An Interface-Preserving Moving Mesh in Multiple Space Dimensions

MARIA ALKÄMPER, JIM MAGIERA, and CHRISTIAN ROHDE, University of Stuttgart, Institute of Applied Analysis and Numerical Simulation, Germany

An interface-preserving moving mesh algorithm in two or higher dimensions is presented. It resolves a moving \((d - 1)\)-dimensional manifold directly within the \(d\)-dimensional mesh, which means that the interface is represented by a subset of moving mesh cell-surfaces. The underlying mesh is a conforming simplicial partition that fulfills the Delaunay property. The local remeshing algorithms allow for strong interface deformations. We give a proof that the given algorithms preserve the interface after interface deformation and remeshing steps. Originating from various numerical methods, data is attached cell-wise to the mesh. After each remeshing operation, the interface-preserving moving mesh retains valid data by projecting the data to the new mesh cells.

An open source implementation of the moving mesh algorithm is available at Reference [1].

CCS Concepts: • Mathematics of computing → Mesh generation; Discretization;
Additional Key Words and Phrases: Moving mesh, local dynamic remeshing, interface tracking, mesh adaptation, Delaunay triangulation

ACM Reference format:
Maria Alkämper, Jim Magiera, and Christian Rohde. 2024. An Interface-Preserving Moving Mesh in Multiple Space Dimensions. ACM Trans. Math. Softw. 50, 1, Article 5 (March 2024), 28 pages.
https://doi.org/10.1145/3630000

1 INTRODUCTION

For many applications in science and engineering, mesh-based methods are utilized to solve time-dependent partial differential equations. The quality of the numerical solutions strongly depends on the quality of the mesh. This is especially the case for sharp-interface approaches, where the interface should be resolved within the mesh, which means that the interface should coincide with mesh surfaces. This can be used, for example, to track discontinuities, such as material faults, the dynamic position of a phase transition in a liquid, or a growing crack in a medium.

One big challenge of those models is that the interface is free and should be able to move according to the dynamics of the model. That means the mesh should be able to incorporate large...
Fig. 1. (a) Initial configuration. (b) Strong interface deformation without and (c) with mesh topology changes.

deformations of the interface. However, a naive implementation by, e.g., a Lagrangian approach that does not perform topological changes within the mesh may yield poor meshes—see Figure 1.

To remedy this problem, we propose an interface-preserving moving mesh (IPMM) that has the following advantages:

- Strong deformations of the interface are possible.
- The interface has an explicit and connected representation within the mesh, from which all geometric information can be inferred, e.g., the local curvature.
- A high resolution of the interface is achievable.
- Remeshing is performed in a local fashion.

The moving mesh method for one space dimension has been proposed in Reference [27] and further developed in Reference [41]. A noteworthy contribution to simulate fluid interfaces using front tracking is presented in References [46, 48], where fluid interfaces are represented by a separate, lower-dimensional moving mesh that exists on top of a static background mesh.

The groundwork for our moving mesh approach was laid in References [11, 13]. In these contributions hyperbolic conservation laws are discretized using a moving mesh in one and two dimensions, which tracks discontinuous PDE-solutions within the mesh.

The moving mesh methods applied in References [4, 17, 34, 36, 37, 47] are related to our work but differ in methodological approaches. For an overview on moving mesh methods, we also refer to Reference [28] and the references therein. In Reference [34] the authors used mesh motion based on spring analogy in combination with mesh flips to maintain optimal mesh connectivity but did not include insertion nor removal of points. Unlike in the latter, in Reference [4] insertion and removal of points was included in a mesh enrichment and mesh coarsening step but only for the two-dimensional setting. Nevertheless, the mesh attained a worse quality after the interface has passed through. In References [17, 36, 37, 47] both the generalization to the three-dimensional case as well as the poor mesh quality was tackled. The authors presented the insertion and removal of vertices via the edge bisection and edge collision methods as well as the local optimization of the mesh quality via edge swapping and mesh smoothing operations.

In contrast to all aforementioned moving mesh approaches, we use a background mesh to retain an initially good mesh quality away from the interface, even after an interface has passed through some part of the mesh. This is especially beneficial if the initial mesh is a highly optimized (Delaunay) mesh with respect to quality, which can be constructed by well-established methods. Furthermore, we retain the Delaunay property of the mesh. Thus, mesh modifications are realized by
using built-in operations on Delaunay meshes (insert, remove, find conflicts) only without altering the mesh manually, which gives comparably simple algorithms.

We seek to present our moving mesh algorithm in a generic way, to be applicable to a wide range of models (see, e.g., Reference [12]), as well as to arbitrary dimensions. The algorithm is also formulated in a way to be easily implemented in already existing mesh frameworks. The most important advantages of our moving mesh method become apparent if we regard it in conjunction with the numerical discretization schemes that can be built on top of it (see, for example, Section 4.3). One advantage is that no interpolation of the data across the interface is needed—which could break two-phase flow simulations of compressible fluids. For example, the interpolated data could be an averaged fluid density over different phases, which cannot be uniquely assigned to a phase and thus would not have a physical meaning. As another advantage, it is possible to apply an easily exchangeable, separate model to describe the interface dynamics, enabling us to include microscale effects directly at the interface.

In this article, we will first introduce our moving mesh method in multiple dimensions by proposing algorithms that allow us to move vertices in the mesh, while taking care of data associated with the mesh (Section 2). Subsequently, these methods are used to formulate the IPMM-algorithm in Section 3. In Section 4, we present computational test cases, showing the capabilities of the method.

To ensure a comprehensible method and reproducible results, we provide an open source implementation of the moving mesh algorithm at Reference [1].

2 MOVING MESH ALGORITHM IN MULTIPLE DIMENSIONS

In this section, we present the moving mesh approach for $d \in \{2, 3\}$ dimensions, but all algorithms are also applicable for $d > 3$. The moving mesh approach is based on simplicial meshes, therefore, we begin by defining the basic mesh entities.

Definition 2.1 (Simplex, Facet, and Edge). A $d$-dimensional simplex $C$ is a polytope defined by $d + 1$ distinct vertices $v_0, \ldots, v_d \in \mathbb{R}^d$, i.e.,

$$C := \{ x \in \mathbb{R}^d : x = \sum_{i=0}^{d} \theta_i v_i, \sum_{i=0}^{d} \theta_i = 1, \theta_i \geq 0 \text{ for all } i = 0, \ldots, d \}.$$

A simplex $C$ is called nondegenerate if the vertices $v_0, \ldots, v_d \in \mathbb{R}^d$ are not coplanar, i.e., do not lie on a $(d-1)$-dimensional plane.

The surface of a $d$-dimensional simplex $C$ consists of $d + 1$ facets $S_l$, $l = 0, \ldots, d$, given by

$$S_l := \{ x \in \mathbb{R}^d : x = \sum_{i=0}^{d} \theta_i v_i, \sum_{i=0, i \neq l}^{d} \theta_i = 1, \theta_i \geq 0 \text{ for all } i = 0, \ldots, d, i \neq l \}.$$

Note that a facet is a $(d-1)$-dimensional simplex in a $d$-dimensional space. An edge $e_{ij}$ of a $d$-dimensional simplex, for $d \geq 2$, is defined by two vertices $v_i, v_j, i, j = 0, \ldots, d, i \neq j$,

$$e_{ij} := \{ x \in \mathbb{R}^d : x = \theta_i v_i + \theta_j v_j \text{ with } \theta_i + \theta_j = 1 \text{ and } \theta_i, \theta_j \geq 0 \}.$$

Consequently, a $d$-dimensional simplex has $\frac{1}{2}d(d+1)$ edges.

The simplex surfaces are called facets to be consistent with the three-dimensional case. In two dimensions, both facets and edges correspond to the edges of a triangle.

To begin, we define the static mesh, which is a partition of a given domain $\Omega \subset \mathbb{R}^d$ into a set of simplices. Implicitly, this defines the set of vertices $\mathcal{V} \subset \mathbb{R}^d$ by the simplex vertices. Consequently, for domains with a smooth boundary, an exact meshing with simplices is in general not possible.
We assume that all vertices $v \in V$ are in general position [49], i.e., no $i + 1$ vertices lie in a subspace of dimension $i - 1$, for $i = 1, \ldots, d$, and no vertices are cospherical, which means no $d + 2$ vertices $v \in V$ share the same circumsphere.

For most cases Assumption 2.2 does not yield severe restrictions but simplifies the presentation of our method by avoiding the discussion of degenerate cases. Still, our method is presented on a sufficiently abstract level, that if all underlying geometric operations are appropriately implemented, these cases are covered as well.

Definition 2.3 (Mesh). Let a domain $\Omega \subset \mathbb{R}^d$ with piecewise linear boundary be given. A mesh $\tau := \{C_j : j = 1, \ldots, N\}$ is a partition of $\Omega$ into a set of $N \in \mathbb{N}$ nondegenerate $d$-dimensional simplices $C_j$. The vertex set $V \subset \mathbb{R}^d$ is defined as the set of all simplex vertices. The vertices of the $j$th cell $C_j$ are denoted by $v_j^i \in V$ for $i = 0, \ldots, d$.

We refer to the simplices also as cells in the context of meshes, and the corresponding facets of the mesh are also called cell surfaces.

We propose a Delaunay mesh [18, 19, 40] as the base of the moving mesh.

Definition 2.4 (Delaunay Mesh). Let a mesh $\tau$ with vertex set $V$ be given. The mesh is called a Delaunay mesh (or: Delaunay triangulation) iff the circumsphere of each cell $C_j \in \tau$ of the mesh does not contain any other vertex from $V$ than the cell vertices $v_j^i$, $i = 0, \ldots, d$.

Using a Delaunay mesh has several advantages. It is uniquely defined [19] in the non-cospherical case, and we write in this case

$$\tau = \tau(V)$$

for the Delaunay mesh of the vertex set $V$. In case of $d + 2$ cospherical vertices the Delaunay mesh can still be defined, but is not unique anymore. From a numerical point of view, a Delaunay mesh is usually a good choice among the set of all possible triangulations, as it maximizes the minimum angle inside its cells [7, 18]. There exist efficient and already implemented algorithms for the insertion and removal of vertices from a Delaunay mesh (see, e.g., References [6, 22]).

Next, we define the moving mesh introducing time dependence. Hereby, the moving mesh is driven by the motion of the vertices. Let a monotone sequence of points in time $(t_k)_{k \in \mathbb{N}_0}$, with $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$, be given with $t_k \to \infty$. For any time interval $[t_k, t_{k+1})$ let a mesh $\tau_k$ with vertex set $V_k$ be given. Assume that for each $v \in V_k$ there exists a continuous function

$$m_v: [t_k, t_{k+1}) \to \mathbb{R}^d, t \mapsto m_v(t),$$

with $m_v(t_k) = 0$, describing the relative motion of the vertex $v$. The position of the vertex $v$ at time $t \in [t_k, t_{k+1})$ is given by $v(t) := v + m_v(t)$. Then, we introduce the moving mesh.

Definition 2.5 (Moving Mesh). If for all $k \in \mathbb{N}_0$ and for each $t \in [t_k, t_{k+1})$, all moving cells with vertices $v_j^i(t) := v_j^i + m_v^i(t)$, where $C_j \in \tau_k$ and $v_j^i$ are vertices of $C_j$, defined as

$$C_j(t) := \left\{ x \in \mathbb{R}^d : x = \sum_{i=0}^d \theta_i v_j^i + m_v^i(t), \sum_{i=0}^d \theta_i = 1, \theta_i \geq 0 \text{ for all } i = 0, \ldots, d \right\},$$

1This is, for example, the case for the CGAL-framework [43], where the developers provide a reliable implementation of basic geometry operations. See Reference [30] for some examples that highlight problems that arise by using inexact arithmetic in geometric applications.
form a mesh of $\Omega$, then we call $\mathcal{T} = (\tau_k, \{m_v\}_{v \in \mathcal{V}_k})_{k \in \mathbb{N}_0}$ a moving mesh. The set
\[ \tau_k(t) := \{C_j(t) : C_j \in \tau_k\}, \text{ for } t \in [t_k, t_{k+1}), \]
is the moving mesh at time $t \in [t_k, t_{k+1})$, with time-dependent vertices $v(t)$ and cells $C_j(t)$.

In general, the Delaunay property of the mesh has to be restored after motion. Thus, we have to be aware that the topology of the mesh, or more precisely the topology of the underlying graph of the mesh, may change at certain points in time. That means that due to the motion, the number of cells as well as neighborhood relations may change. Note that, in Definition 2.5, we have defined the entire moving mesh $\mathcal{T}$ as a sequence of base-meshes $\tau_k$ for every point in time $t_k, k \in \mathbb{N}_0$. Based on these meshes $\tau_k$, we define time-dependent sub-meshes $\tau_k(t)$ in the time interval $t \in [t_k, t_{k+1})$, which have time-dependent vertices but a fixed mesh topology. Implicitly, this restricts the vertex motion to avoid degenerate mesh structures. The mesh topology only changes at each time $t_{k+1}$ from the base-mesh $\tau_k$ to the next base-mesh $\tau_{k+1}$ (see Figure 2 for an illustration of a moving mesh).

The advantage of this approach is that we are able to formulate numerical algorithms on the sub-meshes, where smooth, time-dependent vertex motions are possible, but no topology change takes place. The necessary topology changes, to allow for, e.g., large vertex motions, take place only at discrete points in time $t_k, k \in \mathbb{N}_0$. Consequently, data associated with the mesh has to be transferred only at these discrete points in time, which will be discussed later. To avoid special treatment of the domain boundary and some edge cases, we make the assumption that the vertices on the domain boundary $\partial \Omega$ do not change their positions, i.e., for all $v \in \mathcal{V}_k$ with $v \in \partial \Omega$, we assume $m_v \equiv 0$.

Remark 2.6. Our main concern is about the treatment of interfaces in a meshed domain (see Section 3 below). The interface dynamics close to the domain boundary $\partial \Omega$, the case of a (transversal) intersection of the interface with $\partial \Omega$, or settings with time-dependent domains $\Omega = \Omega(t)$ are delicate to realize if the vertices on the domain boundary are fixed. Let us note that the current 3D implementation is in fact designed for interfaces within the domain $\Omega$ and being close to $\partial \Omega$, e.g., a hydromechanical layer. The local nature of the moving mesh algorithm guarantees that the boundary vertices do not experience a motion.

To make the moving mesh $\tau(t)$ unique for a given vertex set $\mathcal{V}$ at every discrete point in time $t_k$, $k \in \mathbb{N}_0$, we base the moving mesh on Delaunay meshes. Beside their uniqueness, they are generally considered well-shaped meshes for numerical applications.
Definition 2.7 (Delaunay Moving Mesh). Let a moving mesh
\[ \mathcal{T} = (\tau_k, \{m_v\}_{v \in \mathcal{V}_k})_{k \in \mathbb{N}_0}, \]
for a monotone sequence of points in time \((t_k)_{k \in \mathbb{N}_0}\) be given. We call \(\mathcal{T}\) a Delaunay moving mesh, iff \(\tau_k(t_k)\) is a Delaunay mesh for all \(t_k, k \in \mathbb{N}_0\).

Obviously, if a moving mesh \(\mathcal{T}\) is given with a Delaunay mesh \(\tau_k\), then we cannot expect that \(\tau_k(t)\) is Delaunay for \(t > t_k\). This happens if points move into the circumspheres of other cells and therefore are in conflict with them. However, the Delaunay Moving Mesh is a conforming mesh for every \(t \in [0, \infty)\): The base-meshes \(\tau_k\) are conforming by the definition of Delaunay meshes and the continuous deformation of the latter cannot generate hanging vertices.

Definition 2.8 (Conflict, Conflict Zone, Resulting Hole). Let a vertex set \(\mathcal{V}\) and its corresponding Delaunay mesh \(\tau(\mathcal{V})\) be given. A point \(\vec{v} \in \mathbb{R}^d\) is in conflict with a cell \(C \in \tau(\mathcal{V})\) if it lays inside or on the boundary of the circumsphere of \(C\). We call the union of all cells that are in conflict with a point \(\vec{v}\) the conflict zone of \(\vec{v}\). Furthermore, we call the union of all cells that are adjacent to a vertex \(v \in \mathcal{V}\) the resulting hole of \(v\).

Due to arising conflicts, a change of topology, meaning a redefinition of the cells, will be necessary to restore the Delaunay property. This may change the number of cells. Moreover, we even allow changing the number of vertices. These topology changes take place at the discrete time points \((t_k)_{k \in \mathbb{N}_0}\) from Definition 2.5. The affected areas in this process will be the conflict zones and the resulting holes of the involved vertices.

We do not want to consider a moving mesh by itself. Ultimately, we are interested in performing numerical simulations with it. More specifically, we focus on the class of finite volume methods. Therefore, we introduce the notion of generic, cell-wise defined data
\[ U(t) := (U_1, \ldots, U_N) \in \mathbb{R}^{N \times m}, \]
with \(N = |\tau(t)| \in \mathbb{N}\) being the number of all cells in \(\tau(t)\), and \(m \in \mathbb{N}\) the dimension of the data. Note that the data is implicitly related to the cell, i.e., the cell \(C_j\) corresponds to \(U_j \in \mathbb{R}^m\), for all \(j = 1, \ldots, N\). The data defines a function \(U: \Omega \times (0, \infty) \to \mathbb{R}^m\) by
\[ U(x, t) := \sum_{j=0, \ldots, N} U_j 1_{C_j(t)}(x), \quad \text{with } C_j(t) \in \tau_k(t), \ t \in [t_k, t_{k+1}), k \in \mathbb{N}_0, \]  
(2)
where \(1_{C_j}(x)\) denotes the indicator function of the cell \(C_j\).

Remark 2.9. Note that the notion of cell-wise constant data is only due to our focus on finite volume methods; other types of data are also possible. In that case, the treatment of the data in the following algorithms has to be adapted accordingly. The connection to data can even be dropped altogether, without any impact on the presented algorithms.

Due to the topology changes, we have to transfer the data from \(\tau_k(t_{k+1})\) to \(\tau_{k+1}(t_{k+1})\) in a meaningful way during the mesh adaption from \(\tau_k\) to \(\tau_{k+1}\). Depending on the application, the data can be transferred in a conservative way via either averaging or \(L^2\)-projections. How this is done is explained in the following:

For the presented mesh adaptions, it suffices to consider the data locally on a stencil \(\Delta \subset \tau\), which is a subset of connected cells \(C_j\) in a mesh \(\tau\). During mesh adaptions, we encounter an old stencil \(\Delta\) (before the mesh adaption) and a new stencil \(\tilde{\Delta}\) (after adapting the mesh) (see Figure 3 for (b) a sketch of an old stencil \(\Delta\) and (d) a sketch of a new stencil \(\tilde{\Delta}\)). Both the old and the new stencil...
are used to compute the data on the resulting mesh. We assume that both stencils are congruent in the sense that

$$\text{dom}(\Delta) := \bigcup_{C_j \in \Delta} C_j = \bigcup_{\tilde{C}_j \in \tilde{\Delta}} \tilde{C}_j.$$  

A standard choice for transferring the data $U$ from an old stencil $\Delta$ to a new stencil $\tilde{\Delta}$ on the new mesh would be the $L^2$-projection: find $\tilde{U}_{|\text{dom}(\Delta)}(x, t) = \sum_{\tilde{C}_j \in \tilde{\Delta}} \tilde{U}_j 1_{\tilde{C}_j}(x)$, such that

$$\int_{\text{dom}(\Delta)} \left\| U(x, t) - \tilde{U}(x, t) \right\|^2_2 dx \to \min,$$

where $U(x, t)$ is defined as in Equation (2). Another way to transfer the data is by local averaging, i.e., setting the new data

$$\tilde{U}_j = U_{|\Delta} := \frac{1}{|\text{dom}(\Delta)|} \sum_{C_j \in \Delta} U_j |C_j|,$$

for all $\tilde{C}_j \in \tilde{\Delta}$.

From here on, we only consider Delaunay moving meshes, therefore, if we say moving mesh, then we always mean Delaunay moving mesh.

In the following, we present methods to manipulate vertices in a mesh (remove, insert, and move vertex), while retaining a Delaunay mesh and therefore a (Delaunay) moving mesh. The vertex motion in context of Definition 2.5 is hereby understood as an algorithmic transformation from $\tau_k$ to $\tau_{k+1}$, updating the vertex positions from time $t_k$ to $t_{k+1}$ in accordance to the vertex motions $m_v(t_k+1)$. In the following, we drop the time dependency and focus on the motion of a single vertex $v$ to its new position $\tilde{v} = v + m_v$ in a vertex set $V_0 \subset \mathbb{R}^d$.

The algorithmic implementation of the vertex-motion is based on two crucial functions: First, we remove the vertex from the old position, then we insert it at its new position. It is important
that both the insertion and the removal of the vertex consistently retain the cell-wise, underlying data.

The removal of a single vertex works as follows:

**ALGORITHM 2.10: Remove Vertex**

**Description:** Removes vertex \( v \in \mathcal{V} \) in a Delaunay mesh \( \tau(\mathcal{V}) \).

**Algorithm**

1. **Old stencil \( \Lambda \):** Get data \( U \) and all cells \( C \) that are incident to \( v \).
2. **Remove** \( v \) from the Delaunay mesh \( \tau(\mathcal{V}) \) and remesh the resulting hole, i.e., compute \( \tau(\mathcal{V} \setminus \{v\}) \).
3. **New stencil \( \tilde{\Lambda} \):** Get all cells \( \tilde{C} \) that are in conflict with the old vertex \( v \).
4. **Compute data** \( \tilde{U} \) on the new stencil \( \tilde{\Lambda} \) given the data on the old stencil \( \Lambda \).

Removing vertices from Delaunay meshes has been described in, e.g., References [21, 22]. Removing a single vertex can be implemented with the computational complexity \( O(q \log q) \) [20], where \( q \in \mathbb{N} \) is the number of created cells. In the worst case, the entire Delaunay mesh might be affected, leading to a complexity of \( O(N_{\mathcal{V}}^{\lceil d/2 \rceil}) \) with \( N_{\mathcal{V}} = |\mathcal{V}| \) (cf. References [16, 20]).

The insertion of a point into an existing Delaunay mesh is similarly straightforward. Again, we have to make sure that the data is transferred correctly during the mesh adaption.

**ALGORITHM 2.11: Insert Vertex**

**Description:** Insert vertex \( \tilde{v} \in \Omega \subset \mathbb{R}^d \), in a Delaunay mesh \( \tau(\mathcal{V}) \), with \( \tilde{v} \notin \mathcal{V} \).

**Algorithm**

1. **Old stencil \( \Lambda \):** Get data \( U \) and all cells \( C \in \tau(\mathcal{V}) \) that are in conflict with \( \tilde{v} \).
2. **Insert** \( \tilde{v} \) into the Delaunay mesh. Remesh the conflict zone around \( \tilde{v} \) and restore the Delaunay property.
3. **New stencil \( \tilde{\Lambda} \):** Get all cells \( \tilde{C} \) that are incident to \( \tilde{v} \).
4. **Compute data** \( \tilde{U} \) on the new stencil \( \tilde{\Lambda} \) given the data on the old stencil \( \Lambda \).

Inserting a vertex implies that a Delaunay mesh has to be computed (see, e.g., Reference [6]). Finding a Delaunay mesh for a given vertex set is equivalent to computing the convex hull for a lifted point set (cf. Reference [9]). As such, the complexity of inserting a vertex is at its worst \( O(N_{\mathcal{V}}^{\lceil d/2 \rceil}) \) with \( N_{\mathcal{V}} = |\mathcal{V}| \) (see Reference [6]). In practice, such worst-case scenarios rarely occur, and if the mesh fulfills further (mild) requirements, then the expected runtime scales with \( O(N_{\mathcal{V}} \log N_{\mathcal{V}}) \), as shown in Reference [16].

To transfer the data on the new mesh, we use local averaging (4), or alternatively \( L^2 \)-projection (3).

Finally, moving a vertex is a combination of both methods above. To preserve the locality of the algorithm and avoid degenerating meshes, we restrict the motion distance of a vertex \( v \in \tau \) by the variable \( \omega(\tau, v) > 0 \), which is given in such a way that the new position \( \tilde{v} \) is guaranteed to be
inside the polytope defined by the incident cells of \( v \), i.e.,

\[
\omega(\tau, v) := \min_{x \in \partial K} \text{dist}(v, x), \quad \text{with } K := \bigcup_{\text{C incident to } v} \bar{C}.
\]  

(5)

For a given mesh \( \tau \), we can compute the overall minimum distance, denoted by

\[
\omega(\tau) := \min_{v \in V \setminus \partial \Omega} \omega(\tau, v).
\]  

(6)

Note that the maximum vertex motion distance is restricted by \( \omega(\tau) \), which in turn can be roughly estimated by the smallest insphere diameter of the mesh \( \tau \), i.e.,

\[
\omega(\tau) \geq \min_{C \in \tau} \text{inspherediam}(C),
\]

where \( \text{inspherediam}(C) \geq 0 \) denotes the insphere diameter of a cell \( C \). For nondegenerate meshes and if all insphere diameters can be uniformly bounded from below by a positive constant for all time, then any mesh motion can be realized by a finite number of timesteps.

With this, we are able to formulate the algorithmic version of the vertex motion, which is also illustrated in Figure 3.

**ALGORITHM 2.12: Move Vertex**

**Description**: Move vertex \( v \in V \) to the new position \( \bar{v} \in \mathbb{R}^d \) with

\[
\|v - \bar{v}\|_2 < \omega(\tau, v)
\]  

(7)

in a Delaunay mesh \( \tau(V) \).

**Algorithm**

1. Old stencil \( \Delta \): Get data \( U \) and all cells \( C \) that are incident to \( v \) and that are in conflict with \( \bar{v} \).
2. Move vertex:
   - (a) Remove \( v \) from the Delaunay mesh.
   - (b) Remesh the resulting hole.
   - (c) Insert \( \bar{v} \) into the Delaunay mesh.
   - (d) Remesh the conflict zone for \( \bar{v} \) and restore the Delaunay property.
3. New stencil \( \bar{\Delta} \): Get all cells \( \bar{C} \) that are incident to \( \bar{v} \) and that are in conflict with \( v \).
4. Compute data \( \bar{U} \) on the new stencil \( \bar{\Delta} \) given the data on the old stencil \( \Delta \).

Algorithm 2.12 is a combination of Algorithm 2.10 and Algorithm 2.11. The data transfer involves no further cells despite those that are involved in the vertex removal/insertion. As such, the additional computational costs are asymptotically negligible. Consequently, the computational complexity of the algorithm is at most \( O\left(N^\lceil d/2 \rceil\right) \).

In practice, Equation (7) is a timestep restriction, i.e., if a motion from \( v \) to \( \bar{v} \) does not satisfy Equation (7), smaller timesteps have to be chosen that satisfy Equation (7). Alternatively, other vertices that are too close to the new position could be removed (via Algorithm 2.10) before moving the vertex.

Algorithm 2.12 relies only on local operations within the stencil, which makes the algorithm highly efficient (see Section 4 for numerical benchmarks). Furthermore, local mesh operations are the foundation for a possible parallelization of the whole method.

Algorithms 2.10, 2.11, and 2.12 are already sufficient for implementing \( r \)-adaptive mesh refinement (i.e., adaptivity by vertex relocation), albeit without considering interfaces within the mesh.
The steps (1), (2), and (3) in all three algorithms can easily be implemented by using built-in methods of a Delaunay mesh implementation.

Remark 2.13. Generating well-shaped three- or d-dimensional meshes is no trivial task. In three dimensions there is the well-known phenomena of sliver tetrahedra (observed, for example, in Reference [10]). These slivers are almost degenerate tetrahedra whose vertices lie close to the equator of their circumsphere. We refer to References [5, 15] for a more in-depth classification of badly shaped simplices. Even well-spaced vertices do not prevent the occurrence of slivers in Delaunay meshes [42].

There are many methods to remove slivers and improve the mesh quality. For example, choosing appropriate weights in a weighted Delaunay mesh can remove slivers (see Reference [15]). This means no points are added/removed or perturbed, but the resulting mesh is not Delaunay anymore. Another approach is to perform Laplacian smoothing [25], optimal Delaunay triangulation smoothing (ODT-smoothing) [14], Lloyd-smoothing [23], or advanced approaches that use combinations of aforementioned methods in the construction process (see, e.g., Reference [45]).

However, in these cases, vertices are moved, therefore the boundary and interface vertices have to be treated differently. Slivers can also be removed by perturbing single vertices incident to slivers [24, 44]. Yet another approach is to remove slivers by refining the mesh [31, 32]. The drawback hereby is that new points are added and insertions near mesh interfaces have to be treated separately.

We note that the occurrence of slivers in itself is not a problem when aiming at numerical simulations. For the case of finite volume methods, sliver cells can be easily merged virtually with neighboring nondegenerate cells.

The algorithms in this section enable us to handle a moving mesh \( T \) algorithmically, given vertex motions \( m_v \), that respect the constraint (7). In Figure 2, we illustrate the relevant steps. During each time interval \( t \in [t_k, t_{k+1}) \), we define the time-depending moving mesh \( \tau_k(t) \) according to the vertex motions \( m_v(t) \) and Definition 2.5, retaining the topology of the mesh. This yields the mesh \( \tau_k(t) \), for \( t \in [t_k, t_{k+1}) \), which may not be Delaunay anymore, or even close to degenerating. This makes remeshing a necessity. The transition from \( \tau_k \) to \( \tau_{k+1} \)—including the data—is performed by applying Algorithm 2.12 for every vertex in \( \tau_k \) corresponding to their motion \( m_v = m_v(t_{k+1}) \). Algorithmically this allows for topology changes and restores the Delaunay property of the resulting mesh \( \tau_{k+1} \).

### 3 INTERFACE-PRESERVING ALGORITHMS

The moving mesh methods (Algorithms 2.10, 2.11, 2.12) are the prerequisite for our main goal. We consider the motion of an interface within a meshed domain. The interface is represented as a subset of cell surfaces within the moving mesh. Ideally, the remeshing process does not change the interface at all. The succeeding algorithms ensure preservation of the interface topology such that after remeshing, no two bulk vertices will be connected by an edge crossing the new position of the interface. To achieve interface preservation in this sense, and depending on the local properties of the prescribed motion, the position of the interface at some \( \tau_k \) can slightly change. The changes in the position of the interface lead to a geometrical error presumably of the order of the mesh size. Altogether, this is what we call an interface-preserving moving mesh (IPMM), which is our main novel contribution.

First, we define an interface within a mesh.

**Definition 3.1 (Mesh Interface).** An interface \( \Gamma \) in a mesh \( \tau \) is a \((d-1)\)-dimensional mesh that consists of \((d-1)\)-dimensional facets of the cells in \( \tau \), which form a surface in \( \mathbb{R}^d \). It is denoted...
Fig. 4. Sketch of a moving interface $\Gamma(t)$ that divides two time-dependent subdomains $\Omega_+(t)$ and $\Omega_-(t)$.

by $\Gamma_\tau$. The set of all interface vertices $v \in \Gamma_\tau$ is denoted by $\mathcal{V}_\Gamma$. We call all non-interface vertices $v \in \mathcal{V} \setminus \mathcal{V}_\Gamma$ bulk vertices.

Note that the definition above excludes the case of an interface with junction points (see Figure 5). Again, in principle, the algorithm may be extended to such cases, but requires special treatment of junctions. However, we refrain from doing so, as the focus of this work is on closed interfaces for the description of phase transitions.

Definition 3.1 addresses the static case and will serve as an initial state for the interface-preserving moving mesh. The construction of an initial (and thus static) surface interfaces within a full dimensional surrounding mesh may be realized in various well-known ways, e.g., by level-set approaches or direct parametrization.

In the next step, we define a moving interface.

**Definition 3.2 (Moving Mesh Interface).** Let a moving mesh $\mathcal{T} = (\tau_k, \{m_v\}_{v \in \mathcal{V}_k})_{k \in \mathbb{N}_0}$ and an (initial) interface $\Gamma_{\tau_0}$ be given. We call

$$\Gamma_{\tau} := \left(\Gamma_{\tau_k}, \{m_v\}_{v \in \mathcal{V}_k \cap \mathcal{V}_{\tau_k}}\right)_{k \in \mathbb{N}_0}$$

a moving mesh interface if the following holds true.

1. The vertices of the interface $\Gamma_{\tau_k}$ move in time according to the corresponding vertices of the moving mesh $\tau_k(t)$. This defines the time-dependent interface $\Gamma_{\tau_k}(t)$, $t \in [t_k, t_{k+1})$.

2. The interface $\Gamma_{\tau_k}(t)$ as a polygonal shape is retained during each transition from $\tau_k(t_{k+1})$ to $\tau_{k+1}(t_{k+1})$, for each $k \in \mathbb{N}_0$. That means that $\Gamma_{\tau_k}(t_{k+1})$ equals $\Gamma_{\tau_{k+1}(t_{k+1})}$ as sets in $\mathbb{R}^d$.

A direct consequence of Definition 3.2 in conjunction with Definition 2.5 is that the interface retains its topology globally in time. In particular, if the interface is closed, then the encapsulated domain $\Omega_-(t)$ (cf. Figure 4, see Figure 5) stays connected for all time.

As stated in Section 2, to restore the Delaunay property of the entire mesh, we have to adapt the mesh topology. Notably, this should be done in a fashion, such that the moving mesh interface topology remains invariant. Strictly speaking, our moving mesh method is not a moving interface in the sense of Definition 3.2; instead, we use the following relaxed definition of a moving interface:

**Definition 3.3 ($\epsilon$-Moving Mesh Interface).** Let $\epsilon \geq 0$, a moving mesh

$$\mathcal{T} = (\tau_k, \{m_v\}_{v \in \mathcal{V}_k})_{k \in \mathbb{N}_0},$$

ACM Transactions on Mathematical Software, Vol. 50, No. 1, Article 5. Publication date: March 2024.
and an (initial) interface $\Gamma_{\tau_0}$ be given. We call

$$\Gamma_{\tau} := \left( \Gamma_{\tau_k}, \{ m_v \}_{v \in \mathcal{V}_k \cap \mathcal{V}_{\tau_k}} \right)_{k \in \mathbb{N}_0}$$

an $\epsilon$-moving mesh interface if the following holds true:

- The vertices of the interface $\Gamma_{\tau_k}$ move in time according to the corresponding vertices of the moving mesh $\tau_k(t)$.
- The interface $\Gamma_{\tau}$ is a single closed surface.
- **Approximation property**: For each $k \in \mathbb{N}_0$ the $\epsilon$-neighborhood in $\mathbb{R}^d$ of the interface of $\tau_{k+1}(t_{k+1})$ contains the interface of $\tau_k(t_{k+1})$.
- **Preservation property**: No two bulk vertices are connected by an edge that intersects the interface of $\tau_{k+1}(t_{k+1})$.

We will succeed to construct an interface motion algorithm by exploiting the stronger Gabriel property \cite{26} for interface facets. All subsequent algorithms will result in a moving mesh with an $\epsilon$-moving mesh interface.

**Definition 3.4 (Gabriel).** Let a $(d-1)$-dimensional facet $S$ in a mesh be given. We call a sphere a witness sphere of $S$ if it contains all vertices of $S$ on its boundary and does not contain any vertex of the mesh in its interior. A facet has the Gabriel property if its smallest circumscribing sphere is a witness sphere. Independently, the smallest circumscribing sphere of a facet is called Gabriel sphere (regardless whether the facet has the Gabriel property).

We note that any facet that has the Gabriel property is part of the Delaunay mesh.

### 3.1 Interface Motion

We propose an algorithm, where only interface vertices are moved, which is based on Algorithm 2.12. As for the vertex motion, the algorithmic treatment of the interface motion is understood in the context of the remeshing from the base mesh $\tau_k$ to $\tau_{k+1}$ (cf. Definition 2.5).

The main idea of the interface motion algorithm is depicted in Figure 6 and reads as follows: Before moving interface vertices, we detect all bulk vertices in the Gabriel spheres of all interface facets at their new positions and delete all of these bulk vertices. If additionally there are no interface vertices inside the Gabriel sphere, then the interface facets must be Gabriel after interface motion. As mentioned before, the interface facets are consequently also part of the Delaunay mesh after the vertex motion. Nevertheless, if there are further interface vertices inside the Gabriel sphere, then a remeshing of the interface surface is possible. However, the preservation property of the interface is guaranteed, which we will later on prove within Theorem 3.11 and Corollary 3.12. The resulting interface is an $\epsilon$-moving mesh interface. Therefore, even though the moving interface facets may be affected by topology changes resulting from the reconstruction of the Delaunay property, the interface surface is preserved according to Definition 3.3.
The Gabriel property alone, however, does not prevent the collision of vertices. Consequently, we have to ensure that two vertices are always at least $\Delta x_{\min} > 0$ apart from each other. To achieve this, we traverse all interface vertices $v \in \mathcal{V}_I(t)$ and compute the distances to each adjacent vertex $\tilde{v}$ of $v$. If an adjacent vertex $\tilde{v}$ is closer than $\Delta x_{\min}$ to $v$ and does not belong to the interface, i.e., $\tilde{v} \notin \mathcal{V}_I(t)$, then we remove it using Algorithm 2.10. For the coarsening of interface vertices, we refer to Section 3.2.2.

So far, vertices are only moved or removed. As a consequence, the mesh only turns coarser in time. A simple solution of this drawback is to store the removed bulk vertices in a list $L_{del}$ and insert them whenever they are not inside a Gabriel sphere of any interface facet and are at least $\Delta x_{\min}$ apart from any interface vertex. In that way, we keep track of the initial mesh vertices. This means that the initial mesh can be understood as a background mesh.

| Algorithm 3.5: Move Interface |
|-------------------------------|
| **Description:** Move all interface vertices $v \in \mathcal{V}_I(t)$ to their new positions $\tilde{v} \in \mathbb{R}^d$, with $\|v - \tilde{v}\|_2 < \omega(\tau)/2$ (as defined in (6)), in a Delaunay mesh. |
| **Parameters:** Minimum interface distance $\Delta x_{\min} > 0$. |
| **Algorithm** |
| -- **Ensure interface:** For all $v \in \mathcal{V}_I(t)$, do: Find all bulk vertices that are inside the Gabriel spheres of all interface facets that are incident to $v$ with new interface vertex position $\tilde{v} \in \mathbb{R}^d$. Remove (Algorithm 2.10) all detected bulk vertices, while appending the position of the vertices to the list $L_{del}$. If the new vertex $\tilde{v} \in \mathbb{R}^d$ is outside the domain $\Omega$, then throw an exception (boundary conflict). |
| -- **Move:** For all $v \in \mathcal{V}_I(t)$, do: Move (Algorithm 2.12) the interface vertex $v$ to $\tilde{v}$. |
| -- **Coarse bulk:** Remove (Algorithm 2.10) all bulk vertices $\tilde{v} \in \mathcal{V}(t) \setminus \mathcal{V}_I(t)$ that are adjacent to an interface vertex $v \in \mathcal{V}_I(t)$ and fulfill $\|\tilde{v} - \tilde{v}\|_2 < \Delta x_{\min}$. Append all removed vertices to the list $L_{del}$. |
| -- **Refine bulk:** Insert (Algorithm 2.11) all vertices from the list $L_{del}$ that are not inside a Gabriel sphere of any interface facet and are at least $\Delta x_{\min}$ apart from any interface node. |

In comparison to the Algorithm 2.12 (move single vertex), we now move a whole set of vertices with Algorithm 3.5. Therefore, we took a stronger restriction on the vertex displacement. Precisely speaking, we check in Algorithm 3.5 that the vertices are not moved further than $\omega(\tau)/2$, which is sufficient to ensure that no vertices collide or move out of their incident cell polytope $K$ (see Equation (5)). The algorithm is a combination of the previous algorithms and thus the worst case complexity amounts to $O(N_{\mathcal{V}}^{d/2})$. In practice, the expected complexity should be lower than observed in the benchmarks in Section 4.

Algorithm 3.5 can be used to move interface vertices from their given positions $v$ to arbitrary new positions $\tilde{v}$, as long as the stated restriction (8) is fulfilled. To make a link back to the introduction, we now use the time-dependent vertices $v(t) = v + m_v(t)$ given by the flow map from Equation (1) and thus the new vertex positions are

$$\tilde{v} = v + m_v(t_{k+1}).$$
If we assume for simplicity that \( m_v \) is differentiable, then Equation (8) writes as \( \|m_v(t_{k+1})\|_2 \leq \Delta t \sup_{t \in [t_k, t_{k+1}]} \|m'_v(t)\|_2 < \omega(\tau)/2 \) for all interface vertices \( v \in \mathcal{V}_\Gamma(t) \). This is a restriction on the timestep size \( \Delta t = t_{k+1} - t_k \), since \( m_v(t_k) = 0 \).

The exception (boundary conflict) points to an interaction of the domain boundary with the interface and has to be handled by a separate (sub-) model.

In Algorithm 3.5 no explicit remeshing is mentioned. However, every call of the remove, insert, and move Algorithms contains a local remeshing step. Executing all those remeshing operations guarantees that the resulting mesh is still Delaunay after moving the interface.

As mentioned before, Algorithm 3.5 gives a moving mesh with an \( \varepsilon \)-moving mesh interface. In other words, the interface is not exactly retained as polygonal shape. Instead, the interface \( \tau_{k+1}(t_{k+1}) \) resulting from moving and remeshing contains the interface \( \tau_k(t_{k+1}) \) in its \( \varepsilon \)-neighborhood, which is the interface resulting from the movement alone. The exact value of \( \varepsilon \) is not an input parameter but a result of the corresponding motion and is limited by the mesh width. Since remeshing in the \( \varepsilon \)-neighborhood is allowed, topology changes of the interface surface are in fact possible. Whenever topological changes occur, we cannot exclude junction points, which is not a problem for the IPMM but may be challenging for the numerical methods used together with the IPMM. An experimental extension of the provided implementation (Version 2.0) that supports topology changes is also available at Reference [1].

Since we keep track of deleted vertices in \( L_{\text{del}} \) the initial mesh returns (locally) to its initial configuration as soon as the interface has passed through and is sufficiently far away from the considered region. More precisely, a cell corresponds to a cell from the initial mesh if its circum-sphere does not contain any deleted (background) vertex from \( L_{\text{del}} \). In the two-dimensional case, those are all cells with circumsphere that have a distance of at least

\[
\max\left(\Delta x_{\text{min}}, \max_{\text{interface}\, \text{facet} \, S} \left(\text{length}(S)\right)\right)
\]

(9)

to the interface.
Using Algorithm 3.5, we are now able to define an interface $\Gamma_T(t)$ preserving moving mesh $T$. The steps involved are illustrated in Figure 7. Starting with a base-mesh $\tau_k$, we have a time-dependent sub-mesh $\tau_k(t)$ with interface $\Gamma_{\tau_k}(t)$ on the time interval $t \in [t_k, t_{k+1})$. At the end of this time interval, at $t_{k+1}$, we generate the new base-mesh $\tau_{k+1}$ from $\tau_k$ and $\Gamma_{\tau_k}(t_{k+1})$, by means of Algorithm 3.5.

### 3.2 Interface Refinement and Coarsening

During the interface motion, very large or very small interface facets can occur. This can have a negative impact on numerical simulations with moving interfaces. To retain a high-quality interface mesh, it is important to be able to refine and coarsen the interface mesh.

We present two algorithms for interface mesh refinement and coarsening in Algorithms 3.6 and 3.9, respectively. We note that these algorithms are able to run in conjunction with an adaptive mesh refinement/coarsening in the bulk mesh.

#### 3.2.1 Interface Refinement.

The interface $\Gamma_T$ is refined by refining its facets. To decide which facets should be refined, we introduce the threshold $\Delta x_{\Gamma_T, \text{max}} > 0$. If a facet has an edge that is larger than $\Delta x_{\Gamma_T, \text{max}}$, then it is refined by inserting the midpoint of the longest edge as an additional interface vertex.

**Algorithm 3.6: Refine Interface**

**Description:** Refine all interface facets that have an edge larger than the threshold value $\Delta x_{\Gamma_T, \text{max}} > 0$, while retaining the mesh interface.

**Parameters:** edge length threshold $\Delta x_{\Gamma_T, \text{max}} > 0$.

**Algorithm**

For all facets $S$ in $\Gamma_T$ do

1. Compute the maximum edge length $l_{\text{max}}$ of all edges $e$ of $S$.
2. If $l_{\text{max}} > \Delta x_{\Gamma_T, \text{max}}$: Insert (Algorithm 2.11) the midpoint of the longest edge of $S$ into the mesh.

The insertion of a new interface vertex usually results in remeshing. Nevertheless, we do not need to ensure the Gabriel property of the new interface facets, as opposed to the interface motion Algorithm 2.12, due to the following reason:

---

ACM Transactions on Mathematical Software, Vol. 50, No. 1, Article 5. Publication date: March 2024.
Fig. 8. Minimum covering sphere of the vertex set \{v_1, \ldots, v_5\}. The set \{v_1, \ldots, v_5\} is hereby the resulting hole in a two-dimensional interface after the removal of the interface vertex \(v_0\).

Before inserting the new interface facet there were no two bulk vertices from different bulk domains that were connected via an edge. This means that there is no witness sphere for any pair of such vertices. After the insertion of an additional interface vertex there are even more restrictions on the witness sphere and therefore the connecting edge cannot exist after the insertion either. This means that the interface surface is preserved according to Definition 3.3. In the two-dimensional setting the interface is even invariant according to the stronger Definition 3.2.

### 3.2.2 Interface Coarsening

To coarsen the interface, we remove single interface vertices from the mesh. To decide which facet should be coarsened, we introduce another threshold \(\Delta x_{\Gamma, \text{min}} > 0\). As for the interface refinement, we decide whether an interface vertex should be removed, if an interface edge is shorter than \(\Delta x_{\Gamma, \text{min}}\). In this case, we remove one of its vertices. Before removing the interface vertex, we require some mesh modifications to preserve the interface, which is guaranteed via the minimum covering sphere.

**Definition 3.7 (Minimum Covering Sphere).** Let a non-empty set of vertices \(V = \{v_1, \ldots, v_m\}\) be given. We call a sphere a *minimum covering sphere* of \(V\) if it is the smallest sphere that contains all vertices of \(V\) inside or on its boundary.

**Remark 3.8.** The minimum covering sphere is unique and can be computed in linear time \(O(m)\) [33].

Assume that an interface vertex \(v_0\) with adjacent interface vertices \(\{v_1, \ldots, v_m\}, m \geq d\) as in Figure 8 has to be deleted. Then there is a resulting hole in the interface mesh that will be remeshed automatically by the Delaunay mesh. *A priori*, we do not know how the resulting hole will be remeshed. However, we need to preserve the interface surface—in other words, we need to ensure that no two bulk vertices from two different phases will be connected via an edge running through the new resulting hole. We do this by the following simple approach:

- Build the minimum covering sphere containing the resulting hole, i.e., all vertices \(v_1, \ldots, v_m\).
- Remove all bulk vertices that are inside or on the minimum covering sphere.
The interface surface is then preserved. We first state the resulting Algorithm 3.9 and subsequently prove the preservation property within the following Theorem 3.11 and Corollary 3.12.

**ALGORITHM 3.9: Coarsen Interface**

**Description:** Coarsen all interface facets that have an edge smaller than the threshold value \( \Delta x_{T, \text{min}} > 0 \), while retaining the mesh interface.

**Parameters:** edge length threshold \( \Delta x_{T, \text{min}} > 0 \).

**Algorithm**

For all vertices \( v \) in \( \Gamma_T \) do

- Compute the minimum edge length \( l_{\text{min}} \) and average edge length \( l_{\text{avg}} \) of all interface edges \( e \) incident to \( v \).
- If \( l_{\text{min}} < \Delta x_{T, \text{min}} \): Write the vertex \( v \) and the average edge length \( l_{\text{avg}} \) into a sorted list \( L_{\text{coarsen}} \) with ascending \( l_{\text{avg}} \).

While \( L_{\text{coarsen}} \) is not empty do

- Take the first vertex \( v \) of \( L_{\text{coarsen}} \) and remove it from the list.
- Build the minimum covering sphere containing the resulting hole of \( v \) in \( \Gamma_T \), i.e., all incident interface vertices.
- Remove (Algorithm 2.10) all bulk vertices from the mesh that are inside and on the minimum covering sphere.
- Remove (Algorithm 2.10) the interface vertex \( v \) from the mesh.
- Update the list \( L_{\text{coarsen}} \): For all interface vertices \( \hat{v} \) adjacent to \( v \), compute the new minimum edge length \( l_{\text{min}} \) and the new average edge length \( l_{\text{avg}} \). If \( l_{\text{min}} \geq \Delta x_{T, \text{min}} \), delete the vertex \( \hat{v} \) from the list. Otherwise, update the position of the vertex \( \hat{v} \) in the list according to \( l_{\text{avg}} \).

**Remark 3.10.** Algorithm 3.9 works in arbitrary space dimensions and generalizes the following procedure in a two-dimensional mesh: In two space dimensions, we always get as resulting hole a segment \( \overline{v_1v_2} \) whose minimum covering circle is the Gabriel circle. Therefore, if all bulk vertices are removed from the Gabriel circle and additionally no other interface vertex lies inside the Gabriel circle, then the segment \( \overline{v_1v_2} \) has the Gabriel property and is thus part of the mesh. In this case the interface is obviously preserved. In three or more space dimensions the preservation of the interface is not as straightforward as in two space dimensions, since the minimum covering sphere generally does not coincide with the Gabriel sphere. However, the interface is still preserved, as we will show in Theorem 3.11 and Corollary 3.12 below.

Both the refining and the coarsening algorithm use the edge length as an indicator whether the interface should be refined or coarsened, however, the algorithms are not restricted to this case and other indicators such as the local curvature can be employed.

**Theorem 3.11 (Empty Minimum Covering Sphere).** Let a set of vertices \( \{v_1, \ldots, v_m\} \), \( m \geq d \) and its minimum covering sphere \( M \) be given. Furthermore, let two vertices \( p_1 \) and \( p_2 \) be given that are outside \( M \) with the segment \( \overline{p_1p_2} \) intersecting the convex hull of \( v_1, \ldots, v_m \) (see Figure 9 for the setting).

Then there is no witness sphere of the segment \( \overline{p_1p_2} \).

This leads directly to the desired property for interface preservation.
**Corollary 3.12 (Empty Minimum Sphere Preserves Interface).** Let a Delaunay mesh $\tau$ with vertex set $\mathcal{V}$ and interface $\Gamma_\tau$ be given. Let a subset $Q := \{v_1, \ldots, v_m\} \subset \mathcal{V}_\tau$, $m \geq d$, of interface vertices be given such that the minimum covering sphere does not contain any other vertices than $v_1, \ldots, v_m$. For any other two vertices $p_1, p_2 \in \mathcal{V} \setminus Q$ the following statement holds. If the segment $p_1p_2$ intersects the convex hull of $Q$, then the segment is not an edge in the Delaunay mesh $\tau$.

The corollary is important for Algorithm 3.9 in the following sense: When removing a vertex from the interface, we get a resulting hole in the interface mesh. We take the interface vertices of the resulting hole as the set $Q$ in the Corollary 3.12 and compute their minimum covering sphere. Corollary 3.12 ensures that no edge between two bulk vertices $p_1$ and $p_2$ crosses the newly meshed interface by removing all bulk vertices that are inside the minimum covering sphere.

**Proof.** We give the proof of Theorem 3.11 for $d = 3$. For the other cases, we refer to the remark below. The proof is carried out by contradiction. Assume that $p_1p_2$ has a witness sphere $W$. This implies that $W$ does not contain any of the points $v_1, \ldots, v_m$.

1. First, we know that $p_1p_2$ has an intersection with $\text{conv}((v_1, \ldots, v_m))$. Therefore, there is a facet of a triangulation of the convex hull, which intersects with $p_1p_2$—the special case that we hit an edge is discussed at the end of the proof. We call the vertices of this facet $v_a$, $v_b$, and $v_c$ (in Figure 9(a) this fact is, for example, given through $v_1$, $v_5$, and $v_3$). We denote the intersecting point by $v_{\text{int}}$. The intersecting point must then be both in the interior of the minimum covering sphere $M$ as well as in the interior of $W$.

2. The points $v_a$, $v_b$, and $v_c$ are outside the sphere $W$, since it is a witness sphere. The segment from $v_{\text{int}}$ to $v_a$ goes from the inside of $W$ to the outside of $W$ and is fully located inside $M$. Therefore, there is a point $\widehat{v}_a$ along the segment, which is on the sphere $W$ as well as inside the minimum covering sphere $M$. The same arguments hold for the vertices $v_b$ and $v_c$. Thus, there are vertices $\widehat{v}_a$, $\widehat{v}_b$, $\widehat{v}_c$ inside $M$ and on $W$.

3. For $p_1$ and $p_2$, we have by assumption on $M$ and by construction of $W$ that $p_1$, $p_2$ are outside $M$ and on $W$. 

**Fig. 9.** Exemplary sketch of (a) the setting and (b) the construction points of the proof of Theorem 3.11.
(4) From (2) and (3), we know that $p_1$ and $\hat{v}_a$ are on the sphere $W$, and thus, we can find a geodesic on $W$ connecting both points. Furthermore, $p_1$ is outside $M$ and $\hat{v}_a$ is inside $M$ and thus there must be an intersecting point of the geodesic with $M$. With this argument, we get six distinct points that are on both spheres $W$ and $M$, by relating $p_1$ with $\hat{v}_a$, $\hat{v}_b$, $\hat{v}_c$ and $p_2$ with $\hat{v}_a$, $\hat{v}_b$, $\hat{v}_c$. If two spheres coincide on four or more points in general position, then they must be equal. This implies that $W = M$, which is a contradiction to $W$ being a witness sphere, since $W = M$ contains all vertices $v_1, \ldots, v_m$.

In the special case that in step (1) the intersecting point $v_{\text{int}}$ lies on an edge and not a full facet, we get only two vertices $v_a$ and $v_b$ instead of three vertices. However, we can use that the points $p_1, p_2, v_a,$ and $v_b$ are coplanar. Here, the witness sphere and the minimum covering sphere cannot be shown to be equal (like in step (4)), but their cross-section circles coincide in the plane and contain the vertices $v_a$ and $v_b$. Therefore, even in this case, the segment $\overline{p_1p_2}$ cannot have a witness sphere.

One can see from the proof that both points $p_1$ and $p_2$ must be outside $M$. If only either $p_1$ or $p_2$ is outside $M$, we only get three points where the spheres coincide, which is not enough for concluding the equality.

Remark 3.13. The proof is independent of the spatial dimension. The only changes are that the number of vertices of a facet changes to $d$ and the number of equal points of $\partial W$ and $\partial M$ changes to $2 \cdot d$, which is obviously more than $d + 1$. If $d + 1$ points in general position coincide for two $d$-dimensional spheres, then they must be equal.

4 TEST CASES

For the implementation in two or three space dimensions, most frameworks can be applied that are capable of inserting and removing points from a Delaunay mesh. To implement the method, we choose the CGAL-framework [43], which provides these standard operations. For background on the technical details and the implementation within CGAL, we refer to References [2, 3, 8, 29, 35, 39, 50].

Our open source implementation of the moving mesh algorithm can be found at Reference [1]. To test the algorithms, we consider several test cases in two and three space dimensions. The algorithms 2.12, 3.6, and 3.9 require the thresholds $\Delta x_{\text{min}}, \Delta x_{\Gamma, \text{max}},$ and $\Delta x_{\Gamma, \text{min}}$. In all test cases, we choose

$$\Delta x_{\text{min}} = 0.9 h_{\text{min}}, \quad \Delta x_{\Gamma, \text{min}} = (1 - p) h_{\Gamma, \text{min}}, \quad \Delta x_{\Gamma, \text{max}} = (1 + p) h_{\Gamma, \text{max}},$$

with $h_{\text{min}}$ the minimum edge length of all bulk edges and $h_{\Gamma, \text{min}}$ and $h_{\Gamma, \text{max}}$ the minimum and maximum edge length of all interface edges, respectively. The percentage $p > 0$ is chosen as smallest value such that the conditions

$$\Delta x_{\Gamma, \text{max}} \geq 3 \Delta x_{\Gamma, \text{min}} \quad \text{and} \quad p \geq 0.1$$

are fulfilled. Note that for a sensible choice $\Delta x_{\Gamma, \text{max}}$ must be chosen bigger than twice $\Delta x_{\Gamma, \text{min}}$. Otherwise, the refinement of the interface would directly be undone in the interface coarsening step. Furthermore, $p$ must be positive to guarantee that the interface is neither refined nor coarsened if no size changes occur. To be robust in the presence of heterogeneous mesh sizes, we compute $h_{\text{min}}, h_{\Gamma, \text{min}},$ and $h_{\Gamma, \text{max}}$ as the 1% and 99% percentile of the edge lengths, respectively.

All computations in this section were performed sequentially on a desktop computer equipped with an AMD Ryzen Threadripper 2950X 16-core processor and 128 GB RAM. In the following, we present the test cases:
4.1 Star-shaped Deformation of a Sphere

In this first test case, we start with a given interface $\Gamma(0)$ and move its vertices according to a prescribed vector field

$$M : [t_0, t_{\text{end}}] \times \mathbb{R}^d \rightarrow \mathbb{R}^d : (t, x) \mapsto M(t, x),$$

with the time interval from $t_0 = 0$ to $t_{\text{end}} = 3$. In each timestep $t_{k+1} = t_k + \Delta t$ with step size $\Delta t > 0$, each interface vertex $v \in \mathcal{V}_f(t_k)$ is then moving according to

$$v(t_{k+1}) = v(t_k) + \Delta t M(t_k, v(t_k)).$$

We measure the performance of the interface moving Algorithm 3.5 in two and three space dimensions and compare it with the performance of the interface refinement and coarsening Algorithms 3.6, 3.9 as well as with the insertion and removal Algorithms 2.10, 2.11 for the bulk vertices.

4.1.1 The Two-dimensional Algorithm. For this test, we consider a moving interface $\Gamma(t)$ that starts as a circle with radius 0.5 and center in the origin. The simulations run from $t_0 = 0$ to $t_{\text{end}} = 3$ with timesteps $\Delta t = 2 \cdot 10^{-4}$. In each timestep all interface points are moved according to the vector field

$$M : [t_0, t_{\text{end}}] \times \mathbb{R}^2 \rightarrow \mathbb{R}^2 : (t, x) \mapsto -\sin(2\pi t) \cos\left(2\left\lfloor t \right\rfloor \text{atan2}(x_2, x_1)\right)x.$$

This results in a deformation of the circle into a star-shaped curve and returns into circular shape for integer-valued times $t$, as seen in Figures 10(a)–(c) for one simulation.

To test the algorithm, we generate meshes at several resolutions, parametrized by maximum edge lengths $\Delta x > 0$ during the generation. This example shows, as motivated in the introduction, that Algorithm 3.5 can handle strong interface deformations. The interface length in Figure 10 varies in a range between 3.14 and 5.22 while the quality of the surrounding mesh does not differ from the quality of the initial one. In fact, the surrounding mesh reverts to its initial configuration as soon as the distance to the interface is big enough (see Reference (9)).

The computational time needed for each timestep is shown in Figure 11. From those figures, one can see that the largest share of the runtime is asymptotically spent on the bulk coarsening. Interface and bulk modifications, which allow for these strong deformations, are comparatively inexpensive.

4.1.2 The Three-dimensional Algorithm. We generalize the setup from the previous example to three space dimensions. This time the moving interface $\Gamma(t)$ starts as a sphere with radius 0.5 and
An Interface-Preserving Moving Mesh in Multiple Space Dimensions

Fig. 11. Computational time per step for the star-shaped deformation example in two space dimensions. In the left figure the lines represent the 90% percentile of the data over all timesteps. The bars in the right figure indicate the mean computational time, which is split into the respective mesh operation contributions: move interface, refine/coarse bulk (in Algorithm 3.5), and refine/coarse interface.

Fig. 12. The moving interface (blue edges) for the star-shaped deformation example in three space dimensions. The plots show a slice through the computational domain at $x_3 = 0$ of an interface-preserving moving mesh with in average 627,000 cells.

Again, this results in a deformation of the sphere into a star-shaped object, as seen in Figures 12(a)–(c).

We consider different mesh resolutions, where the initial mesh was generated with several maximum edge lengths $\Delta x$.

The computational time needed for each timestep is shown in Figure 13.

We observe from Figure 13 that, again, as in the two-dimensional test case, the bulk coarsening will have asymptotically the largest proportion. The computations at hand were done with the implementation given in Reference [1], which serves as a comprehensible proof of concept. Consequently, for runtime optimizations one should tackle the bulk coarsening method first and...
Fig. 13. Computational time per step for the star-shaped deformation example in three space dimensions. In the left figure the lines represent the 90% percentile of the data over all timesteps. The bars in the right figure indicate the mean computational time, which is split into the respective mesh operation contributions: move interface, refine/coarse bulk (in Algorithm 3.5), and refine/coarse interface.

introduce more sophisticated data an interface-preserving moving mesh with in average 2,191,000 cells structures to achieve optimal runtime for the whole interface-preserving moving mesh.

4.2 Single Vortex Benchmark

A benchmark problem, which is widely used in the context of Volume-of-Fluid (VoF) methods with surface reconstruction, is the deformation of a sphere into a single vortex by a prescribed velocity field (see, e.g., Reference [38]).

4.2.1 The Two-dimensional Algorithm. In two space dimension the single vortex is induced by the velocity field

\[ M: [t_0, t_{\text{end}}] \times \mathbb{R}^2 \to \mathbb{R}^2 : (t, x) \mapsto \cos \left( \frac{\pi t}{t_{\text{end}}} \right) \left( 2 \cos(\pi x_1) \sin(\pi x_1) \sin^2(\pi x_2), 2 \cos(\pi x_2) \sin(\pi x_2) \sin^2(\pi x_1) \right) \]

with \( t_{\text{end}} = 8 \). The moving interface \( \Gamma(t) \) is initially a circle with radius 0.15 and center at \((0.5, 0.75)\) inside the domain \([0, 1]^2\). The velocity field transforms the initial circle to a spiral shape. The factor \( \cos \left( \frac{\pi t}{t_{\text{end}}} \right) \) reverts the velocity field at time \( t = 4 \) such that the interface returns to its initial circular shape by the time \( t_{\text{end}} = 8 \). The results computed with the timestep \( \Delta t = 10^{-4} \) are shown in Figures 14(a)–(d). In contrast to VoF methods with reconstruction, no breakup of the interface occurs by construction of the interface-preserving moving mesh, and the interface returns into almost perfect circular shape. The enclosed area at initial time \( t_0 \) and at final time \( t_{\text{end}} \) is 7.068 \( \cdot 10^{-2} \) and 7.070 \( \cdot 10^{-2} \), which gives an absolute error of 3.352 \( \cdot 10^{-5} \) and 2.489 \( \cdot 10^{-4} \) in the area, respectively. The signed deviation of all interface vertices from the circle is in a range from \(-2.720 \cdot 10^{-3}\) to \(3.141 \cdot 10^{-3}\) with a mean deviation of \(3.592 \cdot 10^{-4}\).

4.2.2 The Three-dimensional Algorithm. Similarly, in the three-dimensional setting the single vortex is induced by the velocity field

\[ M: [t_0, t_{\text{end}}] \times \mathbb{R}^3 \to \mathbb{R}^3 : (t, x) \mapsto \cos \left( \frac{\pi t}{t_{\text{end}}} \right) \left( 2 \cos(\pi x_1) \sin(\pi x_1) \sin^2(\pi x_2), 2 \cos(\pi x_2) \sin(\pi x_2) \sin^2(\pi x_1), 0 \right) \]
Fig. 14. The moving interface (blue edges) for the vortex benchmark in two space dimensions. The depicted example was computed on an interface-preserving moving mesh with in average 202,000 cells.

Fig. 15. The moving interface (blue surface) for the vortex benchmark in three space dimensions. The depicted example was computed on with $t_{end} = 6$. Again, the moving interface $\Gamma(t)$ is initially a sphere with radius 0.15 and center at $(0.5, 0.75, 0)$ inside the domain $[-1, 1]^2 \times [-0.5, 0.5]$. The velocity field transforms the initial sphere to a spiral shape by the time $t = 3$ and transforms it back to its initial spherical shape by the time $t_{end} = 6$. The resulting moving interface computed with the timestep $\Delta t = 2 \cdot 10^{-4}$ is depicted in Figures 15(a)–(d). Unlike in the two-dimensional case the strong deformation of the sphere causes changes of the interface surface. More precisely, interface facets may flip on the interface, which gives in combination with the interface remeshing a less precise resolution of the interface surface for the three-dimensional setting. Nevertheless, the final state is still in good agreement with the predicted spherical shape. The enclosed volume at initial time $t_0$ and at final time $t_{end}$ is $1.408 \cdot 10^{-2}$ and $1.380 \cdot 10^{-2}$, which gives an absolute error of $4.108 \cdot 10^{-3}$ and $2.373 \cdot 10^{-2}$ in the volume, respectively. The signed deviation of all interface vertices from the sphere is in a range from $1.196 \cdot 10^{-2}$ to $3.473 \cdot 10^{-3}$ with a mean deviation of $-1.044 \cdot 10^{-3}$.

In summary, this test case confirms that with the IPMM even extreme interface deformations are possible. The interface surface is directly resolved within the full-dimensional mesh, which bypasses surface reconstruction steps and easily enables the computation of curvature, e.g., by least-squares fitting spheres. Moreover, the interface surfaces stay connected for all times both in two and three space dimensions.

4.3 Circular Advection Example

In the previous examples, we presented some calculations, where the motion of the interface was given explicitly. In this final example, we will treat the case, where the motion of the interface depends dynamically on the numerical solution of an (advective) PDE on the full dimensional
Fig. 16. Two sliced disks that are rotated via circular advection. The initially upper disk is treated with the IPMM-FV method, whereas the initially lower disk is treated with the standard FV method. The first picture shows the initial state in a mesh with 32,708 cells and mesh edge lengths in a range from 0.0426 to 0.1266. The second picture shows an intermediate state at \( t = \pi / 4 \). The third picture shows the final state at \( t_{\text{end}} = \pi / 2 \) after a quarter rotation. For the computation, a fixed timestep \( \Delta t = 10^{-4} \) was used.

mesh. As such, the numerical data on the surrounding full-dimensional mesh interacts directly with the interface motion.

4.3.1 The Two-dimensional Algorithm. We regard the circular advection around the origin in two space dimensions. The model problem reads

\[
\frac{d}{dt} (u(t, x)) + \frac{d}{dx_1} (x_2 u(t, x)) + \frac{d}{dx_2} (x_1 u(t, x)) = 0, \quad \text{for } x \in \Omega, t \in (0, t_{\text{end}}), \quad u(0, x) = u_0(x), \quad \text{for } x \in \Omega.
\]

The exact solution is a counterclockwise rotation around the origin of the initial data \( u_0 \). As initial state, we choose an indicator function on a sliced disk domain, \( \Omega = [-4, 4]^2 \) and \( t_{\text{end}} = \pi / 2 \) (see also Figure 16). The setting is chosen similar to the seminal Zalesak benchmark problem [51].

To compute the cell data, as well as the interface motion, we will make use of a Finite Volume method on top of the interface-preserving moving mesh, which we will call interface-preserving moving mesh finite volume method (IPMM-FV method). The numerical approach is using standard monotone numerical fluxes for the transport term, which includes the flux in Equation (10) and for the flux due to geometric motion. The precise description of the method using an explicit Euler time-stepping can be found in Reference [13].

In Table 1, we list runtime, numerical error, and the experimental order of convergence (EOC) for the standard finite volume (FV) method versus the IPMM-FV method. The EOC is given by log\( (e_1/e_2) / \log(h_1/h_2) \) where \( e_1 \) and \( e_2 \) are the numerical errors on a mesh with mesh width \( h_1 \) and \( h_2 \), respectively, which gives an approximation to the order of convergence of the numerical methods. One can see from the table that the \( L^1 \)-error of the IPMM-FV method is very close to the initial \( L^1 \)-error, which results from the piecewise linear approximation of the interface. The FV method takes approximately half of the runtime of the IPMM-FV method on meshes with similar discretization widths. Nevertheless, even the FV method on the finest mesh does not reach the accuracy of the IPMM-FV method on the coarsest mesh (see also Figure 17).

4.3.2 The Three-dimensional Algorithm. By using the same set of equations (10), we can define the circular advection around the \( z \)-axis in three space dimensions.

We generalize the previous setting by choosing an indicator function on a sliced sphere as initial state and \( \Omega = [-4, 4]^2 \times [-2, 2] \).
Table 1. Comparison of the $L^1$-error at Time $t_{\text{end}} = \pi/2$ and Runtime of the IPMM-FV Method and of the FV Method for the Sliced Disk Benchmark Case Computed with a Fixed Timestep of $\Delta t = 10^{-4}$ in Two Space Dimensions

| Initial $L^1$-error | IPMM-FV method-- 2d | FV method-- 2d |
|----------------------|---------------------|----------------|
|                      | mesh width | $L^1$-error | time/step [s] | EOC | mesh width | $L^1$-error | time/step [s] | EOC |
| 4.9 · 10^{-2}        | 5.2 · 10^{-1} | 4.8 · 10^{-2} | 6.6 · 10^{-4} | 5.2 · 10^{-1} | 3.1 · 10^{0} | 2.8 · 10^{-4} |
| 2.4 · 10^{-2}        | 3.9 · 10^{-1} | 2.3 · 10^{-2} | 1.1 · 10^{-3} | 2.43 | 3.9 · 10^{-1} | 2.8 · 10^{0} | 5.3 · 10^{-4} | 0.34 |
| 7.2 · 10^{-3}        | 2.3 · 10^{-1} | 8.1 · 10^{-3} | 2.9 · 10^{-3} | 1.99 | 2.3 · 10^{-1} | 2.4 · 10^{0} | 1.6 · 10^{-3} | 0.31 |
| 3.8 · 10^{-3}        | 1.5 · 10^{-1} | 3.6 · 10^{-3} | 6.6 · 10^{-3} | 1.92 | 1.5 · 10^{-1} | 2.1 · 10^{0} | 4.1 · 10^{-3} | 0.34 |
| 2.0 · 10^{-3}        | 1.1 · 10^{-1} | 2.1 · 10^{-3} | 1.3 · 10^{-2} | 2.00 | 1.1 · 10^{-1} | 1.9 · 10^{0} | 7.9 · 10^{-3} | 0.33 |
| 9.7 · 10^{-4}        | 7.5 · 10^{-2} | 1.4 · 10^{-3} | 3.3 · 10^{-2} | 0.87 | 7.5 · 10^{-2} | 1.6 · 10^{0} | 2.5 · 10^{-2} | 0.39 |
| 2.0 · 10^{-4}        | 3.8 · 10^{-2} | 8.8 · 10^{-4} | 2.9 · 10^{-1} | 0.72 | 3.8 · 10^{-2} | 1.2 · 10^{0} | 2.4 · 10^{-1} | 0.42 |

The mesh widths are the average edge lengths that appear during the simulation.

Fig. 17. Comparison of the $L^1$-error in dependence of the mesh width for the benchmark cases from Table 1 and Table 2.

Again, we list runtime, numerical error, and EOC for the standard FV method versus the IPMM-FV method (see Table 2). In the three-dimensional setting, the IPMM-FV method takes between 2 to 12 times the runtime of the FV method on meshes with similar discretization widths. The finer the mesh is chosen, the closer the runtime gets for both methods. This indicates that the runtime complexity is similar to the one needed for the numerical operations. For more complex applications the runtime for the IPMM may even become negligible in comparison to further numerical computations. Nevertheless, we see that also in three space dimensions the FV method does not reach the accuracy of the IPMM-FV method.

In both compared methods a standard finite volume method in the bulk regions was used. Of course, the interface-preserving moving mesh is not limited to finite volume methods only but can also be used in combination with higher-order Galerkin methods, which is a subject of ongoing research.

5 CONCLUSIONS

Moving mesh methods are a key tool to enable the robust numerical solution of PDE-based free boundary value problems. We introduced in this contribution a new moving mesh (method) for tracking the motion of the free boundary interface as a codimension-1 manifold in a meshed bulk domain such that the manifold is represented for all times as a subset of facets of the bulk mesh. While the approach works for arbitrary bulk dimensions, an open source implementation for the two- and three-dimensional case can be found at Reference [1]. Remeshing by interface coarsening and refinement is integrated in the entire algorithm, preventing the formation of numerically
disadvantageous cells and thus ensuring a desired mesh resolution. Notably, the remeshing operations are performed in a local manner, which is even shown rigorously in Theorem 3.11 for the interface coarsening algorithm. In this way, our approach is radically different from the widely used Lagrangian approaches (see, e.g., Figure 1 as well as the interface-reconstruction approaches (e.g., piecewise linear interface construction (PLIC) in the context of VoF methods or cut-cell approaches)). The performance of the IPMM is confirmed by the results of the test cases in Section 4. Moreover, we show that it is capable to resolve strong interface deformations and large changes of the interface geometry, while always preserving the \((d - 1)\)-dimensional interface within the mesh. We provide evidence that the runtime complexity per timestep is similar to the one needed for the numerical operations.

However, the IPMM does not come without limitations. First, topology changes, especially the creation of new interfaces, are not supported out-of-the-box. Also, we cannot guarantee that some mesh quality metric is fulfilled in the vicinity of the interface, and slivers may appear (for \(d \geq 3\)). However, in the bulk phases, we retain the initial mesh quality, thanks to the background mesh. Finally, there is no direct way to parallelize the method for distributed meshes.

Nonetheless, the IPMM is easily adapted to modern solver techniques for PDE-based problems such as high-order finite element or discontinuous Galerkin methods. To be consistent with high-order numerics the geometrical error of the interface position should scale in the same way, leading to isoparametric representations of the interface facets. Future work will aim at exploiting the local nature of the algorithms to run the algorithms in parallel on a distributed mesh. Since dynamic (bulk) mesh adaption became a standard tool in the past decade, adaptive control of the moving mesh becomes desirable. In the current 3D implementation, the vertices on the domain boundary are fixed. This is a technical limitation that should be overcome to cover situations with interfaces transversally intersecting the domain boundary or the handling of time-dependent domains. Finally, our moving mesh propels the design of conforming numerical discretization methods for free boundary problems with inherent surface dynamics models like Marangoni convection.

### REFERENCES

1. Maria Alkämper and Jim Magiera. 2021. Interface Preserving Moving Mesh (Code). DOI: https://doi.org/10.18419/darus-1671
2. Pierre Alliez, Clément Jamin, Laurent Rineau, Stéphane Tayeb, Jean Tournois, and Mariette Yvinec. 2020. 3D mesh generation. In CGAL User and Reference Manual (5.0.1 ed.). CGAL Editorial Board. https://doc.cgal.org/5.0.1/Manual/packages.html#PkgMesh3

Table 2. Comparison of the \(L^1\) error at Time \(t_{\text{end}} = \pi/2\) and Runtime of the IPMM-FV Method and of the FV Method for the Sliced Sphere Benchmark Case Computed with a Fixed Timestep of \(\Delta t = 10^{-4}\) in Three Space Dimensions

| initial \(L^1\)-error | IPMM-FV method– 3d | FV method– 3d |
|------------------------|---------------------|--------------|
|                        | mesh width | \(L^1\)-error | time/step [s] | EOC | mesh width | \(L^1\)-error | time/step [s] | EOC |
| 2.0 \(\times 10^{-1}\) | 1.3 \(\times 10^0\) | 2.6 \(\times 10^{-1}\) | 2.4 \(\times 10^{-2}\) | 0.12 | 1.4 \(\times 10^0\) | 5.2 \(\times 10^0\) | 2.0 \(\times 10^{-3}\) | 0.28 |
| 1.9 \(\times 10^{-1}\) | 1.2 \(\times 10^0\) | 2.6 \(\times 10^{-1}\) | 2.6 \(\times 10^{-2}\) | 0.08 | 1.3 \(\times 10^0\) | 5.1 \(\times 10^0\) | 2.4 \(\times 10^{-3}\) | 0.28 |
| 1.9 \(\times 10^{-1}\) | 1.1 \(\times 10^0\) | 2.6 \(\times 10^{-1}\) | 2.6 \(\times 10^{-2}\) | 0.08 | 1.1 \(\times 10^0\) | 4.9 \(\times 10^0\) | 3.2 \(\times 10^{-3}\) | 0.28 |
| 1.6 \(\times 10^{-1}\) | 8.5 \(\times 10^{-1}\) | 2.6 \(\times 10^{-1}\) | 3.0 \(\times 10^{-2}\) | 0.04 | 8.6 \(\times 10^{-1}\) | 4.7 \(\times 10^0\) | 5.5 \(\times 10^{-3}\) | 0.16 |
| 1.2 \(\times 10^{-1}\) | 5.9 \(\times 10^{-1}\) | 2.3 \(\times 10^{-1}\) | 4.6 \(\times 10^{-2}\) | 0.34 | 5.8 \(\times 10^{-1}\) | 4.3 \(\times 10^0\) | 1.5 \(\times 10^{-2}\) | 0.22 |
| 5.8 \(\times 10^{-2}\) | 4.4 \(\times 10^{-1}\) | 1.9 \(\times 10^{-1}\) | 8.1 \(\times 10^{-2}\) | 0.60 | 4.4 \(\times 10^{-1}\) | 3.9 \(\times 10^0\) | 3.8 \(\times 10^{-2}\) | 0.33 |
| 3.0 \(\times 10^{-2}\) | 2.9 \(\times 10^{-1}\) | 1.2 \(\times 10^{-1}\) | 2.6 \(\times 10^{-1}\) | 1.15 | 2.9 \(\times 10^{-1}\) | 3.5 \(\times 10^0\) | 1.6 \(\times 10^{-1}\) | 0.27 |
| 7.6 \(\times 10^{-3}\) | 2.2 \(\times 10^{-1}\) | 9.5 \(\times 10^{-2}\) | 6.5 \(\times 10^{-1}\) | 0.84 | 2.2 \(\times 10^{-1}\) | 3.2 \(\times 10^0\) | 3.9 \(\times 10^{-1}\) | 0.32 |
| 7.6 \(\times 10^{-3}\) | 1.5 \(\times 10^{-1}\) | 7.4 \(\times 10^{-2}\) | 2.6 \(\times 10^0\) | 0.63 | 1.5 \(\times 10^{-1}\) | 2.8 \(\times 10^0\) | 1.4 \(\times 10^0\) | 0.34 |

The mesh widths are the average edge lengths that appear during the simulation.
[3] Pierre Alliez, Sylvain Pion, and Ankit Gupta. 2020. Principal component analysis. In *CGAL User and Reference Manual* (5.0.1 ed.). CGAL Editorial Board. Retrieved from https://doc.cgal.org/5.0.1/Manual/packages.html#PkgPrincipalComponentAnalysis

[4] Timothy Baker and Peter Cavallo. 1999. Dynamic adaptation for deforming tetrahedral meshes. In *Proceedings of the 14th Computational Fluid Dynamics Conference*. DOI: https://doi.org/10.2514/6.1999-3253

[5] Marshall Bern, Paul Chew, David Eppstein, and Jim Ruppert. 1995. Dihedral bounds for mesh generation in high dimensions. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA ’95)*. Society for Industrial and Applied Mathematics, 189–196. Retrieved from https://dl.acm.org/doi/10.1145/205555.313694

[6] Jean-Daniel Boissonnat, Olivier Devillers, and Samuel Hornus. 2009. Incremental construction of the Delaunay graph in medium dimension. In *Proceedings of the 25th Annual Symposium on Computational Geometry*. 208–216. DOI: https://doi.org/10.1145/1542362.1542403

[7] Jean-Daniel Boissonnat and Mariette Yvinec. 1998. *Algorithmic Geometry*. Cambridge University Press. DOI: https://doi.org/10.1017/CBO9781139172998

[8] Hervé Brönnimann, Andreas Fabri, Geert-Jan Giezeman, Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Stefan Schirra. 2020. 2D and 3D linear geometry kernel. In *CGAL User and Reference Manual* (5.0.1 ed.). CGAL Editorial Board. Retrieved from https://doc.cgal.org/5.0.1/Manual/packages.html#PkgKernel23

[9] Kevin Q. Brown. 1979. Voronoi diagrams from convex hulls. *Inform. Process. Lett.* 9, 5 (1979), 223–228. DOI: https://doi.org/10.1016/0020-0190(79)90074-7

[10] James C. Cavendish, David A. Field, and William H. Frey. 1985. An approach to automatic three-dimensional finite element mesh generation. *Int. J. Numer. Meth. Eng.* 21, 2 (1985), 329–347. DOI: https://doi.org/10.1002/nme.1620210210

[11] C. Chalons, P. Engel, and C. Rohde. 2014. A conservative and convergent scheme for undercompressive shock waves. *SIAM J. Numer. Anal.* 52, 1 (2014), 554–579. DOI: https://doi.org/10.1137/120897821

[12] Christophe Chalons, Jim Magiera, Christian Rohde, and Maria Wiebe. 2017. A finite-volume tracking scheme for two-phase compressible flow. In *Theory, Numerics and Applications of Hyperbolic Problems I*, Christian Klingenberg and Michael Westdickenberg (Eds.). *Springer Proceedings in Mathematics & Statistics*, 309–322. DOI: https://doi.org/10.1007/978-3-319-91545-6_25

[13] Christophe Chalons, Christian Rohde, and Maria Wiebe. 2017. A finite volume method for undercompressive shock waves in two space dimensions. *ESAIM: M2AN* 51, 5 (2017), 1987–2015. DOI: https://doi.org/10.1051/m2an/2017027

[14] Long Chen. 2004. Mesh smoothing schemes based on optimal Delaunay triangulations. In *Proceedings of the 13th International Meshing Roundtable*. 109–120.

[15] Siu-Wing Cheng, Tamal K. Dey, Herbert Edelsbrunner, Michael A. Facello, and Shang-Hua Teng. 2000. Sliver exudation. *J. ACM* 47, 5 (Sept. 2000), 883–904. DOI: https://doi.org/10.1145/355483.355487

[16] Kenneth L. Clarkson, Kurt Mehlhorn, and Raimund Seidel. 1993. Four results on randomized incremental construction. *Comput. Geom.* 3, 4 (1993), 185–212. DOI: https://doi.org/10.1016/0925-7701(93)90009-U

[17] Meizhong Dai and David P. Schmidt. 2005. Adaptive tetrahedral meshing in free-surface flow. *J. Comput. Phys.* 208, 1 (2005), 228–252. DOI: https://doi.org/10.1016/j.jcp.2005.02.012

[18] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars (Eds.). 2008. *Computational Geometry: Algorithms and Applications (3rd ed).* Springer. DOI: https://doi.org/10.1007/978-3-540-77974-2

[19] Boris Nikolaevich Delaunay. 1934. Sur la sphère vide. *Bulletin de l’Académie des Sciences de l’URSS. VII. Série* 1934, 6 (1934), 793–800.

[20] Olivier Devillers. 2002. On deletion in Delaunay triangulations. *Int. J. Comput. Geom. Appl.* 12 (2002), 193–205. DOI: https://doi.org/10.1142/S0218195902000815

[21] Olivier Devillers. 2009. *Vertex Removal in Two Dimensional Delaunay Triangulation: Asymptotic Complexity Is Pointless*. Research Report RR-7104. INRIA. Retrieved from: https://hal.inria.fr/inria-00433107

[22] Olivier Devillers and Monique Teillaud. 2003. Perturbations and vertex removal in a 3D Delaunay triangulation. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms*. 313–319. Retrieved from https://hal.inria.fr/inria-00166710

[23] Qiang Du, Vance Faber, and Max Gunzburger. 1999. Centroidal Voronoi tessellations: Applications and algorithms. *SIAM Rev.* 41, 4 (1999), 637–676. DOI: https://doi.org/10.1137/S0036144599352836

[24] Herbert Edelsbrunner, Xiang-Yang Li, Gary Miller, Andreas Stathopoulos, Dafna Talmor, Shang-Hua Teng, Alper Üngör, and Noel Walkington. 2000. Smoothing and cleaning up slivers. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC’00)*. Association for Computing Machinery, New York, NY, 273–277. DOI: https://doi.org/10.1145/335305.335338

[25] David A. Field. 1988. Laplacian smoothing and Delaunay triangulations. *Commun. Appl. Numer. Meth.* 4, 6 (1988), 709–712. DOI: https://doi.org/10.1002/cnm.1630040603

[26] K. Ruben Gabriel and Robert R. Sokal. 1969. A new statistical approach to geographic variation analysis. *System. Biol.* 18, 3 (Sept. 1969), 259–278. DOI: https://doi.org/10.2307/2412323
[27] Ami Harten and James M. Hyman. 1983. Self adjusting grid methods for one-dimensional hyperbolic conservation laws. *J. Comput. Phys.* 50, 2 (1983), 235–269. DOI: https://doi.org/10.1016/0021-9991(83)90066-9

[28] Weizhang Huang and Robert D. Russell. 2011. Adaptive Moving Mesh Methods (Applied Mathematical Sciences, Vol. 174). Springer, New York. DOI: https://doi.org/10.1007/978-1-4419-7916-2

[29] Clément Jamin, Sylvain Pion, and Monique Teillaud. 2020. 3D triangulations. In *CGAL User and Reference Manual* (5.0.1 ed.). CGAL Editorial Board. Retrieved from https://doc.cgal.org/5.0.1/Manual/packages.html#PkgTriangulation3

[30] Lutz Kettner, Kurt Mehlihorn, Sylvain Pion, Stefan Schirra, and Chee Yap. 2008. Classroom examples of robustness problems in geometric computations. *Comput. Geom.* 40, 1 (2008), 61–78. DOI: https://doi.org/10.1016/j.comgeo.2007.06.003

[31] Xiang-Yang Li. 2003. Generating well-shaped d-dimensional Delaunay Meshes. *Theoret. Comput. Sci.* 296, 1 (2003), 145–165. DOI: https://doi.org/10.1016/S0304-3975(02)00437-1

[32] Xiang-Yang Li and Shang-Hua Teng. 2001. Generating well-shaped Delaunay meshed in 3D. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA’01). Society for Industrial and Applied Mathematics, 28–37. Retrieved from https://dl.acm.org/doi/10.5555/365411.365416

[33] Nimrod Megiddo. 1983. Linear-time algorithms for linear programming in $R^3$ and related problems. *SIAM J. Comput.* 12, 4 (1983), 759–776. DOI: https://doi.org/10.1137/0212052

[34] Blair Perot and Ramesh Nallapati. 2003. A moving unstructured staggered mesh method for the simulation of incompressible free-surface flows. *J. Comput. Phys.* 184, 1 (2003), 192–214. DOI: https://doi.org/10.1016/S0021-9991(02)00027-X

[35] Sylvain Pion and Mariette Yvinec. 2020. 2D triangulation data structure. In *CGAL User and Reference Manual* (5.0.1 ed.). CGAL Editorial Board. Retrieved from: https://doc.cgal.org/5.0.1/Manual/packages.html#PkgMesh2

[36] Shaoping Quan, Jing Lou, and David P. Schmidt. 2009. Modeling merging and breakup in the moving mesh interface tracking method for multiphase flow simulations. *J. Comput. Phys.* 228, 7 (2009), 2660–2675. DOI: https://doi.org/10.1016/j.jcp.2008.12.029

[37] Shaoping Quan and David P. Schmidt. 2007. A moving mesh interface tracking method for 3D incompressible two-phase flows. *J. Comput. Phys.* 221, 2 (2007), 761–780. DOI: https://doi.org/10.1016/j.jcp.2006.06.044

[38] William J. Rider and Douglas B. Kothe. 1998. Reconstructing volume tracking. *J. Comput. Phys.* 141, 2 (1998), 112–152. DOI: https://doi.org/10.1006/jcph.1998.5906

[39] Laurent Rineau. 2020. 2D conforming triangulations and meshes. In *CGAL User and Reference Manual* (5.0.1 ed.). CGAL Editorial Board. Retrieved from https://doc.cgal.org/5.0.1/Manual/packages.html#PkgTDS2

[40] Hanan Samet. 2006. *Foundations of Multidimensional and Metric Data Structures* (1st ed.). Morgan Kaufmann.

[41] Richard Sanders. 1985. The moving grid method for nonlinear hyperbolic conservation laws. *SIAM J. Numer. Anal.* 22, 4 (1985), 713–728. DOI: https://doi.org/10.1137/0722043

[42] Dafna Talmor. 1997. *Well-spaced Points for Numerical Methods*. Ph.D. Dissertation, School of Computer Science, Carnegie Mellon University, Pennsylvania.

[43] The CGAL Project. 2020. *CGAL User and Reference Manual* (5.0.1 ed.). CGAL Editorial Board. Retrieved from https://doc.cgal.org/5.0.1/Manual/packages.html

[44] Jane Tournois, Rahul Srinivasan, and Pierre Alliez. 2009. Perturbing slivers in 3D Delaunay meshes. In *Proceedings of the 18th International Meshing Roundtable*, Brett W. Clark (Ed.). Springer, Berlin, 157–173. DOI: https://doi.org/10.1007/978-3-642-04319-2_10

[45] Jane Tournois, Rahul Srinivasan, and Pierre Alliez. 2009. Perturbing slivers in 3D Delaunay meshes. In *Proceedings of the 18th International Meshing Roundtable*, Brett W. Clark (Ed.). Springer, Berlin, 157–173. DOI: https://doi.org/10.1007/978-3-642-04319-2_10

[46] Jane Tournois, Rahul Srinivasan, and Pierre Alliez. 2009. Perturbing slivers in 3D Delaunay meshes. In *Proceedings of the 18th International Meshing Roundtable*, Brett W. Clark (Ed.). Springer, Berlin, 157–173. DOI: https://doi.org/10.1007/978-3-642-04319-2_10

[47] Zeljko Tukovic and Hrvoje Jasak. 2008. Simulation of free-rising bubble with soluble surfactant using moving mesh finite volume/area method. In *Proceedings of the 6th International Conference on CFD in Oil & Gas, Metallurgical and Process Industries*.

[48] Salih Ozen Unverdi and Grétar Tryggvason. 1992. A front-tracking method for viscous, incompressible, multi-fluid flows. *J. Comput. Phys.* 100, 1 (1992), 25–37. DOI: https://doi.org/10.1006/jcph.1992.90307-K

[49] Paul B. Yale (Ed.). 1968. *Simulation of free-rising bubble with soluble surfactant using moving mesh finite volume/area method*. In *Proceedings of the 18th International Meshing Roundtable*, Brett W. Clark (Ed.). Springer, Berlin, 157–173. DOI: https://doi.org/10.1007/978-3-642-04319-2_10

[50] Mariette Yvinec. 2020. 2D triangulation. In *CGAL User and Reference Manual* (5.0.1 ed.). CGAL Editorial Board. Retrieved from https://doc.cgal.org/5.0.1/Manual/packages.html#PkgTriangulation2

[51] Steven T. Zalesak. 1979. Fully multidimensional flux-corrected transport algorithms for fluids. *J. Comput. Phys.* 31, 3 (1979), 335–362. DOI: https://doi.org/10.1016/0021-9991(79)90051-2

Received 22 December 2021; revised 31 July 2023; accepted 18 September 2023

ACM Transactions on Mathematical Software, Vol. 50, No. 1, Article 5. Publication date: March 2024.