Navigating Intrinsic Triangulations

NICHOLAS SHARP, Carnegie Mellon University
YOUSUF SOLIMAN, Caltech
KEENAN CRANE, Carnegie Mellon University

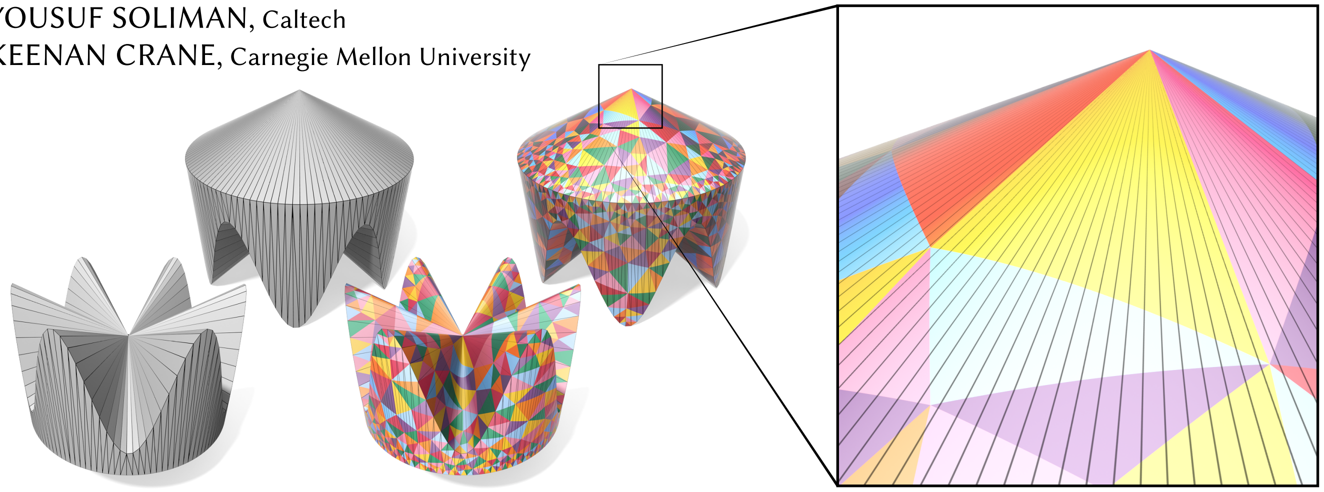


Fig. 1. Our data structure makes it possible to treat a crude input mesh *(left)* as a high-quality *intrinsic triangulation (right)* while exactly preserving the original geometry. Existing algorithms can be run directly on the new triangulation as though it is an ordinary triangle mesh. Here, a mesh with tiny input angles becomes a geometrically identical Delaunay triangulation with angles no smaller than 30°—a feat impossible for traditional, extrinsic remeshing.

We present a data structure that makes it easy to run a large class of algorithms from computational geometry and scientific computing on extremely poor-quality surface meshes. Rather than changing the geometry, as in traditional remeshing, we consider *intrinsic triangulations* which connect vertices by straight paths along the exact geometry of the input mesh. Our key insight is that such a triangulation can be encoded implicitly by storing the direction and distance to neighboring vertices. The resulting *signpost data structure* then allows geometric and topological queries to be made on-demand by tracing paths across the surface. Existing algorithms can be easily translated into the intrinsic setting, since this data structure supports the same basic operations as an ordinary triangle mesh (vertex insertions, edge splits, *etc.*). The output of intrinsic algorithms can then be stored on an ordinary mesh for subsequent use; unlike previous data structures, we use a constant amount of memory and do not need to explicitly construct an *overlay mesh* unless it is specifically requested. Working in the intrinsic setting incurs little computational overhead, yet we can run algorithms on extremely degenerate inputs, including all manifold meshes from the *Thingi10k* data set. To evaluate our data structure we implement several fundamental geometric algorithms including intrinsic versions of Delaunay refinement and optimal Delaunay triangulation, approximation of Steiner trees, adaptive mesh refinement for PDEs, and computation of Poisson equations, geodesic distance, and flip-free tangent vector fields.

Authors' addresses: Nicholas Sharp, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA, 15213; Yousuf Soliman, Caltech; Keenan Crane, Carnegie Mellon University.

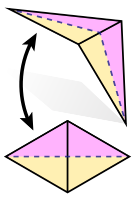
0730-0301/2019/7-ART55 $15.00
https://doi.org/10.1145/3306346.3322979

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

Additional Key Words and Phrases: remeshing, discrete differential geometry

ACM Reference Format:
Nicholas Sharp, Yousuf Soliman, and Keenan Crane. 2019. Navigating Intrinsic Triangulations. *ACM Trans. Graph.* 38, 4, Article 55 (July 2019), 16 pages. https://doi.org/10.1145/3306346.3322979

1 INTRODUCTION



The geometry of a polyhedron has little to do with the way it is triangulated. For instance, flipping a diagonal of a triangulated cube does not change its shape; in general, any two neighboring faces of a triangulation can be laid out flat and connected along the opposite diagonal (see inset). Although the new edge looks bent when drawn on the surface, each triangle is still described by three ordinary edge lengths. Such *intrinsic triangulations* effectively provide "scaffolding" on top of a fixed geometric space: no information about shape is lost by changing the way vertices are connected. However, the choice of triangulation can have significant impact on the behavior of algorithms.

Intrinsic triangulations of geometric spaces have a long history in mathematics, but have seen limited use in practical algorithms: existing data structures support only simple *edge flips*, precluding their use for general geometry processing. Yet a full-blown intrinsic data structure is quite powerful, since it decouples the triangulation used to describe the domain from the one used to implement algorithms on that domain. Hence, rather than trying to make algorithms more robust one at a time, we can immediately run a large class of *existing* algorithms on low-quality inputs, with little to no modification.

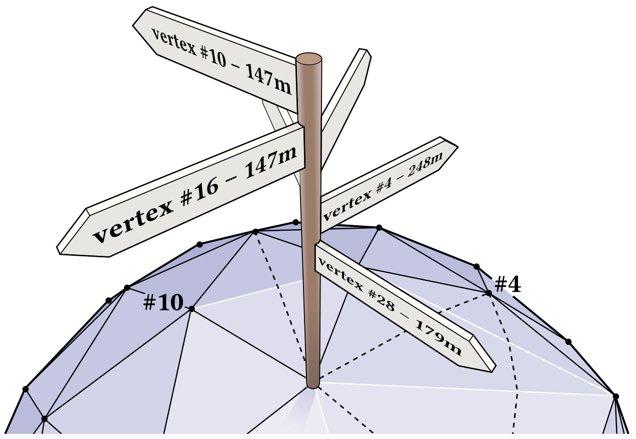


Fig. 2. Our *signpost data structure* stores the direction and distance to each neighbor, making triangulations easy to update and to query on-demand.

The basic idea of our *signpost data structure* is to implicitly encode an intrinsic triangulation by storing the direction and distance from each vertex to its neighbors. Edges can then be traced out on-demand by simply walking along the surface. However, all of this machinery is easily abstracted away: since we support the same standard queries as an ordinary mesh, the fact that an intrinsic triangulation is being used "under the hood" can largely be hidden from a developer of geometric software. This situation is reminiscent of the usage pattern in, say, numerical linear algebra: although libraries may perform sophisticated matrix transformations to improve accuracy or stability, a developer need not think about (or even know about) these transformations in order to express high-level algorithms. Likewise, we seek to abstract away the particular choice of surface tessellation for geometry processing applications.

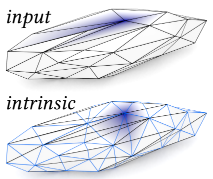


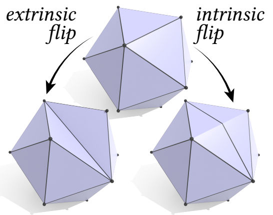
Fig. 3. Linear basis functions on an input vs. intrinsic triangulation.

Two major categories of algorithms naturally fit into the intrinsic framework. First, it enables one to apply standard techniques from Euclidean computational geometry to the polyhedral setting without simultaneously having to worry about geometric approximation: the polyhedron can be viewed as an unchanging "background" domain, just like the Euclidean plane. Second, for finite element methods it allows one to decouple the mesh used to specify the domain from the mesh used to define basis functions, providing the best of both worlds: a concise description of the geometry, together with a small number of high-quality elements (Figure 3).

Our main contribution is a new general-purpose data structure for intrinsic triangulations that enables fundamental geometric algorithms to be implemented in the intrinsic setting for the very first time (Section 3). Most notably, *intrinsic Delaunay refinement* (Section 4.2) provides a way to obtain a high-quality intrinsic triangulation for any input domain (*i.e.*, not merely Delaunay, but also good angles, areas, *etc.*). These tools appear to be the "missing ingredients" that allow intrinsic triangulations to be applied to a much broader range of real-world geometry processing problems.

1.1 Related Work

Geometry Processing in the Wild. There has been significant recent work on robust geometric algorithms for low-quality, real-world inputs. Much of this work depends on a *volumetric* interpretation of geometry [Zhou et al. 2016; Hu et al. 2018; Sellán et al. 2019], using functions defined over the ambient three-dimensional space to achieve robustness [Jacobson et al. 2013; Barill et al. 2018]. Likewise, robust surface triangulation methods from computational geometry are largely based around tetrahedralization of a three-dimensional domain [Boissonnat and Oudot 2005; Cheng et al. 2012]. To our knowledge, ours is the first general approach to manifold *surface* processing "in the wild," which neither changes the input geometry nor constructs an auxiliary volumetric data structure.



Data Structures. Few data structures are available for intrinsic triangulations. Geodesic data structures from computational geometry focus on planar regions [Goodrich and Tamassia 1997], making them unsuitable for 3D geometry processing; other work analyzes sampling criteria, but does not consider practical data structures [Boissonnat et al. 2013]. Work on the discrete Laplace-Beltrami operator of an *intrinsic Delaunay triangulation* [Bobenko and Springborn 2005] inspired development of the first (and to date, only) practical data structure for intrinsic triangulations, the *incremental overlay* of Fisher et al. [2007] which maintains an explicit list of edge crossings. We provide a detailed comparison in Section 6.3, but most importantly this incremental overlay is designed to support *edge flips* and little else—how to implement other operations is not clear, apart from a tracing strategy as developed in this work. Several other algorithms construct intrinsic Delaunay triangulations by tracing geodesics from an intrinsic Voronoi diagram [Xin et al. 2011, 2012; Liu et al. 2017], but do not discuss data structures for subsequent processing. Importantly, although we explore triangulation algorithms in Section 4, we emphasize that our data structure is *not specifically aimed at Delaunay triangulations*; rather, it provides a general framework for any kind of intrinsic geometry processing (Section 5).

Further discussion of related work in the context of specific applications can be found throughout Sections 4 and 5.

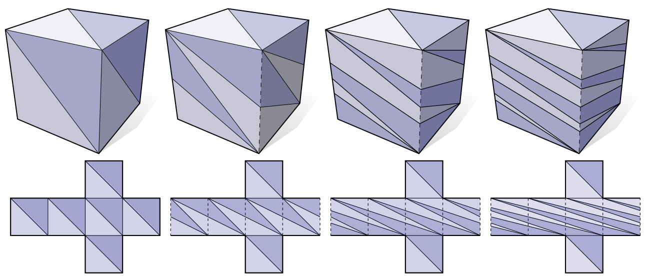


Fig. 4. An intrinsic edge can cross an extrinsic edge many times, as seen with this increasingly "twisted" cube. Rather than track these crossings explicitly, we introduce an implicit encoding that has constant size.

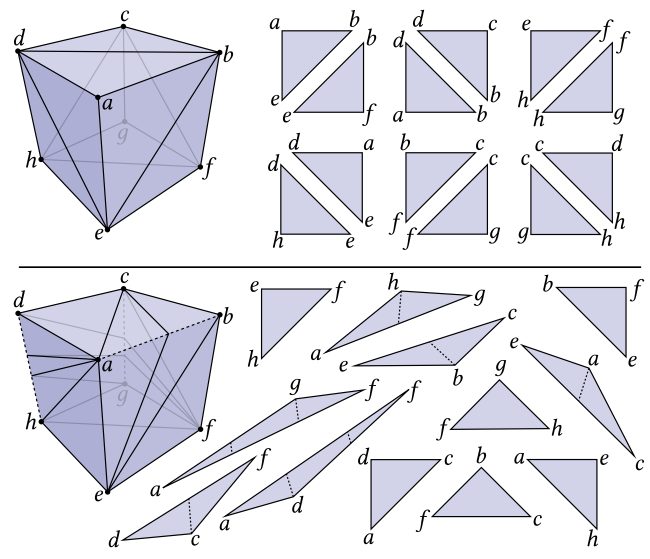


Fig. 5. Unlike an ordinary triangle mesh *(top)*, an intrinsic triangulation *(bottom)* can include edges that take any straight path between vertices. However, each region bounded by three edges can still be unfolded into a single planar triangle.

2 BACKGROUND

The *intrinsic* perspective on geometry means that surfaces are described without reference to the way they sit in space, using only measurements *along* the surface (lengths, areas, *etc.*). Although our initial triangulation typically comes from a standard triangle mesh, we do not require that subsequent triangulations be realizable as collections of planar triangles in $\mathbb{R}^3$—only that each individual face can be drawn as a triangle in the Euclidean plane (Figure 5). We will assume throughout that the underlying domain has manifold connectivity, and that any n-gons have been triangulated (though in principle one could construct a similar representation with intrinsic polygonal faces). We first give an account of intrinsic triangulations *in isolation*; Section 3 describes a practical data structure that can be used to encode an intrinsic triangulation of a given mesh.

2.1 Connectivity

The connectivity of an intrinsic triangulation is given by an abstract triangulation $\mathsf{M} = (\mathsf{V}, \mathsf{E}, \mathsf{F})$ of a polyhedral surface. We denote vertices of M by indices $i \in \mathsf{V}$, edges by pairs $ij \in \mathsf{E}$, and faces by triples $ijk \in \mathsf{F}$; each edge is associated with two oppositely oriented *halfedges* $ij, ji \in \mathsf{H}$. When indices appear on both sides of an equation, all sums and products are restricted to elements containing those indices. For instance, the expression $a_i := \sum_{ij} b_{ij}$ indicates that the value of a at vertex i is equal to the sum of b over all edges ij containing vertex i.

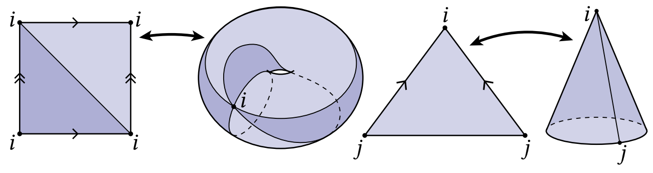
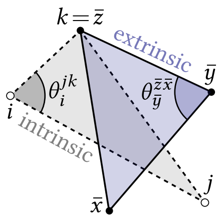


Fig. 6. To provide maximum flexibility, we permit intrinsic triangulations that are not *simplicial*, allowing for instance triangles with repeated vertices *(left)* or self edges *(right)*.

In general, M need not be simplicial: we allow *irregular triangulations* where (for instance) two edges of a triangle are glued together, or all three vertices coincide, as depicted in Figure 6. Formally, M is a Δ*-complex*, as defined by Hatcher [2002, Section 2.1]. This more general construction can represent important objects such as *intrinsic Delaunay triangulations*, yet leads to no additional implementation complexity beyond using an appropriate mesh data structure (see Section 3.1). In practice, irregular vertices almost never occur, except for a small number of extreme examples (*e.g.*, in the *Thingi10k* data set). Moreover, every Δ-complex can be subdivided into a standard simplicial complex [Lundell and Weingram 1969, Theorem 6.1].



Extrinsic vs. Intrinsic Triangulation. Even though the input mesh is not strictly required to be embedded in $\mathbb{R}^n$, for clarity we refer to the initial triangulation as the **extrinsic triangulation** and any subsequent re-triangulation as the **intrinsic triangulation**. We use a bar to denote extrinsic vertices ($\bar{x}, \bar{y}, \bar{z}$, *etc.*) and ordinary letters for intrinsic vertices (i, j, k, *etc.*); note however that intrinsic and extrinsic vertices will often coincide. In figures, we will often use the inset styling to distinguish between intrinsic and extrinsic elements.

2.2 Discrete Metric

The geometry of an intrinsic triangulation is completely determined by a *discrete metric*, *i.e.*, a collection of edge lengths $\ell : \mathsf{E} \to \mathbb{R}_{>0}$ satisfying the triangle inequality $\ell_{ij} + \ell_{jk} > \ell_{ki}$ in each face $ijk \in F$. Other geometric quantities, like the interior angle θ_i^{jk} at corner i of triangle $ijk \in F$, can be determined purely from the edge lengths, as discussed in Section 3.1.1. If a triangulation has vertex coordinates $f : \mathsf{V} \to \mathbb{R}^n$, then initial edge lengths can of course be obtained from the Euclidean distance $\ell_{ij} = |f_j - f_i|$. Importantly, however, subsequent intrinsic edge lengths will *not* generally agree with the Euclidean distance—see Figure 9.

2.3 Tangent Spaces

Tangent vectors on a polyhedral surface can be encoded in local polar coordinates $(r, \varphi) \in \mathbb{R} \times [0, 2\pi)$ relative to an arbitrary (but fixed) reference direction e at each vertex, edge, or face, which corresponds to the direction $\varphi = 0$ (Figure 8). At edges we simply let e_{ij} be the direction along the edge; likewise, at faces we let e_{ijk} be parallel to one of

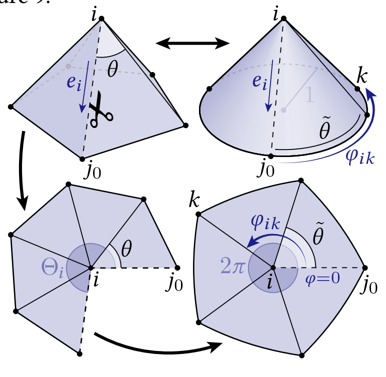


Fig. 7. Intrinsically, the neighborhood of a vertex i looks like a circular cone. The direction of any outgoing edge ik can be expressed via an angle φ_{ik} around this cone.

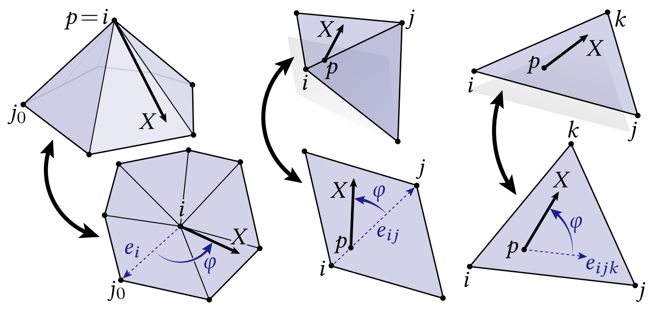


Fig. 8. A tangent vector X at a point p is encoded by an angle φ relative to some fixed reference direction e at each vertex i, edge ij, or face ijk.

the three edges. At vertices, we take advantage of the fact that—*intrinsically*—the local neighborhood looks like a circular cone (Figure 7). Following Knöppel et al. [2015, Section 3.1], we therefore let e_i be the direction from i to some canonical neighbor j_0, and interpret the angle φ as the arc length along the base of the cone. More precisely, let

$$\Theta_i := \textstyle\sum_{ijk} \theta_i^{jk} \tag{1}$$

be the total angle around vertex i, and define augmented angles

$$\tilde{\theta}_i^{jk} := 2\pi\theta_i^{jk}/\Theta_i$$

which (by construction) sum to 2π. All tangent directions at this vertex are then expressed with respect to normalized angles. In particular, if $ij_0, ij_1, \ldots, ij_N$ denote the outgoing edges in counter-clockwise order, then the directions φ_{ij_a} of these edges are given by the cumulative sums

$$\varphi_{ij_a} := \textstyle\sum_{n=0}^{a-1} \tilde{\theta}_i^{j_n, j_{n+1}}, \tag{2}$$

where the index $n+1$ is taken modulo N. Note that these coordinate systems remain fixed even if we modify the intrinsic triangulation (*e.g.*, by flipping the reference edge ij_0).

3 DATA STRUCTURE

Whereas Section 2 defines intrinsic triangulations in isolation, we now describe a data structure that encodes an intrinsic triangulation of a given extrinsic mesh, as well as routines to modify and query the triangulation. To do so we store, at each intrinsic vertex, the distance and direction that one must travel in order to reach each neighboring vertex—we call this construction the *signpost data structure*, in analogy with signs used to mark trajectories to neighboring cities (Figure 2).

Our signpost data structure has at its core just two operations—*signpost updates* and *tracing queries*—on top of which all other operations can be expressed (Section 3.2). From here one can easily implement common geometric and topological primitives such as edge flips and vertex insertions (Section 3.3). We also describe efficient strategies for evaluating correspondence between the intrinsic and extrinsic triangulation (Section 3.4). Here we discuss only generic, low-level operations; we defer discussion of how to actually construct a "good" triangulation to Section 4. Detailed pseudocode for most methods is provided in Appendix B; we also plan on releasing an open source implementation following publication.

3.1 Signpost Data Structure

Our signpost data structure can be built on top of any standard mesh data structure, such as a vertex-face adjacency list; we find it most convenient to use a *halfedge mesh*, which supports irregular triangulations (Section 2.1). Starting with the connectivity $M = (V, E, F)$, our signpost data structure then amounts to two additional pieces of data:

- positive edge lengths $\ell : E \to \mathbb{R}_{>0}$, and
- angles $\varphi : H \to [0, 2\pi)$ for each halfedge.

The lengths ℓ_{ij} describe the shape of each triangle; the angle φ_{ij} gives the direction of the halfedge from vertex i to vertex j, in the local polar coordinate system at vertex i (Section 2.3). For each vertex i inserted into the intrinsic mesh, we also store its location via a pointer to the extrinsic edge or triangle it belongs to, and its (two or three, *resp.*) barycentric coordinates b_i within that element.

3.1.1 Geometric Quantities. Geometric quantities on the intrinsic triangulation can easily be computed from the edge lengths ℓ (rather than the vertex positions f, which would generally yield incorrect results). Expressions for several quantities are given in Figure 9; note that in floating point there are more accurate ways to evaluate such quantities—see for instance [Shewchuk 1999]. The extrinsic location $f_i \in \mathbb{R}^3$ of any intrinsic vertex i can still be obtained via barycentric interpolation of extrinsic vertex coordinates if necessary (*e.g.*, for visualization, or sampling a function on $\mathbb{R}^3$). Other quantities can be deduced, *e.g.*, by isometrically unfolding local neighborhoods into the plane and applying standard Euclidean formulas.

3.2 Atomic Operations

We now describe the atomic local operations on the signpost data structure: a *tracing query*, which follows a signpost to its destination; a *signpost update*, which updates the direction of a single signpost; and a *vertex update*, which combines these two operations to update the signposts around a vertex. As with many data structures, these operations may initially appear abstract, but ultimately allow all other operations to be implemented in a natural way (consider for the *splice* operation in a quad edge mesh [Guibas and Stolfi 1985]). In particular, they will be used to implement low level operations in Sections 3.3 and 3.4, as well as higher-level triangulation algorithms in Section 4.

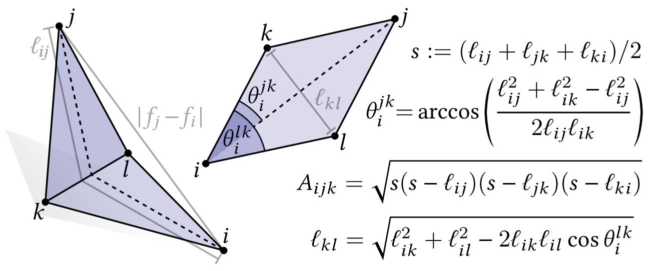
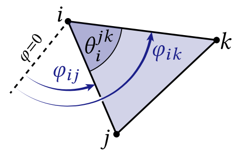

j
ℓ_{ij}
$|f_j - f_i|$
l
k
i
k
j
θ_i^{jk}
θ_i^{lk}
ℓ_{kl}
i
l
$s := (\ell_{ij} + \ell_{jk} + \ell_{ki})/2$
$\theta_i^{jk} = \arccos\left(\frac{\ell_{ij}^2 + \ell_{ik}^2 - \ell_{ij}^2}{2\ell_{ij}\ell_{ik}}\right)$
$A_{ijk} = \sqrt{s(s-\ell_{ij})(s-\ell_{jk})(s-\ell_{ki})}$
$\ell_{kl} = \sqrt{\ell_{ik}^2 + \ell_{il}^2 - 2\ell_{ik}\ell_{il}\cos\theta_i^{lk}}$

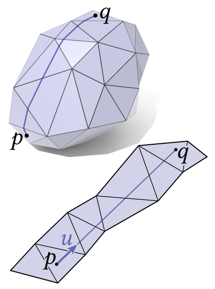
Fig. 9. *Left:* when working with an intrinsic triangulation, the length ℓ_{ij} of an edge will in general be different from the extrinsic distance $|f_j - f_i|$ between its endpoints. *Right:* instead, quantities like triangle areas A_{ijk} and interior angles θ_i^{jk} can be easily computed from the intrinsic edge lengths.



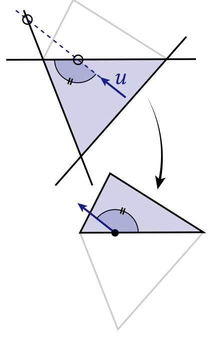
3.2.1 Signpost Update. Maintaining the signpost data structure requires that we be able to update the direction of halfedges from known length and angle information—this update is an atomic operation used in edge flips, vertex insertions, *etc.*, as well as for initializing the signpost data structure itself (see Algorithm 4). Consider in particular a triangle ijk where the angle φ_{ij} and all three edge lengths are already known. A *signpost update* computes the direction of halfedge $ik \in H$ via the relationship

$$\varphi_{ik} = \varphi_{ij} + \tfrac{2\pi}{\Theta_i}\theta_i^{jk}, \tag{3}$$

where the interior angle θ_i^{jk} is computed from the edge lengths, as discussed in Section 3.1.1. In other words, it simply computes the direction of the next edge around the vertex by adding the normalized Euclidean angle (Section 2.3).

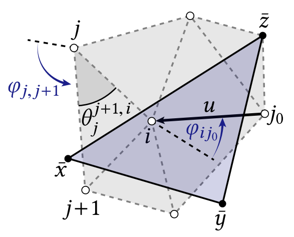


3.2.2 Tracing Query. At any point p, a *tracing query* computes the point q reached by walking a given distance $s > 0$ in a given unit direction u (along either the intrinsic *or* extrinsic mesh). Such queries correspond to an evaluation of the discrete *exponential map*, producing a straightest polyhedral geodesic in the sense of Polthier and Schmies [1998]. Conceptually, this query amounts to little more than unfolding triangles along the path and drawing a straight line (see inset). Since tracing is a purely intrinsic operation, all calculations can be carried out in a local 2D coordinate system for each triangle—we do not need to work in 3D, or explicitly constrain the path to the surface.



Algorithmically, each vertex of the path is found by computing 2D ray-line intersections with the edges of the current triangle, and moving to the closest intersection point. The direction of the ray is then transformed into the coordinate system of the next triangle (by constructing a vector that makes the same angle with the shared edge), and the process is repeated. The final output is the barycentric coordinates of the point q, as well as a pointer to the triangle containing q. In some situations it is also useful to maintain a list of points crossed along the path (given by 1D barycentric coordinates and edge pointers). Discussion of floating point implementation is discussed in Appendix A.

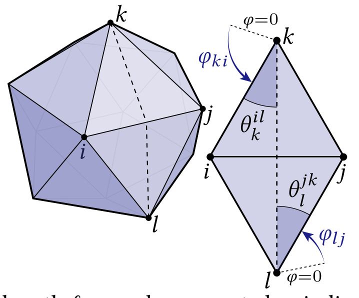
Note that to support the atomic operations of our data structure, we do not need to consider paths *through* vertices (a case which is carefully considered by Polthier and Schmies [1998]). The only subtlety is tracing a vector u that *starts* at a vertex i. Assuming the direction of u is given by a normalized angle $\varphi_u \in [0, 2\pi)$, we first iterate over the neighbors j_n of i until we find the triangle containing u, *i.e.*, until $\varphi_{ij_n} \leq \varphi_u < \varphi_{ij_{n+1}}$. We then initiate a tracing query starting at vertex i of triangle ij_nj_{n+1} in the direction $\Theta_i\varphi_u/2\pi$, *i.e.*, we "un-normalize" the angle so that we can just work in ordinary coordinates. This procedure is described in Algorithm 2.



3.2.3 Vertex Update. Several local operations (Section 3.3) compute new lengths ℓ_{ij} for the edges incident on a single vertex i; the *vertex update* uses these new lengths to update our other quantities. To update the incoming angles φ_{ji} we simply apply a signpost update (Section 3.2.1) to each halfedge ji using the known angles $\varphi_{j,j+1}$ and $\theta_j^{j+1,i}$. To update the outgoing angles φ_{ij} we first establish the direction of an initial edge ij_0 by performing a tracing query from j_0 to i along the direction u, then measure the angle φ_{ij_0} between $-u$ and the reference direction of the extrinsic triangle $\overline{xyz}$ containing the intrinsic vertex i. This tracing query also provides the new barycentric coordinates b_i. To obtain the remaining angles φ_{ij}, we add cumulative sums of interior angles $\theta_i^{j,j+1}$ (as in Equation 2) to the initial angle φ_{ij_0}. Note that to facilitate subsequent tracing queries it is critical to express the outgoing signpost angles φ_{ij} with respect to the canonical coordinate system for the extrinsic triangle $\overline{xyz}$ (as defined in Section 2.3). Otherwise, there is no way to determine how an intrinsic tangent vector u at i gets mapped to an extrinsic tangent vector for subsequent tracing queries (and vice versa).

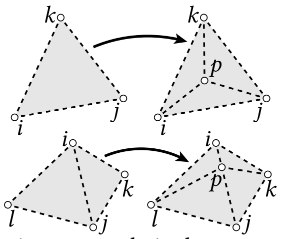
3.3 Local Operations

Local mesh operations such as edge flipping and splitting can easily be implemented using the atomic routines defined in the previous section. This section develops standard operations needed for many applications, including those in Sections 4 and 5. Other common operations (such as *edge insertion* [Bern et al. 1993]) could also be implemented. However, operations that *remove* extrinsic vertices (such as *edge collapses*) do not have an obvious interpretation, since intrinsically it is not clear what the resulting geometry should be; in general, only vertices that were previously inserted can be removed without changing the geometry of the domain.

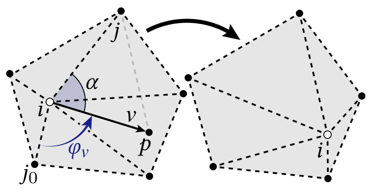


3.3.1 Edge Flip. An *edge flip* replaces a pair of triangles $ijk, jil \in F$ with the triangle pair klj, lki (see inset). To update our signpost data structure, we simply need to compute the length of the new edge, and the angles of the two new halfedges (no barycentric coordinates change during a flip). The new length ℓ_{kl} can be computed as indicated in Figure 9; the angles φ_{lk} and φ_{kl} can be obtained by applying signpost updates (Section 3.2.1) using the known angles $\varphi_{lj}, \theta_l^{jk}$ and $\varphi_{ki}, \theta_k^{il}$, *resp.* This procedure corresponds to the edge flip briefly discussed in Sharp et al. [2019, Section 5.4]. Note that, as with planar triangulations, an edge flip cannot be performed if the two triangles form a nonconvex quadrilateral, or if either endpoint has degree one (as in Figure 6, *right*). Interestingly, neither of these conditions needs to be checked explicitly in the special case of *Delaunay* flips—see Section 4.1 for further discussion.

3.3.2 Vertex Insertion. Given the barycentric coordinates for a point p on the interior of an intrinsic triangle ijk, a *vertex insertion* replaces ijk with three new intrinsic triangles ijp, jkp, kip. The new edge lengths ℓ_{ip}, ℓ_{jp}, and ℓ_{kp} can be obtained by applying the distance formula given in Algorithm 6, and the remaining data is computed via the vertex update procedure (Algorithm 7). Inserting a vertex on the interior of an edge is nearly identical (Algorithm 8).



3.3.3 Vertex Repositioning. Intrinsic vertices i that have been inserted in the triangulation can be moved to a new location p without changing the geometry of the surface. In general, one can simply remove vertex i, triangulate the resulting polygon (which is intrinsically flat), and insert a vertex at p *à la* Section 3.3.2. For smaller motions which only affect the vertex neighborhood of i, it is simpler and more efficient to just update the local lengths and angles. In particular, let v be the vector from i to p; a conservative check for this case is that $|v|$ is smaller than the minimal incident triangle height. The new edge lengths can then be computed as



$$\ell_{pj} = \sqrt{|v|^2 + \ell_{ij}^2 - 2|v|\ell_{ij}\cos\alpha},$$

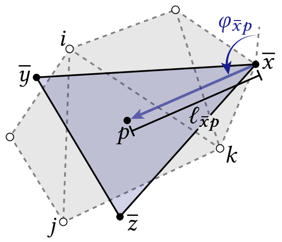
where $\alpha := \varphi_{ij} - \varphi_v$ is the angle between v and the original edge ij; the angles φ of all incident halfedges are then computed via a vertex update (Section 3.2.3). An implementation is given in Algorithm 10—for the algorithms in this paper, we need only this local update.

3.4 Correspondence

There is always a 1:1 correspondence between points on an intrinsic triangulation and the underlying extrinsic triangulation. An attractive feature of the signpost data structure is that we can query this relationship directly, without building an auxiliary structure (Section 3.4.1). This functionality makes our data structure easily compatible with sample-based rendering like ray-tracing, *e.g.*, by interpolating a texture map or evaluating a finite element solution.

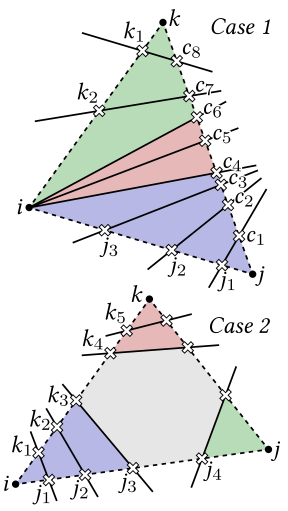
For downstream applications that do require a standard polygon mesh, one can optionally extract an explicit *overlay mesh* (Section 3.4.2), which is the common subdivision of the intrinsic and extrinsic triangulation. Piecewise linear functions on the intrinsic triangulation then become (finer) piecewise linear functions on the overlay mesh, and can hence be used or visualized within a standard geometry pipeline. (Note that, in general, this subdivision will *not* exhibit special properties of the intrinsic mesh—*e.g.*, an intrinsic Delaunay mesh will not have a Delaunay overlay.) Extraction of the overlay need only be performed once, *after* all triangulation operations. In contrast, the data structure of Fisher et al. [2007] must maintain the overlay during each local operation, which can incur significant cost (Section 6.3). Moreover, since our extraction algorithm is local to each triangle, it can easily be executed in parallel (in principle, even in a shader program).

3.4.1 Point Query. Tracing queries can be used to locate a given extrinsic point p on the intrinsic triangulation (and vice versa): if $\overline{xyz}$ is the triangle containing p, one can just trace along the intrinsic triangulation in the direction $\varphi_{\bar{x}p}$ for a distance $\ell_{\bar{x}p}$ (see inset). Any data associated with the intrinsic mesh can then be evaluated using the resulting barycentric coordinates. Pseudocode is given in Algorithm 11. Note that these *point queries* work only because we carefully maintain angles φ with the same meaning on the intrinsic and extrinsic triangulation, *i.e.*, they are always expressed with respect to the same local polar coordinate system.



3.4.2 Overlay Extraction. If an overlay mesh is needed, we first perform a tracing query along each intrinsic edge $ij \in E$, and store a sorted list of barycentric coordinates $t \in [0, 1]$ where ij crosses an extrinsic edge (marked by "×" in the figure below). For each intrinsic triangle ijk, we emit a collection of extrinsic triangles and quads; there are just two cases to consider (illustrated inset below):

(1) All extrinsic edges intersecting ijk cross a common edge jk.
(2) There is no common edge, a sequence of extrinsic edges clips each corner.



Case 1. For an intrinsic triangle ijk, let $c_1, \ldots, c_r$ be the crossings along the common edge jk, let $j_1, \ldots, j_n$ be the crossings along edge ji, and let $k_1, \ldots, k_m$ be the crossings along edge ki. For each crossing along ji, emit a segment connecting j_p and c_p; likewise, for each crossing along ki emit a segment connecting k_p and c_{r-p+1}. Finally, connect all remaining crossings c_p to vertex i.

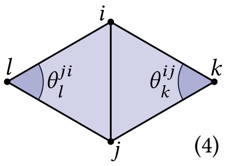
Case 2. For each vertex i of the intrinsic triangle ijk, let $j_1, \ldots, j_n$ be the crossings along edge ij, and let $k_1, \ldots, k_m$ be the crossings along edge ik. We simply need to connect a segment between j_p and k_p until the sum of remaining crossings is less than or equal to the number of crossings on edge jk. This algorithm is then repeated for the other two vertices.

To get polygons, we simply emit faces corresponding to consecutive segments crossing the intrinsic triangle (as well as the final hexagon in Case 2). If we assign a unique index to each crossing, the final output is a standard vertex-face adjacency list description of the overlay mesh (which can be triangulated if necessary). For some applications, it may also be convenient to encode the relationship between the overlay and the intrinsic or extrinsic mesh, *e.g.*, by tagging each overlay triangle with an index to its associated intrinsic face.

4 INTRINSIC RETRIANGULATION

As a key application, we show how different classes of retriangulation algorithms can be easily implemented using the signpost data structure. In contrast to classic *remeshing*, where one must balance triangle quality (angles, areas, *etc.*) with geometric approximation quality (*e.g.*, closeness to the input surface), the intrinsic approach allows one to focus solely on *triangulation* of a given polyhedron, knowing that the underlying geometry will remain unchanged. *Delaunay triangulations* are particularly important since they furnish valuable numerical and structural properties across a wide range of algorithms; other special triangulations (such as those that minimize total length or maximum vertex degree) would also be interesting to consider in the intrinsic setting.

Intrinsic Delaunay Triangulation (iDT). A triangulation exhibits the *intrinsic Delaunay property* if for each edge $ij \in E$ opposite a pair of vertices $k, l \in V$



$$\theta_k^{ij} + \theta_l^{ji} \leq \pi, \tag{4}$$

which for planar triangulations is equivalent to the standard "empty circumcircle" condition. Although other generalizations of the Delaunay condition have been studied for polyhedra (such as *restricted Delaunay* [Cheng et al. 2012, Chapter 13]), the intrinsic Delaunay condition is essential in many applications because it implies positivity of the *cotangent weights*, *i.e.*,

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

This condition in turn guarantees good behavior of a wide variety of algorithms and numerical methods—for instance, it ensures that many standard finite element matrices are *M-matrices*, which facilitates the use of "best in class" numerical solvers [Spielman 2010]. It also ensures that the finite element Laplacian L exhibits a *maximum principle*: a function u satisfying the discrete Laplace equation $\mathsf{L}\mathsf{u} = 0$ will smoothly interpolate fixed boundary data, and has no interior extrema, *i.e.*, no interior vertices that are local maxima or minima [Bobenko and Springborn 2005]. This property is invaluable for numerous applications; with our data structure, it also extends to *vector*-valued functions, as discussed in Section 5.5.

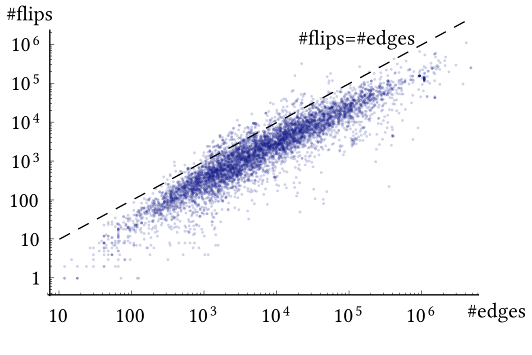


Fig. 10. By running the Delaunay edge flipping algorithm on a large data set we observe a surprising trend: the number of flips required in practice is rarely more than the number of edges in the input mesh. Here each blue dot represents a mesh from the *Thingi10k* dataset.

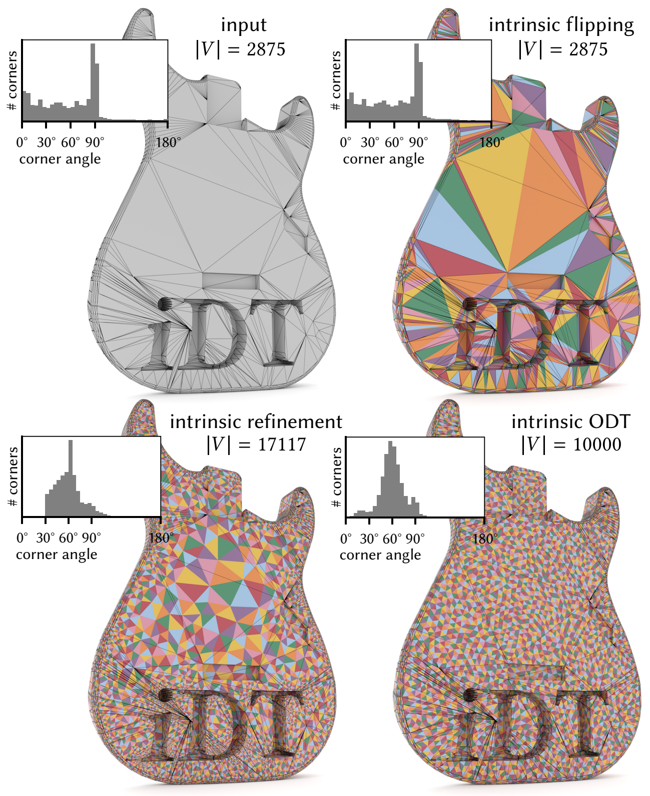
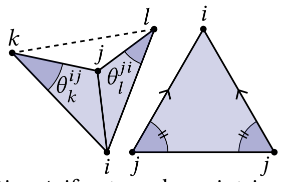


Fig. 11. Intrinsic versions of classic algorithms yield high quality triangulations for any mesh. We can obtain a Delaunay triangulation without increasing the vertex count *(top right)*, achieve lower angle bounds *(bottom left)*, or balance angles and triangle areas *(bottom right)*.

4.1 Delaunay Flipping



An iDT can always be obtained via a simple flipping algorithm [Bobenko and Springborn 2005]: enqueue all edges; while the queue is not empty, check if the next edge ij satisfies Equation 4; if not, apply an intrinsic edge flip and enqueue the four neighbors ik, il, jk, and jl. Note that any edge that needs to be flipped can be flipped: for instance, an edge shared by a nonconvex pair of triangles will have an opposite angle sum less than π; likewise, the two angles opposite an edge with a degree-1 endpoint must belong to the same triangle, and hence cannot sum to greater than π (see inset). For planar triangulations this algorithm requires at most $O(|V|^2)$ flips [Fortune 1993]; for intrinsic triangulations there is no upper bound that depends solely on the number of vertices, since there can be infinitely many geodesics between a given vertex pair (see for example Figure 4). However, the intrinsic algorithm is surprisingly efficient in practice (Figure 10), perhaps because triangulations that come from an embedded surface cannot have extremely long, winding edges. Note that one could also initialize our signpost data structure with an iDT obtained from a geodesic Voronoi diagram [Liu et al. 2017]. However, flipping remains essential for subsequent algorithms which must incrementally improve a triangulation (Sections 4.2 and 4.3).

4.2 Intrinsic Delaunay Refinement (iDR)

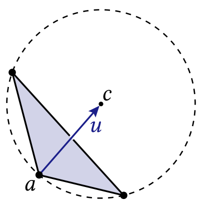
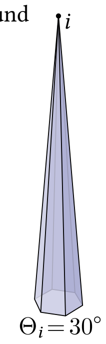


Fig. 13. We locate the intrinsic circumcenter by tracing along a vector u obtained from the planar layout.

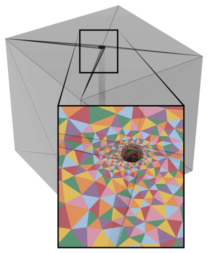
In practice one often cares about criteria beyond the Delaunay property, such as bounds on angles or edge lengths. *Delaunay refinement* progressively inserts vertices to achieve user-specified criteria, while maintaining the Delaunay property. This approach has been studied extensively in the plane, but the intrinsic or *geodesic* case has received little attention—Shewchuk [1997, Section 5.3.2] merely suggests that the basic lemmas needed for Chew's algorithm should also apply in the geodesic setting. Our signpost data structure makes implementation of intrinsic Delaunay refinement straightforward; in particular, we implement an intrinsic version of *Chew's 2nd algorithm* [Chew 1993; Shewchuk 1997]:

- Until the specified minimum angle bound is satisfied:
  - Flip to Delaunay (Section 4.1)
  - Find any triangle ijk that violates the angle bound
  - Insert the circumcenter p of ijk (Section 3.3.2)



In the intrinsic setting we have to decide where to insert the circumcenter p if it is not contained in its triangle—previous literature does not provide a clear answer in this case. If the circumcenter lies outside the triangle ijk then there must be an obtuse angle at a unique vertex a. Our approach is then to layout the triangle in the plane, construct the vector u from a to the planar circumcenter, then trace u along the surface to a point p (see Figure 13). Note also that in the intrinsic setting, the underlying domain may have skinny needle vertices i where the total angle Θ_i is already smaller than the user-specified minimum angle bound (see inset). The only change we make here is that we do not perform refinement when the skinny angle is incident on a degree-1 vertex, and hence cannot possibly be improved.

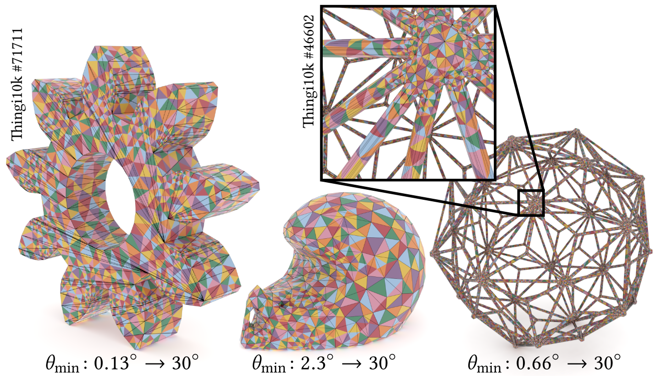
In practice this intrinsic refinement algorithm appears to work quite well, reproducing the behavior of the planar algorithm. For instance, on the *MPZ* dataset (away from only four needle-like vertices) we achieve a minimum interior angle bound of 30° (and hence an upper bound of 120°), which agrees with the guarantee made in the planar case [Chew 1987]. Formally analyzing the behavior of the intrinsic algorithm, and generalizing it to handle boundary or constrained edges is an interesting question for future work.



A typical example from the *MPZ* dataset is shown in Figure 12, *center*; on average the number of inserted vertices was about 60% of the number of input vertices. We also observe the characteristic *grading* behavior of Delaunay refinement that adapts to different feature sizes (see inset). Note that one can also continue refining according to any other quantity of interest, such as triangle areas, or using PDE error estimates as we explore in Section 5.4.

4.3 Intrinsic Optimal Delaunay Triangulation (iODT)

An *optimal Delaunay triangulation* seeks to improve quality not by pure refinement, but also by adjusting the placement of vertices [Chen and Xu 2004]. We apply this idea in the intrinsic setting by optimizing the location of inserted vertices—modifying the input vertices is of course undesirable, since it would change the surface geometry rather than just the triangulation. The basic strategy is to iteratively move all vertices toward the triangle area weighted sum of the circumcenters of incident triangles, then apply the flipping algorithm from Section 4.1 [Chen and Holst 2011, Equation 4.13]. In the intrinsic setting we can locate circumcenters as described in Section 4.2; rather than averaging these locations, we simply average the vectors *to* these locations, then use this average as our update direction. Examples are shown in Figures 11, 14, and 16. Since vertices of the input mesh cannot be moved, we insert new vertices on each iteration by splitting edges longer than a user-defined target length (*à la* Tournois et al. [2009]). In general we observe the same behavior as in the Euclidean case: in contrast to Delaunay refinement, we get a better distribution of areas, at the cost of some skinnier angles.



Thingi10k #71711
Thingi10k #46602
$\theta_{\min}: 0.13° \longrightarrow 30°$
$\theta_{\min}: 2.3° \longrightarrow 30°$
$\theta_{\min}: 0.66° \longrightarrow 30°$

Fig. 12. As in the Euclidean case, we find that intrinsic Delaunay refinement can achieve a minimum angle bound of 30°, even on challenging models.

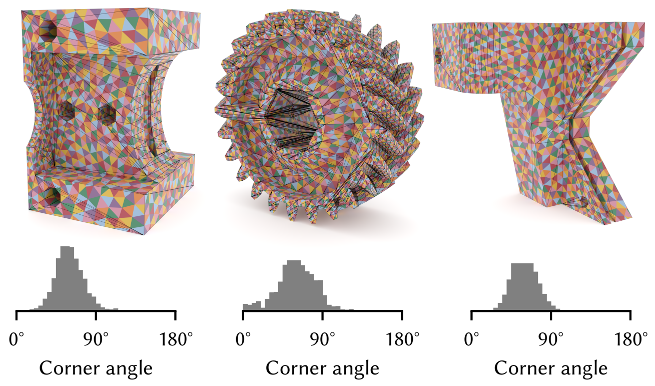
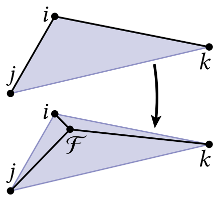


Fig. 14. Our signpost data structure makes it easy to implement an intrinsic version of optimal Delaunay triangulation (iODT), which provides an excellent balance between element size and angle distribution while exactly preserving the input geometry.

5 INTRINSIC GEOMETRY PROCESSING

Our data structure enables a broad range of techniques from computational geometry and scientific computing to be directly applied in the polyhedral setting; it also helps improve the accuracy and reliability of existing surface processing algorithms. In this section we consider several such examples.

5.1 Steiner Tree Approximation



Many applications require a short cut passing through all vertices of a polyhedron, so that it can be isometrically unfolded into the plane. A common approach is to cut through extrinsic edges which approximate the *Steiner tree* of the vertex set [Sheffer and Hart 2002; Erickson and Har-Peled 2004]; for a genus-0 surface the shortest such cut is simply the minimal spanning tree (MST). We can use our signpost data structure to get an even shorter cut, by applying strategies previously only available in the plane. Here, the Delaunay triangulation has the shortest possible MST [Toussaint 1980]; an even shorter tree can be found by inserting additional *Steiner points*. If a triangle ijk has angles all smaller than $2\pi/3$, its local Steiner tree is obtained by inserting the *Fermat point* $\mathcal{F}$, given by barycentric coordinates $b_i := \csc(\theta_i^{jk} + \pi/3)$. Smith et al. [1981] therefore suggest a strategy akin to Kruskal's algorithm: enqueue all edges and per-triangle Steiner trees; greedily add the shortest element from the queue as long as it does not form a cycle. We can directly implement this strategy in the intrinsic setting by flipping to the iDT (*à la* Section 4.1), and using the vertex insertion operation from Section 3.3.2. The resulting cut can be significantly shorter than the extrinsic MST—for instance, by moving the left and right vertices apart in Figure 15, *top*, the length ratio approaches 20%. Even shorter cuts could be obtained by applying more sophisticated techniques from the Euclidean setting (such as [Laarhoven and Ohlmann 2011]).

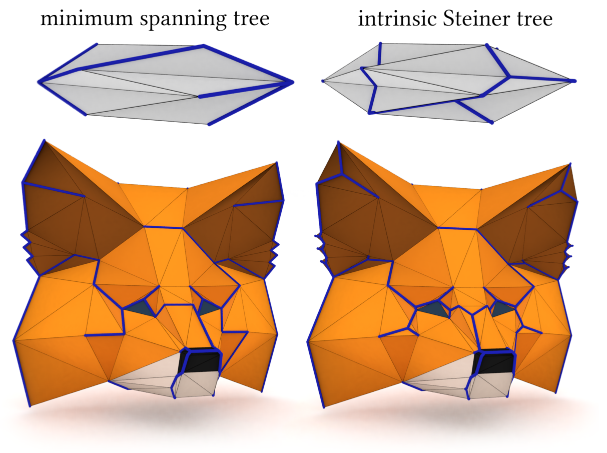


Fig. 15. Our data structure allows planar computational geometry algorithms to be easily translated into the polyhedral setting. Here we apply a classic Euclidean algorithm for *Steiner tree approximation* to find shorter cuts through polyhedral surfaces.

5.2 Finite Elements

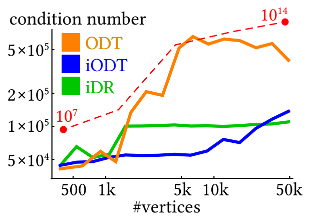


Fig. 17. Condition number for FEM Laplacian with different triangulations; values for regular 4-1 refinement (red) are off the chart.

The numerical behavior of finite element methods depends critically on the quality of basis elements; however, for a given budget of triangles there is a trade off between geometric accuracy and element quality. The intrinsic approach offers the best of both worlds, since the triangulation used to define a finite element basis can be decoupled from the mesh used to describe the domain geometry. Moreover, unlike finite element schemes that use curved elements [de Goes et al. 2016; Feng et al. 2018] or adapt basis functions to accommodate low-quality triangulation [Schneider et al. 2018], an intrinsic triangulation can be used with any existing finite element code that supports ordinary triangular elements, *including* those based on adaptive basis refinement.

To study numerical performance, we solve a standard Poisson equation $\Delta u = f$ on several retriangulations of a coarse input mesh with poor element quality. The stiffness matrix is built exactly as with an ordinary triangle mesh, via the *cotan formula* [MacNeal 1949, Equation 3.19]; we likewise use the standard Galerkin mass matrix. For an equal number of vertices, the intrinsic retriangulation (via either iDR or iODT) yields better condition numbers (Figure 17) than an extrinsic version of ODT, which is more restricted in the vertices it can reposition (and hence the angles it can improve). Intrinsic triangulations also produced significantly lower L^2 error—even for meshes with very little extrinsic curvature (Figure 16). Asymptotically (as edge length goes to zero) intrinsic and extrinsic schemes will behave similarly, but the intrinsic approach provides significant error reduction while still keeping mesh sizes small.

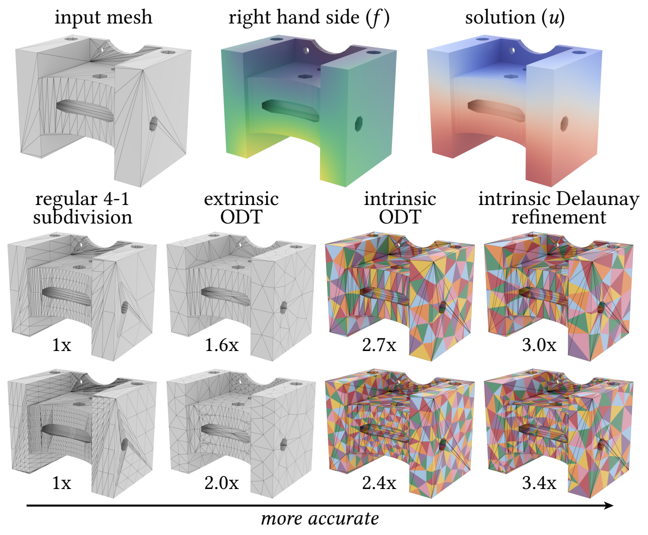


Fig. 16. Even for domains with large, flat regions, solving a Poisson equation is about twice as accurate with an intrinsic triangulation. Rows show two refinements, with matching element counts in the last three columns.

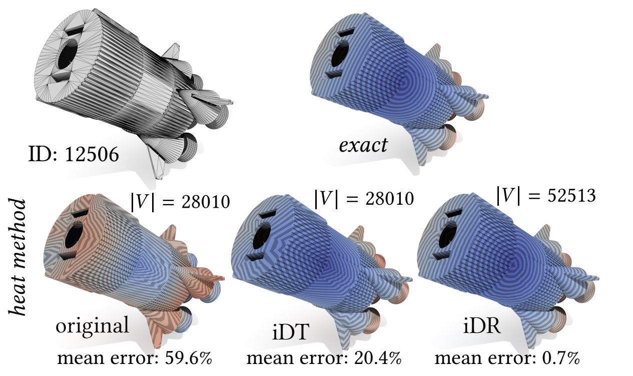


Fig. 18. Working with intrinsic triangulations allows one to accurately and efficiently approximate the exact polyhedral distance using PDE based methods like the heat method.

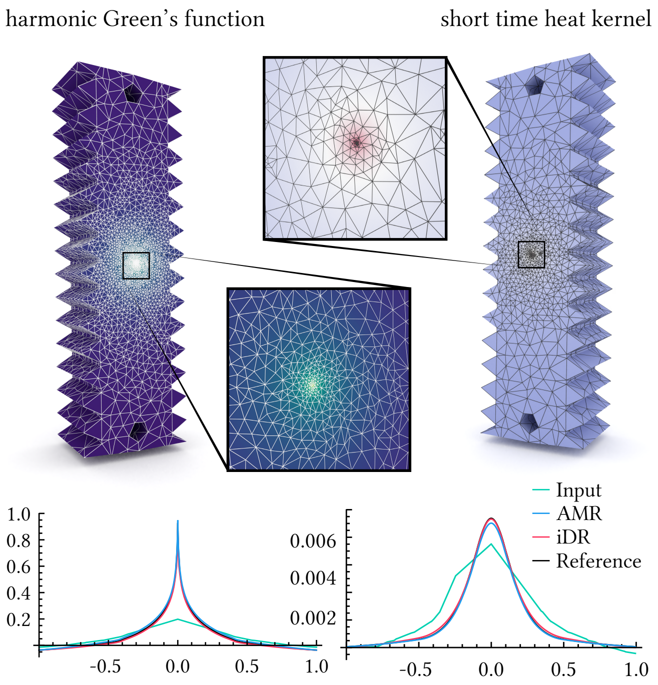
5.3 Geodesic Distance

PDE-based methods for computing geodesic distance such as the *heat method* [Crane et al. 2013] are attractive because they are easy to implement and can take advantage of fast linear solvers; *window-based* methods provide the exact polyhedral distance, but are generally slower and more difficult to implement. We can use an intrinsic triangulation to get the best of both worlds: highly accurate distance via fast PDE-based methods. Figure 18 shows the heat method on several intrinsic triangulations: by just using the iDT the error is dramatically reduced; further refinement with iDR reduces the error to less than 1%.

5.4 Adaptive Mesh Refinement

Many algorithms in geometry processing depend on PDEs where interesting behavior is highly localized in space—for such problems we can use our signpost data structure to adaptively refine the region of interest, rather than uniformly refining the entire surface. Previously, *adaptive mesh refinement (AMR)* on surfaces has been achieved via tetrahedralization of the enclosed volume [Demlow and Olshanskii 2012]; the signpost data structure enables us to apply AMR directly to the surface mesh, while taking advantage of the larger space of intrinsic triangulations. The basic idea is to use a local *a posteriori* error estimate as an additional refinement criterion in iDR (Section 4.2). We apply the basic error estimates described in Morin et al. [2002] and Mekchay and Nochetto [2005] (see supplement for details), though more sophisticated estimators could be used just as easily.

In Figure 19 we compare uniform versus adaptive refinement for computing the *harmonic Green's function* and *short-time heat kernel*, finding that adaptive refinement is 2–10 times faster than uniform refinement (taking all computation into account); such kernels are widely used for shape analysis and distance approximation [Patané 2016]. In Figure 20 we compute a simple surface parameterization via harmonic mapping. Here the iDT ensures injectivity due to the maximum principle (Section 4), though distortion persists near the boundary; our intrinsic AMR scheme provides additional resolution exactly where it is needed. Such techniques could also be applied to more sophisticated mapping algorithms, such as those needed for quadrilateral remeshing [Bommes et al. 2013].



	harmonic Green's function			short time heat kernel		
	$\lvert V\rvert$	time	error	$\lvert V\rvert$	time	error
input	214	0.006 s	0.341	214	0.007 s	0.622
AMR	3029	2.299 s	0.008	1551	0.898 s	0.007
iDR	54916	5.611 s	0.006	81702	8.466 s	0.008

Fig. 19. Intrinsic AMR allows extremely accurate computations of standard kernels from geometry processing with very few elements. *Left:* harmonic Green's function. *Right:* short time heat kernel. Performing uniform iDR to achieve the same accuracy requires 18x and 54x as many vertices, respectively.

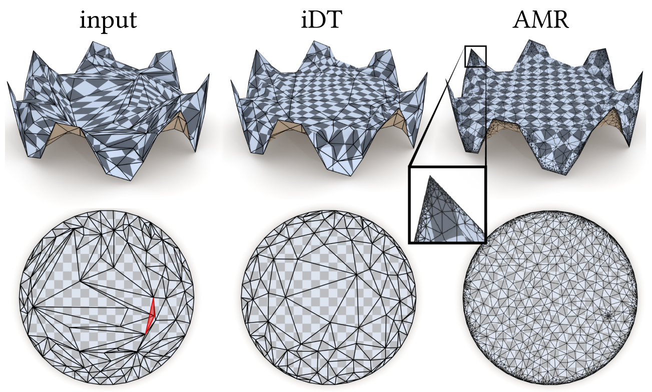


Fig. 20. Mappings computed on low-quality meshes can exhibit flipped triangles, as shown here for a harmonic mapping *(left)*. An iDT guarantees the map is flip-free *(center)*, and AMR intelligently reduces distortion *(right)*.

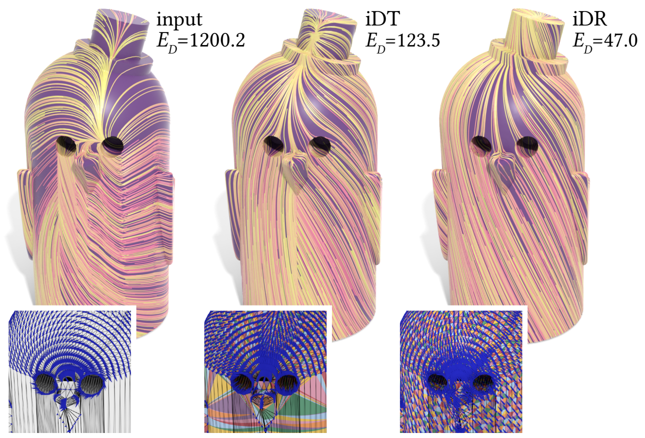


Fig. 21. Our signpost data structure is also well suited for tangent vector field processing. Here we draw streamlines of the vector field with smallest Dirichlet energy E_D (showing vectors in the inset); using an intrinsic triangulation yields a much smoother field.

5.5 Tangent Vector Field Processing

Since our signpost data structure already maintains tangent coordinate systems across all operations, it is naturally suited for problems involving tangent vector fields, as well as other tangent direction fields (*e.g.*, cross fields).

Flip-Free Vector Fields. In a variety of applications one requires a smooth tangent direction field, *e.g.*, for field-aligned quad meshing [Bommes et al. 2013] or for guiding texture synthesis [Knöppel et al. 2015]. Such a field can be obtained by prescribing vectors at the boundary and solving the vector-valued Laplace equation $\Delta^\nabla X = 0$, where Δ^∇ denotes the *connection Laplacian.* Here we can use the finite difference connection Laplacian described in Sharp et al. [2019, Section 5.3]. The intrinsic Delaunay criteria provides a vector-valued analogue of the scalar maximum principle for this discretization: tangent vectors at each point will be contained in the *convex cone* of their immediate neighbors [Sharp et al. 2019, Section 5.4] as seen in Figure 22. The signpost data structure enables this property to be extended to refinements of the input mesh, rather than simply the iDT (which may still provide poor accuracy).

Globally Optimal Direction Fields. We can compute the smoothest direction field on a domain *without* boundary by minimizing the *vector Dirichlet energy*; this amounts to solving an eigenvalue problem via the finite element connection Laplacian described in Knöppel et al. [2013, Sec. 6.1.1]. Figure 21 shows a dramatic increase in smoothness going from the input triangulation, to the iDT, to the iDR. In general, we can find polyhedra where the minimal vector Dirichlet energy is *not* always achieved by the *iDT* (in contrast with *Rippa's theorem* in the scalar case [Rippa 1990]), though in practice it always appears to provide much smoother vector fields. An interesting question, therefore, is whether there is a natural energy for which an iDT always yields the smoothest field.

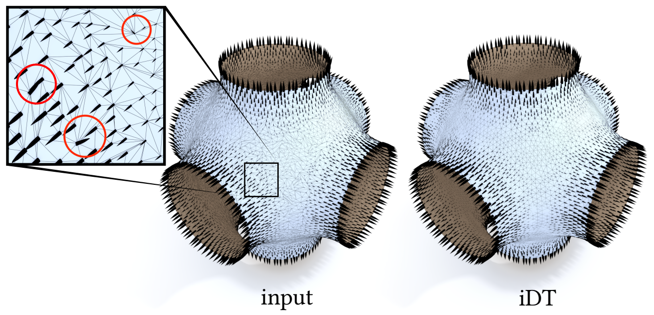


Fig. 22. Intrinsic triangulations that satisfy the Delaunay condition help prevent flips in tangent direction fields. Here we compute the smoothest field interpolating given vectors at the boundary.

6 EVALUATION

We use the applications described in Section 4 and Section 5 to analyze the numerical robustness and runtime performance of our signpost datastructure. With careful treatment of floating point calculations, we found that our data structure is robust enough to operate on all models found "in the wild." Moreover, since the majority of processing involves familiar local mesh operations, there are no big surprises in terms of speed—for instance, running iDR and extracting the overlay on a mesh of about 50k triangles (which makes heavy use of all operations) takes about 0.57 seconds.

6.1 Experimental Setup

Implementation. Signposts can in principle be used to augment a number of different underlying mesh data structures; we found it convenient to use a halfedge mesh, which supports all the necessary topological operations, as well as irregular triangulations. In particular, we implemented our data structure in C++ on top of a standard halfedge mesh library; Appendix A provides details of the numerical implementation. Timings were measured using a single thread of an Intel Core i7 3.5GHz CPU; for quad precision calculations (used only for the robustness benchmarks in Section 6.2), we used the *GCC libquadmath* library [Free Software Foundation 2008].

Data Set. We performed experiments on a wide variety of real-world models from the *Princeton Shape Benchmark* [Shilane et al. 2004] (1800 meshes), the *MPZ* data set from [Myles et al. 2014, Section 8] (117 meshes), and the manifold meshes found in the *Thingi10k* dataset [Zhou and Jacobson 2016] (~8k meshes) which includes many extremely degenerate models (zero-area faces, poorly triangulated CAD models, *etc.*). For *Thingi10k* we used *MeshLab* to convert all files to PLY format [Cignoni et al. 2008], and omitted 10 meshes that either contained invalid data (such as B-splines) or were nonmanifold after conversion. The only preprocessing we perform is to remove a few zero-area triangles by extrinsically flipping or collapsing edges opposite acute angles smaller than 10^{-10} radians or obtuse angles greater than $\pi - 10^{-10}$ radians, *resp.*. This process yields an utterly negligible change in the geometry, and was also necessary for obtaining geometrically meaningful results from the algorithm of Fisher et al. [2007].

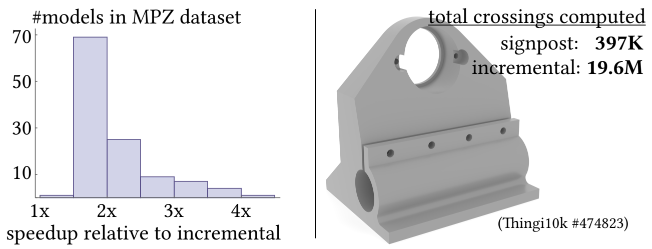


Fig. 23. Even for basic tasks like finding crossings for an intrinsic Delaunay triangulation, our data structure is faster than the incremental overlay of Fisher *et al. (left)*. In extreme cases, the incremental scheme may compute far more crossings than are actually needed for the final overlay *(right)*.

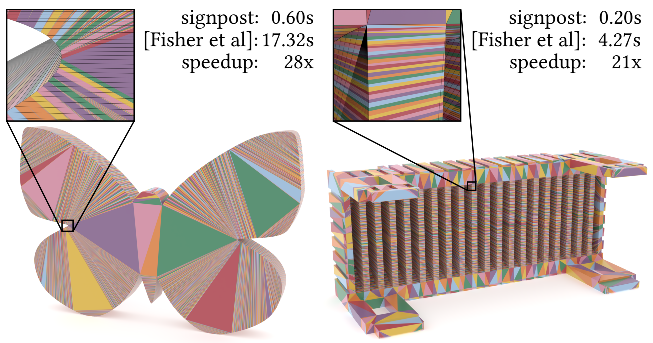


Fig. 24. Flipping to the intrinsic Delaunay triangulation and extracting the edge crossings in the overlay mesh is extremely efficient with the signpost data structure. Prior methods update the overlay mesh every flip, while our approach reads it off after the fact. (*Thingi10k* #79741 & 1432740)

6.2 Robustness

Judicious management of numerical precision is important for any intrinsic triangulation data structure. Note that there can be no robustness issues due to *discretization* error: in exact arithmetic, all calculations yield an exact description of the intrinsic geometry. Therefore, problems arise only due to floating point error, typically on pathological inputs. For instance, the basic edge flipping scheme (which simply keeps track of edge lengths) can produce INF or NaN values due to inaccurate floating point calculation; even with correct length calculations, the incremental overlay [Fisher et al. 2007] can yield meshes that are topologically valid, but where intrinsic edges look nothing like geodesics due to inaccurate calculation of barycentric coordinates (see Figure 27). The main challenge with the signpost data structure is ensuring that tracing queries are topologically valid—see Appendix A for a detailed discussion.

In practice, we found the signpost data structure to be remarkably robust. To examine numerical robustness, we performed the Delaunay flip algorithm (Section 4.1) on the data sets described in Section 6.1, and verified that the overlay extracted *à la* Section 3.4.2 describe the correct combinatorics of the common subdivision. This test succeeds for all 1800 meshes in the Princeton Shape Benchmark and 117 meshes in the *MPZ* dataset. On the more challenging *Thingi10k* dataset, extraction succeeds on 97% of the meshes when using double precision; the remaining 3% can be handled by either randomly perturbing vertex coordinates by 10^{-8} of the bounding box diameter (which yields no appreciable difference in geometry), or by falling back to quad precision (in which case geometry is preserved exactly). Overall, for all practical purposes, our data structure is able to operate on 100% of meshes found "in the wild."

6.3 Comparisons

Incremental Overlay. We compared our signpost data structure to the *incremental overlay* scheme of [Fisher et al. 2007], which represents an intrinsic triangulation by continually maintaining an explicit list of edge crossings. In contrast, the signpost data structure can simply extract crossings once, after all other processing has terminated (Section 3.4.2). Both data structures were implemented on top of the same halfedge data structure, and in both cases we performed extensive profiling and code-level optimization. Further speedups could easily be achieved in our extraction procedure (which is trivially parallelizable across edges), though here we use a serial implementation. Figure 23, left shows the relative performance for computing the edge crossings of an intrinsic Delaunay triangulation on the *MPZ* dataset. (Note that more sophisticated algorithms like iDR cannot be compared since they are not supported by the incremental scheme.) On the most challenging examples in the *Thingi10k* dataset, the signpost data structure was as much as 28x faster (Figure 24). The basic reason for the performance gap is that the cumulative number of crossings computed by the incremental overlay may be dramatically larger than the number of crossings needed for the final triangulation (Figure 23, right).

Extrinsic Delaunay. We also compared to the method of Liu et al. [2015] (using their implementation), which uses extrinsic edge splits and flips of flat edges to achieve the intrinsic Delaunay condition; the benefit is that the output is a standard (extrinsic) triangle mesh. There are however two downsides, namely (i) it may be necessary to generate a large number of elements, and (ii) even though elements are Delaunay, they may otherwise have poor quality (*e.g.*, small angles or areas). The former can be addressed via simplification, at the cost of changing the geometry; however Liu *et al.* propose no approach for improving element quality. In practice, we observe that on models from *Thingi10k*, Liu *et al.*'s algorithm can initially inflate mesh size by a factor of 10–100x (for instance, model #97588). In

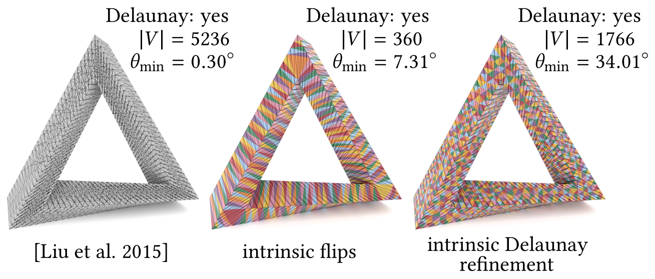


Fig. 25. *Left:* an extrinsic Delaunay triangulation which preserves geometry requires dramatically more vertices. *Center:* the intrinsic Delaunay triangulation requires no additional elements. *Right*: Even a modest number of additional intrinsic vertices can dramatically increase element quality.

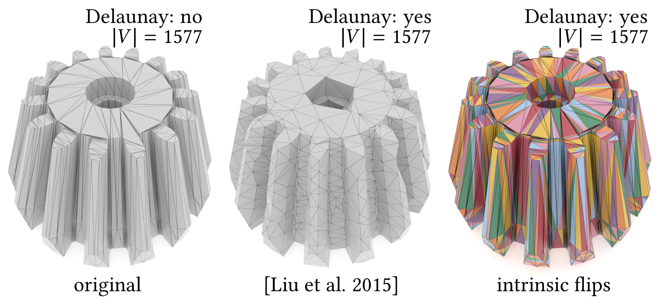


Fig. 26. A mechanical part *(left)* is made Delaunay with extrinsic splits and simplifications *(center)*, or with intrinsic flips *(right)*. For an equal number of elements, the extrinsic method loses important geometric detail, whereas the intrinsic triangulation exactly preserves the given shape.

contrast, the iDT exactly preserves the geometry and has the same number of elements as the input; any subsequent increase in mesh size serves purely to improve triangulation quality. In Figure 25 the coarsest simplification that can be produced via Liu *et al.* without modifying the geometry has about 15x more triangles than the iDT; moreover, iDR can produce a far higher-quality triangulation, using far fewer elements. Figure 26 likewise shows that for the same budget of triangles, the intrinsic approach yields a much better approximation of geometry. A recent method of Yi et al. [2018] uses global nonlinear optimization to obtain extrinsic Delaunay meshes with lower geometric error, albeit at dramatically higher cost—for instance, over 1000x times slower than building and extracting our iDT on the three models in Yi et al. [2018, Table 3]. This method also does not consider element quality (angles, areas, etc).

7 LIMITATIONS AND FUTURE WORK

The essential trade-offs of working with intrinsic triangulations are well-known. At the most basic level, one simply does not have an ordinary triangle mesh, though as we have seen, most algorithms can still be executed as usual. Some basic translation into the intrinsic setting may be needed (*e.g.*, computing areas from edge lengths rather than vertex positions), but these are typically easy to abstract away. Although extracting the overlay mesh yields an extrinsic representation of the triangulation, the overlay mesh is generally not suitable for computation, and will not inherit desirable properties of the intrinsic triangulation (such as the Delaunay property). Nonetheless, many applications do not require the overlay mesh: a typical pipeline might consist of constructing an intrinsic triangulation, solving a PDE, and finally copying the solution back to the original triangulation. In such an application, the overlay mesh might be used only to visualize the intermediate data structure.

The intrinsic operations we present are formulated from the perspective of improving the triangulation of a given domain while preserving that domain exactly, and thus do not address concerns such as denoising, or repairing spurious topological features. Furthermore, we assume throughout that inputs are manifold meshes; an interesting question is how to extend our data structure to general *nonmanifold* meshes, *e.g.*, by augmenting signposts with local topological information.

Since the applications described in Section 4 and Section 5 serve primarily to evaluate our data structure, there are of course many unexplored questions. For instance, in the case of Delaunay refinement we do not consider domains with boundary, though this topic has been extensively studied in the plane. Finally, it would be valuable to consider more generalized notions of intrinsic triangulations, especially those that are not required to include the vertices of the input mesh. In fact, the signpost data structure can already be used as-is to describe embedded graphs with geodesic edges; extending it to perform operations that actually *remove* extrinsic vertices (such as edge collapses) would open the door to yet further algorithms such as intrinsic simplification. On the whole, the ability to easily work with intrinsic triangulations provides fertile soil for new developments in robust geometry processing.

ACKNOWLEDGMENTS

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

REFERENCES

Gavin Barill, Neil Dickson, Ryan Schmidt, David I.W. Levin, and Alec Jacobson. 2018. Fast Winding Numbers for Soups and Clouds. *ACM Transactions on Graphics* (2018).

Marshall W. Bern, Herbert Edelsbrunner, David Eppstein, Scott A. Mitchell, and Tiow Seng Tan. 1993. Edge Insertion for Optimal Triangulations. *Discrete & Computational Geometry* 10 (1993).

A. I. Bobenko and B. A. Springborn. 2005. A discrete Laplace-Beltrami operator for simplicial surfaces. *ArXiv Mathematics e-prints* (March 2005). arXiv:math/0503219

Jean-Daniel Boissonnat, Ramsay Dyer, and Arijit Ghosh. 2013. *Constructing Intrinsic Delaunay Triangulations of Submanifolds.* Research Report RR-8273. INRIA.

Jean-Daniel Boissonnat and Steve Oudot. 2005. Provably good sampling and meshing of surfaces. *Graphical Models* 67, 5 (2005).

David Bommes, Bruno Lévy, Nico Pietroni, Enrico Puppo, Claudio Silva, Marco Tarini, and Denis Zorin. 2013. Quad-Mesh Generation and Processing: A Survey. *Computer Graphics Forum* 32, 6 (2013).

Long Chen and Michael J. Holst. 2011. Efficient Mesh Optimization Schemes based on Optimal Delaunay Triangulations. *Comput. Methods Appl. Mech. Engrg.* 200 (2011).

Long Chen and Jin-chao Xu. 2004. Optimal delaunay triangulations. *Journal of Computational Mathematics* (2004).

Siu-Wing Cheng, Tamal K. Dey, and Jonathan Shewchuk. 2012. *Delaunay Mesh Generation* (1st ed.). Chapman & Hall/CRC.

L. P. Chew. 1987. Constrained Delaunay Triangulations. In *Proceedings of the Third Annual Symposium on Computational Geometry (SCG '87).* ACM.

L. P. Chew. 1993. Guaranteed-quality Mesh Generation for Curved Surfaces. In *Proceedings of the Ninth Annual Symposium on Computational Geometry (SCG '93).* ACM.

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.* The Eurographics Association.

Keenan Crane, Clarisse Weischedel, and Max Wardetzky. 2013. Geodesics in heat: A new approach to computing distance based on heat flow. *ACM Transactions on Graphics (TOG)* 32, 5 (2013).

Fernando de Goes, Mathieu Desbrun, Mark Meyer, and Tony DeRose. 2016. Subdivision exterior calculus for geometry processing. *ACM Trans. Graph.* 35, 4 (2016).

Alan Demlow and Maxim A Olshanskii. 2012. An adaptive surface finite element method based on volume meshes. *SIAM J. Numer. Anal.* 50, 3 (2012).

Jeff Erickson and Sariel Har-Peled. 2004. Optimally Cutting a Surface into a Disk. *Discrete & Computational Geometry* 31, 1 (2004).

Leman Feng, Pierre Alliez, Laurent Busé, Hervé Delingette, and Mathieu Desbrun. 2018. Curved optimal delaunay triangulation. *ACM Trans. Graph.* 37, 4 (2018).

M. Fisher, B. Springborn, P. Schröder, and A. I. Bobenko. 2007. An algorithm for the construction of intrinsic delaunay triangulations with applications to digital geometry processing. *Computing* 81, 2 (01 Nov 2007).

Steven Fortune. 1993. A note on Delaunay diagonal flips. *Pattern Recognition Letters* 14, 9 (1993).

Free Software Foundation. 2008. GCC libquadmath.

Michael T Goodrich and Roberto Tamassia. 1997. Dynamic Ray Shooting and Shortest Paths in Planar Subdivisions via Balanced Geodesic Triangulations. *Journal of Algorithms* 23, 1 (1997).

Leonidas Guibas and Jorge Stolfi. 1985. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi. *ACM Trans. Graph.* 4, 2 (April 1985).

A. Hatcher. 2002. *Algebraic Topology.* Cambridge University Press.

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

Alec Jacobson, Ladislav Kavan, and Olga Sorkine-Hornung. 2013. Robust Inside-outside Segmentation Using Generalized Winding Numbers. *ACM Trans. Graph.* 32, 4, Article 33 (July 2013).

Felix Knöppel, Keenan Crane, Ulrich Pinkall, and Peter Schröder. 2013. Globally optimal direction fields. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 59.

Felix Knöppel, Keenan Crane, Ulrich Pinkall, and Peter Schröder. 2015. Stripe patterns on surfaces. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 39.

Jon W. Van Laarhoven and Jeffrey W. Ohlmann. 2011. A randomized Delaunay triangulation heuristic for the Euclidean Steiner tree problem in $\Re^d$. *J. Heuristics* 17, 4 (2011).

Yong-Jin Liu, Dian Fan, Chun-Xu Xu, and Ying He. 2017. Constructing Intrinsic Delaunay Triangulations from the Dual of Geodesic Voronoi Diagrams. *ACM Trans. Graph.* 36, 2, Article 15 (April 2017).

Yong-Jin Liu, Chun-Xu Xu, Dian Fan, and Ying He. 2015. Efficient Construction and Simplification of Delaunay Meshes. *ACM Trans. Graph.* 34, 6, Article 174 (Oct. 2015).

Albert T. Lundell and Stephen Weingram. 1969. *Regular and Semisimplicial CW Complexes.* 77–115.

Richard MacNeal. 1949. *The Solution of Partial Differential Equations by Means of Electrical Networks.* Ph.D. Dissertation. California Institute of Technology.

Khamron Mekchay and Ricardo H Nochetto. 2005. Convergence of adaptive finite element methods for general second order linear elliptic PDEs. *SIAM J. Numer. Anal.* 43, 5 (2005).

Pedro Morin, Ricardo H Nochetto, and Kunibert G Siebert. 2002. Convergence of adaptive finite element methods. *SIAM review* 44, 4 (2002).

Ashish Myles, Nico Pietroni, and Denis Zorin. 2014. Robust field-aligned global parametrization. *ACM Transactions on Graphics (TOG)* 33, 4 (2014).

Giuseppe Patané. 2016. STAR: Laplacian Spectral Kernels and Distances for Geometry Processing and Shape Analysis. In *Proceedings of the 37th Annual Conference of the European Association for Computer Graphics: State of the Art Reports.* Eurographics Association.

Konrad Polthier and Markus Schmies. 1998. Straightest Geodesics on Polyhedral Surfaces. (1998).

Samuel Rippa. 1990. Minimal roughness property of the Delaunay triangulation. *Computer Aided Geometric Design* 7, 6 (1990).

Max Schindler and Evan Chen. 2012. Barycentric Coordinates in Olympiad Geometry.

Teseo Schneider, Yixin Hu, Jeremie Dumas, Xifeng Gao, Daniele Panozzo, and Denis Zorin. 2018. Decoupling Simulation Accuracy from Mesh Quality. 37, 5 (2018).

Silvia Sellán, Herng Yi Cheng, Yuming Ma, Mitchell Dembowski, and Alec Jacobson. 2019. Solid Geometry Processing on Deconstructed Domains. *Computer Graphics Forum* (2019).

Nicholas Sharp, Yousuf Soliman, and Keenan Crane. 2019. The Vector Heat Method. *ACM Trans. Graph.* 38, 2 (2019).

Alla Sheffer and John C. Hart. 2002. Seamster: Inconspicuous Low-distortion Texture Seam Layout. In *Proceedings of the Conference on Visualization '02 (VIS '02).* IEEE Computer Society.

Jonathan Shewchuk. 1999. *Lecture Notes on Geometric Robustness.* Technical Report. University of California at Berkeley.

Jonathan Richard Shewchuk. 1997. *Delaunay Refinement Mesh Generation.* Ph.D. Dissertation. Carnegie Mellon University. Tech Report CMU-CS-97-137.

Philip Shilane, Patrick Min, Michael Kazhdan, and Thomas Funkhouser. 2004. The Princeton Shape Benchmark. In *Shape Modeling International.*

J. Macgregor Smith, D. T. Lee, and Judith S. Liebman. 1981. An O(n log n) heuristic for steiner minimal tree problems on the euclidean metric. *Networks* 11, 1 (1981).

Daniel A Spielman. 2010. Algorithms, Graph Theory, and Linear Equations in Laplacian Matrices. In *Proceedings of the International Congress of Mathematicians*, Vol. 4.

Jane Tournois, Camille Wormser, Pierre Alliez, and Mathieu Desbrun. 2009. Interleaving Delaunay refinement and optimization for practical isotropic tetrahedron mesh generation. 28, 3 (2009), 75.

Godfried Toussaint. 1980. The Relative Neighborhood Graph of a Finite Planar Set. *Pattern Recognition* 12 (1980).

Shi-Qing Xin, Shuang-Min Chen, Ying He, Guo-Jin Wang, Xianfeng Gu, and Hong Qin. 2011. Isotropic Mesh Simplification by Evolving the Geodesic Delaunay Triangulation. In *ISVD.* IEEE Computer Society.

Shi-Qing Xin, Xiang Ying, and Ying He. 2012. Constant-time All-pairs Geodesic Distance Query on Triangle Meshes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '12).* ACM.

Ran Yi, Yong-Jin Liu, and Ying He. 2018. Delaunay Mesh Simplification with Differential Evolution. In *ACM Transactions on Graphics*, Vol. 37.

Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. 2016. Mesh Arrangements for Solid Geometry. *ACM Transactions on Graphics (TOG)* 35, 4 (2016).

Qingnan Zhou and Alec Jacobson. 2016. Thingi10K: A Dataset of 10,000 3D-Printing Models. *arXiv preprint arXiv:1605.04797* (2016).

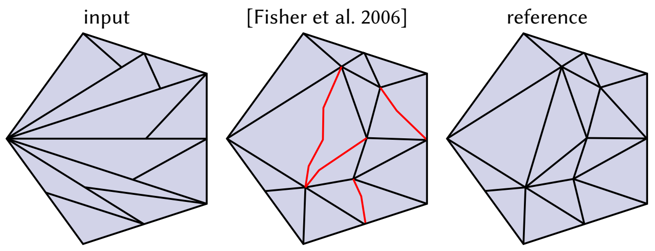


Fig. 27. Even though the incremental overlay of Fisher *et al.* guarantees topological correctness, it can still yield geometrically inaccurate edges—as shown here for the Delaunay triangulation of a near-degenerate input.

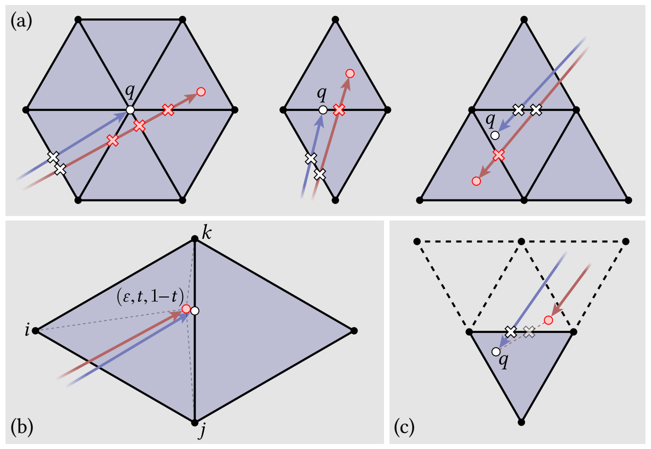


Fig. 28. Even on a nice mesh, a naïve tracing query from p to q may generate spurious crossings (in red) due to floating point error. Our robust tracing procedure helps provide reliable behavior even on highly degenerate models.

A NUMERICS

For most meshes, one can simply implement our data structure exactly as described in Section 3 in double precision, using no special treatment of numerics. However, as noted by Fisher et al. [2007, Section 2.2], careful treatment of floating point can improve the robustness of intrinsic operations for particularly challenging meshes. For the worst 5% of meshes in *Thingi10k*, we found it valuable to carefully treat the floating point evaluation of tracing queries, as described below. For triangle pairs where three vertices are nearly collinear, we found we could improve the accuracy of the length update formula described in Fisher et al. [2007, Sec. 2.2] by computing a Taylor series expansion around large and small angles. We also found that an alternative angle representation (described in supplemental material) helps to improve run time performance (by avoiding transcendental functions), but is not critical for robustness. Ultimately, *all* operations needed for our signpost data structure can be expressed using only rational arithmetic ($+, -, *, \div$) and square roots, meaning that in principle it could be implemented using exact numerical libraries that do not support transcendental operations.

Tracing Queries. Even on high-quality input meshes, a naïve floating point implementation of tracing can yield a sequence of edge crossings that does not encode the correct path. We apply several common-sense improvements, such as only performing ray-edge intersections with edges ij that do not contain the ray origin, and clamping edge barycentric coordinates to the interval $[0, 1]$. Similarly, if a tracing query ever terminates at a point with barycentric coordinates $(b_i, b_j, b_k) = (\varepsilon, t, 1 - t)$ for some ε close to zero (as depicted in Figure 28, (b)), we simply snap this vertex to the edge jk and update the ray direction u accordingly. In situations where we know the target point q *a priori* (*e.g.*, in the case of overlay extraction), we can also use purely topological invariants to improve numerical robustness. In particular, we know that the sequence of crossings produced by a correct path should end with a crossing *into* the star $\mathrm{St}(\sigma)$ of the simplex σ containing q. Removing any subsequent crossings (indicated by red crosses in Figure 28, (a)) and connecting the final valid crossing to the target q therefore yields a path that is topologically valid, even if there is error in the barycentric coordinates. Another possibility is that we do not quite reach $\mathrm{St}(\sigma)$ (as in Figure 28, (c)); in this case, it is almost always sufficient to search immediate neighbors of the triangle containing the path endpoint for σ, then connect to q along a straight line segment. Although more elaborate strategies could be applied, this simple scheme enables us to handle all but the most pathological examples in standard precision (see Section 6.2).

B PSEUDOCODE

Pseudocode for building and manipulating an intrinsic triangulation is provided below. We assume that the intrinsic triangulation is initialized from a standard triangle mesh with vertex positions in $\mathbb{R}^3$, which represents the most common use case. For brevity, we will use $\mathsf{S} = (\mathsf{M}, \ell, \varphi, b, \Theta)$ to denote a signpost mesh, which includes the underlying mesh $\mathsf{M} = (\mathsf{V}, \mathsf{E}, \mathsf{F})$ together with the edge lengths $\ell : \mathsf{E} \to \mathbb{R}_{>0}$, halfedge angles $\varphi : \mathsf{H} \to \mathbb{R}$, and barycentric coordinates $b : \mathsf{V} \to [0, 1]^3$. We also include the vertex angle sums $\Theta : \mathsf{V} \to \mathbb{R}_{>0}$ (Equation 1), which are invariant with respect to changes to the triangulation and can hence be computed just once during initialization (rather than recomputing it from the edge lengths ℓ each time). Finally, we will use ℓ_{ijk} as shorthand for the tuple of edge lengths $(\ell_{ij}, \ell_{jk}, \ell_{ki})$.

All routines not explicitly given below are either elementary geometric calculations, or basic topological operations whose implementation depends on the choice of mesh data structure:

- Degree(M, i)—number of edges in the mesh M that contain vertex i.
- NewVertex(M)—add a new vertex to M and return it.
- EraseTriangles($\mathsf{M}, i_1j_1k_1, i_2j_2k_2, \ldots$)—remove the given triangles from the mesh M.
- InsertTriangles($\mathsf{M}, i_1j_1k_1, i_2j_2k_2, \ldots$)—add the given triangles to the mesh M.
- Angle($\ell_{ij}, \ell_{jk}, \ell_{ki}$)—for a triangle with edge lengths ℓ_{ij}, ℓ_{jk}, ℓ_{ki}, returns the interior angle θ_i^{jk}.
- BaseLength(a, b, θ)—returns the third side length of a triangle with sides of length a and b meeting at an angle θ
- AngleBetween(α, β)—given two angles $\alpha, \beta \in \mathbb{R}$ encoding points on the unit circle, returns the *smallest* (unsigned) angle between them.
- Argument(u, v)—given two vectors $u, v \in \mathbb{R}^2$ gives the angle from u to v in the range $[0, 2\pi)$.

The only exception is the routine TraceVector(S, ijk, p, u), which returns the triangle ijk and barycentric coordinates $q \in [0, 1]^3$ for the point reached by starting at a point p (given in barycentric coordinates) and walking along the direction $u/|u|$ for a distance $|u|$; implementation of this routine is discussed in Section 3.2.2 and Appendix A.

Algorithm 1 UpdateSignpost(S, ijk)

Input: A triangle ijk of a signpost mesh S. Assumes ijk has valid edge lengths and a valid angle φ_{ij}.
Output: An updated signpost mesh with valid angle φ_{ik}.

```
1: $\theta_i^{jk} \leftarrow$ Angle($\ell_{ijk}$)
2: $\varphi_{ik} \leftarrow \varphi_{ij} + 2\pi\theta_i^{jk}/\Theta_i$        ▷Section 3.2.1
3: return S
```

Algorithm 2 TraceFromVertex($\mathsf{S}, i, r, \varphi$)

Input: A tangent vector at a vertex i specified via a magnitude r and an angle $\varphi \in [0, 2\pi)$.
Output: An extrinsic triangle $\overline{xyz}$ and point in barycentric coordinates $p \in [0, 1]^3$.

```
1: $n \leftarrow 0$
2: $\theta \leftarrow \varphi_{ij_0}$
3: while $\theta < \varphi$ do
4:     $\theta \leftarrow \theta + 2\pi\theta_i^{j_nj_{n+1}}/\Theta_i$
5:     $n \leftarrow n + 1$
6: end while
7: $\tilde{\varphi} \leftarrow \Theta_i(\varphi - \theta)/2\pi$
8: returnTraceVector($ij_nj_{n+1}, (1, 0, 0), r, \tilde{\varphi}$)
```

Algorithm 3 UpdateVertex(S, i)

Input: A vertex i. Assumes all edge lengths of S are valid, and all angles φ in the link of i are known.
Output: An updated signpost mesh with valid angles φ_{ij} and φ_{ji} for each edge ij incident on i.

```
1: for $n = 0, \ldots,$ Degree(M, $i$) $- 1$ do        ▷update incoming angles
2:     UpdateSignpost(S, $j_nj_{n+1}i$)
3: $(\overline{xyz}, b_i) \leftarrow$ TraceFromVertex(S, $j_0, \ell_{j_0i}, \varphi_{j_0i}$)
4: $\varphi_{ij_0} \leftarrow$ Argument($e_{\overline{xyz}}, -u_{j_0\to i}$)
5: for $n = 1, \ldots,$ Degree(M, $i$) $- 1$ do        ▷update outgoing angles
6:     $\varphi_{ij_n} \leftarrow \varphi_{ij_{n-1}} +$ Angle($\ell_{ij_{n-1}j_n}$)
7:
```

Algorithm 4 InitSignpostMesh(M, f)

Input: A triangle mesh M and vertex positions $f : V \to \mathbb{R}^3$.
Output: A signpost mesh S encoding the input triangulation.

```
1: $S \leftarrow (M, \ell, \varphi, b, \Theta)$                ▷ allocate storage
2: for each $ij \in E$ do $\ell_{ij} \leftarrow |f_j - f_i|$         ▷ get edge lengths
3: for each $i \in V$ do
4:     $\Theta_i \leftarrow 0$        ▷ compute total angle around vertex i
5:     for $n = 0, \ldots, \text{Degree}(M, i) - 1$ do
6:         $\Theta_i \leftarrow \Theta_i + \text{Angle}(\ell_{ij_nj_{n+1}})$
7:     $\varphi_{ij_0} \leftarrow 0$              ▷ compute signpost directions
8:     for $n = 0, \ldots, \text{Degree}(M, i) - 2$ do
9:         UpdateSignpost(S, $ij_nj_{n+1}$)
10: return S
```

Algorithm 5 FlipEdge(S, ij)

Input: An interior edge ij, with opposite vertices k and l.
Output: An updated signpost mesh, with edge ij flipped (Section 3.3.1).

```
1: $\theta_i^{lk} \leftarrow \text{Angle}(\ell_{ijk}) + \text{Angle}(\ell_{ilj})$
2: EraseTriangles(M, $ijk, jil$)                 ▷ update connectivity
3: InsertTriangles(M, $ilk, jkl$)
4: $\ell_{kl} \leftarrow \text{BaseLength}(\ell_{ik}, \ell_{il}, \theta_i^{lk})$      ▷ length of flipped edge
5: UpdateSignpost(S, $ljk$)               ▷ direction of halfedge lk
6: UpdateSignpost(S, $kil$)               ▷ direction of halfedge kl
7: return S
```

Algorithm 6 Distance($\ell_{12}, \ell_{23}, \ell_{31}, p, q$)

Input: Three lengths $\ell_{12}, \ell_{23}, \ell_{31} \in \mathbb{R}_{>0}$ satisfying the triangle inequality, and barycentric coordinates $p, q \in [0, 1]^3$ for two points in this triangle.
Output: The distance between the points specified by p and q (using a formula from Schindler and Chen [2012, Section 3.2]).

```
1: $u \leftarrow q - p$
2: $d \leftarrow -(\ell_{12}^2 u_1 u_2 + \ell_{23}^2 u_2 u_3 + \ell_{31}^2 u_3 u_1)$
3: return $\sqrt{d}$
```

Algorithm 7 InsertVertex(S, b)

Input: A point on the interior of a triangle ijk, specified via positive barycentric coordinates $b_i + b_j + b_k = 1$.
Output: An updated signpost mesh, with a vertex inserted at b (Section 3.3.2).

```
1: $p \leftarrow \text{NewVertex}(M)$                ▷ update connectivity
2: EraseTriangles(M, $ijk$)
3: InsertTriangles(M, $ijp, jkp, kip$)
4: $\ell_{ip} \leftarrow \text{Distance}(\ell_{ijk}, i, b)$          ▷ update edge lengths
5: $\ell_{jp} \leftarrow \text{Distance}(\ell_{ijk}, j, b)$
6: $\ell_{kp} \leftarrow \text{Distance}(\ell_{ijk}, k, b)$
7: UpdateVertex(S, $p$)                  ▷ update signpost angles
8: return S
```

Algorithm 8 SplitEdge(S, b)

Input: A point on the interior of an edge ij with opposite vertices k, l, specified via positive barycentric coordinates $b_i + b_j = 1$.
Output: An updated signpost mesh, with a vertex inserted at b (Section 3.3.2).

```
1: $p \leftarrow \text{NewVertex}(M)$                ▷ update connectivity
2: EraseTriangles(M, $ijk, jil$)
3: InsertTriangles(M, $ipk, kpj, jpl, lpi$)
4: $\ell_{ip} \leftarrow \ell_{ij} b_i$                       ▷ update edge lengths
5: $\ell_{jp} \leftarrow \ell_{ij} b_j$
6: $\ell_{kp} \leftarrow \text{Distance}(\ell_{ijk}, k, (b_i, b_j, 0))$
7: $\ell_{lp} \leftarrow \text{Distance}(\ell_{jil}, l, (b_i, b_j, 0))$
8: UpdateVertex(S, $p$)                       ▷ update angles
9: return S
```

Algorithm 9 VectorToPoint($\ell_{ij}, \ell_{jk}, \ell_{ki}, p$)

Input: The three side lengths of a triangle ijk, and barycentric coordinates $p_i + p_j + p_k$ for a point in this triangle.
Output: The polar coordinates (r, φ) for the vector from i to p, where φ is expressed relative to edge ij.

```
1: $r_{pi} \leftarrow \text{Distance}(\ell_{ijk}, i, p)$          ▷ distance from p to i
2: $r_{jp} \leftarrow \text{Distance}(\ell_{ijk}, j, p)$          ▷ distance from j to p
3: $\varphi_{ip} \leftarrow \text{Angle}(\ell_{ij}, r_{jp}, r_{pi})$        ▷ angle from ij to ip
4: return $(r_{ip}, \varphi_{ip})$
```

Algorithm 10 MoveVertex(S, i, iab, p)

Input: An inserted vertex i, and a point in an intrinsic triangle iab, specified by nonnegative barycentric coordinates $p_i + p_a + p_b = 1$.
Output: An updated signpost mesh, where i has been moved to p (Section 3.3.3).

```
1: $(r, \varphi) \leftarrow \text{VectorToPoint}(\ell_{iab}, p)$            ▷ vector from i to p
2: $\varphi \leftarrow \varphi_{ia} + \varphi$          ▷ $\Theta_i = 2\pi$, since i is inserted vertex
3: for $n = 0, \ldots, \text{Degree}(M, i) - 1$ do       ▷ iterate over neighbors
4:     $\alpha \leftarrow \text{AngleBetween}(\varphi, \varphi_{ij_n})$
5:     $\ell_{pj_n} \leftarrow \sqrt{r^2 + \ell_{ij_n}^2 - 2r\ell_{ij_n}\cos\alpha}$     ▷ update edge lengths
6: UpdateVertex(S, $i$)                   ▷ update signpost angles
7: return S
```

Algorithm 11 PointQuery(S, $\overline{xyz}$, p)

Input: A point in triangle $\overline{xyz}$ of the extrinsic mesh, given in barycentric coordinates $\bar{p}_i + \bar{p}_j + \bar{p}_k = 1$.
Output: The intrinsic triangle ijk and barycentric coordinates p_i, p_j, p_k for the corresponding point on the intrinsic mesh.

```
1: $(r, \varphi) \leftarrow \text{VectorToPoint}(\ell_{\overline{xyz}}, p)$        ▷ vector from $\bar{x}$ to p
2: $(ijk, p) \leftarrow \text{TraceVector}(S, \overline{xyz}, \bar{x}, r, \varphi)$
3: return $(ijk, p)$
```