cut the existing grid cells and tetrahedrize the intersection [Bridson and Doran 2014; Bronson et al. 2013; Doran et al. 2013; Labelle and Shewchuk 2007], whereas some deform the grid to match the original boundary [Molino et al. 2003]. Some guarantees on average element quality exist for low-curvature objects that are sufficiently convex (that is, have a high volume-to-surface ratio). However, element quality either degrades considerably near the surface, or many elements are required. We show a comparison against the representative method Quartet [Bridson and Doran 2014] in Section 6.

Advancing front. Some methods (e.g.,[Alauzet and Marcum 2014; Cuillière et al. 2013; Frey et al. 1996; Haimes 2015]) start from a given front (the boundary surface) and propagate meshes inwards. While seemingly achieving good element quality near the boundary, these methods suffer from problematic cases when fronts meet in the interior of the volume, making the generation of high-quality elements in these places challenging. We are not aware of any robust and publicly available representative implementation for this category.

In the wild. A considerable portion of methods do not inherently assume the boundary is valid or watertight. Thus, a resulting tet mesh would not necessarily be conforming. One such approach is envelope meshing (e.g., [Hu et al. 2018; Mandad et al. 2015; Shen et al. 2004]), where the surface is approximated to some volumetric tolerance, or an "envelope" around the original boundary components. While these methods solve an inherently different problem, we show a comparison against the representative method TetWild in Section 6, where we show that the boundary is not exactly represented.

2.3 Robust geometric predicates

Numerical robustness is a common issue in many meshing algorithm implementations. If standard floating point arithmetic is used with no specific care, an implementation may easily crash or fail to converge [Li et al. 2005]. Implementations can be made robust by using exact arithmetic kernels [Fabri and Pion 2009]. However, this solution is often too slow for practical applications. Alternatively, robustness can be achieved by simply guaranteeing that the program flow is correct while accepting a rounded output [Li et al. 2005]. The program flow is determined by its branches that, in turn, are governed by the value of geometric predicates analyzing the relative position of points: when their coordinates are read from an input file, floating-point filtering techniques and adaptive precision can combine speed and robustness [Shewchuk 1997]. This is not sufficient when some of the points being analyzed are constructed by the algorithm. Indeed, in this case, coordinates can be rounded and predicates are no longer guaranteed to give the correct answer. However, if these intermediate points can be expressed as simple combinations of input points, we can still exploit the concept of indirect predicates [Attene 2020] to leverage the efficiency of the floating point hardware. This solution was successfully used to create polyhedral meshes in [Diazzi and Attene 2021], though the arbitrarily bad shape of cells severely limits their potential application.

3 BACKGROUND

Unless explicitly stated otherwise, all the concepts described in this section are defined in three-dimensional space.

3.1 Problem statement

Our input is a piecewise-linear complex (PLC) [Miller et al. 1996]. A PLC is a collection of vertices, segments, and polygonal facets with the typical characteristics of a complex: the boundary of any element is made of lower-dimensional elements of the PLC, and the set is closed under intersection; that is, the intersection of any two elements of a PLC is either empty or it is the union of other elements of the PLC. Our objective is to compute a CDT that preserves the original PLC. Specifically, if P is our PLC, we want to compute a tetrahedrization T of the convex hull of P such that T has all, and only, the vertices of P, and each facet of P is the union of some triangular facets of T. Furthermore, we require that T is constrained Delaunay with regards to P; that means that the interior of the circumsphere of each tetrahedron does not contain any *visible* vertex of P (see Fig. 2). In turn, a vertex v is visible from within a tetrahedron t if any point in the interior of t can be connected to v using a straight segment that does not intersect P.

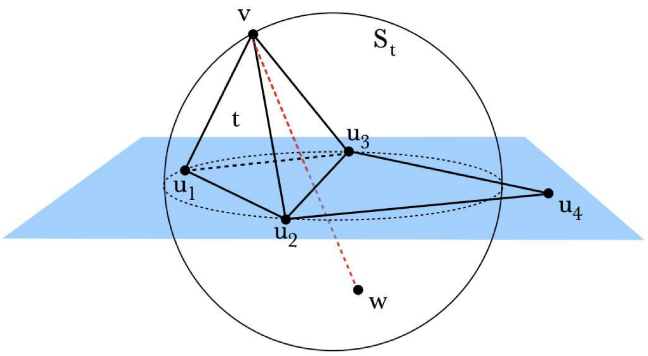


Fig. 2. A constrained Delaunay Tetrahedron t. The blue region is the supporting plane to PLC faces $\langle u_1, u_2, u_3 \rangle$ and $\langle u_2, u_3, u_4 \rangle$. Vertices v and w belong to opposite half-spaces defined by the plane, and thus w is not visible from the interior of tetrahedron $t = \langle u_1, u_2, u_3, v \rangle$. In contrast, vertex u_4 is visible from t. S_t is the circumsphere of tetrahedron t that encloses w but not u_4. Therefore, t is constrained Delaunay.

3.2 Characteristics of a PLC

As mentioned in the introduction, not all PLCs admit a CDT. Schönhardt's polyhedron [Schönhardt 1928] and Chazelle's polyhedron [Chazelle 1984] are typical examples of PLCs that do not have a CDT. A sufficient condition for a PLC to admit a CDT is that all its segments are *strongly* Delaunay (cf. CDT theorem [Shewchuk 1998]): a segment is strongly Delaunay if it admits a circumsphere that neither contains nor touches other PLC vertices. Interestingly enough, this characteristic deals with segments only; therefore, splitting PLC segments at appropriate points is sufficient to turn a generic PLC into one that admits a CDT. The split points are denoted as Steiner points. We thus allow such splitting for PLCs that do not admit CDTs. Note that splitting is a purely topological operation that does not modify the geometric realization of the input. Among the possible approaches to compute Steiner points, we chose the method used by Si [Si and Gärtner 2005] in his `tetgen` software, as it produces the smallest number of Steiner points in practice.

3.3 Segment recovery

Si observes that if all the segments of a PLC belong to the Delaunay tetrahedrization of its vertices, then a CDT exists and can be computed using local operations (see [Si and Gärtner 2005], thm. 2). When a PLC segment is not in the Delaunay tetrahedrization, it is called a *missing* segment and cannot be strongly Delaunay. In this case, the idea is to split it so that the resulting (shorter) sub-segments have more chances to satisfy the condition. The exact location of the split is crucial to guarantee termination.

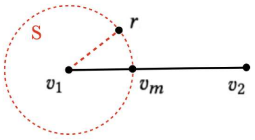
To determine split points, Si uses the concept of *encroaching point*. Let $e = \langle v_1, v_2 \rangle$ be a missing segment, D be the smallest (diametral) sphere by v_1 and v_2, and V_D be the set of vertices enclosed in D, excluding v_1 and v_2. Note that V_D cannot be empty because e would be strongly Delaunay and not missing. The vertices in V_D are called *encroaching points* for e. The algorithm picks the encroaching point r for which the circle defined by v_1, v_2 and r has the maximum radius, and declares it to be the *reference* point for the segment.

Another important concept in Si's approach is the classification of an *acute* vertex: a vertex v is acute if two PLC segments incident at v form an acute angle. Based on this, PLC segments are classified into three categories:

(1) Segments having no acute vertices;
(2) Segments having only one acute vertex;
(3) Segments having two acute vertices.

Missing segments of the 3^{rd} category are split at their midpoint, so that the resulting sub-segments belong to the 2^{nd} category. Any other missing (sub)segment $\langle v_1, v_2 \rangle$ with reference point r is further subdivided by inserting a point v_m, until no segment is missing.

If e is of the 1^{st} category, Si's method considers the two spheres S_1 and S_2 centered at v_1 and v_2 respectively, and both touching r. If the radii of S_1 and S_2 are both bigger than half of the length of e, v_m is set to the midpoint of e; otherwise, v_m is the intersection of e and the sphere that has the smallest radius (see inset).



If e is in the 2^{nd} category and w is its acute vertex, all its sub-segments remain in the same 2^{nd} category and *remember* the original acute vertex w.

w is used as center for a sphere S which touches the reference point r. Let v_w be the endpoint closest to w, v_o be the other endpoint, and p be the intersection point of S and e. If p is closer to r than to v_o, v_m is set to p. Otherwise, if the distance d between r and p is less than half the distance between v_w and p, shift p towards v_w by a distance d. Otherwise, move p to the midpoint of the segment $< v_w, p >$. v_m is set to p.

Missing segments are split one after the other in an iterative process which is guaranteed to converge to a PLC that admits a CDT.

Note that v_m may be determined by the intersection of a segment with a sphere, and hence its coordinates can be irrational numbers even if all the input vertices are in floating point precision.

3.4 Face recovery

Once all the PLC segments are part of the Delaunay tetrahedrization, Si's method verifies whether all PLC faces are represented by the union of Delaunay triangles. A face that is not represented is denoted as a *missing* face that must be *recovered*. That means that the region around the face must be retetrahedrized in order to enforce the requirement. A PLC-face f is determined to be missing if it is pierced by at least one edge of the tetrahedrization. Let T_f be the union of all the tetrahedra that are incident to these face-piercing edges. Each face-connected subset of T_f forms a *cavity* (see Fig. 3) that requires retetrahedrization (two tetrahedra are face-connected if they share a triangle).

The cavity is split along the plane of f into two half-cavities, C_1 and C_2, where the vertices that belong to the plane of f are assigned to both cavities. For each of the C_i, the vertices are used to compute a local Delaunay tetrahedrization D_i, which is used to fill the respective half cavity with new tetrahedra. Note that D_i is convex but C_i may be concave, meaning that not all the tets in D_i are necessarily used. Then, the final retetrahedrization of the cavity

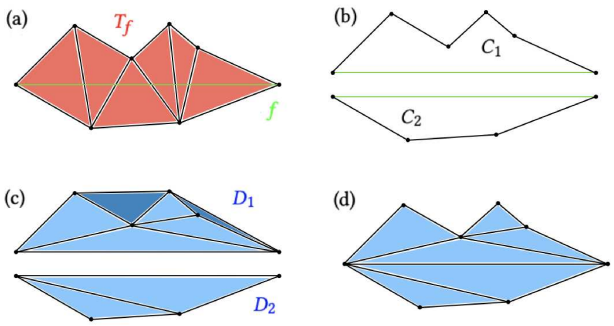


Fig. 3. Face recovery in 2D. The missing PLC 2D-face f (green) is intersected by mesh triangles T_f (red)(a), which are removed to create two half-cavities (b). The convex hull of each half-cavity is then Delaunay-triangulated (blue) (c). Triangles outside the cavity (dark blue) are removed whereas the remaining triangles are used to fill the cavity while preserving f (d).

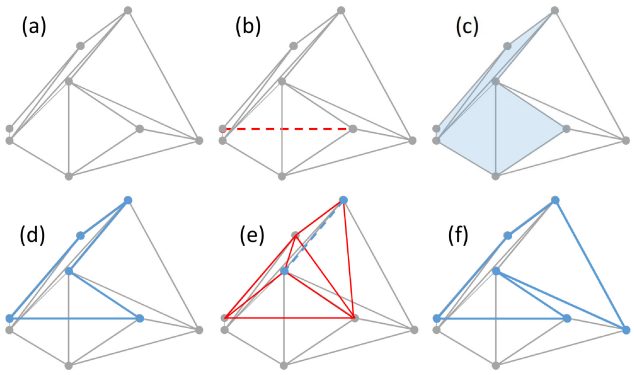


Fig. 4. A 2D example showing a failure of the cavity expansion described in [Si and Gärtner 2005], where 3D face recovery corresponds to 2D segment recovery. We start from the configuration in (a), which is plausible since having recovered any segment before, the Delaunay property of the vertices may not hold anymore. Consider the missing segment being recovered (b), its corresponding cavity (c), and the upper half cavity (d). The Delaunay triangulation of the vertices of the half-cavity is in red (e), where the dashed blue segment is missing, and thus the algorithm expands across it (f). However, this results in a half-cavity that intersects with the lower half-space.

is the union of the used tets from D_1 and D_2. All missing faces are recovered one after the other in an iterative process.

In order for this method to work, the triangles bounding C_i should result in triangles in D_i. If a triangle $\tau \in C_i$ is not in D_i, the half-cavity is expanded by adding the opposite tetrahedron to τ. In that case, C_i is updated and D_i is recomputed. This process is iterated as long as the boundary of C_i is not entirely represented by triangles in D_i. This might result in a failure of the algorithm, as we point out in the following.

3.5 Possible failures in tetgen

While efficient, Si's method has two potential weaknessess. First, as stated in Sec. 3.3, Steiner points may have irrational coordinates, meaning that a correct implementation requires predicates and data types that can robustly deal with irrational numbers (see Sec. 4). However, the only existing implementation (`tetgen`) uses floating-point arithmetics. Even if filtered exact predicates [Shewchuk 1997] are employed in `tetgen`, their guarantees are lost as soon as a Steiner point is rounded to its closest floating point representable position.

Second, the cavity expansion process described in Sec. 3.4 may fail regardless of numerics. The algorithm implicitly assumes that the local tetrahedrizations of the two half-cavities have disjoint interiors, and that their boundaries match at the face being recovered. However, there are cases where the expansion process adds a tetrahedron from across the plane of the recovered face (see Fig. 4), leading to an intersection of the two tetrahedrizations. Such cases are rare (this happens in just 2 of the 4408 models we tested). However, this is an algorithmic failure that cannot be overlooked.

4 ROBUST CDT

As mentioned in the introduction, representing Steiner points using floating point coordinates introduces a major weakness in any possible implementation. Employing thresholds does not solve the inherent problem, and might still lead to failure (Sec. 6).

In the following, we describe the individual components of our robust CDT algorithm. They comprise (1) a standard Delaunay tetrahedrization of the input vertices based on incremental insertion; (2) a robust segment recovery procedure; (3) a robust face recovery phase; (4) an internal/external characterization of the tetrahedra. While for (1) and (4) we adopt standard methods, our novel contributions reside in steps (2) and (3). Our segment recovery proceeds as in [Si and Gärtner 2005], but we resolve its numerical fragility thanks to our new implicit point type described in the remainder. Similarly, our face recovery also proceeds as in [Si and Gärtner 2005] while employing our novel numerical kernel. Nonetheless, we also detect and resolve the cavity expansion theoretical issue and, in these cases, we proceed with an alternative algorithm (Sec. 4.4), so as to guarantee a successful termination in all circumstances.

4.1 tetgen with an exact number type

The easiest solution to the fragility problem is to reimplement Si's approach with *exact* arithmetic kernels. Nevertheless, one then needs the number type to be closed under the square-root operation. As an example, Steiner points may be at the intersection of a segment and a sphere (Sec. 3.3), where then their coordinates could be irrational.

An appropriate library for that task is CORE [Karamcheti et al. 1999]. CORE provides a number type that is closed under all the standard arithmetic operations, including the square root. CORE is based on the representation of intervals, and is able to evaluate the sign of expressions with guaranteed correctness. This however comes at a rather high computational cost. Even while exploiting the lazy evaluation paradigm provided by CGAL [Fabri and Pion 2009] to speed up calculations based on CORE, the resulting algorithm is far too slow for practical use (hours are necessary even for moderately small input files, see Sec. 6). We note that this implementation is

numerically exact, but still does not solve the theoretical flaw defined in Sec. 3.5.

4.2 Rational Steiner points

To mitigate the performance problem with CORE, we can employ a simple fix that allows us to work with (arbitrarily precise) *rational* numbers. We observe that Steiner points must be placed exactly on the segment they split. If their exact position is *snapped* to an approximated location (e.g., due to the need to represent coordinates using floating point numbers) we are actually deforming the input PLC. In turn, this renders all the theoretical proofs invalid and leads to non-robust implementations that can easily crash. Nevertheless, as long as a Steiner point remains exactly on its originating segment, its position along the segment can change up to a certain amount without causing a change in the mesh connectivity. We then opt to represent the approximate Steiner point with rational coordinates that are arbitrarily close to the exact location defined by Si's algorithm (Sec. 3.3). Specifically, we parameterize this position by a linear combination of the segment endpoints with a rational $t \in (0, 1)$, so that the position p is $tv_1 + (1-t)v_2$. This allows reimplementing the algorithm using faster number types such as GNU GMP rationals [Granlund 1996]. The value of t can be approximated by rounding its exact (possibly irrational) value to its nearest floating point number using the standard `sqrt` library function. If this is not precise enough, the precision can be iteratively doubled as long as necessary. However, our experiments reveal that in practice there is no actual need for such an increase in precision: using the initial approximation always leads to exactly the same results we could obtain using the exact implementation based on CORE. Even in this case, CGAL's lazy evaluation mechanism can be exploited to speed up the operations of a factor of 1.5×-4× (Sec. 6.1).

4.3 Implicit Steiner points

The rational version of our algorithm opens the door to a further, and major, optimization: we can replace GMP with custom indirect geometric predicates [Attene 2020].

In a nutshell, indirect predicates use standard floating point calculations to derive **exact** information regarding the mutual position of so-called *implicit* points. In turn, an implicit point is an unevaluated expression representing a position in space as a function of other known positions. In 3D, examples of implicit points are LPI (Line-Plane Intersection) and TPI (Three-Planes Intersection). When its floating point coordinates are known, a point may be called an *explicit* point. With this terminology, an LPI is a function of five explicit points, two representing the line and three representing the plane being intersected. LPIs and TPIs, along with standard `orient2D` and `orient3D` geometric predicates operating on them, were successfully used for diverse applications [Cherchi et al. 2020; Diazzi and Attene 2021; Du et al. 2022]. Herewith, we introduce a new type of implicit point representing a linear combination of two known points, and show how to derive indirect versions of the `orient3d` and `inSphere` predicates necessary to construct a CDT.

Let v_1 and v_2 be two explicit points, and let t be a floating point number in the range $(0, 1)$. The expression $tv_1 + (1-t)v_2$ is called an LNC (LiNear Combination) implicit point. LNCs can be effectively used to represent Steiner points during all the phases of our CDT algorithm. Indeed, any non-explicit point treated in our approach is a point that subdivides an input segment, meaning that it can be expressed as a linear combination of its two (explicit) endpoints. Note that, thanks to our *local-global* approach to exploit symbolic perturbation (Sec. 4.4), we do not need any additional points to remove the so-called *local degeneracies* discussed in Sec. 6 of [Si and Gärtner 2005].

4.3.1 Indirect `orient3d` *and* `inSphere` *predicates.* The basic idea behind indirect predicates is to combine the expression of the predicate itself with the expression of the argument points. If the combined expression is either a polynomial or a ratio of polynomials, its sign can be determined without errors using arithmetic filtering and, if necessary, floating point expansions [Attene 2020].

In our context, argument points can be: (1) explicit points whose coordinates come directly by reading an input file or (2) implicit LNCs representing Steiner points. In either case, the point expression is a simple polynomial, and the result of `orient3d`(p_1, p_2, p_3, p_4), where all the four argument points are explicit, corresponds to the sign of the following determinant, which is another polynomial:

$$\begin{vmatrix} p_{1x} - p_{4x} & p_{1y} - p_{4y} & p_{1z} - p_{4z} \\ p_{2x} - p_{4x} & p_{2y} - p_{4y} & p_{2z} - p_{4z} \\ p_{3x} - p_{4x} & p_{3y} - p_{4y} & p_{3z} - p_{4z} \end{vmatrix} \tag{1}$$

If the first argument point is an LNC $p_1 = tv_1 + (1-t)v_2$, while the other three points are explicit, the combined expression can be obtained by simply replacing p_{1x} with $tv_{1x} + (1-t)v_{2x}$ (and similarly for p_{1y} and p_{1z}). Therefore, the corresponding indirect `orient3d` predicate evaluates as:

$$\begin{vmatrix} tv_{1x} + (1-t)v_{2x} - p_{4x} & tv_{1y} + (1-t)v_{2y} - p_{4y} & tv_{1z} + (1-t)v_{2z} - p_{4z} \\ p_{2x} - p_{4x} & p_{2y} - p_{4y} & p_{2z} - p_{4z} \\ p_{3x} - p_{4x} & p_{3y} - p_{4y} & p_{3z} - p_{4z} \end{vmatrix} \tag{2}$$

Similarly, the result of `inSphere`$(p_1, p_2, p_3, p_4, p_5)$, where all the five argument points are explicit, corresponds to the sign of the following determinant:

$$\begin{vmatrix} p_{1x} - p_{5x} & p_{1y} - p_{5y} & p_{1z} - p_{5z} & \|p_1 - p_5\|^2 \\ p_{2x} - p_{5x} & p_{2y} - p_{5y} & p_{2z} - p_{5z} & \|p_2 - p_5\|^2 \\ p_{3x} - p_{5x} & p_{3y} - p_{5y} & p_{3z} - p_{5z} & \|p_3 - p_5\|^2 \\ p_{4x} - p_{5x} & p_{4y} - p_{5y} & p_{4z} - p_{5z} & \|p_4 - p_5\|^2 \end{vmatrix} \tag{3}$$

The matrix above can be modified by simple substitution as done for `orient3d` to derive indirect predicates.

In principle, any combination of explicit and implicit points determines one different indirect predicate, meaning that we need to account for 16 different versions of the `orient3d` predicate and 32 versions of the `inSphere`. Fortunately, since swapping rows in the matrix affects the determinant sign in a predictable manner, one can limit the possible variety to 5 and 6 versions respectively. One version accounts for the total number of argument points that are in implicit form and assumes these points are the first in the argument list. When calling the predicate, the code swaps contiguous rows to move all implicit points to the beginning of the list and keeps track of the parity of these swaps to determine the sign. For example, if the only implicit point is p_2, the result of `inSphere`$(p_1, p_2, p_3, p_4, p_5)$ is computed as `-inSphere`$(p_2, p_1, p_3, p_4, p_5)$, and the minus sign is there

because the number of necessary swaps is odd. If p_2 and p_3 are implicit, `inSphere`$(p_1, p_2, p_3, p_4, p_5)$ is equal to `inSphere`$(p_2, p_3, p_1, p_4, p_5)$, because the number of swaps is even. Filter values for fast calculation using floating point arithmetic are given in Appendix A.

4.4 Modified gift-wrapping algorithm

To cope with the cavity expansion failures, we describe an alternative algorithm based on [Shewchuk 2000b]. Shewchuk's method is guaranteed to produce the CDT out of a PLC that admits one. It is essentially a modification of a naïve gift-wrapping approach: first, each face in the input PLC is triangulated using a local 2D CDT. Then, each resulting triangle is connected to one *apex* vertex to form a tetrahedron. If the input vertices are in general position, the CDT is unique, meaning that only one vertex in the set is a valid apex for any given triangle. Hence, when such a valid apex is selected among the input vertices, the tetrahedron is guaranteed to be part of the eventual CDT.

In our method, we combine a local version of this gift-wrapping algorithm with a coherent symbolic perturbation technique [Edelsbrunner and Mücke 1990] that guarantees conformity everywhere, even if the points are not in general position. Specifically, we construct one cavity at a time as described in Sec. 3.4 and split it into two half-cavities. For each half-cavity, we build one tetrahedron at a time as in [Shewchuk 2000b] while considering three key aspects:

- The triangles common to the opposite half-cavity are unknown when tetrahedrizing the first of the two halves;
- The triangulation induced by the tetrahedrization of the two half-cavities must match on the common face, even if the vertices are not in general position;
- The implementation may not tolerate numerical errors, and therefore all predicates and checks must be exact.

For any missing PLC-face f we first delete all tetrahedra T_f whose interior intersects f and keep track of all their vertices V_f. When done, we keep track of all the triangles ∂C that bound the resulting cavity C, each oriented so that the normal points toward the exterior. Then, we create the half-cavities C_1 and C_2 by splitting C through f and, at the same time, we split ∂C in two subsets ∂C_1 and ∂C_2. Furthermore, we split the set V_f in two subsets V_1 and V_2 whose vertices are over (or on) and below (or on) f respectively.

We then fill one half-cavity at a time. Let C_1 be the first. The set ∂C_1 contains *known* triangles that bound C_1, but does not completely enclose the half-cavity. Nonetheless, we have sufficient information to proceed with the creation of tetrahedra.

During our iterative process, a set of triangles ∂C_{cur} defines the *current* boundary of the half-cavity being filled. At the beginning, $\partial C_{cur} = \partial C_1$.

We iteratively pick a triangle σ from ∂C_{cur} and search for a suitable apex vertex w in V_1 such that the resulting tetrahedron t is *valid*. When we find it, we create the tetrahedron, update ∂C_{cur} and process the next σ. After the first iteration, σ might be on the plane of f: in this case we simply skip it and move to the next one. The process terminates when all the triangles in ∂C_{cur} are processed, see Fig. 5.

During this process the tetrahedron t, made by joining the triangle σ and the apex w, is valid if it satisfies all the following conditions:

i) w is in the opposite half-space with respect to σ outgoing normal (i.e. t has positive volume);
ii) if t intersects a triangle of ∂C_{cur} then the intersection is a common subsimplex;
iii) no vertex in V_1 is in the circumsphere of t, except those whose visibility is occluded by ∂C_1.

We observe that, as soon as a vertex in V_1 is no longer *usable* as an apex (e.g., because, as the wrapping proceeds, it becomes completely surrounded by tetrahedra) it might be removed from the search list. However, condition iii) must still check all the vertices in the original half-cavity.

When C_1 is completed, we repeat the same process on C_2, though in this case we reuse the triangles produced on f while filling C_1.

Note that, in the case of cospherical points in the cavity, the CDT may be not unique. This means that, when creating a tetrahedron for C_2, we are no longer sure that it will eventually induce a mesh that conforms with the triangles on f inherited from C_1. Avoiding the inheritance (and hence proceeding for C_2 with an open boundary as we do for C_1) would not solve the problem, because the common triangulation induced on f might not match.

We solve this problem by exploiting symbolic perturbation [Edelsbrunner and Mücke 1990] as follows: let v_1, ..., v_4 be the four vertices of a tetrahedron, and let q be a query vertex. Our exact `inSphere` predicate (see Sect. 4.3.1) states whether q is *inside*, *outside*, or *exactly on* the sphere defined by v_1, ..., v_4. When the result is *exactly on*, we take a decision between *inside* and *outside* based on the order in which the five vertices are stored in memory (Alg. 1). To guarantee that all the cavities are tetrahedrized conformally with the other parts of the mesh, we create the subvectors representing V_1 and V_2 so that any two vertices in V_1 (resp. V_2) are stored in the same order as they are stored in the *global* mesh vector.

ALGORITHM 1: perturbedInCircumphere(i_1, i_2, i_3, i_4, i_5)

Input: i_1, ..., i_5: indexes of the five points in a global vector V, where i_1, ..., i_4 are the four vertices of a valid tetrahedron, whereas i_5 is a query point
Output: -1 if i_5 is inside or on the circumsphere of i_1, ..., i_4, 1 if it is outside or on the circumsphere.

```
r := inSphere(V[i1], V[i2], V[i3], V[i4], V[i5])
if r ≠ 0 then return r ;
sort i1, ..., i5 in ascending order by n iterative swaps
r := orient3D(V[i2], V[i3], V[i4], V[i5])
if n is odd then r := -r ;
if r ≠ 0 then return r ;
r := orient3D(V[i1], V[i3], V[i4], V[i5])
if n is even then r := -r ;
return r
```

We use indirect predicates for all the intersection and visibility checks required by conditions $(i)-(iii)$, so as to guarantee exactness without the need for slow exact arithmetic. Checking conditions (ii) and (iii) is particularly complex, as one must take into account all the possible configurations in which a triangle and a tetrahedron may be arranged in three-dimensional space. We describe these conditions in terms of geometric predicates in App. B.

The fact that this process leads to a CDT is given for granted in [Shewchuk 2002] but may be not obvious at a first sight. We

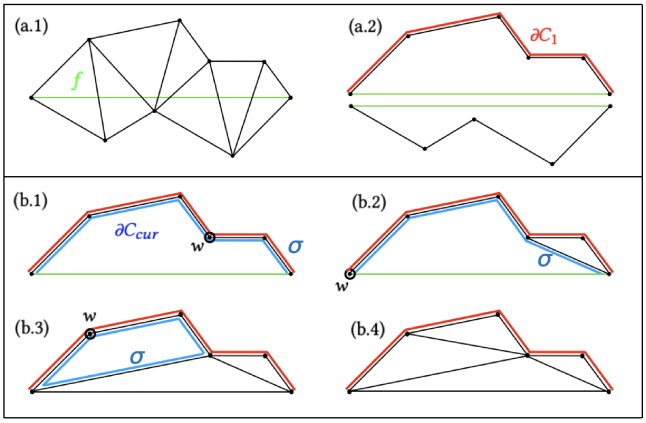


Fig. 5. A 2D example of gift wrapping. The missing PLC segment f (green) is intersected by mesh triangles (a.1), which are removed to create two half-cavities (a.2). The upper cavity is then triangulated (b.1-b.4), by exploiting gift-wrapping algorithm discussed in Sec. 4.4. The cavity boundary ∂C_1 (red) does not change during triangulation, while the current cavity boundary ∂C_{curr} (blue) does. At each step an edge σ of ∂C_{curr} is connected to an apex w (circled) to create a *valid* triangle.

observe that a tetrahedrization is constrained Delaunay if and only if all the internal triangles are *locally* constrained Delaunay. In turn, an internal triangle is locally constrained Delaunay if (1) it is a constraint (i.e. part of the input PLC) or (2) each of its two incident tetrahedra has a circumsphere that does not contain the apex vertex of the opposite incident tetrahedron. This precondition holds before digging the cavity, and still holds for all the triangles created within the cavity because the new tetrahedra are *valid*. Each triangle on ∂C is also locally constrained Delaunay because it must satisfy one of the following two conditions: (1) it is a constraint or (2) its incident tetrahedron outside the cavity does not contain any visible vertex (because the mesh was a CDT before digging the cavity) thus including the apex of its (newly created) opposite tetrahedron. Hence, the resulting mesh after gift-wrapping is a CDT wrt all the facets recovered so far.

On average, our modified gift-wrapping algorithm is fast enough and allows us to successfully tetrahedrize all the models in our reference dataset (Sec. 6). Nevertheless, since the approach in [Si and Gärtner 2005] is faster in practice, our implementation uses the latter unless the cavity expansion fails (Sec. 3.5), where then it switches to the modified gift-wrapping approach, see Fig. 6. Note that cavity expansion fails in 2 out of 4408 models in our dataset.

We observe that, as an alternative to our modified gift-wrapping, cavities may be retetrahedrized using iterative flips as in [Shewchuk 2003]. However, this would require implementing further indirect predicates operating with four-dimensional points.

4.5 Interior/exterior characterization

If the input unambiguously separates the space into internal and external parts, we can similarly characterize each tet in our final tetrahedrization. For this step, which is optional in our pipeline, we rely on a flood-filling approach previously adopted in other meshing

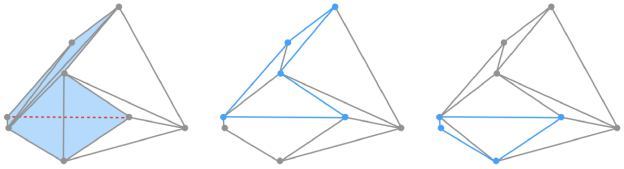


Fig. 6. If the half-cavity expansion fails as for the 2D example on the left (see Fig. 4), we switch and fill the original (unexpanded) half-cavity with valid *triangles* using gift-wrapping (middle), we then use it again for the other half-cavity (right).

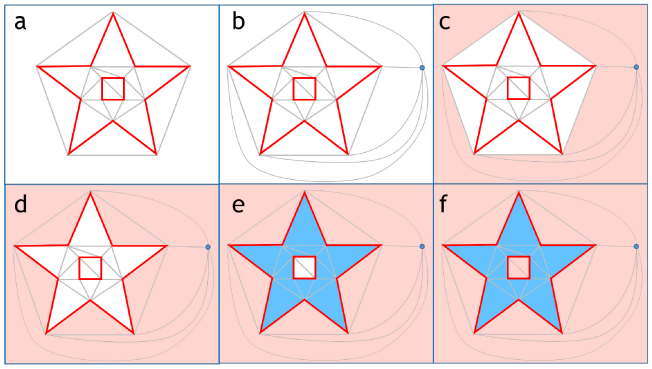


Fig. 7. A 2D example of propagating internal (blue) and external (light red) labels. A PLC (red segments) is depicted along with its CDT (gray) (a). We add a ghost vertex (blue) (b), and connect it to all the boundary edges to form ghost *triangles*. Ghost triangles are tagged as *external* (c). The *external* label propagates without crossing PLC edges (d), switches to *internal* and propagates again (e), and finally switches back to *external* and propagates on all the remaining triangles (f).

algorithms [Hu et al. 2020; Xiao et al. 2016]. We use the common concept of *ghost* vertex to ensure that each tetrahedron always has four neighbors. The ghost is essentially a virtual vertex "at infinity" that is connected to all the triangles of the convex hull, so that each triangle produces a *ghost tetrahedron*. Adoption of ghosts significantly simplifies data management [Marot et al. 2019].

Our interior/exterior characterization proceeds by first assigning an *external* label to all the ghost tetrahedra (Fig. 7(c)), and then by propagating this label across *unconstrained* triangular faces (i.e., faces which are not part of the input PLC, Fig. 7(d)). When propagation stops (i.e., because all the faces reached are constrained), we switch the label from *external* to *internal* across PLC faces, and keep propagating across unconstrained faces (Fig. 7(e)). If propagation stops on other constrained faces we switch the tag back to *external* and so on, until all the tetrahedra are reached (Fig. 7(f)).

This simple process assumes that the input defines a well-defined volume or, equivalently, that each segment in the PLC has an even number of incident faces. If this is not the case one may decide to find a plausible characterization using winding numbers [Jacobson et al. 2013] or graph labeling [Diazzi and Attene 2021], but this is out of the scope of this work.

4.6 Implicit Steiner CDT algorithm

Having defined the individual steps, we next describe the entire algorithm which includes the possible refinement of the input PLC to make it admit an output CDT. With reference to Alg. 2, we first compute the Delaunay tetrahedrization D of the input vertices using a classical Bowyer-Watson incremental insertion [Bowyer 1981; Watson 1981] . Then, we proceed with the segment recovery (Sec. 3.3). This step is enclosed in a **while** loop because a non-missing segment might become missing when D is modified due to a Steiner point insertion. A proof of convergence is given in [Si and Gärtner 2005].

The subsequent face recovery works similarly. Here the outer loop is necessary because a non-missing face might become missing due to the cavity expansion. Even in this case, proof of convergence is given in [Si and Gärtner 2005]. Note that gift-wrapping is used only if the expansion fails, which means it does not have an impact on guaranteed convergence.

ALGORITHM 2: SteinerCDT(P)

Input:
$P =< V, E, F >$: a valid PLC with vertices V, edges E and faces F.
Output:
A Steiner CDT of P

D := Delaunay(V) `// [Bowyer 1981]`
while *at least a segment is missing in* D **do**
  foreach $e \in E$ **do**
    if e *is missing in* D **then**
      calculate a Steiner point s `// Sec. 3.3 and 4.2`
      split e at s
      insert s in D `// [Bowyer 1981]`
    end
  end
end
while *at least a face is missing in* D **do**
  foreach $f \in F$ **do**
    create half-cavities C_1 and C_2 `// Sec. 3.4 and 4.4`
    if C_1 *and* C_2 *can both be expanded* **then**
      recover f in D by local Delaunay `// Sec. 3.4`
    else
      recover f in D by gift-wrapping `// Sec. 4.4`
    end
  end
end
return D

5 POST-PROCESSING AND APPLICATIONS

5.1 Floating point representation

Because the coordinates of LNC points can be losslessly converted to rational numbers, our implementation can save the output CDT to a file with no approximations. However, a typical requirement in downstream applications is that the vertex coordinates are in floating-point precision, regardless of the number type used by the algorithm. Unfortunately, rounding our implicit points to their closest floating-point position may make a nearly degenerate (though valid) element into a flat or inverted tetrahedron. In contrast to 2D Delaunay triangulations, the empty-sphere property does not always result in optimal element quality in 3D [Alexa 2019]. In particular, 3D Delaunay meshes often exhibit bad-shaped elements called *slivers*. A sliver is a valid tetrahedron with no short edges, but rather four nearly-coplanar vertices. Some methods remove these elements from a fully-Delaunay tetrahedrization (e.g.,[Cheng et al. 2000]); we however must constrain the output to exactly preserve the input PLC. Though PLC-preserving sliver-removal algorithms have been designed (e.g., [Cheng and Dey 2002]), they are extremely complicated and refine the mesh everywhere, adding extra unnecessary elements.

Our approach is to remove bad-shaped tetrahedra by iterating local connectivity modifications so as to monotonically increase the element quality. While this is not a full mesh optimization such as in [Hu et al. 2018], we found it is sufficient to prevent the introduction of degenerate elements when rounding.

We modify the connectivity by iterative face and edge swaps. A face swap (also called a 2–3 swap) replaces two tetrahedra sharing a face f with three tetrahedra sharing an edge that connects the two vertices opposite to f in the initial configuration (see Fig. 8). An (3-2) edge swap is essentially the inverse operation to face swap, when applicable. Typical implementations use the said 3-2 swaps, 4-4 swaps, or even 5-6 swaps [Hu et al. 2018]. We use a single, but general, edge-swap operation that works as follows: we first split an edge $e = (v_1, v_2)$ by inserting a virtual point, then immediately collapse the new point to one of its neighbors different from v_1 and v_2. The temporary point is not assigned any position as it is immediately destroyed, hence the term *virtual*. An edge shared by n tetrahedra can be swapped in $n - 2$ different ways, depending on the neighboring vertex used to collapse the virtual point. When $n = 3, 4, 5$, our generic edge-swap corresponds to the standard 3-2, 4-4, and 5-6 swap respectively.

In our mesh improvement algorithm, each swap is operated only if both these conditions hold: (1) no tetrahedron is inverted or flattened due to the change; (2) the maximal AMIPS energy [Hu et al. 2018] of tetrahedra strictly decreases due to the change. When swapping an edge we randomly select a neighboring vertex to which we collapse the virtual point, where the conditions would hold; if no such vertex is found the swap is rejected. To ensure that the resulting mesh is still conformal with the input PLC, no swap is operated if the interior of the affected region contains constrained facets. After this process, the rounding may still introduce invalid tetrahedra, but in practice this possibility is dramatically reduced (Sec. 6). Clearly, after this process the mesh is no longer guaranteed to be constrained Delaunay. Note that, although an FP-rounded Steiner point is no longer *exactly* on its originating input segment, that segment is unique and known, therefore the Steiner point can inherit boundary conditions from the input with no ambiguity.

5.2 Delaunay refinement

The CDTs produced by our method can be effectively used within a plethora of mesh refinement algorithms, each striving to maximize/minimize some particular metric depending on the target application. Most importantly, the fact that our meshes are *constrained Delaunay* guarantees that Delaunay refinement algorithms converge to reliably good meshes [Shewchuk 2000a, 2002]. To demonstrate

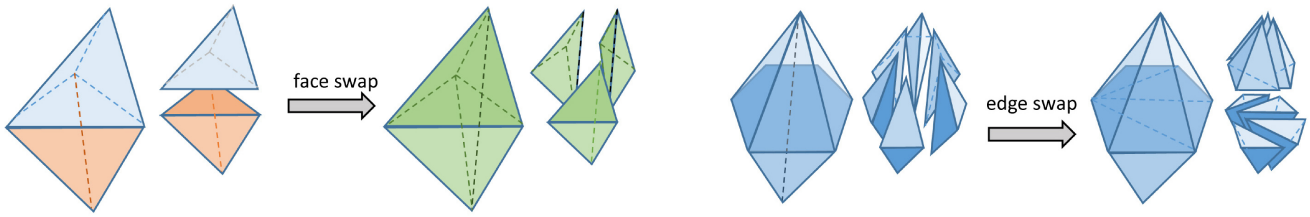


Fig. 8. The two basic topological operators used to remove bad-shaped elements.

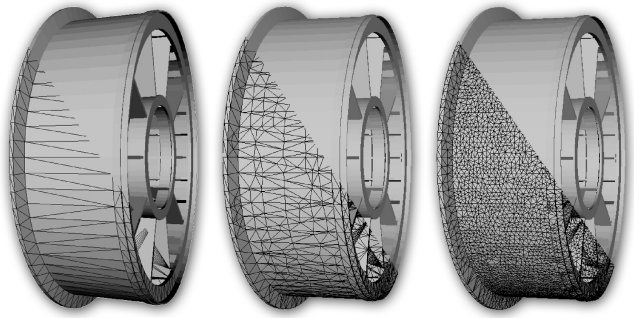


Fig. 9. An input PLC that makes tetgen crash (left), admits a valid CDT by our method (middle), which is optimized by running tetgen in *refine-only* mode on our CDT (right).

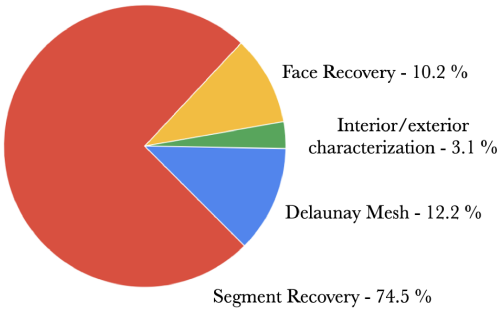


Fig. 10. A division of our running time to the various steps of the pipeline. These data refer to the time spent processing the 4408 Thingi10k models defining a PLC.

this property, we use the Delaunay refinement algorithm implemented in `tetgen` [Shewchuk and Si 2014], where we set `tetgen` to bypass its initial CDT generation and instead provide the CDT from our algorithm as input. Fig. 9 shows an example.

If a certain amount of displacement is tolerable, other refinement algorithms may provide better results. We further tested our meshes as input to the algorithm in [Dobrzynski and Frey 2008], for which an open-source implementation is available [MMG3D 2004]. We demonstrate this in Fig. 1-(d, e).

6 RESULTS AND DISCUSSION

We implemented our algorithm as a standalone tool in C++. For this, we modified the Indirect Predicates library [Attene 2019] to support LNC points. We can plug in several number types to our code to represent coordinates (Secs. 4.1—4.3). This enables a fair comparison of the various versions of the algorithm, in terms of performance and robustness, while being sure that exactly the same algorithm is run. We use the exact number types provided in CGAL [Fabri and Pion 2009] to implement the exact and rational versions. Our source code is freely available at *https://github.com/MarcoAttene/CDT*. Our algorithm was run on a Linux-based machine equipped with an AMD EPYC 7452 CPU and 1Tb RAM. All the experiments were run on a single core.

6.1 Results

We tested our implementation on the meshes of the Thingi10k dataset [Zhou and Jacobson 2016] that satisfy the PLC conditions (i.e. that do not self-intersect), for a total of 4408 models. Our experiments demonstrate that our method is fast enough for most practical applications: 76% of the models are processed in less than 1 second, and the average execution time is 4.3 seconds. Furthermore, only 1.2% of the models require more than 60 seconds, in which the worst case takes 33 minutes. If one considers this small tail of models to represent outliers, the expected (average) time for the remaining 98.8% of the models is just 1.8 seconds. We do not observe any clear relation between the number of input triangles and elapsed time. We argue that geometric characteristics of the input (i.e. the local feature size) are an overriding factor in this sense.

Our algorithm may be divided into the following main phases: Delaunay tetrahedrization of the input vertices, segment recovery, face recovery, and interior-exterior characterization of the tets. Fig. 10 gives an idea of how each of these phases impacts the overall execution time, and reveals that segment recovery accounts for 74.5% of the total. This is an average value on the entire dataset, and clearly depends on the number of Steiner points required by each specific model (see Fig. 11). For the model requiring the longest execution time, segment recovery takes 86% of the total and adds 10216352 Steiner points. This shows where future research should focus to target further performance improvement (Sec. 6.3).

Memory usage is also kept within reasonable bounds and allows processing all the selected 4408 Thingi10k models even on an average home desktop PC. On average, the peak memory allocated while processing a model is 46.5 Mb (as measured by *getrusage()* [1]), while worst case requires 8.88 Gb.

[1] https://www.gnu.org/software/libc/manual/html_node/Resource-Usage.html

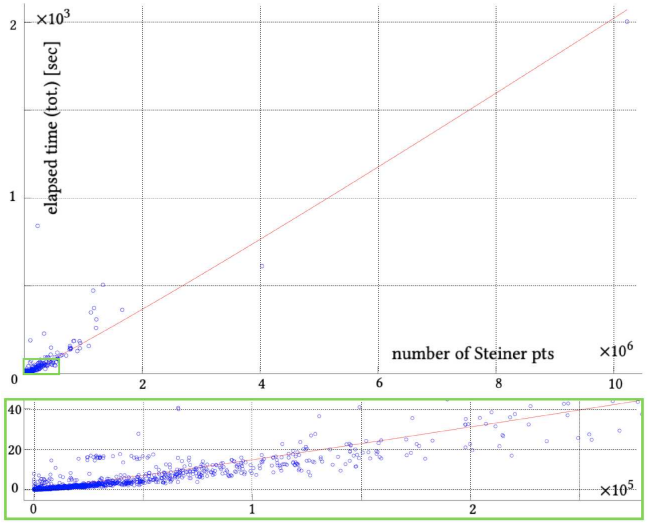


Fig. 11. Elapsed time vs. number of Steiner points inserted. Each blue circle represents one of the 4408 models in our dataset. The red line is a least-square fit of a function of the form $t = A + Bn\log(n)$, with $A = 0.452$ and $B = 0.0000125$. A zoom-in near the origin is shown in the bottom part.

Input File	Indirect Predicates - Ind.Pred. TIME (ms)	Lazy GNU GMPQ TIME (ms)	Lazy GNU GMPQ / Ind.Pred.	GNU GMPQ TIME (ms)	GNU GMPQ / Ind.Pred.	Lazy CORE TIME (ms)	Lazy CORE / Ind.Pred.	CORE TIME (ms)	CORE / Ind.Pred.
100028.off	4	54	13,50	74	18,50	110	27,50	99	24,75
100031.off	28	213	7,61	382	13,64	56981	2035,04	66542	2376,50
100033.off	16	175	10,94	215	13,44	1639	102,44	2564	160,25
100034.off	3	56	18,67	75	25,00	1510	503,33	4830	1610,00
100035.off	94	598	6,36	1712	18,21	146180	1555,11	137907	1467,10
100070.off	204	2247	11,01	7296	35,76	319051	1563,98	415202	2035,30
100071.off	96	786	8,19	2233	23,26	1119005	11656,30	2377043	24760,86
100072.off	10	104	10,40	167	16,70	343	34,30	4578	457,80
100073.off	8	125	15,63	251	31,38	171	21,38	86	10,75
100077.off	9	152	16,89	341	37,89	250	27,78	92	10,22
100173.off	11	83	7,55	215	19,55	30780	2798,18	31801	2891,00
100336.off	3091	27795	8,99	100913	32,65	4519715	1462,22	17932263	5801,44
100349.off	168	3069	18,27	11365	67,65	15404	91,69	28862	171,80
100388.off	1899	46088	24,27	170792	89,94	108730	57,26	443243	233,41
100423.off	33	160	4,85	575	17,42	19385	587,42	17346	525,64
100506.off	1029	14464	14,06	54172	52,65	1170495	1137,51	3999830	3887,10
100507.off	825	11145	13,51	41336	50,10	872822	1057,97	3110036	3769,74
1005285.off	16	140	8,75	285	17,81	304	19,00	107	6,69
100681.off	13	101	7,77	271	20,85	8302	638,62	8161	627,77
100728.off	1	30	30,00	23	23,00	69	69,00	29	29,00
Average	377,9	5379,25	12,86	19634,65	31,27	419562,30	1272,30	1429031,05	2542,86

Fig. 12. This table reports processing times for the first 20 models in our dataset while using our algorithm implemented with different number types as described in section 4. Yellow columns show how slower each version is when compared with the fast version based on indirect predicates.

Our algorithm inserts less than 50k Steiner points in 91% of our test dataset. About 25k Steiner points are used on average, with a worst case requiring 10 million points. The average decreases to 4.9k if the 9% of models requiring more than 50k Steiner points are not considered.

Depending on the number type used, our tool has fairly different performances, although it remains robust in all cases. Fig. 12 reports a comparison of the various versions for the first twenty models in our dataset. All the versions were compiled with full optimization and run on the same machine. Yellow columns show how slower each version is when compared with the fast version based on indirect predicates, revealing that CORE is thousands of times slower, even when wrapped around a lazy evaluation framework. GMP rationals perform better, but are still more than one order of magnitude slower.

6.2 Comparison

Our method is fundamental in all cases where downstream applications expect the mesh to be valid (i.e. no flipped or degenerate elements allowed) and exactly match the input surface. Furthermore, downstream applications hardly support exact rational coordinates, meaning that the aforementioned valid mesh must be representable using floating-point coordinates. These requirements are rather common in practice, but nonetheless strongly limit the set of appropriate tools that are comparable to ours. We compare against the Delaunay mesher `DelPSC` [Cheng et al. 2007], `CGAL` meshing with sharp feature preservation [Rineau and Yvinec 2007], the isosurface stuffing implemented in `Quartet` [Bridson and Doran 2014], the recent `tetwild` algorithm [Hu et al. 2018], the state-or-the-art software `tetgen` [Si and Gärtner 2005], and the recent fast and robust polyhedral meshing algorithm DA2021 [Diazzi and Attene 2021].

We used the implementations provided by their authors. When mandatory input parameters are required, we tuned them to obtain the best possible result within 1 hour.

`DelPSC`, `CGAL`, `Quartet`, and `tetwild` produce approximate boundary conformity, even when sharp feature preservation is used. Hence, the meshes produced do not exactly match the input (see Fig. 13). In principle, `CGAL` can be configured to force any non-flat edge to be preserved in the output, hence leading to an exactly conformal mesh. However, in many cases, the algorithm fails to converge with this setting. For example, the result shown in Fig. 13 was obtained by asking `CGAL` to preserve all edges whose incident triangle normals form an angle of at least 15 degrees. Any attempt to lower this value leads to convergence failure.

We consequently focus our comparison on the remaining two algorithms, `tetgen` and DA2021, which can exactly preserve the input. On the subset of 4030 models where both these methods succeed, we compare the elapsed time and memory usage (Fig. 14). In terms of performance, our algorithm is comparable with both `tetgen` and DA2021, while requiring consistently less memory. Our implementation can process 19.4% of the models in less than 0.01 seconds, whereas `tetgen` achieves this speed for 14.4% of the models and DA2021 for 10.2% (file reading and writing times are excluded). In contrast, our method can process 95.2% of the models within 10 seconds, compared to 98.5% for `tetgen` and 97.7% for DA2021.

The results produced by `tetgen` are always representable using floating-point coordinates, which is not surprising as `tetgen` uses this number type in its entire pipeline. Nonetheless, it fails in 378 out of 4408 models (8.6%) in our test dataset. One might argue that `tetgen` fails because it snaps together input vertices if they are closer than a certain threshold (by default set to 10^{-8}), introducing invalid input configurations. To counter that, we re-run the tests after having set the threshold to zero (`-T0` option in `tetgen`), which in fact increased the number of failures from 378 to 720.

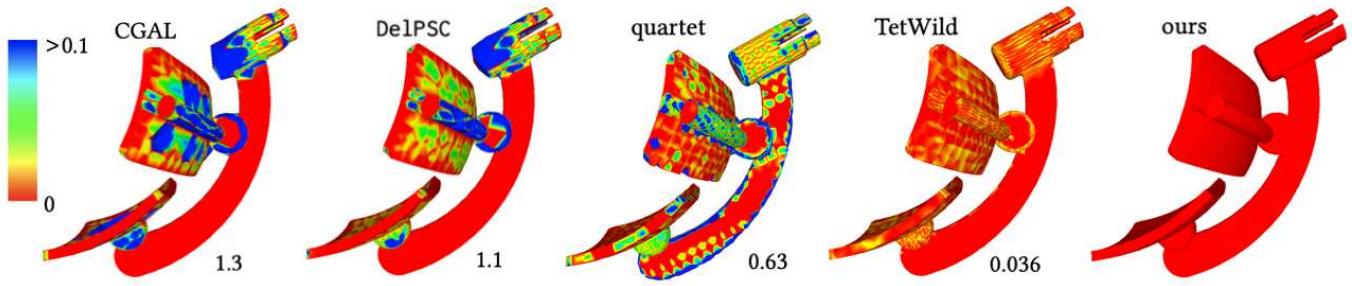


Fig. 13. Tetrahedral meshes produced by different tools on the same input (Thingi10k model no. 112544). Colors depict the Hausdorff distance between the tetrahedral mesh surface and the input PLC. The maximum distance is indicated alongside each model. CGAL was configured with *threshold angle*=15 and *minimum radius edge ratio*=2, DelPSC with *radius edge ratio*=2 and *feature angle*=120, quartet with *grid spacing*=0.5 and *sharp features threshold angle*=170. Other methods were run with default settings. On this model tetgen fails, whereas DA2021 degenerates after rounding to floating points. Our method is the only one that *exactly* conforms with the input PLC while being representable in floating point precision.

Conversely, DA2021 succeeds on all the models in our dataset, but only 61.7% of the results can be represented using floating-point coordinates with no flips and/or flattenings. When comparing against DA2021 we must first consider that this method was not designed to produce tetrahedra, though the convex polyhedral cells created can always be tetrahedrized. To do this, the authors suggest first triangulating the faces and then inserting a point at the barycenter of each non-tetrahedral cell. Approximating this barycenter using floating-point coordinates might easily invalidate the mesh. Indeed, since a polyhedral cell can be arbitrarily bad-shaped, there is no guarantee that its internal volume contains a point representable using floating-point coordinates. One might argue that the same problem may affect the Steiner points produced by our method, that is, tetrahedra may flip after having rounded the coordinates to their closest FP-representable values. While this is indeed the case, in practice our meshes are constrained Delaunay, meaning that their quality is not as arbitrarily bad as in DA2021. Thus, our meshes are still valid after rounding in 93.22% of the cases, compared to the 61.7% for DA2021. Furthermore, with the additional post-processing discussed in Sec. 5.1, the mesh connectivity can be modified to eliminate most of the tets that degenerate or flip after rounding, hence increasing the percentage of valid rounded models produced by our method to 99.77%, without any auxiliary vertices. In our implementation this latter post-processing step is optional: users can choose whether the output file must have exact rational coordinates or floating point coordinates and, in the latter case, whether to run the post-processing or not.

A theoretical question remains: is it always possible to create a valid tet mesh, possibly with Steiner points, which exactly conforms to an input surface even after rounding? The answer is no, and a counterexample is as follows. Floating-point numbers are discrete, and points whose coordinates belong to this set form a grid in space. This grid is not regular as it is denser near the origin, but for our discussion, we may imagine it to be uniform. Let ϵ be the difference between two consecutive representable numbers, and let $p = (x, y, z)$ be one representable point. The eight points $p_i = (x + i\epsilon, y + j\epsilon, z + k\epsilon)$, with $i, j, k \in \{0, 1\}$, form a cube. Now, we can take three points from its lower base (e.g. third coordinate = z) and other three points from the upper base (third coordinate = $z + \epsilon$) and

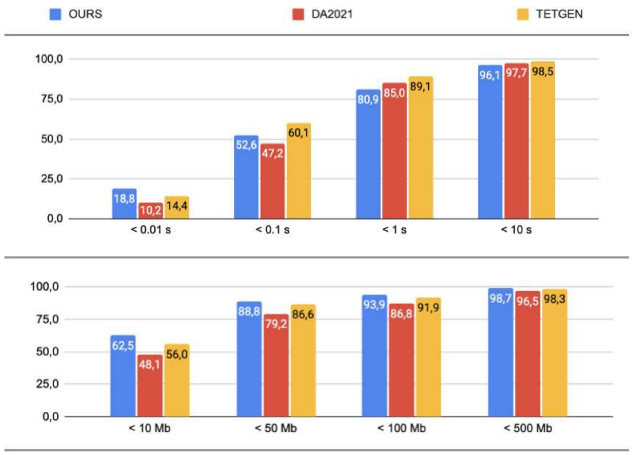


Fig. 14. Time (top) and memory usage (bottom) comparisons between our method, DA2021, and `tetgen`. Each bar represents the percentage of models in the dataset that could be processed within the limit condition at its bottom.

form a Schönhardt polyhedron out of them [Schönhardt 1928]. This polyhedron is completely contained in the aforementioned cube and, in order to be tetrahedrized, would require at least one Steiner point within the cube itself. However, the cube does not contain any other representable point within the ϵ resolution.

6.3 Limitations and discussion

While robust, our algorithm has some limitations. As one example, the use of edge-based Steiner points means we do not conform to the *connectivity* of the original PLC. Even if Steiner points can be moved to the interior (Sec. 7 in [Si and Gärtner 2011]), our tool does not support this feature as it would require an additional type of implicit point for the interior Steiner vertices. Even if our algorithm would be extended this way in the future, a PLC may still have coplanar triangular faces that form a non-Delaunay 2D triangulation and, in these cases, we cannot guarantee a conformal connectivity while still being constrained Delaunay.

Furthermore, as mentioned, the 3D Delaunay condition does not necessarily correspond to high-quality meshes [Alexa 2019] and our tool does not include a full-fledged mesh optimization phase. Mesh optimizers based on Delaunay refinement [Shewchuk and Si 2014] are mostly implemented using floating-point arithmetic that might spoil their convergence guarantees. Hence, even if our CDTs can be used to initialize these algorithms, there is no guarantee of success. To have such guarantees we would need to define new implicit points and indirect predicates to cope with mesh refinement, whose robust reimplementation is a very interesting direction for further research.

In general, converting implicit points to floating-point coordinates with no flips remains an unsolved problem (though rare in practice), and further research is needed to discover which conditions make a PLC admit a *floating-point representable* CDT.

Finally, we do not exploit modern parallel architectures. Parallelization already proved to be beneficial when computing unconstrained Delaunay tetrahedrizations [Marot et al. 2019], and we believe it represents a feasible improvement even for the constrained case.

7 CONCLUSIONS

We showed that, through a clever exploitation of floating point arithmetic, algorithmic efficiency and numerical robustness can be combined when calculating Steiner CDTs. Also, our research has uncovered an algorithmic issue that invalidates the theoretical guarantees of the widely used `tetgen` software. This made us able to implement a theoretically correct version of the algorithm which is also robust and fast, while bringing the failure rate from 8.6% of the cases to zero. This represents a significant advancement in this area because, to the best of our knowledge, no previous algorithm was capable of computing CDTs as robustly as we do.

As an interesting direction for future research, it is worth trying to optimize our CDTs with convergence guarantees to produce high quality meshes. In principle, the algorithm in [Shewchuk and Si 2014] provides such guarantees, but in practice its floating point based implementation available in `tetgen` may easily fail or not terminate.

ACKNOWLEDGMENTS

L. Diazzi is partly supported by the Unimore FAR Mission Oriented project 2021 “Artificial Intelligence-based Mathematical Models and Methods for low dose CT imaging”. M. Attene is partly supported by CNR STM Project on “Robust, Flexible and Performing Algorithms to Mesh 3D Domains”. This work was partially supported by the NSF grants OAC-1835712 and CHS-1908767. We would like to thank Silvia Sellán and Alec Jacobson for fruitful discussions about radical number types.

REFERENCES

Frédéric Alauzet and David Marcum. 2014. A closed advancing-layer method with changing topology mesh movement for viscous mesh generation. In *Proceedings of the 22nd international meshing roundtable*. Springer, 241–261.

Marc Alexa. 2019. Harmonic Triangulations. *ACM Trans. Graph.* 38, 4, Article 54 (jul 2019), 14 pages. https://doi.org/10.1145/3306346.3322986

Marc Alexa. 2020. Conforming Weighted Delaunay Triangulations. *ACM Trans. Graph.* 39, 6, Article 248 (nov 2020), 16 pages. https://doi.org/10.1145/3414685.3417776

Pierre Alliez, David Cohen-Steiner, Mariette Yvinec, and Mathieu Desbrun. 2005. Variational tetrahedral meshing. In *ACM SIGGRAPH 2005 Papers*. 617–625.

M. Attene. 2019. Indirect Predicates Library. https://github.com/MarcoAttene/Indirect_Predicates.

Marco Attene. 2020. Indirect Predicates for Geometric Constructions. *Computer-Aided Design* 126 (2020), 102856. https://doi.org/10.1016/j.cad.2020.102856

Adrian Bowyer. 1981. Computing Dirichlet tessellations. *Comput. J.* 24, 2 (1981), 162–166. https://doi.org/10.1093/comjnl/24.2.162

Robert Bridson and Crawford Doran. 2014. Quartet: A tetrahedral mesh generator that does isosurface stuffing with an acute tetrahedral tile. *https://github. com/crawford-doran/quartet* (2014).

Jonathan Bronson, Joshua A Levine, and Ross Whitaker. 2013. Lattice cleaving: A multimaterial tetrahedral meshing algorithm with guarantees. *IEEE transactions on visualization and computer graphics* 20, 2 (2013), 223–237.

C. Burnikel, K. Mehlhorn, and S. Schirra. 1996. *The LEDA Class Real Number*. Max-Planck-Institut für Informatik. https://books.google.it/books?id=ND5LvwEACAAJ

Daniela Cabiddu and Marco Attene. 2017. epsilon-maps: Characterizing, detecting and thickening thin features in geometric models. *Computers & Graphics* 66 (2017), 143–153. https://doi.org/10.1016/j.cag.2017.05.014 Shape Modeling International 2017.

Bernard Chazelle. 1984. Convex partitions of polyhedra: a lower bound and worst-case optimal algorithm. *SIAM J. Comput.* 13, 3 (1984), 488–507.

Siu-Wing Cheng, Tamal Dey, and Joshua Levine. 2007. A Practical Delaunay Meshing Algorithm for a Large Class of Domains*. *Proceedings of the 16th International Meshing Roundtable*, 477–494. https://doi.org/10.1007/978-3-540-75103-8_27

S. W. Cheng and T. K. Dey. 2002. Quality meshing with weighted Delaunay refinement. *Proc. 13th ACM-SIAM Sympos.Discrete Algorithms (SODA2002)*, 137–146.

S. W. Cheng, T. K. Dey, H. Edelsbrunner, M. A. Facello, and S.-H. Teng. 2000. Sliver exudation. *Journal of ACM* 47 (2000), 883–904.

Gianmarco Cherchi, Marco Livesu, Riccardo Scateni, and Marco Attene. 2020. Fast and Robust Mesh Arrangements Using Floating-Point Arithmetic. *ACM Trans. Graph.* 39, 6, Article 250 (nov 2020), 16 pages. https://doi.org/10.1145/3414685.3417818

L. P. Chew. 1989. Constrained Delaunay triangulations. *Algorithmica* 4 (1989), 97–108.

David Cohen-Steiner, Eric Colin De Verdiere, and Mariette Yvinec. 2002. Conforming Delaunay triangulations in 3D. In *Proceedings of the eighteenth annual symposium on Computational geometry*. 199–208.

Jean-Christophe Cuillière, Vincent Francois, and Jean-Marc Drouet. 2013. Automatic 3D mesh generation of multiple domains for topology optimization methods. In *Proceedings of the 21st International Meshing Roundtable*. Springer, 243–259.

Tamal K. Dey and Wulue Zhao. 2003. Approximating the Medial Axis from the Voronoi Diagram with a Convergence Guarantee. *Algorithmica* 38 (2003), 179–200.

Lorenzo Diazzi and Marco Attene. 2021. Convex polyhedral meshing for robust solid modeling. *ACM Transactions on Graphics (TOG)* 40 (2021), 1 – 16.

C Dobrzynski and P. Frey. 2008. Anisotropic Delaunay mesh adaptation for unsteady simulations. In *Proceedings of the 17th international Meshing Roundtable*.

Crawford Doran, Athena Chang, and Robert Bridson. 2013. Isosurface stuffing improved: acute lattices and feature matching. In *ACM SIGGRAPH 2013 Talks*. 1–1.

Qiang Du and Desheng Wang. 2003. Tetrahedral mesh generation and optimization based on centroidal Voronoi tessellations. *International journal for numerical methods in engineering* 56, 9 (2003), 1355–1373.

Xingyi Du, Qingnan Zhou, Nathan Carr, and Tao Ju. 2022. Robust Computation of Implicit Surface Networks for Piecewise Linear Functions. *ACM Trans. Graph.* 41, 4, Article 41 (jul 2022), 16 pages. https://doi.org/10.1145/3528223.3530176

Herbert Edelsbrunner and Ernst Peter Mücke. 1990. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics (tog)* 9, 1 (1990), 66–104.

Andreas Fabri and Sylvain Pion. 2009. CGAL: The computational geometry algorithms library. In *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*. 538–539.

Pascal J. Frey, Houman Borouchaki, and Paul Louis George Inria. 1996. Delaunay Tetrahedralization using an Advancing-Front Approach.

PL. George, F. Hecht, and E. Saltel. 1991. Automatic mesh generator with specified boundary. *Comp Methods Appl Mechanics and Engineering* 92 (1991), 269–288.

Torbjörn Granlund. 1996. Gnu mp. *The GNU Multiple Precision Arithmetic Library* 2, 2 (1996).

Zhenqun Guan, Chao Song, and Yuanxian Gu. 2006. The boundary recovery and sliver elimination algorithms of three-dimensional constrained Delaunay triangulation. *Internat. J. Numer. Methods Engrg.* 68, 2 (2006), 192–209. https://doi.org/10.1002/nme.1707 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.1707

Robert Haimes. 2015. MOSS: multiple orthogonal strand system. *Engineering with Computers* 31 (2015), 453–463.

Yixin Hu, Teseo Schneider, Bolun Wang, Denis Zorin, and Daniele Panozzo. 2020. Fast Tetrahedral Meshing in the Wild. *ACM Trans. Graph.* 39, 4, Article 117 (aug 2020), 18 pages. https://doi.org/10.1145/3386569.3392385

Yixin Hu, Qingnan Zhou, Xifeng Gao, Alec Jacobson, Denis Zorin, and Daniele Panozzo. 2018. Tetrahedral Meshing in the Wild. *ACM Trans. Graph.* 37, 4, Article 60 (July

2018), 14 pages. https://doi.org/10.1145/3197517.3201353
Alec Jacobson, Ladislav Kavan, and Olga Sorkine-Hornung. 2013. Robust inside-outside segmentation using generalized winding numbers. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 1–12.
Clément Jamin, Pierre Alliez, Mariette Yvinec, and Jean-Daniel Boissonnat. 2015. CGALmesh: a generic framework for delaunay mesh generation. *ACM Transactions on Mathematical Software (TOMS)* 41, 4 (2015), 1–24.
B. Joe. 1991. GEOMPACK — A software package for the generation of meshes using geometric algorithms. *Adv. Engin. Software* 51 (1991), 325–331.
V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. 1999. A Core Library for robust numeric and geometric computation. In *Procs 15th ACM Symp on Computational Geometry (SoCG)*. 351–359.
François Labelle and Jonathan Richard Shewchuk. 2007. Isosurface Stuffing: Fast Tetrahedral Meshes with Good Dihedral Angles. *ACM Trans. Graph.* 26, 3 (jul 2007), 57–es. https://doi.org/10.1145/1276377.1276448
A. Lagae and P. Dutre. 2008. Accelerating Ray Tracing using Constrained Tetrahedralizations. *Computer Graphics Forum* 4 (2008), 1303–1312.
Der-Tsai Lee and Arthur K. Lin. 1986. Generalized Delaunay Triangulations for Planar Graphs. *Discrete & Computational Geometry* 1 (1986), 201–217.
C. Li, S. Pion, and C.K. Yap. 2005. Recent progress in exact geometric computation. *The Journal of Logic and Algebraic Programming* 64, 1 (2005), 85 – 111. https://doi.org/10.1016/j.jlap.2004.07.006 Practical development of exact real number computation.
Manish Mandad, David Cohen-Steiner, and Pierre Alliez. 2015. Isotopic approximation within a tolerance volume. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 1–12.
Célestin Marot, Jeanne Pellerin, and Jean-François Remacle. 2019. One machine, one minute, three billion tetrahedra. *Internat. J. Numer. Methods Engrg.* 117, 9 (2019), 967–990.
Gary L Miller, Dafna Talmor, Shang-Hua Teng, Noel Walkington, and Han Wang. 1996. Control volume meshes using sphere packing: Generation, refinement and coarsening. *Proc. of 5th Intl. Meshing Roundtable* (1996).
MMG3D. 2004. Mmg Platform - Robust open-source and multidisciplinary software for remeshing. https://www.mmgtools.org/.
Neil Molino, Robert Bridson, and Ronald Fedkiw. 2003. Tetrahedral mesh generation for deformable bodies. In *Proc. Symposium on Computer Animation*, Vol. 8.
Michael Murphy, David M Mount, and Carl W Gable. 2001. A point-placement strategy for conforming Delaunay tetrahedralization. *International Journal of Computational Geometry & Applications* 11, 06 (2001), 669–682.
Laurent Rineau and Mariette Yvinec. 2007. A generic software design for Delaunay refinement meshing. *Computational Geometry* 38, 1-2 (2007), 100–110.
J. Ruppert. 1995. A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation. *Journal of Algorithms* 18, 3 (1995), 548–585. https://doi.org/10.1006/jagm.1995.1021
Jim Ruppert and Raimund Seidel. 1992. On the difficulty of triangulating three-dimensional Nonconvex Polyhedra. *Discrete & Computational Geometry* 7, 3 (mar 1992). https://doi.org/10.1007/BF02187840
E. Schönhardt. 1928. Über die Zerlegung von Dreieckspolyedern in Tetraeder. *Math. Ann.* 86 (1928), 309–312. https://doi.org/10.1007/BF01451597
Chen Shen, James F O'Brien, and Jonathan R Shewchuk. 2004. Interpolating and approximating implicit surfaces from polygon soup. In *ACM SIGGRAPH 2004 Papers*. 896–904.
Jonathan Shewchuk. 1998. A Condition Guaranteeing the Existence of Higher-Dimensional Constrained Delaunay Triangulations. *14th Ann. ACM Symp. Comp. Geom.* (06 1998). https://doi.org/10.1145/276884.276893
Jonathan Richard Shewchuk. 1997. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry* 18 (1997), 305–363. https://doi.org/10.1007/PL00009321
Jonathan Richard Shewchuk. 2000a. Mesh generation for domains with small angles. In *Proceedings of the sixteenth annual Symposium on Computational Geometry*. 1–10.
Jonathan Richard Shewchuk. 2000b. Sweep Algorithms for Constructing Higher-Dimensional Constrained Delaunay Triangulations. In *Proceedings of the Sixteenth Annual Symposium on Computational Geometry* (Clear Water Bay, Kowloon, Hong Kong). Association for Computing Machinery, New York, NY, USA, 350–359. https://doi.org/10.1145/336154.336222
Jonathan Richard Shewchuk. 2002. Constrained Delaunay Tetrahedralizations and Provably Good Boundary Recovery. In *International Meshing Roundtable Conference*.
Jonathan Richard Shewchuk. 2003. Updating and constructing constrained Delaunay and constrained regular triangulations by flips. In *Proceedings of the nineteenth annual symposium on Computational geometry*. 181–190.
Jonathan Richard Shewchuk and Hang Si. 2014. Higher-quality tetrahedral mesh generation for domains with small angles by constrained delaunay refinement. In *Proceedings of the thirtieth annual symposium on Computational geometry*. 290–299.
H. Si. 2008. Adaptive tetrahedral mesh generation by constrained Delaunay refinement. *Internat. J. Numer. Methods Engrg.* 75, 7 (2008), 856–880. https://doi.org/10.1002/nme.2318 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.2318
Hang Si and Klaus Gärtner. 2005. Meshing Piecewise Linear Complexes by Constrained Delaunay Tetrahedralizations. *Proceedings of the 14th International Meshing Roundtable*, 147–163. https://doi.org/10.1007/3-540-29090-7_9
Hang Si and Klaus Gärtner. 2011. 3D boundary recovery by constrained Delaunay tetrahedralization. *Int. J. Numer. Meth. Engng* 85 (2011), 1341–1364.
J Tournois, C Wormser, P Alliez, and M Desbrun. 2009. Interleaving Delaunay Refinement and Optimization. *ACM Trans. Graphics* 28, 3 (2009).
David F. Watson. 1981. Computing the n-dimensional Delaunay Tessellation with Application to Voronoi Polytopes. *Comput. J.* 24, 2 (1981), 167–172.
NP. Weatherill and O. Hassan. 1994. Efficient three-dimensional Delaunay triangulation with automatic point creation and imposed boundary constraints. *Intnl J Num Methods in Engineering* 37 (1994), 2005–2039.
Zhoufang Xiao, Jianjun Chen, Yao Zheng, Jianjing Zheng, and Desheng Wang. 2016. Booleans of triangulated solids by a boundary conforming tetrahedral mesh generation approach. *Computers Graphics* 59 (2016), 13–27. https://doi.org/10.1016/j.cag.2016.04.004
Qingnan Zhou and Alec Jacobson. 2016. Thingi10K: A Dataset of 10,000 3D-Printing Models. *arXiv preprint arXiv:1605.04797* (2016).

A FLOATING POINT FILTERS

The expression for all the versions of the `orient3D` and `inSphere` predicates can be obtained by replacing plain coordinates in Eqns. 1 and 3 respectively with the expression of LNC coordinates as shown, for example, in Eqn. 2. In the following, extensions in the predicate name indicate the type of argument points (e.g. in `orient3D_LEEE` the first point is implicit whereas the other three are explicit, in `inSphere_LLLLL` all the points are implicit). The first implicit point is denoted with i_1 and is equal to $t_1 p_1 + (1 - t_1) q_1$, hence its x coordinate is $t_1 p_{1x} + (1 - t_1) q_{1x}$. With this notation, we can now define the filters for all the versions of the predicates used. Filter values were calculated using [Attene 2020]. If the absolute value of the predicate calculated using floating point arithmetic is less than the filter value ε_Δ, then it can be re-evaluated with more precision using intervals and, if necessary, exactly through expansion arithmetic [Attene 2020].

$$\texttt{orient3d\_LEEE}(i_1, p_2, p_3, p_4): \quad \varepsilon_\Delta = 1.718625242119744\ 10^{-13} \delta_\Delta^6$$
$$\texttt{orient3d\_LLEE}(i_1, i_2, p_3, p_4): \quad \varepsilon_\Delta = 2.495781359357355\ 10^{-13} \delta_\Delta^6$$
$$\texttt{orient3d\_LLLE}(i_1, i_2, i_3, p_4): \quad \varepsilon_\Delta = 3.836930773104546\ 10^{-13} \delta_\Delta^6$$
$$\texttt{orient3d\_LLLL}(i_1, i_2, i_3, i_4): \quad \varepsilon_\Delta = 5.68434188608081\ 10^{-13} \delta_\Delta^6$$

$$\texttt{inSphere\_LEEEE}(i_1, p_2, p_3, p_4, p_5): \quad \varepsilon_\Delta = 5.295763827462003\ 10^{-13} \delta_\Delta^7$$
$$\texttt{inSphere\_LLEEE}(i_1, i_2, p_3, p_4, p_5): \quad \varepsilon_\Delta = 2.218669692410916\ 10^{-12} \delta_\Delta^8$$
$$\texttt{inSphere\_LLLEE}(i_1, i_2, i_3, p_4, p_5): \quad \varepsilon_\Delta = 9.019007762844938\ 10^{-12} \delta_\Delta^9$$
$$\texttt{inSphere\_LLLLE}(i_1, i_2, i_3, i_4, p_5): \quad \varepsilon_\Delta = 3.581668295282733\ 10^{-11} \delta_\Delta^{10}$$
$$\texttt{inSphere\_LLLLL}(i_1, i_2, i_3, i_4, i_5): \quad \varepsilon_\Delta = 1.991793396882719\ 10^{-10} \delta_\Delta^{10}$$

The value of δ_Δ for an `orient3D` version with n implicit arguments is:

$$\delta_\Delta = \max\{\delta_{\Delta i}, \delta_{\Delta e}\}$$
$$\delta_{\Delta i} = \max_{k \in 1..n} \{|p_{kx}|, |p_{ky}|, |p_{kz}|, |q_{kx} - p_{kx}|, |q_{ky} - p_{ky}|, |q_{kz} - p_{kz}|, |t_k|\}$$
$$\delta_{\Delta e} = \max_{k \in n+1..4} \{|p_{kx}|, |p_{ky}|, |p_{kz}|\}$$

The value of δ_Δ for an `inSphere` version with n implicit arguments is:

$$\delta_\Delta = \max\{\delta_{\Delta i}, \delta_{\Delta e}, \delta_{\Delta w}\}$$

$$\delta_{\Delta i} = \max_{k \in 1..n} \{|p_{kx}|, |p_{ky}|, |p_{kz}|, |q_{kx} - p_{kx}|, |q_{ky} - p_{ky}|, |q_{kz} - p_{kz}|, |t_k|\}$$

$$\delta_{\Delta e} = \max_{k \in n+1..4} \{|p_{kx} - p_{5x}|, |p_{ky} - p_{5y}|, |p_{kz} - p_{5z}|\}$$

$$\delta_{\Delta w} = \begin{cases} 0 & \text{if } n = 5 \\ \max\{|p_{5x}|, |p_{5y}|, |p_{5z}|\} & \text{otherwise} \end{cases}$$

B TETRAHEDRON VALIDITY FOR GIFT-WRAPPING

With reference to section 4.4, a tetrahedron $T =< t_0, t_1, t_2, w >$ is *valid* if conditions *i*), *ii*) and *iii*) hold. Condition *i*) is equivalent to checking whether `orient3d`$(t_0, t_1, t_2, w) > 0$. Condition *ii*) holds if, for each triangle $\tau \in \partial C_{curr}$, one of the following holds:

- τ and T share three vertices (i.e. τ is a face of T);
- τ and T share two vertices **and** at least one of the tets obtained by replacing one of the unshared vertices in T with the unshared vertex in τ have negative volume;
- τ and T share one vertex **and** the other two vertices of τ are not contained in the volume of T **and** no edge of τ intersects a face of T **and** no edge of T intersects τ except for the common vertex;
- τ and T have no common vertices **and** no vertex of τ is contained in the volume of T **and** no edge of τ intersects a face of T **and** no edge of T intersects τ.

Note that this approach requires *point in tetrahedron* and *segment-triangle intersection* tests only. All can be checked exactly through simple calls to `orient3d` (see Algorithms 3 and 4).

ALGORITHM 3: point_in_tetrahedron(p, T)

Input:
p: query point to be checked;
$T =< t_0, t_1, t_2, t_3 >$: reference tetrahedron.
Output:
true if p belongs to the volume of T (including its boundary).

---

if `orient3d`$(t_0, t_1, t_2, p) < 0$ **then return false** ;
if `orient3d`$(t_0, t_1, p, t_3) < 0$ **then return false** ;
if `orient3d`$(t_0, p, t_2, t_3) < 0$ **then return false** ;
if `orient3d`$(p, t_1, t_2, t_3) < 0$ **then return false** ;
return true

ALGORITHM 4: segment_intersects_triangle(s, t)

Input:
$s =< s_1, s_2 >$: segment;
$t =< v_0, v_1, v_2 >$: triangle.
Output:
true if s and t intersect while not being coplanar.

---

if `orient3d`(v_0, v_1, v_2, s_1) `=` `orient3d`(v_0, v_1, v_2, s_2) **then return false** ;
if `orient3d`(v_0, v_1, s_1, s_2) `*` `orient3d`$(v_1, v_2, s_1, s_2) < 0$ **then return false** ;
if `orient3d`(v_1, v_2, s_1, s_2) `*` `orient3d`$(v_2, v_0, s_1, s_2) < 0$ **then return false** ;
if `orient3d`$(2_0, v_0, s_1, s_2)$ `*` `orient3d`$(v_0, v_1, s_1, s_2) < 0$ **then return false** ;
return true

Condition *iii*) holds if, for each vertex v in the half-cavity, either v is not in the circumsphere of T or it is not visible from within T. v is not visible from within T even if just a portion of T's relative interior is occluded by the initial half-cavity boundary ∂C_i. Hence, in order for v to be visible, all the internal points in T must be visible from v. Equivalently, v is visible from within T if and only if, for each triangle τ in ∂C_i, $I(CH(T \cup v)) \cap \tau = \emptyset$, where $I(.)$ denotes the *interior* operator and $CH(.)$ denotes the *convex hull* operator. Because $CH(T \cup v)$ can be made of one or two tetrahedra, the previous approach to detect triangle-tetrahedron intersection can be reused.