Integer Coordinates for Intrinsic Geometry Processing

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

This paper describes a numerically robust data structure for encoding intrinsic triangulations of polyhedral surfaces. Many applications demand a correspondence between the intrinsic triangulation and the input surface, but existing data structures either rely on floating point values to encode correspondence, or do not support remeshing operations beyond basic edge flips. We instead provide an integer-based data structure that guarantees valid correspondence, even for meshes with near-degenerate elements. Our starting point is the framework of *normal coordinates* from geometric topology, which we extend to the broader set of operations needed for mesh processing (vertex insertion, edge splits, *etc.*). The resulting data structure can be used as a drop-in replacement for earlier schemes, automatically improving reliability across a wide variety of applications. As a stress test, we successfully compute an intrinsic Delaunay refinement and associated subdivision for all manifold meshes in the Thingi10k dataset. In turn, we can compute reliable and highly accurate solutions to partial differential equations even on extremely low-quality meshes.

CCS Concepts: • **Mathematics of computing → Mesh generation**.

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

ACM Reference Format:
Mark Gillespie, Nicholas Sharp, and Keenan Crane. 2021. Integer Coordinates for Intrinsic Geometry Processing. *ACM Trans. Graph.* 40, 6, Article 1 (December 2021), 13 pages. https://doi.org/10.1145/3478513.3480522

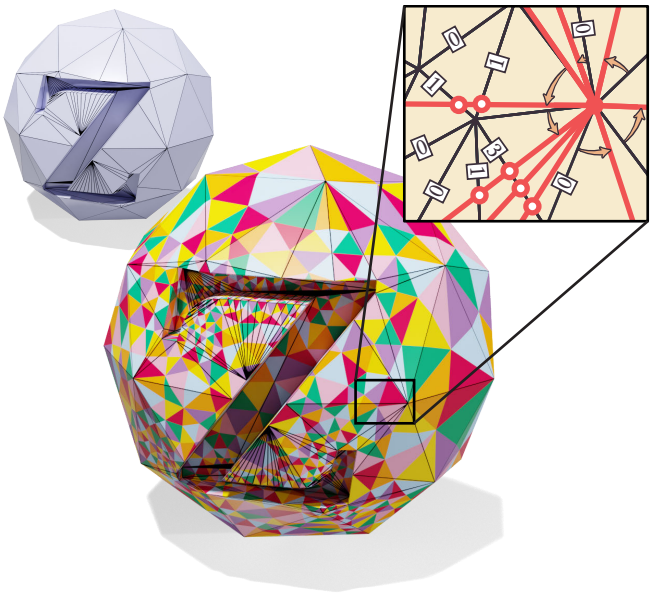


Fig. 1. Intrinsic triangulations enable one to compute with a high-quality triangulation even on a low-quality input mesh. We introduce an integer-based intrinsic triangulation data structure that is dramatically more robust than past alternatives.

1 INTRODUCTION AND RELATED WORK

Polyhedral surfaces play a central role in graphics, geometry processing, scientific computing, and computer vision, but meshes from these domains are often not directly suitable for computation. For instance, a mesh that is perfectly good for visualization may not be suitable for solving *partial differential equations (PDEs)*, which are a basic component of many modern geometric algorithms. Significant work hence focuses on algorithms that are robust to mesh quality [Zhou et al. 2016; Schneider et al. 2018; Sellán et al. 2019; Sawhney and Crane 2020; Jiang et al. 2020], including extremely robust remeshing [Hu et al. 2018, 2020]. However, current best-in-class remeshing techniques are volumetric in nature, making them less than ideal for surface problems: for instance, they do not preserve boundaries or self-intersections, and are orders of magnitude more expensive (seconds to minutes) than traditional 2D remeshing (on the order of milliseconds). More fundamentally, standard *extrinsic* approaches to remeshing based on vertex positions in $\mathbb{R}^n$ must negotiate a trade-off between mesh size, the quality of mesh elements, and geometric approximation of the input domain.

Intrinsic triangulations provide a framework for surface mesh processing with computational cost and behavior similar to ordinary 2D meshing. The basic idea is to trace out a secondary mesh along straight (*i.e.*, geodesic) paths that triangulate the same vertex set—opening up a dramatically larger space of triangulations for computation. Though intrinsic triangles may appear bent, they are fully described by just three ordinary edge lengths, since they can be unfolded into the plane without any distortion (Figure 2). Hence, algorithms designed for ordinary triangle meshes can be run on the intrinsic mesh with little to no modification. Sharp et al. [2021] provides an in-depth introduction, including a detailed discussion of practical applications.

This larger space of triangulations also makes it possible to improve element shape without introducing geometric approximation error (Figure 1). Encapsulating this machinery in a standard interface provides robustness as a subroutine: rather than make algorithms more robust one at a time, we can transform low-quality input into a high-quality intrinsic mesh, execute a "non-robust" algorithm, then read back the results in a variety of ways.

However, to make this approach truly reliable one must develop numerically robust data structures for encoding the correspondence between the input mesh T^0 and an intrinsic triangulation T^1 traced out over T^0. The earliest example is the *overlay mesh* of Fisher et al. [2006], which explicitly tracks the *common subdivision* obtained by "slicing up" T^0 along the edges of T^1. This approach guarantees

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

0730-0301/2021/12-ART1
https://doi.org/10.1145/3478513.3480522

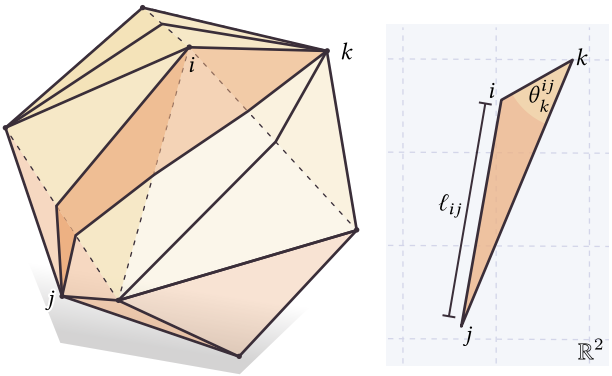


Fig. 2. *Left:* we triangulate the vertices of a polyhedral surface by edges along geodesic paths (*left*). *Right:* since the three edge lengths of an intrinsic triangle ijk satisfy the usual triangle inequalities, we can express quantities like angles and areas via standard formulas from Euclidean geometry.

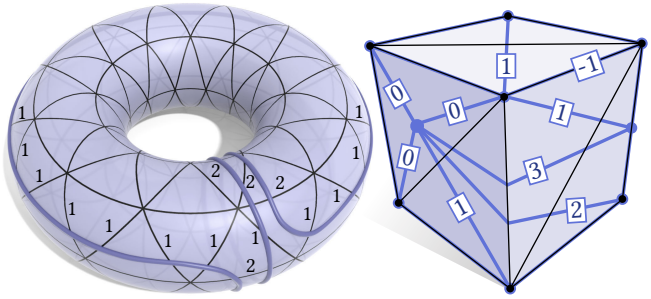


Fig. 3. *Left: normal coordinates* count how many times a given curve crosses each edge of a triangulation. *Right:* more generally, we can count the total number of times each edge of one triangulation crosses any edge of another. Remarkably, one integer per edge is sufficient to recover the geometry of the intersections.

correct connectivity, but even basic operations like edge flips are non-local and expensive to evaluate; further operations (such as vertex insertion) were never described. Sharp et al. [2019a] encode correspondence implicitly via *signposts* at vertices, which give the direction and length of each outgoing intrinsic edge. A variety of local operations are now cheap to evaluate, but the connectivity of the common subdivision is no longer guaranteed to be correct since it is encoded via inexact floating-point values. For instance, an intrinsic edge ij traced from vertex i may fail to reach the neighborhood of the other vertex j.

Integer-Based Encoding. Our data structure offers the best of both worlds: an implicit encoding of correspondence that supports fast local operations, but also guarantees correct connectivity. Since we augment traditional integer-valued normal coordinates with additional integer-based *roundabouts à la* Gillespie et al. [2021], we refer to the overall encoding as *integer coordinates*. Like signposts, integer coordinates support many operations beyond edge flips (Section 3), but are dramatically more robust. A key example is *intrinsic Delaunay refinement* (Section 4.2), which significantly improves robustness for PDE-based geometry processing (Section 5)—but is often useful only if one can reliably extract the *common subdivision* over which the solution is interpolated (Section 6).

Our starting point is the concept of *normal coordinates* from geometric topology[1]—the basic idea is to count how many times each edge of a triangulation is crossed by a given curve (Figure 3, *left*), as detailed in Section 2.3. As long as the curve is *normal* (*i.e.*, does not enter and exit any face through the same edge), these numbers alone are sufficient to recover the original curve, up to a homotopy that does not pass through vertices. More generally, one can consider a collection of curves—such as the edges of a second triangulation (Figure 3, *right*). Normal coordinates were originally developed to study surfaces in 3-manifolds [Kneser 1929; Haken 1961; Hass and Trnkova 2020], and appear throughout topology (*e.g.*, for encoding 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, since the total bits required to store a long winding curve can be exponentially smaller than storing explicit segments along the curve [Erickson and Nayyeri 2013]. However, we must augment classical normal coordinates in several ways in order to make them suitable for geometry processing.

One challenge is that past literature rarely considers operations beyond edge flips, and even then only for closed loops (*e.g.* [Schaefer et al. 2002, Section 5.4]). For triangulations, it is essential to handle open curves terminating at vertices. A second issue is that normal coordinates alone are not enough to uniquely identify traced curves with the logical edges of a mesh. Gillespie et al. [2021, Section 5.2] introduce so-called *roundabouts* to address this issue, but again consider only edge flips. Finally, whereas normal coordinates typically encode topological curves (or perhaps hyperbolic geodesics [Mosher 1988]), we use them to encode a geodesic triangulation of a Euclidean polyhedron. This distinction is important since not all normal coordinates describe such a triangulation; this specialization in turn helps to establish procedures not yet seen.

Contributions. Overall, we make the following contributions:

- We describe how normal coordinates (plus roundabouts) can be used as a general representation for intrinsic triangulations of Euclidean polyhedra, including triangulations where new vertices have been inserted.
- We extend existing integer-based data structures to local operations beyond edge flips.
- We extend the intrinsic Delaunay refinement algorithm from Sharp et al. [2019a, Section 4.2] to surfaces with boundary, and prove correctness and quality guarantees watertight surfaces.
- We present an accurate way to transfer piecewise linear functions between input and intrinsic triangulations.

Robustness of our technique is evaluated by performing intrinsic Delaunay refinement on all manifold meshes in the Thingi10k dataset and extracting the common subdivision (Section 6), which in turn improves accuracy for various geometry processing tasks geometry processing (Section 5).

[1]Not to be confused with *geodesic normal coordinates* from Riemannian geometry.

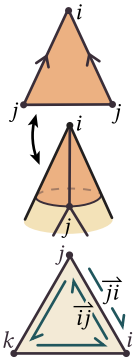
2 DATA STRUCTURE

Our data structure consists of a fixed input triangulation T^0, plus a dynamic *intrinsic triangulation* T^1 sitting on top of T^0 (Section 2.1). We use V^i, E^i, F^i to denote the vertices, edges, and faces of T^i, *resp.* Initially T^0 and T^1 are identical, but subsequent local operations may add vertices to T^1. We never remove vertices of T^0 since, in general, such vertices cannot be removed without changing the geometry of the surface. Hence, $V^1 \supseteq V^0$, and $V^\star := V^1 \setminus V^0$ gives the set of inserted vertices. For clarity, we use a, b, c for vertices of T^0 and i, j, k for vertices of T^1 (which may also be in T^0). We also associate four basic quantities with T^1:

- lengths $\ell_{ij} \in \mathbb{R}_{>0}$ for each edge $ij \in E^1$ (Section 2.2),
- normal coordinates $n_{ij} \in \mathbb{Z}$ for each edge $ij \in E^1$ (Section 2.3),
- roundabouts $r_{\overrightarrow{aj}} \in \mathbb{Z}_{\geq 0}$ for each halfedge $\overrightarrow{aj} \in H^1$ rooted at a vertex $a \in V^0$ (Section 2.4), and
- barycentric coordinates q_i^0 relative to T^0 for each inserted vertex $i \in V^\star$ (Section 2.2).

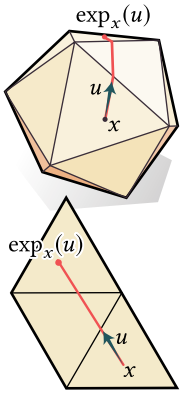
Note that this scheme differs from Gillespie et al. [2021, Section 5], who always assume that $V^1 = V^0$—and hence cannot support many of the local operations described in Section 3.

2.1 Connectivity



Formally, we consider triangulations $T = (V, E, F)$ of a fixed polyhedral surface M. By *triangulation* we mean a Δ-*complex* [Hatcher 2002, Section 2.1], which allows, *e.g.*, a single intrinsic triangle to "wrap around" the surface and meet along its own edge (inset, *top*). In general, the vertices of a face or edge in a Δ-complex need not be distinct, and two distinct elements can have identical vertices. For example, there can be several edges ij with the same endpoints $i, j \in V$, and we may also have $i = j$. Each edge ij is associated with two oriented *halfedges* $\overrightarrow{ij} \neq \overrightarrow{ji} \in H$ (inset). In practice a Δ-complex can be implemented via, *e.g.*, the *halfedge data structure* [Botsch et al. 2010, Chapter 2], or an *edge gluing map* [Sharp and Crane 2020b, Section 4.1]. We use u_σ to denote a quantity u at a vertex, edge, face, or halfedge σ; u_i^{jk} is a quantity u at corner i of triangle ijk.

2.2 Geometry



Edge Lengths. The intrinsic geometry of a triangulation is given by edge lengths $\ell : E \to \mathbb{R}_{>0}$ that satisfy the triangle inequality $\ell_{ki} + \ell_{ij} > \ell_{jk}$ for each triangle corner $\overset{jk}{i}$. Initially, $T^0 = T^1$ and both triangulations have identical edge lengths $\ell_{ij} = |f_j - f_i|$, where $f : V \to \mathbb{R}^3$ are the input vertex positions. The edge lengths of T^1 will later be modified by operations like intrinsic edge flips (Section 3.2). Even then, we can draw individual faces $ijk \in F^1$ as ordinary triangles in the plane, and obtain quantities like corner angles θ_i^{jk} via standard formulas. Finally, we use $\exp_x(u)$ to denote the exponential map at x, which gives the point reached by walking in the tangent direction u for a distance $|u|$ (see [Sharp et al. 2021, Section 2.4.2] for implementation details).

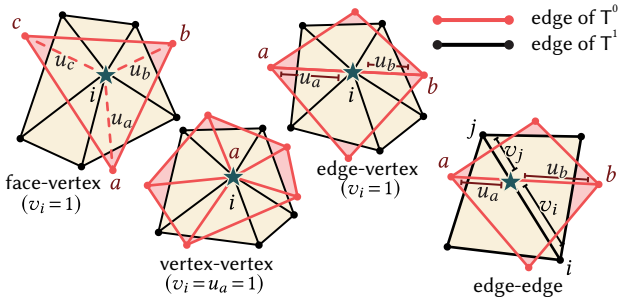


Fig. 4. Points are encoded relative to both triangulations T^0 and T^1. For each triangulation we store the simplex containing the point, and the barycentric coordinates within that simplex.

Barycentric Coordinates. We encode points $x \in M$ by barycentric coordinates relative to a vertex, edge, or triangle, *e.g.*, three positive values $u_i + u_j + u_k = 1$ for a triangle ijk (and for a vertex i we have a single coordinate $u_i = 1$). Importantly, for each point x we store barycentric coordinates u and v relative to both T^0 *and* T^1, *resp.* (Figure 4). We use q^0 and q^1 to denote a barycentric coordinate together with its associated simplex, on either T^0 or T^1.

2.3 Normal Coordinates

We use normal coordinates (Figure 3) to encode how two triangulations T^0, T^1 cross each-other. Classically this data can be stored on either triangulation, *i.e.*, one can count the number of times each edge of T^1 is crossed by T^0 or vice versa. But to insert new vertices in T^1 we must store normal coordinates on edges of T^1—otherwise the encoding becomes ambiguous. We hence think of each edge of T^0 as a geodesic curve crossing edges of T^1 (Figure 5, *right*).

More explicitly, the value $n_{ij} \in \mathbb{Z}$ indicates how many times each edge $ij \in E^1$ is crossed transversally by edges of E^0; if an edge of T^0 runs along ij, we let $n_{ij} = -1$. In this latter case, no edge of T^0 can cross ij. The value $n_{ij}^+ := \max(n_{ij}, 0)$ hence gives the number of transversal crossings; the value $n_{ij}^- := -\min(n_{ij}, 0)$ is 1 on shared edges and 0 otherwise. (Note that Gillespie et al. [2021] adopt a different convention, where $n_{ij} = 0$ for parallel edges.)

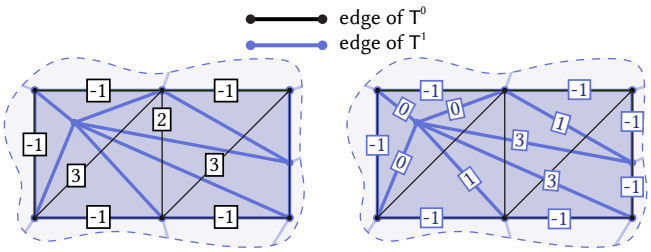


Fig. 5. The number of crossings can be encoded on either triangulation. However, storing values on the edges of T^0 does not account for vertices inserted in T^1 *(left)*; we hence store normal coordinates on T^1 *(right)*.

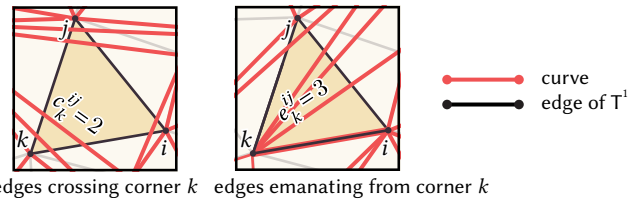


Fig. 6. The value c_i^{jk} gives the number of curves crossing corner $^{jk}_{i}$, while e_i^{jk} gives the number of curves emanating from corner $^{jk}_{i}$.

Using the values n_{ij}^+ we can count how many curves cross and emanate from each corner of T^1, *resp.* (Figure 6):

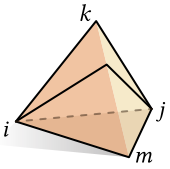
$$c_k^{ij} := \tfrac{1}{2}\left(\max\left(0, n_{jk}^+ + n_{ki}^+ - n_{ij}^+\right) - e_i^{jk} - e_j^{ki}\right). \tag{1}$$

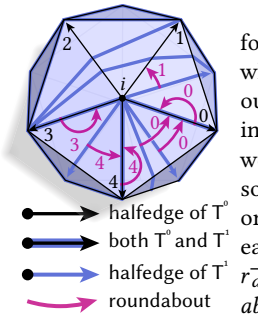
$$e_k^{ij} := \max(0, n_{ij}^+ - n_{jk}^+ - n_{ki}^+), \tag{2}$$

(For curves that do not touch vertices, the corner coordinates c are essentially dual to the normal coordinates n—see [Erickson and Nayyeri 2013, Section 2.3].)

2.4 Roundabouts

Normal coordinates allow us to trace the edges of T^0 as geodesic curves along T^1 (Section 3.1). However, they do not uniquely determine the correspondence between traced curves and logical edges $ab \in E^0$. For instance, flipping edge mk of a tetrahedron results in two distinct intrinsic edges between vertices i and j (inset). Given a curve between vertices i and j on the tetrahedron, it is not *a priori* obvious which of the two logical edges the curve corresponds to.





Gillespie et al. [2021, Section 5.2] therefore introduce *roundabouts* $r : H^1 \to \mathbb{Z}_{\geq 0}$, which for each vertex describe how the outgoing edges from both T^0 and T^1 are interleaved (inset). In particular, suppose we enumerate the halfedges of T^0 around some vertex $a \in V^0$ in counter-clockwise order, starting at any halfedge. Then for each halfedge $\overrightarrow{aj} \in H^1$, the roundabout $r_{\overrightarrow{aj}}$ gives the index of the first halfedge $ab \in H^0$ following $\overrightarrow{aj}$ (which may be $\overrightarrow{aj}$ itself). Unlike Gillespie *et al.* we may insert new vertices—but need only store roundabouts at shared vertices $a \in V^0$, since edges of E^0 always terminate at vertices in V^0.

2.5 Crossings

Finally, we call a point where edges of T^0 and T^1 intersect a *crossing*. A *combinatorial crossing* $\zeta = (\overrightarrow{ij}, p)$ describes a crossing as an ordinal p along a halfedge $\overrightarrow{ij}$—*i.e.*, *which* crossing, as we go from i to j. The same crossing with respect to the opposite halfedge is given by $\overline{\zeta} := (\overrightarrow{ji}, n_{ij} - p - 1)$; this reversal operation is helpful when tracing curves. A *geometric crossing* $z = (\overrightarrow{ij}, p, u, v)$ also gives the barycentric coordinates u, v along the edges from T^0 and T^1, *resp.*

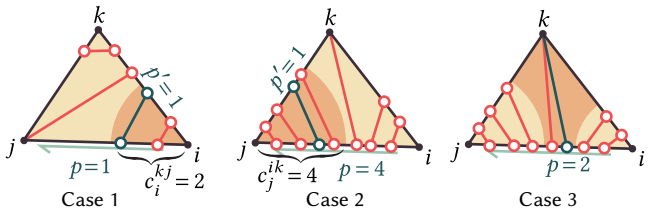


Fig. 7. A curve entering triangle jik along edge ij can proceed in 3 ways: it can exit along edge ik, in which case it is counted by c_i^{kj} (*left*); it can exit along edge kj, in which case it is counted by c_j^{ik} (*center*); or it can terminate at vertex k (*right*).

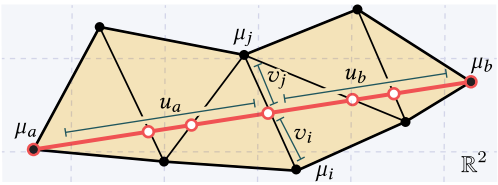


Fig. 8. We compute barycentric coordinates by laying out a triangle strip in the plane.

3 ALGORITHMS

We next describe basic operations on our data structure—detailed pseudocode is given in Appendix B of the supplemental material.

3.1 Extracting Curves

Our first task is to recover a curve on T^1 from its normal coordinates—because our curves are geodesic, normal coordinates imply curve geometry. Explicitly, ExtractCurve (Algorithm 2) takes any combinatorial crossing ζ along a curve γ, and computes the trajectory of γ along T^1 as a sequence $(i, z_1, \ldots, z_k, j)$ of geometric crossings with start and end vertices $i, j \in V^1$. This procedure mirrors Gillespie et al. [2021, Section 6], albeit in a more general setting.

We proceed in two steps, first determining the triangle strip that γ passes through (which depends purely on integers n_{ij}) before computing its geometry (which only then depends on floating point).

The first step calls a subroutine TraceFrom (Algorithm 1), which takes $\zeta = (\overrightarrow{ij}, p)$ 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 (Figure 7). The direction in which to trace is determined by the orientation of $\overrightarrow{ij}$.

ExtractCurve calls TraceFrom once in either direction, obtaining the sequence of 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. Intersecting this line with each of the intermediate edges yields the geometric crossings along γ (Figure 8). Note that the geodesic is guaranteed not to cross the strip boundary—otherwise, the normal coordinates would have encoded such crossings.

Finally, one can convert a single combinatorial crossing into a geometric crossing by calling ExtractCurve and returning the desired crossing. We refer to this procedure as ExtractGeometricCrossing.

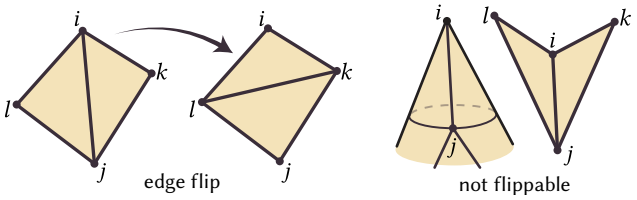


Fig. 9. An edge flip replaces an edge with its opposite diagonal (*left*). An edge ij is not be flippable if it would leave a vertex with degree zero, or if its neighboring faces form a nonconvex quadrilateral (*right*).

3.2 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 9).

Mesh Update. We replace edge ij with an edge kl. We compute the new edge length ℓ^1_{lk} 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:

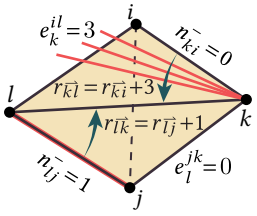
$$n_{kl} = c^{jk}_l + c^{ij}_k + \tfrac{1}{2}\left|c^{il}_j - c^{ki}_j\right| + \tfrac{1}{2}\left|c^{lj}_i - c^{jk}_i\right| - \tfrac{1}{2}e^{ji}_l - \tfrac{1}{2}e^{ij}_k + e^{lj}_i + e^{jk}_i + e^{il}_j + e^{ki}_j + n^-_{ij}. \tag{3}$$

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:

$$r_{\overrightarrow{kl}} = \mod\left(r_{\overrightarrow{ki}} + e^{il}_k + n^-_{ki}, \deg_0(k)\right), \tag{4}$$

where $\deg_0(k)$ is the degree of vertex k in triangulation T^0. The quantity $e^{il}_k + n^-_{ki}$ counts how many edges of T^0 are between $\overrightarrow{ki}$ and $\overrightarrow{kl}$: e^{il}_k counts edges strictly between them, and n^-_{ki} adds one if there is also an edge lying exactly along $\overrightarrow{kl}$. We only perform this update for halfedges whose source is in V^0.



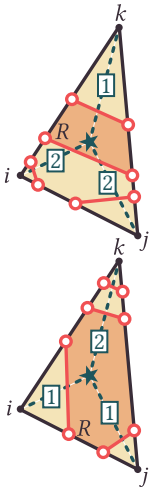
3.3 Face Split

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 ℓ^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 v. Schindler and Chen [2012, Section 3.2] show that the length of a displacement vector δv_i in barycentric coordinates is given by

$$\|\delta v_i\|^2 = -\ell^2_{ij}\delta v_i \delta v_j - \ell^2_{jk}\delta v_j \delta v_k - \ell^2_{ki}\delta v_k \delta v_i. \tag{5}$$



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 (implemented via a simple cross product). We may misclassify points extremely close to a region's boundary due to floating point error, yet even then we insert a valid point in the identified region, at a nearly 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 A for details).

3.4 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 $\overrightarrow{ij} \in H^1$. 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 10, *top*). However, if $n_{ij} < 0$ (which is common in practice—*e.g.* Section 4.2), we perform an explicit edge split to insert the new vertex along the coincident curve (Figure 10, *bottom*).

Note that due to edge splits a single edge $ab \in E^0$ may actually correspond to a sequence of geodesic segments meeting at intermediate vertices $i \in V^\star$. But since each segment terminates at vertices, one can still encode these curves via normal coordinates.

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

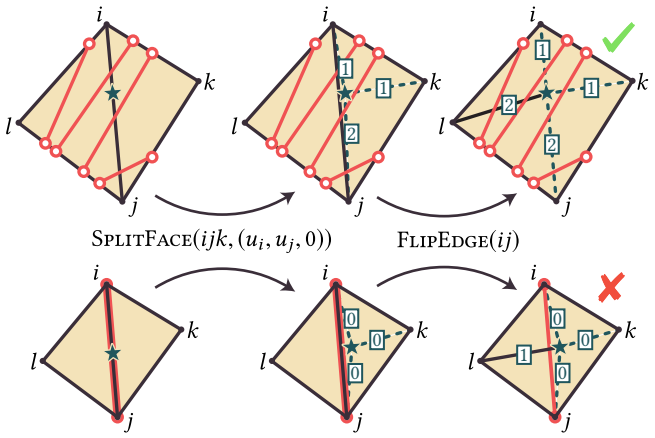
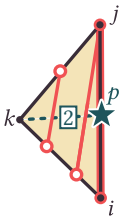


Fig. 10. Generally, one can split edge ij by performing a face split on a neighboring face followed by an edge flip (*top*). However, if ij carries a curve, this strategy will cause the inserted vertex to miss the curve (*bottom*). We hence provide a different edge split procedure for this case in Section 3.4.



Normal Coordinates & Roundabouts. When $n_{ij} < 0$ the new normal coordinates depend only on the old ones. Since $n_{ij} < 0$, 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

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

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.

3.5 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 $i \in V^\star$ 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).

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 do so in the topological setting where the necessary edge flips are always valid—they do not consider the convexity condition (Section 3.2).

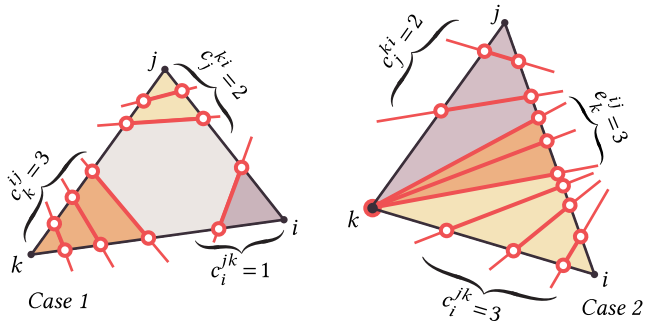


Fig. 11. We extract the connectivity of common subdivision within each triangle using its normal coordinates.

3.6 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].

3.7 Common Subdivision

As noted previously, the *common subdivision*[2] 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.

The common subdivision thus serves as an essential "bridge" between an intrinsic triangulation and the original extrinsic domain: it provides the minimal piecewise linear basis on which both intrinsic data at vertices and extrinsic vertex positions can simultaneously be interpolated. S can then be used to pull back functions from the abstract intrinsic setting to an ordinary mesh sitting in space.

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 4.3), 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,

[2] Also known as the *supermesh* in the FEM literature, *e.g.* Farrell et al. [2009, Section 2].

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. 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. Note that one can construct pathological cases in which there are quadratically many intersections between T^0 and T^1 (or worse—consider a triangulation with many *Dehn twists*, as pictured in [Sharp et al. 2019a, Figure 4]). However, we do not observe such extreme behavior in practice (see Section 7).

Connectivity. We subdivide T^1 independently in each face ijk (Figure 11). There two cases to consider. In case 1, when no curves emanate from any corner, we simply connect the first c_i^{jk} crossings along edge ij to the first c_i^{jk} 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 emanating curves is $e_k^{ij} > 0$. We 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_j^{ki} 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 (Section 2.2). Using these values, one can linearly interpolate data from T^0 or T^1 to the vertices of S. 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 the 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.

3.8 Visualization

To produce figures depicting intrinsic triangulations (*e.g.* Figure 12), we compute the common subdivision (Section 3.7) 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.

3.9 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 manipulating intrinsic triangulations on near-degenerate input meshes is *intrinsic mollification*, introduced by Sharp and Crane [2020b]. Mollification provably ameliorates near-degenerate meshes by adding a small value δ to every edge length, ensuring that every triangle satisfies the triangle inequality with slack at least ϵ. This operation only changes the geometry if some triangle is within ϵ of being degenerate, and even then changes the geometry by a negligible amount. Intrinsic mollification works particularly well with our data structure compared to past approaches: the signpost data structure of Sharp et al. [2019a] relies on tracing edges along the surface, which become *less* accurate when mollification is applied. Integer coordinates have no such problem. In our experiments we mollify with $\epsilon = 10^{-5}h$, where h is the mean edge length, 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 nuanced policy (see Appendix B) which always yields valid normal coordinates.

3.10 Other Operations

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.

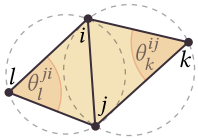
4 RETRIANGULATION AND TRANSFER

Retriangulation. On top of the basic operations from Section 3, one can start to build up the same kind of fundamental building blocks as in extrinsic geometry processing. In this section we focus on *intrinsic Delaunay refinement*, which is a critical tool guaranteeing that a high-quality triangulation can always be built on top of any (*e.g.*, low-quality) input mesh. In particular, we give a new proof about guaranteed quality for surfaces without boundary, and extend the algorithm to surfaces with boundary (without proof).

Transfer. Given a function on a high-quality intrinsic Delaunay mesh, there are then several ways to transfer this function back to an ordinary (extrinsic) mesh. One is to simply copy values at shared vertices, but this approach is often less than satisfactory since the intrinsic and extrinsic basis functions can look quite different. Another is to interpolate it over the common subdivision—made far more robust by our data structure. In this section, we also describe a third, alternative route that finds the best representation of an intrinsic function in a basis on the original mesh (Section 4.3), again made possible by robust extraction of the common subdivision.

Fig. 12. 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*).

4.1 Intrinsic Delaunay Triangulations



One key application of intrinsic triangulations is the computation of *intrinsic Delaunay triangulations* (Figure 12, *top*). A triangulation is said to be Delaunay if the sum of angles opposite every edge is at most π, *i.e.* for every $ij \in E$ we have

$$\theta_k^{ij} + \theta_l^{ji} \leq \pi. \tag{7}$$

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

$$\cot \theta_k^{ij} + \cot \theta_l^{ji} \geq 0. \tag{8}$$

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 13, *left* [Bobenko and Springborn 2007, Proposition 10].

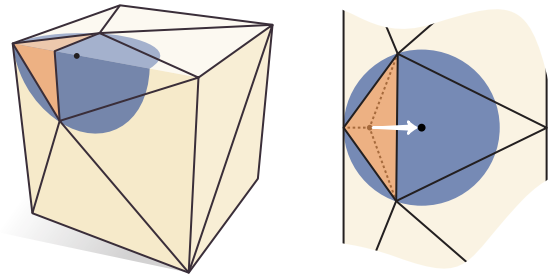


Fig. 13. Triangles in Delaunay meshes have empty circumdisks, and thus well-defined circumcenters (*left*). When necessary, we locate a triangle's circumcenter by walking outwards from its barycenter (*right*).

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].

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 12, *bottom*). Here we modify *Chew's second algorithm* to perform intrinsic Delaunay refinement [Chew 1993; Shewchuk 1997]. This problem has been extensively studied in the plane, but an 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 13, *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 13, *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]:

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

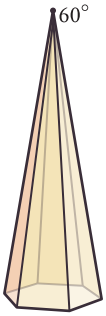
and then normalize to obtain barycentric coordinates

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

To locate the circumcenter on the surface, we then evaluate the exponential map (Section 2.2) starting at the barycenter $w_i = w_j = w_k = 1/3$, 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}. While Xia considers only the planar setting, their proof (which is based on triangle strips) applies without modification to intrinsic Delaunay triangulations of surfaces.

Observe also that, as in the planar case, Delaunay refinement only ever removes *previously-inserted* vertices. Hence, as promised in Section 2, the original extrinsic vertex set V^0 is still preserved.

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 Attribute Transfer

Intrinsic triangulations can drastically improve the quality of solutions to PDEs on low-quality meshes, as will be explored in Section 5. In practice, however, 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. This strategy has been widely explored in the FEM literature (*e.g.* Jiao and Heath [2004]) in the context of extrinsic triangulations. With our reliable common subdivision we can now apply this technique in the intrinsic setting. Here, we restrict 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 f on the intrinsic triangulation, we seek $\hat{f}$ on the original mesh that minimizes the squared L^2 distance

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

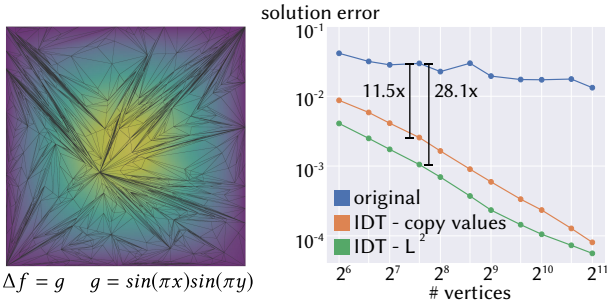


Fig. 14. Accuracy is improved by transferring PDE solutions back to an original triangulation as the L^2-nearest solution, evaluated via the common subdivision. Here we generate random low-quality meshes of the unit square by random edge splits (*left*), and plot the error in the solution of a Poisson 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.

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 matrx M as

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

where M is constructed as in [Strang and Fix 2008, Chapter 10, (32)]. However, in the intrinsic setting f and $\hat{f}$ are encoded over *different* triangulations; they are members of different function spaces. The key observation is that the common subdivision S (Section 3.7) provides exactly the structure needed to evaluate $\|f - \hat{f}\|^2_{L^2}$, as both functions are linear on each triangle of S. In fact, we have

$$\|f - \hat{f}\|^2_{L^2} = (P_1 f - P_0 \hat{f})^T \mathsf{M}_S (P_1 f - P_0 \hat{f}), \tag{13}$$

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 the following positive-definite linear system:

$$\mathsf{M}_0 \hat{f} = P_0^T \mathsf{M}_S P_1 f. \tag{14}$$

Here M_0 is the Galerkin mass matrix of T^0, and 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 14, we show how this transfer indeed improves the accuracy of PDE solutions as measured on the original low-quality mesh. This machinery is enabled because our integer coordinates efficiently and robustly compute the common subdivision. More broadly, this paradigm

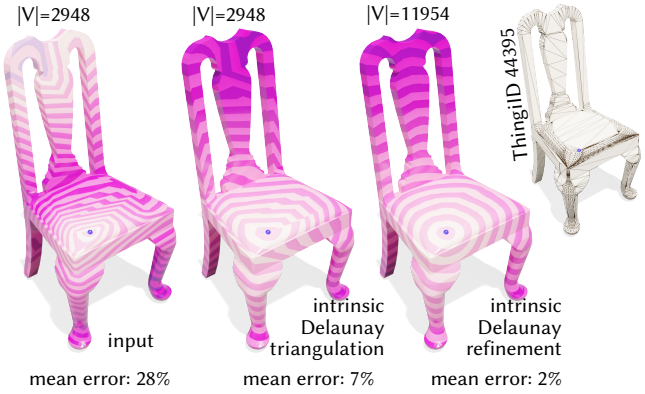


Fig. 15. Running PDE-based algorithms such as the heat method on poor triangulations (*left*) can lead to inaccurate solutions. Flipping to intrinsic Delaunay (*center*) and performing Delaunay refinement (*right*) can drastically improve the results.

opens the door to a wide variety of future finite-element formulations involving intrinsic triangulations.

5 APPLICATIONS

The applications of intrinsic triangulations to robust geometry processing have been extensively explored in past work, including parameterization, surface embedding, distance computation, spectral filtering, surface editing, *etc.* [Fisher et al. 2006; Bobenko and Izmestiev 2008; Sun et al. 2015; Sharp et al. 2019a; Sharp and Crane 2020b,a; Fumero et al. 2020; Gillespie et al. 2021]. The particular importance of our new data structure is not that it enables new applications on the intrinsic mesh—in fact, to merely execute most methods, one needs only the basic intrinsic connectivity and edge lengths. Rather, the point is that we can now provide a valuable guarantee of robustness—namely that the connectivity of the common subdivision is always properly recovered—and hence we can exactly represent intrinsic solutions on the extrinsic surface (Section 6 provides experimental evaluation on a large dataset). In turn, data computed on a high-quality mesh can reliably be used for downstream tasks (texture mapping, deformation, *etc.*) in the original context. In this section we explore this approach using several established algorithms.

5.1 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 numerical solvers. Many such methods depend only on intrinsic data, and are hence a natural application of intrinsic Delaunay triangulations and refinements. For low-quality input meshes, simply running standard algorithms on an intrinsic triangulation (instead of the original mesh) yields solutions of dramatically higher quality.

In Figures 15 and 16, we show several representative examples of this paradigm: 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]. Unfortunately, the

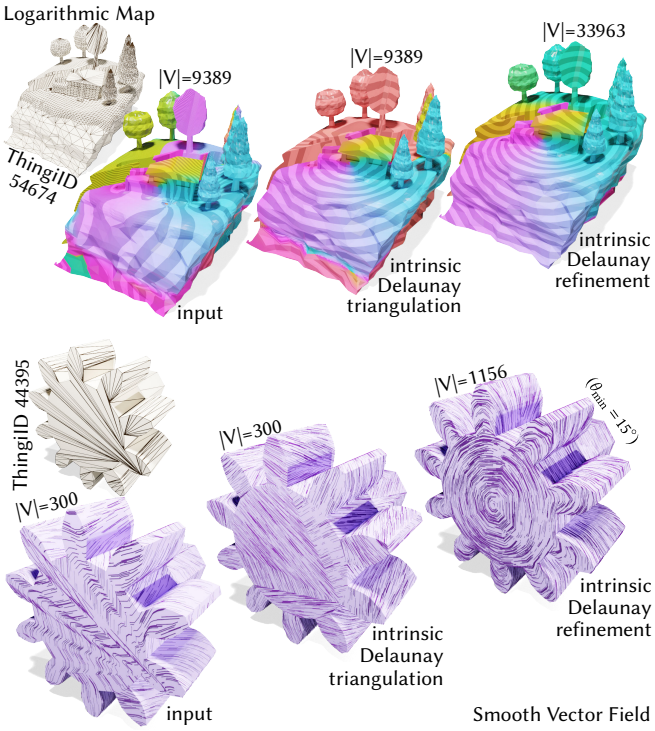


Fig. 16. Here we compute a local parameterization (the *logarithmic map*, *top*), and a smooth vector field (*bottom*) using the connection Laplacian. Both procedures yield inaccurate results on near-degenerate inputs (*left*)—intrinsic Delaunay triangulations (*center*) and intrinsic Delaunay refinements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

high-quality intrinsic solution is not a piecewise-linear function on the original triangulation—and hence cannot be used immediately downstream. Here the common subdivision comes to the rescue: it allows one to represent the intrinsic solution exactly on an ordinary (extrinsic) mesh, for visualization, deformation, or other purposes. Alternatively, we can transfer the solution back to the original basis *à la* Section 4.3, again taking advantage of the common subdivision. Importantly, unlike past schemes, our integer encoding guarantees that this subdivision can be correctly recovered.

5.2 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 [2020a] 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

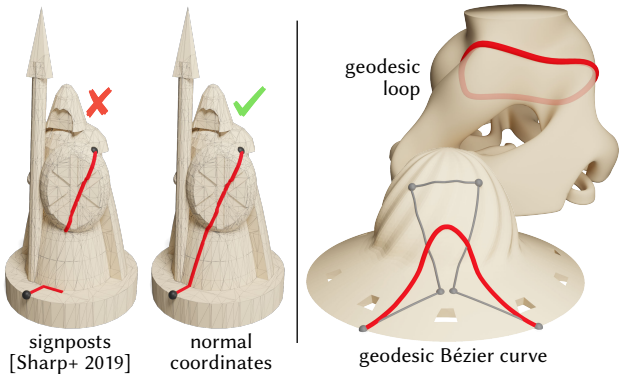


Fig. 17. We construct geodesic paths by flipping edges in a normal coordinate intrinsic triangulation, as in [Sharp and Crane 2020a]. Normal coordinates guarantee a valid path, even on degenerate inputs (*left*). This unlocks advanced applications of geodesics in normal coordinates with the same guarantees, such as geodesic loops and Bézier curves (*right*).

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.

integer coordinates is particularly appealing, because geodesic algorithms are notoriously difficult to implement robustly [Sharp and Crane 2020a, 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 17, *left*). It also enables higher-level tasks involving geodesic paths to be safely run on low-quality input, such 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 17, *right*. Experimentally, we repeat the benchmark of Sharp and Crane [2020a, Section 5.1] and validate that our integer scheme successfully generates a connected polyline along the surface across ∼62,000 trials on the Thingi10k dataset.

6 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

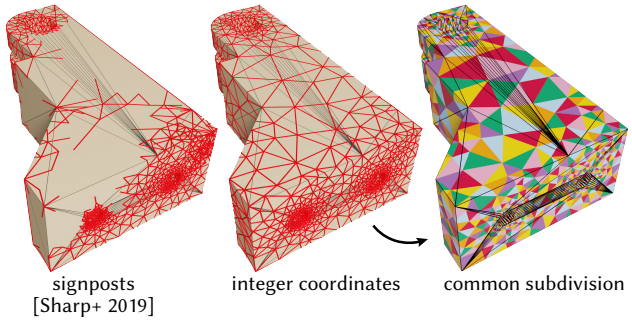


Fig. 18. Past methods extracted edges by tracing "signposts" along the mesh, which may fail in the presence of degenerate triangles. In contrast, our integer coordinates always yield a topologically-valid common subdivision, even on extremely poor quality inputs.

on a single core of an Intel i9-9980XE with 32 GB of RAM. An implementation is provided at https://github.com/MarkGillespie/intrinsic-triangulations-demo.

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 19, *top*) whereas signposts take only 1.5 minutes, but on such meshes signposts generally fail to compute a valid common subdivision. Section 7 discusses hybrid routines which may give the best of both worlds.

6.1 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 3.9). 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 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 19, *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,

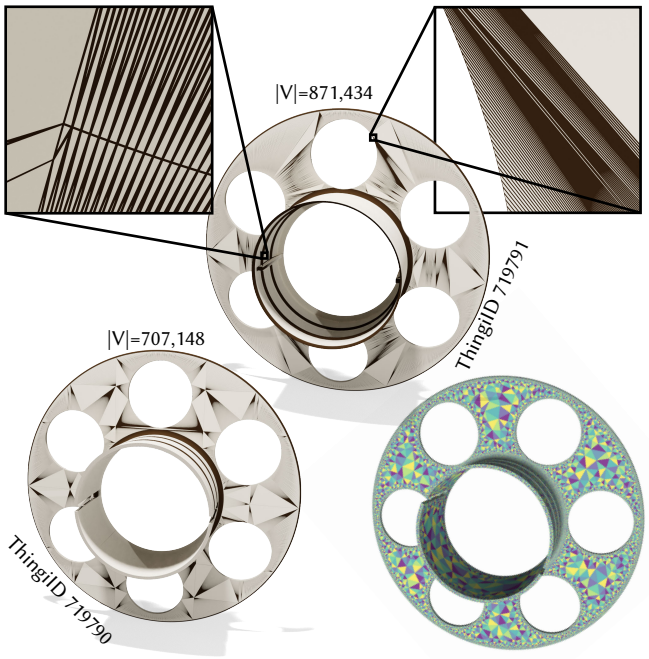


Fig. 19. We fail to compute an explicit mesh of the common subdivision following Delaunay refinement on one Thingi10k model (*top*). Its common subdivision would contain 34 million vertices and our program runs out of memory. We succeed on a nearly identical model (*bottom*), whose common subdivision contains merely 27 million vertices.

but may not successfully recover the common subdivision on degenerate inputs (Figure 18). 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.

7 LIMITATIONS AND FUTURE WORK

Limitations. The common subdivisions that we compute after Delaunay refinement can be quite large: on the difficult Thingi10k dataset, the mean increase in $|V|$ is $20x$, and the 95^{th} 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 95^{th} percentile increase of $7.8x$. However, for applications that rely on the common subdivision (*e.g.* Section 4.3), 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.

More broadly, we inherit the usual tradeoffs of the intrinsic paradigm. There is no clear application to some extrinsic problems (such as bending energies), and algorithms applied in the intrinsic setting must be adapted to take edge lengths as input (though this translation is generally straightforward). As with past work on intrinsic triangulations, our data structure always exactly preserves the input geometry. This imposes a strong restriction that $V^0 \subseteq V^1$ in general position, and furthermore precludes any mesh repair-style operations that might fix spurious holes or handles in the input. Relaxing these assumptions is an important area of ongoing work.

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 conjecture that 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, avoiding the quadratic costs that may arise when explicitly tracking edge crossings. Nonetheless, flipping and insertion remain fast, and connectivity is exact, if one accepts 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.*.

8 ACKNOWLEDGMENTS

Thanks to Boris Springborn and Dylan Thurston for valuable conversations, and to Derek Liu for pointing us to the existing FEM literature on supermeshes. 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). https://pypi.python.org/pypi/flipper.

Mark Bell. 2015. *Recognising mapping classes.* Ph.D. Dissertation. University of Warwick.

Alexander I Bobenko and Ivan Izmestiev. 2008. Alexandrov's theorem, weighted Delaunay triangulations, and mixed volumes. In *Annales de l'institut Fourier*, Vol. 58. 447–505.

Alexander I Bobenko and Boris A Springborn. 2007. A discrete Laplace–Beltrami operator for simplicial surfaces. *Discrete & Computational Geometry* 38, 4 (2007), 740–756.

Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy. 2010. *Polygon Mesh Processing.* CRC press.

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 Delauny 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 Amir 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*. Princeton University Press.
Patrick E Farrell, Matthew D Piggott, Christopher C Pain, Gerard J Gorman, and Cian R Wilson. 2009. Conservative interpolation between unstructured meshes via supermesh construction. *Computer methods in applied mechanics and engineering* 198, 33-36 (2009), 2632–2642.
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.
Marco Fumero, Michael Möller, and Emanuele Rodolà. 2020. Nonlinear spectral geometry processing via the tv transform. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 1–16.
Mark Gillespie, Boris Springborn, and Keenan Crane. 2021. Discrete Conformal Equivalence of Polyhedral Surfaces. *ACM Trans. Graph.* 40, 4 (2021).
Wolfgang Haken. 1961. Theorie Der Normalflächen: Ein Isotopiekriterium Für Den Kreisknoten. *Acta Math.* 105, 3-4 (1961).
Joel Hass and Maria Trnkova. 2020. Approximating Isosurfaces by Guaranteed-quality Triangular Meshes. *Computer Graphics Forum* (2020).
Allen Hatcher. 2002. *Algebraic Topology*. Cambridge University Press.
Yixin Hu, Teseo Schneider, Bolun Wang, Denis Zorin, and Daniele Panozzo. 2020. Fast tetrahedral meshing in the wild. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 117–1.
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 Clémençon. 2001. Voronoi diagrams on piecewise flat surfaces and an application to biological growth. *Theoretical Computer Science* 263, 1-2 (2001), 263–274.
Zhongshi Jiang, Teseo Schneider, Denis Zorin, and Daniele Panozzo. 2020. Bijective projection in a shell. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 1–18.
Xiangmin Jiao and Michael T Heath. 2004. Common-refinement-based data transfer between non-matching meshes in multiphysics simulations. *Internat. J. Numer. Methods Engrg.* 61, 14 (2004), 2402–2427.
Hellmuth 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).
Dimas Martínez Morera, Paulo Cezar Carvalho, and Luiz Velho. 2008. Modeling on triangulations with geodesic curves. *The Visual Computer* 24, 12 (2008), 1025–1037.
Lee Mosher. 1988. Tiling the Projective Foliation Space of a Punctured Surface. *Trans. Amer. Math. Soc.* (1988).
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 Stefankovic. 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.
Teseo Schneider, Yixin Hu, Jérémie Dumas, Xifeng Gao, Daniele Panozzo, and Denis Zorin. 2018. Decoupling simulation accuracy from mesh quality. *ACM transactions on graphics* (2018).
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. You can find geodesic paths in triangle meshes by just flipping edges. *ACM Trans. on Graphics (TOG)* 39, 6 (2020), 1–15.
Nicholas Sharp and Keenan Crane. 2020b. A Laplacian for Nonmanifold Triangle Meshes. *Computer Graphics Forum (SGP)* 39, 5 (2020).
Nicholas Sharp, Mark Gillespie, and Keenan Crane. 2021. Geometry Processing with Intrinsic Triangulations. (2021).
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).
Jian Sun, Tianqi Wu, Xianfeng Gu, and Feng Luo. 2015. Discrete conformal deformation: algorithm and experiments. *SIAM Journal on Imaging Sciences* 8, 3 (2015), 1421–1456.
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 BARYCENTRIC COORDINATES RECOVERY

In Section 3.3, 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 know. 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

$$\begin{pmatrix} v_i^{(0)} & v_i^{(1)} & \cdots & v_i^{(\rho)} \\ v_j^{(0)} & v_j^{(1)} & \cdots & v_j^{(\rho)} \\ v_k^{(0)} & v_k^{(1)} & \cdots & v_k^{(\rho)} \end{pmatrix} \begin{pmatrix} \xi_0 \\ \xi_1 \\ \vdots \\ \xi_\rho \end{pmatrix} = \begin{pmatrix} v_i \\ v_j \\ v_k \end{pmatrix}, \tag{15}$$

and then set

$$u_a := \sum_m u_a^{(m)} \xi_m, \quad u_b := \sum_m u_b^{(m)} \xi_m, \quad u_c := \sum_m u_c^{(m)} \xi_m. \tag{16}$$

Note that while one often seeks a nonnegative ξ, any solution will suffice here: we only use ξ to interpolate in Equation 16.

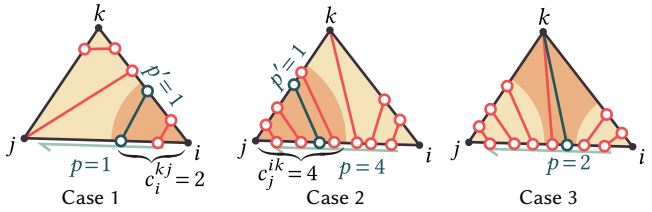


Fig. 20. A curve entering triangle jik along edge ij can proceed in 3 ways: it can exit along edge ik, in which case it is counted by c_i^{kj} (*left*); it can exit along edge kj, in which case it is counted by c_j^{ik} (*center*); or it can terminate at vertex k(*right*). This forms the core of algorithm TraceFrom [Reproduced from Figure 7 for convenience].

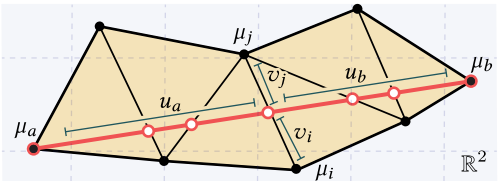


Fig. 21. In procedure ExtractCurve, we compute barycentric coordinates by laying out a triangle strip in the plane [Reproduced from Figure 8 for convenience].

Supplemental Material

This supplement provides additional pseudocode for the procedures described in Section 3 and proofs of correctness.

B 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.

Algorithm 1 TraceFrom(ζ)

Input: Any combinatorial crossing $\zeta = (\overrightarrow{ij}, p)$ along some curve γ lying along T^1.
Output: The half of the curve γ as a sequence of points $(\zeta_0, \zeta_1, \ldots, \zeta_n, k)$ along M, where $\zeta = \zeta_0$ and k is the vertex at which γ terminates.

currentHalfedge $\leftarrow \overrightarrow{ij}$
$\gamma \leftarrow$ [(currentHalfedge, p)]
while True **do** ▷*Walk until the curve terminates at a vertex*
  ▷*Let i and j refer the the tail and tip of the current halfedge*
  $\overrightarrow{ij} \leftarrow$ currentHalfedge
  $k \leftarrow$ OppositeVertex(Twin(currentHalfedge))
  if $p < c_i^{kj}$ **then** ▷*Case 1 of Figure 20 (γ goes right)*
    currentHalfedge $\leftarrow \overrightarrow{ik}$ ▷*Move to $\overrightarrow{ik}$*
    $p \leftarrow p$
    Append(γ, (currentHalfedge, p))
  else if $p \geq n_{ij} - c_j^{ik}$ **then** ▷*Case 2 of Figure 20 (γ goes left)*
    currentHalfedge $\leftarrow \overrightarrow{kj}$ ▷*Move to $\overrightarrow{kj}$*
    $p \leftarrow n_{kj} + p - n_{ij}$
    Append(γ, (currentHalfedge, p))
  else ▷*Case 3 of Figure 20 (γ ends at k)*
    return (γ, k)

Algorithm 2 ExtractCurve(ζ)

Input: Any combinatorial crossing $\zeta = (\overrightarrow{ij}, p)$ along some curve γ
Output: The entire trajectory of γ as a sequence of geometric crossings $(a, z_0, z_1, \ldots, z_k, b)$ along M.

if runs along edge **then**
  return e
else
  $\gamma_{\text{front}}, b \leftarrow$ TraceFrom(ζ) ▷*Trace forwards along γ*
  $\gamma_{\text{back}}, a \leftarrow$ Reverse(TraceFrom($\overline{\zeta}$)) ▷*Trace backwards*
  $\gamma_{\text{Combinatorial}} \leftarrow$ Append($\gamma_{\text{back}}, \gamma_{\text{front}}$)
  ▷*Compute positions in $\mathbb{R}^2$ for the triangle strip containing γ*
  $\mu \leftarrow$ LayOutTriangleStrip($\gamma_{\text{Combinatorial}}$)
  $\gamma_{\text{Geometric}} \leftarrow$ []
  for $\zeta = (\overrightarrow{ij}, p) \in \gamma_{\text{Combinatorial}}$ **do**
    ▷*Find the intersection of ab and ij in the plane (Figure 21)*
    $u, v \leftarrow$ IntersectionBarycentric($\mu_a, \mu_b, \mu_i, \mu_j$)
    $z \leftarrow (\overrightarrow{ij}, p, u, v)$
    Append($\gamma_{\text{Geometric}}, z$)
  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 $(a, z_0, z_1, \ldots, z_l, b)$ along M

$k \leftarrow$ local index of $\overrightarrow{ab}$ about vertex a ▷*Find preceding halfedge*
$\overrightarrow{ai} \leftarrow \text{argmax}_{\overrightarrow{ai}}\{r_{\overrightarrow{ai}} \ : \ r_{\overrightarrow{ai}} \leq k\}$ ▷*Might wrap cyclically*
if $r_{\overrightarrow{ai}} = k$ and $n_{ai} = -1$ **then** ▷*Shared edge*
  return $\overrightarrow{ai}$
else
  $p \leftarrow k - r_{\overrightarrow{ai}} - 1$
  $\gamma \leftarrow$ ExtractCurve(Next($\overrightarrow{ai}$), p)
  ▷*If γ does not end at a vertex of V^0 we must keep tracing*
  while γ does not end at a vertex of V^0 **do**
    $i \leftarrow$ endpoint of γ
    ▷*Our curves only pass through vertices inserted via edge splits. Hence, there is a unique other crossing ζ emanating from j that we must trace along*
    $\zeta \leftarrow$ other crossing emanating from j
    $(i, z_1, \ldots, z_s, j) \leftarrow$ ExtractCurve(ζ)
    ▷*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*
    AdjustBarycentricCoordinates($z_1, \ldots, z_s$)
    Append($\gamma, (z_1, \ldots, z_s, j)$)
  return γ

Algorithm 4 SplitFace_Case1(ijk, u)

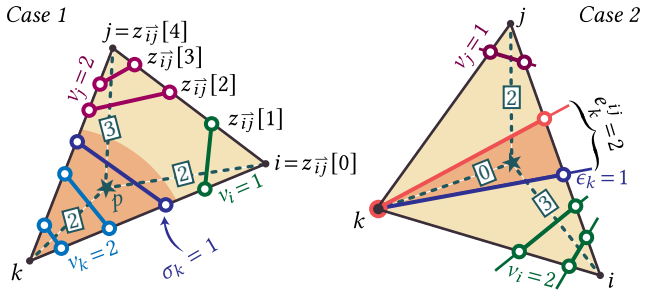


Fig. 22. In SplitFace, we do a sequence of line-side tests to compute a value of v at each corner.

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.

▷ *Gather all geometric crossings*
for $\zeta = (\vec{ij}, p) \in$ CombinatorialCrossings(ijk) **do**
  $z_{\vec{ij}}[p+1] \leftarrow$ ExtractGeometricCrossing(ζ)
▷ *Compute new normal coordinates*
for $c_i^{jk} \in$ CornersOf(ijk) **do** ▷ *for each corner*
  ▷ *Identify which corner curves, if any, contain u*
  $\mathcal{C}_i \leftarrow \left\{\xi : 0 \le \xi < c_i^{jk}, u \in \text{Triangle}\left(i, z_{\vec{ij}}[\xi], z_{\vec{ik}}[\xi]\right)\right\}$
  $v_i \leftarrow \min\left(c_i^{jk} \cup \mathcal{C}_i\right)$ ▷ *Take the closest such corner*
  $\sigma_i \leftarrow c_i^{jk} - v_i$ ▷ *"Slack" left at corner*
▷ *In exact arithmetic, only one σ_i may be nonzero. In floating point multiple could be nonzero, so we keep the biggest and round the others to zero*
if $\sigma_i \ge \sigma_j, \sigma_k$ **then**
  $\sigma_j, \sigma_k \leftarrow 0, 0$
else if $\sigma_j \ge \sigma_k, \sigma_i$ **then**
  $\sigma_k, \sigma_i \leftarrow 0, 0$
else if $\sigma_k \ge \sigma_i, \sigma_j$ **then**
  $\sigma_i, \sigma_j \leftarrow 0, 0$
for $c_i^{jk} \in$ CornersOf(ijk) **do** ▷ *include slack crossings*
  $n_{pi} \leftarrow v_i + \sigma_j + \sigma_k$
▷ *Compute everything else*
$r \leftarrow$ UpdateRoundabouts$(n_{pi}, n_{pj}, n_{pk}, r)$ ▷ *Equation 4*
$\ell^1_{pi}, \ell^1_{pj}, \ell^1_{pk} \leftarrow$ UpdateEdgeLengths(ℓ^1, u) ▷ *Equation 5*
$R \leftarrow$ RegionFromNormalCoordinates(n_{pi}, n_{pj}, n_{pk})
$q_p^0 \leftarrow$ RecoverBarycentric(R, Z, u) ▷ *Appendix A*

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 $n_{ij} \ge n_{jk} + n_{ki}$

Output: An updated integer coordinate intrinsic triangulation.

▷ *Gather all geometric crossings*
for $\zeta = (\vec{ij}, p) \in$ CombinatorialCrossings(ijk) **do**
  $z_{\vec{ij}}[p+1] \leftarrow$ ExtractGeometricCrossing(ζ)
▷ *Compute new normal coordinates*
for $c_i^{jk} \in$ CornersOf(ijk) **do** ▷ *for each corner*
  ▷ *Identify which corner curves, if any, contain u*
  $\mathcal{C}_i \leftarrow \left\{\xi : 0 \le \xi < c_i^{jk}, u \in \text{Triangle}\left(i, z_{\vec{ij}}[\xi], z_{\vec{ik}}[\xi]\right)\right\}$
  $v_i \leftarrow \min\left(c_i^{jk} \cup \mathcal{C}_i\right)$ ▷ *Take the closest such corner*
  $\sigma_i \leftarrow c_i^{jk} - v_i$ ▷ *"Slack" left at corner*
▷ *Note that $v_k = \sigma_k = 0$*
if $\sigma_i \ge \sigma_j$ **then** ▷ *Ensure that at most one σ is nonzero*
  $\sigma_j \leftarrow 0$
else if $\sigma_j \ge \sigma_i$ **then**
  $\sigma_i \leftarrow 0$
▷ *Check for intersections with emanating edges*
if $v_i < c_i^{jk}$ **then** ▷ *In corner i*
  $v_j \leftarrow v_j + e_k^{ij}$
else if $v_j < c_j^{ki}$ **then** ▷ *In corner j*
  $v_i \leftarrow v_j + e_k^{ij}$
else ▷ *In middle of fan region*
  ▷ *Identify which emanating curves, if any, contain u*
  $\mathcal{E}_k \leftarrow \left\{\xi : 0 \le \xi < e_k^{ij}, u \in \text{Triangle}\left(i, z_{\vec{ij}}[\xi + c_i^{jk}], k\right)\right\}$
  $\epsilon_k \leftarrow \min\left(e_k^{ij} \cup \mathcal{E}_k\right)$ ▷ *Take the curve closest to i*
  $v_i \leftarrow v_i + \epsilon_k$ ▷ *Edge pi crosses the first ϵ_k such curves*
  $v_j \leftarrow v_j + e_i^{jk} - \epsilon_k$ ▷ *Edge pj crosses the rest*
for $c_i^{jk} \in$ CornersOf(ijk) **do** ▷ *include opposite corners*
  $n_{pi} \leftarrow v_i + \sigma_j + \sigma_k$
▷ *Compute everything else*
$r \leftarrow$ UpdateRoundabouts$(n_{pi}, n_{pj}, n_{pk}, r)$ ▷ *Equation 4*
$\ell^1_{pi}, \ell^1_{pj}, \ell^1_{pk} \leftarrow$ UpdateEdgeLengths(ℓ^1, u) ▷ *Equation 5*
$R \leftarrow$ RegionFromNormalCoordinates(n_{pi}, n_{pj}, n_{pk})
$q_p^0 \leftarrow$ RecoverBarycentric(R, Z, u) ▷ *Appendix A*

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

if $n_{ij} \ge 0$ **then**
  $k \leftarrow$ OppositeVertex$(\vec{ij})$
  SplitFace$(ijk, (u_i, u_j, 0))$
  FlipEdge(ij)
else
  $k \leftarrow$ OppositeVertex$(\vec{ij})$
  if InInterior(ij) **then**
    $l \leftarrow$ OppositeVertex$(\vec{ji})$
  $T^1 \leftarrow$ insert a vertex p along ij ▷ *Update combinatorics*
  $n_{pj}, n_{pi} \leftarrow n_{ij}, n_{ij}$ ▷ *Compute new normal coordinates*
  $n_{pk} \leftarrow \max(n_{ki}, n_{jk}, 0)$
  ▷ *Compute everything else*
  $r \leftarrow$ UpdateRoundabouts$(n_{pi}, n_{pj}, n_{pk}, n_{pl}, r)$ ▷ *Equation 4*
  $\ell^1_{pi}, \ell^1_{pj}, \ell^1_{pk}, \ell^1_{pl} \leftarrow$ UpdateEdgeLengths(ℓ^1, u) ▷ *Equation 5*
  $q_p^0 \leftarrow u_i q_i^0 + u_j q_j^0$

Algorithm 7 RemoveVertex(i)

Input: An inserted vertex i
Output: Updated triangulation i removed

while i has degree > 3 **do**
  $ij \leftarrow$ flippable edge incident on i
  FlipEdge(ij)
DeleteVertexAndIncidentEdges(i)

Algorithm 8 ComputeCommonSubdivision()

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

▷*Index common subdivision vertices*
$\mathcal{I}^v \leftarrow$ index common subdivision vertices
▷*Compute connectivity*
polygons $\leftarrow []$
for $ijk \in F^1$ **do** ▷*Always orient such that* $n_{ij} \geq n_{jk}, n_{ki}$
  if $e_k^{ij} = 0$ **then**
    Append(polygons, SubdivideFace_Case1($ijk, \mathcal{I}^v$))
  else
    Append(polygons, SubdivideFace_Case2($ijk, \mathcal{I}^v$))
▷*Compute intersection geometry. We denote the locations on* T^0 *by* Q^0 *and the locations on* T^1 *by* Q^1
for $i \in V^1$ **do** ▷*Vertices of* T^1
  $k \leftarrow \mathcal{I}_i^v$ ▷*Index of vertex in S*
  $Q_k^0 \leftarrow q_i^0$ ▷*Location on* T^0 *computed when i was inserted*
  $Q_k^1 \leftarrow (i, 1)$ ▷*Location on* T^1 *is just i itself*
for $ab \in E^0$ **do** ▷*Edge intersections*
  $\gamma = (a, z_1, \ldots, z_l, b) \leftarrow$ ExtractEdge(ab)
  for $z = (\overrightarrow{ij}, p, u, v) \in \gamma$ **do**
    $k \leftarrow \mathcal{I}_{ij}^{\gamma_s}[p+1]$ ▷*Index of crossing in S*
    $Q_k^0 \leftarrow (\overrightarrow{ab}, u)$ ▷*Position on* T^0 *along* $\overrightarrow{ab}$
    $Q_k^1 \leftarrow (\overrightarrow{ij}, v)$ ▷*Position on* T^1 *along* $\overrightarrow{ij}$
return polygons, Q^0, Q^1

Algorithm 9 DelaunayRefinement($\theta_{\min}$)

Input: A minimum allowed angle $\theta_{\min}$.
Output: An intrinsic triangulation T^1 whose corner angles are all at least θ_{min}

FlipToDelaunay()
while T^1 has triangles with angles less than $\theta_{\min}$ **do**
  $ijk \leftarrow$ any triangle with an angle less than $\theta_{\min}$
  ▷*Find the circumcenter via the exponential map*
  $v_c \leftarrow$ circumcenter barycentric coordinates ▷*Equations 9, 10*
  ▷*Barycentric coordinate offset from barycenter to circumcenter*
  $\delta v_c \leftarrow v_c - (1/3, 1/3, 1/3)$
  ▷*Transform offset to face tangent space*
  $V \leftarrow$ BarycentricOffsetToTangentVector(δv_c)
  ▷*Evaluate exponential map from face barycenter*
  $c \leftarrow$ Exp(Barycenter(ijk), V)
  if c lies inside the mesh **then**
    InsertCircumcenter(ijk)
  else
    $lm \leftarrow$ boundary edge separating c from ijk
    $p \leftarrow$ SplitEdge($lm, 0.5$)
    ▷*Must flip to Delaunay before computing Dijkstra ball*
    FlipToDelaunay()
    ▷*Remove inserted vertices from lm's diametral ball*
    ball $= \{i \in V^1 :$ DijkstraDistance(E^1, i, p) $< \ell_{lm}\}$
    for $i \in$ ball **do**
      RemoveVertex(i)
  FlipToDelaunay()

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 removes 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 6.1).

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.

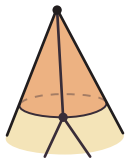
It remains to show that DelaunayRefinement never creates an edge of length less than δ. It is convenient to convert the angle bound α to a *circumradius-to-shortest-edge ratio* bound $B = \frac{1}{2\sin\alpha}$ [Shewchuk 1997, Section 3.1]. Having corner angles at least $\alpha = 30°$,

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.

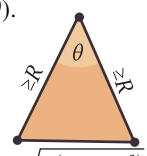
Gluing together the two ends of a self edge yields a loop; we will split into cases based on the homotopy class of this loop on the punctured surface (before the insertion of i). First, note that the loop cannot be contractible to a point, since the original edge is geodesic. Then we will split in to 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 $\cos 60° = \frac{1}{2}$, 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 $\gamma_{\min}$ in the homotopy class is non-constant. By Lemma C.5, we can take $\gamma_{\min}$ to touch some vertex a, and note that since $\gamma_{\min}$ is the *shortest* loop our original self edge must be at least as long as $\gamma_{\min}$. Then by Lemma C.3 a has an edge at least a long as $\gamma_{\min}$, 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. □

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*

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

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 $d_{ij} > 0$. □

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]. □

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. □

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. □

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

D SIMPLICIAL VERTEX REMOVAL

In Section 3.5, 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 3.2). 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 3.2). As always, the convex quadrilateral is defined in the sense of the intrinsic geometry determined by edge lengths. The endpoint degree constraint is automatically

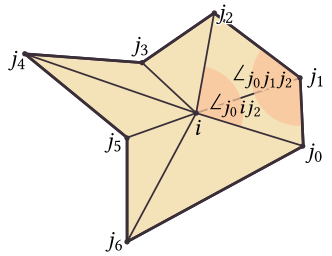


Fig. 23. The relevant angles for Theorem D.1.

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 23). The outer angles $\angle ij_{k-1}j_k$ and $\angle ij_{k+1}k$ are corners of Euclidean triangles, and thus are necessarily at most π, so we need to find an edge ij_k for which the angles $\angle j_{k-1}ij_{k+1}$ and $\angle j_{k+1}j_kj_{k-1}$ are also at most π.

First we consider the inner corners $\angle j_{k-1}ij_{k+1}$. At most two of these angles can be greater than π. To see why, suppose there were three $\angle j_{k-1}ij_{k+1} > \pi$. 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 $\angle j_{k+1}j_kj_{k-1}$ 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. □

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 [2020a, 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.

Received January 20XX