The Cutting Pattern Problem for Tetrahedral Mesh Generation

Xiaotian Yin[1], Wei Han[1], Xianfeng Gu[2], and Shing-Tung Yau[1]

[1] Mathematics Department, Harvard University, MA, U.S.A. {xyin,weihan,yau}@math.harvard.edu
[2] Computer Science Department, Stony Brook University, NY, U.S.A. gu@cs.sunysb.edu

Summary. In this work we study the following cutting pattern problem. Given a triangulated surface (i.e. a two-dimensional simplicial complex), assign each triangle with a triple of ± 1, one integer per edge, such that the assignment is both complete (i.e. every triangle has integers of both signs) and consistent (i.e. every edge shared by two triangles has opposite signs in these triangles). We show that this problem is the major challenge in converting a volumetric mesh consisting of prisms into a mesh consisting of tetrahedra, where each prism is cut into three tetrahedra. In this paper we provide a complete solution to this problem for topological disks under various boundary conditions ranging from very restricted one to the most flexible one. For each type of boundary conditions, we provide efficient algorithms to compute valid assignments if there is any, or report the obstructions otherwise. For all the proposed algorithms, the convergence is validated and the complexity is analyzed.

Key words: cutting pattern, graph labeling, tetrahedral mesh, prism

1 Introduction

1.1 Motivation

Volumetric meshes are widely used in various areas, such as geometric modeling, computer aided design and physical simulation. Tetrahedral mesh is one of the most commonly used representations, because it is a simplicial complex that allows many topological algorithms and finite element solvers to be applied. There are many methods to generate tetrahedral meshes, such as Advancing Front techniques [1, 2], Octree methods [3] and Voronoi Delaunay based methods [4, 5, 6].

In this work we look at a specific problem: converting a *prismal mesh* to a tetrahedral mesh without introducing new vertices. A prismal mesh consists of a set of prisms, which are volumetric elements bounded by two triangles from top and bottom and three quadrilaterals from sides. A typical example is a 3D space-time slab, which is generated by extruding a 2D triangular mesh in temporal direction. It has been shown in [7] that such a prismal mesh needs to be split into a tetrahedral mesh to adapt existing unstructured tetrahedral solvers. It has also been shown in [8, 9] that such a conversion is necessary in computer graphics, especially when one wants to apply efficient algorithms for volume rendering and iso-contouring that exist for purely tetrahedral meshes.

The conversion from a prismal mesh to a tetrahedral mesh has been addressed in a lot of work, such as [8, 9, 10, 11, 12]. However, none of these methods allows the user to control the triangulation on the boundary of the output volumetric mesh. In another word, they only solve the free boundary version of this problem. In this work, we aim at tackling this problem in a more general setting that includes both free boundary and fixed boundary cases, where the latter incurs more challenges than the former.

In order to do this conversion without introducing new vertices, an intuitive way is to cut each prism into three tetrahedra using two planes passing through certain corners of the prism (figure 1b-d). The options of cutting a prism can be encoded as a 3-tuple of ± 1, where $+1$ (or -1) means the corresponding wall is sliced along the diagonal (figure 1a left) or anti-diagonal (figure 1a right). Therefore, the conversion of the whole volumetric mesh can be reduced to assigning the underlying triangular surface mesh with a set of 3-tuples of ± 1.

This ± 1 assigning task is challenging in multiple aspects.

- In a local consideration, not every combination of three ± 1 represents a valid cutting, see figure 1(e) for a counterexample. A valid cutting should have both $+1$ and -1 in each triangle.
- In a global consideration, the cutting must be consistent; i.e. for any pair of prisms adjacent to each other by a quadrangular wall, their cutting lines must meet on that wall. It means the cutting options in the bottom triangles should have opposite signs across any common edge shared by two triangles.
- From the user's point of view, the user should be able to control the cutting options on the boundary surface of the volumetric mesh, and the internal cutting should be subject to the boundary values.

Considering all these requirements, we convert this mesh conversion problem to an equivalent 2D graph problem in the next section.

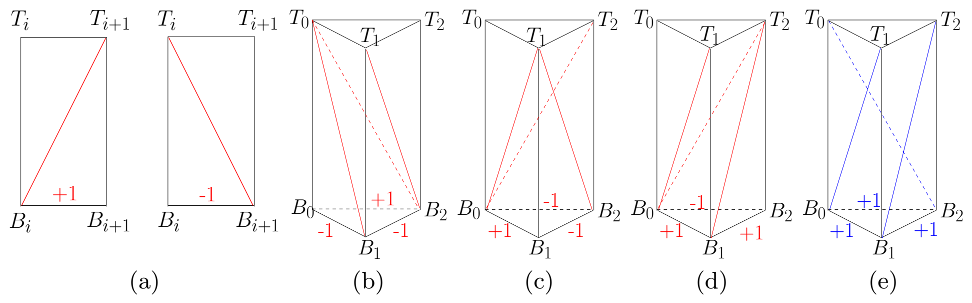

Fig. 1. Options of cutting a prism into tetrahedra are encoded by assigning +1 or -1 to each edge of the bottom triangle $B_0B_1B_2$ (a). A valid cutting pattern consists of both signs (b, c and d), while an invalid one consists of only one sign (e).

1.2 Problem Statement

The major problem that we need to solve is as follows.

Definition 1 (The Cutting Pattern Problem). *Given a triangular mesh D, assign each triangle with a 3-tuple of ± 1, one integer per edge, such that:*

- *Every triangle has both $+1$ and -1 in its 3-tuple;*
- *Every edge shared by two adjacent triangles is assigned with opposite values in these triangles;*

In practice, the assignments on boundary edges may or may not be prescribed. If they are prescribed, there are various ways to set the boundary values. In this work, we consider the following types of boundary conditions, varying from the most restricted one to the most flexible one.

Definition 2 (Boundary Conditions). *The cutting pattern problem can be subject to the following types of boundary conditions:*

1. *Restricted boundary: Every boundary edge of D has a prescribed ± 1 assignment. In particular,*
   - *For any triangle with only one edge exposed on the boundary of D, the prescribed assignment for this boundary edge can take an arbitrary value of ± 1;*
   - *For any triangle with at least two edges exposed on the boundary of D, two of such boundary edges must have opposite prescribed assignments.*
2. *General boundary: Every boundary edge of D has a prescribed assignment that can take an arbitrary value of ± 1;*
3. *Free boundary: None of the boundary edges has a prescribed assignment.*

These boundary conditions reflect different levels of user control in the original problem of triangulating a prismal mesh, that is, whether and how one wants to control the boundary triangulation of the output tetrahedral mesh. And as discussed later, different boundary conditions result in different solvability of the problem and different ways to find a solution if there is any. If the boundary triangulation is prescribed without any special constraint, it corresponds to the general boundary condition and the problem may or may not have a solution. If the boundary triangulation is prescribed and satisfies the restricted boundary condition, the problem can always be solved using our algorithm. If the user does not want to specify the boundary triangulation, our algorithm can automatically specify the boundary and is guaranteed to find a triangulation of the whole volume.

1.3 Contributions

The cutting pattern problem is the major challenge of converting a prismal mesh to a tetrahedral mesh. If the former can be solved, so is the latter. Meanwhile, the cutting pattern problem itself is an interesting graph problem that can be categorized as graph labeling (see [13] for a survey). To our best knowledge, however, this specific problem has not been addressed in the literature.

In this paper we make efforts to tackle this problem for topological disks with a single boundary, and propose a complete set of solutions under all kinds of boundary conditions. In particular:

1. For restricted boundary conditions, we show that solution always exists and can be found using an efficient algorithm proposed in the paper. (section 2)
2. For general boundary conditions, we show that solution exists if there is no cutting pattern obstruction (definition 6). We also propose an efficient algorithm that either reports such an obstruction if there is any or outputs a valid solution otherwise. (section 3)
3. For free boundary conditions, we show that the problem can be always turned into a restricted one, therefore solution always exists and can be found using an algorithm modified from the restricted one. (section 4)

For the algorithm proposed for each case, the convergence is validated and the complexity is analyzed.

1.4 Definitions and Notations

Here we introduce the concepts and notations used in later elaborations.

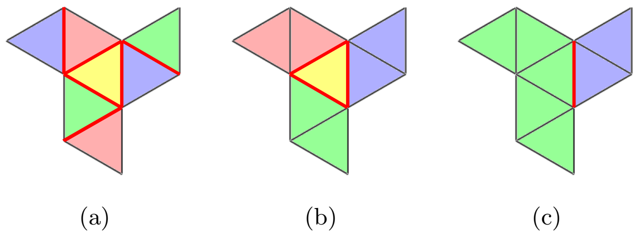

Fig. 2. Separating edges (a), breaking edges (b) and a minimal set of breaking edges (c).

In a triangular mesh D consisting of a set of triangles $F = \{f_1, f_2, \cdots, f_n\}$, the *edge-valence* (or *valence* in short) of a triangle $f_i \in F$ is the number of triangles adjacent to f_i across some edges. It should be a non-negative integer that is at most 3. A triangle is called a *dangling triangle* if its valence is 1. A dangling triangle has exactly two edges exposed on the boundary of D. A triangle is called a *singular triangle* if its valence is 0. In a singular triangle, all three edges are on the boundary of D.

For an edge e shared by two triangles $f_1, f_2 \in D$, we say it is of (i, j)-*type* if f_1 and f_2 have valence i and j respectively $(1 \le i \le j \le 3)$. An edge e is called a *separating edge* if it is shared by two triangles and the end vertices of e are both on the boundary of D (figure 2a). An edge e is called a *breaking edge* if it is a separating edge and at least one of its adjacent triangles has valence 3 (figure 2b and 2c).

A *sub-component* C is a sub-mesh of D that is *edge-connected* (or *connected* in short), meaning that every triangle in C is adjacent to at least one other triangle in C across a common edge.

Given an edge e in triangle f, its *assignment* (denoted as $a\,(e, f)$) is an integer 0 (*unsolved*) or ± 1 (*solved*). An edge e is *completely solved* (or *solved* for short) if and only if it is solved in every triangle enclosing e. An triangle f is *completely solved* (or *solved* for short) if and only if all three edges of f have been solved. A mesh D (or sub-component C) is *completely solved* (or *solved* for short) if and only if all its triangles are solved. Obviously, a valid cutting pattern over a given mesh D is a set of ± 1 assigned to every edge in every triangle such that both completeness and consistency are satisfied.

Given a mesh D (or sub-component C), its *boundary assignment* is the set of assignments of its boundary edges. A boundary assignment is *safe* if and only if every dangling or singular triangle f (if there is any) has two boundary edges with opposite non-zero assignments $+1$ and -1 (figure 3a). It is *moderately dangerous* if and only if it is not safe and at least two boundary edges have opposite non-zero assignments (figure 3b). It is *extremely dangerous* if and only if it is not safe and all the non-zero assignments on boundary edges equal to a single value $a \in \{+1, -1\}$ (figure 3c), which is called the *forbidding value* of this boundary assignment. The last two cases are both called *dangerous* boundary assignments.

2 Solution for Restricted Boundary Conditions

For the cutting pattern problem under restricted boundary conditions, we propose the following algorithm to look for a valid cutting pattern. We will show that this algorithm always ends up

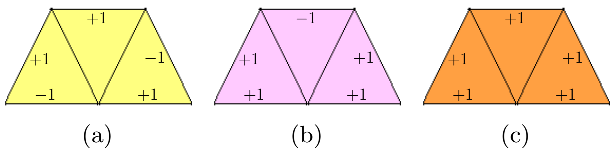

Fig. 3. Sub-components with different types of boundary assignments: (a) safe, (b) moderately dangerous, (c) extremely dangerous.

with a valid cutting pattern and therefore gives a constructive proof for the existence of solutions under restricted boundary conditions.

Given a topological disk represented as a triangular mesh D, the goal is to compute a valid cutting pattern for D, i.e. to assign every edge in every triangle with an integer $b \in \{+1, -1\}$. Upon input, all the boundary edges have prescribed assignments ± 1 while all the other edges are initialized with assignment 0. The input mesh D is processed iteratively. In each iteration, the unsolved part of D is partitioned into a set of sub-components that have special types, and each sub-component is either completely solved or partially solved. After this, the remaining unsolved triangles will be brought into another iteration. Repeat this process until the whole mesh is solved.

Algorithm 2.1 (Cutting Pattern Algorithm - I)

- *Input: A triangular mesh D with a restricted boundary condition.*
- *Output: A valid cutting pattern for D.*
- *Procedures:*
  1. *Initialize all the internal edges of D with assignment 0 (i.e. unsolved).*
  2. *Repeat the following procedures on D until no unsolved triangle left.*
     a) *Partition D into a minimal set of sub-components of basic types and solve the newly exposed boundary edges (section 2.1).*
     b) *Solve each sub-component completely or partially according to its type (2.2 and 2.3).*
     c) *If there is no unsolved triangle, exit; otherwise, set D to be the unsolved part and go back to step 2a.*

2.1 Constructing Sub-components

In step 2a we partition mesh D into a set of sub-components $\{C_i \mid 1 \leq i \leq k\}$ that are *triangle-disjoint* (i.e. there is no triangle $f \in S_i \cap C_j$, $i \neq j$) and *covering* (i.e. $D = \cup_{i=1}^{k} C_i$). Figure 4 shows several examples of mesh partitioning. Furthermore, each sub-component should have one of the following basic types:

Definition 3 (Classification of Sub-components).

- *Aggregated sub-component (figure 6): a sub-component where every triangle has edge-valence at least 2;*
- *Linear sub-component (figure 5): a sub-component consisting of a sequence of triangles, where two triangles at the ends have edge-valence 1 (i.e. dangling triangles) and all the others have edge-valence 2;*
- *Singular sub-component: a sub-component consisting of only one triangle with edge-valence 0 (i.e. a singular triangle).*

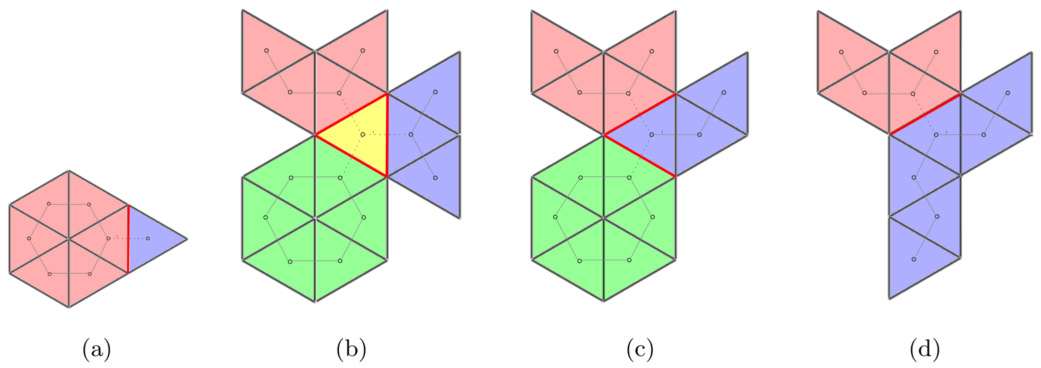

Fig. 4. Partition a mesh into a minimal set of sub-components. (a) to (d) show several different meshes and their partitions.

One can get such a partition by slicing D along all the breaking edges. However, such a partition may have unnecessary small pieces (figure 2b). What we need in the algorithm is a *minimal set of sub-components* of basic types, or equivalently a set of *maximal sub-components*, meaning that the union of any two or more sub-components in this set is not of any basic type (figure 2c). Therefore, we need to find *a minimal set of breaking edges*, meaning that a smallest set of breaking edges so that the resulting sub-components are all maximal. This can be done by addressing all the breaking edges in D and then exclude unnecessary ones by checking other edges in two enclosing triangles.

The major reason of choosing a minimal set rather than the full set of breaking edges is to simplify the algorithm and the analysis. For example, as we will show later, this will result in simple cases to be considered when solving a breaking edge, and therefore simplify the proof of convergence of the algorithm.

Once the minimal set of breaking edges (denoted as E_b) is addressed, we partition the given mesh into a set of sub-components, and every edge $e \in E_b$ will be exposed on the boundary of two sub-components and needs to be solved here. In order to solve edges in E_b, we first classify them based on their positions in the mesh and the types of their adjacent sub-components.

Definition 4 (Classification of Breaking Edges). *A breaking edge e shared by sub-components C_1 and C_2 can only have one of the following types:*

- *A2A: if both C_1 and C_2 are aggregated;*
- *A2Lm (or Lm2A): if C_1 is aggregated and C_2 is linear, and e belongs to a triangle in the middle of C_2;*
- *A2Le (or Le2A): if C_1 is aggregated and C_2 is linear, and e belongs to a triangle at one end of C_2;*
- *A2S (or S2A): if C_1 is aggregated and C_2 is singular;*
- *Lm2Lm: if both C_1 and C_2 are linear, and e belongs to a middle triangle in C_1 and another middle triangle in C_2;*
- *Lm2Le (or Le2Lm): if both C_1 and C_2 are linear, and e belongs to a middle triangle in C_1 and an end triangle in C_2;*
- *Lm2S (or S2Lm): if C_1 is linear and C_2 is singular, and e belongs to a middle triangle in C_1;*

Note that these are not all the combinations of two sub-components with basic types. However, other combinations, including Le2Le, Le2S and S2S, are impossible in our algorithm.

For example, a Le2Le combination means that two linear sub-components connect to each other at one end and can thus form a larger linear sub-component; therefore it breaks the rule of maximal sub-components and is not a valid combination. The combination of Le2S and S2S can be excluded in a similar way.

Once the breaking edges are classified, they can be solved according to their types using the following procedure.

Procedure 2.1 (Solving A Breaking Edge) *Given a breaking edge e shared by two triangles f_1 and f_2 in sub-components C_1 and C_2 respectively, do the following:*

- *If e is A2A (i.e. between aggregated C_1 and aggregated C_2): Assign e with $+1$ on C_1 side and -1 on C_2 side.*
- *If e is A2Lm or Lm2Lm: similar to the A2A case.*
- *If e is A2Le (i.e. between aggregated C_1 and linear C_2): We will first solve e on C_2 side and then on C_1 side. On C_2 side, e is a boundary edge in an end triangle f_2. Let e' be the other boundary edge in f_2;*
  - *If e' is already solved in f_2, suppose its assignment is $a'_2 \in \{+1, -1\}$, then assign e in f_2 with $a_2 = -a'_2$.*
  - *If e' is not solved in f_2 yet, then assign e with an arbitrary value $a_2 \in \{+1, -1\}$ (so that e' can be solved later with assignment $a'_2 = -a_2$).*

  Once e receives assignment $a_2 \in \{+1, -1\}$ in $f_2 \in C_2$, we assign e with $a_1 = -a_2$ in $f_1 \in C_1$.
- *If e is Lm2Le: similar to the A2Le case.*
- *If e is A2S (i.e. between aggregated C_1 and singular C_2): We will first solve e on C_2 side and then on C_1 side. On C_2 side, there is only one triangle $f_2 \in C_2$, and e is one of the three edges in f_2, let e' and e'' be the other two. Then within f_2,*
  - *If at least one other edge (say e') is already solved in f_2, suppose its assignment is $a'_2 \in \{+1, -1\}$, then assign e with $a_2 = -a'_2$ in f_2.*
  - *If neither e' nor e'' is solved in f_2, then assign e with an arbitrary value $a_2 \in \{+1, -1\}$ (so that e' and e'' can be solved later with assignment ${a_2}' = -a_2$ and ${a_2}'' = -a_2$).*

  Once e receives assignment $a_2 \in \{+1, -1\}$ in $f_2 \in C_2$, we assign e with $a_1 = -a_2$ in $f_1 \in C_1$.
- *If e is Lm2S: similar to the A2S case.*

This procedure guarantees that every breaking edge in E_b receives opposite assignments on two sides, and every resulting sub-component has a safe boundary assignment. After this process, every sub-component C has a solved boundary that satisfies the restricted boundary condition and is ready to be solved for the inner edges based on the type of C. For a singular sub-component, all the edges are on the boundary and are already solved, no further process is needed. For linear and aggregated sub-components, their inner edges need extra efforts to solve (see section 2.2 and 2.3 respectively).

2.2 Solving Linear Sub-components

A linear sub-component C consists of a sequence of triangles,

$$f_0,\ f_1,\ \cdots,\ f_n$$

where f_0 and f_n are end triangles (with valence 1) that each has two edges $e^B_{i,1}$ and $e^B_{i,2}$ on the boundary of C ($i = 0, n$), while any other f_i is a middle triangle (with valence 2) that has only one edge e^B_i on the boundary of C ($1 \le i \le n-1$). These boundary edges are already solved. Meanwhile, each pair of consecutive triangles f_{i-1} and f_i ($1 \le i \le n$) share a common internal edge e^I_i; these internal edges need to be solved here.

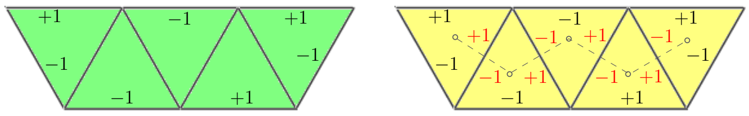

Fig. 5. A linear sub-component can be completed solved via a stabbing line (dotted).

In order to solve the internal edges, we stab the sequence of triangles with a line from f_0 all the way to f_n (figure 5). Let's take an arbitrary value $b \in \{+1, -1\}$ as an initial assignment. When the line penetrates an internal edge e_i^I, we assign b and $-b$ to its entrance side (in f_{i-1}) and exit side (in f_i) respectively. In the end, all the internal edges are completely solved and the whole sub-component is thus solved.

2.3 Solving Aggregated Sub-components

Since every sub-component in our algorithm is guaranteed to be a topological disk, every aggregated sub-component C must have only one boundary, and there must be a layer of most outside triangles (the *frontier*) that can be distinguished from the rest (the *interior*) in the following way (see figure 6).

Consider the dual graph C^*. Any triangular face $f_i \in C$ corresponds to a vertex $v_i^* \in C^*$, edge $e_j \in C$ to edge $e_j^* \in C^*$, vertex $v_k \in C$ to face $f_k^* \in C^*$. Note that C is a topological disk and every face $f_i \in C$ has at most one boundary edge. Accordingly, C^* is also a topological disk and its boundary is a loop connecting n vertices sequentially: $v_1^*, v_2^*, \cdots, v_n^*, v_1^*$. This boundary loop of C^* corresponds to a looping sequence of n triangular faces in C:

$$f_1, f_2, \cdots, f_n, (f_1)$$

These triangles are called the *frontier* of C, denoted as C^F, and the remaining triangles the *interior*, denoted as C^I. In this step we will solve C^F completely.

Edges in C^F can be grouped into three sets E^B, E^I and E^O, where E^B consists of edges on the boundary loop of C, E^O consists of edges on the border between C^F and C^I, and E^I consists of edges e_i^I shared by consecutive triangles f_{i-1} and f_i in C^F. Edges in E^B have already been solved, while the other two sets need to be solved here.

To solve E^I, we use a stabbing process similar to that for linear sub-components. We take arbitrary $b \in \{+1, -1\}$ as the initial assignment, stab the sequence of triangles in C^F with a circle; when penetrating an edge $e_i^I \in E^I$, assign it with b on the f_{i-1} side and $-b$ on the f_i side.

To solve E^O, we first consider the C^I side, where E^O serves as the boundary of C^I. Check every face $f \in C^I$ that contains at least one edge in E^O. If f contains only one such edge, we assign it with $+1$. If f contains at least two such edges, we assign $+1$ to one of them and -1 to the rest. Once an edge $e \in E^O$ is solved on the C^I side with assignment b, we assign it on the C^F side with an opposite value $-b$.

Once all the edges in $E^I \cup E^O$ are solved, the whole frontier C^F is completely solved, while C^I is only solved on its boundary and the remainder will be recursively solved later.

2.4 Validation and Analysis

In this part we discuss the convergence and complexity of algorithm 2.1. From the above discussion about the algorithm, several invariants of the algorithm can be induced.

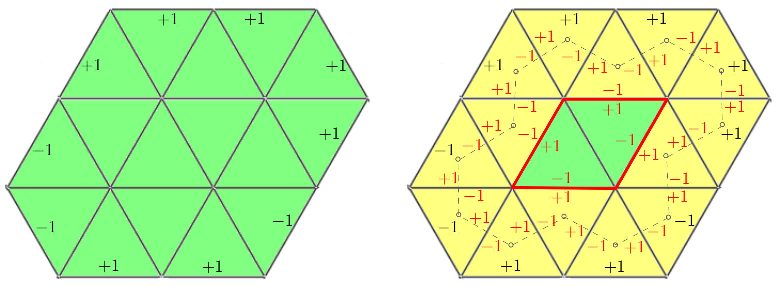

Fig. 6. An aggregated sub-component is only solved in the frontier (in yellow); the interior (in green) is left to later iterations.

Definition 5 (Invariants of Algorithm 2.1).

- *Completeness: whenever a triangle is completely solved, its 3-tuple of edge assignments should consist of both* $+1$ *and* -1*;*
- *Consistency: whenever an internal edge (shared by two triangles) is completely solved, it should have opposite assignments in two enclosing triangles;*
- *Safeness: the unsolved part (a sub-mesh) should have a safe boundary assignment at the beginning of step 2a, and every sub-component should also have a safe boundary assignment at the end of step 2a.*

We can prove that all these invariants hold in the algorithm. To do this, we need to guarantee that none of these invariants can be violated in our algorithm.

Completeness and consistency can only be violated when an edge or a face is solved. In algorithm 2.1 there are three procedures that need to be checked. The first is in step 2a, where breaking edges between adjacent sub-components are solved (section 2.1). The second and third are both in step 2b, where linear and aggregated sub-components are solved (section 2.2 and 2.3). These violations are guaranteed not to happen by Lemma 1, 2 and 3 respectively.

Safeness can only be violated when edges are newly exposed to the boundary of a sub-mesh or a sub-component. In algorithm 2.1 there are two procedures that need to be checked. The first is in step 2a, where breaking edges between adjacent sub-components are solved (section 2.1). The second is in step 2b, where aggregated sub-components are solved in the frontier (2.3). These violations are guaranteed not to happen by Lemma 4 and 5 respectively.

Lemma 1. *The completeness and consistency invariants hold when solving a breaking edge using procedure 2.1.*

Proof. In procedure 2.1, we solve all the breaking edges along which the unsolved sub-mesh is partitioned into basic type sub-components. Given such a breaking edge e shared by $f_1 \in C_1$ and $f_2 \in C_2$,

- If e is A2A: e is assigned with $+1$ and -1 in $f_1 \in C_1$ and $f_2 \in C_2$ respectively, therefore consistency holds on e.
- If e is A2Lm or Lm2Lm: similar to the A2A case.
- If e is A2Le: On C_2 side, f_2 is an end triangle that has two boundary edges, e and e'. In f_2, e always receives an assignment opposite to that of e', therefore completeness holds for f_2. On C_1 side, e in f_1 always receives an assignment opposite to that of e in f_2, therefore consistency holds for e.

- If e is Lm2Le: similar to the A2Le case.
- If e is A2S: On C_2 side, f_2 is a singular triangle that all three edges are on the boundary. In f_2, e always receives an assignment opposite to that of another edge $e' \in f_2$, therefore completeness holds for f_2. On C_1 side, e in f_1 always receives an assignment opposite to that of e in f_2, therefore consistency holds for e.
- If e is Lm2S: similar to the A2S case.

Lemma 2. *The completeness and consistency invariants hold when solving a linear sub-component.*

Proof. In section 2.2 we use a stabbing procedure to solve all the internal edges $\{e_1^I,\ e_2^I,\ \cdots,\ e_n^I\}$ in a linear sub-component C and thus solve all the triangles $\{f_0,\ f_1,\ \cdots,\ f_n\}$.

Every internal edge $e_i^I = f_{i-1} \cap f_i$ $(1 \leqslant i \leqslant n)$ receives assignment b and $-b$ in f_{i-1} and f_i respectively, where b is an initial assignment arbitrarily chosen from $\{+1, -1\}$. Therefore consistency holds for every newly solved e_i^I.

Every middle triangle f_i $(1 \leqslant i \leqslant n-1)$ has two internal edges e_i^I and e_{i+1}^I and receives opposite assignments on them during the stabbing. In addition, due to the safeness invariant, every end triangle (f_0 and f_n) already has opposite assignments on two boundary edges before the stabbing. Therefore completeness holds for every triangle in C.

Lemma 3. *The completeness and consistency invariants hold when solving an aggregated sub-component.*

Proof. In section 2.3 we solve the internal edges E^I of the frontier layer C^F and the edges E^O between the frontier C^F and the interior C^I, thus all the faces $\{f_1, f_2, \cdots, f_n\}$ in C^F are solved.

E^I are solved by a stabbing procedure, where every $e_i^I = f_{i-1} \cap f_i$ receives assignment b and $-b$ in f_{i-1} and f_i respectively, where b is an initial assignment arbitrarily chosen from $\{+1, -1\}$. Therefore consistency holds for every edge in E^I.

E^O are first solved on C^I side and then on C^F side, and every edge in E^O receives opposite assignments on two sides. Therefore consistency holds for every edge in E^O as well.

Every triangle f_i in C^F has two internal edges e_i^I and e_{i+1}^I and receives opposite assignments on them during the stabbing. Therefore completeness holds for every triangle solved in this process.

Lemma 4. *The safeness invariant holds when solving a breaking edge using procedure 2.1.*

Proof. In procedure 2.1, we partition the unsolved sub-mesh along a set of breaking edges. These edges are exposed on the boundary of resulting sub-components. Given such a breaking edge e shared by $f_1 \in C_1$ and $f_2 \in C_2$, we need to check whether their assignments violate the safeness of the boundary of C_1 and C_2.

- If e is A2A: since C_1 aggregated, f_1 is not singular or dangling, therefore any assignment will be safe; i.e. safeness holds for C_1. The same argument holds for C_2 as well.
- If e is A2Lm or Lm2Lm: similar to the A2A case.
- If e is A2Le: On C_1 side, f_1 is not singular or dangling, therefore any assignment will be safe. Therefore safeness holds for C_1. On C_2 side, f_2 is an end (dangling) triangle containing two boundary edges e and e', they receive opposite assignments. By definition this does not violate the safeness on the boundary of C_2.
- If e is Lm2Le: similar to the A2Le case.
- If e is A2S: On C_1 side, f_1 is not singular or dangling, therefore any assignment will be safe. Therefore safeness holds for C_1. On C_2 side, f_2 is a singular triangle containing three boundary edges, two of them receive opposite assignments. By definition this does not violate the safeness on the boundary of C_2.

- If e is Lm2S: similar to the A2S case.

Lemma 5. *The safeness invariant holds when solving an aggregated sub-component.*

Proof. In section 2.3 we completely solve the frontier C^F of a sub-component C and expose the edges in E^O as the boundary of the interior C^I. Those new boundary edges E^O are solved on C^I side. Recall that for face $f \in C^I$ that has at least two boundary edges (in E^O), this f receives assignments of both signs. By definition this gives a safe boundary assignment for C^I, therefore safeness holds for C^I.

Based on the above lemmas, we have the following conclusion about the convergence of the algorithm:

Theorem 1. *Algorithm 2.1 always terminates (due to the safeness invariant); at termination, all the triangles are solved and the resulting assignments represent a valid cutting pattern (due to the completeness and consistency invariants).*

Now consider the time complexity of this algorithm. Let $|V|$, $|E|$ and $|F|$ be the number of vertices, edges and triangles in the input mesh D. The cost of the algorithm lies in the following aspects.

- Addressing breaking edges to partition the unsolved part of D. Since a breaking edge is always connecting boundary vertices, we only need to check the boundary vertices of C; and to avoid unnecessary breaking edges, we only need to check its two adjacent triangles. Therefore the cost in each iteration is bounded by the number of boundary vertices of the unsolved part of D in that iteration. Since every vertex in the original mesh D appears at most once on the boundary of the unsolved part, the total cost for partitioning is bounded by $O(|V|)$.
- Solving edges. This happens when we solve a linear sub-component completely or solve an aggregated sub-component in its frontier. From an overall view, every edge in the original mesh D is solved only once; therefore this part is bounded by $O(|E|)$.
- Addressing the frontier of aggregated sub-components. From an overall view, every triangle in D will appear in the frontier of an aggregated sub-component for at most once; therefore this part is bounded by $O(|F|)$.

In summary, the time complexity of algorithm 2.1 is $O(|V| + |E| + |F|)$, which is linear.

3 Solution for General Boundary Conditions

A general boundary condition is different to a restricted one in that the boundary assignment for the input mesh is allowed to be dangerous, and it degenerates to the latter if the boundary assignment is safe. Inspired by this fact, we design an algorithm that tries to resolve all the danger and tries to turn this general problem into a restricted one. If the danger cannot be removed from some part of the mesh, the algorithm will report this part as an obstruction and terminate. Otherwise, it will continue to run as the restricted algorithm.

Algorithm 3.1 (Cutting Pattern Algorithm - II)

- *Input: A triangular mesh D with a general boundary condition.*
- *Output: A valid cutting pattern for D or an unsolvable sub-mesh.*
- *Procedures:*

1. *Check the boundary assignment. If it is safe, jump to step 4; otherwise, do the following.*
2. *Partition mesh D into a minimal set of basic type sub-components, try to solve the breaking edges between sub-components. If any obstruction is detected, report it and exit. (section 3.1)*
3. *Solve moderately dangerous sub-components. (section 3.2)*
4. *Run algorithm 2.1 on the unsolved sub-mesh from the previous step.*

In this algorithm there are two steps where we need to resolve danger, step 2 (section 3.1) and step 3 (section 3.2).

3.1 Resolving Danger via Partition

The first place that we can resolve danger in algorithm 3.1 is step 2. In this step, we first partition the input mesh D into a minimal set of basic type sub-components (as in section 2.1) along a set of breaking edges (which will be initialized with assignment 0), then try to solve these breaking edges using the following procedure.

Procedure 3.1 (Resolving Danger via Partition) *Given a minimal set of basic type sub-components, solve the breaking edges between sub-components as follows.*

1. *Repeat the following until all the extremely dangerous sub-components (if there is any) have been transformed into moderately dangerous or safe ones:*
   a) *Address all the extremely dangerous sub-components, put them in a set $\mathfrak{C}$;*
   b) *For every extremely dangerous sub-component $C \in \mathfrak{C}$, check each adjacent sub-component C'_i as the following, where we use e^B_i to denote the boundary edge between C and C'_i.*
      - *If there is an adjacent C'_i that is safe, then assign e^B_i with $a_i = -b$ in C (thus C becomes moderately dangerous or safe) and $a'_i = b$ in C'_i (thus C'_i remains safe). Remove C from $\mathfrak{C}$.*
      - *Otherwise, if there is an adjacent C'_i that is moderately dangerous, then assign e^B_i with $a_i = -b$ in C (thus C becomes moderately dangerous or safe) and $a'_i = b$ in C'_i (thus C'_i remains moderately dangerous or becomes safe). Remove C from $\mathfrak{C}$.*
      - *Otherwise, if there is an adjacent C'_i that is extremely dangerous with opposite forbidding value $-b$, then assign e^B_i with $a_i = -b$ in C (thus C becomes moderately dangerous or safe) and $a'_i = b$ in C'_i (thus C'_i becomes moderately dangerous or safe). Remove C from $\mathfrak{C}$.*
      - *Otherwise, all the adjacent sub-components must be extremely dangerous and have the same forbidding value b; Then keep C in $\mathfrak{C}$ to wait for a second chance at a later point when some adjacent sub-component is transformed into a less dangerous one.*
   c) *Check the set $\mathfrak{C}$:*
      - *If $\mathfrak{C}$ is empty, jump to step 2 of this procedure.*
      - *Otherwise, if $\mathfrak{C}$ contains less sub-components than it does before this iteration, go back to step 1a and start another iteration.*
      - *Otherwise, no one in $\mathfrak{C}$ can be resolved, report $\mathfrak{C}$ and exit the procedure.*
2. *For every sub-component C that is moderately dangerous, do the following:*
   a) *For every new boundary edge e^B_i in C that has assignment 0 (i.e. still unsolved):*
      - *If the corresponding adjacent sub-component C'_i is moderately dangerous, then assign e^B_i with an arbitrary value $a_i \in \{+1, -1\}$ in C (thus C remains moderately dangerous or becomes safe) and $a'_i = -a_i$ in C'_i (thus C'_i remains moderately dangerous or becomes safe).*

- *Otherwise, the corresponding adjacent sub-component C'_i must be safe, then assign e_i^B with arbitrary value $a_i \in \{+1, -1\}$ in C (thus C remains moderately dangerous or becomes safe) and $a'_i = -a_i$ in C'_i (thus C'_i remains safe).*

As one can see from the procedure, solving breaking edges will make a dangerous boundary assignment less or equally dangerous but not more dangerous. For example, an extremely dangerous one can be transformed to another extremely dangerous one, a moderately dangerous one, or even a safe one; A moderately dangerous one can be transformed to another moderately dangerous one or a safe one, but not to an extremely dangerous one; A safe one can only remain safe and cannot be transformed to a dangerous one.

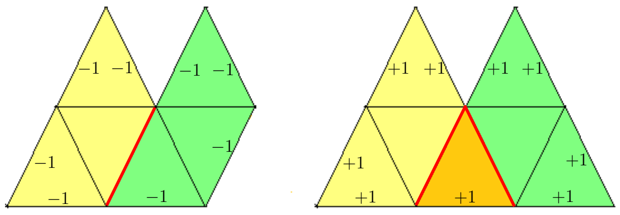

Fig. 7. Two examples of cutting pattern obstruction (definition 6).

At the end of step 1c in procedure 3.1, we detect a set of sub-components with extremely dangerous boundary assignments that cannot be transformed to less dangerous ones (figure 7). This set of unsolvable sub-components is an obstruction that keeps the algorithm from finding a valid solution.

Definition 6 (Cutting Pattern Obstruction). *A cutting pattern obstruction in an input mesh D is a maximal edge-connected component $D' \subseteq D$ consisting of a set of sub-components $\mathbb{C} = \{C_i \mid 1 \leq i \leq m\}$, such that:*

- *Every pair of sub-components C_i and C_j are triangle-disjoint to each other.*
- *Every C_i is a maximal basic type sub-component of basic types. Or equivalently, the set $\mathbb{C}$ is a minimal set of basic type sub-components.*
- *All of the sub-components in $\mathbb{C}$ have extremely dangerous boundary assignments and share the same forbidding value $a \in \{+1, -1\}$.*

3.2 Solving Moderately Dangerous Sub-components

After the previous step, if no obstruction is detected, we will reach step 3 in algorithm 3.1, which is a second place where we can resolve danger. At this point, all the extreme dangerous sub-components have been transformed, only moderately dangerous and safe ones are left. In this step, we will solve all the moderately dangerous sub-components.

A moderately dangerous sub-component, by definition, must be linear. Recall that a linear sub-component with safe boundary conditions can be completely solved using a stabbing procedure (section 2.2). As we will show in below, moderately dangerous sub-components can also be completely solved using a different stabbing procedure.

Given a moderately dangerous sub-component C, it contains a sequence of triangles $[f_0, \cdots, f_n]$. And by definition there must be two boundary edges $e_{i_1}^B$ and $e_{i_2}^B$ in two different triangles f_{i_1} and f_{i_2} ($i_1 < i_2$) that have opposite assignments, $a\left(e_{i_1}^B, f_{i_1}\right) = -a\left(e_{i_2}^B, f_{i_2}\right) = b \in \{+1, -1\}$. Taking f_{i_1} and f_{i_2} as two intermediate stops, we can partition C into three segments (in general) that can be stabbed separately (see figure 8).

- $C_{01} = [f_0, \cdots, f_{i_1}]$: stab this segment with a line from f_0 to f_{i_1} using initial assignment $-a(e^B_{0,1}, f_0)$ (i.e. opposite to the assignment for one of the boundary edges in f_0).
- $C_{12} = [f_{i_1}, \cdots, f_{i_2}]$: stab this segment with a line from f_{i_1} to f_{i_2} using initial assignment $-b$ (i.e. opposite to the assignment for boundary edge $e^B_{i_1}$ in f_{i_1}).
- $C_{2n} = [f_{i_2}, \cdots, f_n]$: stab this segment with a line from f_n to f_{i_2} using initial assignment $-a(e^B_{n,1}, f_n)$ (i.e. opposite to the assignment for one of the boundary edges in f_n).

Note that if there are more than one pair of $(e^B_{i_1}, e^B_{i_2})$ with opposite signs, one can choose an arbitrary pair and there is no impact on the convergence and correctness of the above algorithm. As another note, if $e^B_{i_1}$ (or $e^B_{i_2}$) appears in an end triangle of C, segment C_{01} (or C_{02}) would degenerate and therefore only two segments left, the algorithm can also work. With all these considerations, after this step there is no dangerous sub-components left. All the reminders are safe ones and can be solved using algorithm 2.1.

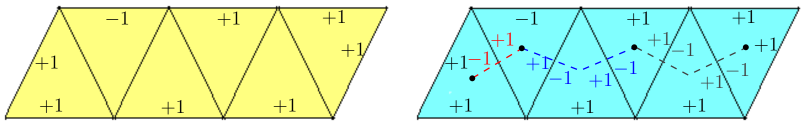

Fig. 8. A moderately dangerous sub-component must be linear; it can be completely solved via a stabbing line with three segments (separated by solid dots).

3.3 Validation and Analysis

In this part we discuss the convergence and complexity of algorithm 3.1. From the above discussion, this algorithm differs from the restricted algorithm 2.1 by several extra preprocessing steps 1, 2 and 3. As we will see in Lemma 6, this part of the algorithm always terminates.

Lemma 6. *The preprocess part (step 1, 2 and 3) of algorithm 3.1 always terminates; at termination, it either reports a cutting pattern obstruction, or outputs an unsolved sub-mesh with a safe boundary assignment.*

Proof. We check each preprocess step separately.

- In step 1: we check whether the boundary assignment is safe. If it is safe, the given mesh is left to step 4; otherwise, goes to step 2.
- In step 2: from the discussion of procedure 3.1 in section 3.1, if there is any cutting pattern obstruction in the input mesh D, it will be reported from step 1c in procedure 3.1, and the whole algorithm terminates. Otherwise, all the extremely dangerous sub-components are downgraded to moderately dangerous ones or safe ones (step 1), and then all the moderately dangerous sub-components are transformed to either moderately dangerous ones or safe ones (step 2). After this process, we have a set of basic type sub-components with completed boundary assignments, either moderately dangerous or safe.
- In step 3: based on the discussion in section 3.2, all the moderately dangerous sub-components are completely solved using a stabbing procedure, which guarantees to produce a valid set of assignments on those linear sub-components. After this process, the unsolved sub-mesh consists of a set of basic type sub-components with safe boundary conditions.

After step 3, if no obstruction is reported, the algorithm will get into step 4 with an unsolved sub-mesh with a safe boundary assignment, which has been proved in Theorem 1 to be solvable using algorithm 2.1. Therefore we can conclude:

Theorem 2. *Algorithm 3.1 always terminates; at termination, it either reports a cutting pattern obstruction or generate a valid cutting pattern for the original mesh.*

Now consider the time complexity of the algorithm. The cost of the algorithm lies in the following aspects.

- Step 1. This step only involves the boundary edges and is therefore bounded by $O(|E|)$.
- Step 2. The major cost lies in procedure 3.1. Step 1 in this procedure checks every extremely dangerous sub-component (for at most twice) and its adjacent sub-components, the cost is bounded by the total number of breaking edges in D. Similarly, step 2 in this procedure checks every moderately dangerous sub-component and its adjacent sub-components, the cost is also bounded by the total number of breaking edges in D. The total cost of this part is therefore bounded by $O(|E|)$.
- Step 3. Solving a moderately dangerous sub-component has the same complexity as that for solving a linear sub-component. Therefore this part is bounded by $O(|E|)$.
- Step 4. This step runs algorithm 2.1, which is bounded by $O(|V| + |E| + |F|)$ according to section 2.4.

In summary, the time complexity of algorithm 3.1 is $O(|V| + |E| + |F|)$, which is linear.

4 Solution for Free Boundary Conditions

Under a free boundary condition, the boundary edges of the input mesh D will not be prescribed with ± 1. Instead, these values can be set by the algorithm. It turns out that for a topological disk D, we can always generate a safe boundary assignment for D, so that the problem is converted to a restricted one (section 2). The algorithm pipeline is outlined in below.

Algorithm 4.1 (Cutting Pattern Algorithm - III)

- *Input: A triangular mesh D with a free boundary condition.*
- *Output: A valid cutting pattern for D.*
- *Procedures:*
  1. *Generate a safe boundary assignment for D.*
  2. *Run algorithm 2.1 on D with restricted boundary condition.*

Step 1 can be easily implemented. Check every triangle $f \in D$ that contains at least one boundary edge. If f contains only one boundary edge, assign it with $+1$. If f contains at least two such edges, assign $+1$ to one of them and -1 to the rest.

Obviously step 1 is guaranteed to terminate in time of $O(|F|)$. Therefore the convergence and complexity of this algorithm follows the results in section 2.4 for algorithm 2.1.

5 Conclusion and Discussion

In this paper we study the cutting pattern problem, a 2D graph labeling problem that is the major challenge of converting a prismal mesh to a tetrahedral mesh. We solve this problem for topological disks with a single boundary under three different boundary conditions. For restricted and free boundary conditions, we propose algorithms that is guaranteed to find a valid cutting pattern, therefore prove that solution always exists under these conditions. For general boundary conditions, we show that solutions exist if there is no cutting pattern obstruction; an algorithm is proposed to detect such obstructions if there is any or find a valid cutting pattern otherwise.

Note that for general boundary conditions, we only show that the absence of obstructions is a sufficient condition for the existence of solutions; but we are inclined to believe it is also necessary, and this will be an interesting topic for future exploration. Another interesting topic would be generalizing this work to other input meshes with more complicated topologies.

Acknowledgement

This work was partially supported by ONR N000140910228, NSF CCF-1081424, NSF Nets 1016829, NIH R01EB7530 and DARPA 22196330-42574-A. The first two authors were also supported by Harvard University Mathematics Department Funds. The authors would like to acknowledge and thank the anonymous reviewers for their insightful and constructive comments.

References

1. Rainald Löhner and Paresh Parikh. Generation of three-dimensional unstructured grids by the advancing-front method. *International Journal for Numerical Methods in Fluids*, 8(10):1135–1149, 1988.
2. Peter Möller and Peter Hansbo. On advancing front mesh generation in three dimensions. *International Journal for Numerical Methods in Engineering*, 38(21):3551–3569, 1995.
3. Mark S. Shephard and Marcel K. Georges. Automatic three-dimensional mesh generation by the finite octree technique. *International Journal for Numerical Methods in Engineering*, 32(4):709–749, 1991.
4. Nigel P. Weatherill and Oubay Hassan. Efficient three-dimensional delaunay triangulation with automatic point creation and imposed boundary constraints. *International Journal for Numerical Methods in Engineering*, 37(12):2005–2039, 1994.
5. Herbert Edelsbrunner. *Geometry and topology for mesh generation.* Cambridge University Press, 2001.
6. Tamal K. Dey. Delaunay mesh generation of three dimensional domains. In *Technical report.* Ohio State University, 2007.
7. A. M. Froncioni, P. Labbé, A. Garon, and R. Camarero. Interpolation-free space-time remeshing for the burgers equation. *Communications in Numerical Methods in Engineering*, 13(11):875–884, 1997.
8. Nelson Max, Barry Becker, and Roger Crawfis. Flow volumes for interactive vector field visualization. In *Proceedings of the 4th IEEE conference on Visualization*, pages 19–24, 1993.
9. Guy Albertelli and Roger A. Crawfis. Efficient subdivision of finite-element datasets into consistent tetrahedra. In *Proceedings of the 8th IEEE conference on Visualization*, pages 213–219, 1997.
10. Shahyar Pirzadeh. Advancing-layers method for generation of unstructured viscous grids. In *the 11th AIAA Applied Aerodynamics Conference*, pages 420–434, 1993.
11. Rainald Löehner. Matching semi-structured and unstructured grids for navier-stokes calculations. In *the 11th AIAA Computational Fluid Dynamics Conference*, pages 555–564, 1993.
12. Julien Dompierre, Paul Labbé, Marie-Gabrielle Vallet, and Ricardo Camarero. How to subdivide pyramids, prisms, and hexahedra into tetrahedra. In *Proceedings of the 8th International Meshing Roundtable*, pages 195–204, 1999.
13. Joseph A. Gallian. A dynamic survey of graph labeling. *The Electronic Journal of Combinatorics*, 17, 2010.