LINEAR-ALGEBRAIC REPRESENTATION AND TRANSFORMATION OF UNSTRUCTURED MESHES

Daniel Shapero[1]

[1] *University of Washington, Seattle, WA, USA, shapero@uw.edu*

ABSTRACT

This paper will show some new approaches for implementing common transformations to the connectivity or topology of an unstructured mesh. The key enabling technology for our approach is to borrow ideas from algebraic topology: we use the *boundary operators* of a *chain complex* to represent the mesh. Boundary operators are really just integer matrices. By representing the objects of study using the language of linear algebra, we can use linear algebraic reasoning and intuition to define transformations. We show how to implement a few common transformations on the boundary operator representation: merging cells, splitting cells on a new vertex, and splitting a cell on facets. The source code for these transformations becomes short and easy to test.

Keywords: mesh generation, computational geometry, algebraic topology

1. INTRODUCTION

Nearly all constructions in unstructured meshing require the ability to perform local transformations to the mesh topology. For example, to compute the Delaunay triangulation, the Lawson algorithm uses a sequence of edge flips, while the Bowyer-Watson algorithm is based on splitting star-shaped polytopes along a vertex [1]. Algorithms for mesh coarsening, on the other hand, apply a sequence of edge or face collapses [2]. Implementing these low-level transformation kernels on common mesh data structures can be difficult and error-prone. Are there other mesh data structures that make common algorithms easier to implement?

The idea of *linear-algebraic representation* is to describe the mesh topology using a sequence of linear operators between certain vector spaces or modules. The idea comes from algebraic topology: the linear operators are the *boundary operators* on a certain *chain complex*. **If we can describe the mesh topology using linear algebra, then we can transform it using linear algebra as well.** Other domains of science and engineering have seized on the idea of using linear algebra as the common language for building applications as well. For example, the GraphBLAS project aims to implement common algorithms in graph theory using linear algebra [3].

This paper will show three transformations on the linear-algebraic representation of a polygonal mesh: (1) splitting a cell on a vertex, (2) merging adjacent cells, and (3) subdividing a cell along a collection of facets. The main advantage of this linear-algebraic approach is that the transformation kernels are easy to write down and to code. We show how other higher-level transformations can be implemented in terms of these primitive operations. Finally, as a proof-of-concept, we implemented algorithms for computing convex hulls in arbitrary dimensions and constrained Delaunay triangulations in 2D.

2. RELATED WORK

The key innovation of this paper is using chain complexes to implement certain topological transformations, mainly bistellar flips. Many previous works have used bistellar flips and a few have used chain complexes. To our knowledge, no work has combined both.

Nearly all methods for improving the quality of a mesh by transforming its topology use bistellar flips [4, 5, 6,

7]. One notable exception is polyhedron reconnection [8]. These works all assume a half-edge or array-based data structure to describe a simplicial complex.

The closest work to ours is DiCarlo and others [9], which defines the Euler operators and a bisection operator on chain complexes. The operators defined in that paper work on only two cells at a time; ours operate on an arbitrary number of cells. Other papers use chain complexes to implement Catmull-Clark subdivision [10] or to compute arrangements [11].

Our main results are explicit formulas for certain transformations in arbitrary dimensions. These are equations (18) and (19) for vertex splits; (24) for merges; and (32) and (33) for face splits. These formulas have not appeared in the meshing literature before. Equation (18) has appeared in the homological algebra literature [12] but not for practical computation. The GAP and polymake packages both include routines for computing topological cones and performing bistellar moves on simplicial complexes [13, 14, 15]. But both of these packages use array-based data structures rather than the linear-algebraic representation. Finally, these routines are not used for meshing as such but to find homeomorphisms of piecewise-linear manifolds.

3. THEORY

In this section, we will describe polytopal complexes, a class which includes all common types of meshes. The boundary operators on a polytopal complex can be used as a data structure to represent the complex. Next, we will describe how to convert between a polytopal complex and the more familiar representation of a simplicial complex as an array describing which vertices are contained in each simplex. Finally, we'll then introduce the concept of a topological cone and give explicit formulae for the boundary operators of a cone. The transformations that we describe in the next section all "factor through" the topological cone.

3.1 Polytopal complexes

We will assume as given the notions of simplices, cubes, polytopes, and meshes or spatial subdivisions made up of these objects. The mathematical theory that underlies the rest of the paper is basic algebraic topology. We will present very abbreviated definitions; we refer to [9] for a longer exposition of the algebraic-topological point of view on solid modeling and to [16] for the theory of algebraic topology.

Let Ω be a polytopal mesh of some domain in Euclidean space. Given a k-dimensional polytope σ and a $k-1$-dimensional face τ of σ, the *incidence number* is an integer that describes the handedness or orientation with which τ is attached into σ. We will write the incidence number as $[\sigma, \tau]$. An informal definition of the incidence number is

$$[\sigma,\tau] \equiv \begin{cases} +1 & \tau \text{ is attached positively in } \sigma \\ -1 & \tau \text{ is attached negatively in } \sigma \\ 0 & \tau \text{ is not in } \sigma \end{cases}. \quad (1)$$

This informal definition is enough for the constructions that follow. For example, if $e = \langle v_0, v_1 \rangle$ is an edge connecting the vertices v_0 and v_1, then $[e, v_0] = -1$ and $[e, v_1] = +1$. Likewise, if we consider a positively-oriented triangle $t = \langle v_0, v_1, v_2 \rangle$, then the incidence number of t to the edges $\langle v_0, v_1 \rangle$, $\langle v_1, v_2 \rangle$, and $\langle v_2, v_0 \rangle$ are all positive. The most important property of incidence numbers is that, if ω is a $k+1$-cell and τ is a $k-1$-cell, then

$$\sum_{\sigma} [\omega, \sigma] \cdot [\sigma, \tau] = 0 \quad (2)$$

where the sum is over all k-cells σ. This is a property that requires proof for different classes of cell types. The proofs for simplices and cubes use only combinatorial facts about taking faces of these cells; the proof for general polytopes is more involved [16, 17]. We will assume this property in the following.

Let n_k be the number of cells of Ω of dimension k. Suppose that we have numbered all of the cells, so the k-cells can be enumerated as $\{\sigma_j\}_{j=1}^{n_k}$. We associate to the set of all k-dimensional cells a $\mathbb{Z}$-module $C^k\Omega$ of dimension n_k. Each k-cell σ_j corresponds to the j-th standard basis vector e_j. These modules are called the *chain spaces* and an element of a chain space is referred to as a chain or a k-chain.

The utility of chains is that taking the boundary of a cell is a one-to-many relation – a single simplex or cube has many faces. A vector with non-zero entries corresponding to these faces summarizes which cells are in the boundary, and their signs describe the incidence. The boundary operators extend this notion beyond a single cell to a linear combination of cells, i.e. a chain. The boundary operators are linear mappings from k-chains to $k-1$-chains:

$$\partial_k : C^k\Omega \to C^{k-1}\Omega. \quad (3)$$

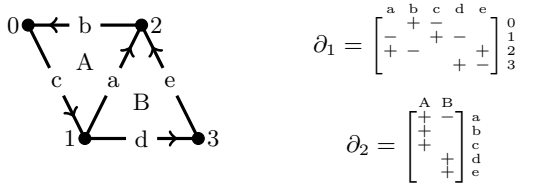

Figure 1: Pair of adjacent triangles (left) and their boundary matrices (right). Vertices are labelled with numbers, edges with lower-case letters, and polygons with upper-case letters.

The representation of a boundary operator in the basis described above is an $n_{k-1} \times n_k$ matrix; its entries are the incidence coefficients. Suppose that the $k-1$-cells are enumerated as $\{\tau_i\}_{i=1}^{n_{k-1}}$. The i, j entry of the matrix ∂_k is then

$$(\partial_k)_{ij} = [\sigma_j, \tau_i]. \tag{4}$$

By reading off columns of each boundary matrix, we can see which $k-1$-cells are faces of a given k-cell and how they are attached. To continue with the example above, we might say that if $e = \langle v_0, v_1 \rangle$ is an edge, then $\partial e = +v_1 - v_0$. Likewise, if $t = \langle v_0, v_1, v_2 \rangle$ is a triangle, then $\partial t = \langle v_0, v_1 \rangle + \langle v_1, v_2 \rangle + \langle v_2, v_0 \rangle$. Figure 1 shows the boundary matrices for two adjacent triangles.

Equation (2) lets us prove the most important fact about the boundary operators. **The boundary of a boundary is always equal to zero:**

$$\partial_k \cdot \partial_{k+1} = 0. \tag{5}$$

A collection of modules $\{C^k\}$ together with linear operators $\partial_k : C^k \to C^{k-1}$ that satisfy equation (5) is called a *chain complex*. From a polytopal complex one can get a chain complex, but chain complexes can be studied on their own.

There is one final definition that will clarify a later construction at a critical juncture. The definitions above assume that the mesh terminates at the points or 0-dimensional cells. We will instead include a single *bottom* cell $\perp$ of dimension -1 and we will say that the bottom cell is positively incident to every vertex, i.e.

$$[v, \perp] = +1 \tag{6}$$

for all vertices v. We can then define the 0-boundary operator ∂_0 as a row vector of all 1s:

$$\partial_0 = \mathbb{1}^*. \tag{7}$$

The condition that $\partial_0 \partial_1 = 0$ now implies that the boundary of every edge has one positive and one negative vertex: $\partial e = v_i - v_j$ for some i, j. We cannot have that, say, $\partial e = v_i + v_j$. This outcome would be undesirable but we have not explicitly forbidden it otherwise. Equation (7) will reappear when we define the topological cone.

The boundary matrices do not capture important geometric data describing how cells are embedded into Euclidean space. We could have the same connectivity structure for two different triangular meshes, one which uses only piecewise linear facets and another that uses curved polynomial mappings. In the remainder, we consider only transformations to the connectivity structure or topology and not to the geometry.

What kinds of alterations or transformations to a set of boundary matrices preserve the fundamental relation $\partial\partial = 0$? Suppose A, B are integer matrices, $\{\partial_0, \ldots, \partial_n\}$ are boundary matrices, and we define

$$\partial'_k = \partial_k \cdot A, \quad \partial'_{k+1} = B \cdot \partial_{k+1}. \tag{8}$$

What conditions do we need on A and B in order to guarantee that $\partial'_k \cdot \partial'_{k+1} = 0$? One sufficient condition is that *the image of ∂_{k+1} is an invariant subspace of $A \cdot B$*. An important particular case is $A \cdot B = I$, which includes both permutations and sign flips. We can reorder the columns of ∂_k so long as we apply the inverse permutation to the rows of ∂_{k+1}. Likewise, we can flip the sign of any column of ∂_k as long as we flip the sign of the corresponding row of ∂_{k+1}. Speaking informally, we can say that individual incidence numbers don't matter as much as the relation between all cells and faces. The more general case where $A \cdot B$ is not equal to the identity matrix but its image is an invariant subspace of ∂_{k+1} is needed for other transformations, such as merging multiple cells into one or deleting cells.

3.2 Simplicial complexes

For many problems in meshing, the goal is to produce a mesh where all the cells are simplices or cubes rather than general polytopes. The utility of polytopal meshes is that they can represent intermediate states that are not of the right type. For example, one might merge a set of triangles into a polygon and then split or subdivide the polygon into a different triangulation incrementally. A polytopal mesh can represent the polygonal intermediate states, whereas if one could only represent triangular data then this transformation has to be completed all in one shot. But the ability to use polytopal meshes, or more specifically the boundary operators, as an intermediate step relies on the ability to convert between the two.

We assume that the standard data structure for a pure, orientable k-dimensional simplicial complex is an array of size $n \times (k+1)$ where n is the number of top simplices. Each row of this array stores the vertices of the corresponding simplex, ordered in such a way to give a positive orientation. For a k-dimensional complex embedded in $\mathbb{R}^k$, positive orientation is the usual condition that the determinant of the matrix formed by a simplex's points in homogeneous coordinates is positive. The order is non-unique up to any permutation with positive parity.

Simplicial → polytopal. Let $\sigma = \langle v_0, \ldots, v_k \rangle$ be an oriented k-simplex. If we re-order the vertices of σ by some permutation p, then we get the same simplex if p has positive parity and the "opposite" simplex if p has negative parity:

$$\langle v_{p(0)}, \ldots, v_{p(k)} \rangle = \text{parity}(p) \cdot \langle v_0, \ldots, v_k \rangle \tag{9}$$

The incidence number from σ to the face τ =

$\langle v_0, \ldots, \hat{v}_j, \ldots, v_k \rangle$ obtained by removing the jth vertex is positive if j is even and negative if j is odd:

$$[\sigma, \tau] = (-1)^j. \tag{10}$$

We can motivate this formula by observing that if σ is in $\mathbb{R}^k$, the oriented half-space formed by the jth face contains σ if j is even and does not contain σ if j is odd. Given a numbering scheme for all the simplices of the mesh, the previous equation gives us the incidence numbers that we need to fill in the entries of the boundary matrices for each simplex.

Polytopal → simplicial. Suppose that $\{\partial_0, \ldots, \partial_k\}$ are the boundary operators of a single simplex with vertices $v_0, \ldots, v_k$. We do not know whether the right order of the vertices is $\langle v_0, \ldots, v_k \rangle$ or some orientation-reversing permutation. Now let $\{\partial'_0, \ldots, \partial'_k\}$ be the boundary matrices of a simplex using the canonical choice of incidence numbers from equation (10). The goal is to find an isomorphism of the two complexes. We take the starting permutation and sign flip matrices to be $P_0 = I_{n_0}$ where n_0 is the number of vertices and $s_0 = \mathbb{1}$. For each dimension $1 \leq j < k$, we find a permutation matrix P_j and sign flips s_j such that

$$\mathrm{diag}(s_{j-1}) \cdot P^*_{j-1} \cdot \partial_j \cdot P_j \cdot \mathrm{diag}(s_j) = \partial'_j. \tag{11}$$

The permutations can be found by finding columns with the same sets of non-zero entries. Finally, at dimension k, we will find that $\mathrm{diag}(s_{k-1}) \cdot P^*_{k-1}\partial_k$ is a column vector of all +1 or -1. If the resulting vector is positive, then the ordering $\langle v_0, \ldots, v_k \rangle$ will result in boundary matrices that are isomorphic to the input boundary matrices. Otherwise, any orientation-reversing permutation of that order is the correct choice. We can then apply this logic to each top-dimensional cell of a whole complex provided that it is known to be simplicial. If at any point a set of permutations and sign flips cannot be found, the resulting polytopal complex was not simplicial.

3.3 Cone spaces

In the next section, we will show several transformations on a polytopal complex that are expressible as operations on their boundary matrices. Each of these transformations is a type of *bistellar move*, also referred to in the literature as a Pachner move after U. Pachner [18, 19, 20]. A bistellar move of a set of cells of a mesh first embeds them into the boundary of a higher-dimensional ball, and then replaces them with the closure of their complement in the boundary of the ball. These transformations are known to preserve important invariants and other global properties. The fact that the new cells have the same boundary as the old ensures that the the fundamental relation $\partial\partial = 0$ is preserved. But how we polygonize the higher-dimensional ball is a matter of choice. The constructions that follow are all based on a particular polygonization called the topological *cone*.

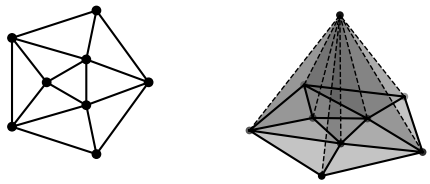

Figure 2: A 2D footprint polygon (left) and its topological cone (right).

To describe what a topological cone is, it is helpful to first describe the concept of a *join*. The join of two subsets C and C' of Euclidean space is the set

$$C * C' = \{\lambda \cdot c + (1 - \lambda)c' : c \in C, c' \in C', \lambda \in [0, 1]\}. \tag{12}$$

Intuitively, the join of two spaces consists of the set of all lines between them. If C and C' are, respectively, m- and n-dimensional submanifolds, then $C * C'$ is $m + n + 1$-dimensional. An important fact for the transformations that we'll define is that the join of an m-simplex and an n-simplex is an $m + n + 1$-simplex.

The topological cone of a space Ω, which we will write as $\mathrm{cone}(\Omega)$, is the join with a single point. We will refer to this added point as the *apex*. An illustration is shown in Figure 2. Suppose that we have a set of boundary matrices $\{\partial_0, \ldots, \partial_n\}$ for a space Ω. **We can express the boundary matrices $\{\partial'_0, \ldots, \partial'_{n+1}\}$ for cone(Ω) in terms of the boundary matrices of the original space.** This construction underlies both the vertex- and face-split transformations described below.

First, if the original space has N vertices, the cone space adds one more. So ∂_0 is a row vector of all 1s with N columns, and ∂'_0 is a row vector of all 1s with $N + 1$ columns. Next, the cone space includes all the 1-cells or edges of the original space, so we know that ∂'_1 will include ∂_1 as a sub-matrix. Forming the cone space adds edges between every original vertex and the apex of the cone. Any one of these edges could have negative or positive incidence on the apex. But by applying sign flips to the columns of ∂'_1, which as per the previous section does not alter the topology, we can always guarantee that every edge has positive incidence to an original vertex and negative incidence to the new cone vertex. In linear algebraic terms,

$$\partial'_1 = \begin{bmatrix} \partial_1 & I \\ 0 & -\mathbb{1}^* \end{bmatrix} \tag{13}$$

where $\mathbb{1}$ is the vector of all 1s. Moreover, we can ob-

serve here that the row vector of all 1s is the same as the 0-boundary matrix:

$$\partial_1' = \begin{bmatrix} \partial_1 & I \\ 0 & -\partial_0 \end{bmatrix}. \tag{14}$$

We can then state that the number of edges or 1-cells in the cone space is equal to #vertices + #edges.

Now we need to determine what the remaining boundary operators should be. We can start by counting how many 2-cells there should be in the cone. Every 2-cell of the original space must also be present in the cone. The added 2-cells are formed through a topological join of the original 1-cells with the cone vertex. So we can again state that the number of 2-cells in the cone space is equal to #2-cells + #1-cells. Again, in principle all of the new 2-cells could have a completely arbitrary incidence w.r.t. to the original 1-cells, but by a sign flip we can ensure instead that all of these incidences are positive. In other words, so far, we know that the 2-boundary matrix has the form

$$\partial_2' = \begin{bmatrix} \partial_2 & I \\ 0 & [?] \end{bmatrix} \tag{15}$$

where the question mark is a matrix that has to be determined. We have a constraint, however, that $\partial_1' \cdot \partial_2' = 0$. The upper-right block of the product $\partial_1' \cdot \partial_2'$ is

$$\partial_1 \cdot I + I \cdot [?] \tag{16}$$

and a solution that would make this expression equal to zero is to take $[?] = -\partial_1$. In other words, we make the ansatz

$$\partial_2' = \begin{bmatrix} \partial_2 & I \\ 0 & -\partial_1 \end{bmatrix}. \tag{17}$$

Calculating each block by hand, we find that this guess does indeed make $\partial_1' \cdot \partial_2' = 0$.

Equations (14) and (17) both have the same form but with the dimensions incremented by 1. We can then guess that, in general, all of the boundary matrices of the cone space up to dimension n have the form

$$\partial_k' = \begin{bmatrix} \partial_k & I \\ 0 & -\partial_{k-1} \end{bmatrix}. \tag{18}$$

Again, a rudimentary hand-computation by blocks shows that $\partial_k' \cdot \partial_{k+1}' = 0$. To complete the construction, the final boundary matrix is

$$\partial_{n+1}' = (-1)^n \begin{bmatrix} -I \\ \partial_n \end{bmatrix}. \tag{19}$$

We will explain the sign convention below. **These two equations are the crux of the paper.**

A few consequences of equations (18), (19) are then apparent. First, the number of k-cells in cone(Ω) is equal to the number of k-cells + the number of $k-1$-cells in Ω. Second, while we have chosen particular signs for the incidence numbers, we can flip these signs as we see fit. For example, in the vertex-split transformation we will write the n-boundary matrix as

$$\partial_n' = \begin{bmatrix} \partial_n & \mathrm{diag}(\partial_n \cdot \mathbb{1}) \\ 0 & -\partial_{n-1} \cdot \mathrm{diag}(\partial_n \cdot \mathbb{1}) \end{bmatrix} \tag{20}$$

with an equivalent sign flip applied to the $n+1$-boundary matrix. Finally, we can write down a formula for the boundary matrices of the *suspension* of a space – the topological join with two isolated points instead of one – in the same way:

$$\partial_{k+1}' = \begin{bmatrix} \partial_k & I & I \\ 0 & -\partial_{k-1} & 0 \\ 0 & 0 & -\partial_{k-1} \end{bmatrix} \tag{21}$$

and for the top cells

$$\partial_{n+1}' = (-1)^n \begin{bmatrix} -I & +I \\ +\partial_n & 0 \\ 0 & -\partial_n \end{bmatrix}. \tag{22}$$

Suspensions will appear again when we consider 2-3 flips of tetrahedra and multi-face retriangulation.

Before we proceed to the transformations themselves, we should explain the sign convention in equation (19). The sign of the final boundary matrix is ultimately arbitrary – we can multiply on the right by any sign flip and still preserve $\partial\partial = 0$. But we can pick a normalization by the following argument. The standard construction defines the boundary of a k-simplex according to equation (10). But we can also construct the simplex inductively as a cone space. A 0-simplex is a single point, and a $k+1$-simplex is the cone of a single point and the k-simplex with boundary matrices chosen according to equations (18) and (19). Choosing the normalization factor $(-1)^n$ makes the repeated cone construction isomorphic to the positively-oriented simplex through a sequence of permutations and sign flips. With no normalization at all, the even-dimensional simplices under the cone construction are instead isomorphic to the standard construction *but with the opposite orientation.*

4. TRANSFORMATIONS

Having written down the explicit formulas (18) and (19) for the boundary matrices of a cone, we can then apply them to define transformations.

4.1 Merging

A *merge* of a set of k-cells replaces them with a single cell. Merging is a column operation on the matrix ∂_k. In the simplest case, the result column is the sum of

all the columns to be merged, but in general we might need to flip some signs:

$$\partial'_k = \partial_k \cdot s, \tag{23}$$
$$\partial'_{k+1} = e_i^* \cdot \partial_{k+1}. \tag{24}$$

where e_i is again the i-th standard basis vector and the entries s_i of the vector s are all ± 1. The signs are chosen so that any higher-dimensional cell σ has the same incidence with respect to any of the cells τ to be merged. The transformation to the rows of ∂_{k+1} collapses all incidence to any of the desired k-cells into incidence to the merged k-cell. For merging cells of top dimension n, there are no higher-dimensional cells to apply equation (24) to and this step is left out.

We can also write a merge as a bistellar move, in which case the extended boundary matrices are:

$$\partial'_k = \begin{bmatrix} -\partial_k & \partial_k \cdot s \end{bmatrix} = \partial_k \cdot \begin{bmatrix} -I & s \end{bmatrix} \tag{25}$$
$$\partial'_{k+1} = \begin{bmatrix} \partial_{k+1} \\ e_i^* \partial_{k+1} \end{bmatrix} = \begin{bmatrix} I \\ -e_i^* \end{bmatrix} \partial_{k+1} \tag{26}$$

and the $k+2$-boundary matrix, if any, is unaltered.

Edge collapsing, the key transformation in surface simplification algorithms [2], is a merge of two vertices.

4.2 Vertex-splitting

A *vertex-split* divides the union of several polytopes along a vertex. The key correctness criteria for this operation are that (1) every newly-created polytope contains the splitting vertex and (2) the boundary of the sum of all polytopes does not change. This second condition can be expressed mathematically as

$$\partial'_n \cdot \mathbb{1} = \begin{bmatrix} \partial_n \cdot \mathbb{1} \\ 0 \end{bmatrix}. \tag{27}$$

The cells of the vertex split can be obtained as the "top" of the topological cone of the original cells, so equations (18) and (19) are applicable with some modifications. Boundary matrices ∂'_0 through ∂'_{n-1} all transform according to equation (18).

For dimension n, we want to do three things differently. First, the cone over a space has 1 higher dimension, but we want to transform a space to one of the same dimension. We thus discard the $n+1$-th boundary matrix from the cone when we form a vertex split. Second, the cone over a space includes the space itself as a sub-complex, whereas we want to remove the original polytopes. The resulting modification is that we discard the lower- and upper-left blocks of ∂'_n from the usual formula for cones. Finally, we want to preserve the original boundary of the starting polytopes, as specified in equation (27). The alteration to the usual cone formula is to multiply ∂'_n on the right by

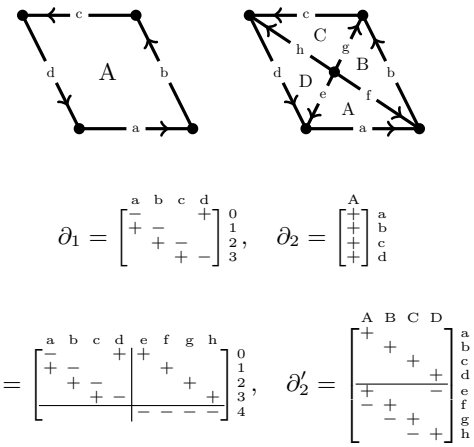

Figure 3: Quadrilateral before and after splitting on a new vertex in the center (top) and boundary matrices before and after (bottom).

the diagonal matrix $\mathrm{diag}(\partial_n \cdot \mathbb{1})$. Putting all of these together, we find that

$$\partial'_n \equiv \begin{bmatrix} \partial_n & I \\ 0 & -\partial_{n-1} \end{bmatrix} \cdot \underbrace{\begin{bmatrix} 0 \\ I \end{bmatrix}}_{\substack{\text{cut left} \\ \text{column}}} \cdot \underbrace{\mathrm{diag}(\partial_n \cdot \mathbb{1})}_{\substack{\text{preserve} \\ \text{boundaries}}}$$
$$= \begin{bmatrix} \mathrm{diag}(\partial_n \cdot \mathbb{1}) \\ -\partial_{n-1} \cdot \mathrm{diag}(\partial_n \cdot \mathbb{1}) \end{bmatrix} \tag{28}$$

We can see that this choice of ∂'_n satisfies equation (27) by using the fact that $\mathrm{diag}(z) \cdot \mathbb{1} = z$ for any vector z.

A final pruning step is necessary when we split the union of more than one polytope. The vector $\partial_n \cdot \mathbb{1}$ will have zero entries along any interior faces. Consequently, some $n-1$-cells of the newly-generated complex are not in the boundary of any n-cell. We can then remove these zero rows from ∂'_n and columns from ∂'_{n-1}. Proceeding down by dimension, we remove any row from ∂'_k that is all zeros and the corresponding column from ∂'_{k-1} for $k = n-1, \ldots, 1$.

Figure 3 illustrates the split transformation on a single quadrilateral and shows the boundary matrices before and after. The Bowyer-Watson algorithm for computing Delaunay triangulations and all common algorithms for computing convex hulls require only the vertex-split transformation [1].

More transformations can be obtained by combining splits and merges; figure 4 shows how to perform flips in 2D and 3D.

4.3 Face-splitting

A *face-split* divides a polytope into several cells along a collection of splitting faces. We assume that the initial

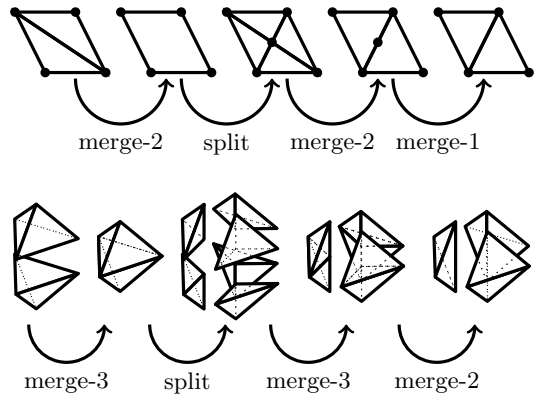

Figure 4: 2-2 and 2-3 flips implemented as sequences of merges and splits. The vertex added in the split step is deleted when the two edges are merged in the final transformation. We've shown the tetrahedra in an "exploded" view to help with visualization.

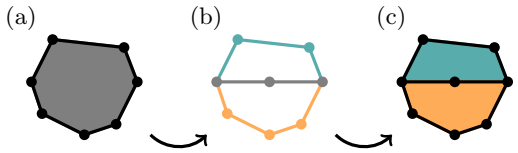

Figure 5: (a) An initial polygon. (b) Add separator cells, shown in grey; the connected components are in teal and orange. A path from a teal edge to an orange edge does not have to pass through a grey edge, but it does have to pass through one of its faces, i.e. a grey vertex. (c) Split the polygon in two along the separator.

polytope is not incident upon the splitting faces. The correctness criteria for this operation are that (1) each new polytope is incident on the splitting faces and (2) the boundary of the sum of all polytopes does not change.

In order to be able to split the top cell, we need to assume a certain connectivity structure between its faces. A *k-path* in Ω is a collection $\{f_0, s_0, f_1, s_1, \ldots, f_{m-1}, s_{m-1}, f_m\}$ such that s_i is a $k-1$-face of both f_i and f_{i+1} for each i. Given two collections of k-cells F_1, F_2, we say that a third collection of k-cells F is a *separator* for F_1, F_2 if any path from a cell f_1 in F_1 to a cell f_2 in F_2 must pass through a $k-1$-face s of some f in F. The definition is similar to that of graph theory but with some important differences in the case of polytopal complexes. See Figure 5 for an illustration.

In order to be able to subdivide P into multiple cells, we need to be able to partition its faces into two or more groups $\{F_1, \ldots, F_m\}$ with a common separator F_0. Moreover, we assume at first that $[P, F_0] = 0$ and $[P, F_i] \neq 0$ for $i \geq 1$. We can find separators and subgroups through a procedure analogous to breadth-first search but with adjacency defined by sharing common subfaces.

The $n-1$-boundary matrix will have the form

$$\partial_{n-1} = \begin{bmatrix} [S, F_0] & [S, F_1] & \cdots & [S, F_m] \end{bmatrix}, \tag{29}$$

where S denotes the set of all subfaces or $n-2$-dimensional cells contained in P, and

$$\partial_n = \begin{bmatrix} 0 \\ [F_1, P] \\ \vdots \\ [F_m, P] \end{bmatrix}. \tag{30}$$

The fundamental equation (5) then implies that

$$\sum_i [S, F_i] \cdot [F_i, P] = 0. \tag{31}$$

We will not alter the $n-1$-boundary matrix at all, only the n-boundary matrix.

Now we make the ansatz that the transformed n-boundary matrix will have one column for each connected component; each column will have one block for the separator F_0; and column k will have a non-zero block for the corresponding component F_k. In all, the new matrix will have the form

$$\partial_n' = \begin{bmatrix} [F_0, P_1] & \cdots & [F_0, P_m] \\ [F_1, P] & & \\ & \ddots & \\ & & [F_m, P] \end{bmatrix} \tag{32}$$

where the incidences $[F_0, P_i]$ need to be solved for. The incidences $[F_i, P]$ are the same as in the initial polytope. If we can show that $\sum_i [F_0, P_i]\mathbb{1} = 0$, then the boundary of the new polytopes will be the same as the old. To preserve the fundamental equation $\partial\partial = 0$, we need that

$$[S, F_0][F_0, P_i] = -[S, F_i][F_i, P] \tag{33}$$

for each i. We can then attempt to find a solution of this collection of linear systems of equations. The existence of a solution is not guaranteed a priori, but if we can find one then it defines a valid subdivision of the original cell. The system is also rectangular, so it may be under- or over-determined depending on the number of faces and subfaces. We can obtain a square system by instead opting to solve

$$[S, F_0]^*[S, F_0][F_0, P_i] = -[S, F_0]^*[S, F_i][F_i, P]. \tag{34}$$

We can compute solutions to integer linear systems by first finding the Hermite normal form of the system

matrix [21]. Let $[S, F_0]^+$ denote the pseudo-inverse of this system or the matrix $\{[S, F_0]^*[S, F_0]\}^{-1}[S, F_0]$ if the system is solvable. Then

$$\sum_i [F_0, P_i]\mathbb{1} = \\ -[S, F_0]^+ \sum_i [S, F_i][F_i, P] \tag{35}$$

which equals zero by equation (31), so indeed the new polytopes have the same boundary as the old.

We can also look at this transformation as a bistellar move. Both the old polytope and the new polytopes are the common boundary of the $n+1$-polytope with boundary matrices

$$\partial'_n = \begin{bmatrix} 0 & [F_0, P_1] & \cdots & [F_0, P_m] \\ -[F_1, P] & +[F_1, P] & & \\ & & \ddots & \\ -[F_m, P] & & & +[F_m, P] \end{bmatrix} \tag{36}$$

and $\partial'_{n+1} = \mathbb{1}$. We then remove the $n+1$-dimensional cell and the original n-cells.

Finally, observe that the size of the linear system (34) is equal to the number of faces of the separator F_0. When there is only one face, the linear system reduces to a scalar equation and questions about solvability reduce to divisibility.

4.4 Tetrahedral mesh operations

The simplest topological transformations used in tetrahedral mesh improvement are 2-3, 3-2, and 4-4 face flips [1]. These transformations work on a relatively small number of tetrahedra at a time. Several publications have argued that two families of transformations, edge removal and multi-face removal, give superior results for increasing common mesh quality measures [5]. A third transformation, multi-face retriangulation, can be written as a composition of multi-face removal followed by edge removal in order to retriangulate a polygon [6]. These transformations operate on many more tetrahedra at a time, but are, according to an author of a popular mesh generator, "particularly tedious to implement" [8]. Here we show how to implement these transformations on the linear algebraic representation of the mesh.

Both transformations operate on tetrahedra that are assumed to have special structure. The starting mesh for multi-face removal is assumed to consist of a set of triangles that are sandwiched between a pair of vertices. The result is a set of tetrahedra that all meet at a common central edge. Edge removal goes in the opposite direction. In either case, we can observe that the sandwich scenario is really the topological suspension of the filling, i.e. the join with two vertices. We will exhibit both of these states as parts of the boundary of a higher-dimensional polytope.

Let $\{\partial_0, \partial_1, \partial_2\}$ be the boundary matrices of the filling faces. We will take the cone of this polytopal complex twice, which will lift it into 4D. The boundary matrices of the first cone are

$$\partial'_1 = \begin{bmatrix} \partial_1 & I \\ & -\mathbb{1}^* \end{bmatrix}, \ \partial'_2 = \begin{bmatrix} \partial_2 & I \\ & -\partial_1 \end{bmatrix}, \ \partial'_3 = \begin{bmatrix} -I \\ \partial_2 \end{bmatrix} \tag{37}$$

and the boundary matrices of the iterated cone are

$$\partial''_1 = \begin{bmatrix} \partial_1 & I & I \\ & -\mathbb{1}^* & & +1 \\ & & -\mathbb{1}^* & -1 \end{bmatrix}, \quad \partial''_2 = \begin{bmatrix} \partial_2 & I & I & \\ & -\partial_1 & & I \\ & & -\partial_1 & -I \\ & & & -\mathbb{1}^* \end{bmatrix},$$

$$\partial''_3 = \begin{bmatrix} -I & +I & \\ +\partial_2 & & +I \\ & -\partial_2 & -I \\ & & \partial_1 \end{bmatrix}, \quad \partial''_4 = \begin{bmatrix} I \\ I \\ -\partial_2 \end{bmatrix}. \tag{38}$$

Notice the first two columns of ∂''_3, highlighted in orange – these are identical to equation (22) for the top boundary matrix of a suspension. In other words, the starting state for multi-face removal and the ending state of edge removal are both isomorphic to a subcomplex of the iterated cone. The complement of these tetrahedra, highlighted in teal, is the starting state of edge removal and the final state of multi-face removal. The column of ∂''_1 corresponding to the edge that joins the apices in the starting state of edge removal / ending state of multi-face removal is highlighted in purple.

The fact that the end states of either operation are both complementary components of the boundary of a higher-dimensional polytope suggests an implementation strategy. First, find an isomorphism of the starting tetrahedra with a subcomplex of the iterated cone. Then remove the starting tetrahedra; the remainder are the desired final state. If we take the original 2D filling faces to be a single triangle, then the previous equations give an alternative implementation of $2 \leftrightarrow 3$ face flips.

5. DEMONSTRATION

As a proof of concept, we developed a Python package called *zmsh* which implements the transformations defined above. The main external dependency is numpy for matrix algebra and array operations. For geometric predicates, we compute determinants using interval arithmetic; if the result interval contains zero, we try again with exact rationals. We then implemented several common algorithms in computational geometry using the linear-algebraic representation of topological transformations described in the previous section.

These transformations and algorithms are built around a small core functionality. The two key data structures are simplicial and polytopal topologies. Functions for querying and operating on both data types are in the modules `simplicial` and `polytopal` respectively.

`polytopal.Topology` : An n-dimensional polytopal topology is a list of $n+1$ integer matrices. The 0th matrix is a row vector of all 1s. A topology is considered valid if it obeys the fundamental relation $\partial\partial = 0$, all its entries are between -1 and +1, and the number of non-zero entries in every column of ∂_k is either 0 or $\geq k+1$.

`simplicial.Topology` : An n-dimensional simplicial topology is an array with one row for every top simplex and $n+1$ columns.

The condition on the number of non-zero entries in each column for a polytopal complex is necessary to ensure regularity.

The simplicial module includes routines for checking that simplices are consistently oriented with respect to each other. The remaining features are all in the polytopal module.

`closure` : Given the integer IDs of a set of k-cells $\{\sigma_\alpha\}$, return the IDs of their immediate faces, their faces' faces, and so on.

`subcomplex` : Given the integer IDs of a set of cells $\{\sigma^k_\alpha\}$ for each dimension $k = 0, \ldots, n$, return the boundary matrices corresponding to the subcomplex formed by these cells.

`find_isomorphism` : Given a pair of complexes, return the permutations and sign flips that transform one into the other if they exist.

`to_simplicial` : Given a polytopal complex, convert it to a simplicial complex if possible. For each top cell of the input complex, find an isomorphism of the subcomplex consisting of this cell and its closure to the boundary operators of the standard simplex using the `find_isomorphism` routine.

`from_simplicial` : Given a simplicial complex, return the equivalent polytopal complex by assigning cell IDs to all lower-dimensional simplices and repeatedly applying equation (10).

`join_vertex` Given a polytopal complex, return the join with a single point (i.e. the cone) as shown in equations (18) and (19).

`join_vertices` Given a polytopal complex, return the join with two points (i.e. the suspension) as shown in equations (21) and (22).

The source code for the `join_vertex` routine is shown in Figure 6; the source code for the `join_vertices` function is similar. The main advantage of working with the linear-algebraic representation is that many key routines are easy to implement and test.

```
1  import numpy as np
2
3  Topology = list[np.ndarray]
4
5  def join_vertex(D: Topology) -> Topology:
6      n = len(D) - 1
7      num_vertices = D[0].shape[1]
8      # Create the boundary matrices for 1 <= k < n
9      E = [ones((1, num_vertices + 1))]
10     for k in range(1, n + 1):
11         num_cells = D[k].shape[1]
12         num_sub_faces, num_faces = D[k - 1].shape
13         I = eye(num_faces)
14         Z = zeros((num_sub_faces, num_cells))
15         E_k = np.block([[D[k], I], [Z, -D[k - 1]]])
16         E.append(E_k)
17
18     # Create the top-dimensional boundary matrix
19     num_cells = D[n].shape[1]
20     I = eye(num_cells)
21     E_n = (-1) ** n * np.vstack((-I, D[n]))
22     E.append(E_n)
23
24     return E
```

Figure 6: Python source code for the split transformation. Lines 15 and 21 correspond to equations (18) and (19) respectively.

The very concise source code for `join_vertex` shown here is an example. We do not claim that using the linear-algebraic representation has superior performance characteristics to the conventional approach based on applying transformations directly to the simplicial data type.

5.1 Convex hulls and vertex-split

The vertex-split transformation is the key computational kernel for computing convex hulls in any dimension. The implementation of the vertex-split transformation consists of a small modification of the `join_vertex` routine shown in Figure 6.

We used the vertex-split transformation to implement a convex hull algorithm that works in arbitrary dimensions. To test the convex hull code, we used (1) random point sets up to dimension 5 and (2) several synthetic input sets with various degeneracies such as coplanarity.

Our implementation of the convex hull algorithm takes in a function as argument which supplies the procedure for computing the signed volume predicate. Computing Delaunay triangulations is equivalent to computing the lower half of a convex hull of the input points lifted to a paraboloid. We also implemented a Delaunay triangulation method which reuses our convex hull algorithm by supplying an insphere function instead of the usual signed volume function.

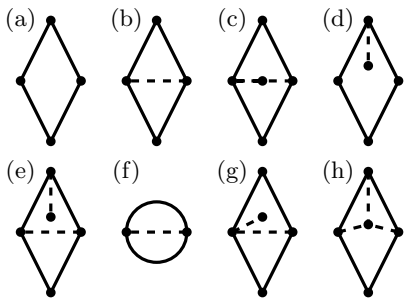

Figure 7: Test cases for splitting polygons along edges: (a) nothing to split at all, (b) splitting along an edge, (c) splitting along multiple edges, (d) nothing to split but with a hanging face, (e) splitting along an edge but with a hanging face, (f) straightforward split but with linearly dependent columns of ∂_1, (g) extraneous hanging edge, (h) three-way split.

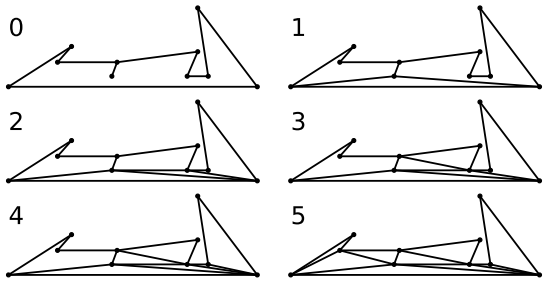

Figure 8: The steps of the cavity retriangulation algorithm. The hanging edge at the start and the non-simplicial intermediate states are all representable as boundary operators.

5.2 Constrained Delaunay triangulation and face-split

The vertex-split transformation is sufficient for computing convex hulls and unconstrained Delaunay triangulations. Computing a constrained Delaunay triangulation (CDT) requires merges and face-splits. We follow the approach of [22] for computing CDTs. First, we compute an unconstrained triangulation of the input point set. We then add constrained edges one by one. To add a constrained edge, we (1) merge all of the triangles that contain it into one polygon and remove any edges that cross the constrained edge; (2) add the constrained edge, dividing this polygon in two; and (3) retriangulating the remaining polygonal cavities in order to maintain the constrained Delaunay property.

A significant complication that can arise when inserting constrained edges is the presence of *hanging edges* of the polygonal cavity [1]. At the initial insertion of the constrained edge and subsequent splitting of the polygonal cavity, we know that the separator consists only of the constrained edge. At some later step, the hanging edge will be connected by two more edges and become part of the separator. So we can assume that at least one face of the separator is known from the outset, but at later stages we might not know what the entire separator is. Our implementation uses a breadth-first search to identify the entire separator from the starting faces. We then perform further searches to mark the remaining connected components.

We first implemented the face-split transformation and tested it on several cases including ones with hanging edges. The test cases we chose are shown in Figure 7. Some of these cases should never show up in real problems, but we check for them anyway in the interest of completeness. Case (h), with only two out of the three separator edges known to start, is the scenario where a hanging edge is incorporated into a CDT.

We then tested several realistic configurations for cavity retriangulation. Figure 8 shows one such test case from Figure 3.13 of [1]. This test case is a non-convex cavity with a hanging edge. The algorithm proceeds entirely by subdividing polygons into smaller ones with no special cases to handle hanging edges beyond the breadth-first searches described above. By contrast, the CavityCDT algorithm described in [1] requires special cases to re-traverse hanging edges and to delete triangles of the wrong orientation.

5.3 Multi-face retriangulation

We implemented part of the multi-face removal and retriangulation transformations described in [6]. These transformations are assumed to operate on the suspension of a set of filling triangles. In [6], the filling triangles are identified via a search procedure and are then transformed into another triangulation of the same polygon. Here we focus entirely on the transformation step and assume that we are given both the filling triangles and the tetrahedra that they fill.

For multi-face re-triangulation, the goal is to take the suspension S_1 of a set of triangles and replace it with the suspension S_2 of a given set of triangles T_2. The Python source code for our implementation is shown in Figure 9. The implementation is correct if it can find an isomorphism between the boundary subcomplexes of S_1 and the suspension of T_2.

For multi-face removal, the goal is to take the suspension S_1 of a set of triangles T_1 and (1) merge the triangles of T_1 into a single polygon P, (2) form the cone of the cone of P, and (3) replace S_1 with its complement in the iterated cone of P. The implementation is correct if the original triangles are no longer present

in the final tetrahedralization, there is an edge connecting the apexes of the suspension, and if the two boundaries are isomorphic. See Figure 10 for an illustration. The source code is in Figure 9.

We evaluated the correctness of both procedures by running a set of randomized test cases. To make the inputs, we generate random triangulations of a polygon of n vertices for $n = 4, 5, \ldots, 10$. We then form the suspension of these triangles and apply a random permutation and sign flip. This permutation and sign flip is included because, in realistic 3D meshes, it is not generally the case that the triangular filling of a set of tetrahedra that we wish to transform is coherently oriented. In other words, two tetrahedra should have opposite incidence to their common triangle, but two triangles need not have opposite incidence to their common edge. For multi-face retriangulation, we generate a second random triangulation of the same polygon as the other input. Our implementations passed all of the randomized tests according to the correctness criteria outlined above for both problems.

The main takeaway here is that the transformation code shown in Figure 9 is very concise and doesn't require deeply-nested loops with large bodies. These routines do rely on other pre-defined library functions such as procedures to compute the closure of a complex and to find isomorphisms, but these procedures are easy to specify and test in isolation.

```python
import numpy as np
from numpy import flatnonzero as nonzero, count_nonzero
from zmsh.polytopal import (
    closure, subcomplex, merge,
    join_vertex, join_vertices,
    find_isomorphism, Topology,
)

def boundary_complex(T: Topology) -> Topology:
    face_ids = nonzero(T[-1].sum(axis=1))
    cell_ids = closure(T[:-1], face_ids)
    return subcomplex(T[:-1], cell_ids)

def retriangulate(
    S_1: Topology,
    T_2: Topology,
) -> Topology:
    dS_1 = boundary_complex(S_1)
    S_2 = join_vertices(T_2)
    dS_2 = boundary_complex(S_2)
    return S_2, find_isomorphism(dS_1, dS_2)

def multi_face_removal(
    S: Topology, face_ids: list[int]
) -> Topology:
    # Merge the triangle filling into a polygon
    cells_ids = closure(S[:-1], face_ids)
    d_0, d_1, d_2 = subcomplex(S[:-1], cells_ids)
    interior_edge_ids = \
        nonzero(count_nonzero(d_2, axis=1) == 2)
    s = merge(poly, interior_edge_ids)
    d_2 = (d_2 @ s).reshape((-1, 1))

    # Form the iterated cone of the polygon and
    # return the complement of its suspension
    cone = join_vertex([d_0, d_1, d_2])
    f_0, f_1, f_2, f_3, f_4 = join_vertex(cone)
    return [f_0, f_1, f_2, f_3[:, 2:]]
```

Figure 9: Source code for multi-face retriangulation and multi-face removal.

5.4 Edge collapse

Finally, we implemented the edge collapse operation used in mesh simplification [2]. This procedure is error-prone enough on edge-based data structures that some authors have resorted to using SMT solvers [23]. The source code is shown in figure 11.

An edge collapse merges two vertices v_0, v_1 into one. We implemented this in three phases. First, we sum the rows in ∂_1 corresponding to v_0 and v_1 together; call the merged vertex v^*. This step creates a host of degeneracies. If we had a third vertex u and edges $e_0 = \langle u, v_0 \rangle$, $e_1 = \langle u, v_1 \rangle$, then after this first step we are left with two copies of the edge $\langle u, v^* \rangle$. From a linear algebra perspective, the columns corresponding to the two copies are scalar multiples of each other. In the second phase, we apply column operations to ∂_1 to remove such redundant edges and row operations to ∂_2 to collapse any adjacency to the redundant edges down to one remaining representative. We then repeat this procedure at each higher dimension. This elimination procedure is equivalent to a merge operation but applied to the dual complex. Finally, we remove any k-cells with fewer than $k + 1$ faces.

The results of the edge collapse operation on a simple and a less-simple case are shown in figure 12. The first case is straightforward and produces well-shaped triangles. In the second case, we can compute the collapsed topology, but the resulting geometry is invalid because it introduces triangles of zero or negative area. Mesh simplification algorithms have to detect and reject these cases [2].

6. CONCLUSION

Doubly-connected edge lists have historically been popular for mesh generation because they offer a simple interface for traversing the topology [24]. Here we propose that boundary operators are a viable choice of representation if the goal is to perform topological transformations. This idea has appeared before, for example in the work of DiCarlo and others [9].

Boundary operators are only necessary for representing a small patch of the topology at a time. Once the transformed patch is computed, it can be translated back to a set of simplices. We adopted this approach in our implementation of each of the algorithms described above. Boundary operators are useful for de-

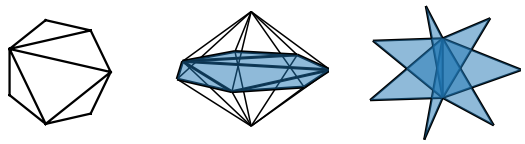

Figure 10: A randomly-generated triangulation of a polygon, its suspension, and the interior facets after multi-face removal / edge insertion.

```
def edge_collapse(
    D: Topology, vertex_ids: int
) -> Topology:
    # Form the matrix for the initial edge collapse
    P = np.eye(D[1].shape[0], dtype=np.int8)
    P[np.ix_(vertex_ids, vertex_ids)] = \
        np.array([[1, 1], [0, 0]])

    # Remove redundant and empty cells
    E = [D[0].copy(), P @ D[1], *D[2:]]
    for k in range(1, n := len(D) - 1):
        A, B = make_reduction_matrices(E[k])
        E[k], E[k + 1] = E[k] @ A, B @ E[k + 1]

    empty_cell_ids = \
        nonzero(count_nonzero(E[-1], axis=0) <= n)
    E[-1] = np.delete(E[-1], empty_cell_ids, axis=1)
    return E
```

Figure 11: The source code the edge collapse transformation. The routine `make_reduction_matrices` (not shown) forms the row and column operations to remove redundant cells.

scribing non-simplicial intermediate states of a transformation; they are not space-optimal for describing an entire simplicial complex.

We focused here on serial algorithms to produce a simplicial mesh. The transformations that we defined here are also applicable to hex or poly meshing. Moreover, large-scale 3D finite element analysis requires the mesh to be decomposed into subdomains that are then distributed to different processes. The connectivity between these subdomains does not necessarily have a simplicial or cubical structure. We can then describe the overall topology hierarchically – a simplicial representation inside a domains and a polytopal complex between domains.

We showed how to implement two common topological operations for tetrahedral mesh improvement. These operations assume a particular structure for the tetrahedra that they operate on. The *small polyhedron reconnection* (SPR) approach, on the other hand, works on any set of input tetrahedra at the cost of a more expensive search step [8]. The linear-algebraic representation might be able to accelerate SPR by instead bisecting cavities into smaller polytopes that only become simplicial by the end. The branch and bound procedure over polytopes might be faster or easier to implement than its counterpart over tetrahedra.

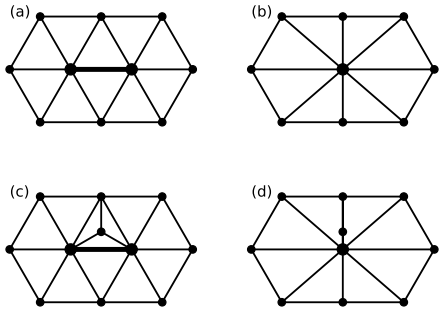

Figure 12: Edge collapsing before (a) and after (b) in the ideal case. In a bad case (c) we can compute the topology but the resulting geometry is always invalid (d).

We evaluated our code correctness by testing that we could compute convex hulls, Delaunay triangulations, etc. We did not give formal proofs that each operation is guaranteed to result in a topologically valid end state. For example, applying the face-split transformation assumes that a certain integer linear system is solvable. Are there cases where this system has either no solution or multiple solutions but the polytope is splittable? Are there cases where the system has a unique solution but not one that produces the desired final topology? We found empirically that the linear system was solvable in all the cases that we tried and gave the expected results. But an interesting future direction would be to use formal methods to verify these transformations using established results, for example on total unimodularity of boundary matrices [25].

When the objects of study can be represented as linear operators, we can apply linear algebraic reasoning to define transformations and verify that they preserve all of the important invariants. The condition in equation (5) that the product of two boundary operators is zero is a powerful invariant for ensuring the validity of the underlying topology.

ACKNOWLEDGEMENTS

Thanks to Matt Knepley, Tobin Isaac, Mauricio del Razo, and Leila de Floriani for many helpful discussions.

References

[1] Cheng S.W., Dey T.K., Shewchuk J. *Delaunay mesh generation*. CRC Press Boca Raton, 2013

[2] Guéziec A. "Surface simplification with variable tolerance." *Second Annual Symposium on Medical Robotics and Computer Assisted Surgery, 1995*. 1995

[3] Mattson T., Bader D., Berry J., Buluc A., Dongarra J., Faloutsos C., Feo J., Gilbert J., Gonzalez J., Hendrickson B., et al. "Standards for graph algorithm primitives." *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–2. IEEE, 2013

[4] Freitag L.A., Ollivier-Gooch C. "Tetrahedral mesh improvement using swapping and smoothing." *International Journal for Numerical Methods in Engineering*, vol. 40, no. 21, 3979–4002, 1997

[5] Klingner B.M., Shewchuk J.R. "Aggressive tetrahedral mesh improvement." *Proceedings of the 16th international meshing roundtable*, pp. 3–23. Springer, 2008

[6] Misztal M.K., Bærentzen J.A., Anton F., Erleben K. "Tetrahedral mesh improvement using multiface retriangulation." *Proceedings of the 18th international meshing roundtable*, pp. 539–555. Springer, 2009

[7] Hu Y., Zhou Q., Gao X., Jacobson A., Zorin D., Panozzo D. "Tetrahedral meshing in the wild." *ACM Trans. Graph.*, vol. 37, no. 4, 60, 2018

[8] Marot C., Verhetsel K., Remacle J.F. "Reviving the search for optimal tetrahedralizations." *Proceedings of the 28th International Meshing Roundtable. Zenodo, Buffalo, New York, USA*, 2020

[9] DiCarlo A., Milicchio F., Paoluzzi A., Shapiro V. "Solid and physical modeling with chain complexes." *Proceedings of the 2007 ACM symposium on Solid and physical modeling*, pp. 73–84. 2007

[10] Mueller-Roemer J.S., Altenhofen C., Stork A. "Ternary sparse matrix representation for volumetric mesh subdivision and processing on GPUs." *Computer Graphics Forum*, vol. 36, pp. 59–69. Wiley Online Library, 2017

[11] Paoluzzi A., Shapiro V., DiCarlo A., Furiani F., Martella G., Scorzelli G. "Topological computing of arrangements with (co) chains." *ACM Transactions on Spatial Algorithms and Systems (TSAS)*, vol. 7, no. 1, 1–29, 2020

[12] Gelfand S.I., Manin Y.I. *Homological algebra*, vol. 38. Springer Science & Business Media, 1994

[13] Gawrilow E., Joswig M. "Polymake: a framework for analyzing convex polytopes." *Polytopes—combinatorics and computation*, pp. 43–73. Springer, 2000

[14] Björner A., Lutz F.H. "Simplicial manifolds, bistellar flips and a 16-vertex triangulation of the Poincaré homology 3-sphere." *Experimental Mathematics*, vol. 9, no. 2, 275–289, 2000

[15] Effenberger F., Spreer J. "simpcomp: a GAP toolbox for simplicial complexes." *ACM Communications in Computer Algebra*, vol. 44, no. 3/4, 186–189, 2011

[16] Hatcher A. *Algebraic Topology*. Cambridge University Press, 2002

[17] Massey W.S. *A basic course in algebraic topology*, vol. 127. Springer, 2019

[18] Pachner U. "Shellings of simplicial balls and pl manifolds with boundary." *Discrete mathematics*, vol. 81, no. 1, 37–47, 1990

[19] Pachner U. "PL homeomorphic manifolds are equivalent by elementary shellings." *European Journal of Combinatorics*, vol. 12, no. 2, 129–145, 1991

[20] Casali M.R. "A note about bistellar operations on PL-manifolds with boundary." *Geometriae Dedicata*, vol. 56, no. 3, 257–262, 1995

[21] Kannan R., Bachem A. "Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix." *siam Journal on Computing*, vol. 8, no. 4, 499–507, 1979

[22] Anglada M.V. "An improved incremental algorithm for constructing restricted Delaunay triangulations." *Computers & Graphics*, vol. 21, no. 2, 215–223, 1997

[23] Papadakis M., Bernstein G.L., Sharma R., Aiken A., Hanrahan P. "Seam: Provably safe local edits on graphs." *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, 1–29, 2017

[24] Guibas L., Stolfi J. "Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams." *ACM transactions on graphics (TOG)*, vol. 4, no. 2, 74–123, 1985

[25] Dey T.K., Hirani A.N., Krishnamoorthy B. "Optimal homologous cycles, total unimodularity, and linear programming." *Proceedings of the forty-second ACM symposium on Theory of computing*, pp. 221–230. 2010