Numer Algor (2018) 77:1029–1068
DOI 10.1007/s11075-017-0352-7

ORIGINAL PAPER

Robust intersection of structured hexahedral meshes and degenerate triangle meshes with volume fraction applications

Frida Svelander[1,2] · Gustav Kettil[1,2] · Tomas Johnson[1] · Andreas Mark[1] · Anders Logg[1,2] · Fredrik Edelvik[1]

Received: 16 May 2016 / Accepted: 23 May 2017 / Published online: 1 July 2017

Abstract Two methods for calculating the volume and surface area of the intersection between a triangle mesh and a rectangular hexahedron are presented. The main result is an exact method that calculates the polyhedron of intersection and thereafter the volume and surface area of the fraction of the hexahedral cell inside the mesh. The second method is approximate, and estimates the intersection by a least squares plane. While most previous publications focus on non-degenerate triangle meshes, we here extend the methods to handle geometric degeneracies. In particular, we focus on large-scale triangle overlaps, or double surfaces. It is a geometric degeneracy that can be hard to solve with existing mesh repair algorithms. There could also be situations in which it is desirable to keep the original triangle mesh unmodified. Alternative methods that solve the problem without altering the mesh are therefore presented. This is a step towards a method that calculates the solid area and volume fractions of a degenerate triangle mesh including overlapping triangles, overlapping meshes, hanging nodes, and gaps. Such triangle meshes are common in industrial applications. The methods are validated against three industrial test cases. The validation shows that the exact method handles all addressed geometric degeneracies, including double surfaces, small self-intersections, and split hexahedra.

Keywords Cut-cell · Volume fraction · Mesh repair · Overlapping triangles · Split hexahedra

✉ Frida Svelander
frida.svelander@fcc.chalmers.se; frisve@chalmers.se

[1] Fraunhofer-Chalmers Research Centre for Industrial Mathematics, Gothenburg, Sweden

[2] Department of Mathematical Sciences, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden

1 Introduction

The need for computing intersections between polyhedral objects is common in many applications, such as computer graphics, simulations, and robotics [12]. Polyhedral objects are typically used as discrete representations of real objects, for example, when computational fluid dynamics (CFD) is used to solve multiphase flow problems. The surface of a solid object can then be represented by a triangle mesh and immersed in a background fluid grid consisting of rectangular hexahedra. The mesh needs to be connected to the background grid. In the connection step or later in the solution process, it can be necessary or useful to compute the intersection between the triangle mesh and each fluid cell, to find information about the cut cells.

Two well-known methods for connecting the mesh and the grid are cut cell methods [2, 11, 26] and immersed boundary (IB) methods [19, 22]. In cut cell methods, the fluid cells that are intersected by the mesh are refined to fit the mesh. In IB methods, body-fitted grids and cut cells are replaced by other approaches for connecting the mesh and the grid. The cut cell geometric information is clearly needed in the connection step of a cut cell method, but can be of interest also in an IB method. A typical application is in the calculation of fluxes. There is potential to improve the accuracy in the simulated fluxes if the fraction of each cell face inside the triangle mesh is known. This information can be extracted from the cut cell information.

Several methods for extracting the geometry of a cut cell are found in the literature. Some of these are reviewed in Section 2. However, few of the publications focus on degenerate triangle meshes. Degeneracies such as hanging nodes (T-vertices), gaps, cracks, overlapping meshes, or overlapping triangles are common in triangle meshes used for engineering applications. This aspect is important to consider when designing or using an algorithm that extracts the geometry of a cut cell. One option is to repair the mesh before it is given as input to the geometry extraction algorithm. Mesh repair is a broad field, partly surveyed in Section 2. Another, less common, option is to let the geometry extraction algorithm itself handle the degeneracies. This is done for example in [27].

In this article, we present a method similar to [1, 2, 11] for extracting the exact geometric information about a Cartesian cell cut by a triangle mesh. In contrast to earlier works, our method is geometrically robust in the sense that it can handle large-scale triangle overlaps, what we will also call double surfaces, without the need for previously repairing the mesh. To some extent, our method also handles more general self-intersecting meshes. This is an improvement since large-scale triangle overlaps appear frequently in engineering applications and are hard to repair without undesirable side effects (see Section 2). Moreover, none of [1, 2, 11] go into details about the method and implementation, that is why there is still room for explanation of important concepts. We will consider some of these concepts here.

The method is in the following referred to as ”the exact method.” It is not completely exact since we use finite precision floating point arithmetics in our intersection routines. What we mean by exactness is that the exact polygons and polyhedrons of intersection are calculated given the intersection points. An approximate method where a least squares plane is fitted to the cell-triangle mesh intersection is also

developed and compared to the exact method. The approximate method is intended for highly resolved hexahedral grids, for which it is reasonable to approximate the cell-triangle mesh intersection with a plane. It has some limitations and is a complement to the exact method.

What we essentially have solved is the purely geometric problem of intersecting a degenerate triangle mesh and a rectangular hexahedron. However, the proposed methods are specially designed to be used in an immersed boundary method [20]. As such, we are interested in both the cell-triangle mesh intersection and the face-triangle mesh intersections. This has been taken into account in the development of the methods.

2 Related work

Different methods for extracting the geometry of a cut cell are found in the literature. In 1994, Quirk [23] introduces the Cartesian boundary method for complex two-dimensional geometries, where the cut cells are identified and classified into one of twelve types depending on how the solid and the cell boundary intersect. The intersection between the geometry and the cell is approximated by a line. This approximation is motivated by the fact that the existence of a cell with more than two intersection points indicates that the grid is not fine enough to resolve the solid properly.

Yang et al. [26] take this a bit further and describe a cut cell method in three dimensions where the part of the triangle mesh in the interior of the cell is approximated by a non-planar quadrilateral. They find the area of the quadrilateral and each cut face and use Gauss's divergence theorem to calculate the volume of the part of the cell located inside the triangle mesh.

In [1, 2], Aftosmis et al. present an approach for completely resolving the geometry of a cut cell in three dimensions. They introduce triangle polygons, face polygons, and face segments to describe the polyhedron that represents a cut cell. To construct the triangle polygons, they use the Sutherland-Hodgman algorithm [25] for clipping each triangle against the cell boundary. They mention that face polygons are easily formed by connecting face segments with the edges of the cut cell. Cieslak et al. [11] also present a cut cell method that preserves the real geometry by finding the exact cut out polygons and polyhedron. They find the face polygons through connectivity criteria such as common faces or common triangles.

In [11, 23, 26], little or nothing is mentioned about geometrically degenerate triangle meshes, while Aftosmis et al. [2] adopt a volumetric approach to degenerate and overlapping triangle meshes. Volumetric approaches belong to one of two groups of mesh repair methods. The other group consists of surface-oriented methods. Surface-oriented methods such as [7, 15, 18] operate directly on the degenerate geometry, while volumetric methods such as [9, 17, 21] create an intermediate volumetric representation of the geometry. The intermediate representation is used to create a new mesh without degeneracies. Most repair methods either have restrictions on the input mesh or drawbacks [4, 10]. Volumetric methods are typically robust, but destroy

connectivity structures of the input geometry and could lead to loss of model features. Surface-oriented methods are often better at preserving details of the input mesh, but are typically not as robust as the volumetric methods. A survey and categorization of mesh repair approaches was done by Attene, Campen, and Kobbelt in 2013 [4].

A robust surface-oriented repair method that focuses particularly on self-intersecting meshes and double surfaces was proposed by Attene in 2014 [5]. It computes the exact outer hull of the input triangle mesh by finding the self-intersections, subdividing the triangles along the self-intersections, and finally, removing excess triangles. Exact arithmetics is used to ensure correct intersections, but only when ordinary floating point operations are not sufficiently accurate.

Some approaches to a combination of mesh repair and geometric operations are found in the literature, where the method by Aftosmis et al. [2] is already mentioned. Another approach was recently proposed by Zhou et al. [27], who integrate mesh repair and set operations such as difference and union of an arbitrary number of input meshes. They assign a generalized winding number to different parts of space, defined by the intersections between the input meshes. The winding numbers are used to repair the meshes or perform set operations.

The repair method proposed in [5] could be used to remove the self-intersecting double surfaces, and [27] could be used to find the requested cut cell information if each hexahedral cell is first triangulated. In contrast to the three-dimensional set operations in [27], we present a method that operates on the two-dimensional faces of the intersection between the triangle mesh and the hexahedron. Thus, our method is adapted for finding both cell-mesh and face-mesh intersections. Only minor changes to the original method are required to handle double surfaces and some other types of self-intersections. Due to the small computational overhead of handling the degenerate cases, we claim that our method is a good alternative to the previous methods.

3 Problem formulation

Let $\mathscr{T} = \{T_i\}_{i=1}^{n_T}$ be a triangle mesh, where $T_i \subset \mathbb{R}^3$, $i \in \{1, \ldots, n_T\}$, are triangles. $\mathscr{T}$ is represented by a set of indexed vertices $\{\mathbf{v}_i\}_{i=1}^{n_v}$, and each triangle T_i is defined by three vertices. Note that $n_v \leq 3\,n_T$, since vertices are unique and shared between triangles. The triangular mesh is oriented by the triangle normals $\{\mathbf{n}_i\}_{i=1}^{n_T}$ pointing out of $\mathscr{T}$. If $\mathscr{T}$ encloses a bounded volume, the interior of $\mathscr{T}$ is denoted $\Omega_{\mathscr{T}}$.

Let $\mathscr{C} \subset \mathbb{R}^3$ be a Cartesian cell, i.e., an axis-aligned cuboid consisting of its boundary $\partial\mathscr{C}$ and interior $\Omega_{\mathscr{C}}$. Then, $\partial\mathscr{C} = \bigcup_{i=1}^{6} F_i$, where $\{F_i\}_{i=1}^{6}$, are the rectangular cell faces. A cell $\mathscr{C}$ also has twelve edges $\{E_i\}_{i=1}^{12}$ and eight vertices $\{V_i\}_{i=1}^{8}$.

In this paper, we consider the problem of calculating the percentage or *solid volume fraction* of $\mathscr{C}$ inside $\mathscr{T}$. The solid volume fraction α is given by

$$\alpha = \frac{V(\overline{\Omega_{\mathscr{T}}} \bigcap \mathscr{C})}{V(\mathscr{C})}, \tag{1}$$

where $V(X)$ denotes the volume of a subset X of $\mathbb{R}^3$. We also consider the percentage or *solid area fraction* of each cell face F_i inside $\mathscr{T}$. The solid area fraction β_i corresponding to face F_i is given by

$$\beta_i = \frac{A(\overline{\Omega_{\mathscr{T}}} \bigcap F_i)}{A(F_i)}, \tag{2}$$

where $A(X)$ is the area of a two-dimensional object X. A cell $\mathscr{C}$ intersecting a triangular mesh $\mathscr{T}$ is seen in Fig. 1a, and $\overline{\Omega_{\mathscr{T}}} \bigcap \mathscr{C}$ is seen in Fig. 1b.

More generally, we consider multiple meshes $\mathscr{T}_1, \ldots, \mathscr{T}_N$, where α and β_i are the fractions of the cell and faces inside $\bigcup_{i=1}^{N} \mathscr{T}_i$. Each mesh is allowed to include overlapping triangles or *double surfaces*. To define the concept of a double surface, let $\mathscr{T}_1 = \{T_i^1\}_{i=1}^{n_T^1}$ and $\mathscr{T}_2 = \{T_i^2\}_{i=1}^{n_T^2}$ be proper triangle meshes (without geometric degeneracies). Let $\mathscr{S}_1 \subset \mathscr{T}_1$ and $\mathscr{S}_2 \subset \mathscr{T}_2$ be such that $\bigcup_{T_i \in \mathscr{S}_1 \bigcup \mathscr{S}_2} T_i$ lie in a plane and $(\bigcup_{T_i \in \mathscr{S}_1} T_i) \bigcap (\bigcup_{T_i \in \mathscr{S}_2} T_i) \neq \emptyset$. Then, $\mathscr{D} = (\bigcup_{T_i \in \mathscr{S}_1} T_i) \bigcap (\bigcup_{T_i \in \mathscr{S}_2} T_i)$ is a double surface. A typical double surface is seen in Fig. 2.

4 Algorithms

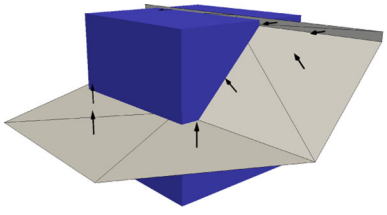
In this section, the two proposed methods to calculate the solid volume fraction α and solid area fractions β_i are presented. Both methods find a polyhedron P that represents $\overline{\Omega_{\mathscr{T}}} \bigcap \mathscr{C}$ and calculate α and β_i by

$$\alpha = \frac{V(P)}{V(\mathscr{C})} \tag{3}$$

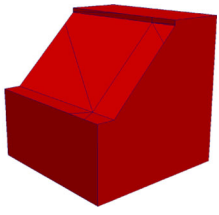
and

$$\beta_i = \frac{A(P \bigcap F_i)}{A(F_i)}, \tag{4}$$

$i \in \{1, \ldots, 6\}$. The first method exactly calculates $\overline{\Omega_{\mathscr{T}}} \bigcap \mathscr{C}$. It is presented in Section 4.3. The second method is an approximate method based on approximating



(a) A cell $\mathscr{C}$ intersecting a triangular mesh $\mathscr{T}$. The arrows represent triangle normals pointing out of $\Omega_{\mathscr{T}}$.



(b) The polyhedron $\overline{\Omega_{\mathscr{T}}} \bigcap \mathscr{C}$ resulting from intersecting $\mathscr{C}$ and $\mathscr{T}$.

Fig. 1 Intersection between a triangle mesh $\mathscr{T}$ and a Cartesian cell $\mathscr{C}$

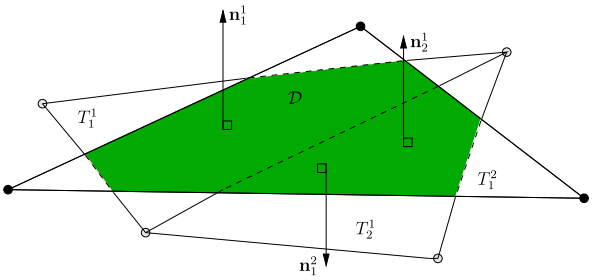


Fig. 2 Triangles from different meshes overlap in a double surface. The double surface is the intersection of triangles T_1^1 and T_2^1 from one triangle mesh $\mathscr{T}_1$, and triangle T_1^2 from a second triangle mesh $\mathscr{T}_2$

$\overline{\Omega_{\mathscr{T}} \bigcap \mathscr{C}}$ from a least squares fit of the intersection $\mathscr{T} \bigcap \mathscr{C}$ between the mesh and the cell. It is presented in Section 4.4.

In this first phase, we require that the triangle meshes are non-degenerate. This restriction is removed in Section 5, where we consider double surfaces and multiple meshes. Pseudo code for the algorithms is presented, with some helper methods outlined in Appendix A. The algorithms in the pseudo code are sometimes simplified, but cover the key steps of a successful implementation.

4.1 Preliminaries

Before we proceed, we define some important concepts: *face polygon*, *cell polygon*, and *intersection point*. We use a notation similar to that of [1, 2]. Given a polyhedron P and cell faces F_i, $i \in \{1, \ldots, 6\}$, the sets $\overline{\left(P \bigcap F_i\right)^\circ}$, are called face polygons, and we say that $\overline{\partial P \bigcap \Omega_{\mathscr{C}}}$ consists of a number of cell polygons (see Fig. 3). Here, $\overline{\left(P \bigcap F_i\right)^\circ}$ is the closure of the interior of the two-dimensional set $P \bigcap F_i$ and $\overline{\partial P \bigcap \Omega_{\mathscr{C}}}$ is the closure of the three-dimensional set $\partial P \bigcap \Omega_{\mathscr{C}}$. Now, let S be a subset of a two-dimensional space, and let $\partial_0(S)$ be the extreme points of S. The set

$$\mathscr{I} = \bigcup_{i=1}^{n_T} \bigcup_{j=1}^{6} \partial_0 \left(T_i \bigcap F_j \right) \tag{5}$$

is called the intersection points between $\mathscr{C}$ and $\mathscr{T}$ (see Fig. 4 where, for a particular triangle T_i, $\bigcup_{j=1}^{6} \partial_0 \left(T_i \bigcap F_j \right)$ is marked by spheres). The set

$$\mathscr{I}_0 = \Omega_{\mathscr{C}} \bigcap \bigcup_{i=1}^{n_v} \mathbf{v}_i \tag{6}$$

are referred to as the triangle vertices of $\mathscr{T}$ inside $\mathscr{C}$.

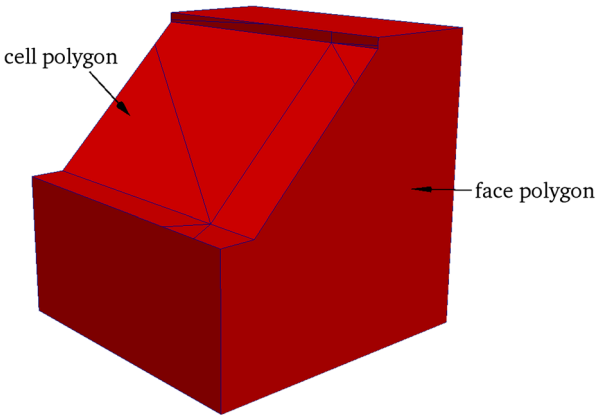


Fig. 3 The polyhedron in Fig. 1b seen as a composition of face polygons and cell polygons. A face polygon is the intersection between a cell face F_i and the polyhedron P. A cell polygon is a face of P in the interior of the cell $\mathscr{C}$

The divergence theorem is used to calculate the volume of a polyhedron P. Applying the theorem to the vector field $\mathbf{F}(x,\ y,\ z) = \dfrac{1}{3}(x,\ y,\ z)$, we get

$$V(P) = \frac{1}{3}\sum_i \mathbf{c}_i \cdot \hat{\mathbf{n}}_i\, A_i, \tag{7}$$

where $\mathbf{c}_i$ is the centroid, $\hat{\mathbf{n}}_i$ the unit normal, and A_i the area of the i:th face S_i of the surface of P. To see this, note that $\nabla \cdot \mathbf{F} = 1$ and

$$\begin{aligned} V(P) &= \textstyle\int_P dV = \int_P \nabla\cdot\mathbf{F}\, dV = \int_{\partial P} \mathbf{F}\cdot\hat{\mathbf{n}}\, dS = \frac{1}{3}\sum_i \int_{S_i}(x,\ y,\ z)\cdot\hat{\mathbf{n}}_i\, dS_i \\ &= \tfrac{1}{3}\textstyle\sum_i \hat{\mathbf{n}}_i \cdot \int_{S_i}(x,\ y,\ z)\, dS_i = \frac{1}{3}\sum_i \hat{\mathbf{n}}_i\cdot\mathbf{c}_i\, A_i. \end{aligned} \tag{8}$$

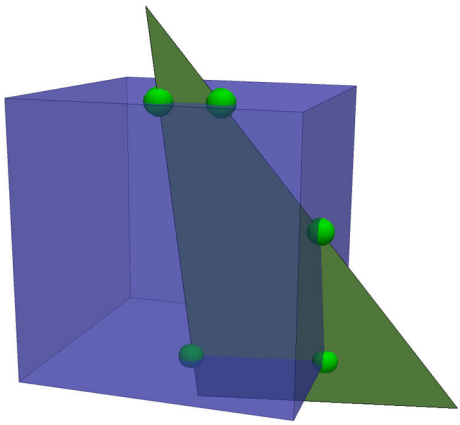


Fig. 4 Intersecting a triangle and a Cartesian cell results in intersection points, here marked by spheres

The areas and centroids in (7) are calculated as in [8]:

$$A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i), \tag{9}$$

and

$$\begin{aligned} c_x &= \tfrac{1}{6A}\sum_{i=0}^{n-1}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i), \\ c_y &= \tfrac{1}{6A}\sum_{i=0}^{n-1}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i), \end{aligned} \tag{10}$$

where $\mathbf{c} = (c_x, c_y)$ and the polygon has vertices $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1}), (x_n, y_n)$, with $(x_n, y_n) = (x_0, y_0)$. A change of coordinates is performed so that (x_i, y_i) are the coordinates of vertex i in a basis for the two-dimensional space defined by the polygon plane.

4.2 Triangle-cell intersections

To calculate the area and volume fractions, we first need to find the intersection points between the triangle mesh and the hexahedral cell according to (5). What we basically want to do is to intersect a number of triangles and an axis-aligned hexahedron in a robust way. By robustness, we mean that we want to locate the points of intersection, we do not want to miss an intersection due to numerical imprecision, and we want related intersection tests to give consistent results. The latter requirement could fail for example if an edge that is shared between two triangles is not represented consistently between the two triangles [12]. We also want unique intersections, meaning that an intersection point on a triangle edge is shared between the two triangles meeting at the edge, and that an intersection point in a triangle vertex is shared between all triangles that meet in the vertex.

The above robustness criteria makes the triangle-box intersection problem hard to solve. Determining whether a triangle and an axis-aligned box intersect or not is a well-studied problem, discussed for example in [3]. Robustness issues are discussed by among others [12, 24]. It gets more complicated when the locations of the intersection points are also required. One approach to this is to perform a sequence of simpler intersection tests, where the triangle edges are intersected with the rectangular faces of the box, and the box edges are intersected with the triangle. Such line segment-rectangle and line segment-triangle intersection tests are common in several applications, often build upon the parametric representation of the geometric objects, and summarized for example in [12].

To get unique intersections, we have introduced vertices and edges that are shared between triangles. Similarly, we have introduced box vertices and box edges that are shared between the rectangular faces of the box. These joint edges and vertices are then used in a sequence of simpler intersection tests, as described above. Before the intersection points are calculated numerically, we, as far as possible, discretely determine how many intersections there are. This is done using coordinate comparisons similar to the outcodes described in [1, 2]. The coordinate comparison approach is appealing since the cell is axis-aligned, and the x-, y-, and z-coordinates can be

considered separately. Neither are there any numerical issues with the coordinate comparisons, since no floating point calculations have to be performed.

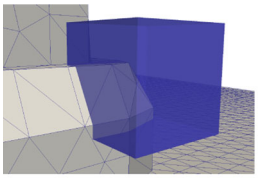
4.3 Exact method

In the exact method, we find the exact polyhedron of intersection $\overline{\Omega_{\mathscr{T}}} \bigcap \mathscr{C}$ (see Fig. 1b). We apply (1)–(2) and (7) to calculate the exact area and volume fractions of $\mathscr{C}$. The main steps in the method can be summarized as follows:

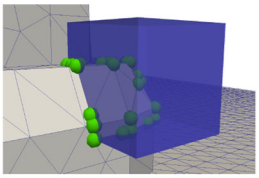
1. Find the intersection points $\mathscr{I}$ between the mesh $\mathscr{T}$ and the cell $\mathscr{C}$
2. Find the triangle vertices $\mathscr{I}_0$ of $\mathscr{T}$ inside the cell $\mathscr{C}$
3. For $i \in \{1, \ldots, 6\}$, connect intersection points in $\mathscr{I} \bigcap F_i$ to polygons Π_{ij}, and calculate the area fraction β_i as the total area of the polygons Π_{ij} divided by the area of the face F_i
4. For each triangle T_i intersecting the cell $\mathscr{C}$, connect $\left(\mathscr{I} \bigcup \mathscr{I}_0\right) \bigcap T_i$ to a cell polygon Π_{T_i}
5. The face and cell polygons Π_{ij} and Π_{T_i} in steps 3–4 define the polygonal faces of the polyhedron $\overline{\Omega_{\mathscr{T}}} \bigcap \mathscr{C}$. Calculate volume fraction of $\overline{\Omega_{\mathscr{T}}} \bigcap \mathscr{C}$ according to (1) and (7).

The intersection points in step 1 are given by (5), and the triangle vertices $\mathscr{I}_0$ in step 2 by (6). More details about the construction of polygons in steps 3–5 are found in Section 4.3.1. The main steps in the algorithm are illustrated in Fig. 5.

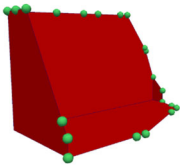
Pseudo code for an algorithm that computes the exact solid volume and area fractions is presented in Algorithm 1. In the algorithm, we introduce the notation $\mathscr{T}_{\parallel} = \mathscr{T}_{\parallel}(\mathscr{C})$ for the set of triangles $\mathscr{T}_{\parallel} \subseteq \mathscr{T}$ that are coplanar to some cell face



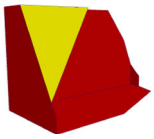
(a) A Cartesian cell $\mathscr{C}$ intersected by a triangle mesh $\mathscr{T}$.



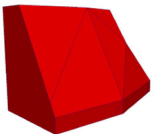
(b) All intersection points $\mathscr{I}$ between the mesh and cell are found as well as triangle vertices inside the cell, $\mathscr{I}_0$.



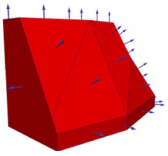
(c) For each face F_i of $\mathscr{C}$ the intersection points on that face, $\mathscr{I} \bigcap F_i$, are connected to face polygons.



(d) Each cell polygon Π_{T_i} is found by joining the intersection points on each intersecting triangle.



(e) When all intersecting triangles are processed the polyhedron P is complete.



(f) The volume fraction is calculated according to (1) and (7), where normals, areas and centroids are required.

Fig. 5 Description of how the polyhedron of intersection is found in the exact method

$F_k \subset \mathscr{C}$. We denote by $\mathscr{T}_\uparrow = \mathscr{T}_\uparrow(\mathscr{C})$ the set of triangles $\mathscr{T}_\uparrow \subseteq \mathscr{T}$, with $T_i \in \mathscr{T}_\uparrow$ satisfying $\mathscr{I} \bigcap T_i \subset F_k$ for some $k \in \{1, \ldots, 6\}$, and $\mathscr{I}_0 \bigcap T_i = \emptyset$. This means that an edge or vertex of $T_i \in \mathscr{T}_\uparrow$ touches a cell face, but the intersection of the triangle and the interior of the cell is empty.

The method normal = UnitNormal(polygon) returns a unit normal of the input polygon. The method UnitNormals(polygons) returns a list of normals, one for each polygon in polygons. The method Area(polygon) and Centroid(polygon) are implementations of (9) and (10), respectively. Centroids(polygons) calls Centroid(polygon) for each polygon in polygons. The method polygons = ConnectPolygons(intersection points, target facet) connects the points in points to polygons. It is described in Section 4.3.1.

Algorithm 1 $\{\alpha,\ \{\beta_i\}\} = \text{CalculateExactAreaAndVolumeFraction}(\mathscr{T}, \mathscr{C})$

Data: A triangle mesh $\mathscr{T}$ and a hexahedral cell $\mathscr{C}$.
Result: α: fraction of $\mathscr{C}$ inside $\mathscr{T}$.
$\{\beta_i\}$: fractions of F_i inside $\mathscr{C}$, where $F_i,\ i \in \{1, \ldots, 6\}$, are the faces of $\mathscr{C}$.

$\mathscr{I} \leftarrow \bigcup_{i=1}^{n_T} \bigcup_{j=1}^{6} \partial_0 \left(T_i \bigcap F_j\right)$
$\mathscr{I} \leftarrow \mathscr{I} \bigcap (\mathscr{T} \setminus (\mathscr{T}_\parallel \bigcup \mathscr{T}_\uparrow)$
$\mathscr{I}_0 \leftarrow \Omega_{\mathscr{C}} \bigcap \bigcup_{i=1}^{n_v} \mathbf{v}_i$
$\alpha \leftarrow 0$
for $i \in \{1, \ldots, 6\}$ **do**
  $\{\Pi_{ij}\} \leftarrow \text{ConnectPolygons}(\mathscr{I} \bigcap F_i, F_i)$
  $\{\hat{\mathbf{n}}_{ij}\} \leftarrow \text{UnitNormals}(\{\Pi_{ij}\})$
  $\{\mathbf{c}_{ij}\} \leftarrow \text{Centroids}(\{\Pi_{ij}\})$
  $\beta_i \leftarrow \sum_j \text{Area}(\Pi_{ij})/\text{Area}(F_i)$
  $\alpha \leftarrow \alpha + \sum_j \hat{\mathbf{n}}_{ij} \cdot \mathbf{c}_{ij} * \text{Area}(\Pi_{ij})$
for $T_i \in \mathscr{T} : (\mathscr{I} \bigcup \mathscr{I}_0) \bigcap T_i \neq \emptyset \wedge \neg \left(T_i \in \mathscr{T}_\parallel\right) \wedge \neg \left(T_i \in \mathscr{T}_\uparrow\right)$ **do**
  $\Pi_{T_i} \leftarrow \text{ConnectPolygons}\left((\mathscr{I} \bigcup \mathscr{I}_0) \bigcap T_i, T_i\right)$
  $\hat{\mathbf{n}}_{T_i} \leftarrow \text{UnitNormal}(\Pi_{T_i})$
  $\mathbf{c}_{T_i} \leftarrow \text{Centroid}(\Pi_{T_i})$
  $\alpha \leftarrow \alpha + \hat{\mathbf{n}}_{T_i} \cdot \mathbf{c}_{T_i} * \text{Area}(\Pi_{T_i})$
$\alpha \leftarrow \alpha/3$

In the algorithm, we do not create cell polygons from triangles in $\mathscr{T}_\parallel$ or $\mathscr{T}_\uparrow$, and we remove intersection points in $\mathscr{T} \setminus (\mathscr{T}_\parallel \bigcup \mathscr{T}_\uparrow)$. We do not create cell polygons from coplanar triangles to avoid that they contribute with the same area both to cell and face polygons, which would result in an error in (7). Since we do not create cell polygons from coplanar triangles, the coplanar triangles have to be accounted for in the face polygons. Instead of resolving every coplanar triangle in the face polygon algorithm, we remove from $\mathscr{I}$ the set $\{x \in \mathscr{I} : x \notin \mathscr{T} \setminus \mathscr{T}_\parallel\}$; that is the intersection

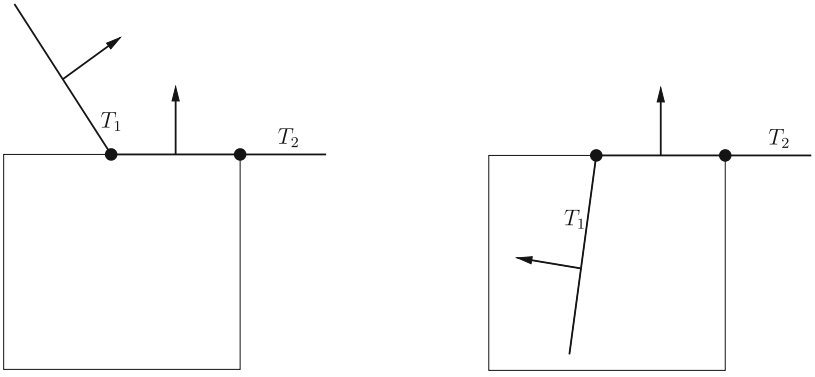
points that are only in coplanar triangles. If a cell face F_k has no intersection points after the removal, we know that the area fraction of F_k is 0.0 or 1.0. To find out which, we check if a point $x \in F_k$ is on the inside or outside of $\mathscr{T}$. This is done using a method based on angle weighted pseudo normals, similar to that of Bærentzen and Aanæs [6].

Since we do not consider triangles in $\mathscr{T}_{\parallel}$, we must also disregard triangles in $\mathscr{T}_{\uparrow}$. To see why, consider Fig. 6a, where a cell intersected by a triangulation $\mathscr{T}$ including triangles $T_1 \in \mathscr{T}_{\uparrow}$ and $T_2 \in \mathscr{T}_{\parallel}$ is seen in cross section. One of the intersection points marked by spheres belongs to both T_1 and T_2, and the other one only belongs to T_2. The intersection point that is only in T_2 is removed since $T_2 \in \mathscr{T}_{\parallel}$. Assuming symmetry, the area fraction of the top cell face is 1.0. Not removing the intersection points in T_1 would erroneously give an area fraction less than 0.5. Therefore, we also remove from $\mathscr{I}$ the set $\{x \in \mathscr{I} : x \notin \mathscr{T} \setminus (\mathscr{T}_{\parallel} \bigcup \mathscr{T}_{\uparrow})\}$, the points that are only in coplanar triangles or triangles pointing out of the cell.

As seen in Fig. 6b, we can not remove all $x \in \mathscr{I} \bigcap \mathscr{T}_{\parallel}$. Again assuming symmetry, the top area fraction is now approximately 0.6, since there is a face polygon that in cross section is the line between the two intersection points. We need to keep the intersection point that belongs to both T_1 and T_2 to construct this polygon. The difference to Fig. 6a where we removed both intersection points is that T_1 now does not belong to $\mathscr{T}_{\uparrow}$ but points into the cell.

4.3.1 Connecting intersection points to polygons

In this section, we describe how the intersection points in $\mathscr{I} \bigcup \mathscr{I}_0$ are connected to polygons. It is critical that the intersection points are processed in correct order so that (9)–(10) can be used to calculate area and centroid. As in [11], the connectivity of the triangle mesh and the cell is used to achieve this. While [11] briefly states that



(a) $T_1 \in \mathscr{T}_{\uparrow}$ and $T_2 \in \mathscr{T}_{\parallel}$.

(b) $T_2 \in \mathscr{T}_{\parallel}$. Since $T_1 \notin \mathscr{T}_{\uparrow}$ the intersection point that is both in T_1 and T_2 can not be removed.

Fig. 6 Cross sections of a cell intersected by a triangulation $\mathscr{T}$ including triangles T_1 and T_2

the face polygons are found through connectivity criteria such as common faces or common triangles, we describe how this is done.

There are two types of polygons that need to be created. The first type is the cell polygon, which is the intersection between a triangle T_i and $\overline{\Omega_{\mathscr{T}}}$. The second type is the face polygon that for each cell face F_i, $i \in \{1, \ldots, 6\}$, has vertices $\mathscr{I} \bigcap F_i$. While [1, 2] uses the Sutherland-Hodgman algorithm [25] to clip each triangle T_i against the cell faces to create the cell polygons Π_{T_i}, we use the connectivity to connect all points in $\left(\mathscr{I} \bigcup \mathscr{I}_0\right) \bigcap T_i$. This will be convenient when detecting double surfaces (see Section 5.1.1). To create the face polygons $\{\Pi_{ij}\}$, we use the connectivity to connect all points in $\mathscr{I} \bigcap F_i$.

Both types of polygons can be constructed in a similar way, which can be described after introducing target facets (F^T) and search facets (F^S). A target facet is the triangle T_i when the cell polygon Π_{T_i} is constructed, and the cell face F_i when the face polygons $\{\Pi_{ij}\}$ are constructed. The goal is to connect all intersection points on the target facet to one or more polygons. To do this, we need search facets. A search facet is a triangle, a triangle edge, a cell face, or a cell edge in which to search for the next vertex of the polygon that is under construction.

Given a correct current search facet, there should always be a uniquely determined intersection point to take as next polygon vertex. The main idea is to find the next vertex on the current search facet, and then change search facet. The next search facet is determined by the location of the newly added vertex with respect to the triangle mesh and the cell. This procedure, taking the next vertex and updating the search facet, is repeated until the polygon is completed, and until all intersection points on the target facets have been assigned to a polygon. Examples are given in Figs. 7 and 8, where a simple face polygon and a simple cell polygon are constructed.

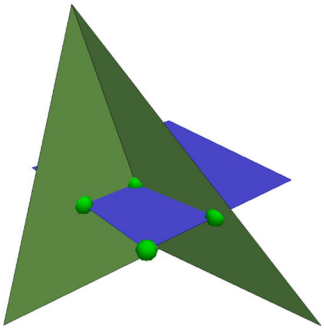
In Algorithm 1, the recently described polygon connection step is performed by the method ConnectPolygons(intersection points, target facet). It is given in pseudo code in Algorithm 3 (see Appendix A.1).

4.4 Approximate method

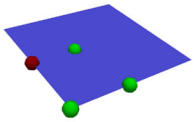
In the approximate method, $\overline{\Omega_{\mathscr{T}}} \bigcap \mathscr{C}$ is approximated by a convex polyhedron P with convex polygonal faces. P is defined by a least squares plane fitted to the points in $\mathscr{I} \bigcup \mathscr{I}_0$ and a normal of the plane. The plane splits $\mathscr{C}$ in two parts, P and $\mathscr{C} \setminus P$. The main steps in the method can be summarized as follows:

1. Find the intersection points $\mathscr{I}$ between the mesh $\mathscr{T}$ and the cell $\mathscr{C}$
2. Find the triangle vertices $\mathscr{I}_0$ of $\mathscr{T}$ inside the cell $\mathscr{C}$
3. Fit a least squares plane π to the intersection points $\mathscr{I}$ and the vertices found in steps 1–2
4. Identify P and $C \setminus P$
5. For $i \in \{1, \ldots, 6\}$, calculate the area of $P \bigcap F_i$ and the area fraction β_i
6. Calculate the volume of P and the volume fraction α.

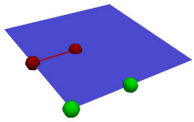
We now comment on the details of this procedure. The intersection points in step 1 are defined by (5), and the triangle vertices $\mathscr{I}_0$ in step 2 by (6). In step 3, a total



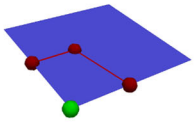
(a) Two triangles are intersecting the cell face - find the face polygon Π.



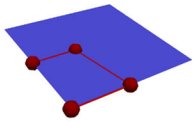
(b) The intersection point on the left face side is taken as start vertex v_0. Current search facet F^S is the left triangle.



(c) Take next vertex in F^S, it is on a triangle edge. Let F^S be the second triangle that contains the edge.



(d) F^S is the right triangle, take next vertex in F^S. New current intersection point is on a face side. Let F^S be that face side.



(e) Take next vertex in F^S. New intersection point in face corner, let F^S be the second face side that has the corner. Taking the next vertex in F^S closes the polygon.

Fig. 7 Description of how a face polygon is created according to Algorithm 3

least squares plane π is fitted to $\mathscr{I} \bigcup \mathscr{I}_0$. The plane with minimal total orthogonal squared distance to m points $\{\mathbf{p}_i\}_{i=1}^m$ in $\mathbb{R}^3$ satisfies

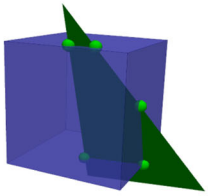
$$\min_{||\mathbf{n}||=1} \sum_{i=1}^{m} ((\mathbf{p}_i - \mathbf{p}_0) \cdot \mathbf{n})^2, \tag{11}$$

where $\mathbf{p}_0$ is a point in the plane and $\mathbf{n}$ is the plane normal. We can take [13]

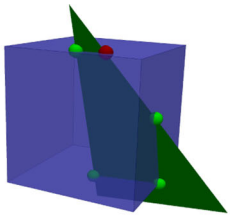
$$\mathbf{p}_0 = \frac{1}{m} \sum_{i=1}^{m} \mathbf{p}_i. \tag{12}$$

Let A be the matrix

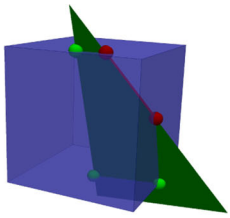
$$A = \begin{bmatrix} \mathbf{p}_1 - \mathbf{p}_0 \\ \vdots \\ \mathbf{p}_m - \mathbf{p}_0 \end{bmatrix}. \tag{13}$$



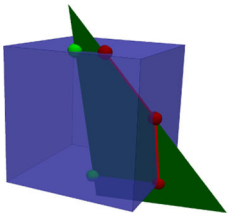
(a) A triangle T_i intersecting a cell gives a cell polygon Π_{T_i}.



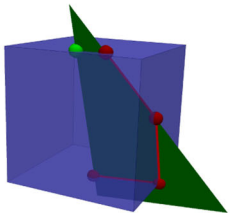
(b) The top right intersection point is taken as start vertex v_0. Let F^S be the triangle edge in which v_0 is included.



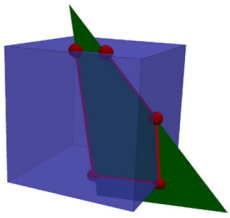
(c) Take next vertex in F^S. New current vertex is in a face, let F^S be that face.



(d) Take next vertex in F^S. New vertex is in a face side, let F^S be the second cell face that contains the side.



(e) Take next vertex in F^S. New vertex is in a triangle edge, let F^S be that triangle edge.



(f) Take next vertex in F^S. New vertex is in a face, let F^S be that face. Taking the next vertex in F^S closes the polygon.

Fig. 8 Description of how a cell polygon is created according to Algorithm 3

Then, (11) can be written

$$\min_{||\mathbf{n}||=1} ||A\mathbf{n}||^2, \tag{14}$$

where $||\cdot||$ denotes the Euclidean norm. The minimum can be found by computing the singular value decomposition

$$A = USV^T \tag{15}$$

of A. One then finds the minimum $\mathbf{n}$ as the third column of V. To see this, note that

$$||A\mathbf{n}||^2 = ||USV^T\mathbf{n}||^2 = ||SV^T\mathbf{n}||^2 = ||S\mathbf{z}||^2 = \lambda_1 z_1^2 + \lambda_2 z_2^2 + \lambda_3 z_3^2, \tag{16}$$

where $\lambda_1 \geq \lambda_2 \geq \lambda_3$ are the squared singular values of A, and $\mathbf{z} = V^T\mathbf{n}$. The second equality in (16) follows since U is an orthogonal matrix, and the last equality since $S(i,\ i) = \sqrt{\lambda_i}$ and $S(i,\ j) = 0,\ i \neq j$. The expression in (16) is thus minimized when $\mathbf{z} = \mathbf{e}_3 = (0,\ 0,\ 1)$, or equivalently when

$$\mathbf{n} = V\mathbf{e}_3, \tag{17}$$

the third column of V.

In step 4, the correct sign of the plane normal $\mathbf{n}_\pi$ is critical to determine what is P and what is $\mathscr{C} \setminus P$. One heuristic to determine the sign is to compare $\mathbf{n}_\pi$ to the area weighted average $\mathbf{n}_{\text{aw}}$ of all normals of triangles intersecting the cell,

$$\mathbf{n}_{\text{aw}} = \sum_{T_i \bigcap \mathscr{C} \neq \emptyset} A(T_i)\, \hat{\mathbf{n}}_i, \tag{18}$$

where $\hat{\mathbf{n}}_i$ is the unit normal of T_i. If $\mathbf{n}_{\text{aw}} \cdot \mathbf{n}_\pi > 0$, the sign of $\mathbf{n}$ is assumed correct; otherwise, it is changed. A second alternative is to replace $A(T_i)$ by $A(T_i \bigcap \mathscr{C})$ in (18). This will have advantages when the mesh includes double surfaces (see Section 5), but is more expensive since $T_i \bigcap \mathscr{C}$ has to be found. In addition, $T_i \bigcap \mathscr{C}$ will in general have a more complex shape than T_i, which makes the area calculation more expensive.

For steps 5–6, we need the cell and face polygons of P (see Fig. 3). The only cell polygon Π is convex and defined by

$$\Pi = \overline{\partial P \bigcap \Omega_{\mathscr{C}}}. \tag{19}$$

It is constructed from the points in

$$\mathscr{I}_\pi = \bigcup_{i=1}^{12} \partial_0(\pi \bigcap E_i), \tag{20}$$

the intersection between the plane π and the edges of $\mathscr{C}$. The face polygon on cell face F_i is denoted by Π_i, and defined by

$$\Pi_i = P \bigcap F_i. \tag{21}$$

It is constructed by connecting the points in $\mathscr{I}_\pi \bigcap F_i$ with the cell vertices $\{V_j\}_{j=1}^{8}$ on the correct side of the plane π. The correct side is determined by the plane normal $\mathbf{n}_\pi$. What remains is to calculate the volume of the polyhedron according to (7) and divide it by the cell volume to get the volume fraction α. The face area fraction β_i is given by the area of Π_i divided by the area of F_i.

The main steps in the algorithm are illustrated in Fig. 9. Pseudo code for an algorithm that computes P and hence the solid volume and area fractions of $\mathscr{C}$ is described in Algorithm 2. The method plane = LeastSquaresPlane(points) on line 3 is an implementation of the least squares plane algorithm described in (11)–(17). The method CorrectNormal(plane, triangle mesh, cell) on line 4 corrects the normal $\mathbf{n}_\pi$ of the input plane according to the sign of $\mathbf{n}_{\text{aw}} \cdot \mathbf{n}_\pi$ as described above, where $\mathbf{n}_{\text{aw}}$ is given by (18). The method FindCellVerticesInPolyhedron(cell, plane) returns a list of the vertices of the input cell on the correct side of the input plane, where the correct side is determined by the plane normal. The method normal = UnitNormal(polygon) returns a unit normal of the input polygon. The methods area = Area(polygon) and centroid = Centroid(polygon) are implementations of (9) and (10), respectively. Finally, polygon = ConvexPolgon(points, normal) takes as input a list of polygon vertices and the normal of the polygon, and returns a list of the vertices sorted in clockwise or counterclockwise order. Pseudo code for this method is presented in Algorithm 11 in Appendix A.2.

Algorithm 2 $\{\alpha,\ \{\beta_i\}\} = \text{CalculateApproximateAreaAndVolumeFraction}(\mathscr{T}, \mathscr{C})$

Data: A triangle mesh $\mathscr{T}$ and a hexahedral cell $\mathscr{C}$.
Result: α: approximate fraction of $\mathscr{C}$ inside $\mathscr{T}$.
$\{\beta_i\}$: approximate fractions of F_i inside $\mathscr{C}$, where $F_i,\ i \in 1, \ldots, 6$, are the faces of $\mathscr{C}$.

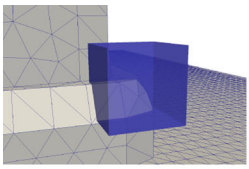
$\mathscr{I} \leftarrow \bigcup_{i=1}^{n_T} \bigcup_{j=1}^{6} \partial_0 \left(T_i \bigcap F_j\right)$
$\mathscr{I}_0 \leftarrow \Omega_{\mathscr{C}} \bigcap \bigcup_{i=1}^{n_v} \mathbf{v}_i$
$\pi \leftarrow \text{LeastSquaresPlane}\left(\mathscr{I} \bigcup \mathscr{I}_0\right)$
CorrectNormal(π, $\mathscr{T}$, $\mathscr{C}$)
$\mathscr{I}_\pi \leftarrow \bigcup_{i=1}^{12} \partial_0(\pi \bigcap E_i)$
$\mathscr{I}_V \leftarrow \text{FindCellVerticesInPolyhedron}(\mathscr{C}, \pi)$
for $i \in \{1, \ldots, 6\}$ **do**
  $\hat{\mathbf{n}}_i \leftarrow \text{UnitNormal}(\Pi_i)$
  $\Pi_i \leftarrow \text{ConvexPolygon}(\mathscr{I}_\pi \bigcup (\mathscr{I}_V \bigcap F_i),\ \hat{\mathbf{n}}_i)$
  $\mathbf{c}_i \leftarrow \text{Centroid}(\Pi_i)$
  $\beta_i \leftarrow \text{Area}(\Pi_i)/\text{Area}(F_i)$
$\hat{\mathbf{n}} \leftarrow \text{UnitNormal}(\pi)$
$\Pi \leftarrow \text{ConvexPolygon}(I_\pi,\ \hat{\mathbf{n}})$
$\mathbf{c} \leftarrow \text{Centroid}(\Pi)$
$\alpha \leftarrow 1/3 * \left[\hat{\mathbf{n}} \cdot \mathbf{c} * \text{Area}(\Pi) + \sum_{i=1}^{6} \hat{\mathbf{n}}_i \cdot \mathbf{c}_i * \text{Area}(\Pi_i)\right]$

5 Geometric complications

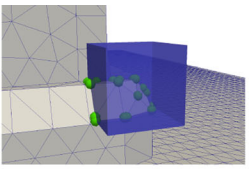
The algorithms described in Section 4 calculate solid area and volume fractions for a non-degenerate triangle mesh. In practice, it is not unusual that the mesh is degenerate in some sense. It can include T-vertices (hanging nodes), gaps, and cracks, or consist of overlapping meshes. The latter leads to what we call *double surfaces*, the intersection of overlapping triangles. In this section, it is described how the algorithms in Section 4 are modified to handle double surfaces with normals of the overlapping triangles pointing in opposite directions. It is also shown that the algorithms handle multiple meshes and split hexahedra. These are steps towards a method that calculates the solid area and volume fractions of a degenerate triangle mesh including overlapping triangles, overlapping meshes, hanging nodes, and gaps. Such triangle meshes are common in industrial applications.

5.1 Double surfaces

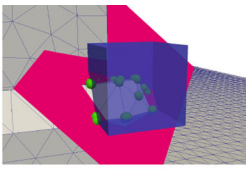
Recall the definition of a double surface from Section 3. In Fig. 2, we saw a double surface including triangle normals with opposite directions. In Fig. 10, there is another, schematic, example of a mesh including a double surface. Two typical cases of intersection between a cell and the double surface are seen in Fig. 11.



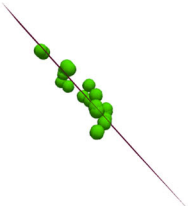
(a) A Cartesian cell $\mathscr{C}$ intersected by a triangle mesh $\mathscr{T}$.



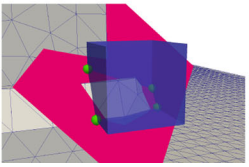
(b) The intersection points $\mathscr{I}$ between the mesh and cell are found as well as triangle vertices inside the cell, $\mathscr{I}_0$.



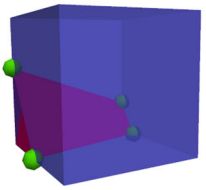
(c) A least squares plane π is fitted to the intersection points in $\mathscr{I} \bigcup \mathscr{I}_0$.



(d) The least squares plane π and the intersection points in $\mathscr{I} \bigcup \mathscr{I}_0$ from another view angle.



(e) The intersection points $\mathscr{I}_\pi$ between the plane and the cell are located.



(f) The approximate polyhedron P is defined by $\mathscr{I}_\pi$ and the normal of π.

Fig. 9 Description of how the polyhedron of intersection is found in the approximate method

When $\mathscr{T}_1$ and $\mathscr{T}_2$ are handled as separate meshes both algorithms correctly accounts for the double surface as explained in Section 5.2. To solve the degenerate case when the two meshes are treated as one, the algorithms are modified as described below. In the rest of the section, we consider double surfaces on a single mesh $\mathscr{T} = \mathscr{T}_1 \bigcup \mathscr{T}_2$.

5.1.1 Exact method

The exact method in Section 4.3 is modified to handle both cases in Fig. 11. It is not necessary that the double surface is axis-aligned. The cell polygons are unproblematic since a cell polygon is created by connecting all intersection points in a particular triangle. The contribution to the volume fraction from the triangles in the double surface will cancel out in (7). Possible problems occur when the face polygons are

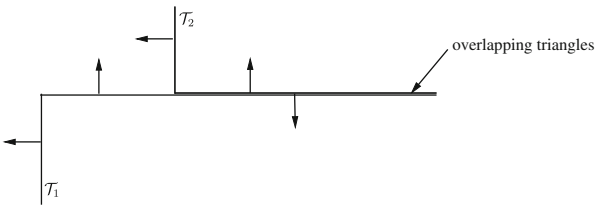
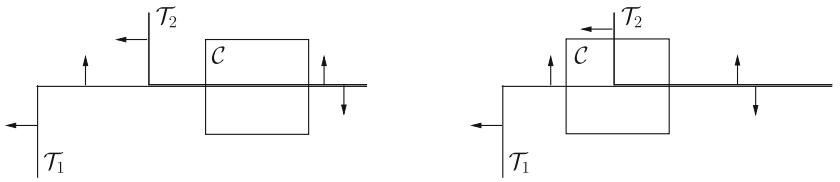


Fig. 10 Cross section of two triangle meshes $\mathscr{T}_1$ and $\mathscr{T}_2$ glued together. The *arrows* represent triangle normals pointing out of the interior of the respective mesh. The overlapping triangles give rise to a double surface



(a) A cell $\mathscr{C}$ intersecting a double surface.

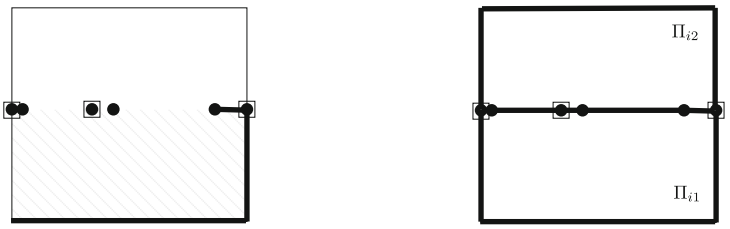
(b) A cell $\mathscr{C}$ intersecting both a double surface and another part of the mesh.

Fig. 11 Cross section of typical cases of intersection between a cell and a double surface

created. Then, it has to be made sure that intersection points from the underlying meshes $\mathscr{T}_1$ and $\mathscr{T}_2$ are connected to separate polygons. The solution is to use the connectivity of $\mathscr{T}$ to extract information about the underlying meshes $\mathscr{T}_1$ and $\mathscr{T}_2$, so that the two parts of $\mathscr{T}$ can be handled separately.

Figure 12 shows a cell face F_i and intersection points $\mathscr{I} \bigcap F_i$ from Fig. 11a. The intersection points are marked by spheres and boxes. The spheres represent $\mathscr{I} \bigcap F_i \bigcap \mathscr{T}_1$, and the boxes represent $\mathscr{I} \bigcap F_i \bigcap \mathscr{T}_2$. The whole cell is inside, so the face polygons $\{\Pi_{ij}\}$ should cover F_i. In Fig. 12b, the points in $\mathscr{I} \bigcap F_i \bigcap \mathscr{T}_1$ have been connected to one polygon Π_{i1} and the points in $\mathscr{I} \bigcap F_i \bigcap \mathscr{T}_2$ to a second polygon Π_{i2}, where $F_i = \Pi_{i1} \bigcup \Pi_{i2}$ as expected.

When the search facet is a triangle, there is no ambiguity in the choice of the next polygon vertex. When the next intersection point is searched on a face side, as in Fig. 12a, the presence of the double surface becomes apparent. Then, it has to be made sure that only intersection points from the correct part of $\mathscr{T}$ can be taken. Normally, the next intersection point on the face side is the one closest to and on the correct side of the current intersection point (see Algorithm 7 in Appendix A.1). At a double surface, there are two intersection points at the same position. The solution



(a) A polygon Π_{i1} is to be created. The next intersection point is in the leftmost face side. There are two alternative intersection points at the same location. The only correct alternative is to take the sphere, which belongs to $\mathscr{T}_1$.

(b) The intersection points marked by spheres are correctly connected to a polygon Π_{i1}, and the intersection points marked by boxes to another polygon Π_{i2}. As expected, $\Pi_{i1} \bigcup \Pi_{i2} = F_i$.

Fig. 12 Face polygon connection at a double surface. Intersection points marked by *spheres* come from $\mathscr{T}_1$ and intersection points marked by *boxes* come from $\mathscr{T}_2$

is to make sure that an intersection point in a triangle with normal pointing against the current intersection point cannot be taken. This is a check that the polygon that is created is actually located on the inside of the underlying mesh $\mathscr{T}_1$ or $\mathscr{T}_2$, since the triangle normals point of $\Omega_{\mathscr{T}_1}$ and $\Omega_{\mathscr{T}_2}$ (at a double surface the surface normal of $\mathscr{T}$ is undefined). To account for the presence of double surfaces, Algorithm 7 has to be modified according to this observation. The result is presented in Algorithm 10 in Appendix A.1.

5.1.2 Approximate method

To see that the approximate method in Section 4.4 is inaccurate for meshes with double surfaces, consider Fig. 11a. The cell is intersected by a double surface, which will make all intersection points lie in a plane in the middle of the cell. The unmodified algorithm takes this plane as the approximating plane and gives a volume fraction around 0.5, though the whole cell is inside. The solution is to check if all normals of triangles intersecting the cell point in the same or opposite direction, and if the triangles lie in the same plane. The volume fraction is then set to 1.0.

In Fig. 11b, a cell is intersected both by a double surface and a regular part of $\mathscr{T} = \mathscr{T}_1 \bigcup \mathscr{T}_2$. The intersection points at the double surface will erroneously affect the plane. This problem is not handled as a special case in the approximate method.

As pointed out in Section 4.4, it is better to use the normal weighted with the area of the part of a triangle inside the cell than the area of the whole triangle when determining the sign of the least squares plane normal. To see an example of why, consider Fig. 13, where a cross section of a cell intersected by a triangle mesh including triangles T_1^1, T_2^1, and T_1^2 is seen. Parts of T_2^1 and T_1^2 overlap in a double surface. The dashed line and normal represent the approximating plane π in the approximate method. Now assume the area of T_2^1 is much larger than that of T_1^1 and T_1^2. The comparison normal can then be approximated by that of T_2^1, and we would erroneously take the wrong sign for the normal of π by the sign test of the dot product. If we weight with the area inside the cell, the double surface will be canceled out and will not affect the comparison normal.

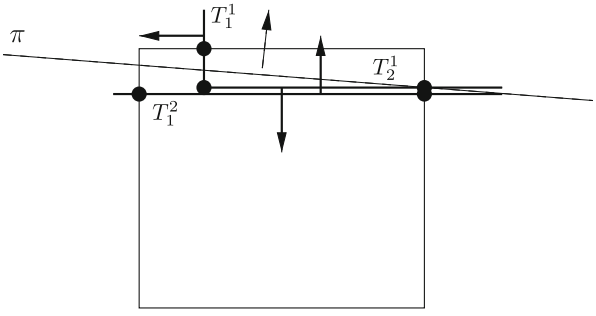


Fig. 13 Cross section of cell intersected by a double surface created by T_2^1 and T_1^2. The approximating plane π in the approximate method is marked by a *dashed line*

5.2 Several meshes

Both algorithms in Section 4 can be applied on multiple meshes $\mathcal{T}_1, \ldots, \mathcal{T}_N$ by considering each $\mathcal{T}_j$ separately and summing the area fractions β_{ij}, $i \in \{1, \ldots, 6\}$, and volume fraction α_j from each mesh according to

$$\alpha = \sum_{j=1}^{N} \alpha_j, \tag{22}$$

and

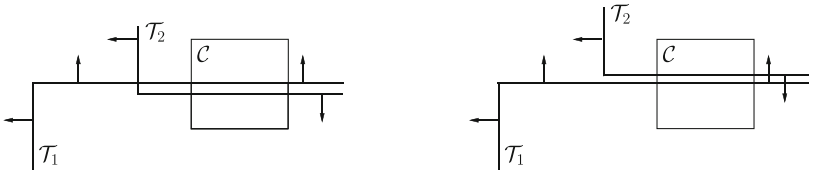
$$\beta_i = \sum_{j=1}^{N} \beta_{ij}. \tag{23}$$

5.3 Small self-intersections or overlaps

When floating points are used, numerical issues could make the triangles in a double surface slightly overlap or be slightly separated, as in Fig. 14. The proposed algorithms work even if the triangles in the double surface are slightly separated and not overlapping. However, an extra condition has to be added if small overlaps are to be handled.

A limit ε for the maximal overlap allowed can be introduced and used to indicate when a double surface is found. Two intersection points at a separation distance smaller than ε are assumed to lie on a double surface if they also belong to triangles with opposite normals. Due to numerical issues, it will be necessary to have a limit on the oppositeness of normals. In the following, we assume that two normals are opposite if $\pi - \theta < \delta$, where θ is the angle between the normals and δ is a prescribed limit.

In the approximate method, it is enough to use the limits to determine when two triangles with opposite normals are coplanar. In the exact method, the limits are used in the face polygon connection step to indicate whether a certain intersection point can be taken as the next vertex in the face polygon or not. In the following, we discuss the treatment of double surfaces in the exact algorithm.



(a) A cell $\mathscr{C}$ intersecting an overlapping double surface.

(b) A cell $\mathscr{C}$ intersecting a double surface with a small separation between the triangles.

Fig. 14 Cross section of typical cases of intersection between a cell and a double surface. The double surface includes small overlaps or small separations between triangles

When a double surface intersects a cell, as in Fig. 11, we wish to connect the intersection points on $\mathscr{T}_1$ to one polyhedron of intersection, and the intersection points on $\mathscr{T}_2$ to another polyhedron of intersection as described in Section 5.1. The result after a correct polygon connection step is visualized in Fig. 15, where the different striped patterns correspond to the different polyhedrons of intersection, one for $\mathscr{T}_1$ and one for $\mathscr{T}_2$.

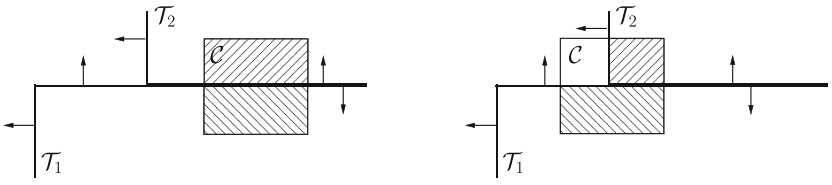
For an overlapping double surface, the goal is to get to the result in Fig. 16. The two meshes $\mathscr{T}_1$ and $\mathscr{T}_2$ still contribute with one polyhedron of intersection each, but there will now be an overlap between the polyhedrons. This overlap corresponds to the overlap of the double surface. The area and volume of the overlap is counted twice in the solid area and solid volume fraction calculations. To get to the result in Fig 16, the face polygon connection step has to be modified, or the result will. be as in Fig. 17 where only a fraction of the correct polyhedrons of intersection is found.

The problem with the cases in Fig. 17 is that the algorithm cannot distinguish the overlapping double surface from a thin or sharp-edged part of the mesh. Examples of thin and sharp-edged meshes are found in Fig. 18a and b, respectively. A sharp edge is in this case formed when two triangles meet at an angle less than δ. For the cases in Fig. 18, it is correct to connect the two close intersections at the left and right cell faces, while it is wrong to do the same thing for the cases in Fig. 17.

To avoid the problem demonstrated in Fig. 17, the maximal overlap limit ε is used as an upper limit on how close two nearby vertices of a face polygon can be. If the distance between the current vertex and a candidate for the next vertex is closer than ε, and the two vertices belong to triangles with opposite normals; the candidate can not be taken as the next vertex. With this modification of the polygon connection algorithm, the result will be as in Fig. 16. A consequence is that sharp-edged meshes and meshes thinner than ε cannot be handled.

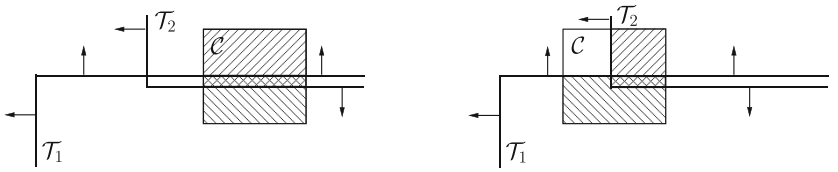
The above modification of the exact algorithm is necessary only if the double surface belongs to a single mesh $\mathscr{T} = \mathscr{T}_1 \bigcup \mathscr{T}_2$. If $\mathscr{T}_1$ and $\mathscr{T}_2$ are handled separately, the double surface is resolved by finding the face polygons and polyhedrons of intersection for one mesh at a time as described in Section 5.2. The area and volume fractions from the different meshes are then added.

For separate meshes $\mathscr{T}_1$ and $\mathscr{T}_2$, the overlaps that can be handled are not restricted to double surfaces. It is possible to handle arbitrary overlaps, as long as it is reasonable to count the overlap volume and area twice.



(a) A cell $\mathscr{C}$ intersecting a double surface. (b) A cell $\mathscr{C}$ intersecting both a double surface and another part of the mesh.

Fig. 15 Cross section of typical cases of intersection between a cell and a double surface. The intersections have been connected into the correct polyhedrons of intersection, one for $\mathscr{T}_1$ and one for $\mathscr{T}_2$



(a) A cell $\mathscr{C}$ intersecting a double surface. (b) A cell $\mathscr{C}$ intersecting both a double surface and another part of the mesh.

Fig. 16 Cross section of typical cases of intersection between a cell and an overlapping double surface. The intersections have been connected into the correct polyhedrons of intersection, one for $\mathscr{T}_1$ and one for $\mathscr{T}_2$

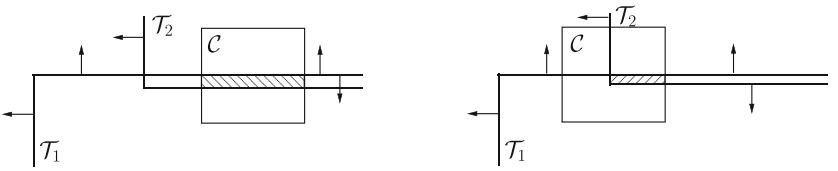
5.4 Split hexahedra

When the hexahedral grid is coarse or the geometry is thin or detailed, a cell can be cut in several disconnected parts by one or more polyhedrons of intersection. This leads to the formation of so called split cells, or split hexahedra. An example of a split cell in two dimensions is found in Fig. 19.

The exact method easily handles most cases of split hexahedra by construction. The cell polygons are treated triangle by triangle as described in Section 4.3.1 and in Algorithm 3, which is not complicated by split hexahedra. The face polygons are uniquely determined by the connectivity of the triangle mesh under the same assumptions of self-intersections as for double surfaces (see Section 5.3), and not either complicated by most forms of split hexahedra. The face polygon construction algorithm described in Section 4.3.1 and in Algorithm 3 finds as many polygons of intersection as needed until all intersection points have been assigned to a polygon.

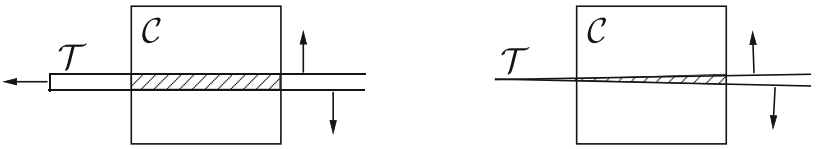
One case that is not covered by the algorithm described this far is when the triangle mesh intersects a cell face and forms one or more polygonal "holes," as in Fig. 20. This happens when a connected part of the triangle mesh intersects the face without intersecting the boundary of the face. The algorithm cannot determine what is the inside and what is the outside of the "hole," since the triangle normals are not used in the connection step.

To handle polygonal holes, signed polygon areas are used. The sign of the area is determined by the normals of the triangles that form the polygon. The area is positive



(a) A cell $\mathscr{C}$ intersecting a double surface. (b) A cell $\mathscr{C}$ intersecting both a double surface and another part of the mesh.

Fig. 17 Cross section of typical cases of intersection between a cell and a double surface. The intersections have in both cases been connected erroneously to only one polyhedron of intersection



(a) A cell $\mathscr{C}$ intersecting a thin mesh $\mathscr{T}$. (b) A cell $\mathscr{C}$ intersecting a sharp-edged mesh $\mathscr{T}$.

Fig. 18 Meshes that cannot be distinguished from an overlapping double surface in the method based on a maximal limit of the allowed overlap

if the normals point out of the polygon, and negative if the normals point into the polygon. To determine if the normals point into the polygon or not, a method similar to [6, 14] is used. The closest polygon vertex v to an arbitrary face edge is first found, the vertex is projected onto the edge, and the vector from the projection v_e to the vertex is formed. The sign of the dot product $(v_e - v) \cdot n_v$ determines if the triangle normals point into or out of the polygon according to

$$\text{sgn}(A) = \begin{cases} 1, & (v_e - v) \cdot n_v > 0, \\ -1, & (v_e - v) \cdot n_v < 0, \end{cases}$$

where sgn(A) is the sign of the polygon area and n_v is the average of the triangle normals of the triangles meeting in v. An illustration is found in Fig. 21.

If none of the face polygons intersect the boundary of the face, the area of the whole face has to be added to the total area of intersection if the "outermost" polygon is a hole. This is the case in Fig. 20b. The outermost polygon is found by locating the vertex closest to an arbitrary face edge, and identifying the corresponding polygon.

6 Numerical examples and results

Three test cases have been studied for evaluation of the algorithms. The cases were designed to test the algorithms on double surfaces, numerical or small overlaps, and

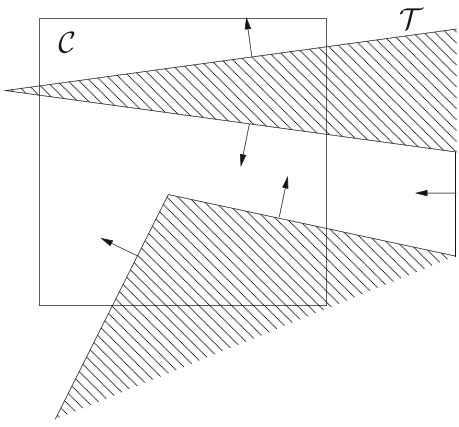
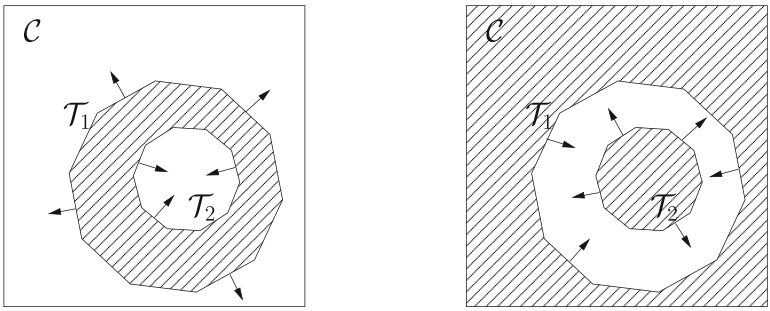


Fig. 19 The triangle mesh $\mathscr{T}$ splits the cell C into four parts. Two of the four parts are disconnected polyhedrons of intersection



(a) The inner polygon is a hole, and the outer is not. (b) The outer polygon is a hole, and the inner is not.

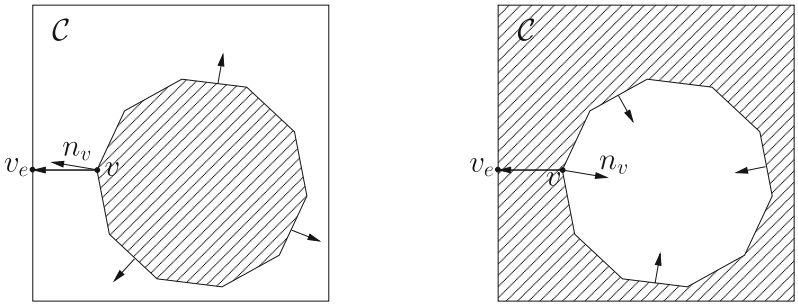
Fig. 20 Two triangle meshes $\mathscr{T}_1$ and $\mathscr{T}_2$ intersecting a cell $\mathscr{C}$ in a way that splits the hexahedral cell in three parts, without intersecting the boundary of any cell face

split hexahedra. In Section 6.1, double surfaces and split hexahedra is tested with a geometry representing a heat sink used for air cooling of CPU:s. In Section 6.2, we test the exact method for double surfaces and small overlaps on a geometry consisting of two cylinders whose surfaces should meet in a double surface, but the surfaces are not perfectly matching. In Section 6.3, the geometry consists of four cylinders placed inside each other. It is used to test the exact method for split hexahedra and holes.

The algorithms were implemented in the C++-based multiphase flow framework IBOFlow [16]. All computations were carried out on a machine equipped with a 3.50-GHz Intel Core i7 processor (5930K) and 64 GB of RAM (1.066 GHz DDR4).

6.1 Heat sink with a double surface

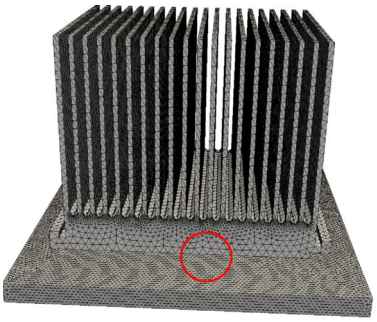
The exact method is validated and compared to the approximate method on a geometry including a double surface. The geometry represents a heat sink and is taken from an industrial application. The whole mesh is seen in Fig. 22a, and a zoom in on the part of the mesh that is marked by a circle is shown in Fig. 22b. The triangle pattern



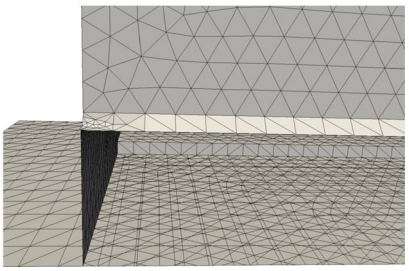
(a) The area is positive since n_v points out of the polygon.

(b) The area is negative since n_v points into the polygon.

Fig. 21 The sign of the dot product $(v_e - v) \cdot n_v$ is used to determine if the polygon is a hole or not



(a) Test geometry representing a heat sink.



(b) Zoom in on the part in the left figure marked by a circle. The overlapping triangles indicate that there is a double surface.

Fig. 22 The test case

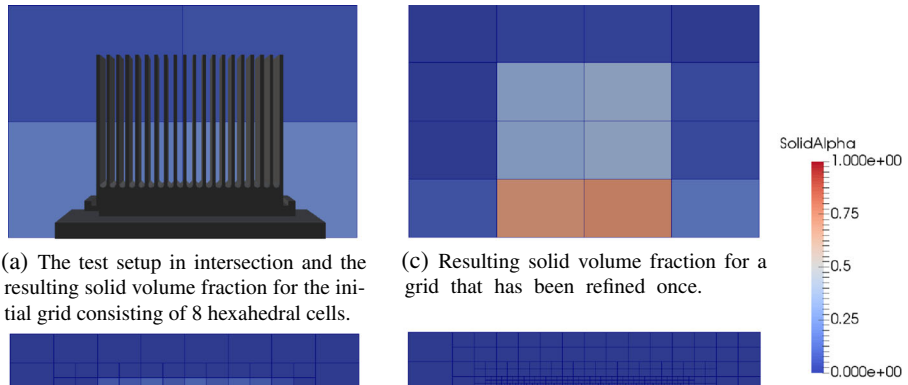
in the lower right part of the zoom in reveals that the geometry contains a double surface.

Grid convergence and CPU times were analyzed and compared. For the grid study, the fluid grid was stored in an octree and initially consisted of eight rectangular cells with sides $(\Delta x,\ \Delta y,\ \Delta z) = (0.0250\,\text{m},\ 0.0208\,\text{m},\ 0.0312\,\text{m})$. It was refined up to eight times around the triangular mesh, resulting in a finest grid with 3, 329, 947 cells of size $(\Delta x,\ \Delta y,\ \Delta z) = (9.77 \cdot 10^{-5}\,\text{m},\ 8.13 \cdot 10^{-5}\,\text{m},\ 1.22 \cdot 10^{-4}\,\text{m})$ or larger. At each refinement level, the smallest cell size was halved and eight new cells were formed. The test setup and the solid volume fractions for the base grid is seen in Fig. 23a. The solid volume fractions for grids with one, three, and six refinements are seen in Fig. 23b, c, d, respectively. The exact method was used in all four cases.

Results of the grid study are presented in Fig. 24a. The total volume of the interior of the geometry was calculated by running the exact and approximate volume fraction algorithms for each cell. This was repeated for each refinement level. The calculated volume was compared to the real volume $1.45 \cdot 10^{-5}\,\text{m}^3$, which was found by applying (7) to the whole triangle mesh.

A plot of the CPU time against the number of fluid cells is seen in Fig. 24b. The times are the average results from ten runs. A Cartesian grid with initial cell size $(\Delta x,\ \Delta y,\ \Delta z) = (4.16{\cdot}10^{-3}, 4.16{\cdot}10^{-3}, 4.16{\cdot}10^{-3}\,\text{m})$ was used for the time study. It was refined up to five times around the triangular mesh, resulting in a finest grid with 8, 438, 270 cells of size $(\Delta x,\ \Delta y,\ \Delta z) = (1.30 \cdot 10^{-4}\,\text{m},\ 1.30 \cdot 10^{-4}\,\text{m},\ 1.30 \cdot 10^{-4}\,\text{m})$ or larger. In summary, the exact method is independent of cell size, while the approximate method is second-order accurate. The exact method is a constant factor slower than the approximate method. The constant factor is close to 2.0 when the number of cells is large.

The grid study in Fig. 24a and the test setup in Fig. 25 indicate that the exact method handles split hexahedra well. In Fig. 25a, the initial grid has been refined three times, and several cells are split in two or more parts by the triangle mesh. One of the split cells is marked green. A zoom in on this cell and the polyhedrons



(a) The test setup in intersection and the resulting solid volume fraction for the initial grid consisting of 8 hexahedral cells.

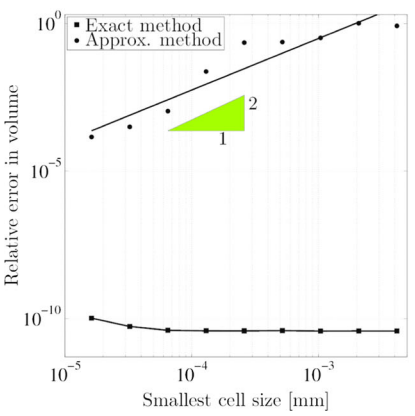
(c) Resulting solid volume fraction for a grid that has been refined once.

(b) Resulting solid volume fraction for a grid that has been refined three times.

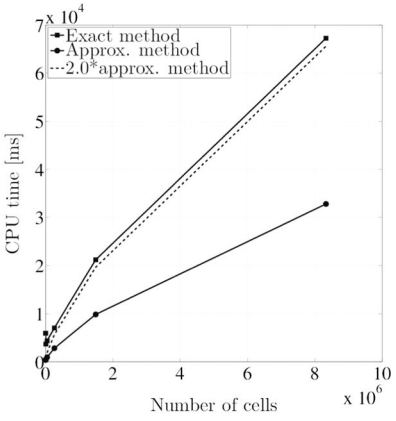
(d) Resulting solid volume fraction for a grid that has been refined six times.

Fig. 23 Test setup and resulting solid volume fraction for different number of refinements of the initial grid

of intersection are seen in Fig. 25b. The calculated volume of the two polyhedrons of intersection agrees well with the theoretical volume $9.51 \cdot 10^{-9}\,\mathrm{m}^3$. The thickness of the first fin is $6.15 \cdot 10^{-4}\,\mathrm{m}$ and the thickness of the second fin is $5.55 \cdot 10^{-4}\,\mathrm{m}$, extending in the z-direction. The dimensions of the cell are $(\Delta x,\ \Delta y,\ \Delta z) = (3.13 \cdot 10^{-3}\,\mathrm{m},\ 2.60 \cdot 10^{-3}\,\mathrm{m},\ 3.90 \cdot 10^{-3}\,\mathrm{m})$.

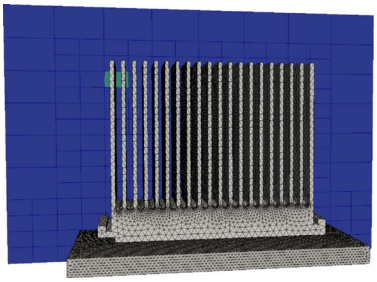


(a) Grid convergence of volume fraction algorithms run on the heat sink case.

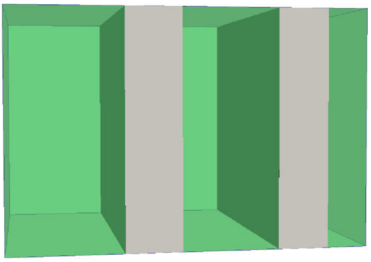


(b) CPU time of volume fraction algorithms run on the heat sink case.

Fig. 24 Results from grid and CPU time studies



(a) The test setup for three refinements. One cell that is split in several parts by the triangle mesh is colored green.

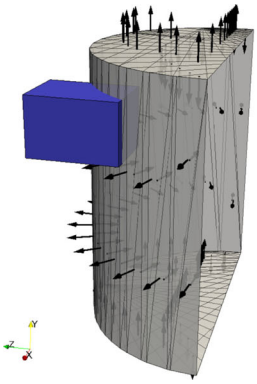


(b) The resulting polyhedrons of intersection that split the green cell in several parts.

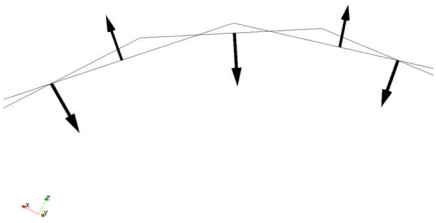
Fig. 25 The test setup for three refinements results in a number of split hexahedra

6.2 Overlapping cylinders

Two identical co-located cylinders of radius 0.5 m were triangulated, joined to one, and intersected by an axis-aligned hexahedral cell of dimensions $(\Delta x,\ \Delta y,\ \Delta z) = (0.23\,\text{m},\ 0.208\,\text{m},\ 0.312\,\text{m})$. The co-located cylinders simulate a double surface. Different triangulation schemes were used for the two cylinders so that the resulting triangle meshes slightly overlapped instead of forming a perfect double surface. The resulting setup is seen in Fig. 26. In Fig. 26a, it is seen how the hexahedron intersects the triangle meshes, and in Fig. 26b, we have zoomed in on parts of a cross section of the triangle meshes to show that the triangles overlap.



(a) Two overlapping cylinder triangle meshes are intersected by a hexahedral cell (blue). The arrows represent triangle normals pointing out of the interior of the meshes. Only half of the triangle meshes are shown.



(b) Part of a circular cross-section of the triangle meshes, which shows that the triangle meshes slightly overlap. The overlaps have been exaggerated in the figure.

Fig. 26 Setup that tests the algorithm for a double surface formed by slightly overlapping, non axis-aligned triangles

The algorithm handles the overlapping double surface according to Section 5.3, where $\varepsilon = 0.01\Delta x$ m and $\delta = \frac{2\pi}{45}$ rad is chosen. The polygons and polyhedrons of intersection are correctly connected, as demonstrated in Fig. 27. One of the polyhedrons is colored red and the other black, and they belong to one cylinder each. Both overlaps and gaps had to be handled to construct the face polygons. This is seen from the face polygons for face 2 (y-top) in Fig. 28. For visibility, the overlaps and gaps have been exaggerated in the figure. At the bottommost face edge, the factual overlap between the triangles is $0.002\Delta x$ m, which is within the tolerance ε. The angle θ between the triangle normals satisfies $\pi - \theta = 1.23 \cdot 10^{-1}$ rad, which is less than δ.

6.3 Split hexahedra

Four cylinders were placed one inside the other in a configuration that intersected an axis-aligned hexahedron in two different ways. The dimensions of the hexahedron were $(\Delta x,\ \Delta y,\ \Delta z) = (6 \cdot 10^{-3}\,\text{m},\ 10^{-2}\,\text{m},\ 10^{-2}\,\text{m})$. The radii of the cylinders were $r_1 = 3 \cdot 10^{-4}$ m, $r_2 = 9 \cdot 10^{-4}$ m, $r_3 = 1.5 \cdot 10^{-3}$ m, and $r_4 = 1.8 \cdot 10^{-3}$ m.

In the first setup, the centerline of each cylinder was aligned with the y-axis and passing through the center of the xz-faces of the hexahedron (Fig. 29). In the second setup, the cylinders were rotated $\frac{\pi}{4}$ rad about the x-axis from their initial position (Fig. 30). For these setups, the analytic volume fraction is to three significant figures $\alpha^1 = 8.95 \cdot 10^{-2}$, $\alpha^2 = 1.13 \cdot 10^{-1}$, and the area fractions are

$$\beta_i^1 = \begin{cases} 0, & i = 0,\ 1,\ 4,\ 5, \\ 8.95 \cdot 10^{-2}, & i = 2,\ 3, \end{cases} \tag{24}$$

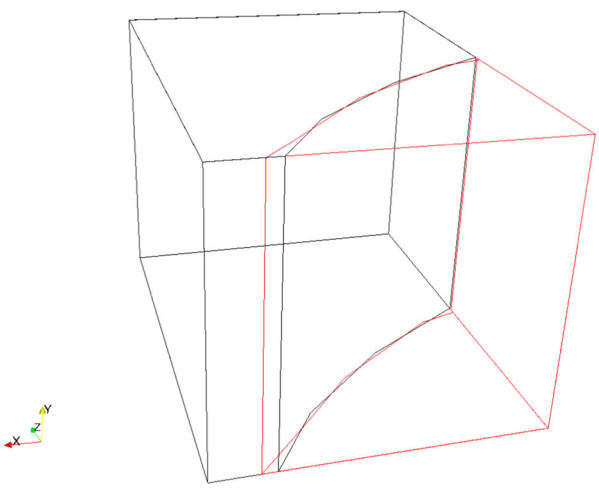


Fig. 27 Polyhedrons of intersection between two overlapping cylinders and a cell. The polyhedron with *red border* corresponds to one cylinder, and the polyhedron with *black border* corresponds to the other cylinder. Together the polyhedrons cover the whole of the cell, as expected. The overlaps and gaps have been exaggerated in the figure

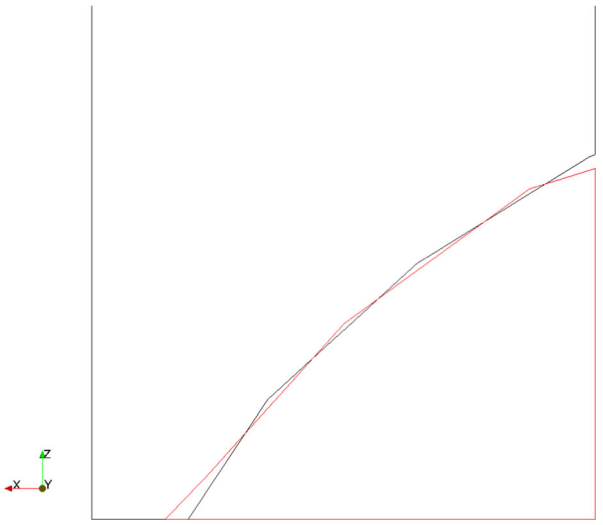


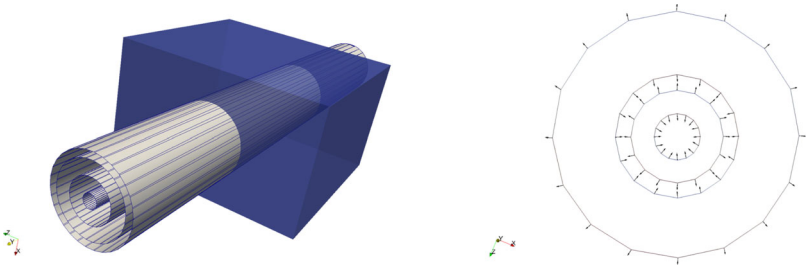
Fig. 28 At the rightmost face edge (x-low), there is a gap between the triangles, and at the bottommost face edge (z-low), the triangles slightly overlap. The overlap and the gap have been exaggerated in the figure

and

$$\beta_i^2 = \begin{cases} 0, & i = 0,\ 1, \\ 6.33 \cdot 10^{-2}, & i = 2,\ 3,\ 4,\ 5, \end{cases} \tag{25}$$

where the superscripts 1 and 2 correspond to the first and second test cases, respectively. The area fractions in (24) and (25) are calculated from the formulas for the area of a circle and an ellipse, respectively. The first volume fraction, α^1, is simply calculated from the formula for the volume of a cylinder. The second volume fraction is

$$\alpha^2 = \frac{V_1 - V_2 + V_3 - V_4}{\Delta x\, \Delta y\, \Delta z}, \tag{26}$$



(a) The test setup.

(b) The cylinder triangle meshes in intersection, with triangle normals pointing out of the interior of each mesh.

Fig. 29 Four axis-aligned cylinders with the same center line and different radii intersecting an axis-aligned hexahedron

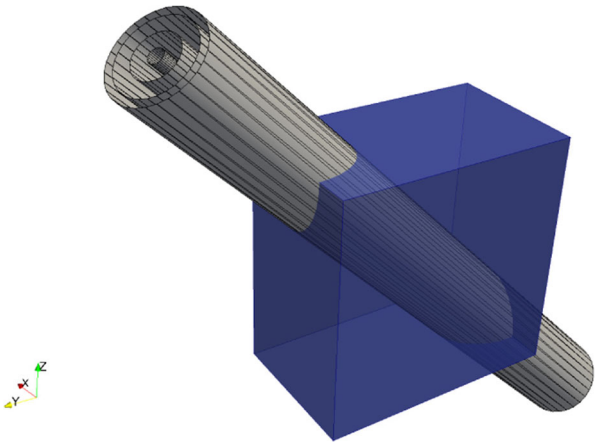


Fig. 30 Four cylinders with the same center line and different radii intersecting an axis-aligned hexahedron. The cylinders are rotated 45 degrees about the x-axis

where

$$V_i = \pi\, r_i^2 \sqrt{(\Delta y)^2 + (\Delta z)^2} - \frac{8}{3} r_i^3, \quad i = 1,\ 2,\ 3,\ 4, \tag{27}$$

is the volume of intersection of the cylinder with radius r_i and the hexahedron. In (27), V_i is the sum of the volume of three parts. The first part is the largest cylinder C that fits into the intersection between the rotated cylinder and the cell. The remaining parts are the two identical bodies that are formed as the difference of the whole volume of intersection and C. The volume of the two identical bodies was calculated from the definition of the volume integral. The area and volume fractions calculated with the proposed algorithm agree with the theoretical results.

7 Discussion and conclusion

Two methods for calculation of solid area and volume fractions of the intersection between degenerate triangle meshes and a hexahedral grid have been proposed. The algorithms have been implemented in the multiphase flow framework IBOFlow and will be used to improve the accuracy in the calculation of fluxes between fluids and solids. The solid area and volume fractions indicate how much of each fluid cell that is intersected by the solid. The exact algorithm is the main result that handles all geometric complications addressed in this paper. The approximate method is intended for highly resolved grids, for which it is reasonable to approximate the cell-mesh intersection with a plane.

There are some comments on what could have been done differently in the two methods. In the exact method, since both hexahedra and triangles are convex, the cell polygons are also convex. This could be used in an alternative method to connect the intersection points to cell polygons, for example as described in Section 4.4 and Appendix A.2. In the approximate method, since it is known that the vertices to be

sorted are the result of intersecting a plane with a hexahedral cell, an alternative is to use the connectivity of the cell to connect the polygons. A method similar to, but simpler than, the one used in the exact method in Section 4.3 and Appendix A.1 could also have been adopted. The advantages and drawbacks of these alternatives have not been investigated.

A limitation of the current implementation is the numerical computation of the triangle-cell intersection points, which could fail due to numerical round-off. In case this would happen, we currently use the approximate algorithm as a backup. Since the approximate algorithm cannot handle all cases, a better alternative would be to switch to exact arithmetics when the floating point precision algorithms fail. An even better approach would be to improve the discrete method for determining the number of intersection point and their location. If this could be done perfectly, all numerical problems would be eliminated.

A next step towards a geometrically robust method is to extend the exact algorithm to handle more general triangle overlaps and hanging nodes. Overlaps could be found by using the fact that each underlying triangle mesh is handled separately. Hanging nodes could be handled by introducing triangle subedges, and account for these in the polygon connection algorithms.

Acknowledgments This work was supported in part by the Swedish Governmental Agency for Innovation Systems, VINNOVA, through the FFI Sustainable Production Technology program, and in part by the Sustainable Production Initiative and the Production Area of Advance at Chalmers University of Technology. The support is gratefully acknowledged.

Appendix A: pseudo code for various algorithms

We here present pseudo code for the helper methods of the algorithms in Section 4. Algorithms for the exact method are described in Section 1, and algorithms for the approximate method are described in Section 10. The algorithms in the pseudo code are sometimes simplified, but cover the key steps of a successful implementation.

A.1 Exact method

Pseudo code for the exact algorithm is given in Algorithm 1 of Section 4.3. It includes connecting intersection points to polyhedral face polygons as described in Section 4.3.1. This is done in the method ConnectPolygons, which is given in pseudo code in Algorithm 3. ConnectPolygons uses several subroutines, which are outlined in Algorithms 4–10. An overview of the algorithms and where they are called is presented in Table 1.

In Algorithm 3, it is assumed that an intersection point is either taken or untaken. An intersection point is taken if it is already included in a polygon in the target facet.

Table 1 Overview of algorithms in the exact method

Method	Reference	Used in
CalculateExactAreaAndVolumeFraction	Algorithm 1	–
ChangeSearchFacet	Algorithm 8	Algorithm 3
ChangeSearchFacetAndUpdateSearchDirection	Algorithm 9	Algorithm 3
ConnectPolygons	Algorithm 3	Algorithm 1
GetNextVertexInPolygon (cell polygon)	Algorithm 5	Algorithm 3
GetNextVertexInPolygonOnFaceSide	Algorithm 7, 10	Algorithm 6
GetNextVertexInPolygon (face polygon)	Algorithm 6	Algorithm 3
GetSearchFacet	Algorithm 4	Algorithm 3

Algorithm 3 polygons = ConnectPolygons(intersection points, target facet)

Data: A triangle or a cell face, and the intersection points located on that triangle or cell face.
Result: The cell polygon or face polygons associated with the triangle or cell face.

```
1  polygons ← ∅
2  i ← 0
3  while i < number of elements in intersection points do
4      Π ← ∅
5      v_0 ← first untaken intersection point in target facet
6      F^S ← GetSearchFacet(target facet, v_0)
7      v_i ← v_0
8      t_0 ← true
9      initialize d
10     while (v_i ≠ v_0) ∨ t_0 do
11         if target facet is a triangle then
12             v_{i+1} ← GetNextVertexInPolygon(v_i, F^S)
13         else
14             v_{i+1} ← GetNextVertexInPolygon(v_i, F^S, d, t_0)
15         mark v_{i+1} as taken
16         if target facet is a triangle then
17             F^S ← ChangeSearchFacet(v_{i+1}, F^S)
18         else
19             {F^S, d} ←
               ChangeSearchFacetAndUpdateSearchDirection(v_{i+1}, F^S, d)
20         let v_{i+1} be the next vertex of Π
21         v_i ← v_{i+1}
22         t_0 ← false
23         i ← i + 1
24     append Π to polygons
```

Algorithm 4 next search facet = GetSearchFacet(target facet, intersection point)

Data: A triangle or a cell face, and the intersection point representing the first vertex in a cell polygon or face polygon associated with the triangle or cell face.
Result: The search facet on which to find the next vertex in the cell polygon or face polygon.

```
1  if target facet is a triangle then
2      if intersection point is in a triangle edge or a triangle vertex then
3          next search facet ← the triangle edge or triangle vertex in which the
           intersection point is included
4      else
5          next search facet ← the face side or face corner in which the
           intersection point is included
6  else
7      next search facet ← the triangle in which the intersection point is included
```

Algorithm 5 next vertex = GetNextVertexInPolygon(current intersection point, current search facet)

Data: The current vertex in a cell polygon under construction, and the search facet in which to find the next vertex in the cell polygon.
Result: The next vertex in the cell polygon.

```
1  if current search facet is a triangle edge then
2      if no untaken intersection point in the current search facet then
3          // current intersection point is in a triangle vertex
4          next vertex ← the untaken intersection point in the other triangle edge
5          in which the current intersection point is included
6      else
7          next vertex ← the untaken intersection point in the current search facet
8  else
9      // current search facet is a cell face
10     if no untaken intersection point in the current search facet then
11         if current intersection point is in a face side then
12             next vertex ← next untaken intersection point in the second cell
               face in which the current intersection point is included
13         else
14             // current intersection point is in a face corner
15             next vertex ← next untaken intersection point in one of the other
               two faces in which the current intersection point is included
16     else
17         next vertex ← the untaken intersection point in the current search facet
```

Every intersection point is untaken by default. We also use that a face side (cell edge) is a directed line segment. Thus, each intersection point located on a face side has a unique position relative to the face side, represented as a number in the interval $[0,\ 1]$. In Algorithm 3, we introduce the notation d for the search direction on a face side. If F^S is a face side, d determines if the relative position of the next vertex is smaller or larger than the relative position of the current vertex.

Algorithm 6 next vertex = GetNextVertexInPolygon(current intersection point, current search facet, current search direction, first time)

```
Data: The current vertex in a face polygon under construction, the search facet
      in which to find the next vertex in the face polygon, a search direction
      used if the search facet is a face side, and a boolean value indicating if
      this method has been called before with this face polygon.
Result: The next vertex in the face polygon.
1  if current search facet is a triangle then
2      if no untaken intersection point in the current search facet then
3          if first time then
4              if no untaken intersection point in any of the other triangles in
               which the current intersection point is included then
5                  if current intersection point is in a face side then
6                      current search facet ← the face side in which the current
                       intersection point is included
7                      next vertex ←
                       GetNextVertexInPolygonOnFaceSide(current intersection
                       point, current search facet, current search direction)
8                  else
9                      // current intersection point is in a face corner
10                     current search facet ← one of the other face sides in which
                       the current intersection point is included
11                     next vertex ←
                       GetNextVertexInPolygonOnFaceSide(current intersection
                       point, current search facet, current search direction)
12             else
13                 next vertex ← next untaken intersection point in one of the
                   other triangles in which the current intersection point is included
14     else
15         next vertex ← the untaken intersection point in current search facet
16 else
17     // current search facet is a face side
18     next vertex ← GetNextVertexInPolygonOnFaceSide(current intersection
       point, current search facet, current search direction)
```

Algorithm 7 next vertex = GetNextVertexInPolygonOnFaceSide(current intersection point, face side, search direction)

Data: The current vertex in a face polygon under construction, the face side in which to find the next vertex in the face polygon, and a search direction for the face side.
Result: The next vertex in the face polygon.

```
1  s ← relative position of the current intersection point on the face side
2  d ← a value larger than 1.0
3  for each intersection point p in the face side do
4      s_p ← relative position of p on the face side
5      if p on correct side of the current intersection point given search direction
       then
6          d_p ← |s_p − s|
7          if d_p < d then
8              d ← d_p
9              next vertex ← p
```

Algorithm 8 next search facet = ChangeSearchFacet(current intersection point, current search facet)

Data: The current vertex in a cell polygon under construction, and the search facet in which the current vertex was found.
Result: The search facet in which to find the next vertex in the polygon

```
1  if current intersection point is in a triangle vertex v then
2      next search facet ← the triangle edge including v that is not the current
       search facet
3  else if current intersection point is in a triangle edge e and e is not the current
   search facet then
4      next search facet ← e
5  else if current intersection point is in a face side or face corner then
6      next search facet ← a cell face in which the side or corner is included, and
       which has not already been searched
7  else
8      // current intersection point is in a cell face f
9      next search facet ← f
```

Algorithm 9 {next search facet, next search direction} = ChangeSearchFacetAndUpdateSearchDirection(current intersection point, current search facet, current search direction)

Data: The current vertex in a cell polygon under construction, the search facet in which the current vertex was found, and the current search direction.
Result: The search facet in which to find the next vertex in the polygon, and the next search direction which needs update if the next search facet is a face side.

```
1  finished ← false
2  next search direction ← current search direction
3  if current intersection point is in a triangle edge or a triangle vertex then
4      if there is a a triangle t in which the current intersection point is included,
       and t has not already been searched then
5          next search facet ← t
6          finished ← true
7  else
8      // current intersection point is in the interior of a triangle t
9      if current search facet is a face side then
10         next search facet ← t
11         finished ← true
12 if ¬ finished then
13     // next search facet is a face side, so we have to update the search direction
14     if current intersection point is in a face side s then
15         next search facet ← s
16         if current search facet is a triangle with normal n then
17             if the projection n_s of n on s is not zero then
18                 next search direction ← the direction corresponding to −n_s
19             else
20                 find a triangle in which the current intersection point is
                   included, and which normal has a nonzero projection n_s on s
21                 next search direction ← the direction corresponding to −n_s
22         else
23             next search direction ← current search direction
24     else
25         // current intersection point is in a face corner c
26         if current search facet is a face side then
27             next search facet ← the face side in which c is included, and which
               is not the current search facet
28             next search direction ← current search direction
29         else
30             // current intersection point is in a triangle t, find face side to search
               in and search direction
31             next search facet ← the face side in which c is included, and which
               is located on the inside of t assuming that the normal of t points
               outwards
32             next search direction ← the direction of the next search facet
               pointing away from c
```

Algorithm 10 next vertex = GetNextVertexInPolygonOnFaceSide(current intersection point, face side, search direction)

Data: The current vertex in a face polygon under construction, the face side on which to find the next vertex in the face polygon, and a search direction for the face side.
Result: The next vertex in the face polygon.

```
1  s ← relative position of the current intersection point on the face side
2  d ← a value larger than 1.0
3  for each intersection point p in the face side do
4      s_p ← relative position of p on the face side
5      if p on correct side of the current intersection point given search direction
       then
6          if (p − current intersection point) · normal of triangle including p > 0.0
           then
7              d_p ← |s_p − s|
8              if d_p < d then
9                  d ← d_p
10                 next vertex ← p
```

A.2 Approximate method

In the approximate method outlined in Algorithm 2 in Section 4.4, we need to sort the vertices of a convex polygon in clockwise or counterclockwise order. This is performed by the method ConvexPolygon in Algorithm 11, which sorts a list of vertices given the normal of the polygon. Pseudo code for SortVerticesRecursive, which performs the sorting and returns a list with the sorted input points, is presented in Algorithm 12. Pseudo code for the method PartitionVertices used in Algorithm 12 is presented in Algorithm 13. An overview of the methods and where they are called is presented in Table 2.

Table 2 Overview of algorithms in the approximate method

Method	Reference	Used in
CalculateApproximateAreaAndVolumeFraction	Algorithm 2	–
ConvexPolygon	Algorithm 11	Algorithm 2
PartitionVertices	Algorithm 13	Algorithm 11
SortVerticesRecursive	Algorithm 12	Algorithm 11

Algorithm 11 polygon = ConvexPolygon(vertices, normal)

Data: Vertices of a convex polygon, and the normal of the polygon.
Result: The polygon vertices sorted in clockwise or counterclockwise order.

```
1 pop last element in vertices and assign to v_0
2 lines ← ∅
3 for each vertex v in vertices do
4 |   append (v - v_0) to lines
5 polygon ← SortVerticesRecursive(vertices, v_0, lines, normal)
6 append v_0 to polygon
```

Algorithm 12 polygon = SortVerticesRecursive(vertices, origin, lines, normal)

Data: All but one vertices of a convex polygon, the last vertex of the polygon chosen as origin, lines from the origin to all other vertices, and the normal of the polygon.
Result: The polygon vertices in vertices sorted in clockwise or counterclockwise order.

```
1  if vertices == ∅ then
2  |   return ∅
3  if number of elements in vertices == 1 then
4  |   return vertices
5  else
6  |   pop last element in vertices and assign to pivot
7  |   pop last element in lines and assign to pivot line
8  |   {left vertices, right vertices, left lines, right lines} ←
   |   PartitionVertices(vertices, normal, lines, pivot line)
9  |   polygon ← SortVerticesRecursive(left vertices, origin, left lines, normal)
10 |   append pivot to polygon
11 |   tmp ← SortVerticesRecursive(right vertices, origin, right lines, normal)
12 |   append tmp to polygon
13 |   return polygon
```

Algorithm 13 {left vertices, right vertices, left lines, right lines} = PartitionVertices(vertices, normal, lines, pivot line)

Data: Vertices of a convex polygon, the normal of the polygon, lines from an origin to all input vertices, and a reference line from the origin to a reference vertex to sort about.

Result: The input vertices sorted to the left and to the right of the reference line, and the input lines sorted to the left and to the right of the reference line.

```
1  left lines ← ∅
2  right lines ← ∅
3  left vertices ← ∅
4  right vertices ← ∅
5  line normal ← (pivot line)×normal
6  for v ∈ {first index of vertices, ..., last index of vertices} do
7      if (line normal) · lines(v) > 0.0 then
8          append vertices(v) to left vertices
9          append lines(v) to left lines
10     else
11         append vertices(v) to right vertices
12         append lines(v) to right lines
```

References

1. Aftosmis, M.J., Berger, M.J., Melton, J.E.: Robust and efficient Cartesian mesh generation for component-based geometry. AIAA J. **36**(6), 952–960 (1998)
2. Aftosmis, M.J., Berger, M.J., Melton, J.E.: Adaptive Cartesian mesh generation. In: Weatherill, N.P., Soni, B.K., Thompson, J. (eds.) Handbook of Grid Generation, pp. 22-1–22-21. CRC Press, Boca Raton (1998)
3. Akenine-Möller, T.: Fast 3D triangle-box overlapping test. J. Graph. Tools **6**(1), 29–33 (2001)
4. Attene, M., Campen, M., Kobbelt, L.: Polygon mesh repairing: an application perspective. ACM Comput. Surv. **45**(2), Article No. 15 (2013)
5. Attene, M.: Direct repair of self-intersecting meshes. Graph. Model. **76**(14), 658–668 (2014)
6. Brentzen, J.A., Aans, H.: Signed distance computation using the angle weighted pseudonormal. IEEE Trans. Vis. Comput. Graph. **11**(3), 243–253 (2005)
7. Barequet, G., Sharir, M.: Filling gaps in the boundary of a polyhedron. Comput.-Aided Geom. Des. **12**(2), 207–229 (1995)
8. Bashein, G., Detmer, P.R.: Centroid of a polygon. In: Heckbert, P.S. (ed.) Graphics Gems IV, pp. 3–6. Academic, New York (1994)
9. Bischoff, S., Pavic, D., Kobbelt, L.: Automatic restoration of polygon models. Trans. Graph. **24**(4), 1332–1352 (2005)
10. Botsch, M., Pauly, M., Kobbelt, L., Alliez, P., Levy, B.: Geometric modeling based on polygonal meshes http://lgg.epfl.ch/publications/2008/botsch_2008_GMPeg.pdf. Accessed 27 March 2016 (2007)
11. Cieslak, S., Ben Khelil, S., Choquet, I., Merlen, A.: Cut cell strategy for 3-D blast waves numerical simulations. Shock Waves **10**, 421–429 (2001)
12. Ericson, C.: Real-Time Collision Detection. Morgan Kaufmann, Burlington (2004)
13. Gander, W., Hrebicek, J.: Solving Problems in Scientific Computing Using MAPLE and MATLAB. Springer Verlag, Heidelberg (1993)
14. Gouraud, H.: Continuous shading of curved surfaces. IEEE Trans. Comput. **20**(6), 124–133 (1971)

15. Guéziec, A., Taubin, G., Lazarus, F., Horn, B.: Cutting and stitching: converting sets of polygons to manifold surfaces. IEEE Trans. Vis. Comput. Graph. **7**(2), 136–151 (2001)
16. IBOFlow. http://www.iboflow.com. Accessed 27 March 2016
17. Ju, T.: Robust repair of polygonal models. ACM Trans. Graph. (TOG) **23**(3), 888–895 (2004)
18. Liepa, P.: Filling holes in meshes. In: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on geometry processing, pp 200–205 (2003)
19. Mark, A., Rundqvist, R., Edelvik, F.: Comparison between different immersed boundary conditions for simulation of complex fluid flows. Fluid Dyn. Mater. Process. **7**(3), 241–258 (2011)
20. Mark, A., Svenning, E., Edelvik, F.: An immersed boundary method for simulation of flow with heat transfer. Int. J. Heat Mass Transf. **56**, 424–435 (2013)
21. Nooruddin, F.S., Turk, G.: Simplification and repair of polygonal models using volumetric techniques. IEEE Trans. Vis. Comput. Graph. **9**(2), 191–205 (2003)
22. Peskin, C.S.: Numerical analysis of blood flow in the heart. J. Comput. Phys. **25**, 220–252 (1977)
23. Quirk, J.J.: An alternative to unstructured grids for computing gas dynamic flows around arbitrarile complex two-dimensional bodies. Comput. Fluids **23**, 125–142 (1994)
24. Schirra, S.: Robustness and precision issues in geometric computation. In: Sack, J.-R., Urrutia, J. (eds.) Handbook of Computational Geometry, pp. 597–632. Elsevier (1997)
25. Sutherland, I.E., Hodgman, G.W.: Reentrant polygon clipping. Comm. ACM **17**(1), 32–42 (1974)
26. Yang, G., Causon, D.M., Ingram, D.M.: Calculation of compressible flows about complex moving geometries using a three-dimensional Cartesian cut cell method. Int. J. Numer. Meth. Fluids **33**, 1121–1151 (2000)
27. Zhou, Q., Grinspun, E., Zorin, D., Jacobson, A.: Mesh arrangements for solid geometry. ACM Trans. Graph. **35**(4), Article No. 39 (2016)