HexHex: Highspeed Extraction of Hexahedral Meshes

TOBIAS KOHLER, University of Bern, Switzerland
MARTIN HEISTERMANN, University of Bern, Switzerland
DAVID BOMMES, University of Bern, Switzerland

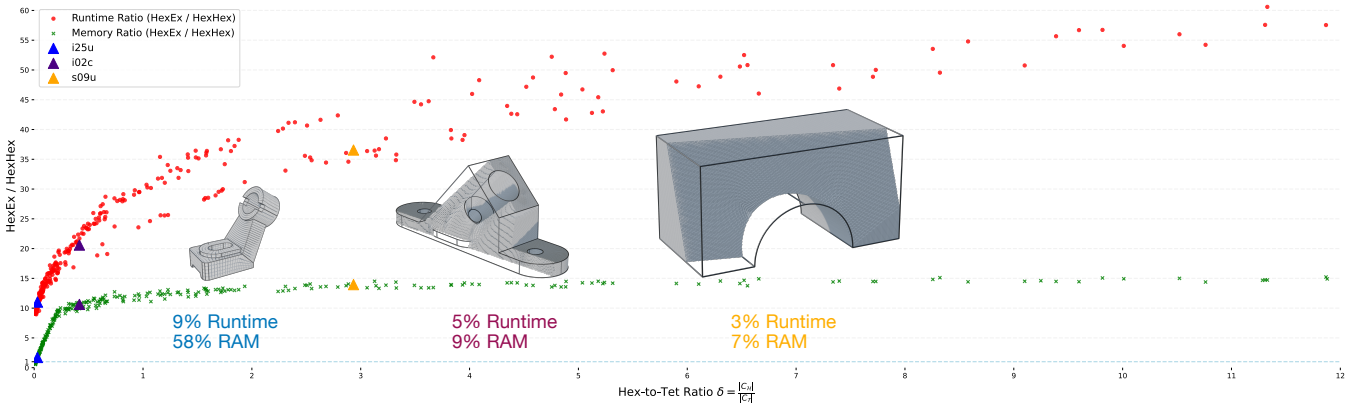


Fig. 1. When evaluated on the representative HexMe dataset [Beaufort et al. 2022], our novel **HexHex** algorithm extracts a hexahedral mesh from a tetrahedral mesh and locally injective integer-grid map significantly faster than the state-of-the-art **HexEx** [Lyon et al. 2016] while also using less memory. For coarse inputs where the resulting hex-mesh contains notably fewer hexahedral elements (= $C_\mathcal{H}$) than tetrahedral element in the input (= $C_\mathcal{T}$), HexEx takes up to 30 times longer than our HexHex. Since HexHex scales particularly well for large hex-to-tet ratios $\delta = \frac{|C_\mathcal{H}|}{|C_\mathcal{T}|}$, the speedup increases to an approximate factor of 50 for $\delta \approx 10$, while at the same time a reduction in peak-memory usage of over one order of magnitude can be observed.

Modern hexahedral mesh generation relies on integer-grid maps (IGM), which map the Cartesian grid of integer iso-surfaces to a structure-aligned and conforming hexahedral cell complex discretizing the target shape. The hexahedral mesh is formed by iso-surfaces of the map such that an extraction algorithm is needed to convert the *implicit* map representation into an *explicit* mesh. State-of-the-art algorithms have been designed with two goals in mind, i.e., (i) unconditional robustness and (ii) tolerance to map defects in the form of inverted or degenerate tetrahedra. Because of significant advancements in the generation of locally injective maps, the tolerance to map defects has become irrelevant. At the same time, there is a growing demand for efficiently handling significantly larger mesh complexities, unfortunately not well served by the state-of-the-art since the tolerance to map defects induces a high runtime cost. Consequently, we present HexHex, a novel (unconditionally robust) hexahedral mesh extraction algorithm for locally injective integer-grid maps designed for maximal performance and scalability. Key contributions include a novel and highly compact mesh data structure based on so-called *propellers* and a conservative rasterization technique, significantly reducing the number of required exact predicate tests. HexHex not only offers lower asymptotic runtime complexities from a theoretical perspective but also lower constants, enabling in practice a 30x speedup for medium-sized examples and a larger speedup for more complex inputs, specifically when the hex-to-tet ratio is large. We provide a C++ reference implementation, supporting multi-core parallelization and the extraction of curved (piecewise-linear) hexahedral mesh edges and faces, e.g., valuable for subsequent higher-order mesh generation.

CCS Concepts: • **Applied computing** → *Computer-aided design*; • **Computing methodologies** → **Volumetric models**.

Additional Key Words and Phrases: hexahedral meshing, integer-grid map, optimization

Authors' Contact Information: Tobias Kohler, University of Bern, Bern, Switzerland, tobias.kohler@unibe.ch; Martin Heistermann, University of Bern, Bern, Switzerland, martin.heistermann@unibe.ch; David Bommes, University of Bern, Bern, Switzerland, david.bommes@unibe.ch.

1 Introduction

Hexahedral meshes are a popular choice for volumetric domain discretization [Pietroni et al. 2022], specifically for simulation in engineering applications with a high demand for performance and accuracy. For high-quality hexahedral mesh generation, there is a lack of automatic algorithms such that semi-manual workflows, where a human designs a coarse block decomposition, are still the industry standard. However, Integer-Grid Map (IGM) based algorithms [Liu et al. 2018; Nieser et al. 2011] are a promising research direction since state-of-the-art methods [Liu and Bommes 2023] are becoming increasingly robust while delivering high-quality block decompositions comparable to those manually designed by humans.

Since an IGM encodes the hexahedral mesh only *implicitly* through its integer iso-contours in the co-domain of the map, as explained in more detail in Section 3.2, an algorithm is needed to extract the *explicit* hexahedral mesh. Compared to standard contouring of implicit iso-surfaces, e.g. through Marching Cubes [Lorensen and Cline 1987] or Dual Contouring [Ju et al. 2002], there are two

major differences, which complicate the task: (i) IGMs correspond to an arrangement of many iso-contours where explicit intersection resolution is required, and (ii) singularities of the hexahedral mesh necessitate a map representation with coordinate charts and transition functions, typically inducing numerical discontinuities in floating-point representations [Brückler et al. 2022; Mandad and Campen 2019].

The state-of-the-art HexEx algorithm [Lyon et al. 2016] offers a robust solution through a sanitization pre-process to resolve numerical discontinuities in combination with exact geometric predicates, and explicit tracing of intersection curves (edges of the hexahedral mesh). The algorithm guarantees to correctly extract the uniquely defined connectivity of a *locally injective* piecewise-linear IGM, i.e. a map defined through a tetrahedral mesh, which does not contain degenerate or inverted elements in domain or co-domain. Without local injectivity there are no longer strong guarantees but the HexEx algorithm attempts to tolerate local defects in a process that annihilates multiply-covered regions caused by inversions. Such a repair mechanism has been highly valuable during the earlier stages of IGM research, where local injectivity of the map could often not be ensured, specifically close to (noisy) singular arcs. Unfortunately, the tolerance to defects requires redundant storage and tracing of edges in the corresponding dart representation, which compromises runtime and memory consumption.

Table 1. Runtimes for models with increasingly more hexahedral cells in the mesh generation pipeline. Listed are the model name, the number of hex-cells, and the runtime for generating an IGM and the mesh extraction as well as the contribution of the latter in the entire pipeline. Our method performs significantly better for meshes with many hex-cells, no longer making the extraction a bottleneck. Note that the underlying optimization problem to generate an IGM actually becomes easier for meshes with more hex-elements, hence its runtime decrease.

Mesh	$\lvert\mathcal{H}\rvert$	IGM [s]	Extract. [s]	Extract. [s] (ours)
i25u	6653	1035.35	3.07 (0.16%)	0.65 (0.03%)
i02c	431752	720.14	48.88 (3.00%)	1.58 (0.10%)
s09u	998730	41.71	105.21 (30.18%)	2.27 (0.92%)
s17c	7903126	28.14	3366.59 (94.59%)	9.72 (4.81%)

One key advantage of IGM-based algorithms over alternative techniques is that the map construction time does not directly depend on the complexity of the output hexahedral mesh but is mostly influenced by the structural complexity of the block decomposition. Hence, generating hexahedral meshes with millions of cells – often required for practical simulation applications – is comparatively cheap. Unfortunately, in those scenarios, the hexahedral mesh extraction, where the runtime does directly depend on the output mesh complexity, becomes a critical bottleneck of the IGM-based hexahedral mesh generation pipeline as demonstrated in Table 1 for a few examples.

Thanks to major advances in the generation of locally injective maps [Garanzha et al. 2021; Nigolian et al. 2023, 2024] and IGMs [Jiang et al. 2013; Li et al. 2012; Liu and Bommes 2023], the handling of defects during the mesh extraction has become irrelevant, offering potential for acceleration. Consequently, we propose the novel hexahedral mesh extraction algorithm HexHex, which under the assumption of local injectivity significantly outperforms HexEx in terms of runtime and memory consumption, without compromising robustness. For example, as seen in Table 1, for s17c with nearly 8 million hex-elements, the runtime of the mesh extraction is reduced from nearly one hour, contributing 95% to the entire pipeline's runtime, to less than 10 seconds, contributing only 5% to the pipeline's runtime.

HexHex extracts a hex mesh through the following three main (plus optional fourth) steps.

(1) *Preprocessing*: Numerical inaccuracies in the parametrization due to the solver's numerical tolerance are eliminated.
(2) *Geometry Extraction*: Hex-vertices are extracted by conservatively rasterizing each tet.
(3) *Connectivity Extraction*: Propellers corresponding to directed hex-edges are extracted per generator and then traced once through the parametrization to their opposites.
(4) *Piecewise Linear Extraction (Optional)*: Piecewise-linear edge arcs, and piecewise-linear face patches are generated during the connectivity extraction.

The main performance gain in comparison to HexEx result from (i) a specifically designed conservative rasterization technique for the vertex extraction (Section 5.2), (ii) a novel propeller data structure employed in the connectivity extraction, effectively eliminating the redundancy of the dart representation (Section 4.2), and (iii) taking advantage of locally constant connectivity configurations (Section 5.3). In Section 5.6 we show that compared to HexEx the asymptotic runtime complexity is significantly reduced. More precisely, assuming a tetrahedral mesh of constant complexity, the runtime of HexHex depends linearly on the number of hexahedral cells, while the dependency is quadratic for HexEx.

Our C++ reference implementation[1] offers multi-threading, includes a command line application, a set of simple API functions, and an *OpenFlipper* [Möbius and Kobbelt 2012] plugin. The optional extraction of piecewise-linear edge arcs and face patches provides a valuable basis for the fitting of higher-order hexahedral meshes guided by the IGM.

2 Related Work

Hexahedral mesh generation and its one-dimensional lower counterpart of quadrilateral mesh generation are actively researched topics. In the following, we will focus on previous work closely related to our contributions, and refer the reader for a broader overview to the available excellent surveys [Armstrong et al. 2015; Bommes et al. 2013b; Campen 2017; Pietroni et al. 2022].

Integer-Grid Maps. The notion of an integer-grid map (IGM) has been introduced in [Bommes et al. 2013a] in the context of quadrilateral mesh generation. However, the underlying idea of parametrization with coordinate charts, where the transition functions are grid automorphisms, originates from the earlier work of [Bommes et al. 2009; Kälberer et al. 2007; Ray et al. 2006].

The concept of IGMs has been extended to volumetric domains [Nieser et al. 2011] for hexahedral mesh generation. Early attempts

[1] https://github.com/cgg-bern/libHexHex

suffered from non-meshable singularity or feature configurations, often leading to degeneracies in the IGM [Liu et al. 2018]. Through local meshability repair of singular arcs [Jiang et al. 2013; Li et al. 2012] and singular nodes and geometric features [Liu and Bommes 2023], the robustness of volumetric IGM techniques has been evolved to a practically relevant level. Recent major advances in the robust generation of volumetric maps [Garanzha et al. 2021; Hinderink et al. 2024; Hinderink and Campen 2023; Nigolian et al. 2023, 2024] ensure that in case of meshable singularities, a defect-free IGM can be obtained through the generation of a motorcycle complex [Brückler et al. 2022], robust quantization [Brückler et al. 2022; Brückler et al. 2024], and elimination of zero-quantized cells [Brückler and Campen 2023].

Quad/Hex Mesh Extraction. QEx [Ebke et al. 2013] is the state-of-the-art algorithm for the robust extraction of a quadrilateral mesh from a given integer-grid map. It introduced the idea of eliminating numerical discontinuities of IGMs represented in floating-point precision, which is called *sanitization*. The sanitization step is essential to enable provably-correct and unique geometric decisions with the help of exact predicates [Attene 2020; Richard Shewchuk 1997], e.g. when tracing edges of the quad mesh in the co-domain of the IGM. Subsequently, in [Mandad and Campen 2019] the idea of sanitization has been extended to an algorithm, which projects a floating-point configuration x, satisfying linear constraints $Ax = b$ only up to some numerical tolerance $||Ax - b|| < \epsilon$ onto a close-by floating-point configuration $\bar{x}$ with exact constraint satisfaction $||A\bar{x} - b|| = 0$.

The volumetric counterpart of QEx for hexahedral mesh extraction is HexEx [Lyon et al. 2016]. Both algorithms are conceptually similar and – in addition to robustness – attempt to tolerate local defects of an integer-grid map, QEx trough vertex merging, and HexEx through even more powerful dart annihilation. By the best of our knowledge, QEx and HexEx are the only published and publicly available mesh extraction algorithms, which is probably related to the fact that their implementation is nontrivial and requires the handling of a multitude of different geometric configurations.

Our novel mesh extraction algorithm Hex^2 is strongly inspired by QEx and HexEx. By including the sanitization step into our algorithm, we inherit the robustness of state-of-the-art mesh extraction algorithms. However, we deviate from QEx and HexEx by requiring local injectivity of the IGM, i.e., not tolerating map defects. Relying on the advances in the robust generation of volumetric IGMs, we instead target maximal performance.

3 Terminology

In this section, we introduce the required definitions for a mesh and an integer-grid map (IGM). From now on, we use the abbreviation *Hex^2* (= Hex · Hex) for our algorithm to visually better differentiate it from HexEx.

3.1 Mesh

For our purposes, a (volumetric) *mesh* is a 3-dimensional polytopal complex $\mathcal{M} = (V, E, F, C)$ of (0−dimensional) vertices V, each with a position $\boldsymbol{p}(v) \in \mathbb{R}^3$, (1−dimensional) edges E, (2−dimensional) faces F, and (3−dimensional) cells C, collectively referred to as *entities*. In contrast to surface meshes, volumetric meshes also represent the interior. For simplicity, $\mathcal{M}$ refers to the mesh itself or the set $V \cup E \cup F \cup C$ of its entities. We write $\sigma < \rho$ if the dimensionality of $\sigma \in \mathcal{M}$ is less than that of $\rho \in \mathcal{M}$.

An entity σ is *incident* to an entity ρ of higher dimensionality if it is entirely part of the boundary of ρ and the two entities σ and ρ are *adjacent* if they have the same dimensionality and share a common entity of one lower dimensionality on their boundary. We write both relations as $\sigma \sim \rho$. As a small abuse of this definition, we consider a cell incident to itself.

We only consider manifold meshes whose incidence relations are conforming and that do not contain any isolated entities.

The *cell valence* $\mathrm{val}_c(e) \in \mathbb{N}_{\geq 1}$ of an edge $e \in E$ is the number of its incident cells. Analogously, $\mathrm{val}_f(e) \in \mathbb{N}_{\geq 2}$ is the number of faces incident to an edge. They only differ for boundary edges where $\mathrm{val}_f(e) = \mathrm{val}_c(e) + 1$.

A tetrahedral mesh or *tet-mesh* $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T}, F_\mathcal{T}, C_\mathcal{T})$ is a simplicial complex, meaning each cell $c \in C_\mathcal{T}$ is a tetrahedron consisting of four triangular faces.

In a hexahedral mesh or *hex-mesh* $\mathcal{H} = (V_\mathcal{H}, E_\mathcal{H}, F_\mathcal{H}, C_\mathcal{H})$, each cell $c \in C_\mathcal{H}$ is a hexahedron consisting of six quadrilateral faces, and a *singular edge* is defined as any inner edge or boundary edge with cell valence other than four or two, respectively. Otherwise, the edge is called *regular*.

Our algorithm relies on *OpenVolumeMesh (OVM)* [Kremer et al. 2013] as a data structure for meshes which extends the ideas of *OpenMesh* [Botsch et al. 2002] to the third dimension. Each edge is split into two directed edges [Campagna et al. 1998], and each face is split into two half-faces, each with an incident cell if it is not on the boundary of the mesh. Conventionally, the four vertices of a tet are positively oriented, and the vertices of its four half-faces are oriented such that the tet volume is the intersection of the respective defined half-spaces.

3.2 Integer-Grid Map

The idea of *parametrization* based hexahedral meshing is to map a tetrahedral mesh onto the regular cartesian grid such that the preimage of the intersection of the integer grid with the parametrized tet-mesh defines a boundary-aligned, conforming hexahedral mesh. The task of computing this preimage is trivial for globally injective parametrizations where topologically adjacent tets remain geometrically adjacent. However, such a mapping would not allow singularities and consequently hurt mesh quality. To support singularities, an integer-grid map allows for non-identity transitions between adjacent tets, leading to gaps (for valences less than four) or overlaps (for valences greater than four) in the case of inner edges. This makes extracting the mesh connectivity nontrivial as it requires a careful tracing through the parametrization while regarding the transitions as explained in Sections 5.3 and 5.4.

A 3d *integer-grid map (IGM)* is the union of per-cell affine mappings $\boldsymbol{f} = \{\boldsymbol{f}_c : \mathbb{R}^3 \to \mathbb{R}^3\}_{c \in C_\mathcal{T}}$ that map the four vertex positions of each tet $c \in C_\mathcal{T}$ to four respective *parameters*. The preimage of the intersection of the parametrization with the regular cartesian grid induces the hex-mesh as visualized in Figure 2. A position can be mapped to different parameters in different cell charts which are related by the *transition functions* τ_{ij} that map the chart of a tet c_i to

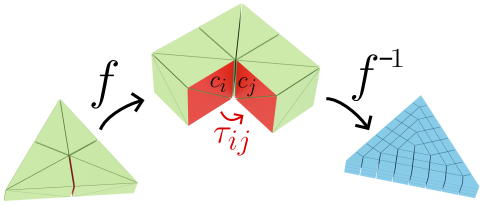


Fig. 2. A tet-mesh is mapped onto the regular integer-grid, which implies the hex-mesh structure by the map's preimage. To allow for a resulting singular hex-edge of valence 3 in the center, the mesh is cut open around it resulting in non-identity transitions (red) between adjacent cells.

the chart of an adjacent tet c_j, meaning $\tau_{ij}(\boldsymbol{f}_{c_i}(p)) = \boldsymbol{f}_{c_j}(p)$ for any point p shared by both parametrized tets. If not specified, $\boldsymbol{f}$ is the per-cell mapping of an arbitrary cell incident to the parametrized simplex and we write $\boldsymbol{f}(v)$ instead of $\boldsymbol{f}(\boldsymbol{p}(v))$ and $\boldsymbol{f}(\sigma)$ instead of $(\boldsymbol{f}(v_1), ..., \boldsymbol{f}(v_d))$ for the d vertices of σ. In the context of integer-grid maps, a *singular edge* is either an inner or boundary tet-edge where the sum of parametric dihedral face angles around it is different than 2π or π respectively, and a *singular vertex* is a vertex incident to other than zero or two singular edges [Lyon et al. 2016].

As explained in [Brückler et al. 2022; Liu et al. 2018; Lyon et al. 2016], an IGM satisfies the following four constraints.

(IGM1) *Conformity* The transition functions are automorphisms of the form $\tau_{ij}(u) = R_{ij}(u) + t_{ij}$ where $t_{ij} \in \mathbb{Z}^3$ is an integer translation and $R_{ij} \in O$ is one of the 24 octahedral rotations that preserves orientations. Since $\tau_{ji} = \tau_{ij}^{-1}$ it follows that $R_{ji} = R_{ij}^{-1}$ and $t_{ji} = R_{ji}(-t_{ij})$.

(IGM2) *Boundary integer alignment* For all boundary faces $f \in \partial F$ there exist $z \in \mathbb{Z}, a, b, c \in \mathbb{R}^2, R \in O$ such that $\boldsymbol{f}(f) = (R(z, a), R(z, b), R(z, c))$.

(IGM3) *Singularity integer alignment* For all *singular* edges e there exist $z \in \mathbb{Z}^2, a, b \in \mathbb{R}, R \in O$ such that $\boldsymbol{f}(e) = (R(z, a), R(z, b))$ and $\boldsymbol{f}(v) \in \mathbb{Z}^3$ for all *singular* vertices v.

(IGM4) *Local injectivity* The per-cell mappings have consistent, positive orientation, meaning $\text{ori3D}(\boldsymbol{f}(c)) > 0$ for all tets $c \in C_{\mathcal{T}}$ where $\text{ori3D}(A, B, C, D) = \text{sgn}(\det([A - D \mid B - D \mid C - D]))$ describes the side on which one point lies compared to the half-space defined by the other three.

An IGM satisfying the first three constraints only approximately, due to the numerical tolerance of the solver that generated it, and not necessarily satisfying the local injectivity constraint is referred to as a *relaxed integer-grid map* by [Lyon et al. 2016] and serves as the parametrization in HexEx. The omission of (IGM4) allows for flipped (ori3D < 0) or degenerate (ori3D $= 0$) tets, causing numerous hurdles like multiple hex-vertices being extracted at the same parameter and flipped regions in the parametrization that need to be fixed in a dart annihilation postprocessing step by [Lyon et al. 2016].

Based on the achievements by [Liu and Bommes 2023], Hex^2 expects (IGM4) to always be satisfied. We refer to such an IGM as a *locally injective relaxed integer-grid map*, as the first three constraints might still only be satisfied approximately.

4 Data Structures

We introduce the dart data structure used in HexEx and the new propeller data structure used in Hex^2.

4.1 The Darts of HexEx

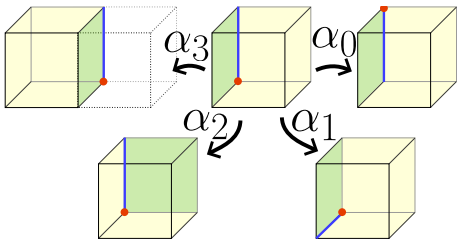


Fig. 3. A dart (v, e, f, c) consists of a vertex (red), edge (blue), face (green) and cell (yellow) that are pairwise incident. It is connected to the four unique darts that each share but one entity.

HexEx defines the extracted hex mesh using the *dart* data structure [Kraemer et al. 2014]. A dart is a 4−tuple $(v, e, f, c) \in V \times E \times F \times C$ whose entities are pairwise incident. Each dart stores four connections to other darts, α_0 to α_3, where $\alpha_i(d)$ refers to the unique dart which shares all the entities with d except the i−dimensional entity. For example, $\alpha_0((v, e, f, c)) = (v', e, f, c)$ is the dart consisting of the same edge, face, and cell, but different vertex. For boundary faces, α_3 is ignored. As each hex consists of six faces, each face of four edges and each edge of two vertices, there are $6 \cdot 4 \cdot 2 = 48$ darts per hex and $48|C_{\mathcal{H}}|$ darts with $4 \cdot 48 = 192|C_{\mathcal{H}}|$ connections stored in total.

4.2 The Propellers of HexHex

Intuitively, a propeller is half of an edge. A pair of a propeller and one of its blades are a quarter of a hex-face or quad-corner. A triple of a propeller and two consecutive blades is an eighth of a hex-cell or hex-corner.

More formally, for a mesh $\mathcal{M} = (V, E, F, C)$, we define the set of $2|E|$ *propellers* as the directed edges

$$\mathcal{P} = \{p_{v,e} = (v, e) \in V \times E : v \sim e\}$$

where each propeller stores $1 + \text{val}_f(e)$ connections to other propellers. Each propeller stores one connection to its *opposite*

$$\text{opposite}(p_{v,e}) = p_{v',e} \in \mathcal{P} \text{ s.t. } v \neq v'$$

which is the propeller corresponding to the same edge but different vertex. Additionally, let $f^0_{v,e}, f^1_{v,e}, ..., f^{\text{val}_f(e)-1}_{v,e}$ be the faces ordered around a directed edge such that $f^i_{v,e}$ and $f^{i+1}_{v,e}$ are incident to a common cell. Then, the i-th *blade*, $\text{blade}_i(p_{v,e})$, of a propeller $p_{v,e}$ is the propeller on the same vertex but different edge such that the two edges are incident to the shared face $f^i_{v,e}$.

$$\text{blade}_i(p_{v,e}) = p_{v,e^i_v} \in \mathcal{P} \text{ s.t. } e \neq e^i_v \wedge e^i_v \sim f^i_{v,e}$$

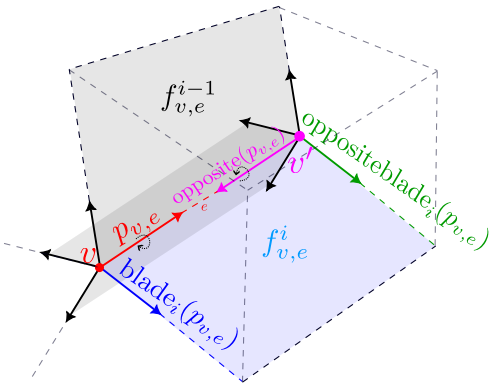


Fig. 4. A propeller (red) on a vertex v is connected to its opposite (pink) on a common (valence 4) edge e but different vertex v'. Its $i-$th blade (blue) encloses a shared face $f_{v,e}^{i}$ with the corresponding opposing blade (green). $f_{v,e}^{i-1}$ and $f_{v,e}^{i}$ are incident to a common hex-cell (gray, dashed).

For each edge, a unique *connection offset* $l_e \in \{0, ..., \text{val}_f(e) - 1\}$ is defined such that

$$f_{v,e}^{i} = f_{v',e}^{(l_e - i)}$$

This offset exists if we enforce the ordering of blades to always be counterclockwise (or to always be clockwise) around a propeller. The last and first faces are incident to a common cell only if the edge does not lie on the boundary or if $\text{val}_f(e) = 2$. The *i-th opposite blade* of a propeller $p_{v,e}$ is the blade of its opposite that encloses a shared face with its $i-$th blade and is implicitly given by the opposite and blade connections as follows:

$$\text{oppositeblade}_i(p_{v,e}) = \text{blade}_{(l_e - i)}(\text{opposite}(p_{v,e}))$$

All indices are to be interpreted as cyclic modulo the face valence or number of blades.

4.3 Properties

A propeller corresponding to a half-edge (v, e) essentially combines $2 \cdot \text{val}_c(e)$ darts into a single object and corresponds to the orbit $\langle \alpha_2, \alpha_3 \rangle(d)$ [Kraemer et al. 2014] of a dart $d = (v, e, \cdot, \cdot)$, meaning it combines all darts reachable via α_2 (different face) or α_3 (different cell) connections.

To describe a hexahedral mesh, we need $2|E_\mathcal{H}|$ propellers, and there are $2 \sum_e \text{val}_f(e) + 1 = 8|F_\mathcal{H}| + 2|E_\mathcal{H}|$ interconnections since each quad face consists of four edges. Consider an infinite regular hexahedral mesh where each hex-vertex is an inner vertex with 8 incident cells. Since there are 3 incident faces per incident cell, and 2 incident edges per incident face, a hex-vertex has $8 \cdot 3 \cdot 2 = 48$ incident darts. It has 6 incident edges, resulting in only 6 incident propellers per hex-vertex. Furthermore, per hex-vertex, there are $4 \cdot 48 = 192$ dart connections and $6 \cdot (4+1) = 30$ propeller connections. Consequently, the usage of propellers requires $\frac{48}{6} = 8$ times less objects and $\frac{192}{30} = 6.4$ less connections compared to darts. This highlights the primary advantage of this data structure for our application, although in a finite mesh with the presence of boundary vertices, these factors are slightly lower.

Whereas darts can directly describe nonmanifold meshes, this is not the case for propellers but could be remedied by storing whether or not two consecutive blades are incident to a shared cell. Furthermore, although we use propellers to describe a hexahedral mesh only, they could also be applied to other polyhedral meshes. These observations are summarized in Table 2.

In both the dart and the propeller data structures, only the vertices are stored explicitly, whereas the other entities are stored implicitly via a collection of darts or propellers per vertex.

Table 2. Comparison of the dart data structure and the propeller data structure defining a hex-mesh in terms of some basic properties. The factors to compare the two different units of measurements (cells for darts and edges, faces for propellers) consider an infinite regular grid.

Property	Darts	Propellers	$\frac{\text{Darts}}{\text{Propellers}}$
`Nonmanifolds`	✓	✓	
`Polyhedral meshes`	✓	✓	
`#Objects (hex)`	$48\|C_\mathcal{H}\|$	$2\|E_\mathcal{H}\|$	8
`#Connections (hex)`	$192\|C_\mathcal{H}\|$	$8\|F_\mathcal{H}\| + 2\|E_\mathcal{H}\|$	6.4

5 Algorithm

We now explain the individual steps of the Hex^2 pipeline in more detail. In Section 5.1, we explain how and why the input has to be preprocessed. In Section 5.2, Section 5.3, and Section 5.4, we go over the extraction process consisting of a vertex extraction for which we developed a conservative rasterization algorithm based on standard techniques, and a connectivity extraction for which we use the propellers. For both, we highlight how it improves upon HexEx in terms of performance. Lastly, in Section 5.5, we present the optional extraction of piecewise-linear elements.

The input for our algorithm is a tetrahedral mesh $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T}, F_\mathcal{T}, C_\mathcal{T})$ and a locally injective relaxed integer grid map $\boldsymbol{f} = \{\boldsymbol{f}_c\}_{c \in C_\mathcal{T}}$, and the output is the induced hexahedral mesh $\mathcal{H} = (V_\mathcal{H}, E_\mathcal{H}, F_\mathcal{H}, C_\mathcal{H})$. For clarity, some illustrations are in $2d$.

To clarify the differences and improvements from HexEx to Hex^2, we will, for each subtask, give a brief explanation on how it is executed in HexEx and why there are possible improvements, before explaining our version in more detail.

5.1 Preprocessing

The preprocessing phase consists of caching some frequently used properties as well as a *sanitization* of the integer-grid map that is necessary because it fulfills (IGM1) to (IGM3) only approximately due to the solver's numerical tolerance. This phase remains mostly the same compared to HexEx, apart from utilizing multiple threads for each substep and a more efficient parameter propagation if no nontrivial transitions are present around a vertex. Normally, previous steps of the state-of-the-art hex-meshing pipeline already compute the edge valences and rotations. In that case, the values are not recalculated by Hex^2.

5.1.1 Extracting the Transition Functions. The parametrization is provided solely as parametric position per tet corner, thus requiring

a computation of the transition functions. Given the vertex parameters u_{i1}, u_{i2}, u_{i3} and u_{j1}, u_{j2}, u_{j3} of a face in the charts of two adjacent tets $c_i, c_j \in C_{\mathcal{T}}$, the transition τ_{ij} from c_i to c_j is implicitly provided as

$$R_{ij} = \text{argmin}_{R \in O}(\|u'_{j2} - R(u'_{i2})\|^2 + \|u'_{j3} - R(u'_{i3})\|^2)$$
$$t_{ij} = \text{round}(u_{j1} - R_{ij}(u_{i1}))$$

where $u'_{kl} = u_{kl} - u_{k1}$ shift the first vertex parameters to the origin. In most cases (around 90% on average), the vertex parameters are the same in both charts making this computation unnecessary and τ_{ij} is set to the identity.

5.1.2 Computing Singularities. As defined in Section 3.2, singularities in the IGM are defined by the sum of parametric dihedral face angles around an edge. Computing this sum $\alpha_e \in \mathbb{R}$ for each edge $e \in E_{\mathcal{T}}$ is straightforward, and the edge is deemed singular if a hex-edge extracted on it would be singular.

$$\text{singular}(e) := \begin{cases} \text{round}(\frac{2\alpha_e}{\pi}) \neq 4, & e \text{ is inner} \\ \text{round}(\frac{2\alpha_e}{\pi}) \neq 2, & e \text{ is boundary} \end{cases}$$

5.1.3 Sanitization. If the input is not already an exact IGM, it is still possible that transforming a parameter from one chart to an adjacent one does not result in the correct parameter, i.e., $\tau_{ij}(u_i) \neq u_j$. To remedy this, the strategy of [Ebke et al. 2013] is applied in which the vertex parameter in the chart of an incident cell is truncated according to the maximum coordinate of a vertex parameter over all incident cell charts and then propagated to all incident charts according to the respective transition functions. This propagation is optimized in Hex^2 as follows: When a vertex does not have any incident nonidentity transitions, its parameter is set directly in all incident tet charts instead of wandering between incident tets and applying an accumulated transition function. To ensure the exact fulfillment of (IGM2) and (IGM3), singular vertices, edges, and boundary faces are snapped to the nearest integer point, -line and -plane respectively. After the exact fulfillment of (IGM1) to (IGM4) is guaranteed, to make the algorithm robust against the inaccuracies of floating point arithmetic, we employ *exact predicates* as provided by [Richard Shewchuk 1997] for all the various geometric tests required in the following phases of the algorithm. Specifically, all tests build on the implementation of *ori3D* that determines the orientation of four $3d$ points (cf. (IGM4)).

5.2 Vertex Extraction

An integer point $z \in \mathbb{Z}^3$ intersecting a parametrized tet-element $\sigma \in \mathcal{T}$ implies a hex-vertex $v \in V_{\mathcal{H}}$. This simplex is referred to as the *generator* of the hex-vertex by [Lyon et al. 2016]. For the later local connectivity extraction presented in Section 5.3, it is crucial that we know the generator per hex-vertex as it reduces the number of possible cases to consider. A generator of a hex-vertex can be a tetrahedral vertex, edge, face or cell as visualized in Figure 5. We first present how the vertices are extracted in HexEx, and how we use the principles of rasterization and a `Generator` function to reduce the number of exact predicate checks and, consequently, the runtime of the geometry extraction step.

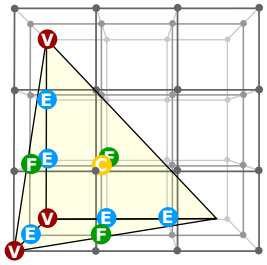


Fig. 5. A single parametrized tet cell which intersects 12 integer points on either its vertices (V, red), edges (E, blue), faces (F, green), or in its interior (C, yellow), corresponding to 12 hex-vertices. The generator of a hex-vertex is the simplex of the tet mesh whose parametric interior the corresponding integer-grid point intersects. There are three vertex generators, three edge generators (two of which both contain two points), three face generators, and one cell generator.

HexEx: To prevent duplicate hex-vertices from being extracted due to intersecting multiple tet images on their boundaries, HexEx iterates over all vertices, edges, faces, and cells separately and excludes their respective boundaries. Let us formally define the *parametric volume* of a simplex $\sigma \in \mathcal{T}$ as the convex hull of its vertex parameters

$$F_c(\sigma) = \text{Conv}(\boldsymbol{f}_c(\sigma))$$

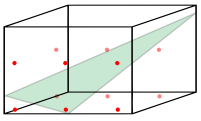
and its *parametric interior* as the parametric volume without its boundary

$$\mathring{F}_c(\sigma) = F_c(\sigma) \setminus \bigcup_{\substack{\rho < \sigma \\ \rho \sim \sigma}} F_c(\rho)$$

Algorithm 1 HexEx: Vertex Extraction

```
for σ ∈ V_T ∪ E_T ∪ F_T ∪ C_T do
    c ← any c ∈ C_T s.t. σ ~ c
    if σ ∈ E_T then
        for z ∈ F̊_c(σ) ∩ ℤ³ do
            extract hex-vertex with edge generator σ and position f_c^{-1}(z).
    if σ ∈ V_T ∪ F_T ∪ C_T then
        for z ∈ AABB(F_c(σ)) ∩ ℤ³ do
            if z ∈ F̊_c(σ) then
                extract hex-vertex with generator σ and position f_c^{-1}(z).
```

For each simplex $\sigma \in \mathcal{T}$ and each parameter $z \in \mathring{F}(\sigma) \cap \mathbb{Z}^3$, a hex-vertex with generator σ is extracted at position $\boldsymbol{f}^{-1}(z)$. This position is not unique if the cell chart is degenerate. For faces and cells, HexEx tests every integer-grid point in the axis-aligned bounding box (AABB) of the parametrized simplex, resulting in a large number of unnecessary exact predicate checks.



For example, consider a triangle with a normal direction far from any coordinate axis. Its bounding box will be significantly larger than the triangle, meaning a lot of points are tested even though a "randomly positioned" triangle in $3d$ space will very rarely intersect any integer point. Such an example is shown in the inset figure where a triangle does not intersect any of the integer locations within its bounding box. Even a tet with three integer-aligned faces only fills one-sixth of its bounding box. Additionally, since faces are tested independently of cells, HexEx uses a special

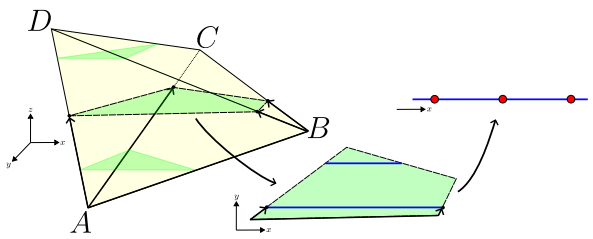


Fig. 6. In Hex^2, a tetrahedron in $3d$ is rasterized by sweeping along one axis. The resulting intersections with an integer-grid plane are $2d$ polygons. These are rasterized analogously resulting in $1d$ intervals on which the enumeration of integers is trivial.

exact predicate function, requiring up to 7 orientation checks per face and integer point. A cell always requires 4 orientation checks per point, one w.r.t. each half-face, an edge requires up to 3, and a vertex requires none since one only needs to check if the vertex parameter is an integer. The algorithm is shown in Algorithm 1. Note that the axis-aligned bounding box of a single vertex is just the point itself. For edges, one coordinate axis is chosen in which the $1d$ integers are linearly interpolated to the $3d$ line, rounded to the nearest integer-grid point, and tested against the line with a specific exact predicate.

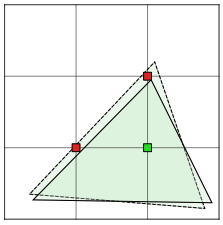
Hex^2: We reduce the set of candidate integer-grid points through a conservative rasterization approach, which iterates in parallel over only the tetrahedra. Each vertex, edge, face and cell $\sigma \in V_{\mathcal{T}} \cup E_{\mathcal{T}} \cup F_{\mathcal{T}} \cup C_{\mathcal{T}}$ is assigned to an arbitrary incident tet denoted by $\texttt{ParentTet}(\sigma) \in C_{\mathcal{T}}$ during the preprocessing phase to prevent duplicate hex-vertices from being extracted. Due to the local injectivity, there is no ambiguity when mapping back an integer parameter to the mesh domain. The high-level algorithm is shown in Algorithm 2.
For numerical robustness, we increase the candidate set of integer-grid points by enlarging our tets by some global tolerance $\varepsilon \geq 0$ as visualized in Figure 7 in $2d$. A candidate point is then checked using an exact predicate. In the following, we explain the algorithm in more detail.

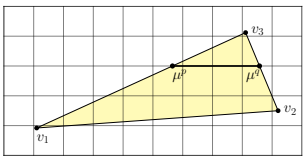
Algorithm 2 HexHex: Vertex Extraction

```
1: parallel for c ∈ C_T do
2:     if AABB(F_c(c)) ∩ ℤ³ ≠ ∅ then
3:         for z ∈ F_c(c) ∩ ℤ³ do                    // conservative rasterization
4:             if (σ ← Generator(z, c)) ≠ ⊥ and ParentTet(σ) = c then
5:                 extract hex-vertex v with generator σ, parameter f_c(v) = z, and position f_c^{-1}(z).
6:
```

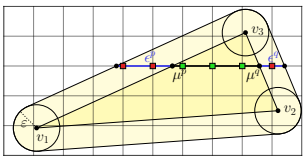
5.2.1 Rasterization. In our context, rasterization refers to efficiently enumerating all integer points that intersect a tetrahedron, as conceptually visualized in Figure 6. For each tet $c \in C_{\mathcal{T}}$, we first check if the axis-aligned bounding box of its parametric volume contains any integer-grid point. If there is at least one such candidate point, the axis of sweeping is chosen such that the number of scanplanes is minimized, meaning it is equal to the coordinate in which the extent of the integer bounding box of the tet image is minimal. Without loss of generality, we assume this to be the z−axis and the four points $A, B, C, D \in \mathbb{R}^3$ of the tet to be sorted in descending order such that $A_z \leq B_z \leq C_z \leq D_z$. The level of sweeping starts at $z = \lceil A_z \rceil$ and is increased by one after each step until $z = \lfloor D_z \rfloor$. At every level, the points of the intersection polygon are computed. The intersection of the integer-grid plane with constant z−coordinate $z \in \mathbb{Z}$ with the tetrahedron is a (convex) quadrilateral if $B_z < z < C_z$ and a triangle otherwise, except if the tet does only touch the plane with one or two of its vertices. This polygon is rasterized analogously along the y−axis where at each level $y \in \mathbb{Z}$, the intersection of an integer-grid line and the polygon is a line segment from (x_l, y, z) to (x_r, y, z). Due to the convexity of a tetrahedron, the points in between are also contained in the tet and do not need to be tested.

5.2.2 Conservative Rasterization. The geometric computations of the rasterization as explained above are inexact due to the limited precision of floating point operations. These numerical errors can lead to missing candidate integer points (see inset figure). Simply padding the intersection line segments by a nonnegative value is not enough as entire scanlines might be missing when an integer-grid point lies within a tet, but below or above the computed intersection polygon (cf. upper red vertex in inset figure).

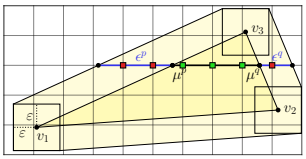




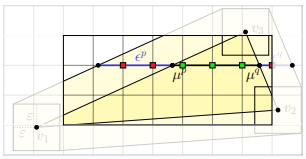
(a) The intersection of an integer-aligned plane with a tet.



(b) The tet is enlarged to the convex hull of the ε−balls around its vertices.



(c) ε−boxes are considered as a simple, conservative approximation. Consequently, the intersection polygon p, q is enlarged by $\epsilon^p, \epsilon^q \geq 0$.



(d) The candidate set is clamped to the integer bounding box of the tet.

Fig. 7. For robustness and performance, the tetrahedron is conservatively enlarged by some $\varepsilon \geq 0$. For clarity, the tet is visualized in $2d$ given by vertices v_1, v_2, v_3.

To remedy this, we conceptually enlarge a tetrahedron (Figure 7a) such that the distance between the faces of the tet and the faces of the inflated tet is equal to a global tolerance $\varepsilon \geq 0$. This corresponds to the convex hull of the ε−balls around the tet-vertices (Figure 7b). As a conservative approximation of this inflated tet, and in particular to avoid square root computations, we instead consider the convex hull of the ε−cubes (Figure 7c). The rasterized region is clamped to the

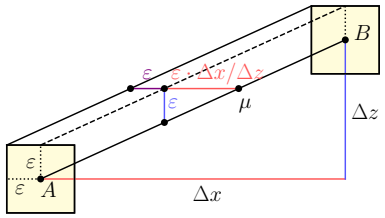


Fig. 8. The ϵ corresponding to the distance, in x, of the intersection point μ on the tet-edge to the intersection point on the enlarged tet's edge is computed using similar triangles.

integer bounding box of the tet (Figure 7d) to prevent superfluous predicate tests.

If the intersection point of an integer plane $z \in \mathbb{Z}$ with a tetrahedron lies in the parametric interior of a tet-edge $A, B \in \mathbb{R}^3$, meaning w.l.o.g. $A_z < z < B_z$, it is computed as an $\epsilon-$rectangle $p = (\mu^p, \epsilon^p)$ given by a center

$$\mu^p_{xy} = A_{xy} + \frac{z - A_z}{B_z - A_z} \cdot (B_{xy} - A_{xy})$$

and half size

$$\epsilon^p_{xy} = \varepsilon + \varepsilon \cdot \frac{|B_{xy} - A_{xy}|}{B_z - A_z}$$

as visualized in Figure 8 in $2d$ for clarity. The intersection point of the conservatively enlarged tet's edge with the integer plane then corresponds to one of the four extreme points of the $\epsilon-$rectangle p. In the case where an intersection point corresponds exactly to a tet-vertex $A \in \mathbb{R}^2 \times \{z\}$ and does not lie in the parametric interior of a tet-edge, we instead set $(\mu_{xy}, \epsilon_{xy}) = (A_{xy}, (0, 0))$.

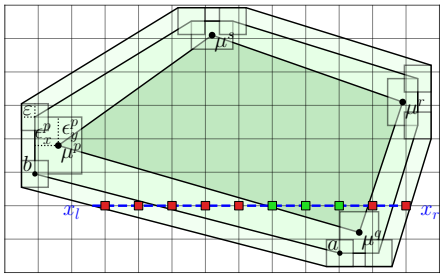


Fig. 9. The computed intersection polygon of a plane with a tet is given by its $\epsilon-$rectangles p, q, r, s. This inflated polygon, enlarged again by the global tolerance ε, is rasterized analogously to the tetrahedron.

To compute the intersection interval of an $\epsilon-$polygon given by three or four $\epsilon-$rectangles, as visualized in Figure 9, with an integer-grid line $y \in \mathbb{Z}$, the individual intersections of the line with each polygon's edge $E = (p, q)$ are computed. There are three cases:

(1) E does not intersect the line. Then, the intersected range is empty.

(2) The $y-$ranges of the $\epsilon-$rectangles p, q overlap, meaning the edge is close to parallel to the $x-$axis. Then, we set

$$x^E_l = \min(\mu^p_x - \epsilon^p_x, \mu^q_x - \epsilon^q_x)$$
$$x^E_r = \max(\mu^p_x + \epsilon^p_x, \mu^q_x + \epsilon^q_x)$$

(3) Otherwise, the intersection is computed analogously to the tet-case. Let $a, b \in \mathbb{R}^2$ be two extreme points of the segment's convex hull (w.l.o.g. $a_y < b_y$). Then we compute

$$x^E_{lr} = \left(a_x + \left[\frac{y - a_y}{b_y - a_y}\right]_{0,1} \cdot (b_x - a_x)\right) \mp \left(\varepsilon + \varepsilon \cdot \frac{|b_x - a_x|}{b_y - a_y}\right)$$

where $[\cdot]_{0,1}$ means clamping a value between 0 and 1.

The entire intersection interval of an $\epsilon-$polygon with the integer-grid line corresponds to the maximum range of intersection points over the relevant set of edges.

$$(x_l, x_r) = (\min_E x^E_l, \max_E x^E_r)$$

Since this conservative approach entails that we generally enumerate more points than are contained in the tet, we subsequently verify with exact predicates whether or not a candidate integer point lies within the tetrahedron. To that end, a parameter $z \in \mathbb{Z}^3$ in the chart of tet $c \in C_{\mathcal{T}}$ is checked using the function

$$\texttt{Generator} : \mathbb{Z}^3 \times C_{\mathcal{T}} \to \mathcal{T} \cup \{\bot\}, (z, c) \mapsto \sigma$$

which returns the unique generator simplex $\sigma \in \mathcal{T} \cup \{\bot\}$ incident to c such that the parameter z is integer and contained in the parametric interior of σ, meaning $z \in \mathring{F}_c(\sigma) \cap \mathbb{Z}^3$, or $\bot$ if the parameter is outside the image of the tet.

Since a tetrahedron corresponds to the intersection of four half-spaces given by its four triangular half-faces, the four respective orientation tests of z against each half-space already provide the generator simplex, as described in more detail in Section A.1, which eliminates the need for checking vertices, edges and faces separately. The number of exact predicate checks can be further reduced by exploiting convexity of the tetrahedron, i.e., after identifying the leftmost and rightmost interior integer points of a scanline, all integer points in-between are guaranteed to be interior as well.

5.2.3 Choice of Epsilon. By default we choose $\varepsilon = 10^{-6}$, which is empirically motivated by the evaluation presented in Section 6.2.1. On the one hand, we did not observe any failure cases for $\varepsilon > 10^{-17}$, while on the other hand the runtime overhead for $\varepsilon < 10^{-2}$ is negligible. Hence, $\varepsilon = 10^{-6}$ offers sufficient numerical margin in both directions to avoid undesired practical behavior. Importantly, for increasing ε we never generate candidate points outside of the integer bounding box of a tetrahedron such that in the limit case of $\varepsilon = \infty$ there is an effective fallback to the candidate set of HexEx, which is guaranteed to be correct. Note that the practically unlikely case of an insufficiently large ε can always be detected in the subsequent connectivity extraction and is easily resolved by re-running the vertex extraction with $\varepsilon = \infty$. While such fallback to $\varepsilon = \infty$ was never necessary in all our experiments, it is nevertheless crucial to guarantee robustness without requiring assumptions on ε.

The result of our geometry extraction is a list $\mathcal{G}$ of generators and a

list $V_{\mathcal{H}}(g)$ of hex-vertices per generator. Each hex-vertex v has a position $\boldsymbol{p}(v) \in \mathbb{R}^3$ in the mesh domain and one parameter $\boldsymbol{f}_c(v) \in \mathbb{Z}^3$ in the co-domain for each tet c incident to its generator.

5.3 Local Connectivity Extraction

The extracted hex-vertices define the (linear) hex-mesh's geometry, but we are still missing any information about the mesh's connectivity, the hex-edges, -faces and -cells. We split the connectivity extraction into two parts. The *local connectivity* refers to the local structure of outgoing incident hex-elements in a neighborhood around each hex-vertex. The *global connectivity* (Section 5.4) then refers to the interconnections between vertices.

HexEx: In HexEx, the local connectivity is expressed via darts (Section 4.1) which are enumerated on each hex-vertex. We define an integer-grid edge (unit length line segment), -face (unit square) and -cell (unit cube), respectively originating from a parameter $z \in \mathbb{Z}^3$ and extending in orthonormal axis aligned directions d_1, d_2, d_3 as

$$\mathcal{E}_{d_1}(z) = \{z + t_1 d_1 : 0 < t_1 < 1\}$$
$$\mathcal{F}_{d_1,d_2}(z) = \{z + t_1 d_1 + t_2 d2 : 0 < t_1, t_2 < 1\}$$
$$C_{d_1,d_2,d_3}(z) = \{z + t_1 d_1 + t_2 d2 + t_3 d_3 : 0 < t_1, t_2, t_3 < 1\}$$

For any hex-vertex $v \in V_{\mathcal{H}}$ with generator $g \in \mathcal{T}$ and parameter $z \in \mathbb{Z}^3$ in the chart of tet $c \in C_{\mathcal{T}}$ incident to g, an outgoing hex-edge is implied if

$$F_c(c) \cap \mathcal{E}_{d_1}(z) \neq \emptyset \tag{IGE}$$

an outgoing hex-face is implied if

$$F_c(c) \cap \mathcal{F}_{d_1,d_2}(z) \neq \emptyset \tag{IGF}$$

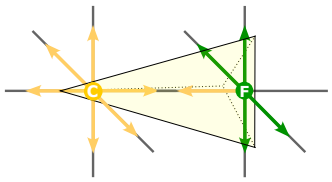
and an outgoing hex-cell is implied if

$$F_c(c) \cap C_{d_1,d_2,d_3}(z) \neq \emptyset \tag{IGC}$$

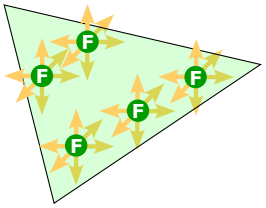
For each hex-vertex $v \in V_{\mathcal{H}}$, each tet $c \in C_{\mathcal{T}}$ incident to its generator g and each triple of orthonormal axis aligned directions d_1, d_2, d_3, a dart, corresponding to a tuple $(v, e, f, c) \in V_{\mathcal{H}} \times E_{\mathcal{H}} \times F_{\mathcal{H}} \times C_{\mathcal{H}}$ of incident hex-entities is extracted if the three aforementioned conditions, IGE, IGF and IGC, are satisfied.

Hex^2: Our approach for the local connectivity extraction uses the propeller data structure as defined in Section 4.2. We notice that it suffices to explicitly extract the local connectivity on a small subset of hex-vertices only. This is due to two observations, illustrated in Figure 10, and explained in the following, which we collectively refer to as the *constant local connectivity property*.

5.3.1 Constant Local Connectivity Property. First, all hex-vertices with a cell generator are locally identical. Every such vertex is an inner vertex with six outgoing hex-edges, one per axis-aligned direction in the parametrization, and eight incident hex-cells surrounding it. Similarly, all hex-vertices extracted on a boundary face generator are locally identical up to an octahedral rotation due to (IGM2). They will have five outgoing hex-edges and four incident hex-cells. The two cases are visualized in Figure 10a. We consider the information implicitly given and skip the local connectivity extraction for these types. Since most inner hex-vertices are extracted strictly inside cells and most boundary hex-vertices have boundary face



(a) All cell generators are locally identical, having six propellers extending into the cell (yellow). All boundary face generators are locally identical, having one propeller extending into the cell (yellow) and four into the face (green).



(b) All points on the same generator are locally identical.

Fig. 10. Illustration of the constant local connectivity property.

generators, skipping the local connectivity extraction for the two types significantly reduces the number of hex-vertices that need to be processed. Second, the local connectivity is the same for every point on a generator, as shown in Figure 10b for a face generator. For any fixed generator g and direction d, the integer-grid edge $\mathcal{E}_d(z)$ extends into the same incident simplex ρ, no matter the origin $z \in \mathring{F}_c(\sigma)$. This means it suffices to extract the local connectivity once per generator instead of per hex-vertex, which makes this step of the algorithm independent of the number of hex-elements per tet-element.

The local connectivity extraction consists of three substeps: The propeller extraction (Section 5.3.2) provides information about outgoing hex-edges, the blade enumeration (Section 5.3.3) identifies the propeller pairs that span hex-faces and the corner enumeration (Section 5.3.4) identifies the propeller triples that span hex-cells.

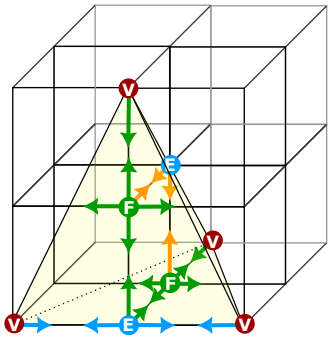


Fig. 11. Different types of propellers based on their holder, the simplex into which they extend. Within the parametrized tet are 4 propellers extending into edges (blue), 12 into faces (green), and 4 into the cell (yellow).

5.3.2 Propeller Extraction. For each tet $c \in C_{\mathcal{T}}$ incident to the generator g of an arbitrary hex-vertex $v \in V_{\mathcal{H}}(g)$, we test for each of the six axis-aligned directions $d_1 \in \mathcal{D} := \{\pm(1,0,0)^\top, \pm(0,1,0)^\top, \pm(0,0,1)^\top\}$ whether the integer-line segment originating from the parameter $z = \boldsymbol{f}_c(v)$ of the hex-vertex in the chart of c and going in direction d_1 intersects the

parametric volume of c (Equation (IGE)) using a function

$$\texttt{Holder} : \mathcal{G} \times \mathbb{Z}^3 \times \mathcal{D} \times C_{\mathcal{T}} \to \mathcal{T} \cup \{\perp\}, (g, z, d_1, c) \mapsto \rho$$

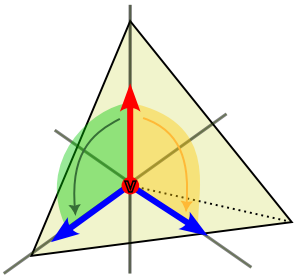
which works similarly to the `Generator` function. Given a generator g, any parameter $z \in \mathring{F}_c(g)$, a direction d_1 and a tet c, the holder function returns the unique simplex ρ, incident to c such that the integer-grid edge $\mathcal{E}_{d_1}(z)$ extends into the parametric interior $\mathring{F}_c(\rho)$. It returns $\perp$ if the integer-grid edge only shares its origin with the tet. The exact implementation is provided in Section A.2. If $\rho \neq \perp$, a *local propeller* with *holder* ρ is extracted, representing $|V_{\mathcal{H}}(g)|$ many hex-half-edges, one per hex-vertex on the generator. The holder is always an edge, face, or cell, and either the generator itself or an incident simplex of higher dimensionality, meaning there are 10 combinations of generator- and holder types, some of which are displayed in Figure 11. To avoid duplicates on a shared edge or face, the propeller is ignored if the assigned tet of its holder does not match the tet in which the propeller was discovered, analogously to Section 5.2. For all tets c_k incident to the holder ρ of the propeller, the direction of the propeller is stored as $d_{c_k}(p) = R_{c,c_k}(d_1)$ where τ_{c,c_k} is an accumulated transition $\tau_{c,c_k} = \tau_{c_{k-1},c_k} \circ \ldots \circ \tau_{c,c_1}$ through k tets from c to c_k around ρ, such that $\rho \sim c_j, c \sim c_1$ and $c_j \sim c_{j+1}$ for all j.

Since every local propeller represents one half-edge per vertex, the total number of hex-edges is given by

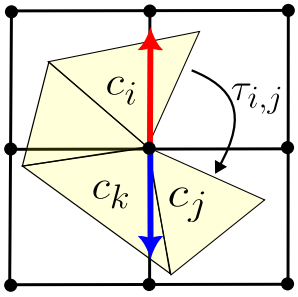
$$|E_{\mathcal{H}}| = \frac{1}{2} \sum_{g \in \mathcal{G}} |V_{\mathcal{H}}(g)| \cdot |\mathcal{P}_l(g)|$$

where $\mathcal{P}_l(g)$ is the set of all local propellers on generator g.

5.3.3 Blade Enumeration. With the propellers extracted, we could technically already trace along their directions through the parametrization until the opposite hex-vertex is reached. However, without information about the propeller blades and, in particular, how the blades of two opposing propellers are related via the connection offset, the result would not provide any knowledge about the hex-faces or -cells. Therefore, before connecting the propellers to their opposites, we first connect them to their blades. A blade of a local propeller is another local propeller on the same generator such that the two propellers enclose a common integer-grid face, meaning they are perpendicular. In the easiest case, the blade has an image in the same tet chart as the propeller. If this is not the case, the task is made nontrivial by the presence of nonidentity transitions, as the notion of orthogonality is somewhat ambiguous. For example, it is possible that a propeller and its blade are collinear when considering the images in the tets they were discovered in as illustrated in Figure 12b. Even when acknowledging possible transitions between tets, there are multiple ways to transition from one tet to another. For example, we can rotate either clockwise or counterclockwise around an edge. If we now consider an inner valence three singular edge, like the one in Figure 2, one way, between two tets between which the parametrization is cut open, leads to an accumulated transition of some rotation while the other way leads to the identity. Hence, we need a condition that determines through which face we leave a tet to travel from a propeller to its blade. But first, we need to know in what directions we need to look for the blades.



(a) A casing of a propeller (red) to one of its blades (blue) is the simplex into which the integer-grid face extends. To the left, the propeller extends from its holder, an edge, into the face (green), to the right into the cell (yellow).



(b) The direction of a propeller (red) on a valence 3 singularity within the chart of cell c_i is collinear to the direction of its blade (blue) in the chart of cell c_k due to the nonidentity transition τ_{ij}.

Fig. 12

Starting from a local propeller $p \in \mathcal{P}_l(g)$ on a generator g, we first iterate over each tet c incident to the holder h of p, in a counterclockwise manner, and evaluate for each direction d_2 orthogonal to the direction $d_1 = d_c(p)$ of p in the chart of c the function

$$\texttt{Casing} : (E_{\mathcal{T}} \cup F_{\mathcal{T}} \cup C_{\mathcal{T}}) \times \mathbb{Z}^3 \times \mathcal{D}^2 \times C_{\mathcal{T}} \to \mathcal{T} \cup \{\perp\}$$
$$(h, z, d_1, d_2, c) \mapsto \zeta$$

which works analogously to the `Generator` function for hex-vertices and the `Holder` function for propellers. A *casing* ζ of a propeller is the unique simplex incident to c such that the integer-grid face $\mathcal{F}_{d_1,d_2}(z)$ extends into its parametric interior $\mathring{F}_c(\zeta)$ as visualized in Figure 12a. It is either a face or a cell and is either the holder itself, or an incident simplex of higher dimensionality. Duplicates are avoided analogously to the generators and holders by checking the preassigned tet per simplex. The number of casings per propeller is equal to its number of blades. Therefore, it is also equal to the face valence of one of its $|V_{\mathcal{H}}|$ many resulting hex-edges. The consistent enumeration of the casings in a counterclockwise manner will ensure the existence of the connection offset in Section 5.4.

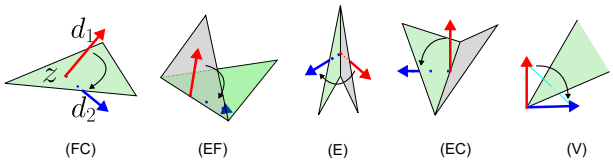


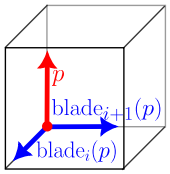
Fig. 13. The cases to consider when rotationally tracing from a propeller (red) to its blade (blue) through multiple tets. Only the last two require exact predicate tests. (FC): From a face generator, the generator itself is the only possible transition face. (EF): From a propeller on an edge generator with a face holder, there is only one other face. (E): When rotating around an edge and entering the current tet through a face, there is only one other face to exit through. (EC): For a propeller extending from an edge into a cell, the face on the correct side is picked. (V): On a vertex generator, the rotational trace must intersect the open triangle.

The next step is to follow the directions of the casings by rotating 90 degrees from a propeller to its blade, for which the strategy is similar to how the α_1 dart interconnections are determined in HexEx. For each local propeller p and each casing ζ of p with direction d_2 in the chart of a tet c, while we do not find the blade in the current tet c, we leave the tet c through a face, incident to the holder, that intersects the spanned integer-grid face, into an adjacent tet c', computed by a function `PickNextHalffaceToBlade` (Algorithm 7). When entering tet c', our parameters are updated according to the transition function between c and c'. The required predicate tests can be simplified depending on the generator and holder type. For example, when rotating around an edge generator and entering the current tet through a face, the only possible exit face is the other face incident to the edge. For a holder that is identical to the generator, the function `PickNextHalffaceToBlade` is not needed as the blade will always have an image in the same chart as the propeller. All cases to consider are shown in Figure 13. Since the blade relationship between propellers is symmetric, meaning one propeller is a blade of another if and only if that propeller is a blade, we only rotationally trace once between two propellers.

After this step, the number of hex-faces is given by

$$|F_{\mathcal{H}}| = \frac{1}{4}\sum_{g\in\mathcal{G}} |V_{\mathcal{H}}(g)| \cdot \frac{1}{2}\sum_{p\in\mathcal{P}_l(g)} |\text{blades}(p)|$$

5.3.4 Hex-Corner Enumeration. As preparation for the final hex-cell extraction step, the hex-corners are enumerated, which are the triplets of local propellers that span a hex-cell (see inset figure). Because the local connectivity for each hex-vertex is identical, one (local) hex-corner represents the corner of $|V_{\mathcal{H}}(g)|$ hex-cells.



For each generator $g \in \mathcal{G}$, we enumerate the set of its local corners $C_l(g)$ as all triples $(p_1, p_2, p_3) \in \mathcal{P}_l(g)^3$ such that $p_2 = \text{blade}_i(p_1)$ for some index i and $p_2 = \text{blade}_{i+1}(p_1)$. If the holder of the propeller p_1 is part of the tet mesh interior, we consider its list of blades to be cyclic, meaning $(p_1, \text{blade}_{|blades(p_1)|-1}(p_1), \text{blade}_0(p_1))$ is another hex-corner.

Because the blades were enumerated in counterclockwise order, all hex-corners are right-hand oriented and we do not need to do any exact predicate checks. Implicitly, a hex-corner corresponds to an integer-grid cell extending from an integer-grid point into the parametrization. Only one of the three possible orderings of propellers per hex-corner is stored to avoid duplicates.

After the enumeration of hex-corners, the number of hex-cells is given by

$$|C_{\mathcal{H}}| = \frac{1}{8}\sum_{g\in\mathcal{G}} |V_{\mathcal{H}}(g)| \cdot |C_l(g)|$$

5.4 Global Connectivity Extraction

The global connectivity extraction is the final step to the hex-mesh.

HexEx: [Lyon et al. 2016] interconnect their darts as follows: For each enumerated dart d, of which there are $48|C_{\mathcal{H}}|$ many, the dart $\alpha_i(d)$, for $i = 0, 1, 2, 3$, refers to the dart that shares all but the $i-$dimensional entity with d. If it cannot be found in the same tet chart as d, the tet is exited through the face f, which intersects the integer-grid edge, -face and -cell given by $\alpha_i(d)$. Except for the entering face, every face of the tet is tested, even if, considering the α_2 connection, it is not incident to the tet-simplex on which the dart-edge lies. Because of this, this process is costly in HexEx. Additionally,the list of darts per tet (which grows linearly with the hex-to-tet ratio) is searched linearly to find a matching connection, and every type of connection is checked independently.

Hex^2: Unlike the blade connections, the opposite connections of propellers also depend on the hex-vertex, not just the generator. Therefore, we introduce the notion of a *global propeller* as a pair $(v, p) \in V_{\mathcal{H}}(g) \times \mathcal{P}_l(g)$ of a hex-vertex and a local propeller. One global propeller corresponds to exactly one hex half-edge. The propeller tracing works analogously to the rotational blade tracing explained in Section 5.3.3 and is shown in Algorithm 8.

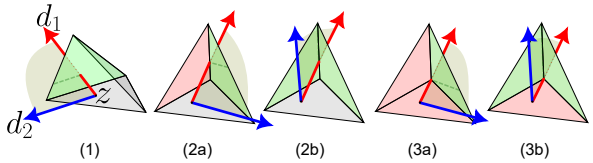


Fig. 14. A propeller (red) with its blade (blue) is traced to its opposite from tet to tet through an intersected face. To find the opposite blade, the secondary direction is considered if more than one face is intersected. (1): The face is exited through its interior, meaning the secondary direction does not need to be considered. (2a): The integer-edge intersects an edge shared by two faces, the secondary direction determines which face to pick. (3b): For an integer-edge intersecting a vertex shared by three faces, the secondary direction is considered to pick one of the three faces. (2b,3b): Two faces are intersected by the propeller and the integer-grid face. Either one is picked.

Starting from a hex-vertex v and propeller p with respective images $z \in \mathbb{Z}^3$ and $d \in \mathcal{D}$ in tet c_i, we leave the current tet through the face f, determined by a function `PickNextHalffaceToOpposite` (Algorithm 9), which intersects the integer-grid edge from z to $z + d$ into the next tet c_j and update our images according to the transition τ_{ij}, meaning z becomes $\tau_{ij}(z)$ and d becomes $R_{ij}(d)$. The different cases are shown in Figure 14. This process is repeated until $z + d$ lies in the parametric volume of the current tet, which is checked using a hashmap per tet that, for each integer parameter in the parametric volume of the tet, stores the corresponding hex-vertex. In case of a match, the opposite propeller of (v, p) must have an image in the chart of the current tet c. The integer-grid edge might leave a tet through one of its vertices or edges. For this ambiguity, a secondary direction, corresponding to one of the propeller's blades is considered. This is necessary, as otherwise, we might twirl around a higher singularity edge and end up in a wrong cell chart. Additionally, it allows us to compute the connection offset that determines which blade of the opposite propeller corresponds to which blade. Namely, if we consider the secondary direction of the $i-$th blade and find that it corresponds to the $j-$th blade of the opposite, the connection offset is $l = i + j$. The connection offset means that we need to trace each propeller only once, instead of the amount of its blades many

times.

The blade connections per local propeller and opposite connections per global propeller now define the entire hexahedral mesh. For each hex-vertex and each hex-corner on that vertex, the seven other corners of the same hex-cell are enumerated via opposite and implicit oppositeblade connections.

5.5 Piecewise-Linear Edge Arcs and Face Patches

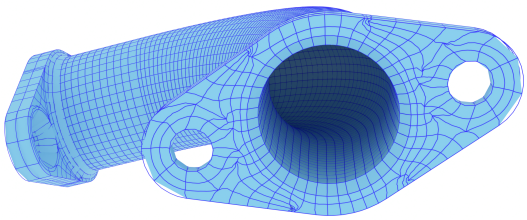


Fig. 15. A (linear) hex-mesh (model i09u from the HexMe dataset) with overlaid piecewise-linear edge arcs extracted by Hex^2. These are visibly more detailed than their straight counterparts and reveal distortions of the IGM near singularities.

As a new addition to the pipeline, we implemented the option to extract edges and faces as piecewise-linear segments corresponding to the intersections of an entire integer-grid edge or -face with the parametrized tet-mesh. These piecewise-linear curves and surfaces could then be used to fit higher-order meshes. We leave that to future work. Normally, the edges of the hex-mesh are straight line segments and the faces are quads. In contrast, piecewise-linear edges are a collection of multiple line segments that might differ in their directions, and piecewise-linear faces are polygonal meshes where individual polygons might have different normals. An example is shown in Figure 15.

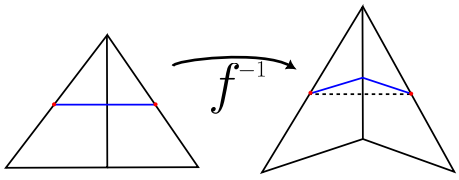


Fig. 16. Mapping the intersection point of an integer-grid edge between parametrized tets (left) back can lead to a point that does not lie on the straight edge (right) resulting in more detailed elements.

The extraction of piecewise-linear edges is straightforwardly integrated into the propeller tracing (Section 5.4). When tracing through a face, its intersection with the integer-grid edge represents an additional point on the piecewise-linear segment when being mapped back to the domain. This point does not have to lie on the straight line segment between the endpoints if the two tets get warped via the IGM f as illustrated in Figure 16.

Extracting piecewise-linear faces whose linear planar segments correspond to the intersection of the entirely linear hex-face with the tetrahedra requires a bit more work, though the idea remains simple. Starting from a tet that intersects the parametric interior of the corresponding integer-grid face, all other intersecting tets are enumerated using a flood-fill approach. From a tet, an adjacent tet is only checked if the common face intersects the parametric interior of the integer-grid face. Using the piecewise-linear segments from the edge extraction, these intersection polygons are cropped to up to an octagon, and mapped back to the domain, where they form the linear patches of a hex-face.

5.6 Runtime Analysis

We now analyze the asymptotic runtimes of the individual subtasks of HexEx and Hex^2 and explain the impact of our contributions. Two key quantities influence the algorithm's runtime: the number of elements in the input tet-mesh and the number of elements in the output hex-mesh induced by the IGM.

Let us consider an input tetrahedral mesh $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T}, F_\mathcal{T}, C_\mathcal{T})$ resulting in a hexahedral mesh $\mathcal{H} = (V_\mathcal{H}, E_\mathcal{H}, F_\mathcal{H}, C_\mathcal{H})$. We denote the hex-to-tet ratio by $\delta = \frac{|C_\mathcal{H}|}{|C_\mathcal{T}|}$ and define $L = \delta^{\frac{1}{3}}$. L is a measure for the parametrization refinement and is proportional to the average length of a parametrized tet-edge. In the following, we review each step of the extraction process and compare HexEx and Hex^2 analytically. Table 3 shows a summary of the observations.

Table 3. Asymptotic runtimes of the HexEx counterparts to the Hex^2 subroutines.

Task	HexEx	Hex^2
`Preprocessing`	$O(\mathcal{T})$	$O(\mathcal{T})$
`Vertex Extraction`	$O(\mathcal{T}L^3)$	$O(\mathcal{T}L^2)$
`Local Connectivity`	$O(\mathcal{T}L^6)$	$O(\mathcal{T})$
`Edge Extraction`	$O(\mathcal{T}L^6)$	$O(\mathcal{T}L^3)$
`Cell Extraction`	$O(\mathcal{T}L^3)$	$O(\mathcal{T}L^3)$

5.6.1 Preprocessing. The preprocessing phase is asymptotically identical for HexEx and Hex^2. It iterates over the elements of the input mesh to extract transition functions, calculate singularities, and sanitize errors in the parametrization, so it is linear in the number of tet-elements. It does not depend on the number of hex-elements. Since Hex^2 optimizes the vertex-parameter updates as explained in Section 5.1 and can use precomputed transition functions and valences, its hidden runtime constants are lower than the ones of its predecessor.

5.6.2 Vertex Extraction. Both algorithms extract the vertices linearly in the number of tet-elements. However, since HexEx iterates over all vertices, edges, faces, and cells separately and Hex^2 iterates over the cells only, the number of tet-elements contributes more to HexEx than to Hex^2. Testing every point in the axis-aligned bounding box of a face or tet is cubic in L, whereas the rasterization is only quadratic in L. This is because, for each tet, both the number of scanplanes and number of scanlines per scanplane are linear in L. Once we have a $1d$ scanline, we do not need to check every point on it anymore, removing one factor L from the complexity.

5.6.3 Local Connectivity. For HexEx, we count everything to the local connectivity that gives us the same connectivity information as the propellers and their blade connections. This includes the dart

extraction and the connections of the $\alpha_1, \alpha_2, \alpha_3$ pointers. Darts are enumerated per hex-vertex, and the number of hex-vertices per generator is cubic in L. For each dart, to find a corresponding dart, a list of darts per tet is searched linearly, and the number of darts per tet is, again, cubic in L, giving us a complexity of $O(\mathcal{T}L^6)$ in total. In contrast, Hex^2 does enumerate its local propellers once per generator, independent of the number of hex-vertices on it, and the connections to blades are also computed on this local level. This means our local connectivity extraction only grows in the possible number of generators, bounded by the number of tet-elements, no matter the size of the parametrization, i.e., $O(\mathcal{T})$.

5.6.4 Edge Extraction. Hex-edges are defined by the α_0 connections between darts in HexEx. Since these connections are computed analogously to the other dart connections, the asymptotic runtime is identical to the one of the local connectivity extraction: $O(\mathcal{T}L^6)$. The number of hex-edges per generator grows cubically in L. For Hex^2, the edge extraction corresponds to the global connectivity extraction. Since we use a hashmap to find a hex-vertex within a tet, instead of linearly searching through a list, Hex^2 only takes $O(\mathcal{T}L^3)$ time.

5.6.5 Cell Extraction. Constructing the hex-mesh requires an iteration over all darts, of which there are $O(|C_{\mathcal{H}}|) = O(\mathcal{T}L^3)$ many. The postprocessing step, which we also consider as part of this step, requires an iteration over all hex-vertices, which is asymptotically the same. In Hex^2, we iterate over all hex-vertices and their incident hex-corners to extract the hex-cells, which takes $O(\mathcal{T}L^3)$ time.

In total, we get a runtime $O(\mathcal{T}\delta^2)$ for HexEx, and $O(\mathcal{T}\delta) = O(\mathcal{H})$ for HexEx, evaluating to a factor of $O(L^3) = O(\delta)$. We show in Section 6.1 that Hex^2 is significantly faster than HexEx, in particular for large hex-to-tet ratios, but also if the hex-to-tet ratio is small (< 1).

6 Results

In the following, we compare our algorithm Hex^2 to its predecessor HexEx on examples of various complexity from the *HexMe* dataset [Beaufort et al. 2022] with parametrizations generated using the state-of-the-art algorithm by [Liu and Bommes 2023]. This yielded 103 valid IGMs. Some samples are displayed in Figure 17. The generated integer-grid maps (IGMs) deliberately result in coarse hex-meshes to measure meshes with a low hex-to-tet ratio as well as a larger hex-to-tet ratio by upscaling the parametrization, which corresponds to a refinement of the hexahedral mesh. The hex-mesh extraction algorithm only depends on the number of tet- and hex-elements. It does not depend on the structural mesh complexity indicated by the prefix s (simple), n (nasty) or i (industrial). Hence, some studies in Section 6.2 were evaluated on the relatively simple sphere model `s17c` without loss of generality. All reported evaluations were run on an Apple M1 Ultra. For Hex^2, if not stated otherwise, precomputed edge valences and transitions were utilized, no piecewise-linear elements were extracted, and parallelization was disabled. HexEx does not offer the option to utilize precomputed values during preprocessing. However, we show in Section 6.2.2 that Hex^2 is notably faster even with recomputing the transitions and valences. Furthermore, the impact of the preprocessing becomes negligible with large hex-to-tet ratios.

6.1 Runtimes

Our algorithm outperforms HexEx on all models in the HexMe data set. We measured the runtimes of HexEx, Hex^2 without parallelization, and Hex^2 utilizing 8 cores on the 103 models with different parametrization refinement factors (1, 2, 4) that increase the number of hexes being extracted as explained in Section 6.2.3. Single-core Hex^2 is about $10-40$ times faster than its predecessor, depending on the hex-to-tet ratio $\delta = \frac{|C_{\mathcal{H}}|}{|C_{\mathcal{T}}|}$. Multicore Hex^2 is, on average, over 50 times faster than HexEx. The total runtimes vary from $0.59 - 69.86s$ in HexEx, $0.02 - 3.7s$ in single-core Hex^2, and $0.01 - 1.21s$ in multi-core Hex^2. The timings for a selection of models are presented in Table 4.

Not only is the total runtime reduced from HexEx to Hex^2, but the runtimes of every subtask are also reduced. We observed that the hex-to-tet ratio δ strongly influences the performance gain, confirming the theoretical analysis given in Section 5.6. For coarse meshes with a low δ, most generators will contain no more than a single hex-vertex, reducing the possible impact of the local connectivity extraction and the rasterization. For such meshes, the preprocessing is the dominant part. The more refined the hex-mesh, i.e. the larger δ is, the more significant the optimizations become. In particular, the impact of the local connectivity extraction on the total runtime decreases to a negligible part with an increase of the hex-to-tet ratio since it does not depend on the number of hex-vertices per generator. In HexEx, this part of the mesh extraction process instead increases to about 25% of the total runtime. This can be seen in Figure 18, which shows a trend for how the proportions of the different subtasks change with different δ. For large δ, in HexEx, four dominant subtasks remain: Local and global connectivity extraction (edge tracing), hex-mesh construction, and postprocessing. In Hex^2, the dominant subtasks for large δ are the global connectivity extraction and the hex-mesh construction. The performance costs of the hex-mesh construction partially stem from our use of OpenVolumeMesh [Kremer et al. 2013].

6.2 Experiments

We now present the impact of certain parameters and properties on the performance of Hex^2. Namely, we evaluated the effects of (i) using different epsilons for the vertex extraction, (ii) utilizing precomputed transitions and edge valences on the preprocessing, (iii) the hex-to-tet ratio on the total runtime, (iv) the rasterization on the hex-vertex extraction, (v) using a hashmap on the connectivity extraction, and (vi) the effects of the piecewise-linear curve and surface extraction on the runtime.

6.2.1 Epsilon. We run all models for IGM scaling factors 1, 2 and 4 and $\varepsilon \in \{10, 1, 10^{-1}, ..., 10^{-21}, 10^{-22}, 0\}$. For $\varepsilon = 10^{-16}$, the rasterization succeeds for all models. Even for $\varepsilon = 0$, we capture all points for 101/103 models. For $\varepsilon \leq 10^{-17}$, there are two models where the rasterization misses a single integer-grid point: `n07u` and `n08c`. Hence, we defined the default $\varepsilon = 10^{-6}$ to still be much larger than any observed failure cases without any performance costs as

Table 4. Evaluation on different models of the HexMe dataset with different parametrization scalings. Displayed are from left to right the model, the parametrization refinement s, the number of tetrahedral elements in the input, the number of hexahedral elements in the output, their ratio δ, the runtime of HexEx, the runtime of Hex^2 using a single core (SC) and 8 cores (MC), their respective ratio compared to HexEx, the peak memory usage of HexEx, the peak memory usage of Hex^2 (single-core) and their ratio. The last three rows show the averages over all meshes for the three refinement levels.

Input					Runtime			Memory		
Model	s	$\lvert C_{\mathcal{T}}\rvert$	$\lvert C_{\mathcal{H}}\rvert$	δ	HexEx [s]	Hex^2 (SC/MC) [s]	$\frac{\text{Hex}^2\text{ (SC/MC)}}{\text{HexEx}}\downarrow$	HexEx [MB]	Hex^2 [MB]	$\frac{\text{Hex}^2}{\text{HexEx}}\downarrow$
s04b	1	66304	5428	0.08	1.25	0.09/0.03	7.02%/2.20%	63	15	23.84%
s04b	2	66304	43424	0.65	5.29	0.18/0.07	3.48%/1.28%	503	41	8.23%
s04b	4	66304	347392	5.24	40.55	0.77/0.35	1.90%/0.86%	4019	281	6.99%
s09u	1	100130	4590	0.05	1.58	0.13/0.03	7.97%/2.13%	51	21	40.61%
s09u	2	100130	36720	0.37	5.47	0.25/0.08	4.55%/1.44%	408	38	9.42%
s09u	4	100130	293760	2.93	31.12	0.85/0.37	2.74%/1.19%	3256	233	7.17%
s10u	1	28217	4986	0.18	0.85	0.05/0.02	5.85%/2.23%	56	7	13.25%
s10u	2	28217	39888	1.41	4.53	0.13/0.05	2.84%/1.17%	438	35	7.91%
s10u	4	28217	319104	11.31	33.72	0.59/0.28	1.74%/0.83%	3502	237	6.78%
s17c	1	28028	4608	0.16	0.78	0.05/0.02	6.20%/2.36%	51	7	14.29%
s17c	2	28028	36864	1.32	4.06	0.12/0.05	2.98%/1.25%	408	32	7.93%
s17c	4	28028	294912	10.52	30.30	0.54/0.26	1.79%/0.87%	3261	217	6.67%
n03u	1	33719	5274	0.16	0.94	0.06/0.02	6.23%/2.14%	62	9	14.00%
n03u	2	33719	42192	1.25	4.78	0.14/0.06	3.02%/1.18%	473	38	8.13%
n03u	4	33719	337536	10.01	34.86	0.65/0.31	1.85%/0.89%	3687	247	6.70%
n04c	1	7122	4233	0.59	0.59	0.02/0.01	3.97%/1.57%	49	5	9.58%
n04c	2	7122	33864	4.75	3.84	0.07/0.03	1.91%/0.83%	380	28	7.38%
n04c	4	7122	270912	38.04	46.58	0.39/0.22	0.84%/0.46%	3037	202	6.67%
n07c	1	39652	5638	0.14	1.09	0.07/0.02	6.28%/2.08%	65	10	14.98%
n07c	2	39652	45104	1.14	5.38	0.17/0.06	3.15%/1.20%	520	42	8.10%
n07c	4	39652	360832	9.10	37.23	0.73/0.35	1.97%/0.93%	4151	286	6.89%
n10u	1	133239	6511	0.05	2.23	0.19/0.05	8.40%/2.41%	74	29	39.01%
n10u	2	133239	52088	0.39	7.74	0.37/0.12	4.76%/1.57%	589	54	9.19%
n10u	4	133239	416704	3.13	44.36	1.22/0.54	2.74%/1.21%	4700	324	6.89%
n12b	1	72814	5508	0.08	1.38	0.10/0.03	7.04%/2.39%	64	15	24.05%
n12b	2	72814	44064	0.61	5.68	0.22/0.09	3.87%/1.51%	510	47	9.22%
n12b	4	72814	352512	4.84	38.32	0.84/0.40	2.18%/1.04%	4072	300	7.37%
i01c	1	137233	6190	0.05	2.24	0.23/0.07	10.48%/3.02%	71	29	41.44%
i01c	2	137233	49520	0.36	7.41	0.44/0.14	5.94%/1.90%	564	54	9.60%
i01c	4	137233	396160	2.89	43.68	1.26/0.57	2.89%/1.29%	4502	321	7.12%
i02c	1	123043	6389	0.05	2.08	0.18/0.06	8.82%/2.81%	73	26	35.86%
i02c	2	123043	51112	0.42	7.39	0.36/0.13	4.86%/1.69%	580	55	9.41%
i02c	4	123043	408896	3.32	44.75	1.28/0.56	2.87%/1.26%	4626	322	6.96%
i09u	1	793884	7188	0.01	10.03	1.11/0.25	11.07%/2.45%	84	156	186.51%
i09u	2	793884	57504	0.07	19.19	1.61/0.40	8.41%/2.07%	661	175	26.47%
i09u	4	793884	460032	0.58	69.86	3.70/1.21	5.30%/1.73%	5150	466	9.05%
i18c	1	73118	5465	0.07	1.48	0.11/0.04	7.56%/2.65%	64	16	25.50%
i18c	2	73118	43720	0.60	5.86	0.24/0.09	4.04%/1.50%	508	45	8.94%
i18c	4	73118	349760	4.78	37.18	0.86/0.39	2.30%/1.05%	4049	293	7.24%
i25u	1	200188	6050	0.03	2.93	0.27/0.07	9.08%/2.49%	70	41	58.50%
i25u	2	200188	48400	0.24	8.46	0.48/0.15	5.70%/1.79%	554	55	9.97%
i25u	4	200188	387200	1.93	43.06	1.38/0.59	3.21%/1.38%	4415	323	7.32%
Average	1	107821	5345	0.15	1.77	0.15/0.04	8.69%/2.52%	62	23	37.68%
Average	2	107821	42762	1.19	6.13	0.29/0.10	4.66%/1.57%	485	48	9.85%
Average	4	107821	342103	9.53	39.15	0.94/0.41	2.39%/1.05%	3843	275	7.15%

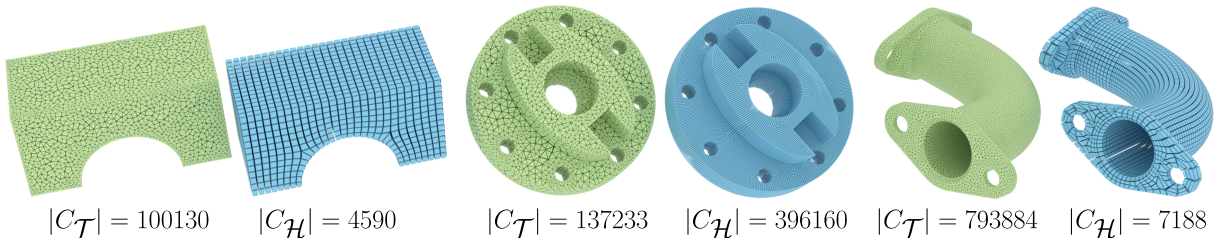


Fig. 17. Three sample tet-meshes (green) from the HexMe dataset, s09u, i01c and i09u, and their extracted hex-meshes (blue). Annotated are the respective number of tetrahedra and hexahedra.

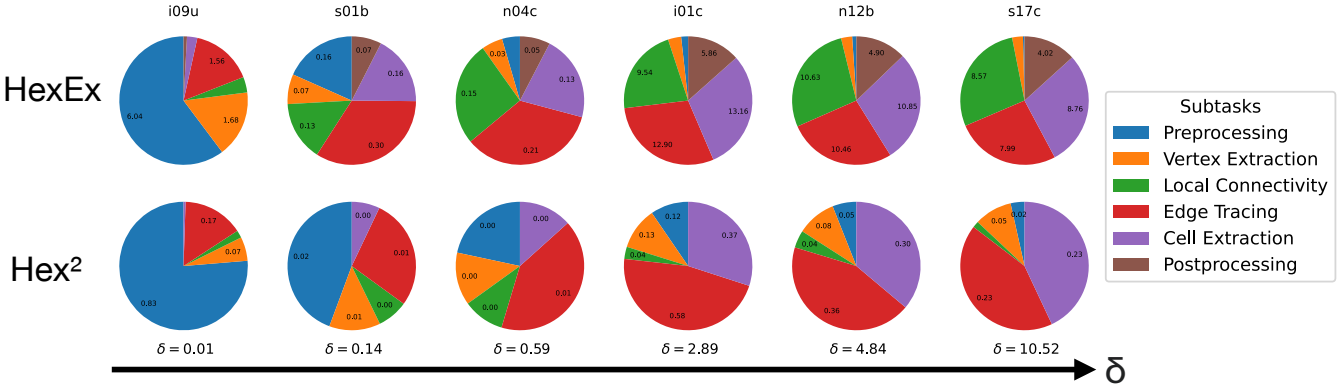


Fig. 18. Distribution of subtasks in HexEx (top) and Hex^2 (bottom) for models with different hex-to-tet ratios δ. The models are sorted from left (coarse/small δ) to right (fine/large δ). The runtimes of the individual tasks are shown in seconds and are omitted for readability if they take less than 5% of the total runtime. Particularly notable is the proportion of the local connectivity extraction (green), which grows in HexEx, but becomes negligible in Hex^2.

shown in Figure 19. Interestingly, a translation of the parameters by $(10^9, 10^9, 10^9)$ to reduce precision led to no missing hex-vertices for these two models, even for $\varepsilon = 0$, likely a consequence of the precision truncation in the sanitization of the IGM.

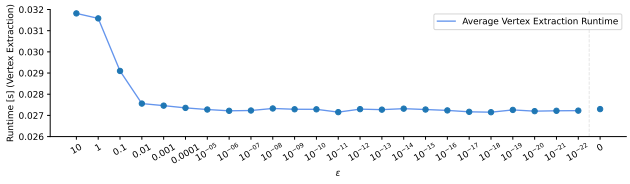


Fig. 19. Average (over 103 meshes with scaling factors 1, 2, 4) runtime of the vertex extraction for different rasterization tolerances $\varepsilon \geq 0$ (default $\varepsilon = 10^{-6}$). For small $\varepsilon \ll 1$, there is no notable difference in performance.

To further test our algorithm, we generated several million random tetrahedra within the required 32−bit integer range. None of these tetrahedra were missing any vertices, even for $\varepsilon = 0$.

6.2.2 Preprocessing. Since the parametrizations generated by [Liu and Bommes 2023] already contain transition rotations and edge valences, there is no need to recompute these values. We measured the runtimes of the preprocessing phase of Hex^2 for different models of the HexMe dataset when manually computing the values compared to utilizing the precomputed values as well as the preprocessing of HexEx. The runtime of the preprocessing step increases with the size of the tet-mesh. Even when recomputing the transitions and valences, Hex^2 performed better than HexEx. This shows that our optimization for the vertex parameter propagation explained in Section 5.1 contributes to a decrease in runtime as well. The results are visualized in Figure 20.

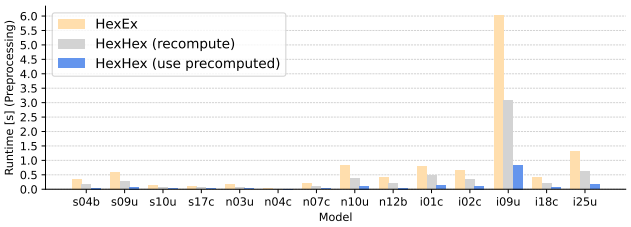


Fig. 20. Runtimes of the preprocessing phase for different models of the HexMe dataset when computing edge valences and transition functions manually (gray) compared to using precomputed values in the input (blue). For comparison, the runtime in HexEx is also displayed (beige).

6.2.3 Hex-Mesh Density. To measure the effect of parametrization refinement or hex-to-tet ratio δ, we measured the runtimes on a

model (s17c) with an increasing number of hex-elements. Given a tet-mesh and IGM, the resolution of the resulting hex-mesh can be increased by multiplying all per-cell-vertex parameters by an integer factor $s \in \mathbb{N}_{\geq 1}$ resulting in a hex-mesh with s^3 as many cells. The sphere mesh s17c is visualized in Figure 21 for $s = 1, 2, 3, 4$. Note that this is not exactly equivalent to subdividing a coarser hex-mesh as explained in Section 5.5, although the effects are similar.

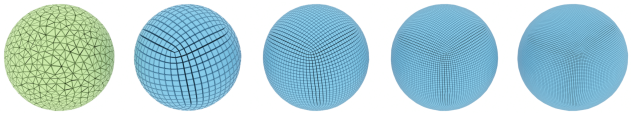


Fig. 21. The sphere model s17c and the extracted hex-meshes for four refinement levels 1, 2, 3 and 4 in which the number of hex-cells grows cubically.

As we increase the parametrization scaling factor, the runtime of Hex^2 increases significantly slower than the runtime of HexEx, as shown in Table 5. For the sphere model s17c consisting of 28 028 tets and with a parametrization scaling factor of $s = 12$, resulting in 7 962 624 hexes, the runtime of HexEx exploded to over an hour whereas HexEx takes humanly bearable 9.33 seconds. The runtime ratio between Hex^2 and HexEx becomes smaller with larger δ (from 6.3% for $s = 1$ to 0.23% for $s = 12$), confirming that our algorithm scales much better with the number of hex-elements as analysed in Section 5.6.

Table 5. Runtimes of HexEx and Hex^2 for the model s17c, consisting of 28 028 tetrahedra, where the parametrization gets more and more upscaled, resulting in more hex-cells. The larger the scaling factor, the more efficient our algorithm is relative to HexEx.

s	$\lvert C_{\mathcal{H}} \rvert$	δ	HexEx [s]	Hex^2 [s]	$\frac{\text{Hex}^2}{\text{HexEx}} \downarrow$
1	4 608	0.16	0.80	0.05	6.30%
2	36 864	1.32	4.00	0.14	3.61%
3	124 416	4.44	12.28	0.28	2.27%
4	294 912	10.52	30.71	0.57	1.85%
5	576 000	20.55	66.09	0.98	1.48%
6	995 328	35.51	136.38	1.49	1.09%
7	1 580 544	56.39	265.35	2.21	0.83%
8	2 359 296	84.18	496.82	3.21	0.65%
9	3 359 232	119.85	886.96	4.37	0.49%
10	4 608 000	164.41	1 534.54	5.74	0.37%
11	6 133 248	218.83	2 534.19	7.33	0.29%
12	7 962 624	284.10	4 049.37	9.33	0.23%

6.2.4 Rasterization. For the model s17c, we measured the impact of the hex-vertex extraction by rasterizing the tets compared to testing every integer-grid point in the bounding box. This experiment was run using different parametrization scaling factors to highlight the different asymptotic runtimes of the two methods, which are displayed in Figure 22. The rasterization is faster for all scaling factors, and the speedup is more significant, the larger the factor becomes.

6.2.5 Hashmap. We evaluated the impact of using a hashmap instead of a list to store and access the extracted data structure during

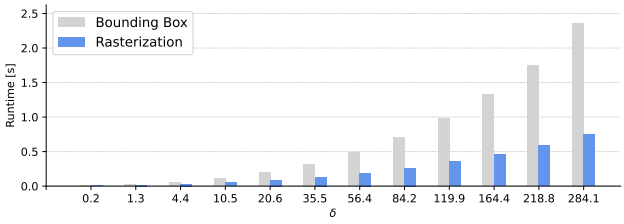


Fig. 22. Runtimes of the hex-vertex extraction on a single core for the model s17c with increasing hex-to-tet ratio δ using the rasterization in Hex^2 (blue) compared to the naive approach of testing each point in the bounding box of a tet (gray).

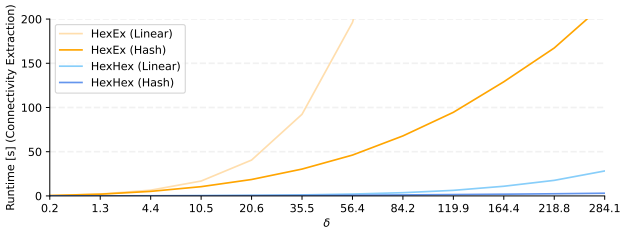


Fig. 23. Runtimes of the connectivity extraction of HexEx and HexHex, with using a hashmap and using a list to access darts or propellers per tet, for the sphere model s17c with an increasingly higher hex-to-tet ratio δ.

the connectivity extraction. This means, we both implemented a hashmap in HexEx, so darts can be accessed on a per tet- and parameter level, as well as a linear list in Hex^2, so propellers are only accessed on a per tet level. For these four options, we measured the runtimes of the connectivity extraction, including both local and global connectivity, as shown in Figure 23 and Table 6, for the example model s17c. For coarse hex-meshes where each generator contains mostly just a single hex-vertex, meaning there are only a few darts or (global) propellers per tet, the difference between the two containers in terms of performance is basically zero, as expected. The more refined the parametrization, the faster a hashmap becomes compared to the list. Even when searching all propellers stored per tet linearly, Hex^2's connectivity extraction is still notably faster than HexEx's, whether darts are accessed via a hashmap or searched for in a list. Noticeably, using a hashmap makes the algorithm scale much better for higher hex-to-tet ratios, which coincides with our theoretical analysis. Within the range of evaluated refinements ($\delta = 1$ to 12), Hex^2 with a hashmap is the only version exhibiting linear growth in δ, while the others grow quadratically. We conclude that, while using a hashmap does contribute to a faster connectivity extraction, the advantages of our data structure as well as the utilization of the constant local connectivity per generator described in Section 5.3 are more impactful.

6.2.6 Piecewise-Linear Extraction. We measured how the runtime of Hex^2 increases when enabling the additional extraction of piecewise linear elements. When enabling only the piecewise linear edge extraction, there is a small overhead since it entails both computing the intersections of integer-grid edges, mapping them back to the

Table 6. Impact of using a hashmap versus a linear list to access darts or propellers per tet. The table shows, for the model s17c using different parametrization scalings, the runtimes of the connectivity extraction, local and global, for HexEx using a list (as-is), for HexEx using a hashmap, for Hex^2 using a list, and for Hex^2 using a hashmap (default).

Input			HexEx (Linear)	HexEx (Hash)	Hex^2 (Linear)		Hex^2 (Hash)				
s	$\lvert C_{\mathcal{H}}\rvert$	δ	HexEx [s]	HexEx [s]	Hex^2 [s]	$\frac{\text{Hex}^2\text{ (L)}}{\text{HexEx (L)}}$	Hex^2 [s]	$\frac{\text{Hex}^2\text{ (H)}}{\text{HexEx (H)}}$	$\frac{\text{HexEx (H)}}{\text{HexEx (L)}}$	$\frac{\text{Hex}^2\text{ (H)}}{\text{Hex}^2\text{ (L)}}$	$\frac{\text{Hex}^2\text{ (H)}}{\text{HexEx (L)}}$
1	4608	0.16	0.44	0.41	0.01	2.87%	0.01	3.16%	93.00%	102.09%	2.93%
2	36864	1.32	2.11	1.94	0.06	2.71%	0.06	3.30%	92.32%	112.40%	3.05%
3	124416	4.44	6.48	5.13	0.15	2.28%	0.12	2.40%	79.19%	83.68%	1.90%
4	294912	10.52	16.83	10.45	0.32	1.88%	0.23	2.18%	62.14%	71.93%	1.36%
5	576000	20.55	40.53	18.57	0.62	1.54%	0.41	2.20%	45.82%	65.44%	1.01%
6	995328	35.51	92.31	30.35	1.15	1.24%	0.57	1.89%	32.88%	50.01%	0.62%
7	1580544	56.39	196.18	46.18	2.12	1.08%	0.82	1.77%	23.54%	38.51%	0.42%
8	2359296	84.18	395.34	67.85	3.65	0.92%	1.12	1.65%	17.16%	30.78%	0.28%
9	3359232	119.85	745.36	94.54	6.28	0.84%	1.52	1.61%	12.68%	24.27%	0.20%
10	4608000	164.41	1341.28	129.08	10.92	0.81%	1.95	1.51%	9.62%	17.84%	0.15%
11	6133248	218.83	2285.84	167.29	17.60	0.77%	2.43	1.45%	7.32%	13.82%	0.11%
12	7962624	284.10	3735.47	217.63	28.21	0.76%	3.03	1.39%	5.83%	10.76%	0.08%

mesh domain, and constructing a second mesh. When also enabling the extraction of piecewise linear faces, the extraction time is up to four times as large because, for every hex-face, we compute all of its intersections with the parametrization, which requires numerous exact predicate tests and polygon clipping. The resulting runtimes are visualized in Figure 24.

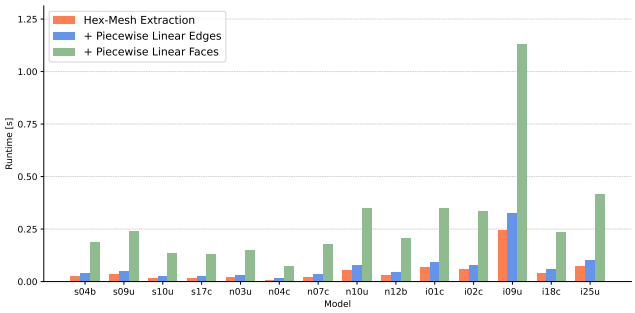


Fig. 24. The runtimes of Hex^2 (using 8 threads) for different models of the HexMe dataset when extracting a normal hexahedral mesh with no piecewise linear elements (red), when extracting a hexahedral mesh plus piecewise linear edges (blue), and when extracting a hexahedral mesh plus both piecewise linear edges and piecewise linear faces (green).

6.3 Memory

Lastly, we evaluated the peak memory usage when running both algorithms, presented in Table 4. Although this was not the main focus when Hex^2 was developed, the propeller data structure, as well as the utilization of constant local connectivity property, naturally leads to lower overhead compared to the dart data structure. Peak storage consumption indeed decreased from HexEx to Hex^2 for all but one model, on average by half for the coarsest inputs. The notable exception is i09 with a scaling factor of 1, the model with the lowest $\delta = 0.01$, for which our algorithm required nearly twice as much memory as HexEx. However, this mesh only requires 156 MB (84 MB with HexEx). For inputs with large memory consumption in HexEx, i.e. inputs with a large hex-to-tet ratio δ, the peak memory usage is reduced from several GB in HexEx to a few hundred MB in Hex^2.

7 Conclusion and Future Work

We presented HexHex (=Hex^2), a hexahedral mesh extraction algorithm that is significantly faster than HexEx for state-of-the-art integer-grid maps without defects. It extracts piecewise-linear edges and -faces if desired. We attribute the speedup of HexHex to its use of the propeller data structure, rasterization methods, and utilization of the constant local topology property.

The extraction of piecewise-linear elements could serve as a basis for extracting higher-order meshes with HexHex.

We designed and implemented the propeller data structure specifically for the hex-mesh extraction process. Exploring other applications of propellers as a mesh representation could be interesting.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 853343).

References

Cecil G Armstrong, Harold J Fogg, Christopher M Tierney, and Trevor T Robinson. 2015. Common Themes in Multi-block Structured Quad/Hex Mesh Generation. *Procedia Engineering* 124 (2015). doi:10.1016/j.proeng.2015.10.123

Marco Attene. 2020. Indirect Predicates for Geometric Constructions. *Computer-Aided Design* 126 (2020), 102856. doi:10.1016/j.cad.2020.102856

Pierre-Alexandre Beaufort, Maxence Reberol, D. Kalmykov, H. Liu, Franck Ledoux, and D. Bommes. 2022. Hex Me If You Can. *Computer Graphics Forum* 41 (Oct. 2022), 125–134. doi:10.1111/cgf.14608

David Bommes, Marcel Campen, Hans-Christian Ebke, Pierre Alliez, and Leif Kobbelt. 2013a. Integer-Grid Maps for Reliable Quad Meshing. *ACM Trans. Graph.* 32, 4 (July 2013). doi:10.1145/2461912.2462014

David Bommes, Bruno Lévy, Nico Pietroni, Enrico Puppo, Claudio Silva, Marco Tarini, and Denis Zorin. 2013b. Quad-mesh generation and processing: A survey. In *Computer Graphics Forum*, Vol. 32.
David Bommes, Henrik Zimmer, and Leif Kobbelt. 2009. Mixed-integer quadrangulation. *ACM Trans. Graph.* 28, 3 (July 2009). doi:10.1145/1531326.1531383
M. Botsch, S. Steinberg, Stephan Bischoff, and Leif Kobbelt. 2002. OpenMesh - a generic and efficient polygon mesh data structure. (Feb. 2002).
Hendrik Brückler, David Bommes, and Marcel Campen. 2022. Volume parametrization quantization for hexahedral meshing. *ACM Trans. Graph.* 41, 4 (July 2022). doi:10.1145/3528223.3530123
Hendrik Brückler, David Bommes, and Marcel Campen. 2024. Integer-Sheet-Pump Quantization for Hexahedral Meshing. In *Computer Graphics Forum*, Vol. 43.
Hendrik Brückler and Marcel Campen. 2023. Collapsing Embedded Cell Complexes for Safer Hexahedral Meshing. *ACM Trans. Graph.* 42, 6 (Dec. 2023). doi:10.1145/3618384
Hendrik Brückler, Ojaswi Gupta, Manish Mandad, and Marcel Campen. 2022. The 3D Motorcycle Complex for Structured Volume Decomposition. *Computer Graphics Forum* 41, 2 (2022), 221–235. doi:10.1111/cgf.14470
Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. 1998. Directed edges – a scalable representation for triangle meshes. *Journal of Graphics tools* 3, 4 (1998).
Marcel Campen. 2017. Partitioning surfaces into quadrilateral patches: A survey. In *Computer Graphics Forum*, Vol. 36.
Hans-Christian Ebke, David Bommes, Marcel Campen, and Leif Kobbelt. 2013. QEx: Robust quad mesh extraction. *ACM Trans. Graph.* 32, 6 (2013).
Vladimir Garanzha, Igor Kaporin, Liudmila Kudryavtseva, François Protais, Nicolas Ray, and Dmitry Sokolov. 2021. Foldover-free maps in 50 lines of code. *ACM Trans. Graph.* 40, 4 (July 2021). doi:10.1145/3450626.3459847
Steffen Hinderink, Hendrik Brückler, and Marcel Campen. 2024. Bijective Volumetric Mapping via Star Decomposition. *ACM Trans. Graph.* 43, 6 (Nov. 2024). doi:10.1145/3687950
Steffen Hinderink and Marcel Campen. 2023. Galaxy Maps: Localized Foliations for Bijective Volumetric Mapping. *ACM Trans. Graph.* 42, 4 (July 2023). doi:10.1145/3592410
Tengfei Jiang, Jin Huang, Yuanzhen Wang, Yiying Tong, and Hujun Bao. 2013. Frame field singularity correction for automatic hexahedralization. *IEEE Transactions on Visualization and Computer Graphics* 20, 8 (Aug. 2013). doi:10.1109/TVCG.2013.250
Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. 2002. Dual contouring of hermite data. *ACM Trans. Graph.* 21, 3 (July 2002), 339–346. doi:10.1145/566654.566586
Pierre Kraemer, Lionel Untereiner, Thomas Jund, Sylvain Thery, and David Cazier. 2014. CGoGN: N-dimensional Meshes with Combinatorial Maps. doi:10.1007/978-3-319-02335-9_27
Michael Kremer, David Bommes, and Leif Kobbelt. 2013. OpenVolumeMesh – A versatile index-based data structure for 3D polytopal complexes. In *Proceedings of the 21st International Meshing Roundtable*. Springer. doi:10.1007/978-3-642-33573-0_31
Felix Kälberer, Matthias Nieser, and Konrad Polthier. 2007. QuadCover - Surface Parameterization using Branched Coverings. *Computer Graphics Forum* 26, 3 (2007), 375–384. doi:10.1111/j.1467-8659.2007.01060.x
Yufei Li, Yang Liu, Weiwei Xu, Wenping Wang, and Baining Guo. 2012. All-hex meshing using singularity-restricted field. *ACM Trans. Graph.* 31, 6 (Nov. 2012). doi:10.1145/2366145.2366196
Heng Liu and David Bommes. 2023. Locally Meshable Frame Fields. *ACM Transactions on Graphics* 42, 4 (2023). doi:10.1145/3592457
Heng Liu, Paul Zhang, Edward Chien, Justin Solomon, and David Bommes. 2018. Singularity-Constrained Octahedral Fields for Hexahedral Meshing. *ACM Trans. Graph.* 37, 4 (July 2018). doi:10.1145/3197517.3201344
William E. Lorensen and Harvey E. Cline. 1987. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Comput. Graph.* 21, 4 (Aug. 1987), 163–169. doi:10.1145/37402.37422
Max Lyon, David Bommes, and Leif Kobbelt. 2016. HexEx: Robust Hexahedral Mesh Extraction. *ACM Trans. Graph.* 35, 4 (July 2016). doi:10.1145/2897824.2925976
Manish Mandad and Marcel Campen. 2019. Exact Constraint Satisfaction for Truly Seamless Parametrization. *Computer Graphics Forum* 38, 2 (2019). doi:10.1111/cgf.13625
Jan Möbius and Leif Kobbelt. 2012. OpenFlipper: An Open Source Geometry Processing and Rendering Framework. In *Curves and Surfaces*. LNCS, Vol. 6920. Springer Berlin / Heidelberg, 488–500. doi:10.1007/978-3-642-27413-8_31
Matthias Nieser, Ulrich Reitebuch, and Konrad Polthier. 2011. CubeCover – Parameterization of 3D Volumes. *Computer Graphics Forum* 30, 5 (2011). doi:10.1111/j.1467-8659.2011.02014.x
Valentin Zénon Nigolian, Marcel Campen, and David Bommes. 2023. Expansion Cones: A Progressive Volumetric Mapping Framework. *ACM Trans. Graph.* 42, 4 (July 2023). doi:10.1145/3592421
Valentin Zénon Nigolian, Marcel Campen, and David Bommes. 2024. A Progressive Embedding Approach to Bijective Tetrahedral Maps driven by Cluster Mesh Topology. *ACM Trans. Graph.* 43, 6 (Nov. 2024). doi:10.1145/3687992
Nico Pietroni, Marcel Campen, Alla Sheffer, Gianmarco Cherchi, David Bommes, Xifeng Gao, Riccardo Scateni, Franck Ledoux, Jean Remacle, and Marco Livesu. 2022. Hex-mesh generation and processing: a survey. *ACM Trans. Graph.* 42, 2 (2022).
Nicolas Ray, Wan Chiu Li, Bruno Lévy, Alla Sheffer, and Pierre Alliez. 2006. Periodic global parameterization. *ACM Trans. Graph.* 25, 4 (Oct. 2006), 1460–1485. doi:10.1145/1183287.1183297
Jonathan Richard Shewchuk. 1997. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry* 18 (1997). doi:10.1007/PL00009321

A Exact Predicates

Hex^2 implements three exact predicate functions that use ori3D by [Richard Shewchuk 1997] to evaluate its various geometric tests. These are presented in the following. It is crucial that the tetrahedra of the input mesh are oriented positively

$$\forall c \approx (v_1, v_2, v_3, v_4) \in C_{\mathcal{T}} : \texttt{ori3D}(\boldsymbol{p}(v_1), \boldsymbol{p}(v_2), \boldsymbol{p}(v_3), \boldsymbol{p}(v_4)) > 0$$

where $\boldsymbol{p} : V_{\mathcal{T}} \to \mathbb{R}^3$ is the position or geometric embedding of a vertex. Furthermore, half-faces are always oriented such that the fourth vertex of the incident cell lies within the defined half-space. This means that for any (inner) half-face identified by the vertices (v_1, v_2, v_3) and its incident cell consisting of the vertices v_1, v_2, v_3 and v_4, we have $\texttt{ori3D}(\boldsymbol{p}(v_1), \boldsymbol{p}(v_2), \boldsymbol{p}(v_3), \boldsymbol{p}(v_4)) > 0$.

A.1 Generator

Algorithm 3 HexHex: `Generator`

Input
parameter $z \in \mathbb{R}^3$
tet $c \in C_{\mathcal{T}}$ **with halffaces** hf_0, hf_1, hf_2, hf_3
Output
generator $\sigma \in V_{\mathcal{T}} \cup E_{\mathcal{T}} \cup F_{\mathcal{T}} \cup C_{\mathcal{T}}$ s.t. $z \in \mathring{F}_c(\sigma)$

$v_{i1}, v_{i2}, v_{i3} \leftarrow$ halfface vertices, $i = 0, 1, 2, 3$
for $i = 0, 1, 2, 3$ **do**
  $o_i \leftarrow \text{ori3D}(\boldsymbol{f_c}(v_{i1}), \boldsymbol{f_c}(v_{i2}), \boldsymbol{f_c}(v_{i3}), z)$
  if $o_i < 0$ **then return** $\perp$ //outside tet
if $\sum o_i = 1$ **then return** v s.t. $v \sim hf_i, hf_j, hf_k$ where $o_i = o_j = o_k = 0$ //on vertex
if $\sum o_i = 2$ **then return** e s.t. $e \sim hf_i, hf_j$ where $o_i = o_j = 0$ //on edge
if $\sum o_i = 3$ **then return** hf_i s.t. $o_i = 0$ //on face
if $\sum o_i = 4$ **then return** c //in tet

To compute the generator of a given parameter $z \in \mathbb{Z}^3$, we evaluate it on each of the four half-spaces whose intersection represents the tetrahedron. This results in four orientations $o_1, o_2, o_3, o_4 \in \{-1, 0, 1\}$. If any o_i is negative, the generator is not part of c nor is it c itself. Otherwise, the number of orientations being zero tells us whether the generator is one of the four vertices (intersects three half-faces), one of the six edges (intersects two half-faces), one of the four faces (intersects one half-face) or the cell itself (intersects no half-faces) as shown in Algorithm 3.

Given the generators σ_l and σ_r of two endpoints of a scanline in a tetrahedron $c \in C_{\mathcal{T}}$, all integer-grid points in between have a common generator σ that can be determined as follows:

(1) If $\sigma_l = c$ or $\sigma_r = c$ then $\sigma = c$
(2) If $\sigma_l = \sigma_r$ then $\sigma = \sigma_l$
(3) If $\sigma_l, \sigma_r \in V_{\mathcal{T}}$ then $\sigma \in E_{\mathcal{T}}$ s.t. $\sigma_l, \sigma_r \sim \sigma$
(4) If $\sigma_l, \sigma_r \in E_{\mathcal{T}}$ then $\sigma \in F_{\mathcal{T}}$ s.t. $\sigma_l, \sigma_r \sim \sigma$ or $\sigma = c$ otherwise
(5) If $\sigma_l, \sigma_r \in F_{\mathcal{T}}$ then $\sigma = c$
(6) If $\sigma_l < \sigma_r$ then $\sigma = \sigma_r$ if $\sigma_l \sim \sigma_r$ or $\sigma = \sigma_r + 1$ s.t. $\sigma \sim \sigma_l, \sigma_r$ otherwise

where we assume w.l.o.g. $\sigma_l \leq \sigma_r$. In practicality, only the first two cases are checked for simplicity. Otherwise, the generator σ is manually computed once for an arbitrary integer point between the two endpoints.

A.2 Holder

To compute the holder entity of a local propeller, we utilize the type of the generator. For example, a propeller cannot have a holder with lower dimensionality than its generator g. The idea is the same as in the `Generator` function: Given a parameter $z \in \mathring{F}_c(g)$ in the parametric interior of a generator g, and a direction d, we evaluate the orientations of the parameter $z + d$ against each half-face of the tet that is incident to the generator. This means, for a vertex generator, we need to test against three half-faces, for edge generators, we need to test against two half-faces, and for a face generator, we need to test against one half-face; itself. For a cell generator, no exact predicates are required as every direction extends into the same cell. As part of the constant local connectivity property, we skip the propeller extraction on cell generators anyway. If any of the orientations is negative, we do not point into the image of the tet. Otherwise, the number of positive orientation tests give us the holder entity. The algorithm is provided in Algorithm 4.

Algorithm 4 HexHex: `Holder`

Input
generator g =`Generator`(z, c)
parameter $z \in \mathbb{R}^3$
direction $d \in \mathcal{D}$
tet $c \in C_{\mathcal{T}}$
Output
holder $\rho \in E_{\mathcal{T}} \cup F_{\mathcal{T}} \cup C_{\mathcal{T}}$

if $g \in C_{\mathcal{T}}$ **then return** g
if $g \in F_{\mathcal{T}}$ **then**
  $v_1, v_2, v_3 \leftarrow$ halfface vertices of (g, c)
  $o \leftarrow \text{ori3D}(\boldsymbol{f}_c(v_1), \boldsymbol{f}_c(v_2), \boldsymbol{f}_c(v_3), z + d)$
  if $o > 0$ **then return** c *//from face into cell*
  if $o = 0$ **then return** g *//from face along fac*
  return $\perp$ *//from face into other side*
if $g \in E_{\mathcal{T}}$ **then**
  $hf_0, hf_1 \leftarrow$ halffaces of c incident to g
  $v_{i1}, v_{i2}, v_{i3} \leftarrow$ halfface vertices, $i = 0, 1$
  for $i = 0, 1$ **do**
    $o_i \leftarrow \text{ori3D}(\boldsymbol{f}_c(v_{i1}), \boldsymbol{f}_c(v_{i2}), \boldsymbol{f}_c(v_{i3}), z + d)$
    if $o_i < 0$ **then return** $\perp$ *//outside tet*
  if $\sum o_i = 0$ **then return** g *//from edge along edge*
  if $\sum o_i = 1$ **then return** hf_i s.t. $o_i = 0$ *//from edge into face*
  if $\sum o_i = 2$ **then return** c *//from edge into cell*
if $g \in V_{\mathcal{T}}$ **then**
  $hf_0, hf_1, hf_2 \leftarrow$ halffaces of c incident to g
  $v_{i1}, v_{i2}, v_{i3} \leftarrow$ halfface vertices, $i = 0, 1, 2$
  for $i = 0, 1, 2$ **do**
    $o_i \leftarrow \text{ori3D}(\boldsymbol{f}_c(v_{i1}), \boldsymbol{f}_c(v_{i2}), \boldsymbol{f}_c(v_{i3}), z + d)$
    if $o_i < 0$ **then return** $\perp$ *//outside tet*
  if $\sum o_i = 1$ **then return** e s.t. $e \sim hf_i, hf_j$ where $o_i = o_j = 0$ *//from vertex into edge*
  if $\sum o_i = 2$ **then return** hf_i s.t. $o_i = 0$ *//from vertex into face*
  if $\sum o_i = 3$ **then return** c *//from vertex into cell*

A.3 Casing

Computing a casing entity works analogously to the `Generator` and `Holder` functions. The algorithm is displayed in Algorithm 5. For a parameter $z \in \mathring{F}_c(g)$ in the parametric interior of a generator g, a direction d_1 pointing into the holder h, and a second, orthogonal, direction d_2, we evaluate the orientations of the parameter $z + d_2$ against each half-face of the tet that is incident to the holder.

Algorithm 5 HexHex: `Casing`

Input
holder h =`Holder`$(\text{Generator}(z, c), z, d_1, c)$
parameter $z \in \mathbb{R}^3$
directions $d_1, d_2 \in \mathcal{D}$
tet $c \in C_{\mathcal{T}}$
Output
casing $\zeta \in F_{\mathcal{T}} \cup C_{\mathcal{T}}$

if $h \in C_{\mathcal{T}}$ **then return** h
if $h \in F_{\mathcal{T}}$ **then**
  $v_1, v_2, v_3 \leftarrow$ halfface vertices of (h, c)
  $o \leftarrow \text{ori3D}(\boldsymbol{f}_c(v_1), \boldsymbol{f}_c(v_2), \boldsymbol{f}_c(v_3), z + d_2)$
  if $o > 0$ **then return** c
  if $o = 0$ **then return** h
  return $\perp$
if $h \in E_{\mathcal{T}}$ **then**
  $hf_0, hf_1 \leftarrow$ halffaces of c incident to h
  $v_{i1}, v_{i2}, v_{i3} \leftarrow$ halfface vertices, $i = 0, 1$
  for $i = 0, 1$ **do**
    $o_i \leftarrow \text{ori3D}(\boldsymbol{f}_c(v_{i1}), \boldsymbol{f}_c(v_{i2}), \boldsymbol{f}_c(v_{i3}), z + d_2)$
    if $o_i < 0$ **then return** $\perp$
  if $\sum o_i = 1$ **then return** hf_i s.t. $o_i = 0$
  if $\sum o_i = 2$ **then return** c

A.4 Connectivity Extraction

Algorithm 6 HexHex: Local Connectivity Extraction

parallel for each generator $g \in \mathcal{G} \setminus (C_{\mathcal{T}} \cup \partial F_{\mathcal{T}})$ **do**
  pick any hex-vertex $v \in V_{\mathcal{H}}$ with generator g
  //Propeller extraction
  for $c \in C_{\mathcal{T}}$ incident to g **do**
    for $d \in \mathcal{D}$ **do**
      if $(\rho \leftarrow \text{Holder}(g, \boldsymbol{f}_c(v), d, c)) \neq \perp$ **and** $\text{ParentTet}(\rho) = c$ **then**
        $p \leftarrow$ new local propeller with holder ρ and direction $d_c(p) = d$
        $\mathcal{P}_l(g) += p$
  //Blade enumeration
  for $p \in \mathcal{P}_l(g)$ with holder h **do**
    dirs2blades$[p] \leftarrow [\,]$
    for $c \in C_{\mathcal{T}}$ incident to h in CCW order **do**
      $d_1 \leftarrow d_c(p)$
      for $d_2 \in \mathcal{D}$ s.t. $d_2 \perp d_1 \wedge \text{Casing}(h, \boldsymbol{f}_c(v), d_1, d_2, c) \neq \perp$ in CCW order **do**
        if $\text{ParentTet}(\zeta) = c$ **then**
          dirs2blades$[p] \leftarrow$ dirs2blades$[p] + (c, d_2)$
  for $p \in \mathcal{P}_l(g)$ with holder h **do**
    for $i = 0, 1, \ldots |\text{dirs2blades}[p]| - 1$ **do**
      if blades$[p][i]$ is valid **then**
        continue
      $c, d_2 \leftarrow$ dirs2blades$[p][i]$
      $d_1 \leftarrow d_c(p)$
      $(c, f)_{exit} \leftarrow (c, \perp)$
      $z \leftarrow \boldsymbol{f}_c(v)$
      while $(p' \leftarrow$ propeller in $\mathcal{P}_l(g)$ s.t. $d_c(p') = d_2)$ is not valid **do**
        $(c, f)_{exit} \leftarrow \text{PickNextHalffaceToBlade}((c, f)_{exit}, z, d_1, d_2)$
        $z, d_1, d_2 \leftarrow \tau_{(c,f)_{exit}}(z, d_1, d_2)$
        $(c, f)_{exit} \leftarrow$ opposite halfface of $(c, f)_{exit}$
      blades$[p][i] \leftarrow p'$
      $j \leftarrow$ index s.t. dirs2blades$[p'][j] = (c, d_1)$
      blades$[p'][j] \leftarrow p$
  //Corner enumeration
  for $p \in \mathcal{P}_l(g)$ with holder h **do**
    for $i = 0, \ldots,$ len(blades$[p]) - 2$ **do**
      $C_l(g) += (p, \text{blades}[p][i], \text{blades}[p][i + 1])$
    if $h \notin \partial\mathcal{T}$ **then**
      $C_l(g) += (p, \text{blades}[p][0], \text{blades}[p][\text{len}(\text{blades}[p]) - 1])$

The local connectivity extraction as explained in Section 5.3 is provided in Algorithm 6 in its three parts: Propeller extraction, blade connections, and corner enumeration. In the blade connection part, it is crucial that the blades around a propeller are consistently ordered counterclockwise. We first enumerate the directions to the blades before rotationally tracing to them, so we do not connect the same pair twice.

The rotational tracing from a local propeller to one of its blades, provided the starting tet and directions in this chart, requires the function `pickNextHalffaceToBlade` which determines which adjacent tet should be traced into in case the blade does have no image in the current one. The algorithm considers the cases in Figure 13 and is provided in Algorithm 7.

Algorithm 7 `PickNextHalffaceToBlade`

Input
p propeller on generator g
(f, c) entering halfface. Invalid or incident to g and c
c current cell chart
d_1, d_2 propeller root and blade directions in the chart of c
Output
(f', c) exiting halfface incident to c and not equal to (f, c) which is intersected by the integer-grid face

if $g \in F$ **then** //Face-cell-propeller
  return (g, c) //(FC): only one option
inFirstCell $\leftarrow (f, c)$ is invalid //still in starting chart?
if $g \in E$ **then**
  if ¬inFirstCell **then** //Edge-generator, not inFirstCell
    return unique (f', c) s.t. $f' \neq f, g \sim f'$ //only one option
  if holder$[p] \in F$ **then** //Edge-face-propeller, inFirstCell
    return unique (f', c) s.t. $f' \neq$ holder$[p], g \sim f'$ //(EF): only one option
$z \leftarrow f_c(v_h)$ //v_h can be any hex-vertex with generator g
for $(f', c) \sim c$ s.t. $f' \neq f$ and $g \sim f'$ **do**
  if $g \in E$ **then** //(EC): Edge-cell-propeller, inFirstCell
    $u, v, w \leftarrow f_c((f', c))$ ordered s.t. $f_c(g) = \{u, v\}$ //halfface parameters
    $\text{ori}_{d_2} \leftarrow \text{ORI3D}(u, v, z + d_1, z + d_2)$
    if $\text{ori}_{d_2} = 0$ **then** //face cuts through generator edge
      return (f', c) //both faces incident to g would be ok
    $\text{ori}_w \leftarrow \text{ORI3D}(u, v, z + d_1, w)$
    if $\text{ori}_{d_2} = \text{ori}_w$ **then** //face lies on correct side, see (EC)
      return (f', c)
  else //(V): Vertex-generator
    $u, v, w \leftarrow f_c((f', c))$ ordered s.t. $f_c(g) = u$ //halfface parameters
    $\text{ori}_1 \leftarrow \text{ORI3D}(u, v, u + d_1, u + d_2)$
    $\text{ori}_2 \leftarrow \text{ORI3D}(w, u, u + d_1, u + d_2)$
    if $\text{ori}_1 = \text{ori}_2$ **then** //Rotation goes through inner part of face triangle
      return (f', c) //(both oris must be < 0)
    if ($\text{ori}_1 = 0$ and $\text{ori}_2 < 0$) or ($\text{ori}_1 < 0$ and $\text{ori}_2 = 0$) **then** //Rotation goes through edge uv or uw
      return (f', c)

The tracing from a global propeller to one of its opposites, provided the starting tet and parameter, the primary direction of the propeller and the secondary direction of one of its blades with a casing incident to the starting tet, requires the function `PickNextHalffaceToOpposite` which determines which adjacent tet should be traced into in case the opposite vertex is not contained in the current one. The algorithm considers the cases in Figure 14 and is provided in Algorithm 9.

Algorithm 8 HexHex: Global Connectivity Extraction

parallel for each generator $g \in \mathcal{G}$ **do**
  for each global propeller $(v, p) \in V_{\mathcal{H}}(g) \times \mathcal{P}_l(g)$ **do**
    if opposite$[v, p]$ is valid **then**
      continue
    $z \leftarrow f_c(v)$
    $c, d_2 \leftarrow$ dirs2blades$[p][0]$
    $d_1 \leftarrow d_c(p)$
    $(c, f)_{exit} \leftarrow (c, \epsilon)$
    while $v' \leftarrow$ hex-vertex with $f_c(v') = z + d_1$ is not valid **do**
      $(c, f)_{exit} \leftarrow$ `PickNextHalffaceToOpposite`$((c, f)_{exit}, z, d_1, d_2)$
      $z, d_1, d_2 \leftarrow \tau_{(f,c)_{exit}}(z, d_1, d_2)$
      $(c, f)_{exit} \leftarrow$ opposite halfface of $(c, f)_{exit}$
    $p' \leftarrow$ propeller with $d_c(p') = -d_1$
    $j \leftarrow$ index s.t. dirs2blades$[p'][j] = (c, d_2)$
    opposite$[v, p] \leftarrow (v', p')$
    opposite$[v', p'] \leftarrow (v, p)$
    connectionOffset$[v, p] \leftarrow$ connectionOffset$[v', p'] \leftarrow j$

Algorithm 9 `PickNextHalffaceToOpposite`

Input
(f, c) entering halfface. Invalid or incident to c
c current cell chart
z hex-vertex parameter in the chart of c
d_1, d_2 propeller root and blade directions in the chart of c
Output
(f', c) exiting halfface incident to c and not equal to (f, c) which is intersected by the integer-grid face and by the integer-grid face

for $(f', c) \sim c$ s.t. $f' \neq f$ **do**
  $u, v, w \leftarrow f_c((f', c))$
  if ORI3D$(u, v, w, z) \leq 0$ or ORI3D$(u, v, w, z + d_1) \geq 0$ **then**
    continue //Root does not even cut through the face plane
  $\text{ori}_{uv} \leftarrow \text{ORI3D}(u, v, z, z + d_1)$ //get the orientations of the three
  $\text{ori}_{vw} \leftarrow \text{ORI3D}(v, w, z, z + d_1)$ //tets formed around $z, z + d_1$
  $\text{ori}_{wu} \leftarrow \text{ORI3D}(w, u, z, z + d_1)$
  if $\text{ori}_{uv} = \text{ori}_{vw} = \text{ori}_{wu}$ **then** //all oris are < 0
    return (f', c) //(1): Root does cut through inner part of face triangle
  if any of these three oris is positive and any of them is negative **then**
    continue //Root does not cut through the face triangle
    //Otherwise, the root cuts through the triangles boundary (edge or vertex)
  $n \leftarrow (\text{ori}_{uv} = 0) + (\text{ori}_{vw} = 0) + (\text{ori}_{wu} = 0)$ //Get number of edge intersections. 1 or 2
  if $n = 1$ **then** //cuts through edge
    cycle u, v, w s.t. $\text{ori}_{uv} = 0$ //u, v is intersected edge
    ori $\leftarrow \text{ORI3D}(u, v, z + d_1, z + d_2)$
    if ori$= 0$ **then** //Integer-grid plane cuts through edge
      return (f', c) //(2b): both faces incident to edge would be ok
    if ori$=$ ORI3D$(u, v, z + d_1, w')$ **then**
      return (f', c) //(2a): integer-grid face intersects triangle face
    else //(2a): integer-grid face intersects other incident triangle face
      return unique (f'', c) incident to intersected edge s.t. $f' \neq f''$
  else //$n = 2$, (3): cuts through vertex
    cycle u, v, w s.t. $\text{ori}_{uv} = \text{ori}_{wu} = 0$ //u is intersected vertex
    if ORI3D$(v, u, z, z + d_2) < 0$ or ORI3D$(u, w, z, z + d_2) < 0$ **then**
      continue //$z, z + d_2$ is not on correct side
    return (f', c) //(3a), (3b)