TopoCut: Fast and Robust Planar Cutting of Arbitrary Domains

XIANZHONG FANG, Ningbo University, China and Zhejiang University, China
MATHIEU DESBRUN, Inria/Ecole Polytechnique, France
HUJUN BAO, State Key Lab of CAD&CG, Zhejiang University, China
JIN HUANG*, State Key Lab of CAD&CG, Zhejiang University, China

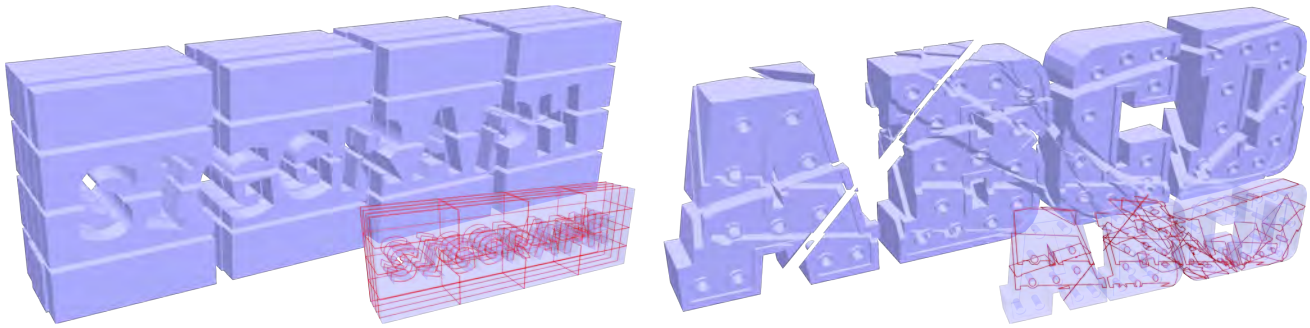


Fig. 1. **TopoCut at a glance.** We introduce a new bottom-up construction of generalized cut-cell meshes which split an input domain (left: a box with hollowed-out text; right: a letter-shaped domain) based on grid-aligned (left, five along each dimension) or arbitrary (right, ten random cuts) cutting planes. The resulting cut-cell meshes are slightly shrunk to show the interior; red lines show intersections between cut planes and input models.

Given a complex three-dimensional domain delimited by a closed and non-degenerate input triangle mesh without any self-intersection, a common geometry processing task consists in cutting up the domain into cells through a set of planar cuts, creating a "cut-cell mesh", i.e., a volumetric decomposition of the domain amenable to visualization (e.g., exploded views), animation (e.g., virtual surgery), or simulation (finite volume computations). A large number of methods have proposed either *efficient* or *robust* solutions, sometimes restricting the cuts to form a regular or adaptive grid for simplicity; yet, none can guarantee *both* properties, severely limiting their usefulness in practice. At the core of the difficulty is the determination of topological relationships among large numbers of vertices, edges, faces and cells in order to assemble a proper cut-cell mesh: while exact geometric computations provide a robust solution to this issue, their high computational cost has prompted a number of faster solutions based on, e.g., local floating-point angle sorting to significantly accelerate the process — but losing robustness in doing so. In this paper, we introduce a new approach to planar cutting of 3D domains that substitutes topological inference for numerical ordering through a novel mesh data structure, and revert to exact numerical evaluations only in the few rare cases where it is strictly necessary. We show that our novel concept of *topological cuts* exploits the inherent structure of cut-cell mesh generation to save computational time while still guaranteeing exactness for, and robustness to, arbitrary cuts and surface geometry. We demonstrate the superiority of our approach over state-of-the-art methods on almost 10,000 meshes with a wide range of geometric and topological complexity. We also provide an open source implementation.

CCS Concepts: • **Computing methodologies** → **Mesh geometry models**; • **Mathematics of computing** → **Mesh generation**.

Additional Key Words and Phrases: Robust geometry processing, cut-cell meshes, volumetric meshes.

ACM Reference Format:
Xianzhong Fang, Mathieu Desbrun, Hujun Bao, and Jin Huang. 2022. TopoCut: Fast and Robust Planar Cutting of Arbitrary Domains. *ACM Trans. Graph.* 41, 4, Article 40 (July 2022), 15 pages. https://doi.org/10.1145/3528223.3530149

1 INTRODUCTION

Given its boundary geometry $\partial\Omega$, a three-dimensional domain Ω can be split into a volumetric decomposition through a series of planar cuts. While this basic geometric operation is required in a large number of geometry processing applications, it has gathered increased attention in recent years due its increasingly common use in virtual surgery simulation [Sifakis et al. 2007; Patterson et al. 2012] and in the construction of "cut-cell" meshes for fluid flow simulation [Meinke et al. 2013; Edwards and Bridson 2014; Jaśkowiec et al. 2016; Azevedo et al. 2016; Tao et al. 2019], where a possibly large and intricate input surface geometry is intersected with a background (Cartesian or adaptive) grid to allow for efficient finite-volume computations. Assuming that the boundary surface geometry is provided as a triangle mesh, one would expect that the task of computing the volumetric decomposition induced by the input mesh and the various planar cuts to be trivial: after all, it only involves intersections of planes and linear facets — which can all be evaluated exactly with rational numbers — along with topological components (cut-cells, cut-faces, cut-edges and cut-vertices) and their combinatorial structure.

*Corresponding author

Authors' addresses: Xianzhong Fang, Ningbo University, Ningbo, Zhejiang, China, fangxianzhong@nbu.edu.cn; Mathieu Desbrun, Inria Saclay/Ecole Polytechnique, Palaiseau, France, mathieu.desbrun@inria.fr; Hujun Bao, Jin Huang, State Key Lab of CAD&CG, Zhejiang University, Hangzhou, Zhejiang, China, {bao,hj}@cad.zju.edu.cn.

1.1 Related work

Unfortunately, such a process remains a challenge in practice. While the use of exact arithmetic [GMP 2021] and/or exact predicates [Shewchuk 1997] guarantees correctness [Zhou et al. 2016] (and thus, robustness to arbitrary cuts), their respective computational cost is simply impractical when large meshes or many planar cuts are involved, spurring many authors to try to improve the speed of geometric tests. Exact geometric computing concepts [Yap and Sharma 2008] like delayed evaluations or interval analysis as used in CGAL [CGAL 2021] can maintain correctness while saving a good amount of computations, but computational complexity remains too high for most practical applications. Indirect geometric predicates [Attene 2020] used in [Cherchi et al. 2020; Diazzi and Attene 2021] leverage lazy intermediate representations to make the geometric tests exact, yet much faster than previous methods for fast and robust solid modeling operations. Recent approaches for general Boolean operations [Bernstein and Fussell 2009; Campen and Kobbelt 2010] have also tackled cut-mesh generation as it is akin to a form of *clipping* [Bajaj and Pascucci 1996; Wang and Manocha 2013] which can be efficiently implemented via a plane-based Binary Space Partition (BSP) tree. However, the use of exact arithmetic still renders these methods computationally intensive; this prompted [Nehring-Wirxel et al. 2021] to formulate a new scalable approach combining a local BSP in octree cells and 256-bit integer arithmetic in order to process an input mesh using 26-bit coordinates, improving computational times tenfold.

Compared to exact or high-precision arithmetic, the use of single-precision or double-precision floating-point operations is far more efficient. Yet, their well-documented round-off and digit-cancellation errors irremediably bring about both geometric *and* topological inconsistencies, leading to erroneous volumetric decompositions on which downstream applications fail. Heuristics based for instance on numerical perturbations [Edwards and Bridson 2014] can dramatically reduce occurrences of failure without significant computational overhead, but at the price of topologically inconsistent outputs. By further restricting to axis-aligned cut-planes, Tao *et al.* [2019] recently proposed a novel approach to infer topological information through polar orderings. However, while the resulting *Mandoline* tool was designed to guarantee topological correctness, they still rely on floating-point evaluations, which prevent robustness as demonstrated in Sec. 6.4. Another existing tool restricted to Cartesian cut-cell meshes, called Carl3D [Aftosmis et al. 1998], uses instead adaptive precision arithmetic and simulation-of-simplicity tie-breaking. This latter tool is commonly used in computational fluid dynamics due to its robustness, but incur significant computational overhead for complex boundary meshes as local grid refinements are triggered — at times, unreasonably so.

1.2 Contributions

This paper revisits the fundamental geometric operation of domain cutting by introducing a new method dubbed *TopoCut*.

- This name refers to the simple, yet powerful idea that we exploit to dramatically accelerate domain cutting: we remove the need for *local ordering of edges or faces* (see Fig. 2) as proposed in previous works to treat planar cuts by *leveraging the current mesh connectivity to determine which nearby cell to cut instead.*
- These *topological cuts* allow for the efficient, yet provably robust construction of a volumetric decomposition from an input simplicial complex and a set of planar cuts by relying on purely topological information to assemble the final cell complex, reverting to exact numerical evaluations *only* when strictly necessary — i.e, when the local topological situation is ambiguous.
- By exploiting the combinatorial topology of cell complexes and respecting their hierarchical construction from uniquely-labeled cut-vertices all the way to cut-faces, we show how to leverage equivalence classes to remove redundancy in the sets of cut-cell elements, and to bypass line and cycle ordering to infer most cuts using purely topological information — for instance, a d-dimensional convex cell split by a $(d-1)$-dimensional cut that goes through one of the $(d-2)$-dimensional elements of the convex cell can be established without any knowledge of the coordinates of the vertices involved (see Figs. 3, 4 and 16).
- Consequently, almost all of the bottom-up topological construction that we perform using half-edge and half-face data structures can be done exactly, yet very efficiently. We show that

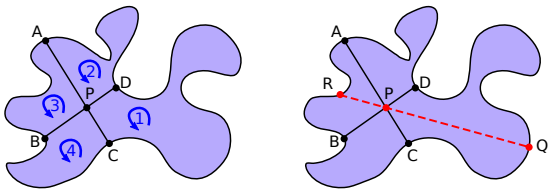


Fig. 2. **Didactic example of TopoCut**: For a input domain with already two cuts (left), every cut-face is topologically related to a boundary loop (enumerated from 1 to 4); so a new cut (Q, P, R) can directly identify the loops (here, 1 and 3) that need to be cut in two *without* numerical evaluation. Previous methods like Mandoline would rely instead on edge-cycle sorting around the cut-vertex P to proceed, which is slower and error prone.

Table 1. **Summary of our notations.**

Notation	Explanation
Ω	3D solid domain
$\mathcal{M}$	Input triangle mesh
$\mathcal{P}$	Input cut planes
$\overline{\mathcal{V}}$	Input vertices
$\overline{\mathcal{E}}$	Input edges
$\overline{\mathcal{T}}$	Input triangles
$\overline{\mathcal{F}}$	Input planar faces (triangles and planes)
$\mathcal{V}$	Cut-vertices
$\mathcal{E}$	Cut-edges
$\mathcal{F}$	Cut-faces
$\mathcal{C}$	Cut-cells
$(\overline{f}_i, \overline{f}_j, \overline{f}_k)$	A triple of faces
$(\overline{f}_i, \overline{f}_j)$	A tuple of faces
$H(\overline{f})$	Half-edge data structure of face $\overline{f}$
$\mathcal{H}_F^k$	k-th half-face data structure after k cuts
$\mathcal{H}_F$	Final half-face data structure

only a few *easily identifiable* cases require numerical evaluation to remove ambiguity, incurring an overall marginal overhead.

This novel strategy to output a generalized cut-cell mesh leads to significant computational gains (typically, between 3× and 10× faster on average for our implementation compared to the fastest existing methods [Tao et al. 2019; Diazzi and Attene 2021]) while matching the output of slow, but exact methods as demonstrated on nearly 10,000 models of various shape complexity and topology. We also discuss ways to compute a floating-point representation of our cut-cell mesh for downstream processing — a lossy conversion known to introduce self-intersections and degeneracies [Milenkovic and Nackman 1990; Devillers et al. 2018], but for which we still outperforms existing approaches.

2 OVERVIEW OF TOPOCUT

Given an orientable, non-self-intersecting and watertight input triangle mesh $\mathcal{M}$ with vertices $\overline{\mathcal{V}}$, edges $\overline{\mathcal{E}}$ and triangles $\overline{\mathcal{T}}$, and a set of arbitrary cut planes $\mathcal{P} = \{p_1, p_2, ..., p_n\}$, we want to compute a volumetric decomposition, denoted as the "cut mesh" for brevity. We summarize the notations used in this paper in Tab. 1.

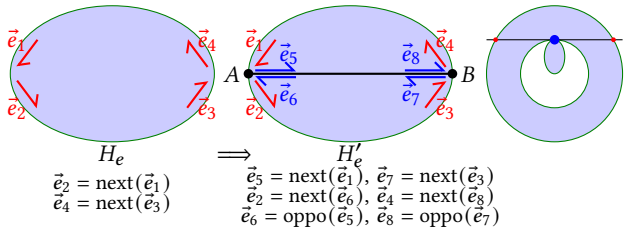


Fig. 3. **Example of topological face cut**. When cutting the half-edge loop of an existing cut-face by a cut-edge whose cut-vertices are already in the face (left), existing topological information is sufficient to update the data structure and perform the cut (middle). For a non-manifold cut-vertex (right, blue point), the situation is ambiguous, and edge-cycle ordering is required to choose between the two possible options.

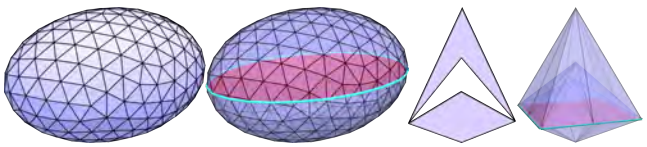


Fig. 4. **Example of 3D topological cut.** While cutting the half-face data structure of a cut-cell (left) by a cut-face whose cut-edges are already part of the cell (middle), the same type of topology-only update of the cut-cell as in Fig. 3 can be performed. For a model with non-manifold edges, we need to resort to face-cycle ordering to deal with the resulting ambiguity.

2.1 Input Mesh

Before providing a bird's eye view of our TopoCut approach, we discuss the few requirements that the input mesh must satisfy:

- The input triangle mesh $\mathcal{M}$ forms the boundary of a simply-connected 3D solid domain Ω with finite volume, watertight, and without self-intersection (see App. A). All the normals of the input triangles are consistently oriented outwards. Furthermore, we assume that $\overline{\mathcal{V}}$ has no duplicated vertices (no two vertices should be collocated), and each triangle in $\overline{\mathcal{T}}$ is not degenerated.
- The input set $\mathcal{P}$ of cut planes has no duplicated planes, and each plane $p \in \mathcal{P}$ is defined by $\mathbf{n}_p^T\mathbf{x} + d_p = 0\ \forall \mathbf{x} \in p$, for a normal $\mathbf{n}_p \neq \mathbf{0}$ and an scalar offset d_p.

Similar to many previous works and to ensure the accuracy of intermediate computations, we assume that all the coordinates and plane parameters to be rational numbers, without loss of generality since most inputs typically use floating-point coordinates. With a high-precision numerical library such as GMP [GMP 2021], exact arithmetic in the field $\mathbb{Q}$ of rational numbers provides the basis for zero-error evaluation if need be.

2.2 Output Cut Mesh

The output cut mesh will be composed of cut-vertices $\mathcal{V}$, cut-edges $\mathcal{E}$, cut-faces $\mathcal{F}$ and cut-cells $\mathcal{C}$, such that the domain cutting problem can be formulated as

$$\{\mathcal{M}(\overline{\mathcal{V}}, \overline{\mathcal{E}}, \overline{\mathcal{T}}), \mathcal{P}\} \quad \xrightarrow{\text{Cut}} \quad \{\mathcal{V}, \mathcal{E}, \mathcal{F}, \mathcal{C}\}.$$

While more thorough descriptions of the domain cutting problem have been mentioned in previous works such as [Tao et al. 2019], we review its precise definition here so as to properly define what we mean by "robustness" and "correctness" of our construction. If we denote by $\overline{\mathcal{F}} = \overline{\mathcal{T}} \cup \mathcal{P}$ all the faces (triangles & planes) involved in the input, correctness of the TopoCut connectivity means that:

- The union of all cut-cells covers the entire domain bounded by $\mathcal{M}$. The boundary of cut-cells are formed by cut-faces, the boundary of cut-faces are cut-edges, and the boundary of cut-edges must be cut-vertices, i.e.,

$$\mathcal{V} \xleftarrow{\partial} \mathcal{E} \xleftarrow{\partial} \mathcal{F} \xleftarrow{\partial} \mathcal{C}.$$

  Each element in $\mathcal{E}, \mathcal{F}$ and $\mathcal{C}$ is thus properly represented by its boundary. All of these elements are involved in the cut-cell complex, and none of these elements are duplicates or degenerate (i.e., no zero volumes, areas, or lengths).
- Each cut-vertex is the intersection of exactly three faces in $\overline{\mathcal{F}}$, i.e., $\forall v \in \mathcal{V}, v = \overline{f}_i \cap \overline{f}_j \cap \overline{f}_k,\ \overline{f}_i, \overline{f}_j, \overline{f}_k \in \overline{\mathcal{F}}$. *Thus,* triples *of faces in* $\overline{\mathcal{F}}$ *can be used to represent cut-vertices.* Note that there may exist two or more triples representing identical spatial coordinates; this will be handled in the end of Sec. 3.1.
- Each cut-edge is at the intersection of two faces in $\overline{\mathcal{F}}$, i.e., $\forall e \in \mathcal{E}$, $e \subset \bar{f}_i \cap \bar{f}_j$, with $\bar{f}_i, \bar{f}_j \in \overline{\mathcal{F}}$. Two cut-vertices along an intersection line between two faces form a cut-edge that is devoid of any other cut-vertex; *thus a linear ordering of cut-vertices determines cut-edges* (see Fig. 6, left).
- Each cut-face is a subset face of $\overline{\mathcal{F}}$, i.e., $\forall f \in \mathcal{F}, f \subseteq \bar{f} \in \overline{\mathcal{F}}$, enclosed by a chain of cut-edges. In the chain, two cut-edges around a common cut-vertex forms a corner of the cut-face if there are no other cut-edges in between (see Fig. 5, left). As a consequence, *cut-faces are defined via edge-cycle ordering* (see Fig. 6, middle).
- Each cut-edge belongs to two or more adjacent cut-faces, and two cut-faces adjacent to a common cut-edge form the wedge (dihedral angle) of a cut cell if no other cut-face lies in between (see right of Fig. 5), so *cut cells are defined via face-cycle ordering* (see Fig. 6, right).

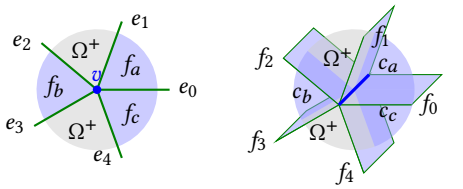


Fig. 5. **Corners and wedges.** Adjacent cut-edges to a cut-vertex in a cut-face form a corner (left); adjacent cut-faces of a cut-edge in a cut-cell form a wedge (right). Here, Ω^+ means the outside of domain Ω.

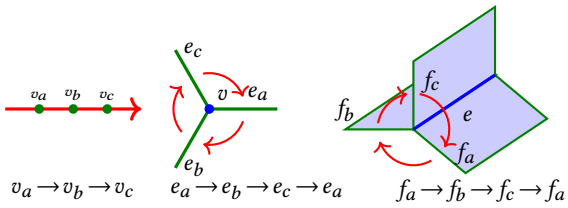


Fig. 6. **Orders of mesh elements.** Left: linear order of cut-vertices. Middle: edge-cycle order of the one-ring of a cut-vertex. Right: face-cycle order of the one-ring of a cut-edge.

Due to the hierarchical nature of the cell complex construction of the volumetric decomposition, we see therefore that *given cut-vertices, local orderings of mesh elements define the cut mesh connectivity — and vice-versa.* In our TopoCut approach, we will enforce the correctness of the domain cutting description by construction, making heavy use of the aforementioned orderings but *inferring them from topological clues* found in the current mesh connectivity. Robustness of TopoCut simply refers to the fact that our approach terminates after generating a cut-cell complex satisfying all the conditions listed above, and its result does *not* differ from what a (far slower) method using exact arithmetic would produce.

Holes and voids. One issue worth mentioning early on is that, while most cut-faces and cut-cells after cutting have only one connected boundary, there can technically be multiple connected boundaries if they contain *holes* or *voids*. Indeed, a simply-connected cut-face is enclosed by cut-edges, but it can look like a simple disk — in this case, only one connected set of cut-edges defines its boundary — or like an annulus for instance — in which case, there are two such connected sets. As a matter of definition, we will say that two cut-edges e_i and e_j are connected if there exists a path of adjacent cut-edges to go from one to the other. Similarly, because a cell can have a ball-like or cheese-like topology, we say that two cut-faces are *connected* if there is a path of adjacent cut-faces joining the two. In the general case, there can be several holes in a cut-face, or several voids in a cut-cell, so one must extract its multiple connected boundaries, and some of these boundaries will have opposite orientation. Because the boundary is piecewise linear, there is no difficulty in computing the signed volume exactly with rational numbers, therefore, each cut-face or cut-cell has a well defined "inside" or "outside" label. Simply put, cut-faces and cut-cells with negative volume actually enclose parts that are outside of Ω. Depending on the application being targeted, one may want to keep this detailed topology of each cut-face and cut-cell, or simply remove the holes and voids by cutting the cut-faces or cut-cells open through their homology generators. Note that the total sum of all the oriented volumes of the cut-cells will be equal to the input domain volume.

2.3 Modified half-edge and half-face data structure

To help with the construction of a cut-cell mesh, we will use a volumetric variant of the common half-edge data structure, traditionally used for surface meshes. More precisely, we start from the *half-face data structure* [OpenVolumeMesh 2021] to represent the relationship between faces and cells. Each cut face $f \in \mathcal{F}$ along with an adjacent cell $c \in C$ form an oriented half-face $\vec{f}(c) \in \vec{\mathcal{F}}$. Its "opposite" oriented half-face, stored as opposite$(\vec{f})$ in the half-face structure, is thus the same face f but this time associated with another adjacent cell $c' \in C$, where $f = c \cap c'$. If there is no such "opposite" cell (i.e., at the boundary of the domain), the opposite half-face pointer is set to empty. This data structure provides efficient local operations to query neighboring cells from faces or edges. Since we also need to efficiently represent the relationship between points and edges, we use an extension of the usual *half-edge data structure* as well, and integrate it into a half-face data structure, similar to [OpenMesh 2021]. For each cut-edge e along with an adjacent oriented half-face $\vec{f}$, one can uniquely define a half-edge $\vec{e}(\vec{f}) \in \vec{\mathcal{E}}$ for $e \in \mathcal{E}$ and with $e \subset \partial f$ and $\vec{f} \in \vec{\mathcal{F}}$. The common "opposite" operation on a half-edge queries the half-edge on the adjacent face, which is ill-defined for edges with more than two adjacent faces. Note that the orientation of the half-face $\vec{f}$ defines an oriented cycle around edge e as shown in Figs. 6 and 7, which uniquely defines the half-edge on the "next adjacent" face. As a consequence, we replace the common opposite operation acting on half-edges on a face by an "adjacent" operation with the following definition:

$$\text{adj}(\vec{f}, e) = \vec{e}(\vec{f}'), \tag{1}$$

where f' is the next adjacent face of f along the cycle. It is easy to verify that this new operation is consistent with the common "opposite" operation if e is adjacent to two consistently oriented faces. It should be noticed here that the input mesh can also be viewed as an extreme case of cut-cell mesh with a simple, complicated cell Ω and no cuts, which can be consistently represented by this new data structure. In order to simplify our explanations in the remaining of our paper, we do not use the half-edge $\vec{e}(\vec{f})$ explicitly, but instead use a tuple $(\vec{f}, e)$, and use the function adj$(\vec{f}, e) \in \vec{\mathcal{F}}$ to describe the adjacent half-face of $\vec{f}$ along e in the same cell.

Our half-edge/half-face data structure will be used to represent the connectivity and ordering information among cut-vertices, cut-edges, cut-faces and cut-cells, and has no geometric (numerical) information. To have a complete description of the output cut mesh, the coordinates of all the cut-vertices are also required, but they will be separately stored in a set Q. Giving a cut-vertex with index v, we can retrieve its coordinates coord(v) as a rational number (or, equivalently, a floating-point value) from this cut-vertex set.

2.4 Algorithm pipeline and key concept

In an overall manner similar to [Tao et al. 2019], our algorithm consists in first computing the cut-vertices, then assembling cut-edges, cut-faces and cut-cells in order, see Figs. 8 and 9. The various stages of our construction differ sharply from previous works, however; they are summarized as follows:

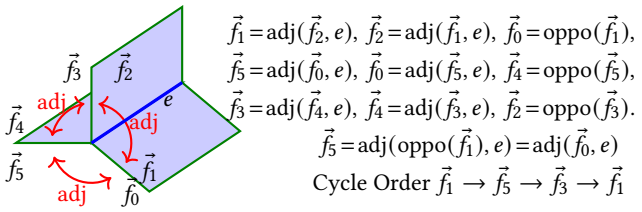


Fig. 7. **Face-cycle order** of the one-ring of edge e, along with its associated half-face structure.

(1) Compute all intersection lines and points, and represent them symbolically by tuples and triples respectively, containing the indices of the faces (triangles or planes) that intersect (see Sec. 3).

(2) Sort the intersection points on each intersection line, removing duplicate points and outside parts to get cut-vertices $\mathcal{V}$ and cut-edges $\mathcal{E}$ (see Sec. 3).

(3) Cut every face $\overline{f}$ in $\overline{\mathcal{F}}$ (i.e., each triangle $\bar{t}$ in $\overline{\mathcal{T}}$ and cut-plane p in $\mathcal{P}$) by the cut-edges found above, and store them in an individual half-edge data structure $H(\overline{f})$ (i.e., $H(\bar{t})$ and $H(p)$ in Fig. 8) — see Sec. 4.

(4) From the half-edge data structures $\{H(\bar{t})\}$ related to all the triangles in $\mathcal{M}$, generate an initial half-face data structure $\mathcal{H}_F^0$ enclosing the volumetric domain surrounded by $\mathcal{M}$ (see Sec. 5.2).

(5) Given the current half-face data structure $\mathcal{H}_F^k$, cut it into the next "refined" half-face data structure $\mathcal{H}_F^{k+1}$ by a half-edge data structure $H(p)$ of an unprocessed cut-plane $p \in \mathcal{P}$ (see Sec. 5.3); repeat until no cut-plane is left.

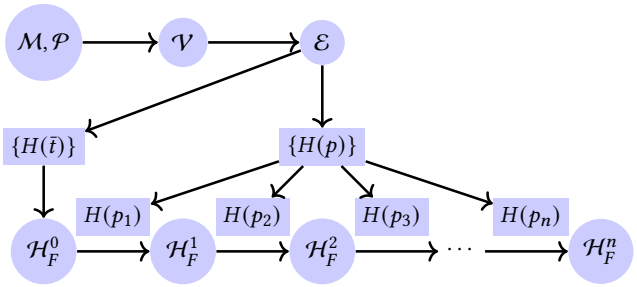


Fig. 8. **Data flow.** Overview of our algorithm.

While our algorithm (schematically summarized in Fig. 8) could rely on a number of linear, edge-cycle, and face-cycle orderings as was proposed in recent works (possibly through delayed evaluation or adaptive accuracy for efficiency), the overall cost of these operations does not scale well with the complexity of the input mesh and the number of planes. Instead, TopoCut leverages the fact that numerical sorting is in fact unnecessary in most cases, as illustrated in Fig. 2 and as shown in practice in Figs. 3 and 4: one can directly infer, from the current adjacency graph, which surrounding cell(s) to cut. Fully exploiting the efficacy of topology-only cutting will significantly save computational time, while keeping the results exact, i.e., equivalent to using exact arithmetic. Only a few cases that are topologically ambiguous will need exact evaluation, requiring a negligible amount of computations overall and scaling extremely well with input complexity.

3 GENERATION OF CUT-VERTICES AND CUT-EDGES

Our construction begins by finding all cut-vertices. As mentioned earlier, each cut-vertex is the intersection point of three faces in $\overline{\mathcal{F}} = \overline{\mathcal{T}} \cup \mathcal{P}$, i.e., an intersection between triangle faces in $\overline{\mathcal{T}}$ and/or infinite large planes in $\mathcal{P}$. Each cut-vertex $v \in \mathcal{V}$, whose location in space is defined with three coordinates in $\mathbb{Q}^3$, is thus defined by a face triple in $\mathbb{Z}^3$ through

$$(\overline{f}_i, \overline{f}_j, \overline{f}_k) := v = \overline{f}_i \cap \overline{f}_j \cap \overline{f}_k, \text{ where } \{\overline{f}_i, \overline{f}_j, \overline{f}_k\} \subset \overline{\mathcal{F}}. \quad (2)$$

Not all face triples are forming a cut-vertex, and a valid face triple should have only one intersection point among its three triangles and/or cut-planes. However, a cut-vertex may be associated with multiple triples composed of indices of the different faces which just happen to intersect at the same spatial location. It is worth noting that our representation is different from the triples in [Tao et al. 2019] as it applies to arbitrary cuts instead of just axis-aligned cuts, and it only includes integer indices, with no numerical values stored. We now elaborate how to find all such triples and subsequently generate cut-edges, before finally merging duplicate triples to generate the final list of distinct cut-vertices.

3.1 Intersection lines between faces

Directly enumerating all triples of faces in $\overline{\mathcal{F}} = \overline{\mathcal{T}} \cup \mathcal{P}$ to check if they intersect can be done with complexity $|\overline{\mathcal{F}}|^3$. In practice, we begin by finding all the *pairs of faces* intersecting along a line first, for a complexity of $|\overline{\mathcal{F}}|^2$: this automatically group cut-vertices on the same line together, and intersection between this line and all other faces is performed in a second step. Intersection lines are represented as unordered tuples of faces of three possible types:

- $(\bar{t}_i, \bar{t}_j) \in (\overline{\mathcal{T}}, \overline{\mathcal{T}})$: intersection line of two triangles, i.e., an input edge given our input requirements;
- $(\bar{t}, p) \in (\overline{\mathcal{T}}, \mathcal{P})$: a triangle $\bar{t}$ and cut-plane p intersect at a point or along a line $\mathbf{n}_p \times \mathbf{n}_{\bar{t}} \neq \mathbf{0}$, where $\mathbf{n}_p$ and $\mathbf{n}_{\bar{t}}$ are the normals of p and $\bar{t}$ respectively; note that if the triangle lies within the plane, there is no actual cut;
- $(p_i, p_j) \in (\mathcal{P}, \mathcal{P})$: intersection line of two planes p_i and p_j, implying that $p_i \cap p_j \neq \emptyset$, or equivalently, $\mathbf{n}_i \times \mathbf{n}_j \neq \mathbf{0}$, where $\mathbf{n}_i$ and $\mathbf{n}_j$ are the normals of p_i and p_j respectively.

Exact arithmetic computation is used in this step for robustness, see App. B for practical details. Then, for each intersection line of two faces $\overline{f}_a$ and $\overline{f}_b$, we enumerate all the remaining faces in $\overline{\mathcal{F}}$ to check for possible intersection with this line. If faces $\overline{f}_a, \overline{f}_b$ and $\overline{f}_c$ intersect at a single point (even if it is outside of the input domain), $(\overline{f}_a, \overline{f}_b, \overline{f}_c)$ is marked as a point on the line $(\overline{f}_a, \overline{f}_b)$. As we describe next, we must now order all these points so as to segment the intersection line; note that all the line segments outside of the region delimited by the input mesh $\mathcal{M}$ will be removed later on.

Line sorting. All the points $(\overline{f}_a, \overline{f}_b, \overline{f}_{c_k}), k = 1, ..., m$ on an intersection line $(\overline{f}_a, \overline{f}_b)$ must be sorted. If the intersection line $(\overline{f}_a, \overline{f}_b)$ is degenerate, which is possible in the $(\overline{\mathcal{T}}, \mathcal{P})$ case (only a corner of the triangle is on the plane), there is no cut-edge on it. Otherwise, we proceed as in [Tao et al. 2019]: we pick a direction $\vec{\ell}$ that is not orthogonal to the line $(\overline{f}_a, \overline{f}_b)$, compute the rational coordinates $x \in \mathbb{Q}^3$ corresponding to the triple, and then $(\overline{f}_a, \overline{f}_b, \overline{f}_{c_k})$ along line

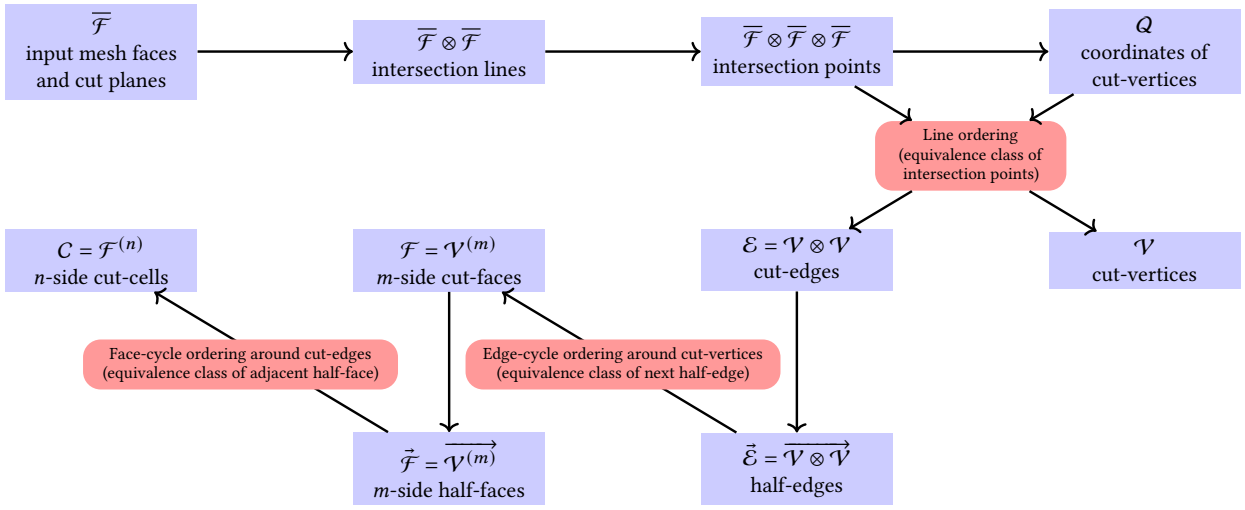


Fig. 9. **Data structure abstraction.** Schematic description of the data structure in our algorithm.

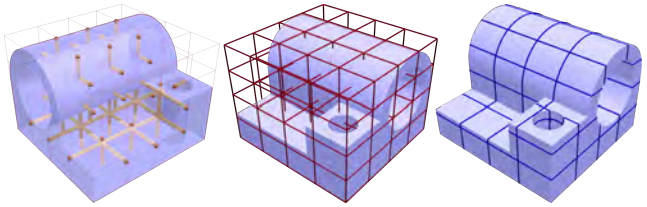


Fig. 10. **Split Joint**. The Joint mesh is cut by a regular arrangement of planes. Left: inner cut-edges (orange), middle: outer parts of intersection lines (red), right: boundary intersection lines (blue).

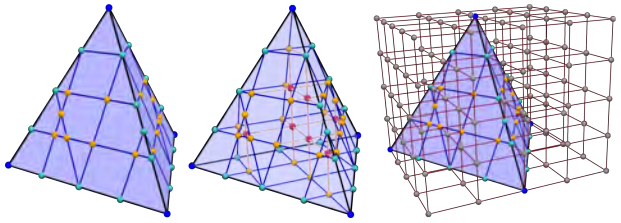


Fig. 11. **Cut-vertices and cut-edges for a sliced tetrahedron.** Cut-vertices are indicated with blue points when on the input vertices, green when on the edges of the input, orange when on the input triangles, and dark red when they are at the intersection of cut-planes inside Ω. Cut-edges are shown in blue when between cut-planes and input triangles, and yellow when between cut-planes inside Ω.

$(\overline{f}_a, \overline{f}_b)$ is parameterized by $\gamma_k \in \mathbb{Q}$ with

$$\gamma_k = x \cdot \vec{\ell}. \tag{3}$$

Using the parameterized coordinates γ_k, we can now sort the triples on $(\overline{f}_a, \overline{f}_b)$ into an ordered sequence $\{(\overline{f}_a, \overline{f}_b, \overline{f}_{c_l})\}_l$ based on their values γ_k. If the same γ_k value is found for multiple vertices, we call these points "equivalent" since they share identical coordinates, and we merge them into a single cut-vertex. Note that any $\vec{\ell}$ that is not orthogonal to the line $(\overline{f}_a, \overline{f}_b)$ will result in the same sorting results up to a possible order reversal. The ordered sequence of non-equivalent points thus defines a set of non-degenerate segments.

3.2 Identifying inner and outer cut-edges

The oriented input mesh $\mathcal{M}$ defines three distinct regions: the domain boundary $\partial\Omega$, the inside of the domain Ω^-, and the outer space Ω^+. Relevant cut-edges must thus be classified into two sets: the cut-edges on $\partial\Omega$, and cut-edges belonging to the inside of Ω, see Fig. 10. From the non-degenerated segments of $(\overline{\mathcal{T}}, \overline{\mathcal{T}})$ and $(\overline{\mathcal{T}}, \mathcal{P})$ (i.e., those induced by a triangle facet and with a non-zero length), we obtain all boundary cut-edges — and if two segments have the same end-points, they are identified as the same cut-edge. From tuples of the $(\mathcal{P}, \mathcal{P})$ type, however, cut-edges may be inside, outside, or on the boundary of Ω, and only inner cut-edges are relevant in our domain cutting context. Noting that boundary cut-edges from $(\mathcal{P}, \mathcal{P})$ must have already been identified from the $(\overline{\mathcal{T}}, \mathcal{P})$ and $(\overline{\mathcal{T}}, \overline{\mathcal{T}})$ types, we just need to know which cut-edges are inside of the domain. This determination will also help to generate inner cut-faces later on.

The intersection lines $\ell = p_i \cap p_j$ for plane tuples $(p_i, p_j) \in (\mathcal{P}, \mathcal{P})$ are segmented by the input mesh into an inner part, and an outer and boundary part. We just have to locate the transitions, which can be achieved through a ray casting method typically used in determining if a point is inside a polyhedron [Kalay 1982; Feito and Torres 1997] by counting the number of times a ray passes through the boundary $\partial\Omega$ (see Fig. 12 for a 2D example). Define a binary function $s(.) \in \{0, 1\}$ indicating whether an oriented line $\vec{\ell}$ in the direction of line ℓ gets into or out of Ω at $v \in \ell \cap \partial\Omega$: when $\vec{\ell}$ passes from inside to outside or from outside to inside at v, $s(v) = 1$; otherwise $s(v) = 0$. Given all previously-found triples of type $(\mathcal{P}, \mathcal{P}, \overline{\mathcal{T}})$ in $(p_i, p_j) \in (\mathcal{P}, \mathcal{P})$, (i.e., the triples representing cut-vertices on $\ell \cap \partial\Omega$), we remove duplicate triples and get a set of vertices $\{v_1, v_2, ..., v_m\}$ along $\vec{\ell}$ that we sort; for each v in this sorted set, we calculate $s(v)$ as explained in App. C, and obtain a series of binary values $\{s_1, s_2, ..., s_m\}$. Then cut-edges (v_k, v_{k+1}) are identified as inside or not by simply checking parity:

$$(v_k, v_{k+1}) \in \begin{cases} \Omega^-, & \text{if } (\sum_{i=1}^{k} s_i) \bmod 2 = 1, \\ \Omega^+ \cup \partial\Omega, & \text{otherwise.} \end{cases} \tag{4}$$

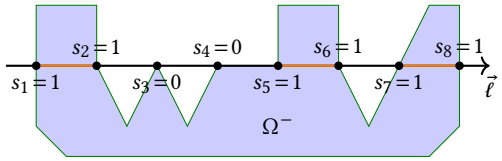


Fig. 12. **2D example of inner cut-edges** (orange segments) along a line ℓ.

3.3 Extract cut-vertices and cut-edges

We have already evaluated the exact coordinates for all the cut-vertices (i.e., face triples) right before our line sorting, so it is trivial to group the triples with the exact same coordinates together into a unique cut-vertex; note that we can leverage the line order we computed to accelerate this check. We end up with a set of different cut-vertices $\mathcal{V}$; cut-edges, which are unordered tuples in $\mathcal{V} \otimes \mathcal{V}$, have thus no duplicates — completing this stage. Cut-vertices for a tetra mesh example with axis-aligned cuts are shown in Fig. 11.

4 CUT-FACE GENERATION THROUGH HALF-EDGES

Once we have found all the tuples of intersection lines, cut-vertices $\mathcal{V}$, and cut-edges $\mathcal{E}$, we are ready to construct cut-faces. As we are about to see, we proceed one face at a time and collect all of its elements touching a boundary triangle, before building an initial half-edge structure and cutting it by all the intersection lines represented by face tuples. The key idea at this stage is to properly set the "next" and "opposite" of each cut-edge in the half-edge data structure, which boils down to exactly and efficiently ordering the cut-edges around each cut-vertex, see Figs. 13 and 15.

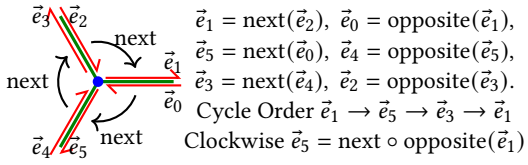


Fig. 13. **The cycle order of the one-ring edges of a vertex** and its half-edge structure; notice that the ordering can be deduced from the opposite and next relations of the half-edges.

4.1 Initialization from boundary triangles

For all faces in $\overline{\mathcal{F}}$, we need to find the entities (intersection lines, cut-vertices and cut-edges) that are part of a boundary triangle. Consequently, for each triangle $\overline{f} \in \overline{\mathcal{T}}$, we first find the tuples containing $\overline{f}$ in $(\overline{\mathcal{T}}, \overline{\mathcal{T}})$, i.e., we find the sets:

$$\{(\overline{f}, \bar{t}) \in (\overline{\mathcal{T}}, \overline{\mathcal{T}}) \,|\, \forall \bar{t} \in \overline{\mathcal{T}}\}. \tag{5}$$

For each cut plane $\overline{f} \in \mathcal{P}$, besides the part outside of Ω, we also exclude the parts within a boundary triangle because they will already be in the half-edge data structures of boundary triangles. Therefore, we just have to find the tuples containing $\overline{f}$ that form an intersection line with $\overline{f}$:

$$\{(\overline{f}, \bar{t}) \in (\overline{\mathcal{T}}, \mathcal{P}) \,|\, \forall \bar{t} \in \overline{\mathcal{T}} \text{ and } \bar{t} \not\subset \overline{f}\}. \tag{6}$$

For each tuple, we already sorted all the triples (cut-vertices) along the intersection line in Sec. 3.1, thus we already have the cut-edges within $\overline{f}$. Note that if $\overline{f}$ is a boundary triangle, all the intersection lines mentioned above are on its boundary. In degenerate cases, there may be inner cut-edges as well if $\overline{f}$ is a cut plane (see inset), but they only require a few additional checks. When there exist coplanar triangles on the cut plane $\overline{f}$, and since these triangles will have their own half-edge structures, we do not need to consider them. Some tuples in $(\overline{f}, \bar{t})$ are related to the cut-edges in $\partial(\overline{f} \cap \Omega) \backslash \partial(\overline{f} \cap \Omega^-)$, which can also be removed. Now, according to the two adjacent regions of each cut-edge on $\overline{f}$, the boundary of $\overline{f} \cap \Omega^-$ has three types of cut-edges: inner-outer, inner-inner and inner-boundary, see (1),(2),(3) in Fig. 14 respectively; but the boundary of $\overline{f} \cap \Omega$ has another two types, namely outer-boundary and outer-outer, which are not useful, see (4),(5) in Fig. 14. Each tuple $(\overline{f}, \bar{t})$ is related to one intersection line between $\overline{f}$ and $\bar{t}$; if this intersection line does not belong to $\partial\bar{t}$, the tuple is kept since the line is of inner-outer type; otherwise, the intersection line is on the boundary $\partial\bar{t}$, i.e., it is an input edge $\bar{e} \in \overline{\mathcal{E}}$. We then check its type via face-cycle sorting of normals of adjacent triangles of $\bar{e}$ and plane $\overline{f}$: if it is an outer-boundary or outer-outer type, we remove it.

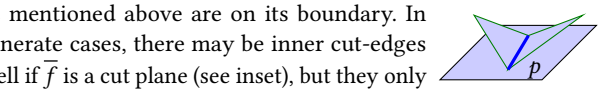


Fig. 14. **Types of tuple** $(p, \bar{t})$**.** Line indicates intersection between p and $\bar{t}$.

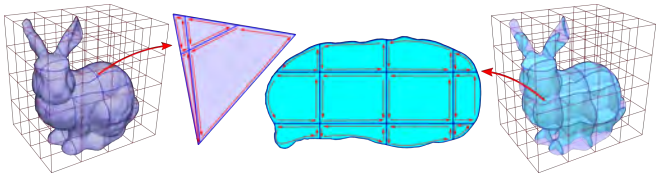
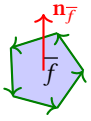


Fig. 15. **Half-edge data structure.** Examples of half-edge structures for (left) a triangle and (right) a cut plane formed by axis-aligned planar cuts.

Now that we have gathered all the cut-edges related to boundary triangle for face $\overline{f} \cap \Omega$, we can construct the initial half-edge data structure $H^0(\overline{f})$. Cut-edges are unordered tuple of vertex indices of the type $(v_a, v_b), v_a, v_b \in \mathcal{V}$, and we want to turn them into oriented half-edges, producing ordered tuples of vertex indices $[v_a, v_b]$. We simply pick the orientation consistent with the (outward) face normal of $\overline{f} \in \overline{\mathcal{T}}$ or the plane normal $\mathbf{n}$ for $\overline{f} \in \mathcal{P}$. That is to say, for a triangle, the cut-edges in its boundary have an orientation consistent with its boundary vertex ordering; for a plane p, the orientation of the cut-edges in $p \cap t$ are chosen to be consistent with $\mathbf{n}_p \times \mathbf{n}_t$, where t is a triangle on the boundary of the input mesh, see Fig. 15. Its next half-edge must be of the form $[v_b, v_x]$, where v_x is determined by cycle ordering of boundary cut-edges around v_b. If there are only two cut-edges (v_a, v_b) and (v_b, v_c) adjacent to v_b, the next of $[v_a, v_b]$ is simply $[v_b, v_c]$; the edge-cycle ordering around vertices v_b allows the handling of non-manifold 1D boundaries.



We do not explicitly record information for the face of a half-edge: the "next" operation provides an equivalence relationship among half-edges, and the equivalence class of a half-edge is a loop surrounding a face. All the faces in the half-edge data structure are

thus the quotient set of all the half-edges under the "next" operation. The "opposite" can be simply constructed for pairs of half-edges $[v_a, v_b]$, $[v_b, v_a]$ in the current half-edge data structure. Because we assumed our mesh input to be an orientable watertight surface, the operations for "next" and "opposite" are always well defined.

4.2 Repeatedly cutting by planes

Given the initial half-edge structure $H^0(\overline{f})$, we now can recursively cut it by intersection lines within $\overline{f}$. All the intersection lines that are parts of a face $\overline{f}$ can be found by tuples of the form:

$$\{(\overline{f}, p) \in (\overline{\mathcal{T}}, \mathcal{P}) \cup (\mathcal{P}, \mathcal{P}) \,|\, \forall p \in \mathcal{P}\}. \tag{7}$$

If a tuple in this set is colinear to one from $\{\overline{f}, t\}$ in the initial half-edge structure; we can safely discard it. Otherwise, we recursively cut the face $\overline{f}$ with the current half-edge data structure $H^k(\overline{f})$ by each intersection line from the set in Eq. (7). In each cut, we need to add the cut-edges on the intersection line $(\overline{f}, p)$ to H^k and update the next and opposite pointers to maintain the half-edge data structure. Similarly to the earlier case, the "next" pointer is determined through cycle ordering. Then each cut-edge (v_a, v_b) on $(\overline{f}, p)$ provides two half-edges $[v_a, v_b]$, $[v_b, v_a]$ with opposite orientations.

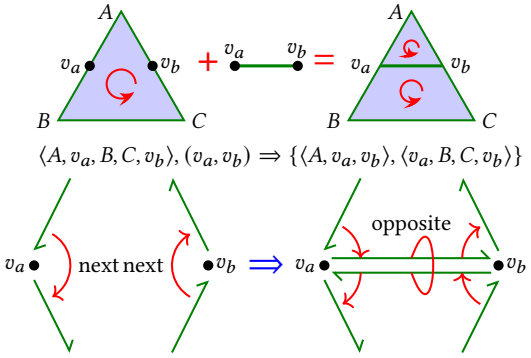


Fig. 16. **Topological 2D cut.** One simple example for 2D topological cut (top), which can be entirely inferred from existing topological information in the half-edge data structure (bottom).

4.3 Cycle ordering of cut-edges

In the algorithm above, the only step that seemingly requires spatial coordinates is the edge-cycle ordering around vertices. One can of course apply exact arithmetic, but we can significantly reduce the cost of plane cutting in Sec. 4.2 by exploiting the following properties. The line $(\overline{f}, p)$ and H^k intersect at cut-vertices, which split the line $(\overline{f}, p)$ into several segments, and only the inner segments cut $\overline{f}$ (see Sec. 3.2). Each inner segment only shares two cut-vertices with H^k at its two ends. The only difficulty in deducing the order is if one of the two ends connects more than two half-edges in H^k. Indeed, the edge-cycle ordering is uniquely defined from H^k and its two ends v_a, v_b in the following cases:

(a) there are only two half-edges in H^k adjacent to each of the two ends, even if there is not a single loop connecting them.
(b) there is only one loop (or equivalence class of "next") containing both ends, and in this loop, there exists only two half-edges adjacent to v_a, v_b.

In practice, most cycle orderings in Sec. 4.2 fall into one of these two cases, and we can directly infer the next of a cut-edge without resorting to any numerical computation. Such a shortcut only involves topological information represented by tuples and triples, and we name it a "topological cut" to distinguish it from the far more costly "numerical cut". For the few cases where we cannot use our topological treatment because the two cases above do not apply, we revert to exact arithmetic evaluation, see Fig. 17.

4.4 Cut-faces in $H(\overline{f})$

The cut-faces in $H(\overline{f})$ are indeed half-edge loops in $H(\overline{f})$: for each half-edge $\vec{h}$ in $H(\overline{f})$, we use the *next* pointer to visit all next half-edges starting from $\vec{h}$, and get all starting cut-vertices from them, which are the boundary-ordered cut-vertices of a cut-face. The reversed loop is its opposite half-face.

5 CUT-CELL GENERATION VIA HALF-FACES

Having all the cut-faces as well as the half-edge structures $H(\overline{f})$ for each $\overline{f} \in \overline{\mathcal{F}}$ at hand, it is time to generate the cut-cells that they bound. The algorithm is very similar to the generation of cut-faces: we first build the initial half-face structure from all the boundary cut-faces, then cut the resulting original cell enclosed by the initial half-face structure recursively by each cut plane. The key is to seek the "adjacent" face of each cut-face in a cut-cell, so that these two faces properly form a wedge on an edge of a cut-cell; and of course, we use face-cycle ordering around cut-edges instead of cut-vertices.

5.1 Adjacent half-face

While the "next" operation in a half-edge data structure is well known, the "adjacent" operation (see Fig. 19) is unusual, so we discuss it further before elaborating on our algorithm. Remember that the "adjacent" operation of a half-face $\vec{f} = \overrightarrow{f_1 f_2 \ldots f_n}$ is a function

$$\mathrm{adj}(\vec{f}, e) : \vec{\mathcal{F}} \otimes \mathcal{E} \to \vec{\mathcal{F}}, \tag{8}$$

where $e = (v_a, v_b), v_a, v_b \in \mathcal{V}$ is part of $\vec{f}$. We first orient e into $\vec{e}$ according to the orientation of $\vec{f}$, then the oriented cut-edge $\vec{e}$ defines, through face-cycle ordering around it, the next cut-face $\vec{g} = \overrightarrow{g_1, g_2, \ldots, g_m}$ of $\vec{f}$. It is then easy to verify that

$$\mathrm{adj}(\mathrm{adj}(\vec{f}, e), e) = \vec{f}. \tag{9}$$

Consequently, the inherited face normal points towards the outside of the cell. Next, we show how to bypass cycle-ordering through topological inference. For example, if a cut-edge is adjacent to only two cut-faces, the two associated half-faces are obviously adjacent.

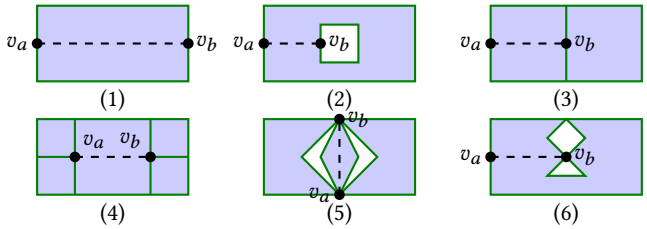


Fig. 17. **Ambiguous vs. unambiguous 2D topological cuts.** Cases (1) and (2) satisfy condition (a); cases (3) and (4) satisfy condition (b); but cases (5) and (6) cannot be done through topological cut.

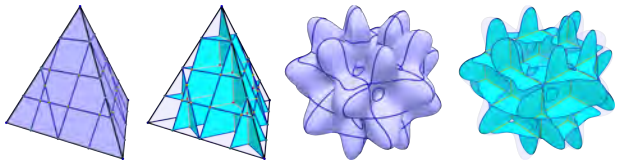


Fig. 18. **Cut-faces** on the boundary and on cut planes for the cutting of a tetrahedron domain (left) and a bumpy-torus domain (right).

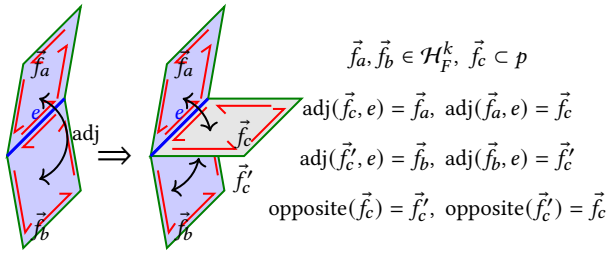


Fig. 19. **"Adjacent" operation.** The *adjacent* operation is a function of a half-face and a cut-edge, detailed in this schematic description.

5.2 Initial half-face structure from boundary triangles

The initial half-face structure only includes boundary entities, i.e., cut-vertices, cut-edges and cut-faces from the half-edge data structures $H(\bar{t})$ created from all the boundary triangles $\bar{t} \in \overline{\mathcal{T}}$. All such entities are encoded by the index of distinct vertices, and removing duplication is thus simple to achieve. Since the half-edge data structures created for boundary triangles are consistent with the (outward) face normals, the original order $\overrightarrow{f_1 f_2 \ldots f_n}$ defined by the "next" operation on a half-edge points outwards too. The opposite of a boundary half-face is set to $\emptyset$, while for inner half-faces generated in the previous section, it is set to the reversed sequence. Similar to the half-edge data structure, the "opposite" pointer is irrelevant to the correctness of the output, but helps with query efficiency.

Given all boundary half-faces and boundary cut-edges, we simply visit each cut-edge and find its adjacent half-face for all the half-faces containing this cut-edge. Most cut-edges on the boundary now have only two adjacent half-faces (on the input boundary), and we know that they are adjacent to each other without resorting to any numerical calculation. Only when the cut-edge is on a non-manifold edge of $\overline{\mathcal{E}}$ do we need to use the face-cycle orders of adjacent triangles to get the adjacency relationship. The resulting half-face structure is denoted as $\mathcal{H}_F^0$, representing the input mesh $\mathcal{M}$ cut by all the cut planes.

5.3 Cutting by planes

Beginning with $\mathcal{H}_F^0$, we iteratively cut the current half-face data structure $\mathcal{H}_F^k$ by a cut-plane $p \in \mathcal{P}$, into a "refined" structure $\mathcal{H}_F^{k+1}$. We first eliminate the cut-faces in $H(p)$ already in $\mathcal{H}_F^k$ by checking its sequence of vertex ids (i.e., triples), then assemble the remaining cut-faces (denoted $\mathcal{F}_p$ here) into $\mathcal{H}_F^k$. After splitting each cut-face into a pair of opposite half-faces (remember that each of the two half-faces has a predefined normal according to $H(p)$ with which their orientation is consistent, and that these two normals are opposite), the final task is to compute the adjacent face for all related half-faces:

$$\{(\vec{f}, e) | f \in \mathcal{F}_p \cup \mathcal{H}_F^k, e \in \mathcal{E}_p\}, \tag{10}$$

where $\mathcal{E}_p$ indicates the cut-edges in $\mathcal{F}_p$. If e is not in $\mathcal{H}_F^k$, it is only adjacent to cut-faces in $\mathcal{F}_p$, and the computation of adjacent half-face is trivial. If e is also in $\mathcal{H}_F^k$, it is at the intersection of $\mathcal{H}_F^k$ and p (i.e., $\mathcal{F}_p$). One can thus use the face-cycle ordering around such cut-edges to compute the adjacent function for all the related half-faces in both $\mathcal{H}_F^k$ and $\mathcal{F}_p$. As for the cut-face generation, many of these face-cycle orderings can be replaced by topological cuts just as before, without loss of exactness.

Once all the half-faces and associated adjacency are set, we extract the cut-cells. Similar to the extraction of cut-faces, it is all about equivalence classes. We say two half-faces $\vec{f}$ and $\vec{g}$ are equivalent if they are adjacent to each other on a shared cut-edge e, i.e.,

$$\text{adj}(\vec{f}, e) = \vec{g}. \tag{11}$$

Intuitively, we pick a half-face as a seed and recursively visit all its adjacent half-faces by pivoting along each of its boundary cut-edges. All visited half-faces are then the boundary of the cell on the positive side of the seed half-face.

5.4 Cycle ordering of cut-faces

Between the half-face structure $\mathcal{H}_F^k$ and plane p, there are common cut-edges, and these cut-edges split p into several 2D sub-regions: each sub-region has intersected $\mathcal{H}_F^k$ at one or more loops $\{\vec{\ell}_i\}$ composed by cut-edges. As in the edge-cycle ordering case, there are two cases that allow us to skip numerical evaluation altogether when performing the cut:

(a) Only two adjacent half-faces exist in $\mathcal{H}_F^k$ for each cut-edge of loop $\vec{\ell}_i$.

(b) There is only one equivalent class in $\mathcal{H}_F^k$ containing all the cut-edges of the loop, and in this equivalent class, there exists only two adjacent half-faces for each cut-edges of the loop $\vec{\ell}_i$.

Obviously, if each intersection loop between $\mathcal{H}_F^k$ and p satisfies condition (a) or (b), the insertion is unique, and we call it a 3D topological cut, see Fig. 20. If a loop satisfies condition (a) and (b), we can update the *adjacent* pointer of half-faces in $\mathcal{H}_F^k$ and p that are adjacent to cut-edges in the loop; otherwise, we use face-cycle sorting to find the adjacent half-faces. Once we get the adjacent half-faces by topological cut or cycle-ordering around common cut-edges $e \in \ell_i$, we can update the *adjacent* pointer as explained in Fig. 19. After inserting all cut planes, we get the final half-face structure $\mathcal{H}_F$. All we have to do is then to extract the equivalence class for each half-face to get the cut-cells of our domain decomposition.

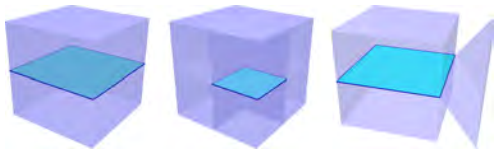


Fig. 20. **Unambiguous vs. ambiguous 3D topological cuts.** The left-side example satisfies condition (a), the middle one satisfies condition (b), but the example on the right side does not satisfy any of these two conditions, and thus requires a full numerical treatment. Blue loops are the intersection lines between the cut plane (light blue) and the current half-face structure.

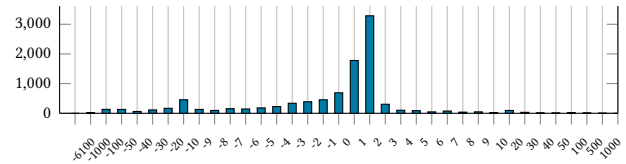


Fig. 21. **Tetwild dataset.** Distribution of the Euler characteristic for our input meshes from [Hu et al. 2018].

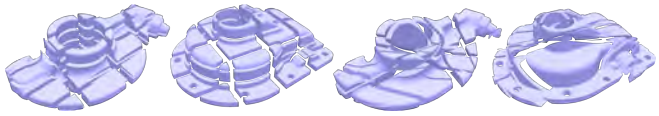


Fig. 22. **RockerArm and Cast meshes.** Left: domains shaped as RockerArm and Caster meshes, with axis-aligned cuts; right: same for arbitrary cuts.

6 RESULTS

In order to show robustness and efficiency of our method, we tested our implementation on a large number of models from TetWild [Hu et al. 2018] and compared our TopoCut method with Mandoline [Tao et al. 2019], libigl [Panozzo and Jacobson 2019] and VolumeMesher [Diazzi and Attene 2021] whose codes are available. Additional results are also included in the supplemental material. For completeness, we provide our implementation as an open-source library at `https://xzfang.top/topocut`.

Dataset. Based on the output tetrahedral meshes of TetWild [Hu et al. 2018], we extract their boundaries to get a total of 9841 triangle meshes. These meshes are very diverse, both in complexity and genus (see their Euler characteristic distribution in Fig. 21). For instance, the triangle mesh in Fig. 23 is a high-genus case.

Cut planes. We tested mostly two types of mesh cutting scenarios: one where the cut planes are uniform spaced in the bounding box as in the typical cut-cell mesh generation (we will refer to this case as the $n\times n\times n$ case (or "An") to indicate that there are n cut planes perpendicular to each axis); and one where arbitrary planes are generated by randomly choosing pairs $(\mathbf{n}_p, d_p)$ (we will refer to this case as "Rn"). In Fig. 22, the left two examples use axis-aligned cut planes, the right two examples use random cut planes.

Line sorting speedup. To speed up our line sorting, we adopt an adaptive approach: we use floating-point numbers to perform coarse sorting, then use exact numerical calculation for fine sorting when the distance between two adjacent cut-vertices is smaller than 10^{-3} times the average edge length of the input triangle mesh.

Output. The connectivity information of the resulting volumetric decomposition is now easily extracted from the constructed half-edge and half-face structures: our method outputs the cut mesh through a set of polyhedra formed by several boundary polygons, and a set of coordinates for the cut-vertices that are represented by triples of triangles/planes indices.

6.1 Post-processing for downstream applications

Cut-faces and cut-cells with multiple boundaries. As mentioned in Sec. 4.4, a cut-face is an oriented boundary loop, which is associated to an oriented plane or an oriented boundary triangle (see Fig. 15). When the corresponding normals are opposite, the cut-face is in fact a "hole" rather than a "solid" (see the inner cylinder boundary in inset) as already discussed in Sec. 2.2. In case the output mesh should use cut-faces only as "solid" planar regions, one can easily add an inner cut line in order to join two boundaries into a single one — possibly repeatedly if there are more than one hole. The added inner line cuts will guarantee that each cut-face has only one boundary which encloses a solid region. The same principle applies to cut-cells: the boundary of a cut-cell is an oriented watertight polyhedron composed of a set consistently oriented half-faces. If the half-faces form a "void" (i.e., a negative volume, when all of its half-faces come from voids in the input mesh), one can use ray-casting to find the first solid boundary enclosing it; then, this solid boundary and all the inner void boundaries form a solid cell.

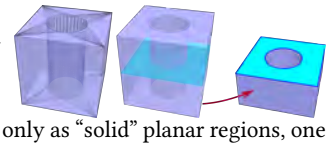


Table 2. **Self-intersection testing statistics.** First row indicates the ratio of results *without self-intersection* for each type of cutting scenario for the dataset tests, where An refers to $n\times n\times n$ grid cuts while Rn indicates the use of n arbitrary cuts.

	A5	A10	A20	R10	R20	R30	R100
Ours	95.1%	98.3%	98.4%	100%	99.99%	99.98%	99.98%
Mandoline	<92%	<84%	<76%	n/a	n/a	n/a	n/a

Floating-point coordinates. While our approach uses triples of faces to refer to a cut-vertex, one may need to embed the final cut mesh to use it as an input for fluid simulation or any other downstream application. While we can easily use rational or arbitrary-precision floating-point numbers [GMP 2021] without affecting the validity of the resulting cut mesh, the use of 32-bit or 64-bit floating-point coordinates is often needed in standard geometry processing libraries. However, directly converting rational coordinates into floating-point ones can result in both *degenerate elements* and *self-intersections* due to this vertex rounding [Milenkovic and Nackman 1990; Fortune 1997]: our topological data structure is no longer matching the topology of this rounded-coordinate embedding. Theoretically, one can perform a conversion from rational coordinates to doubles which prevents degeneracy (in the sense of length, area and volume respectively) and self-intersection by enforcing the proper vertex-vertex (line-order) and vertex-face (Eq. (13)) orders: the final embedding of all cut-vertices $\{v_i\}$ can be found by minimizing the sum of their squared distances to their associated positions $\{v_i^*\}$ directly obtained via forceful casting from rational coordinates to double coordinates, *subject to* preserving all the orders along cut-lines and all sidedness of cut-vertices with respect to cut-faces. It should be noted that after rounding, vertices of a planar polygon may not be co-planar anymore, which will require to change the formal definition of a face or of the vertex-vertex order along a line to account for the added perturbation. However, such an optimization can be time-consuming due to the number of constraints to check. We thus employ a heuristic instead: we first convert all coordinates from rational representation to double floating-point representation; then, for each intersection line, we get its normalized direction t in double representation using the beginning and the end points on it. Then, we sweep through the points along the line to verify their order by checking that $(v_{i+1}-v_i)\cdot t > 0$; if a vertex v_{i+1} fails this test, we update its coordinates by displacing them along t by the

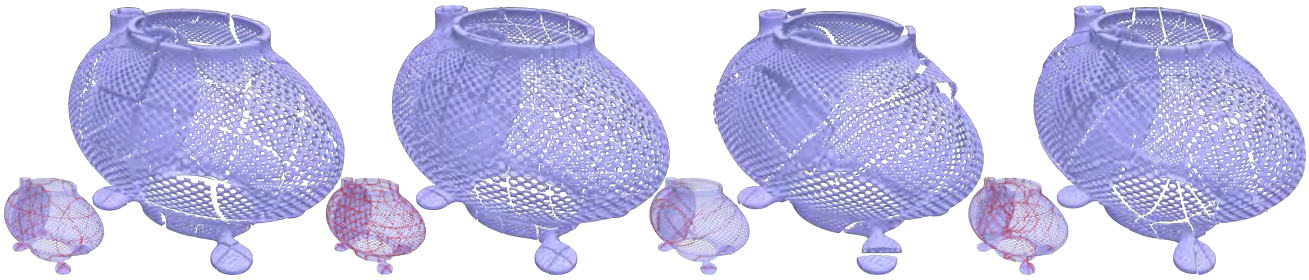


Fig. 23. **Robustness to high genus.** A high genus input mesh with 5×5×5 and 10×10×10 cuts, and with 10 and 20 random cuts.

smallest double floating-point change that satisfies the ordering test. In most cases, no displacement is even needed. We then consider the vertex-face orders: for each vertex, we check all the near faces; if one inequality constraint is not satisfied, we move the vertex position along the face normal until the constraint is satisfied. In a few cases, these local checks will not necessarily lead to a proper global embedding, unfortunately; but we will demonstrate in Sec. 6.4 that this simple heuristic already outperforms existing approaches relying on 64-bit floating-point coordinates, since we only incur errors during this *final projection* instead of accumulating numerical errors during ordering operations, which can lead to dramatic failures (Fig. 26).

6.2 Robustness and correctness

In the nearly 10K triangle meshes we tried, we never failed to generate the final half-face structure of the expected cut-mesh, see Fig. 29 and Tab. 4. We confirmed that the Euler characteristics for all input meshes match the ones of our output cut-meshes match. We also calculated the volume of the input triangle meshes vs. our output cut-meshes embedded using floating-point coordinates (either through direct conversion from rational coordinates or through the heuristic presented in Sec.6.1), and the average relative error was always smaller than 10^{-4}. Finally, we provide statistics of cell volumes over all the cutting tests we made in Tab. 3.

Table 3. **Volume statistics.** With An referring to $n\times n\times n$ grid cuts and Rn indicating n arbitrary cuts, we show the average minimum and maximum cell volumes after cutting all the models in the dataset.

cut type	A5	A10	A20	A50	R10	R20	R30	R100
min vol.	3.9e-4	1.3e-5	8.5e-7	2.6e-8	3.9e-5	2.3e-6	1.8e-7	8.7e-9
max vol.	4.6e-2	6.7e-3	9.8e-4	1.2e-4	0.19	0.10	0.065	0.013

6.3 Performance

All our tests were run on a PC with an Intel Xeon® W-2255 CPU and 64GB RAM. We provide in Fig. 24 (top) the computational time for the cutting of a domain by different choices of cut-planes, each averaged over the entire dataset. Note that our timings include the post-processing steps as well, involving rounding and adjustment of coordinates of cut-vertices, the connection of multi-boundary cut-faces, and the triangulation of cut-faces. We also profiled the cutting process by timing each step, see Fig. 24 (bottom). Finally, it is worth mentioning than when performing axis-aligned cuts, we do *not* leverage the prior knowledge that planes are parallel or orthogonal to each others; so the difference in execution time between regular cuts and arbitrary cuts is virtually nonexistent (modulo mesh-specific properties).

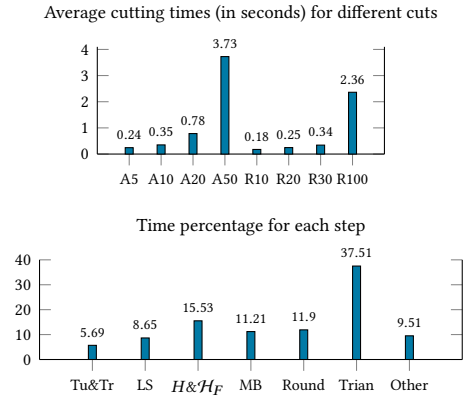


Fig. 24. **Computational efficiency.** Top: our computational time averaged over the whole dataset for different cutting planes, where Rn indicates the use of n arbitrary cuts while An refers to $n\times n\times n$ grid cuts. Down: Profiling each step: computation of tuples and triples (Tu&Tr), line sorting (LS), half-edge and half-face structure ($H\&\mathcal{H}_F$) generation, multi-boundary cut-face post-processing (MB), rounding & adjustment of cut-vertices (Round), triangulation of cut-faces (Trian) and other.

Numerical sorting vs. topological sorting. In practice, our topological cut idea induces a surprisingly huge boost in performance. To understand why, we counted the number of cut-vertices and cut-edges that needed numerical cycle-sorting evaluation because topological cutting was ambiguous, compared to the total number of cycle-sorting evaluations that would have been needed without TopoCut. As shown in the histogram in Fig. 25, *very few* domains require numerical evaluation for cycle ordering. In fact, when using 5×5×5 cut planes, there are 8,158 input meshes among the 9,841 meshes of the dataset that do not need numerical sorting for cut-vertices at all, and 7,858 input meshes that do not need numerical sorting for cut-edges at all. Because the topological sorting only involves the *opposite* and *next* operations in the half-edge or half-face data structure, the computational cost of a topological cut is marginal compared to the numerical cycle-sorting version, explaining our improved performance.

6.4 Comparisons

Using the nearly 10K-mesh dataset, we compare our method with the multithreaded implementation of Mandoline provided by the authors for 10 threads in Fig. 30. As mentioned before, we do not use any prior on the planes being axis aligned. On average, our method achieves a speed-up factor of two to three, see Fig. 28. Moreover,

while it claims "experimental robustness", Mandoline can fail in practice on fairly simple input meshes due to their use of floating-point arithmetic. As shown in Fig. 26, the staircase triangle mesh is cut by three different set of axis-aligned cut planes (4×4×4, 5×5×5, and 6×6×6 respectively) covering exactly the bounding box of the input mesh (and thus creating a slew of degenerate cases); Mandoline succeeds on the first two, but fails on the middle one: their approach uses fast numerical evaluations that often provide correct results, but do not guarantee correctness. Fig. 27 shows another case where Mandoline returns incomplete cells or improperly includes outside elements as the connectivity information is incorrect, due to inaccurate cycle sorting. Finally, we also show in Tab. 2 that after embedding our results using floating-point coordinates, our results also contain far less self-intersecting outputs as tested by Cinolib [Livesu 2019] after triangulating all the cut-faces into triangles without adding any new vertex: when cutting using a 10×10×10 grid for instance, our approach has only 167 self-intersecting meshes, while Mandoline has 1,600+ self-intersecting meshes out of 9,841 input meshes. We also compare TopoCut with the state-of-the-art mesh boolean method VolumeMesher [Diazzi and Attene 2021] using their own code to process $n \times n \times n$ grid cuts. Fig. 30 shows various performance comparisons for axis-aligned cut planes, showing that we improve on VolumeMesher by an average factor of three for 5×5×5 grid cuts, and a factor of five on 10×10×10 grid cuts in efficiency. Finally, we compare TopoCut with the mesh boolean method of [Zhou et al. 2016] implemented in libigl [Panozzo and Jacobson 2019], and our tests show that TopoCut is about an order of magnitude faster.

7 CONCLUSIONS

Cutting a complex domain by arbitrary planes efficiently is challenging: for the sake of efficiency, numerical approximations that cannot guarantee correctness or robustness are too often used. In this paper, we construct the geometric elements of a generalized cut-cell mesh in a bottom-up fashion, and carefully use topological information already gathered earlier to replace most of the exact arithmetic computations while still guaranteeing the same construction — an approach that we called topological cutting. Experimental results show that topological cuts are surprisingly ubiquitous in practice, with only a small percentage of exact arithmetic computations needed as a fallback solution in case of topological ambiguity. Moreover, TopoCut applies to arbitrary cuts instead of only the typical axis-aligned cuts to which most of the existing libraries are restricted. Despite this added generality, we outperform all state-of-the-art approaches for axis-aligned cut-cell mesh generation.

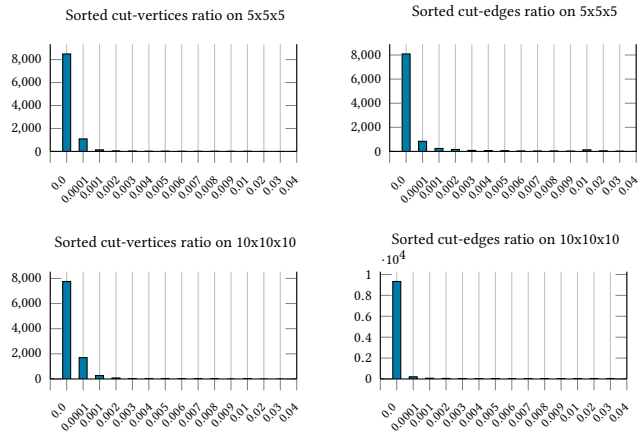


Fig. 25. **Topological vs. Numerical Cuts**. In this histogram, we evaluate the frequency of 2D or 3D topological cuts vs. numerical ordering around cut-vertices or cut-edges. Left: Horizontal axis represents the ratio of numerically-sorted cut-vertices, and the vertical axis is the number of triangle meshes. Right: same histograph, but for cut-edges this time.

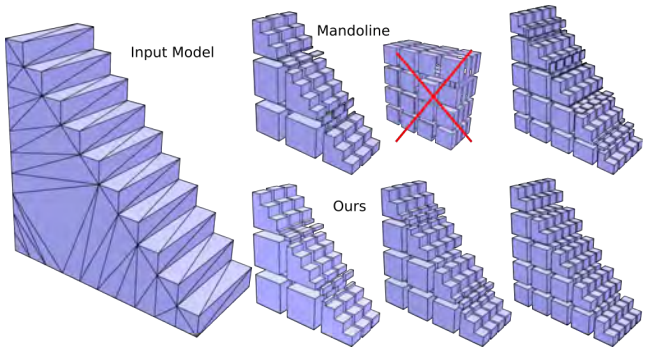


Fig. 26. **Staircase triangle mesh.** Left to right: For a staircase domain (left), we use 4×4×4, 5×5×5 and 6×6×6 grid cuts. Mandoline fails sometimes (top), while our approach always outputs the correct cut-cell meshes.

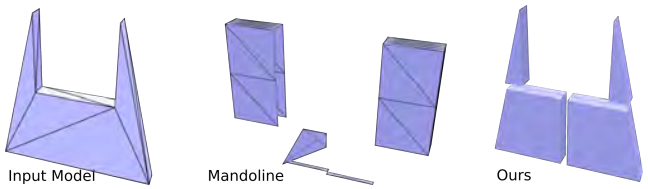


Fig. 27. **Simple fail case for Mandoline.** Left to right: input triangle mesh, the result of Mandoline, and our result using the same set of cut planes.

Table 4. Statistics for the domain cutting examples shown in this paper.

model	$\lvert\overline{\mathcal{V}}\rvert$	$\lvert\overline{\mathcal{T}}\rvert$	$\lvert\mathcal{P}\rvert$	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\lvert\mathcal{F}\rvert$	$\lvert\mathcal{C}\rvert$	T(sec)	$\lvert\mathcal{P}\rvert$	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\lvert\mathcal{F}\rvert$	$\lvert\mathcal{C}\rvert$	T(sec)
armadillo	21k	43k	15	26094	73565	47648	60	0.531	10	25616	72602	47142	41	0.438
block	2k	4k	15	3827	9510	5852	64	0.077	10	3663	9240	5744	80	0.0728
bumpySphere	6k	11k	15	8353	22282	14093	65	0.15	10	8310	22223	14073	71	0.131
bumpyTorus	17k	34k	15	21961	60559	38779	60	0.343	10	21949	60607	38852	102	0.351
bunnyBotsch	55k	111k	15	62548	180596	118211	52	1.203	10	62012	179490	117636	40	1.205
carter	30k	60k	15	41747	113897	72832	468	0.888	10	35336	100503	65305	28	0.599
cast	20k	40k	15	25345	70543	45352	50	0.448	10	24592	69028	44587	47	0.436
dragonstand	52k	104k	15	60632	173267	112828	77	0.89	10	59524	171037	111695	64	0.914
elephant	25k	50k	15	30121	85013	55068	61	0.487	10	30927	86644	55889	63	0.485
gearbox	64k	128k	15	75977	216400	140480	65	1.346	10	71167	206747	135604	42	1.176
rockerArm	10k	20k	15	13741	37352	23783	57	0.222	10	13572	37052	23650	72	0.229
siggraph	20k	40k	15	30428	80810	50558	92	0.553	10	28543	77007	48653	99	0.484

Limitations. One main limitation of our work, common to all previous works which are not using exact arithmetic systematically, is that the conversion of the index triples of cut-vertices into floating-point coordinates cannot guarantee the absence of self-intersection when assuming straight-boundary face and cells. Developing a better heuristic or guaranteed projection to double-precision coordinates may require a better strategy to enforce the topological conditions we mentioned, or even relaxing the exact straightness of the cutting planes and cut-lines.

Future work. Besides a better final embedding of our results to guarantee exactness, we point out that our approach is sharply different from previous approach as we do *not* rely on the straightness

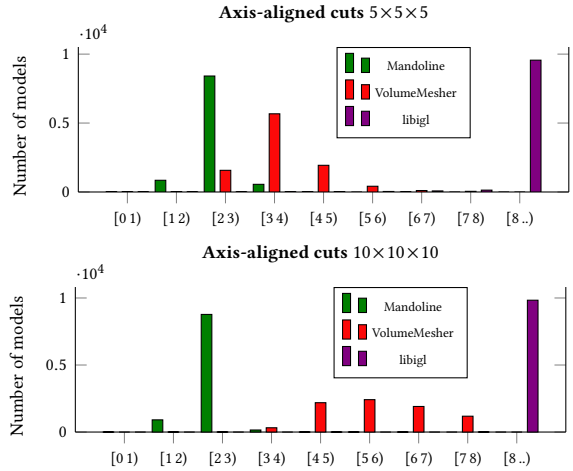


Fig. 28. **Efficiency comparisons.** Histograms show binning of acceleration factors with respect to [Tao et al. 2019] (i.e., Timing(Mandoline)/Timing(TopoCut)), [Diazzi and Attene 2021] and [Zhou et al. 2016], where the vertical axis indicates the input model counts for which this factor is achieved.

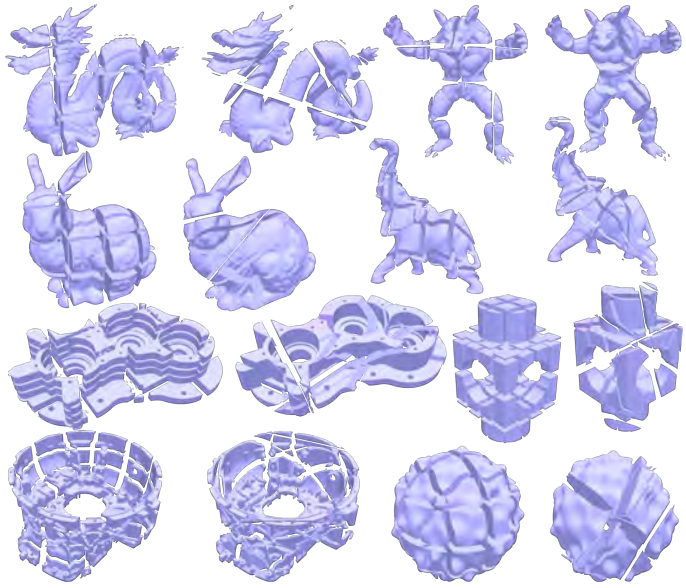


Fig. 29. **Cutting Zoo.** A few selected results of our method for axis-aligned and arbitrary cuts, see Tab. 4 for details.

of the cuts. It would thus be interesting to see if we can extend our algorithm to *curved cuts*: if the cuts remain 2-manifold and their pairwise intersections remain 1-manifold, TopoCut should still apply — as long as the computations of cut-vertices do not become overly expensive. It may also help to extend this idea to more general Boolean operations involving curved lines/faces.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their comments and suggestions. Most of models we used for testing are originally from Thingiverse (right of Fig. 1, Fig. 23), Stanford Scanning Repository (Bunny, Armadillo, Dragon) and AIM@SHAPE Repository (Figs. 10, 22 and 29). J. Huang was supported by National Key R&D Program of China (No. 2020AAA0108901) and Zhejiang Provincial Science and Technology Program in China under Grant 2021C01108. X. Fang acknowledges support from Zhejiang Provincial Natural Science Foundation of China under Grant No. LQ22F020025, and the Open Project Program of the State Key Lab of CAD&CG (Grant No. A2201), Zhejiang University. MD acknowledges the generous support of a Choose France Inria chair, and of Ansys, Inc.

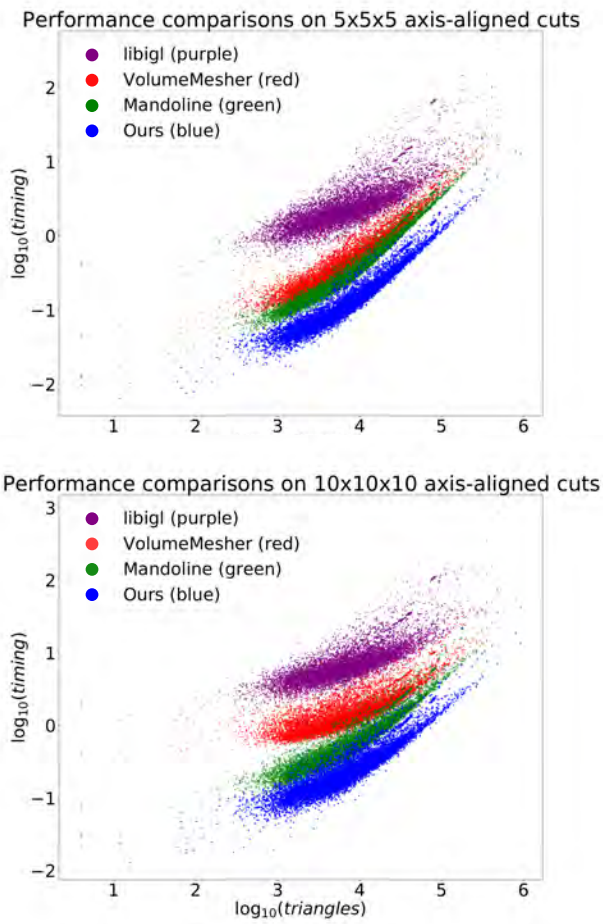


Fig. 30. **Performance comparisons.** Log-log plot showing computational times for VolumeMesher (red), Mandoline (green), libigl (purple), and TopoCut (blue) as a function of triangle count for various models of TetWild.

REFERENCES

M. J. Aftosmis, M. J. Berger, and J. E. Melton. 1998. Robust and Efficient Cartesian Mesh Generation for Component-Based Geometry. *AIAA Journal* 36, 6 (1998), 952–960. https://doi.org/10.2514/2.464

Marco Attene. 2020. Indirect Predicates for Geometric Constructions. *Computer-Aided Design* 126 (2020), 102856. https://www.sciencedirect.com/science/article/pii/S001044852030049X

Vinicius C. Azevedo, Christopher Batty, and Manuel M. Oliveira. 2016. Preserving Geometry and Topology for Fluid Flows with Thin Obstacles and Narrow Gaps. *ACM Trans. Graph.* 35, 4, Article 97 (July 2016), 12 pages.

Chandrajit L. Bajaj and Valerio Pascucci. 1996. Splitting a Complex of Convex Polytopes in Any Dimension. In *Proceedings of the Twelfth Annual Symposium on Computational*

Geometry (SCG '96). Association for Computing Machinery, New York, NY, USA, 88–97.
Gilbert Bernstein and Don Fussell. 2009. Fast, Exact, Linear Booleans. *Computer Graphics Forum* 28, 5 (2009), 1269–1278.
Marcel Campen and Leif Kobbelt. 2010. Exact and Robust (Self-)Intersections for Polygonal Meshes. *Computer Graphics Forum* 29, 2 (2010), 397–406.
CGAL. 2021. CGAL - Computational Geometry Algorithms Library. https://www.cgal.org
Gianmarco Cherchi, Marco Livesu, Riccardo Scateni, and Marco Attene. 2020. Fast and Robust Mesh Arrangements Using Floating-Point Arithmetic. *ACM Trans. Graph.* 39, 6, Article 250 (Nov. 2020), 16 pages.
Olivier Devillers, Sylvain Lazard, and William J. Lenhart. 2018. 3D Snap Rounding. In *34th International Symposium on Computational Geometry (SoCG 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Bettina Speckmann and Csaba D. Tóth (Eds.), Vol. 99. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 30:1–30:14.
Lorenzo Diazzi and Marco Attene. 2021. Convex Polyhedral Meshing for Robust Solid Modeling. *ACM Trans. Graph.* 40, 6, Article 259 (dec 2021), 16 pages.
Essex Edwards and Robert Bridson. 2014. Detailed Water with Coarse Grids: Combining Surface Meshes and Adaptive Discontinuous Galerkin. *ACM Trans. Graph.* 33, 4, Article 136 (July 2014), 9 pages.
F.R. Feito and J.C. Torres. 1997. Inclusion test for general polyhedra. *Computers & Graphics* 21, 1 (1997), 23 – 30.
Steven Fortune. 1997. Vertex-Rounding a Three-Dimensional Polyhedral Subdivision. *Discrete Comput. Geom* 22 (1997), 116–125.
GMP. 2021. GMP - The GNU Multiple Precision Arithmetic Library. https://gmplib.org
Yixin Hu, Qingnan Zhou, Xifeng Gao, Alec Jacobson, Denis Zorin, and Daniele Panozzo. 2018. Tetrahedral Meshing in the Wild. *ACM Trans. Graph.* 37, 4, Article 60 (July 2018), 14 pages.
J. Jaśkowiec, P. Pluciński, and A. Stankiewicz. 2016. Discontinuous Galerkin method with arbitrary polygonal finite elements. *Finite Elements in Analysis and Design* 120 (2016), 1 – 17.
Yehuda E Kalay. 1982. Determining the spatial containment of a point in general polyhedra. *Computer Graphics and Image Processing* 19, 4 (1982), 303 – 334.
Marco Livesu. 2019. *Cinolib: A Generic Programming Header Only C++ Library for Processing Polygonal and Polyhedral Meshes.* Springer Berlin Heidelberg, Berlin, Heidelberg, 64–76.
Matthias Meinke, Lennart Schneiders, Claudia Günther, and Wolfgang Schröder. 2013. A cut-cell method for sharp moving boundaries in Cartesian grids. *Computers & Fluids* 85 (2013), 135 – 142. International Workshop on Future of CFD and Aerospace Sciences.
V. J. Milenkovic and L. R. Nackman. 1990. Finding compact coordinate representations for polygons and polyhedra. *IBM Journal of Research and Development* 34, 5 (1 Jan. 1990), 753–769.
Julius Nehring-Wirxel, Philip Trettner, and Leif Kobbelt. 2021. Fast Exact Booleans for Iterated CSG using Octree-Embedded BSPs. *Computer-Aided Design* 135 (2021), 103015.
OpenMesh. 2021. OpenMesh - A generic and efficient polygon mesh data structure. https://www.openmesh.org
OpenVolumeMesh. 2021. OpenVolumeMesh - A Generic and Versatile Index-Based Data Structure for Polytopal Meshes. https://www.openvolumemesh.org
Daniele Panozzo and Alec Jacobson. 2019. libigl: Prototyping Geometry Processing Research in C++. In *Eurographics 2019 - Tutorials*, Wenzel Jakob and Enrico Puppo (Eds.). The Eurographics Association. https://doi.org/10.2312/egt.20191037
Taylor Patterson, Nathan Mitchell, and Eftychios Sifakis. 2012. Simulation of Complex Nonlinear Elastic Bodies Using Lattice Deformers. *ACM Trans. Graph.* 31, 6, Article 197 (2012), 10 pages.
Jonathan R. Shewchuk. 1997. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete Comput. Geom.* 18, 3 (1997), 305–363.
Eftychios Sifakis, Kevin G. Der, and Ronald Fedkiw. 2007. Arbitrary Cutting of Deformable Tetrahedralized Objects. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '07)*. Eurographics Association, Goslar, DEU, 73–80.
Michael Tao, Christopher Batty, Eugene Fiume, and David I. W. Levin. 2019. Mandoline: Robust Cut-cell Generation for Arbitrary Triangle Meshes. *ACM Trans. Graph.* 38, 6, Article 179 (Nov. 2019), 17 pages.
Charlie C. L. Wang and Dinesh Manocha. 2013. Efficient Boundary Extraction of BSP Solids Based on Clipping Operations. *IEEE Transactions on Visualization and Computer Graphics* 19, 1 (Jan. 2013), 16–29.
Chee K. Yap and Vikram Sharma. 2008. *Robust Geometric Computation.* Springer US, Boston, MA, 788–790.
Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. 2016. Mesh Arrangements for Solid Geometry. *ACM Trans. Graph.* 35, 4, Article 39 (July 2016), 15 pages.

A INPUT CONDITION

When the mesh $\mathcal{M}$ has no duplicated vertices, it is called free of self-intersection if:

$$\forall e \in \overline{\mathcal{E}}, t \in \overline{\mathcal{T}},\ e \cap t = \emptyset \text{ or } e \cap t = e \text{ or } e \cap t = \text{vert}(e) \cap \text{vert}(t).$$

That is, for all pairs made of an edge and a triangle, they are separate, or intersecting, or the edge is one of the boundary edge of the triangle, or they share a common vertex. Otherwise, self-intersection is present.

B TUPLES AND TRIPLES GENERATION

For completeness, we provide implementation details about tuples and triples generation next.

Tuples of $(\overline{\mathcal{T}}, \overline{\mathcal{T}})$. For two triangles, $t_i, t_j \in \overline{\mathcal{T}}$, $(t_i, t_j) \in (\overline{\mathcal{T}}, \overline{\mathcal{T}})$ if $t_i \cap t_j \in \overline{\mathcal{E}}$.

Tuples of $(\mathcal{P}, \mathcal{P})$. In order to generate $(\mathcal{P}, \mathcal{P})$ easily, we compute a function T_{pp} such that for two given planes $p_i(\mathbf{n}_i, d_i)$ and $p_j(\mathbf{n}_j, d_j) \in \mathcal{P}$, and with $\mathbf{s}_{ij} = \text{sign}(\mathbf{n}_i \times \mathbf{n}_j)$, $s_d = \text{sign}((d_j\mathbf{n}_i - d_i\mathbf{n}_j)^T\mathbf{n}_i)$, we define

$$T_{pp}(p_i, p_j) = \begin{cases} \emptyset, & \mathbf{s}_{ij} = \mathbf{0}, s_d \neq 0 \text{ (Parallel)}, \\ \boxminus, & \mathbf{s}_{ij} \neq \mathbf{0} \text{ (Intersected)}, \\ \blacksquare, & \mathbf{s}_{ij} = \mathbf{0}, s_d = 0 \text{ (Coplanar)}. \end{cases} \tag{12}$$

Then trivially, $(p_i, p_j) \in (\mathcal{P}, \mathcal{P})$ if $T_{pp}(p_i, p_j) = \boxminus$.

Tuples of $(\overline{\mathcal{T}}, \mathcal{P})$. In order to generate $(\overline{\mathcal{T}}, \mathcal{P})$, we define a function T_{pv} dependent on the relation between input vertices $\overline{\mathcal{V}}$ and input cut planes $\mathcal{P}$: given a plane $p \in \mathcal{P}$ and a vertex $\overline{v} \in \overline{\mathcal{V}}$, define

$$T_{pv}(p, \overline{v}) = \text{sign}(\mathbf{n}_p^T\overline{v} + d_p) \in \{-1, 0, 1\}. \tag{13}$$

Here, $T_{pv}(p, \overline{v}) = 0$ means $\overline{v} \in p$, i.e., $\overline{v}$ is on the plane p; a value of 1 means instead that $\overline{v}$ is on top of the plane, and -1 means that $\overline{v}$ is under the plane. By using T_{pv}, we also can easily deduce the relationship T_{pt} between $\overline{\mathcal{T}}$ and $\mathcal{P}$: given $t \in \overline{\mathcal{T}}$ and $p \in \mathcal{P}$,

$$T_{pt}(p, t(\overline{v}_i, \overline{v}_j, \overline{v}_k)) = \begin{cases} \emptyset, & T_{pv}(p, \overline{v}_i) = T_{pv}(p, \overline{v}_j) = T_{pv}(p, \overline{v}_k) \neq 0, \\ \boxdot, & |T_{pv}(p, \overline{v}_i) + T_{pv}(p, \overline{v}_j) + T_{pv}(p, \overline{v}_k)| = 2, \\ \blacksquare, & T_{pv}(p, \overline{v}_i) = T_{pv}(p, \overline{v}_j) = T_{pv}(p, \overline{v}_k) = 0, \\ \boxminus, & \text{otherwise}. \end{cases} \tag{14}$$

Here, $\emptyset$ means $p \cap t = \emptyset$, $\boxdot$ means that $p \cap t$ is one point, $\boxminus$ means that $p \cap t$ is a line, while $\blacksquare$ means $t \subset p$. Thus, when $T_{pt}(p, t) = \boxdot$ or $\boxminus$, we know that $(t, p) \in (\overline{\mathcal{T}}, \mathcal{P})$.

Triples of $(\overline{\mathcal{T}}, \overline{\mathcal{T}}, \overline{\mathcal{T}})$. We do not need to construct these triples explicitly because they are just the input vertices! For each vertex, any three different adjacent triangles contribute a triple in $(\overline{\mathcal{T}}, \overline{\mathcal{T}}, \overline{\mathcal{T}})$.

Triples of $(\mathcal{P}, \mathcal{P}, \mathcal{P})$. If $T_{pp}(p_i, p_j) = T_{pp}(p_j, p_k) = T_{pp}(p_k, p_i) = \boxminus$ and $\text{sign}(\mathbf{n}_i \times \mathbf{n}_j \cdot \mathbf{n}_k) \neq 0$, then we can directly deduce that $(p_i, p_j, p_k) \in (\mathcal{P}, \mathcal{P}, \mathcal{P})$.

Triples of $(\overline{\mathcal{T}}, \overline{\mathcal{T}}, \mathcal{P})$. Before constructing such triples, we introduce a function T_{pv} to check how a cut plane $p \in \mathcal{P}$ intersects with an input edge $e(\overline{v}_i, \overline{v}_j) \in \overline{\mathcal{E}}$ shared by two boundary triangles and

with vertices $\overline{v}_i, \overline{v}_j$:

$$T_{pe}(p, e(\overline{v}_i, \overline{v}_j)) = \begin{cases} \varnothing, & T_{pv}(p, \overline{v}_i) = T_{pv}(p, \overline{v}_j) \neq 0, \\ \boxminus, & T_{pv}(p, \overline{v}_i) = T_{pv}(p, \overline{v}_j) = 0, \\ \boxdot, & \text{otherwise.} \end{cases} \tag{15}$$

Thus, $\forall t_a, t_b \in \overline{\mathcal{T}}$ such that $t_a \cap t_b = e(\overline{v}_i, \overline{v}_j) \in \overline{\mathcal{E}}$, then for any plane $p \in \mathcal{P}$, if $T_{pe}(p, e(\overline{v}_i, \overline{v}_j)) = \boxdot$, then we know that $(t_a, t_b, p) \in (\overline{\mathcal{T}}, \overline{\mathcal{T}}, \mathcal{P})$.

Triples of $(\mathcal{P}, \mathcal{P}, \overline{\mathcal{T}})$. Given two planes p_i and p_j that satisfy $T_{pp}(p_i, p_j) = \boxminus$, a boundary triangle t contributes a triple of $(\mathcal{P}, \mathcal{P}, \overline{\mathcal{T}})$ in the following three cases:

- $(T_{pt}(p_i, t) = \boxdot, T_{pv}(p_j, p_i \cap t) = 0)$: the plane p_i intersects t just on a corner vertex v, and v is on plane p_j;
- $(T_{pt}(p_j, t) = \boxdot, T_{pv}(p_i, p_j \cap t) = 0)$: Similar to the above case;
- $(T_{pt}(p_i, t) = T_{pt}(p_j, t) = \boxminus,\ p_i \cap p_j \cap t \neq \varnothing, p_i \cap t \neq p_j \cap t)$: the two intersection lines from (p_i, t) and (p_j, t) just intersect at a single point in t.

In the above equations, we used the fact that if $T_{pt}(p_i, t) = \boxdot$, then $p_i \cap t$ must be a vertex of the input mesh. Given the values of T_{pv}, T_{pt}, the first two cases are easy. The last case involves a bit more computation. Giving planes $p_i(\mathbf{n}_i, d_i)$ and $p_j(\mathbf{n}_j, d_j)$ and a triangle $t(\overline{v}_a, \overline{v}_b, \overline{v}_c)$, every point in t can be parameterized by λ_1, λ_2 as $x = (\overline{v}_b - \overline{v}_a, \overline{v}_c - \overline{v}_a)(\lambda_1, \lambda_2)^T + \overline{v}_a$, where $0 \leq \lambda_i,\ \lambda_1 + \lambda_2 \leq 1$. If a point $x(\lambda_1, \lambda_2)$ is on both planes $\mathbf{n}_i^T x + d_i = 0,\ \mathbf{n}_j^T x + d_j = 0$, we have

$$M(\lambda_1, \lambda_2)^T = b,$$

where $M = (\mathbf{n}_i, \mathbf{n}_j)^T(\overline{v}_b - \overline{v}_a, \overline{v}_c - \overline{v}_a)$, $b = -(d_i + \mathbf{n}_i^T \overline{v}_a, d_j + \mathbf{n}_j^T \overline{v}_a)^T$. When $|M| = 0$, there exists more than one intersection point or no intersection. Otherwise, we check whether the point is in the triangle. In short, (p_i, p_j, t) contributes such a triple if and only if:

$$|M| \neq 0, \text{ and } \lambda_1, \lambda_2, \lambda_1 + \lambda_2 \in [0, 1].$$

Exact arithmetic is used to solve this 2×2 linear system.

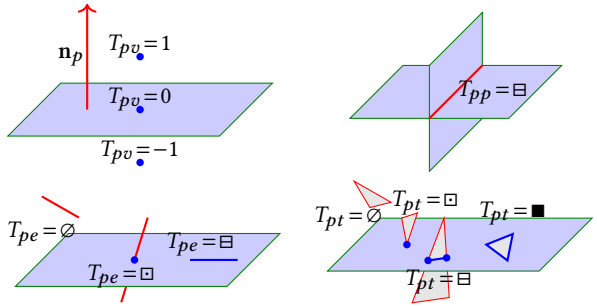


Fig. 31. Intersection relations between input data.

C SWITCH FUNCTION

Below, we introduce how to calculate $s(v)$ on the directed intersection line $\vec{\ell}$ ($\ell = p_i \cap p_j$) for the three cases in Fig. 32. After merging duplicated triples, we easily know whether one cut-vertex is on the triangle, edge and vertex of the input mesh, which will be helpful to compute $s(v)$.

Passing triangle. When $v \in t \in \mathcal{T}, v \notin \partial t$, define the normal of t as $\mathbf{n}_t$, if $\text{sign}(\mathbf{n}_t^T \vec{\ell}) = 0$, $s(v) = 0$, otherwise

$$s(v) = 1. \tag{16}$$

Passing edge. If e is an edge with only two adjacent triangles, and $v \in e = t_i \cap t_j \in \overline{\mathcal{E}}, v \notin \partial e$, given $t_i = (\overline{v}_a, \overline{v}_b, \overline{v}_c)$, $t_j = (\overline{v}_a, \overline{v}_d, \overline{v}_b)$, $\vec{e} = \langle \overline{v}_a, \overline{v}_b \rangle$, define $\mathbf{n}_i = (\overline{v}_b - \overline{v}_a) \times (\overline{v}_c - \overline{v}_a)$, $\mathbf{n}_j = (\overline{v}_d - \overline{v}_a) \times (\overline{v}_b - \overline{v}_a)$, $D_{ij} = \text{sign}(\mathbf{n}_i^T \vec{\ell})\text{sign}(\mathbf{n}_j^T \vec{\ell})$, $r = \text{sign}((\overline{v}_b - \overline{v}_a)^T(\mathbf{n}_i \times \mathbf{n}_j))$. If $e \subset \ell$, $s(v) = 0$; otherwise, we formulate $s(v)$ as below

$$s(v, t_i, t_j) = \begin{cases} 1 & \text{if } D_{ij} > 0 \vee (D_{ij} = 0, r < 0), \\ 0 & \text{if } D_{ij} < 0 \vee (D_{ij} = 0, r \geq 0). \end{cases} \tag{17}$$

When e is a non-manifold edge on the boundary, we need to check all its adjacent triangles. We can classify the adjacent triangles into several subsets, each subset has two triangles t_i, t_j, which are adjacent within the domain, i.e. the region between t_i and t_j is inside Ω. We check each subset with this relation, then get the sum of them to deduce:

$$s(v) = \left(\sum_{\{t_i, t_j\} \subset \mathcal{N}_t(e)} s(v, t_i, t_j) \right) \bmod 2. \tag{18}$$

Here, $\{t_i, t_j\}$ are adjacent in the interior of Ω.

Passing vertex. When $v \in \overline{\mathcal{V}}$, we can work directly in 2D instead. We choose a plane p_i as the 2D domain, and the adjacent region of v in p_i is split by adjacent cut-edges from $(\overline{\mathcal{T}}, \mathcal{P})$. The adjacent cut-edges $\mathcal{E}_v$ are obtained from the intersection lines between adjacent triangles and p_i. We set the direction of each cut-edge in $\mathcal{E}_v$ starting from v, and define the set of their directions as $\vec{\mathcal{E}}_v$. Then we sort $\vec{\mathcal{E}}_v$ clockwise along $\mathbf{n}_{p_i}$ (cycle sorting) and get $\vec{\mathcal{E}}_v' = \{\vec{e}_1, \vec{e}_1, ...\}$. Then we check whether $\vec{\ell}$ and $-\vec{\ell}$ pass inside the region between $\vec{e}_i$ and $\vec{e}_{i+1}$, each fan region belongs to $\Omega^-, \partial\Omega$ or Ω^+ which is determined by adjacent triangles (in fact, each edge is related to tuple (p_i, t)). If $\vec{\ell}$ passes one fan region in Ω^- and $-\vec{\ell}$ not, or $-\vec{\ell}$ passes one fan region in Ω^- and $\vec{\ell}$ does not, then $s(v) = 1$; otherwise, $s(v) = 0$.

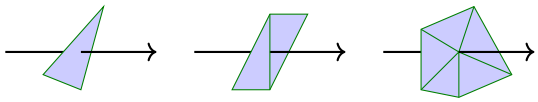


Fig. 32. Intersection between triangles and line.