

Integer Coordinates for Intrinsic Geometry Processing

MARK GILLESPIE, NICHOLAS SHARP, and KEENAN CRANE, Carnegie Mellon University

In this work, we present a general, efficient, and provably robust representation for intrinsic triangulations. These triangulations have emerged as a powerful tool for robust geometry processing of surface meshes, taking a low-quality mesh and retriangulating it with high-quality intrinsic triangles. However, existing representations either support only edge flips, or do not offer a robust procedure to recover the common subdivision, that is, how the intrinsic triangulation sits along the original surface. To build a general-purpose robust structure, we extend the framework of *normal coordinates*, which have been deeply studied in topology, as well as the more recent idea of *roundabouts* from geometry processing, to support a variety of mesh processing operations like vertex insertions, edge splits, etc. The basic idea is to store an integer per mesh edge counting the number of times a curve crosses that edge. We show that this paradigm offers a highly effective representation for intrinsic triangulations with strong robustness guarantees. The resulting data structure is general and efficient, while offering a guarantee of always encoding a valid subdivision. Among other things, this allows us to generate a high-quality intrinsic Delaunay refinement of all manifold meshes in the challenging Thingi10k dataset for the first time. This enables a broad class of existing surface geometry algorithms to be applied out-of-the-box to low-quality triangulations.

CCS Concepts: • Mathematics of computing → Mesh generation.

Additional Key Words and Phrases: remeshing, intrinsic triangulation, Delaunay triangulation, discrete differential geometry

1 INTRODUCTION AND RELATED WORK

Geometric data plays a growing role in applications from computational fabrication to autonomous driving to augmented reality, but data in these applications is increasingly difficult to deal with due to the poor quality of meshes generated by non-expert users, or by algorithms targeted at visualization rather than mesh processing—there have hence been significant recent efforts to make geometric algorithms more robust [Zhou et al. 2016; Hu et al. 2018; Sellán et al. 2019; Sawhney and Crane 2020]. One basic tool is to remesh the input to obtain higher-quality elements, but for this approach to work on difficult, near-degenerate inputs, remeshing algorithms must themselves be extremely robust. Moreover, traditional approaches to remeshing based on vertex positions in \mathbb{R}^n must negotiate the trade-off between mesh size, the quality of mesh elements, and geometric approximation of the input domain.

Intrinsic Triangulations. A promising idea is to approach geometry processing from the *intrinsic* point of view: rather than considering the embedding of the geometry in space, one focuses only on point-to-point distances along the surface, as encoded by the edge

Authors' address: Mark Gillespie; Nicholas Sharp; Keenan Crane, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA, 15213.

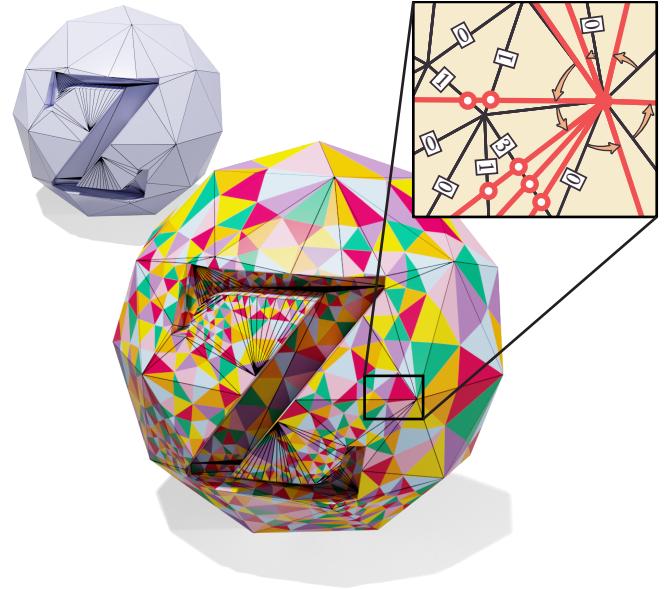


Fig. 1. We extend the machinery of normal coordinates beyond just edge flips, to enable a broader set of local mesh operations such as vertex insertion. By doing so, we dramatically improve the robustness of algorithms like Delaunay refinement in the intrinsic setting.

lengths of a triangulation. This perspective is quite natural for problems in geometry processing and scientific computing, since many objects in these domains are themselves intrinsic—for instance, the Laplace-Beltrami operator, which appears in numerous algorithms and fundamental partial differential equations (PDEs). In this paper we consider so-called *intrinsic triangulations*, whose edges no longer need to be straight line segments in Euclidean space, but can instead be any straight or *geodesic* path across the input polyhedron (see Figure 2). This construction still captures the input geometry exactly, but provides a dramatically larger space of possible triangulations, lending enormous flexibility to geometric algorithms. For instance, it de-couples the quality of elements used for simulation from the elements used to describe the geometry, side-stepping the trade-off encountered in traditional meshing. Moreover, this perspective enables the input polyhedral surface to serve as a background domain—analogous to the Euclidean plane in traditional computational geometry—allowing one to “port” trusted algorithms from the plane to curved surfaces. Most importantly, by encapsulating all this machinery in an interface that resembles an ordinary mesh, one can provide robustness as a subroutine: rather than make existing algorithms more robust one by one, we can transform the input

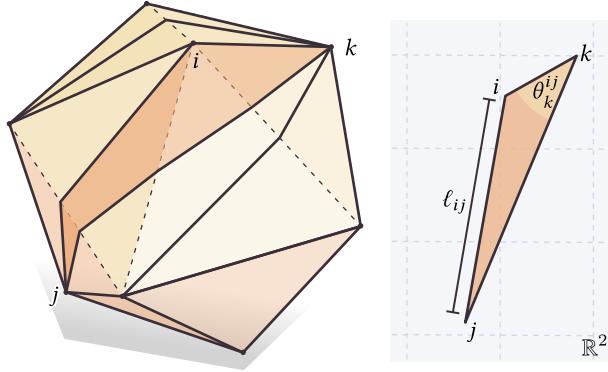


Fig. 2. Edges of an intrinsic triangulation are allowed to be geodesic paths along a surface (left). The faces of such a triangulation can be laid out in the plane as ordinary triangles (right).

into an intrinsic triangulation, execute an ordinary (“non-robust”) algorithm, and then read off the results in a variety of ways.

A remaining impediment to making the intrinsic approach truly reliable is to develop *data structures* for intrinsic triangulations that provide all the expected operations from standard mesh processing, while simultaneously providing strong guarantees of correctness. The basic challenge is encoding the correspondence between the input mesh T^0 and an intrinsic triangulation T^1 sitting atop it, so that data on one triangulation can be transferred to the other. The first such data structure was the *overlay mesh* of Fisher et al. [2006]. The overlay uses a halfedge mesh decorated with special vertex and edge attributes to maintain the *common subdivision* of T^0 and T^1 , i.e., the polygon mesh obtained by “slicing up” the underlying surface along the edges of both T^0 and T^1 . This approach guarantees correct connectivity, but ordinarily-local operations such as edge flips become non-local and expensive to evaluate—moreover, edge flips are the only operation supported by this data structure. Sharp et al. [2019a] instead encode the correspondence implicitly by storing so-called *signposts* at vertices, which give the direction and length of each intrinsic edge. This approach is somewhat complementary to the overlay mesh: local mesh operations are now cheap to evaluate, but the encoding of connectivity now depends on floating-point values, and is hence not guaranteed to be correct. For instance, when tracing out intrinsic edges small floating point errors can cause one to “miss” the target vertex. A key development here, however, was extending intrinsic triangulations to operations beyond edge flips.

Integer-Based Encoding. In this paper, we introduce an integer-based data structure for intrinsic triangulations that offers the best of both worlds: an implicit encoding of correspondence that supports fast local operations, but which is also guaranteed to correctly describe connectivity. Like the signposts, our data structure also supports a wide variety of local mesh operations (Section 3). As demonstrated in Section 5, we get dramatically improved robustness for difficult tasks, e.g., we achieve a 100% success rate for extracting a high-quality Delaunay refinement of low-quality input data. In turn, any algorithm that relies on a high-quality triangulation (e.g., for

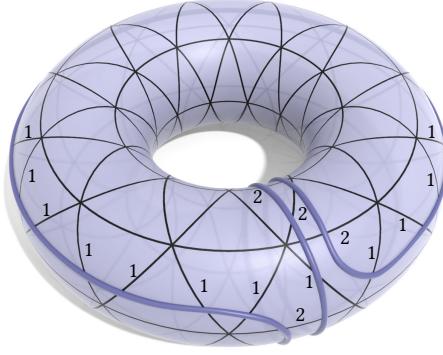


Fig. 3. We build on the idea of *normal coordinates* count how many times a curve crosses each edge of a triangulation.

solving PDEs) can immediately benefit from this improved robustness. For instance, in Section 4 we observe improved robustness for computing geodesic distance, local parameterization, constructing geodesics, and finding smooth vector fields.

The basic starting point for our data structure is the concept of *normal coordinates* from geometric topology (not to be confused with *geodesic normal coordinates* from Riemannian geometry). However, we must augment this construction in several ways in order to make it suitable for geometry processing. As detailed in Section 2.3, the basic idea of normal coordinates is to simply count how many times each edge of a triangulation is crossed by some curve (Figure 3). Such coordinates were originally developed to study not curves, but rather embeddings of surfaces in 3-manifolds [Kneser 1929; Haken 1961; Hass and Trnkova 2020], and subsequently appear in several places in mathematics (e.g., for studying the *mapping class group* [Farb and Margalit 2011]), including significant work on algorithms [Bell 2015, 2018; Schaefer et al. 2008]. In theoretical computer science, normal coordinates are also viewed as a means of “compressing” curves, e.g., the total number of bits required to store a long winding curve can be exponentially smaller than storing explicit segments along the curve [Erickson and Nayyeri 2013].

One challenge with using normal coordinates for geometry processing is that existing literature rarely considers operations beyond edge flips: the little that does considers only closed loops e.g. [Schaefer et al. 2002, Section 5.4], whereas curves that terminate at vertices are absolutely essential for encoding triangulations. A second issue is that normal coordinates alone are not enough to uniquely identify curves implied by the coordinates with logical edges of a mesh. Very recently, Gillespie et al. [2021, Section 5.2] proposed a solution to this issue using what they call *roundabouts*, but again do not consider operations beyond edge flips. Third, whereas most literature assumes that normal coordinates encode homotopy classes of curves in a purely topological setting (or perhaps hyperbolic geodesics), we must make a significant departure this perspective and assume that the normal coordinates encode a triangulation of a Euclidean polyhedron by geodesic edges. This distinction is important since, in general, not all normal coordinates describe a valid Euclidean geodesic triangulation. Considering this special case in turn enables us to establish procedures not previously seen.

Contributions. Overall, we make the following contributions:

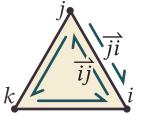
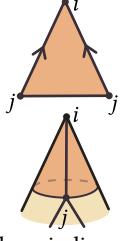
- We describe normal coordinates as a representation for general intrinsic triangulations, including the case where vertices have been added to the triangulation.
- We extend integer-based data structures for geometric intrinsic triangulations to include local operations beyond edge flips.
- We prove the correctness and quality of a Delaunay refinement algorithm for intrinsic triangulations of surfaces without boundary.
- We also extend intrinsic Delaunay refinement to surfaces with boundary.
- We introduce a new, more accurate way of transferring functions between bases on different triangulations.

We also experimentally validate the robustness of our technique, including generating intrinsic Delaunay refinements for all manifold meshes in the Thingi10k dataset, and demonstrate robustness for a variety of basic algorithms from geometry processing.

2 NOTATION AND CONVENTIONS

2.1 Connectivity

Throughout we assume that our domain is an oriented manifold surface M , possibly with boundary. We write $T = (V, E, F)$ to denote a triangulation of M with vertices V , edges E , and faces F . In general we allow triangulations that are not be simplicial, but can instead be a Δ -complex in the sense of Hatcher [2002, Section 2.1]—we allow, e.g., two edges of the same triangle to be glued together (see inset). We will refer to vertices $i \in V$, edges $ij \in E$, and faces $ijk \in F$ by one, two, or three indices, resp.. Note that as our triangulations need not be simplicial, the vertices i, j of an edge may not be distinct, and do not necessarily identify the edge—there may be multiple edges between i and j . We will refer to oriented halfedges $\vec{ij} \in H$, and will use u_i^{jk} to denote a value u at corner i of triangle ijk , and $u_{\vec{ij}}$ to denote a value u at halfedge \vec{ij} . Throughout, we consider a fixed input triangulation T^0 of M , as well as a dynamic *intrinsic triangulation* T^1 sitting atop M . T^1 must contain all vertices of T^0 , i.e. $V^1 \supseteq V^0$, but may include additional inserted vertices which we denote by $V^* := V^1 \setminus V^0$. We will generally use indices a, b, c for vertices of T^0 and indices i, j, k for vertices of T^1 (which may also be in T^0).



2.2 Geometry

The geometry of a triangulation is determined by a collection of edge lengths $\ell : E \rightarrow \mathbb{R}_{>0}$ satisfying the triangle inequalities in each face. For instance, we typically begin by assigning E^0 and E^1 the same edge lengths $\ell_{ij} = |f_j - f_i|$ determined by vertex coordinates $f : V \rightarrow \mathbb{R}^3$. While the lengths ℓ^0 remain fixed, the edge lengths ℓ^1 of T^1 may change due to operations like intrinsic edge flips (Section 3.3). Even after changing the edge lengths, each individual triangle $ijk \in F^1$ can always be drawn as an ordinary triangle in the Euclidean plane, allowing us to compute quantities such as face areas or corner angles. In general, we will not need to simultaneously embed all triangles of T^1 in \mathbb{R}^3 . Finally, we use $\exp_x : T_x M \rightarrow M$

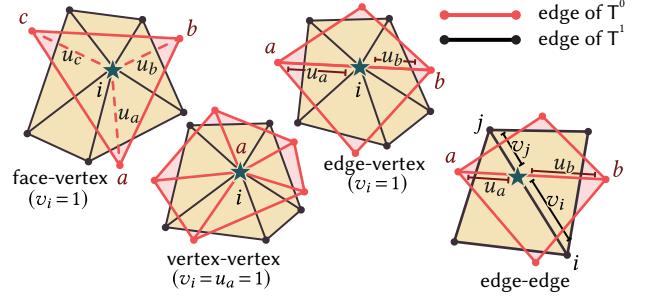
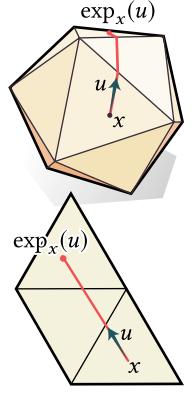


Fig. 4. Points are encoded relative to both triangulations T^0 and T^1 . For each triangulation we store the simplex containing the point, and the barycentric coordinates within that simplex. Here we show a few examples.

to denote the exponential map at x . Given a tangent vector u at x , $\exp_x(u)$ is the point reached by walking straight along the surface in the direction of u for a distance $|u|$ (see inset). (In practice this can be implemented as in [Sharp et al. 2019a, Section 3.2.2].)

Points $x \in M$ can be expressed in *barycentric coordinates* relative to some simplex (vertex, edge, or triangle) of a triangulation T , e.g., a point in a triangle ijk is given by three coordinates $u_i, u_j, u_k \in [0, 1]$ such that $u_i + u_j + u_k = 1$. For vertices, we set the single coordinate u_i to 1. Importantly, we will need to encode points x with respect to *two different* triangulations T^0 and T^1 , using barycentric coordinates u and v , resp. (Figure 4). Note that for vertices that are shared by both triangulations, we have $u_i = v_i = 1$. We will use q^0 to denote a location on T^0 represented by simplex along with a barycentric coordinate, and similarly will use q^1 for a location on T^1 .



2.3 Integer Coordinates

Normal coordinates. We use normal coordinates to count the number times each edge of T^1 crosses the edges of T^0 (see Figure 5, left). In principle, normal coordinates could either be defined as a value per edge of T^0 , counting the number of crossings from T^1 , or as a value per edge of T^1 . In our setting, we take the latter approach, as it remains fully-informative even when we insert new vertices in to T^1 . It is then natural to think of T^1 as the primary triangulation, with T^0 as a collection of geodesic curves sitting along it (Figure 5, right). To emphasize this abstract viewpoint, we will refer to the edges of E^0 as *curves*. Because we allow edges to be split, a single edge $ab \in E^0$ may actually correspond to a sequence of curves expressed in normal coordinates which meet at intermediate vertices $i \in V^*$.

Precisely, we use $n : E^1 \rightarrow \mathbb{Z}$ to denote the normal coordinates. For each edge $ij \in E^1$, the quantity n_{ij} indicates how many times ij is crossed by edges of E^0 : if $n_{ij} > 0$, then this is a count of crossings (formally, transversal intersections), whereas if $n_{ij} = -1$, it indicates that a curve runs along edge ij (Figure 5, left). Note that n_{ij} is never less than -1 because we are working with triangulations, and multiple edges of a triangulation cannot lie along the exact same

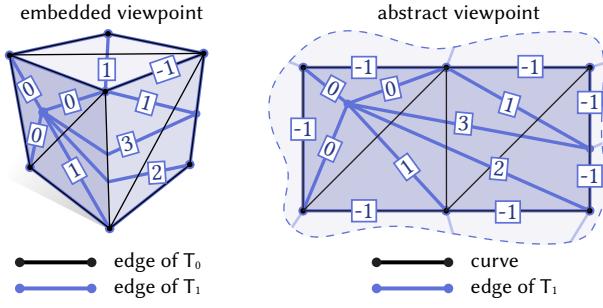


Fig. 5. The intrinsic triangulation T^1 is often presented embedded in \mathbb{R}^3 on top of the input triangulation T^0 (left). However, in our setting it is helpful to think of T^1 as an abstract intrinsic triangulation, given only by connectivity and edge lengths, which carries a collection of geodesic curves (right).

path. We use n_{ij}^+ to denote the number of transversal crossings, i.e. $n_{ij}^+ := \max(n_{ij}, 0)$, and similarly define $n_{ij}^- := -\min(n_{ij}, 0)$, which is 1 on shared edges and 0 otherwise.

Additionally, we define two quantities at each corner of T^1 , which count how many curves emanate from the interior of the corner and cross the corner resp. (Figure 6):

$$e_k^{ij} := \max(0, n_{ij}^+ - n_{jk}^+ - n_{ki}^+), \quad (1)$$

$$c_k^{ij} := \frac{1}{2} \left(\max(0, n_{jk}^+ + n_{ki}^+ - n_{ij}^+) - e_i^{jk} - e_j^{ki} \right). \quad (2)$$

Crossings. A point where a curve crosses an edge $ij \in E^1$ can be described either in a combinatorial or geometric sense. A *combinatorial crossing* is given by a pair $\zeta = (\overrightarrow{ij}, p)$, such that ζ is the p^{th} crossing along oriented edge $\overrightarrow{ij} \in H^1$. A *geometric crossing* is similarly given by $z = (\overrightarrow{ij}, p, u, v)$, where u, v encode the location of the crossing along the curve and along the \overrightarrow{ij} (resp.) in barycentric coordinates. Importantly, both of these crossings are *oriented*: the choice of halfedge \overrightarrow{ij} versus \overrightarrow{ji} indicates which side of the edge one is “coming from” and “going to,” which will be important when tracing out curves along the surface. We let $\bar{\zeta} := (\overrightarrow{ji}, n_{ij} - p - 1)$ denote a reversal of orientation.

Roundabouts. Normal coordinates alone do not fully encode the correspondence between T^0 and T^1 , because we cannot necessarily determine which curve along T^1 corresponds to which edge of T^0 (recall that there may be multiple edges of T^0 between the same pair

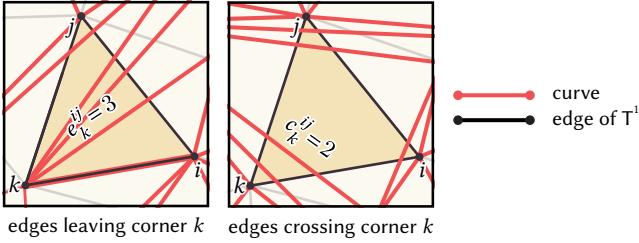


Fig. 6. It is often useful to count the curves emanating from the interior of each corner (left), and the curves crossing each corner (right).

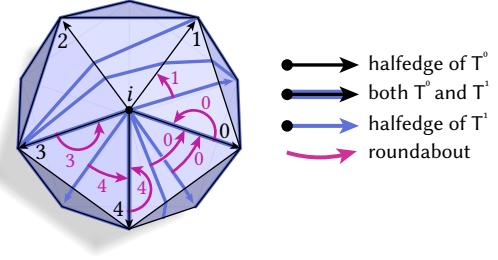


Fig. 7. We employ *roundabouts* to encode how edges of T^0 and T^1 are interleaved around vertices.

of vertices). Thus we additionally store *roundabouts* $r : H^1 \rightarrow \mathbb{Z}_{\geq 0}$, introduced by Gillespie et al. [2021, Section 5.2], which describe how the edges of T^0 and T^1 are interleaved around vertices. Unlike Gillespie et al., we may insert new vertices—however, we still store roundabouts only at halfedges pointing away from shared vertices $a \in V^0$. These are sufficient to disambiguate the identity of any traced curve, since all edges of E^0 must start and end at vertices in V^0 (if an edge has been split, then the *sequence* of curves starts and ends at vertices in V^0).

Precisely, for each halfedge $\overrightarrow{aj} \in H^1$ starting at a shared vertex $a \in V^0$, the roundabout stores the first halfedge $\overrightarrow{ab} \in H^0$ following \overrightarrow{aj} . This is encoded as an index $r_{\overrightarrow{aj}} \in \mathbb{Z}_{\geq 0}$, where we enumerate the halfedges of T^0 about vertex a in counterclockwise order (Figure 7)

3 ALGORITHMS AND DATA STRUCTURES

In this section, we provide descriptions of our data structure and the operations that it supports. Detailed pseudocode can be found in Appendix A.

3.1 Data Structure

The most essential data to maintain is a mesh of triangulation $T^1 = (V^1, E^1, F^1)$; recall that $V^1 \supseteq V^0$. We use a halfedge mesh, since halfedge meshes can represent general Δ -complexes (Section 2.1), though one could also use a vertex-face adjacency list plus a small amount of additional data [Sharp and Crane 2020a, Section 4.1]. On top of this mesh, our data structure maintains four quantities:

- lengths $\ell_{ij} \in \mathbb{R}_{>0}$ for each edge $ij \in E^1$,
- normal coordinates $n_{ij} \in \mathbb{Z}$ for each edge $ij \in E^1$,
- roundabouts $r_{\overrightarrow{aj}} \in \mathbb{Z}_{\geq 0}$ for each halfedge $\overrightarrow{aj} \in H^1$ incident on a vertex a shared with T^0 ,
- barycentric coordinates q_i^0 relative to T^0 for each $i \in V^1$.

This differs from the scheme of Gillespie et al. [2021, Section 5] in a few key ways. The essential difference is that their data structure assumes that T^0 and T^1 share the same vertex set ($V^1 = V^0$). This has numerous consequences—e.g. they use nonnegative normal coordinates $n_{ij} \in \mathbb{Z}_{\geq 0}$, they do not store input positions q_i^0 . Most importantly, it is impossible to perform many of the local mesh operations we describe in the next section without changing the vertex set of T^1 ; our generalization of the representation is essential if one wants to perform tasks such as Delaunay refinement (Section 4.2).

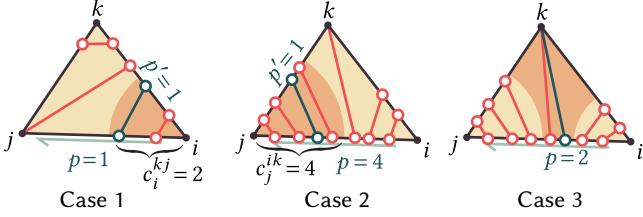


Fig. 8. A curve entering triangle jik along edge ij can proceed in 3 ways: it can exit along edge ik , in which case it is counted by c_i^{kj} (left); it can exit along edge kj , in which case it is counted by c_j^{ik} (center); or it can terminate at vertex k (right). This forms the core of procedure `TRACEFROM`.

3.2 Extracting Curves

Our first task is to recover a curve on T^1 from its normal coordinates—because our curves are geodesic, we can determine the exact geometry from these normal coordinates. In particular, we describe a procedure `EXTRACTCURVE` (Algorithm 2) which takes in any combinatorial crossing ζ along a curve, and computes the curve’s trajectory along T^1 as a sequence (i, z_1, \dots, z_k, j) of geometric crossings along with start and end vertices $i, j \in V^1$. We note that this mirrors the discussion in Gillespie et al. [2021, Section 6], albeit in a more general setting; we include a full description here for completeness.

We proceed in two steps, first determining the triangle strip that the curve passes through and only then computing the curve’s geometry. Note that the triangle strip depends solely on the integer-valued normal coordinates n , while geometric data and floating point computation are relegated to the second step.

The first step is performed by `TRACEFROM` (Algorithm 1), which takes some combinatorial crossing $\zeta = (\vec{ij}, p)$ along a curve γ , and traces out the remaining combinatorial crossings until γ terminates at a vertex. `TRACEFROM` proceeds iteratively, taking the crossing where γ enters a triangle, and using the triangle’s normal coordinates to determine where γ exits (see Figure 8). The direction in which to trace the curve is determined by the orientation of \vec{ij} .

To determine the triangle strip containing γ , `EXTRACTCURVE` calls `TRACEFROM` once in either direction, yielding the sequence of all combinatorial crossings along γ . `EXTRACTCURVE` then unfolds this triangle strip in an arbitrary planar coordinate system and draws γ as a straight line between its endpoints. The intersection of this line with each of the intermediate edges determines the geometric crossings along γ (Figure 9). Note that unlike Gillespie et al. [2021, Algorithm 1], we may have to invoke `EXTRACTCURVE` multiple times to extract a single edge $ab \in E^0$, as it may pass through several vertices of T^1 , e.g. due to *edge splits* (Section 3.5).

Finally, it is sometimes useful to convert a single combinatorial crossing ζ into a geometric crossing z . We will refer to this operation as `EXTRACTGEOMETRICCROSSING`; it may be implemented by calling `EXTRACTCURVE` and then returning the single desired crossing.

3.3 Edge Flip

Edge flips are a well-studied operation, but we include a discussion here for completeness. We may flip an edge if and only if (i) both endpoints have degree at least one after the flip, and (ii) the two triangles containing the edge form a convex quadrilateral (Figure 10).

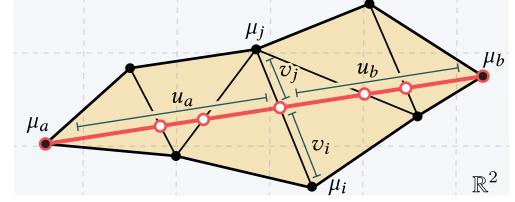


Fig. 9. We compute barycentric coordinates by laying out a triangle strip in the plane.

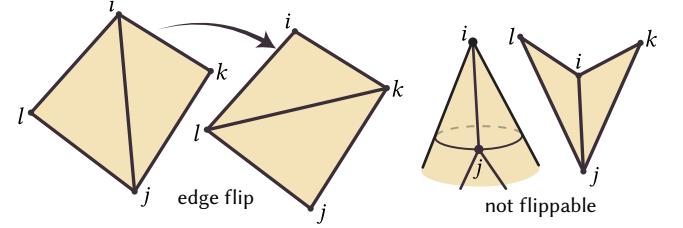


Fig. 10. An edge flip replaces an edge with its opposite diagonal (left). An edge ij is not flippable if it would leave a vertex with degree zero, or if its neighboring faces form a nonconvex quadrilateral (right).

Mesh Update. We replace edge ij with an edge kl . We compute the new edge length ℓ_{lk}^1 by laying out the two old triangles ijk, lji in the plane and measuring the length of the appropriate diagonal.

Normal Coordinates & Roundabouts. The new normal coordinate n_{kl} does not depend at all on the geometry of T^1 . It is given by the following formula:

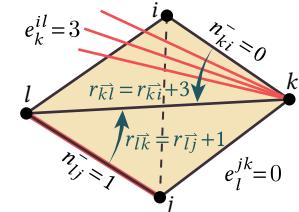
$$n_{kl} = c_l^{jk} + c_k^{ij} + \frac{1}{2} |c_j^{il} - c_j^{ki}| + \frac{1}{2} |c_i^{lj} - c_i^{jk}| - \frac{1}{2} e_l^{ji} - \frac{1}{2} e_k^{ij} + e_i^{lj} + e_i^{jk} + e_j^{il} + e_j^{ki} + n_{ij}^- \quad (3)$$

This differs slightly from the formula of Gillespie et al. [2021], which did not allow for inserted vertices.

We can update each roundabout from its previous neighbor:

$$r_{\vec{k}\vec{l}} = \text{mod} \left(r_{\vec{k}\vec{i}} + e_k^{il} + n_{ki}^-, \deg_0(k) \right), \quad (4)$$

where $\deg_0(k)$ is the degree of vertex k in triangulation T^0 . The quantity $e_k^{il} + n_{ki}^-$ counts how many edges of T^0 are between $\vec{k}\vec{i}$ and $\vec{k}\vec{l}$: e_k^{il} counts edges strictly between them, and n_{ki}^- adds one if there is also an edge lying exactly along $\vec{k}\vec{l}$.



We only perform this update for halfedges whose source is in V^0 .

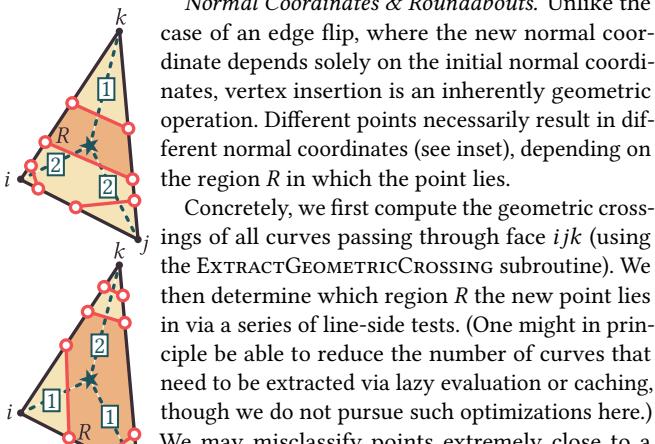
3.4 Face Split

We now describe procedure **SPLITFACE** (Algorithms 4 and 5), the first of several new routines to mutate triangulation T^1 while tracking the correspondence with T^0 . Note that Schaefer et al. [2002, Section 5.4] describe a similar face split operation in the topological setting, but do not provide the ability to insert a point at a particular geometric location, which is essential in our setting of Euclidean polyhedra.

In particular, suppose we wish to insert a vertex at a point $x \in M$, given by barycentric coordinates v on a triangle $ijk \in F^1$. To do so, we need to update the connectivity T^1 , edge lengths t^1 , normal coordinates n , and roundabouts r . Additionally, we need to compute the position q^0 of this new vertex in barycentric coordinates on T^0 .

Mesh Update. We update the connectivity of T^1 with a new vertex and three new edges and faces. We compute new edge lengths as a formula of the barycentric coordinates u . Schindler and Chen [2012, Section 3.2] show that the length of a displacement vector δu_i in barycentric coordinates is give by

$$\|\delta u_i\|^2 = -t_{ij}^2 \delta u_i \delta u_j - t_{jk}^2 \delta u_j \delta u_k - t_{ki}^2 \delta u_k \delta u_i. \quad (5)$$



Normal Coordinates & Roundabouts. Unlike the case of an edge flip, where the new normal coordinate depends solely on the initial normal coordinates, vertex insertion is an inherently geometric operation. Different points necessarily result in different normal coordinates (see inset), depending on the region R in which the point lies.

Concretely, we first compute the geometric crossings of all curves passing through face ijk (using the **EXTRACTGEOMETRICCROSSING** subroutine). We then determine which region R the new point lies in via a series of line-side tests. (One might in principle be able to reduce the number of curves that need to be extracted via lazy evaluation or caching, though we do not pursue such optimizations here.) We may misclassify points extremely close to a region's boundary due to floating point error, in which case we insert a valid point in the identified region, at a virtually identical location. Note that this behavior is perfectly reasonable in, e.g., retriangulation algorithms (see Section 4.2), where the insertion location is not computed exactly anyway.

The roundabouts on any new halfedges emanating from original vertices (*i.e.*, vertices in $\{i, j, k\} \cap V^0$) can be set from their neighbors via Equation 4.

Position on T^0 . To determine q^0 , we must locate the triangle $abc \in F^0$ containing x , as well as the barycentric coordinates of x within abc . We do so via interpolation from the corners of R . Explicitly, the corners of R are all geometric crossings with known barycentric coordinates in some triangle $abc \in F^0$ (computed in **EXTRACTGEOMETRICCROSSING**); we can then solve a small linear system to recover the barycentric coordinates of p in the same triangle. Intuitively, we recover generalized barycentric coordinates for p with respect to the polygon R and apply them on abc to recover standard barycentric coordinates in T^0 (see Appendix B for details).

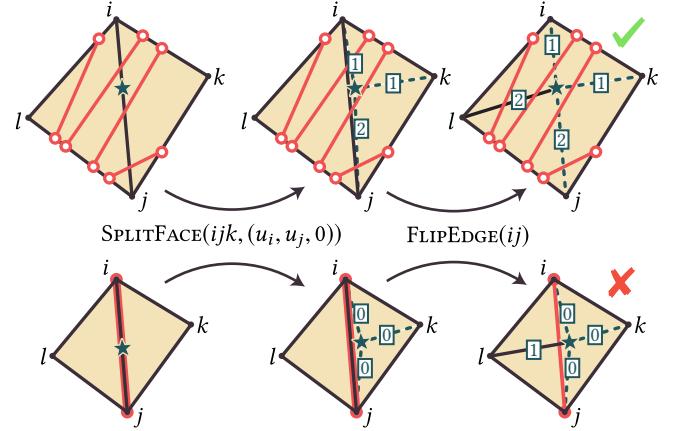


Fig. 11. Generally, one can split edge ij by performing a face split on a neighboring face followed by an edge flip (*top*). However, if ij carries a curve, this strategy will cause the inserted vertex to miss the curve (*bottom*). We hence provide a different edge split procedure for this case in Section 3.5.

3.5 Edge Split

We also introduce an operation **SPLITEDGE** (Algorithm 6), which takes as input a point given by barycentric coordinates v along an oriented edge $ij \in H^1$. If ij does not have a curve running along it (*i.e.* $n_{ij} \geq 0$), then this is implemented as a face split followed by an edge flip (Figure 11, *top*). However, if $n_{ij} < 0$ (which is common in practice—*e.g.* Section 4.2), we perform an explicit edge split which inserts the new vertex along the coincident curve (Figure 11, *bottom*).

Mesh Update. We insert a new vertex and triangulate any adjacent faces, computing the new edge lengths via Equation 5.

Normal Coordinates & Roundabouts. When $n_{ij} < 0$ the new normal coordinates are simple functions of the old ones, since every curve in face ijk must emanate from i or j , or cross k . The number of such curves is $\max(n_{ki}, n_{jk}, 0)$ and edge pk crosses them all. We hence set n_{pj} and n_{pi} equal to n_{ij} , and set

$$n_{pk} = \max(n_{ki}, n_{jk}, 0). \quad (6)$$

As with face splits, roundabouts on any new halfedges emanating from original vertices can be set from their neighbors (Equation 4).

Position on T^0 . To determine q^0 , we must locate the edge $ab \in E^0$ containing x , as well as the barycentric coordinates of x within ab . Since, i and j necessarily have known locations along some edge in $ab \in E^0$, we can simply interpolate by v to compute the location q^0 .

3.6 Vertex Removal

In general, a vertex which is present in the original triangulation cannot be removed without distorting the intrinsic metric because any curvature at that vertex would be lost. However, inserted vertices $i \in V^*$ have no curvature, and can hence be removed safely. In fact this operation will be necessary for Delaunay refinement of domains with boundary (Section 4.2).

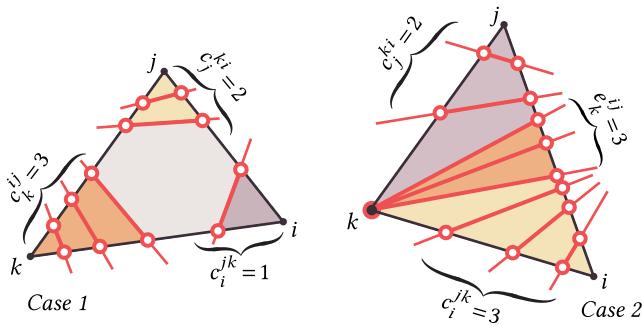


Fig. 12. We extract the connectivity of common subdivision within each triangle using its normal coordinates.

The basic strategy behind REMOVEVERTEX (Algorithm 7) is to flip edges incident on the vertex to be removed until it has degree three, then delete the three edges incident on the vertex as well as the vertex itself. No other data needs to be updated, since the edges of the resulting triangle already appear in the triangulation. Algorithm 7 describes this procedure, and Theorem D.1 proves its correctness for simplicial complexes. A nearly identical procedure can be used to remove an inserted boundary vertex. Schaefer et al. [2002, Section 5.4] also suggest a similar flipping procedure, but work in the topological setting where the necessary edge flips are always valid—they do not consider the convexity condition (Section 3.3).

3.7 Moving Inserted Vertices

Given the previous operations, we can easily define a procedure for moving around inserted vertices. Specifically, given a vector v in the tangent space of an inserted vertex i , we can move i along v in the following way:

- First, compute the new location $p = \exp_i(v)$.
- Insert p using SPLITFACE.
- Remove i using REMOVEVERTEX.

We insert p first since the removal procedure could flip edges incident on the triangle containing p , invalidating its barycentric coordinates. Note that Sharp *et al.* propose an alternative strategy for local vertex displacement [Sharp et al. 2019a, Section 3.3.3].

3.8 Common Subdivision

As noted previously, the *common subdivision* S of T^0 and T^1 is the polygon mesh obtained by “slicing up” the underlying surface along the edges of both T^0 and T^1 . The vertices of S are hence a superset of V^0 and V^1 , and every edge or face of T^0 and T^1 can be expressed as union of edges or faces of S (*resp.*). Moreover, the faces of S are always planar and convex. Most importantly in our setting, any piecewise-linear function on T^0 or T^1 can be represented exactly as a piecewise-linear function on S . Note however that even if T^0 and T^1 have nice elements, S is not in general a high-quality mesh, and may not itself be suitable for, *e.g.*, solving PDEs. Rather, it plays a complementary role in the geometry processing pipeline, enabling (for instance) transfer of data between triangulations (Section 4.4), or visualization of data downstream via standard rendering tools.

We compute the common subdivision by cutting T^1 along the edges of T^0 . First we extract the connectivity of S , using only the normal coordinates n_{ij} . Then we recover the intersection geometry, allowing us to interpolate data stored at the vertices of T^0 or T^1 to S —most commonly, vertex positions on T^0 along with any solution data on T^1 . Note that this procedure was previously described by Sharp et al. [2019a, Section 3.4.2]; we recap it here for completeness, and to give a convenient description using our integer coordinates.

Connectivity. We subdivide T^1 independently in each face ijk . The normal coordinates alone determine the connectivity of S within this face. There are just two cases to consider, illustrated in Figure 12. Case 1 occurs when no curves emanate from any corner, so we simply need to connect the first c_i^{jk} crossings along edge ij to the first c_j^{ki} crossings along ik (in order), and likewise for corners j and k . In Case 2 curves emanate from some corner; without loss of generality, let this corner be k so that the number of such curves is $e_k^{ij} > 0$ and there are more curves crossing edge ij than the other two edges. Hence, we can walk from i to j , connecting the first c_i^{jk} crossings to those along ik , the next e_i^{jk} crossings to vertex k , and the remaining c_k^{ji} crossings to those along edge kj . Note that curves running along edges ($n_{ij} < 0$) require no special treatment.

Intersection Geometry. Next, we associate each vertex i of the common subdivision with a point in T^0 and a point in T^1 , encoded in barycentric coordinates relative to some simplex (vertex, edge, or face) of the appropriate triangulation. Using this, one can linearly interpolate data in the usual way. Again, there are just two cases: each vertex i in S is either a vertex of T^1 or the intersection of an edge of T^0 with an edge of T^1 . In the first case, the position on T^1 is given by i itself, and its position q_i^0 on T^0 was computed when i was inserted. In the second case, we compute the desired barycentric coordinates using EXTRACTCURVE (see Algorithm 8 for details).

3.9 Transposing Coordinates

Throughout, we store normal coordinates $n : E^1 \rightarrow \mathbb{Z}$ which count how many times edges of E^1 cross edges of E^0 . It is sometimes useful to observe that tracing an edge $ab \in E^0$ over T^1 counts how many times ab crosses edges of T^1 . If T^1 has more vertices than T^0 , then the edges of E^1 are not *normal* over T^0 —they can start and end in the middle of faces of T^0 —and these crossing counts do not uniquely encode the structure of T^1 . However, if T^0 and T^1 do have the same vertex set, then these crossing counts provide an implicit representation of T^1 as a collection of curves over T^0 .

3.10 Visualization

In the following, we show examples of intrinsic triangulations on top of meshes (*e.g.* Figure 13). To produce these figures, we compute the common subdivision (Section 3.8) and draw the edges of the input mesh with a black wireframe while coloring the intrinsic triangles in arbitrarily-chosen colors. In figures displaying functions defined on intrinsic triangulations (*e.g.* Figure 15), we interpolate the solutions along the common subdivision for rendering.

3.11 Robust Implementation

Our integer coordinates are guaranteed to encode a triangulation sitting atop T^1 . The geometric accuracy of this triangulation, of course, depends on floating point arithmetic, which can become inaccurate in near-degenerate configurations. *Exact predicates* have been applied with great success to similar problems [Devillers and Pion 2003]. Unfortunately they do not directly apply to intrinsic triangulations, as the predicates that we evaluate are not fixed functions of the input data; an intrinsic edge length can depend upon arbitrarily many input edge lengths. Hence, we focus on fast and robust implementations using ordinary floating point arithmetic.

One essential tool for dealing with intrinsic triangulations on near-degenerate input meshes is *intrinsic mollification*, introduced by Sharp and Crane [2020a]. Mollification improves degenerate meshes by adding a small ϵ to every edge length, provably improving triangle quality. This changes the geometry by a negligible amount, and moreover we only mollify if some triangle is within ϵ of being degenerate. This procedure works particularly well with our data structure compared to signposts: the signpost data structure relies on tracing queries along the surface which become *less* accurate when mollification is applied. Our integer coordinates have no such problem: we always get the correct edge sequence, even if the mesh geometry is slightly modified. In our experiments we mollify with $\epsilon = 10^{-5}$, and find that it resolves almost all numerical difficulties.

Even after mollification, it is still beneficial to use care when working with floating point. For example, there are well-conditioned triangles on which the Delaunay condition (Equation 8, discussed in the next section) is difficult to evaluate; in practice, we only enforce Equation 8 up to some ϵ tolerance. As a further example, when computing new normal coordinates in SPLITFACE, one could lay out the face in the plane, and independently count intersections along the new edges. However, this can produce invalid normal coordinates in floating point. We apply a more complicated policy (see Appendix A) which always yields valid normal coordinates. For additional details on all procedures, we refer the reader to our implementation, which will be made available after review.

3.12 Other Algorithms

Normal coordinates also enable a wide variety of other operations not detailed here. For instance, Schaefer et al. [2002, Section 5] provide algorithms for counting connected components, checking if crossings are part of the same curve, checking if curves are isotopic, and computing the oriented intersection number. Erickson and Nayyeri [2013] provide an asymptotically-fast algorithm for tracing normal curves across a surface. Finally, Dynnikov [2020, Proposition 13] provides an algorithm for computing how many times curves represented by normal coordinates intersect.

4 APPLICATIONS

4.1 Intrinsic Delaunay Triangulations

One key application of intrinsic triangulations is the computation of *intrinsic Delaunay triangulations* (Figure 13, top). A triangulation is said to be Delaunay if the sum of angles opposite

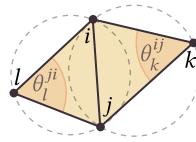


Fig. 13. Using our integer-based data structure, we can not only improve near-degenerate meshes by generating *intrinsic Delaunay triangulations* (top), but can also extract the common subdivision after computing a high-quality *intrinsic Delaunay refinement* (bottom).

every edge is at most π , i.e. for every $ij \in E$ we have

$$\theta_k^{ij} + \theta_l^{ji} \leq \pi. \quad (7)$$

Delaunay triangulations have a number of beneficial properties. One consequence of Equation 7 is that edges of a Delaunay triangulation must have nonnegative cotan weights:

$$\cot \theta_k^{ij} + \cot \theta_l^{ji} \geq 0. \quad (8)$$

In fact, Equation 8 is equivalent to Equation 7 above, and provides a convenient formula for checking the Delaunay property in an intrinsic triangulation. Moreover, Equation 8 ensures that the finite element Laplacian L satisfies the *maximum principle*, guaranteeing that discrete harmonic functions do not have local extrema in the interior of the domain [Bobenko and Springborn 2007, Proposition 19]. Similarly, Equation 8 also ensures that discrete harmonic vector fields are “flip-free” [Sharp et al. 2019b, Section 5.4]. Furthermore, the local Delaunay condition implies the *empty circumcircle property*: each triangle’s geodesic circumdisk contains no vertices, illustrated in Figure 14, left [Bobenko and Springborn 2007, Proposition 10].

The Delaunay triangulation can be computed via a simple greedy algorithm: flip any non-Delaunay edge until all edges satisfy Equation 7 [Bobenko and Springborn 2007, Propositions 11 and 12].

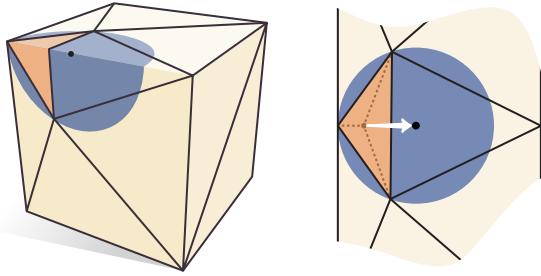


Fig. 14. Triangles in Delaunay meshes have empty circumdisks, and thus well-defined circumcenters (left). When necessary, we locate a triangle’s circumcenter by walking outwards from its barycenter (right).

4.2 Intrinsic Delaunay Refinement

Delaunay refinement inserts vertices in order to produce a Delaunay mesh whose triangles all satisfy a minimum angle bound (Figure 13, bottom). Here we modify *Chew’s second algorithm* to perform intrinsic (i.e. geodesic) scheme was only recently proposed by Sharp et al. [2019a, Section 4.2]. However, they did not handle meshes with boundary—here we resolve the essential difficulties of the boundary case, and show how refinement can be implemented using our integer-based data structure.

In the plane, the basic algorithm is to greedily pick any triangle which violates the minimum angle bound, insert a vertex at its circumcenter, then flip to Delaunay. This process continues until all triangles satisfy the angle bound. If a triangle’s circumcenter is outside the domain, then the boundary edge ij separating the triangle from its circumcenter is split at its midpoint; subsequently, all interior vertices within at least a distance of $\ell_{ij}/2$ are removed—though removing additional interior vertices causes no issues (Appendix C.1). One can prove that this process succeeds for minimum angle bounds up to 25.65 degrees on planar domains with boundary angles at least 60° [Shewchuk 1997, Section 3.4.2]. More advanced versions of this procedure can achieve better angle bounds, e.g. [Rand 2011], but here we restrict our attention to the basic algorithm for simplicity.

There are two difficulties in adapting this algorithm to the intrinsic setting: locating circumcenters and computing (geodesic) distances. As mentioned earlier, intrinsic Delaunay triangulations obey the empty circumcircle property; hence each triangle has an intrinsically-flat circumdisk with a well-defined center (Figure 14, left). So long as this center corresponds to a point on the surface, it can be found by walking from the triangle’s barycenter (Figure 14, right). In practice, we compute triangle ijk ’s circumcenter in homogeneous (i.e., unnormalized) barycentric coordinates \hat{v}_i via the following formula [Schindler and Chen 2012, Section 2.3]:

$$\hat{v}_i := \ell_{jk}^2 (\ell_{ij}^2 + \ell_{ki}^2 - \ell_{jk}^2), \quad (9)$$

and then normalize to obtain barycentric coordinates

$$v_i := \frac{\hat{v}_i}{\hat{v}_i + \hat{v}_j + \hat{v}_k}. \quad (10)$$

To locate the circumcenter on the surface, we then evaluate the exponential map (Section 2.2) starting at the barycenter $w_i = w_j =$

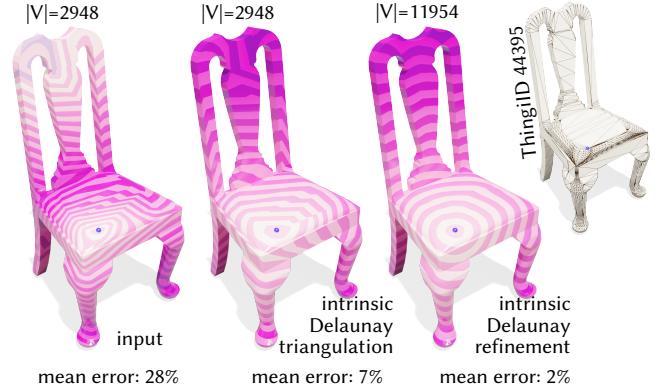
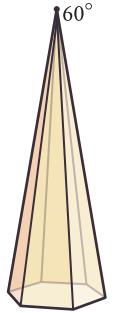


Fig. 15. Running PDE-based algorithms such as the heat method on poor triangulations (left) can lead to inaccurate solutions. Flipping to intrinsic Delaunay (center) and performing Delaunay refinement (right) can drastically improve the results.

$w_k = 1/3$, along the vector $v - w$. If we hit a boundary edge ij while tracing out this path, then the circumcenter is not contained in the surface, so we split ij at its midpoint and flip to Delaunay. We must then remove all inserted interior vertices within a geodesic ball of radius $\ell_{ij}/2$ centered at the inserted point. Computing geodesic distance on a surface mesh is nontrivial, but Xia [2013, Corollary 1] shows that on a Delaunay triangulation any vertex inside a geodesic ball of radius r will also be inside the Dijkstra ball of radius $2r$ (i.e. points whose distance along the edge graph are at most $2r$). We hence remove all interior inserted vertices within a Dijkstra distance of ℓ_{ij} . Note that while Xia considers only the planar setting, their proof (which is based on triangle strips) applies without modification to intrinsic Delaunay triangulations of surfaces.

On meshes with narrow cone vertices or boundary angles, it may be impossible to find any triangulation satisfying a given angle bound. In such cases, we do not insert circumcenters of intrinsic triangles which are incident on exactly one narrow vertex, or are entirely contained in a triangle of T^0 which is incident a narrow vertex, and ignore such triangles when computing the minimum corner angle of the output mesh. Although the final output may violate the angle bound, such triangles appear only near narrow vertices. In analogy with the planar case, we set 60° as the minimum allowed angle sum (see inset); in practice the vast majority of meshes obey this constraint at all vertices (97.2% of Thingi10k), and even on those which do not we obtain high-quality triangulations.



4.3 PDE-Based Geometry Processing

PDE-based methods abound in geometry processing, as they are generally simple to implement and benefit from decades of research into linear solvers. Many such methods depend only on intrinsic data, and are hence a natural application of our intrinsic Delaunay triangulations and refinements. On near-degenerate inputs, simply running the standard algorithm on an intrinsic triangulation instead

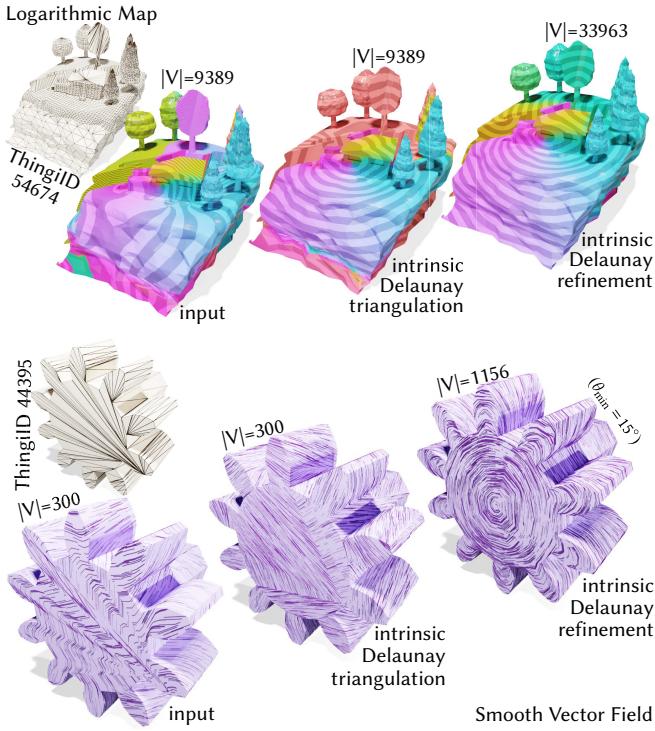


Fig. 16. Here we compute a local parameterization (the *logarithmic map*, top), and a smooth vector field (bottom) using the connection Laplacian. Both procedures yield inaccurate results on near-degenerate inputs (left)—intrinsic Delaunay triangulations (center) and intrinsic Delaunay refinements (right) greatly improve solution quality. Whether our solution is a scalar function or vector field, we can visualize it on the common refinement.

of the original mesh yields solutions of dramatically higher quality (Figures 15 and 16).

We show several examples: fast geodesic distance computation [Crane et al. 2017], local parameterization via the logarithmic map [Sharp et al. 2019b], and smooth vector fields [Knöppel et al. 2013]. Further examples on tasks such as parameterization, minimal surfaces, and surface editing can be found in [Sharp et al. 2019a; Sharp and Crane 2020a]. Across the board, normal coordinates offer improved robustness guarantees, and open doors to higher solution accuracy with the common subdivision.

4.4 Attribute Transfer

Intrinsic triangulations can drastically improve the quality of solutions to PDEs on low-quality meshes. However in practice, one often needs to represent the solution on the input mesh. Past approaches have simply “copied back” the solution values at vertices of the original mesh, but this strategy is ad-hoc and suboptimal. A more principled approach is to choose the function on the original mesh which is closest to the intrinsic solution. Here, we restrict our treatment to piecewise-linear bases and L^2 distance for simplicity, though the same strategy could easily be applied to other basis functions and notions of distance. Precisely, given a function

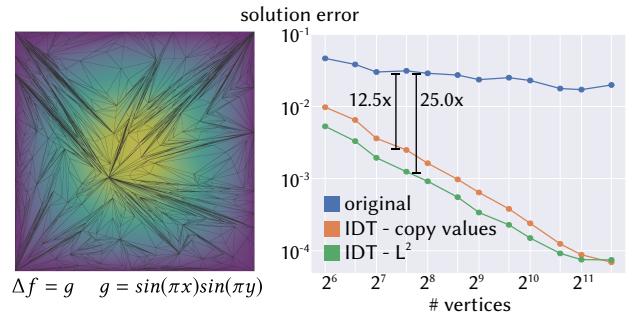


Fig. 17. Accuracy is improved by transferring PDE solutions back to an original triangulation as the L^2 -nearest solution, evaluated via the common subdivision. Here we generate random low-quality meshes of the unit square by random edge splits (left), and plot the error in the solution of a Poisson equation compared to analytic ground truth, always represented in the basis of the original triangulation (right). Each data point is the average error over 100 trials. As expected, solving on the intrinsic Delaunay triangulation dramatically increases accuracy, but further improvements are gained by choosing the solution on the original mesh which is L^2 -nearest to the intrinsic solution, rather than naively copying vertex values.

f on the intrinsic triangulation, we seek \hat{f} on the original mesh that minimizes the squared L^2 distance

$$\|f - \hat{f}\|_{L^2}^2 := \int_M |f(x) - \hat{f}(x)|^2 dx. \quad (11)$$

Here, f and \hat{f} are functions represented in finite-dimensional bases with nodal values at the vertices of the intrinsic triangulation and the original mesh *resp.* In traditional finite elements, this integral commonly arises over a *single* triangulation, in which case it can be evaluated via the Galerkin mass matrix M as

$$\|f - \hat{f}\|_{L^2}^2 = (f - \hat{f})^T M (f - \hat{f}), \quad (12)$$

where M is constructed as in [Strang and Fix 2008, Chapter 10, (32)]. However, in our setting f and \hat{f} are encoded over *different* triangulations; they are members of different function spaces. Our key observation is that the common subdivision S (Section 3.8) provides exactly the structure needed to evaluate $\|f - \hat{f}\|_{L^2}^2$, as both functions are linear on each triangle of S . In fact, we have

$$\|f - \hat{f}\|_{L^2}^2 = (P_1 f - P_0 \hat{f})^T M_S (P_1 f - P_0 \hat{f}), \quad (13)$$

where now M_S is the Galerkin mass matrix of the common subdivision, and P_0, P_1 are interpolation matrices which map piecewise-linear functions on original and intrinsic triangulations to piecewise-linear functions on S , *resp.* In particular, P_0 is a $|V^0| \times |V^S|$ matrix, where each row corresponds to a vertex of S , and has that vertex’s barycentric coordinates on T^0 as entries. P_1 is defined likewise for T^1 . We then find the function \hat{f} which minimizes Equation 13 as the solution to a linear least-squares system, which can be prefactored if desired to efficiently transfer many functions.

We can leverage this formulation to transfer functions from any intrinsic triangulation back to the original mesh. In Figure 17, we show how this transfer indeed improves the accuracy of PDE solutions as measured on the original low-quality mesh. This machinery

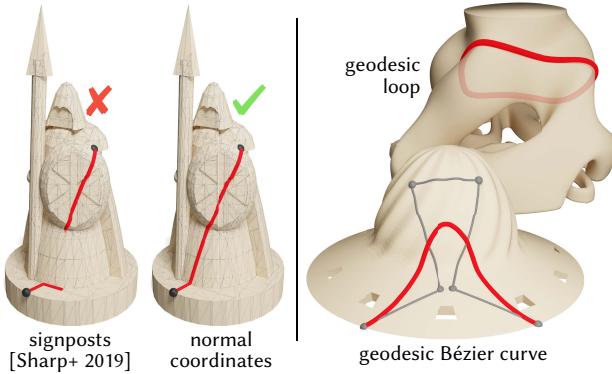


Fig. 18. We construct geodesic paths by flipping edges in a normal coordinate intrinsic triangulation, as in [Sharp and Crane 2020b]. Normal coordinates guarantee a valid path, even on degenerate inputs (*left*). This unlocks advanced applications of geodesics in normal coordinates with the same guarantees, such as geodesic loops and Bézier curves (*right*).

is enabled because our integer coordinates efficiently and robustly compute the common subdivision. More broadly, this paradigm opens the door to a wide variety of future finite-element formulations involving intrinsic triangulations.

4.5 Flip-Based Geodesic Paths

The previous sections have demonstrated the value of intrinsic triangulations as a high-quality basis for discretizing functions on surfaces; more broadly, these triangulations also provide simple and robust solutions to other tasks across geometry processing. As an example, the recent FLIPOUT procedure of Sharp and Crane [2020b] computes exact geodesic paths on surfaces via a simple intrinsic edge flipping strategy, introducing the geodesic as a path of edges in the triangulation. This method is easily implemented in our integer representation in terms of the mesh operations in Section 3, and the resulting geodesic paths may then be recovered with the EXTRACTEDGE subroutine. Computing geodesics with our robust integer coordinates is particularly appealing, because geodesic algorithms are notoriously difficult to implement robustly [Sharp and Crane 2020b, Section 5.3]. Even the method of Sharp *et al.* uses the signpost data structure, which may fail to reconstruct a connected path along the surface for degenerate inputs. In contrast, implementing FLIPOUT in our integer coordinate representation extends the benefits of our approach to this task, including a guarantee of valid connectivity in the output (Figure 18, *left*). It also enables higher-level tasks involving geodesic paths to be safely run on low-quality input, such as constructing geodesic loops on surfaces, and even geodesic Bézier curves, using a de Casteljau-style scheme due to Morera *et al.* [2008] as shown in Figure 18, *right*.

5 EVALUATION

We implemented all algorithms in C++; since basic vertex-face adjacency list cannot represent a general Δ -complex (Section 2.1), we use a halfedge data structure for triangle meshes. Timings are measured on a single core of an Intel i9-9980XE with 32 GB of RAM.

Method	Intrinsic Delaunay Triangulation	Intrinsic Delaunay Refinement
Explicit Overlay	100 %	-
Signpost Tracing	96.0 %	69.1 %
Integer Coordinates	100 %	100 %

Table 1. The success rate of our method and past approaches for building high-quality intrinsic triangulations in the Thingi10k dataset. For each we construct a Delaunay triangulation, either on the original vertex set or with Delaunay refinement to a 25° minimum angle bound, and attempt to recover the connectivity of the common subdivision. The explicit overlay method does not support refinement.

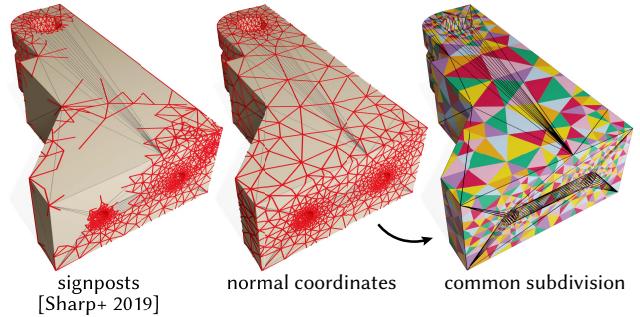


Fig. 19. Past methods extracted edges by tracing “signposts” along the mesh, which may fail in the presence of degenerate triangles. In contrast, our integer coordinates always yield a topologically-valid common subdivision, even on extremely poor quality inputs.

Performance. Generally our data structure is quite fast, computing Delaunay refinements for complex meshes in seconds. For example, computing the Delaunay refinement Figure 15 takes 0.2s, and the Delaunay refinement in Figure 16 (*top*) takes 0.6s. Because we lazily recover intersection geometry from our integer coordinates when inserting vertices, routines such as Delaunay refinement which perform many insertions may become moderately expensive on large near-degenerate inputs. For instance we take 4 minutes to perform Delaunay refinement on 719791 (Figure 20, *top*) which signposts does in 1.5 minutes, but on such meshes signposts generally fails to compute a valid common subdivision at the end. Section 6 discusses hybrid routines which may give the best of both worlds.

5.1 Robustness

We validate robustness by successfully computing Delaunay triangulations, refinements, and their common subdivisions on all manifold meshes in Thingi10k [Zhou and Jacobson 2016]. In particular, we used *MeshLab* to convert each mesh to the PLY file format [Cignoni *et al.* 2008], resulting in 7696 valid manifold meshes. We begin by mollifying each mesh to a tolerance of 10^{-5} (Section 3.11). For each model we compute the intrinsic Delaunay triangulation (Section 4.1) with a tolerance of 10^{-5} , as well as an intrinsic Delaunay refinement (Section 4.2) with a 25° angle bound. We verify that the algorithms

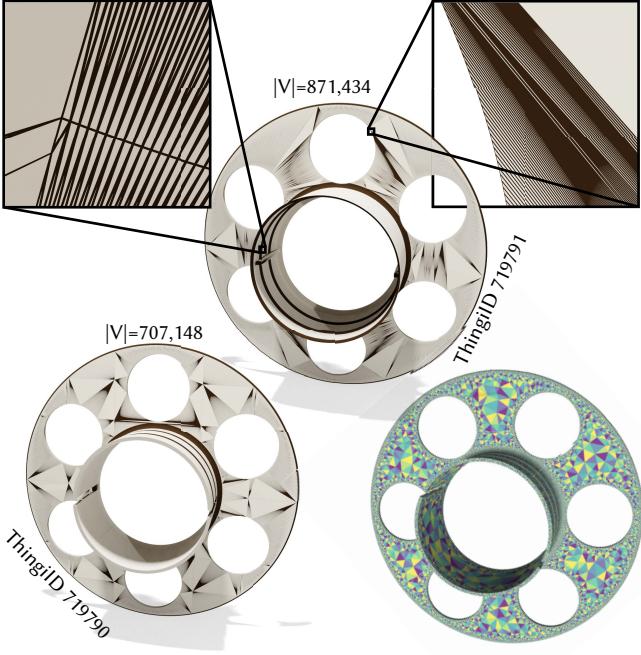


Fig. 20. We fail to compute an explicit mesh of the common subdivision following Delaunay refinement on one Thingi10k model (*top*). Its common subdivision would contain 34 million vertices and our program runs out of memory. We succeed on a nearly identical model (*bottom*), whose common subdivision contains merely 27 million vertices.

terminate with the expected conditions. Additionally, we successfully extract an explicit mesh of the common subdivision in both cases, except for 1 model in the case of refinement whose common subdivision contains around 30 million vertices (Figure 20, *top*).

We compare against the explicit overlay representation of Fisher et al. [2006] and the signpost representation of Sharp et al. [2019a] (Table 1). The overlay representation similarly offers a guarantee of valid connectivity, but does not provide a constant-time edge flip operation (like normal coordinates do). More importantly it does not support operations beyond edge flips and thus cannot perform Delaunay refinement. Signposts support a wide range of operations, but may not successfully recover the common subdivision on degenerate inputs (Figure 19). The statistic reported here differs from the result in Sharp et al. [2019a], because no preprocessing of meshes is performed. For refinement Sharp et al. [2019a] do not treat the boundary case, so we evaluate only on models without boundary.

6 LIMITATIONS AND FUTURE WORK

Limitations. The common subdivisions that we compute after Delaunay refinement can be quite large: the mean increase in $|V|$ is $20x$, and the 95th percentile increase is $45x$. We emphasize again that the common subdivision is not generally a high-quality mesh anyway: one should perform numerical computations on the intrinsic triangulation instead. The intrinsic mesh has much higher element quality and is generally much smaller with a mean increase in $|V|$ of $3.7x$ and 95th percentile increase of $7.8x$. However, for applications

that rely on the common subdivision (e.g. Section 4.4), it would still be beneficial to explore strategies for simplifying S .

A related issue is that Delaunay refinement sometimes generates meshes with many small triangles. One can prove that Delaunay refinement in the plane produces *well-graded* meshes, meaning essentially that it only places small triangles in regions with small features, and our Delaunay refinement on surfaces seems to behave similarly. Nonetheless, on poorly-conditioned input meshes, Delaunay refinement can insert many small triangles. This can cause problems for diffusion-based algorithms (e.g. the logarithmic map computation in Figure 16), which use the mean edge length to determine a suitable diffusion time. We found that computing the diffusion time on the original mesh and then performing diffusion on the intrinsic triangulation produced the best results.

Hybrid Data Structures. At this point, there are several intrinsic triangulation data structures, but no single one is perfect:

- Overlay (explicit) provides exact connectivity; flipping can be slow; no vertex insertion.
- Signposts (implicit) provide inexact connectivity; flipping and vertex insertion are both fast.
- Integer coordinates (implicit) provide exact connectivity; flipping is fast; vertex insertion can be slow.

We propose a good way to get the best of all worlds would be to use a hybrid signpost + integer coordinate data structure. This is fully implicit, so you don't pay the $O(n^2)$ cost when you have $O(n)$ edges crossing $O(n)$ edges. But, flipping and insertion are both fast, and connectivity is exact, if you accept the inserted locations.

Even further in the implicit direction, storing edge lengths is an “optimization” in our data structure. One could *just* store the normal coordinates and original triangulation, recovering edge length whenever necessary via a layout operation. This is appealing, since it is truly an integer-only representation for intrinsic triangulations.

General geodesic curves. It would also be natural to use this machinery as a representation for general geodesic curves on surfaces, which commonly arise in geometry processing tasks such as cutting, segmentation, *etc.*

7 ACKNOWLEDGMENTS

This work was supported by a Packard Fellowship, NSF Award 1717320, DFG TRR 109, an NSF Graduate Research Fellowship, and gifts from Autodesk, Adobe, and Facebook.

REFERENCES

- Mark Bell. 2013–2018. *flipper* (Computer Software). pypi.python.org/pypi/flipper.
- Mark Bell. 2015. *Recognising mapping classes*. Ph.D. Dissertation, University of Warwick.
- Alexander I Bobenko and Boris A Springborn. 2007. A discrete Laplace–Beltrami operator for simplicial surfaces. *Discrete & Computational Geometry* 38, 4 (2007), 740–756.
- L Paul Chew. 1993. Guaranteed-quality mesh generation for curved surfaces. In *Proceedings of the ninth annual symposium on Computational geometry*. 274–280.
- Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. 2008. Meshlab: an open-source mesh processing tool. In *Eurographics Italian chapter conference*, Vol. 2008. Salerno, Italy, 129–136.
- Keenan Crane, Clarisse Weischedel, and Max Wardetzky. 2017. The Heat Method for Distance Computation. *Commun. ACM* 60, 11 (Oct. 2017), 90–99.
- Olivier Devillers and Sylvain Pion. 2003. Efficient Exact Geometric Predicates for Delaunay Triangulations.. In *Proc. 5th Workshop Algorithm Eng. Exper.* 37–44.

- Ivan Dynnikov. 2020. Counting intersections of normal curves. *arXiv preprint arXiv:2010.01638* (2020).
- Jeff Erickson and Amin Nayyeri. 2013. Tracing compressed curves in triangulated surfaces. *Discrete & Computational Geometry* 49, 4 (2013), 823–863.
- Benson Farb and Dan Margalit. 2011. *A primer on mapping class groups (pms-49)*. Princeton University Press.
- Matthew Fisher, Boris Springborn, Alexander I Bobenko, and Peter Schröder. 2006. An algorithm for the construction of intrinsic Delaunay triangulations with applications to digital geometry processing. In *ACM SIGGRAPH 2006*, 69–74.
- Mark Gillespie, Boris Springborn, and Keenan Crane. 2021. Discrete Conformal Equivalence of Polyhedral Surfaces. *ACM Trans. Graph.* 40, 4 (2021).
- W. Haken. 1961. Theorie Der Normalflächen: Ein Isotopiekriterium Für Den Kreisknoten. *Acta Math.* 105, 3-4 (1961).
- Joel Hass and Maria Trmka. 2020. Approximating Isosurfaces by Guaranteed-quality Triangular Meshes. *Computer Graphics Forum* (2020).
- Allen Hatcher. 2002. *Algebraic Topology*. Cambridge University Press.
- Yixin Hu, Qingnan Zhou, Xifeng Gao, Alec Jacobson, Denis Zorin, and Daniele Panozzo. 2018. Tetrahedral meshing in the wild. *ACM Trans. Graph.* 37, 4 (2018), 60–1.
- Claude Indermitte, Th M Liebling, Marc Troyanov, and Heinz Cléménçon. 2001. Voronoi diagrams on piecewise flat surfaces and an application to biological growth. *Theoretical Computer Science* 263, 1-2 (2001), 263–274.
- H. Kneser. 1929. Geschlossene Flächen in Dreidimensionalen Mannigfaltigkeiten. *Jahresber. Dtsch. Math.-Ver.* 38 (1929).
- Felix Knöppel, Keenan Crane, Ulrich Pinkall, and Peter Schröder. 2013. Globally optimal direction fields. *ACM Trans. Graph.* 32, 4 (2013).
- D. Morera, P. Carvalho, and L. Velho. 2008. Modeling on Triangulations with Geodesic Curves. *The Visual Computer* 24, 12 (Dec 2008).
- Alexander Rand. 2011. Where and How Chew’s Second Delaunay Refinement Algorithm Works.. In *CCCG*.
- Rohan Sawhney and Keenan Crane. 2020. Monte Carlo Geometry Processing: A Grid-Free Approach to PDE-Based Methods on Volumetric Domains. *ACM Trans. Graph.* 39, 4 (2020).
- Marcus Schaefer, Eric Sedgwick, and Daniel Štefankovič. 2002. Algorithms for normal curves and surfaces. In *International Computing and Combinatorics Conference*. Springer, 370–380.
- Marcus Schaefer, Eric Sedgwick, and Daniel Štefankovič. 2008. Computing Dehn Twists and Geometric Intersection Numbers in Polynomial Time.. In *CCCG*, Vol. 20. 111–114.
- Max Schindler and Evan Chen. 2012. Barycentric Coordinates in Olympiad Geometry. *Olympiad Articles* (2012), 1–40.
- Silvia Sellán, Herng Yi Cheng, Yuming Ma, Mitchell Dembowski, and Alec Jacobson. 2019. Solid geometry processing on deconstructed domains. In *Computer Graphics Forum*, Vol. 38. Wiley Online Library, 564–579.
- Nicholas Sharp and Keenan Crane. 2020a. A Laplacian for Nonmanifold Triangle Meshes. *Computer Graphics Forum (SGP)* 39, 5 (2020).
- Nicholas Sharp and Keenan Crane. 2020b. You can find geodesic paths in triangle meshes by just flipping edges. *ACM Trans. on Graphics (TOG)* 39, 6 (2020), 1–15.
- Nicholas Sharp, Yousuf Soliman, and Keenan Crane. 2019a. Navigating intrinsic triangulations. *ACM Trans. on Graphics (TOG)* 38, 4 (2019), 1–16.
- Nicholas Sharp, Yousuf Soliman, and Keenan Crane. 2019b. The Vector Heat Method. *ACM Trans. Graph.* 38, 3 (2019).
- Jonathan R Shewchuk. 1997. *Delaunay refinement mesh generation*. Ph.D. Dissertation. Carnegie-Mellon Univ School of Computer Science.
- Gilbert Strang and George J Fix. 2008. An analysis of the finite element method (2 ed.). 212 (2008).
- Ge Xia. 2013. The stretch factor of the Delaunay triangulation is less than 1.998. *SIAM J. Comput.* 42, 4 (2013), 1620–1659.
- Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. 2016. Mesh arrangements for solid geometry. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–15.
- Qingnan Zhou and Alec Jacobson. 2016. Thingi10K: A Dataset of 10,000 3D-Printing Models. *arXiv preprint arXiv:1605.04797* (2016).

A PSEUDOCODE

We assume all algorithms have access to triangulations T^0 and T^1 , their edge lengths ℓ^0, ℓ^1 , the normal coordinates n , and the roundabouts r .

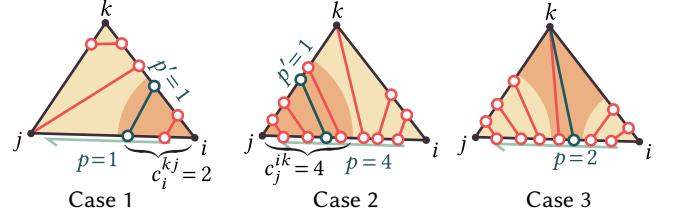


Fig. 21. A curve entering triangle jik along edge ij can proceed in 3 ways: it can exit along edge ik , in which case it is counted by c_i^{kj} (left); it can exit along edge kj , in which case it is counted by c_j^{ik} (center); or it can terminate at vertex k (right). This forms the core of algorithm TRACEFROM [Reproduced from Figure 8 for convenience].

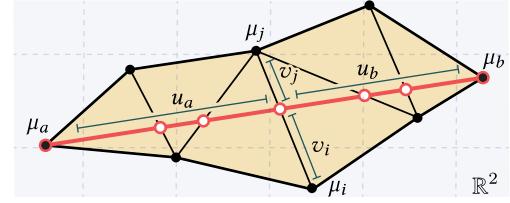


Fig. 22. In procedure EXTRACTCURVE, we compute barycentric coordinates by laying out a triangle strip in the plane [Reproduced from Figure 9 for convenience].

Algorithm 1 TRACEFROM(ζ)

Input: Any combinatorial crossing $\zeta = (\vec{ij}, p)$ along some curve γ lying along T^1 .

Output: The half of the curve γ as a sequence of points $(\zeta_0, \zeta_1, \dots, \zeta_n, k)$ along M , where $\zeta = \zeta_0$ and k is the vertex at which γ terminates.

```

1: currentHalfedge  $\leftarrow \vec{ij}$ 
2:  $\gamma \leftarrow [(\text{currentHalfedge}, p)]$ 
3: while True do Walk until the curve terminates at a vertex
4:   Let i and j refer to the tail and tip of the current halfedge
5:    $\vec{ij} \leftarrow \text{currentHalfedge}$ 
6:    $k \leftarrow \text{OPPOSITEVERTEX}(\text{TWIN}(\text{currentHalfedge}))$ 
7:   if  $p < c_i^{kj}$  then Case 1 of Figure 21 ( $\gamma$  goes right)
8:     currentHalfedge  $\leftarrow \vec{ik}$  Move to  $\vec{ik}$ 
9:      $p \leftarrow p$ 
10:    APPEND( $\gamma$ , (currentHalfedge, p))
11:   else if  $p \geq n_{ij} - c_j^{ik}$  then Case 2 of Figure 21 ( $\gamma$  goes left)
12:     currentHalfedge  $\leftarrow \vec{kj}$  Move to  $\vec{kj}$ 
13:      $p \leftarrow n_{kj} + p - n_{ij}$ 
14:     APPEND( $\gamma$ , (currentHalfedge, p))
15:   else Case 3 of Figure 21 ( $\gamma$  ends at k)
16:     return  $(\gamma, k)$ 

```

Algorithm 2 EXTRACTCURVE(ζ)

Input: Any combinatorial crossing $\zeta = (\vec{ij}, p)$ along some curve γ

Output: The entire trajectory of γ as a sequence of geometric crossings $(a, z_0, z_1, \dots, z_k, b)$ along M .

```

1: if runs along edge then
2:   return e
3: else
4:    $\gamma_{\text{front}}, b \leftarrow \text{TRACEFROM}(\zeta)$            ▷ Trace forwards along  $y$ 
5:    $\gamma_{\text{back}}, a \leftarrow \text{REVERSE}(\text{TRACEFROM}(\bar{\zeta}))$     ▷ Trace backwards
6:    $\gamma_{\text{Combinatorial}} \leftarrow \text{APPEND}(\gamma_{\text{back}}, \gamma_{\text{front}})$ 
7:   ▷ Compute positions in  $\mathbb{R}^2$  for the triangle strip containing  $\gamma$ 
8:    $\mu \leftarrow \text{LAYOUTTRIANGLESTRIP}(\gamma_{\text{Combinatorial}})$ 
9:    $\gamma_{\text{Geometric}} \leftarrow []$ 
10:  for  $\zeta = (\vec{ij}, p) \in \gamma_{\text{Combinatorial}}$  do
11:    ▷ Find the intersection of  $ab$  and  $ij$  in the plane (Figure 22)
12:     $u, v \leftarrow \text{INTERSECTIONBARYCENTRIC}(\mu_a, \mu_b, \mu_i, \mu_j)$ 
13:     $z \leftarrow (\vec{ij}, p, u, v)$ 
14:    APPEND( $\gamma_{\text{Geometric}}, z$ )
15:  return ( $a, \gamma_{\text{Geometric}}, b$ ),  $\gamma_{\text{Combinatorial}}$ 

```

Algorithm 3 EXTRACTEDGE(ab)

Input: An edge $ab \in E^0$

Output: The entire trajectory of ab as a sequence of geometric crossings $(a, z_0, z_1, \dots, z_l, b)$ along M

```

1:  $k \leftarrow$  local index of  $\vec{ab}$  about vertex  $a$  ▷ Find preceding halfedge
2:  $\vec{ai} \leftarrow \text{argmax}_{\vec{ai}} \{r_{\vec{ai}} : r_{\vec{ai}} \leq k\}$           ▷ Might wrap cyclically
3: if  $r_{\vec{ai}} = k$  and  $n_{\vec{ai}} = -1$  then                                ▷ Shared edge
4:   return  $\vec{ai}$ 
5: else
6:    $p \leftarrow k - r_{\vec{ai}} - 1$ 
7:    $\gamma \leftarrow \text{EXTRACTCURVE}(\text{NEXT}(\vec{ai}), p)$ 
8:   ▷ If  $\gamma$  does not end at a vertex of  $V^0$  we must keep tracing
9:   while  $\gamma$  does not end at a vertex of  $V^0$  do
10:     $i \leftarrow$  endpoint of  $\gamma$ 
11:    ▷ Our curves only pass through vertices inserted via edge splits. Hence, there is a unique other crossing  $\zeta$  emanating from  $j$  that we must trace along
12:     $\zeta \leftarrow$  other crossing emanating from  $j$ 
13:     $(i, z_1, \dots, z_s, j) \leftarrow \text{EXTRACTCURVE}(\zeta)$ 
14:    ▷ EXTRACTCURVE returns geometric crossings whose points have barycentric coordinates  $u$  computed relative to endpoints  $i$  and  $j$ . We should return barycentric coordinates relative to  $a$  and  $b$  instead. Since  $q_i^0, q_j^0$  give barycentric coordinates for  $i$  and  $j$  along  $ab$ , this amounts to linear interpolation of those coordinates
15:    ADJUSTBARYCENTRICCOORDINATES( $z_1, \dots, z_s$ )
16:    APPEND( $\gamma, (z_1, \dots, z_s, j)$ )
17:  return  $\gamma$ 

```

Algorithm 4 SPLITFACE_CASE1(ijk, u)

Input: The location to insert a vertex on T^1 , as barycentric coordinates u in a face $ijk \in F^1$.

Output: An updated integer coordinate intrinsic triangulation.

```

1: ▷ Gather all geometric crossings
2: for  $\zeta = (\vec{ij}, p) \in \text{COMBINATORIALCROSSINGS}(ijk)$  do
3:    $z_{\vec{ij}}[p+1] \leftarrow \text{EXTRACTGEOMETRICCROSSING}(\zeta)$ 
4: ▷ Compute new normal coordinates
5: for  $^{jk}_i \in \text{CORNERSOF}(ijk)$  do          ▷ for each corner

```

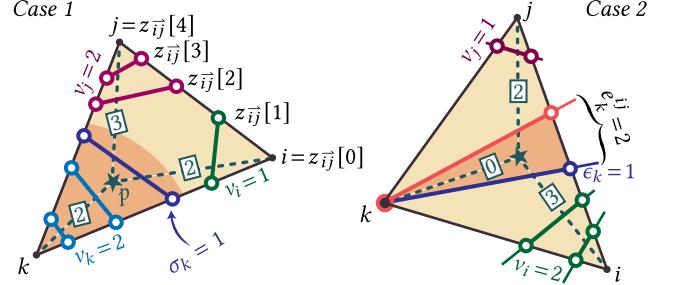


Fig. 23. In SPLITFACE, we do a sequence of line-side tests to compute a value of v at each corner.

```

6:   ▷ Identify which corner curves, if any, contain  $u$ 
7:    $C_i \leftarrow \{\xi : 0 \leq \xi < c_i^{jk}, u \in \text{TRIANGLE}(i, z_{\vec{ij}}[\xi], z_{\vec{ik}}[\xi])\}$ 
8:    $v_i \leftarrow \min(c_i^{jk} \cup C_i)$           ▷ Take the closest such corner
9:    $\sigma_i \leftarrow c_i^{jk} - v_i$                   ▷ "Slack" left at corner
10:  ▷ In exact arithmetic, only one  $\sigma_i$  may be nonzero. In floating point multiple could be nonzero, so we keep the biggest and round the others to zero
11:  if  $\sigma_i \geq \sigma_j, \sigma_k$  then
12:     $\sigma_j, \sigma_k \leftarrow 0, 0$ 
13:  else if  $\sigma_j \geq \sigma_k, \sigma_i$  then
14:     $\sigma_k, \sigma_i \leftarrow 0, 0$ 
15:  else if  $\sigma_k \geq \sigma_i, \sigma_j$  then
16:     $\sigma_i, \sigma_j \leftarrow 0, 0$ 
17:  for  $^{jk}_i \in \text{CORNERSOF}(ijk)$  do          ▷ include slack crossings
18:     $n_{pi} \leftarrow v_i + \sigma_j + \sigma_k$ 
19:  ▷ Compute everything else
20:   $r \leftarrow \text{UPDATEROUNABOUTS}(n_{pi}, n_{pj}, n_{pk}, r)$           ▷ Equation 4
21:   $\ell_{pi}^1, \ell_{pj}^1, \ell_{pk}^1 \leftarrow \text{UPDATEEDGELLENGTHS}(\ell^1, u)$           ▷ Equation 5
22:   $R \leftarrow \text{REGIONFROMNORMALCOORDINATES}(n_{pi}, n_{pj}, n_{pk})$ 
23:   $q_p^0 \leftarrow \text{RECOVERBARYCENTRIC}(R, Z, u)$           ▷ Appendix B

```

Algorithm 5 SPLITFACE_CASE2(ijk, u)

Input: The location to insert a vertex on T^1 , as barycentric coordinates u in a face $ijk \in F^1$. In Case 1, ijk must be oriented so that $n_{ij} \geq n_{jk} + n_{ki}$

Output: An updated integer coordinate intrinsic triangulation.

```

1: ▷ Gather all geometric crossings
2: for  $\zeta = (\vec{ij}, p) \in \text{COMBINATORIALCROSSINGS}(ijk)$  do
3:    $z_{\vec{ij}}[p+1] \leftarrow \text{EXTRACTGEOMETRICCROSSING}(\zeta)$ 
4: ▷ Compute new normal coordinates
5: for  $^{jk}_i \in \text{CORNERSOF}(ijk)$  do          ▷ for each corner
6:   ▷ Identify which corner curves, if any, contain  $u$ 
7:    $C_i \leftarrow \{\xi : 0 \leq \xi < c_i^{jk}, u \in \text{TRIANGLE}(i, z_{\vec{ij}}[\xi], z_{\vec{ik}}[\xi])\}$ 
8:    $v_i \leftarrow \min(c_i^{jk} \cup C_i)$           ▷ Take the closest such corner
9:    $\sigma_i \leftarrow c_i^{jk} - v_i$                   ▷ "Slack" left at corner
10:  ▷ Note that  $v_k = \sigma_k = 0$ 
11:  if  $\sigma_i \geq \sigma_j$  then          ▷ Ensure that at most one  $\sigma$  is nonzero

```

```

12:    $\sigma_j \leftarrow 0$ 
13: else if  $\sigma_j \geq \sigma_i$  then
14:    $\sigma_i \leftarrow 0$ 
15: Check for intersections with emanating edges
16: if  $v_i < c_i^{jk}$  then ▷ In corner i
17:    $v_j \leftarrow v_j + e_k^{ij}$ 
18: else if  $v_j < c_j^{ki}$  then ▷ In corner j
19:    $v_i \leftarrow v_i + e_k^{ij}$ 
20: else ▷ In middle of fan region
21:   Identify which emanating curves, if any, contain u
22:    $\mathcal{E}_k \leftarrow \{\xi : 0 \leq \xi < e_k^{ij}, u \in \text{TRIANGLE}(i, z\vec{ij}[\xi + c_i^{jk}], k)\}$ 
23:    $\epsilon_k \leftarrow \min(e_k^{ij} \cup \mathcal{E}_k)$  ▷ Take the curve closest to i
24:    $v_i \leftarrow v_i + \epsilon_k$  ▷ Edge pi crosses the first  $\epsilon_k$  such curves
25:    $v_j \leftarrow v_j + e_i^{jk} - \epsilon_k$  ▷ Edge pj crosses the rest
26: for  $j^k \in \text{CORNERSOF}(ijk)$  do ▷ include opposite corners
27:    $n_{pi} \leftarrow v_i + \sigma_j + \sigma_k$ 
28: Compute everything else
29:    $r \leftarrow \text{UPDATEROUNDABOUTS}(n_{pi}, n_{pj}, n_{pk}, r)$  ▷ Equation 4
30:    $\ell_{pi}^1, \ell_{pj}^1, \ell_{pk}^1 \leftarrow \text{UPDATEEDGELENGTHS}(\ell^1, u)$  ▷ Equation 5
31:    $R \leftarrow \text{REGIONFROMNORMALCOORDINATES}(n_{pi}, n_{pj}, n_{pk})$ 
32:    $q_p^0 \leftarrow \text{RECOVERBARYCENTRIC}(R, Z, u)$  ▷ Appendix B

```

Algorithm 6 SPLITEDGE(\vec{ij}, u)

Input: The location to insert a vertex on T^1 , as barycentric coordinates u on a halfedge $\vec{ij} \in H^1$

Output: An updated integer coordinate intrinsic triangulation

```

1: if  $n_{ij} \geq 0$  then
2:    $k \leftarrow \text{OPPOSITEVERTEX}(\vec{ij})$ 
3:    $\text{SPLITFACE}(ijk, (u_i, u_j, 0))$ 
4:    $\text{FLIPEDGE}(ij)$ 
5: else
6:    $k \leftarrow \text{OPPOSITEVERTEX}(\vec{ij})$ 
7:   if ININTERIOR( $\vec{ij}$ ) then
8:      $l \leftarrow \text{OPPOSITEVERTEX}(\vec{ji})$ 
9:      $T^1 \leftarrow \text{insert a vertex } p \text{ along } ij$  ▷ Update combinatorics
10:     $n_{pj}, n_{pi} \leftarrow n_{ij}, n_{ij}$  ▷ Compute new normal coordinates
11:     $n_{pk} \leftarrow \max(n_{ki}, n_{jk}, 0)$ 
12:   Compute everything else
13:    $r \leftarrow \text{UPDATEROUNDABOUTS}(n_{pi}, n_{pj}, n_{pk}, n_{pl}, r)$  ▷ Equation 4
14:    $\ell_{pi}^1, \ell_{pj}^1, \ell_{pk}^1, \ell_{pl}^1 \leftarrow \text{UPDATEEDGELENGTHS}(\ell^1, u)$  ▷ Equation 5
15:    $q_p^0 \leftarrow u_i q_i^0 + u_j q_j^0$ 

```

Algorithm 7 REMOVEVERTEX(i)

Input: An inserted vertex i

Output: Updated triangulation i removed

```

1: while  $i$  has degree  $> 3$  do
2:    $ij \leftarrow$  flippable edge incident on  $i$ 
3:    $\text{FLIPEDGE}(ij)$ 
4:    $\text{DELETEVERTEXANDINCIDENTEDGES}(i)$ 

```

Algorithm 8 COMPUTECOMMONSUBDIVISION()

Input: Nothing beyond the usual data (i.e. T^0, T^1, \dots)

- 1: **Index common subdivision vertices**
- 2: $\mathcal{I}^v \leftarrow$ index common subdivision vertices
- 3: **Compute connectivity**
- 4: $\text{polygons} \leftarrow []$
- 5: **for** $ijk \in F^1$ **do** ▷ Always orient such that $n_{ij} \geq n_{jk}, n_{ki}$
- 6: **if** $e_k^{ij} = 0$ **then**
- 7: APPEND(polygons, SUBDIVIDEFACE_CASE1(ijk, \mathcal{I}^v))
- 8: **else**
- 9: APPEND(polygons, SUBDIVIDEFACE_CASE2(ijk, \mathcal{I}^v))
- 10: **Compute intersection geometry. We denote the locations on T^0 by Q^0 and the locations on T^1 by Q^1**
- 11: **for** $i \in V^1$ **do** ▷ Vertices of T^1
- 12: $k \leftarrow \mathcal{I}_i^v$ ▷ Index of vertex in S
- 13: $Q_k^0 \leftarrow q_i^0$ ▷ Location on T^0 computed when i was inserted
- 14: $Q_k^1 \leftarrow (i, 1)$ ▷ Location on T^1 is just i itself
- 15: **for** $ab \in E^0$ **do** ▷ Edge intersections
- 16: $\gamma = (a, z_1, \dots, z_l, b) \leftarrow \text{EXTRACTEDGE}(ab)$
- 17: **for** $z = (\vec{ij}, p, u, v) \in \gamma$ **do**
- 18: $k \leftarrow \mathcal{I}_{\vec{ij}}^v[p+1]$ ▷ Index of crossing in S
- 19: $Q_k^0 \leftarrow (ab, u)$ ▷ Position on T^0 along ab
- 20: $Q_k^1 \leftarrow (\vec{ij}, v)$ ▷ Position on T^1 along \vec{ij}
- 21: **return** polygons, Q^0, Q^1

Algorithm 9 DELAUNAYREFINEMENT(θ_{\min})

Input: A minimum allowed angle θ_{\min} .

Output: An intrinsic triangulation T^1 whose corner angles are all at least θ_{\min}

```

1:  $\text{FLIPTODELAUNAY}()$ 
2: while  $T^1$  has triangles with angles less than  $\theta_{\min}$  do
3:    $ijk \leftarrow$  any triangle with an angle less than  $\theta_{\min}$ 
4:   Find the circumcenter via the exponential map
5:    $v_c \leftarrow$  circumcenter barycentric coordinates ▷ Equations 9, 10
6:   Barycentric coordinate offset from barycenter to circumcenter
7:    $\delta v_c \leftarrow v_c - (1/3, 1/3, 1/3)$ 
8:   Transform offset to face tangent space
9:    $V \leftarrow \text{BARYCENTRICOFFSETTOTANGENTVECTOR}(\delta v_c)$ 
10:  Evaluate exponential map from face barycenter
11:   $c \leftarrow \text{EXP}(\text{BARYCENTER}(ijk), V)$ 
12:  if  $c$  lies inside the mesh then
13:     $\text{INSERTCIRCUMCENTER}(ijk)$ 
14:  else
15:     $lm \leftarrow$  boundary edge separating  $c$  from  $ijk$ 
16:     $p \leftarrow \text{SPLITEDGE}(lm, 0.5)$ 
17:    Must flip to Delaunay before computing Dijkstra ball
18:     $\text{FLIPTODELAUNAY}()$ 
19:    Remove inserted vertices from lm's diametral ball
20:    ball =  $\{i \in V^1 : \text{DIJKSTRADISTANCE}(E^1, i, p) < \ell_{lm}\}$ 
21:    for  $i \in \text{ball}$  do
22:       $\text{REMOVEVERTEX}(i)$ 

```

B BARYCENTRIC COORDINATES RECOVERY

In Section 3.4, we recover the barycentric coordinates of a newly inserted vertex on T^0 via interpolation along a polygonal subregion R of triangle $abc \in F^0$. Here we give a full expression for the necessary small linear system.

Precisely, let $3 \leq \rho \leq 6$ denote the number of corners of R . Let the m^{th} corner of R have barycentric coordinates $u_a^{(m)}, u_b^{(m)}, u_c^{(m)}$ on $abc \in F^0$ and barycentric coordinates $v_i^{(m)}, v_j^{(m)}, v_k^{(m)}$ on $ijk \in F^1$, all of which are known. We also know the barycentric coordinates v_i for $p \in ijk$. We then want to solve for the corresponding u_a on abc . We proceed in two steps: first, we express v as a linear combination ξ of the $v^{(m)}$. Then, we apply this same linear combination to the $u^{(m)}$ to obtain u . Concretely, we first solve for the minimum-norm solution of the underdetermined system

$$\begin{pmatrix} v_i^{(0)} & v_i^{(1)} & \dots & v_i^{(\rho)} \\ v_j^{(0)} & v_j^{(1)} & \dots & v_j^{(\rho)} \\ v_k^{(0)} & v_k^{(1)} & \dots & v_k^{(\rho)} \end{pmatrix} \begin{pmatrix} \xi_0 \\ \xi_1 \\ \vdots \\ \xi_\rho \end{pmatrix} = \begin{pmatrix} v_i \\ v_j \\ v_k \end{pmatrix}, \quad (14)$$

and then set

$$u_a := \sum_m u_a^{(m)} \xi_m, \quad u_b := \sum_m u_b^{(m)} \xi_m, \quad u_c := \sum_m u_c^{(m)} \xi_m. \quad (15)$$

Note that while one often seeks a nonnegative ξ , any solution will suffice here: we only use ξ to interpolate in Equation 15.

C DELAUNAY REFINEMENT DETAILS

C.1 Removing Extra Vertices

When Chew’s second algorithm splits an edge, it removes all inserted circumcenters within a geodesic ball centered at the edge’s midpoint. These vertices must be removed, but it is okay to remove additional interior inserted vertices. Shewchuk [1997, Section 3.4.2] observes that the algorithm can only perform finitely many edge splits. As long as one removes all interior inserted vertices within the geodesic ball—and never removes vertices along the boundary—the algorithm will still perform only finitely many edge splits. Hence, it must terminate as usual following the final edge split, even if one removes extra circumcenters during edge splits.

C.2 Proof of Correctness on Watertight Meshes

Here we seek to prove that DELAUNAYREFINEMENT (Algorithm 9) succeeds, in the basic case of a closed surface with bounded cone angles. We will not prove the more general boundary case here, but experimentally we observe success on a large dataset (Section 5.1).

THEOREM C.1 (DELAUNAY REFINEMENT, NO BOUNDARY). *On meshes without boundary, with vertex angle sums at least 60° , Algorithm 9 produces a Delaunay mesh with triangle corner angles at least 30° .*

PROOF. By definition, DELAUNAYREFINEMENT only terminates when the triangulation is a Delaunay triangulation which satisfies the angle bound, so we just need to prove that termination occurs after a finite number of iterations. We will show this by establishing that DELAUNAYREFINEMENT maintains a minimum spacing between

all vertices in the mesh, so the number of insertions is bounded by surface area. Our argument will generally follow the planar proof of Shewchuk [1997, Section 3.2.1], though extra care is needed in the intrinsic case, where self edges may connect a vertex to itself.

In particular, we consider the length of the shortest edge in the initial mesh’s intrinsic Delaunay triangulation, $\delta := \min_{ij} \ell_{ij}$. We will show that the minimum edge length in each subsequent Delaunay triangulations is at least δ . Then all vertices must be separated by a distance at least δ , since Lemma C.3, each vertex is connected to its geodesic nearest neighbor. Hence, each vertex is contained in an open disk of radius $\frac{1}{2}\delta$ which is disjoint from all other disks. As the input mesh has finite surface area, we conclude that Algorithm 9 can only insert finitely many vertices, and thus must terminate.

It remains to show that DELAUNAYREFINEMENT never creates an edge of length less than δ . It is convenient to convert the angle bound α to a *circumradius-to-shortest-edge ratio* bound $B = \frac{1}{2\sin\alpha}$ [Shewchuk 1997, Section 3.1]. Having corner angles at least $\alpha = 30^\circ$, is equivalent to a circumradius-to-shortest-edge ratio of at most $B = 1$, and thus we insert the circumcenters of triangles with $B > 1$.

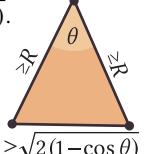
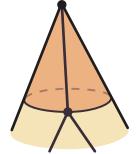
We proceed by induction. All initial edges have length at least δ by definition. Now consider inserting vertex i at the circumcenter of triangle jkl with circumradius R . Since we only split triangles with $B > 1$, and jkl ’s edges have length at least δ , we must have $R > \delta$. By Lemma C.4 all new edges in the Delaunay triangulation must be incident on i , and since jkl had an empty geodesic circumcircle, there can be no other vertices within distance $R > \delta$. Thus new all edges to other vertices have length at least δ . We must now consider self edges connecting the new vertex i to itself.

Gluing together the two ends of a self edge yields a loop; we will split into cases based on the homotopy class of this loop on the punctured surface (before the insertion of i). First, note that the loop cannot be contractible to a point, since the original edge is geodesic. Then we will split in to two cases: either the loop contracts around a single vertex, or it does not.

If the loop contracts around a single vertex, then the self edge encloses a degree-1 vertex. The degree-1 vertex must have distance at least R to the inserted vertex, and has angle sum at least 60° . Thus, by the law of cosines, the length of the self edge must be at least

$$\sqrt{R^2 + R^2 - 2R^2 \cos \theta} = R\sqrt{2(1 - \cos \theta)}.$$

Since $\cos 60^\circ = \frac{1}{2}$, and $1 - \cos \theta$ is increasing with θ , this shows that the self edge has length at least R whenever θ is at least 60° .



If the loop is not in a homotopy class contractible about a single vertex, then the shortest loop γ_{\min} in the homotopy class is non-constant. By Lemma C.5, we can take γ_{\min} to touch some vertex a , and note that since γ_{\min} is the *shortest* loop our original self edge must be at least as long as γ_{\min} . Then by Lemma C.3 a has an edge at least as long as γ_{\min} , and thus the self edge has length $\geq |\gamma_{\min}| \geq \delta$.

Thus, we conclude that Algorithm 9 never introduces an edge of length less than δ , which means that it must terminate after inserting finitely many vertices. \square

LEMMA C.2. For any pair of vertices $i, j \in V$, let Γ_{ij} be the set of non-constant geodesics connecting i to j . Then

$$d_{ij} := \inf_{\gamma \in \Gamma_{ij}} \text{length}(\gamma) > 0.$$

PROOF. This follows directly from [Indermitte et al. 2001, Proposition 1], which states that for any $L > 0$, the number of geodesic arcs from i to j of length at most L is finite. Since any geodesic of length 0 is constant, and thus not in Γ_{ij} , this implies that $d_{ij} > 0$. \square

LEMMA C.3. For any vertex $i \in V$, the intrinsic Delaunay triangulation contains an edge to i 's nearest neighbor.

PROOF. This is a standard result, which we include for completeness. Let j be i 's nearest neighbor, i.e. $j := \operatorname{argmin}_j d_{ij}$. Note that j may equal i , and $d_{ij} > 0$ by Lemma C.2. Consider the disk D of radius d_{ij} centered at i . Since j is i 's nearest neighbor, D contains no vertex other than i . Thus, the circle which goes through i and j and is tangent to D at j has empty interior, and its boundary contains no vertices other than i and j . We conclude that ij is in the Delaunay triangulation [Bobenko and Springborn 2007, Definition 3]. \square

LEMMA C.4. All edges created in DELAUNAYREFINEMENT following the insertion of a vertex i and flipping to Delaunay are incident on i .

PROOF. Again, we follow the planar proof of Shewchuk [1997, Lemma 12]. We wish to prove that all Delaunay edges which are not incident on i were Delaunay before inserting i . This follows from the fact that edges of a Delaunay triangulation satisfy an empty circumcircle condition [Bobenko and Springborn 2007, Definition 3]. If an edge's circumcircle is empty after inserting vertex i , it must have been empty before too, so the edge was already Delaunay. \square

LEMMA C.5. Any geodesic loop γ is isotopic to a geodesic loop γ' of the same length which touches a vertex.

PROOF. γ can “slide” until it touches a vertex without changing its length. Precisely, consider a unit-speed motion of γ within the surface along its outward normal direction. During the motion, $\frac{d}{dt}|\gamma| = \int_\gamma \kappa(s) ds$, where κ is the geodesic curvature of γ . Since γ is a geodesic, $\kappa = 0$: its length does not change. Thus we can construct γ' by sliding γ along the surface until it touches a vertex. \square

As an aside, we note that geodesic loops which do not touch a vertex only occur in non-generic configurations.

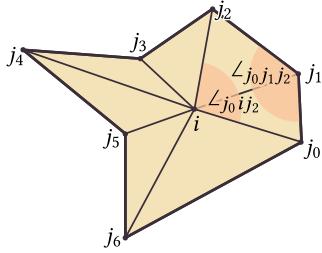


Fig. 24. The relevant angles for Theorem D.1.

D SIMPLICIAL VERTEX REMOVAL

In Section 3.6, we consider removing a vertex by flipping edges until the vertex has degree three and then deleting it. Past work has also proposed this approach in the purely-topological setting [Schaefer et al. 2002, Section 5.4], but here we must respect geometric constraints. In particular, edges can only be flipped geometrically if they are contained in a convex quadrilateral (Section 3.3). Here we prove that flipping edges to remove vertices is indeed a viable strategy in the Euclidean setting as well: one can always find an edge to flip.

THEOREM D.1 (VERTEX REMOVAL, SIMPLICIAL). *If a vertex i in a simplicial complex has cone angle 2π and degree $d > 3$, then some edge ij incident on i can be flipped to decrease the degree of i .*

PROOF. Recall that an edge can be flipped if both endpoints will have degree at least 1 after the flip, and the edge is contained in a convex quadrilateral (Section 3.3). As always, the convex quadrilateral is defined in the sense of the intrinsic geometry determined by edge lengths. The endpoint degree constraint is automatically satisfied on a simplicial complex, so we only need to show that the geometric convexity constraint is satisfied, which is equivalent to showing that all angles of the edge's quadrilateral are at most π .

Denote the neighboring vertices of i as j_k , with j_{k+1} etc. implicitly indexed modulo the vertex degree d (Figure 24). The outer angles $\angle_{ijk_{k+1}}$ and $\angle_{ijk_{k+1}k}$ are corners of Euclidean triangles, and thus are necessarily at most π , so we need to find an edge ijk_k for which the angles $\angle_{jk_{k-1}ijk_k}$ and $\angle_{jk_{k+1}ijk_k}$ are also at most π .

First we consider the inner corners $\angle_{jk_{k-1}ijk_{k+1}}$. At most two of these angles can be greater than π . To see why, suppose there were three $\angle_{jk_{k-1}ijk_{k+1}} > \pi$. Since the degree of i is $d > 3$, then some pair of those three large angles would correspond to disjoint angular sectors around the vertex, and summing their angles yields a value greater than 2π , which is impossible because the angle sum of i is 2π . Thus all but at most two of the edges incident on i have inner corners with angle at most π .

Likewise, at least three of the outer corners $\angle_{jk_{k+1}ijk_{k-1}}$ are at most π . This is because the sum of all d outer corners must be $(d - 2)\pi$. Since they are nonnegative, at most $d - 3$ of them can be strictly greater than π , implying that at least 3 will be π .

Thus at least three outer corners are at most π , and at most two of the inner corners are *not* at most π , so there must be at least one edge for which both the inner and outer corners are at most π . This edge can then be flipped, reducing the vertex degree. \square

Importantly, this proof *does not* handle the full general case of a Δ -complex, where there may exist self-edges which cause flips to not make progress. However, we note that Sharp and Crane [2020b, Appendix A] proves that a similar flip-removal strategy works in the case of a Δ -complex, and we conjecture that an analogous technique could be applied to generalize Theorem D.1. Also, note that the “equality” case of Theorem D.1 is a possibility, such as a degree four cross configuration where all angles = $\pi/2$. Fortunately the resulting skinny triangle after the edge is a non-issue, because the center vertex is about to be removed.