Construction of a conformal hexahedral mesh from volume fractions : theory and applications

Nicolas Le Goff

université
PARIS-SACLAY

Construction of a conformal hexahedral mesh from volume fractions: theory and applications

Thèse de doctorat de l'Université Paris-Saclay

École doctorale n° 580, Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat: Informatique
Unité de recherche: Université Paris-Saclay, Univ Evry, IBISC, 91020, Evry-Courcouronnes, France
Référent: Université d'Évry Val d'Essonne

Thèse présentée et soutenue à distance, le 17 décembre 2020, par

Nicolas LE GOFF

Composition du jury:

Pascale LE GALL Professeur, CentraleSupélec	Présidente
Guillaume DAMIAND Directeur de recherche, CNRS LIRIS Université Claude Bernard	Rapporteur & Examinateur
Julien TIERNY Chargé de recherche, CNRS LIP6 Sorbonne Université, HDR	Rapporteur & Examinateur
Sylvie ALAYRANGUES Maître de conférences, Université de Poitiers (XLIM)	Examinatrice
Jean-Christophe JANODET Professeur, Université d'Évry-Val-d'Essonne (IBISC)	Directeur
Franck LEDOUX Docteur, Commissariat à l'énergie atomique et aux énergies alternatives	Co-Encadrant
Steven J. OWEN Docteur, Sandia National Laboratories	Invité

Thèse de doctorat

NNT: 2020UPASG033

Acknowledgements

I would like to thank again the members of my dissertation committee for having kindly agreed to evaluate this work, and I have enjoyed the questions and exchanges during the defense.
Likewise, I would like to thank my supervisors; Jean-Christophe has steered the work away from purely obtaining something that works in favor of also getting to know why it does work, and has shown great enthusiasm throughout those three years. As for Franck, I have been working with him for quite a few years now and I feel privileged to have been able to work alongside him both in a software engineer capacity and during this thesis.
Without the agreement and support of the managers at my company, this PhD would not have taken place and I am thankful for the trust that it represents.
Through existing collaborations we have worked with other people both for research and production purposes; in particular it has been gratifying to exchange and compare ideas on this topic with Steve over the years.
Finally, I am grateful to my colleagues, friends and family, who provided help, encouragements and support whenever the going got though.

I hope that this document will prove useful to you, the reader, and wish you all the best.

Contents

Chapter 1

Introduction

Numerical simulation is used to study the evolution of a system, and usually comes, when possible, with building prototypes, making experiments and careful observations. The physical phenomena themselves are handled in what are called *simulation codes*; they modelize the behavior of the system, which can be computationally intensive and can require to run on supercomputers.

In order to make a numerical simulation code run properly, one needs to prepare and craft some data. Such data can carry geometric pieces of information, like the description of the studied domain (a car, a bridge, a skull or body, etc.) and physical ones (pressure, temperature, material description, etc.). Both types of data are of interest in this work: geometric and physical. Whatever the applicative field is, one usually needs to discretize the geometric domain of interest into elementary cells that form what is called a *mesh*. A mesh will provide a partition of the domain into basic elements that are supports to apply traditional numerical methods, like Finite Element Methods (FEM) or Finite Volumes Methods (FVM), which rely on basic functions that are defined onto finite elements or finite volumes. Depending on the numerical methods, many types of meshes can be used. For instance, Figure 1.1 exhibits different kinds of 2D meshes that are made of triangular cells (a) or quadrilateral cells (b and c) , and that can be structured (b), i.e. all the internal nodes of the mesh are adjacent to the same number of cells[1], or unstructured (a and c), i.e. each node can be adjacent to a different number of cells. 3D meshes can be similarly classified.

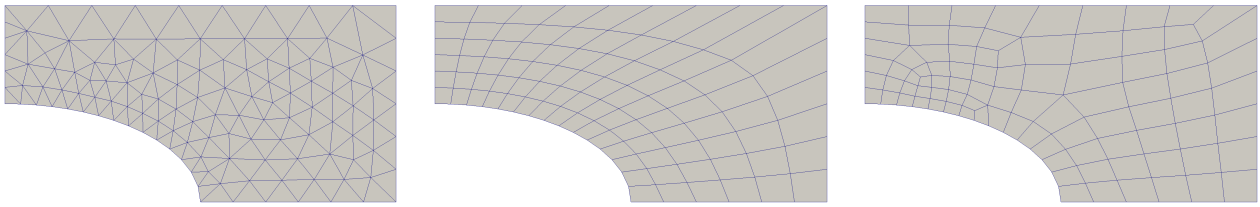


Figure 1.1: Different types of 2D meshes that were built starting from the same boundary discretization. In (a), an unstructured triangular mesh; in (b) a fully structured quadrilateral mesh; in (c) an unstructured quadrilateral mesh.

Looking at the discrete representation of the study domain, meshes can also differ in the way they match simulation materials. Let us consider a study domain made of two materials A and B that are in movement, and depicted in various ways on Figure 1.2. The first line shows the "physical" evolution without showing the mesh discretization, where A is colored in blue and B in red. The materials are moving according to the effect of the simulated physical phenomenon, leading to an expansion of A that tries to fill the whole domain, while B is contracted. The meshes are shown on the three remaining lines. At the initial time, the meshes are the same: a *pure* quadrilateral mesh, that is a mesh where each 2D cell is a quad and contains a single material (A or B). Three approaches are then possible:

- **Euler.** On the second line, the mesh is fixed and the materials move through the cells; when several materials are present inside one cell, the cell is called *mixed* and we denote such a mesh as being *Eulerian*. In this case the interface between A and B is lost.

[1] In this example, all the internal nodes are adjacent to exactly 4 faces while boundary nodes are adjacent to 1 or 2 faces.

- **Lagrange.** On the third line, the mesh moves at the same speed as materials do. Cells remain pure during all the simulation but their geometry can drastically change. We note such a mesh as *Lagrangian*, and the interface between A and B is totally defined by some mesh edges.

- **ALE.** Eventually, on the fourth line, the mesh moves but not at the same speed as materials do. Cells can become mixed and their geometry remains controlled. We qualify such a mesh as being *ALE*, for *Arbitrary Lagrangian-Eulerian*.

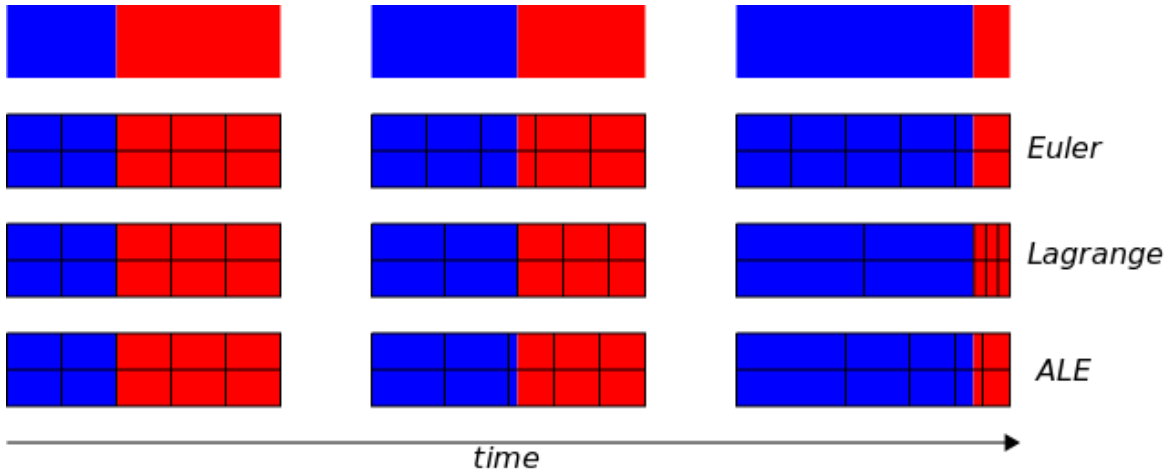


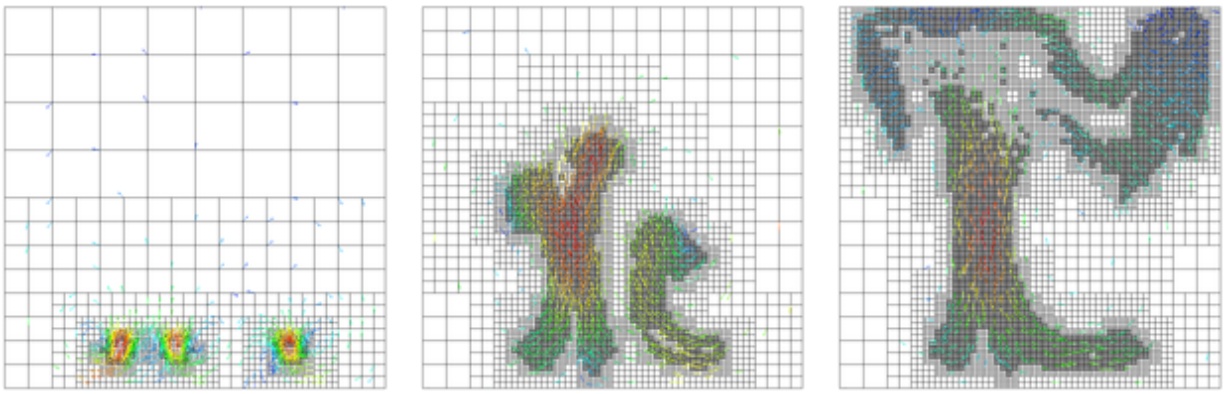
Figure 1.2: A 2D domain made of two materials (first line) and discretized in an Eulerian (second line), a Lagrangian (third line) and an ALE ways (fourth line).

1.1 From Euler to Lagrange intercode

The main context of our work is about converting data from Eulerian to Lagrangian codes, where the first ones act on Eulerian meshes and the latter ones on Lagrangian meshes. In the former, the mesh will in most cases be easy to generate, typically a grid, and the cells will be able to contain several materials, which will be expressed as *volume fractions*. Some of those Eulerian codes have Adaptive Mesh Refinement (AMR) capabilities, meaning that, usually by use of an octree-like data structure, the mesh is locally refined or coarsened to respectively track a phenomenon of interest or to reduce the execution time and the memory footprint of the simulation (see Figure 1.3). In the latter, i.e. the Lagrangian codes, a cell can only contain one material, which means that the mesh follows the interfaces between the components that form the domain of study, such as the different parts of a CAD assembly for instance. The requirements on those meshes can vary wildly: it can go from unstructured tetrahedral meshes to boundary aligned block-structured hexahedral meshes (see Figure 1.4), and it depends on what the simulation code can handle.

Conversion of data between codes are usual in industrial studies where many different codes are used to solve a complex multi-physics problem. It can be done in different ways going from loosely coupled codes, where codes are assembled in a pipeline and communicate by reading and writing files, to tightly-coupled codes, that are interleaved in a simulation loop and share data in memory. We focus here on loosely-coupled codes, where simulation codes are chained one after another. As each code has its own input and output requirements, an intercode tool is required to convert the output of one code into the input of the next one. In the case of converting the output of an Eulerian code into the input of a Lagrangian code, this task is not trivial at all, especially as we target the generation of a full hexahedral Lagrangian 3D mesh (see Figures 1.5 and 1.6). Right now in practice at CEA[2], the process of converting an Eulerian code output to an input that fits Lagrangian codes is done using interactive tools. This process requires from several days to weeks for an experimented engineer to create the expected hexahedral Lagrangian mesh. Our final aim is to provide a way to do it in an automatic manner with selecting a little set of parameters.

[2]To our knowledge, the same process is used in other laboratories facing the same issue.



(a) (b) (c)

Figure 1.3: Example of the evolution of the mesh when an Adaptive Mesh Refinement (AMR) technique is activated during a simulation from the deal.II [Bangerth et al. 2007] software suite tutorial.

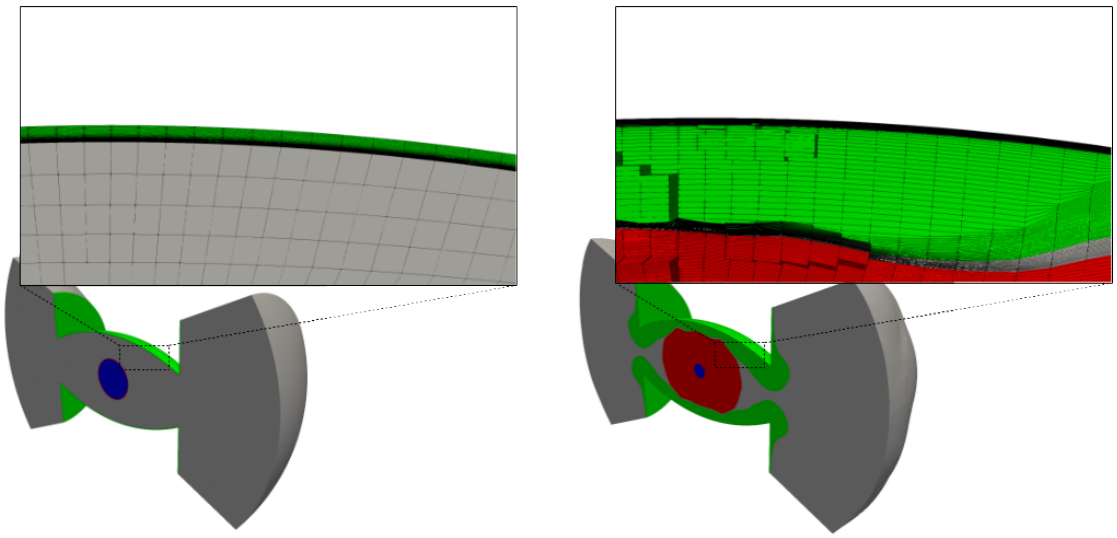


Figure 1.4: Example of the mesh deformation during a laser simulation. As the simulation moves forward from left to right we can see that the green and red materials have expanded greatly at the cost of the grey material; close-ups show the extent of the mesh deformation and illustrate the need for a "good" starting mesh, as no cell can become tangled during the simulation.

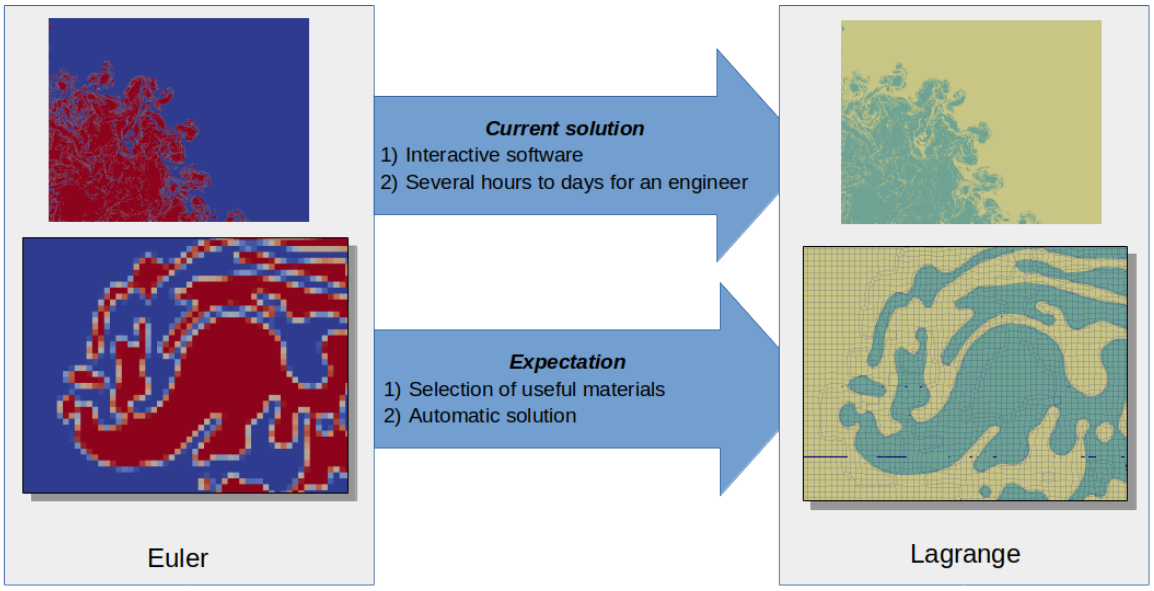


Figure 1.5: Example of the purpose of our work in 2D, with our input (left) a grid mesh carrying volume fractions in a two material CFD case from [Toro 2009] and our output (right) an unstructured quadrilateral mesh containing only pure cells. The presented hexahedral mesh (on the right) is obtained automatically with this work. Using interactive tools, we would get another hexahedral mesh (more structured) but that would necessitate a lot of engineer interactions.

Our goal is to further the automation of Lagrangian meshing from Eulerian data outputs. It can be split into two sub-objectives:

1. **Provide a valid geometrical model extraction process.** The current process to interactively generate a Lagrangian mesh from an Eulerian one is to: first extract a geometrical representation of the material interfaces, which are 3D surfaces connected along curves and points, then to inject those surfaces into a CAD-meshing software so that it can be meshed in 3D. In order to be usable, such surfaces need to be quite smooth and the topology of the overall model (surfaces, curves, points) must be as clean as possible. It means, for instance, that when two surfaces have to intersect one another along a curve, this curve must be simple and not made of a lot of disconnected pieces, and we must not have a lot of noisy tiny surfaces created. If we are not able to provide that, the engineering time – already huge in the best of cases – one has to spend on cleaning the model and meshing could be prohibitive. Extracting a geometrical model is useful for the current pipeline used at CEA (see Figure 1.5) but also for getting a geometry reference for the expected fully automatic solution.

2. **Automatically generate a valid hexahedral mesh.** Directly obtaining a usable mesh would be most beneficial; while what defines a "good" mesh varies between the codes and the cases run, some geometric quality criteria on the cells are common among the simulation codes which cannot operate when even one single cell becomes too distorted [P. M. Knupp 2001].

An overall constraint is to **avoid to deviate too much from the input data**, namely the material volume fractions. Indeed, the physical quantities must be preserved as best as possible between the two simulation codes that are chained. This constraint is both strong and loose:

- Strong because as this process is used in a physical simulation pipeline, it is mandatory to preserve physical quantities as best as possible to get “high-fidelity” results. In other words, we need to convert data from one code to another one without too many approximations. In our case, one of the main issues is the ability to generate a material interface between materials that are implicitly defined in the first code.

- Loose because this problem is overconstrained and so approximations are unavoidable. Getting both high-fidelity material preservation and a smooth clean geometrical definition of the material interfaces is quite difficult in many cases. Moreover during those intercode data transfers, some

materials can be deemed as being of higher priority by the engineer that drives the simulation or he can decide to remove some non-significant materials.

We can summarize our goal as follows:

> *Starting from a 3D Eulerian mesh M_E with a set of materials $\mathcal{M}$, we want to generate both an interface geometrical model $GeomM$, and a full hexahedral mesh M_L such that :*
>
> 1. *All the surfaces of $GeomM$ are as smooth as possible;*
> 2. *The topological structure of $GeomM$ is as "clean" as possible;*
> 3. *Cells of M_L fit minimum quality requirements to be used by a FEM or FVM simulation codes;*
> 4. *Every material $m \in \mathcal{M}$ is preserved as best as possible, in the meaning that the overall volume of m is similar in M_E and M_L and it is located at the same spatial location.*

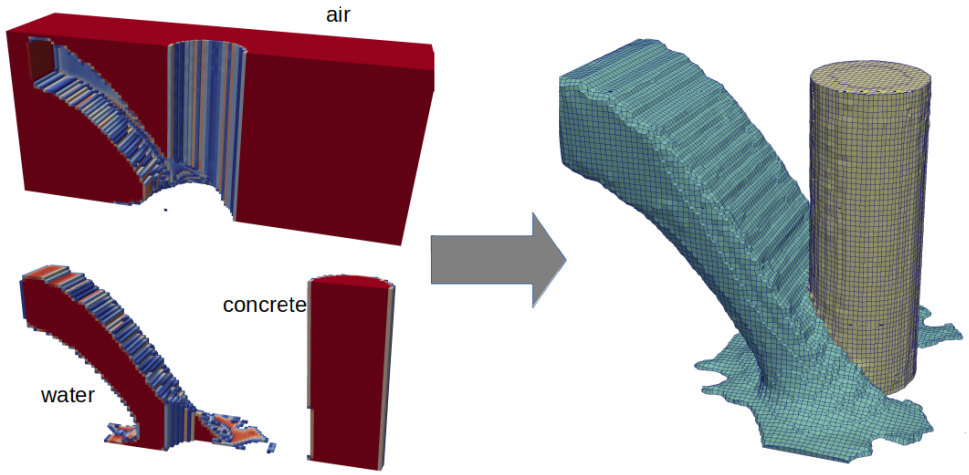
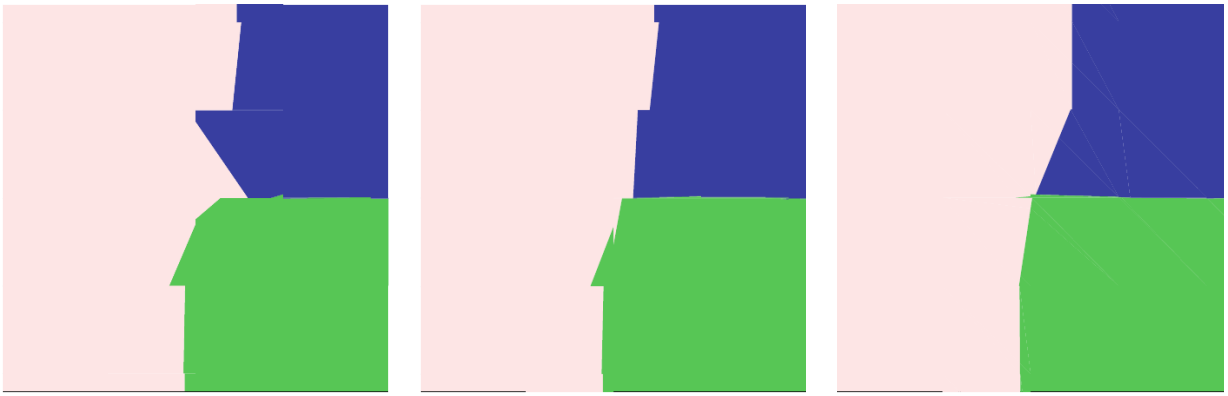


Figure 1.6: A 3D Example of what we want to achieve after having run the simulation code Gridfluid [Guy 2019] with water being poured against a concrete pillar. On the left, our input, a grid mesh carrying the volume fractions of respectively the water, the concrete pillar and the air. On the right our output Lagrangian mesh (the air is meshed but not shown here).

1.2 Proposed solution and main contributions

When looking at existing techniques we consider that two different categories of approaches seem to be able to address our problems: interface reconstruction methods and overlay-grid meshing algorithms.

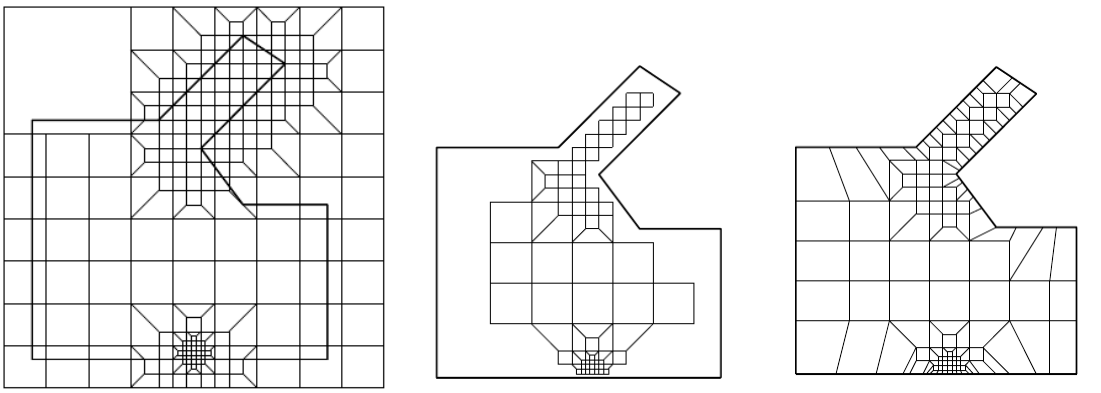
First are the interface reconstruction methods [Kucharik et al. 2010]. These methods take an Eulerian mesh as an input and reconstruct the interfaces between materials inside each cell; they are classically used in ALE simulation codes and in associated scientific visualization software [Childs et al. 2012; Ahrens et al. 2005]. While the volume fractions preservation is actually enforced by design, a limitation of those methods is that the obtained interfaces do not fit our purpose, as they are jagged, not continuous and potentially with small slivers of materials (see Figure 1.7) and we would be hard pressed to use them as an input in a CAD-meshing software and for projecting mesh nodes associated to those surfaces without any significant modification, which would in turn render the volume preservation property null and void.



(a) VOF green first (b) VOF-PD (c) MOF

Figure 1.7: Illustrations from [Kucharik et al. 2010] showing several interface reconstruction methods applied on the triple point problem. VOF stands for "Volume-of-Fluid", VOF-PD for "VOF with power diagram" and MOF for "Moment-of-Fluid".

Secondly figure among the variety of hexahedral meshing techniques the overlay-grid methods [R. Schneiders 1996] where a shape – an explicit geometrical CAD model – that needs to be meshed is embedded into a mesh that discretizes its bounding box. Said mesh will usually be a grid, easy to generate and possibly refined locally [Robert Schneiders et al. 1999] (see Figure 1.8); its cells are assigned to the components of the models and those outside discarded, and the mesh is then deformed or a padding layer is inserted in order to capture the geometric features of the model. Contrary to the interfaces reconstruction methods, extracting a geometrical model from that mesh will give a relatively smooth model with a clean topology, but as a drawback one does not preserve the volume of the materials as overlay grid algorithms have difficulty capturing features with sharp angles. There are additional incompatibilities with our aim: first the expected inputs of those methods are CAD models, not meshes carrying volume fractions; secondly, most of those methods are designed to mesh only one component and cannot be used to mesh a CAD model that is the assembly of several pieces, while we have several materials. Thirdly, those methods heavily rely on the ability to generate an adequate initial mesh, possibly with local refinements to offer more robustness, to better capture the CAD or to provide a modicum of volume preservation [X. Gao, Shen, et al. 2019]. This is again in direct conflict with what we need, as in our case the input mesh is fixed as part of our input.



(a) initial mesh (b) kept cells (c) padding layer inserted

Figure 1.8: Illustrations from [Robert Schneiders et al. 1999] showing an overlay-grid method based on an octree mesh. (a) an initial single englobing cell is repeatedly split and refined to be adapted to the domain; (b) only the in-domain cells are kept; (c) boundary cells are added to fill-in the whole domain.

As stated before, our first aim is to get a clean geometrical and topological model. We propose to adopt an overlay-grid technique that meets our concerns, and more specifically, we decided to extend the Sculpt algorithm [Owen, M. L. Staten, and Sorensen 2012; Owen, Brown, et al. 2017], which implements an overlay-grid approach considering volume fractions data as an input. A complete presentation of the Sculpt algorithm will be given in Section 2.3. This work has required to make

several improvements in order to fit our requirements, and it led to the definition of a new algorithm, called ELG that stands for *Euler to LaGrange remeshing*, which is a complete pipeline of different treatments (see Figure 1.10) that works both in dimension 2 and 3. It comes with the following main contributions:

- **Volume preservation measure and control:** We implemented a post-process algorithm where, as the volume fractions are not preserved by SCULPT, we wanted to evaluate the difference between the output of the algorithm and its input. It is done in the last step of our process in the "*Discrepancy-Driven Mesh Deformation*" box in our pipeline. We defined in [Le Goff, Ledoux, and Owen 2018] a discrepancy criterion, which is computed on each cell of the volume fraction mesh. It gives us locality information, on the meaning that locally to each input cell we know how each material is preserved in terms of volume. Additionally, we used the computation of the discrepancy to provide us with a feedback on whether cells of the output mesh are too large or too small (meaning that we locally have too much or too little of a material), and introduced a discrepancy-controlled loop so as to reduce it by moving the interface nodes while ensuring the cell quality did not drop below a user-defined threshold;

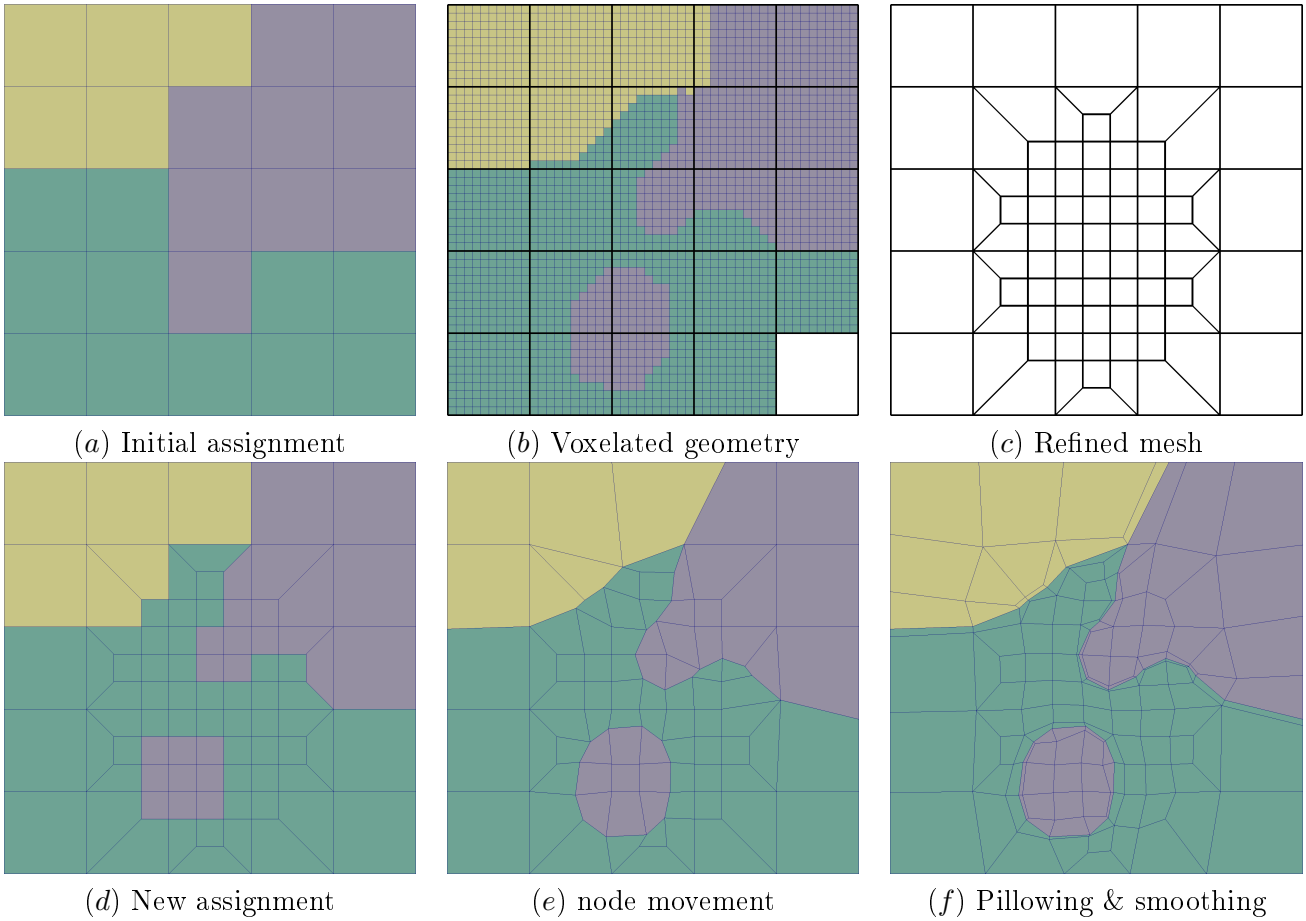
- **Geometrical model and material interfaces extraction:** we studied a discrete interface reconstruction method, where the problem is defined as follows: all the mixed cells and their adjacent pure cells are subdivided into so-called voxels, and we assign a material on those voxels. Voxels spawned from pure cells are assigned to the material of their corresponding pure coarse cell, leaving those spawned from mixed coarse cells as "free" to be assigned. We implemented several methods that we published and presented in [Le Goff, Ledoux, and Janodet 2019a; Le Goff, Ledoux, Janodet, and Owen 2019], with some inspired from graph partitioning techniques. From this we can extract an alternative geometrical model that can be used to project mesh nodes on. In the pipeline shown on Figure 1.10, it is the first computation that we do since it will help us to guide mesh deformation afterward;

- **Hexahedral mesh quality control:** We introduced several modifications to the initial pipeline, especially as it does not consider the cell quality until the very last smoothing step (see Figure. 2.22 at page 35); basically in such an algorithm we move the nodes, change the mesh topology and hope for the best with a smoother (see Figures 2.22.f, g and h). In our context, where our input is the output of a simulation code, we have shown in [Le Goff, Ledoux, Janodet, and Owen 2019] that depending on the mesh resolution the code ran with, we can end up with a good quality mesh (no inverted cells) or a bad quality one. It is unrealistic to ask engineers to run their simulations again with different resolutions at random, assuming it is even feasible; this exhibits the need for an algorithm that consistently works. We proposed in [Le Goff, Ledoux, Janodet, and Owen 2019; Le Goff, Ledoux, and Janodet 2019b] an alternative where the ELG pipeline controls the mesh quality at each step. It is done at the "*Quality-driven mesh projection*" stage;

- **Input mesh controlled adaptation:** Looking at the geometrical model we build in stage 1.2 on Figure 1.10, we can decide to refine the initial mesh in order to fit it as best as possible. Our aim here is to mimic the techniques available in some overlay-grid methods. For instance, let us consider the example of Figure 1.9 where the initial cell assignment in (a) does not have the same structure of materials than the voxelated geometry in (b). As a consequence, we refine the initial mesh (c), compute new volume fractions on it using (b) and we reassign materials (d) before moving the interface nodes in (e), and applying topological modifications and smoothing in (f).

1.3 Manuscript structure

The remainder of this document is structured as follows. In chapter 2, we introduce notions and definitions that are used throughout this manuscript, along with a survey of mesh generation methods with an emphasis on hexahedral meshes, and more precisely on overlay-grid methods and the SCULPT algorithm, which serves as the starting point of our study. This work led us to adapt the



(a) Initial assignment (b) Voxelated geometry (c) Refined mesh

(d) New assignment (e) node movement (f) Pillowing & smoothing

Figure 1.9: Example of input mesh refinement in order to fit the built geometrical model.

algorithm in many ways to finally propose our own ELG pipeline, which is illustrated on Figure 1.10. Individual steps of this pipeline will be presented in the following chapters.

In chapter 3, we describe the post-process that we devised and applied on Lagrangian meshes in order to measure and improve their volume preservation compared to the Eulerian meshes they originated from. Even if it is the last step of our pipeline, it also motivates why we had to adapt the initial SCULPT algorithm to fit our requirements.

In chapter 4, we study several variations on discrete interface reconstructions and the way we link the extracted geometrical model to its overlay-grid counterpart in order to project the mesh nodes associated to these models.

In chapter 5, we introduce mesh quality control and mesh refinement stages in order to ensure getting both a minimum cell quality (which is required by simulation codes) and a mesh that fits the topology of the geometrical model extracted in chapter 4.

Eventually, in chapter 6, we will conclude on this work and give some evolutions the study of which could benefit our proposed solution, whether by improving the results (pillowing in Section 6.2.1) or extending the usefulness of our method (cell size control in Section 6.2.2). We will also present technical details about our implementation, namely the multithreaded (6.2.4) and the components programming model (6.2.3) we used[3].

As a foreword, one should note that :

- the techniques depicted in this work are not limited to Eulerian to Lagrangian intercode problems: they are relevant as long as one is able to provide a mesh carrying volume fractions, which is typically the case for the examples issued from CAD models that we have used;
- similarly, while the majority of the inputs that we will show in our study are grid meshes, we are not limited to those meshes and can handle any unstructured conformal hexahedral meshes.

[3]Both of them were presented at [Le Goff, Ledoux, and Janodet 2018].

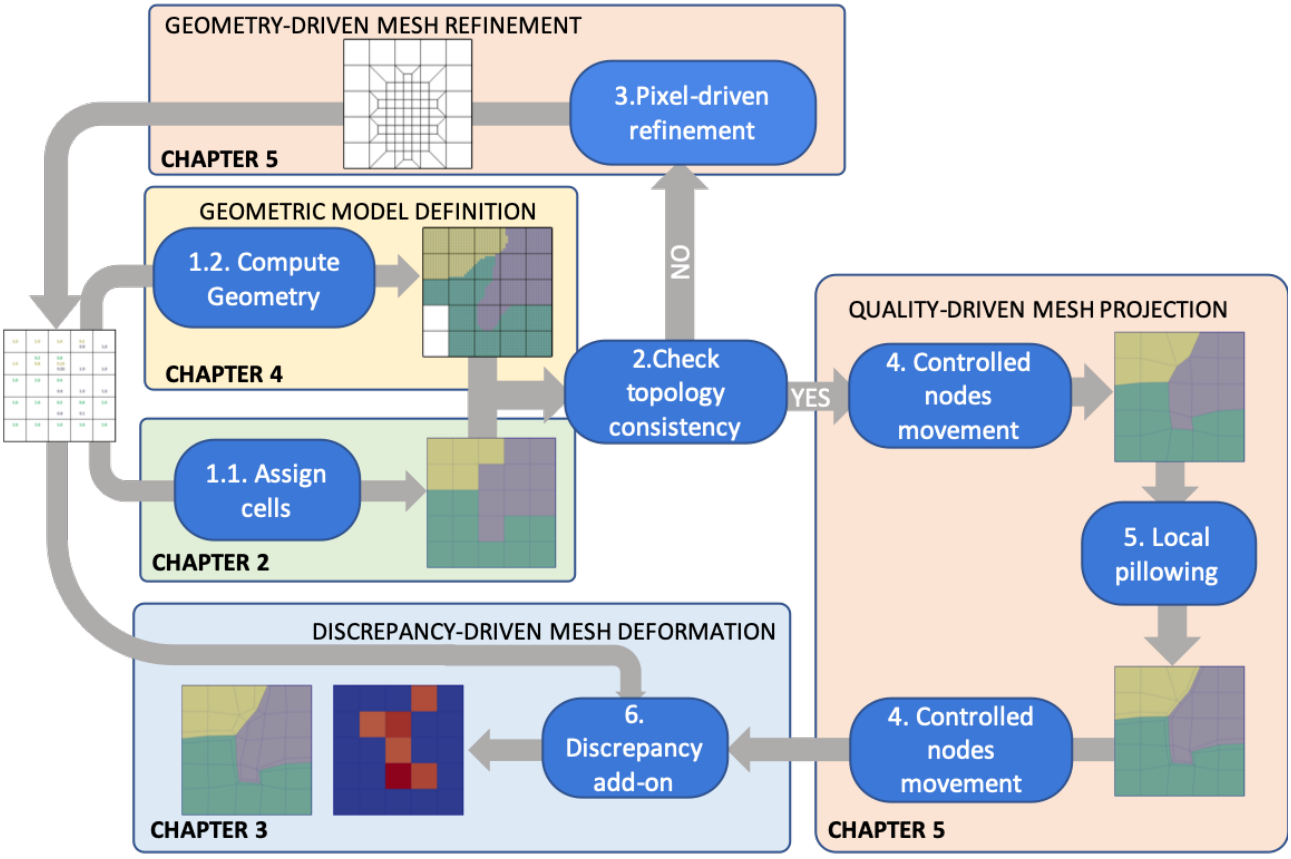


Figure 1.10: The ELG pipeline that we developed.

We do not handle non-conformal meshes, such as those shown in Figure 1.3 because as our method consists in using the input mesh as a base for our overlay-grid algorithm this base mesh should meet the requirements, first of all being conformal;

- Most of the steps are not restricted to hexahedra and can directly accommodate other types of cells, such as tetrahedra and prisms, with the caveat of course that our output would not be an hexahedral mesh, but it is not the focus of our work.

Our publications

Le Goff, Nicolas, Franck Ledoux, and Jean-Christophe Janodet (2018). “A Parallel Shared-Memory Implementation of an Overlay Grid Method”. oral talk. Symposium on Trends in Unstructured Mesh Generation.

— (2019a). “An Overlay Grid Driven Geometric Model Extraction”. oral talk. Symposium on Trends in Unstructured Mesh Generation.

— (2019b). “Hexahedral Overlay Grid Method with Guaranteed Element Quality”. oral talk. International Conference on Adaptive Modeling and Simulation.

Le Goff, Nicolas, Franck Ledoux, Jean-Christophe Janodet, and Steven J. Owen (2019). “Guaranteed quality-driven hexahedral overlay grid method”. In: *Proceedings of the 28th International Meshing Roundtable*.

Le Goff, Nicolas, Franck Ledoux, and Steven J. Owen (2018). “Hexahedral mesh modification to preserve volume”. In: *Computer-Aided Design* 105, pp. 42–54.

Chapter 2

Hexahedral meshing: definitions and main algorithms

Our work focuses on converting Eulerian data into a Lagrangian hexahedral mesh that could be used by a simulation code. To understand it, many notions have to be handled; we describe those that are useful for our work just after in Section 2.1. As our work consists in generating hexahedral meshes, Section 2.2 is dedicated to an overview of the main families of hexahedral meshing techniques and we conclude this chapter with a description of the SCULPT algorithm, which is the starting point of our work.

2.1 Notions and definitions

Many definitions and notions do not depend on the dimension n of the embedded space or the object we consider, n being a strictly positive integer value. That is why we will often talk about "n-dimensional objects", i.e. objects of dimension n. Note that in this work we focus on spaces of dimension 2 and 3.

2.1.1 Meshes and cells

Considering an n-dimensional geometric domain Ω, a mesh M of Ω partitions Ω into a finite set of simple n-dimensional elements, which are called n-dimensional cells, n-cells, or more simply cells. With M being a partition of Ω, it roughly means that Ω is totally covered by cells that belong to M that do not overlap with one another (see Figure 2.1). In other words, a single point of Ω belongs to one and only one cell of M.

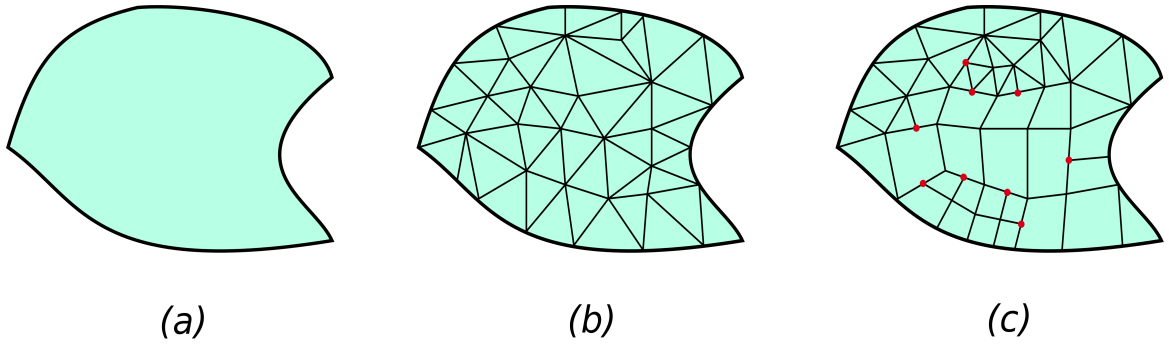


Figure 2.1: 2D examples showing a simple domain Ω in (a), a triangular conformal mesh of Ω in (b) and a non-conformal mixed mesh in (c) where red dots correspond to non-conformal nodes.

An n-dimensional cell is composed of i-cells with $0 \leq i \leq n$. 0-cells are usually called *nodes* or *vertices*, 1-cells are called *edges*, 2-cells are called *faces* and 3-cells are called *regions*. An n-dimensional mesh is said to be *conformal* if any couple of n-cells of M share at most one $(n-1)$-cell. Otherwise, it is *non-conformal*. Cells of a mesh can be classified by looking at their combinatorial structure, or topology. For instance, very generic and simple meshes are the *simplicial* meshes,

which are made of triangles in 2D and tetrahedra in 3D. As they only contain a single type of n-cells, such meshes are qualified as being *full*-triangular or *full*-tetrahedral. A 3D mesh only made of hexahedral 3D cells will similarly be called *full*-hexahedral. On the contrary, an n-mesh made of several types of n-cells, such as triangles, quadrilaterals and polygons in 2D, will be qualified as being *mixed*. Eventually, we also recall the notions of *structured* and *unstructured* meshes, which are relative to the overall topological structure of a mesh. An n-mesh is *structured* if all of its inner nodes are shared by the same number of n-cells and unstructured otherwise. In 2D or 3D, the simple case of structured mesh is a *grid*.

In the context of this work, an n-mesh is used as a support for numerical simulation and it can be an *Eulerian* or a *Lagrangian* mesh. Considering a set of materials $\mathcal{M} = \{m_1, m_2, \ldots, m_p\}$, every n-cell of a Lagrangian mesh is *pure*, i.e. made of a single material of $\mathcal{M}$, while an n-cell of an Eulerian mesh can contain several materials of $\mathcal{M}$. It leads us to the two following definitions that will be useful in the remainder of this document.

Definition 1 (volume fraction) *Let Ω be a geometric domain partitioned by a set of materials $\mathcal{M}$, $V(.)$ a function that gives the volume of an entity, M be a mesh that discretizes Ω and a cell $c \in M$, then the* volume fraction $f_{c,m}$ *of a material m in c is defined as:*

$$f_{c,m} = f(c,m) = \frac{V(c \cap m)}{V(c)} \tag{2.1.1}$$

where $c \cap m$ is the geometric intersection between c and m. By definition, for any cell $c \in M$, we have $\sum_{m \in \mathcal{M}} f_{c,m} = 1$.

Definition 2 (material assignment) *Let $\mathcal{M}$ be a set of materials, M a mesh and a cell $c \in M$ then the material assignment of c is a function that returns a material $m \in \mathcal{M}$.*

2.1.2 The classification relation between a mesh and a geometric domain

Defining a geometrical domain Ω can be done in many ways. For instance, it can be defined using a function that indicates for any point p of the space whether p belongs to Ω. In 3D, it can also be obtained as the result of a series of Boolean operations, which are unions, differences and intersections of solid primitives, or defined as a set of explicitly connected curves and surfaces defining the boundary of Ω, denoted $\partial\Omega$. The process of building Ω with Boolean operations is called CSG, for *Constructive Solid Geometry*, while only representing $\partial\Omega$ as a set of connected geometrical curves and surfaces is called *BRep*, for *Boundary Representation*.

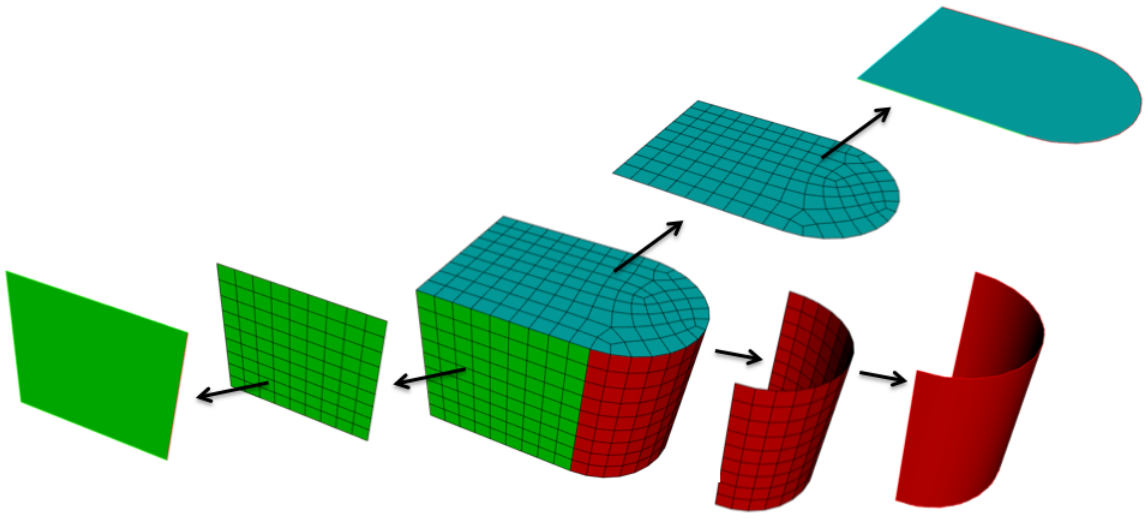


Figure 2.2: Classification of a mesh onto a geometrical model; cells assigned to the same geometrical boundary entity are drawn with the same color.

In the case of Lagrangian meshes, we often need to know if a cell is inside Ω or outside. In 3D, regions are by definition assigned to be inside or outside of Ω depending on whether they contain the

Ω material or not. But faces, edges and nodes can be inside, outside or on the boundary of Ω. When we use a BRep representation G_Ω for Ω, faces, edges and nodes of M are said to be *classified* on surfaces, curves or points of G_Ω or inside a material (see Figure 2.2). Such a classification is useful in order to perform geometric smoothing operations that intend to move nodes in order to improve the geometric quality of the adjacent cells. When moving a node classified on a surface or a curve, a constraint is added, which is to remain as close as possible to this geometric entity.

2.1.3 Geometric quality of a mesh

In numerical simulation, defining "mesh quality" strongly depends on the simulation code requirements; however, a simple and frequently admitted criterion is related to the geometric quality of the cells that compose the mesh: usually, cells with a negative volume are to be avoided. It is for instance the case for numerical codes based on the finite element methods, or FEM. When discretizing the problem with finite elements, shape functions are usually defined for each cell of the mesh. These functions allow the connexion of any cell to a *reference element*, which is perfectly regular. For example, the reference element for an hexahedron is the unit cube (see Figure 2.3). The resolution settings are well-known for the reference element. For each cell c, a shape function T_c maps the reference element onto c, with a change of variables between the reference space and the real space. In dimension 3, we note (x, y, z) (respectively (x, y) in dimension 2) the coordinates of a point in the real space (or physical space) and (ξ, η, ζ) (respectively (ξ, η) in dimension 2) the coordinates in the reference space (or logical space).

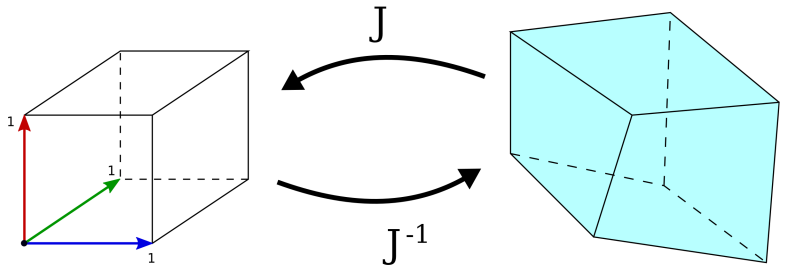


Figure 2.3: From the reference cube to a physical hexahedral element and vice versa.

Each shape function is bijective at any point on the reference element or on its boundary. To any point of the reference element corresponds one and only one point of the physical element, and vice versa. Similarly, each boundary part of the reference element corresponds to a boundary portion of the real element. A shape function is bijective if the Jacobian matrix J corresponding to the change of variables is not singular. It can be checked by verifying that the determinant of J is never equal to zero for any point of the reference element. In other words, it is strictly positive. In dimension 2, the Jacobian matrix J_2 is defined by the following relationship:

$$\begin{pmatrix} \frac{\partial T}{\partial \xi} \\ \frac{\partial T}{\partial \eta} \end{pmatrix} = \begin{pmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{pmatrix} \begin{pmatrix} \frac{\partial T}{\partial x} \\ \frac{\partial T}{\partial y} \end{pmatrix} = J \begin{pmatrix} \frac{\partial T}{\partial x} \\ \frac{\partial T}{\partial y} \end{pmatrix}. \tag{2.1.2}$$

Similarly, in dimension 3, we obtain the definition of J_3:

$$\begin{pmatrix} \frac{\partial T}{\partial \xi} \\ \frac{\partial T}{\partial \eta} \\ \frac{\partial T}{\partial \zeta} \end{pmatrix} = \begin{pmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta} \end{pmatrix} \begin{pmatrix} \frac{\partial T}{\partial x} \\ \frac{\partial T}{\partial y} \\ \frac{\partial T}{\partial z} \end{pmatrix} = J \begin{pmatrix} \frac{\partial T}{\partial x} \\ \frac{\partial T}{\partial y} \\ \frac{\partial T}{\partial z} \end{pmatrix}. \tag{2.1.3}$$

In dimensions 2 or 3, the Jacobian matrix of a triangle, respectively a tetrahedron, T, defined by vertices $\{s_i(x_i, y_i, z_i)\}_{i\in[0,n]}$, with $n = 2$ or 3 can be computed at the vertex s_0 by respectively:

$$|J_2| = \begin{vmatrix} x_1 - x_0 & y_1 - y_0 \\ x_2 - x_0 & y_2 - y_0 \end{vmatrix} \quad \text{and} \quad |J_3| = \begin{vmatrix} x_1 - x_0 & y_1 - y_0 & z_1 - z_0 \\ x_2 - x_0 & y_2 - y_0 & z_2 - z_0 \\ x_3 - x_0 & y_3 - y_0 & z_3 - z_0 \end{vmatrix} \tag{2.1.4}$$

To verify that a cell is regular, it is thus necessary to know if the associated shape function is bijective. A cell is detected as irregular if the determinant of the Jacobian matrix is negative or zero for at least one of its nodes. For a single hexahedron, there will be eight such matrices, one per corner of the hexahedron (for additional discussion on elements with multiple Jacobian matrices see [P. M. Knupp 2001]). The minimal determinant of these eight matrices is known as the *Jacobian* metric of the hexahedral element. The lengths of the sides of the hexahedron can be normalized to provide a "scaled" version of this metric that will have values between -1 and 1 (see Figure 2.4). A scaled Jacobian determinant between -1 and 0 indicates non-convexity of the element. A scaled Jacobian value of 1.0 indicates an hexahedron with interior angles between common edges of 90 degrees. Generally acceptable scaled Jacobian values range between 0.2 and 1.0.

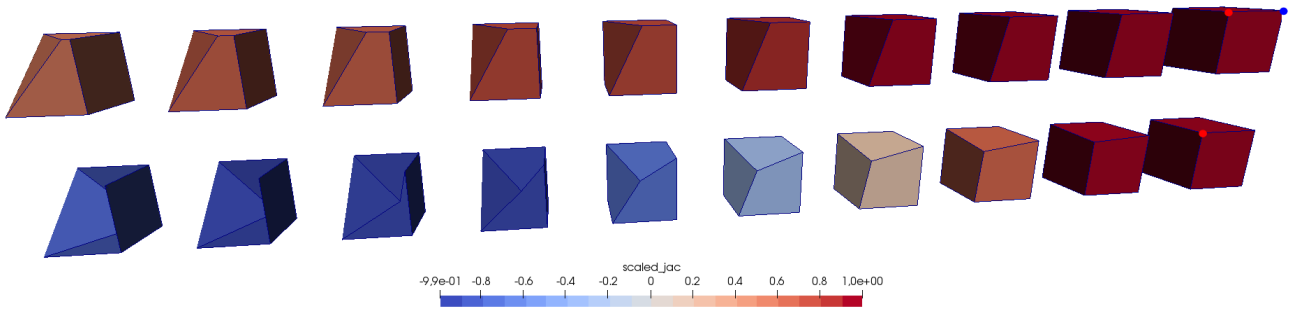


Figure 2.4: Evolution of the scaled Jacobian value when progressively moving one corner of a perfect cube from right to left. On the top row, the red corner is progressively moved to get to the blue one; on the bottom row, the red corner moves to the opposite node in the hex (along the diagonal).

2.1.4 Hexahedral meshing : the issue of dealing with global constraints

Tetrahedral mesh generation, while still the subject of improvements on many aspects, such as cell quality, constraints on cell sizing, edge length gradation control, mesh adaptation during simulations, performances,..., can be considered feasible. Given an arbitrarily complex CAD model, software like [Distene 2020; Si 2015; Geuzaine and Remacle 2009] are in most cases capable of producing a good quality tetrahedral mesh. On the contrary, reliable automatic hexahedral mesh generation is still an open problem. The main reason that explains such a difficulty is the topological structure of hexahedral meshes that does not allow algorithms to be built based on local modification operations. Let's take a look at simple operations such as inserting a point or collapsing an edge in a mesh that we can see in the 2D cases shown in Figures 2.5 and 2.6. In both cases, a triangular and a quadrangular meshes are modified.

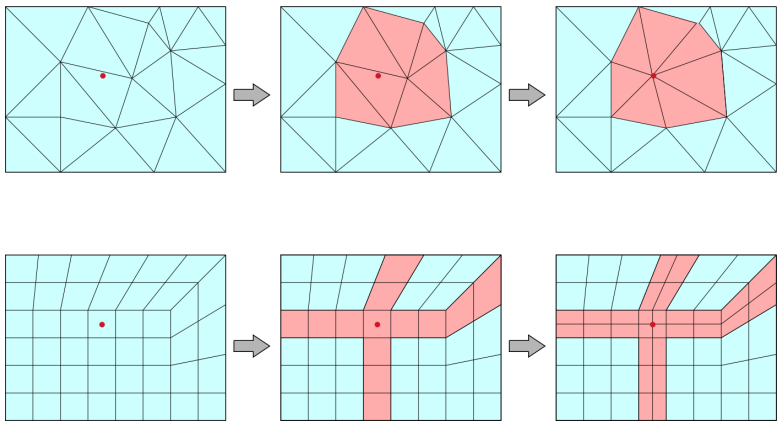


Figure 2.5: Example of point insertion for a triangular (top) and a quadrilateral meshes (bottom).

In Figure 2.5, a point is inserted in the mesh. A classical way to proceed in a triangular mesh is to first detect the triangle the point belongs to, then traverse adjacent triangles while a certain

condition is met. This set of triangles (in red) defines a cavity, which will be made empty by removing all the triangles it contains before being triangulated anew by creating one triangle for each of the cavity boundary edge and the new point. In the case of quadrilateral meshes, after detecting the quadrilateral that contains the point, we traverse the lines of quadrilaterals it belongs to (two lines or one self-intersecting line). Those lines are inflated to insert two new lines of quadrilaterals in the mesh. We can note that such an operation is not restricted to the local vicinity in the mesh and can propagate far away. In Figure 2.6, another common operation is performed, the edge collapse. Again, we can see that collapsing an edge in a triangular mesh is local. For the quadrilateral mesh, a direct edge collapse would lead to creating two triangles in the mesh, so we have to propagate the edge collapse along a complete line of quads in order to preserve the full-quadrilateral nature of the mesh.

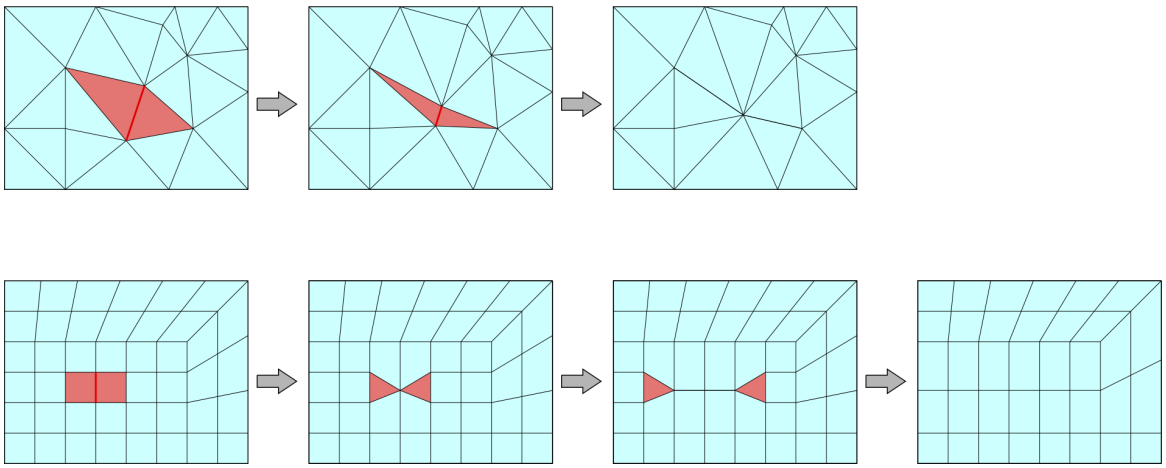


Figure 2.6: Example of edge collapsing for a triangular (top) and a quadrilateral meshes (bottom).

In 3D, hexahedral meshes are structured in the same way as quadrilateral ones (see Figure 2.7). Their dual structures are simple arrangements of curves (for quadrilateral meshes) and surfaces (for hexahedral meshes). This structure was first introduced with the term *spatial twist continuum* [Murdoch et al. 1997]. As a consequence, modifying such meshes require to modify their structure. In practice, in 3D, it can be done by performing operations on layers of hexahedral cells, i.e. a set of hexahedral cells that corresponds to a dual surface, or columns of hexahedral cells, i.e. the set of hexahedral cells that corresponds to the intersection of two dual surfaces.

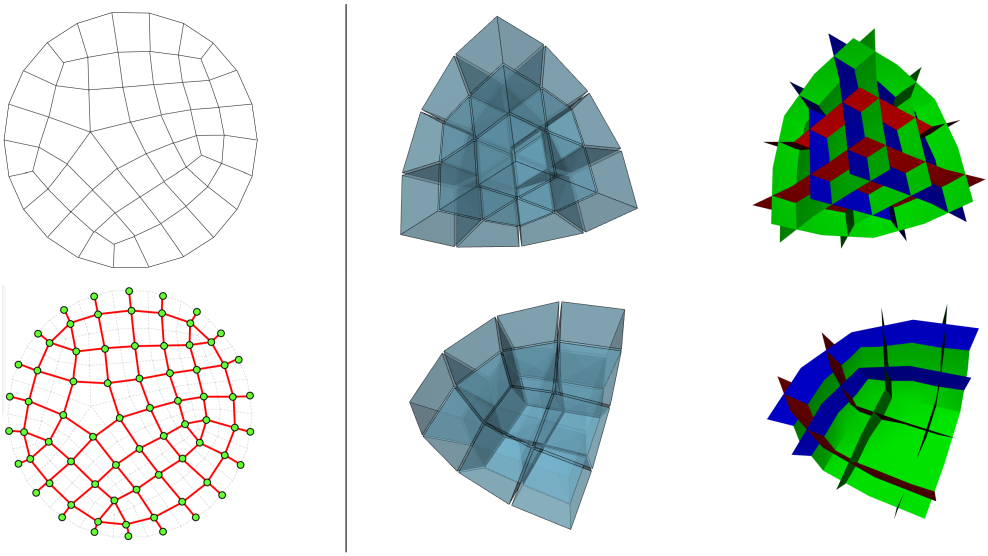


Figure 2.7: The dual of a quad mesh (left) and an hexahedral mesh (right) are respectively structured as a simple arrangement of curves and surfaces.

2.1.5 Why use hexahedral meshes?

Unlike tetrahedral meshes, two main benefits of hexahedral meshes are broadly accepted:

1. From a quantitative point of view, for the same edge size, the number of hexahedral elements to partition a geometrical domain is far smaller than the number of tetrahedral elements;
2. From a qualitative point of view, hexahedral elements are preferred in many physics fields (structure mechanics, computational fluid dynamics – CFD) since the hexahedral structure allows the alignment of the cells in the direction of the fluid displacement, or a wave shock, leading to minimizing some numerical diffusion issues. For this reason many finite-element solvers behave better when hexahedral elements are used.

The first point is particularly important for simulations performed on a single computer where memory is a limitation: for the same degree of accuracy, the memory footprint is lower with an hexahedral mesh than with a tetrahedral mesh. Additionally, the execution time of the simulation is generally lower with hexahedral elements. Indeed, numerous approaches are based on Finite-Element Methods (FEM), where n-cells ($n = 2$ or 3) are traversed numerous times, making the number of cells crucial. Moreover, the time step used during the simulation often depends on the geometrical size of cells (the smallest edge length for instance); again, the number of cells is an important factor. Concerning the second point, it must be noted that the behavior of finite-element solvers strongly depends on the applicative field. For CFD solvers, it can be motivated by the alignment of hexahedral elements with fluid displacement direction [Ferziger and Peric 2002]. It also means that not every hexahedral mesh is usable: it sometimes must respect alignment properties. For the same reasons, in aerodynamics, structured hexahedral meshes are preferred to discretize thin layers, i.e. interface areas between an object – an aircraft wing for instance – and a fluid in motion that surrounds it. In fact, we will find mathematical approaches that best work with hexahedral meshes in several fields like explicit dynamic [Zienkiewicz, Rojek, et al. 1998; Zienkiewicz, Taylor, et al. 2013; T. Hughes 2000] or cardio-vascular simulations [Liu et al. 2004].

For these reasons, a lot of the simulation codes used in the industry rely on physical models that can only be applied on or get better results with hexahedral meshes. An engineer who wants to use those codes must then be able to provide such meshes. Unfortunately, generating hexahedral meshes is very time-consuming; this task even dominates the overall simulation process. In [T. J. R. Hughes 2004], the author stated that about one million finite element analysis was performed each day in 2004. The meshing time was then about 80% of the full analysis process time. The importance of the meshing step is underlined in the same proportions in [T. Blacker 2000; Boggs et al. 2005].

If a pure quantitative point of view can justify to use hexahedral meshes, the qualitative point of view suggests that not all hexahedral meshes can be used for a specific applicative field. As a consequence, we often require to have well-shaped elements along some geometrical domain features (its boundaries), or simulation-specific features (fluid interfaces). The automatic generation of hexahedral meshes is a very difficult task and, up to now, there does not exist a satisfying solution. By satisfying we generally mean that the following criteria must be considered (see Figure 2.8 for an illustration):

1. **Individual quality of hexahedral elements** - The shape of an hexahedron must often be as close as possible to the shape of a cube or a block, i.e. it should have planar faces, with opposite edges similar in size and adjacent edges forming 90 degree angles;
2. **Global structure** - The "perfect" structure of an hexahedral mesh is a regular grid. In other words, each inner vertex should be adjacent to exactly eight hexahedral elements. Such a vertex is said *regular*. Otherwise, it is said *irregular* or *singular*. It is often impossible to verify both criterion 1 and this criterion, thus algorithms try to minimize the number of singular vertices;
3. **Boundary alignment** - Hexahedral elements must be aligned along the geometrical domain boundary. It basically means that they should form layers of hexahedral elements that follow the geometrical domain boundary. If we only consider the surfaces that compose the geometrical domain boundary, generated quadrilateral elements must respect main curvature directions and sharp features;

4. **Respect of a size map** - A size map should be considered in order to specify what the expected size of a cell is in any location of the geometrical domain. When the expected size strongly changes for close locations it is difficult to verify this criterion and criteria 1 and 2 at the same time.

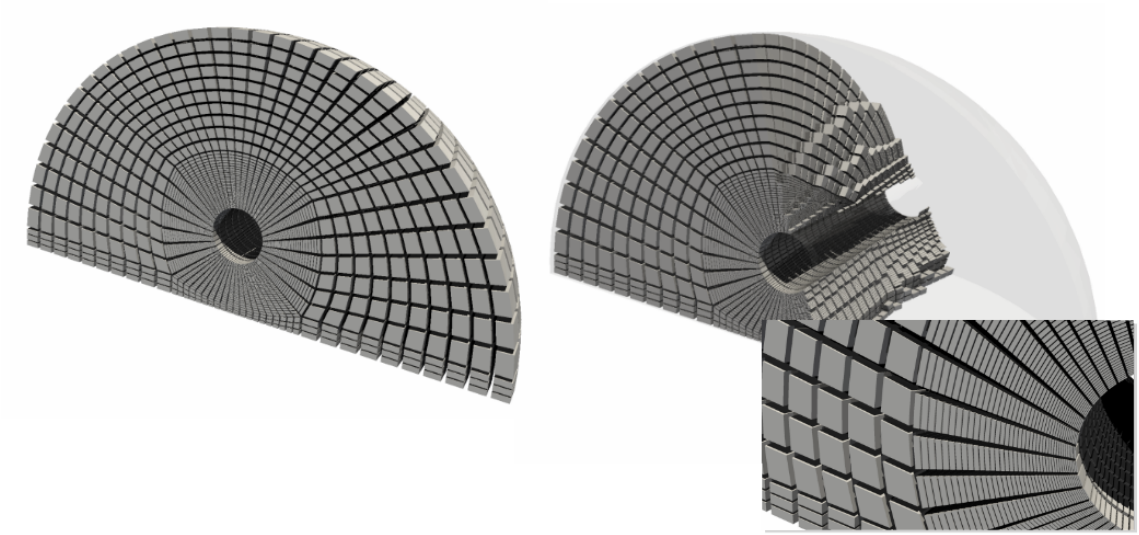


Figure 2.8: Example of a block-structured hexahedral mesh.

These criteria only consider the geometrical and topological quality of a mesh, but do not consider the simulated physical phenomena. In fact, the prior criteria to satisfy is the adequacy of the mesh and the simulation codes. The mesh can be very good from a geometrical and topological point of view, but unusable in practice for a particular simulation code or study.
Still, these criteria are usually the ones that can allow us to determine if a mesh can actually be used in practice. For instance, let us consider a simple way to generate an hexahedral mesh. It consists in generating a tetrahedral mesh[1], then splitting each tetrahedron into four hexahedra (see Fig. 2.13). This approach is automatic and robust. Unfortunately, such a mesh does not fulfill most of the previously given criteria. For instance, it does not consider any global structure to get a regular mesh or the boundary alignment constraint. For the same reasons, *overlay grids* are seldom used. The mesh is well-structured inside the geometrical domain but not at all along the geometrical boundary. For such algorithms, the insertion of hexahedral layers along the boundary improves the quality along said boundary (criterion 3) but does not make for a better global structure (criterion 2).

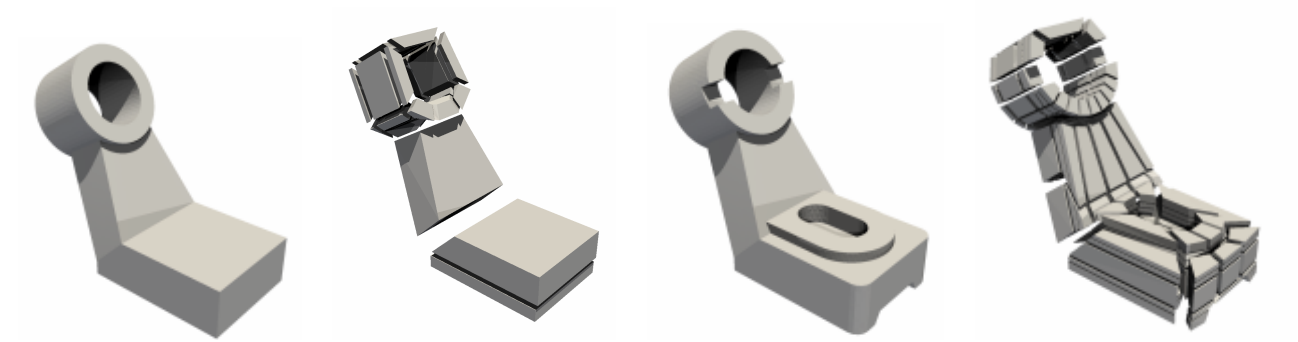
Software like CUBIT [Cubit 2019] or ICEM-CFD [ICEM 2013] propose solutions where the user must prepare the geometrical domain to be meshable. It mainly consists in subdividing it into meshable parts, where a basic algorithm can be applied. In CUBIT, the underlying basic algorithm is the *sweeping* one [T. D. Blacker 1997; Eloi Ruiz-Gironés et al. 2009] (see Fig. 2.11), while ICEM-CFD relies on interactive tools to split the geometrical domain into hexahedral blocks (see Figure 2.10). The obtained mesh is then heavily controlled by the user, but the time one has to dedicate to this task can be prohibitive. The expertise level of the user is thus preponderant in order to get a good resulting mesh. Moreover, these semi-automatic approaches do not take the physical properties of the simulation into account, and can not be extended to be used in an adaptive loop during the simulation itself.

2.2 State of the art in hex meshing

As seen previously, due to the lack of local operators on hexahedral meshes, the only shape that can reliably be meshed using hexahedra is the block, or more precisely the hexahedral shaped block; it

[1] Numerous tetrahedral meshing algorithms are available [Frey and George 2008].

can be meshed with a grid mesh slightly deformed using a transfinite interpolation [Faux and Pratt 1979]. The current meshing process for preparing the input for numerical simulations does just that, meaning that given a geometrical model, an engineer will process it and decompose it into elementary hexahedral blocks that can then be meshed, as seen in Figure 2.9. One should note that this example only shows one piece; when considering a whole assembly of CAD pieces and the fact that blocks need to propagate through the pieces in order to obtain a conformal mesh, it comes as no surprise that this phase can take several weeks to months of engineering time.



(a) (b) (c) (d)

29 blocks, 15 minutes 174 blocks, 2 hours

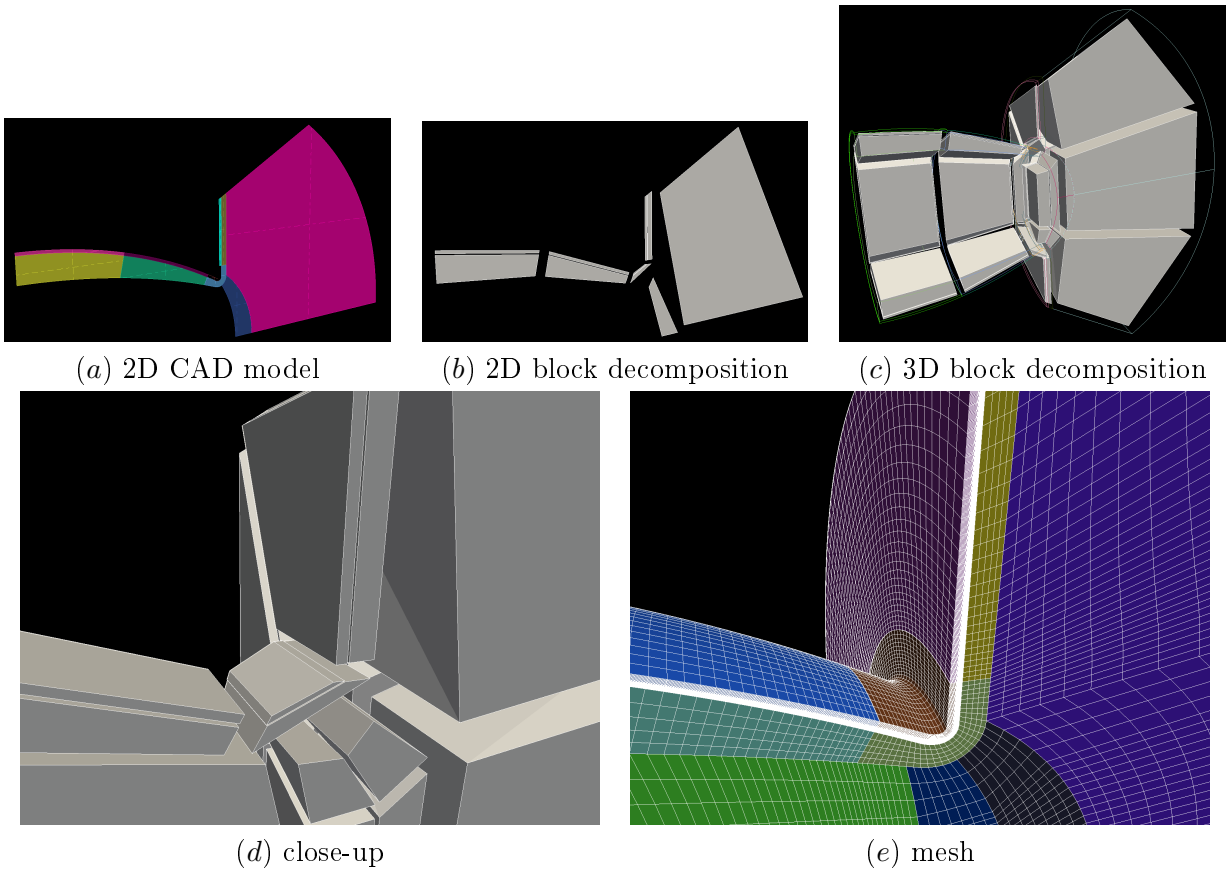
Figure 2.9: Example taken from [Ledoux 2018] where a CAD-model (a) is manually decomposed into blocks (b) with the engineering time taken by an operator familiar with the software; (c and d) the same with a more complex model that includes small details.

In the study [Boggs et al. 2005], the authors recognize that the geometric manipulation step – of which one goal is often to make it easier to mesh – and the meshing step account for a large fraction of the overall process time. As such some developments in industrial CAD-meshing software focus on the ease of use of the software. An example of those capabilities is shown in Figure 2.10 where a 2D CAD model is built and decomposed into quad blocks. The user can then automatically create an axisymmetric 3D CAD and block structure and finally a mesh is obtained that is fit for purpose for a simulation as seen in Figure 1.4 page 9.

Other developments have extended the range of shapes that can be meshed. The sweeping algorithms [P. Knupp 1998] lift the limitation on hexahedral-shaped blocks; instead the requirements are that a source and a target surface of similar topology have to be linked by a mappable surface or surfaces (see Figure 2.11). The many-to-one and the many-to-many sweeping [Lai et al. 2000] were then introduced to further expand the capabilities of this type of algorithm.

We have seen that, despite many improvements, generating a mesh is still a time-consuming task as it requires a user to manually operate a somewhat complex software. In order to shorten this step quite a few methods were and are still devised so as to automate the process. In the following sections we will attempt to present those methods, thematically grouped but knowing that more than a few of those are not really restricted to one group and could be part of several families. The following surveys and courses [Sarrate et al. 2014; Owen 2005; Ledoux 2014] have guided our depiction of those methods. The families we consider are depicted in Figure 2.12 and are the following ones:

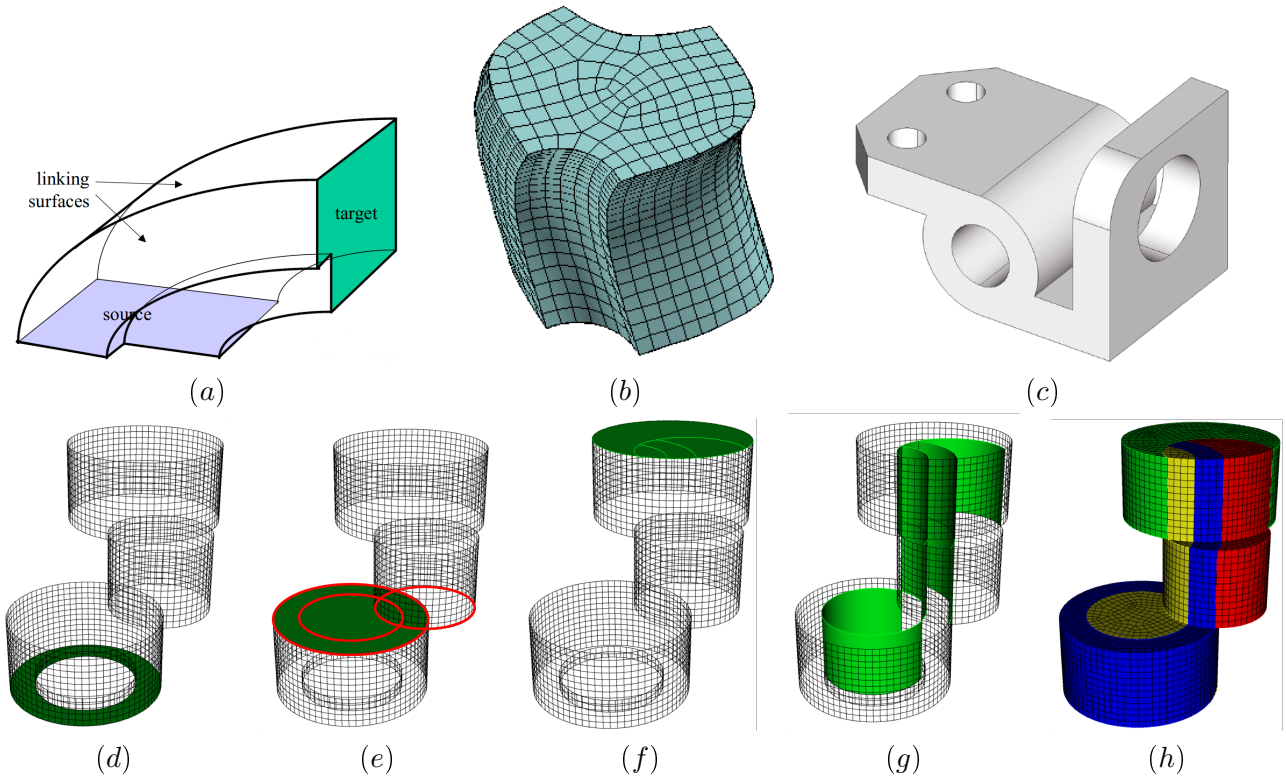
- **Geometry-first.** In order to discretize a geometric domain Ω, those algorithms rely on the geometrical properties of Ω. Starting from the boundary $\partial\Omega$ of Ω, they will for instance try to discretize Ω by iteratively inserting cells, or extracting geometric pieces of information, like a medial object, to help build the mesh.
- **Mesh-first.** In this case, a mesh is first built then adapted to Ω. The underlying intuitive idea is that it is easier to modify a mesh than to create it from scratch. But it is often more complicated to ensure the right cell quality near $\partial\Omega$.
- **Cartesian Idealization.** Those methods are kind of a compromise where we are going to "deform" the geometrical model Ω to get a model that is easier to mesh with quadrilateral (in 2D) or hexahedral (in 3D) cells.



(a) 2D CAD model (b) 2D block decomposition (c) 3D block decomposition

(d) close-up (e) mesh

Figure 2.10: Example of a laser target from [Ledoux 2018] similar to what can be used to run simulations and prepare studies at experimental facilities [CEA 2020].



(a) (b) (c)

(d) (e) (f) (g) (h)

Figure 2.11: Sweeping examples from [Owen 2005]. (c) is not sweepable without some additional manipulations because of its multiple directions.

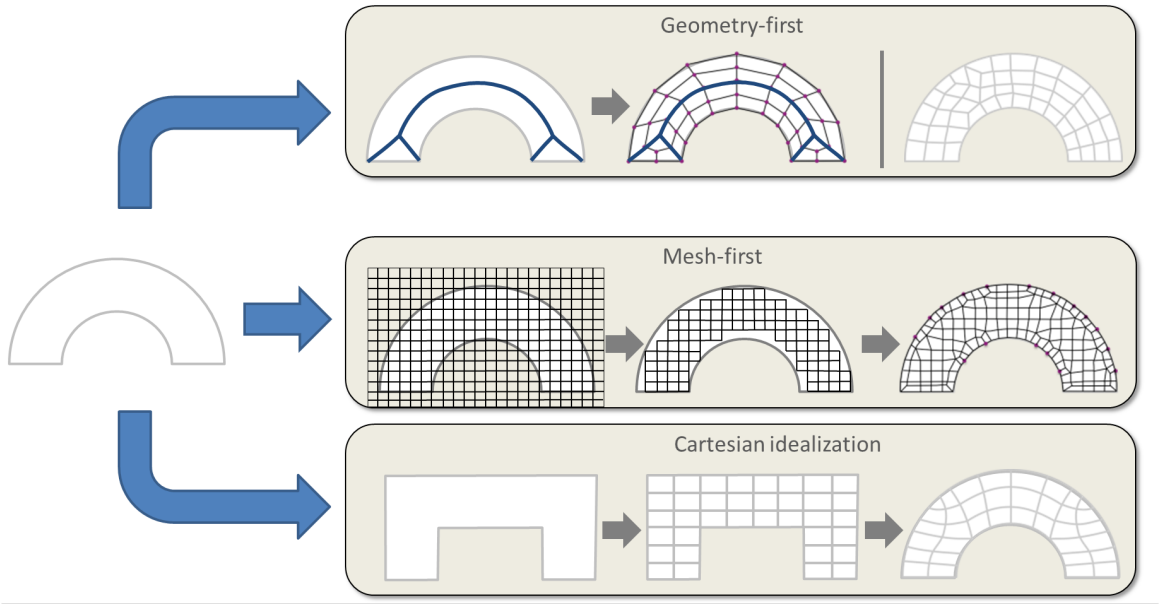


Figure 2.12: Three major types of hexahedral meshing algorithms.

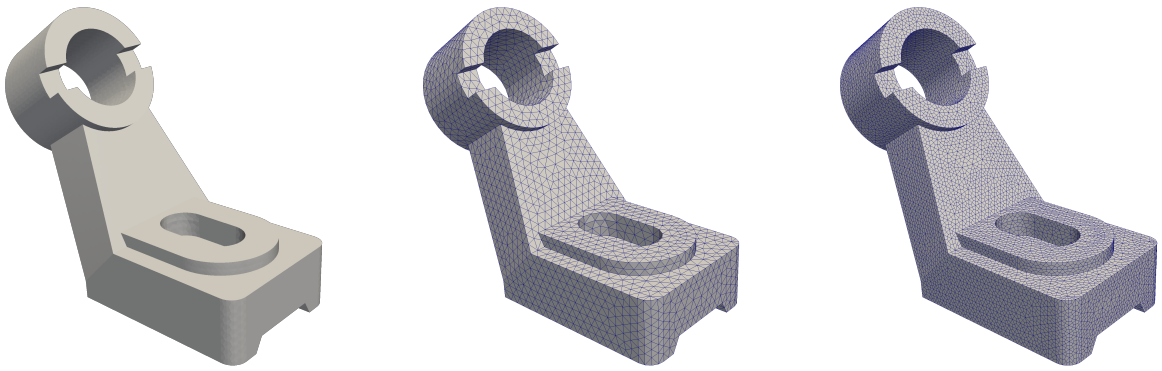
2.2.1 Geometry-first

The approaches that consider the geometry first are the most numerous. They are either some attempts (with or without success) to extend existing techniques or driven by natural ideas and decompositions that are used to split geometric domains in interactive commercial tools.

2.2.1.1 Automatic full domain hex-meshing

Indirect methods. The first methods we consider are based on tetrahedral meshing. We have seen in Section 2.1.4 that it is possible to reliably generate a tetrahedral mesh, even on a complicated domain, and we group in this family the methods that directly make use of this tetrahedral mesh. Some other methods also make use of a tetrahedral mesh as a first stage, not directly but to help with computations or to localize nodes in the domain.

- **Tethex.** The idea behind this method is to first generate a tetrahedral mesh and then to split every tetrahedron into 4 hexahedra (triangles are split into 3 quads in 2D) as seen in Figure 2.13.



(a) (b) (c)

Figure 2.13: Example from [Ledoux 2018] of the tethex method applied on a CAD model (a) that is first meshed using tetrahedra (b), which are each split in order to obtain an hexahedral mesh (c).

- **Recombination.** The recombination methods are in a sense the opposite of the tethex: starting from an initial tetrahedral mesh, these methods assemble groups of tetrahedra in order to form a single hexahedron. There are two main drawbacks to these methods: first, they do not produce full hexahedral meshes, but hex-dominant ones instead. A lot of patches of tetrahedra

remain that could not be grouped together into hexahedra. Secondly, it is not possible to control where in the domain those remaining tetrahedra are located; in [Baudouin et al. 2014], the authors attempt to mitigate this by prioritizing the creation of hexahedra along the boundaries. In [Pellerin et al. 2018] the authors extend the recombination so as to consider previously missed possible groupings by not relying on predefined templates, then build a graph where the vertices are the possible recombinations and are deemed neighbors when they conflict with one another, typically when they share a tetrahedron. The recombinations that will effectively be done are selected by extracting an independent set from the graph. Even so, the method's output mesh is hex-dominant with a number of non-hexahedron cells of the same order of magnitude as the number of hexahedra.

Front propagation. As its name suggests, this family gathers the methods where the mesh generation starts from the boundary $\partial\Omega$ and propagates to the interior of the domain. They usually require that the boundary of the domain be already meshed, and while it is in itself a limitation it is also a selling point as these are the only methods capable of meshing a volume when its delimiting surfaces are meshed or partially meshed. Those methods are mainly designed as extension of 2D algorithms and they are driven by geometry or topology first.

In the paving algorithm [T. D. Blacker and Stephenson 1991] and its extension to 3D [T. D. Blacker and Meyers 1993], new quadrilaterals and hexahedra are added incrementally starting from an already meshed boundary until there are only some inner cavities left. Decisions to add a cell are driven by geometric considerations. The filling of the remaining cavities and the joining of fronts prove to be where the method encounters robustness issues. In Q-Morph [Owen, M. L. Staten, Canann, et al. 1999] and H-Morph [Owen and Saigal 2000] the authors follow the same principle but start from a first triangular or tetrahedral mesh and select the nodes of the original mesh that will serve as vertices of the newly built cell; this is a way to both facilitate geometric computations and to prescribe a target mesh size. The original mesh is then updated in order to insert the new edges and faces that were not present initially (see Figure 2.14).

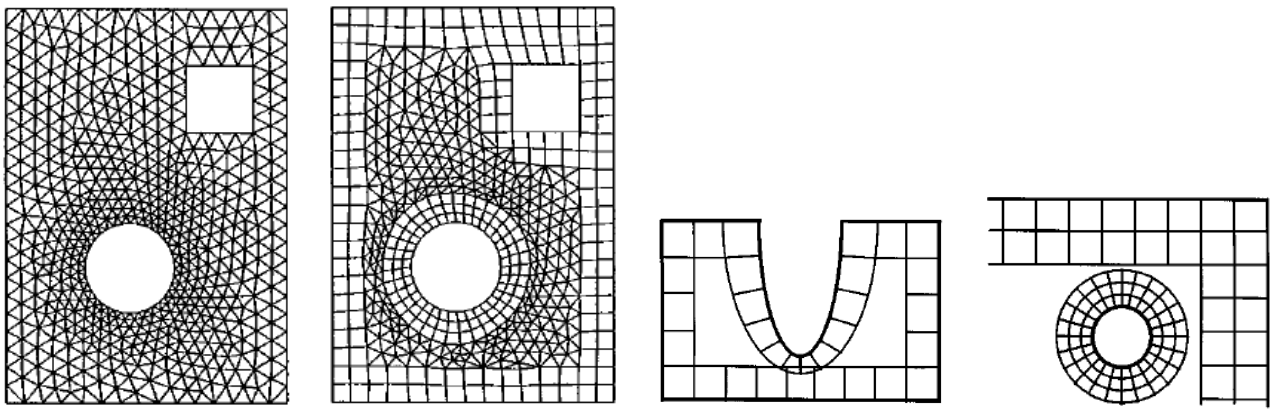


Figure 2.14: The Q-Morph algorithm with the initial triangular mesh (a) and the mesh after a few iterations in (b). Subfigures (c) and (d) show the problem of front joining. Examples from [Owen, M. L. Staten, Canann, et al. 1999].

Some other methods, classified as topology-driven, also propagate from the boundary but choose to avoid geometric computational issues and the handling of lots of special cases by making use of the structured nature of hexahedral meshes that we talked about in Section 2.1.4. Instead of directly adding cells to the mesh, they attempt to build the dual of an hexahedral mesh by inserting whole sheets – which is akin to inserting a whole layer of hexahedra at a time – inside the domain. Working on the topology of the mesh in place of its geometry is the basic principle of the Reliable Whisker Weaving algorithm [Folwell and Mitchell 1999]. The authors of [Ledoux and Jean-Christophe Weill 2007] introduce back some measure of geometry in the decision process on the premise that not doing so tends to produce meshes of too low quality.

Block decomposition. In an attempt to mimic the results of handmade meshes as seen at the beginning of this section, and considering that having a block-structure is usually a sought after property for the mesh, several methods aim to split the domain into simple meshable blocks.

- **Medial axis.** Medial axis methods [Tam and Armstrong 1991] consist in retrieving the set of points that are equidistant from at least two points located on the boundary of the domain $\partial\Omega$; these points form a skeleton that subdivides the domain into simple blocks as seen in Figure 2.15. The problems with these methods reside in first the difficulty of building said skeleton, secondly in the fact that the blocks that are obtained are not all quadrilateral or hexahedral and thus cannot be meshed using a grid without an additional treatment and thirdly this decomposition is not optimal in terms of mesh quality. These drawbacks could seem prohibitive, but the medial object has the main benefit of being mathematically well-defined and even if it does not provide the expected block decomposition, it could be very helpful to help build final blocks. Some recent works as [Quadros 2014; Papadimitrakis et al. 2019] show how it could be used in practice for getting blocks.

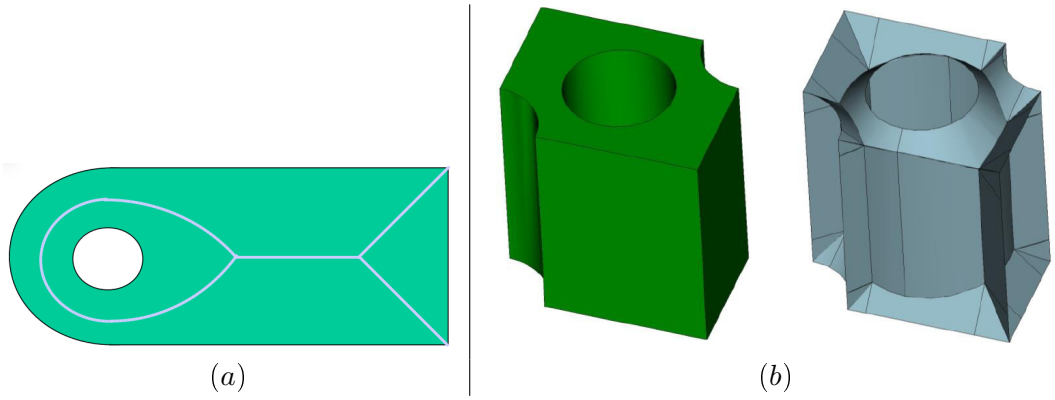


Figure 2.15: Examples of a medial axis decomposition in 2D in (a) and a 3D model with its medial surface in (b), from [Owen 2005; Quadros 2014]

- **Frame fields.** For about a decade, different works have tried and extended 2D approaches based on cross-fields to 3D. The idea is to build a field of frames, which are a "simple" 3D basis made of three orthogonal directions. Frame fields are built to be aligned along $\partial\Omega$ and be as smooth as possible. Such fields bring an extra information to building hexahedral meshes : in every location of Ω, hexahedral cells should be aligned along the three axis of the local frame. Following this approach, several works like [Kowalski et al. 2016] produce the expected block-decomposition of Ω. In many simple cases, as the one shown on Figure 2.16, a frame field is computed on a tetrahedral mesh (a and b) then a topological skeleton is extracted (c), which is necessary to compute a meshable block decomposition (d and e). The current limitation of those approaches is to generalize to more complex domains where the topological skeleton of the frame field does not correspond to a valid block-structured mesh. Some current works [Palmer et al. 2020] focus on this limitation to try and go further.

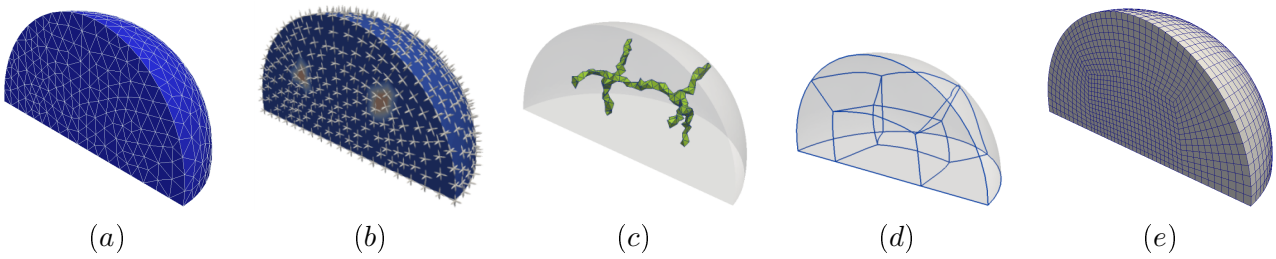
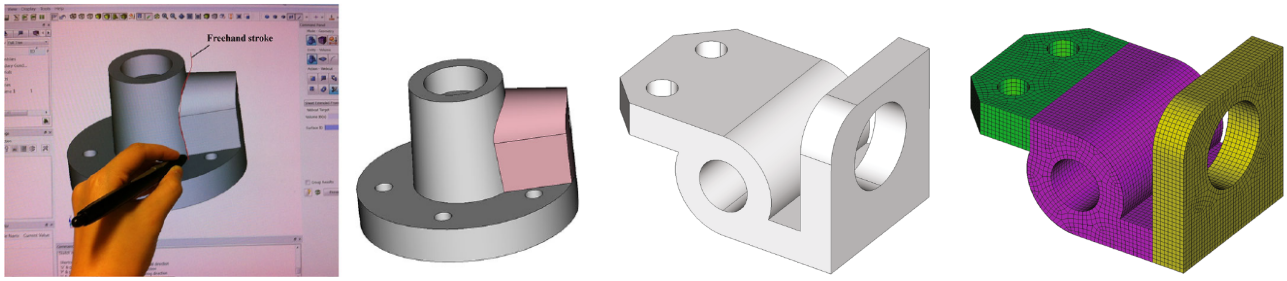


Figure 2.16: 3D block-decomposition pipeline using frame fields. Starting from the shape meshed using tets (a) frames are computed at each node (b). Interior singularity lines are identified (c) and are completed on the boundary to form an hexahedral block-decomposition (d), and finally each block can be meshed using a grid (e). Example from [Kowalski et al. 2016].

- **User-guided hex-meshing.** As previously mentioned, all the automatic hexahedral meshing methods have drawbacks and the current workflows for studies include a manual meshing phase that can take several months of engineering time. As such, some of the research focuses on mixed solutions where the algorithms will no longer be fully automatic but will instead assist the engineer; they could also be seen as semi-automatic methods where the engineer will be prompted to unstuck the program.

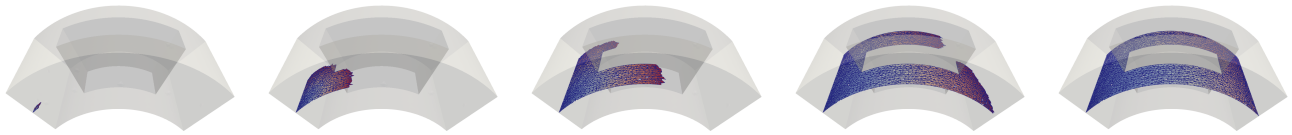
  In [Lu et al. 2014], the authors propose a sketch-based user interface (see Figure 2.17) in order to decompose a model into sweepable blocks. This can be seen as a complement to the automatic sweepable volume decomposition [David R. White et al. 2004; Wu and S. Gao 2014] where the methods are not only run to completion but a feedback is extracted and given to the user, who can then improve upon the proposed solution or provide its input to make them work in the case where they fail.



(a) (b) (c)

Figure 2.17: A sketch-based user interface is shown in (a) with the corresponding decomposition of the domain into sub-volumes. In (b) the CAD model that is decomposed into three sub-volumes (c), each with a different sweeping direction. Example from [Lu et al. 2014].

Similarly, considering that generating a block-decomposition automatically based on frame fields is nowhere near close at hand, especially in 3D, in [Takayama 2019] the authors implemented a frame fields-based method where the user designs boundary loops from which dual sheets are built, thus building the dual of the desired block-decomposition of a domain. In [Calderan et al. 2019], the authors chose to skip this intermediary step and provide an interactive environment where the user can pick points on $\partial\Omega$ from which dual surfaces will spawn directly (see Figure 2.18) and then build a block structure (see Figure 2.19).



(a) (b) (c) (d) (e)

Figure 2.18: The user picks the location of the seed (a) from which a dual sheet will grow (e). Example from [Calderan et al. 2019].

2.2.2 Cartesian Idealization

The methods presented in this section are based upon the idea that since the only shape that can reliably be automatically meshed is the block, it might be advantageous to deform the original shape to a block-like one, which can then be meshed, and transform back the mesh onto the original shape. Two such families of methods are discussed here: the submapping and the polycubes.

Submapping. This method meshes a CAD model by first transposing it as an axis-aligned computational domain, which is then discretized. Whiteley et al. [Whiteley et al. 1996] extended the purely

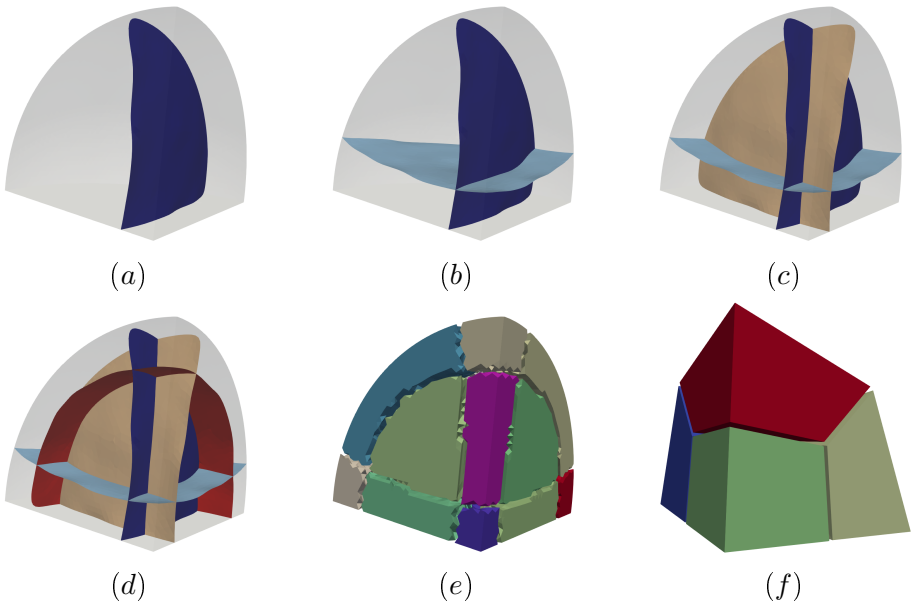


Figure 2.19: In [Calderan et al. 2019], the user interactively picks several dual surfaces (a to d) to build the dual of an hexahedral block structure. This set of surfaces is converted into a set of dual zones (e) which is finally derived into the final block structure (f).

manual user input-based method [David Roger White 1996] by introducing an automated interval assignment solved using integer programming. Ruiz-Gironés et al. [E. Ruiz-Gironés and Sarrate 2010b; E. Ruiz-Gironés and Sarrate 2010a] added the automatic classification of vertices and relaxed the requirement that all angles should approximate multiples of $\frac{\pi}{2}$ (see Figure 2.20). Their extension also handles multiply connected domains, by considering them as simply connected with additional virtual curves traversed twice. Tautges et al. [Cai and Tautges 2015] introduced the manual addition of virtual vertices so as to prevent bad output meshes when encountering a few classic configurations in the CAD.

Submapping methods suffer from a lack of robustness, in particular in 3D as the vertices classification is done locally to each surface, which can result in conflicts during the interval assignment step, i.e. the phase when the actual discretization is determined. Moreover the whole interval assignment process is only based on the boundary classification and it loses some in-volume information.

Polycubes. Unlike the submapping algorithm, the polycube-based method relies on a whole volume deformation process, which keeps in-volume information available and pertinent. The main works on polycube [He et al. 2009; Gregson et al. 2011; Yu et al. 2014] build a polycube, i.e. an axis-aligned polyhedra, and define a transformation between said polycube and its original shape (see Figure 2.21). In some cases, the polycube is adjusted in order to meet some requirements, such as preserving the topology of the CAD.

Polycubes are sensitive to the model orientation: having angles of typically 45 degrees between its curves or surfaces will make for a lot of badly shaped cells on the boundaries. Moreover, like submapping, there exists a lot of classic models that cannot be meshed with a polycube and such approaches do not insert singularities in the domain, potentially leading to distorted elements along some boundary surfaces of Ω.

2.2.3 Mesh-first or overlay-grid methods

Mesh-first methods consider a starting mesh and adapt it so that it meshes the CAD input. The starting mesh should be easy to generate, typically an axis-aligned grid. Starting from the pioneer work of R. Schneiders [R. Schneiders 1996], several authors proposed different solutions to build such type of meshes considering volume fraction inputs like [Owen and Shepherd 2009; Owen, M. L. Staten, and Sorensen 2012; Owen, Brown, et al. 2017; Owen and Shelton 2015], CAD models where sharp features must be preserved first [Maréchal 2009; Maréchal 2016; X. Gao, Shen, et al. 2019] or smooth surfaces for medical applications [Zhang 2016]. In all those works, many improvements are

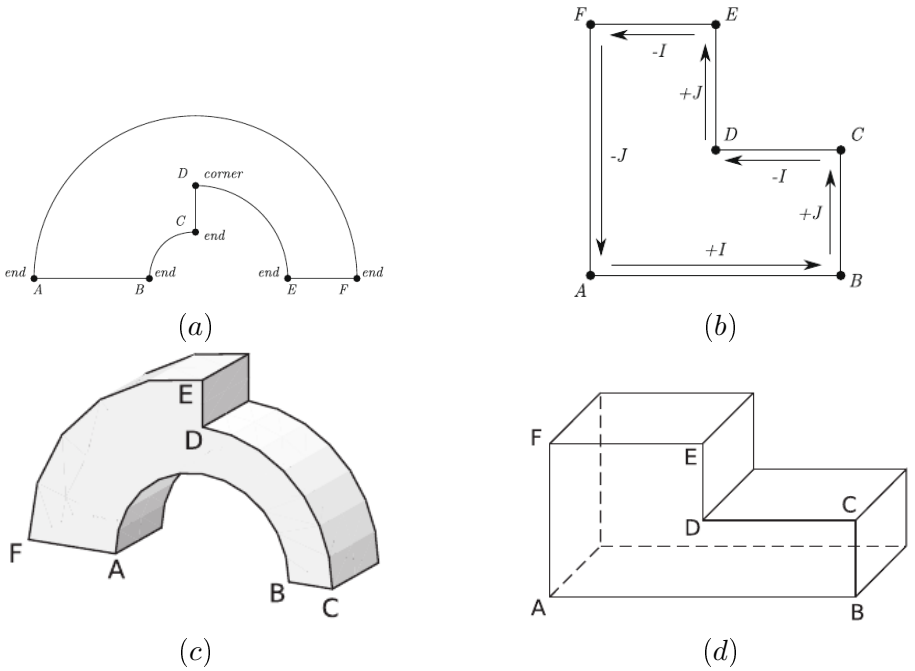


Figure 2.20: Example of the submapping method from [E. Ruiz-Gironés and Sarrate 2010b; E. Ruiz-Gironés and Sarrate 2010a]. In (a) the 2D geometrical model where the vertices are classified which allows for the shape to be transformed into the computational domain as (b). (c and d) a 3D example.

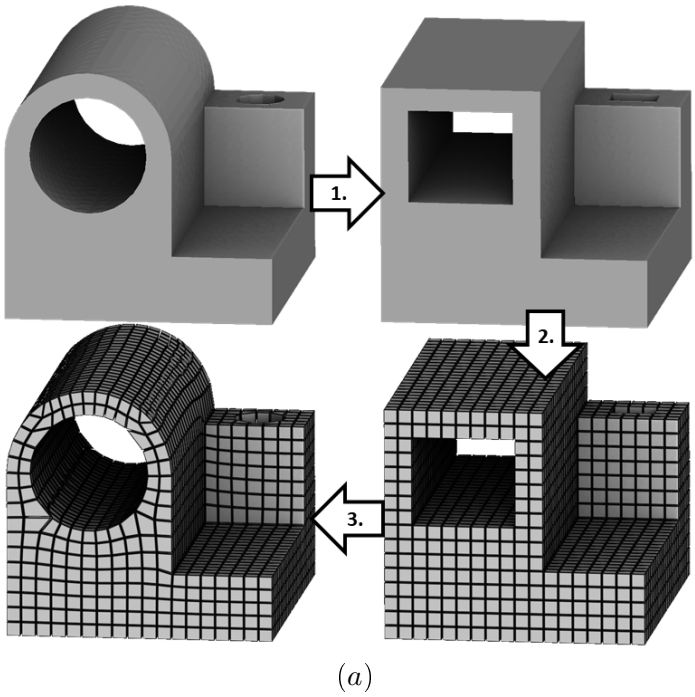


Figure 2.21: Example from [Ledoux 2018] of a polycube-based approach. First the domain is "intuitively" deformed to get all of its boundary faces X, Y or Z-axis aligned; this deformed domain is then easily meshed and finally the mesh is projected back onto the original shape.

done to handle sharp features, small features of the domain Ω that require to refine the input mesh, and multi-domain input.

In practice, mesh-first methods are the most robust hexahedral meshing algorithms but they do not provide expected features for many applications: worst cell quality along the boundary $\partial\Omega$, undesirable patterns encountered inside the domain or on the boundary, loosely CAD capture for some methods, non-alignment along the boundary and inability to get block structure.

2.3 SCULPT

We present here in detail the SCULPT algorithm, introduced in [M. L. Staten and Owen 2010; Owen, M. L. Staten, and Sorensen 2012] and available in the CUBIT software [Cubit 2019]. It takes a mesh carrying volume fraction data as an input, assign a material to each cell and modify the mesh with two goals in mind: first to match the input volume fractions (geometric modification by moving the nodes), and secondly to produce a mesh with good quality cells (moving the nodes during a smoothing phase and topological modifications by inserting layers of hexahedra to try to obtain better quality cells). Contrary to some of the overlay grid methods presented previously that create a mesh and manipulate its dual, it directly works on the primal mesh, which we can see having three main advantages:

- as the input volume fractions are carried by the cells, manipulating those directly avoids the need to compute volume fractions at the nodes using interpolation techniques that may or may not offer guarantees in terms of volume preservation;
- the input mesh can be huge, especially when SCULPT is used in an intercode capacity after an Eulerian simulation, and might already be distributed on several partitions; directly handling it avoids the need to create a distributed dual mesh and having to partition the dual cells located across partition boundaries;
- more simply, while the dual of a grid mesh is another full-hexahedral mesh – it is in fact another grid – that is not true when the mesh is unstructured; not handling unstructured meshes would limit the software functionalities.

The reasons stated above make this method a good fit for our objectives; the algorithm is illustrated in more detail in Figure 2.22 and is comprised of the following steps:

a) The Eulerian mesh, that is to say a mesh that carries volume fraction data, can be given as an input or it could be manufactured using others means, such as imprinting a CAD model onto a grid mesh; the authors introduced a refinement procedure to better represent the geometric details in such a case [Owen, Shih, et al. 2017]. But whatever its origins, it is treated as an input on which we do not have a say;

b) The cells of the mesh are assigned to materials on a majority basis: a cell is assigned to the material of highest volume fraction inside the cell;

c) The assignment correction step is by nature optional; the initial assignment obtained in (b) can conflict with the simulation code requirements or with assumptions that the mesh generation implementation itself made, whether because it is a methodology requirement or because it was easier to code. In the SCULPT case, this phase ensures that the assigned materials each form an assembly of disconnected manifolds (necessary when for example one needs to compute the normal vector at every node of its interface; it is undefined in Figure 2.23) by reassigning the cells to different materials where necessary. When changing materials, cells are reassigned around each problematic nodes to the second best (in the sense that it is closest to the volume fractions) correct assignment and as it could lead to non-manifold configurations appearing in the neighborhood of changed nodes this phase is executed again; the authors slightly modify the volume fractions (this adjustment is only considered for this phase, the original volume fractions are used for the rest of the method) when changing the assignment of a cell so as to avoid an infinite loop scenario (see Figure 2.24);

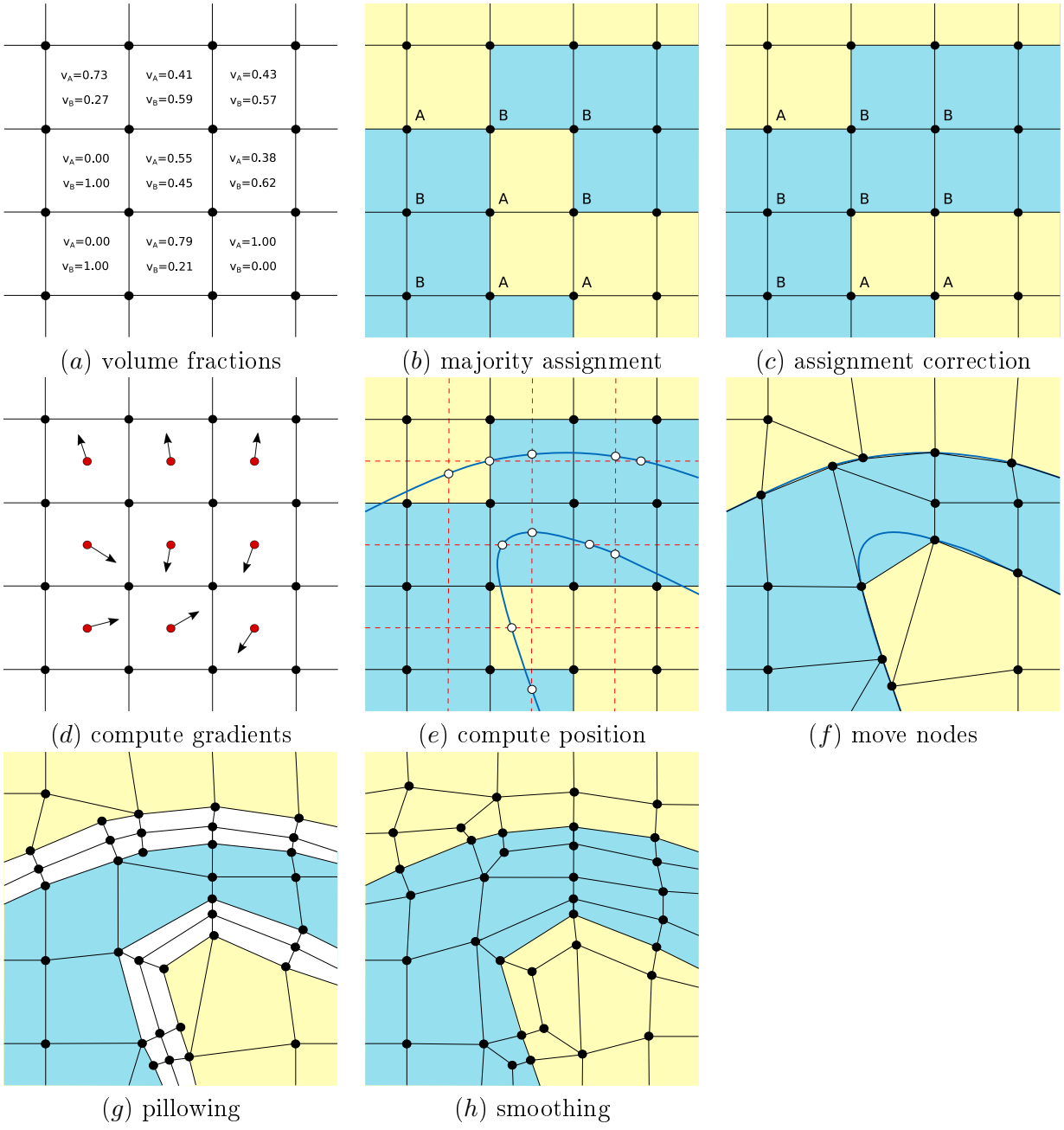


Figure 2.22: Starting from the volume fractions given in (a), grid cells are assigned to each material in two stages (b and c) before computing a new position (d and e) for the interface nodes and then moving those nodes (f); as the mesh quality can end up degraded, a pillowing (g) is applied in order to provide the smoother (h) more degrees of freedom to try and improve the quality.

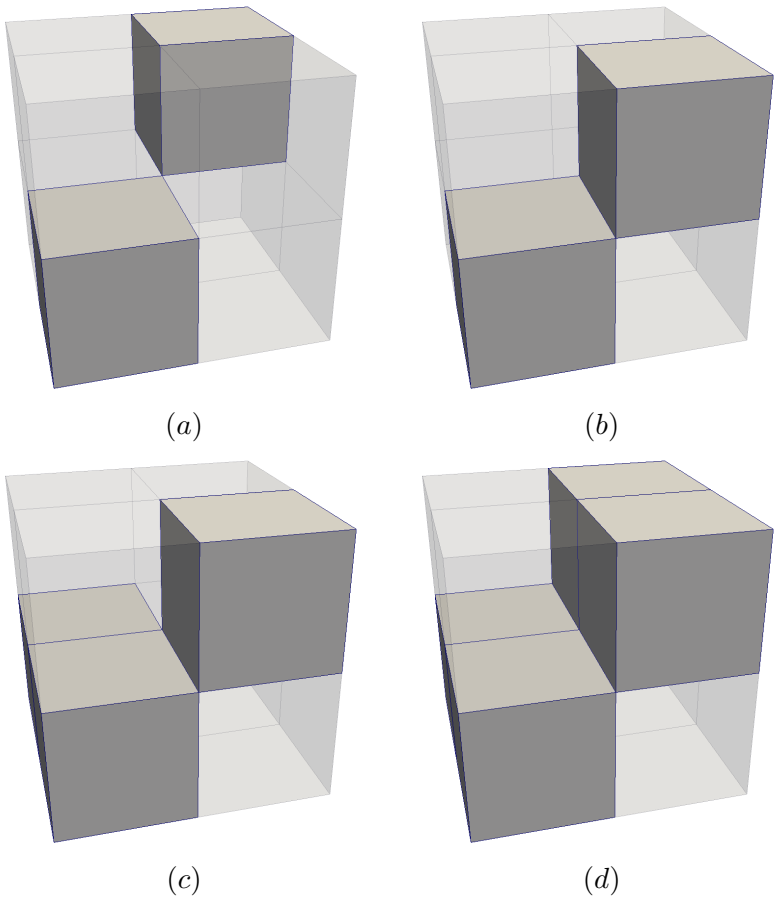


Figure 2.23: Example of non-manifold configurations for a material (non-transparent cells) around the central node.

d) A gradient is computed for every material. It is not an end in itself, it is simply a means to an end as it will be used in the computation of the position of the interface nodes;

e) A target position is computed for the interface nodes;

f) The nodes are moved to their expected new positions; up to this moment the mesh was never modified, staying strictly the same as the input. We can see in Figure 2.22.f that several of the quads are now degenerated, nearly triangle-shaped or even non-convex;

g) A pillow is applied on each side of the interfaces; it is the only step where the mesh is topologically changed. There are still the same degenerated cells in Figure 2.22.g as in the previous figure; this operation's goal is not to directly improve the mesh quality, it will instead provide more degrees of freedom or leeway for the smoother to works with;

h) The mesh is then smoothed and Figure 2.22.h illustrates that in this example it indeed managed to drastically improve the mesh quality.

As previously mentioned, the resulting mesh neither preserve the volume fractions[2] nor does it capture the geometric features when the starting input is a CAD model. Nonetheless, the authors have shown [Owen and Shelton 2015] that in one linear and in one non-linear elastic-plastic cases, running using meshes produced via Sculpt was not detrimental; they even included an orientation-sensitivity study, as it has a huge impact on several meshing algorithms (see Section 2.2.2) and particularly in overlay-grid methods (see Section 2.2.3).

[2] In [Owen, Brown, et al. 2017] the authors proposed an addition to step c, the assignment correction, that reassigns the cells in order to avoid small isolated clusters of materials which will tend to produce perturbed interfaces and bad quality elements; they called this option *defeaturing*. This goes to show that, depending on the user, volume fraction preservation can be dropped in favor of mesh usability.

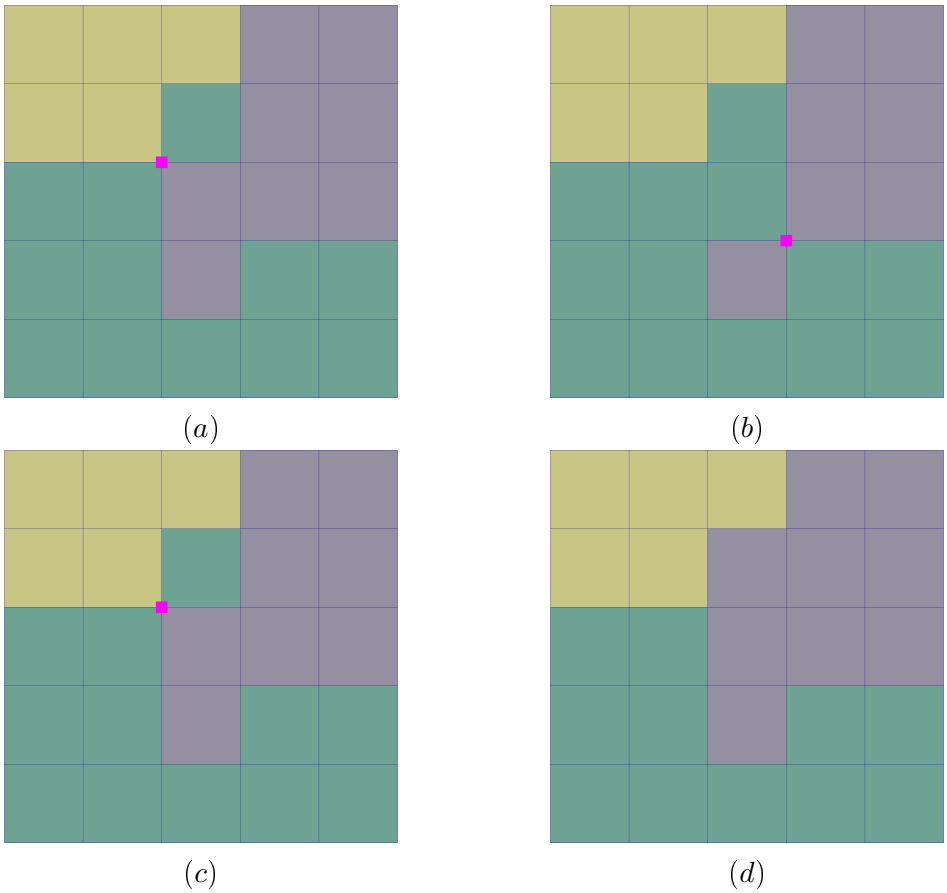


Figure 2.24: Example of non-manifold resolution and live-lock avoidance strategy. (a) the initial material assignment with the non-manifold node highlighted; (b) the new assignment solves the non-manifold at the first node but causes another to appear; (c) the new non-manifold node is solved but we are back to the initial assignment, which was originally the best; (d) the non-manifold is solved again with another assignment because at some point the volume fractions repeated adjustments will lead the algorithm to make another choice, thus avoiding an infinite loop between state (a) and (b).

Chapter digest – *In the course of this chapter we have presented notions and definitions that pertain to our problem. During the state of the art of hexahedral meshing we have seen that there are methods aplenty, but that considering our input – namely a mesh carrying material volume fractions – the number of methods available for us to use as is becomes severely limited. We presented* SCULPT*, which is such a method and intend to study and extend it in the next chapter.*

Chapter 3

SCULPT evaluation and improvement

Data conversion between physics simulation codes requires to address some questions about the preservation of physical quantities. Unlike some computer graphics application where we could focus on frame rate optimization or getting something visually "close to reality" or "pleasing to the eye", we need here to preserve, or at least control, some relevant quantities. In the case of converting an Eulerian mesh M^I to a Lagrangian mesh M^O, we intend to control that the volume of every material of M^I is the same in M^O.

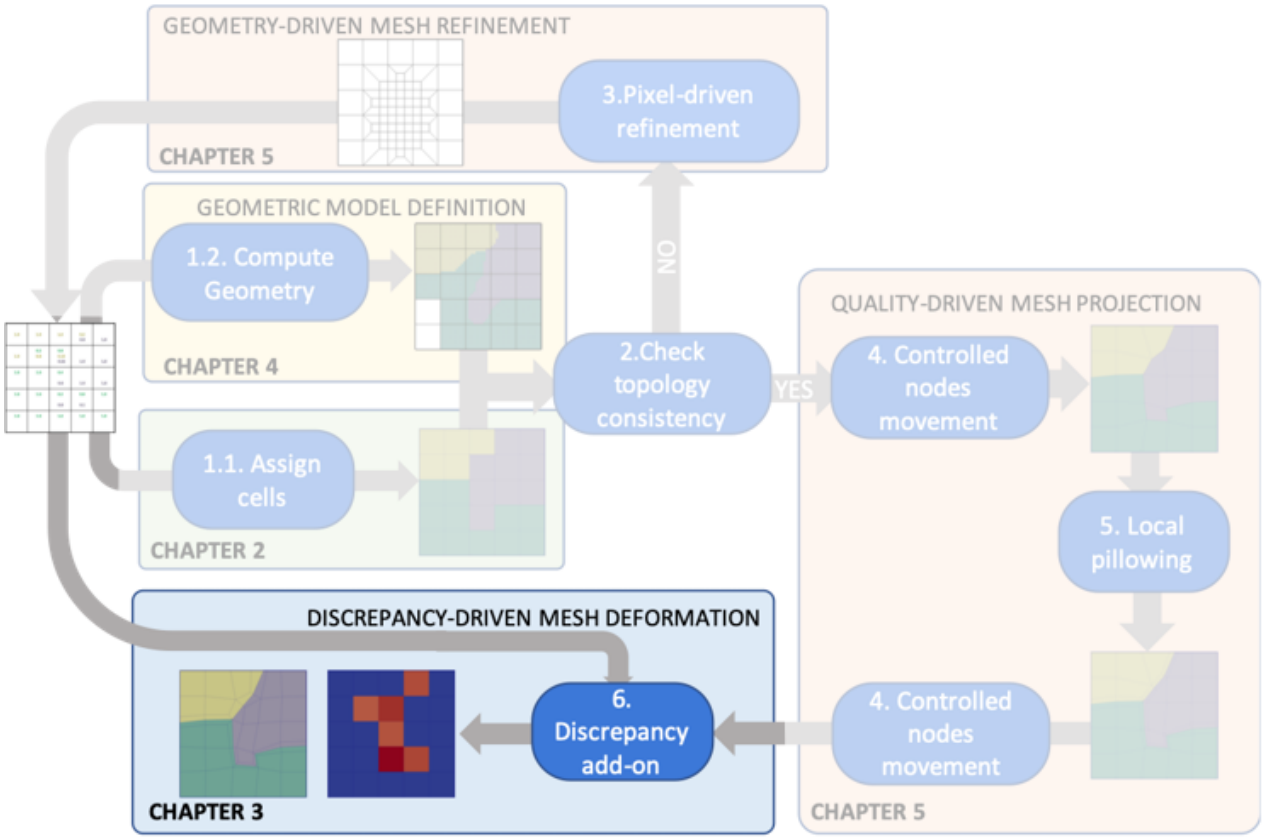


Figure 3.1: The discrepancy evaluation is performed as a post-processing stage in the ELG pipeline leading to a final smoothing stage that is driven by fitting initial data location - the volume fractions in the initial mesh - as best as possible.

Such a consideration is not directly taken into account in overlay-grid meshing algorithms. For instance, the volume fractions are not preserved by the SCULPT algorithm (see Section 2.3). As it is mandatory for our applications, we propose in this chapter to introduce a *discrepancy* criterion to evaluate the difference between the output of this software and its input. This notion is introduced in Section 3.1. It gives us locality information, on the meaning that locally to each input cell of M^I we know how well each material is preserved in terms of volume by the cells of M^O. With such an information, we have a feedback on whether cells of M^O are too large or too small (meaning

that we locally have too much or too little of a material). It allows us to introduce in Section 3.2 a discrepancy-controlled loop so as to reduce it by moving the interface nodes while ensuring that the cell quality does not drop below a user-defined threshold. This procedure has been introduced as a post-processing stage for evaluating the original SCULPT algorithm and to improve our ELG pipeline at the end (see Figure 3.1 for a reminder of the ELG pipeline). It was run on a variety of meshes built from STL models, simple and complex CAD assemblies and volume fraction grids, including outputs of simulation codes; some of those experiments are presented in Section 3.3.

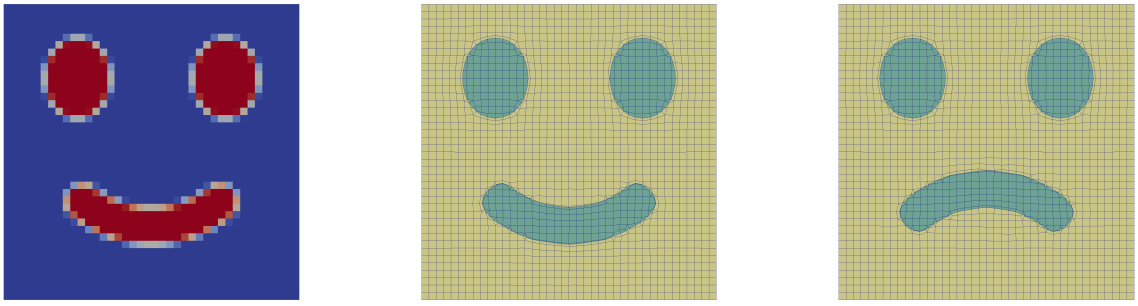
3.1 Discrepancy definition and evaluation

Our aim is to build an output Lagrangian mesh M^O, made of pure cells, from an input Eulerian mesh M^I, made of pure and mixed cells, while preserving as best as possible the volume and locality of each material during this process. In other words, the volume of a material m should be the same in M^O and M^I and it should be at the same geometric location. It leads us to the definition of global material volume differences.

Definition 3 (**Global material volume difference**). *Let A and B be two meshes of the same geometric domain Ω and $\mathcal{M}$ be the set of materials that disjointly fills Ω, the global difference of material volumes between A and B is*

$$\Delta V = \sum_{m \in \mathcal{M}} |\Delta V_m| = \sum_{m \in \mathcal{M}} |V_m^A - V_m^B| \tag{3.1.1}$$

where V_m^A and V_m^B are respectively the volume of a material m in meshes A and B.



(a) (b) (c)

Figure 3.2: Given a grid mesh with the green material volume fractions represented in (a) ranging from 0 in blue to 1 in red, we can see that despite being of the same total volume for each material, the material mesh (b) fits better the volume fraction grid than (c).

Minimizing ΔV as defined in Definition 3 only ensures a global volume preservation, which proves not to be sufficient as it can lead to unexpected results as seen in Figure 3.2. This shows that we need to introduce locality in our comparison criterion. In order to introduce it, we propose to project back each cell of M^O onto M^I to compute a *discrepancy* value localized in each cell of M^I, as illustrated in Figure 3.3. On this example, a Lagrangian mesh made of blue and yellow quadrilateral cells is projected onto an Eulerian mesh shown in dashed red lines. Let us focus on cell $c_j^I \in M^I$, which contains 2 materials A and B with respectively the volume values[1] $f_{j,A}V(c_j^I)$ and $f_{j,B}V(c_j^I)$. In order to compute the discrepancy of a material m in cell c_j^I, we look for the pure cells of M^O containing m (noted $M^O_{|m}$) that intersect c_j^I.

Definition 4 (**Local discrepancy**). *Let M^O and M^I be two meshes of the same geometric domain Ω and $\mathcal{M}$ be the set of materials that disjointly fills Ω. Let c_j^I be a cell of M^I and m be a material of $\mathcal{M}$, we note $d_{j,m}$ the discrepancy of c_j^I relatively to material m and mesh M^O and we define it as*

$$d_{j,m} = d(c_j^I, m) = V(c_j^I \cap M^O_{|m}) - f_{j,m}V(c_j^I) \tag{3.1.2}$$

[1] Computed as the product between the geometric volume of the cell c_j and the volume fraction $f_{j,m} = f(j,m)$ of the material m in c_j.

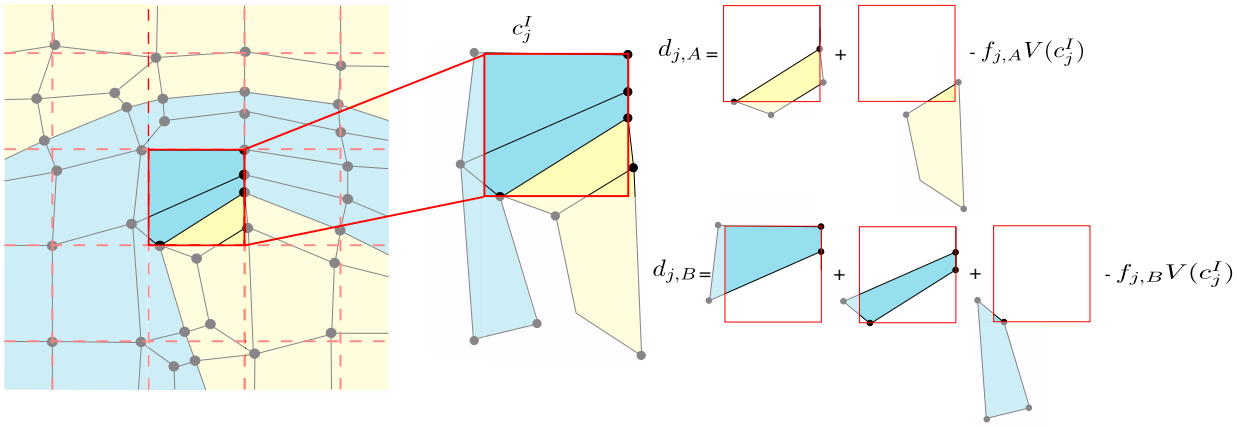


Figure 3.3: Considering one input cell c_j^I of an Eulerian grid, we graphically illustrate the discrepancy computation for materials A and B locally to the cell c_j^I.

where $V(X)$ is the volume of any geometric space X, $M^O_{|m}$ is the output mesh restricted to the pure cells of material m and $c_j^I \cap M^O_{|m}$ is the geometric intersection of c_j^I with the cells of $M^O_{|m}$.

Let us note that in practice, we compute geometric intersections using[2] [Herring et al. 2017]. Looking at the definition of the local discrepancy, it is interesting to note that:

- $d_{j,m} > 0$ means that locally to c_j^I, we have too much of material m;
- $d_{j,m} < 0$ means that locally to c_j^I, we do not have enough of material m.

In order to compare the whole meshes M^I and M^O, we finally get the following definition.

Definition 5 (**Global discrepancy**). *Let M^O and M^I be two meshes of the same geometric domain Ω and $\mathcal{M}$ be the set of materials that disjointly fills Ω. The total discrepancy of a cell $c_j^I \in M^I$ is defined as*

$$d_j = d(c_j^I) = \sum_{m \in \mathcal{M}} |d_{j,m}|, \tag{3.1.3}$$

and the global discrepancy on M^I is defined as

$$d = d_{tot} = \sum_{c_j^I \in M^I} d_j. \tag{3.1.4}$$

Incidentally, for every cell $c_j^I \in M^I$, we have the following inequality

$$d_j \leq 2V(c_j^I), \tag{3.1.5}$$

which is obtained by replacing $d_{j,m}$ in Equation 3.1.3 by its expression from Equation 3.1.2 and applying Minkowski's inequality:

$$d_j = \sum_{m \in \mathcal{M}} |V(c_j^I \cap M^O_{|m}) - f_{j,m} V(c_j^I)| \leq \sum_{m \in \mathcal{M}} |V(c_j^I \cap M^O_{|m})| + \sum_{m \in \mathcal{M}} |f_{j,m} V(c_j^I)| \leq 2V(c_j^I). \tag{3.1.6}$$

3.2 Discrepancy improvement

The global discrepancy allows us to check how a material is preserved between an input mesh M^I and an output mesh M^O. Considering that the global discrepancy is never equal to zero, the intent of the volume-preserving adaptation process is to geometrically modify M^O to best fit the volume of each material defined in M^I. The proposed procedure is an iterative method where each iteration aims to reduce the global discrepancy (see Definition 5) on M^I. The following points outline the proposed structure of the algorithm, with line numbers refering to the specific lines enumerated in Algorithm (1):

[2] The interested reader can directly use the open-source library *portage* [Herring et al. 2017] based on *R3D* [Powell and Abel 2015].

1. At each iteration, we first improve the quality of M^O by moving some inner nodes (line 3), which are nodes adjacent to cells of the same material only. In practice we use the smart Laplacian algorithm [Freitag 1997] or the GETMe agorithm [Vartziotis and Wipper 2012] to improve the mesh quality.

2. Global discrepancy is computed (line 4) and stored to check potential regression during the incoming stage (lines 14 to 17). A regression occurs when the discrepancy does not decrease during two successive iterations.

3. We evaluate the expected target volume for each cell $c_i^O \in M^O$ individually (lines 5 to 7). This process requires to compute the geometric intersection of c_i^O with cells of M^I, and is described in Section 3.2.1.

4. Nodes of M^O are then moved according to the algorithm 2 at page 44 described in Section 3.2.2 (line 9). If some movements induce poor mesh quality (computed at line 10), they are withdrawn (lines 11 to 13 for the node update and line 8 for the previous node location storage).

5. Each iteration stage ends with computing the global discrepancy in order to measure potential regression (line 14). If a regression occurs, it implies canceling the last stage (lines 15 to 17).

Let us now describe the main steps of this algorithm in the following subsections.

Algorithm 1: Global structure of the volume-preservation adaptation algorithm.

Data: M^I, M^O, maxIter, ϵ
Result: Geometrically modified M^O

```
1  iter ← 0; regression ← false;
2  while iter < maxIter & regression == false do
3  |  smoothInside(M^O);
4  |  d_IN ←computeDiscrepancy();
5  |  for c_j^O ∈ M^O do
6  |  |  T_V[c_j^O] ←computeTargetVolumes(c_j^O, M^I) /* see Section 3.2.1 */;
7  |  end
8  |  prev ←storeNodeLocations(M^O);
9  |  moveNodes(M^O, T_V) /* see Section 3.2.2 */;
10 |  q ← computeQuality(M^O);
11 |  if q < ε then
12 |  |  updateSomeNodes(M^O, q, prev);
13 |  end
14 |  regression← (computeDiscrepancy()> d_IN);
15 |  if regression == true then
16 |  |  updateAllNodes(M^O, prev);
17 |  end
18 |  iter ← iter + 1;
19 end
```

3.2.1 Target volume of output cells

For a cell $c_j^I \in M^I$, the discrepancy d_j indicates how well materials are preserved inside of c_j^I. For a specific material m, the discrepancy $d_{j,m}$ indicates how accurately m is preserved locally to c_j^I. It is this second quantity that we use for computing target volumes of the cells of the output mesh M^O. Let $c_i^O \in M^O_{|m}$, as c_i^O is a pure cell, it contains a single material. Let m be this material, the target

volume of c_i^O is defined as:

$$T_V(c_i^O) = V(c_i^O) - \sum_{c_j^I \in M^I} d_{j,m} \frac{V(c_i^O \cap c_j^I)}{V(M^O_{|m} \cap c_j^I)} \tag{3.2.1}$$

$$= V(c_i^O) - \sum_{c_j^I \in \{c^I \in M^I \mid c^I \cap c_i^O \neq \emptyset\}} d_{j,m} \frac{V(c_i^O \cap c_j^I)}{\sum_{c_k^O \in \{c^O \in M^O_{|m} \mid c^O \cap c_j^I \neq \emptyset\}} V(c_k^O \cap c_j^I)} \tag{3.2.2}$$

where $\{c^I \in M^I \mid c^I \cap c_i^O \neq \emptyset\}$ are the cells of M^I that intersect c_i^O and $\{c^O \in M^O_{|m} \mid c^O \cap c_j^I \neq \emptyset\}$ are the cells of $M^O_{|m}$ that intersect c_j^I. Figure 3.4 gives an illustration of Equation (3.2.1) where we look at the output cell c_i^O, which intersects two input cells c_1^I and c_2^I. For each of those cells, we compute a volume contribution, which is one term of the sum in the right term in Equation (3.2.1). Let us consider cell c_1^I for instance. Discrepancy $d_{1,m}$ indicates the quantity of material m that is under (i.e. $d_{1,m} < 0$) or over (i.e. $d_{1,m} > 0$) represented in c_1^I. For example, let us consider the first case when $d_{1,m} < 0$. It indicates that we do not have enough material m in c_1^I. Consequently we must inflate the cells of M^O that contain the material m and that intersect c_1^I. These are the cells c_i^O, c_p^O and c_q^O in our example. The *inflate* weight given to each cell c_i^O, c_p^O and c_q^O by c_j^I is proportional to their geometric intersection with c_1^I. For instance, for c_i^O, it is equal to

$$\frac{V(c_i^O \cap c_1^I)}{V(c_i^O \cap c_1^I) + V(c_p^O \cap c_1^I) + V(c_q^O \cap c_1^I)}.$$

The same computation is also done for the input cell c_2^I. Finally, let us note that as the discrepancy is negative when we do not have enough material in an input cell, the right term of Equation (3.2.1), equal to $T_V(c_i^O)$, is greater than $V(c_i^O)$.

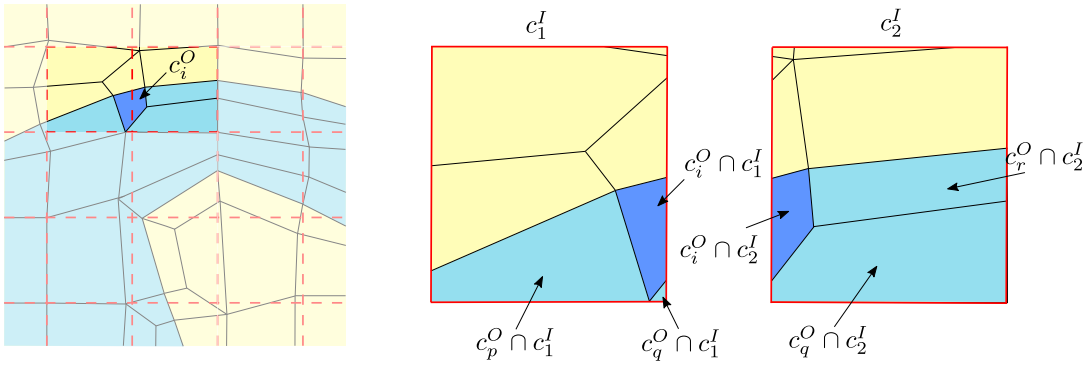


Figure 3.4: Illustration of the different geometric quantities used to compute the target volume of the cell c_i^O.

3.2.2 Interface node movements

Recalling that our aim is to retrieve as accurately as possible the volume of input materials, we try and move nodes located along material interfaces.

3.2.2.1 Overview of the algorithm to move interface nodes

Considering the output mesh M^O, where each cell is pure, the only degrees of freedom we have are the location of the nodes that are on the interface between distinct materials. Let n be such a node (see Figure 3.5). It is surrounded by four pure cells, denoted $\{c_1^O, \ldots, c_4^O\}$, which are all assigned to a specific material and have a target volume computed during the previous stage of the algorithm. The procedure used to move nodes is described in Algorithm 2 and depicted on Figure 3.5. It consists in three main stages:

1. First, we compute an *ideal* deformation factor for every cell c^O having at least one face lying on an interface between distinct materials (lines 1 to 9). Let $\{f_1, \ldots, f_k\}$ be those faces, the

deformation consists in defining one translation vector for each face f_i, with $1 \leq i \leq k$, which is their normals multiplied by the deformation factor.

2. By construction, each interface face f receives two translation vectors $\mathbf{v_1}$ and $\mathbf{v_2}$ computed in step 1, one per adjacent cells[3] (except when the void material is not explicitly meshed in M^O, in that case there is only one vector). We assign the average translation vector $\mathbf{t}_f = \frac{\mathbf{v_1}+\mathbf{v_2}}{2}$ to f (lines 10 to 13).

3. Finally, each node n that belongs to the interface between distinct materials is moved according to the translation vectors previously assigned to adjacent interface faces (lines 14 to 21). Let $\{f_1, \ldots, f_k\}$ be those faces, then n is translated along $\kappa(\mathbf{t}_{f_1}+\mathbf{t}_{f_2}+\ldots+\mathbf{t}_{f_k})/k$. The κ dampening term is progressively increased to 1 until the maximum number of iterations $maxIter$ is reached. Nodes constrained to the domain boundary, usually the bounding box when M^I is a grid and the void material is explicitly meshed, are then projected back onto this boundary.

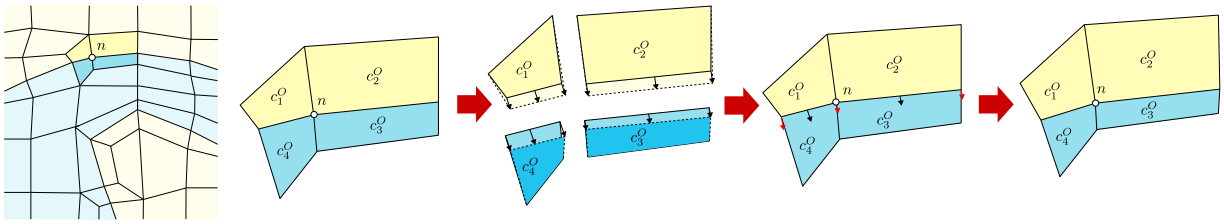


Figure 3.5: Illustration of the node movement procedure. Let n be an interface node and c_1^O, c_2^O, c_3^O and c_4^O its adjacent cells. Ideal shape is computed for each cell c_1^O, c_2^O, c_3^O and c_4^O by moving interface nodes. We then define one translation vector for each interface face, which is averaged at interface nodes to finally move the nodes of the interface all together.

Algorithm 2: Interface node movement

Data: M^O, $T_V : M_3 \rightarrow IR$
Result: $\mathbf{v} : M_0 \rightarrow IR^3$

```
1  t_map ← ∅ /*here map has the C++ STL container capabilities */;
2  for c^O ∈ M^O do
3  |  V_cO ← T_V[c^O];
4  |  {f_1, ..., f_l} ← getMaterialInterfaceFaces(c^O);
5  |  {t_f1, ..., t_fl} ← computeIdealShape(c^O, V_cO, {f_1, ..., f_l});
6  |  for f ∈ {f_1, ..., f_l} do
7  |  |  t_map[f].add(t_f);
8  |  end
9  end
10 f_map ← ∅;
11 for e ∈ t_map do
12 |  f_map[e.first] = (e.second[0] + e.second[1])/2;
13 end
14 for n ∈ M^O do
15 |  {f_1, ..., f_l} ← getMaterialInterfaceFaces(n);
16 |  n_map ← ∅;
17 |  for f ∈ {f_1, ..., f_l} do
18 |  |  n_map.add(f_map(f));
19 |  end
20 |  v[n] ← computeAverageVector(n_map[n]);
21 end
```

[3]Which belongs to distinct materials by definition.

3.2.2.2 Ideal deformation of a single cell

We drive the computation of the ideal shape of a cell c^O along its interface faces. It consists in moving its interface nodes only using each face's contribution to preserve their normals. To illustrate the proposed approach, let us consider the 2D example of Figure 3.6 where we list the five topological configurations we can encounter for a quad cell.

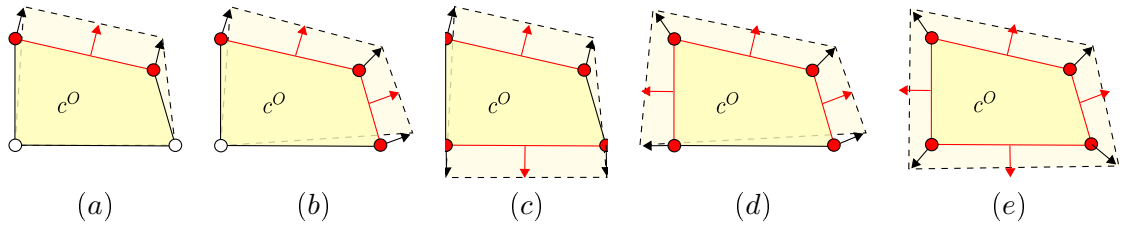


Figure 3.6: The five possible configurations to reshape a 2D cell to fit a specified target volume; the remaining other configurations can be obtained by rotation or symmetry of the node numbering in the 2D cell. Interface faces and free nodes are both colored in red. In (a), only one face is on the interface leading to only two free nodes that can move along the face normal. In (b) and (c), two faces are on the interface but in different topological configurations. In (d) and (e), respectively three and four faces are on the interface.

Those five configurations are the following ones:

1. In the first case (see Fig. 3.6.a), only one face is on an interface. As a consequence, only the two red nodes can be moved and they are constrained to move along the face normal (shown in black).

2. In the second case (see Fig. 3.6.b), two faces sharing a node are on an interface. The end nodes can move along their adjacent face normal, while the common node has to move along the sum of the faces' normals.

3. The three last configurations are built in the same way (see Fig. 3.6.c, 3.6.d and 3.6.e): a node adjacent to one interface face moves along this face normal, while a node adjacent to two interface faces moves along the sum of their normal vectors.

Let us note that the configuration with only one node to move on an interface cannot occur due to the pillowing process applied during the primal contouring procedure.

In practice, we compute the ideal deformation of a cell c^O with the following process. First we start by identifying the free nodes $\{n_1, \ldots, n_l\}$ of c^O and computing their respective direction vectors $\{\mathbf{u}_1, \ldots, \mathbf{u}_l\}$ as in Figure 3.6. A deformation factor for the cell is then computed by deforming c^O along the $\mathbf{u}$ vectors towards a target volume $T_V(c^O)$ using a bisection method. In case this computation fails, a default factor of $\sqrt[3]{T_V(c^O)} - \sqrt[3]{V(c^O)}$ that tries to give a rough estimate of the distance the nodes have to move for $V(c^O)$ to equal $T_V(c^O)$, is chosen[4].

Although the previous discussion illustrates the node movement procedure in 2D, the 3D extension is straightforward.

3.3 Volume preservation: some results

In this section we demonstrate and analyze the results of the proposed method applied to several cases that have been initially meshed using the SCULPT algorithm. Our method, given in Algorithm 1, uses several parameters, which are a maximum number of iterations *maxIter* and a quality threshold ϵ. We fix the value of *maxIter* to 30 in the presented results while the ϵ value depends on the cases. Indeed, since the M^O meshes have different initial minimum scaled Jacobian values, the ϵ value will be specific to each of them. For the test cases illustrated in Table 3.2 we used $\epsilon = 0.2$, and for those in Table 3.3 we used $\epsilon = \frac{3}{4} scaledjacobian_{init}$ that will allow for a controlled decrease in mesh quality by a quarter of its initial value. In all cases, the κ term, as introduced in Section 3.2.2,

[4] It would be the difference in edge length if c^O and deformed c^O were both cubes.

is equal to $0.1 + 0.9\frac{iter}{maxIter}$.

We first demonstrate the procedure using a simple CAD-based model that consists of the brick and cylinder configuration shown in Figure 3.7.a. Even if this model is not representative of our intercode issue, it will help us to show the mesh orientation sensitivity in Section 3.3.1. After that, we list a set of models we used for evaluating our algorthim and we analyse the obtained results for all of them.

3.3.1 Mesh orientation sensitivity

Figure 3.7.a shows a simple CAD-based model that consists of a brick and cylinder. Its corresponding Sculpt output M^O_{init} is shown in Figure 3.7.b, while Figures 3.8.a and b show the value of the discrepancy ratio per cell of M^I respectively before and after applying our algorithm. We can see that in both the input and output the worst cells are located near the sharp geometric features and that our method improved on that criteria. We note that the discrepancy decreases after applying our method and is consistent with Figure 3.9.a that shows the evolution in the distribution of the discrepancy ratio across the cells. The same is true for the computed target volume adjustment (defined as $T_V(c^O) - V(c^O)$) of the cells of M^O (see Fig. 3.8.c and d). Our measures confirm the visual observations from comparing Figures 3.8.a and b with Figure 3.8.a where we can see that the discrepancy is largest around the geometric sharp features, namely the curves and corners of the box and the curves of the cylinder, both in our input and output M^O mesh. It is particularly true along the top curve in (d) where the M^O mesh seems jagged. If we can observe that our solution improves the overall discrepancy, most notably around sharp features, it can also degrade mesh quality, because we are moving the cells at the interfaces, which usually are the cells that have the lowest quality in overlay grid methods. The user should chose a threshold depending on the requirements of the numerical simulation that will be run using this mesh.

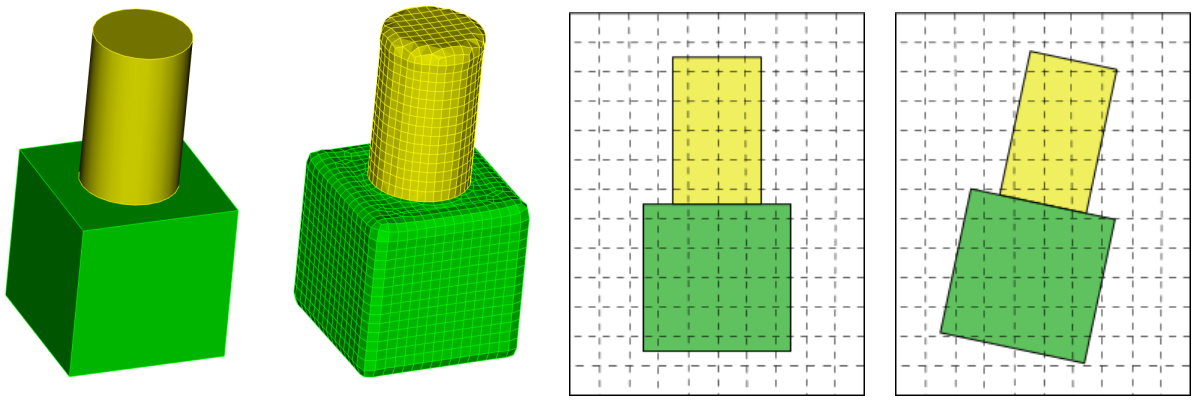
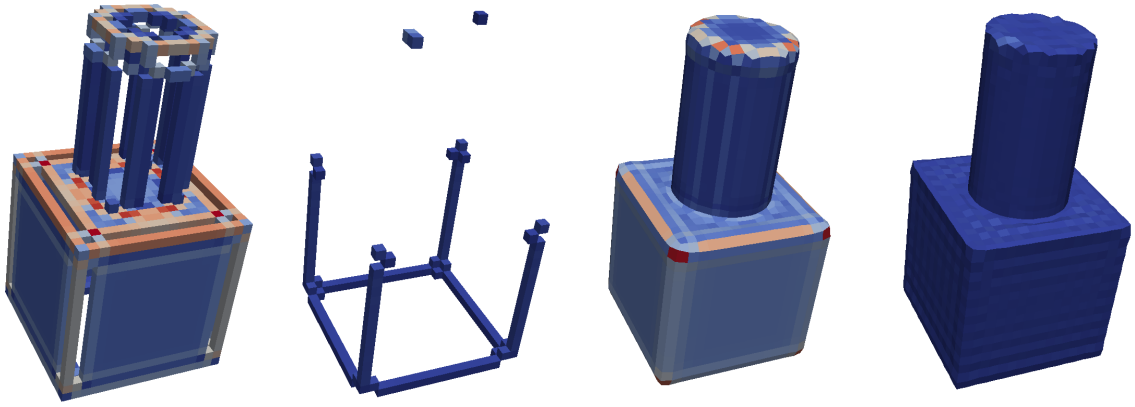


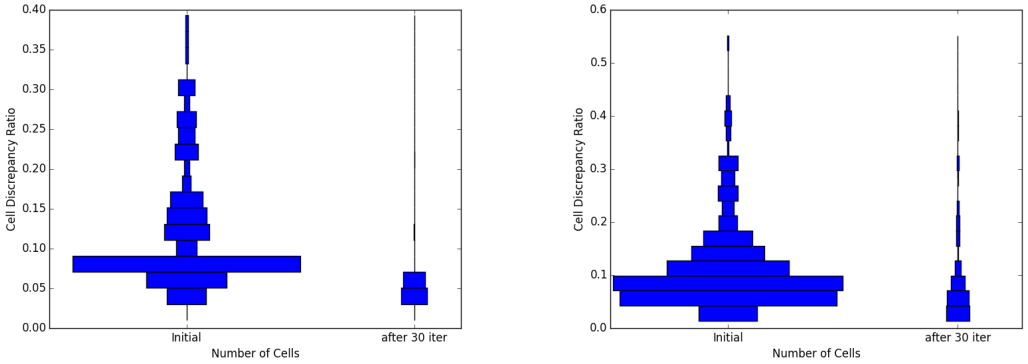
Figure 3.7: Brick cylinder example. (a) CAD model with two materials. (b) Sculpt output used as our M^O_{init} mesh (M^O_{final} can be seen in Figure 3.8.d). (c) 2D representation of CAD model with 0 rotation angle and (d) 10 degree rotation angle with respect to the overlay Cartesian grid.

Overlay grid procedures can be particularly sensitive to the orientation of the overlay Cartesian grid with respect to the reference geometry. In this example, we examine so the effect of the overlay grid orientation on the results of the volume preservation procedure. The following cases use the simple brick-cylinder configuration of the previous example while applying incremental rotations of 10 degrees to the geometry with respect to the initial overlay Cartesian grid. Tables 3.1 and 3.2 illustrate results from varying orientations from 0 to 90 degrees. We observe that our method consistently improves M^O, and note that although discrepancy improvements vary, they are indeed improved in all cases (see Fig. 3.9 and 3.10). In particular, the proposed method decreases the total discrepancy as well as the maximum discrepancy ratio per cell (see columns 6, 7 and 8 in Table 3.2).



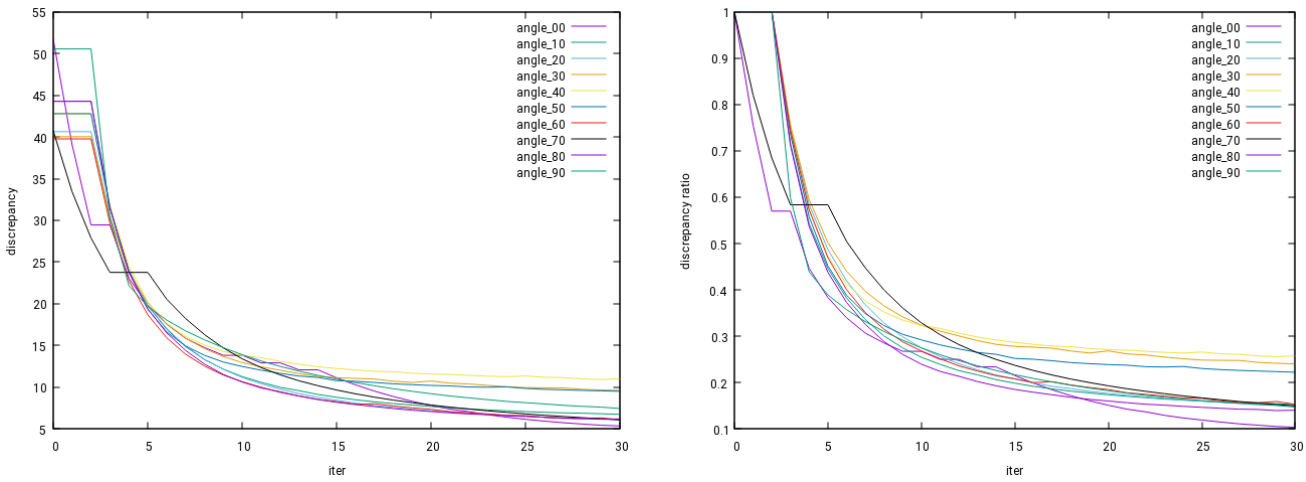
(a) (b) (c) (d)

Figure 3.8: Results for the brick-cylinder case. (a and b) initial and final discrepancy (cells $c^I \in M^I$ that have $\frac{d(c^I)}{V(c^I)} < 0.05$ are not represented) represented with the same color scale; (c and d) initial and final absolute value of the target volume adjustment



(a) (b)

Figure 3.9: Cell discrepancy ratio distribution in the initial and final M^O mesh (cells $c^I \in M^I$ that have $\frac{d(c^I)}{V(c^I)} < 0.05$ are not represented). Lower and narrower is better. (a) for the brick-cylinder with rotation angle 0, it confirms the difference seen between Figures 3.8.a and b. (b) same for the rotation angle 10 example.



(a) (b)

Figure 3.10: (a) Discrepancy evolution for the brick-cylinder cases compared with iteration. Note that a step in the graph is indicative that node positions did not move due to elements reaching a minimum ϵ quality threshold. (b) Same data but with the ratio $\frac{d_{final}}{d_{init}}$.

Table 3.1: Volume survey for the brick cylinder cases (the void material is excluded). We are interested here in evaluating the difference in material volume globally, as defined in Eq. 3.1.1, page 40. While volumes of the materials were already well preserved in the initial mesh our method still improves upon that.

angle $(degrees)$	$\frac{\Delta V_{init}(brick)}{V^I(brick)}$	$\frac{\Delta V_{final}(brick)}{V^I(brick)}$	$\frac{\Delta V_{init}(cyl)}{V^I(cyl)}$	$\frac{\Delta V_{final}(cyl)}{V^I(cyl)}$	$\frac{\Delta V_{init}}{V^I_{tot}}$	$\frac{\Delta V_{final}}{V^I_{tot}}$
0	8.79e-03	1.78e-05	-6.31e-03	-9.54e-06	8.25e-03	1.60e-05
10	2.26e-03	-1.39e-04	-2.07e-03	-5.27e-04	2.22e-03	2.24e-04
20	9.54e-04	-3.67e-04	-4.10e-04	-2.13e-04	8.34e-04	3.33e-04
30	7.99e-04	-5.49e-04	-9.21e-04	-6.77e-05	8.26e-04	4.43e-04
40	9.71e-04	-3.28e-04	-9.94e-04	3.81e-05	9.76e-04	2.65e-04
50	1.02e-03	-4.80e-04	-7.02e-04	9.02e-05	9.47e-04	3.94e-04
60	7.63e-04	-1.59e-04	-9.14e-04	-3.82e-05	7.96e-04	1.32e-04
70	9.62e-04	-2.39e-04	-1.61e-05	-1.97e-04	7.54e-04	2.30e-04
80	2.26e-03	-1.55e-06	-2.08e-03	-4.79e-04	2.22e-03	1.06e-04
90	-1.32e-03	-1.51e-04	4.19e-03	3.27e-04	1.95e-03	1.89e-04

Table 3.2: Discrepancy results for the brick cylinder cases. On the first row is indicated the columns' index. Columns 2 and 3 show the discrepancy as defined in Eq. 3.1.4. Columns 4 and 5 show the ratio of this discrepancy over the total volume of the materials; it expresses how "far" the M^O mesh is from the volume fractions carried by M^I. Column 6 is the ratio between the final and the initial total discrepancy where it is shown that our method has divided the initial discrepancy by a factor 4 to 10 depending on the case. Between columns 7 and 8 can be seen the improvement of the maximum discrepancy ratio per cell (the theoretical bound of those values is 2, as seen in Eq. 3.1.5). In columns 9 and 10 are the minimum scaled jacobian before and after applying our procedure.

1	2	3	4	5	6	7	8	9	10
angle	d_{init}	d_{final}	$\frac{d_{init}}{V_{tot}}$	$\frac{d_{final}}{V_{tot}}$	$\frac{d_{final}}{d_{init}}$	$\max_{c\in M^I}(\frac{d_{init}(c)}{V(c)})$	$\max_{c\in M^I}(\frac{d_{final}(c)}{V(c)})$	$scaledJ_{init}$	$scaledJ_{final}$
0	51.6792	5.36181	4.03e-02	4.18e-03	1.04e-01	4.03e-01	1.28e-01	0.39	0.40
10	44.2667	6.76245	3.45e-02	5.28e-03	1.53e-01	5.66e-01	3.97e-01	0.26	0.20
20	40.6279	6.07868	3.17e-02	4.74e-03	1.50e-01	5.68e-01	3.42e-01	0.33	0.20
30	40.0265	9.62336	3.12e-02	7.51e-03	2.40e-01	5.02e-01	3.90e-01	0.31	0.20
40	42.72	11.0175	3.33e-02	8.60e-03	2.58e-01	5.42e-01	5.37e-01	0.26	0.20
50	42.7854	9.51502	3.34e-02	7.43e-03	2.22e-01	5.42e-01	4.79e-01	0.28	0.20
60	39.7635	6.09284	3.10e-02	4.75e-03	1.53e-01	5.02e-01	3.81e-01	0.31	0.20
70	40.7345	6.10349	3.18e-02	4.76e-03	1.50e-01	5.68e-01	2.08e-01	0.28	0.20
80	44.256	6.20802	3.45e-02	4.84e-03	1.40e-01	5.66e-01	3.89e-01	0.26	0.20
90	50.5343	7.44741	3.94e-02	5.81e-03	1.47e-01	4.08e-01	2.38e-01	0.28	0.20

3.3.2 Our set of validation examples

For additional validation of the proposed method, we applied our approach on multiple types of input data given to Sculpt. We give in Table 3.3 the results from several examples based on different types of inputs, with timings for some of those in Table 3.4. Similar to the brick-cylinder case we note that discrepancies are reduced in all of the cases. We give more details about all those examples in the next paragraphs.

Simple CAD test cases

Anderson et al. [Anderson et al. 2010] 3D examples include two test CAD models depicted in Figure 3.11; a box with corner coordinates $(0.2, 0.2, 0.2)$ to $(0.6, 0.6, 0.6)$ intersected by a sphere of center $(0.6, 0.6, 0.6)$ and radius 0.2, with the space surrounding the two volumes being a third "void" material, and another consisting of five concentric spheres of radii $\frac{1}{13}$, $\frac{2.25}{13}$, $\frac{3.5}{13}$, $\frac{4.75}{13}$ and $\frac{6}{13}$ centered on $(0.5, 0.5, 0.5)$. Our method was successfully applied on both cases and in the box-sphere example in particular we can see in Figure 3.11.d when compared to Figure 10 in [Anderson et al. 2010] that our resulting mesh appears sharper, less "bloated" around the box's sharp curves and corners.

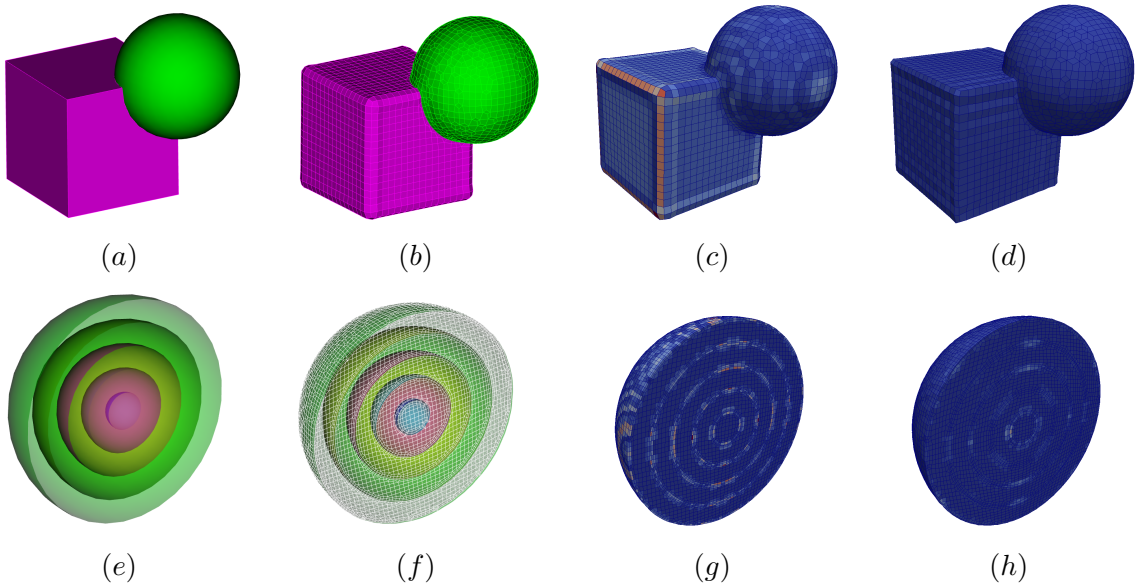


Figure 3.11: Simple example CAD test cases, top is the box-sphere, bottom is the five concentric spheres. (a) and (e) CAD models ; (b) and (f) Sculpt output M^O initial meshes; (c) and (g) M^O initial meshes with the absolute value of the target volume adjustment represented ; (d) and (h) the M^O final meshes after applying our method using the same color scaling as in (c) and (g) respectively.

Smooth models

The asteroid and lumbar models seen in Figure 3.12.a and b are both single volume cases that were given as STL files to Sculpt; they contain no sharp feature.

Microstructure models

The microstructure and the two_phase cases are shown in Figure 3.12.c and d. In the case of two_phase there are no mixed cells, making the M^I effectively a pure hexahedral mesh with perfect volume preservation. The objective behind running the Sculpt framework in this case is to smooth out the stair-like shape of the interfaces between materials.

CAD models single volume

The examples shown on Figure 3.13 are all comprised of a complex single volume which contains sharp curves and corners. They are representative of real manufactured pieces of equipment.

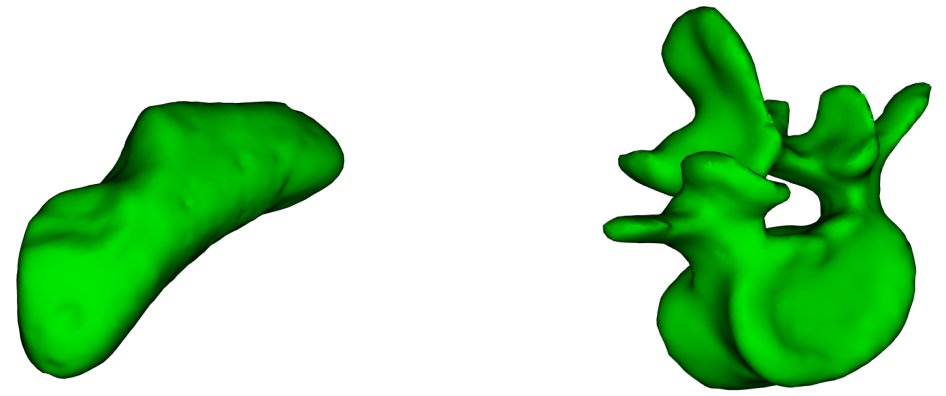
CAD models assemblies

The CAD assembly examples shown in Figure 3.14 are multi-volume models, which are typically formed by combining single volume CAD pieces such as those seen in Figure 3.13.

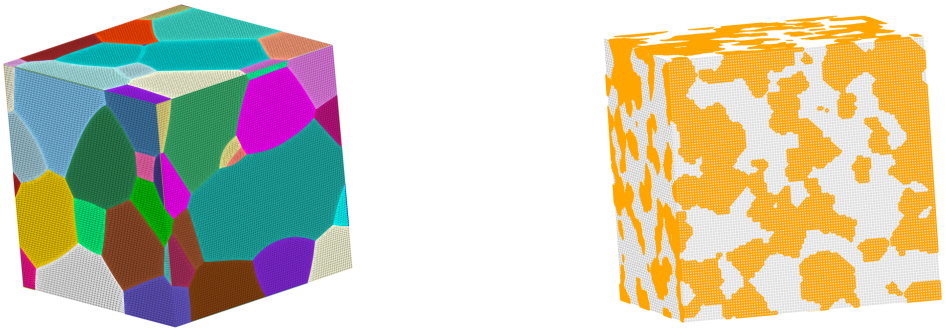
3.3.2.1 Simulation code output

These examples illustrate the usage of our method when applied in an intercode situation, where the output of a first code (an hexahedral mesh carrying materials volume fraction data) is used as the input of a second one (a full-hexahedral mesh with pure cells only). The first example shown in Figures 3.15 and 3.16 is commonly called the "triple point problem" and frequently appears as a test case in computational fluid dynamics; it shows the evolution of three fluids of different densities in a boxed domain.

The second example seen in Figure 3.17 is a domain where there are also three fluids of varying densities, this time laid in layers and stirred by two rotating blades.

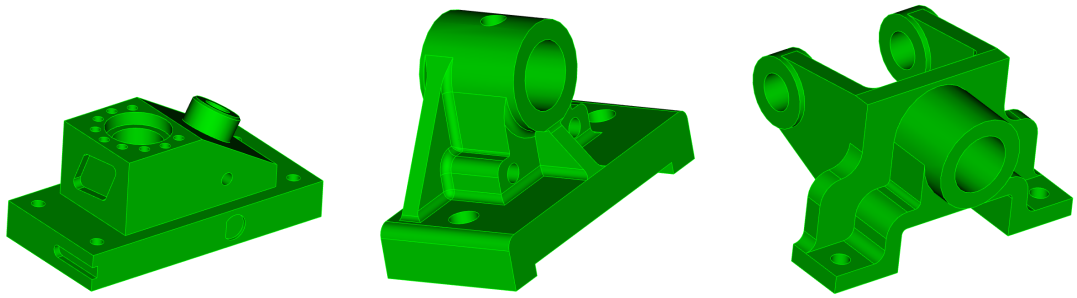


(a) asteroid (b) lumbar

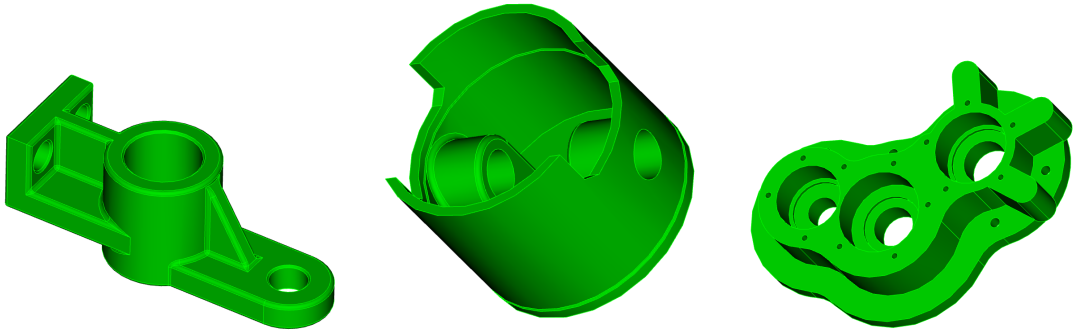


(c) microstructure (d) two_phase

Figure 3.12: Several input data. (a) and (b) are smooth models given as STL files inputs ; (c) and (d) are microstructure cases given as volume fractions carried by the grid.

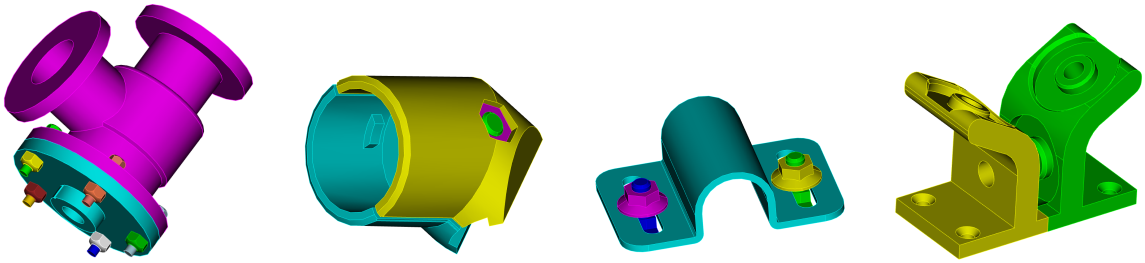


(a) anc101 (b) bearing (c) bevel



(d) bracket (e) piston (f) pump

Figure 3.13: Several single volume complex CAD models. Except for bracket, they all contain numerous sharp features.

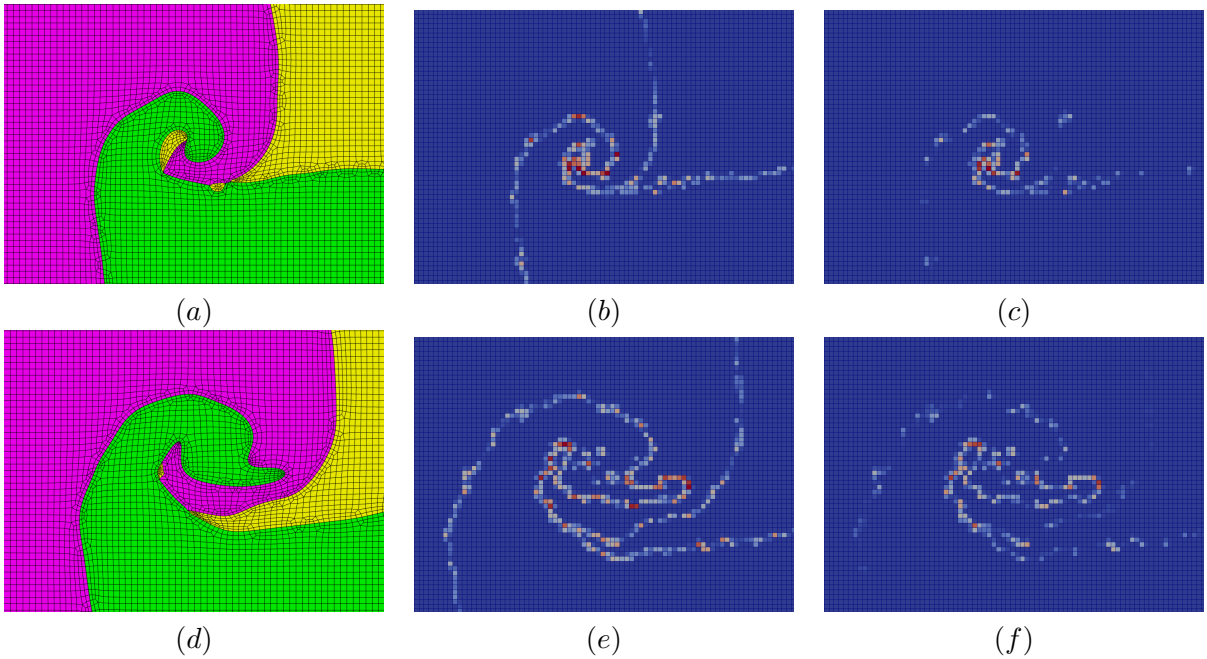


(a) v2_tweaked (b) assembly_1 (c) assembly_2 (d) lagerbock

Figure 3.14: Several multiple volumes CAD models.

(a) $t = 0sec$ (b) $t = 1sec$ (c) $t = 2sec$

Figure 3.15: The initial M^O mesh displayed without the edges at different time steps ($t = 0, 1$ and 2 seconds respectively) for the triple point problem. (a) shows the initial setup with the three fluids, (b and c) the formation of the vortex.



(a) (b) (c)

(d) (e) (f)

Figure 3.16: Close-up of the vortex in the triple point problem. (a) the output of Sculpt at $t = 1sec$, the M^O initial mesh ; (b) the discrepancy before applying our method ; (c) the discrepancy after, using the same color scale; (d, e and f) are the same at $t = 2sec$.

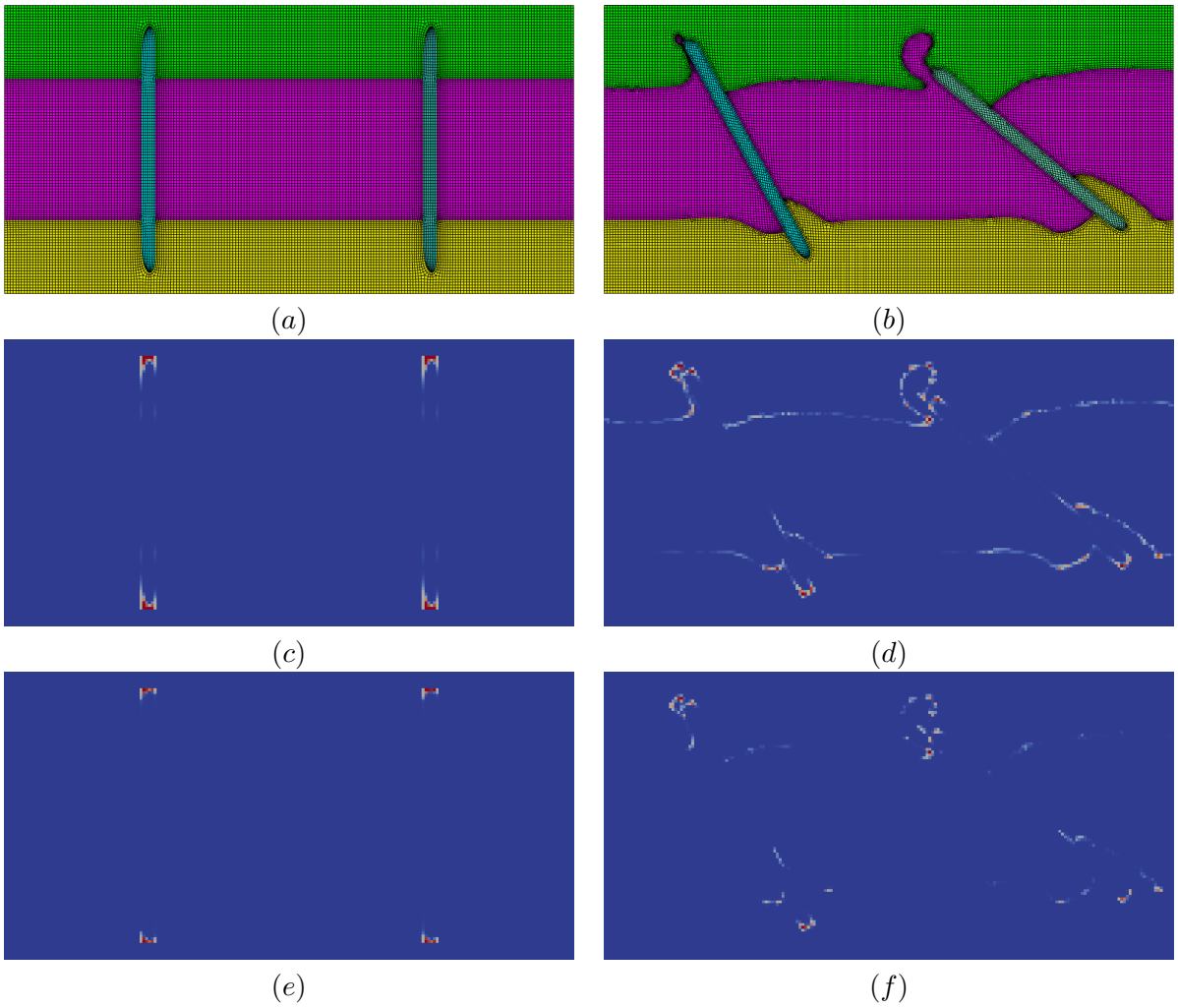


Figure 3.17: The doublebar simulation, at $t = 0sec$ on the left and $t = 0.5sec$ on the right. (a) and (b) the initial M^O mesh ; (c) and (d) the discrepancy on the cells of M^I before applying our method ; (e) and (f) the discrepancy after, using the same color scale.

The automatic nature of Sculpt and our volume preservation improvement method makes it possible to obtain pure hexahedral meshes at different time steps without user interaction, and in this intercode situation to execute the second code on different inputs more easily.

3.3.3 Results analysis

We note several observations from the preceding results. We focus on four specific features that required additional effort to apply or interpret the volume preservation procedures, namely: reloading node positions, fuzzy volume fractions and cell contribution error. The last one, the invalid mesh issue, led us to consider implementing our own pipeline in Chapter 5.

3.3.3.1 Reloading node position.

An initial implementation of our procedure reloaded nodal locations for all the nodes when minimum mesh quality fell below a threshold ϵ scaled Jacobian. This implementation proved problematic, particularly in cases such as illustrated in Figure 3.18.d, where relocating nodes towards an improved discrepancy would result in a few badly shaped elements. When this occurred, the procedure would reload the previous positions of all nodes of the mesh, negating the improvements where mesh quality remained above ϵ just to accommodate those few elements.

This proved to be a limiting factor of our method. As a consequence, we chose to identify such nodes and avoid moving them (see Algorithm (1) lines 11 to 13). This limits the amount of discrepancy improvement at these nodes in favor of preserving a minimum cell quality ϵ.

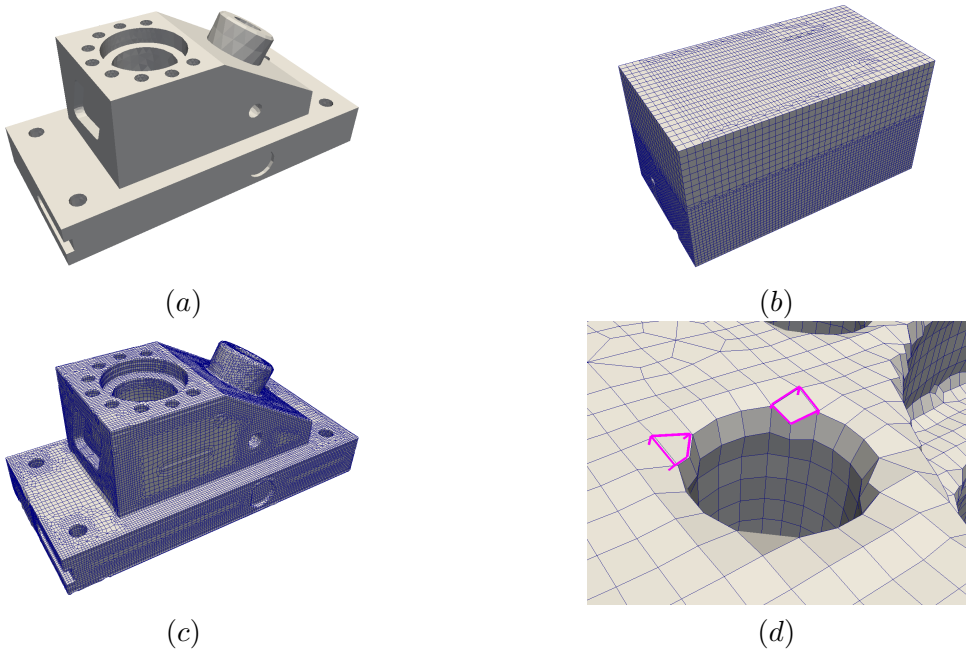
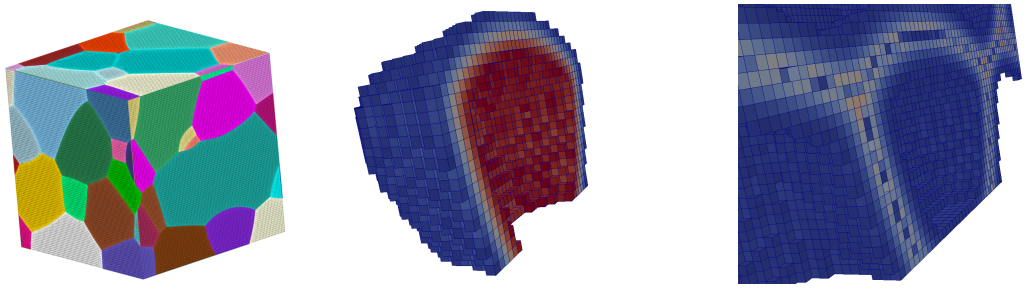


Figure 3.18: The anc101 example. (a) CAD model ; (b) M^I mesh, which is actually an adapted grid in this case ; (c) final M^O mesh ; (d) the two selected elements corner nodes should be moved outward in order to better follow the cylindric hole, but cannot be else the elements become non-convex.

3.3.3.2 Fuzzy volume fractions.

In some cases, input volume fractions will not necessarily denote a sharp interface. Depending on the technology used while acquiring the data or the way the volume fractions are procedurally generated, the transition region between two materials may spread across several cells such as that illustrated in Figure 3.19.b. We observe this phenomenon in the microstructure test case illustrated in Figure 3.19.a. While the proposed method can still decrease the discrepancy it is more difficult to interpret the results since it will stay high whatever the changes brought to M^O. (see results for microstructure

in Table 3.3) We note that this large transition between materials make for many additional non-zero discrepancy cells reducing the reported effectiveness of the procedure (see Fig. 3.19.c).



(a) (b) (c)

Figure 3.19: The microstructure example. (a) M^O mesh ; (b) close-up of the volume fraction of one material. The cells represented are those with a volume fraction ranging from 0.1 to 0.9. In this case we observe that the transition is more than 4 cells wide (c) the discrepancy (cells $c^I \in M^I$ that have $\frac{d(c^I)}{V(c^I)} < 0.05$ are not represented), which is not limited to a width of one or two cells intersecting the interfaces between material, but spreads farther. The cells that actually intersect the interfaces between materials can in fact have a lower discrepancy than their neighbors.

3.3.3.3 Cell contribution error.

We have observed that a cell of M^I, despite having a fraction of its volume composed of material m may have no part in moving the nodes of M^O at the interface of material m. Let c_i^I be a cell of M^I and m be a material where $f_{i,m} \neq 0$ and $c_i^I \cap M^0_{|m} = \emptyset$, in this case c_i^I has no contribution to the target volume of any cell of $M^O_{|m}$ according to Eq. 3.2.1. It is especially visible in the two-phase case : in Table 3.3 the maximum discrepancy ratio per cell $\max_{c^I \in M^I}(\frac{d(c^I)}{V(c^I)})$ is equal to the theoretical maximum of 2 (see Eq. 3.1.5) because some small areas of one material do not appear in M^O.

3.3.3.4 Invalid mesh.

The discrepancy measurement and improvement rely on being able to compute the intersection volumes between cells, which implies having no inverted cells in M^O to begin with. While Sculpt performs well on a wide range of cases it offers no guarantee of providing such a mesh. In particular we were met with difficulties in obtaining a valid mesh in cases issued from hydrodynamics simulations (see Table 3.5); this issue is addressed in Chapter 5.

Chapter digest – *In the course of this chapter we defined a discrepancy criterion that we used to measure the proximity of the output of Sculpt to its input; we then designed a discrepancy-driven algorithm that we applied to a wide range of examples, from different types of inputs, where we demonstrated its effectiveness in reducing this discrepancy by usually a factor of 10. We also included an orientation sensitivity study, as the orientation of the data compared to the axis can greatly impact the results of overlay-grid methods.*

1	2	3	4	5	6	7	8	9	10
case name	d_{init}	d_{final}	$\frac{d_{init}}{V_{tot}}$	$\frac{d_{final}}{V_{tot}}$	$\frac{d_{final}}{d_{init}}$	$\max_{c \in M^I}(\frac{d_{init}(c)}{V(c)})$	$\max_{c \in M^I}(\frac{d_{final}(c)}{V(c)})$	$scaledJ_{init}$	$scaledJ_{final}$
Simple test cases									
boxsphere	0.0030857	0.0004242	3.32e-02	4.57e-03	1.37e-01	3.70e-01	7.09e-02	0.27	0.34
concentric	0.0047499	0.0020702	1.16e-02	5.06e-03	4.36e-01	1.32e-01	1.24e-01	0.22	0.41
Smooth models									
asteroid	12.914	1.87471	5.16e-03	7.48e-04	1.45e-01	6.25e-02	9.80e-02	0.23	0.17
lumbar	0.480773	0.057342	8.18e-03	9.76e-04	1.19e-01	4.36e-02	2.09e-02	0.20	0.15
Microstructures									
microstructure	142693	137804	1.61e-01	1.56e-01	9.66e-01	1.44e+00	1.40e+00	0.11	0.08
two-phase	94746100	55919300	9.47e-02	5.59e-02	5.90e-01	2.00	2.00	0.19	0.14
CAD single vol									
anc101	88975.4	7164.96	1.08e-02	8.73e-04	8.05e-02	5.88e-01	4.52e-01	0.20	0.15
bearing	3079.08	407.641	1.37e-02	1.82e-03	1.32e-01	6.58e-01	5.61e-01	0.18	0.14
bevel	5644.87	740.943	2.21e-02	2.90e-03	1.31e-01	6.22e-01	4.82e-01	0.23	0.18
bracket	3.5111	0.261756	1.63e-02	1.21e-03	7.46e-02	1.31e+00	3.36e-01	0.25	0.19
piston	114.128	8.35095	4.32e-02	3.16e-03	7.32e-02	8.19e-01	6.60e-01	0.21	0.15
pump	53190.9	6015.45	2.34e-02	2.65e-03	1.13e-01	7.97e-01	5.67e-01	0.17	0.13
CAD assemblies									
v2_tweaked	212759	14622.9	2.72e-02	1.87e-03	6.87e-02	1.23e-01	5.75e-02	0.14	0.11
assembly1	2674.03	1725.06	1.16e-01	7.47e-02	6.45e-01	9.22e-01	9.22e-01	0.06	0.04
assembly2	3970.19	1061.27	1.34e-01	3.58e-02	2.67e-01	7.57e-01	7.57e-01	0.11	0.08
lagerbock	0.131722	0.0138628	1.94e-02	2.05e-03	1.05e-01	7.48e-01	5.14e-01	0.21	0.15
Simulation code									
triplepoint_1sec	583.983	248.594	2.53e-03	1.08e-03	4.26e-01	2.00e+00	2.00e+00	0.27	0.21
triplepoint_2sec	1204.86	513.38	5.23e-03	2.23e-03	4.26e-01	2.00e+00	1.82e+00	0.17	0.13
doublebar_0sec	386.834	188.521	6.26e-03	3.05e-03	4.87e-01	2.00e+00	2.00e+00	0.28	0.22
doublebar_0.5sec	852.687	351.924	1.38e-02	5.69e-03	4.13e-01	2.00e+00	2.00e+00	0.21	0.16

Table 3.3: Discrepancy results for several other examples. It is read the same way as Table 3.2.

Table 3.4: Mesh size and execution time for several of the examples.

case name	number of cells	execution time (s)
microstructure	1277808	19484
two-phase	1392928	32372
anc101	383704	13640
v2_tweaked	503566	11596
assembly1	56848	870
assembly2	78690	749
triplepoint_1sec	238518	5450
triplepoint_2sec	240570	5593
doublebar_0sec	71124	2230
doublebar_0.5sec	75216	2267

Table 3.5: Discrepancy comparison with Sculpt and Sculpt + our add-on. Fail indicates that we were unable to proceed due to the presence of inverted elements. We obtained our input mesh using Sculpt in Cubit $v15.4b$ with the *defeaturing* [Owen, Brown, et al. 2017] option activated. We do not claim to be exhaustive in testing Sculpt, which has many options to drive the mesh generation.

case name	grid resolution	*Sculpt*	*Sculpt*+ add-on
triplepoint 1s	420x180x3	583.983	248.594
triplepoint 2s	420x180x3	1204.86	513.38
doublebar 0.5s	200x100x3	852.687	351.924
doublebar 1.0s	200x100x3	fail	fail

Chapter 4

Geometrical model and voxelated interface reconstruction

In the previous section we defined a discrepancy criterion and used it to drive a method that modifies the mesh by moving its nodes to better match our input volume fractions. While we have shown that it is effective, including in improving the capture of sharp geometric details when available, it still remains a roundabout way of dealing with the fact that we do not have an explicit geometrical model. In this chapter, we present several methods that aim to produce such a model. As shown on Figure 4.1, a model will be created at the beginning of the ELG pipeline in order to provide a reference geometry to the next stages of the pipeline.

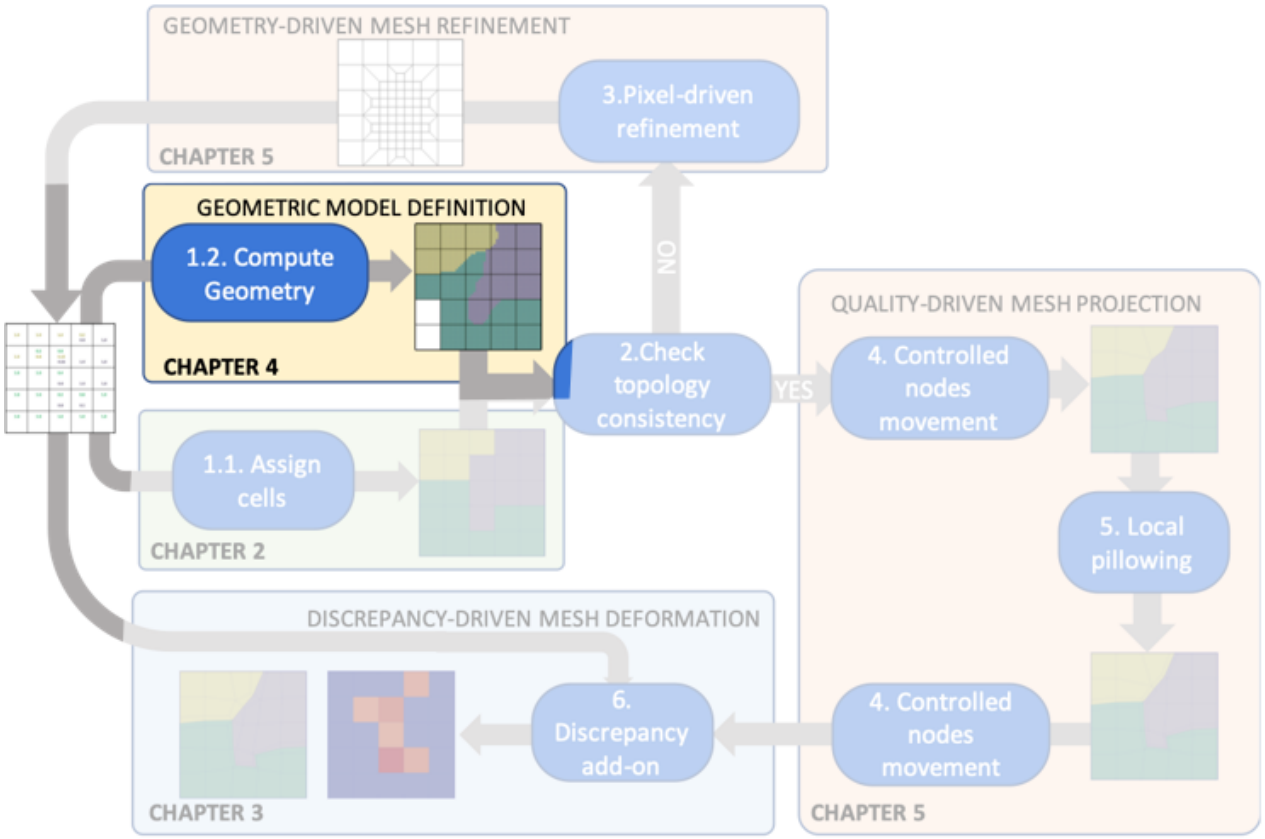


Figure 4.1: The geometrical model is built as the beginning of the ELG pipeline in order to provide a reference to the next stages of the pipeline.

4.1 Discrete interface reconstruction

Up to now we have been using a technique similar to the one in [Owen, M. L. Staten, and Sorensen 2012]) for computing a new position for the interface nodes (the nodes adjacent to cells assigned to different materials). This section focuses on building interfaces between materials that could help in the computation of this prospective new position. Extracting such interfaces will also bring the added benefit of producing a geometrical model.

Material interface extraction raises two main issues in our context: the desired interfaces should be smooth, typically when the goal is to visualize them or to use them to generate a mesh, but at the same time they should also fit the input data as best as possible, namely the volume fractions; those two objectives can conflict with one another. To address these issues we consider that several methods, coming from different fields, are relevant:

- **ALE-simulation interface reconstruction.**
  A domain where material interface reconstruction is extensively studied and applied is Arbitrary Lagrangian-Eulerian (ALE) CFD simulations. While those reconstructed interfaces have a built-in volume fractions preservation, they are not smooth, as they are discontinuous across cells as illustrated in Figure 1.7, page 12, taken from [Kucharik et al. 2010]. Most of these methods also have the additional drawback of being material order-dependent;

- **Original strategy**
  The position computed for the interface nodes in Sculpt [Owen, M. L. Staten, and Sorensen 2012], and that we reproduce in our original implementation, depends on the volume fractions (and their gradient) and the material assignment. It gives us new positions for the nodes and a geometrical model can be extracted that we have shown in Chapter 3 originally fits quite well the input, but we have also shown that it can still be improved. It also does not capture at all the materials that have no majority volume fractions in any of the mesh cells;

- **Voxel decomposition**
  These methods decompose the mixed cells into sub-elements – typically an hexahedron will be refined into a grid of voxels – on which a partitioning strategy is applied with respect to the volume fractions inside each cell. The interfaces at the sub-elements level are aliased, and since these methods originated from visualization purposes the interfaces are usually simplified into smooth triangular surfaces.

Considering that the original strategy – that we have extensively evaluated – can become limited in some cases, typically when the size of the material details is smaller than the mesh cells size, we will further study voxel-based methods in this section. We can note that as we handle both 2D and 3D cases in structured and unstructured cases, throughout this document we will use the term "voxel" as a misnomer in place of pixel (in 2D), sub-cell or sub-element.

4.1.1 How to partition voxels, an overview

The discrete voxel-based interface reconstruction techniques stems from the need to visualize the location of materials in the case where some of the cells are mixed and where the number of materials is greater than two. In the case where the number of materials equals two, classic iso-contouring methods provide a "clean" solution but with more materials small gaps or artifacts can appear that are non-desirable. In [Hege et al. 1997], the authors introduced the decomposition of mixed cells into subcells (or voxels) which are in turn assigned to the materials present in the mixed cells they were spawned from; the work in [Anderson et al. 2008; Anderson et al. 2010] extends it to cases with more than three materials per cell.

The voxel-assignment problem can be simply stated as :

> "*Considering a coarse mixed cell* cc *containing materials* $m \in \mathcal{M}$*, with volume fractions denoted* $f_{cc,m}$ *such that* $\sum_{m \in \mathcal{M}} f_{cc,m} = 1$*, and* cc *discretized as a set* $\mathcal{V}_{cc}$ *of* $nbsub$ *voxels, assign one single material* m *to each voxel of* $\mathcal{V}_{cc}$ *while ensuring material volume preservation*".

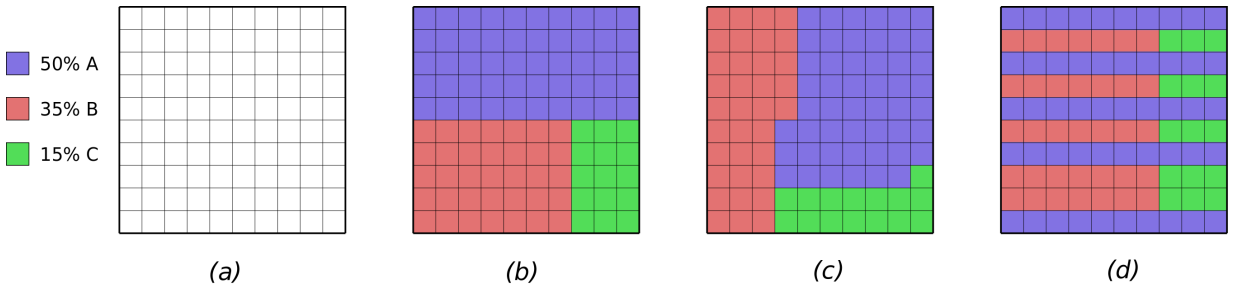


Figure 4.2: Example of the voxel-assignment problem and some unexpected valid results.

Figure 4.2 illustrates such a situation where in (a), a coarse mixed cell, made of 50% of material A, 35% of material B and 15% of material C, is split into 100 voxels. Results given in (b), (c) and (d) are valid solutions for the voxel-assigment problem, as they all respect the volume fractions, but they are wildly different from one another; it probably means that our problem could receive some additional description in order to be appropriately formalized. The voxel-assignment should aim towards several objectives:

- First, in order to enforce the volume preservation of each material, we need to favor solutions having a low $discrepancy$ (see in previous chapter Definition 5 at page 41), defined as the sum over each coarse cell cc of the absolute difference between the volume of each material present in cc (for material m it is $f_{cc,m}V(cc)$) and the sum of the volumes of the voxels of cc assigned to m. It expresses whether the voxels material assignment fits the volume fractions;
- Secondly, as is usual in partitioning algorithms, we want to favor connected components for each material. It translates into minimizing the $edgecut$ function, defined as the sum of the number of pairs of adjacent voxels assigned to different materials;
- Thirdly, surrounding pure cells of cc provides the initialization to our problem. If a mixed cell is bounded by pure cells, then we extend the "voxelization" process to the neighborhood of cc, i.e all the mixed cells and their adjacent pure cells are subdivided into voxels, and voxels spawned from pure cells are already assigned to the material of their corresponding pure coarse cell, leaving those spawned from mixed coarse cells as "free" (see figure 4.3).

In order to solve this problem, we compare the four following methods:

- A mixed-integer linear programming definition, that gives us an optimal solution to the problem. We introduce such a definition to get a formal description of the problem and also reference results on simple toy examples. But it is too expensive in terms of computation time to be used in practice;
- Simulated annealing was used in [Anderson et al. 2008] and can be found implemented in *Visit* [Childs et al. 2012] to solve this problem for scientific visualization purposes;
- Considering the problem as partitioning a graph, techniques like graphcut can be used;
- Finally, we propose a greedy heuristic that is designed to fit our specific requirements.

4.1.1.1 Mixed-Integer Programming Formulation

The problem of voxel-assignment can be formulated as a linear problem, which is the following one:

$$\left\{ \begin{array}{ll} min \sum\limits_{v\in\mathcal{V},m\in\mathcal{M}} |ma_{v,m} - \frac{1}{|N(v)|} \sum\limits_{w\in N(v)} ma_{w,m}| & \\ \text{constrained to} & \\ ma_{v,m} \in \{0,1\} & \forall v,m \\ \sum\limits_{m\in\mathcal{M}} ma_{v,m} = 1 & \forall v \\ \sum\limits_{v\in\mathcal{V}_{cc}} ma_{v,m} = nbsub * f_{cc,m} & \forall m, \forall cc \end{array} \right\}$$

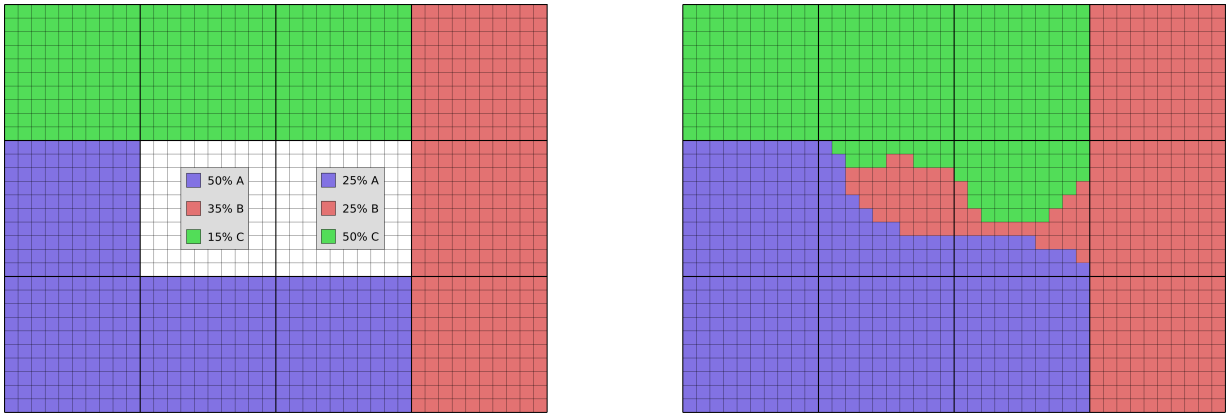


Figure 4.3: Example of two mixed cells surrounded by pure cells which voxels must be partitioned into materials (left). Considering adjacent pure cells and the objective of getting connected areas of same material could lead to the result shown on the right.

where $\mathcal{V}$ is the total set of voxels of the whole domain, $ma_{v,m} = 1$ if voxel v is assigned to material m and $ma_{v,m} = 0$ otherwise; $N(v)$ is the set of voxels adjacent to v. The first two constraints indicate that every voxel has one assignment and only one. The third constraint expresses that we want to have a $discrepancy$ equal to zero ($nbsub$ being the number of voxels in a coarse cell cc). The objective function that we want to minimize reflects the aim for voxels assigned to the same material to be clustered together, i.e. having a low $edgecut$.

Since our variables are integers, we in fact have a mixed-integer linear problem. This type of problem can be solved using software such as GLPK [GLPK 2019], LP_SOLVE [Berkelaar et al. 2004], CPLEX [*CPLEX Optimizer* 2020] or GUROBI [Gurobi Optimization 2020].

4.1.1.2 Simulated Annealing

This method introduced in [Anderson et al. 2008] to be used for the specific problem of voxel-assignment, consists in randomly assigning the voxels to materials with respect to the volume fractions data. Pair of voxels spawned from the same coarse cell will then swap their assignment, as described in Algorithm 3. Since the initial assignment to the voxels fit as best as possible the volume fractions and that only swaps are performed, the resulting voxels material assignment fits just the same. Note that this property does not hold when the mesh is unstructured, or more precisely when the sub-elements are not really voxels because they have different sizes (see Figure 4.6). It is also not so simple to devise a termination criteria; in the implementation depicted in Algorithm 3 available in VISIT, this issue is bypassed in favor of a user-specified timer.

4.1.1.3 Using the Graphcut algorithm

In [Boykov, Veksler, et al. 2001; Boykov and Kolmogorov 2004; Kolmogorov and Zabin 2004] the authors identify classes of energy functions that can be minimized using a graphcut. They formulate a voxel-labeling problem in this form, expressed as the sum of a so-called data (D)and smooth (V) costs where voxels are graph nodes and adjacency connections between voxels are graph edges. We use here our notations, with $l_v \in \mathcal{M}$ being the label (or material assignment) of the voxel $v \in \mathcal{V}$; we want to find the labeling l that minimizes:

$$E(l) = \lambda \sum_{v \in \mathcal{V}} D_v(l_v) + \beta \sum_{\{v,w\} \in \mathcal{V}^2} V_{v,w}(l_v, l_w)$$

with basically the second term being the edgecut:

$$V_{v,w}(l_v, l_w) = \begin{cases} 0. & \text{if } l_v = l_w \\ 1. & \text{otherwise} \end{cases}$$

Algorithm 3: Voxels assignment via simulated annealing (as implemented in VISIT version 2.13.3, a scientific visualization software).

Data: volume fraction VF, voxelated sub-mesh
Result: voxels assignment

```
1  /* temperature stays constant but it could decrease over time */
2  T ← 0.25
3  while time ≤ allotedtime do
4      iter ← 0
5      for iter ≤ 1000 do
6          /* randomly select a mixed (coarse) cell */
7          cc ← getMixedCell()
8          /* randomly select a pair of voxels of cc assigned to different materials; give up after
              ten tries */
9          v0, v1 ← cc.getSwapCandidates()
10         l0 ← label(v0)
11         l1 ← label(v1)
12         /* evaluate the energy with the current labeling and the prospective one where the
              labels are swapped */
13         current_energy ← energy(v0, l0) + energy(v1, l1)
14         future_energy ← energy(v0, l1) + energy(v1, l0)
15         if future_energy < current_energy then
16             swap(v0,v1,l0,l1)
17         else
18             ΔE ← |future_energy − current_energy|
19             if rand(0, 1) < e^{−ΔE/T} then
20                 /* randomly swap anyway depending on the temperature */
21                 swap(v0,v1,l0,l1)
22             else
23                 if future_energy = current_energy and rand(0, 1) < 0.5 then
24                     /* when equal randomly swap */
25                     swap(v0,v1,l0,l1)
26                 end
27             end
28         end
29         iter++
30     end
31 end
```

and the cost of assigning material m to a free voxel v (with $cc(v)$ the coarse cell that spawned v and $nearest(m)$ the nearest voxel issued from a pure cell assigned to m):

$$D_v(m) = (1. - f_{cc(v),m})v.distance(nearest(m)).$$

We should note that the first term D does not enforce matching the volume fractions. In order to apply the graphcut, this term has to be dependent on only v; the expression that we chose tries to emulate that property (assigning material m to v costs less the higher the volume fraction from its coarse cell $f_{cc(v),m}$ is and the nearer there is a voxel spawned from a pure cell assigned to m) but we will see in the results (Figure 4.5-c) that it is far from being effective. A second issue comes from the energy function itself that is the sum of two terms not related to one another, and the values chosen for (λ, β) will impact the results obtained making this method difficult to use.

4.1.2 Our method - Greedy Heuristic

We have implemented a greedy heuristic (see Algorithm 4) where at each iteration the free voxels are assigned volume fractions that depend on the values in their respective coarse cells adjusted to take into account the voxels that were already assigned (see the evolution at several iterations in Figure 4.4); 3D results are shown in Figure 4.7.

Algorithm 4: Voxels assignment greedy heuristic.

Data: volume fraction VF, voxelated mesh
Result: Voxels assignment

```
threshold ← 1.
freeVoxels ← allVoxels
fixedVoxels ← ∅
vf ← (VF, freeVoxels)
for freeVoxels ≠ ∅ do
    /* get the free voxels with a vf higher than the threshold for one material */
    fixedVoxelsToAdd ← extractVoxelsAbove(freeVoxels, threshold)
    fix(fixedVoxelsToAdd)
    if fixedVoxelsToAdd ≠ ∅ then
        reduce threshold
    end
    /* update the vf while substracting the voxels already assigned */
    vf ← update(VF, freeVoxels)
    for iter ≤ maxNbIter||convergence do
        /* kind of a vf smoothing */
        vf ← average(vf) for voxels where < threshold
        normalize(vf)
    end
end
```

The underlying idea of this algorithm is to assign a material to each voxel following an advancing-front strategy. We consider a set S of connected mixed cells as a starting point (orange cells on Figure 4.4-a). Each cell cc of S is split into voxels that we have to assign to a specific material. The material each voxel will be assigned to depends on the volume fractions of materials that compose its parent cell in S. For example, on Figure 4.4, volume fractions of the central cell are given; the central cell should be filled by 40% of green and 60% of grey voxels at the end.

In order to assign a material to a voxel v, we consider the materials that are already assigned in its vicinity (the 8 surrounding pixels in 2D when the case is structured) and we diffuse those materials into the voxel v. Voxel v is assigned to a material m if its newly computed volume fraction is higher than a minimum threshold. The threshold value is iteratively decreased in order to avoid blocking situations where the algorithm is unable to assign a material to any voxel during an iteration. At the end of each iteration, the volume fractions to reach for each material in a cell are updated (see

Figure 4.4-a to e). With this strategy voxels on the boundary of S tend to be assigned first and we get the expected advancing front assignment.

4.1.3 Comparative study

The four methods were applied on several examples, including a 5x5 2D example for which results are given in Figure 4.5. The MIP implementation is impractical, as it does not return a solution in an acceptable time; it can return a valid (meaning that it fits the constraints) but not optimal solution, which is the case in Figure 4.5-a. The graphcut approach tends to return straight interfaces, resulting in a good edgecut, but as we have mentioned is quite bad when considering the $discrepancy$. That leaves us with the simulated annealing, which is a little better than our greedy heuristic regarding the edgecut in the case of a grid, but fares badly concerning the $discrepancy$ in unstructured cases, as shown in Figure 4.6 and 4.8. All of those methods have the same memory limitation, as the submesh, i.e. the set of voxels, can be quite large. In practice, we will use our heuristic to build the voxelated interfaces (see Figure 4.7 for 3D results), as it is a good compromise between structured and unstructured cases and does not rely on tuning parameters depending on the case.

4.1.4 Voxel assignment correction - Repartitioning

We have seen that on "real-life" examples our greedy heuristic fares quite well; on "toy-like" simple examples it can exhibit some traits that may be deemed undesirable, such as a tendency to produce a voxels assignment with a few isolated voxels, which leads us to believe that the edgecut could be improved, as illustrated in Figure 4.9. Our approach tends to clump together voxels assigned to the same material but we do not make it mandatory for a free voxel to be assigned a material one of its neighbors is already assigned to.

In order to counteract this phenomenon we have devised and tested a post-process that can be applied after our greedy algorithm and that spawns from a simple consideration: as the voxels assignment can be seen as a graph partitioning problem, adjusting the obtained partitions can be considered a repartitioning problem. We have thus experimented with two well-known repartitioning algorithms that both start from an initial assignment[1] and that we have adapted to our constraints, the Kernighan-Lin algorithm in Section 4.1.4.1 and the Fiduccia-Mattheyes algorithm in Section 4.1.4.2. The interested reader can find some more up-to-date references on the subject of graph partitioning in [Morais 2016; Barat 2017].

4.1.4.1 Kernighan-Lin

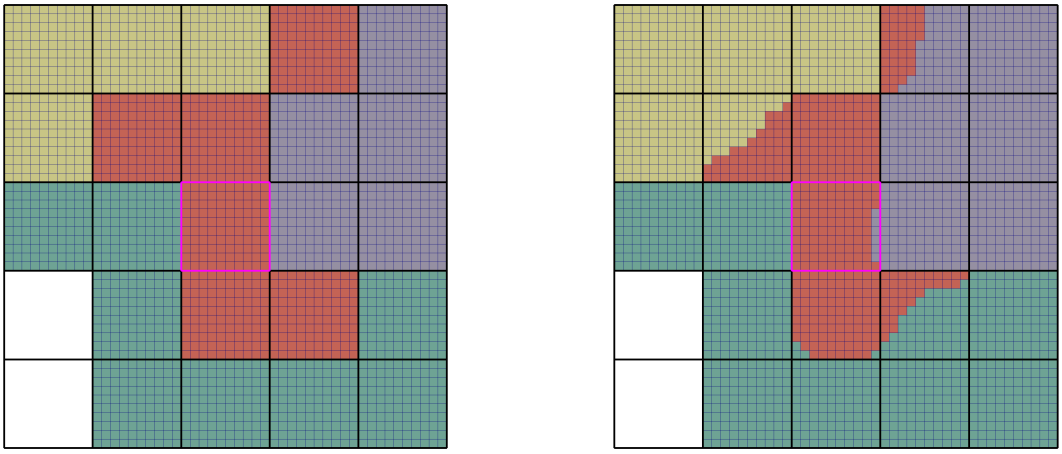
The Kernighan-Lin [Kernighan and Lin 1970] graph bi-repartitioning algorithm takes as an input a graph with its vertices split into two sets and proceeds to improve upon the initial partitioning by exchanging vertices between the sets, two by two. The simulated annealing implementation (see Section 4.1.1.2) is in fact based on the same principle, but the Kernighan-Lin algorithm drives the swaps by determining the sequence of swaps that maximizes the gain[2]; incidentally it allows for "bad" moves, or negative gain moves as long as they are compensated. Our implementation is depicted in Algorithm 5 where our modifications are annotated: they first consist in having to contend with more than two partitions and secondly to restrict the possible exchanges to the swaps between voxels spawned from the same source cell; this is done so that starting from an initial assignment valid in terms of volume fractions constraints the output will still be valid under those same constraints.

Results in Figure 4.9 show that it is quite effective in reducing the *edgecut* when there are isolated assignment artifacts.

As the Kernighan-Lin method base operation is the swap, it has the exact same issue as the simulated annealing when the coarse mesh is unstructured and the voxels do not all have the same volume; it will preserve the number of voxels assigned to each material, and while it may improve on

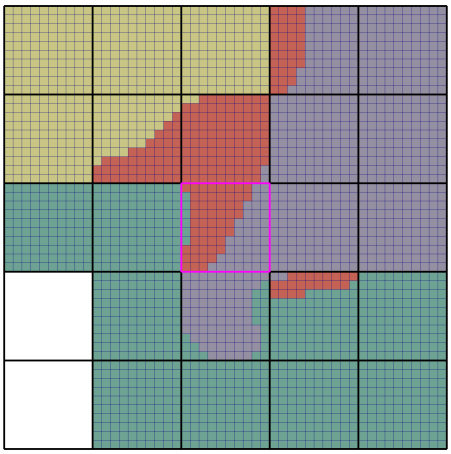
[1] The initial voxels assignment need not be our greedy heuristic, it could come from another method entirely, including a randomized assignment that respects the volume fractions similar to the starting stage of the simulated annealing method.

[2] "gain" in terms of improving the *edgecut*.

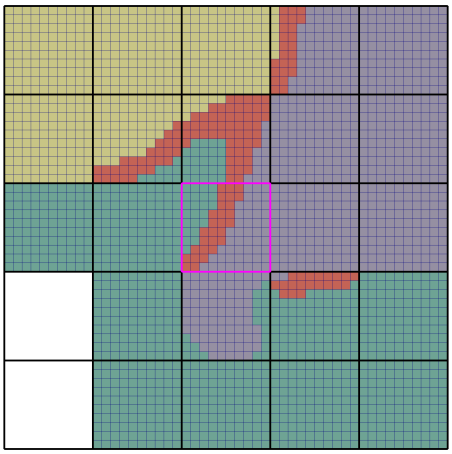


(a) vf=0.40 , vf=0.60

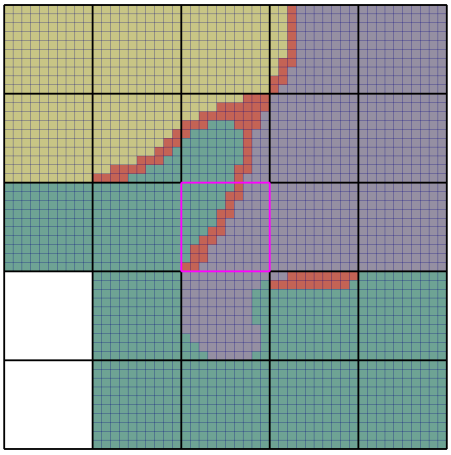
(b) vf=0.43 , vf=0.57



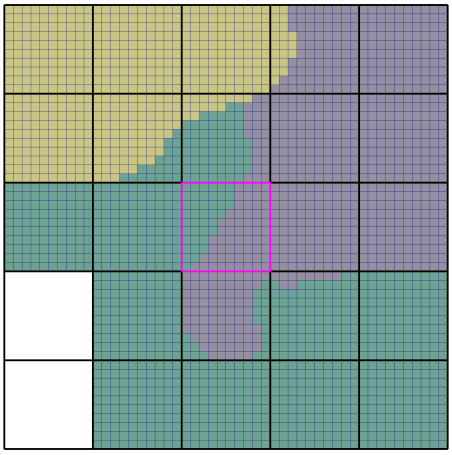
(c) vf=0.62 , vf=0.38



(d) vf=0.58 , vf=0.43

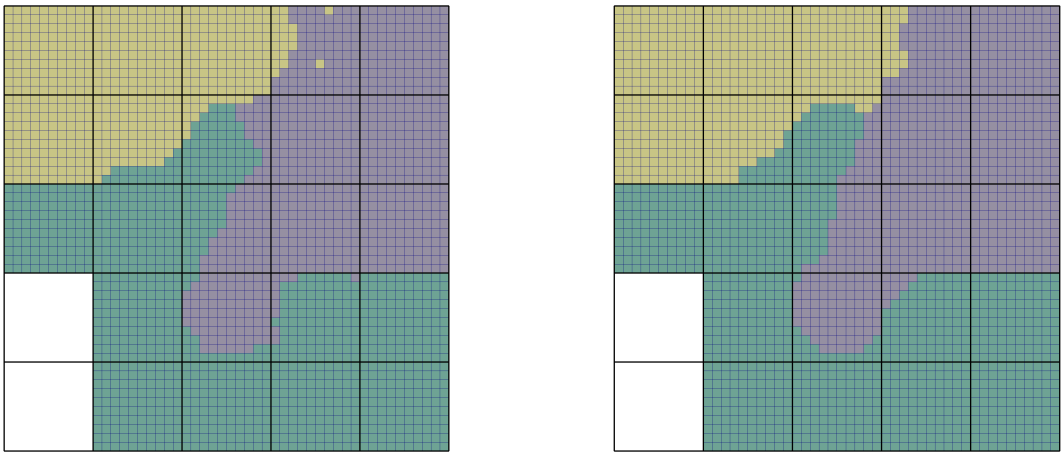


(e) vf=0.53 , vf=0.47



(f)

Figure 4.4: Greedy heuristic applied to the 5x5 example where we can see the evolution of the volume fractions (see Algorithm 4:13) assigned to the free voxels of the central coarse cell below each figure. The wireframe black grid is the coarse mesh and the voxels colored in orange are those not yet assigned.



(*a*) MIP
d=0, edgecut=592

(*b*) simulated annealing
d=0, edgecut=536

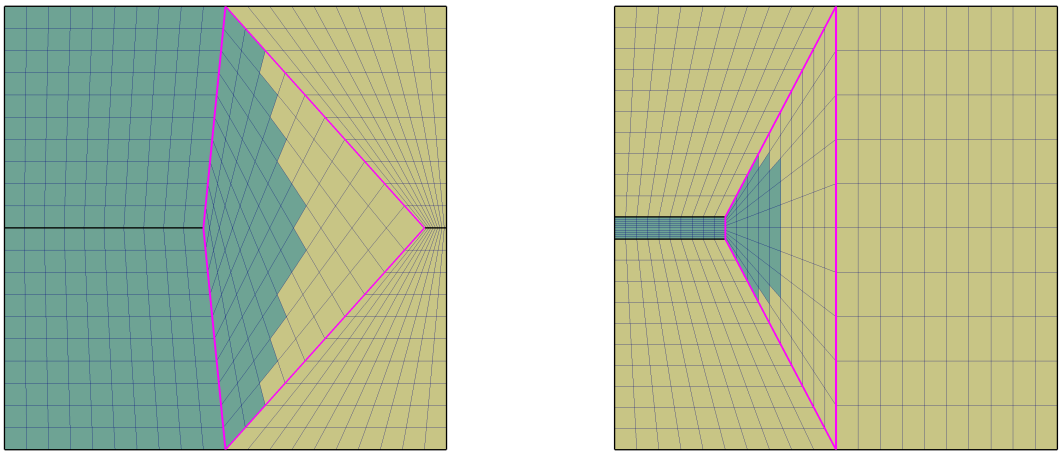
(*c*) graphcut
d=2.18, edgecut=440

(*d*) greedy heuristic
d=0, edgecut=568

Figure 4.5: Comparison of the voxelated interfaces reconstruction methods. Note that we stopped the MIP solver execution after 5 minutes.

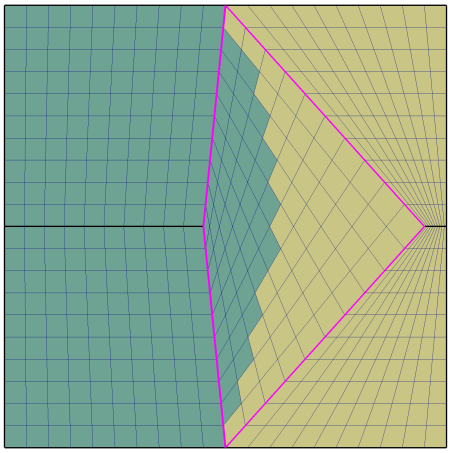
Algorithm 5: Kernighan-Lin.

```
1  while gain_cumul > 0 do
2      matAssign_tmp ← matAssign
3      compute costs for all vertices  /* one cost per material */
4      free all vertices
5      while ∃ possible swap do
6          v0, v1 ← find best swap  /* swaps between voxels of the same source cell */
7          store best swap in sequence
8          lock(v0), lock(v1)
9          swap(matAssign_tmp(v0), matAssign_tmp(v1))
10         update costs for v0, v1 and neighbors
11     end
12     gain_cumul ← find sequence of swaps with maximal cumulative gain
13     if gain_cumul > 0 then
14         matAssign ← execute swaps
15     end
16 end
```

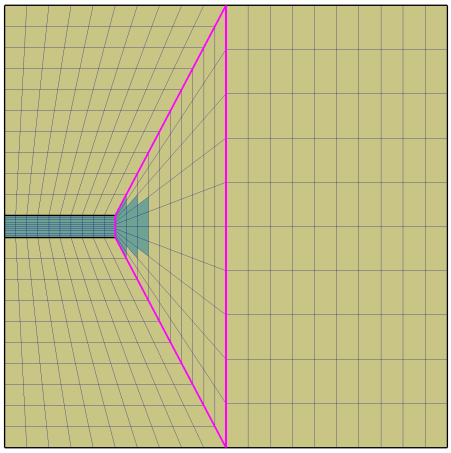


(a) d=0.0768

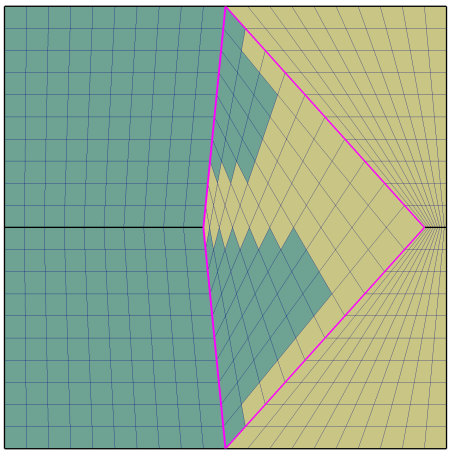
(b) d=0.024



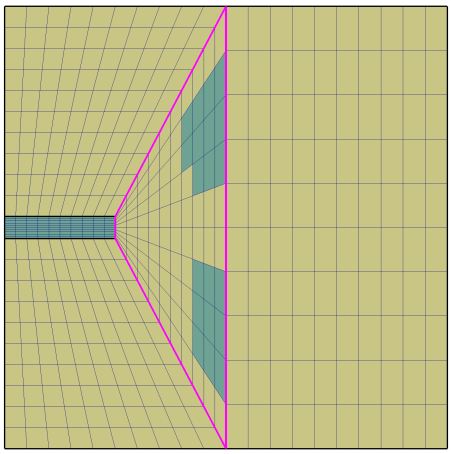
(c) d=0.264



(d) d=0.1444

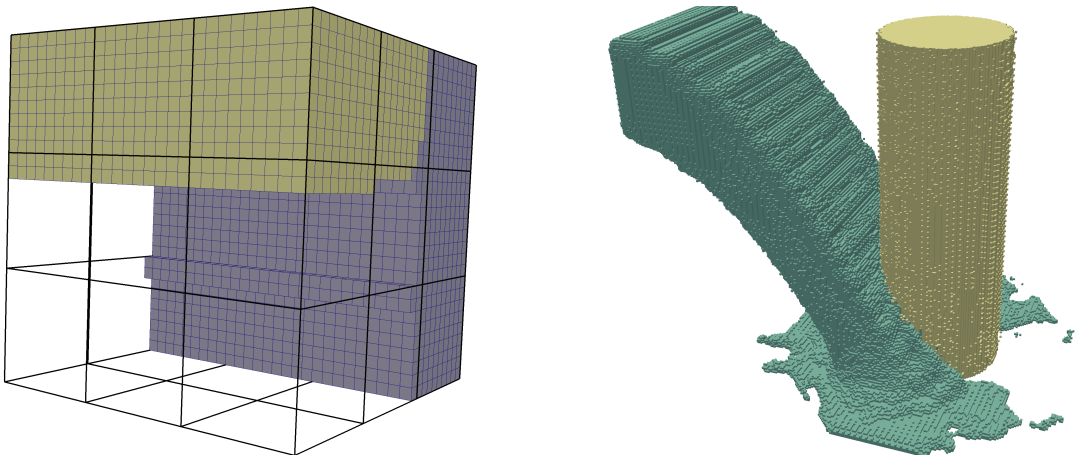


(e) d=0.1344



(f) d=0.1254

Figure 4.6: Greedy heuristic (first row) versus simulated annealing (second and third rows) applied to unstructured cases. Only the highlighted cell is mixed, and the respective volume fractions are (0.5,0.5) in the case on the left, (0.2,0.8) on the right. Two different results (c and e), (d and f) are shown for the simulated annealing method because some cluster of voxels can appear due to the randomness of the initial voxel assignment and the swaps.



(a) (b)

Figure 4.7: Greedy heuristic applied on two 3D cases with three materials. The third material is hidden. (a) an extruded case; (b) a 3D case from a CFD simulation.

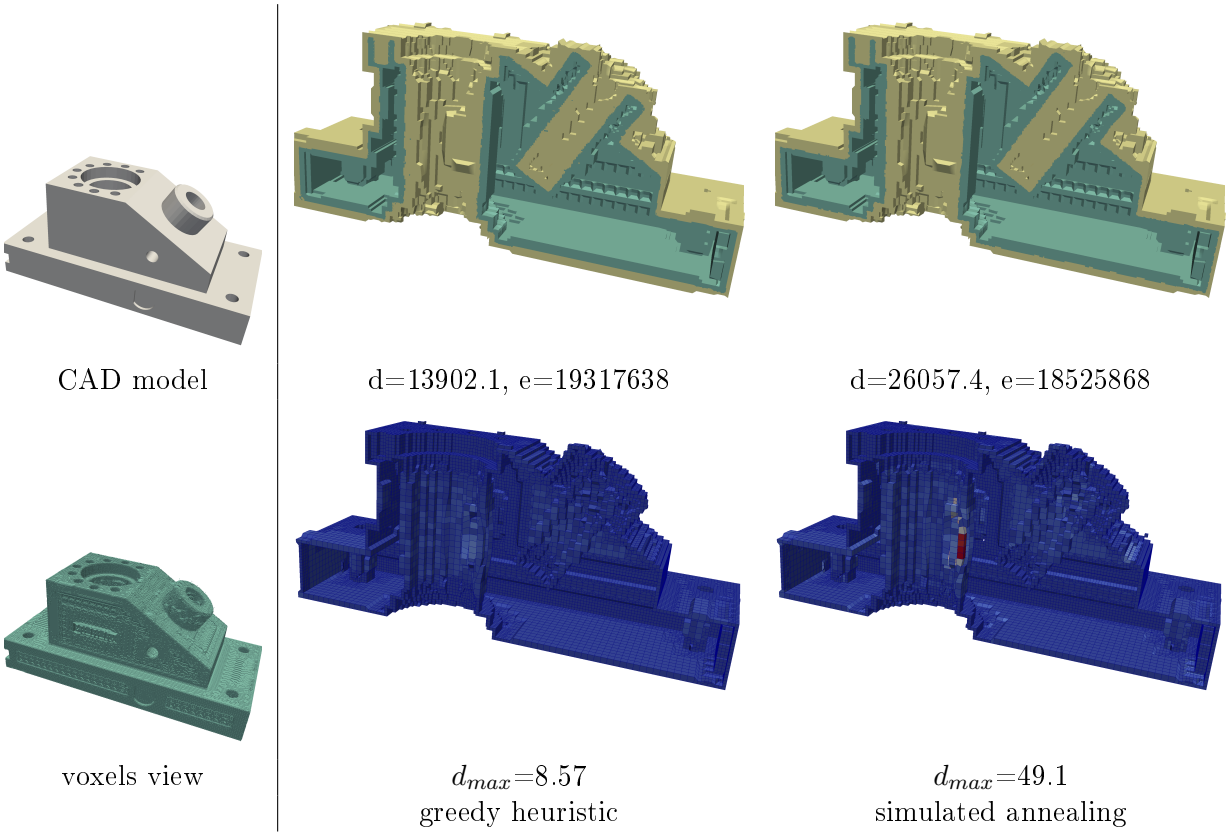


Figure 4.8: Example in a real-life unstructured case. The results show that while our greedy heuristic (left) is a little worse edgecut-wise than the simulated annealing (right), it fares better by a factor of 2 and 5 in terms of discrepancy, the total sum and its cell maximum respectively.

the edgecut it may lead to an increase of the discrepancy (see Figure 4.10). In order to address this issue we applied to our problem the Fiduccia-Mattheyes repartitioning algorithm in Section 4.1.4.2.

4.1.4.2 Fiduccia-Mattheyes

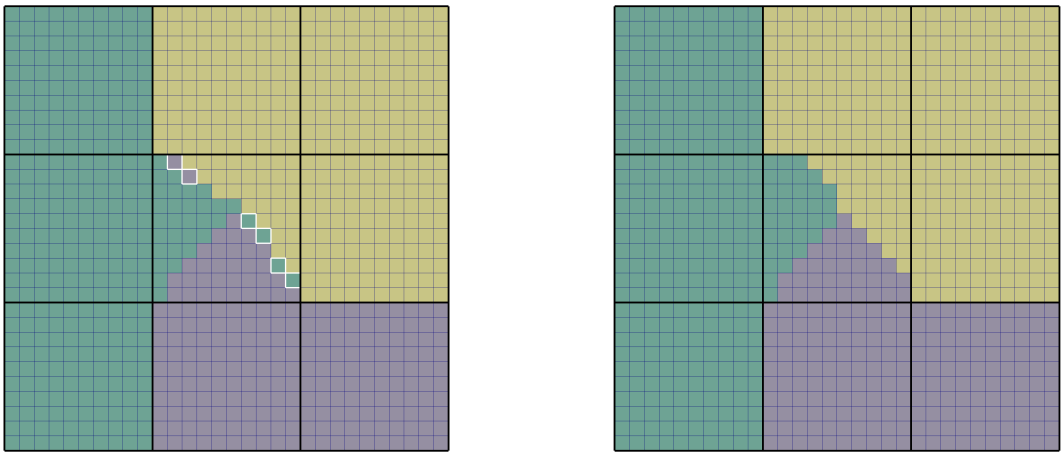
Contrary to the Kernighan-Lin algorithm, the Fiduccia-Mattheyes [Fiduccia and Mattheyses 1982] method only changes the part to which a vertex is assigned instead of swapping two vertices at a time. In particular it means that in the case where the initial partitions are perfectly balanced the algorithm will need to be allowed some wiggle room, i.e. the possibility to increase the imbalance between partitions, to be able to operate. In our problem it translates into our adaptation seen in Algorithm 6 line 6 where a material change for a voxel will only be considered if it does not degrade too much the discrepancy of the coarse cell this voxel is issued from. And again, negative gain moves can be performed, as long as they are compensated afterwards.

Algorithm 6: Fiduccia-Mattheyes.

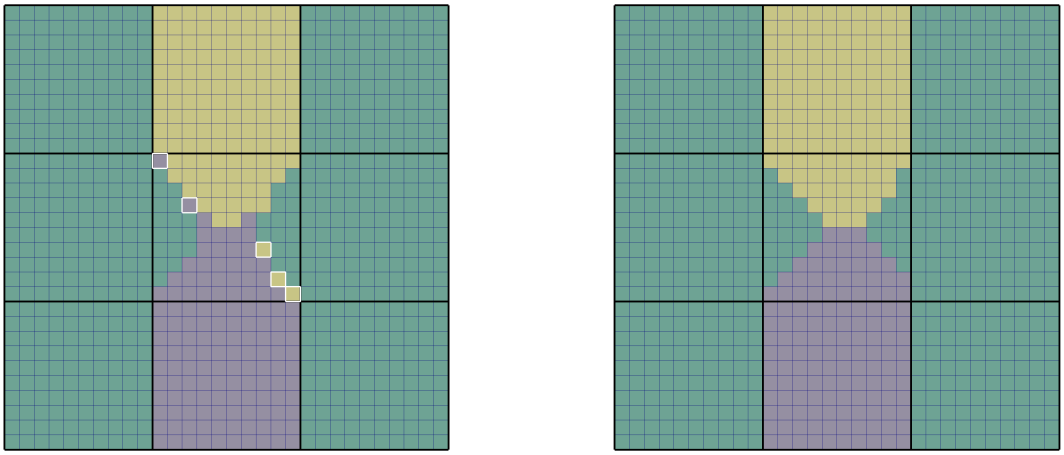
```
1  while gain_cumul > 0 do
2  |  matAssign_tmp ← matAssign
3  |  compute costs for all vertices  /* one cost per material */
4  |  free all vertices
5  |  while ∃ possible material change do
6  |  |  v, m ← find best material change  /* change has to be allowed under volume
   |  |   fractions constraints */
7  |  |  store best material change in sequence
8  |  |  lock(v)
9  |  |  matAssign_tmp(v) ← m
10 |  |  update costs for v and neighbors
11 |  end
12 |  gain_cumul ← find sequence of material changes with maximal cumulative gain
13 |  if gain_cumul > 0 then
14 |  |  matAssign ← execute material changes
15 |  end
16 end
```

In Figure 4.10 is shown the benefits of being able to change the material assignment of the voxels – as is done in the Fiduccia-Mattheyes algorithm – instead of only proceeding by swapping in unstructured cases. The Kernighan-Lin implementation greatly reduces the edgecut at the cost of increasing the discrepancy, while the Fiduccia-Mattheyes manages to reduce both, albeit a little less concerning the edgecut.

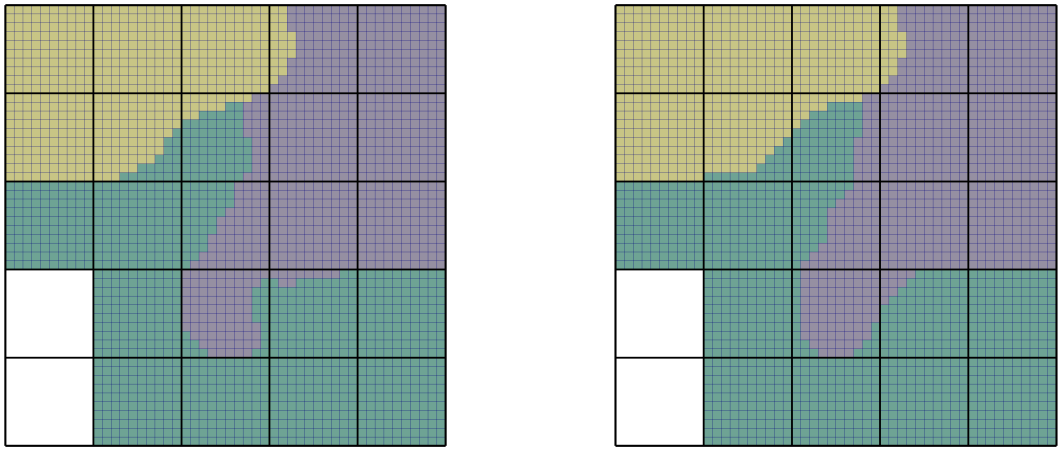
The experiments that we have presented here show that formulating the problem as a graph or mesh partitioning problem and using techniques coming from this field can be worthwhile in extracting better discrete interfaces. The relationship between the coarse cells that carry volume fractions and their spawned voxels implies that compared to classic algorithms additional constraints must be enforced.



(a) e=186 (b) e=144

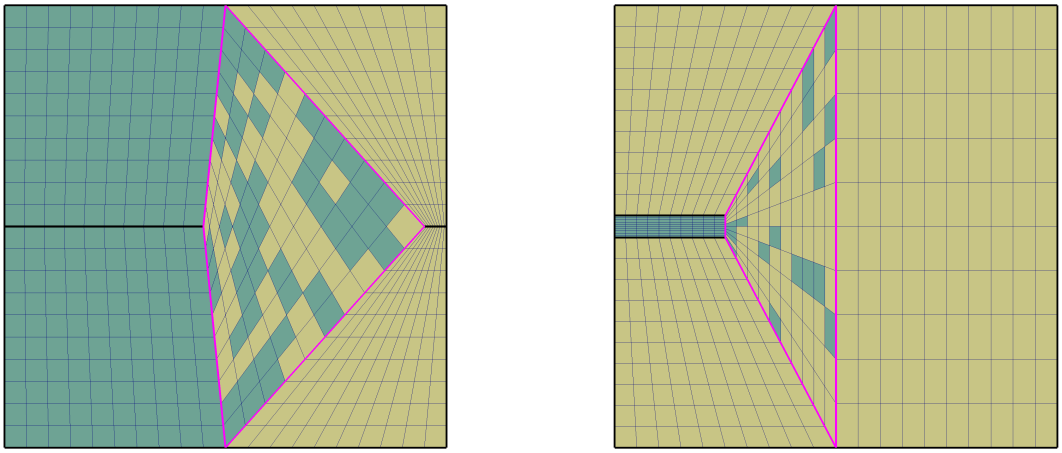


(c) e=222 (d) e=178



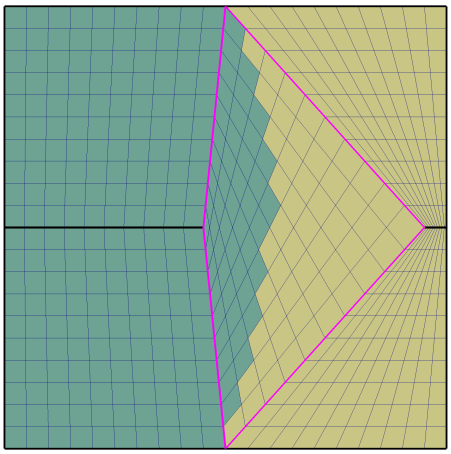
(e) e=568 (f) e=532

Figure 4.9: KL algorithm (right) applied after the greedy heuristic (left). The edgecut is reduced in all the cases, in particular the algorithm changed the assignment of the highlighted isolated voxels.

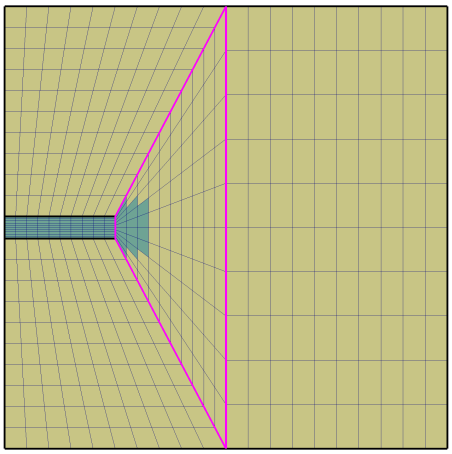


(a) d=0.048, e=444

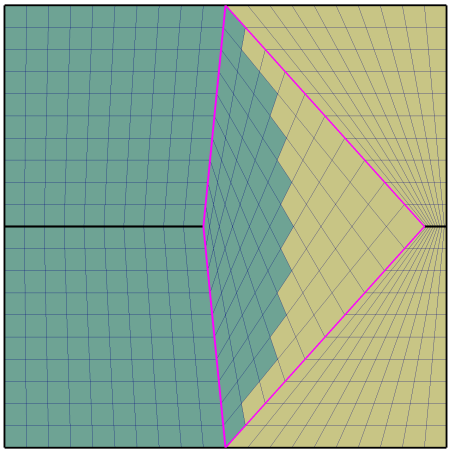
(b) d=0.0038, e=276



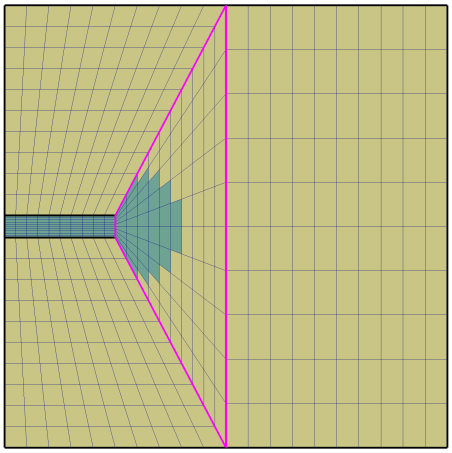
(c) d=0.264, e=84



(d) d=0.1444, e=76



(e) d=0.0128, e=86



(f) d=0.001, e=96

Figure 4.10: Example of re-partitioning applied on two unstructured cases that shows the limits of only swapping the assignments (Kernighan-Lin) instead of changing the assignment (Fiduccia-Mattheyes). (a and b) the initial random assignment similar to the one before applying the simulated annealing; (c and d) after applying the Kernighan-Lin algorithm, where we can see that while the edgecut is reduced, the discrepancy increases; (e and f) after applying the Fiduccia-Mattheyes, which reduces both.

4.2 Geometrical model projection

Starting from our input mesh carrying volume fractions, we have built a finer submesh made of cells, which we called "voxels" and assigned those to materials; we now get the opportunity to extract the interfaces between materials from which we can build an explicit geometrical model. This model can be built at two different levels of discretization (see Figure 4.11): either on the fine mesh made of voxels (top row) or on the coarse mesh (bottom row). By construction, the first one provides high-fidelity to the reconstructed interfaces and preserves material volumes, while the second one is coarser, making it easier to handle (smaller memory footprint, easier to visualize and to use for mesh-to-geometry projection and smoothing). Considering our goal of getting a pure full-hexahedral mesh starting from an Eulerian mesh, we want to build the coarser model.

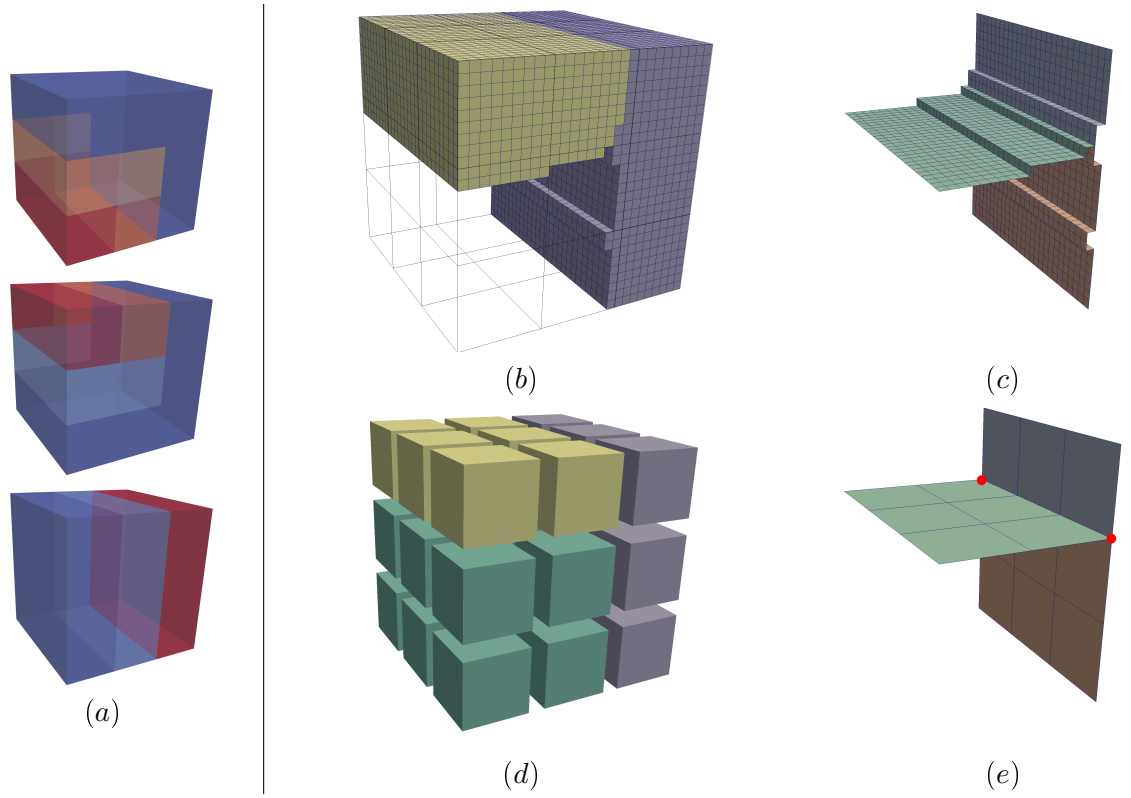


Figure 4.11: Example of explicit geometrical models built from a 3 materials case in a $3 \times 3 \times 3$ grid. (a) volume fractions for each of the three materials, ranging from 0 in blue to 1 in red; (b and c) the voxels assignment and its corresponding geometrical model; (d and e) the same after the assignment of the coarse cells.

Both models (the finest and the coarsest) can be built by first extracting the faces – the edges in 2D – between cells assigned to different materials (see Figure 4.11-c and e), then building a geometrical model $G = (S, C, V)$. Starting from an hexahedral mesh $M = (H, Q, E, N)$, where H are hexahedral cells, Q are quadrilateral faces, E are edges and N are nodes, G can be extracted using the following rules:

- A **multi-surface** of S is a set of faces of Q that are adjacent to the same 2 materials. We get 3 such distinct multi-surfaces in the example on Figure 4.11;
- A **multi-curve** of C is defined as a set of edges bounding the quad of a surface $s \in S$. Considering all the faces forming s, we get the set of edges $E_s \subseteq E$ that bounds those faces. This set of edges is then partitioned into multi-curves as follows: two edges of E_s are assigned to the same multi-curve if they are adjacent to the same set of materials assigned to the cells. For instance, let us consider the green surface on Figure 4.11-c and e; this surface is bounded by 2 curves: the first one corresponds to the intersection between the three surfaces – all the edges are then adjacent to exactly the 3 same materials; the second one is made of the remaining edges, which are adjacent to only 2 materials and located on the boundary of the domain – here the bounding box of the input grid;

- A **multi-vertex** of V corresponds to all the nodes of N that are adjacent to the same multi-curves in C, or in other words, to the same set of materials assigned to the cells of H. Considering the example of Figure 4.11, the two nodes highlighted in (e) define a single multi-vertex.

Characterizing entities of the geometrical models using the material assignment of adjacent cells leads to form **multi**-entities that are potentially non-connected – in particular a vertex can have several spatial locations. For our purpose, this definition is sufficient and we do not further differentiate by splitting those entities into connected parts. In the example of Figure 4.11, the geometrical models extracted are both made of 3 multi-surfaces, 4 multi-curves and 1 multi-vertex with two positions.

Now that we have those two models we will draw the correspondence between them and adapt the coarser representation by constraining it onto the finer model as seen in Figure 4.12. Considering a fine model G_f and a coarse model G_c extracted from compatible data (see Figure 4.12-a and b), multi-entities of G_f and G_c are defined from the same set of materials and so can be identified and associated through this material correspondence. In Figure 4.12, both models are superposed on the left while on the right only the multi-entities of G_c are represented. To adapt G_c to fit G_f as best as possible, we first project each node of the coarse mesh that corresponds to a multi-entity of G_c onto the corresponding multi-entity of G_f (see Figure 4.12-e and f), then we smooth the node positions while keeping them projected onto the corresponding multi-entity of G_f (see Figure 4.12-g and h).

The proposed solution has been widely used on several examples including those shown on Figure 4.13 where the coarse and fine models are similar, and some others on Figure 4.14 where they differ. Those examples are built using a testbed program that assign volume fraction of material on a simple sample grid. This program is used for doing unit-testing and non-regression verifications. Note that the topological differences between the coarse and fine model, that we encounter in Figure 4.14, does not prevent our approach to get a result. However, it can be an issue in the meaning that the materials are not consistent between the two levels of resolution. This is why in Section 5.2 we will present a strategy that addresses this issue by use of mesh refinement and also by use of the voxels material assignment that was presented in this section.

We have also applied our approach on more realistic data, such as the result of a CFD simulation case that is shown in Figure 4.15 where our input is a grid mesh carrying the volume fractions at $t = 1sec$ and $t = 2sec$ of the simulation.

Chapter digest – *In this chapter we designed, implemented and compared several techniques over the extraction of discrete interfaces. We have shown that having an unstructured mesh heavily influences the choice of the method one may use, and we have demonstrated that applying post-processes inspired from graph re-partitioning algorithms can correct some artifacts in the voxels assignment when they occur. We have also demonstrated how a geometrical model built from those interfaces can be used to constrain the model built from the coarse mesh.*

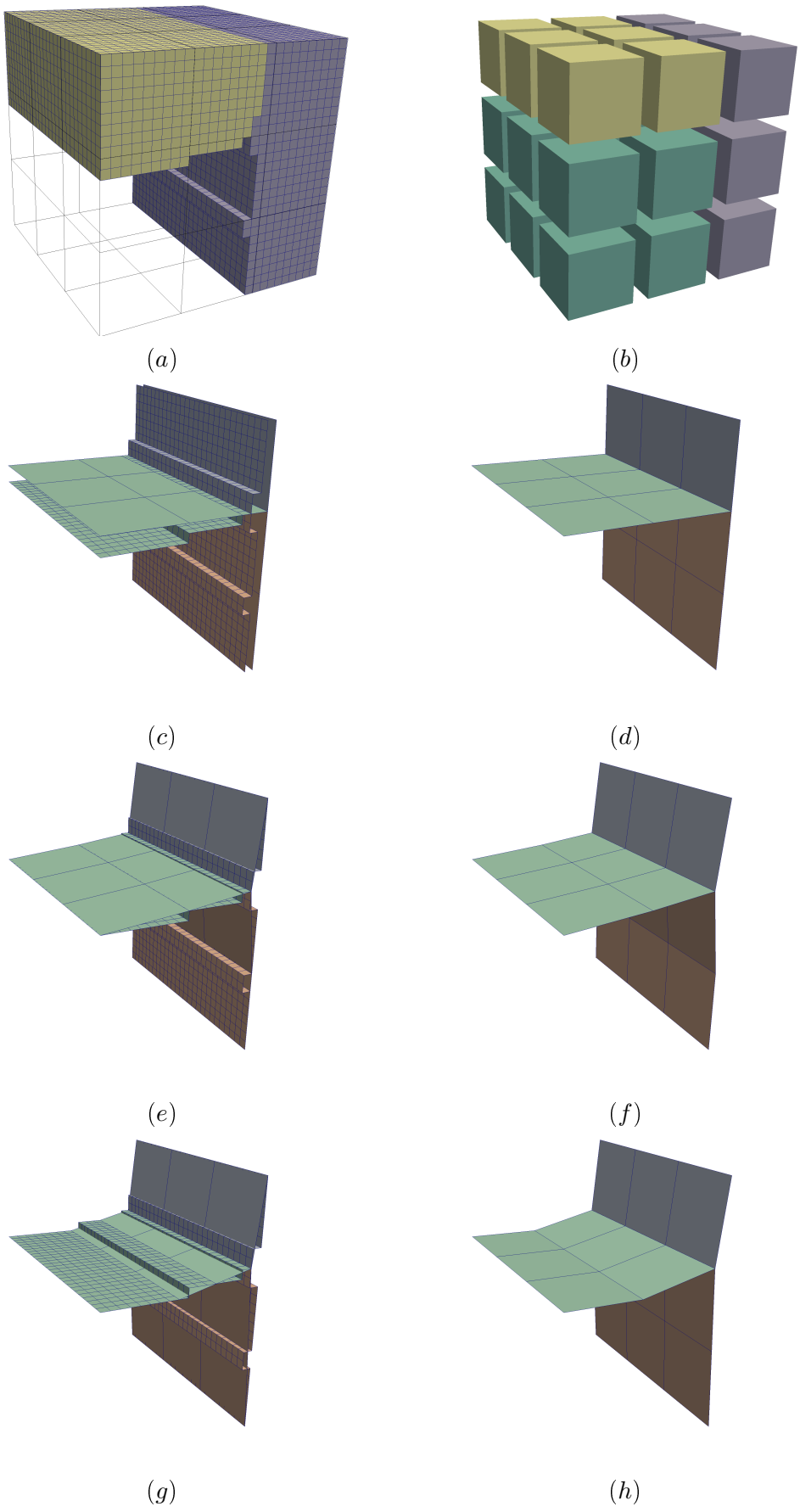


Figure 4.12: Illustration of how the finer geometrical model is used as a support to project and smooth the coarser geometrical model. (c and d) the initial models; (e and f) the coarser model is projected onto the finer one; (g and h) the coarser model is smoothed while being constrained.

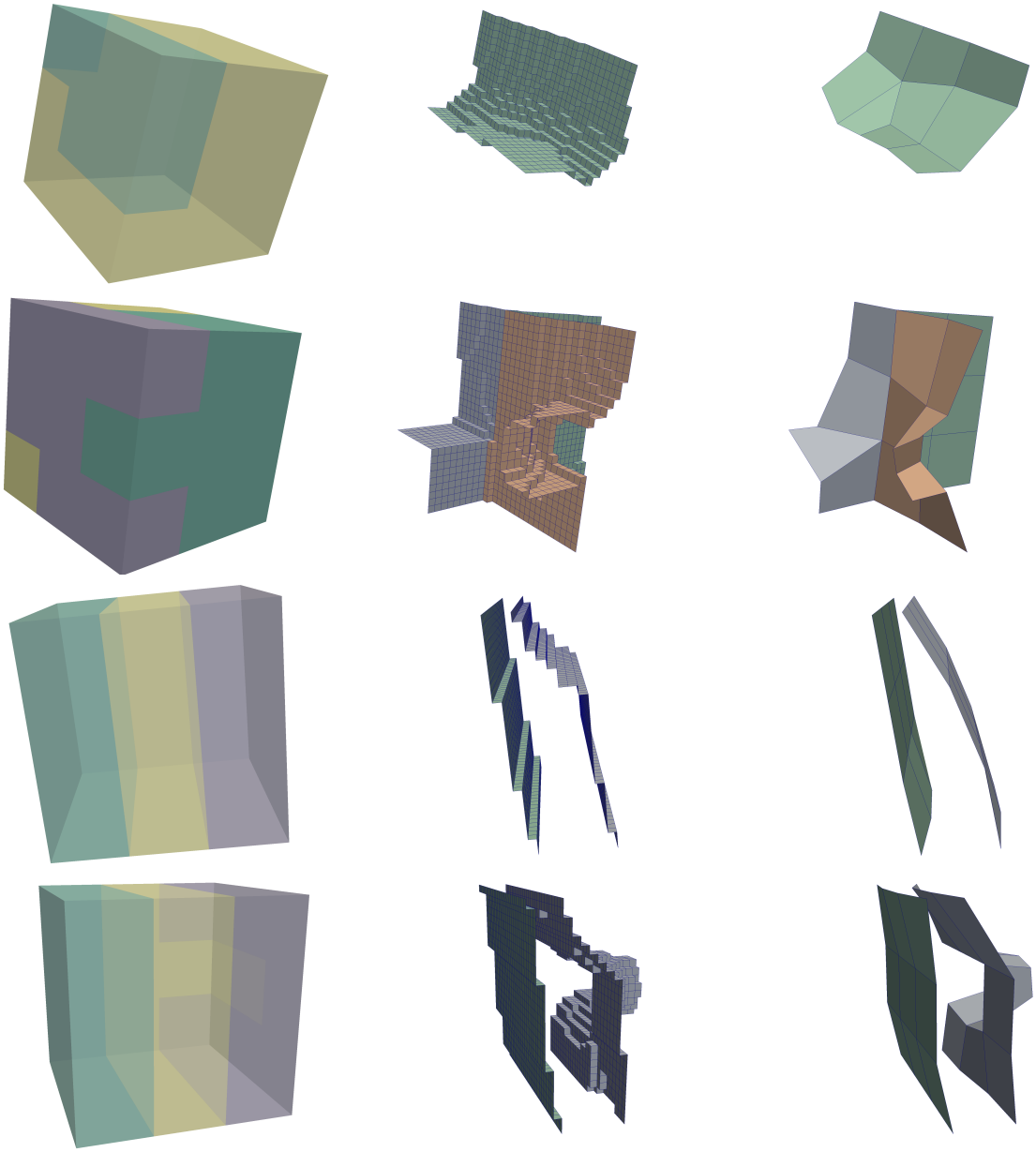


Figure 4.13: Examples where the finer models (middle) are similar to the coarser models (right). On the left are the coarser meshes after material assignment.

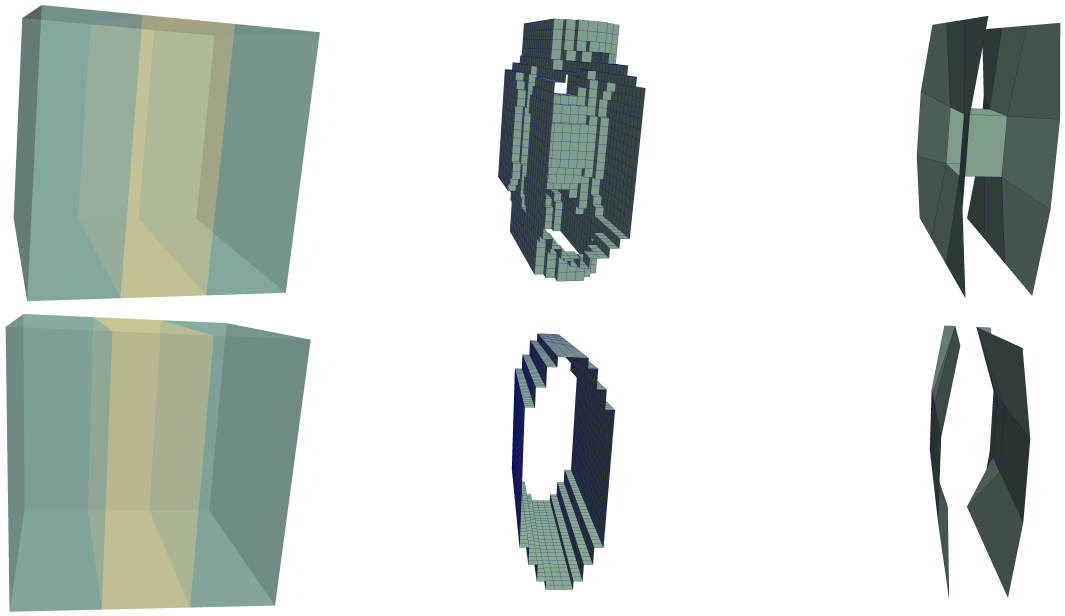


Figure 4.14: Examples where the finer models (middle) differ from to the coarser models (right). On the left are the coarser meshes after material assignment.

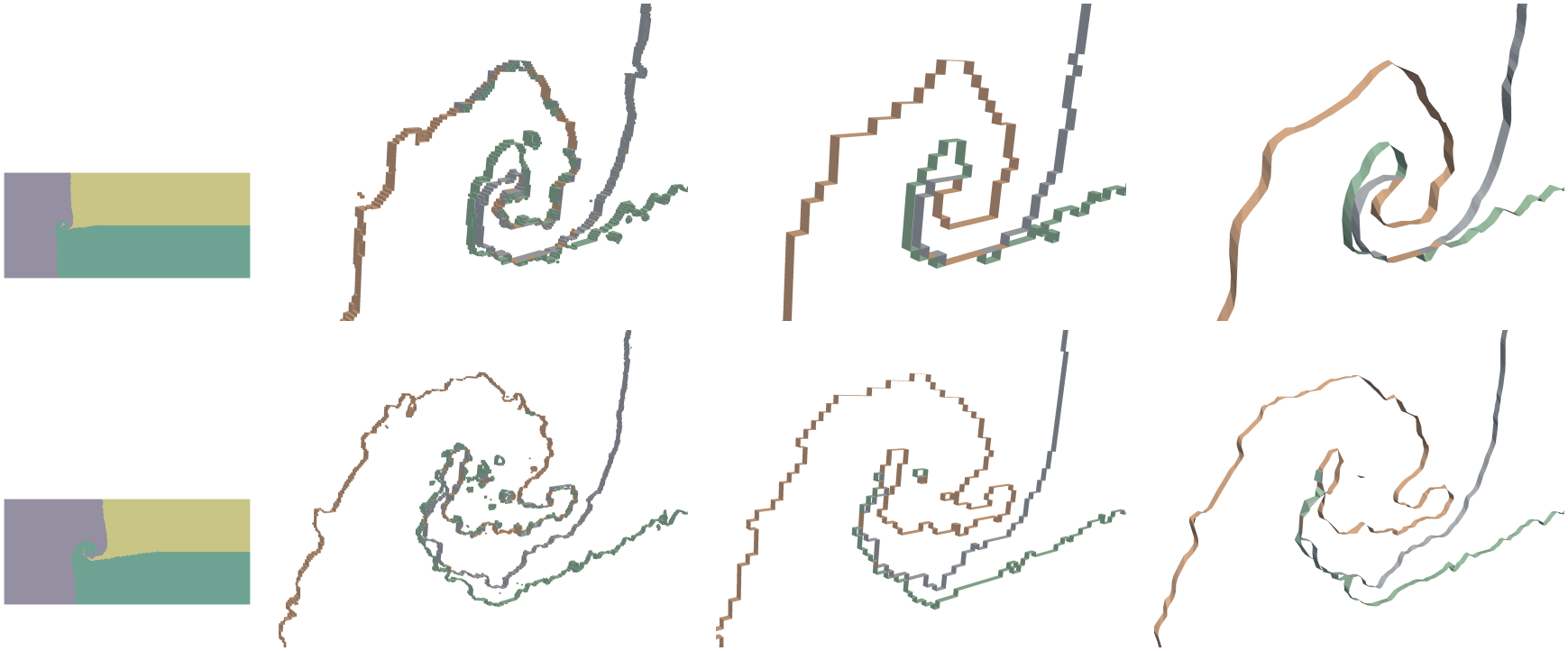


Figure 4.15: Coarse geometrical model (middle) projected and smoothed (right) onto the voxelated one (left) in the triple point problem at $t = 1sec$ (top) and $t = 2sec$ (bottom).

Chapter 5

Guaranteed quality and topological operations

In our context, intercode usage means to generate a Lagrangian mesh M_L with smooth surfaces from an Eulerian mesh M_E while preserving as best as possible the geometry and the volume fractions of M_E locally to each cell of M_E (see the discrepancy definition in Section 3.1, page 40). In practice, getting an acceptable result, that is to say a mesh M_L that fits previous requirements and that provides cells with good-enough quality for the simulation code, requires to perform topological mesh modifications. Such modifications are performed at two levels in the ELG pipeline (see Figure 5.1).

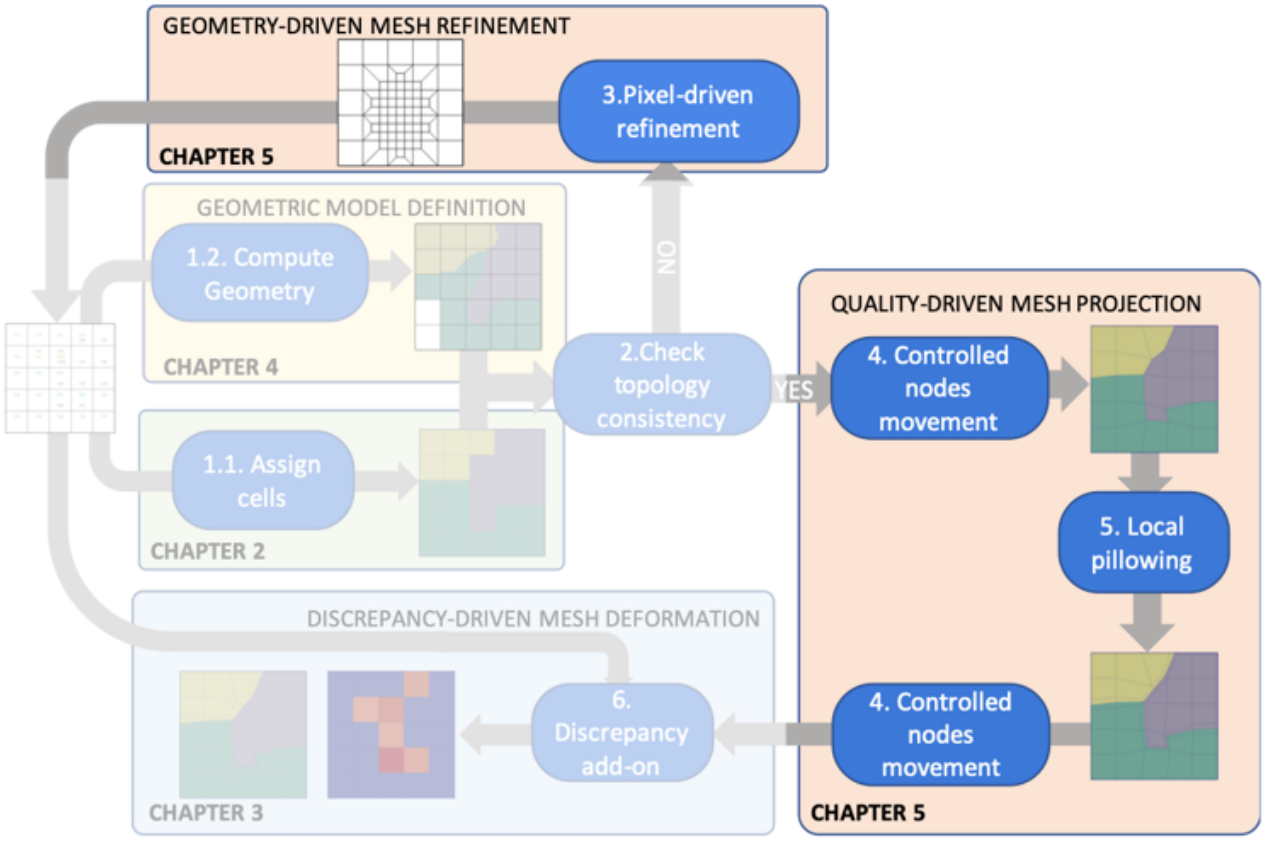
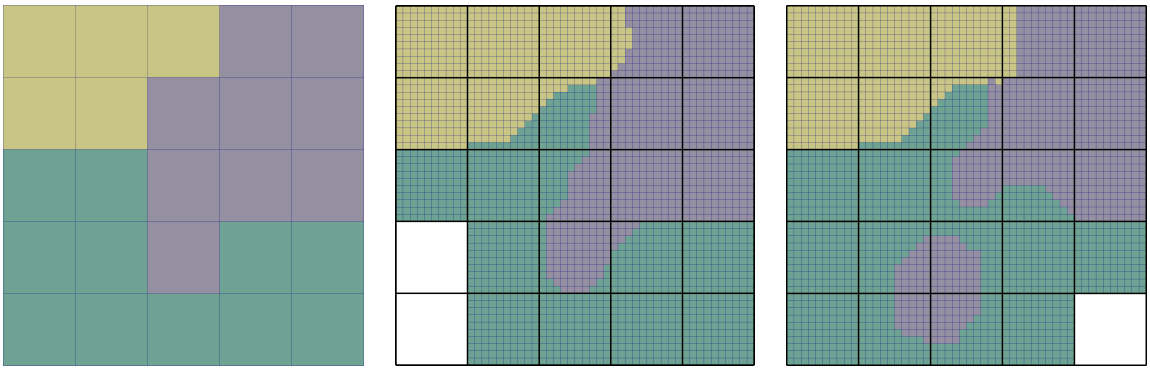


Figure 5.1: Performing topological modifications is required to get a final mesh that both fits the geometrical model - stage 3 on top - and allows us to get a mesh usable by a simulation code - stages 4 and 5 on the right.

An example of topological issues we could meet occurs on the example of Figure 5.2, where cells of M_E are assigned to 3 materials in (a). In (b) a voxel-based geometrical model gives a consistent geometrical model: each material is connected similarly to (a) and located in the same area. On the contrary, in (c), the grey material is made of 2 connected components and the green one has a hole. In the latter situation, if necessary classic overlay grid methods can adapt their mesh to capture overlooked details, but they do have the CAD model on hand. We will explore the possibilities that we

have in Section 5.2; it corresponds to the stage 3 of the ELG pipeline when the topology consistency is not obtained (see Figure 5.1).



(a) (b) $c)$

Figure 5.2: Mesh assignement and geometry definition can provide a compatible decomposition - between (a) and (b) - or an incompatible one - between (a) and (c). Volume fractions slightly differ between (b) and (c), but in both cases the material assignment on the coarse mesh is (a).

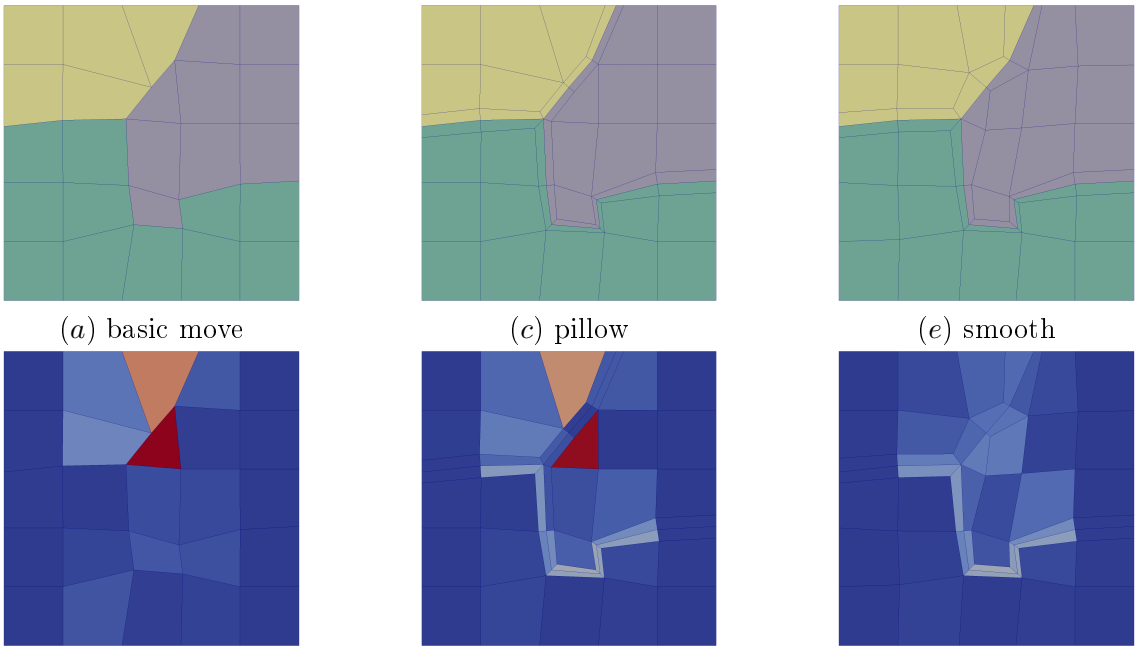
In Chapter 3 we have evaluated and improved, by means of a post-process, our ELG pipeline and the SCULPT procedure. Several limitations were identified and discussed, among which the fact that since we rely on computing the volume of intersection between cells, not having inverted cells is a prerequisite to our post-process, and more importantly, to the simulation code. In some cases, we did not manage to obtain such a mesh using SCULPT, particularly in cases issued from hydrodynamics numerical simulations. The basic overlay grid pipeline (see Figures 5.3 and 5.4) does not consider the cell quality until the very last smoothing step; basically in such an algorithm we move the nodes (see Figure 5.3.a and b), change the mesh topology (see Figure 5.3.c and d) and hope for the best (see Figure 5.3.e and f) with the smoother. In our context, where our input can be the output of a simulation code, we have shown in [Le Goff, Ledoux, and Owen 2018; Le Goff, Ledoux, Janodet, and Owen 2019] that depending on the mesh resolution the code ran with, we can end up with a good quality mesh (no inverted cells) or a bad quality one. It is unrealistic to ask engineers to run their simulations again with different resolutions at random, assuming it is even feasible. This exhibits the need for an algorithm that consistently works. We proposed in [Le Goff, Ledoux, Janodet, and Owen 2019] an alternative pipeline to apply, where we control the mesh quality at each step (see Figure 5.4). More specifically, enforcing a quality control led us to act on three stages (see the right side of Figure 5.1):

1. **Node movement:** instead of simply moving the nodes to their computed position, we iteratively move them and stop if the adjacent cells quality drops below a user defined threshold (see Section 5.1.1). One side effect is that our nodes can end up not at their expected position;

2. **Node position computation:** altering the node movement shows the need to take into account the quality of material interfaces, particularly in 3D where the previous nodes computed locations (done similarly to [Owen, M. L. Staten, and Sorensen 2012]) could lead to low quality quadrangles on the material interfaces (which is a 3D surface). It means that by design the interface nodes can never reach their destination if the nodes are moved with our controlled movement. We propose to modify the computed locations by projecting and smoothing the expected interfaces onto a geometrical model built following the techniques we have implemented in the previous Chapter 4;

3. **Pillowing:** we no longer apply a global pillowing for each material, which we show can have adverse effects, and instead favor a localized one. We apply our geometrical and topological mesh modifications inside submeshes, or cavities, that we insert back into the mesh if the cavity quality meets the requirements (see Section 5.1.2).

In the first part of this chapter, in Section 5.1, we detail the process involved in the "quality-driven mesh projection" stage. While the method can still be improved, it fulfills the goal of keeping the cell quality above a user-specified threshold.



(a) basic move (c) pillow (e) smooth

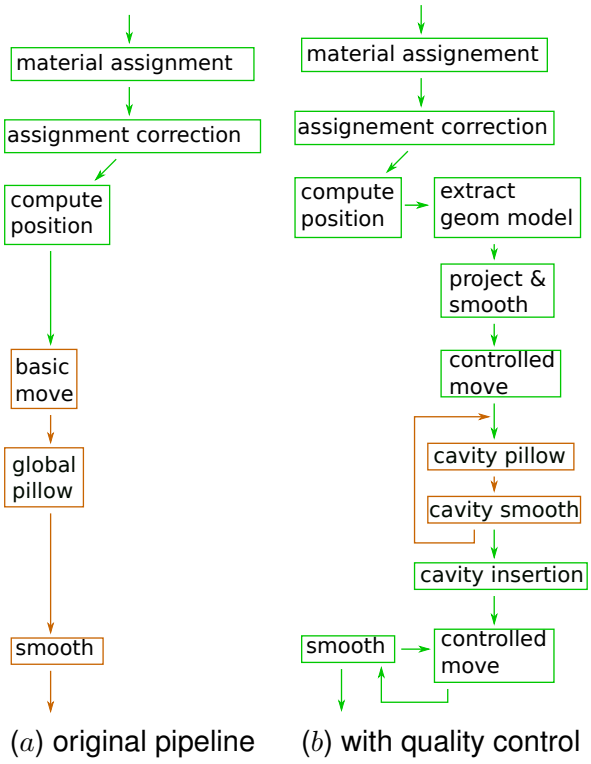
(b) minSJ $= 0.036$ (d) minSJ $= 0.046$ (f) minSJ $= 0.588$

Figure 5.3: Evolution of the cell quality during the base algorithm. In (a) and (b) the node movement causes a sharp decrease of the cell quality; in (c) and (d) the pillowing does not really improve the cell quality; in (e) and (f) the smoothing is efficient in this example.

5.1 Quality-driven mesh projection

In this section, we try and explain the "*quality-driven mesh projection*" stage that appears in the ELG pipeline. As previously said, the aim of this stage is to adapt the mesh to fit the geometry as best as possible under the constraint of getting a mesh that can be used by a Lagrangian simulation code. In order to get such a mesh, the most important constraint to satisfy is to ensure that each cell of the resulting mesh is above a minimal quality threshold, prescribed by the Lagrangian code. In Figure 5.4 on the right, we detail the "*quality-driven mesh projection*" stage and we highlight the parts of the algorithm that keep the mesh quality under control (green boxes) and the ones that do not (orange boxes).

As the two main algorithms of this stage are the node movement and the local pillowing operations, we first show how we try and control mesh quality when moving material interface nodes, and after we introduce the local pillowing operation before showing obtained results in 2D and 3D.



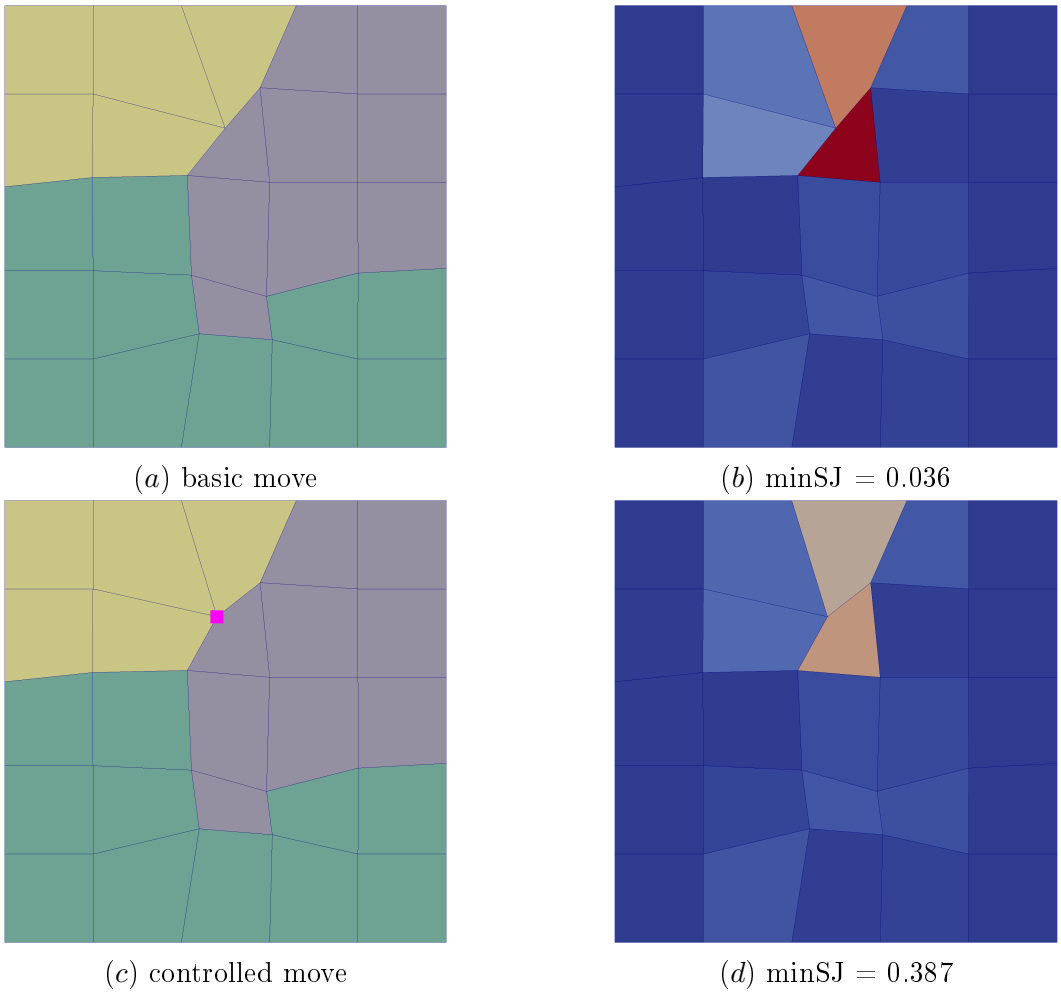
(a) original pipeline (b) with quality control

Figure 5.4: In green the part where we have a guaranteed cell quality, and in orange the part where there is no control over this.

5.1.1 Guarantee by controlled node movement

As seen in Figure 5.3-b, the first phase that may decrease quality is the basic node movement.

Considering that the input mesh meets the quality requirements, our strategy is to avoid moving the nodes when the quality is degraded below a prescribed threshold. For that purpose, we progressively move each interface node n towards their expected location $\mathbf{p}_{ideal}$ on the computed interfaces. At each small movement of n, we check the quality of cells surrounding n. In our implementation, a small movement corresponds to $1/16^{th}$[1] of the distance between the location of n and $\mathbf{p}_{ideal}$. The impact of this controlled movement can be seen in Figure 5.5, and the guarantee over the quality now extends and reaches just before the pillowing stage (see Figure 5.4).



(a) basic move (b) minSJ $= 0.036$

(c) controlled move (d) minSJ $= 0.387$

Figure 5.5: Example of controlled movement. (a) and (b) basic move and the corresponding mesh quality; (c) and (d) controlled move, the marked node could not move all the way and was stopped at a distance of 0.118 from its expected location and the cell quality remains above 0.3.

A direct side effect of this controlled movement is that some of the nodes do not reach their expected position. Let $\mathcal{N}$ be those nodes, we introduce an additional modification to the base algorithm to allow those nodes to move further. After applying the topological modifications (the pillowing), the algorithm enters into a move-smooth loop (see Figure 5.4) so that the nodes of $\mathcal{N}$ can progressively keep moving towards the geometrical interfaces, eventually reaching it in the best cases. In this move-smooth loop, the smoothing stage has two prerequisites: it must not decrease the quality, and the nodes that have reached the geometrical interfaces are fixed. At the end of this stage, the set $\mathcal{N}$ is not necessarily empty, meaning that there could remain nodes that did not reach their expected destination.

So far we have tried to move the nodes towards a location computed using the input volume fractions and the cells material assignment, but it considers each nodes independently and not the interfaces as a whole; in particular no care was taken for the expected interfaces quality. It becomes especially

[1] This fraction was chosen arbitrarily. Directly trying to move to the expected positions and reverting the position of nodes with adjacent faces that became of bad quality had the effect that a lot of nodes stayed in place.

relevant in 3D where the mesh entities forming the interfaces are no longer 1-cells (or edges) but are now 2-cells (or faces). Moving the nodes to their computed ideal location can by-design lead to bad quality faces, hence severely limiting our nodes controlled movement. Figure 5.6-a illustrates it in the asteroid case, where the surface mesh we would obtain by moving the interface nodes at their expected locations has very bad-quality quadrilateral elements. This causes our algorithm to be stuck after the first controlled move, with a mesh still fairly stair-shaped (see Figure 5.6-b). Such a resulting mesh could be considered satisfactory, quality-wise, still we want the interface nodes to be located as close as possible to where the material interfaces were determined to be.

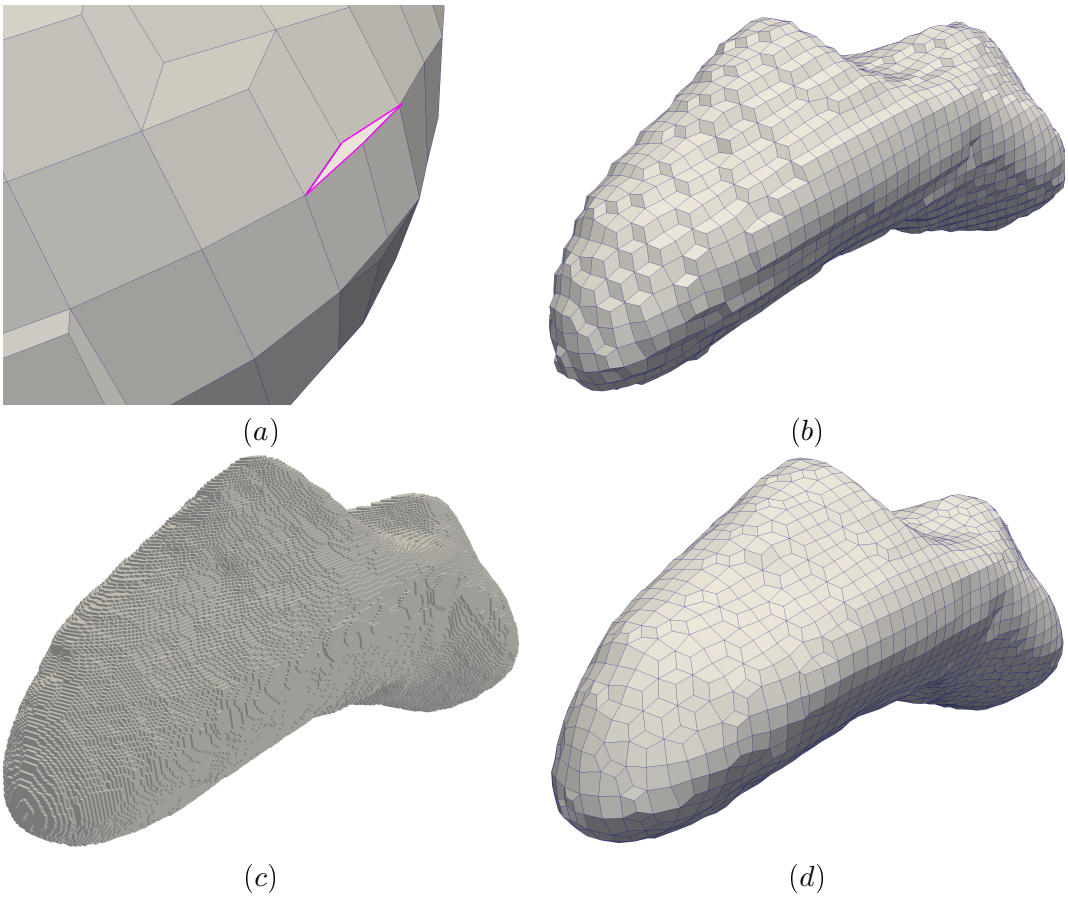
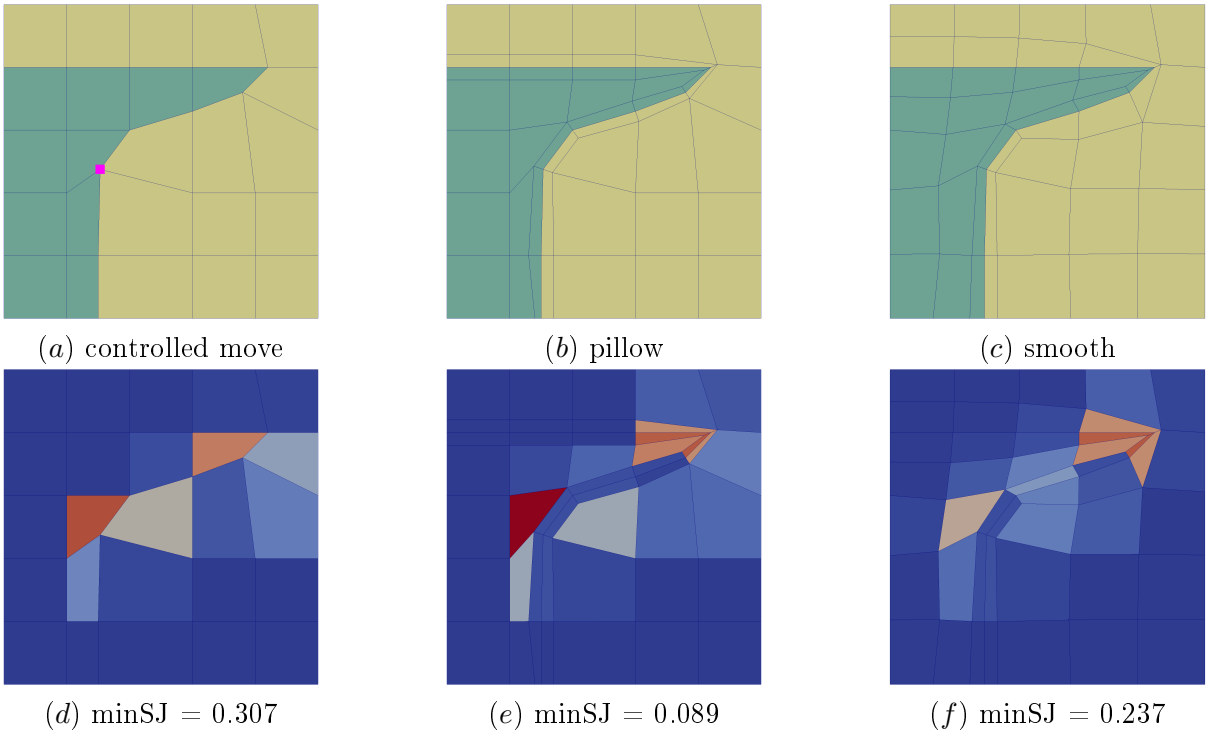


Figure 5.6: Motivation behind the geometrical model extraction and projection smoothing illustrated in the asteroid case in 3D. (a) close-up of the expected surfacic interface mesh where the marked quad has a low quality of 0.068; (b) the mesh after the first controlled move towards the original computed position. We can see that the cells assigned to the asteroid are still fairly stair-shaped, and our algorithm can not go further because by-design we want to move the nodes to bad positions; (c) the voxelated interface between the asteroid and the exterior as obtained using the techniques we developed in Chapter 4 and on which we will smooth the expected interfaces to remove problems such as those seen in (a); (d) the coarse geometrical model obtained using the techniques of the same chapter; it replaces (a) as the new expected positions of the interface nodes.

In order to counteract this we modify the computed nodes locations by employing the methods we introduced in Chapter 4. As the original interface nodes expected positions form a surface with bad faces, we want to apply a surfacic smoothing to avoid those. So as not to stray too far from the input volume fractions data and reduce the impact on the volume preservation of the materials, the positions are constrained on a geometrical model, which in our case is the model that can be extracted from interfaces reconstructed (see Figure 5.6-c) as we have discussed in Section 4.1. This is in fact the technique we introduced in Section 4.2, and it leads to the coarse surfacic model shown in Figure 5.6-d), which becomes the new expected positions for our interface nodes.

5.1.2 Cavity pillowing

In the original pipeline, as in many other overlay grid methods, a pillowing phase is applied, where each material is wholly pillowed without taking cell quality into account, and eventually a final global



(a) controlled move (b) pillow (c) smooth

(d) minSJ = 0.307 (e) minSJ = 0.089 (f) minSJ = 0.237

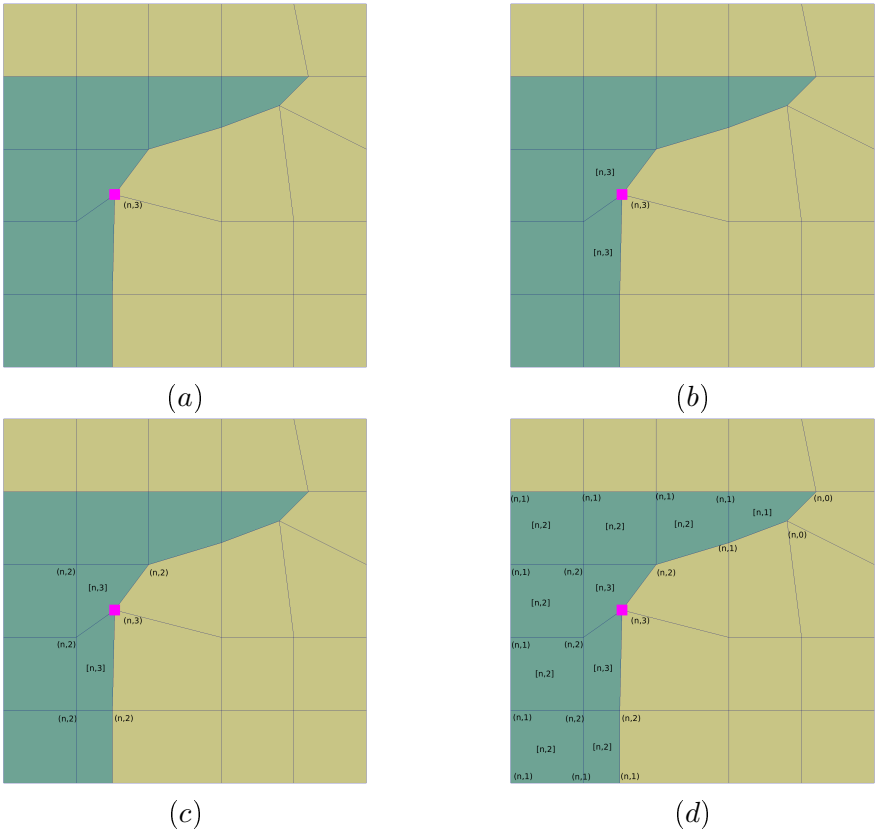
Figure 5.7: Global pillowing quality problem on a 5x5 two materials case. (a and d) mesh and its cell quality after the controlled move with the marked node stopped at a distance of 0.04 from its desired position; (c and d) global pillowing; (e and f) the smoothing makes good use of the additional edge added to the marked node in the green material area to improve quality but cannot improve it at the tip.

smoothing stage is executed so as to improve the overall cell quality. The intermediate pillowing phase usually does not in itself improves quality, but it provides more degrees of freedom for the smoothing algorithm to work with. The example of Figure 5.3 illustrates that such a pipeline can lead to good results. While the cell quality is not strongly controlled, the mesh quality is improved during the process.

The example of Figure 5.7 shows the exact opposite. The mesh quality worsens because of a global pillowing technique that does not take into account some local geometrical features. In this case, performing a pillowing around the right tip of the green area leads to strongly decreasing the inner angle of each quadrilateral cell around this node. Note that the worst cell after proceeding with the pillowing (in red in Figure 5.7-e) was the one hampering the displacement of the marked node. The subsequent smoothing applied does improve the situation (it goes up from 0.089 to 0.237) but the quality of the worst cell is eventually lower than at the beginning (0.237 against 0.307) before applying the pillowing. Note that the worst cells are now the two newly created ones at the tip. In [Cherchi et al. 2019] the authors seek to avoid a global pillowing and improve mesh quality by first identifying a hard set of faces that they want padded, extend this set so as to connect the patches they form and solve a binary problem they formulated to obtain a valid set of faces for sheet insertion.

In the ELG pipeline, we avoid indiscriminately performing a global pillowing by following a new procedure. It aims to help moving nodes of $\mathcal{N}$ by providing more leeway for the smoothing algorithm to work with. The idea is to apply pillowing operations in the vicinity of $\mathcal{N}$ while avoiding to change the mesh topology where not necessary, that is to say where the quality is already good (higher than the threshold) and interface nodes reached their destination. The process we propose can be summarized as follows: for each node n of $\mathcal{N}$ and each material m adjacent to n, we extract cell groups that we are going to pillow. Each cell group is called a cavity. In theory, the idea would be to build many sets of cavities $\{c_i\}_{i>0}$ for (n, m), pillow and smooth each cavity c_i independently and keep the one that gives the best quality. If this quality is higher than the quality threshold S_q then the pillowing of this "best" quality is plugged in the mesh.

As there is a huge number of potential cavities, we consider in practice a maximal distance to the node n to build cavities. This distance corresponds to a node-based traversal of the mesh starting from n. For instance, for distance 1, the cavity of (n, m) contains all the cells of material m that are



(a) (b) (c) (d)

Figure 5.8: Cavity definition for the green material. (a) the node n that could not reach its destination is marked and assigned a range (here it is 3); (b) the adjacent cells are also selected and a reference to n is kept; (c) the selection is extended to the adjacent nodes, a reference to n is kept and the range is decreased; (d) we continue until the range reaches zero or all the cells assigned to the material are selected.

adjacent to n. In all our examples, a maximal distance of 3 is chosen. As an illustration, Figure 5.8 shows our process to create cavities when $\mathcal{N}$ is reduced to a single node. Our implementation choice is to have a greedy approach where we do not perform the pillowing for all the cavities; for a couple (n, m), we publish the first one that provides a quality above S_q. This greedy algorithm is done by progressively decreasing the cavity distance, starting with 3, until we obtain a satisfactory mesh in the cavity. We chose this value because the cavities of several nodes of $\mathcal{N}$ are more likely to merge when they are big enough. Such merging reduces the number of topological changes and provides a big enough cavity for the smoother to work with. Figure 5.9 shows how we can end up with potentially several large-size cavities.

As previously said, our greedy algorithm stops when we encounter a situation with a cavity quality that is above S_q. This is illustrated by Figures 5.10 and 5.11. In Figure 5.10, we first extract a cavity of range 3 for the marked node (in a) and then we pillow and smooth it (in b and c). As the quality is below the $0,3$ threshold, we decrease the cavity range to 2 and start a second iteration (in e, f, g and h) where we get the expected quality. In Figure 5.11, we see the local pillowing made on both materials adjacent to the marked node. Our overall algorithm is summarized by Algorithm 7).

5.1.3 Results

In this section, we demonstrate and analyze the results of the proposed method on several cases, both 2D and 3D; the metrics on the results are shown in Table 5.1 where $dist_{init}$ and $dist_{final}$ are defined as the sum of the distances between the interface nodes and their expected position, respectively after the first controlled move and at the end of our algorithm. Admittedly those values heavily depend on the size of the mesh, so comparisons between cases might not be relevant, but the ratio $\frac{dist_{final}}{dist_{init}}$ represents the improvements (the lower the better) our modified pipeline brings to counteract the drawback of the controlled node movement. We also compare our method with *Sculpt* on some examples regarding execution time and a $discrepancy$ metric (see Definition 5 at page 41).

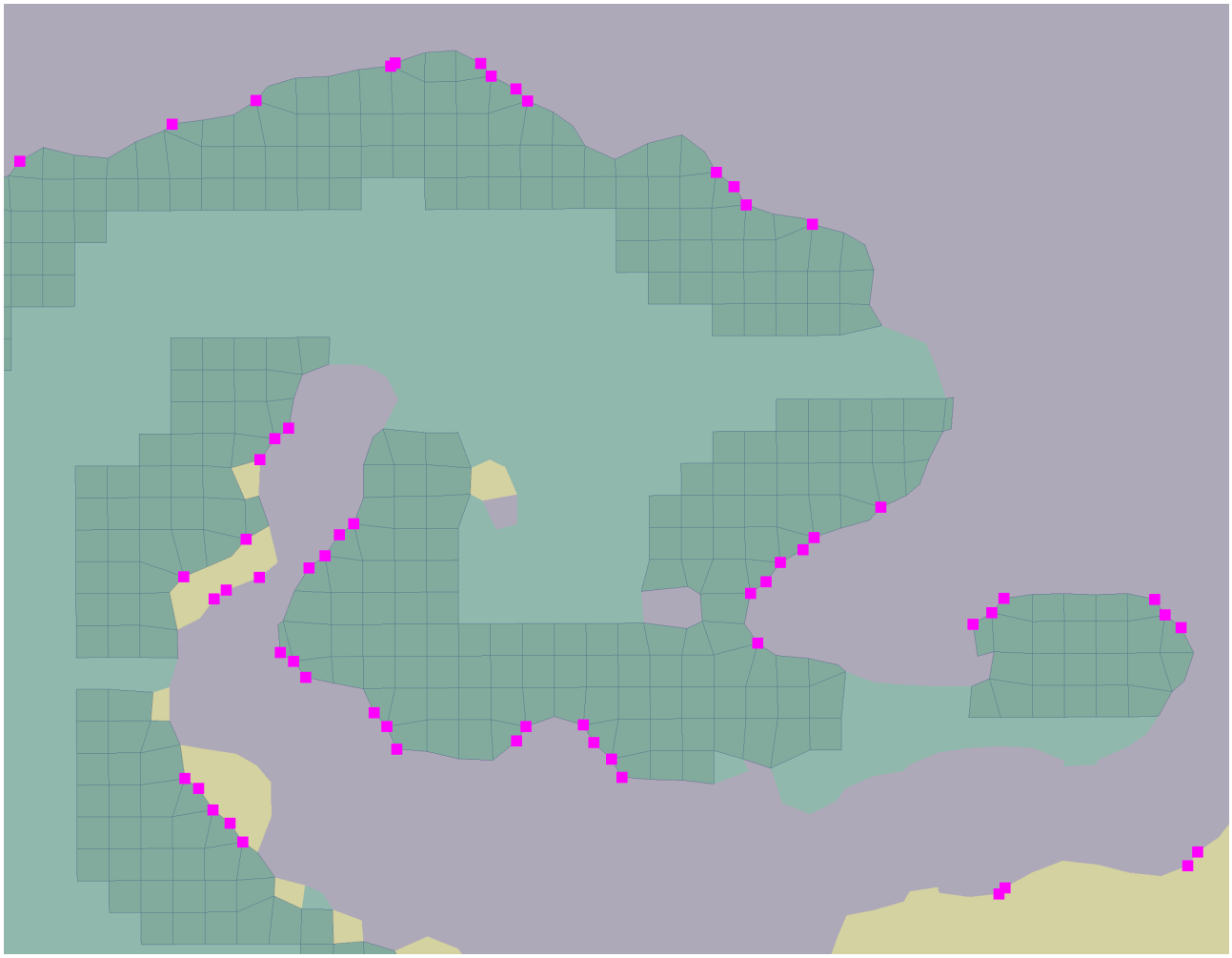
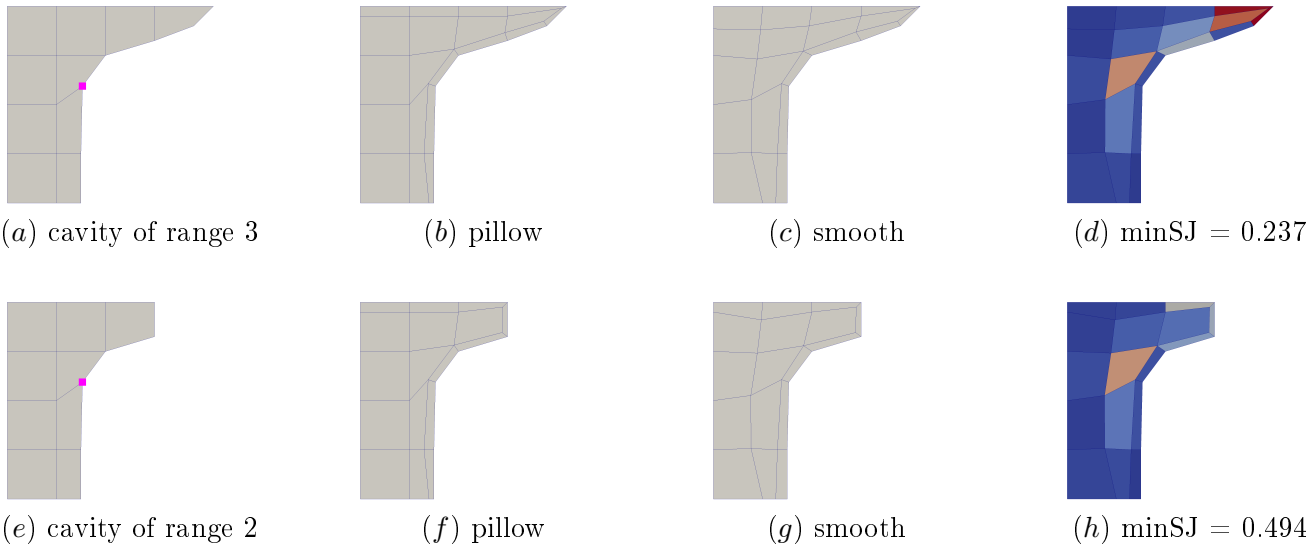


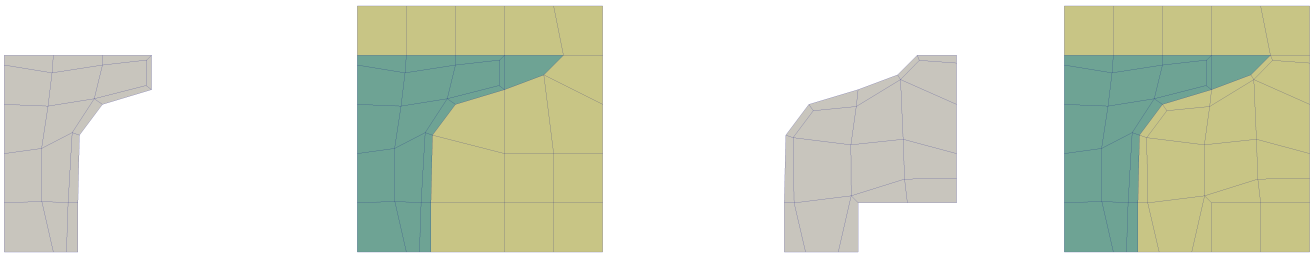
Figure 5.9: Cavity extracted for the green material in the triple point example (see Figure 5.12) where we can see the cavities spawned from different nodes merged to form bigger cavities. Note that marked nodes at the interface between the yellow and grey materials are ignored and do not spawn cavities, as we currently consider one material at a time.



(*a*) cavity of range 3 (*b*) pillow (*c*) smooth (*d*) minSJ = 0.237

(*e*) cavity of range 2 (*f*) pillow (*g*) smooth (*h*) minSJ = 0.494

Figure 5.10: Cavity pillowing and smoothing loop. (*a*, *b*, *c* and *d*) a first iteration with a cavity of size 3 does not give the expected quality; (*e*, *f*, *g* and *h*) a second iteration with size 2 meets the requirements.



(a) (b) (c) (d)

Figure 5.11: Cavity pillowing insertion back into the mesh. (*a* and *c*) the pillowed cavity for the green material and its insertion inside the mesh; (*b* and *d*) the same for the yellow material.

Algorithm 7: Cavity loop.

Data: marked Nodes $\mathcal{N}$
Result: mesh

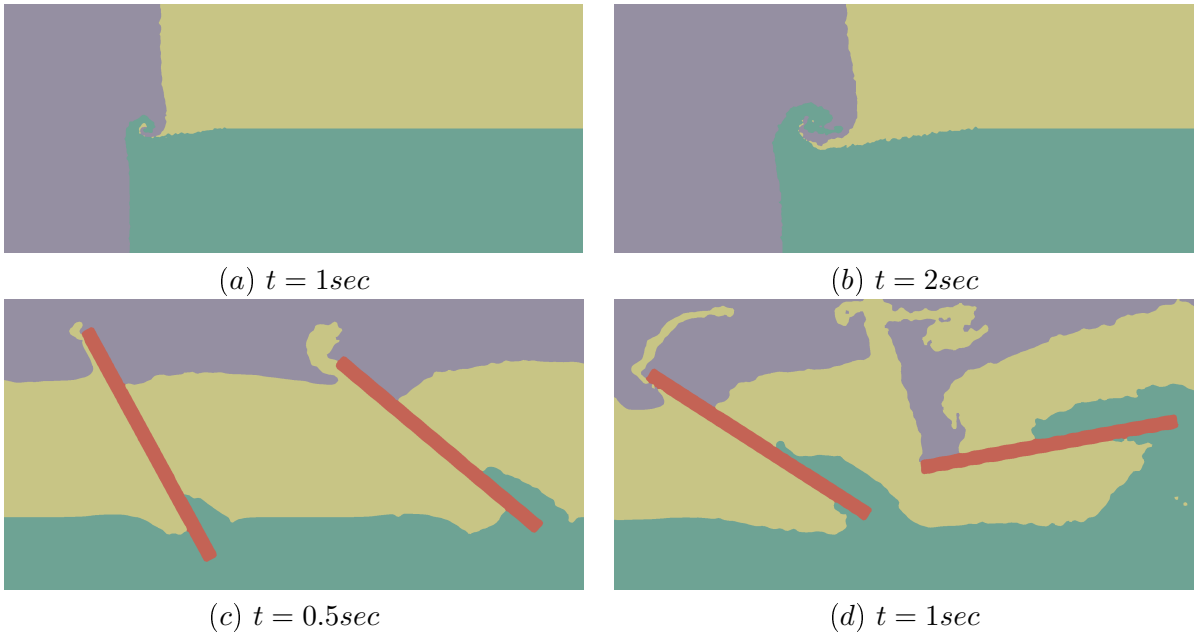
```
1  for m ∈ Materials do
2      /* we initially assign the same user-input range to all the marked nodes */
3      R ← initRange()
4      /* cavity extraction (see Figure 5.8) */
5      cavity ← extractCavity(N, R, m)
6      while cavity ≠ ∅ do
7          /* we will apply modifications to the cavity and reduce it size until it meets the quality
               threshold (see Figure 5.10) */
8          cavity.pillow()
9          cavity.smooth()
10         qual ← cavity.getQuality()
11         if qual > threshold then
12             /* the mesh modifications inside the cavity meet the quality requirements, we can insert
                   it back into the mesh (see Figure 5.11) */
13             mesh.insert(cavity)
14             break
15         end
16         /* cells of the cavity keep a reference to the node that marked them, we decrease the range of
               such nodes for cells of quality below threshold */
17         R ← updateRange()
18         cavity ← extractCavity(N, R, m)
19     end
20 end
```

2D cases. We applied our approach in an intercode context, where our inputs are grid meshes carrying volume fractions data taken from a CFD simulation code at several time steps (see Figure 5.12) for two cases, the *triple point* and the *double bar* problems. They came from simulations run on grids of two different resolutions. The results in Table 5.1 highlight the motivations behind our approach: taking the triple point case at 1 second, we can see that for one grid resolution the base algorithm returns with a mesh containing no inverted cells (but still lower than the 0.3 minimum scaled jacobian threshold chosen by the user). That is not the case for the other resolution, making it unreliable. It is unrealistic to ask users to rerun their simulations with different resolutions at random, assuming it is even feasible, hence the need for an algorithm that consistently works.

Our method was applied on two additional hydrodynamics simulations issued from [Toro 2009] (see Figure 5.13); in all those cases it improves the distance by at least an order of magnitude.

3D extruded cases. The same cases from Figure 5.12 were extruded[2] and run in 3D.

While our method does indeed result in meshes meeting the quality requirements the ratio of $dist_{final}$ over $dist_{init}$ remains much higher than in the 2D cases.



$(a)\ t = 1sec$ $(b)\ t = 2sec$

$(c)\ t = 0.5sec$ $(d)\ t = 1sec$

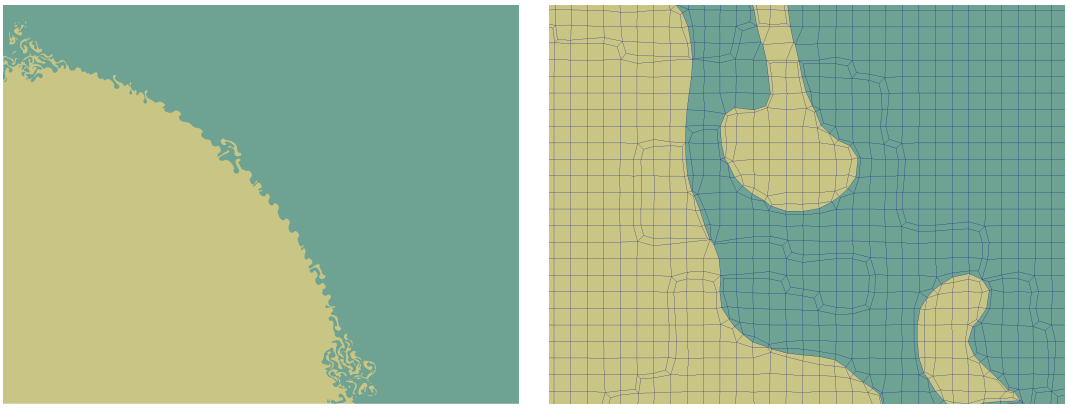
Figure 5.12: Examples of CFD simulations in 2D. Our algorithm was also applied to 3D cases extruded from the 2D. (a and b) triple point problem where three fluids of different densities lead to the formation of a vertex; (c and d) double bar problem where three fluids of different densities are stirred by two rotating blades.

"Real" 3D cases. Fully 3D cases were studied, one of which input is a grid where the volume fractions data were computed by imprinting an asteroid model into the grid (see the example in Figure 5.6). Other examples are Eulerian meshes from hydrodynamic simulations run using [Guy 2019], a ball of liquid that drops in a box taken at several time steps in Figure 5.14, a dam that breaks and a three material case where a liquid is poured onto a concrete pillar in Figure 5.15.

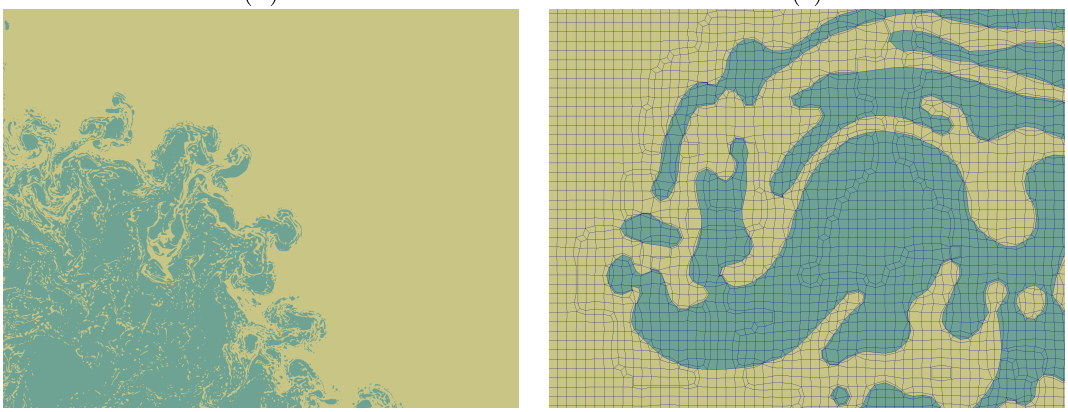
These cases really highlight the need for our updated interface nodes positions, as without it we ended up with stair-shaped meshes. We should also note that in these cases, the first controlled move was executed with a threshold $S_q = 0.3$ which was decreased to 0.2 for the remainder of the algorithm. Without this intermediate threshold the cavity adaptation never manages to produce a good enough one to insert back into the mesh, which means our output mesh would have been the mesh obtained after the first controlled move. The distance $dist_{init}$ and $dist_{final}$ are illustrated in Figure 5.16, and an example of our decreasing cavity-size strategy is shown in Figure 5.17.

We compared our approach with SCULPT (using CUBIT version $15.4b$) and SCULPT with our post-process developed in Chapter 3 aiming to reduce the $discrepancy$; the results can be seen in Ta-

[2]The 3D mesh is created from a 2D quad mesh, lying in the XY plane, by creating successive layers of hexahedral cells along the Z direction. Volume fractions are simply derived for each hexahedral cell from their origin quadrilateral cell.

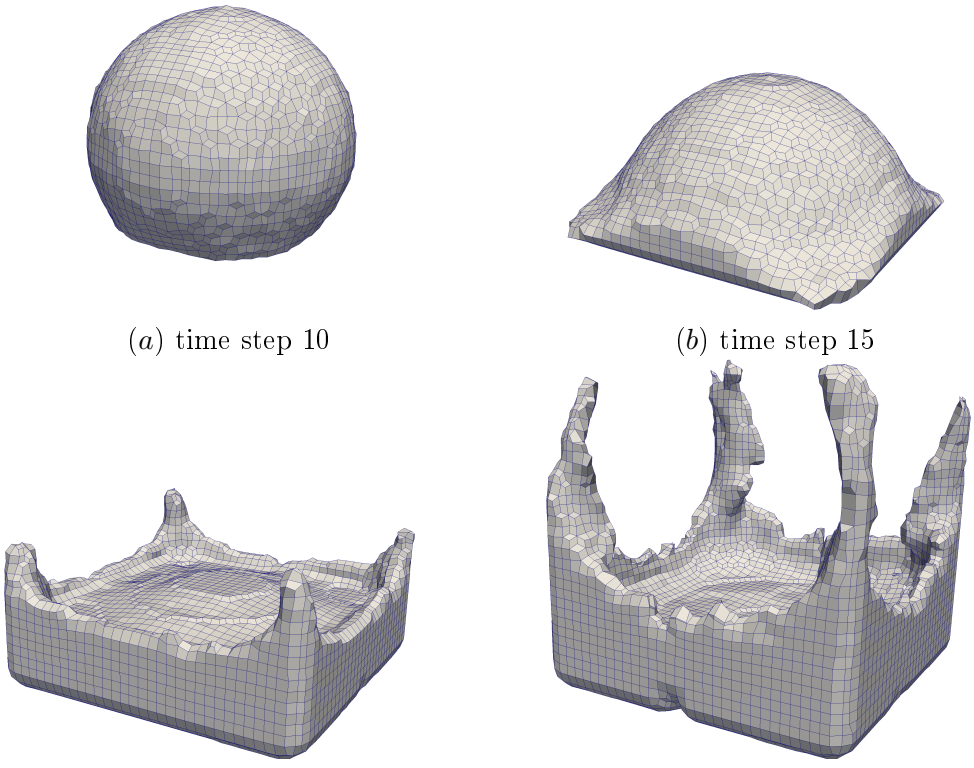


(a) (b)



(c) (d)

Figure 5.13: Other examples of hydrodynamics simulations in 2D [Toro 2009]. (a) and (c) the two cases; (b) and (d) close-ups on our resulting meshes shown respectively.



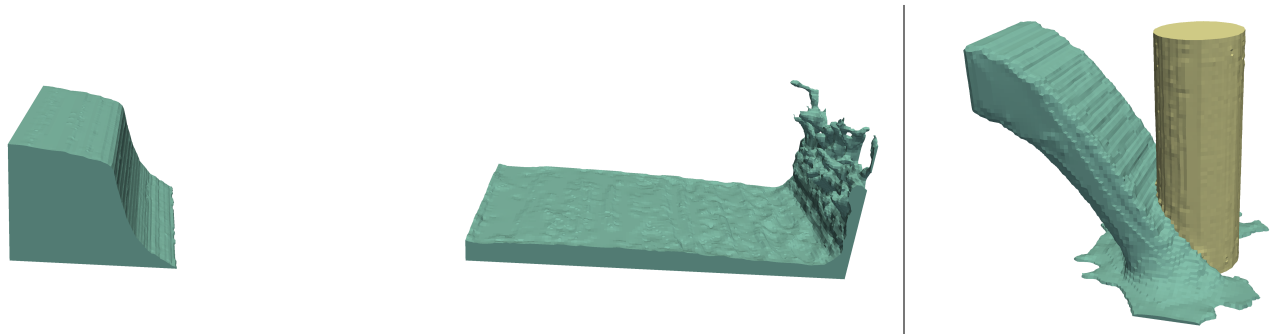
(a) time step 10 (b) time step 15

(c) time step 20 (d) time step 25

Figure 5.14: Resulting meshes from our algorithm applied to the balldrop case.

Table 5.1: Quality and distance metrics for the examples.

case name	$minJS$ *base algo*	$minJS$ *our algo*	$dist_{init}$	$dist_{final}$	$\frac{dist_{final}}{dist_{init}}$
2D					
triplepoint 1s 420x180	0.215	0.322	0.0676	0.0071	0.105
triplepoint 1s 518x222	-0.071	0.310	0.0856	0.0061	0.071
triplepoint 2s 420x180	-0.031	0.311	0.165	0.0138	0.084
triplepoint 2s 518x222	0.097	0.308	0.186	0.0163	0.088
doublebar 0.5s 200x100	0.074	0.306	0.5915	0.0163	0.027
doublebar 0.5s 214x107	0.091	0.301	0.5950	0.0121	0.020
doublebar 1s 200x100	-0.177	0.300	0.5768	0.0319	0.055
doublebar 1s 214x107	-0.109	0.301	0.6146	0.0411	0.067
hydro_toro_a	-0.104	0.300	116.70	6.0177	0.051
hydro_toro_b	-0.994	0.300	1902.3	104.34	0.055
3D					
triplepoint 1s 420x180x3	0.067	0.300	34.048	21.587	0.634
triplepoint 2s 420x180x3	-0.157	0.300	74.5847	44.388	0.595
doublebar 0.5s 200x100x3	0.043	0.300	134.47	26.025	0.193
doublebar 1s 200x100x3	-0.159	0.300	122.24	44.576	0.365



(*a*) time step 10 (*b*) time step 40 (*c*) in_out_flow

Figure 5.15: Resulting meshes from our algorithm applied to the dambreak and in_out_flow cases. The mesh size is between 450000 and 500000 cells in these examples while execution time ranges from 5 minutes to 2.5 hours.

ble 5.3. We do not claim to be exhaustive in testing SCULPT, which has many options to drive the mesh generation.

We can see that our robustness comes at a price, our method being approximately twenty times slower (SCULPT was run on a single MPI process, and our algorithm on a single thread). In Table 5.3 is shown that for the *doublebar* cases, which have the same grid size, the execution time varies significantly: our method is not only sensitive to the grid size, but also to the carried data.

We can also see that our method fits better the volume fractions, even after applying the add-on. It could be explained by the fact that in order for Sculpt to return an output mesh with no negative scaled Jacobian elements in those cases the $defeaturing$ [Owen, Brown, et al. 2017] option was activated, favoring mesh quality by sacrificing the volume fractions preservation (see the impact in Figure 5.18 where some small clusters of material were wiped out). Even with this option on, we were not able to obtain a mesh without inverted elements in the *double bar 1.0s* example. In Figure 5.19 we can see that an aggressive smoothing makes the rotating bar loose its shape.

This $defeaturing$ option can be quite useful in order to help produce good-quality meshes and simpler, smoother material interfaces; by reassigning the cells it basically consists in filtering out small details and clumps of materials. Some simulation codes might not even be able to handle those slivers of materials, or maybe could but at a high computational cost, for example by reducing the timestep. But a user might be interested in capturing those details as they can be relevant for the sim-

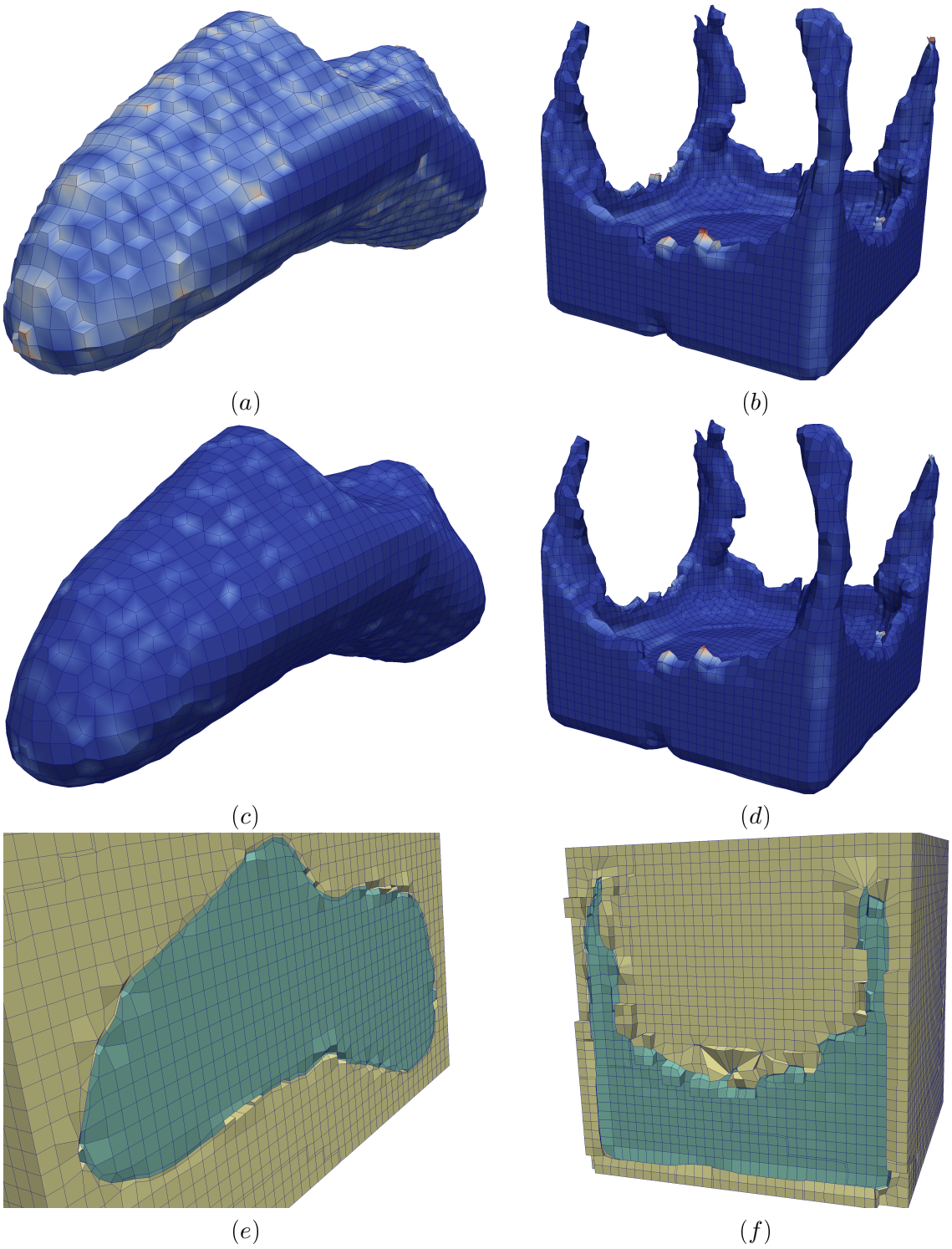
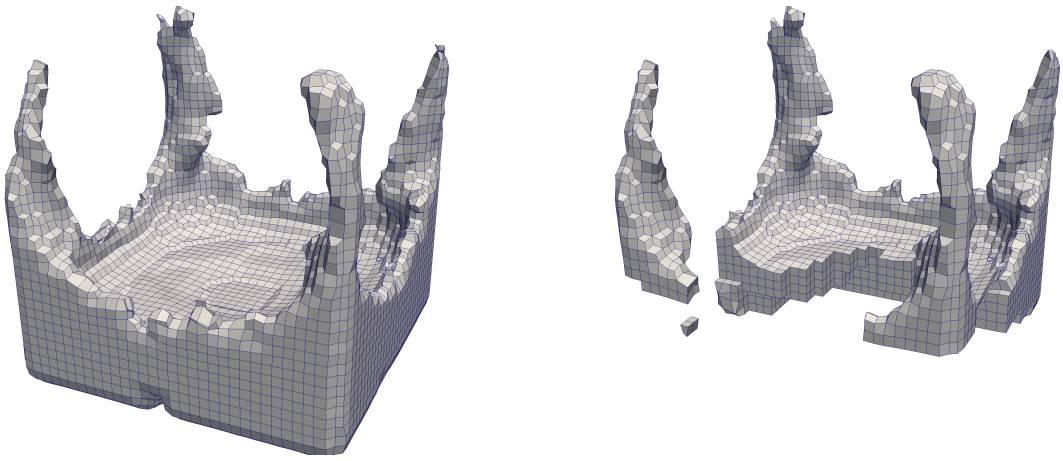
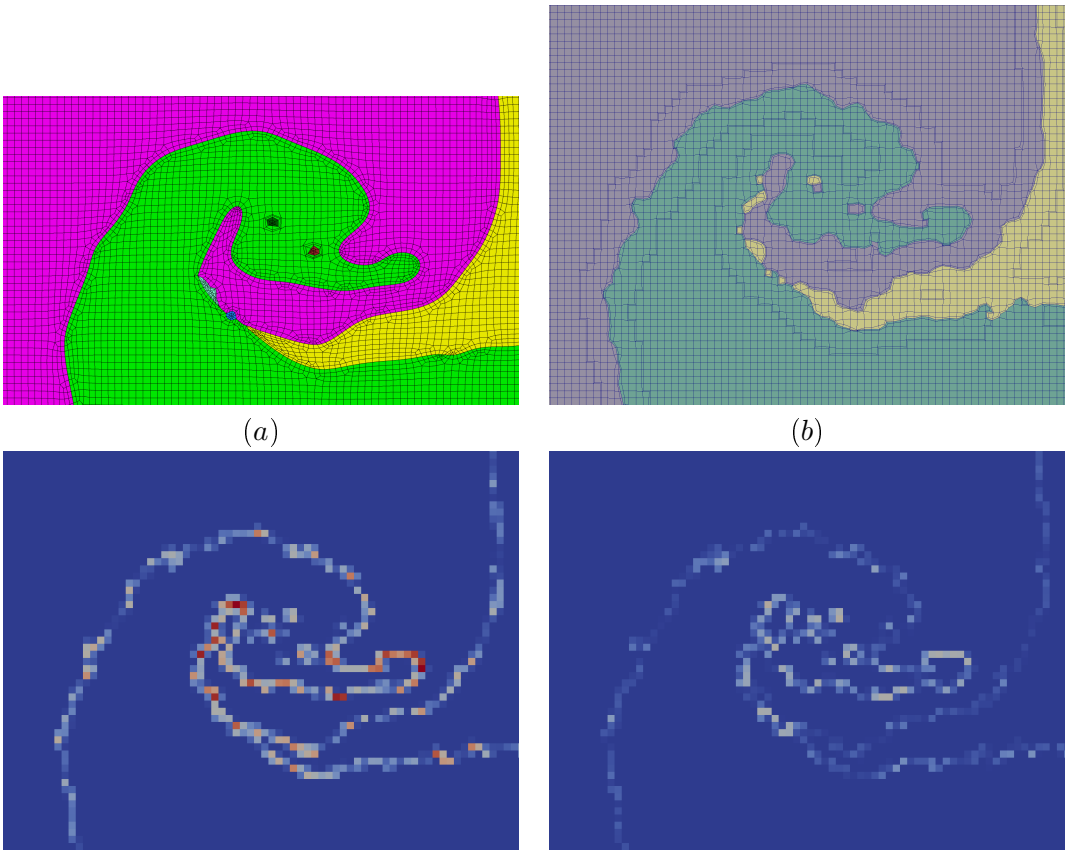


Figure 5.16: 3D cases measuring the impact of our algorithm. (a and c) the distance in the asteroid case between the interface nodes after the first controlled move and their computed position, and the same after applying our whole algorithm; (b and d) the same for the balldrop case at step 25; (e and f) a clipped view of both cases in order to exhibit the exterior.



(a) $nbCells = 9548$ (b) $nbCells = 4115$

Figure 5.17: Cavity downsizing for the pillow on the balldrop case at step 25. (a) first cavity on which the pillow was tried for the fluid material, it is practically the whole fluid; (b) cavity where the pillow and smoothing phases managed to produce a submesh of acceptable quality that was inserted back into the mesh.



(a) (b) (c) (d)

Figure 5.18: Comparison for the *triplepoint 2sec* case. (a) and (b) output meshes from SCULPT and ELG pipeline; (c) and (d) their respective per-cell discrepancy.

Table 5.2: Quality and distance metrics for the examples.

case name	$minJS$ *base algo*	$minJS$ *our algo*	$dist_{init}$	$dist_{final}$	$\frac{dist_{final}}{dist_{init}}$
asteroid	-0.13	0.200	319.874	31.148	0.097
balldrop_10		0.274	41.426	5.5029	0.133
balldrop_15		0.209	35.243	6.3432	0.18
balldrop_20		0.221	35.824	18.149	0.506
balldrop_25		0.200	75.346	39.089	0.519
damnbreak_10		0.200	34.444	17.638	0.512
damnbreak_40		0.200	112.73	67.012	0.594
in_out_flow		0.200	132.75	75.079	0.565

Table 5.3: Discrepancy and execution time comparison with SCULPT, SCULPT + add-on and our approach.

case name	metric	*Sculpt*	*Sculpt*+ add-on	*our algo*
triplepoint 1s 420x180x3	discrepancy	583.983	248.594	205.699
	time (s)	24.57		505.2
triplepoint 2s 420x180x3	discrepancy	1204.86	513.38	433.458
	time (s)	26.5		540.6
doublebar 0.5s 200x100x3	discrepancy	852.687	351.924	270.238
	time (s)	9.14		187.9
doublebar 1.0s 200x100x3	discrepancy	fail	fail	546.159
	time (s)	84		334.8

ulation one builds the mesh for; we present in the next Section 5.2 techniques dedicated to preserving those.

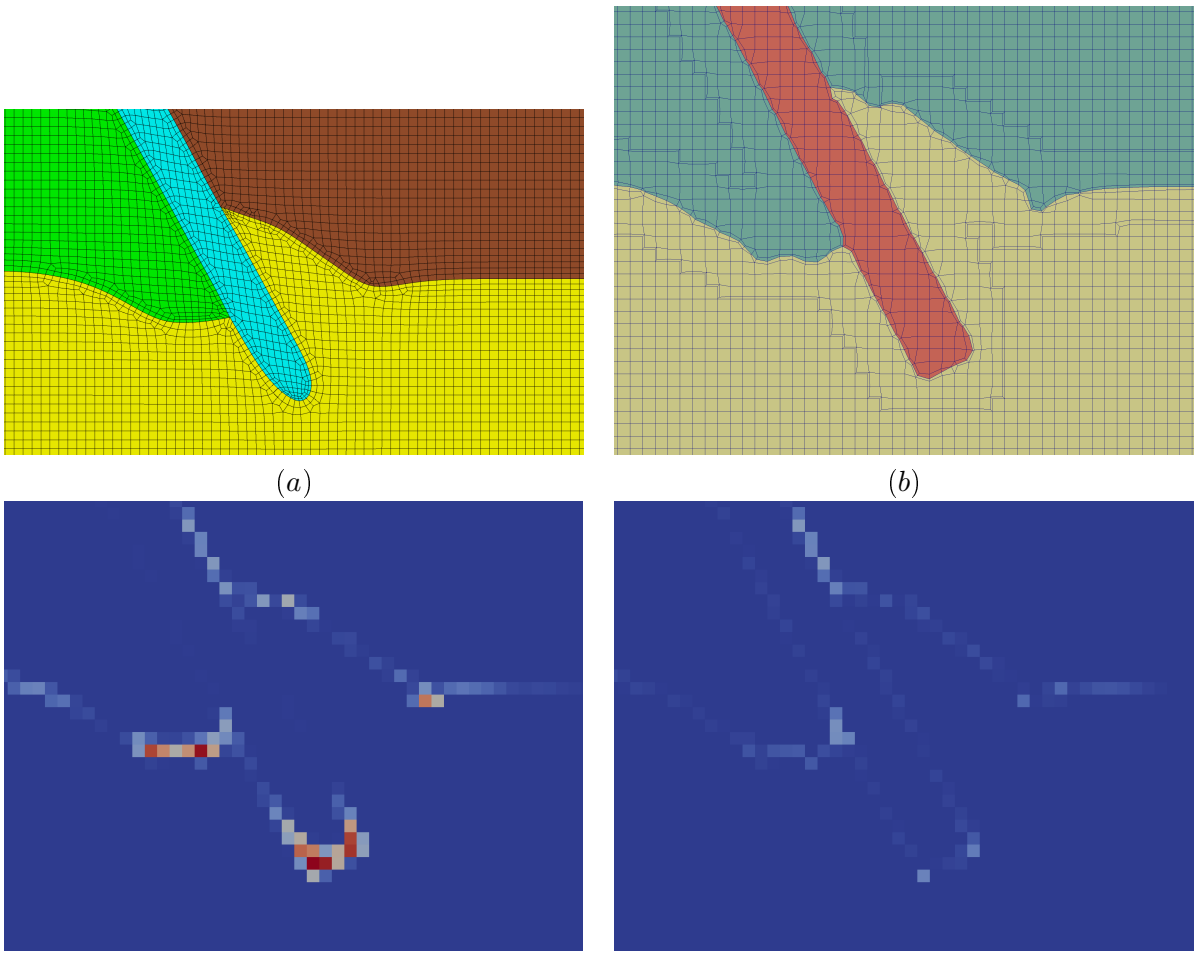


Figure 5.19: Comparison for the *doublebar 0.5sec* case. (a) and (b) output meshes from SCULPT and ELG pipeline; (c) and (d) their respective per-cell discrepancy.

5.2 Mesh refinement for getting usability

In addition to maintaining a minimum cell quality – we showed in the previous section how we enforced it – a user might have expectations on the mesh related to what it features. In our intercode case between an Eulerian code and a Lagrangian one it is especially true; as two separate codes they will most likely differ on what they modelize and consider of interest. Being Eulerian, by definition the mesh cells will not have followed the material interfaces, and a material of interest for the second code could have been neglected and misrepresented by the first one.

Our method, with its cell assignment based on the material with the highest volume fraction, might inaccurately capture some parts of the domain and miss certain geometric features that the user wants to retrieve because they are relevant for the second simulation code.

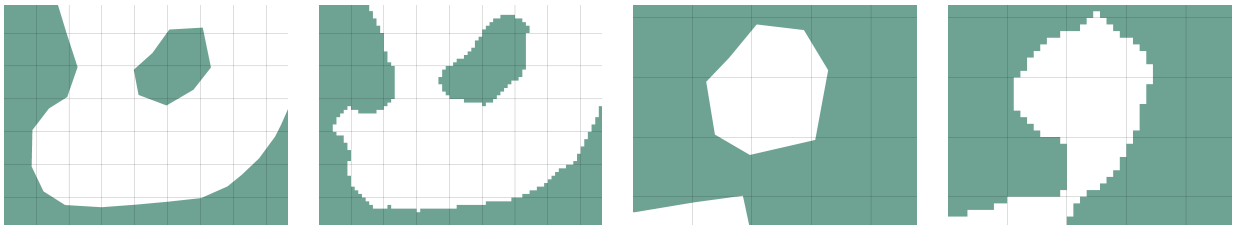


Figure 5.20: Topology comparisons between our final mesh and the voxelated interfaces on close-ups of the case in Figure 5.13 page 87; the black wireframe is the original grid mesh carrying volume fractions data. (a) and (b) both return the same topology for the green material; (c) and (d) the topology differs.

In Figure 5.20 is shown a case where the voxelated geometrical model topologically differs from the coarser one, due to the cell material assignment filtering out the smaller areas where one material is not present enough. Had the Eulerian code run with a finer mesh it might have been captured... or not. In Figure 5.21 an example with a thin layer of one material illustrates the issue where, depending on the resolution of the Eulerian mesh, the volume fractions associated to the thin layer will allow or not our algorithm to extract a geometrical model that matches the initial CAD model; in this example it actually ranges from not being represented at all to being adequately captured. This experiment[3] is made possible here precisely because we have the CAD model as a reference, which we do not have in a real case scenario.

As the expectations on the geometrical model depend on the user and what one plans to do with the resulting mesh, we can provide functionalities that would bias the cell material assignment in order to favor a hierarchical list of materials established by the user; when a material tagged with a higher priority is present in a cell, this cell would be assigned to the material. Such results could be welcomed in the case of thin layers, as seen in Figure 5.22 on the first row where the thin layer which was previously captured in disconnected parts now forms one component, but could make things worse as shown in the second row where the two thin layers[4] were adequately captured without the bias and where the grey thin layer becomes disconnected when the yellow material is favored. It shows that we need to devise another method to ensure the correct representation when needed because simply tweaking the cell assignment is unreliable.

In our problem, as we start from the input mesh and its carried volume fractions we are not at liberty to change the mesh resolution or refine it locally to capture some details of the CAD – which we actually do not have – as is done in many other overlay grid methods. Nevertheless, we propose to mimic this capability by use of our voxel-based interface reconstruction we presented in Chapter 4.

[3] The example is a square domain of size 7 composed of:

- a green material which is a quarter of a disk of radius 5;
- a yellow thin layer of 0.3 thickness;
- an orange material that takes the rest of the domain.

We used grid sizes of 8×8, 12×12 and 32×32 cells.

[4] The example with two thin layers is the same as the previous example, with an additional grey thin layer also of thickness 0.3.

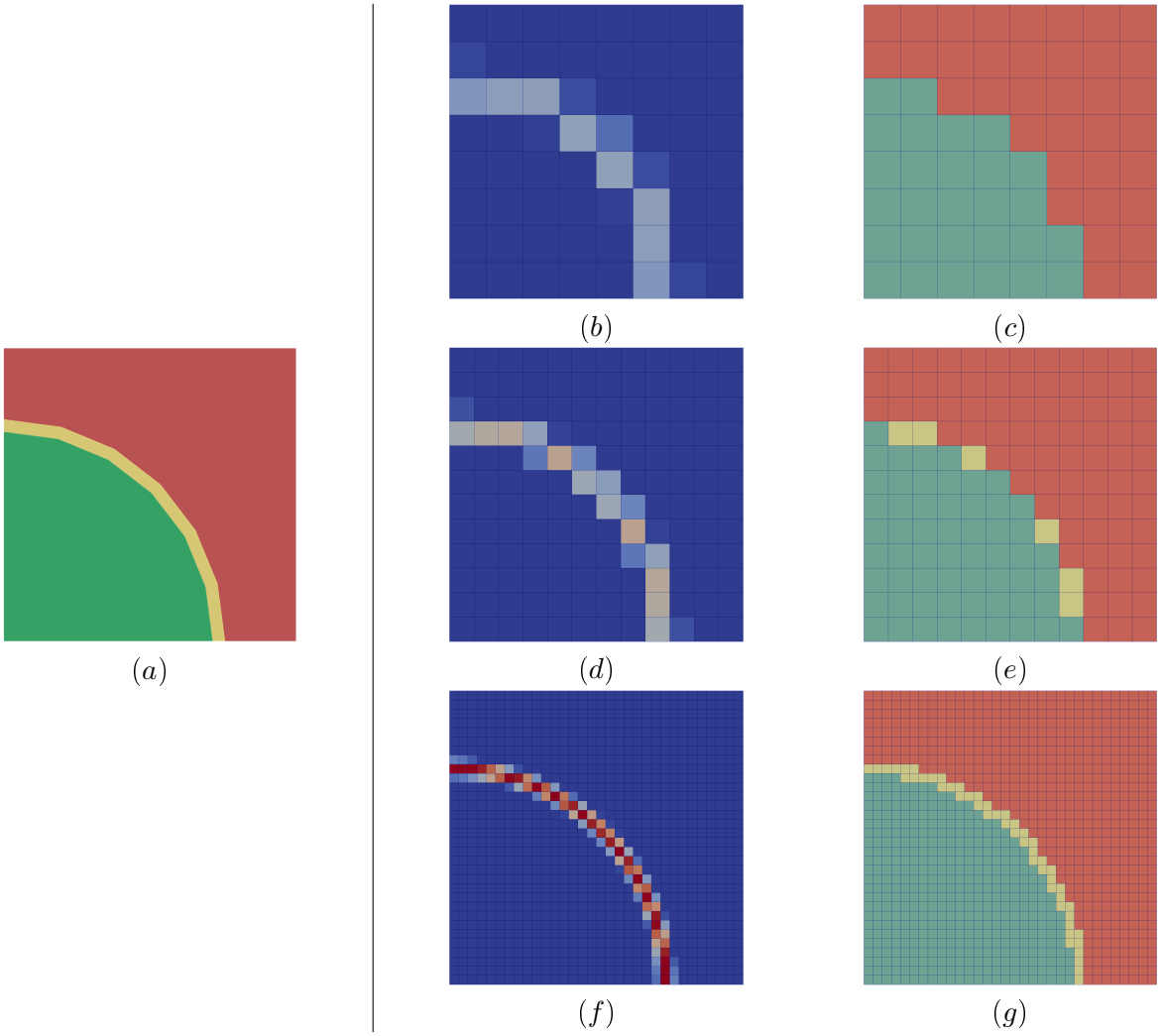


Figure 5.21: Example of the representation of a thin layer of the yellow material in a CAD-based case with different grid sizes; the volume fractions presented on the left column are those of the yellow material, ranging from 0 (blue) to 1 (red). (a) the CAD model; (b and c) the volume fractions and the results of the cell material assignment in our overlay grid method, where the yellow material is not present at all; (d and e) with a finer grid size the yellow material appears but does not match the shape of the CAD; (f and g) with an even finer grid the yellow material matches the CAD.

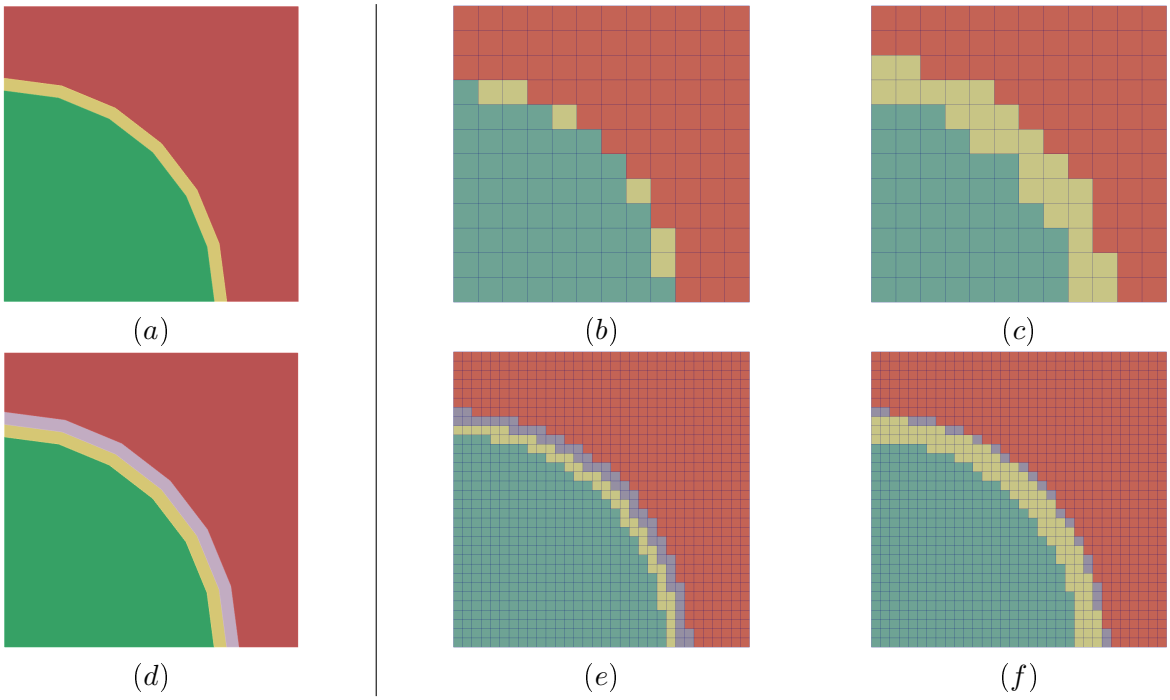


Figure 5.22: Comparison in two cases (left column) between the majority volume fraction cell assignment (middle column) and the assignment where the yellow material was enforced (right column). (a to c) the thin layer example a where the yellow material went from forming small isolated components in b to one single component in c, thus matching the CAD; (d to f) two thin layers example where the cell assignment accurately captured the two layers in e but enforcing the yellow material led to fragment the captured grey layer in f.

5.2.1 Our user-guided process

The overlay grid methods with mesh adaptation capabilities that we want to emulate are based on the principle that when the base mesh would not give satisfactory results, it is adapted to better fit the CAD model and the method is then executed on this new mesh. Our overlay grid method takes as an input a mesh and volume fractions carried by that mesh. While this input mesh can be manipulated and modified, or a whole new mesh could be produced using altogether other means, the input volume fractions are not easily transferable to this new mesh as it does not make sense to average those in the destination cells.

We propose to solve this issue by extracting the material interfaces in the mixed cells; in this document we use our discrete interface reconstruction method based on assigning voxels to materials presented in Section 4.1. Any method that locates the materials inside those cells would do, but the tendency of our discrete interface reconstruction method to produce connected components is a plus as it will avoid spreading isolated trace amount of volume fractions of materials in the destination mesh. We reapply the techniques we presented in Section 3.1 – that we used to compute the $discrepancy$ – to project the voxels assigned to materials onto the new mesh, thus computing new volume fractions. An illustration of our proposed solution can be seen in Figure 5.23: from left to right, starting from the input mesh with volume fractions, our voxels interfaces are built then projected onto a new refined mesh – here it is three times finer – in order to compute volume fractions on that new mesh; finally, our overlay grid method can run on this new data.

We can see that Figure 5.23.d better matches the CAD than what we would obtain initially in Figure 5.21.e, but the yellow material is still extracted in two parts in the single thin layer case; the same goes for the grey material in the two layers case. As it happens during the discrete interface reconstruction phase (see Figure 5.23.b and f), we investigate for a way to compel the voxels material assignment to produce those connected layers. We can make the observation that so far we have formulated and treated the voxel-assignment problem as a graph partitioning problem, and while it is accurate we do have additional data available: since the graph we partition is spawned from a mesh the vertices of the graph do have spatial coordinates. Instead of simply considering reducing the number of pairs of adjacent vertices assigned to different materials, it becomes possible to make use of their spatial positions to compute a weight on the edges of the graph that will introduce a bias in the

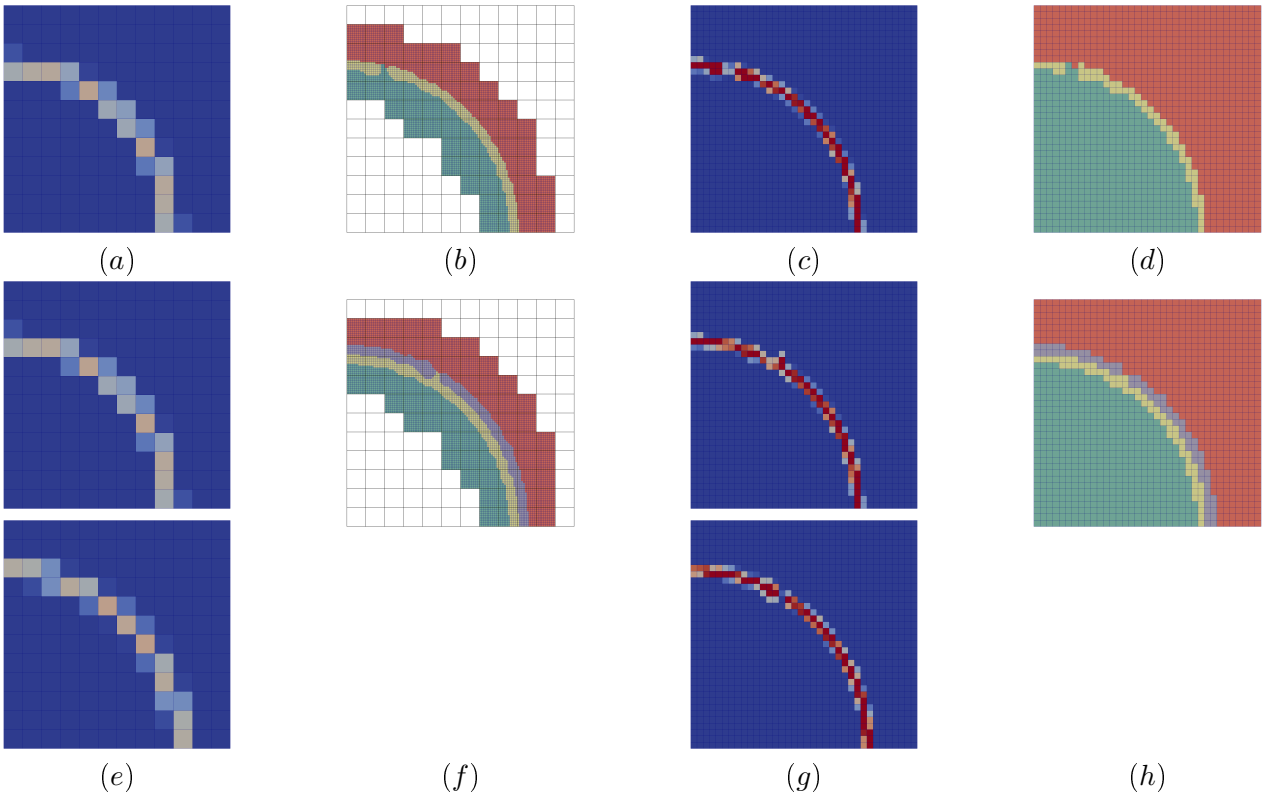


Figure 5.23: Example with thin layers where the mesh is refined. (a) volume fractions of the yellow material on the input mesh ranging from 0 (blue) to 1 (red); (b) the voxels assigned to materials; (c) newly computed volume fractions on a refined grid; (d) cell material assigned by our overlay grid method. $(e$ to $h)$ the same in the two thin layers case with the volume fractions of the yellow and grey materials respectively on the top and the bottom rows.

algorithm guided by some spatial or directional considerations. This bias could be specified by the user; in the thin layer cases that we exhibit here, as we aim for the voxels assigned to the yellow (and grey) material to form a continuous component in the shape of said layer we rely on the gradients of the volume fractions to provide us with a direction orthogonal to which we want our yellow component to spread. In practice we compute the weight carried by the edge between two vertices v and w, with $v \in cc$ its coarse cell and m the material considered as:

$$weight = \frac{1.1 - |(v,\vec{w}).grad_{cc,m}|}{1.1}.$$

The impact of this weight-based variation on the voxel-assignment problem can be seen in Figure 5.24 where the previously split thin layer materials in our two cases now form connected components that reach across the domain.

Those improved reconstructed interfaces are used to again compute new volume fractions on a refined mesh in Figure 5.25 where the cell material assignment now also produces single components for the thin layers, contrary to what was obtained without this directional bias.

Results presented in Figure 5.26 and 5.27 show the effectiveness of our process in 3D cases where our overlay grid, when directly executed on the input mesh (a $3 \times 3 \times 3$ grid), produces a geometrical model topologically different from the one extracted from our voxel material interfaces. In both of those 3D cases with two materials we used a refined mesh of size $9 \times 9 \times 9$ on which volume fractions are computed and our overlay grid algorithm run anew.

We have proposed a method that allows us to mimic the mesh adaptation capabilities of other overlay grid algorithms despite not having a CAD model as an input. we make use of techniques we previously employed across this document that allow us to compute volume fractions on a new mesh, and we have shown the impact of some bias one can use to steer the material capture when one has additional knowledge on how they should appear.

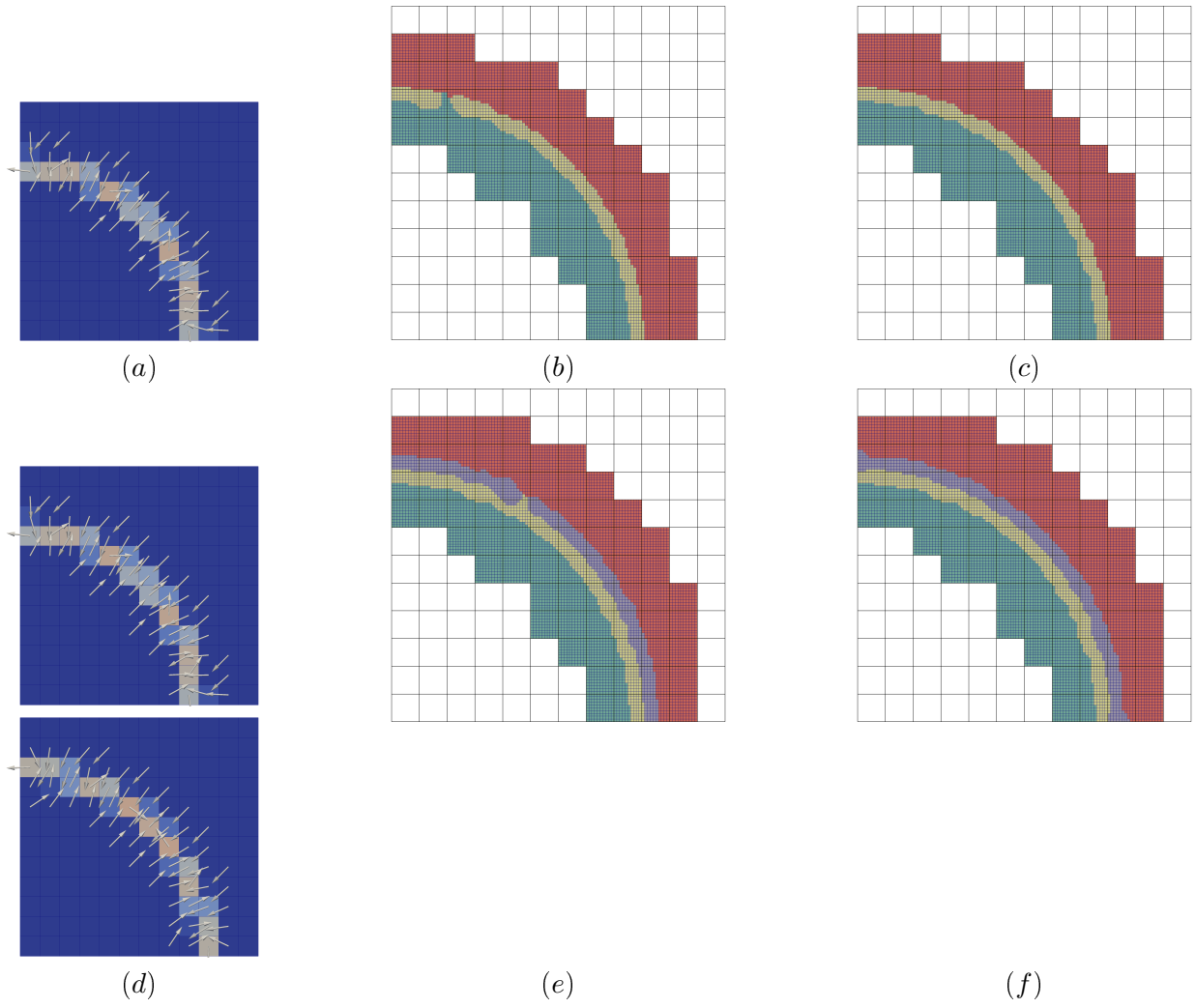


Figure 5.24: Impact of our proposed weighted voxels material assignment. (*a*) the input volume fractions for the yellow material and the computed associated gradient; (*b*) our basic voxel assignment where the yellow layer happens to be split into two components; (*c*) voxel assignment weighted by the gradient orientation that produces a single component for the yellow material. (*d* to *f*) the same in the two thin layers case where the grey layer ended up split.

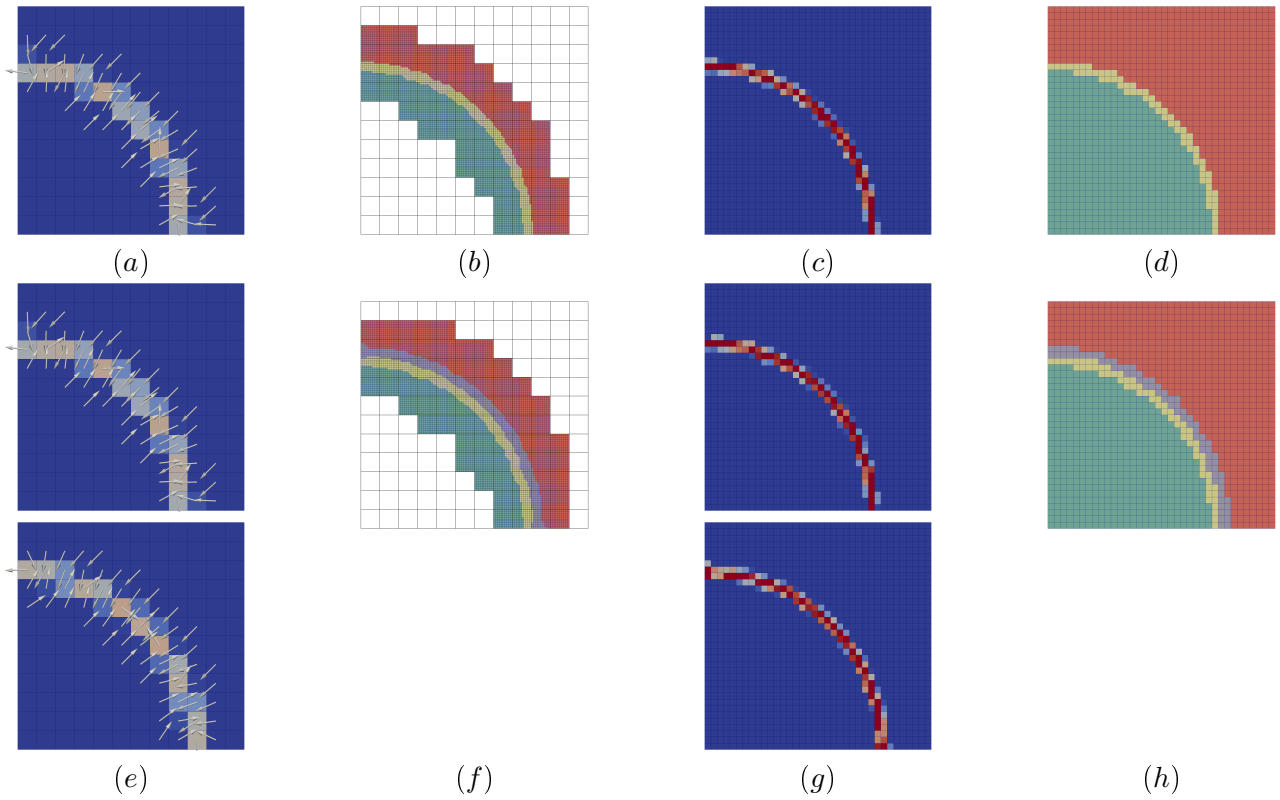


Figure 5.25: Example with thin layers where the mesh is refined but this time with our weighted voxels material assignment. Compared to Figure 5.23 the thin layers are preserved.

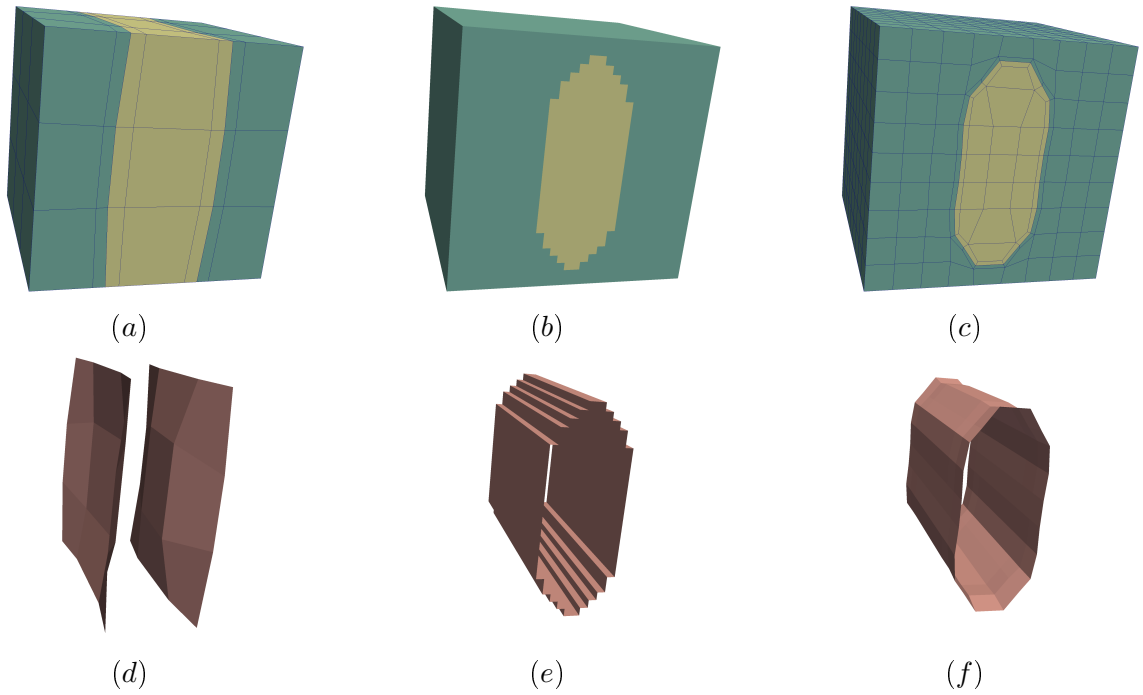


Figure 5.26: 3D example of our process designed to handle mesh refinement. (*a*) our base overlay grid algorithm; (*b*) the voxels material assignment; (*c*) overlay grid algorithm run after computing volume fractions on a finer mesh; (*d* to *f*) the respective extracted geometrical models where we can see the *f* matches *e*.

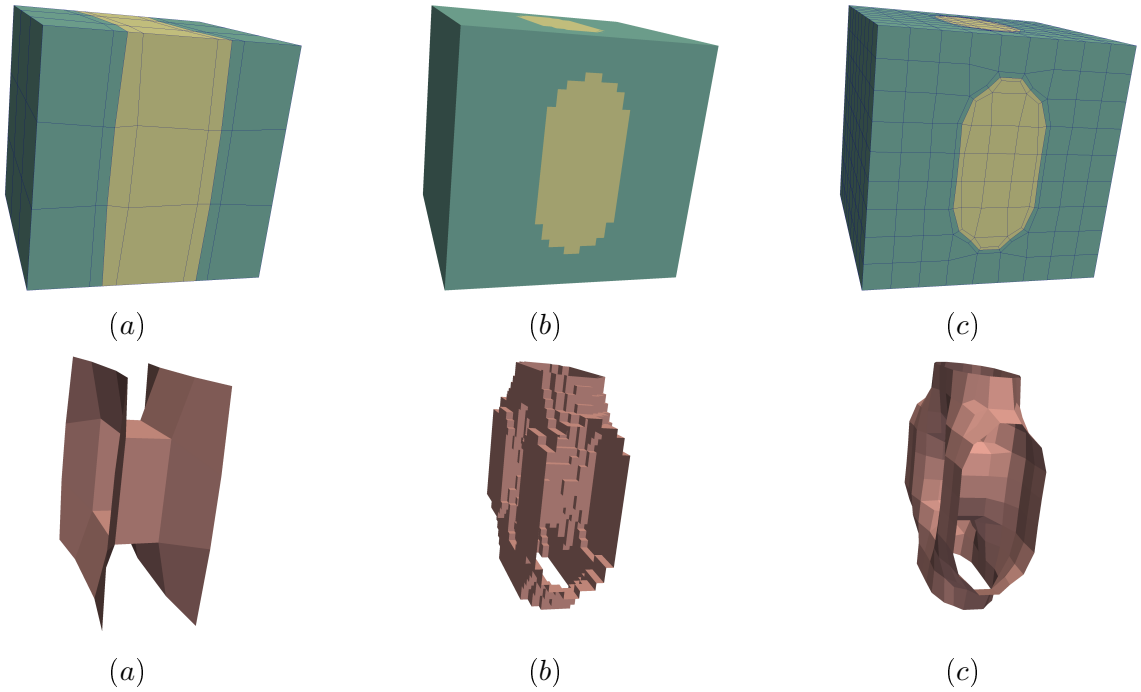


Figure 5.27: Another 3D example of our mesh refinement handling process.

5.2.2 Automatic refinement

We have proposed a method that allows for the transfer of volume fractions carried by one mesh to another so that our overlay grid method can be applied on this new mesh; we will now present a method for obtaining said new mesh. We will here build a new mesh by adapting the original one; we rely on a classic 3-refinement template (see Figures 5.28 and 5.29), as it can easily be applied in unstructured cases, rather than the 2-refinement template where one has to alternate refinement in two directions. Cells that should be refined are marked, as are their nodes, and refinement templates are applied to the cells adjacent to marked nodes; since not all configurations are handled in 3D, the marks are propagated until we end up with only valid configurations.

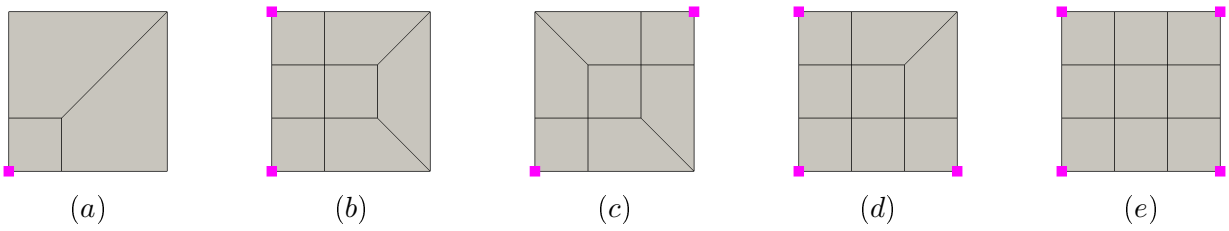
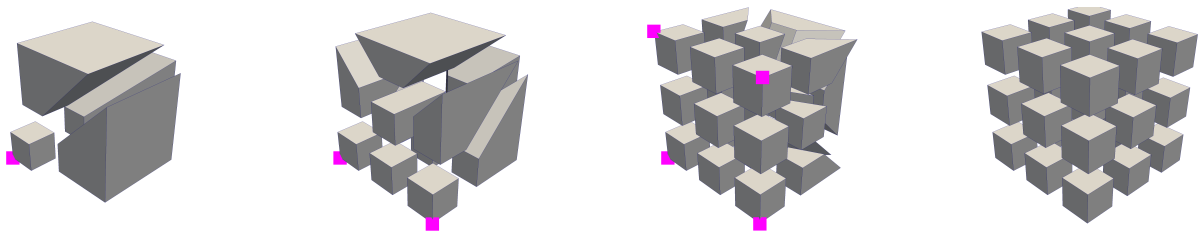


Figure 5.28: Refinement templates in 2D. The marked nodes are those around which the refinement will be done.

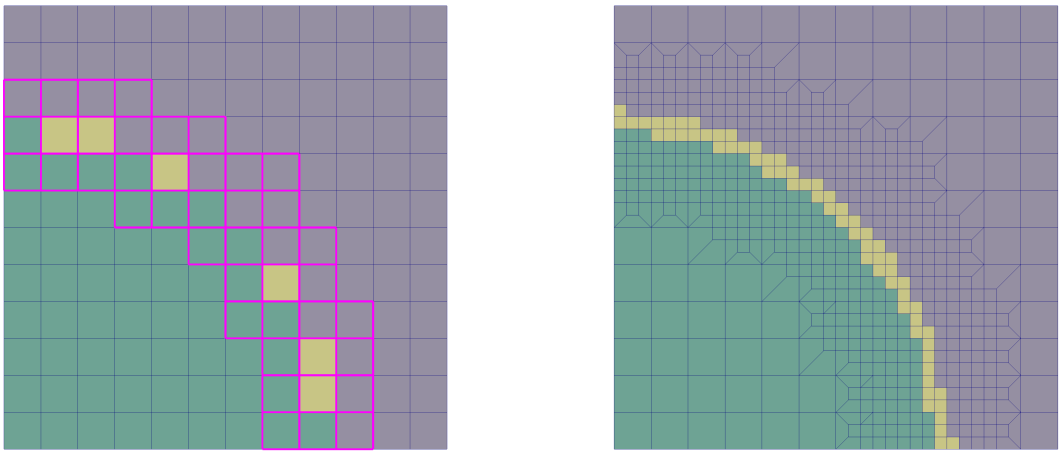
In the examples we have provided in Figures 5.23, 5.24, 5.25, 5.26 and 5.27 the finer mesh we used was a refinement of the whole grid which allowed for a demonstration of our capability to run our overlay grid implementation by computing new volume fractions on this adapted mesh. As seen in the thin layers examples, only the area containing the thin layers is actually of interest as there is no benefit in refining the rest of the domain.

While the user can always select individual cells or areas to refine, we propose the following strategy to identify cells that could benefit from the refinement: drawing inspiration from the *defeaturing* option present in *Sculpt*, we infer that isolated cells assigned to a material are likely to indicate areas where we want to increase the resolution, or *refeature* the domain. We also consider cells that contain a fraction of a material but where none of the adjacent cells were assigned said material; such a selection is illustrated in Figure 5.30 where the cells identified as explained previously, in addition to their adjacent cells, are highlighted. In this example the adapted mesh is twice as small as when refining indiscriminately (668 vs 1296 cells).



(a) (b) (c) (d)

Figure 5.29: Refinement templates in 3D. The marked nodes are those around which the refinement will be done; in case of a cell where the marked nodes do not match one of the above templates, additional nodes are marked.



(a) (b)

Figure 5.30: Example of a targeted refinement. (a) the highlighted cells are to be refined; (b) the adapted mesh after refinement and its cells assigned to materials according to the volume fractions computed on this new mesh.

Chapter digest – *In this chapter we have designed a new pipeline for an overlay grid method that guarantees a minimum cell quality. It relies on the simple notion that if the starting mesh is good enough, not moving a node is always an option; as we have shown it can easily lead to stair-shaped meshes we provided solutions to mitigate this drawback. Additionally to individual cell quality, one has to consider the difference in how two simulation codes modelize the domain and what they consider of interest. As such, we demonstrated some solutions to* refeature *the domain by mimicking mesh adaptation strategies encountered in other overlay grid algorithms, with the twist in our case that our input is not a CAD model but a mesh carrying volume fraction data.*

Chapter 6

Conclusion and future works

We conclude this document by giving first a brief recap about this work. Starting from an industrial requirement, which is to design and implement an intercode component that could be used in an HPC toolchain for 3D numerical simulations, we designed the ELG pipeline. But this work is not over, it even raised several questions and some future work that we should explore in a short, medium or long term.

6.1 The current ELG pipeline

In the work presented in this document, we tried and addressed the issue of intercode data transfer, at the level of the discrete representations that are meshes. This issue occurs quite often in practice: a physical study using a numerical simulation toolchain can make use of several different simulation codes, chained one after another, with each its own requirements. More precisely, in our case we endeavored to convert the output of an Eulerian simulation code, an hexahedral mesh carrying volume fractions, into the input of a Lagrangian one, an unstructured hexahedral mesh with its cells assigned to materials.

Automatic mesh generation is not a given, more so when a user needs to generate an hexahedral one; a classic way of producing a simulation-grade mesh is to use a CAD-meshing software into which the geometrical model is imported and an engineer interactively prepares it for meshing (usually by decomposing it into meshable blocks). As our data is coming from a simulation code we do not have such a model, thus we pursued two objectives: we aimed at extracting a geometrical model that could be used for meshing, and we also tried to directly obtain a usable mesh.

We began our work by identifying an overlay grid method that almost fits our initial requirements, which are to at least take the same input. It is the case with the Sculpt algorithm. We defined then a criterion that allowed us to evaluate how close to the input its output is. Indeed an important feature of the intercode processing is to conserve some quantities (the volume in our case). We also devised a post-process we showed manage to reduce that discrepancy by an order of magnitude, consistently over a wide range of cases. This first part of our work was presented in Chapter 3 and it is the starting point that led us to design the ELG pipeline we presented in the remainder of this document.

In order to drive our process to convert an Eulerian mesh into a Lagrangian one, we consider that it would be useful to have a reference geometrical model as a CAD model can be for other types of applications. That is why we designed and implemented an algorithm to reconstruct the material interfaces from our Eulerian data; by subdividing the input cells into voxels and assigning them to the materials, we obtain a geometrical model. Our method is inspired from the scientific visualization field, but we devised it so as to lift some limitations, mainly when the input mesh is unstructured, and we showed that graph re-partitioning techniques could be used to avoid the appearance of undesirable artifacts. It was the purpose of Chapter 4.

Using this geometrical model as a reference, we proposed in Chapter 5 a complete overhaul of the original pipeline in order to control mesh quality. In order to directly generate a mesh that meets cell quality requirements, our new solution guarantees that the resulting mesh quality does not go below a user-defined threshold (see Figure 6.1). We provided alternatives to some of the topological and

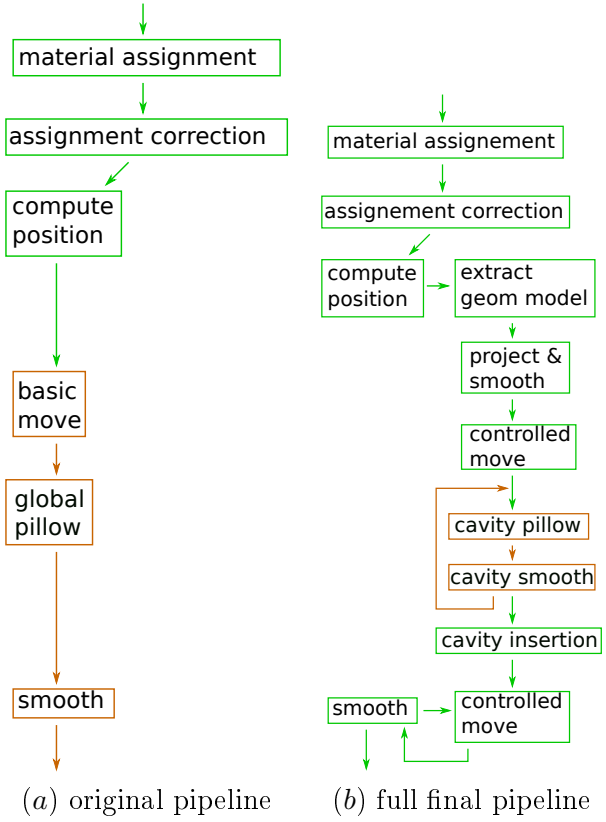


Figure 6.1: From our initial redesign of the SCULPT algorithm (left) to the full ELG pipeline where the final mesh quality is controlled (rigtht). Green boxes ensure cells quality control while orange ones do not.

geometrical modification steps that could degrade the mesh quality, and made use of our extracted geometrical model to still capture the input data and not diverge too much from it.

But usability is not only defined by mesh quality, but also by other criteria; depending on what phenomenon the study will be conducted on, the mesh can be expected to have certain traits related to the input data. We have implemented in Chapter 5 mesh adaptation capabilities that mimic the features available in other overlay grid methods that start from a CAD model and refine the initial mesh to capture details, despite not having said model in our case. We have illustrated how those functionalities can be used in cases where one wants to preserve, or erase, thin layers of materials.

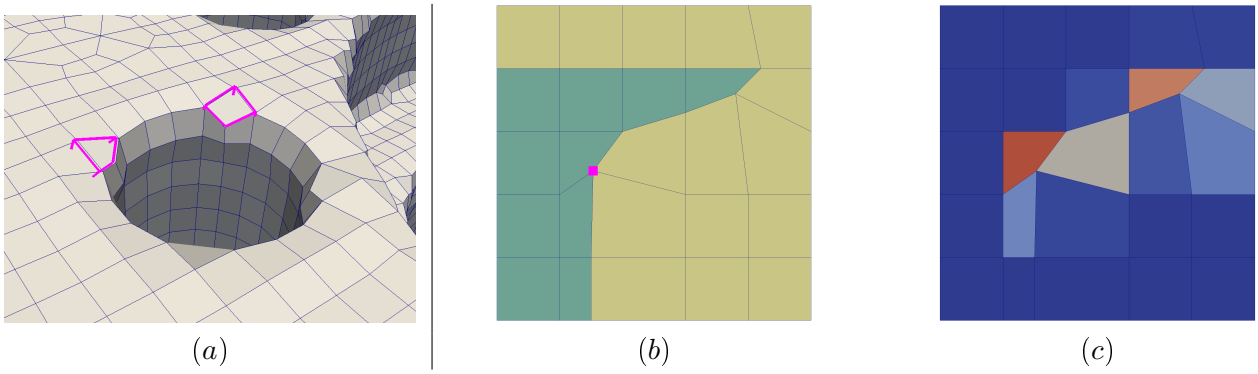
6.2 Future works

In the course of our work, we have identified several areas the study of which could benefit our proposed solution, whether by improving the results (pillowing in 6.2.1), extending the usefulness of our method (cell size control in 6.2.2), potentially making maintenance and developing extensions easier (mesh representation and component-based programing in 6.2.3), and finally making it more practical to use (performances in 6.2.4).

6.2.1 Extending pillowing capabilities

In Section 3.3.3.1, we identified a limitation to our discrepancy improvement algorithm; as it only moved the nodes with no change in the mesh topology some supplementary improvements to the discrepancy could not be reached as can be seen in Figure 6.2.a where the movement of the nodes of the highlighted cells – the nodes at the tip would move outward so as to end up with a cylinder-shaped hole – is limited in favor of keeping a minimum cell quality. One way of allowing for further movement would be to provide more degrees of freedom for those nodes, i.e. to create adjacent edges by use of a pillowing operation that goes across both the CAD piece and the exterior; so far we have implemented pillowing capabilities restricted to one material only.

An issue that we can have with our cavity pillowing is that while we have shown that not blindly executing a global pillowing avoids degrading quality unnecessarily, we could still go further in cases such as the one illustrated in Figure 6.2.b and c; the highlighted node will be the seed of a cavity pillow, but the cell quality shows that this pillow does not in fact need be applied in the yellow material, as it is the green adjacent cell quality that limits the node movement. Our current strategy is to apply a pillow around those nodes in all the adjacent materials and we do not discriminate further, which can lead to problems as we have seen in Section 5.1.2 page 81 where pillowing when not necessary can have adverse effects on mesh quality.



(a) (b) (c)

Figure 6.2: Illustrations of improvements that could benefit our current pillowing strategy. (a) edges should be added – via a cross material pillow – to the nodes on the tip of the highlighted cells so that they could be pushed further back to form a cylinder hole without degrading the cell quality too much; (b) a pillow need to be applied around the highlighted node in the green material but there is no need for that in the yellow one as it can be seen in the cell quality display (c), where blue is good and red is bad, that it is the adjacent cell in the green part that limits the node movement.

In addition to improving the mesh quality pillowing, and other hex mesh operators such as sheet collapsing, could also be used to control the cell size.

6.2.2 Extending cell size control capabilities

In Section 5.2 we have demonstrated that it is possible in our intercode case to provide mesh adaptation capabilities usually available in overlay grid algorithms that take a CAD model as an input. Benefits include the representation of details in the data, and when the output mesh is used in a simulation code cell size is also of importance (when not directly using the output mesh, but instead extracting a geometrical model and importing it into a CAD meshing software controlling the cell size in our algorithm is not important).

In addition to the 3-refinement scheme that we used, it is possible to use pillowing as a way to locally refine the mesh. So far we have only used pillowing in order to improve the mesh quality; our implementation could be used as is, but there remains the problem of identifying where it would be applied.

Coarsening the mesh will prove more difficult; chords and sheets collapsing are not local operations, and while the prescribed cell size might be larger in one part of the domain it could be smaller in another part. Techniques such as [M. Staten et al. 2016] retrieve a block-structure from an unstructured mesh, while repeated collapsing is done in [X. Gao, Panozzo, et al. 2017] to the point that again, a block-decomposition is obtained. Those techniques are not simple to implement and are rife with issues. Most importantly, obtaining an arbitrary block-decomposition, while a huge achievement in itself, is not entirely desirable: as the purpose is to then mesh each block with a grid, producing a conformal mesh will lead to constraints on the subdivisions of the edges and faces of the blocks propagating across the whole domain, meaning that meeting the cell size prescription in some area of the domain can conflict with the prescription in another area.

We feel that the most robust solution in our case would be to reuse the same principles we put in place in Section 5.2, but instead of modifying the input mesh by refining it, use an altogether other mesh that roughly meets the cell size prescription (the easiest would be to use a refined grid), project the volume fractions onto this mesh and run our overlay grid algorithm with this new data as an input.

6.2.3 From graph-based representation to combinatorial maps

As briefly explained in Appendix A, the ELG pipeline used a graph-based mesh data structure for implementing all the proposed software components involved in the pipeline. At the beginning of the PhD, an attempt was done to consider a combinatorial map mesh representation [Damiand and Lienhardt 2014] but we decided to focus on some other parts of the work.

Compared to combinatorial maps, our adjacency-based mesh representation usually has a lower memory footprint, but this does not hold true when nearly all the connectivities are needed (admittedly in our case they are not all needed at the same time; they could be deleted and their occupied memory freed when no longer in use, which would lower the peak memory consumption). We have implemented a combinatorial map data structure but have yet to use this variation apart from a few tests. It would be interesting to compare the two representations, not only memory-wise but also in ease-of-use terms and performances: as our explicit connectivities are not updated on-the-fly, topological mesh modifications such as the pillowing phase seen in Figure A.1 makes most, if not all, of the existing ones invalid, i.e. not coherent with the current mesh. In order to be used by latter components they have to be deleted and built anew.

6.2.4 Improving performances and parallel implementation

The current implementation of the ELG pipeline was done using thread-based parallelism with shared memory. This choice was done considering that most of the studies we have to lead involve Eulerian meshes having several hundred of millions of hexahedral cells. But this number will get larger in the near future and a parallel distributed version of the pipeline must be investigated. Moreover, looking at the current parallelism in ELG (see Appendix B), several observations can be made in order to improve the efficiency of our implementation:

- A cheap way to decrease the overall execution time of our current implementation would be to run with an optimal number of threads, different for each step. This is assuming that the performance profile is roughly invariant with the input data;
- An adjacency connectivity can be seen as a graph or sparse matrix; as our implementations of the connectivities building do not scale well, it should be fruitful to investigate or directly use the implementations of the Kokkos kernels that deal with those objects;
- There are many different ways of writing meshing components, in particular in our case where we do not have a fixed specialized data structure but one where the connectivities can be built on-demand. Identifying costly kernels is of course relevant, but so is considering whether it is beneficial or not to explicitly build and store a connectivity, especially when said connectivity is used in only one algorithmic component. We can also consider radically altering the code by switching from an adjacency-based to a combinatorial map-based mesh representation, making the total execution time the most practical indicator.

Appendix A

Mesh representation using combinatorial maps and a component-based implementation

The implementation of the ELG pipeline relies on our mesh data structure GMDS [Ledoux, Jean-Claude Weill, et al. 2008] that we intensively use. GMDS stands for Generic Mesh Data and Services and it represents any 2D or 3D mesh as an adjacency graph with the connectivities between mesh entities built and stored on-demand. We can handle any type of cells such as triangles, quadrilaterals, polygons, tetrahedra, hexahedra, prisms, etc.

A component-based approach to the algorithm pipeline was adopted due to issues encountered with some of our previous meshing algorithms implementations. By explicitly expressing the data used by each step (so-called *components*), we keep track of the data modifications and inconsistencies that appear inside a component and do not carry along every possible piece of data as global variables or as attributes to an overblown class. We feel that it also provides better re-usability and unit testing.

In Figure A.1 is depicted the pipeline of our initial solution, where algorithmic components are displayed in black and data components in green. Among all the data components we only show the connectivities in order to illustrate the benefits of having written GMDS with a non-fixed connectivity model, as different steps require different sets of connectivities, depending on how they were implemented. In orange are selection components, as some steps only apply to a mesh subset. On this figure, a `X2Y` connectivty means we store the connectivity from `X` cells to `Y` cells. We use `X=N` for nodes, or 0-cells, `X=E` for edges, or 1-cells, `X=F` for faces, or 2-cells, and eventually `X=R` for regions, or 3-cells.

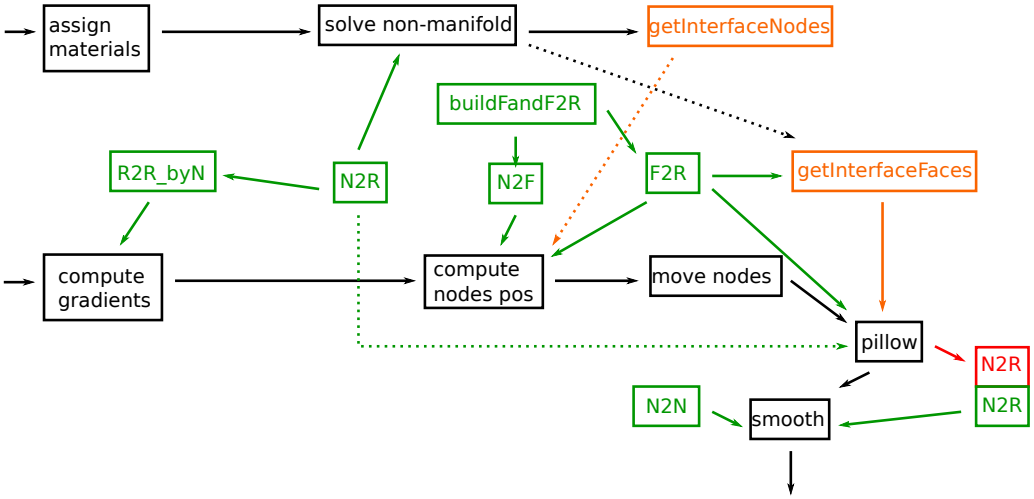


Figure A.1: Our base pipeline displayed as components. In red is shown that the N2R connectivity needed by the smoother is rendered inconsistent by the pillowing step (topological modifications) and needs to be generated anew. The 2D pipeline is similar, with edges in place of faces.

This approach proved helpful, especially when one is not only developing a known fixed algorithm but designing the algorithm itself by trial and error and experimenting with modifications and different assemblies (see the amount of modifications we introduced to the base pipeline in Figure 6.1 page 104).

Figure A.2 also illustrates the many adjacency connectivities needed across our pipeline. While our individual steps implementations each require a limited set of connectivities, the union of all the necessary connectivities is practically the complete set of all the possible ones [1]. One should also note that changing the implementation of one of the steps, or inserting another step, could lead to having to add or remove some of those connectivities.

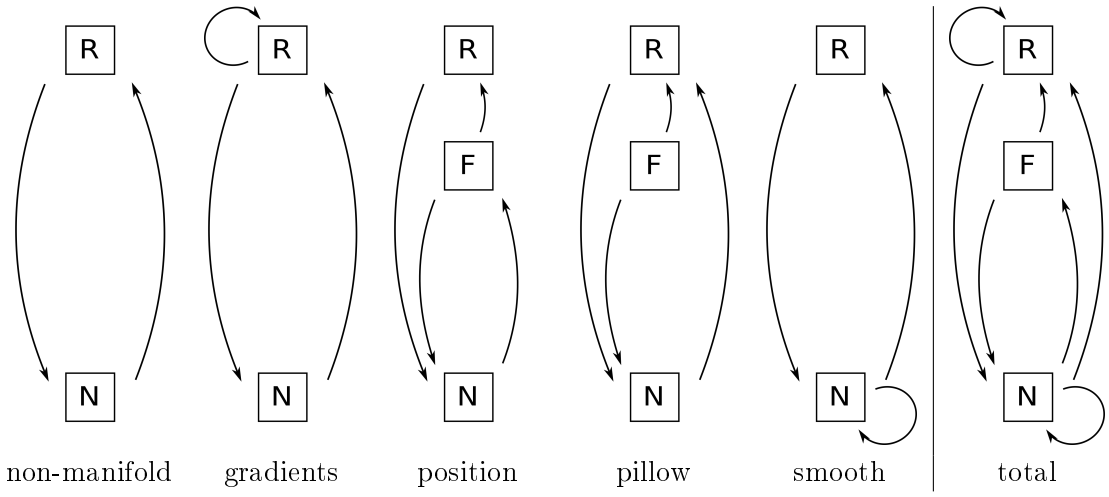


Figure A.2: Adjacency connectivities required for the different steps of the pipeline, with the aggregate on the right.

[1] Some adjacencies are ambiguous; **R2R** usually stands for the regions that share a face while in our case we denote it **R2R_byN** because we needed this adjacency to be the sharing of a node. **N2N** here is when two nodes are linked by an edge (which we do not explicitly represent in our algorithm).

Appendix B

A few words about our parallel implementation

So as to help develop the ELG pipeline, we enriched our mesh data structure GMDS [Ledoux, Jean-Claude Weill, et al. 2008] by providing a set of thread safe functionalities in order to be able to run parallel algorithms. We limited ourselves to a shared-memory implementation, and chose the Kokkos [Carter Edwards et al. 2014] framework as an abstraction layer to work with.

The parallelism strategy we used is of type BSP[1] [Valiant 1990], where when necessary the concurrency between tasks is expressed as edges in a graph from which we extract an independent set, or a coloring when relevant, to work on.

A performances analysis of the base pipeline (see Figure A.1) was realized on three different hardware setups. The hardware used was composed of nodes equipped with "classic" CPUs (Haswell and Skylake) and one manycore setup with a lot of cores at low frequency (KNL):

- Haswell : Intel Xeon E5-2698 v3 (2×16 cores at 2.3ghz, 2HT/core);
- KNL : Intel Xeon-Phi (68 cores in total, 64 available, at 1.4ghz, 4HT/core), with MCDRAM configured in “flat” mode;
- Skylake : Intel Xeon Platinum 8168 (2×24 cores at 2.7ghz, 2HT/core).

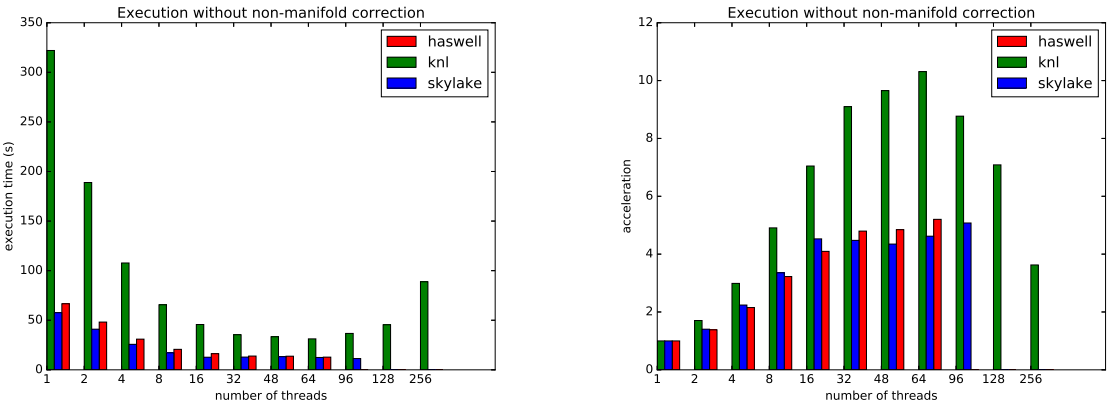
This study was presented in [Le Goff, Ledoux, and Janodet 2018] and a few points deserve to be highlighted here. In Figure B.1 are presented the overall execution time and acceleration of our implementation[2] where we obtained a maximum acceleration of 5 for the total execution time on the Haswell and Skylake setups and 10 on the KNL.

But the total acceleration displayed hides huge discrepancies; despite being part of the same pipeline the phases that compose it are wildly different in nature, and so is the acceleration obtained. In Figure B.2 is shown the results of a computationally intensive step, the computation of the gradients, where the acceleration reaches 10 and 30 depending on the setup.

The lower overall acceleration can be explained by measuring phases of the pipeline with a less favorable ratio of computations over data accesses. In Figure B.3 is displayed the time spent during the gradients computation, the same as Figure B.2.a, in addition to the time spent in building the connectivities necessary for this computation in lighter greens, the $\mathbf{N2R}$ and $\mathbf{R2R\_byN}$ adjacency connectivities (see Figure A.1; note that the usage of the $\mathbf{N2R}$ connectivity is not limited to computing the gradients and appears in other components). Those three steps all have different performance profiles and sweet spots attained at a different number of threads, and our measurements highlight the fact that some inexpensive steps can become preponderant depending on the number of threads (compare the time spent building $\mathbf{N2R}$ at 2 and 256 threads).

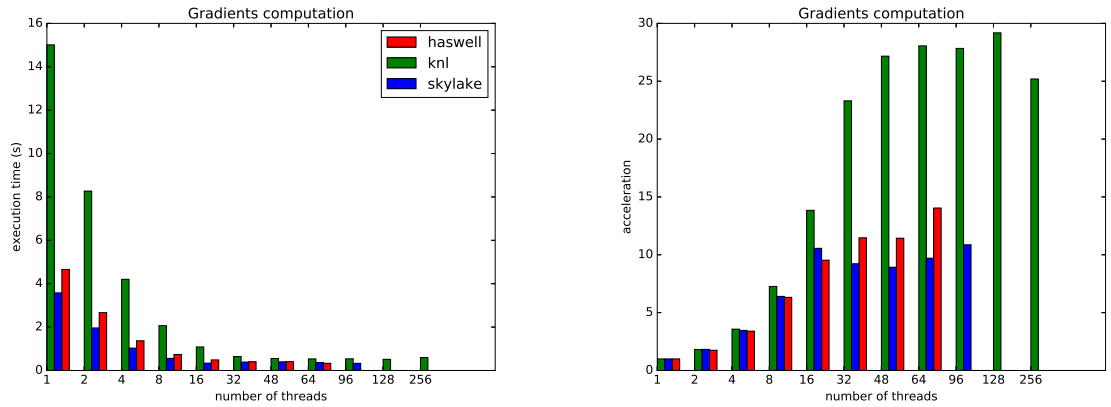
[1] Bulk-Synchronous Parallel

[2] We excluded the non-manifold resolution phase from the measurement because, contrary to the other steps where there is roughly always the same amount of computations, our implementation of it makes its execution dependent on the order in which the nodes are treated, and as treating a node can cascade into producing other non-manifold configurations the total number of resolutions and iterations fluctuates a lot between executions.



(a) (b)

Figure B.1: Execution time and acceleration of the base pipeline (excluding the non-manifold resolution step).



(a) (b)

Figure B.2: Execution time and acceleration of the computation of the gradients step.

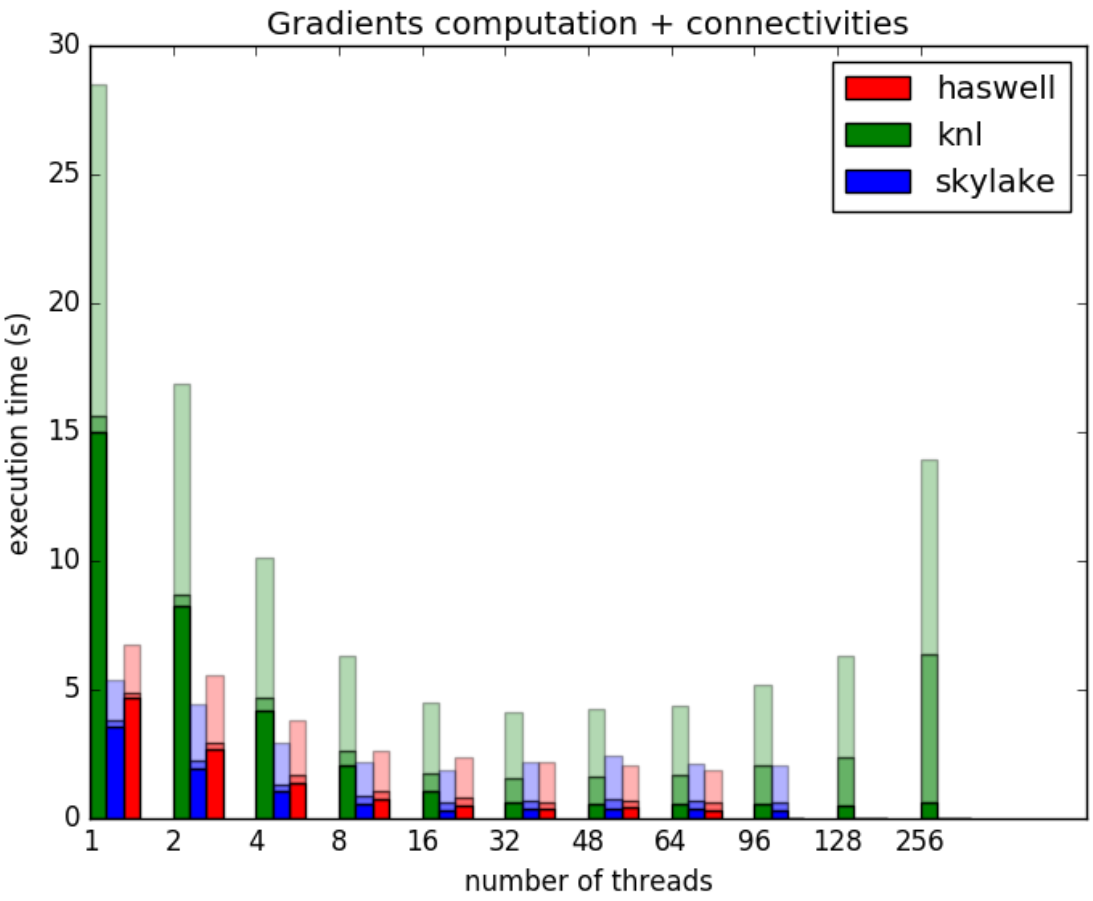


Figure B.3: Execution time of the gradients computation (bottom green) plus building the connectivities N2R (middle green) and R2R_byN (top green).

Appendix C

French summary – résumé en français

Construction d'un maillage hexaédrique conforme à partir d'une grille de fractions de présence: étude et applications

Les codes de simulation numérique reposent sur la discrétisation du domaine de calcul en éléments simples, appelés *cellules* ou *mailles*, qui forment un *maillage*. Ces codes peuvent imposer des contraintes sur les maillages afin de pouvoir tout simplement fonctionner, améliorer les résultats ou les obtenir plus vite en réduisant le temps de simulation : ces contraintes concernent classiquement le type de mailles (triangles, quadrangles, hexaèdres,...), la qualité des mailles (principalement leur forme), l'alignement aux interfaces, leur taille,... La génération du maillage est souvent un processus long et réalisé à la main à l'aide d'outils intéractifs, très consommateur en temps ingénieur, ce d'autant plus lorsque l'objectif est de produire des maillages composés de mailles quadrangulaires en 2D, ou hexaédriques en 3D. Étant donné un volume de forme quelconque et des contraintes souhaitées sur le maillage, il n'existe pas de méthode automatisée permettant de générer ce type de maillages à coup sûr [Sarrate et al. 2014].
Une étude industrielle peut mener à étudier le comportement d'un système soumis à plusieurs phénomènes physiques distincts, et ainsi parfois faire appel successivement à plusieurs codes de calcul différents, chacun avec ses propres particularités. Notre travail poursuit l'objectif de fournir un service de type intercode, qui est de convertir les données issues d'un premier code de calcul eulérien en entrées pour un second code lagrangien. Plus précisément, à partir d'un maillage hexaédrique portant des fractions volumiques nous avons deux sous-objectifs :

1. Extraire un modèle géométrique permettant d'être importé dans un logiciel de CAO-maillage, ou bien de faire tourner des algorithmes de génération de maillage hexaédrique dessus. Il doit être le plus simple possible (pas de petites surfaces isolées) et lisse afin de pouvoir être utilisable ;

2. Éventuellement directement générer automatiquement un maillage valide. Bien que les contraintes concernant le maillage soient propres à chaque code, un critère de qualité commun est d'avoir des mailles qui ne soient pas trop déformées géométriquement [P. M. Knupp 2001].

Une contrainte sous-jacente est que notre résultat ne soit pas trop "éloigné" des données passées en entrée.

Nous nous sommes alors tournés vers des méthodes d'*intersection de grille* (ou *overlay grid*) [R. Schneiders 1996]. Comparées aux autres techniques de génération de maillages hexaédriques (méthodes de recombinaison de tétraèdres, avancée de front, champ d'orientation, axe médian, ...) celles-ci ont l'avantage d'être totalement automatisées et de fonctionner sur un large panel de modèles, en n'ayant besoin que d'un nombre restreint de paramètres fixés par l'utilisateur. Le principe de ces méthodes est de commencer par créer un maillage hexaédrique "facile" à générer, habituellement une grille, qui recouvre le domaine à mailler. Les étapes suivantes consistent à sélectionner et garder les cellules qui feront parties du volume, puis à modifier (topologiquement et géométriquement) ce sous-ensemble du maillage initial pour capturer les éléments du modèle géométrique, pour au final obtenir le maillage demandé.
Mais ces méthodes ont des incompatibilités avec le but poursuivi. Premièrement, les méthodes par *intersection de grille* comme [Maréchal 2009; Zhang 2016] prennent en entrée un modèle géométrique,

ce qui n'est pas ce dont nous disposons. Deuxièmement, la plupart de ces méthodes ne considèrent qu'un seul volume, et non pas un modèle géométrique qui soit un assemblage de plusieurs pièces. Troisièmement, la robustesse de ces méthodes repose sur la capacité à créer un maillage initial adapté, généralement raffiné par endroit pour mieux capturer le modèle de CAO ou bien pour s'en approcher au mieux [X. Gao, Shen, et al. 2019], ce qui est en conflit direct avec le fait que notre maillage d'entrée est pour nous un paramètre qui nous est donné, qu'il convienne ou pas.

Il y a néanmoins parmis ces méthodes l'algorithme SCULPT [Owen, M. L. Staten, and Sorensen 2012; Owen, Brown, et al. 2017] qui prend en entrée un maillage eulérien; nous l'avons adapté et étendu afin qu'il répondre à nos besoins, ce qui nous a conduit à un nouvel algorithme que nous avons appelé ELG pour *Euler to LaGrange remeshing* qui fonctionne à la fois en dimension 2 et 3.

Les principales contributions de cette thèse sont :

- Mesure et amélioration de la préservation du volume. Nous avons introduit en post-traitement à SCULPT un critère appelé $discrepancy$ mesurant l'écart entre le maillage eulérien d'entrée et le maillage lagrangien, ainsi qu'un algorithme piloté par cet écart et permettant de le réduire; ces travaux sont abordés dans le Chapitre 3. Nous avons montré sur un large panel d'exemples venant de modèles STL, de modèles de CAO simples et complexes, de modèles générés procéduralement ainsi que de cas issus de simulations numériques, que notre post-traitement était capable de réduire cette $discrepancy$ en général d'un ordre de grandeur ;

- Reconstruction d'interfaces discrètes et utilisation du modèle géométrique extrait. Nous comparons dans le Chapitre 4 des méthodes permettant de reconstruire les interfaces entre matériaux à partir d'une voxelisation du maillage eulérien. avons élaboré une méthode qui se comporte bien à la fois sur la $discrepancy$ et le $edgecut$ (tel que défini dans les problèmes de partitionnement de graphes) y compris dans des cas où le maillage d'entrée est non-structuré. Nous montrons également comment en extraire un modèle géométrique et l'utiliser pour poursuivre notre objectif ;

- Garantie de qualité sur le maillage. Nous avons modifié notre algorithme dans la Section 5.1 afin que le maillage résultant ait une qualité minimale garantie et ne passe jamais en-dessous d'un seuil choisi par l'utilisateur. À cette occasion nous mettons à profit le modèle géométrique extrait dans le chapitre précédent, et présentons une méthode permettant de faire du *pillowing* localisé, ce qui nous permet ainsi d'éviter certains des problèmes pouvant survenir lorsqu'il est appliqué de manière globale comme il est couramment pratiqué ;

- Adaptation du maillage. Dans la Section 5.2 nous proposons certaines fonctionnalités d'adaptation du maillage, similaires à ce que d'autres méthode de cette famille fournissent, mais dans notre cas sans disposer du modèle de CAO. En réutilisant notre méthode de reconstruction d'interfaces, cela donne la possibilité pour l'utilisateur de piloter les phases initiales de notre algorithme afin d'impacter la manière dont certains des matériaux peuvent être capturés, pour par exemple préserver des couches minces de matériaux d'intérêt pour la simulation.

References

Ahrens, J., Berk Geveci, and Charles Law (2005). “ParaView: An End-User Tool for Large Data Visualization”. In: *Visualization Handbook*.

Anderson, J. C., C. Garth, M. A. Duchaineau, and K. I. Joy (2008). “Discrete Multi-Material Interface Reconstruction for Volume Fraction Data”. In: *Computer Graphics Forum* 27.3, pp. 1015–1022.

— (2010). “Smooth, Volume-Accurate Material Interface Reconstruction”. In: *IEEE Transactions on Visualization and Computer Graphics* 16.5, pp. 802–814.

Bangerth, W., R. Hartmann, and G. Kanschat (2007). “deal.II – A general-purpose object-oriented finite element library”. In: *ACM Transactions on Mathematical Software* 33.4, 24–es.

Barat, Remi (2017). “Load Balancing of Multi-physics Simulation by Multi-criteria Graph Partitioning”. These de doctorat. Bordeaux.

Baudouin, Tristan Carrier, Jean-François Remacle, Emilie Marchandise, François Henrotte, and Christophe Geuzaine (2014). “A frontal approach to hex-dominant mesh generation”. In: *Advanced Modeling and Simulation in Engineering Sciences* 1.1, p. 8.

Berkelaar, Michel, Kjell Eikland, and Peter Notebaert (2004). *lp_solve*. Version 5.1.0.0.

Blacker, Ted (2000). “Meeting the Challenge for Automated Conformal Hexahedral Meshing”. In: *9th International Meshing Roundtable*, pp. 11–20.

Blacker, Ted D. (1997). “The Cooper Tool”. In: *Proceedings of the 5th International Meshing Roundtable*, pp. 217–228.

Blacker, Ted D. and Ray J. Meyers (1993). “Seams and wedges in plastering: A 3-D hexahedral mesh generation algorithm”. In: *Engineering with Computers* 9.2, pp. 83–93.

Blacker, Ted D. and Michael B. Stephenson (1991). “Paving: A new approach to automated quadrilateral mesh generation”. In: *International Journal for Numerical Methods in Engineering* 32.4, pp. 811–847.

Boggs, Paul T., Alan (Exagrid Engineering) Althsuler, Alex R. (Exagrid Engineering) Larzelere, Edward J. Walsh, Ruuobert L. Clay, and Michael F. (Sandia National Laboratories Hardwick (2005). *DART system analysis.* Report. Library Catalog: digital.library.unt.edu Number: SAND2005-4647 Publisher: Sandia National Laboratories.

Boykov, Y. and V. Kolmogorov (2004). “An experimental comparison of min-cut/max- flow algorithms for energy minimization in vision”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.9, pp. 1124–1137.

Boykov, Y., O. Veksler, and R. Zabih (2001). “Fast approximate energy minimization via graph cuts”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 23.11, pp. 1222–1239.

Cai, Shengyong and Timothy J. Tautges (2015). “Optimizing Corner Assignment of Submap Surfaces”. In: *Procedia Engineering*. 24th International Meshing Roundtable 124, pp. 83–95.

Calderan, Simon, Guillaume Hutzler, and Franck Ledoux (2019). “Dual-Based User-Guided Hexahedral Block Generation Using Frame Fields”. In: Proceedings of the 28th International Meshing Roundtable. Library Catalog: Zenodo Publisher: Zenodo.

Carter Edwards, H., Christian R. Trott, and Daniel Sunderland (2014). “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns”. In: *Journal of Parallel and Distributed Computing*. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing 74.12, pp. 3202–3216.

CEA (2020). *Première expérience de fusion nucléaire au Laser Mégajoule*.

Cherchi, G., P. Alliez, R. Scateni, M. Lyon, and D. Bommes (2019). “Selective Padding for Polycube-Based Hexahedral Meshing”. In: *Computer Graphics Forum* 38.1, pp. 580–591.

Childs, Hank, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, David Pugmire, Kathleen Biagas, Mark Miller, Cyrus Harrison, Gunther H Weber, Hari Krishnan, Thomas Fogal, Allen Sanderson, Christoph Garth, E. Wes Bethel, David Camp, Oliver Rübel, Marc Durant, Jean M. Favre, and Paul Návratil (2012). “VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data”. In: *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pp. 357–372.

CPLEX Optimizer (2020).

Cubit (2019). *Sandia National Laboratories: CUBIT Geometry and Mesh Generation Toolkit*.

Damiand, Guillaume and Pascal Lienhardt (2014). *Combinatorial Maps: Efficient Data Structures for Computer Graphics and Image Processing*. A K Peters/CRC Press.

Distene (2020). *Distene's MeshGems suite | Meshing Software Components for CAD and CAE applications from Distene*.
Faux, I. D. and M. J. Pratt (1979). *Computational Geometry for Design and Manufacture*. USA: Halsted Press. 329 pp.
Ferziger, Joel H. and Milovan Peric (2002). *Computational Methods for Fluid Dynamics*. 3rd ed. Berlin Heidelberg: Springer-Verlag.
Fiduccia, C. M. and R. M. Mattheyses (1982). "A linear-time heuristic for improving network partitions". In: *Proceedings of the 19th Design Automation Conference*. DAC '82. IEEE Press, pp. 175–181.
Folwell, N. and Scott Mitchell (1999). "Reliable Whisker Weaving via Curve Contraction". In: *Engineering With Computers* 15, pp. 292–302.
Freitag, Lori A. (1997). "On Combining Laplacian And Optimization-Based Mesh Smoothing Techniques". In: *Trends in Unstructured Mesh Generation*, pp. 37–43.
Frey, Pascal and Paul George (2008). "Mesh Generation: Application to Finite Elements: Second Edition". In: *Mesh Generation: Application to Finite Elements: Second Edition*.
Gao, Xifeng, Daniele Panozzo, Wenping Wang, Zhigang Deng, and Guoning Chen (2017). "Robust structure simplification for hex re-meshing". In: *ACM Transactions on Graphics* 36.6, 185:1–185:13.
Gao, Xifeng, Hanxiao Shen, and Daniele Panozzo (2019). "Feature Preserving Octree-Based Hexahedral Meshing". In: *Computer Graphics Forum* 38.5, pp. 135–149.
Geuzaine, Christophe and Jean-François Remacle (2009). "Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities". In: *International Journal for Numerical Methods in Engineering* 79.11. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.2579, pp. 1309–1331.
GLPK (2019). *GLPK - GNU Project - Free Software Foundation (FSF)*.
Gregson, James, Alla Sheffer, and Eugene Zhang (2011). "All-Hex Mesh Generation via Volumetric PolyCube Deformation". In: *Computer Graphics Forum*. Special Issue of Symposium on Geometry Processing 2011 30.5. Publisher: Wiley, pp. 1407–1416.
Gurobi Optimization, LLC (2020). *Gurobi Optimizer Reference Manual*.
Guy, Ryan (2019). *A PIC/FLIP fluid simulation based on the methods found in Robert Bridson's "Fluid Simulation for Computer Graphics": rlguy/GridFluidSim3D*. original-date: 2015-04-07T05:54:12Z.
He, Ying, Hongyu Wang, Chi-Wing Fu, and Hong Qin (2009). "A divide-and-conquer approach for automatic polycube map construction". In: *Computers & Graphics*. IEEE International Conference on Shape Modelling and Applications 2009 33.3, pp. 369–380.
Hege, Hans-Christian, Martin Seebass, Detlev Stalling, and Malte Zöckler (1997). "A Generalized Marching Cubes Algorithm Based on Non-Binary Classifications". In:
Herring, Angela, Ondrej Certik, Charles Ferenbaugh, Rao Garimella, Brian Jean, Chris Malone, and Chris Sewell (2017). *(U) Introduction to Portage*. Los Alamos National Laboratory (LANL), p. 9.
Hughes, Thomas (2000). *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Vol. 78.
Hughes, Thomas J. R. (2004). *Consider a Spherical Cow - Conservation of Geometry in Analysis: Implications for Computational Methods in Engineering | Institute for Mathematics and its Applications*. URL: https://www.ima.umn.edu/2003-2004/SW5.11-15.04/23254 (visited on 08/19/2020).
ICEM, CFD (2013). *Computational Fluid Dynamics (CFD) Simulation | Ansys*.
Kernighan, B. W. and S. Lin (1970). "An efficient heuristic procedure for partitioning graphs". In: *The Bell System Technical Journal* 49.2. Conference Name: The Bell System Technical Journal, pp. 291–307.
Knupp, Patrick (1998). "Next-Generation Sweep Tool: A Method For Generating All-Hex Meshes On Two-And-One-Half Dimensional Geomtries". In: *7th International Meshing Roundtable*, pp. 505–513.
Knupp, Patrick M. (2001). "Algebraic Mesh Quality Metrics". In: *SIAM J. Sci. Comput.* 23.1, pp. 193–218.
Kolmogorov, V. and R. Zabin (2004). "What energy functions can be minimized via graph cuts?" In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.2, pp. 147–159.
Kowalski, N., F. Ledoux, and P. Frey (2016). "Smoothness driven frame field generation for hexahedral meshing". In: *Computer-Aided Design*. 23rd International Meshing Roundtable Special Issue: Advances in Mesh Generation 72, pp. 65–77.
Kucharik, Milan, Rao V. Garimella, Samuel P. Schofield, and Mikhail J. Shashkov (2010). "A comparative study of interface reconstruction methods for multi-material ALE simulations". In: *Journal of Computational Physics* 229.7, pp. 2432–2452.
Lai, Mingwu, Steven Benzley, and David White (2000). "Automated hexahedral mesh generation by generalized multiple source to multiple target sweeping". In: *International Journal for Numerical Methods in Engineering* 49.1, pp. 261–275.
Le Goff, Nicolas, Franck Ledoux, and Jean-Christophe Janodet (2018). "A Parallel Shared-Memory Implementation of an Overlay Grid Method". oral talk. Symposium on Trends in Unstructured Mesh Generation.
— (2019a). "An Overlay Grid Driven Geometric Model Extraction". oral talk. Symposium on Trends in Unstructured Mesh Generation.
— (2019b). "Hexahedral Overlay Grid Method with Guaranteed Element Quality". oral talk. International Conference on Adaptive Modeling and Simulation.

Le Goff, Nicolas, Franck Ledoux, Jean-Christophe Janodet, and Steven J. Owen (2019). “Guaranteed quality-driven hexahedral overlay grid method”. In: *Proceedings of the 28th International Meshing Roundtable*.

Le Goff, Nicolas, Franck Ledoux, and Steven J. Owen (2018). “Hexahedral mesh modification to preserve volume”. In: *Computer-Aided Design* 105, pp. 42–54.

Ledoux, Franck (2014). “Hexahedral meshing for numerical simulation: representations and algorithms”. habilitation à diriger des recherches. University of Poitiers: University of Poitiers.

— (2018). “Paving the Path Towards Automatic Hexahedral Mesh Generation”. keynote. 27th International Meshing Roundtable, IMR27. Albuquerque.

Ledoux, Franck and Jean-Christophe Weill (2007). “An Extension of the Reliable Whisker Weaving Algorithm”. In: Proceedings of 16th International Meshing Roundtable, pp. 215–232.

Ledoux, Franck, Jean-Claude Weill, and Yves Bertrand (2008). “GMDS: A Generic Mesh Data Structure”. In: *17th International Meshing Roundtable*. United States, ??

Liu, Yutong, Kerem Pekkan, S. Casey Jones, and Ajit P. Yoganathan (2004). “The Effects of Different Mesh Generation Methods on Computational Fluid Dynamic Analysis and Power Loss Assessment in Total Cavopulmonary Connection”. In: *Journal of Biomechanical Engineering* 126.5. Publisher: American Society of Mechanical Engineers Digital Collection, pp. 594–603.

Lu, Jean Hsiang-Chun, Inho Song, William Roshan Quadros, and Kenji Shimada (2014). “Geometric reasoning in sketch-based volumetric decomposition framework for hexahedral meshing”. In: *Engineering with Computers* 30.2, pp. 237–252.

Maréchal, Loïc (2009). “Advances in Octree-Based All-Hexahedral Mesh Generation: Handling Sharp Features”. In: *Proceedings of the 18th International Meshing Roundtable*. Ed. by Brett W. Clark. Springer Berlin Heidelberg, pp. 65–84.

— (2016). “All Hexahedral Boundary Layers Generation”. In: *Procedia Engineering*. 25th International Meshing Roundtable 163, pp. 5–19.

Morais, Sébastien (2016). “Etude et obtention d’heuristiques et d’algorithmes exacts et approchés pour un problème de partitionnement de maillage sous contraintes mémoire. (Study and obtention of exact, and approximation, algorithms and heuristics for a mesh partitioning problem under memory constraints)”. PhD thesis. University of Paris-Saclay, France.

Murdoch, Peter, Steven Benzley, Ted Blacker, and Scott A. Mitchell (1997). “The spatial twist continuum: A connectivity based method for representing all-hexahedral finite element meshes”. In: *Finite Elements in Analysis and Design* 28.2, pp. 137–149.

Owen, Steven J. (2005). “An introduction to mesh generation algorithms”. short course. 14th International Meshing Roundtable. San Diego.

Owen, Steven J., Judith A. Brown, Corey D. Ernst, Hojun Lim, and Kevin N. Long (2017). “Hexahedral Mesh Generation for Computational Materials Modeling”. In: *Procedia Engineering*. 26th International Meshing Roundtable, IMR26, 18-21 September 2017, Barcelona, Spain 203, pp. 167–179.

Owen, Steven J. and Sunil Saigal (2000). “H-Morph: an indirect approach to advancing front hex meshing”. In: *International Journal for Numerical Methods in Engineering* 49.1, pp. 289–312.

Owen, Steven J. and Tim R. Shelton (2015). “Evaluation of grid-based hex meshes for solid mechanics”. In: *Engineering with Computers* 31.3, pp. 529–543.

Owen, Steven J. and Jason F. Shepherd (2009). “Embedding Features in a Cartesian Grid”. In: *Proceedings of the 18th International Meshing Roundtable*. Ed. by Brett W. Clark. Springer Berlin Heidelberg, pp. 117–138.

Owen, Steven J., Ryan M. Shih, and Corey D. Ernst (2017). “A template-based approach for parallel hexahedral two-refinement”. In: *Computer-Aided Design*. 24th International Meshing Roundtable Special Issue: Advances in Mesh Generation 85, pp. 34–52.

Owen, Steven J., Matthew L. Staten, Scott Canann, and Sunil Saigal (1999). “Q-Morph: an indirect approach to advancing front quad meshing”. In: *International Journal for Numerical Methods in Engineering* 44.9, pp. 1317–1340.

Owen, Steven J., Matthew L. Staten, and Marguerite C. Sorensen (2012). “Parallel Hex Meshing from Volume Fractions”. In: *Proceedings of the 20th International Meshing Roundtable*. Ed. by William Roshan Quadros. Springer Berlin Heidelberg, pp. 161–178.

Palmer, David, David Bommes, and Justin Solomon (2020). “Algebraic Representations for Volumetric Frame Fields”. In: *ACM Transactions on Graphics* 39.2, 16:1–16:17.

Papadimitrakis, Dimitrios, Cecil G. Armstrong, Trevor T. Robinson, Alan Le Moigne, and Shahrokh Shahpar (2019). “Building Direction Fields on the Medial Object to Generate 3D Domain Decompositions for Hexahedral Meshing”. In: *Proceedings of the 28th International Meshing Roundtable*. Publisher: Zenodo.

Pellerin, Jeanne, Amaury Johnen, Kilian Verhetsel, and Jean-François Remacle (2018). “Identifying combinations of tetrahedra into hexahedra: A vertex based strategy”. In: *Computer-Aided Design* 105, pp. 1–10.

Powell, Devon and Tom Abel (2015). “An exact general remeshing scheme applied to physically conservative voxelization”. In: *Journal of Computational Physics* 297, pp. 340–356.

Quadros, William Roshan (2014). “LayTracks3D: A New Approach to Meshing General Solids using Medial Axis Transform”. In: *Procedia Engineering*. 23rd International Meshing Roundtable (IMR23) 82, pp. 72–87.

Ruiz-Gironés, E. and J. Sarrate (2010a). “Generation of structured hexahedral meshes in volumes with holes”. In: *Finite Elements in Analysis and Design* 46.10, pp. 792–804.
— (2010b). “Generation of structured meshes in multiply connected surfaces using submapping”. In: *Advances in Engineering Software* 41.2, pp. 379–387.
Ruiz-Gironés, Eloi, Xevi Roca, and Josep Sarrate (2009). “A New Procedure to Compute Imprints in Multi-sweeping Algorithms”. In: *Proceedings of the 18th International Meshing Roundtable*. Ed. by Brett W. Clark. Berlin, Heidelberg: Springer, pp. 281–299.
Sarrate, J., E. Ruiz-Gironés, and X. Roca (2014). “Unstructured and Semi-Structured Hexahedral Mesh Generation Methods”. In: *Computational Technology Reviews* 10, pp. 35–64.
Schneiders, R. (1996). “A grid-based algorithm for the generation of hexahedral element meshes”. In: *Engineering with Computers* 12.3, pp. 168–177.
Schneiders, Robert, R. Schindler, and F. Weiler (1999). “Octree-based Generation of Hexahedral Element Meshes”. In: *Proceedings of the 5th International Meshing Roundtable*.
Si, Hang (2015). “TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator”. In: *ACM Transactions on Mathematical Software* 41.2, 11:1–11:36.
Staten, Matt, Brian Carnes, Corey Mcbride, Clint Stimpson, and Jim Cox (2016). “Mesh Scaling for Affordable Solution Verification”. In: *Procedia Engineering*. 25th International Meshing Roundtable 163, pp. 46–58.
Staten, Matthew L. and Steven J. Owen (2010). *Parallel octree-based hexahedral mesh generation for eulerian to lagrangian conversion.* Report. Library Catalog: digital.library.unt.edu Number: SAND2010-6400 Publisher: Sandia National Laboratories.
Takayama, Kenshi (2019). “Dual Sheet Meshing: An Interactive Approach to Robust Hexahedralization”. In: *Computer Graphics Forum* 38.2. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13617, pp. 37–48.
Tam, T. K. H. and C. G. Armstrong (1991). “2D finite element mesh generation by medial axis subdivision”. In: *Advances in Engineering Software and Workstations* 13.5, pp. 313–324.
Toro, Eleuterio (2009). “Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction”. In: *Riemann Solvers and Numerical Methods for Fluid Dynamics*.
Valiant, Leslie G. (1990). “A Bridging Model for Parallel Computation”. In: *Commun. ACM* 33.8, pp. 103–111.
Vartziotis, Dimitris and Joachim Wipper (2012). “Fast smoothing of mixed volume meshes based on the effective geometric element transformation method”. In: *Computer Methods in Applied Mechanics and Engineering* 201-204, pp. 65–81.
White, David R., Sunil Saigal, and Steven J. Owen (2004). “CCSweep: automatic decomposition of multi-sweep volumes”. In: *Engineering with Computers* 20.3, pp. 222–236.
White, David Roger (1996). *Automatic, Quadrilateral and Hexahedral Meshing of Pseudo-Cartesian Geometries Using Virtual Subdivision*. Google-Books-ID: p91MtwAACAAJ. Brigham Young University. Department of Civil and Environmental Engineering. 130 pp.
Whiteley, M., D. White, S. Benzley, and T. Blacker (1996). “Two and three-quarter dimensional meshing facilitators”. In: *Engineering with Computers* 12.3, pp. 144–154.
Wu, Haiyan and Shuming Gao (2014). “Automatic Swept Volume Decomposition based on Sweep Directions Extraction for Hexahedral Meshing”. In: *Procedia Engineering*. 23rd International Meshing Roundtable (IMR23) 82, pp. 136–148.
Yu, Wuyi, Kang Zhang, Shenghua Wan, and Xin Li (2014). “Optimizing polycube domain construction for hexahedral remeshing”. In: *Computer-Aided Design*. 2013 SIAM Conference on Geometric and Physical Modeling 46, pp. 58–68.
Zhang, Yongjie Jessica (2016). *Geometric Modeling and Mesh Generation from Scanned Images*. 1 p.
Zienkiewicz, O. C., J. Rojek, R. L. Taylor, and M. Pastor (1998). “Triangles and tetrahedra in explicit dynamic codes for solids”. In: *International Journal for Numerical Methods in Engineering* 43.3, pp. 565–583.
Zienkiewicz, O. C., R. L. Taylor, and J. Z. zhu (2013). *The Finite Element Method: its Basis and Fundamentals*. Elsevier.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Construction d'un maillage hexaédrique conforme à partir d'une grille de fractions de présence: étude et applications

Mots clés: maillage, hexaédrique, intersection de grille, intercode, fractions volumiques, Euler vers Lagrange

Résumé: Ces travaux abordent le problème de la génération automatique de maillages hexaédriques pour des codes de simulation, à partir d'un maillage portant des fractions volumiques, c'est-à-dire dont les mailles peuvent contenir plusieurs matériaux. La solution proposée doit construire un maillage hexaédrique dans lequel chaque maille correspond à un seul matériau, et dont les interfaces entre matériaux doivent former des surfaces lisses. D'un point de vue théorique, nous cherchons à adapter et étendre des solutions existantes, et à les appliquer sur une large variété d'exemples : certains issus de modèles de CAO (plaqués sur un maillage pour obtenir des fractions volumiques), d'autres générés procéduralement et enfin d'autres utilisés dans un rôle d'intercode, récupérés en sortie de codes de simulation. Nous définissons une métrique permettant d'évaluer notre (et d'autres) méthodes, ainsi qu'un post-traitement pour améliorer ces résultats; nous introduisons également une méthode de reconstruction d'interfaces discrètes inspirée de méthodes issues du domaine de la visualisation scientifique, et nous proposons un algorithme appelé ELG avec garantie sur la qualité du maillage, faisant intervenir des modifications géométriques et topologiques sur ce maillage.

Title: Construction of a conformal hexahedral mesh from volume fractions: theory and applications

Keywords: mesh, hexahedral, overlay grid, intercode, volume fractions, Eulerian to Lagrangian

Abstract: This thesis addresses the problem of the automatic generation of purely hexahedral meshes for simulation codes when having a mesh carrying volume fraction data as an input, meaning that there can be several materials inside one cell. The proposed approach should create an hexahedral mesh where each cell corresponds to a single material, and where interfaces between materials form smooth surfaces. From a theoretical standpoint, we aim at adapting and extending state-of-the-art techniques and we apply them on examples, some classically issued from CAD models (and imprinted onto a mesh to obtain volume fractions), some procedurally generated cases and others in an intercode capacity where we take the results of a first simulation code to be our inputs. We first define a metric that allows the evaluation of our (or others') results and a method to improve those; we then introduce a discrete material interface reconstruction method inspired from the scientific visualization field and finally we present an algorithmic pipeline, called ELG, that offers a guarantee on the mesh quality by performing geometrical and topological mesh adaptation.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France