Robust Computation of Implicit Surface Networks for Piecewise Linear Functions

XINGYI DU, Washington University in St. Louis, USA
QINGNAN ZHOU, Adobe Research, USA
NATHAN CARR, Adobe Research, USA
TAO JU, Washington University in St. Louis, USA

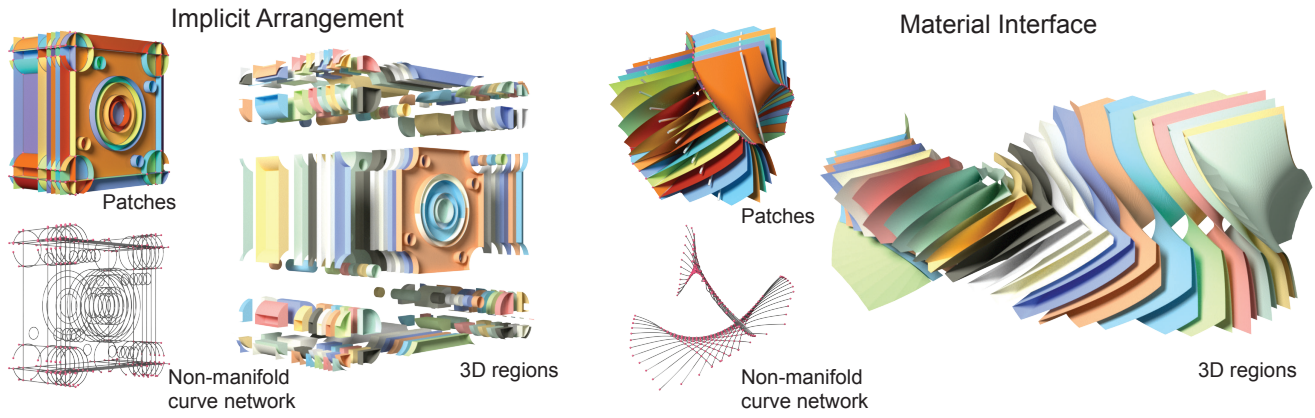


Fig. 1. Implicit surface networks, such as implicit arrangement (left, obtained for primitive geometry defining a CAD object) and material interfaces (right, the Voronoi diagram of rotating 3D lines), produced by our robust algorithms. Each example is visualized by its surface patches, non-manifold curve network as well as the 3D regions partitioned by the surface network.

Implicit surface networks, such as arrangements of implicit surfaces and materials interfaces, are used for modeling piecewise smooth or partitioned shapes. However, accurate and numerically robust algorithms for discretizing either structure on a grid are still lacking. We present a unified approach for computing both types of surface networks for piecewise linear functions defined on a tetrahedral grid. Both algorithms are guaranteed to produce a correct combinatorial structure for any number of functions. Our main contribution is an exact and efficient method for partitioning a tetrahedron using the level sets of linear functions defined by barycentric interpolation. To further improve performance, we designed look-up tables to speed up processing of tetrahedra involving few functions and introduced an efficient algorithm for identifying nested 3D regions.

CCS Concepts: • **Computing methodologies** → **Mesh geometry models**.

Additional Key Words and Phrases: Implicit surfaces, arrangement, material interfaces, predicates, look-up tables, nesting

Authors' addresses: Xingyi Du, Washington University in St. Louis, USA, du.xingyi@wustl.edu; Qingnan Zhou, Adobe Research, USA, qzhou@adobe.com; Nathan Carr, Adobe Research, USA, ncarr@adobe.com; Tao Ju, Washington University in St. Louis, USA, taoju@wustl.edu.

0730-0301/2022/7-ART41
https://doi.org/10.1145/3528223.3530176

ACM Reference Format:
Xingyi Du, Qingnan Zhou, Nathan Carr, and Tao Ju. 2022. Robust Computation of Implicit Surface Networks for Piecewise Linear Functions. *ACM Trans. Graph.* 41, 4, Article 41 (July 2022), 16 pages. https://doi.org/10.1145/3528223.3530176

1 INTRODUCTION

Implicit shape representations are common in computer graphics and geometric processing. While the level set of a single implicit function is routinely used for representing a solid shape with smooth boundary, piecewise smooth, and even non-manifold shapes can be represented using *multiple* implicit functions. These representations typically define a partitioning of the domain into multi-labelled regions by a network of smooth surface patches meeting at non-manifold curves. Consider n functions $\{f_1, \ldots, f_n\}$ in $\mathbb{R}^d$, two commonly used multi-function implicit representations are:

- *Implicit Arrangement* (IA): The surface network is formed by intersecting the zero-level sets of all functions. Equivalently, IA partitions space into labelled regions, where the label at each point x is the set of function signs:

$$L_{IA}(x) = \{sign(f_1), \ldots, sign(f_n)\} \tag{1}$$

  IA is most often used for constructing piecewise smooth shapes from multiple smooth shapes (e.g., CSG [Requicha and Voelcker 1977] and BSH [Du et al. 2021]). It can also be used for modeling geological structures [Bagley et al. 2016; Guo et al. 2021] and 3D curves [Burns et al. 2005; Edelsbrunner and

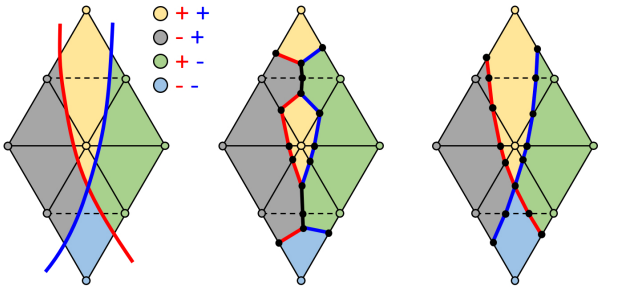


Fig. 2. The Implicit Arrangement (IA) of two functions (a), showing the two zero-level sets (red and blue) and the partitioned regions (colored by sign configurations), and its discretization by either line segments that separate vertices with different labels within each triangle (b) or the IA of the piecewise-linearly interpolated functions (c). Note that (b) has a redundant partition (yellow) not present in (a).

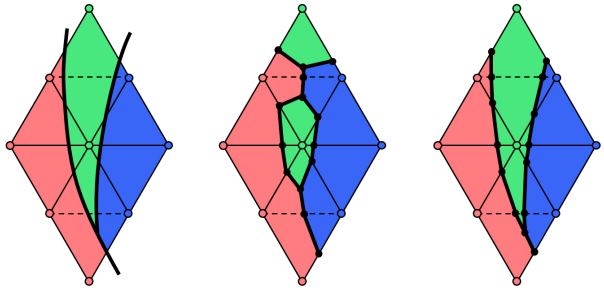


Fig. 3. The Material Interface (MI) of three functions (a), showing the curve network (black) and partitioned regions (colored), and its discretization by either line segments that separate vertices with different labels within each triangle (b) or the MI of the piecewise-linearly interpolated functions (c). Note the redundant partition (green) in (b).

Harer 2002; Ljung and Ynnerman 2003; Thirion and Gourdon 1996].

- *Material Interface* (MI): The labelling at each point x is the set of functions with the maximal values:

$$L_{MI}(x) = \{i | f_i(x) \geq f_j(x), \forall j \neq i\} \quad (2)$$

The surface network consists of points where multiple functions are maximal (i.e., $|L_{MI}(x)| > 1$). MI is a standard representation for partitioned domains, such as anatomical structures [Bertram et al. 2005; Zhang et al. 2008], composite materials [Dillard et al. 2007; Shammaa et al. 2010], bubbles [Zheng et al. 2009] and multi-phase fluids [Kim 2010; Losasso et al. 2006].

To discretize a surface network, the implicit functions are usually sampled on a grid (e.g., cubical or tetrahedral). Most discretization methods generate polygons that separate grid points with different *labels*. While simple to implement, these label-separating methods cannot capture non-trivial interactions between the surface network and the grid, such as multiple intersections on a grid edge (e.g., dotted edges in Figure 2 (a) and 3 (a)). As a result, these methods often produce jagged geometry and incorrect combinatorial structures (e.g., Figure 2 (b) and 3 (b)). To alleviate these artifacts, one may first define continuous functions that interpolate the *values* at the grid points and then compute the surface network of the interpolated functions. The simplest example of this approach is performing linear interpolation in each grid cell on a simplicial (e.g., tetrahedral) grid. The surface network defined by these piecewise linear functions, which consists of planar pieces, often better approximates the underlying surface network than label-separation methods (e.g., Figure 2 (c) and 3 (c)).

However, computing the IA or MI of piecewise linear functions on a tetrahedral grid can be numerically challenging. The computation within each tetrahedron involves intersecting multiple planes, each being the zero-level set of an input function (in IA) or of the difference between two functions (in MI). Existing methods [Bagley et al. 2016; Bonnell et al. 2003; Saye and Sethian 2012] perform these operations on machine-precision numbers, and hence are prone to produce incorrect combinatorial structures, or even algorithm failures, when the level sets are in near-degenerate configurations (e.g., almost co-planar, or multiple level sets nearly intersecting at a common line or point). While the computations can be made exact using rational number representations of coordinates and exact predicates [Fabri et al. 1996], doing so typically incurs a substantial overhead in performance.

We propose a unified, robust and efficient framework for computing both IA and MI on a tetrahedral grid. Our main contribution are algorithms for partitioning a tetrahedron into convex regions by incremental plane intersection (Section 4). The algorithms are made exact and scalable by (1) representing each vertex implicitly using the input functions instead of explicit coordinates and (2) designing a simple predicate for barycentrically interpolated linear functions, which we call the *barycentric predicate* (Section 5). The algorithms are guaranteed to produce the correct combinatorial structure of the IA or MI as defined by the input (floating-point) values at the grid vertices. As secondary contributions, we proposed two mechanisms to improve efficiency on a large grid. First, we pre-computed look-up tables to speed up processing of tetrahedra in which the network is defined by a small number of functions (Section 6). Second, we propose an efficient and numerically robust algorithm that identifies nested 3D regions partitioned by the surface network (Section 7).

Our algorithms are evaluated on both synthetic functions and challenging realistic inputs, such as those shown in Figure 1 (more in Section 8). Our algorithms successfully produced surface networks for all test examples. As a proxy of correctness, we verified that the combinatorial structure of each output is consistent across different orderings of the same input functions. In contrast, a non-robust implementation of the same algorithms frequently fails or generates inconsistent combinatorial structures, which highlights robustness issues in existing methods that are implemented on machine-precision numbers. Performance-wise, our incremental algorithms for partitioning a tetrahedron are two orders of magnitude faster than state-of-the-art exact methods using rational representations [Fabri et al. 1996]. Our algorithms are around 3-4 times slower than label-separating methods, albeit with far fewer artifacts in the

outputs (e.g., Figures 2 and 3). Furthermore, our IA algorithm is much faster than computing mesh-based arrangement [Cherchi et al. 2020; Zhou et al. 2016] on individually discretized level sets. Code and data for this paper are available at https://duxingyi-charles.github.io/publication/robust-computation-of-implicit-surface-networks-for-piecewise-linear-functions/.

2 RELATED WORKS

We briefly review existing works on discretizing implicit surface networks and geometric predicates. We refer readers to [De Araújo et al. 2015] for a survey of polygonization methods of implicit surfaces.

2.1 Discretizing material interfaces

The vast majority of discretization methods for MI aim at separating grid vertices with different labels. Their inputs are typically the functions values or labels (L_{MI} of Equation 2) associated with the vertices of a spatial grid. Some methods determine the combinatorial structure within a grid cell by the labels at the cell vertices, using either heuristic case enumeration [Dillard et al. 2007; Nielson and Franke 1997] or pre-computed tables [Hege et al. 1997; Shammaa et al. 2008; Wu and Sullivan Jr 2003; Yamazaki et al. 2002] that are multi-label extensions of the original Marching Cubes look-up table [Lorensen and Cline 1987]. Other methods compute the surface network on the dual grid [Bertram et al. 2005; Feng et al. 2010; Reitinger et al. 2005; Shammaa et al. 2010], and they can be extended to adaptive grids containing irregular cells [Ju et al. 2002; Zhang et al. 2007; Zhang and Qian 2012]. These methods, however, have inherent difficulties in approximating parts of the MI that intersect a grid element in a non-trivial way, such as multiple intersection points on a grid edge (see Figure 3 (b)) or multiple components of intersection curves on a grid face.

Only a few methods are capable of computing MI for piecewise linear functions. Bloomenthal and Ferguson [1995] trace multiple intersection points on grid edges, but their method cannot handle complex intersections on grid faces (e.g., multiple curve components) or in grid cells (e.g., multiple surface components). Bonnell et al. [2003] compute the MI within a tetrahedron by first mapping the n function values at each vertex into an n-dimensional "material space" and computing the intersection of the mapped tetrahedron with a Voronoi diagram in that space. Saye and colleagues [Saye 2015; Saye and Sethian 2012] construct the MI within a tetrahedron by first computing the polygonal zero-level set of $f_i - f_j$ for each pair of functions f_i, f_j, which is subsequently clipped by the remaining functions f_k ($k \neq i, j$), and then snapping the clipped polygons together. However, these methods all operate on geometry represented by machine-precision coordinates, and hence are prone to numerical errors in near-degenerate configurations. Besides possibly creating incorrect network structures, or structures that are sensitive to the orderings of functions, numerical error might even cause the algorithm to fail. For example, Saye's method [Saye 2015; Saye and Sethian 2012] assumes that clipping a convex polygon by a line results in a convex polygon, which is not always true when working with machine-precision numbers.

A common drawback of grid-based methods, ours included, is that the result may include many poorly shaped triangles (e.g., slivers). Heuristics have been proposed to improve the triangle quality in a post-process while maintaining the combinatorial structure of the surface network [Dillard et al. 2007; Saye 2015], and they can be applied to any polygonalized MI. Alternatively, a class of methods create high-quality triangles directly from the input point samples [Boltcheva et al. 2009; Bronson et al. 2014; Dey et al. 2012; Meyer et al. 2008; Pons et al. 2007]. These algorithms often start by extracting the network of non-manifold curves on the grid, and therefore they could benefit from grid-based algorithms with improved accuracy and robustness.

2.2 Discretizing implicit arrangements

Arrangements of discrete geometry have been extensively studied in computational geometry [Agarwal and Sharir 2000]. For smooth shapes defined by implicit functions, computing geometrically and topologically correct arrangements remains a challenging task, and methods with theoretical guarantees are only known for 2D functions [Alberti et al. 2008; Berberich et al. 2012; Lien et al. 2014] and low-degree 3D polynomials [Dupont et al. 2007; Mourrain et al. 2005; Schömer and Wolpert 2006].

Several algorithms discretize implicit 3D curves, defined as the intersection of the level sets of two functions, on a grid by creating line segments within grid cells [Burns et al. 2005; Ljung and Ynnerman 2003; Thirion and Gourdon 1996]. However, they are typically limited to processing two functions and their outputs do not include surfaces. One could discretize IA by adapting any of the aforementioned label-separating methods for MI, simply by replacing the material label L_{MI} at each grid point with the signs L_{IA} (Equation 1). However, such adaptation would inherit the same difficulties in reproducing fine features of IA, as illustrated in Figure 2 (b), when IA has multiple intersections with a grid element (e.g., the highlighted edges).

We are aware of only a few methods that can compute the IA for piecewise linear functions on a tetrahedral grid [Bagley et al. 2016; Guo et al. 2021; Kim et al. 2000]. All of them adopt a similar grid-refinement approach: the functions are processed sequentially, and for each function, its level set is used to split existing tetrahedra into smaller ones. As the intersection operations are also performed on machine-precision coordinates, like discretization methods for MI for piecewise linear functions, these methods may fail to produce accurate or consistent combinatorial structures of IA due to numerical imprecision.

An alternative, robust way to compute IA for piecewise linear functions is to first extract the level set of each function individually (e.g., using Marching Tetrahedra [Doi and Koide 1991]) as a mesh and then compute the arrangement of these meshes using robust methods like [Cherchi et al. 2020; Zhou et al. 2016]. In our experiments (see Section 8), we found that this two-step approach is much less efficient than computing intersections directly on a grid, because a significant overhead in mesh arrangement is building a spatial query structure (e.g., K-d tree or BSP tree) for locating triangles that intersect.

2.3 Exact predicates and representations

Geometric predicates are key components in robust geometry processing algorithms. Shewchuk [1997] uses fast and adaptive extended floating point operations to robustly conduct orientation and in-circle tests for Delaunay triangulation. CGAL library [Fabri et al. 1996] provides a similar set of predicates using their exact computation kernel, which relies on rational representation of geometric quantities. Lévy [2016] proposed a general purpose Predicate Construction Kit (PCK) for designing arbitrary "exotic" predicates. PCK is available as part of the open source library Geogram [Lévy and Filbois 2015], which contains the current state-of-the-art predicate implementations.

A geometry processing algorithm often needs to construct intermediate quantities, such as the intersection point between a line and a plane that is needed for subsequent intersection tests. To prevent lost of accuracy, all computations can be performed on rational representations, but at a significant computational cost. To circumvent this problem, Sugihara et al. [1989] proposed to use planes as the primary input and to indirectly represent points as intersections of planes. They showed that common predicates such as orientation tests can be expressed efficiently using plane-based representations. This idea proves to be very useful for CSG operations [Bernstein and Fussell 2009; Campen and Kobbelt 2010; Nehring-Wirxel et al. 2021] where all intermediate constructions can be represented exactly using input planes. Attene [2020] proposed indirect predicates which can take both direct or indirect (i.e. using plane-based representation) points as input. Combined with a predicate generation program similar to PCK, the construction of indirect points becomes part of the predicate computation and thus leads to robust computation for a variety of applications [Cherchi et al. 2020; Diazzi and Attene 2021].

As our algorithms also involve intermediate constructions, we follow the idea of plane-based representations and tailor it further to IA and MI. We also develop a new orientation predicate for implicit geometry defined by barycentric interpolation.

3 METHOD OVERVIEW

Our algorithms take as input a tetrahedral mesh M and a vector $\{f_{v,1}, ..., f_{v,n}\}$ at each vertex v, so that $f_{v,i}$ is the value of implicit function $\hat{f}_i : \mathbb{R}^3 \rightarrow \mathbb{R}$ at v. Let f_i be the approximation of $\hat{f}_i$ by continuous, piecewise linear interpolation of vertex values over M. Our algorithms produce either the IA or MI of $\{f_1, \ldots, f_n\}$ within M.

We adopt the same "marching" approach as existing algorithms [Bloomenthal and Ferguson 1995; Bonnell et al. 2003; Saye and Sethian 2012] and process each tetrahedron t in turn. For efficiency, we consider only those functions that contribute to the surface network within t, which we call the *active functions*. In practice, the number of active functions is usually far fewer than n. The criterion of activeness depends on the type of surface network. For IA, a function f_i is active in t if and only if its zero-level set intersects t. This can be easily checked using the signs of $f_{v,i}$ at the four vertices of t - they cannot be all positive or all negative. For MI, it is not easy to determine the active status of f_i without computing the surface network. However, a necessary criterion for f_i being active is that there is no other function f_j ($j \neq i$) that "dominates" f_i, meaning $f_{v,j} > f_{v,i}$ at all vertices v of t. Although some inactive functions might be included, we adopt this criterion due to its simplicity.

Our algorithms proceed in two stages. In the first stage, they extract the IA or MI of the active functions within each tetrahedron using an incremental construction, which is described in Section 4. Section 5 then describes our core contributions, the vertex encoding and predicates, which ensure the robustness and efficiency of the construction. Finally, Section 6 describes the use of look-up tables to speed up the computation for tetrahedra with a small number of active functions.

In the second stage, described in Section 7, the polygonal networks within individual tetrahedra are combined to form the complete surface network and obtain the partitioning of space into 3D regions. Since some regions may be bounded by multiple disconnected surfaces (i.e., an exterior boundary with one or more interior boundaries), we introduce a topological ray-shooting approach that can efficiently and robustly detect all surfaces bounding a 3D region.

4 INCREMENTAL CONSTRUCTION

Consider a tetrahedron t and let $\{f_1, \ldots, f_k\}$ be the set of active, linear functions within t. Our algorithms are motivated by the fact that both IA and MI induce a *convex decomposition* of t. The convexity trivially holds for IA, which partitions t by the planar zero-level sets of all functions. To see why MI is convex, consider the hyperplane in 4-dimensions defined by each function f_i as $w = f_i(\{x, y, z\})$ where x, y, z, w are the 4 coordinate components. MI is the 3D projection of the convex *upper envelope* of these hyperplanes. Figure 4 shows examples of IA and MI of linear functions in a 2D triangle, as well as the corresponding upper envelope in 3D for each MI.

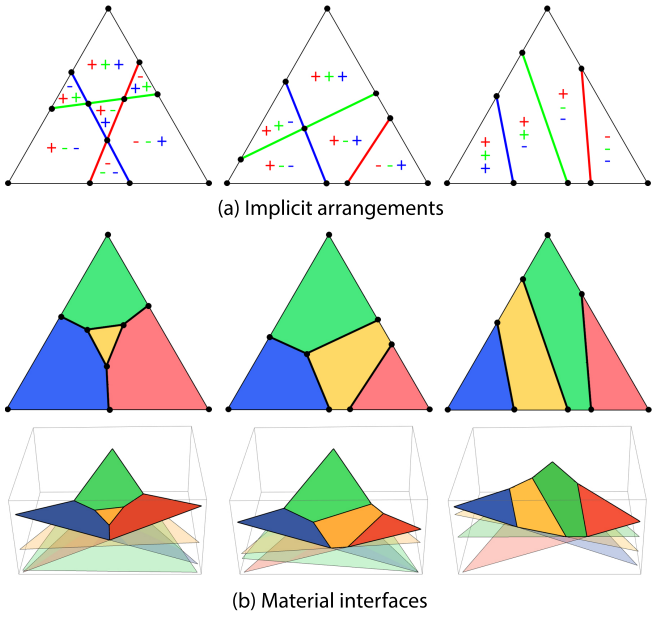


Fig. 4. Examples of IA of 3 functions (a) and MI of 4 functions (b) in a triangle. Each MI is shown below as the upper envelope of planes in 3D.

We compute the IA or MI within t incrementally by processing one function at a time. We represent the convex decomposition induced by the surface network as a cell complex C, whose 0-, 1-, 2-, and 3-dimensional elements are called *vertices*, *edges*, *faces*, and *cells*. Initially C is the tetrahedron t itself. After processing f_i ($i = 1, \ldots, k$ for IA and $2, \ldots, k$ for MI), C is updated to be the decomposition of the first i functions, $\{f_1, \ldots, f_i\}$. We next detail the updating process separately for IA and MI.

4.1 Updating IA

Updating IA amounts to identifying the intersections between the elements of the current cell complex, C, and the planar zero-level set of the next function, f_i. Since all elements of C are convex, and assuming robust computations (see Section 5), the intersection between the level set and any element of C is either empty or a single connected component. This leads to a simple, bottom-up splitting algorithm.

First, the signs of f_i are computed at each vertex of C. We assume that each sign is either + or - (the 0 case is discussed below). Next, for each edge of C whose two ends exhibit different signs, a new *cut-vertex* is created where the edge intersects with the level set, and the edge is split into two. Then, for each face of C whose boundary vertices have different signs, there must be exactly two cut vertices on the boundary edges. These cut vertices are connected to create a new *cut-edge* that splits the face into two. Finally, for each cell of C whose boundary vertices have different signs, the cut edges on the boundary faces must form a single closed loop. A new *cut-face* is created from the loop and splits the cell into two. The algorithm is illustrated in Figure 5 where C consists of a single cell.

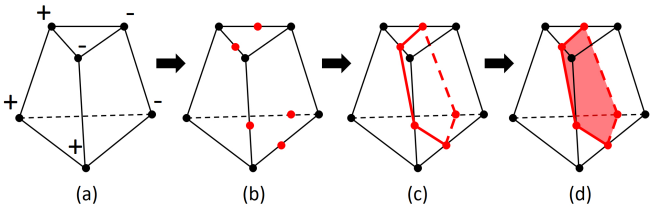


Fig. 5. Splitting a (convex) cell by a plane: computing signs (a), splitting edges (b), splitting faces (c), and splitting the cell (d). The cut-vertices, cut-edges, and cut-faces are highlighted in red.

In the degenerate situation that some vertex signs are 0, meaning that they lie exactly on the level set of f_i, we make the observation that an element e should be split only if the level set of f_i intersects the interior of e and does not completely contain e. We therefore introduce the following changes to the algorithm above to accommodate degeneracy. First, an element (edge, face or cell) is split if it has at least one positive vertex and one negative vertex. Second, a cut edge connects either cut vertices or 0-signed vertices of C. Third, a cut face is formed by a loop of either cut edges or edges of C connecting two 0-signed vertices.

4.2 Updating MI

Updating MI is similar to IA but slightly more involved. To motivate our algorithm, recall that the MI is the projection of the upper envelope of hyperplanes in one higher dimension. Given a new hyperplane, the upper envelope can be updated by (1) computing the intersections between the current envelope elements and the hyperplane, and (2) removing elements that are either under the original envelope or the new hyperplane. Figure 6 bottom illustrates this process for updating the upper envelope of two 3D hyperplanes after adding a third hyperplane.

The two steps mentioned above can be implemented without going to the 4-th dimension. We assume that each vertex v of C is associated with the material labels $L_{MI}(v)$ (Equation 2) from the first $i-1$ functions (the labels are updated together with C). In the first, *splitting step*, each element e of C is intersected with the planar zero-level set of $f_j - f_i$ where $j \in [1, i-1]$ is a common label among all vertices of e. Note that the choice of f_j changes with the element e. The splitting step can be implemented using the same bottom-up algorithm described above for IA (Figure 5), with one crucial modification: the sign at a vertex v of C is computed for $f_j - f_i$ (instead of f_i), where j is any label at v. In the second, *merging step*, all elements of C after splitting that contain at least one negatively signed vertex are merged along their common boundaries. The two steps are illustrated in Figure 6 top.

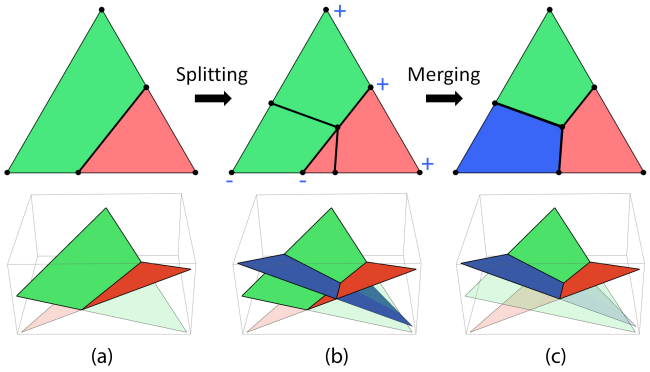


Fig. 6. Top: updating the MI of two functions (a) by adding a third function via splitting (b) and merging (c). The signs at the vertices are indicated in (b). Bottom: the corresponding actions on the hyperplanes in 3D.

4.3 Complexity analysis

The incremental construction of IA takes $O(k^4)$ time for k functions. This is because processing each f_i using the bottom-up splitting algorithm needs to visit all $O(i^3)$ elements in the arrangement of $i-1$ planes. With a more sophisticated algorithm, this complexity can be reduced to $O(i^2)$ (and the overall complexity to $O(k^3)$) [Chazelle and Edelsbrunner 1992]. However, since k is typically small for most tetrahedra, the efficiency gain at large k can be negligible in the overall runtime. Processing f_i for MI takes only $O(i^2)$ time, which is the size of the upper envelope of $i-1$ hyperplanes in 4D [Agarwal and Sharir 2000], and hence the incremental construction of MI takes $O(k^3)$ time. In practice, we observed that this runtime is nearly linear (see Section 8).

5 EXACT COMPUTATION

The successful execution of the algorithm above, and the correctness of its result, both hinge on the robust computation of the vertex signs during bottom-up splitting (Figure 5 (a)). Incorrect signs would lead to a wrong set of elements of the cell complex being split, which in turn would lead to a erroneous combinatorial structure of the surface network. Furthermore, the splitting algorithm may fail to proceed, if the signs at the vertices of an element of C require the creation of multiple cut-elements (e.g., multiple cut-edges on a face or multiple cut-faces in a cell).

Exact signing can be achieved by performing all computations on vertex coordinates represented as rational numbers [Fabri et al. 1996]. However, such computations are much more costly than working with floating-point numbers, and the cost only grows with more iterations of the incremental algorithm, since the newly constructed vertices have even more complex rational representations. To maintain efficiency without sacrificing accuracy, we introduce a fixed-length encoding for vertices and an exact predicate that operates on floating point numbers.

5.1 Vertex encoding

We follow the idea in [Sugihara et al. 1989] to represent points *indirectly* by the planes they lie on. Observe that each vertex in the cell complex C is the *unique intersection of 3 planes*. A plane can be the level set of a function (for IA), the level set of the difference between two functions (for MI), or the supporting plane of a tetrahedral face (for a vertex on the tetrahedral boundary). Specifically, a vertex v that lies on a d-dimensional element of the tetrahedron is the result of intersecting $3-d$ supporting planes of tetrahedral faces with either the zero-level sets of d functions (for IA) or the zero-level sets of pairwise differences among $d+1$ functions (for MI). This is illustrated in Figure 7. In degenerate cases, v may lie on more planes, but there are at least 3 planes that intersect uniquely at v (otherwise v would have been part of an edge or face of C).

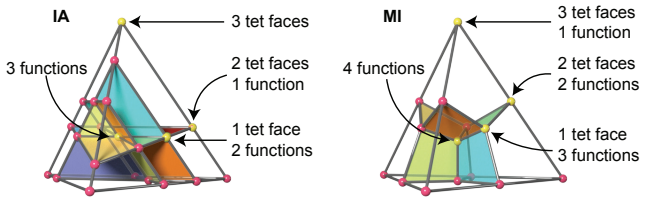


Fig. 7. Vertices in an IA (left) and MI (right) encoded by their defining functions and supporting tetrahedral faces.

We encode each vertex by the indices of the functions and tetrahedra faces associated with the three planes intersecting at the vertex. This encoding is unique for each vertex, and it has a fixed length (3 integers for IA and 4 integers for MI) for all vertices of C regardless of the number of functions.

5.2 Barycentric predicates

We now describe how to check the sign of some function f_i at a vertex using our encoding scheme. Consider a vertex v on a d-dimensional element e of the tetrahedron, for $d \in [0,3]$, and suppose v lies on the zero-level sets of functions $\{g_1,\ldots,g_d\}$. Each g_i is either an input function encoded at v (in IA) or the difference between two encoded functions (in MI). Given a new function g_{d+1}, being either f_i itself (in IA) or the difference between f_i and any function encoded at v, we need to determine the sign of $g_{d+1}(v)$.

We leverage the fact that each function g_i is implicitly defined by the values at the vertices of e, which are floating point numbers given as input. We introduce a simple predicate that involves only those input values. Let $g_{j,i}$ be the value of g_i at the j-th vertex of e, and let $\mathbf{g}_i = \{g_{1,i},\ldots,g_{d+1,i}\}^T$ be the column vector of all values of g_i. The value of function g_i at any point x can be written as

$$g_i(x) = \mathbf{g}_i^T \cdot \mathbf{x} \tag{3}$$

where $\mathbf{x} = \{x_1,\ldots,x_{d+1}\}^T$ is the *barycentric coordinates* of x with respect to the vertices of e, and $\sum_{i=1}^{d+1} x_i = 1$. The barycentric coordinates $\mathbf{v}$ of vertex v, where functions $\{g_1,\ldots,g_d\}$ all evaluate to 0, thus satisfy:

$$A\mathbf{v} = \mathbf{b} \tag{4}$$

where $A = \{\mathbf{g}_1,\ldots,\mathbf{g}_d,\mathbf{1}\}^T$ and $\mathbf{b} = \{0,\ldots,0,1\}^T$. Solving Equation 4 using Cramer's rule and substituting $\mathbf{x} = \mathbf{v}$ into Equation 3 yields:

$$g_{d+1}(v) = \det(A')/\det(A) \tag{5}$$

where $A' = \{\mathbf{g}_1,\ldots,\mathbf{g}_{d+1}\}^T$. Note that $\det(A) \neq 0$ in our algorithm, because our encoding ensures that the zero-level sets of $\{g_1,\ldots,g_d\}$ always have a unique intersection point, implying that the solution to Equation 4 uniquely exists. As a result, $g_{d+1}(v)$ is always well-defined. The sign of $g_{d+1}(v)$ can be determined by the following *barycentric predicate*:

$$\text{sign}(\det(A')) \times \text{sign}(\det(A)) \tag{6}$$

To ensure correctness, we implemented the predicate using an extension of the Predicate Construction Kit (PCK) provided by [Attene 2020]. It employs a tiered evaluation system with three tiers: semi-static filtering, interval arithmetic and exact computation with extended floating point representation. Our implementation takes input function values directly to avoid the accidental introduction of floating point errors due to intermediate value computation. In particular, for MI, we ensure that g_i's, which represent the difference of two input functions, are evaluated inside of the predicates so its associated floating point errors are tracked.

6 LOOK-UP TABLES

The incremental construction algorithm can process any number of active functions in a tetrahedron. In practice, the majority of tetrahedra in a grid contain very few active functions. We found that by far the most common non-empty tetrahedra are ones that contain a single planar polygon, which are defined by 1 (for IA) or 2 (for MI) active functions, followed by tetrahedra that contain several polygons meeting at a non-manifold edge, which are defined by 2 (for IA) or 3 (for MI) active functions.

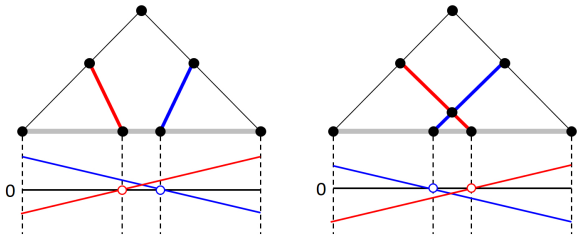
In this section, we describe the design and creation of look-up tables that speed up the processing of these common types of tetrahedra. We only resort to incremental construction during runtime if the tetrahedron contains more than 2 (for IA) or 3 (for MI) functions, or if any vertex sign or edge predicate (see below) is zero.

6.1 IA with 1 function or MI with 2 functions

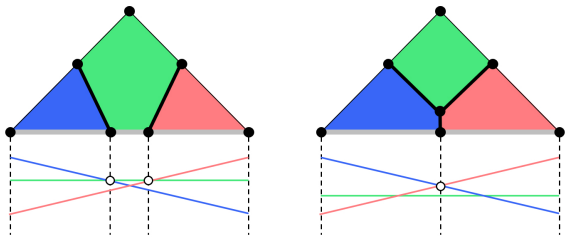
The IA in a tetrahedron containing a single active function can be easily tabulated into a look-up table with $2^4 = 16$ entries, indexed by the sign of the function value at each tetrahedral vertex. A similar table can be made for MI within a tetrahedron containing two active functions, where the sign is taken of the difference between the two function values.

6.2 IA with 2 functions

Now consider the IA in a tetrahedron containing two active functions. The signs of both functions at the tetrahedral vertices are necessary to disambiguate different combinatorial structures of the IA, but they are not sufficient: we also need, on each tetrahedral edge that intersects both level sets, the order of the two intersection points along the edge. We call such an edge *ambiguous* (see Figure 8 (a)). The ambiguity can be resolved by evaluating the barycentric predicate (Equation 6) on that edge.



(a) Ambiguous edge for IA with 2 functions



(b) Ambiguous edge for MI with 3 functions

Fig. 8. Ambiguous edges (gray) for IA (a) and MI (b). An edge is ambiguous if the number or ordering of edge intersections cannot be uniquely determined by the signs (for IA) or ordering (for MI) of functions at the vertices. The functions along each ambiguous edge are plotted underneath.

We could create a single table of $2^{14} = 16,384$ entries indexed by the 8 signs at the vertices (4 signs for each function) and the 6 signs on the edges (one for each predicate). However, looking up this table would require computing the predicates for all 6 edges, regardless of whether they are ambiguous. To avoid unnecessary computations, we use a two-tier table structure. First, we create a *main table* with $2^8 = 256$ entries indexed by the 8 signs at the vertices. Each entry of the table consists of a list of m edges that are ambiguous based on that sign configuration, and a *branch table* with 2^m entries of IA indexed by the m predicate signs on the ambiguous edges. To use the look-up table, we first find the list of ambiguous edges in the main table using the vertex signs, then evaluate the predicate on each ambiguous edge, and finally find the IA in the branch table using the predicate signs.

The look-up tables are created automatically by running the incremental construction algorithm for every combination of vertex signs and every combination of edge predicates on the ambiguous edges determined by the vertex signs. For combinations that lead to ambiguity during splitting, their entries in the tables are left empty, because they will not arise during runtime thanks to our robust implementation.

6.3 MI with 3 functions

Look-up tables for MI with three active functions are created similarly to IA with two functions. We call an edge *ambiguous* if it intersects with the level sets of more than one difference functions (there are three in total), and if the ordering of the intersections along the edge cannot be uniquely determined from the ordering of the functions at the two edge vertices. It can be shown that, for any ambiguous edge, the functions must have opposite orderings at the two vertices (i.e., $f_1 > f_2 > f_3$ at one vertex and $f_1 < f_2 < f_3$ at the other; see Figure 8 (b)). The ambiguity can again be resolved by evaluating the barycentric predicate on that edge.

The structure of the tables are the same as in IA. The main table stores, for each combination of vertex signs, the list of ambiguous edges and a branch table. Each branch table then stores the MI for each combination of predicate signs on the ambiguous edges. The main table consists of $2^{12} = 4096$ entries indexed by the 12 signs at the vertices, 4 for each difference function. The tables are automatically generated, similar to IA.

7 SPACE DECOMPOSITION

After processing all the tetrahedra, polygons on the surface network (IA or MI) within each tetrahedron are combined to form a complete surface network. Using the plane-based encoding (Section 5), duplicate polygon vertices created in adjacent tetrahedra on their common edges or faces can be unambiguously identified and unified. Aided by the cell complexes computed in the tetrahedra, we can then create a topological decomposition of the surface network, including the manifold surface patches, the network of non-manifold curves, and their adjacency (including the ordering of patches meeting at a non-manifold curve), without using geometric coordinates.

To obtain a full decomposition of space, we also need to identify individual 3D regions partitioned by the surface network. Note that a region may be bounded by several disconnected surfaces. This happens either when the region encloses other "nested" regions or when the region's boundary extends to the boundary of the grid M. Correctly identifying regions bounded by multiple boundary components can be numerically challenging, even in 2D, when those components are very close to each other [Bajaj and Dey 1990]. While exact results can be obtained using rational representations [Zhou et al. 2016], this can lead to a significant overhead in performance (see Section 8.3). Leveraging the grid structure and the cell complexes within the tetrahedra, we present a region identification algorithm that is both efficient and numerically robust.

7.1 Topological ray shooting

A straightforward approach for identifying 3D regions is to group all neighboring 3D cells in individual tetrahedra that are not separated by the surface network. While the result is correct, doing so requires traversal of all tetrahedra, including the empty ones. In practice, we observed that this approach quickly becomes the main bottleneck of the algorithm as the grid size increases (see Section 8).

We adopt an alternative approach that is equally robust but more efficient. This approach identifies the surfaces that bound each region using *ray-shooting*. In a nutshell, we shoot one ray from each connected component C of the surface network, so that the ray does not hit C again. If the ray hits another component C' of the surface network, the ray must have travelled through a 3D region bounded by (at least) two separate surfaces, each belonging to either C or C'. After all rays are shot, each group of bounding surfaces that are connected via the rays become the boundary of a 3D region. To avoid computing ray-surface intersections numerically, we use *topological* rays that travel along the edges of the tetrahedral grid M. In this way, intersections between the ray and the surface network can be directly obtained from the cell complexes computed within the tetrahedra. We next describe the algorithm in details.

We define a *shell* as a connected component of the boundary of a region. Each shell can be obtained as a connected set of manifold surface patches oriented towards the same open space. A 2D example of shells is shown in Figure 9 (b) for the curve network in (a). Observe that a region may be bounded by multiple shells due to nesting (e.g., region C) or having open boundaries (e.g., region B). Our algorithm maintains a graph G whose nodes are all shells in the surface network, plus one representing the "outer shell". A described below, each ray connects two nodes of the graph representing shells where the ray starts and ends. After all rays are shot, nodes in each connected component of G are the shells bounding each 3D region (see Figure 9 (d)).

Prior to ray-shooting, we first establish an ordering Φ of all grid vertices. During ray shooting, the ray travels in the "steepest descent" direction according to Φ. That is, after visiting some vertex v, the ray would next visit an adjacent vertex of v with a lower order (if multiple such vertices exist, the one with the lowest order is chosen). To ensure correctness of the algorithm, we seek a Φ such that no two vertices of M have the same order and, importantly, there is only one vertex of M (called the *sink*) that has lower order than all its adjacent vertices. The sink vertex corresponds to the "outer shell" node in the aforementioned graph G. Assuming that M has no inverted or degenerate tetrahedra and its boundary is convex (e.g., a cube), a simple and numerically robust choice of Φ is the lexicographic order of the $\{x, y, z\}$ coordinates of the grid vertices. There is no need to explicitly sort all vertices, because ray-shooting only needs to compare two vertices at a time. The ordering is illustrated in Figure 9 (c).

For each connected component C of the surface network, we select a vertex of C to start the ray so that the ray does not hit C again. To do so, we call a vertex p of the surface network an *edge-point* if it lies on an edge $\{v, u\}$ of M (assuming $\Phi(v) < \Phi(u)$), and we associate p with an *ordering vector* $\{v, u, i\}$ where i is the number of additional edge-points between p and v. Among all edge-points of C, we start

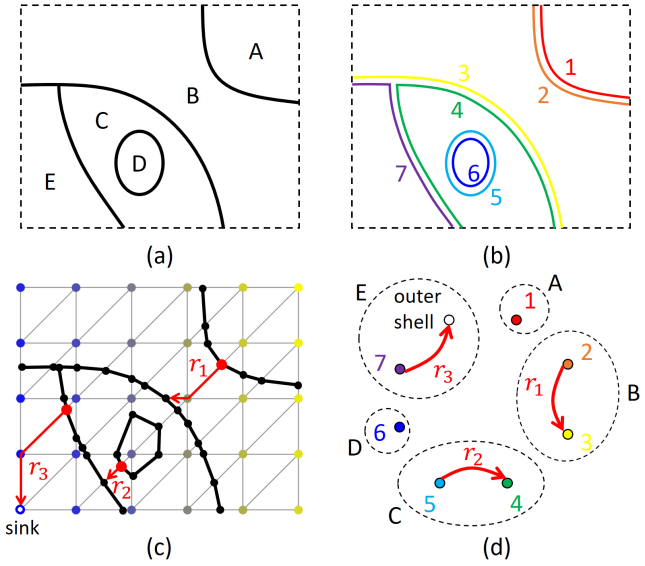


Fig. 9. Topological ray shooting in 2D. (a): A curve network that partitions the plane into five 2D regions (A,B,C,D,E). (b): Shells of the curve network. (c): Grid vertex ordering (as color), the sink vertex, and the topological rays (red arrows, one for each component of the curve network). (d): The graph G in which nodes are shells, edges correspond to rays, and each connected component corresponds to a 2D region.

the ray from the edge-point with the lexicographically smallest ordering vector. We terminate the ray when it either encounters another edge-point or arrives at the sink. In either case, the shells at the two ends of the ray are connected in the graph G. Examples of rays and corresponding graph edges are shown in Figure 9 (c,d).

We prove in Appendix A that the algorithm produces the correct result for any tetrahedral grid M with a simple topology (i.e., no holes or handles) and any implicit surface network that induces a convex decomposition within each tetrahedron (e.g., IA or MI). Compared with the cell-grouping approach, which needs to visit all tetrahedra, ray-shooting only needs to visit the vertices of the surface network and grid edges on the rays, which are typically far fewer in number.

8 RESULTS

We evaluate the robustness and efficiency of the algorithms using both synthetic and real-world data. Our algorithms are implemented in C++. The algorithms have no parameters. Other than the extended PCK [Attene 2020] and the Abseil [Google 2017] library for efficient hash tables, our implementation does not use any external libraries. Our algorithms can work on any type of tetrahedral grids (structured or unstructured). For consistency during evaluation, all tests are performed on tetrahedral grids converted from a uniform cubic grid by dividing each cube into five tetrahedra. All timings were obtained on a MacBook Pro with 2.6GHz 6-Core Intel Core i7 CPU and 16 GB memory.

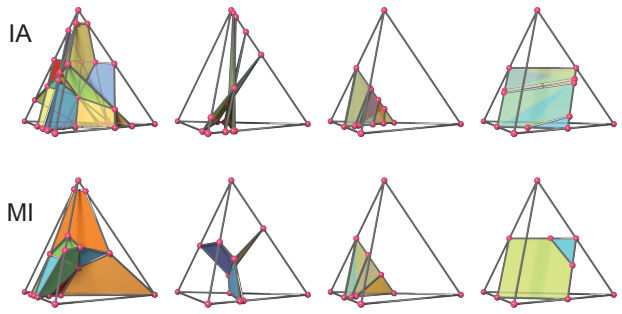


Fig. 10. From left to right: Example cell complexes of IA (top row) and MI (bottom row) computed inside a tetrahedron using our exact algorithm for point-, segment-, tri- and quad-degeneracies in our first test set. Faces, edges, and vertices of the cell complex are shown as colored polygons, gray tubes, and pink balls.

8.1 Robustness

We stress tested our algorithm in various near-degenerate scenarios that are challenging for numerical computations. As a proxy for existing methods that rely on imprecise, floating-point computations [Bagley et al. 2016; Bonnell et al. 2003; Saye and Sethian 2012], we created a variant of our algorithm that replaces the exact implementation of the barycentric predicate (using PCK) by a naive floating-point implementation. Since this version retains our exact, plane-based encoding of vertices, it is already more robust than these previous methods. For each test case, we ran both our algorithm and the floating-point variant twice with different orders of the same functions, and we differentiate between two types of failures. If the algorithm proceeds but outputs a different cell complex structure in each run, we report a *Type I* failure. If the algorithm fails to proceed in either run, due to ambiguity in the splitting algorithm (Section 4.1) caused by incorrect signs, we report a *Type II* failure.

Our first test set consists of functions defined over a tetrahedron whose level sets (for IA) or level sets of their differences (for MI) *almost* intersect at a common point, line segment, triangle, or quadrilateral. We call these scenarios *point-*, *segment-*, *tri-*, and *quad-(near-)degeneracies*. We used 4 functions for IA and 5 functions for MI, and randomly generated 10,000 inputs for each type of near-degeneracy (Figure 10 shows one example for each type). Table 1 summarizes the success rates and failure types for our algorithm ("exact") and the floating-point variant ("float"). Note that our algorithm succeeded 100% of the time, while the floating-point implementation had high failure rates of both types.

Our second test set involves functions defined over a 100^3 grid with slightly shifted level sets (or difference level sets). For IA, each input consists of 4 functions whose zero-level sets are spheres each shifted from a previous one in a fixed direction by a distance of ϵ (the grid has dimension $2 \times 2 \times 2$). For MI, we added one more function that is zero everywhere. We tested with four different ϵ values and, for each value, generated 100 inputs each with a random shifting direction. The results are summarized in Table 2. Observe that our exact algorithm once again achieved 100% success rate, whereas the floating-point variant encountered more failures as ϵ decreases and did not succeed even once for $\epsilon \leq$ 4e-7. Figure 11 shows one example of IA where the floating-point version reports a Type I failure (inconsistent structures). Observe that the floating-point version creates a non-trivial network of non-manifold curves, whereas our exact algorithm correctly finds the (almost coinciding) circles of intersection between the shifted spheres. The resulting IA partitions the space into several extremely thin regions sandwiched between spheres, as shown in the exploded view.

Table 1. Summary for IA and MI with our algorithm ("exact") and a floating-point variant ("float") on 10,000 inputs in each near-degenerate scenario in our first test set.

Degeneracy type	Alg	Success	Failure I	Failure II
[IA] *Point*	exact	100%	-	-
[IA] *Segment*	exact	100%	-	-
[IA] *Tri*	exact	100%	-	-
[IA] *Quad*	exact	100%	-	-
[IA] *Point*	float	68.73%	31.27%	-
[IA] *Segment*	float	3.88%	44.08%	52.04%
[IA] *Tri*	float	27.76%	21.22%	51.02%
[IA] *Quad*	float	13.83%	18.49%	67.68%
[MI] *Point*	exact	100%	-	-
[MI] *Segment*	exact	100%	-	-
[MI] *Tri*	exact	100%	-	-
[MI] *Quad*	exact	100%	-	-
[MI] *Point*	float	41.80%	58.20%	-
[MI] *Segment*	float	12.53%	51.29%	36.18%
[MI] *Tri*	float	67.02%	18.07%	14.91%
[MI] *Quad*	float	52.94%	23.38%	23.68%

Table 2. Result for IA and MI with our algorithm ("exact") and a floating-point variant ("float") on 100 inputs in each ϵ setting in our second test set.

Type	ϵ	Alg	Success	Failure I	Failure II
[IA]	1e-7	exact	100%	-	-
[IA]	4e-7	exact	100%	-	-
[IA]	7e-7	exact	100%	-	-
[IA]	1e-6	exact	100%	-	-
[IA]	1e-7	float	0%	0%	100%
[IA]	4e-7	float	0%	0%	100%
[IA]	7e-7	float	4%	3%	93%
[IA]	1e-6	float	78%	3%	19%
[MI]	1e-7	exact	100%	-	-
[MI]	4e-7	exact	100%	-	-
[MI]	7e-7	exact	100%	-	-
[MI]	1e-6	exact	100%	-	-
[MI]	1e-7	float	0%	0%	100%
[MI]	4e-7	float	0%	0%	100%
[MI]	7e-7	float	6%	23%	71%
[MI]	1e-6	float	76%	12%	12%

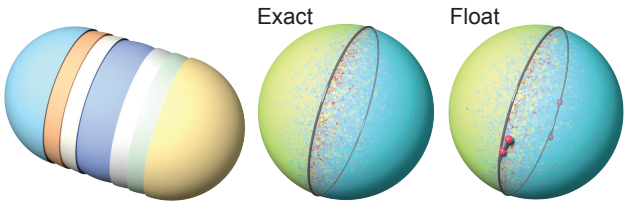


Fig. 11. Right: IA computed by our exact algorithm and a floating-point variant for one input in our shifted-spheres test set. In this and following figures, patches of the surface network are distinguished by colors, and the network of non-manifold curves is visualized as gray tubes connecting at pink balls. The rendering artifacts in this example near the non-manifold curves are due to multiple surfaces being extremely close to each other. Left: an exploded view of the 3D regions partitioned by the IA computed by the exact algorithm.

8.2 Performance of incremental construction

We benchmarked the incremental construction algorithm (Section 4) within a tetrahedron. Figure 12 plots the running time up to a 100 functions for both IA and MI. Consistent with our analysis earlier (Section 4.3), incremental construction of IA shows a polynomial growth in time. Interestingly, running time for MI has a near-linear growth, despite a cubic worst-case complexity. This indicates that the size of MI has a much lower complexity in practice than that of the IA (note that an arrangement of planes always has cubic complexity).

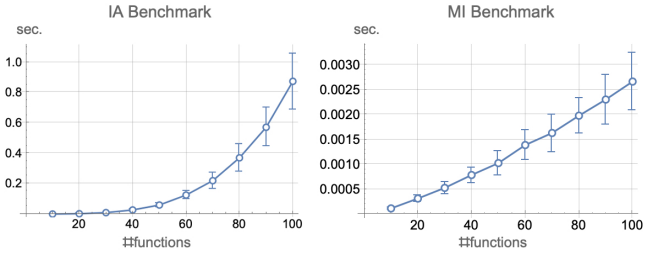


Fig. 12. Running times of incremental construction for IA (left) and MI (right) with increasing number of linear functions within a single tetrahedron.

We further compared our IA construction algorithm with an alternative implementation in CGAL [Fabri and Pion 2009] that uses exact representation of vertex coordinates. Specifically, we partition the tetrahedron using CGAL's "clip" function and adopt the exact_predicate_exact_constructions_kernel, which uses rational numbers to represent coordinates and compute predicates. We ran our algorithm (without look-up table) and CGAL's on 100 instances of 1, 2, or 3 randomly generated functions within a tetrahedron, and the running times are reported in Table 3. Observe that our algorithm is around two orders of magnitude faster than CGAL's. We attribute the significant speed up to our plane-based encoding scheme, which not only has a fixed length but also avoid the need to explicitly compute coordinates, and to our floating-point-based barycentric predicate.

Table 3. Mean running time and standard deviation of incremental construction for IA over 100 instances of N linear functions, using either our algorithm or CGAL's clipping function with exact, rational representation of coordinates.

N	Method	Mean (μs)	Standard Deviation (μs)
1	[CGAL]	666.3	186.1
1	[ours]	6.6	1.1
2	[CGAL]	1754.8	673.6
2	[ours]	13.7	2.5
3	[CGAL]	3302.8	1403.6
3	[ours]	26.3	5.5

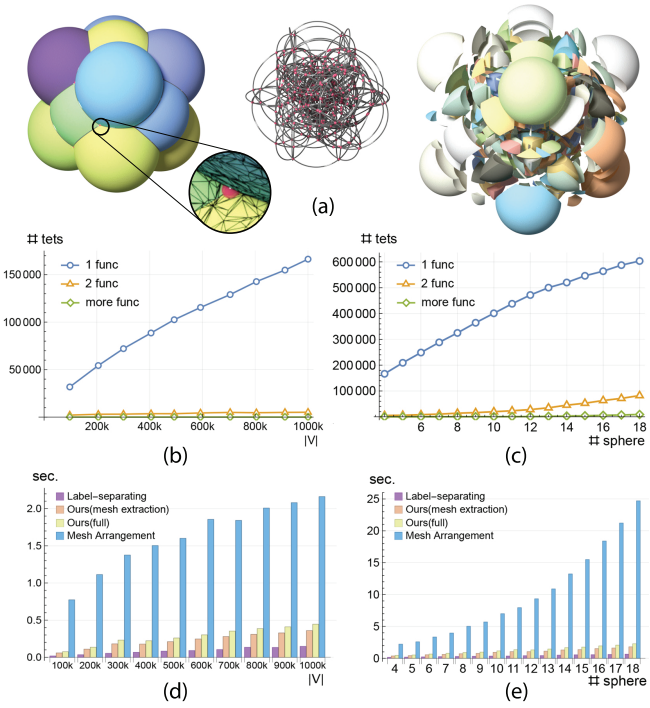


Fig. 13. Performance evaluation for IA. (a): IA computed by our method for 18 spherical functions, showing the patches (with mesh edges in the zoom-in), the non-manifold curve network, and exploded regions. (b,c): Number of tetrahedra containing 1, 2 or more active functions with the increase in either the grid resolution (using 4 functions) or the number of functions (at the highest grid resolution). (d,e): Timing of our algorithm (mesh extraction only and full pipeline), label-separation, and mesh arrangement.

8.3 Performance of complete algorithms

We evaluated the performance of our algorithms on grid data with various resolutions and geometric complexity. For comparison, we implemented a standard label-separating algorithm [Dillard et al. 2007; Nielson and Franke 1997] for both IA and MI, using the definitions of vertex labels in Equations 1 and 2. Since our implementation of label-separation does not produce a full space decomposition into 3D regions, we only report the timing for extracting the polygonal

mesh and compare with the corresponding stage in our algorithm (before space decomposition). For IA, we additionally compared with the two-step approach of first extracting the zero-level set of each function using Marching Tetrahedra (we use an implementation in libigl [Jacobson et al. 2018]) as meshes and then computing mesh arrangement by [Cherchi et al. 2020].

We start with our IA algorithm. Our input consists of functions whose zero-level sets are spheres of the same size and arranged in a regular pattern in a cubic space. We consider two modes of evaluation, either varying the grid resolution while fixing the functions (we use 4), or varying the number of functions (up to 18) while fixing the grid resolution. Figure 13 (a) shows the IA computed by our algorithm for 18 functions in the highest resolution setting (on a 100^3 grid). The statistics of tetrahedra containing 1, 2, or more active functions for each evaluation mode is reported in Figure 13 (b,c). Observe that tetrahedra with only one active function are by far the most common type, although the proportion of tetrahedra with two active functions increases with the total number of functions, due to more frequent intersections between the level sets. Figure 13 (d,e) compare the running time of our algorithm, the label-separating algorithm (mesh extraction only), and the two-step mesh arrangement method in both evaluation modes. Observe that our method (up to mesh extraction) maintains a consistent 1-2 times slow-down relative to label-separation. Mesh arrangement is significantly slower, and it exhibits a non-linear growth as the number of functions increases.

We further evaluate the performance of our ray-shooting algorithm (Section 7.1) in identifying 3D regions bounded by multiple surface components (shells). We consider functions whose zero-level sets are either separate, non-intersecting spheres (Figure 14 left) or concentric, nesting spheres (Figure 14 right). In the former case, the outmost region is bounded by the same number of shells as the number of functions. In the latter case, each 3D region (except for the innermost and outermost ones) is bounded by two shells. Observe that replacing the ray-shooting algorithm by the straightforward cell-grouping approach mentioned at the beginning of Section 7.1 incurs an overhead that is often a few times more than the rest of the entire pipeline. The mesh arrangement approach uses a closest-point-based heuristic for resolving nested 3D regions [Zhou et al. 2016]. Not only this heuristic requires rational representation of coordinates to be robust, its running time grows much faster than our ray-shooting algorithm as the number of bounding shells and (particularly) depth of nesting increases.

To evaluate our MI algorithm, we use functions that are signed distances to spheres of varying sizes and locations. The MI of such functions is the weighted Voronoi diagram of the sphere centers, where the weights are the sphere radii. Such diagrams, also known as the *Apollonius Diagrams*, are useful for analysis of liquids [Voloshin et al. 2002] and proteins [Will 1999], but few methods exist to compute them in 3D [Wang et al. 2020]. Similarly to IA evaluation, we either varied the grid resolution while fixing the same number of functions (4) or varied the number of functions (up to 18) on the highest grid resolution (100^3). Figure 15 shows an example MI with 18 functions, reports the statistics of the tetrahedra, and compares the running times of our algorithm (mesh extraction only and full pipeline) with the label-separating algorithm. Like our IA algorithm,

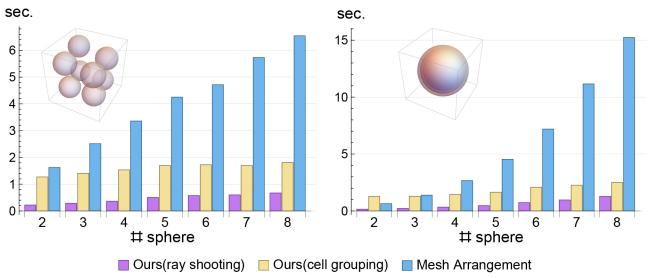


Fig. 14. Timing of our IA algorithm (using either topological ray-shooting or cell-grouping for finding boundaries of 3D regions) and mesh arrangement on increasing number of functions whose level sets are separate (left) or nesting (right) spheres. All tests are performed on a 100^3 grid.

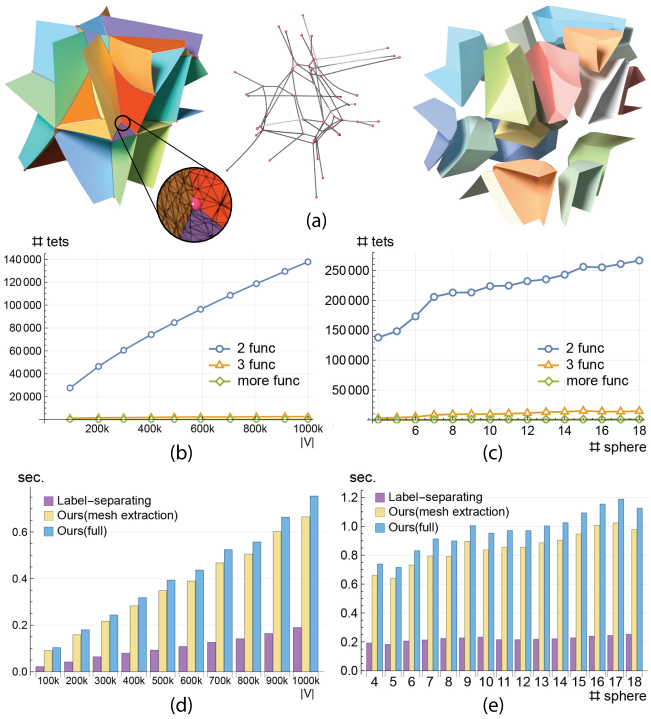


Fig. 15. Performance evaluation for MI. (a): MI computed by our method for 18 functions, which is the Apollonius Diagrams of 18 spheres, showing the patches (with mesh edges in the zoom-in), the non-manifold curve network,and exploded regions. (b,c): Number of tetrahedra containing 2, 3 or more active functions with the increase in either the grid resolution (using 4 functions) or the number of functions (at the highest grid resolution). (d,e): Timing of our algorithm (mesh extraction only and full pipeline) and the label-separating algorithm.

our MI algorithm (up to mesh extraction) maintains a consistent slow-down of 2-3 times compared with label-separation in either evaluation mode.

Finally, we show the break-down of our algorithms' running time into different stages in Figure 16. The information is obtained for computing IA of 18 functions and MI of 18 functions, both on a 100^3

grid. Observe that the use of look-up tables for tetrahedra with 2 (for IA) or 3 (for MI) active functions yields a significant speed-up at these settings, due to the larger proportion of such tetrahedra.

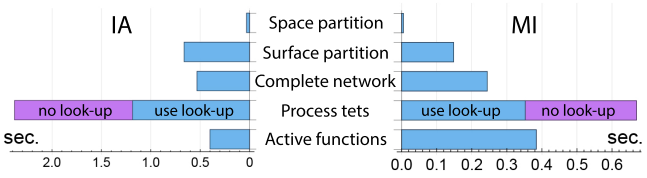


Fig. 16. Breakdown of our IA (left) and MI (right) algorithms' timing into the main stages: identifying active functions within each tetrahedron (including computing the signs at the grid vertices), processing each tetrahedron (showing timing of both using and not using look-up tables for 2 functions in IA or 3 functions in MI), connecting polygons into a complete surface network, computing topological partitioning of the surface network, and computing partitioning of the 3D space.

8.4 Visual comparisons

A key benefit of computing IA or MI on piecewise-linearly interpolation functions is its ability to more faithfully reproduce fine surface features, which are easily missed or distorted using traditional label-separating methods, as illustrated in 2D in Figures 2 and 3. Here we further demonstrate the benefit in 3D.

Figure 17 shows the result of label-separation for computing IA on a pair of slightly shifted spherical functions (similar to those used in our robustness stress test, but with a larger shift distance of ϵ =1e-3) and four intersecting spheres (taken from our performance evaluation), both on a 100^3 grid. Due to the presence of close-by surfaces, both results contain a large number of spurious topological features, such as pockets of small 3D regions and duplicated chains of non-manifold curves (see close-up for the second example).

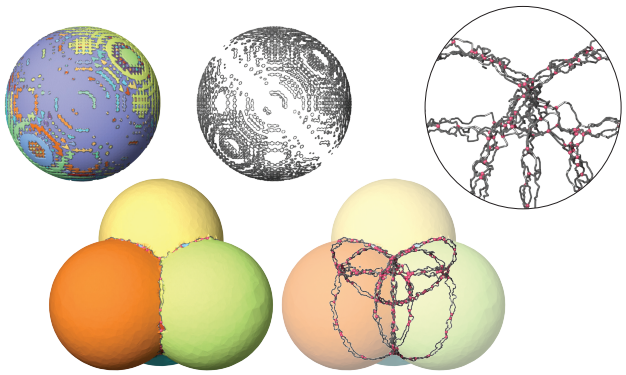


Fig. 17. The IA computed using a label-separation algorithm for a pair of shifted spherical functions (top left and middle) and four intersecting spheres (bottom). Top right shows a zoomed in view of the complex non-manifold structure in the second example.

Figure 18 left compares the MI produced by our algorithm and label-separation on two shifted spherical functions and a constant zero function. To make the case more challenging, we negate one of the spherical functions, so that the true MI should include a thin layer sandwiched between two hemispheres. While our algorithm correctly produces such a result, label-separation once again produces numerous small pockets. Figure 18 right shows the Apollonius Diagram of 4 spheres where one of the patches (blue) has a narrow neck. In addition to creating jagged non-manifold curves, the label-separating algorithm produces an incorrect combinatorial structure wherein that patch is broken into two.

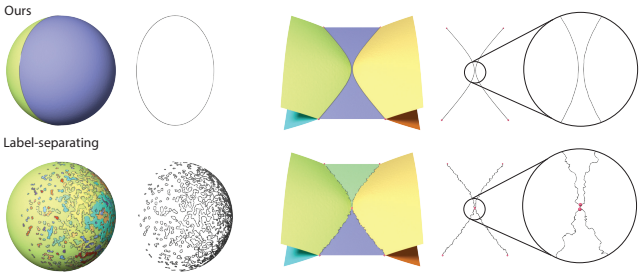


Fig. 18. Comparing MI computed by our algorithm (top) and label-separation (bottom) on a pair of shifted and negatived spherical functions (left) and four spherical distance functions (right).

8.5 More examples

We conclude this section with a few challenging and real-world examples. As mentioned earlier, IA is commonly used for representing complex solid objects from simple primitives, notably via CSG. Figure 19 shows our results on a set of implicit functions obtained from the InverseCSG work of Du et al. [2018]. These are low-degree polynomial functions (e.g., planes, spheres, cylinders, and tori) that represent real-world CAD models. Our algorithm produce high-quality arrangements with many fine surface features, and is significantly faster than the alternative approach of computing mesh arrangement on the extracted level sets (see statistics inside the figure).

As a challenging and visually interesting example of MI, we used our algorithm to compute the Voronoi diagram of a group of 3D lines and circles. Computing such diagrams is an extremely challenging task due to their curved geometry and complex combinatorial structures [Hemmer et al. 2010]. The input to our algorithm is a grid of sampled distances to the given lines and circles. We showcase several examples in Figure 20, including rotating lines (top), lines that sweep along the Mobius strip (bottom left), and two groups of Villarceau circles (bottom right). In addition, we compare our results with those produced by the label-separating algorithm, whose results are both geometrically less smooth and topologically more complex, as shown in the zoomed-in views and the noted numbers of patches and non-manifold curves.

9 DISCUSSION

We presented algorithms for computing two commonly used implicit surface networks, implicit arrangements (IA) and material interfaces (MI), on a tetrahedral grid. To the best of our knowledge, they

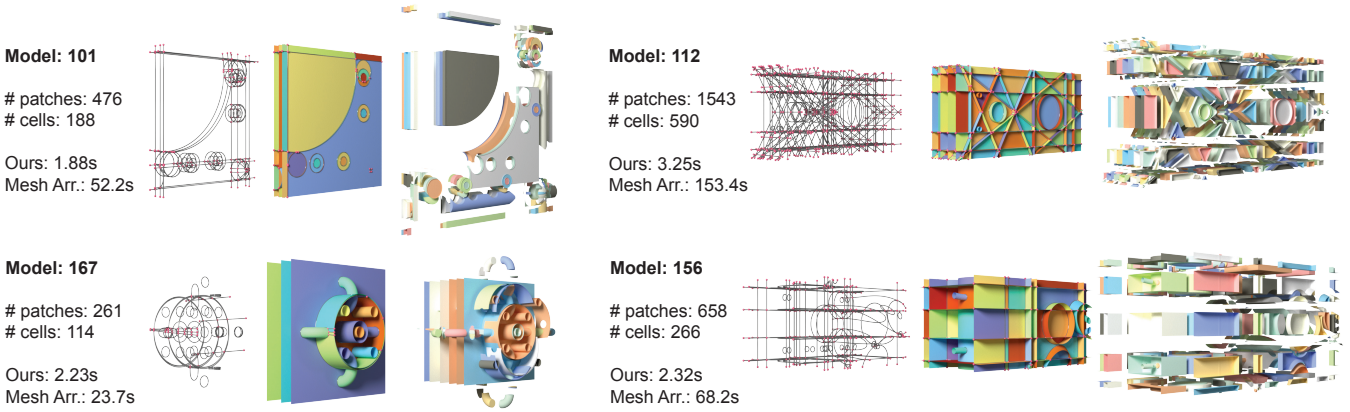


Fig. 19. IA computed by our algorithm on implicit functions obtained from [Du et al. 2018] for representing CAD models. Each example shows the non-manifold curve networks, patches, and 3D regions (in exploded view). Complexity of each example, running time of our method (full pipeline) and mesh arrangement are noted.

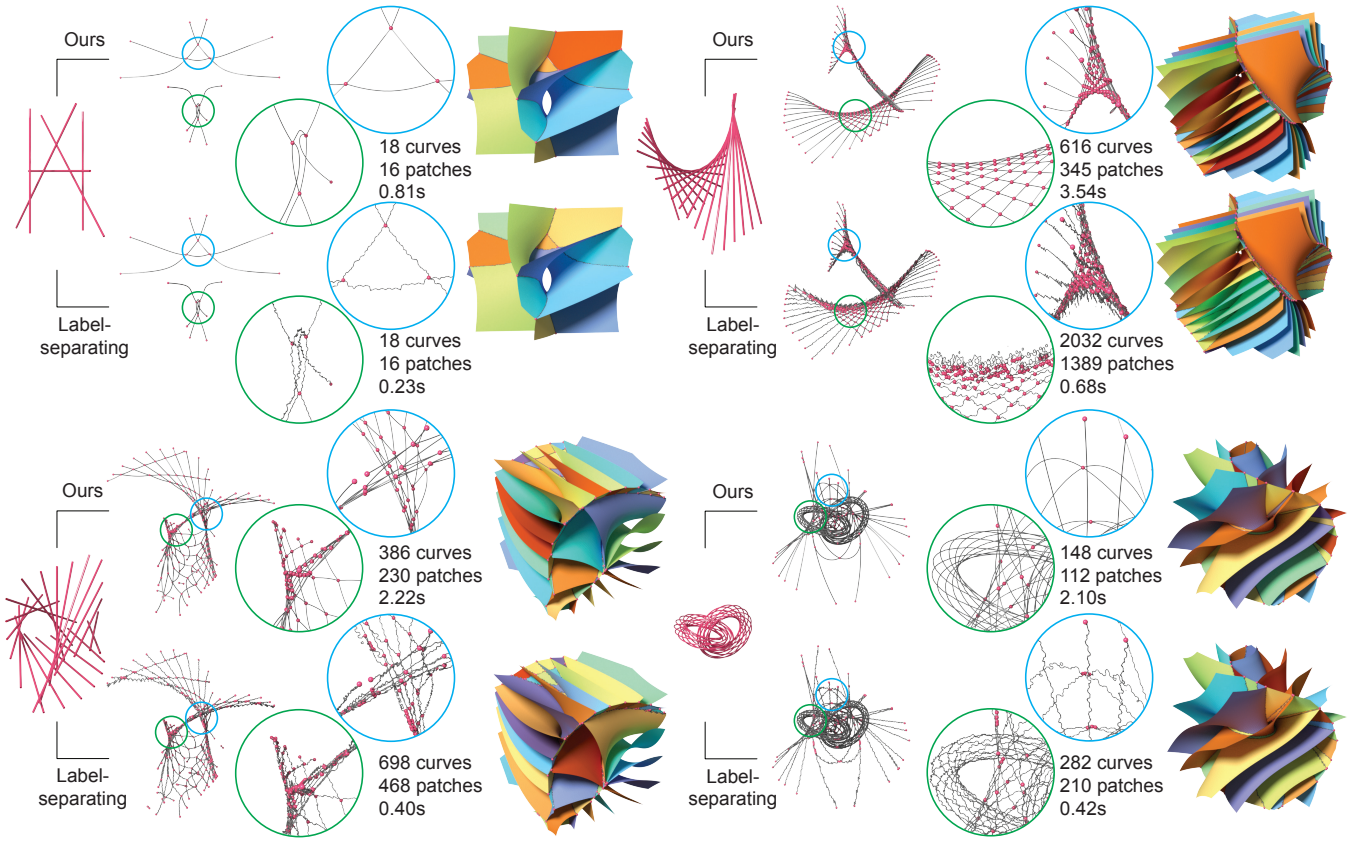


Fig. 20. Voronoi diagrams of 3D lines (top left: 5 rotating lines; top right: 20 rotating lines; bottom left: 21 lines that sweep a Mobius strip) and circles (bottom right: 22 Villarceau circles on two tori) computed by our algorithm and the label-separating algorithm. The zoom-in views highlight regions on the non-manifold curve networks where the two algorithms produce notably different geometry and/or topology. The combinatorial complexity of each surface network and the running times (up to mesh extraction, before space decomposition) are noted.

are the first that can robustly compute the correct combinatorial structure of either surface network for any number of piecewise

linear functions. Our algorithms can reproduce fine surface features much more faithfully than traditional, label-separating methods, at the cost of a three- to four-fold increase in running time.

There are a number of venues for future work. Several stages in our algorithm (e.g., "Active Functions" and "Process Tets" in Figure 16) process each tetrahedron independently, and hence they can be easily parallelized to improve efficiency. Additional speed-up may be possible by adopting successful strategies in polygonalizing implicit surfaces, such as surface tracking. As with other grid-based methods, our algorithms can produce many low-quality triangles (e.g., the zoom-in views in Figures 13 (a) and 15 (a)) that will require further processing [Dillard et al. 2007; Saye 2015]. Alternatively, we can use the non-manifold curve network produced by our algorithms to guide Delaunay-based methods to produce high-quality meshes with an accurate combinatorial structure [Boltcheva et al. 2009; Dey et al. 2012]. In an orthogonal direction, and enabled by our algorithms, we would like to explore grid-generation methods that result in geometrically and topologically correct discretizations of IA or MI for given implicit functions. Finally, while our algorithm is designed for piecewise linear functions on simplicial grids (e.g., triangular or tetrahedral), robust computation of IA or MI for higher-order functions and/or on a non-simplicial grid (e.g., cubical) remains an open and challenging problem.

ACKNOWLEDGMENTS

This work is supported by NIH grant U2C CA233303 and a gift from Adobe Systems. We would like to thank anonymous reviewers for their valuable suggestions.

REFERENCES

Pankaj K Agarwal and Micha Sharir. 2000. Arrangements and their applications. In *Handbook of computational geometry*. Elsevier, 49–119.

Lionel Alberti, Bernard Mourrain, and Julien Wintz. 2008. Topology and arrangement computation of semi-algebraic planar curves. *Computer Aided Geometric Design* 25, 8 (2008), 631–651.

Marco Attene. 2020. Indirect predicates for geometric constructions. *Computer-Aided Design* 126 (2020), 102856.

Brigham Bagley, Shankar Sastry, and Ross Whitaker. 2016. A Marching-tetrahedra Algorithm for Feature-preserving Meshing of Piecewise-smooth Implicit Surfaces. *Procedia Engineering* 163 (12 2016), 162–174.

C. L. Bajaj and T. Dey. 1990. Polygon Nesting and Robustness. *Inf. Process. Lett.* 35, 1 (jun 1990), 23–32.

Eric Berberich, Pavel Emeliyanenko, Alexander Kobel, and Michael Sagraloff. 2012. Arrangement computation for planar algebraic curves. In *Proceedings of the 2011 International Workshop on Symbolic-Numeric Computation*. 88–98.

Gilbert Bernstein and Don Fussell. 2009. Fast, exact, linear booleans. In *Computer Graphics Forum*, Vol. 28. Wiley Online Library, 1269–1278.

Martin Bertram, Gerd Reis, Rolf Hendrik van Lengen, Sascha Köhn, and Hans Hagen. 2005. Non-manifold mesh extraction from time-varying segmented volumes used for modeling a human heart.. In *EuroVis*. Citeseer, 199–206.

Jules Bloomenthal and Keith Ferguson. 1995. Polygonization of non-manifold implicit surfaces. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. 309–316.

Dobrina Boltcheva, Mariette Yvinec, and Jean-Daniel Boissonnat. 2009. Feature Preserving Delaunay Mesh Generation from 3D Multi-Material Images. In *Proceedings of the Symposium on Geometry Processing* (Berlin, Germany) *(SGP '09)*. 1455–1464.

Kathleen Sue Bonnell, Mark A Duchaineau, Daniel R Schikore, Bernd Hamann, and Kenneth I Joy. 2003. Material interface reconstruction. *IEEE Transactions on Visualization and Computer Graphics* 9, 4 (2003), 500–511.

Jonathan R Bronson, Shankar P Sastry, Joshua A Levine, and Ross T Whitaker. 2014. Adaptive and unstructured mesh cleaving. *Procedia engineering* 82 (2014), 266–278.

Michael Burns, Janek Klawe, Szymon Rusinkiewicz, Adam Finkelstein, and Doug DeCarlo. 2005. Line drawings from volume data. *ACM Transactions on Graphics (TOG)* 24, 3 (2005), 512–518.

Marcel Campen and Leif Kobbelt. 2010. Exact and robust (self-) intersections for polygonal meshes. In *Computer Graphics Forum*, Vol. 29. Wiley Online Library, 397–406.

Bernard Chazelle and Herbert Edelsbrunner. 1992. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM (JACM)* 39, 1 (1992), 1–54.

Gianmarco Cherchi, Marco Livesu, Riccardo Scateni, and Marco Attene. 2020. Fast and robust mesh arrangements using floating-point arithmetic. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 1–16.

Bruno Rodrigues De Araújo, Daniel S Lopes, Pauline Jepp, Joaquim A Jorge, and Brian Wyvill. 2015. A survey on implicit surface polygonization. *ACM Computing Surveys (CSUR)* 47, 4 (2015), 1–39.

Tamal K Dey, Firdaus Janoos, and Joshua A Levine. 2012. Meshing interfaces of multi-label data with Delaunay refinement. *Engineering with Computers* 28, 1 (2012), 71–82.

Lorenzo Diazzi and Marco Attene. 2021. Convex polyhedral meshing for robust solid modeling. *ACM Transactions on Graphics (TOG)* 40, 6 (2021), 1–16.

Scott Dillard, John Bingert, Dan Thoma, and Bernd Hamann. 2007. Construction of Simplified Boundary Surfaces from Serial-sectioned Metal Micrographs. *Visualization and Computer Graphics, IEEE Transactions on* 13 (12 2007), 1528–1535.

Akio Doi and Akio Koide. 1991. An Efficient Method of Triangulating Equi-Valued Surfaces by Using Tetrahedral Cells. *IEICE Transactions on Information and Systems* 74 (1991), 214–224.

Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. 2018. Inversecsg: Automatic conversion of 3d models to csg trees. *ACM Transactions on Graphics (TOG)* 37, 6 (2018), 1–16.

Xingyi Du, Qingnan Zhou, Nathan Carr, and Tao Ju. 2021. Boundary-Sampled Halfspaces: A New Representation for Constructive Solid Modeling. *ACM Trans. Graph.* 40, 4 (2021).

Laurent Dupont, Michael Hemmer, Sylvain Petitjean, and Elmar Schömer. 2007. Complete, exact and efficient implementation for computing the adjacency graph of an arrangement of quadrics. In *European Symposium on Algorithms*. Springer, 633–644.

Herbert Edelsbrunner and John Harer. 2002. Jacobi sets of multiple Morse functions. *Foundations of Computational Mathematics, Minneapolis* (2002), 37–57.

Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. 1996. The CGAL kernel: A basis for geometric computation. In *Workshop on Applied Computational Geometry*. Springer, 191–202.

Andreas Fabri and Sylvain Pion. 2009. CGAL: The computational geometry algorithms library. In *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*. 538–539.

Powei Feng, Tao Ju, and Joe Warren. 2010. Piecewise Tri-linear Contouring for Multi-Material Volumes. *Lecture notes in computer science* 6130, 43–56.

Google. 2017. Abseil: C++ Libraries from Google. https://abseil.io/.

Jiateng Guo, Xulei Wang, Jiangmei Wang, Xinwei Dai, Lixin Wu, Chaoling Li, Fengdan Li, Shanjun Liu, and Mark Walter Jessell. 2021. Three-dimensional geological modeling and spatial analysis from geotechnical borehole data using an implicit surface and marching tetrahedra algorithm. *Engineering Geology* 284 (2021), 106047.

Hans-Christian Hege, Martin Seebass, Detlev Stalling, and Malte Zöckler. 1997. A generalized marching cubes algorithm based on non-binary classifications. (1997).

Michael Hemmer, Ophir Setter, and Dan Halperin. 2010. *Constructing the exact Voronoi diagram of arbitrary lines in space*. Ph. D. Dissertation. INRIA.

Alec Jacobson, Daniele Panozzo, et al. 2018. libigl: A simple C++ geometry processing library. https://libigl.github.io/.

Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. 2002. Dual contouring of hermite data. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. 339–346.

Byungmoon Kim. 2010. Multi-phase fluid simulations using regional level sets. *ACM Transactions on Graphics (TOG)* 29, 6 (2010), 1–8.

Dae Hyun Kim, Ulf Doering, and Beat Bruderlin. 2000. Polygonization of non-manifolds with the aid of interval operators. In *Proc. Implicit Surfaces*. 145–151.

Bruno Lévy. 2016. Robustness and efficiency of geometric programs: The Predicate Construction Kit (PCK). *Computer-Aided Design* 72 (2016), 3–12.

Bruno Lévy and Alain Filbois. 2015. Geogram: a library for geometric algorithms. (2015).

Jyh-Ming Lien, Vikram Sharma, Gert Vegter, and Chee Yap. 2014. Isotopic arrangement of simple curves: An exact numerical approach based on subdivision. In *International Congress on Mathematical Software*. Springer, 277–282.

Patric Ljung and Anders Ynnerman. 2003. Extraction of intersection curves from iso-surfaces on co-located 3d grids. In *The Annual SIGRAD Conference. Special Theme-Real-Time Simulations. Conference Proceedings from SIGRAD2003*. Citeseer, 23–28.

William E. Lorensen and Harvey E. Cline. 1987. Marching cubes: A high resolution 3D surface construction algorithm.. In *SIGGRAPH*, Maureen C. Stone (Ed.). ACM, 163–169.

Frank Losasso, Tamar Shinar, Andrew Selle, and Ronald Fedkiw. 2006. Multiple Interacting Liquids. *ACM Trans. Graph.* 25, 3 (jul 2006), 812–819.

Miriah Meyer, Ross Whitaker, Robert M. Kirby, Christian Ledergerber, Hanspeter Pfister, and Senior Member. 2008. Particle-based sampling and meshing of surfaces in multimaterial volumes. *IEEE Transactions on Visualization and Computer Graphics* (2008), 1539–1546.

Bernard Mourrain, Jean-Pierre Técourt, and Monique Teillaud. 2005. On the computation of an arrangement of quadrics in 3d. *Computational Geometry* 30, 2 (2005), 145–164.

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.

Gregory Nielson and R. Franke. 1997. Computing the separating surface for segmented data. 229–233.

J Pons, E Ségonne, Jean-Daniel Boissonnat, Laurent Rineau, Mariette Yvinec, and Renaud Keriven. 2007. High-Quality Consistent Meshing of Multi-Label Datasets. *Information processing in medical imaging : proceedings of the ... conference* 20, 198–210.

Bernhard Reitinger, Alexander Bornik, and Reinhard Beichel. 2005. Constructing smooth non-manifold meshes of multi-labeled volumetric datasets. (2005).

Aristides AG Requicha and Herbert B Voelcker. 1977. Constructive solid geometry. (1977).

RI Saye. 2015. An algorithm to mesh interconnected surfaces via the Voronoi interface. *Engineering with Computers* 31, 1 (2015), 123–139.

Robert I Saye and James A Sethian. 2012. Analysis and applications of the Voronoi implicit interface method. *J. Comput. Phys.* 231, 18 (2012), 6051–6085.

Elmar Schömer and Nicola Wolpert. 2006. An exact and efficient approach for computing a cell in an arrangement of quadrics. *Computational Geometry* 33, 1-2 (2006), 65–97.

M Haitham Shammaa, Yutaka Ohtake, and Hiromasa Suzuki. 2010. Segmentation of multi-material CT data of mechanical parts for extracting boundary surfaces. *Computer-Aided Design* 42, 2 (2010), 118–128.

M Haitham Shammaa, Hiromasa Suzuki, and Yutaka Ohtake. 2008. Extraction of isosurfaces from multi-material CT volumetric data of mechanical parts. In *Proceedings of the 2008 ACM symposium on solid and physical modeling*. 213–220.

Jonathan Richard Shewchuk. 1997. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry* 18, 3 (1997), 305–363.

Kokichi Sugihara, Masao Iri, et al. 1989. A solid modelling system free from topological inconsistency. *Journal of Information Processing* 12, 4 (1989), 380–393.

Jean-Philippe Thirion and Alexis Gourdon. 1996. The 3D marching lines algorithm. *Graphical Models and Image Processing* 58, 6 (1996), 503–509.

VP Voloshin, S Beaufils, and NN Medvedev. 2002. Void space analysis of the structure of liquids. *Journal of molecular liquids* 96 (2002), 101–112.

Peihui Wang, Na Yuan, Yuewen Ma, Shiqing Xin, Ying He, Shuangmin Chen, Jian Xu, and Wenping Wang. 2020. Robust Computation of 3D Apollonius Diagrams. In *Computer Graphics Forum*, Vol. 39. Wiley Online Library, 43–55.

Hans-Martin Will. 1999. *Computation of additively weighted Voronoi cells for applications in molecular biology*. Ph. D. Dissertation. ETH Zurich.

Ziji Wu and John M Sullivan Jr. 2003. Multiple material marching cubes algorithm. *Internat. J. Numer. Methods Engrg.* 58, 2 (2003), 189–207.

Shuntaro Yamazaki, Kiwamu Kase, and Katsushi Ikeuchi. 2002. Non-manifold implicit surfaces based on discontinuous implicitization and polygonization. In *Geometric Modeling and Processing. Theory and Applications. GMP 2002. Proceedings*. IEEE, 138–146.

Yongjie Zhang, Thomas Hughes, and Chandrajit Bajaj. 2007. Automatic 3D Mesh Generation for a Domain with Multiple Materials. *Proceedings of the 16th International Meshing Roundtable*, 367–386.

Yongjie Zhang, Thomas JR Hughes, and Chandrajit L Bajaj. 2008. Automatic 3d mesh generation for a domain with multiple materials. In *Proceedings of the 16th international meshing roundtable*. Springer, 367–386.

Yongjie Jessica Zhang and Jin Qian. 2012. Resolving topology ambiguity for multiple-material domains. *Computer Methods in Applied Mechanics and Engineering* 247 (2012), 166–178.

Wen Zheng, Jun-Hai Yong, and Jean-Claude Paul. 2009. Simulation of bubbles. *Graphical Models* 71, 6 (2009), 229–239.

Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. 2016. Mesh arrangements for solid geometry. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–15.

A CORRECTNESS OF TOPOLOGICAL RAY SHOOTING

We show that the algorithm presented in Section 7.1 correctly identifies all shells bounding each 3D region partitioned by the surface network. Denote the surface network as N, the tetrahedral grid as M, the boundary and interior of M respectively as ∂M and ΩM, the ordering of vertices of M as Φ, and the sink vertex as s. We assume the choice of Φ meets the criteria set forth in Section 7.1.

Our proofs hold for any N, M that meet the following assumptions:

- A1: N partitions ΩM into distinct 3D regions, so that each polygon of N bounds two different regions.
- A2: ΩM is simply connected (i.e., a single component without cavities or handles).
- A3: s is not on N.
- A4: N induces a convex decomposition within each tetrahedron of M.

Assumption A1 implies that each shell S of N is either closed or open only at ∂M. By assumption A2, S must separate ΩM into *inside* and *outside* regions, so that S is oriented towards the outside regions. By definition of shells, each shell has exactly one outside region but could have multiple inside regions (e.g., consider a shell consisting of two oppositely positioned cones that meet at their tips and oriented towards the outside of both cones).

Now consider a connected component C of N and the regions of ΩM partitioned by C. Each region is the outside region of one of the shells of C and is inside all the other shells. Furthermore, due to assumption A3, exactly one of the shells of C has s in its outside region. We call this shell the *exterior shell* of C, and the remaining shells of C the *interior shells*.

The observations above lead to the following characterization of regions of ΩM partitioned by the entire surface network N:

Lemma A.1. *Let R be a region of ΩM partitioned by N, then:*

(1) *No two shells bounding R belong to the same connected component of N.*
(2) *If R contains s, R is bounded by only exterior shells.*
(3) *If R does not contain s, R is bounded by exactly one interior shell and zero or more exterior shells.*

Proof. We prove each statement in turn.

(1) As observed above, the outside regions of two different shells of the same connected component of N are distinct. On the other hand, since R is outside each shell bounding R, the intersection of the outside regions of all of R's bounding shells is non-empty, and hence no two bounding shells can belong to the same connected component of N.
(2) Since R is outside each of its bounding shell, so is s. By definition of exterior shells, each of R's bounding shells is the exterior shell of a connected component of N.
(3) We need to prove that s lies inside exactly one of R's bounding shell but outside the remaining shells. We first show that s must lie inside at least one shell. Consider a path that starts from some point p inside R and ends at s. Since s is not in R, the path must cross the boundary of R for an odd number of times. Therefore it must cross at least one of the R's bounding shells for an odd number of times, implying that s lies in an inside region of that shell. Next we show that it is not possible for s to simultaneously be inside two of R's bounding shells, say S_1, S_2. Otherwise, the inside regions of S_1, S_2 must have a non-empty intersection. Consequently, it is possible to travel from the outside of S_1 to its inside, passing through R and S_2 but not S_1 itself, which contradicts that the inside and outside regions of S_1 are separated by S_1.

□

We define the *extremal* edge-point of a component C as the edge-point with the lexicographically smallest ordering vector. Such edge-points are where the rays start in our algorithm. The next statement holds because of Assumption A4:

Lemma A.2. *The extremal edge-point exists for each connected component C of N and lies on the exterior shell of C.*

Proof. To prove existence, we need to show that C must intersect with some edges of M. We first show that C cannot intersect the interior of a tetrahedron t without intersecting its edges or faces. Otherwise, C must be completely contained inside t, which makes the continuous region surrounding C non-convex. This contradicts with assumption A4. Similarly, we can show that C cannot intersect a triangle face f without intersecting its edges. Otherwise, the intersection of C with f consists of one or more connected component that are disjoint from the edges of f, implying that the region surrounding each component is non-convex. This again contradicts with A4. Therefore C must intersect some edges of M. Since such intersections (i.e., edge-points) are finite in number, the extremal edge-point must exist.

Let the extremal edge-point of C be p with ordering vector $\{v, u, i\}$. Since M has no other sink vertices than s, there exists a path P of grid edges the connect v and s via vertices with decreasing order in Φ (P is empty if $v = s$). Since p's ordering vector is smallest (lexicographically) among all edge-points of C, no other edge-point of C exists on P or on the segment of the grid edge between p and v. This implies that s lies in the outside region of the shell containing p and oriented towards v, which makes that shell the exterior shell. □

We now prove the correctness of the algorithm:

Proposition A.3. *After topological ray-shooting, each connected component of the directed graph G (excluding the outer shell) consists of all shells bounding a 3D region of ΩM partitioned by N.*

Proof. By construction of the algorithm, the interior of each ray does not intersect with N, and hence the starting shell and ending shell (which can be the outer shell if the ray ends at s) of each ray must bound a continuous region. That is, each edge in G connects two shells that bound the same 3D region partitioned by N.

To complete the proof, we need to also show that, for each 3D region R bounded by a set of shells S_R (including the outer shell), the subgraph of G spanning S_R is connected. We consider two different types of R below.

If R does not contain s, then by Lemma A.1, S_R consists of one interior shell and zero or more exterior shells. We only need to consider the case that S_R contains exterior shells. By Lemma A.2 and its argument within, the algorithm traces one ray from each exterior shell into its outside, and since $s \notin R$ the ray must end at another (interior or exterior) shell of S_R. Let S_1, S_2 be the pair of starting and ending shells of a ray. By Lemma A.1, they must lie on different connected components of N, which we denote as C_1, C_2. By construction of the ray, the extremal edge-point of C_2 must have a (lexicographically) smaller ordering vector than that of C_1. As a result, the subgraph of G spanning S_R cannot have cycles (i.e., it is a directed acyclic graph, or a DAG). Since each connected component of a DAG must have at least one node without outgoing edges, and because the only such node in this subgraph is the (single) interior shell, the subgraph must be connected.

If R contains s, then Lemma 3 ensures that S_R consists of only exterior shells plus the outer shell. Using the same arguments as above, we can show that the subgraph of G spanning S_R must be a DAG, and the only node that does not have any outgoing edges is the outer shell. Hence the subgraph is connected. □