Synergistic Geometry Processing: from Robust Geometric Modeling to Efficient Physical Simulations

by

Zhongshi Jiang

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

September, 2022

Professor Daniele Panozzo

Acknowledgements

Abstract

The numerical solution of partial differential equations (PDEs) portrays how a craft would behave even before it is created. In practice, a typical product begins its lifecycle from a digital specification of the shape and material. Computer simulation would then test the behavior to match the intent: the car hood's deformation under pressure or a pan's temperature when heated. In this case, a robust and accurate simulation could lift many iterations of manufacturing to the digital platform. In addition, as it is safer to conduct more stress experiments in a virtual environment, a reliable simulation also promotes safety, especially in designing biomechanical equipment.

There are various approaches to specify a shape, depending on the different geometry characteristics. Shape modeling evolves beyond smooth and regular objects like sculpture or furniture. Algorithm deduced structures have gained popularity since they deliver the promise of maximal durability with limited material. Downstream, computer simulation requires a discrete tessellation of the object. A notable example is a mesh: it contains a list of vertices for the spatial locations and a list of simple polygons or polyhedra. However, it is rarely the case where one mesh fits all. More degrees of freedom give a more accurate description of the desired shape, but it inevitably means more computational resources are needed to handle the computation.

Different representations between design methods and within various adaptations of simulations, even for the same shape, are not reliably correlated. In the general case, the material, attribute, or experiment setting cannot be robustly re-used on another representation without laborious manually fixing. Such a pipeline demands the engineer or designer to understand precisely the properties of the specific numerical methods of choice. It also prohibits an automatic and robust pipeline to compute optimal structure for the desired physical behavior.

My Ph.D. research's central thesis is to develop algorithms that reliably associate different representations of discrete geometry entities at diverse design and simulation stages. Instead of re-assigning properties at various phases of the design-simulation pipeline, or whenever the required budget is different, a good association between geometry at different stages gives more freedom to the design and development of automatic and adaptive solvers. The adaptation can be bi-directional: on the one hand, with coarse and concise tessellation, the algorithm utilizes the information associated with the original design to obtain more faithful simulations. On the other hand, in early-stage prototyping, and automatic data generation for deep learning, the principles I established allows for simplifying the shape and retaining the settings, providing a fast answer. In addition, I use the same principle to design the first large-scale and robust approach for curved mesh representation, which provides accurate representation as well as efficient physical simulations.

Besides the algorithm design, I also explore the underpinning theory to provide guarantees. Theoretical inquiry not only gives us a better understanding of digital geometry and shapes but also lays the foundation for suitable applications. On the other hand, we perform extensive numerical validations with the implemented software, involving tens of thousands of complex different geometry shapes. Finally, I also release the program implementation and detailed specifications to be open source and accessible.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Preliminaries

2 | Bijective Maps with Triangulation Augmentation

2.1 Introduction

The computation of discrete maps is a fundamental problem in computer graphics that has been extensively studied in the last three decades. The problem is challenging due to the large solution space and the non-linearity of the desired properties (in both distortion measures and constraints). Algorithms for robustly and efficiently computing locally injective (i.e. non-flipping) maps have been only recently introduced [???] and are now having a major impact in many research areas outside of traditional texture mapping, including remeshing [?], image editing [?], and cultural heritage [?].

In this chapter, we consider the problem of generating bijective maps, i.e. locally injective maps with non-intersecting boundaries This is a difficult problem, exacerbated by the fact that any pair of boundary elements could overlap, leading to non-linear constraints whose number is quadratic in the size of the boundary. This problem is usually tackled by iteratively deforming an existing map, checking for overlaps after each step, and then preventing the overlap using constraints [?] or penalty forces [?]. These methods require a spatial acceleration structure to find

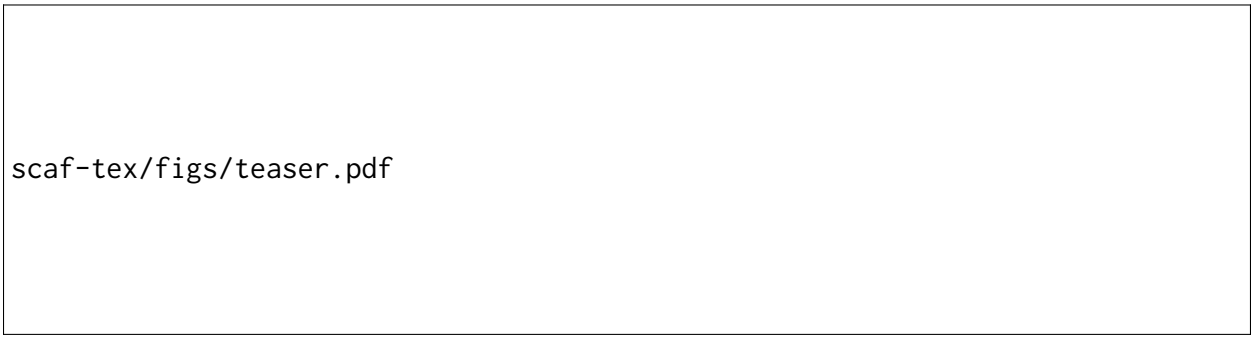


Figure 2.1: The Nefertiti model with prescribed seams is UV mapped by our algorithm. Each chart is bijective mapped into a circle or ring with Tutte's embedding and achieves minimal distortion in less than a second. The layout is further improved interactively and the final parametrized model is shown on the right. Our approach guarantees a valid UV map with no inverted elements or overlapping triangles. See the attaching video for the optimization and manual interaction.

the candidate pairs of overlapping elements and a resolution strategy that updates the map while avoiding the detected overlaps. However, the newly computed displacement might in turn lead to new overlaps, and this process has to be performed iteratively in the hope that no collisions are left. Difficult cases with many collisions might requires tens or hundreds of iterations before all the candidate intersecting pairs are detected.

Our approach sidesteps the need to find candidate self-intersections based on a simple observation: if the the entire ambient space (with a fixed simple boundary) is tessellated, then local injectivity implies global bijectivity [**???**]. We thus propose an optimization framework based on this idea making it possible to leverage recent techniques for locally injective maps; our algorithm is simple to implement, robust, and two orders of magnitude faster than competing methods.

Overview. Given an input bijective map represented by a discrete triangle mesh and its mapped vertex locations, we create a new scaffold mesh for a bounding box that contains the initial, mapped triangle mesh (Figure 2.1) and conforms to its boundary. We then optimize for the desired property of the map (such as distortion, positional constraints, etc.) while ensuring that no triangle will flip. This property is achieved using a variational formulation that combines a user-defined energy for the map with a regularization term that allows the scaffold to freely deform without hindering the optimization of the map properties. During the optimization, we refine and optimize the connectivity of the scaffold mesh to prevent possible locking situations.

While a scaffold mesh has been already used in previous works [**???**], we propose to use an isometric distortion energy on the scaffold mesh, with the reference reset to the current rest pose at each iteration, and an online remeshing strategy. Our scaffold energy aims for "isometry to the current iteration", which resembles how plasticity is usually modeled in elasto-plastic simulations — this leads to a global and natural deformation of the scaffold elements that opens up space for the evolving boundary and allows for an efficient optimization using recent numerical methods for locally injective maps [**?**].

We demonstrate the practical utility of our algorithm in the context of single patch mesh parametrization by producing distortion minimizing bijective maps for a collection of 119 challenging models. Our algorithm is ideal to compute tight UV maps for models with multiple connected components and seams as we demonstrate in our interactive texture packing experiments. Our algorithm can also be easily extended to 3D by replacing the triangular scaffold with one composed of tetrahedra. For the 3D case, we show that our method can be used to deform surfaces preventing self-intersections and to remove self-intersection from existing genus-0 surfaces when paired with a mean conformalized flow [**??**].

In the additional material, we provide a video (showing the optimization iterations) and the input/output meshes for each figure in the paper. To foster replicability of results, we will release an open-source reference implementation of our algorithm.

2.2 Previous Work

Bijective maps find a host of applications in a variety of fields including physical simulation, surface deformation, and parametrization. We review only the most relevant prior works here

and refer to the following surveys for more details [???].

Locally Injective Maps.

There are many methods that focus on creating locally injective maps, which amounts to requiring that triangles maintain their orientation (i.e. they do not flip). In mesh parameterization, many flip-preventing metrics have been developed: the idea is to force the metric to diverge to infinity as triangles become degenerate, inhibiting flips. These metrics optimize various geometric properties such as angle [??] or length [?????] preservation. Similar techniques in the context of deformation have been used to add barrier functions to enforce local injectivity in deformations [?]. Our method uses these techniques to prevent flips in the scaffold.

Many methods have also been developed to optimize these distortion energies including moving one vertex at a time [?], parallel gradient descent [?], as well as other quasi-newton approaches [???]. Other approaches construct such maps by performing a change of basis, projecting to an inversion free space, and then constructing a parametrization from the result [?]. While our method could potentially use any of these optimization methods, we use [?] for its large step sizes. We elaborate on this choice in Section 2.3.1.

Bijective Maps.

In addition to injective constraints, bijective maps have the additional requirement that the boundary does not intersect. One simple method for creating a bijective map in 2D involves constraining the boundary to a convex shape such as a circle [??]. Such parametrizations guarantee a bijective map in 2D but create significant distortion. Even so, these methods are commonly used to create a valid starting point for further optimization [???]. While methods that produce bijective maps with fixed boundaries exist [??], we aim to produce maps where the boundary is free to move to reduce the distortion of the map.

[??] introduced the concept of scaffolding where the free space is triangulated for the purpose of morphing without self-intersection. In [?], the scaffold triangles are given a step function for their error: zero if not flipped, otherwise infinity. Hence, the bijective condition becomes local in that the shape can evolve until a scaffold triangle flips, in which case the free space is retriangulated and the optimization continues. The main limitation of this work is the lack of an evolving triangulation during the line search and the absence of a rotationally invariant metric for the scaffold triangles, which lead to very small steps and an inefficient optimization.

The Deformable Simplicial Complex (DSC) method [?] utilize a triangulation of both the free space and the interior of an object to track the interface between the two volumes. Similar to [?], the DSC retriangulates at degeneracies but also performs operations to improve the shape of the triangles. This method changes the triangulation of the interface that it tracks, which works well for simulation, but it is not allowed in many other applications such as UV mapping.

Air meshes [?] extends the technique of Zhang et al. [?] to add the concept of triangle flipping based on a quality measure during the optimization instead of simply retriangulating at the first sign of a degeneracy. However, this method does not maintain bijective maps as boundaries are allowed to inter-penetrate during optimization: the scaffold is only used to efficiently detect

problematic regions, and the local injectivity requirement is a soft constraint in the optimization. The problem tackled in this chapter is much harder, because we do not allow any overlap during any stage of the optimization to guarantee that the resulting maps will be bijective.

[?] take a different approach: instead of using a scaffold triangulation, the authors introduce a locally supported barrier function for the boundary to prevent intersection and explicitly limit the line search by computing the singularities of both the distortion energy and the boundary barrier function. Such an approach is inspired by traditional collision detection and response methods that are discussed below. Given a bijective starting point, this approach never leaves the space of bijective maps during optimization. Its main limitation is that it is computationally expensive, especially for large models. Our method is two order of magnitude faster (Figure 2.11).

Collision Detection and Response.

While not directly related to our approach, bijective maps inherently involve some form of collision detection and response to avoid overlaps. The field on collision detection is vast, and we refer the reader to a survey [?]. In terms of simulations, methods such as asynchronous contact mechanics [???] ensure the bijective property but are very expensive and designed to operate as part of a simulation. Differently, our approach is specialized for geometric optimization, where we are interested in a quasi-static solution (i.e. we do not want to explicitly simulate a dynamic system, but only find an equilibrium solution).

The work that is closer to ours in term of application (but very different in term of formulation) is [?], where collision detection and response is used to interactively deform shapes while avoiding self-intersections. Similarly to the previous methods, the explicit detection and iterative response is expensive when many collisions happen at the same time. Our work avoids these expensive computations, and can robustly handle hundreds of simultaneous collisions while still making large steps in the optimization.

Seam Creation.

In the context of parametrization, some approaches optimize the connectivity of the charts of the surface during parametrization to obtain a bijective map. [??] parameterize the surface and then split charts based on whether they intersect [?] or based on a level of distortion [?]. Sorkine et al. [?] employ a bottom-up approach and add triangles to a parametrization chart until bijectivity would be violated. The problem we are solving is more general (seams are only useful for texture mapping applications) and constrained (we preserve the prescribed seams). Our algorithm could be used by these algorithms to parametrize single charts, which could reduce the number of additional seams.

2.3 Method

Our method, Simplicial Complex Augmentation Framework (SCAF) utilizes a scaffold structure to robustly compute a bijective map between a pair of simplicial meshes with the same connectivity.

SCAF is specialized for the context of geometric optimization, i.e. we are interested in maps with low distortion and optionally satisfying a set of geometric constraints. We assume our maps are continuous and piecewise affine, i.e. the map deforms every simplex with an affine deformation. Thus, we can fully define the map using the image of its vertices.

Our algorithm uses a discrete, bijective identity map (encoded as a non-overlapping and non-flipping triangle/tetrahedral mesh) as initialization and then iteratively refines it, displacing the vertices while always ensuring that it remains bijective. Our method is composed of three stages: (1) augment the initial mesh with a scaffold, filling a bounding box around the initial map image; (2) optimize the extended mesh (scaffold included), reducing the geometric distortion of the map; (3) update the vertices and scaffold, enlarging the bounding box if necessary, to improve the quality of the triangulation. Steps (2) and (3) are iterated until the quality of the map is deemed sufficient.

2.3.1 General Formulation

Denote the input simplicial mesh by $\mathcal{M} = (\mathcal{V}, \mathcal{F})$ with a single, simple boundary representing a compact d-dimensional manifold embedded in the d-dimensional Euclidean space, where $\mathcal{V}$ is the set of n-vertices and $\mathcal{F}$ is the set of m-simplices. Our goal is to compute a continuous and piecewise affine mapping $\Phi : \mathcal{M} \to \mathbb{R}^d$ with $\Omega := \Phi(\mathcal{M}) = (\mathcal{V}', \mathcal{F})$ the resulting simplicial mesh with the same connectivity as $\mathcal{M}$.

We are interested in the bijective map that minimizes a given type of geometric energy:

$$\begin{aligned} \min_{\mathcal{V}'} \quad & E_{\mathcal{M}}(\Phi) \\ \text{s.t.} \quad & \Phi \text{ is bijective,} \end{aligned} \tag{2.1}$$

where $E_{\mathcal{M}}$ is a user-defined geometric energy.

Reduction to Local Orientation Preservation. A sufficient condition for the simplicial map $\Phi : M \to \Omega$ to be bijective is that the map preserves orientation and its restriction to the boundary $\Phi|_{\partial M} : \partial M \to \partial\Omega$ is bijective [**?**]. In this light, we are able to take advantage of the following simple construction (Figure 2.2): the algorithm extends the axis aligned bounding box of M to get a d-orthotope and fills the enclosed region with a scaffold simplicial complex to form a simplicial complex mesh D that includes and conforms to $M \subset D$. We can now define the continuous and piecewise affine map $\Psi : D \to D'$ with $\Phi = \Psi|_M$ where $\Psi|_{\partial D}$ is the identity map, and denote the scaffold region as $S = D \setminus M$. Now Φ is guaranteed to be bijective if Ψ preserves orientation. Using this observation, we can translate the bijectivity into a local, orientation-preserving requirement defined per simplex.

Variational Formulation Minimizing the distortion of Φ using the augmented map Ψ poses an interesting challenge: what is the desired shape of the scaffold S? Ideally we would like the simplices in the scaffold to maintain their orientation and not affect the optimization in any other way. Such a requirement is difficult to model directly, since it is a discontinuous condition that is not well-suited for the variational framework that we would like to use to minimize $E_{\mathcal{M}}$.

Figure 2.2: The initial mesh $\mathcal{M}$ (in green, left), is embedded in another mesh D (in gray, middle) that covers a box in the ambient space and contains the same triangles as $\mathcal{M}$. D might contain additional points (orange). We denote the triangles that are in D but not in $\mathcal{M}$ as the scaffold S. Our algorithm deforms D, inducing a corresponding deformation on $\mathcal{M}$ (right), while keeping the boundary (blue vertices) fixed and preventing changes in the triangle orientation.

We propose a regularized version of this condition modeled with an energy $E_{\mathcal{M}}(\Psi|_S)$ that still diverges when elements change orientation and that mildly penalizes any non-rigid distortion.

We choose a reweighted version of the symmetric Dirichlet energy $\mathcal{D}$ [?], measured w.r.t. the Jacobian of the map Ψ computed from the rest pose of S for each simplex f,

$$J_f := \nabla \Psi_f \tag{2.2}$$

where Ψ_f is the restriction of Ψ over the simplex f, which is an affine map. We divide the energy of each scaffold simplex f by its area A_f, and sum them up to obtain the final energy that favors an equal contribution regardless of the size of each scaffold simplex:

$$\begin{aligned} E_S(\Psi|_S) &= \sum_{f \in S} \frac{1}{A_f} \mathcal{D}(J_f) \\ &= \sum_{f \in S} (\|J_f\|_F^2 + \|J_f^{-1}\|_F^2 - 2d). \end{aligned} \tag{2.3}$$

The $-2d$ term ensures that the energy is 0 when $J_f = \mathbb{I}$. The map is then computed by summing the two terms:

$$\begin{aligned} \min_D \quad & E_{\mathcal{M}}(\Psi|_{\mathcal{M}}) + \lambda E_S(\Psi|_S) \\ \text{s.t.} \quad & \Psi|_{\partial D} \text{ is Identity} \\ & \Psi \text{ preserves orientation.} \end{aligned} \tag{2.4}$$

where $\lambda > 0$ is balancing the contribution of the two energies, decreasing as the optimization proceeds.

Iterative Regularization Solving this problem leads to a bijective and distortion minimizing map, but the regularizer will affect the stationary points of $E_{\mathcal{M}}$, which is problematic, especially for large deformations. To address this problem we iteratively minimize this energy, regenerating the scaffold at each iteration, and use the new scaffold as a rest pose for the regularization term $E_S(\Psi|_S)$. This iterative procedure has two positive effects: (1) it acts as a proximal regularization term without inhibiting movement since the rest pose is updated at each iteration; (2) the meshing quality of the scaffold is high, which avoids locking configurations.

Interpolation Coefficient. We experimentally observed that our algorithm is robust to different choices of λ, generating indistinguishable results in most cases. However, λ affects the convergence speed (Figure 2.3). We used $\lambda = \frac{1}{100}\frac{E_{\mathcal{M}}(\Psi|_{\mathcal{M}})}{|S|}$ for all our experiments, where $|S|$ is the number of scaffold simplices.

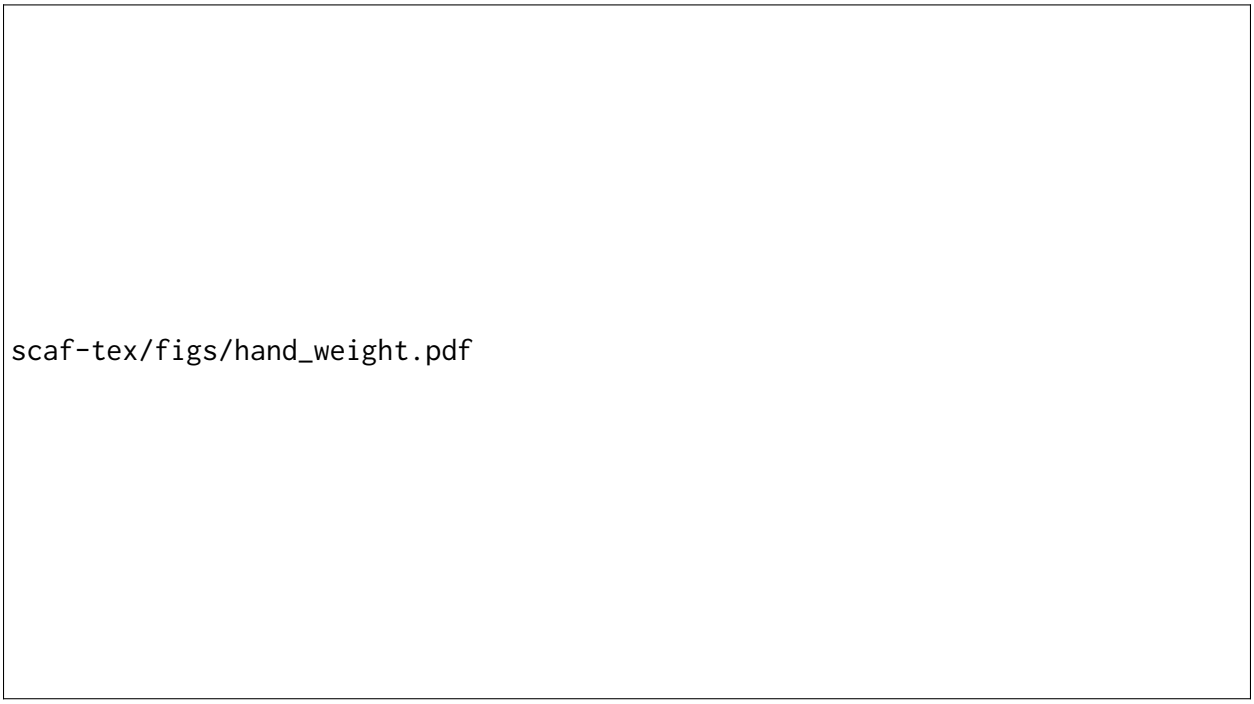


Figure 2.3: Different values of λ do not affect the result, but they change the number of iterations needed. From left to right: we used a large weight (100x ours), our weight, and a small weight (0.01x ours). The optimization took 9,7, and 8 iterations, respectively, to reach the same energy level.

Solver. Since our energy is rotational invariant, we can minimize the energy with the same quadratic proxy proposed in SLIM [**?**], enriching the approach with the equality constraint needed to fix the boundary ∂D. We also employ the orientation-preserving line search [**?**] (with exact predicates [**?**]) to ensure that no triangles can change orientation. Alternatively, other methods such as AQP [**?**] could be used to minimize this energy. Since our approach changes the mesh connectivity at every iteration, AQP loses much of its advantages as the approximate Hessian

must be recomputed at each iteration and not prefactored. Therefore, our approach is an ideal fit for [?], which takes large steps at every iteration without relying on a constant, prefactored matrix at each iteration. For practical applications, SLIM iterations are sufficient to minimize the energy to acceptable levels. For two stress tests (figures 2.4 and 2.11) we used the result of SLIM as a warm start for a Newton optimization (as suggested by [?]), which quickly converges to a numerical minimum.

2.3.2 Surface Parametrization

Our framework can be used for many applications, one of which is computing a bijective surface parametrization from a 3D surface into the UV plane. We follow [?] and assume that each 3D triangle f^{3D} is equipped with a rigid transformation R_f such that applying R_f to f^{3D} maps f^{3D} to the plane. Given this transformed triangle f, we can now measure the distortion of the map using the Jacobian of the affine transformation (a 2×2 matrix) from $R_f(f^{3D})$ to f, the location of the triangle in the parametrization.

Initialization Our method is initialized with Tutte's embedding algorithm [?]:

$$\Phi^0 : \mathcal{M}^{3D} \rightarrow \Omega^0,$$

where Ω^0 is a simplicial disk domain. Then we construct a larger rectangular domain $D^0 \supset \Omega^0$, where ∂D^0 is an axis-aligned rectangle, and use Triangle [?] to triangulate the region in between. We enforce a quality bound of 20° to obtain a graded mesh that is coarse on the boundary and conforming the boundary of Ω^0. This grading implicitly produces an approximate inverse distance weighting of the scaffold space with respect to the error function, which enables a larger deformation per iteration. Then we define $\Psi : D^0 \rightarrow D \subset \mathbb{R}^2$ and restrict $\Psi|_{\partial D^0}$ to be the identity.

Mesh Improvement At the end of each iteration, we improve the quality of the scaffold. The reason for maintaining a good mesh quality is two-fold. First, as observed in [??], fixing the scaffold will potentially prevent movement. Secondly, the quality of the scaffold affects the condition of the linear system in SLIM [?]: a higher quality leads to larger and more efficient iterations.

We resort to Triangle [?] to create the initial scaffold and to regenerate the scaffold mesh in the improvement step. Our experiments show that, in 2D, it is faster to generate the mesh from scratch at every iteration instead of trying to optimize the scaffold using local operations as suggested in [?]. Since our solver makes large steps in each iteration, the scaffold requires significant connectivity changes each iteration, which explains why regenerating the triangulation is faster than local operations. This is in stark contrast with physical simulation scenarios where each iteration represents a small time step and, thus, a minor change in the vertex positions.

We demonstrate the effectiveness of the remeshing strategy in Figure 2.4 where our method recovers from a large rotation — note that the scaffold is updated during the iterations and always leaves space for the map to move freely.

Figure 2.4: A bijective map from a circle (left) to a spiral (right) is computed without (top) and with (bottom) the iterative remeshing step. The slivers in the triangulation locks the optimization (top), preventing it from reaching the target shape.

Sliding & Degeneracy Prevention As the optimization proceeds and some of the boundary elements get closer, some of the scaffold triangles might (and often will) get smaller and smaller, restricting the amount of sliding that is allowed in one iteration as well as introducing numerical difficulties in computing the corresponding Jacobian J_f whose singular values will approach infinity.

To avoid this issue, we replace the degenerating target when computing its Jacobian. For the triangles with an area smaller than ϵ, we use an equilateral triangle with area ϵ to compute the local Jacobian. In our experiment, we traverse through the boundary of the interior of the uv domain at the current iteration to find the minimum edge length l and set $\epsilon = \frac{l^2}{4}$.

A theoretical downside of this modification is that it affects the distortion energy. We experimentally observed that the changes are negligible, and we thus used it for all our experiments. However, on the practical side, it discourages fully degenerate elements — this change, coupled with the orientation preserving line search [**?**], makes our algorithm robust enough for the challenging stress tests shown in Figure 2.5.

2.3.3 Extension to 3D

Our formulation naturally unifies bijective geometric optimization problems in 2D and 3D, so the algorithm readily extends to the 3D case with only one major difference: the scaffold becomes a tetrahedral mesh, which is computationally more challenging to create and update.

Figure 2.5: Two models are cut using [?] and bijectively parametrized using our algorithm. See the additional material for more examples.

Mesh Improvement We use TetGen [?] to generate the initial scaffold and the local operations proposed in [?] to optimize the scaffold's quality in the subsequent iterations. It is unfortunately not possible to directly use TetGen at every iteration as we did with Triangle in the 2D case, since TetGen fails when boundaries get too close, which is common in our experiments.

Guarantee Similarly to the 2D case, we are guaranteed that no flipping or self-intersecting tetrahedra will occur since we are following an interior point strategy. Notice the contrast with Air Mesh [?] where only if penetration happens can the constraints be of effect. However, as pointed out in [Dougherty et al. 2004], the local operations we are performing may not be sufficient to explore the entire space of possible tetrahedralizations. Therefore, we cannot guarantee that the algorithm achieves the globally optimal solution.

2.4 Results

We implemented our algorithm in C++ using Eigen for linear algebra routines. We ran our experiments on a desktop with a 4-core Intel i7 processor clocked at 4 GHz and 32 GB of memory but using only one thread on a single core. For all experiments, the scaffold bounding box is computed by uniformly scaling by three times the bounding box of the image of the current map.

Robustness. To demonstrate the robustness of our algorithm, we computed bijective maps for all the 102 meshes parametrized by the MIQ algorithm [**?**] and for the 17 meshes parametrized by [**?**] in the dataset proposed by [**?**]. The cuts in these meshes have been designed for locally injective parametrization that usually have major self-overlap. We use them as a stress test for the effectiveness and robustness of our method: the cuts introduce a massive distortion in the Tutte initialization and lead to boundaries that are prone to overlap in hundreds of locations. Our method successfully creates bijective parametrizations for all these models with default parameters. We attach all the parametrized models in the additional material and show two examples in Figure 2.5.

Scalability. Our methods scales gracefully to large datasets, similarly to [**?**]. We repeat their scalability experiment, but producing bijective maps instead of just locally injective maps (Figure 2.6). The behaviour is remarkably similar — the density of the model (and consequently of the scaffold) does not affect the number of required iterations.

Texture Atlas Generation UV mapping is a time consuming procedure required in most geometric modeling pipelines. Existing commercial tools provide the ability to flatten single patches and arrange them in UV layouts where multiple patches are tightly packed inside a rectangular domain, which is then loaded in the texture memory of a GPU.

Our algorithm can bijectively parameterize a single patch (Figure 2.7), avoiding the typical manual UV postprocessing required with traditional tools. Our algorithm can also be used to create automatic UV charts of models with multiple connected components (or predefined cut edges). We show an example in Figure 2.8 where we detected the connected components, bijectively map the patches into a set of circles (using a grid layout), and reduce their distortion using our algorithm. The result is a tight and automatic packing without resorting to any user-interaction. Additional interactive tools can further improve the atlas by dragging&dropping regions or translating islands while ensuring that no overlaps are introduced (Figure 2.1). We show interactive sessions using our packing tool in the additional material.

Preventing Self-Intersections Our algorithm can be generalized to handle mixed dimension problems, such as the deformation of 2D surface in 3D space, while preventing self-intersections. In Figure 2.9, we demonstrate the use of our method to resolve self-intersections of surfaces. First we perform a conformalized flow [**?**] using the algorithm proposed in [**?**] to resolve any self-intersections. While [**?**] will resolve the intersections, the resulting surface may be geometrically far from the initial shape (see Figure 2.9). Next we tetrahedralize the ambient space while conforming to the deformed surface mesh and minimize Equation 2.4 with an additional energy term that strives to restore the rest pose geometry of the surface, using the surface ARAP energy proposed in [**?**]. The result is a surface similar to the original mesh, but without self-intersections. In this example, it is possible to observe that even dramatic changes of scale (on the foot) can be robustly handled by our parametrization algorithm.

A more challenging stress test is shown in Figure 2.10, where the bunny model is scaled up inside a box, to 30 times its original size. No self-intersections are introduced, despite the extreme,

Figure 2.6: We compare the distortion energy with respect to the number of iterations on a set of Lucy's meshes with different resolutions (from 1 to 12 million faces). In the center of the plot, we show the 1M Lucy model parametrized by our algorithm.

constrained deformation.

Comparison with [?] The algorithm closest to ours is [?], which tackles a similar problem (restricted to the 2D case). We replicated the space filling curve experiment and obtained remarkably similar results, where our running time is 96s, compared with 8,472s for [?] (88 times faster). We show in Figure 2.11 a more challenging experiment with a subdivided version of the space filling curve to emphasize the performance difference: our algorithm converges in 39 minutes, while [?] did not converge after 5 days and 21 hours. For this example, we used the procedure suggested in [?]: we performed a few iterations minimizing the quadratic proxy and then switch to a traditional newton method until numerical convergence. A video of the optimization is provided in the additional material.

Our method produces results that are visually identical to [?]. In Figure 2.12 we repeat the

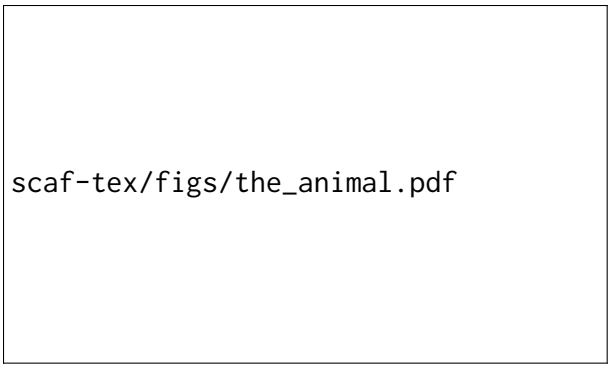


Figure 2.7: A mesh is cut by an artist into a single chart and parametrized using SLIM [?] (left) and with our algorithm (right). Note that local-injectivity is not sufficient for this model, since the global overlaps in the highlighted region prevent this parametrization from being a UV texture map. Our result (right) is guaranteed to be bijective.

experiments shown in [?], stopping our optimization at the same energy value.

Local vs Global Optimization Both [?] and [?] use a construction similar to ours to generate bijective maps (Section 5.2). Both methods explicitly prevent changes of orientation using a local approach: they optimize the map using coordinate descent iterations [?] allowing only one vertex at a time to move in its 1-ring and thus ensuring that no triangle flip. This strategy severely limits the maximal displacement per iteration and restricts the step to the size of the 1-rings. Such a restriction makes these methods impractical for parametrization applications since the difference in scale between the Tutte's embedding and the final result is extreme (the ratio of min and max triangle area is 10^{-6} in Figure 2.13). We show an example of one of our iterations in Figure 2.13, where the highlighted vertex traversed a distance of 150 times the size of the average edge length of its 1-ring in one single step. Using coordinate descent would have required hundreds of iterations to achieve the same effect.

Despite the orientation-dependent box used as scaffold boundary, our optimization produces results that are, in practice, independent of orientation. We show this effect in Figure 2.14 where we initialize the optimization with 1000 randomly rotated Tutte's mappings of the same camel model and run our optimization. The isometric distortion of the model after 50 iterations is quite similar in all trials (the minimum, maximum, average, standard deviation of distortion errors in all 1000 runs are 0.1086, 0.1107, 0.1095, 3.2698e-4 resp.) indicating very little change based on the initial orientation.

Timings The timings for all the results in the paper are reported in Table 2.1.

2.5 Limitations and Concluding Remarks

We proposed a simple and robust algorithm to generate bijective maps, both in 2D and 3D. We demonstrated the practical value of the algorithm in UV mapping and deformation applications,

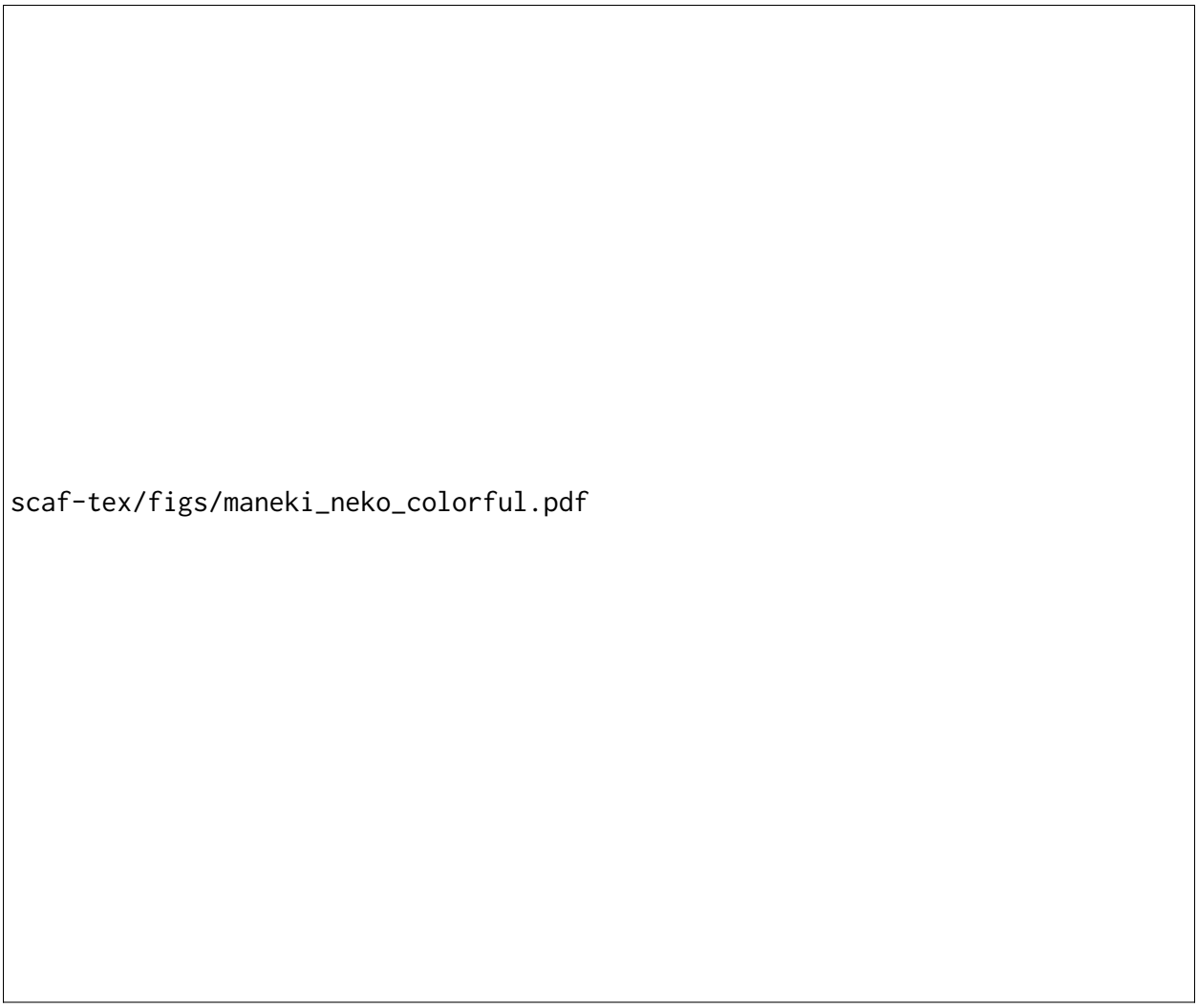


Figure 2.8: A model with multiple chart (left) is automatically parametrized in a texture atlas (bottom-right) by first mapping each component to a circle (top-right) and then minimizing the distortion.

and its robustness with extensive stress tests.

One major venue for future work is the support of hard positional constraints, which are favored over soft constraints in many practical applications. Our current algorithm only supports soft constraints as geometric energy [**?**]. To support hard constraints we would need to generate a bijective starting point that guarantees those constraints, and then preserve them in our optimization. While bijecive maps with hard constraints can be constructed for a 2D patch homeomorphic to a disk [**?**] and for a 3D volume homeomorphic to a ball [**?**], the generic solution is still elusive.

In 3D cases, the generation of the initial scaffold is not as robust as in 2D, since TetGen fails for geometries with self-intersections and other imperfections. Our algorithm is also slower in 3D due to larger and denser linear systems, as well as the need for local mesh refinement operations instead of regenerating the entire tetrahedralization. We believe a more optimized and parallel

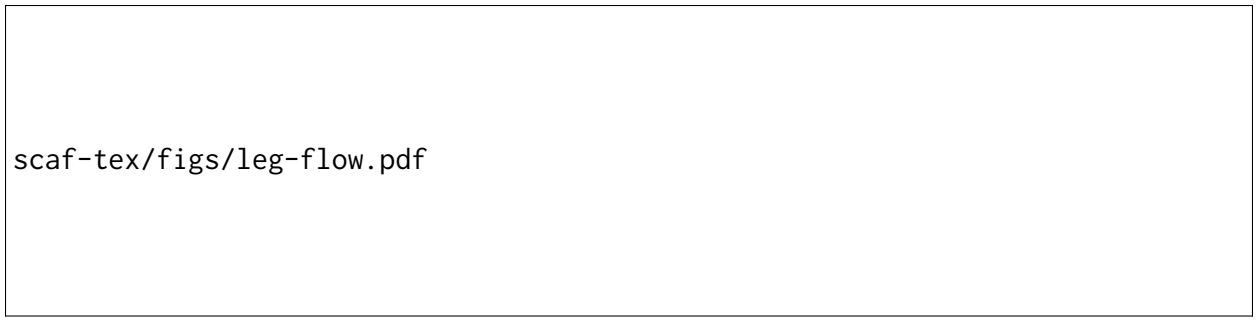


Figure 2.9: We remove the self-intersections from a genus 0 model using the conformalized flow [**??**]. The flow is inverted, while using our algorithm to compute a bijective volumetric map, to recover a self-intersection free version of the original surface. The final model can now be meshed using TetGen, since it is free from self-intersections.

Figure 2.10: We grow a bunny inside a box, while preventing self-intersections. We show the result after 0,10,20,30,40, and 50 iterations.

implementation could reduce this overhead, and plan to explore this in the future.

Figure 2.11: We repeat the challenging test in [?] with a subdivided version of their Hilbert curve to increase the triangle count. Our method starts from a disc (upper left), gracefully extends (upper right), and reaches the same minimum (lower left) in 39 minutes whereas [?] didn't terminate more than 5 days (lower right), highlighting our performance boost of over 200 times.

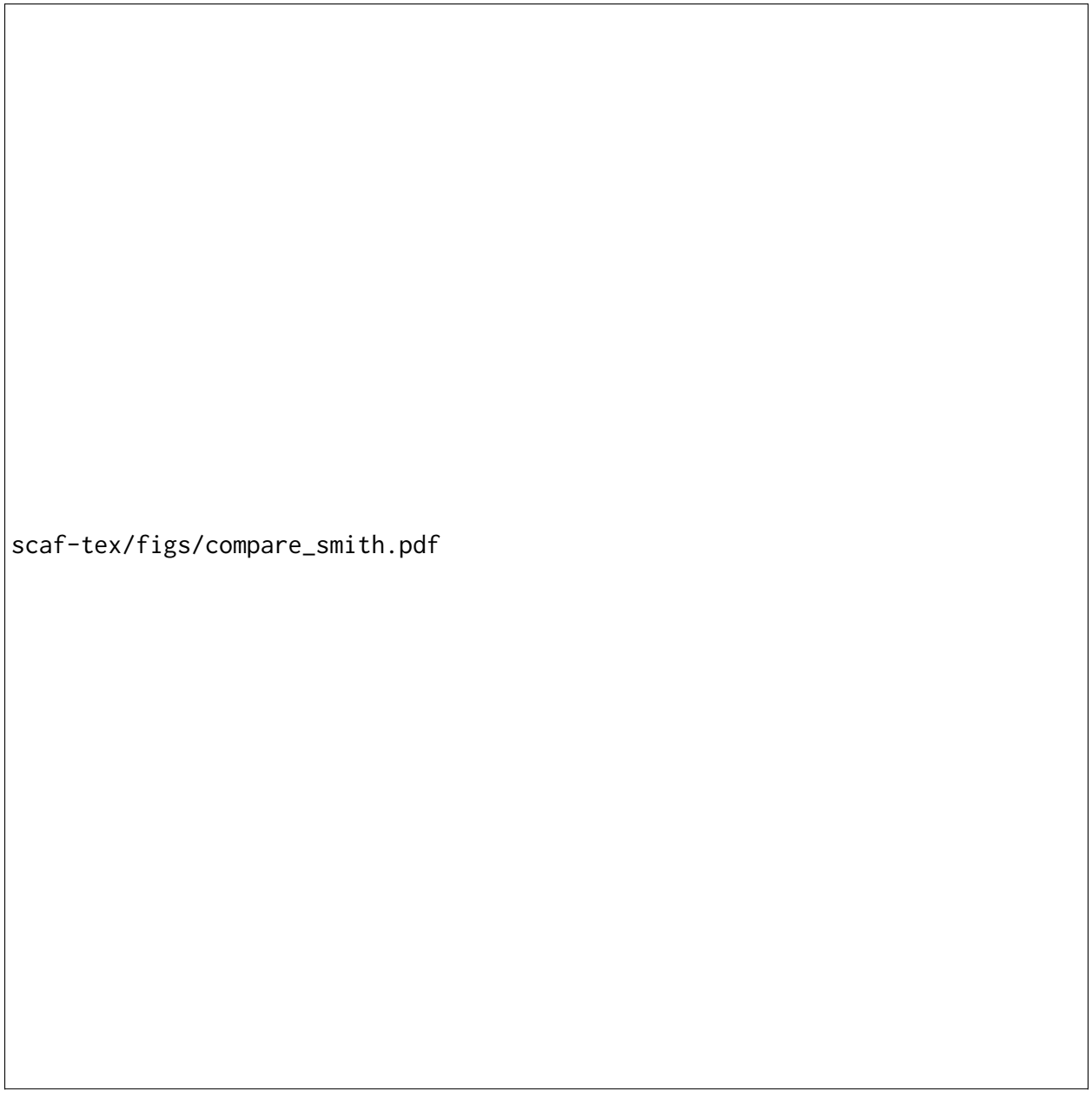


Figure 2.12: We apply our algorithm on 4 models used in [?] (using the same stopping criteria) obtaining visually identical results. Distortion errors produced by our algorithm (outer) and theirs (inner) are shown in black.

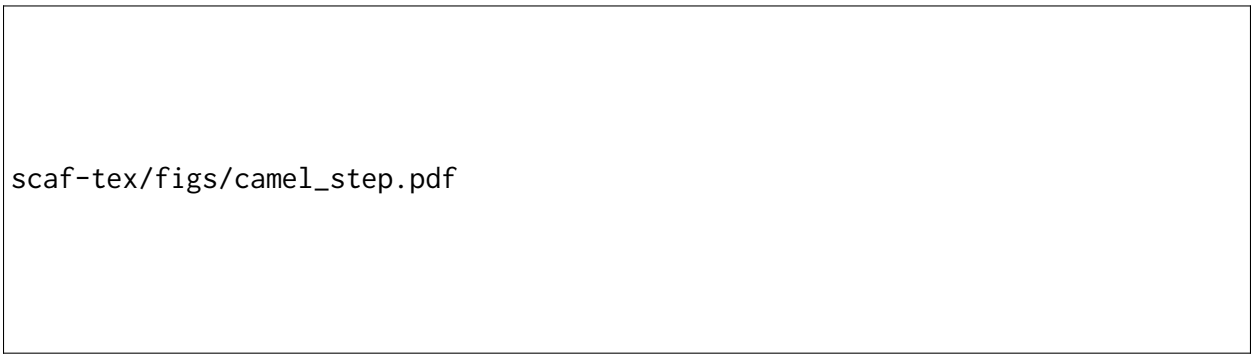


Figure 2.13: A single iteration of our algorithm (from left to right) drastically reduces the distortion. The black vector in the center is 150 times longer than the average edge length of its 1-ring. Iterative methods would need thousands of iterations to achieve a similar progress.

Type	Model	#V	#F	$\#V_S$	$\#F_S$	It.	Total Time (s)	It. Time (s)
Atlas	Nefertiti (Fig. 2.1)	1697	2823	983/ 247	1945 /728	50	0.71	0.01
	Maneki-Neko (Fig. 2.8)	23025	43648	2427 /725	7174/ 3770	50	16.81	0.34
2D	Hand (Fig. 2.3)	2239	4046	347/280	1104/970	7	0.14	0.02
	Spiral (Fig. 2.4)	54	52	78/36	190/106	50(50)	0.04(0.21)	0.01
	Thai Statue (Fig. 2.5, left)	42405	79970	3665/1593	12148/8004	50	28.28	0.56
	Filigree (Fig. 2.5, right)	56062	100000	9160/2627	30422/17356	100	75.99	0.76
	Lucy (Fig. 2.6)	501105	1000000	1856/ 3470	5900/ 5674	100	2524.22	25.24
	Lucy (Fig. 2.6)	1001375	1999999	2284/ 4400	7297/ 7133	100	7251.00	72.51
	Lucy (Fig. 2.6)	2002031	3999999	3587/ 6930	11215/ 10985	100	22500.07	225.00
	Lucy (Fig. 2.6)	3002899	5999999	5135/ 9859	16047/ 15601	100	52235.31	522.35
	Lucy (Fig. 2.6)	4002816	8000000	5140/ 10288	15890/ 15918	100	59413.14	594.13
	Lucy (Fig. 2.6)	5003408	10000000	6194/ 12231	19182/ 19040	100	95247.59	952.47
	Lucy (Fig. 2.6)	6004111	12000000	7357/6418	2291/21036	50	78726.05	1574.52
	Animal (Fig. 2.7)	19937	39040	747/593	2306/1998	50	15.36	0.31
	Space Filling (Fig. 2.11)	79545	146832	90815/88237	181608/176452	200(250)	547.13(1836.58)	5.30
	Horse (Fig. 2.12)	20636	39698	1343/984	4238/3520	30(10)	8.26(12.03)	0.28(1.20)
	Camel (Fig. 2.12)	2032	3576	384/272	1234/1010	30(10)	0.52(1.13)	0.02(0.11)
	Cow (Fig. 2.12)	3195	5804	491/277	1546/1118	30(10)	0.81(1.74)	0.03(0.17)
	Tricera (Fig. 2.12)	3163	5660	544/329	1732/1302	30(10)	0.83(1.77)	0.03(0.18)
3D	Leg (Fig. 2.9)	6617	13230	5016/5021	68521/68544	500	3251.17	6.50
	Bunny (Fig. 2.10)	568	1132	683/706	6209/6289	50	7.16	0.14

Table 2.1: Timings and statistics for the models shown in the paper. From left to right: number of input vertices and simplices, number of initial/final scaffold vertices and simplices, number of iterations, running time in seconds. The numbers in parenthesis refer to the Newton optimization. Note that our timings are considerably higher than those reported in the SLIM paper for the Lucy model since we used the reference implementation in [**?**], which does not use a multi-threaded solver.

Figure 2.14: Our algorithm is independent to the initial orientation. We rotate the initializing Tutte's mapping of the camel model and obtain results with similar isometric distortion.

3 | Bijective Projection in the Shell

3.1 Introduction

Triangular meshes are the most popular representation for discrete surfaces, due to their flexibility, efficiency, and direct support in rasterization hardware. Different applications demand different meshes, ranging from extremely coarse for collision proxies, to high-resolution and high-quality for accurate physical simulation. For this reason, the adaptation of a triangle mesh to a specific set of criteria (surface remeshing) is a core building block in geometry processing, graphics, physical simulation, and scientific computing.

In most applications, the triangular mesh is equipped with attributes, such as textures, displacements, physical properties, and boundary conditions (Figure 3.1). Whenever remeshing is needed, these properties must be transferred on the new mesh, a task which has been extensively studied in the literature and for which robust and generic solutions are still lacking (Section 5.2). Defining a continuous bijective map, more precisely, a homeomorphism where the inverse is also continuous, between two geometrically close piecewise-linear meshes of the same topology is a difficult problem, even in its basic form, when one of these meshes is obtained by adapting the other in some way (e.g., coarsening, refining, or improving triangle shape).a Common approaches to this problem are Euclidean projection [**?**], parametrization on a common domain [**???**], functional maps [**?**], and generalized barycentric coordinates [**?**]. However, the problem is not fully solved, as all existing methods, as we discuss in greater detail in Section 5.2, often fail to achieve bijectivity and/or sufficient quality of the resulting maps when applied to complex geometries. Our focus is on correspondences between meshes obtained during a remeshing procedure, instead of solving the more general problem of processing arbitrary mesh pairs.

In this work, we propose a general construction designed to enable attribute mapping between geometrically close (in a well-defined sense) meshes by jointly constructing: (1) a shell $\mathcal{S}$ around triangle mesh $\mathcal{T}$ spanned by a set of prisms, inducing a volumetric vector field $\mathcal{V}$ in its interior and (2) a projection operator $\mathcal{P}$ that bijectively maps surfaces inside the shell to $\mathcal{T}$, as long as the dot product of the surface face normals and $\mathcal{V}$ is positive (we call such a surface a *section* of $\mathcal{S}$). Given a surface mesh $\mathcal{T}$ and its shell $\mathcal{S}$, it is now possible to exploit the bijection induced by $\mathcal{P}$ in many existing remeshing algorithms by adding to them an additional constraint ensuring that the generated surface is a section of a given shell.

As long as the generated mesh is a section, the projection operator $\mathcal{P}$ can be used to transfer application-specific attributes. At a higher level, the middle surface of our shell can be seen as a common parametrization domain shared by sections within the shell: differently from other

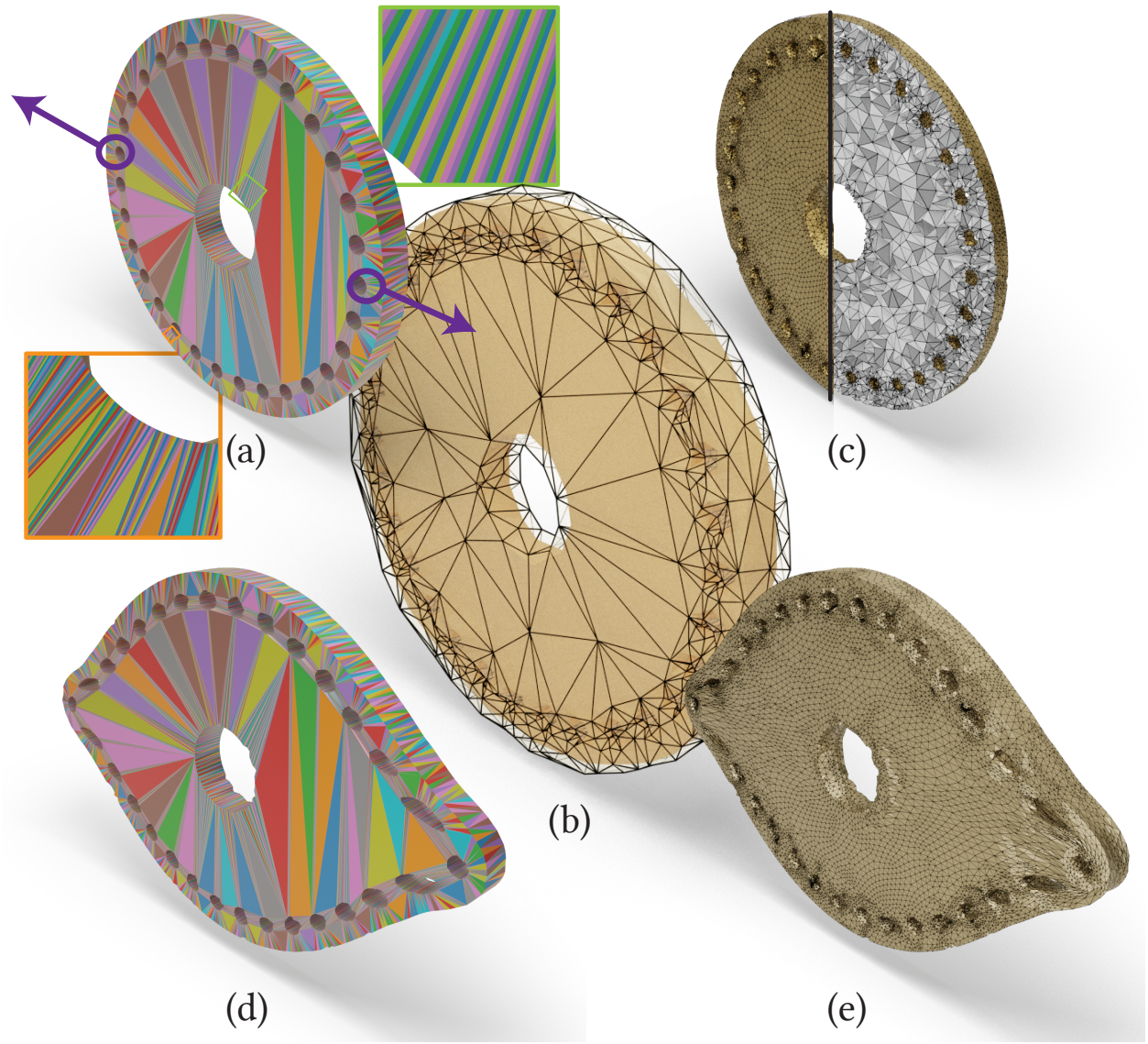


Figure 3.1: A low-quality mesh with boundary conditions (a) is remeshed using our shell (b) to maintain a bijection between the input and the remeshed output. The boundary conditions (arrows in (a)) are then transferred to the high-quality surface (c), and a non-linear elastic deformation is computed on a volumetric mesh created with TetGen (e). The solution is finally transferred back to the original geometry (d). Note that in this application setting both surface and volumetric meshing can be hidden from the user, who directly specifies boundary conditions and analyses the result on the input geometry.

methods that map the triangle meshes to a disk, region of a plane, a canonical polyhedron, or orbifolds, our construction uses an explicit triangle mesh embedded in ambient space as the common parametrization domain. This provides additional flexibility since it is adaptive to the density of the mesh and naturally handles models with high genus, while being numerically stable under floating-point representation (and exact if evaluated with rational arithmetic). The downside is that it is defined only for sections contained within the shell. The construction and optimization of our shell, and corresponding bijective mapping, is computationally more expensive than remeshing-only methods: our algorithms takes seconds to minutes on small and medium sized models, and might take hours on the large models in our tests.

We evaluate the robustness of the proposedapproach by constructing shells for a subset of the models in Thingi10k [**?**] and in ABC [**?**] (Section 3.4). We also integrate it in six common geometry processing algorithms to demonstrate its practical applicability (Section 5.4):

1. *Proxy.* The creation of a proxy, high-quality remeshed surface to solve PDEs (e.g., to compute geodesic distances or deformations), avoiding the numerical problems caused by a low-quality input in commonly used codes. Bijective projection operators associated with a shell enable us to transfer boundary conditions to the proxy mesh, compute the solution on the proxy, and then transfer the solution back to the original geometry.

2. *Boolean operations.* The remeshing of intermediate results of Boolean operations, to ensure high-quality intermediate meshes while preserving a bijection to transfer properties between them.

3. *Displacement Mapping.* The automatic conversion of a dense mesh into a coarse approximation and a regularly sampled displacement height map. Our method generates a bijection that allows us to bake the geometric details in a displacement map.

4. *Tetrahedral Meshing.* The conversion of a surface mesh of low quality into a high-quality tetrahedral mesh, with bijective correspondence.

5. *Geometric Textures.* Generation of complex topological structures using volumetric textures mapped to the volumetric parametrization of a simplified shell defined by $\mathcal{P}$. Our analysis on the initial shell also complements the literature on shell maps.

6. *Nested Cages.* A robust approach to generate a coarse approximation of a surface for collision checking, cage-based deformation, or multigrid approaches.

Our contributions are:

1. An algorithm to build a prismatic shell and the corresponding projection operator around an orientable, manifold, self-intersection free triangle mesh with arbitrary quality;

2. A new definition of bijective maps between *close-by* discrete surfaces;

3. A reusable, reference implementation provided at `https://github.com/jiangzhongshi/bijective-projection-shell`.

3.2 Related Works

We review works in computer graphics spanning both the realization of maps for attribute transfer (Section 3.2.1), and the explicit generation of boundary cages (Section 3.2.2), which are closest to our work.

3.2.1 Attribute Transfer

Transferring attributes is a common task in computer graphics to map colors, normals, or displacements on discrete geometries. The problem is deeply connected with the generation of UV maps, which are piecewise maps that allow to transfer attributes from the planes to surfaces (and composition of a UV map with an inverse may allow transfer between surfaces). We refer to [???] for a complete overview, and we review here only the most relevant works.

Projection Modifying the normal field of a surface has roots in computer graphics for Phong illumination [?], and tessellation [?]. Orthographic, spherical, and cage based projections are commonly used to transfer attributes, even if they often leads to artifacts, due to their simplicity [??]. Projections along a continuously-varying normal field has been used to define correspondences between neighbouring surfaces [????], but it is often discontinuous and non-bijective. While the discontinuities are tolerable for certain graphics applications (and they can be reduced by manually editing the cage), these approaches are not usable in cases where the procedure needs to be automated (batch processing of datasets) or when bijectivity is required (e.g., transfer of boundary conditions for finite element simulation). These types of projection may be useful for some remeshing applications to eliminate surface details [?], but it makes these approaches not practical for reliably transferring attributes. Our shell construction, algorithms, and associated projection operator, can be viewed as guaranteed continuous bijective projection along a field.

Common Domains A different approach to transfer attributes is to map both the source and the target to a common parametrization domain, and to compose the parametrization of the source domain with the inverse parametrization of the target domain to define a map from source to target. In the literature, there are methods that map triangular meshes to disks [??], region of a plane [??????????????], a canonical coarse polyhedra [??], orbifolds [???], Poincare disk [????], spectral basis [???], and abstract domains [???]. While these approaches allow mappings between completely different surfaces, this is a hard problem to tackle in full generality fully automatically, with guarantees on the output (even some instances of the problem of global parametrization, i.e. maps from a specific type of almost everywhere flat domains to surfaces, lack a fully robust automatic solution).

Our approach uses a coarse triangular domain embedded in ambient space as the parametrization domain, and uses a vector field-aligned projection within an envelope to parametrize close-by surfaces bijectively to the coarse triangular domain. Compared to the methods listed above, our approach has both pros and cons. Its limitation is that it can only bijectively map surfaces that are similar to the domain, but on the positive side, it: (1) is efficient to evaluate, (2) guarantees

an exact bijection (it is closed under rational computation), (3) works on complex, high-genus models, even with low-quality triangulations, (4) less likely to suffer from high distortion (and the related numerical problems associated with it), often introduced by the above methods. We see our method not as a replacement for the fully general surface-to-surface maps (since it cannot map surfaces with large geometric differences), but as a complement designed to work robustly and automatically for the specific case of close surfaces, which is common in many geometry processing algorithms, as well as serve as a foundation for generating such close surfaces (e.g., surface simplification and improvement, see Section 5.4)

Attribute Tracking In the specific context of remeshing or mesh optimization, algorithms have been proposed to explicitly track properties defined on the surface [???] after every local operation. By following the operations in reverse order, it is possible to resample the attributes defined on the input surface. These methods are algorithm specific, and provide limited control over the distortion introduced in the mapping. Our algorithm provides a generic tool that enables any remeshing technique to obtain such a map with minimal modifications.

3.2.2 Shell Generation

The generation of shells (boundary layer meshes) around triangle meshes has been studied in graphics and scientific computing.

Envelopes Explicit [??] or implicit [?] envelopes have been used to control geometric error in planar [?], surface [??Cheng et al. 2019], and volumetric [??] remeshing algorithms. Our shells can be similarly used to control the geometric error introduced during remeshing, but they offer the advantage of providing a bijection between the two surfaces, enabling to transfer attributes between them without explicit tracking [?]. We show examples of both surface and volumetric remeshing in Section 5.4. Also, [??] utilize envelopes for function interpolation and reconstruction, where our optimized shells can be used for similar purposes.

Shell Maps 2.5D geometric textures, defined around a surface, are commonly used in rendering applications [????????]. The requirement is to have a thin shell around the surface that can be used to map an explicit mesh copied from a texture, or a volumetric density field used for ray marching. Our shells are naturally equipped with a 2.5D parametrization that can be used for these purposes, and have the advantage of allowing users to generate coarse shells which are efficient to evaluate in real-time. The bijectivity of our map ensures that the volumetric texture is mapped in the shell without discontinuities. We show one example in Section 5.4.

Boundary Layer Boundary layers are commonly used in computational fluid dynamics simulations requiring highly anisotropic meshing close to the boundary of objects. Their generation is considered challenging [???]. These methods generate shells around a given surface, but do not provide a bijective map suitable for attribute transfer.

Collision and Animation Converting triangle meshes into coarse cages is useful for many applications in graphics [?], including proxies for collision detection [?] and animation cages [?]. While not designed for this application, our shells can be computed recursively to create increasingly coarse nested cages. We hypothesize that a bijective map defined between all surfaces of the nested cages could be used to transfer forces from the cages to the object (for collision proxies), or to transfer handle selections (for animation cages). [??] uses a prismatic layer to define volumetric deformation energy, however their prisms are disconnected and only used to measure distortion. Our prisms could be used for a similar purpose since they explicitly tesselate a shell around the input surface.

3.2.3 Robust Geometry Processing

The closest works, in terms of applications, to our contribution are the recent algorithms enabling black-box geometry processing pipelines to solve PDEs on meshes *in the wild*.

[??] refines arbitrary triangle meshes to satisfy the Delaunay mesh condition, benefiting the numerical stability of some surface based geometry processing algorithms. These algorithms are orders of magnitude faster than our pipeline, but, since they are refinement methods, cannot coarsen dense input models. While targeting a different application, [?] offers an alternative solution, which is more efficient than the extrinsic techniques [?] since it avoids the realization of the extrinsic mesh (thus naturally maintaining the correspondence to the input, but limiting its applicability to non-volumetric problems) and it alleviates the introduction of additional degrees of freedom. [?] further generalizes [?] to handle non-manifold and non-orientable inputs, which our approach currently does not support.

TetWild [??] can robustly convert triangle soups into high-quality tetrahedral meshes, suitable for FEM analysis. Their approach does not provide a way to transfer boundary conditions from the input surface to the boundary of the tetrahedral mesh. Our approach, when combined with a tetrahedral mesher that does not modify the boundary, enables to remesh low-quality surface, create a tetrahedral mesh, solve a PDE, and transfer back the solution (Figure 3.1). However, our method does not support triangle soups, and it is limited to manifold and orientable surfaces.

3.2.4 Isotopy between surfaces

[??] presents conditions for two sufficiently smooth surfaces to be isotopic. Specifically, the projection operator is a homeomorphism. [?] extends this idea to make an approximation mesh that is isotopic to a region. However, they did not realize a map suitable for transferring attributes.

3.3 Method

Our algorithm (Figure 3.2) converts a self-intersection free, orientable, manifold triangle mesh $\mathcal{T} = \{V_\mathcal{T}, F_\mathcal{T}\}$, where $V_\mathcal{T}$ are the vertex coordinates and $F_\mathcal{T}$ the connectivity of the mesh, into a shell composed of generalized prisms $\mathcal{S} = \{(B_\mathcal{S}, M_\mathcal{S}, T_\mathcal{S}), F_\mathcal{S}\}$, where $B_\mathcal{S}$, $M_\mathcal{S}$, $T_\mathcal{S}$ are bottom,

Figure 3.2: Overview of our algorithm. We start from a triangle mesh, find directions of extrusion, build the shell, and optimize to simplify it.

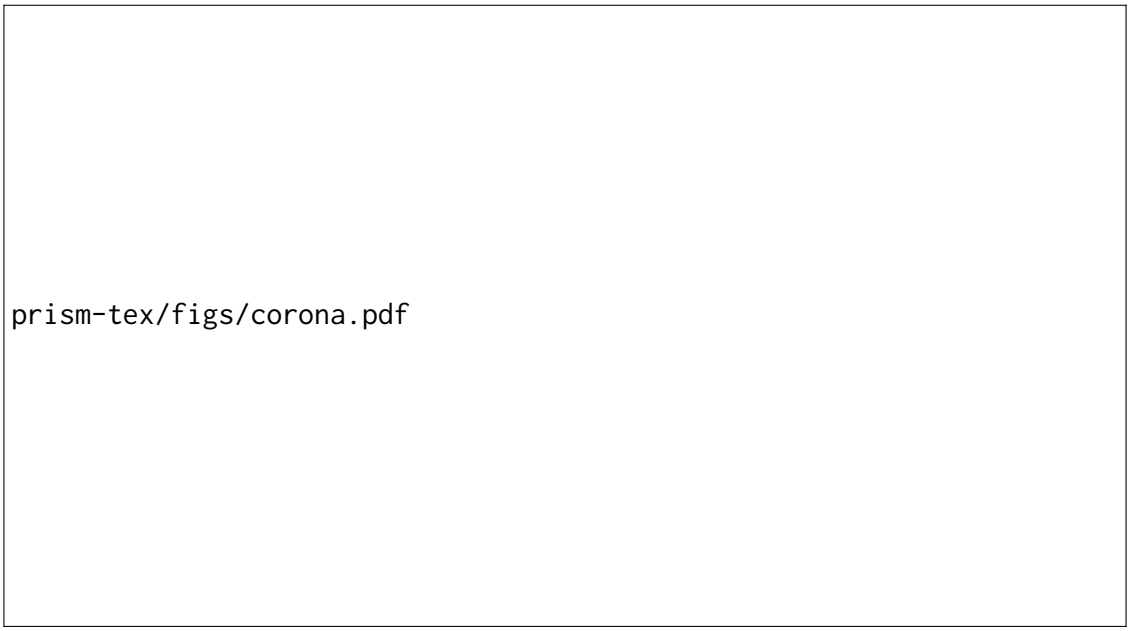


Figure 3.3: Example of the top (left, outer) and bottom (right, inner) surface of the prismatic shell.

middle, and top surfaces of the shell, consisting of bottom, middle, and top triangles of the prisms, and $F_{\mathcal{S}}$ is the connectivity of the prisms (Figure 3.3). The algorithm initially generates a shell $\mathcal{S}$ whose middle surface $M_{\mathcal{S}}$ has the same geometry as the input surface $\mathcal{T}$ (possibly with refined connectivity), and then optimizes it while ensuring that $\mathcal{T}$ is contained inside and projects bijectively to $M_{\mathcal{S}}$. The shell induces a volumetric vector field $\mathcal{V}$ and a projection operator $\mathcal{P}$ in the interior of each of its prisms (Section 3.3.1). This output can be used directly in many geometry processing tasks, as we discuss in detail in Section 5.4.

We first introduce the definition of our projection operator $\mathcal{P}$ and the conditions required for

Figure 3.4: A prism Δ (left) is decomposed into 6 tetrahedra (middle, for clarity, we only draw the 3 tetrahedra of the top slab). Each tetrahedron has a constant vector field in its interior (pointing toward the top surface), which is parallel to the only pillar of the prism that contains the point.

bijectivity of its restrictions to sections of the shell (Section 3.3.1). We then define shell validity (Section 3.3.2), present our algorithm for creating an initial shell (Section 3.3.3) and optimizing it to decrease the number of prisms (Section 3.3.4). To simplify the exposition, we initially assume that our input triangle mesh does not contain *singular points* (defined in Section 3.3.3) and boundary vertices, and we explain how to modify the algorithm to account for these cases in sections 3.3.5 and 3.3.6.

3.3.1 Shell and Projection

Let us consider a single generalized prism Δ in a prismatic layer $\mathcal{S}$ (Figure 3.4 left). The generalized prism Δ is defined by the position of the vertices of three triangles, one at the top, with coordinates t_1, t_2, t_3, one at the bottom, with coordinates b_1, b_2, b_3, and one in the middle, implicitly defined by a per-vertex parameter $\alpha_i \in [0, 1]$, with coordinates $m_i = \alpha_i t_i + (1 - \alpha_i) b_i, i = 1, 2, 3$. We will call the top (bottom) "half" of the prism *top (bottom) slab* (we refer to Appendix A.5 for an explanation on why we need two slabs). For brevity, we will refer to a generalized prism as a prism.

Decomposition in Tetrahedra We decompose each prism Δ into 6 tetrahedra (3 in the top slab and 3 in the bottom one, Figure 3.4 middle), using one of the patterns in [**?**, Figure 4]. The patterns are identified by the orientation (rising/falling) of the two edges cutting the side faces of the prism. While, for a single prism, any decomposition would be sufficient for our purposes, we need a consistent tetrahedralization between neighboring prisms to avoid inconsistencies in the projection operator. To resolve this ambiguity, we use the technique proposed in [**?**]: we define a total ordering over the vertices of the middle surface of Δ (naturally, we use the vertex id) and

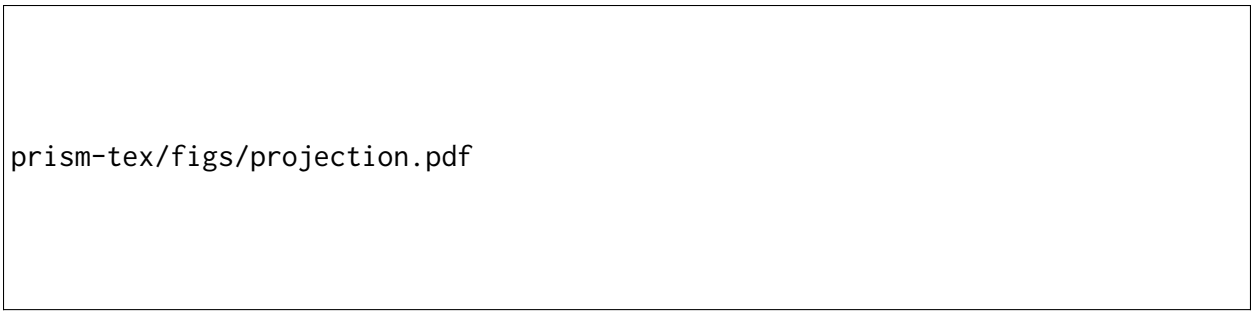


Figure 3.5: A point p (left) is traced through $\mathcal{V}$ inside the top part of the shell. A ray with p as origin and $\mathcal{V}$ as direction is cast inside the orange tetrahedron (middle). The procedure is repeated (on the blue tetrahedron) until the ray hits a point in the middle surface (right).

split (for each half of the prism) the face connecting vertices v_1 and v_2 with a rising edge if $v_1 < v_2$ and a falling edge otherwise.

Forward and Inverse Projection We define a piecewise constant vector field $\mathcal{V}$ inside the decomposed prism, by assigning to each tetrahedron $\boldsymbol{T}_j^\Delta$, $j = 1, \dots, 6$, the constant vector field defined by the only edge of $\boldsymbol{T}_j^\Delta$ which is a *oriented pillar* of Δ connecting the bottom surface to the top surface passing through the middle surface). That is, for any $p \in \boldsymbol{T}_j^\Delta$

$$\mathcal{V}(p) = t_i - b_i, \tag{3.1}$$

where i is the index of the vertex corresponding to the pillar edge of $\boldsymbol{T}_j^\Delta$. Note that $\mathcal{V}$ is constant on each tetrahedron and might be discontinuous on the boundary: we formally define the value of $\mathcal{V}$ on the boundary as any of the values of the incident tetrahedra. This choice does not affect our construction. There is exactly one integral (poly-)line passing through each point of the prism if all the decomposed tetrahedra have positive volumes (Theorem 3.2). This allows us to define the *projection operator* $\mathcal{P}(p)$ for a point $p \in \Delta$ as the intersection of the integral line $f_p(t)$ of the vector field $\mathcal{V}$ passing through p, with the middle surface of Δ (Figure 3.5). Intuitively, we can project any mesh that does not fold in each prism (Figure 3.6) to the middle surface. Formally, we introduce the following definition, to describe this property in terms of the triangle normals of the mesh.

With a slight abuse of the notation, for meshes and collections of prisms A and B, we use $A \cap B$ to denote the intersection of their corresponding geometry.

Definition 3.1. A *section* $\hat{\mathcal{T}}$ of a prism Δ is a manifold triangle mesh whose intersection with Δ is a simply connected submesh $\hat{\mathcal{T}} \cap \Delta$ whose single boundary loop is contained in the boundary of Δ, excluding its top and bottom surface, and such that for every point $p \in \hat{\mathcal{T}} \cap \Delta$ the dot product between the face normal $n(p)$ and the vector field $\mathcal{V}(p)$ is strictly positive. Similarly, a triangle mesh $\hat{\mathcal{T}}$ is a *section of a shell* $\mathcal{S}$, if it is a section of all the prisms of $\mathcal{S}$.

Note that this definition implies that all sections are contained inside the shell. Additionally, the definition implies that the section does not intersect with either bottom or top surface. However,

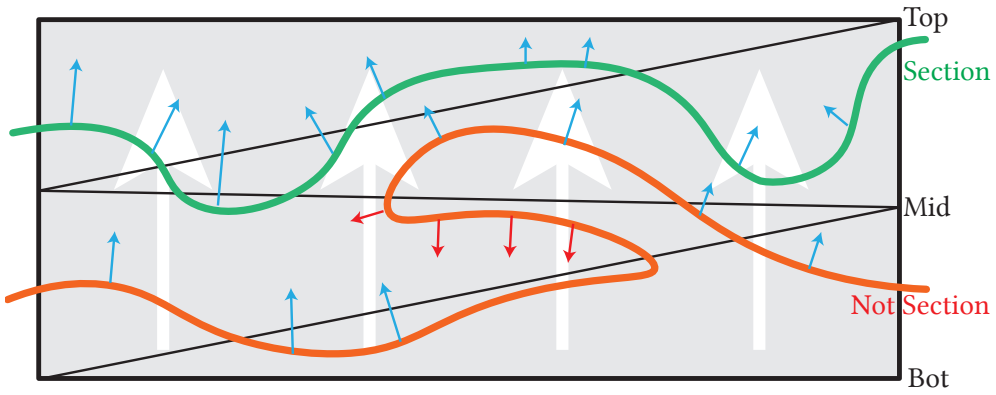


Figure 3.6: A 2D illustration for the normal dot product condition. The blue arrows agrees with the background vector field (white arrows), while the red arrows do not agree.

our definition allows for the bottom or top surface to self-intersect. The intersection of the shell does not invalidate the *local* definition of projection since it is defined per prism. Allowing intersections is crucial to an efficient implementation of our algorithm since it allows us to take advantage of a static spatial data structure in later stages of the algorithm (Section 3.3.4).

Theorem 3.2. *If all 6 tetrahedra T_j^Δ in a decomposition of a prism Δ have positive volume, then the projection operator $\mathcal{P}$ defines a bijection between any section $\hat{\mathcal{T}}$ of Δ and the middle triangle of the prism (M in Figure 3.7).*

Proof. We prove this theorem in Appendix A.1.1. □

The *inverse projection operator* $\mathcal{P}^{-1}$ is defined for a section $\hat{\mathcal{T}}$ as the inverse of the forward projection restricted to $\hat{\mathcal{T}}$. It can be similarly computed by tracing the vector field in the opposite direction, starting from a point in the middle surface of the prism. Note that, differently from the inverse Phong projection [**??**], whose solution depends on the root-finding of a quadric surface, our shell has an explicit form for the inverse and does not require a numerical solve. The combination of forward and inverse projection operators allows to bijectively map between any pair of sections, independently of their connectivity (Figure 3.7). An interesting property of our forward and inverse projection algorithm, which might be useful for applications requiring a provably bijective map, is that our projection could be evaluated exactly using rational arithmetic.

3.3.2 Validity Condition

Shell, projection operator, section definitions, and the bijectivity condition (Theorem 3.2) are dependent on a specific tetrahedral decomposition, which depends on vertex numbering.

To ensure that our shell construction is independent from the vertex and face order, we define the validity of a shell by accounting for all 6 possible tetrahedral decompositions [**?**, Figure 4].

Definition 3.3. We say that a prismatic shell $\mathcal{S}$ is *valid with respect to a mesh* $\hat{\mathcal{T}}$ if it satisfies two conditions for each prism.

Figure 3.7: The composition $\mathcal{P}_{\mathcal{S}_2}^{-1}(\mathcal{P}_{\mathcal{S}_1}(x))$ with $x \in \mathcal{S}_1$, of a direct and an inverse projection operator defines a bijection between two sections $\mathcal{S}_1$ and $\mathcal{S}_2$.

I1 *Positivity.* The volumes of 24 tetrahedra (Appendix A.1.2) corresponding to 6 tetrahedral decompositions are positive.

I2 *Section.* $\hat{\mathcal{T}}$ is a section of $\mathcal{S}$ for all 6 decompositions.

If a shell is valid, from I1, I2, then by Theorem 3.2, it follows that any map between sections induced by the projection operator $\mathcal{P}$ is bijective.

I2 ensures that the input mesh is a *valid section* independently from the decomposition, that is, we require the dot product to be positive with respect to all *three* pillars of a prism inside the convex hull. An interesting and useful side effect of this validity condition is that it ensures the bijectivity of a natural nonlinear parametrization of the prism interior (Appendix A.4, Figure 3.25).

3.3.3 Shell Initialization

We now introduce an algorithm to compute a *valid* prismatic shell $\mathcal{S} = \{(B_{\mathcal{S}}, M_{\mathcal{S}}, T_{\mathcal{S}}), F_{\mathcal{S}}\}$ with respect to a given triangle mesh $\mathcal{T} = \{V_{\mathcal{T}}, F_{\mathcal{T}}\}$ such that $\mathcal{T}$ is geometrically identical to the middle surface of $\mathcal{S}$. We assume that the faces of the triangle mesh are consistently oriented.

Extrusion Direction. The first step of the algorithm is the computation of an extrusion direction for every vertex of $\mathcal{T}$. These directions are optimized to be pointing towards the *outside* of the triangle mesh (which we assume to be orientable), that is, they must have a positive dot product with the normals of all incident faces. More precisely, for a vertex v, we are looking for a direction d_v such that $d_v \cdot n_f > 0$ for each adjacent face f with normal n_f. We can formulate this as the optimization problem

$$\begin{aligned} &\max_{d_v} \min_{f \in N_v} n_f \cdot d_v, \\ &\text{s.t.} \quad n_f \cdot d_v \geq \epsilon, \qquad \forall f \in N_v \\ &\qquad \|d_v\|^2 = 1. \end{aligned} \tag{3.2}$$

A solution, if it exists, can be found solving the following quadratic programming problem (Appendix A.3)

$$\begin{aligned} \min \quad & \|x\|^2, \\ \text{s.t.} \quad & \mathbf{C}x \geq 1, \end{aligned} \tag{3.3}$$

with $d_v = x/\|x\|$ and $\mathbf{C}$ the matrix whose rows are the normals n_f of the faces in the 1-ring N_v of vertex v. Solutions not satisfying $\|x\| \leq 1/\epsilon$ needs to be discarded (Appendix A.3). The QP can be solved with an off-the-shelf solver[??], and in particular it can be solved exactly [?] to avoid numerical problems. Note that the Problem (3.2) is studied in a similar formulation in [?] but their solution requires tolerances in multiple stages of the algorithm to handle cospherical point configurations.

The admissible set of (3.2) might be empty for a vertex v, that is, no vector d_v satisfies $\mathbf{C}d_v \geq \epsilon$. In this case we call v a *singularity*. For example, Figure 3.12 shows a triangle mesh containing a singularity: there exist no direction whose dot product with the adjacent face normals is positive. To simplify the explanation, we assume for the remainder of this section that $\mathcal{T}$ does not contain singularities and also that it does not contain boundaries: we postpone their handling to sections 3.3.5 and 3.3.6.

Proposition 3.4. *Let $\mathcal{T}$ be a closed (without boundary) triangle mesh without singularities and $\mathcal{N}$ be a per-vertex displacement field satisfying $\mathbf{C}\mathcal{N}_i > 0$ for every vertex v_i of $\mathcal{T}$. Then there exist a strictly positive per-vertex* thickness *δ_i such that vertices t_i and b_i obtained by displacing v_i by δ_i in the direction of $\mathcal{N}_i$ and in the opposite direction, define a shell that satisfies invariant I1.*

Proof. We provide a proof in Appendix A.1.2. □

Initial Thickness We first show that a strictly positive per-vertex thickness δ exists for a shell $\mathcal{S}$ with $\mathcal{T}$ as its middle surface, and then discuss a practical algorithm to realize it.

Theorem 3.5. *Given a closed, orientable, self-intersection free triangle mesh $\mathcal{T}$ such that for all its vertices Problem* (3.2) *has a solution, a shell $\mathcal{S}$ exists such that $\mathcal{T}$ is the middle surface and there exist a strictly positive per-vertex thickness δ.*

Proof. We provide a proof in Appendix A.1.3. □

To find a valid per-vertex thickness δ_i to construct the top surface, we initially cast a ray in the direction of $\mathcal{N}$ for each vertex and measure the distance to the first collision with $\mathcal{T}$ (and cap it to a user-defined parameter δ_{max} if no collision is found). An initial top mesh is built with this extrusion thickness; then we test whether any triangle in the top surface intersects $\mathcal{T}$, through triangle-triangle overlap test [?], and iteratively shrink δ_i in this triangle by 20% until we find a thickness that prevents intersections between the input and the top surface. Analogously, we build the bottom shell along the opposite direction. Note that the thickness of a vertex for the top and bottom surface can be different.

Figure 3.8: The vector field aligned with the pillar edge (orange) has a negative dot product with the green triangle normal; the tetrahedron with this field direction meets the green triangle at the red vertex. As a consequence, the mid-surface is not a section. After topological beveling, the shell becomes valid since the dot product between the green normal and the new pillar (purple) is positive.

Validity of the Initial Shell Proposition 3.4 and Theorem 3.5 ensure that the initial shell constructed using a displacement field $\mathcal{N}$ obtained by solving (3.3), satisfies property I1. However, $\mathcal{T}$ might not formally satisfy the conditions for being a section of $\mathcal{S}$, despite being identical to the middle surface of $\mathcal{S}$, due to our definition of the projection operator $\mathcal{P}$. The reason for this can be seen in Figure 3.8. After the initialization, the middle surface is the input mesh. Thus every prism Δ corresponds to a triangle T_Δ of $\mathcal{T}$. The intersection $\Delta \cap \mathcal{T}$, required to check if $\mathcal{T}$ is a section of Δ (Definition 3.1) contains points from both T_Δ and its 1-ring neighborhood.

The projection operator (and thus the definition of the section) is based on all tetrahedral decompositions of Δ; it is possible that multiple tetrahedra (with different vector field values in $\mathcal{V}$) overlap on the boundary of Δ.

Depending on the dihedral angles in the mesh $\mathcal{T}$, it is possible that the dot product between one of the pillars (orange in Figure 3.8) and the face normals of some of the triangles from 1-ring neighborhood (green in Figure 3.8) is negative. While it may seem that this problem could be addressed by changing the definition so that each edge is assigned to one of the incident triangles, so that the field direction only in one incident tetrahedron needs to be considered, this problem is more significant than it may seem, as it leads to instability under small perturbations (e.g., due to floating-point rounding of coordinates). Such small perturbations can change the set of triangles intersecting a prism and thus violate the validity of the shell and, consequently, the bijectivity of $\mathcal{P}$. We propose instead to refine $\mathcal{T}$, without changing its geometry, so that the shell corresponding to the refined mesh satisfies I2 (i.e., its middle surface is a section).

Topological Beveling. We identify a prism Δ_v for which I2 does not hold and use a beveling pattern [**???**] to decompose Δ_v in a way that $\mathcal{T}$ becomes a section for all 6 decompositions (I2). We refer to this operation as *topological beveling*, as it does not change the geometry of the mesh, only its connectivity (Figure 3.10). We use the pattern in Figure 3.9a for Δ_v, and we use the other two patterns (b) and (c) on the adjacent prisms to ensure valid mesh connectivity. The positions

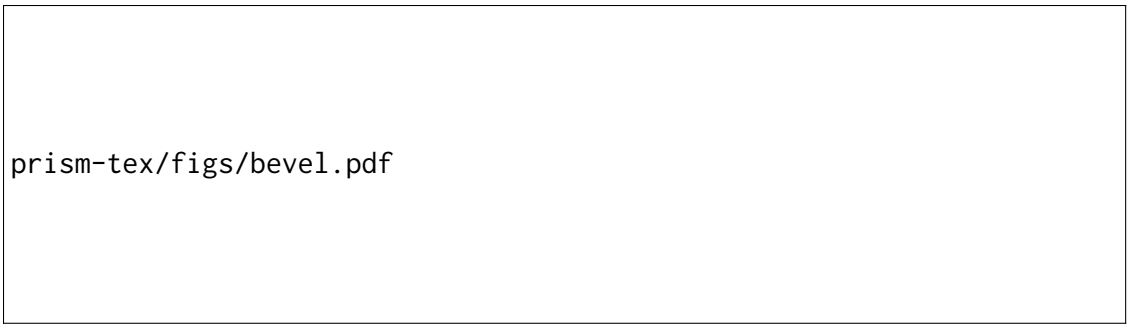


Figure 3.9: The beveling patterns used to decompose prisms for which $\mathcal{T}$ is not a section.

Figure 3.10: Our algorithm refines the input model (left) with beveling patterns (middle) to ensure the generation of a valid shell. The subsequent shell optimizations gracefully remove the unnecessary vertices (right).

of the vertices are computed using barycentric coordinates (we used $t = 0.2$, i.e., the orange dot is at 1/5 of the horizontal edge), and the normals of the newly inserted vertices are copied from the closest vertex (in Figure 3.9, the internal vertices have the normal of the triangle corner with the same color).

Theorem 3.6. *Suppose $\mathcal{T}$ is the middle surface of $\mathcal{S}$, and neither $T_{\mathcal{S}}$ or $B_{\mathcal{S}}$ intersects with $\mathcal{T}$. After*

topological beveling, I2 holds, that is, $\mathcal{T}$ is a section of the shell $\mathcal{S}$ for all 6 decompositions.

Proof. We provide a proof in Appendix A.1.4. □

Output The output of this stage is a valid shell with respect to $\mathcal{T}$ (Section 3.3.2), that is, it satisfies I1 and I2.

3.3.4 Shell Optimization

During shell optimization, we perform local operations (Figure 3.11) on a valid shell to reduce its complexity and increase the quality. Before applying every operation, we check the validity of the operation to ensure that: (1) the resulting middle surface will be manifold [?] and (2) the shell will be valid with respect to $\mathcal{T}$ (to ensure a bijective projection). We forbid any operation that does not pass these checks. We would like to remark that, while there are different choices to guide the shell modification, we experimentally discovered that allowing shell simplification and optimization consistently leads to thicker shells with a richer space of sections.

Theorem 3.7. *Let $\mathcal{S}$ be a valid shell with respect to a mesh $\mathcal{T}$ and let $C = \{\Delta_i\}_{i\in I} \subset \mathcal{S}$ be a collection of prisms such that the middle surface M_C of C is a simply connected topological disk.*

Let $\mathbb{O}$ be an operation replacing C with a new collection of prisms $C' = \{\Delta'_i\}_{i\in I'}$, preserving both geometry and connectivity of the sides of the prism collection C, and ensuring that $M_{C'}$ is a simply connected topological disk.

If these three assumptions hold:

1. *property I1 holds for C',*
2. *the top and bottom surfaces of C' do not intersect $\mathcal{T}$ ($T_{C'} \cap \mathcal{T} = B_{C'} \cap \mathcal{T} = \emptyset$),*
3. *the dot product condition $n(p) \cdot \mathcal{V}(p) > 0$ is satisfied for all points $p \in \mathcal{T} \cap \Delta'_i$ for all pillars of every prism Δ'_i of C',*

then, $\forall i \in I'$, $\mathcal{T} \cap \Delta'_i$ is a simply connected topological disk. In other words, $\mathcal{T}$ is a section of the new shell $\mathcal{S}'$ obtained by applying the operation $\mathbb{O}$ to $\mathcal{S}$.

Proof. We prove this theorem in Appendix A.1.5. □

We note that assumption (2) in Theorem 3.7 prevents the input surface from crossing the bottom/top surface, thus avoiding it to move in the interior of a region covered by more than one prism.

Our local operations (satisfying the definition of $\mathbb{O}$ in Theorem 3.7) are translated from surface remeshing methods [?] since our shell can be regarded as a triangle mesh (middle surface) extruded through a displacement field $\mathcal{N}$. All the local operations described below directly change the middle surface, and consequently affect the extruded shell. After every operation, the middle surface is recomputed by intersecting $\mathcal{T}$ with the edges of the prisms in $\mathcal{S}$.

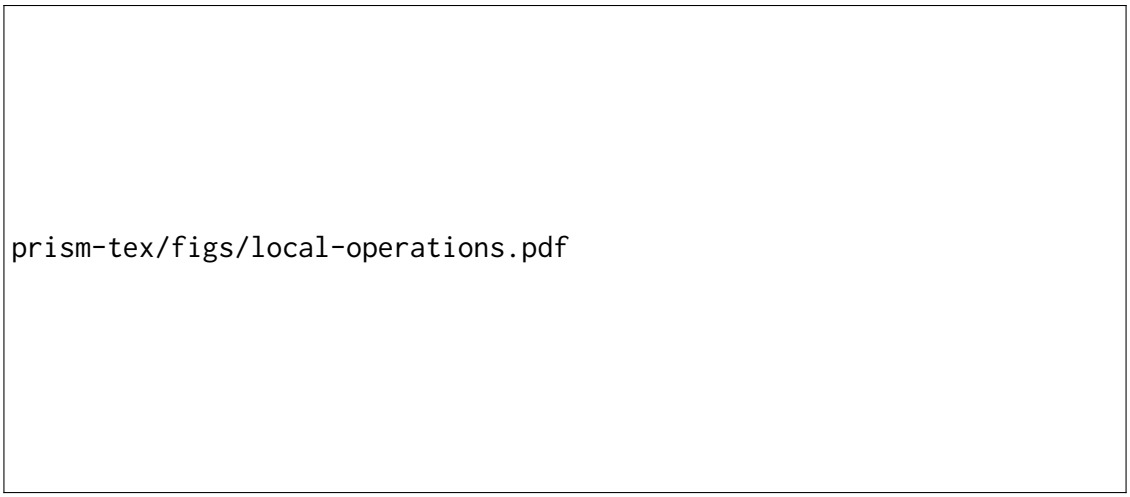


Figure 3.11: Different local operations used to optimize the shell. Mesh editing operations translate naturally to the shell setting. For vertex smoothing, we decompose the operation into 3 intermediate steps: pan, zoom, and rotate.

Shell Quality We measure the quality of the shell $\mathcal{S}$ using the MIPS energy [?] of its middle surface $M_{\mathcal{S}}$. For each triangle T of the middle surface, we build a local reference frame, and compute the affine map $\boldsymbol{J}_T$ transforming the triangle into an equilateral reference triangle in the same reference frame. The energy is then measured by

$$\sum_{T \in M_{\mathcal{S}}} \frac{\mathrm{tr}(\boldsymbol{J}_T^T \boldsymbol{J}_T)}{\det(\boldsymbol{J}_T)}.$$

This energy is invariant to scaling, thus allowing the local operations to coarsen the shell whenever possible while encouraging the optimization to create well-shaped triangles. Good quality of the middle surface decreases the chances, for the subsequent operations, to violate the shell invariants.

Shell Connectivity Modifications. We translate three operations for triangular meshes to the shell settings (Figure 3.11 top). Edge collapse, split, and flip operations can be performed by simultaneously modifying the top and bottom surfaces and retrieve the positions for the middle surface through the intersection. We only accept the operations if they pass the invariant check.

Vertex Smoothing. Due to the additional degree of freedom on vertex-pairs (position, direction, and thickness), we decompose the smoothing operations into three components (Figure 3.11 bottom). *Pan* moves the positions of the top and bottom vertex at the same time, minimizing the MIPS quality of the middle surface. Neither the thickness or direction will be changed. *Rotate* re-aligns the local direction to be the average of the neighboring ones while keeping the position of the middle vertex fixed. *Zoom* keeps the direction and position of the middle vertex, and set the thickness of both top and bottom to be 1.5 times of the neighbor average, capped by the input target thickness.

Invariant Check We use exact orientation predicates [?] to make sure all the prisms satisfy positivity (I1). Further, we ensure that the original surface $\mathcal{T}$ is not intersecting with the bottom and top surface, except at the prescribed singularities. The check is done using the triangle-triangle overlap test [?], accelerated using a static axis-aligned bounding box tree constructed from $\mathcal{T}$. To accelerate the checks for normal condition, for each prism Δ_i, we maintain a list triangles overlapping with its convex hull (an octahedron), and check their respective normals against all the three pillars of Δ_i. These three checks ensure that the three conditions in Theorem 3.7 are satisfied. Note that the vertex smoothing operation is continuous, in the sense that any point between the current position and the optimal one improves the shell. We, however, handle it as a discrete operation to check our conditions: we attempt a full step, and if I1 is not satisfied, we perform a bisection search for a displacement that does. We avoid bisection for the other two conditions since they are expensive to evaluate.

Projection Distortion An optional invariant to maintain (not necessary for guaranteeing bijectivity, but useful for applications), is a bound on the maximal distortion $\mathcal{D}_{\mathcal{P}}(\Delta)$ of $\mathcal{P}$ for a prism Δ. We measure it as the maximal angle between the normals of the set C containing the faces of $\mathcal{T}$ intersecting Δ and $\mathcal{V}$:

$$\mathcal{D}_{\mathcal{P}}(\Delta) = \max_{p \in C} \angle(n_p, \mathcal{V}(p)),$$

where $\angle$ is the unsigned angle in degrees. This quantity is bounded from below by the smallest dihedral angle of $\mathcal{T}$, making it impossible to control exactly. However, we can prevent it from increasing by measuring it and discarding the operations that increase it. In our experiments, we use a threshold of 89.95 degrees.

Scheduling and Termination Our optimization algorithm is composed of two nested loops. The outer loop repeats a set of local operations until the face count between two successive iteration decreases by less than 0.01%. In the inner loop we: (1) flip every edge of $\mathcal{S}$ decreasing the MIPS energy and avoiding high and low vertex valences [?]; (2) smooth all vertices which include pan, zoom, and rotate; and (3) collapse every edge of $\mathcal{S}$ not increasing the MIPS energy over 30. Note that for every operation, we check the invariants, the projection distortion, and manifold preservation and reject any operation violating them. After the outer iteration terminates (i.e. the shell cannot be coarsened anymore), we further optimize the shell with 20 additional iterations of flips and vertex smoothing.

3.3.5 Singularities

Singularities, i.e. vertices of $\mathcal{T}$ for which the constraints set of problem (3.2) is empty, are surprisingly common in large datasets (Figure 3.12 shows an example). For instance, in our subset of Thingi10k [?], although only 0.01% vertices are singular, 8% of the models have at least one singular point. This has been recently observed as a limitation for the construction of nested cages [?, Appendix A], and it is a well-known issue when building boundary layers [???]. There are two main situations that give rise to singular points. The first one naturally generates a singular

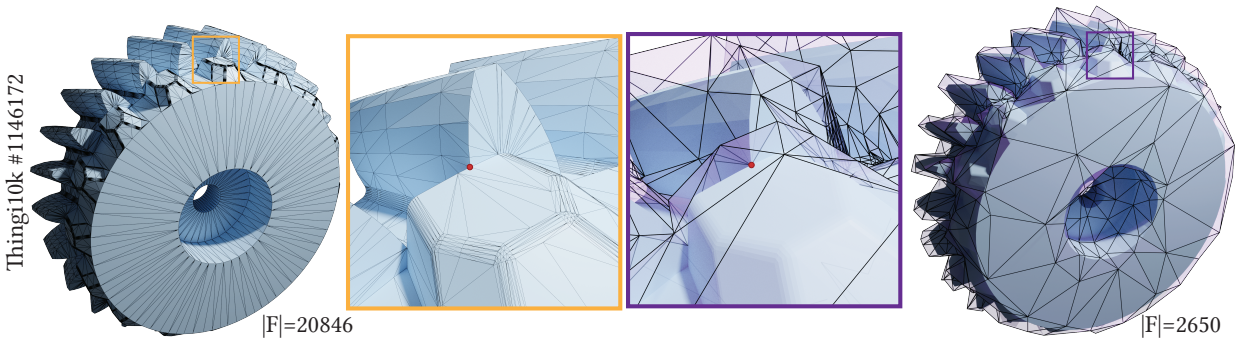


Figure 3.12: A model with 48 singularities and a close up around one (left). Our shell is pinched around each of them without affecting other regions (right).

point when more than two ridge-lines meet (e.g., figures 3.12 and 3.30), thus making the point a feature point. The second one is a pocket-like mesh artifact, often produced as a result of mesh simplification (Figure 3.13).

While singularities might seem fixable by applying local smoothing or subdivision as a pre-process, it is not desirable in the case of a feature point, and is likely to introduce self-intersection or more serious geometric inconsistency. Therefore, due to the above reasons and observing that they are very uncommon, we propose to extend our theory (Appendix A.2) and algorithm to handle isolated singularities by *pinching* the thickness of the shell. Note that in the rare case where two singular points are sharing the same edge, they are *automatically* separated by our topological beveling.

Pinching. We extend our definition of the shell by allowing it to have zero thickness on singularities, thus tessellating the degenerate prism with 4 instead of 6 tetrahedra (Figure 3.13). We further remark that these isolated points must be excluded from Definition 3.1. In the implementation, this requires to change the intersection predicates to skip the singular vertices. With this change, the singularity becomes a trivial point of the projection operator $\mathcal{P}$, and the rest of our shell can still be used in applications without further changes. Since singularities tends to be isolated (they are usually located at the juncture of multiple sharp features), this solution has minimal effects on applications: for example, when our shell is used for remeshing, pinching the shell at singularities will freeze the corresponding isolated vertices while allowing the rest of the mesh to be freely optimized.

The topological beveling algorithm is changed most significantly: for singularities, there is no pillar to copy from. In this case, we apply an additional edge split, to use the pattern in the inset (with the singularity marked by a white dot) in the one-ring neighborhood of the singularity. The newly inserted vertices lie either inside a triangle (uncircled red and orange dots), or in the interior of an edge (circled red and orange dots). Therefore, we assign to the orange vertices the average normal of the two adjacent

prism-tex/figs/singular_bevel.pdf

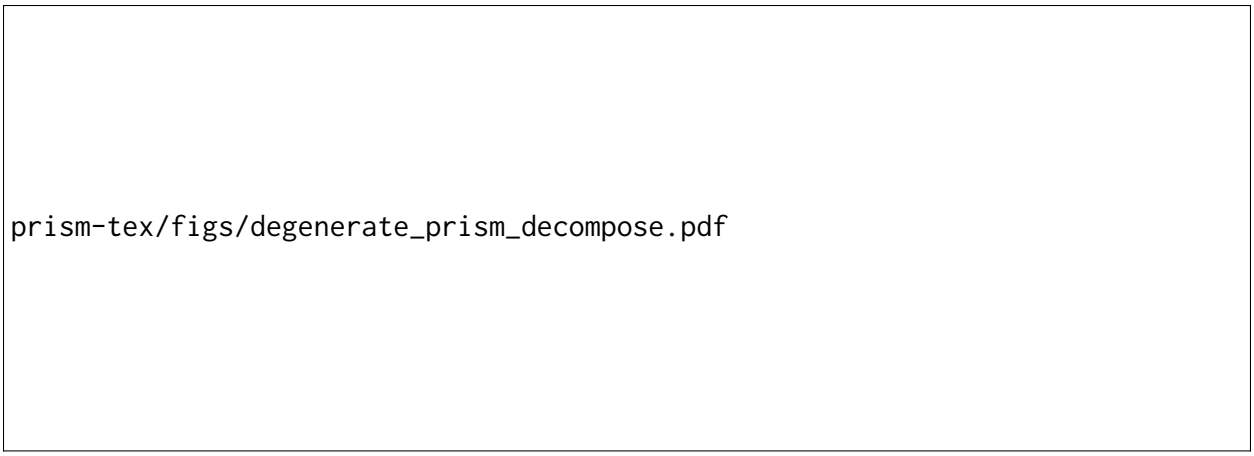


Figure 3.13: Left to right: examples of a singularity as a feature point and as a meshing artefact; and illustration of the degenerate prism with a singularity (red) and its tetrahedral decomposition made of only four tetrahedra.

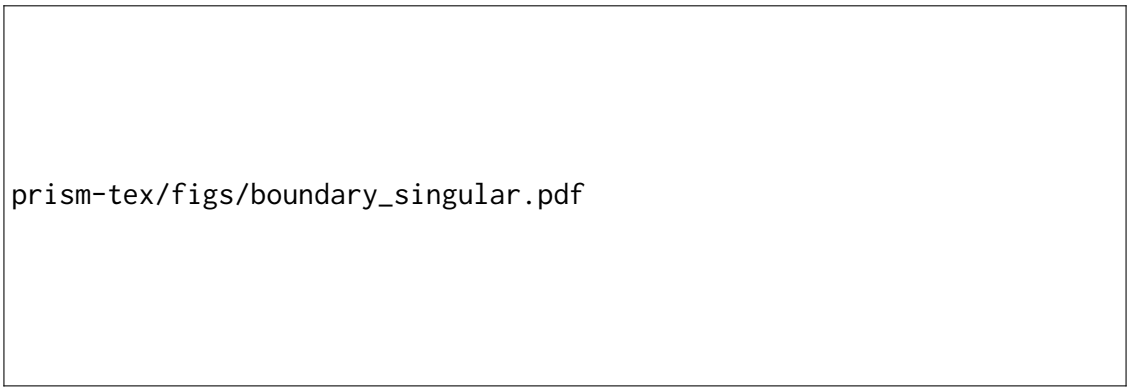


Figure 3.14: AA' is a direction with positive dot product with respect to all its neighboring faces. However, no valid shell can be built following that direction.

triangles, and to red the pillar of the connected orange one.

Additionally, the edges connecting singularities will always be
beveled/split after beveling. Therefore no prism will contain more than one singular point. We discuss the technical extensions for our proofs to shells with pinched prisms in Appendix A.2.

3.3.6 Boundaries.

We introduced our algorithm, assuming that the input mesh does not have boundaries. We will now extend our construction to handle this case, which requires minor variations to our algorithm.

For some vertices on the boundary, it might be impossible to extrude a valid shell (Figure 3.14), even if problem (3.2) has a solution, as Theorem 3.5 does not apply in its original form. We identify such cases by connecting every edge in the 1-ring neighborhood of the boundary vertex to the extruded point and check if they collide with the existing 1-ring triangles (e.g., the triangle $A'AB$

Figure 3.15: An example of a mesh with boundary.

intersects the existing input triangle in Figure 3.14). If it is the case, we consider this vertex as a singularity, and we pinch the shell. Note that this is an extremely rare case and, in our experiments, we detected it only for models where the loss of precision in the STL export introduces rounding noise on the boundary.

Once we pinch all boundary singularities, our construction extends naturally to the boundary. The only necessary modification is in the shell optimization (Section 3.3.4), where we skip all operations acting on boundary vertices to maintain the bijectivity of the induced projection operator (Figure 3.15). We thus freeze these vertices and never allow them to move or be affected by any other modification of the shell. Note that, in certain applications, it might also be useful to freeze additional non-boundary vertices to ensure that these remain on the middle surface during optimization (e.g., to exactly represent a corner of a CAD model).

3.4 Results

Our algorithm is implemented in C++ and uses Eigen [**?**] for the linear algebra routines, CGAL [**?**] and Geogram [**?**] for predicates and spatial searching, and libigl [**?**] for basic geometry processing routines. We run our experiments on cluster nodes with a Xeon E5-2690 v2 @ 3.00GHz. The reference implementation used to generate the results is attached to the submission and will be released as an open-source project.

Robustness For each dataset, we selected the subset of meshes satisfying our input assumptions: intersection-free, orientable, manifold triangle meshes without zero area triangles (tested using a numerical tolerance 10^{-16}). We test self-intersections by two criteria: a ball of radius 10^{-10} around each vertex does not contain non-adjacent triangles; and all the dihedral angles are larger than 0.1 degrees.

We tested our algorithm on two datasets: (1) Thingi10k dataset [**?**] containing, after the filtering due to our input assumptions, 5018 models; and (2) the first chunk of for the ABC dataset [**?**] with 5545 models. The only user-controlled parameter of our algorithm is the target thickness of our shell; in all our experiments (unless stated otherwise), we use 10% of the longest edge of the

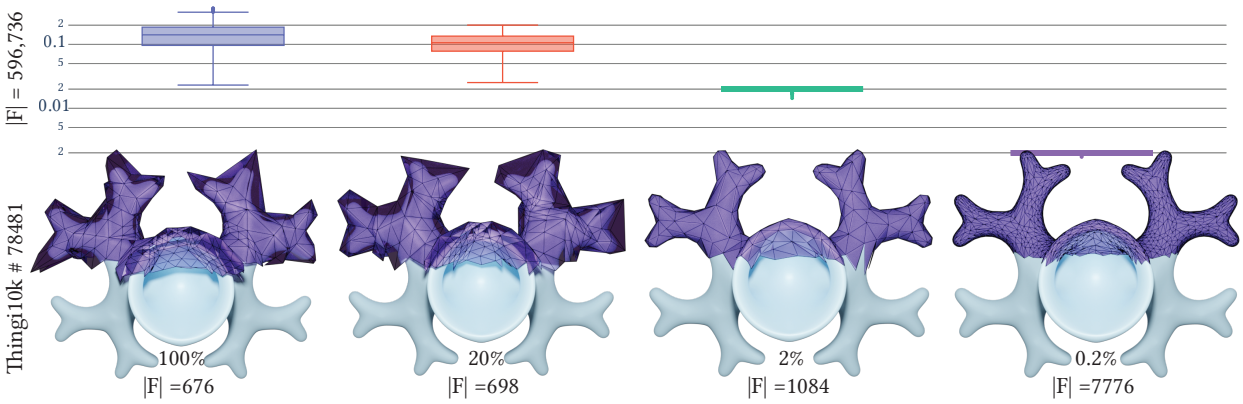


Figure 3.16: The effect of different target thickness on the number of prisms |F| of the final shell, and distribution of final thickness (shown as the box plots on the top).

bounding box. In Figure 3.16, we show how the target thickness influences the usage of the shell: a thicker shell provides a larger class of sections, thus accommodates more processing algorithms, while a thinner one offers a natural bound on the geometric fidelity of the sections.

Our algorithm successfully creates shells for all 5018 models for Thingi10k and 5545 for ABC. We show a few representative examples of challenging models for both datasets in Figure 3.17, including models with complicated geometric and topological details. In all cases, our algorithm produces coarse and thick cages, with a bijective projection field defined.

We report as a scatter plot the number of output faces, the timing, and the memory used by our algorithm (Figure 3.18). In total, the number of prisms generated by our algorithm is 7% and 2% of the number of input triangles for the Thingi10k and ABC dataset respectively and runs with no more than 4.7 GB of RAM. The generation and optimization of the shell takes 5min and 59s in average and up to 8.6 hours for the largest model. 50% of the meshes finish in 3 minutes and 75% in 6 minutes and 15 seconds.

Comparison to Simple Baselines In Figure 3.19 we compare to two baseline methods based on [**?**]. For each method, we generate a coarse mesh, uniformly subdivide it for visualization purposes, and query the corresponding spatial position on the original input to form the subdivided mesh. The *UV based* method is a conventional way of establishing correspondence in the context of texture mapping. However, the robust generation of a UV atlas satisfying a variety of user constraints is still an open problem. We use the state-of-the-art methods [**??**] to generate a low-distortion bijective parametrization, and use seam-aware decimation technique [**?**] to generate the coarse mesh. Due to the complex geometry and the length of the seam (Figure 3.19 second figure), the simplification is not able to proceed beyond the prescribed seam while maintaining the bijectivity, making the pipeline inadequate especially for building computational domains.

We also set up a baseline of the *Naive Cage* method by creating a simplified coarse mesh with [**?**] and use Phong projection to establish the correspondence [**??**]. Such attribute transfer is not guaranteed to be bijective; some face may not be projected (Figure 3.19 third image). With

Thingi10k

ABC

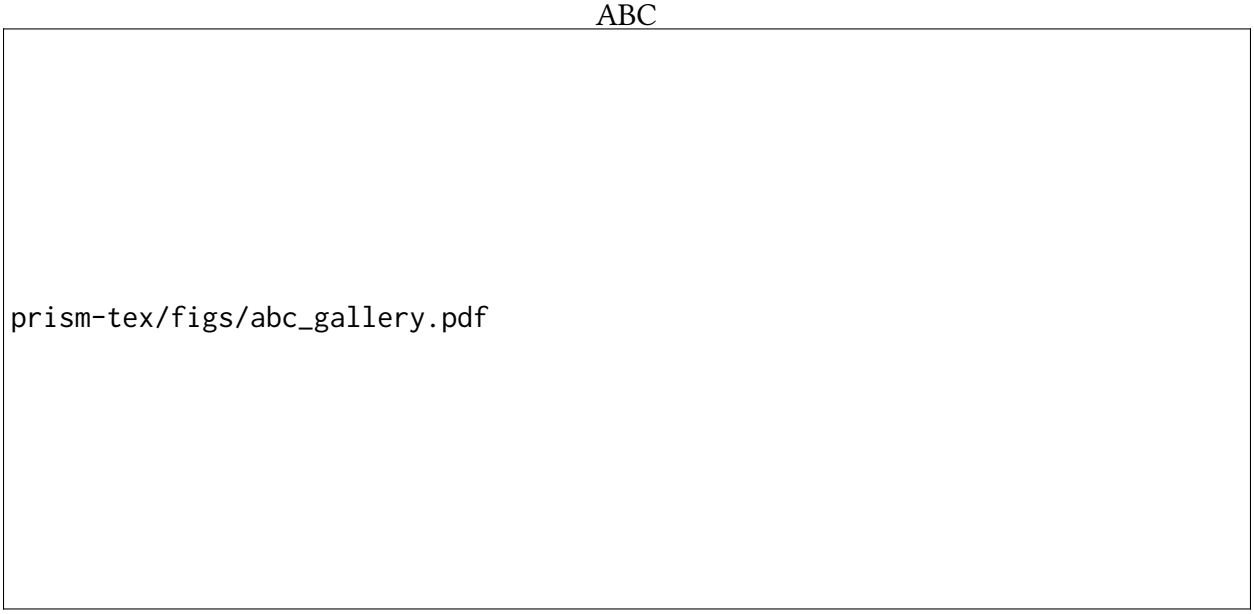


Figure 3.17: Gallery of shells built around models from Thingi10k [?] and ABC [?].

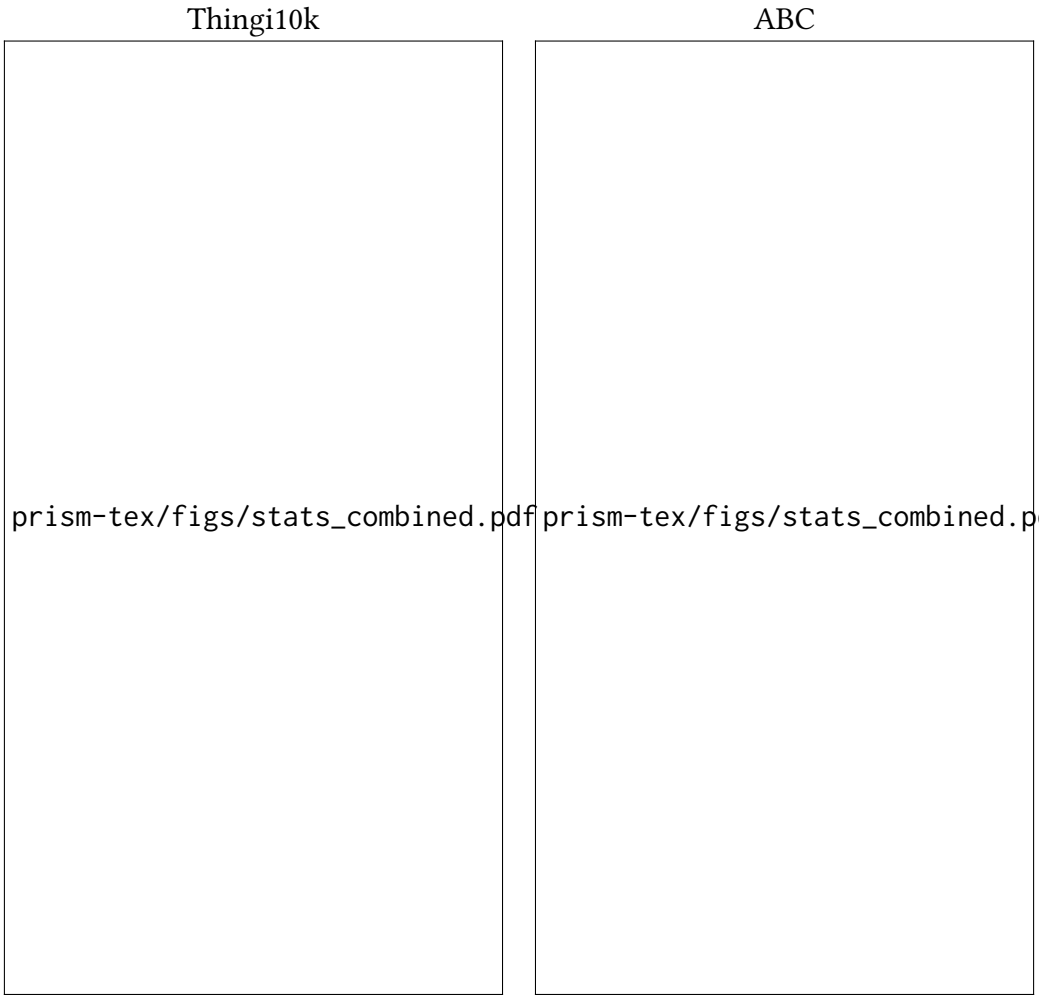


Figure 3.18: Statistics of 5018 shells in Thingi10k dataset [?] (left) and 5545 in ABC Dataset [?] (right).

our method, we can generate a coarse mesh while having a low-distortion bijective projection (Figure 3.19 fourth image).

Numerical accuracy To evaluate the numerical error introduced by our projection operator when implemented with floating-point arithmetic, we *transfer* the vertices of the input mesh to the middle surface and *inverse transfer* them from the middle surface back to the input mesh. We measure the Euclidean distance with respect to the source vertices (Figure 3.20). We compare the same experiment with the Phong projection [?]. This alternative approach exhibits distance errors up to 10^{-5} even after ruling out the outliers for which the method fails due to its lack of bijectivity. The maximal error of our projection is on the order of 10^{-8}; this error could be completely eliminated (for applications requiring an exact bijection) by implementing the projection operator and its inverse using rational arithmetic.

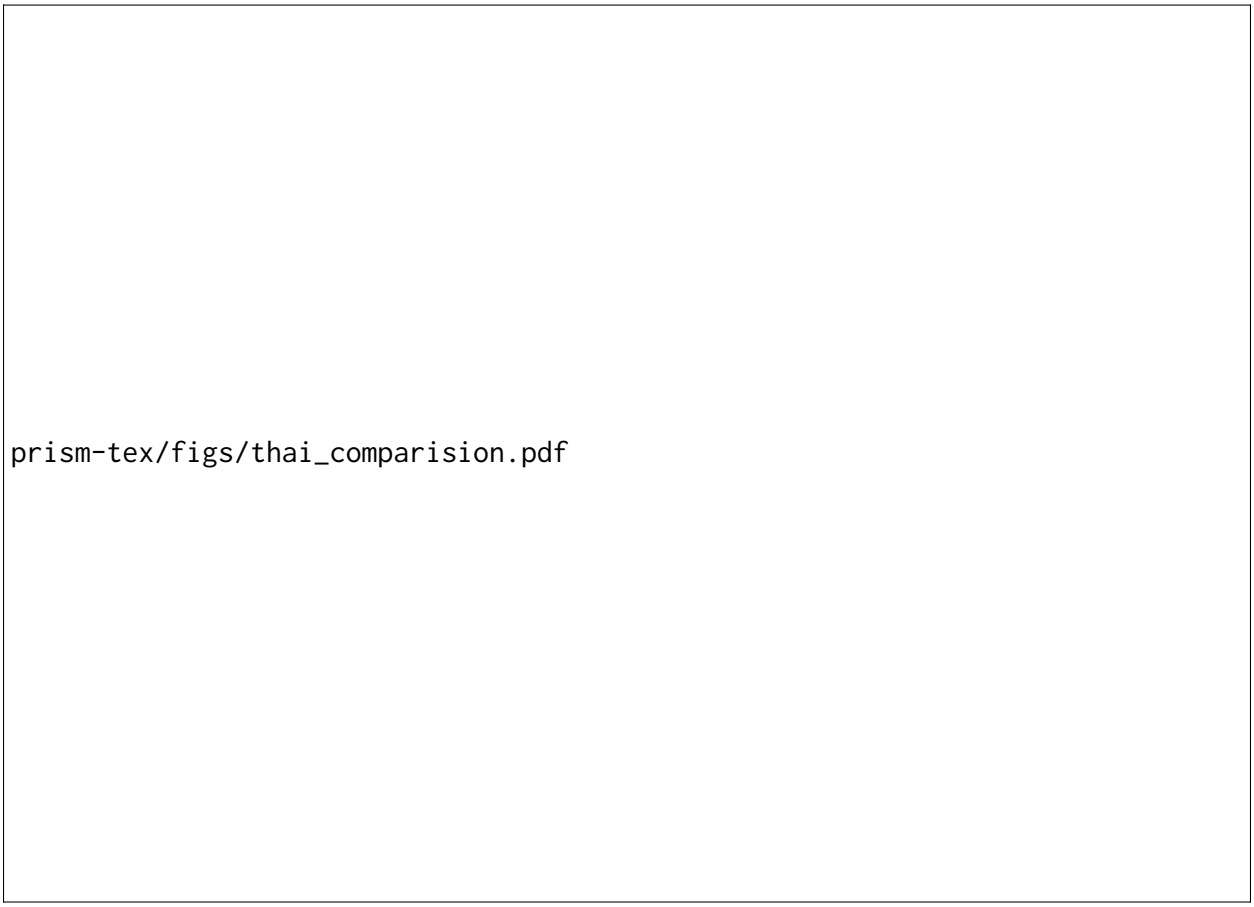


Figure 3.19: The *UV based method* cannot simplify the prescribed seams, and introduces self-intersections. The projection induced by the *Naive Cage* method is not continuous and not bijective; it leads to visible spikes in the reconstructed geometry.

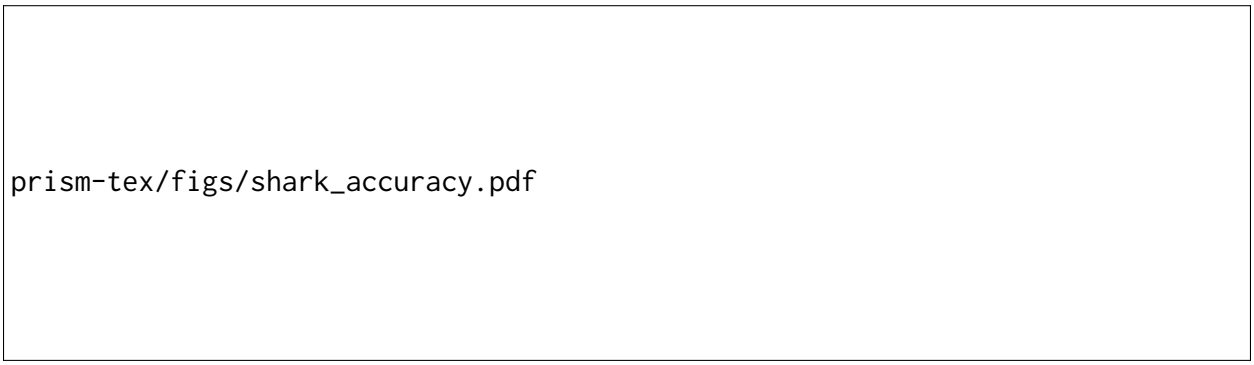


Figure 3.20: Our projection is three orders of magnitude more accurate than the baseline method, and bijectively reconstructs the input vertex coordinates.

3.5 Applications

Using our shell $\mathcal{S}$, we implement the following predicates and functions:

- is_inside(p): returns true if the point $p \in \mathbb{R}^3$ is inside $\mathcal{S}$.
- is_section($\mathcal{T}$): returns true if the triangle mesh $\mathcal{T}$ is a section of $\mathcal{S}$.
- $\mathcal{P}(p)$: returns the prism id (pid), the barycentric coordinates (α, β) in the corresponding triangle of the middle surface, and the relative offset distance from the middle surface (h, which is -1 for the bottom surface, and 1 for the top surface) of the projection of the point p.
- $\mathcal{P}^{-1}(\text{pid}, \alpha, \beta, h)$ is the inverse of $\mathcal{P}(p)$.
- $\mathcal{P}_{\mathcal{T}}(\text{tid}, \alpha, \beta) = \mathcal{P}(q)$, where q is the point in the triangle tid of the mesh $\mathcal{T}$, with barycentric coordinates α, β.
- $\mathcal{P}^{-1}_{\mathcal{T}}(\text{pid}, \alpha, \beta)$ is the inverse of $\mathcal{P}_{\mathcal{T}}$.

As explained in Section 3.3, our shell may self-intersect and we opted to simply exclude the overlapping regions. In practice this affects only the function is_inside(p) which needs to check if p is contained in two or more non-adjacent prisms.

These functions are sufficient to implement all applications below, demonstrating the flexibility of our construction and how easy it is to integrate in existing geometry processing workflows.

Remeshing We integrated our shell in the meshing algorithm proposed in [**?**] by adding envelope checks ensuring that the surface is a section after every operation. After simplification, we can use the projection operator to transfer properties between the original and remeshed surface (e.g., figures 3.22, 3.23). Since the remeshed surface is a section, a very practical side effect of our construction is that the remeshed surface is *guaranteed to be free of self-intersections*, As shown in Figure 3.21, the constraints enforced through our shell prevents undesirable geometric configurations (intersections, pockets, or triangle flips).

Proxy A particularly useful application of our shell is the construction of proxy domains for the solutions of PDEs on low-quality meshes. Additionally, by specifying the target thickness parameter (Figure 3.16), we are able to bound the geometry approximation error to the input as well. Using our method, we can (1) convert a low-quality mesh to a proxy mesh with higher quality and desired density, (2) map the boundary conditions from the input to the proxy using the bijective projection map, (3) solve the PDE on the proxy (which is a standard mesh), and (4) transfer back the solution on the input surface (Figure 3.22). Our algorithm can be directly used to solve volumetric PDEs. by calling an existing tetrahedral meshing algorithm between steps 2 and 3 (figures 3.1, 3.23). In this case, we control the geometric error by setting the target thickness (we use 2% of the longest edge of the bounding box).

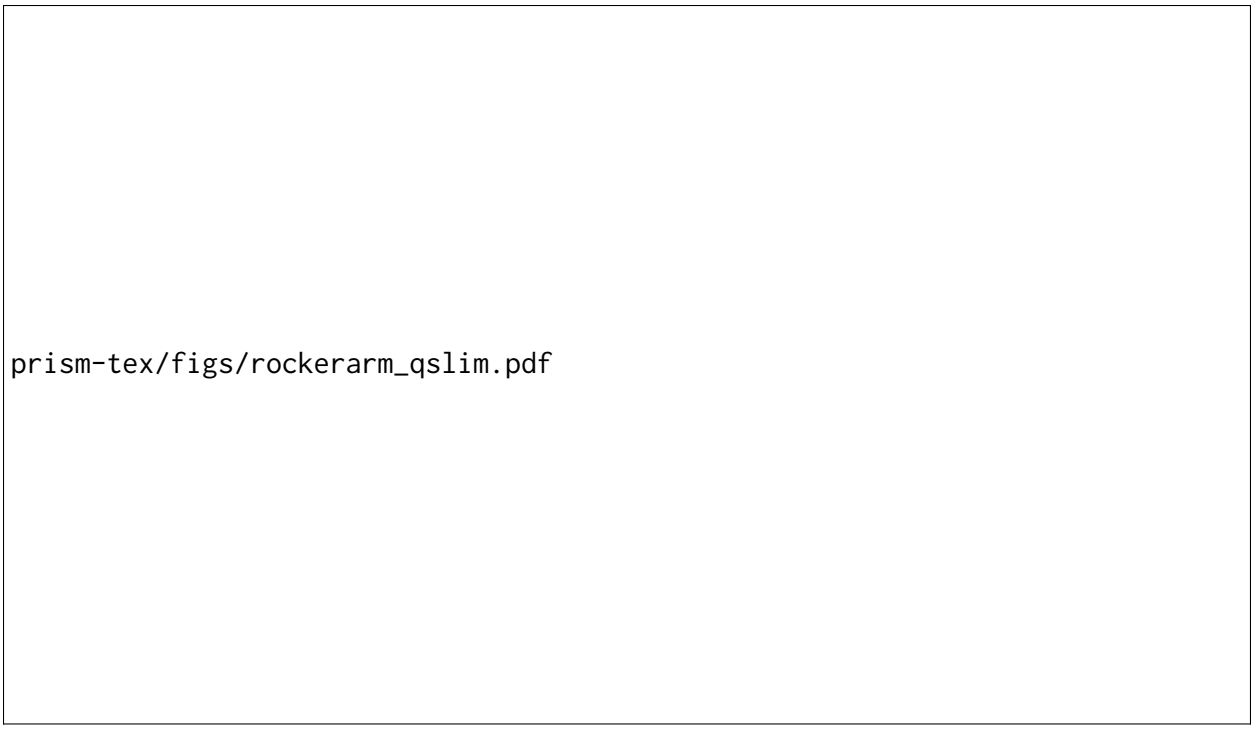


Figure 3.21: We attempt to simplify *rockerarm* (top left) from 20088 triangles to 100. QSlim [**?**] succeeds in reaching the target triangle count (top right) but generates an output with a self-intersection (red) and flipped triangles (purple). With our shell constraints (bottom left), the simplification stagnates at 136 triangles, but the output is free from undesirable geometric configurations. Note that both examples use the same quadratic error metric based sceduling [**?**].

Boolean Operations The mesh arrangements algorithm enables the robust and exact (up to a final floating-point rounding) computation of Boolean operations on PWN meshes [**?**]. However, the produced meshes tend to have low triangle quality that might hinder the performance of downstream algorithms. By interleaving a remeshing step performed with our algorithm after every operation, we ensure high final quality and more stable runtime. The composition of the bijections enables us to transfer properties between different nodes of the CSG tree (Figure 3.24).

Displacement Mapping The middle surface of the shell is a coarse triangular mesh that can be directly used to compress the geometry of the input mesh, storing only the coarse mesh connectivity and adding the details using normal and displacement maps (Figure 3.25). A common way to build such displacement is to project [**??**] the dense mesh on the coarser version. As shown in Appendix A.4, our method guarantees that this natural projection is also bijective as long as the coarse mesh is a section. This alleviates the loss of information even on challenging geometry configurations, and our shell can thus be used to automate the creation of projection cages and displacement maps.

Figure 3.22: The heat method (right top) produces inaccurate results due to a poor triangulation (left top). We remesh the input with our method (left bottom), compute the solution of the heat method [?] on the high-quality mesh (middle bottom) and transfer the solution back to the input mesh using the bijective projection (bottom right). This process produces a result closer to the exact discrete geodesic distance [?] (top middle, error shown in the histograms).

Geometric Textures. The inverse projection operator provides a 2.5D parametrization around a given mesh and can be used to apply a volumetric texture (Figure 3.26). Note that we build the volumetric texture on the simplified shell, while still being able to bijectively transfer the texture coordinates.

3.6 Variants

Input with Self-Intersections Up to this point, we assumed that our input meshes are without self-intersections. This requirement is necessary to guarantee a bijection between any section (e.g., the input mesh) and the middle surface. Such bijection is essential for a key target application, the transfer of boundary conditions for solving PDEs on meshes or mesh-bounded domains.

Figure 3.23: *Knot* simplified with our method and tetrahedralized with TetGen. The original model is converted to 1,503,428 tetrahedra (left) while the simplified surface is converted to only 176,190 tetrahedra (right).

However, our method can be easily extended to meshes containing self-intersections, broadening the class of meshes it can be applied to, at the cost of making the resulting shell usable in fewer application scenarios: for example, if it is used for remeshing, it will likely generate a new surface that still contains self-intersections.

If $\mathcal{T}$ contains self-intersections, our algorithm can be trivially extended to generate a shell which will be *locally injective*, and the bijectivity of the mapping between sections still holds but with respect to the immersion. The only change required is to modify the invariance checks (Section 3.3.4): we have to replace the global intersection check with checking whether local triangles overlap with the current prisms. Figure 3.27 shows an example of a mesh $\mathcal{T}$ with self-intersections, the generated shell, and the isolines of geodesic transferred on the coarser middle surface.

Resolving Shell Self-Intersections For certain applications it might be preferable to have a shell whose top and bottom surfaces do not self-intersect: for example, in the construction of nested cages [**?**] (useful for collision proxies and animation cages), we want to iteratively build nested shells while ensuring no intersections between them (Figure 3.28). With a small modification, our algorithm can be used to generate nested cage automatically and robustly, with the additional advantage of being able to map any quantity bijectively across the layers and to the input mesh. In contrast, [**?**] does not provide guarantees on the success (e.g., the reference implementation of [**?**] fails on Figure 3.19, probably due to the presence of a singularity).

To resolve the self-intersections of the top (bottom) surface, we identify the regions covered by more than one prism by explicitly testing intersections between the tetrahedralized prisms,

Figure 3.24: The union of two meshes is coarsened through our algorithm, while preserving the exact correspondence, as shown through color transfer.

accelerated using [**?**]. For every detected prism, we reduce the thickness by 20%, and iterate until no more intersections are found. Differently from the procedure in Section 3.3.3, where reducing the thickness of the shell always maintains the validity of the shell, at this stage, the shrinking of the shell may make the shell invalid, since $\mathcal{T}$ may not be contained anymore in $\mathcal{S}$. Whenever this happens, we perform one step of red-green refinement [**?**] on the regions we wish to thin, and we iterate until we succeed. This procedure is guaranteed to terminate since, on the limit of the refinement, the middle surface will be geometrically identical to $\mathcal{T}$, and thus Theorem 3.5 holds. In the worst case, the procedure terminates when the size of triangles on the middle surface is comparable to the input; then no refinement is required to shrink below the minimum separation of the input. Figure 3.29 shows how the intersecting shell between the legs of the camel can be shrunk to generate an intersection-free shell.

Pinching Alternative. For certain applications, such as boundary layer meshing, it is necessary to have a shell with non-zero thickness everywhere, including at singularities, and it is tolerable to

Figure 3.25: Top: an input mesh decimated to create a coarse base mesh, and the details are encoded in a displacement map (along the normal) with correspondences computed with Phong projection [?]. Bottom: the decimation is done within our shell while using the same projection as above. With our construction, this projection becomes bijective (Appendix A.4), avoiding the artifacts visible on the ears of the bunny in the top row.

Figure 3.26: Two volumetric chainmail textures (right) are applied to a shell (bottom left) constructed from the *Animal* mesh (top left). The original UV coordinates are transferred using our projection operator.

Figure 3.27: An example of a shell built around a self-intersecting mesh.

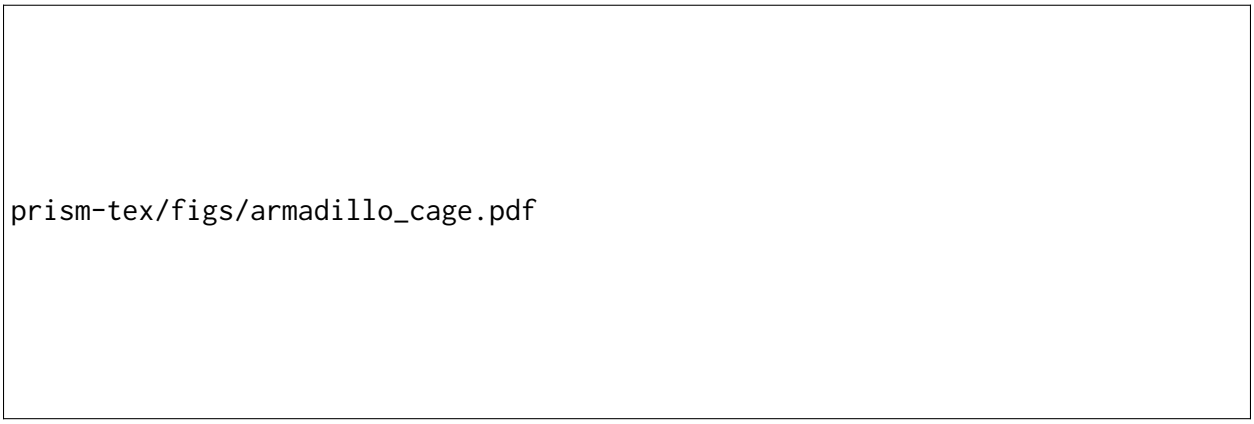


Figure 3.28: The *Armadillo* model with four nested cages. We create a shell from the original mesh, and then rerun our algorithm on the outer shell to create the other three layers. Note that all layers are free of self-intersections, and we have an explicit bijective map between them.

lose bijectivity at the vicinity of a singularity. For these cases, we propose a Boolean construction to *fill* the shell around singularities, and to extend the projection operator $\mathcal{P}$ inside these regions. That is, every point in the filled region will project to the singularity.

Without loss of generality, let us assume that $\mathcal{T}$ has a single singularity (Figure 3.30). We initially construct a pinched shell, with zero thickness at the singularity, construct a valid shell (Section 3.3), and then perform a corefinement [**?**] between a tetrahedron (centered at the singularity and whose size is smaller than the minimal thickness of the neighboring vertices) and the shell. The result of the corefinement operation (Figure 3.30 middle) consists of triangles belong to the tetrahedron, or the shell surface. The remaining part of the tetrahedron is a star-shaped polyhedron with the singularity in its kernel, and sharing a part of its boundary with the shell. This polyhedron can be easily tetrahedralized by connecting its triangulated boundary faces (one of them is highlighted in red in Figure 3.30 middle) with the singularity. For every point p in these tetrahedra, the projection operator $\mathcal{P}$ projects p to the singularity. The remaining triangles are divided into two groups: the triangles with only one new vertex complete the degenerate prisms (one of them is highlighted in blue in Figure 3.30 middle), while the others map to the edges they

Figure 3.29: After optimization, the shell may self-intersect (left). Our postprocessing can be used to extract a non-selfintersecting shell, which is easier to use in downstream applications (right).

Figure 3.30: We generate a pinched shell for a model with a singularity (left). Optionally, we can complete the shell using our Boolean construction.

are attached to.

3.7 Limitations

Currently, our algorithm is limited to manifold and orientable surfaces: its extension to non-manifold and/or non-orientable meshes is a potential venue for future work. With such an extension, the integration of shells with robust tetrahedral meshers [**??**] would allow to solve PDEs on imperfect triangle meshes without ever exposing the user to the volumetric mesh, allowing them to directly work on the boundary representation to specify boundary conditions and to analyze the solution of the desired PDE. Since we rely on the additional checks for the bijective constraints, our method is slower than classical surface mesh adaptation algorithms and it is not suitable for interactive applications.

Integrating our approach into existing mesh processing algorithms might lose some of their guarantees or properties since our shell might prevent some local operations. Other surface processing algorithms guarantee some properties under some regularity assumptions on the input, which might not hold when our bijective constraints are used. For example, QSlim [**?**] might not be able to reach the desired target number of vertices and [**?**] might not be able to achieve the bounded aspect ratio. A practical limitation is that integrating our approach into existing remeshing or simplification implementations requires code level access.

Our shell is ideal for triangle remeshing algorithms employing incremental changes: not every geometry processing algorithm requiring a bijective map can use our construction. For example, it is unclear how isosurface-extraction methods [**?**] could use our shell or how global parametrization algorithms [**????**] could benefit from our method since they already compute a map to a common domain.

3.8 Concluding Remarks

We introduce an algorithm to construct shells around triangular meshes and define bijections between surfaces inside the shell. We proposed a robust algorithm to compute the shell, validated it on a large collection of models, and demonstrated its practical applicability in common applications in graphics and geometry processing.

We believe that many applications in geometry processing could benefit from bijectively mapping spatially close surfaces, and that the idea of using an explicit mesh as a common parametrization domain could be extended to the more general case of computing cross-parametrizations between arbitrary surfaces. To foster research in this direction, we will release our reference implementation as an open-source project.

4 | Bijective and Coarse High-Order Tetrahedral Meshes

4.1 Introduction

Piecewise linear approximations of surfaces are a popular representation for 3D geometry due to their simplicity and wide availability of libraries and algorithms to process them. However, dense sampling is required to faithfully approximate smooth surfaces. Curved meshes, that is, meshes whose element's geometry is described as a high-order polynomial, are an attractive alternative for many applications, as they require fewer elements to achieve the same representation accuracy of linear meshes. In particular, curved meshes have been shown to be effective in a variety of simulation settings in mechanical engineering, computational fluid dynamics, and graphics. Despite their major benefits, they are not as popular as linear meshes: We believe that one of the main reason for their limited usage is the lack of an automatic, robust way of constructing them.

While robust meshing algorithm exists for volumetric linear tetrahedral meshing, there are few algorithms for curved meshes, and even fewer of them having either a commercial or open-source implementation (Section 5.2). Only a few algorithms work directly on arbitrary triangle meshes (most of them require the input geometry to be either a CAD file or an implicit function), and

Figure 4.1: Our pipeline starts from a dense *linear* mesh with annotated features (green), which is converted in a curved shell filled with a high-order mesh. The region bounded by the shell is then tetrahedralized with linear elements, which are then optimized. Our output is a coarse, yet accurate, curved tetrahedral mesh ready to be used in FEM based simulation. Our construction provides a bijective map between the input surface and the boundary of the output tetrahedral mesh, which can be used to transfer attributes and boundary conditions.

none of them can reliably process a large collection of 3D models.

We propose the first robust and automatic algorithm to convert dense piecewise linear triangle meshes (which can be extracted from scanned data, volumetric imaging, or CAD models) into coarse, curved tetrahedral meshes equipped with a bijective map between the input triangle mesh and the boundary of the tetrahedral mesh. Our algorithm takes advantage of the recently proposed bijective shell construction [**?**] to allow joint coarsening, remeshing, and curving of the dense input mesh, which we extend to support feature annotations. Our outputs are guaranteed to have a self-intersection free boundary and the geometric map of every element is guaranteed to be bijective, an important requirement for FEM applications.

The key ingredient of our algorithm is the separation of the curved volumetric meshing problem into a near-surface, surface curving problem, followed by a restricted type of linear volumetric meshing.

We believe that our curved meshing algorithm will enable wider adoption of curved meshes, as it will provide a way to automatically convert geometric data in multiple formats into a coarse tetrahedral mesh readily usable in finite element applications. To showcase the benefits of our approach we study 2 settings: (1) we show that a coarse proxy mesh can be used to compute non-linear deformations efficiently and transfer them onto a high-resolution geometry, targeting real-time simulation (Figure 4.1), and (2) we show that our meshes are ready to use in downstream FEM simulations.

We validate the reference implementation of our approach on a large collection of more than 8000 geometrical models, which will be released as an open-source project to foster the adoption of curved meshes in academia and industry.

Our contributions are:

- An algorithm to convert dense piecewise linear meshes into coarse curved volumetric meshes while preserving a bijective map with the input and bounding the approximation error.
- An extension of the bijective shell construction algorithm to support feature annotations.
- An algorithm for conforming tetrahedral meshing without allowing refinement on the boundary, but allowing internal Steiner points.
- A large-scale dataset of high-order tetrahedral meshes.

4.2 Related Works

We review the literature on the generation of unstructured and structured curved meshes. We also review the literature on boundary-preserving linear meshing, as it is an intermediate step of our algorithm.

4.2.1 Curved Tetrahedral Mesh Generation

High-order meshes are used in applications in graphics [??Suwelack et al. 2013] and engineering analysis [Jameson et al. 2002] where it is important to reduce the geometric discretization error [Babuska and Guo 1988; ?; ?; ?; ?], while using a low number of degrees of freedom. The creation of high-order meshes is typically divided into three steps: (1) linear meshing of the smooth input surface, (2) curving of the linear elements to fit the surface, and (3) optimization to heal the elements inverted during curving. We first cover steps 2 and 3, and postpone the overview of linear tetrahedral algorithms to Section 4.2.3.

Direct methods. Direct methods are the simplest family of curving algorithms, as they explicitly interpolate a few points of the target curved surface or project the high-order nodes on the curved boundary [?Ghasemi et al. 2016; ?; Abgrall et al. 2012; ?; ?; ?]. The curved elements are represented using Lagrange polynomials, [??], quadratic or cubic Bézier polynomials [George and Borouchaki 2012; ?; ?], or NURBS [??]. [??] further optimizes the high-order node distribution according to geometric quantities of interest, such as length, geodesic distance, and curvature. In the case where no CAD information is available, [?], [?] use smooth reconstruction to compute high-order nodes and perform curving.

Deformation methods Deformation methods consider the input linear mesh as a deformable, elastic body, and use controlled forces to deform it to fit the curved boundary. Different physical models have been employed such as linear, [Abgrall et al. 2014, 2012; Dobrzynski and El Jannoun 2017; Xie et al. 2013], and (variants of) non-linear elasticity [Persson and Peraire 2009; ?; ?]. A comparison between different elasticity and distortion energies is presented in [Poya et al. 2016; ?; ?].

Direct and deformation methods have been tested on small collections of simple models, and, to the best of our knowledge, none of them can provide guarantees on the validity of the output or has been tested on large collections of models. There are also no reference implementations we could compare against.

Inversions and Intersections. Most of these methods introduce inverted elements during the curving of the high-order elements. Inverted elements can be identified by extending Jacobian metrics for linear elements [?Knupp 2000] to high-order ones [???Poya et al. 2016; Roca et al. 2012]. Various untangling strategies have been proposed, including geometric smoothing and connectivity modifications [Cardoze et al. 2004; ?; ?; George and Borouchaki 2012; ?; ?; ?; ?; Dobrzynski and El Jannoun 2017; ?; Geuzaine et al. 2015; Roca et al. 2012; ?; ?; ?; ?; ?; ?; ?; ?; ?; ?; ?; ?; ?; ?]. None of these techniques can guarantee to remove the inverted elements.

An alternative approach is to start from an inversion-free mesh and slowly deform it [Persson and Peraire 2009; ?], explicitly avoiding inversions at the cost of possibly inaccurate boundary reproductions. These methods cannot however guarantee that the boundary will not self-intersect. Our approach follows a similar approach but uses a geometric shell to ensure element validity and prevention of boundary self-intersections.

Curved Optimal Delaunay Triangulation [?] generalize optimal delaunay triangulation paradigm to the high-order setting, through iteratively update vertices and connectivity. Their algorithm starts with a point cloud sampled from triangle meshes. However, the success of the method depends on the choice of final vertex number and sizing field, where insufficient vertices may result in broken topology or invalid tetrahedral meshes.

Software Implementation. Despite the large literature on curved meshing generation, there are very few implementations available.

Nektar [?] is a finite element software with a meshing component, which can generate high-order elements. We do not explicitly compare as their documentation (Section 4.5.1.5 Mesh Correction[1]) states that the algorithm is not fully automatic and not designed to process robustly large collections of models. Gmsh [?] is an open source software that supports the curved meshing of CAD models, but it does not support dense linear meshes as input. Despite the difference in the input type, we provide a comparison with Gmsh in Section 4.6.2, as it is the only method that we could run on a large collection of shapes.

To the best of our knowledge, the commercial software that support curved meshing (Pointwise [??]) are also requiring a CAD model as input.

Animation Curved tetrahedral meshes have also gained popularity in the context of fast animation. With fewer degrees of freedom and preserved geometric fidelity, [?] observe the benefit of quadratic tetrahedra in the pipeline of physically based animation. [Suwelack et al. 2013] further investigate the transfer problem when using curved meshes as a proxy.

4.2.2 Curved Structured Mesh Generation

The use of a hexahedral mesh as a discretization for a volume, allows to naturally define C^k splines over the domain, which can be used as basis functions for finite element methods: this idea has been pioneered by IsoGeometric Analyisis (IGA), and it is an active research area. The generation of volumetric, high-order parametrizations that conform to a given input geometry is an extremely challenging problem [??]. Most of the existing methods rely on linear hexahedral mesh generation, which is on its own a really hard problem for which automatic and robust solutions to generate coarse meshes are still elusive [??????] due to the inherently global nature of the problem. The current state of the art for IGA meshing is a combination of manual decomposition of the volume and semi-automated geometrical fitting [??].

In contrast, our approach is automatic, i.e. can automatically process thousands of models without any manual intervention, while providing explicit guarantees on both the validity of the elements and the maximal geometric error. Its downside is the C^0 continuity of the basis on the elements' interfaces. However, it is unclear to us if this is a real limitation: in many common settings in computer graphics and mechanical engineering (Poisson problems, linear and non-linear elasticity) the higher smoothness offered by spline functions does not make noticeable difference experimentally [?] (and it actually leads to much worse conditioning of the system

[1] https://doc.nektar.info/userguide/5.0.0/user-guidese17.html

matrix, which is problematic for iterative solvers), and we thus believe that curved tetrahedral meshing is a very exciting alternative as it dramatically simplifies both the meshing and fitting of high-order elements.

4.2.3 Boundary Preserving Tetrahedral Meshing

We refer to [?] for a detailed overview of linear tetrahedral meshing, and we focus here only on the techniques that target boundary preserving tetrahedral meshing.

The most popular linear tetrahedral meshing methods are based on *Delaunay refinement* [???], i.e. the insertion of new vertices at the center of the circumscribed sphere of the worst tetrahedron in term of radius-to-edge ratio. This approach is used in the most popular tetrahedral meshing implementations [??], and, in our experiments, proved to be consistently successful as long as the boundary is allowed to be refined. A downside of these approaches is that a 3D Delaunay mesh, unlike the 2D case, might still contain "sliver" tetrahedra, thus requiring mesh improvement heuristics [????]. [?] discusses this issue in detail and provides a different formulation to avoid it without the use of a postprocessing. [?] introduces an approach that does not allow insertion of Steiner points, making it not suitable for generic polyhedra domains.

There are many variants of Delaunay-based meshing algorithms including *Conforming Delaunay tetrahedralization* [??], *constrained Delaunay tetrahedralization* [????], and *Restricted Delaunay tetrahedralization* [???].

To the best of our knowledge, all these methods are designed to allow some modifications of the input surface (either refinement, resampling, or approximation). One exception is the constrained Delaunay implementation in TetGen [?] that allows disabling any modification to the boundary. However, this comes at the cost of much lower quality and potential robustness issues, as we show in Appendix A.9.

A different tetrahedral meshing approach has been proposed in [?], and its variants [??], where the problem is relaxed to generate a mesh that is close to the input to increase robustness. However, these approaches are not directly usable in our setting, as we require boundary preservation.

Due to these issues, we propose a novel boundary-preserving tetrahedral meshing algorithm specifically tailored for the shell mesh generated by our curved meshing algorithm.

4.2.4 Curved Surface Fitting

There are many algorithms for fitting curved *surfaces* to dense 3D triangle meshes. The most popular approaches fit spline patches, usually on top of a quadrangular grid. Since generating quadrilateral meshes is a challenging problem for which robust solutions do not exist yet, we refer to [?] for an overview, and only review in this section algorithms for unstructured curved mesh generation, which are more similar to our algorithm. We note that the focus of our paper is volumetric meshing: while we generate an intermediate curved surface mesh, this is not the goal of our algorithm, especially since the generated surface is only C^0 on edges.

[?] fits a smooth surface represented by a point cloud to a curved triangle mesh based on a subdivision surface scheme and an interleaving mesh simplification and fitting pipeline that

preserves sharp features. The algorithm does not provide explicit correspondence to the input: they are defined using distance closest point, which is not bijective far from the surface.

[?] converts dense irregular polygon meshes of arbitrary topology into coarse tensor product B-spline surface patches with accompanying displacement maps. Based on the work [?] that fits triangle surface meshes with Bézier patches, [?] fits triangle surface meshes with high-order B-spline quadrilateral patches and adaptively subdivide the patches to reduce the fitting error. These methods produce smooth surfaces but do not have feature preservation.

Another related topic is the definition of smooth parametric surfaces interpolating triangle meshes. We refer to [?] for an overview of subdivision methods and discuss here the approaches closer to our contribution.

[Hahmann and Bonneau 2003] proposed to use triangular Bézier patches to define smooth surfaces over arbitrary triangle meshes ensuring tangent plane continuity by relaxing the constraint of the first derivatives at the input vertices. Following Hahmann's work, [?] presents a more complete pipeline: perform QEM simplifications, trace the coarse mesh onto the dense one and perform parameterization relaxing and smoothing. Then it fits a hierarchical triangular spline [?] to the surface. More recent work [Tong and Kim 2009] approximates the triangulation of an implicit surface with a G^1 surface. These schemes are usually designed to interpolate existing meshes rather than simplifying a dense linear mesh into a coarse curved mesh and are thus orthogonal to our contribution.

4.3 Shell Preliminaries

We briefly overview [?] as our work uses and extends it. [?] introduces bijective projection shells (which we will abbreviate as shells in the rest of the paper), a new geometry processing tool to perform mesh editing while preserving a close-by bijective map to the input surface.

Shell. The shell is defined by three trianglulated surface meshes $\overline{\mathcal{S}} = \{(B_s, V_s, T_s), F_s\}$ sharing the same mesh connectivity F_s, where B_s, V_s, and T_s are the vertices of the bottom, middle, and top surface respectively. Each triangle in F_s corresponds to a *generalized prism*, defined by connecting the corresponding triangles in the three surfaces with straight edges, called *pillars*. Each prism P has three vertices $v_i \in V_s, t_i \in T_s, b_i \in B_s, i = 1, 2, 3$ on the middle, top, and bottom surface, respectively. Each pillar decomposes into a top $h_i^T = t_i - v_i, i = 1, 2, 3$ and bottom $h_i^B = b_i - v_i, i = 1, 2, 3$ slab, and each slab can be canonically decomposed into 3 tetrahedra, and each tetrahedron contains a constant vector field aligned with the pillars it is connected with [?, Figure 4]. The vector field is used to define a *projection operator* Π within the shell.

Projection Operator. For every prism P, the projection operator Π_P is defined as the tracing of the piece-wise constant vector field V inside the decomposed prism, by assigning to each tetrahedron $T_j^P, j = 1, \ldots, 6$, the constant vector field defined by the only edge of T_j^P which is one of the oriented pillars $h_i^T, h_i^B, i = 1, 2, 3$ ([?, Figure 4]). That is, $\forall p \in T_j^P$

$$\Pi_P(p) = h_i^k,$$

where i is the index of the vertex corresponding to the pillar edge of T_j^P, and k is either the top or bottom surface. The shell projection operator Π is defined as the operator whose restriction to P, $\Pi|_P$ is Π_P, and it defines a bijective map between every pair of specific triangle meshes contained in the shell, called *sections*.

Section. A triangle mesh is a section if it is contained within the shell, and if the dot product of the normal of each of its triangles with the vector field in each of the overlapping tetrahedra is positive. The projection operator Π defines a bijective map between any pair of sections if the shell is *valid*.

Shell Validity. [?] defines a shell $\overline{\mathcal{S}}$ to be *valid* with respect to an input mesh $\mathcal{M}$ if it satisfies two conditions:

1. The volumes of all possible tetrahedral decomposition of a prism (24 of them) are positive.
2. $\mathcal{M}$ is a section for all possible tetrahedral decompositions. That is, the input mesh is contained within the shell, and the dot product between the mesh's normals and the shell's pillar is positive.

Singularity. Singular vertices are a special geometric configuration, where the neighboring triangles of a specific vertice admits a conflicting set of normals. [?] extends the shell construction to allow such cases. Around the (isolated) singular vertices, the prisms are pinched to become generalized pyramids, composed of two tetrahedra instead of three.

The algorithm to build the shell creates an initial valid extrusion, potentially thin and dense, and then iteratively uses the shell local operations (i.e., vertex smoothing, edge collapse, edge split, and edge flip) [?, Section 3.4] to improve its quality while preserving the validity.

4.3.1 Variation from the Original Algorithm

To extend the shell formulation in [?] to accommodate for feature preservation (Section 4.5), we modify the definition of a valid section [?, Definition 3.1], by relaxing the intersection between a triangle and a prism in the discrete case. That is, we do not consider the prism to be intersecting a triangle if they share only one vertex of the triangle; we also ignore the intersection if they intersect on a feature edge on opposite sides. The bijectivity and validity condition of the shell projection trivially holds.

4.4 Curved Tetrahedral Mesh Generation

Input. The input of our algorithm is a collection of oriented manifold, watertight, self-intersection-free triangle mesh $\mathcal{M} = (V, F)$, and a set of points $p_i \in \mathcal{P}$ (possibly empty) on the surface of $\mathcal{M}$ (Figure 4.2, left) where the distance bound ε is prescribed. The collection $\mathcal{M}$ must be consistently oriented such that it is the boundary of an oriented 3-manifold. A set of edges can also be optionally provided as annotated features (Section 4.5).

Figure 4.2: Input triangle mesh $\mathcal{M}$ and points $\mathcal{P}$. Output curved tetrahedral mesh $\mathcal{T}^k$ and bijective map ϕ^k.

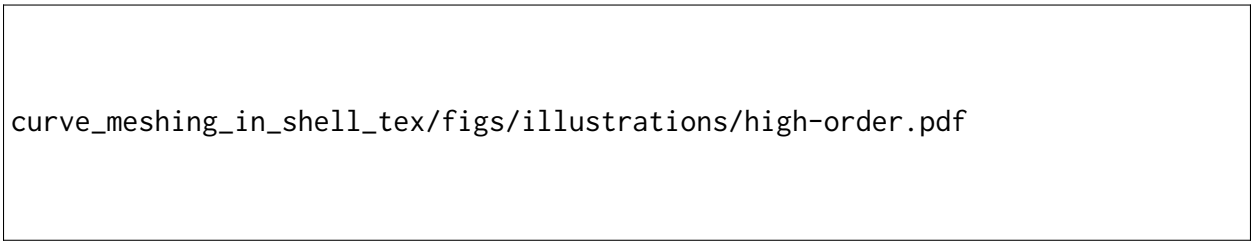


Figure 4.3: Lagrange nodes on the reference element $\hat{\tau}$ for different $k = 1, 2, 3$ and example of geometric mapping g.

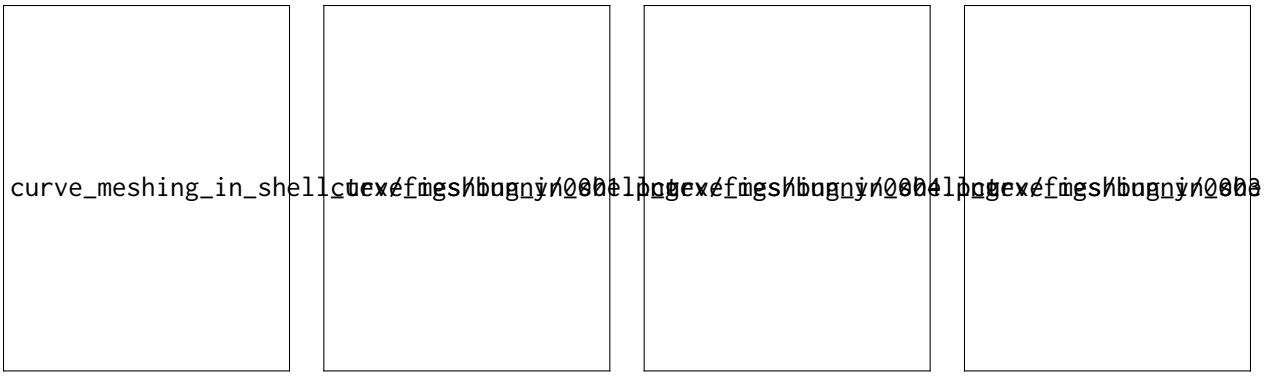
Output. The output of our algorithm is a tetrahedral mesh $\mathcal{T}^k = (V^k, T^k)$ of order k. Formally, each tetrahedron $\tau \in T^k$ is defined through the *geometric map* from the reference tetrahedron $\hat{\tau}$,

$$g^\tau = \sum_{j=1}^{n} c_j^\tau L_j(\hat{u}, \hat{v}, \hat{w}), \tag{4.1}$$

where $\hat{u}, \hat{v}, \hat{w}$ are the local coordinates of a point in $\hat{\tau}$, c_j^τ are the control points for a tetrahedron τ, and L_j are polynomial bases (typically Lagrange bases). For two tetrahedra τ_1 and τ_2 of T^k sharing a face F, the restriction of the maps g^{τ_i}, $i = 1, 2$, to F coincide. Figure 4.3 shows the position of the control points c_j on the reference element for $k = 1, 2, 3$ for the Lagrange bases. We call the tetrahedralization of a curved mesh $\mathcal{T}^k$ *positive* if $\det(J_{g^\tau}) > 0$ everywhere on every τ. In particular, for $k = 1$, since g is affine, J_{g^τ} is constant and the positivity reduces to the positive orientation of the vertices [**?**].

Note that, while bijectivity of the geometric map g^τ implies positivity, the reverse is not true. Therefore, our algorithm not only checks for $\det(J_{g^\tau}) > 0$, but also ensures that the boundary $\partial\mathcal{T}^k$ does not intersect; we show in Appendix A.7 that these two conditions guarantee the bijectivity of g^τ. Furthermore, our algorithm ensures that the distance from any point in $\mathcal{P}$ to $\partial\mathcal{T}^k$ (the surface of $\mathcal{T}^k$) is smaller than an user-controlled parameter ε.

We are not assuming anything on $\mathcal{P}$: a sparse set of points will generate a mesh less faithful to the input geometry, while a dense sampling computed for instance with Poisson disk sampling [**?**]



0 points. 200 points. 500 points. 1000 points.

Figure 4.4: Effect of the choice of the set $\mathcal{P}$ on the output.

will prevent the surface from deviating too much (Figure 4.4).

Our algorithm guarantees that the tetrahedralization is positive and that $\partial\mathcal{T}^k$ does not have self-intersections. It also aims at coarsening $\mathcal{T}^k$ as much as possible while striving to obtain a good geometric quality. To reliable fit $\partial\mathcal{T}^k$ to $\mathcal{M}$ we require a bijective map

$$\phi^k : \mathcal{M} \to \partial\mathcal{T}^k$$

from the input $\mathcal{M}$ to the surface of $\mathcal{T}^k$ (Figure 4.2 right). Our algorithm also generates this map and exposes it as an output for additional uses, such as attribute transfer. Note that, since we build upon the shell construction in [**?**], we also guarantee $\partial\mathcal{T}^k$ is homeomorphic and topology-preserving with respect to $\mathcal{M}$ (Figure 4.5).

To simplify the explanation, we use the bar $\overline{\ }$ to represent quantities on the *straight* linear shell, the tilde $\widetilde{\ }$ for the *curved* shell, and hat $\hat{\ }$ for the reference elements (e.g., $\overline{P}$ is the prism on the straight coarse shell, $\widetilde{P}$ is the curved prism, and $\hat{P}$ is the prism on the reference configuration).

Definition 4.1. We call a curved mesh $\mathcal{T}^k$ and its mapping ϕ^k to $\mathcal{M}$ *valid* if it satisfies the following conditions:

1. ϕ^k is bijective;
2. the distance between any $p \in \mathcal{P}$ and $\partial\mathcal{T}^k$ is less than ε;
3. $\mathcal{T}^k$ is positive (i.e., every geometric map g^τ has positive Jacobian's determinant).

Overview. Our algorithm starts by creating a valid mesh (i.e., it satisfies 4.1), then it performs local operations (Appendix A.8) to improve $\mathcal{T}^k$ (i.e., coarsen it and improve its quality) while ensuring all the conditions remain valid with respect to local modification. To achieve this goal, our algorithm uses two stages: (1) curved shell generation and (2) tetrahedral mesh generation and optimization.

Figure 4.5: Our algorithm maintains free of intersection even on challenging models, without the need of setting adaptive threshold.

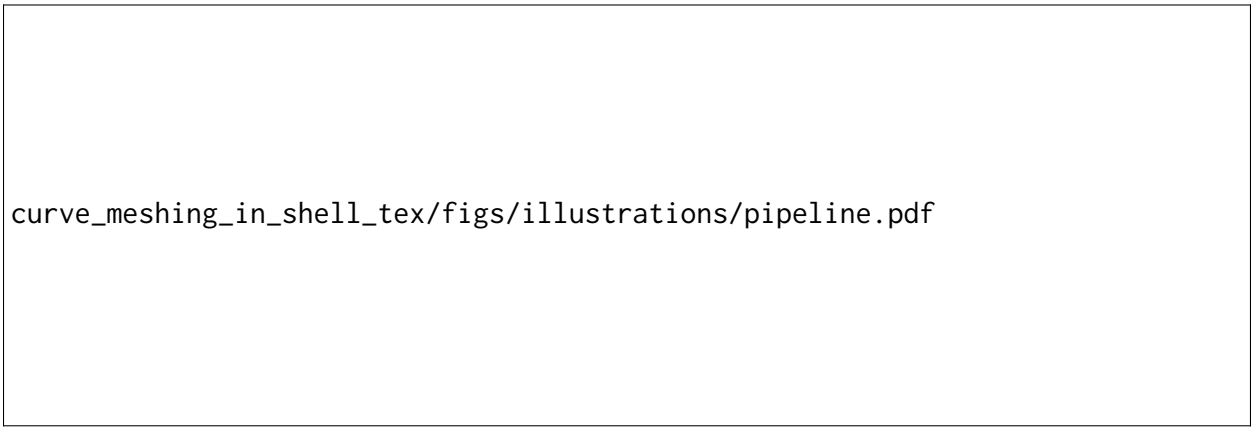


Figure 4.6: Overview of curved mesh generation pipeline.

Stage 1: Curved Shell Construction. In the first stage (Section 4.4.1) we extend the shell construction of [?] by combining the shell projection Π with a high-order mapping

$$\psi^k : \widetilde{\mathcal{S}} \to \overline{\mathcal{S}}.$$

We start form the *valid* shell $\overline{\mathcal{S}}$ constructed from the input mesh $\mathcal{M}$ (i.e., $\mathcal{M}$ is a section $\overline{\mathcal{S}}$). We call $\overline{\mathcal{S}}$ a *projection shell* and call the prismatic projection Π. Together with the construction of $\overline{\mathcal{S}}$, we build an order k *curved prismatic shell* $\widetilde{\mathcal{S}}$ that defines a curved layer around $\mathcal{M}$ and a *bijective* map ψ^k between $\widetilde{\mathcal{S}}$ and $\overline{\mathcal{S}}$ (Figure 4.6, first three figures) that ensures that the distance between $\mathcal{M}$ and $\partial\mathcal{T}^k$ is smaller than ε (Section 4.4.2). That is, $\phi^k(p) < \varepsilon$ for any $p \in \mathcal{P}$. (Note that we do not require $\mathcal{M}$ to be a section of $\widetilde{\mathcal{S}}$.)

To facilitate the volumetric meshing in the next stage (Section 4.4.3), we restrict the top and bottom surface of $\widetilde{\mathcal{S}}$ to be linear (independently from the order of ψ^k). The final output of this first stage is a high-order volumetric shell, a bijective mapping $\phi^k = \Pi \circ \psi^k$, and a *positive* tetrahedralization of $\widetilde{\mathcal{S}}$ with flat boundary. In other words, the tetrahedralization of $\widetilde{\mathcal{S}}$ satisfies 4.1.

Stage 2: Tetrahedral Mesh Generation. In the second stage (Section 4.4.3) we use boundary-conforming tetrahedralization to connect the top and bottom surface of $\widetilde{\mathcal{S}}$ with a background tetrahedral mesh, thus generating a *positive* order k tetrahedralization $\mathcal{T}^k$ of a bounding box around the input, which we can further optimize with local operations to improve its quality (Figure 4.6, last two figures).

To ensure that our first condition is satisfied, we define the mapping ϕ^k as a composition of several mappings, which we ensure are bijective. For the second condition, we initialize our construction with ϕ^k as the identity, and thus, the distance at the sample points is zero. After every operation, we recompute the distance and "undo" the operation if the distance becomes larger than ε. To ensure that the last condition holds, we rely on checking if all prisms (linear and curved) have positive geometric mapping, which ensures that they can be tetrahedralized with a positive tetrahedralization. Ensuring the condition while coarsening $\mathcal{M}$ allows us to generate a coarse *curved* tetrahedral mesh $\mathcal{T}^k$ and the *bijective* map ϕ^k to the input mesh $\mathcal{M}$.

4.4.1 High-order Shells

To simplify the explanation, we first focus on the case where $\varepsilon = \infty$, that is, we aim at generating an as-coarse-as-possible curved mesh. Note that, the trivial solution (i.e., a single tetrahedron) is not necessary a valid $\mathcal{T}^k$ since it would be impossible to build the bijective mapping ϕ^k.

The output of [?] is a coarse shell $\overline{\mathcal{S}}$ with a piecewise linear middle surface. To curve it, we construct a shell $\widetilde{\mathcal{S}}$ and the bijective map ψ^k while constructing $\overline{\mathcal{S}}$. The shell $\widetilde{\mathcal{S}}$ is constructed warping every prism $\widetilde{P}$ of $\widetilde{\mathcal{S}}$ with ψ^k. Since we define ϕ^k as $\Pi \circ \psi^k$, and Π satisfies the first two conditions 4.1, we only need to ensure that ψ^k is bijective, for ϕ^k to be bijective. We define the mapping $\psi^k = \overline{\omega} \circ (\widetilde{\omega}^k)^{-1}$ through two cross-parametrization maps from the reference prism $\hat{P}$:

$$\overline{\omega} : \hat{P} \to \overline{P}, \qquad \widetilde{\omega}^k : \hat{P} \to \widetilde{P}.$$

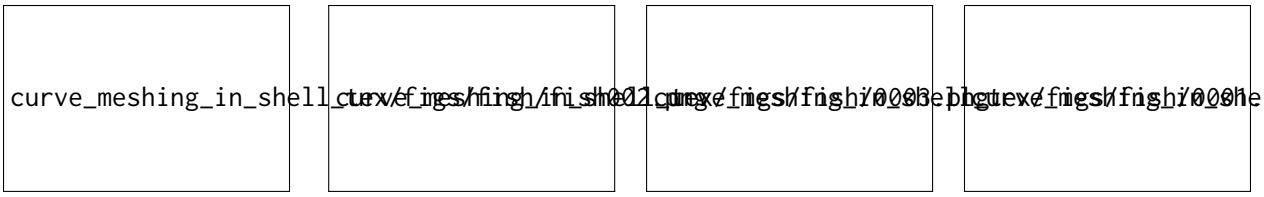


Figure 4.7: A model simplified with different distances.

Both mappings $\overline{\omega}$ and $\widetilde{\omega}^k$ are defined as the tensor product between the base triangular mapping (high-order for $\widetilde{\omega}^k$) and pillar's barycentric heights. That is, for a prism $\widetilde{P}$

$$\widetilde{\omega}^k(\hat{u}, \hat{v}, \hat{h}) = \sum_{j=1}^{n} c_j^P L_j(\hat{u}, \hat{v}) \left((1 - \hat{u} - \hat{v}) h_1^{\widetilde{P}} + \hat{u} h_2^{\widetilde{P}} + \hat{v} h_3^{\widetilde{P}}\right) \hat{h},$$

where $\hat{u}, \hat{v}, \hat{h}$ are the barycentric coordinates in the reference prism, c_j the control points of a triangle, $h_i^{\widetilde{P}}$ the three pillars heights, and L_j a triangle polynomial basis. By ensuring that ψ^k is bijective, we guarantee that any curved tetrahedralization of a prism $\widetilde{P}$ will be a valid tetrahedralization of $\widetilde{\mathcal{S}}$.

We note that to decouple the following tetrahedral mesh generation and the curved shell generation, we ensure that $\widetilde{\omega}^k$ maps the top and bottom face of the curved prism $\widetilde{P}$ to a linear triangle.

After each local operation, we generate samples $\hat{s}_i, i = 1, \ldots, m$ on the parametric base of the prism $\hat{P}$ and use $\Pi^{-1} \circ \overline{\omega}$ to map $\hat{s}_i$ back to $\mathcal{M}$ and $\widetilde{\omega}^k$ to map them to $\partial\mathcal{T}$. Using the mapped point we solve

$$\min_{c_i} \sum_{i=1}^{m} \|(\Pi^{-1} \circ \overline{\omega})(\hat{s}_i) - \widetilde{\omega}[c_i]^k(\hat{s}_i)\|_2^2,$$

where c_i are the control points of $\widetilde{\omega}^k$. As $\widetilde{\omega}[c_i]^k(\hat{s}_i)$ is a linear function of c_i. This is a quadratic optimization problem. The control points of the top and bottom surface are fixed to ensure that $\widetilde{\omega}^k$ maintains the two surfaces as linear. We validate the bijectivity of $\widetilde{\omega}^k$ by checking positivity of the determinant [?] and that the top and bottom surfaces are intersection free. The intersection is simplified in our case since fast and exact algorithms[?] are available since the top and bottom surfaces stay linear.

4.4.2 Distance Bound

In the previous section, we explained how to generate a curved shell $\widetilde{\mathcal{S}}$ that satisfies 4.1. To ensure that the middle surface of $\widetilde{\mathcal{S}}$ has a controlled distance from the points in $\mathcal{P}$, we interleave a distance check in the construction of $\widetilde{\omega}^k$ after each local operation. Formally, after every local operation we use the mapping ϕ^k to map every point $p_i \in \mathcal{P}$ to $\widetilde{p}_i = \phi^k(p_i)$ a point on the coarse curved middle surface of $\widetilde{\mathcal{S}}$ and, if $\|p_i - \widetilde{p}_i\| \geq \varepsilon$ we reject the operation. The initial shell is trivially

Figure 4.8: Two dimensional overview of the five steps of our boundary preserving tetrahedral meshing algorithm.

a valid initialization as ϕ^k is identity and thus, the distance is zero. Note that, $\widetilde{p}_i$ is not necessarily the closest point to p_i on $\partial\mathcal{T}$, thus $\|p_i - \widetilde{p}_i\|$ is an upper bound on the actual pointwise distance. Figure 4.7 shows the effect of the distance bound on the surface; a small distance will lead to a denser mesh with more details, while a large one will allow for more coarsening.

4.4.3 Tetrahedral Meshing

The outcome of the previous stage is a curved tetrahedralization of $\widetilde{\mathcal{S}}$ that closely approximates $\mathcal{M}$ with linear ("flat") boundaries. We now consider the problem of filling its interior (and optionally its exterior) with a tetrahedral mesh, a problem known as *conforming boundary preserving* tetrahedralization.

Several solution exists for this problem (Section 4.2.3) and the most common implementation is TetGen [**?**]. Most algorithms refine the boundary, which allows deriving bounds on the quality of the tetrahedral mesh. However, in our setting, this is problematic, as any change will have to be propagated to the curved shell. To avoid coupling the volumetric meshing problem with the curved shell coarsening, while technically possible it is very challenging to implement robustly, we opt for using a tetrahedral meshing algorithm that preserves the boundary exactly. Not many algorithms support this additional constraint, the only one with a public implementation is the widely used TetGen algorithm. However, we discovered that, when this option is used, it suffers from robustness issues, which we detail in Appendix A.9. To solve this problem in our specific setting, we propose in the following five step algorithm (Figure 4.8) taking advantage of the availability of a shell, based on the TetWild [**?**] algorithm.

Step 1. To generate a boundary preserving linear mesh, we first exploit the shell to extrude the bottom surface B (and top T) further by a positive (potentially small) constant δ such that the newly extruded bottom surface B_e (and top T_e) does not self-intersect. The space between B and B_e (and between T and T_e) consists of prisms divided into positive tetrahedra.

Step 2. Then we insert B_e and T_e in a background mesh $\mathcal{B}$ generated following TetWild algorithm ([**?**, Section 3.1]), that is, we use the triangle of B_e and T_e as the input triangle meshes for the first

stage of the TetWild algorithm, which inserts them into a background mesh $\mathcal{B}$, so that each input triangle is a union of faces of refinement of $\mathcal{B}$.

We interrupt the algorithm after the binary space partitioning (BSP) subdivision (and before the TetWild mesh optimization [**?**, Section 3.2]) to obtain a positive tetrahedral mesh in rational coordinates with a surface with the same geometry of B_e and T_e, but possibly different connectivity as TetWild might refine it during the BSP stage.

Step 3. Our original goal was to compute a mesh conforming to B and T, but we could not do it directly with TetWild as they might be refined. We now replace the mesh generated by TetWild between B_e and T_e with another one conforming to B and T. To achieve this, we delete all tetrahedra between B_e and T_e, and insert the surfaces B and T, which will "float" in the empty space between B_e and T_e. We now want to fill the space between B and B_e with positive tetrahedra conforming to the surfaces B and T.

Step 4. Every prism P, made by a bottom triangle B^T and a bottom extruded triangle B_e^T and its corresponding bottom extruded refined triangle $B_e'^T \in \mathcal{B}$, can be tetrahedralized without refining B: That is, we first decompose the prism B^T, B_e^T in tetrahedra (always possible by construction), then refine every tetrahedron touching $B_e'^T$. By repeating the same operation on the space between T and T_e we will have a positive linear boundary conforming tetrahedral mesh of B and T.

Step 5. The tetrahedra generated in the previous step will have rational coordinates and will also likely have low quality. To round the coordinates to floating-point representation and to improve their quality, we use the mesh optimization stage of TetWild, with the minor variant of keeping the vertices and edges on B and T frozen. Note that the vertices in B and T are already roundable to floating-point representation, as they were part of the input.

Curved Tetrahedral Mesh Optimization After generating the conforming linear tetrahedral mesh. we stitch it with the tetrahedralized $\widetilde{\mathcal{S}}$ to obtain a valid output mesh $\mathcal{T}^k$ (Definition 4.1). However, its quality might be low, in particular in the curved region, as $\widetilde{\mathcal{S}}$ can be thin with large triangles. To improve the quality of $\mathcal{T}^k$ we adapt the local operation of a linear pipeline to our curved settings. We propose three local operations: smoothing, collapse, and flip. Since the surface of $\mathcal{T}^k$ is already coarse and of high quality, as part of the definition of ϕ^k, we prevent any local operation from changing it. We validate every local operation (i.e., check the positivity of $\mathcal{T}^k$) using the convex-hull property [**?**] and reject the operation if it is violated. Our local operations are prototypical, and we leave as future work a more comprehensive study of curved mesh optimization.

Smoothing. As for the linear case, we compute the total energy of a vertex v by summing up the energies of the tetrahedra adjacent to it, which we compute on 56 regularly sampled points. We then perform gradient descent for all high-order nodes in the one-ring neighborhood of v. That is, we collect all edge nodes, face nodes, and cell nodes of the one-ring neighborhood of

Figure 4.9: Input triangle mesh with features and output curved mesh with feature preserved equipped with bijective map ϕ^k.

v. Differently from the linear case, the optimization is expensive since the nodes neighborhood typically contains hundreds of nodes.

Collapse and Swap The collapse and swap are the same as in a linear mesh, and we place the high-order nodes of the newly created face on the linear flat face.

4.5 Feature Preserving Curved Shell

Input We enhance the input to additionally include a set of feature edges $f_i \in \mathcal{F}$ and feature vertices $v_i \in \mathcal{V}$ such that no triangle in F has more than one feature edge (Figure 4.9 left). (This property can be satisfied on any generic mesh by performing 1-to-3 refinement on every triangle with more than one feature edge).

Output Since the input has features, the output curved mesh $\mathcal{T}^k$ will also have curved feature edges $f_i^k \in \mathcal{T}^k$, feature vertices $v_i^k \in \mathcal{V}^k$, and the bijective map ϕ^k preserves features by bijectively mapping $\mathcal{F}$ to $\mathcal{F}^k$ and $\mathcal{V}$ to $\mathcal{V}^k$. Our method makes no assumption on the topology and "quality" of the features. If the features are reasonable, it will produce a high-quality mesh, while if the features are close our algorithm will preserve them and result in smaller triangles on the surface. (Figure 4.10).

The previous construction generates valid curved tetrahedral meshes and the bijective map ϕ^k based on the construction of [?]. However, the shell construction cannot coarsen features: the authors suggest freezing them. For instance, when performing an edge collapse on the feature, the new coarse edge (orange) will not map to the feature (green) anymore (Figure 4.11). To ensure feature preservation we extend Definition 4.1.

Definition 4.2. We call a curved mesh $\mathcal{T}^K$ and its mapping ϕ^k from $\mathcal{M}$ *valid and feature preserving* if they are valid (Definition 4.1) and ϕ^k bijectively maps $\mathcal{F}$ to $\mathcal{F}^k$ and $\mathcal{V}$ to $\mathcal{V}^k$

As for the non-feature preserving case, we always aim to maintain a valid feature preserving $\mathcal{T}^k$.

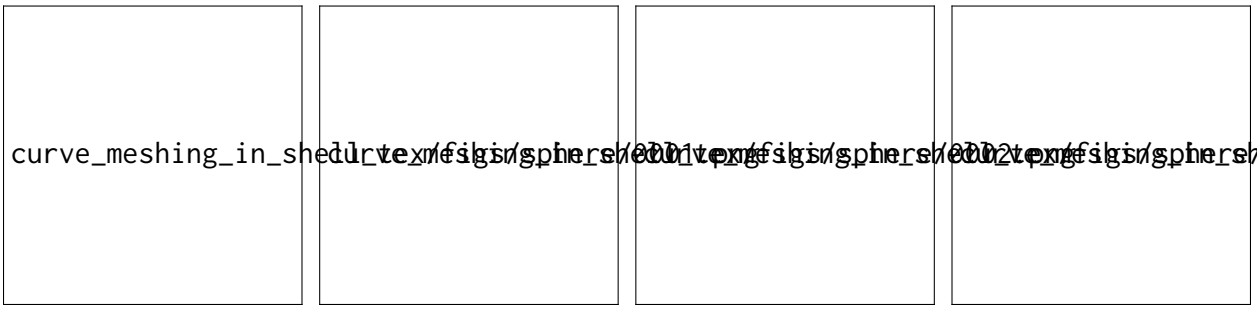


Figure 4.10: A sphere with different marked features (green). As we increase the number of features our algorithm will preserve them all but the quality of the surface suffers.

Figure 4.11: Input feature (green) is not preserved after traditional shell simplification.

Figure 4.12: Overview of the construction of the first stage of our pipeline.

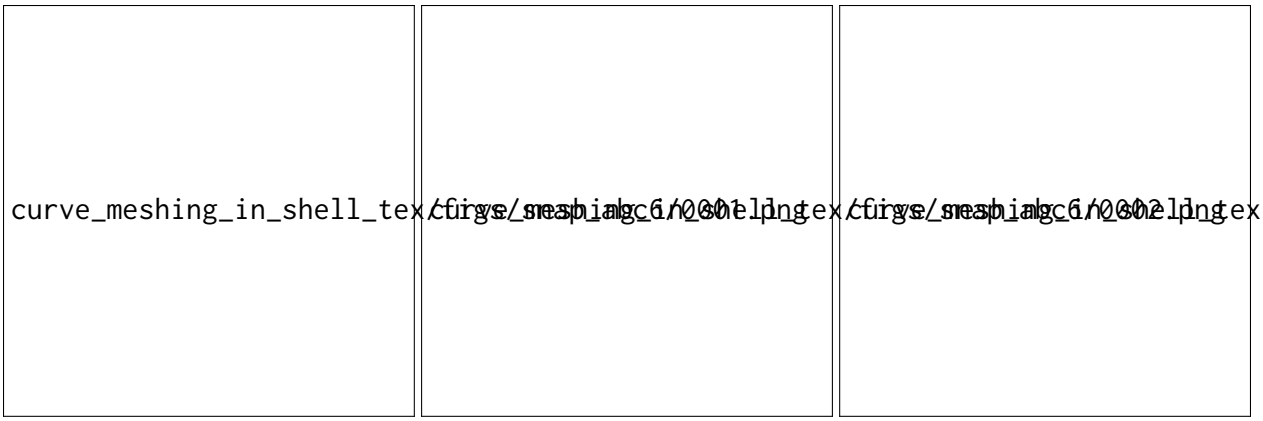


Figure 4.13: The input mesh has feature edges snapped, to create a valid shell, as well as the curved mesh.

Figure 4.14: The input edges feature (green) are grouped together in poly-lines and categorized in graph (left) and loops (right). For every graph we add the nodes (blue) to the set of feature vertices.

To account for features, we propose to change the prismatic map Π, that is, we only need to change the first stage. This is done by snapping the input features (Section 4.5.1). That is, we modify $\mathcal{M}$ to "straighten" the feature to ensure that the coarse prismatic projection preserves them and construct a mapping β between the straight mesh $\overline{\mathcal{M}}$ and $\mathcal{M}$ (Figure 4.13).

The outcome is a *valid* shell $\overline{\mathcal{S}}$ with respect to the straight surface $\overline{\mathcal{M}}$ (i.e., $\overline{\mathcal{M}}$ is a section $\overline{\mathcal{S}}$) that preserve features, the prismatic projection Π, and the bijective map β that can be directly used in the curved pipeline (Section 4.4). That is, the mapping ϕ^k will be defined as $\phi^k = \beta \circ \Pi \circ \psi^k$.

As for the non-preserving feature pipeline we ensure that our conditions are always met, starting from a trivial input and rejecting operations violating them. Our goal is to modify the input mesh $\mathcal{M}$ and create $\overline{\mathcal{M}}$ by moving its vertices. In such a way, the mapping β is simply barycentric. To guarantee bijectivity of ϕ^k we need to ensure that all mappings composing it are bijective, in particular β. To ensure that β is bijective it is enough that $\overline{\mathcal{M}}$ is self-intersection free (guaranteed by the shell construction) and that all its triangles have positive area. By straightening the features of $\mathcal{M}$ we ensure that the edges of the prism will map to the feature. Thus, ϕ^k will be feature preserving.

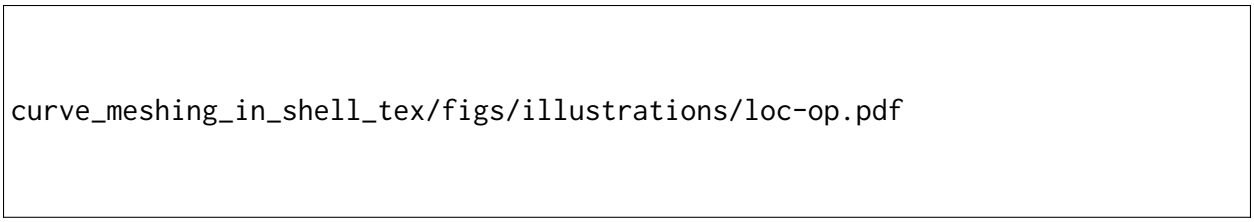


Figure 4.15: Illustration of smoothing on a feature.

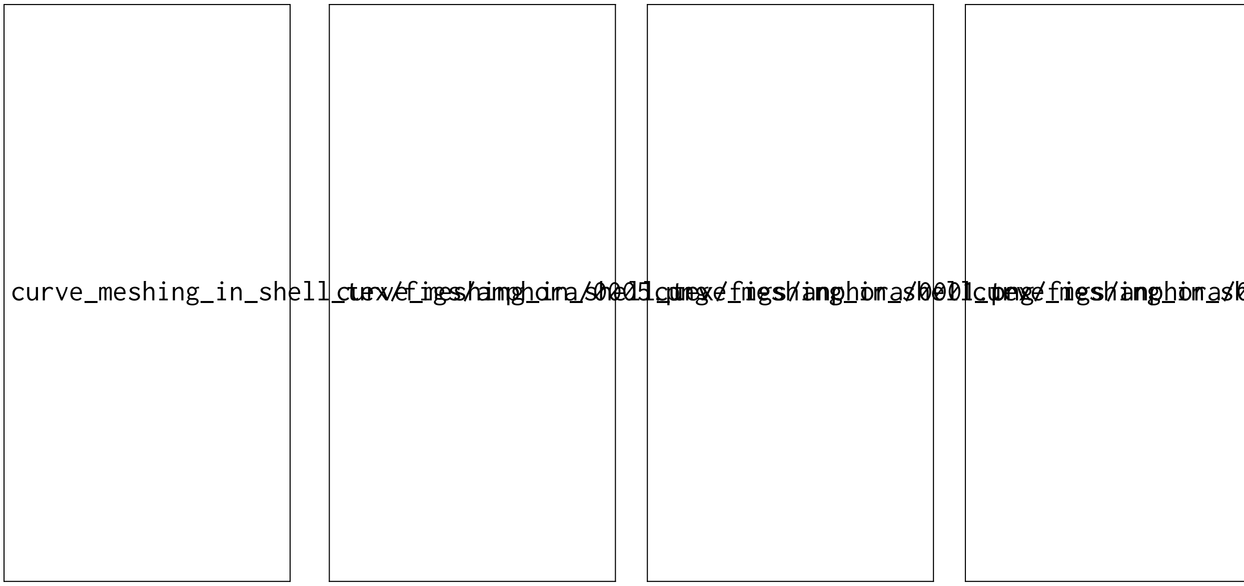
Feature Grouping. The first step of our pipeline consists of grouping successive edges $f_i \in \mathcal{F}$ into poly-lines and identifying two categories: loops and graphs (Figure 4.14). For every graph, we identify its nodes and add them as feature vertices. In other words, we add to $\mathcal{V}$ all the end-points and junction of poly-lines.

4.5.1 Feature straightening.

To allow feature coarsening, we propose to straighten $\mathcal{M}$ to ensure that all features are collinear. In other words, we build, together with the shell $\overline{\mathcal{S}}$, a mesh $\overline{\mathcal{M}} = (\overline{V}, F)$ (i.e., a mesh with the same connectivity F of $\mathcal{M}$) and features $\overline{\mathcal{F}}$ such that every triangle of $\overline{\mathcal{M}}$ has a positive area, $\overline{\mathcal{M}}$ is a section of $\overline{\mathcal{S}}$, and the features in $\overline{\mathcal{F}}$ are collinear. In such a way, the mapping β is trivially defined as piecewise affine ($\overline{\mathcal{M}}$ and $\mathcal{M}$ share the same connectivity) and is locally injective as long as all the triangles on $\overline{\mathcal{M}}$ have positive areas. Note that the bijectivity of β follows from the fact that the shell prevents self-intersections of $\overline{\mathcal{M}}$.

To construct a mesh $\overline{\mathcal{M}}$ with straight features, we start with $\overline{\mathcal{M}} = \mathcal{M}$ (in the beginning all prisms of $\overline{\mathcal{S}}$ cover at most one feature edge). Let $\overline{f}^1 = \{\overline{f}^1_i\}, i = 1, \dots, n$ and $\overline{f}^2 = \{\overline{f}^2_i\}, i = 1, \dots, k$ two chains of feature edge belonging to the same feature $\overline{f} \in \overline{\mathcal{F}}$. For every local operation acting on a feature $\overline{f}_1$ and $\overline{f}_2$ we first construct the new feature $\overline{f}^n = \{\overline{f}^n_i\}, i = 1, \dots, k$ such that the segments $(\overline{f}^n_i, \overline{f}^n_{i+1})$ are collinear and their length is proportional to (f_i, f_{i+1}) (the feature vertices in the input mesh $\mathcal{M}$), that this we use arc-length cross parameterization from f to $\overline{f}^n$ (Figure 4.15 show an example of smoothing a feature). Moving vertices of $\overline{f}^n$ will also move the vertices of $\overline{\mathcal{M}}$ thus, straighten the mesh as the local operations proceed. After the construction of $\overline{f}^n$ we check if the newly constructed $\overline{\mathcal{M}}$ is still a section of $\overline{\mathcal{S}}$ and if the triangles modified by the straightening have areas larger than ϵ. In practice, we choose $\epsilon = 10^{-10}$: a smaller value would lead to numerical instabilities and a larger one to less straightening.

Note that not all features can be straight; for instance, if a triangle has three feature vertices (the snapped feature will result in a degenerate triangle, thus, β will not be bijective) or if the snapping flips the normal ($\overline{\mathcal{M}}$ will no longer be a section of $\overline{\mathcal{S}}$). Both are extremely rare cases in our dataset.



Input $k = 3$ $k = 4$ $k = 5$

Figure 4.16: Curved meshes of different order. The additional degrees of freedom allows for more coarsening.

4.6 Results

Our algorithm is implemented in C++, using Eigen [?] for the linear algebra routines, CGAL [?] and Geogram [?] for predicates and geometric kernel, libigl [?] for basic geometry processing routines, and meshio [?] for converting across the different formats. We run our experiments on cluster nodes with Intel Xeon Platinum 8268 CPU 2.90GHz. The reference implementation and the data used to generate the results will be released as an open-source project.

To simplify the exposition, all meshes presented in this section are quartic meshes ($k = 4$). Our method is flexible and, for lower k, it will generate denser meshes (Figure 4.16).

4.6.1 Large Scale Validation.

We tested the robustness and quality of the result produced by our algorithm on three datasets: (1) Thingi10k dataset [?] containing 3574 models without features; (2) the first chunk of the ABC dataset [?] with 5328 models with features marked from the STEP file and (3) the CAD dataset [?] containing 106 models with semi-manual features.

Note that the original datasets contain more models since, for each of them, we selected meshes satisfying our assumptions: intersection-free (using the same strategy as in [?] with a distance tolerance of 10^{-6} and dihedral angle of 2°) oriented, manifold triangle meshes, smallest triangle area larger than 10^{-8}.

Our method has only the geometry accuracy parameter ε, which we set to 1% of the longest bounding box edge, and the point set $\mathcal{P}$ which we set as the input vertices V. With this basic

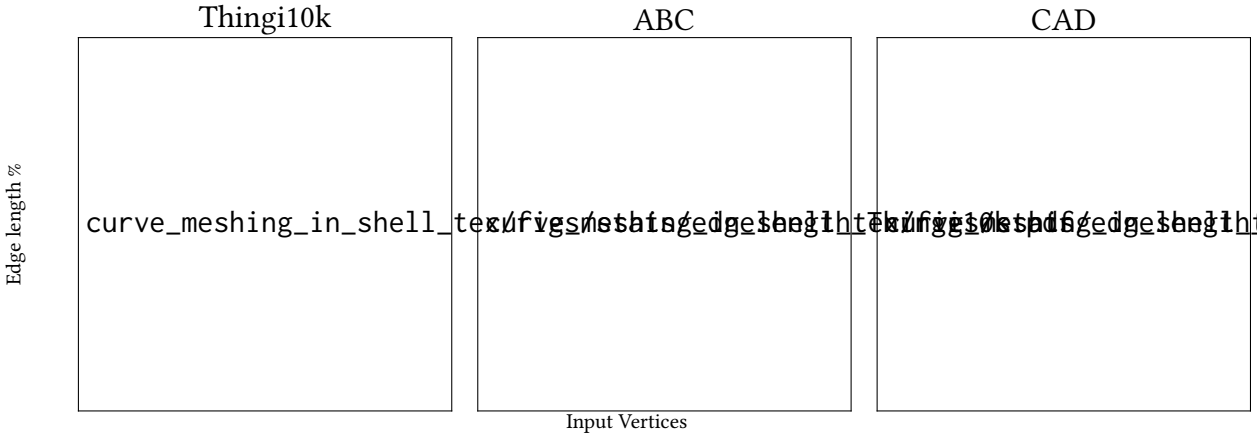


Figure 4.17: Relative average edge length (with respect to longest bounding box edge of each model) of our curved meshes versus number of input vertices.

Figure 4.18: Within the same distance bound (10^{-3} of the longest bounding box side), our method generates a coarser high order mesh, compared to the linear counterpart generated by fTetWild.

setup, our algorithm aims to produce the coarsest possible mesh while preserving features and striving to generate high-quality meshes. Our algorithm successfully generates curved meshes for 3527 for Thingi10k, 5268 for the ABC, and all for CAD dataset within 12 hours; by allowing more time, all models but 3 can be successfully processed. The 3 failures are due to models with a small one-tetrahedra component that "move" inside the shell as it grows. This is an implementation choice: we use collision detection instead of continuous collision detection for efficiency reasons.

Our method successfully generates coarse meshes whose average edge length is 10% of the model size while preserving features (Figure 4.17). Figure 4.18 shows how our method successfully captures the features and coarsen the surface with curved elements, while many linear elements (generated with fTetWild [**?**]) are required to closely approximate the surface.

The output of our algorithm can be directly used in the simulation (Section 4.6.4) since we guarantee that the geometric mapping g is positive. To ensure good conditioning and performance

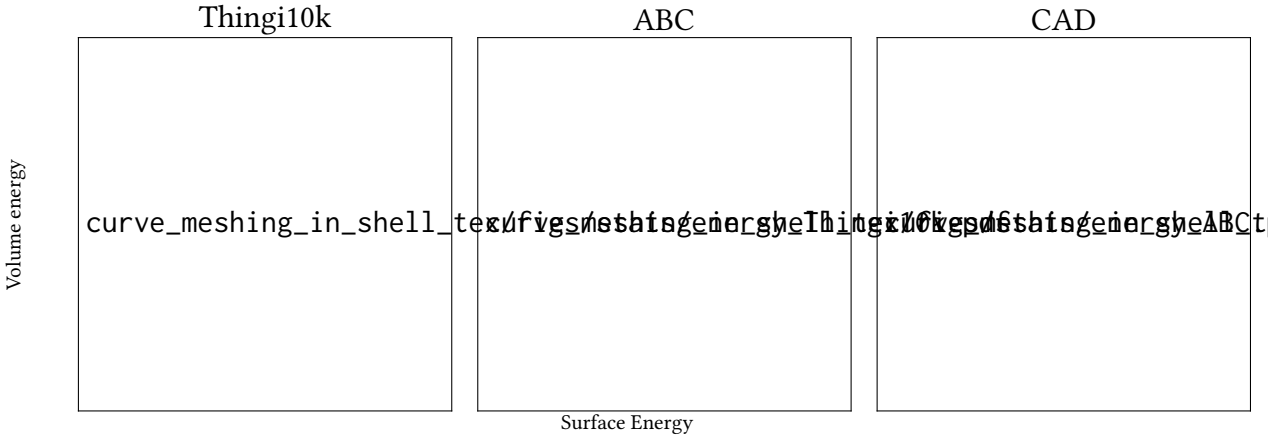


Figure 4.19: Surface and volume average MIPS energy of the output of our method (the CAD volume energy is truncated at 100, excluding 6 models).

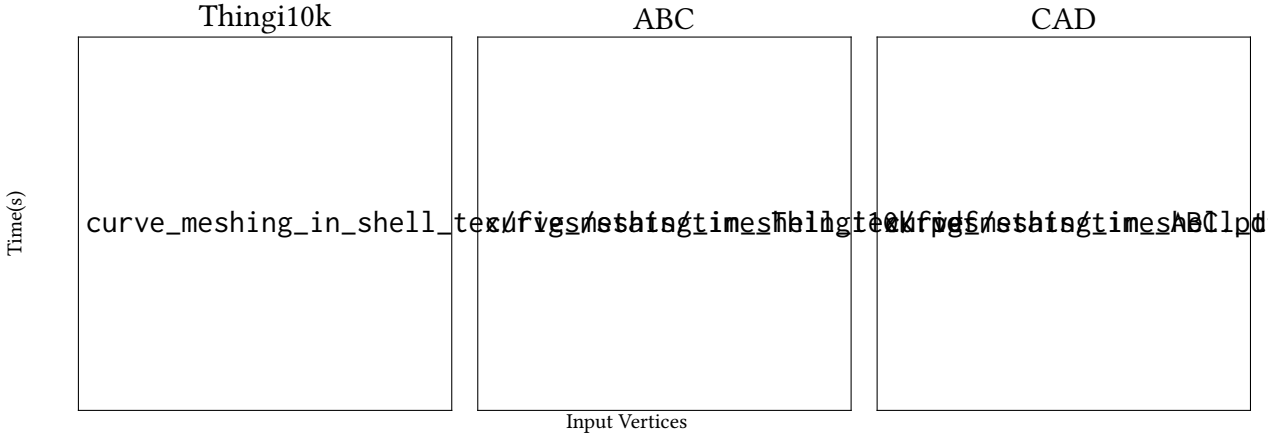


Figure 4.20: Timing of our algorithm versus the input number of vertices.

of the numerical solver, we measure the MIPS energy [**??**] of our output meshes (Figure 4.19).

Figure 4.20 shows the running time of our method with respect to the number of vertices. The running time of our algorithm is linear with respect to the number of input vertices; it takes around an hour for a model with around 10 thousand vertices.

4.6.2 Comparisons

Curved ODT [**?**] is, to the best of our knowledge, the only existing algorithm designed to convert dense triangle meshes into coarse, curved approximations. The input and output are the same as in our algorithm. However, their method does not provide a bijective map between the input and output, does not guarantee to preserve features, has no bound on the distance to the input surface, and does not guarantee that the elements are positive. While our algorithm has been designed to process large collections of data automatically, exposing only a few intuitive options to control the faithfulness to the input, the reference implementation of the curved ODT

Figure 4.21: Compared with Curved ODT, our method does not rely on setting vertex number and sizing field, and can generate coarse valid results.

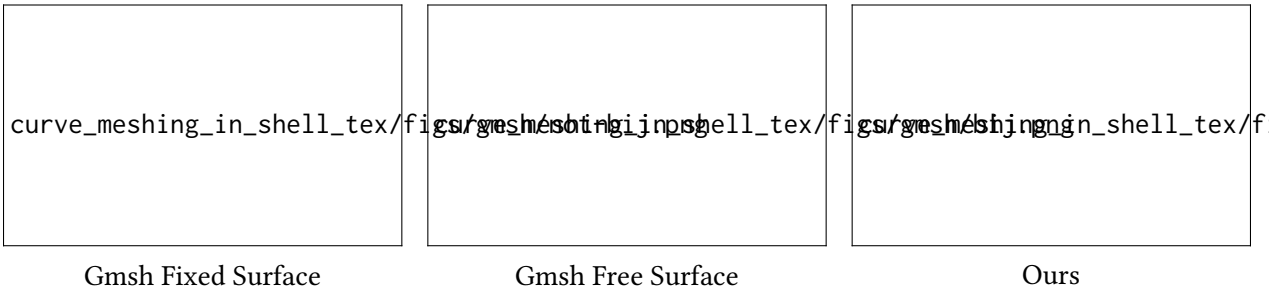


Figure 4.22: Example of a BRep meshed with Gmsh where the optimization fails to untangle elements when fixing the surface. By allowing the surface to be modified, the mesh becomes "wiggly". Our method successfully generate a positive curved mesh.

method provided to us by the authors requires the user to choose multiple *per-model* parameters to achieve good results, and the parameters have a strong effect on the quality and validity of the result (as shown in [**?**, Figure 16]). We thus restricted our comparison to only a small selection of models (see additional material) that the authors of [**?**] processed for us.

From our discussions with the authors, we observed that curved ODT generates a *valid* output when we provide (1) a sufficiently large number of vertices and (2) a good local feature size sizing field (LFS) [**?**] to efficiently spend the vertex budget in the regions with more geometrical details. Figure 4.21 shows an example of a model for which [**?**] fails to converge when using a uniform sizing field, while it succeeds when the sizing field is used.

In contrast, our algorithm can be run automatically on a large collection of geometrical models, it is guaranteed to have positive Jacobian (up to the use of floating-point predicates, Section 4.7), it preserves features, it automatically controls the density of the output depending on the desired user-provided distance threshold, and it provides a bijective map between the input mesh the boundary of the curved surface. For the model in Figure 4.21, our result contains 2037 elements, 18 times less than the curved ODT algorithm.

Coarse Gmsh Dense Gmsh Ours

Figure 4.23: Example of a STEP file meshed with Gmsh where, due to the low mesh density, the tetrahedralization is not positive. Gmsh manages to generate a positive mesh by using a denser initial tessellation. Since our method starts from a dense mesh and coarsen it, it can successfully resolve the geometry.

Gmsh [?] can only generate curved meshes from boundary representation (BRep), that is the input is not exactly the same as ours. To compare both algorithms, we start from the BRep and we generate a dense *linear* mesh that we use for our input. The Gmsh algorithm first constructs a curved mesh by fitting the high-order nodes to the BRep (possibly inverting elements) then performs mesh optimization to untangle them [?]; thus has no guarantee to generate positive meshes while preserving the surface (Figure 4.22, left). Additionally, Gmsh algorithm cannot control the distance from the input when the untangling allows the surface to move, and thus the surface is "wiggly" and denser than our result (Figure 4.22, center). We also observed that if the initial surface mesh is not dense enough, Gmsh closes holes and cannot generate a valid tetrahedral mesh (Figure 4.23).

4.6.3 Flexibility

Since the input to our method is a triangle mesh, our method naturally supports a variety of input that can be easily converted into triangle meshes. For instance, $\mathcal{M}$ can be generated from marching an implicit surface or Catmull-Clark subdivision of a hand-made quad mesh (Figure 4.24). The bijective map ϕ^k is used, for instance, to transfer the geodesic distance field or color information from the input triangle mesh to the coarse curved mesh.

4.6.4 Applications

Large Scale Poisson To show that our meshes are ready for simulation, we solve for Poisson equation

$$\Delta u = f, \quad u|_{\partial\Omega} = g,$$

where Ω is the domain (i.e., the mesh), f is the right-hand side, and g are the Dirichlet boundary conditions. To simplify the setup and the error measurements, we use fabricated solutions [?].

Figure 4.24: Our algorithm processes triangle meshes that can be extracted from different formats: an implicit microstructure geometry from [**?**] or a subdivision surface from [**?**]. The bijective map preserved on the surface allows taking advantage of the plethora of surface algorithms including polyhedral geodesic computation [**?**] and texture mapping.

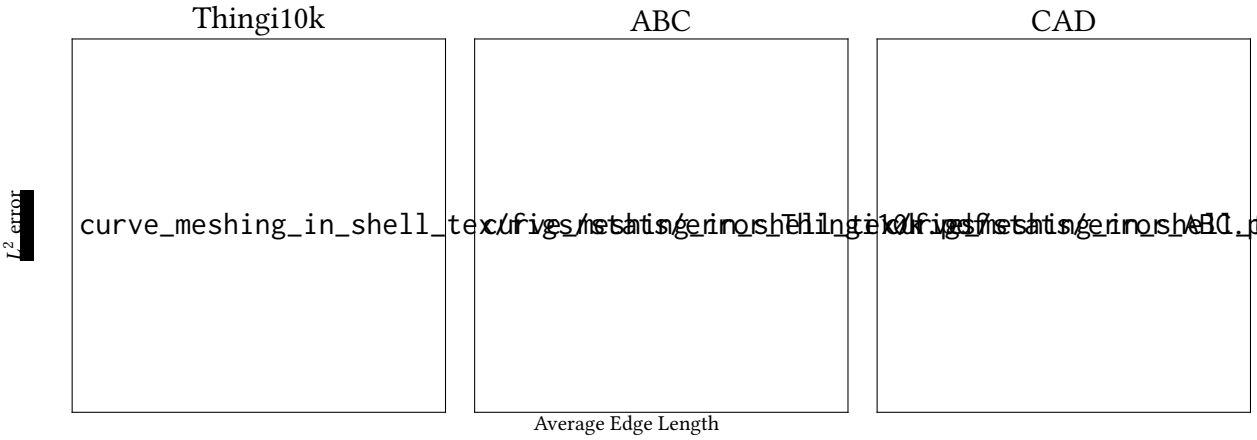
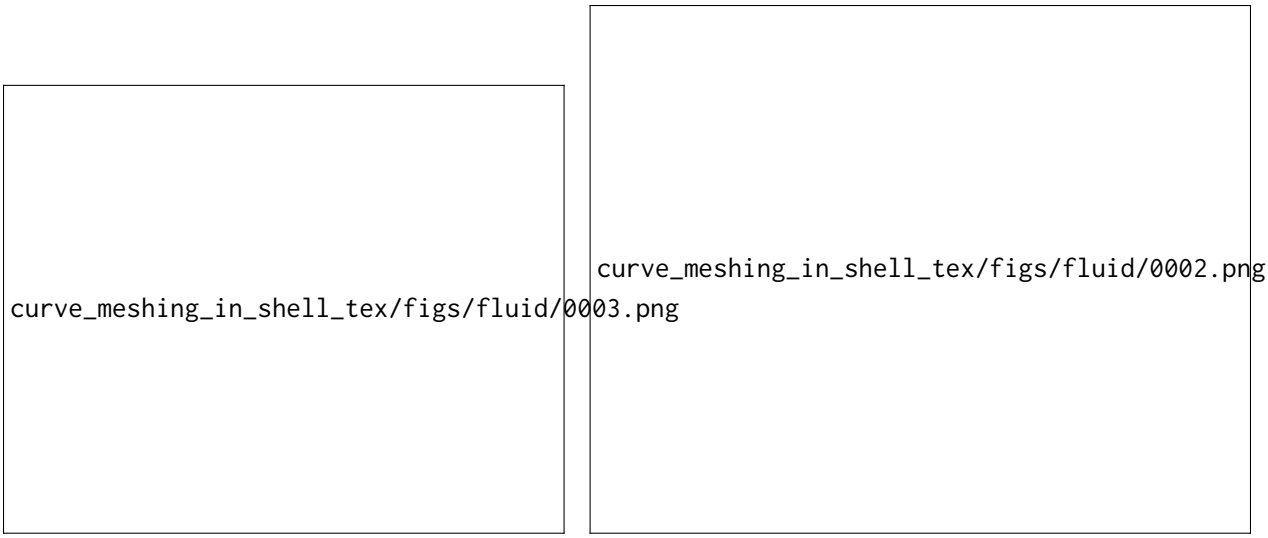


Figure 4.25: L^2 error of the solution of the Poisson equation with respect to model size on our three datasets.



Tetrahedral mesh Curved simulation

Figure 4.26: By meshing the region between a box and a complicated obstacle, we are able to perform non-linear fluid simulation on our curved mesh.

That is, we choose the function u_{exact} to be

$$\begin{aligned} u_{\text{exact}}(x_1, x_2, x_3) = \\ & 3/4\, e^{-((9x_1-2)^2+(9x_2-2)^2+(9x_3-2)^2)/4} + 3/4\, e^{-(9x_1+1)^2/49-(9x_2+1)/10-(9x_3+1)/10} \\ & + 1/2\, e^{-((9x_1-7)^2+(9x_2-3)^2+(9x_3-5)^2)/4} - 1/5\, e^{-(9x_1-4)^2-(9x_2-7)^2-(9x_3-5)^2}, \end{aligned}$$

then we plug it in the equation to obtain f (g is simply u_{exact}). Figure 4.25 shows the L^2 error (average) distribution across our three datasets using our quartic meshes with quadratic approximation of u (i.e., we use superparametric elements).

High Accuracy Fluids Our curved meshes can be directly used to solve different partial differential equations (PDEs). For instance, by meshing the part outside the top shell and discarding the rest, we can generate a curved background mesh for the Navier-Stokes equation (Figure 4.26).

Fast Animation Our coarse curved meshes can be used as animation proxies as in [?Suwelack et al. 2013]. We first compute an as-coarse-as-possible curved mesh (i.e., we set ε to infinity). Then we apply the boundary condition to simulate an elastic distortion of the curved mesh using linear elements. Finally, we use our bijective map φ^4 to map the displacement back to the input high-detailed surface mesh (Figure 4.1). The results are almost indistinguishable to a classical pipeline (i.e., mesh the input mesh), but the runtime is 400 times faster (8s versus over 50 minutes).

4.7 Limitations and Concluding Remarks

We introduce an automatic algorithm to convert dense triangle meshes in coarse, curved tetrahedral meshes whose boundary is within a user-controlled distance from the input mesh. Our algorithm supports feature preservation, and generates meshes with positive Jacobians and high quality, which are directly usable for FEM simulations.

Limitations. Our algorithm generates meshes with a C^0 geometric map. For most FEM applications, this is not an issue. However, for geometric modeling applications, where only the mesh boundary is used, the C^0 geometric map introduces normal discontinuities, which are undesirable. While the surface looks smooth from far away, plotting the reflection lines shows the discontinuity between the normals. We believe an exciting extension of our work would be to study the feasibility of using geometric maps that are C^1 [**?**] or C^2 [**?**]. A second limitation is that, in our implementation, the validity conditions (Definition 4.1) are currently checked using floating point arithmetic, using numerical tolerances to account for rounding errors. While our implementation works on a large collection of models, it is possible that it will fail on others due to the inexact validity predicates. We are not aware of exact predicates for these conditions, and we believe that developing them is an interesting, and challenging, venue for future work.

Future Work. Our current high order mesh optimization pipeline is preliminary, as it only supports vertex smoothing, collapse, and swap. Adding additional operators, allow them to exploit the curved geometric map, and optimizing the boundary could lead to a further increase in mesh quality. While simple at a high-level, this change will require to merge the two parts of our algorithm, a major implementation effort.

Conclusions. We believe that our work will foster adoption of curved meshes, and open the door to a new family of geometry processing algorithms able to take advantage of this highly compact, yet accurate, shape representation.

5 Declarative Specification for Unstructured Mesh Editing Algorithms

5.1 Introduction

«««< Updated upstream Unstructured triangular and tetrahedral meshes are widely used in graphics, engineering, and scientific computing due to their flexibility to represent objects with complex boundaries. Such unstructured meshes find their usage in modeling and rendering 3D objects and scenes, discretizing partial differential equations for physical simulation, collisions detection and response, path planning in robotics, and many other applications. ======= Unstructured triangular and tetrahedral meshes are widely used in graphics, engineering, and scientific computing due to their flexibility to represent objects with complex boundaries. Such unstructured meshes find their usage particularly in modeling and rendering 3D objects and scenes, discretizing partial differential equations for physical simulation, computing collisions and path planning in robotics, and many other applications. »»»> Stashed changes

An unstructured mesh is usually stored in a custom data-structure supporting a set of local operations to add, remove, or change its elements and their properties. A major research effort has been invested in exploring different data-structures and evaluating their generality and efficiency (Section 5.2), which led to the development of mesh libraries such as CGAL [**?**], VCG/meshlab [**?**], OpenMesh [**?**], libigl [**?**], PMP [**?**], and OpenVolumeMesh [**?**]. Commonly, mesh-editing algorithms are tightly coupled with a data-structure and its API, and porting an algorithm from one library to another is a major engineering effort. «««< Updated upstream Code relying on local operations is also inherently error prone, as it usually involves keeping track of properties attached to mesh elements as the mesh itself changes due to the local operations. Parallelizing code using a mesh data structure is also challenging, due to race conditions when multiple threads attempt to change the same region of the mesh.

At a high-level, it is common practice to describe a mesh editing algorithm as a sequence of topological and geometrical editing operations. We argue that this approach is unnecessarily low-level, as it exposes the algorithm designer to technical problems that can be handled automatically by changing the abstraction level. It also makes it challenging to use or customize mesh editing algorithms in larger projects (such as their use in physical simulation for adaptive refinement), as

low-level data structure details percolate in the entire code-base. A particularly difficult challenge in these algorithms is to ensure that a set of conditions (such as manifoldness, being free from self-intersections, minimal quality, maximal geometrical approximation) hold after each operation is applied. This is usually tackled by simulating each operation for the purpose of checking these conditions, an error-prone process that needs to be carefully designed for each pair of operation and condition. The additional presence of attributes attached to vertices, edges, or faces further complicates these problems. ======= Code relying on local operations is also inherently error-prone, as it usually involves keeping track of properties attached to mesh elements as the mesh itself changes due to the local operations. Parallelizing code using a mesh data structure is also challenging due to race conditions when multiple threads attempt to change the same region of the mesh.

At a high level, it is currently common practice to describe a mesh editing algorithm as a sequence of topological and geometrical editing operations. We argue that this approach is unnecessarily low-level, as it exposes the algorithm designer to technical problems that can be handled automatically by changing the abstraction level. It also makes it challenging to use or customize mesh editing algorithms in larger projects (such as their use in physical simulation for adaptive refinement), as low-level data structure details percolate in the entire code-base.

We propose a different way to describe mesh-editing algorithms on manifold meshes: we propose a declarative specification instead of a procedural approach. Instead of focusing on what the algorithm does, we ask the user to specify what are the requirements that the desired mesh should have. We divide these requirements into two groups: invariants and desiderata. The former is a description of hard requirements on the mesh (for example, no inverted elements or no self-intersections), and the latter is a set of desirable properties (such as good quality). A mesh editing algorithm is then described with: (1) a set of per-element invariants (for example, all elements should have correct orientation), (2) a measure for the desiderata (for example, element quality), (3) a set of application-specific attributes attached to mesh elements, and how they are affected by local operations, and (4) a schedule of operation types. We discovered that many existing algorithms for mesh generation, remeshing, and parametrization, can be concisely expressed in this form (Section 5.4), which we denote IDAS (Invariant-Desiderata-Attributes-Schedule). »»»> Stashed changes

We propose a different way to describe mesh-editing algorithms on simplicial manifold meshes, using a declarative specification instead of a more traditional procedural approach. Instead of focusing on what the algorithm does, we ask the user to specify what are the requirements that the desired mesh should have. We divide these requirements into two groups: invariants and desiderata. The former is a description of hard requirements on the mesh (for example, no inverted elements or no self-intersections) and the latter is a set desirable properties (such as good quality). A mesh editing algorithm is then described as: (1) a set of per-element invariants (for example, all elements should have correct orientation), (2) a measure for the desiderata (for example, element quality), (3) a set of application-specific attributes attached to mesh elements, and how they are affected by local operations, and (4) a schedule of operation types. We show that many existing algorithms for mesh generation, remeshing, and parametrization, can be concisely expressed in this form (Section 5.4), which we denote IDAS (Invariant-Desiderata-Attributes-Schedule).

The IDAS specification has been designed with four goals:

1. **Modularity:** The connectivity of the mesh is abstracted from the user, which can only navigate the mesh using a high-level abstraction based on a cell tuple [Brisson 1989]. This reduces the learning curve for a new user, as they only need to learn a navigation API to implement algorithms in IDAS. It will also allow IDAS programs to benefit from continuous progress in data-structure design, as the data structure will be swappable without requiring downstream code changes in the high level IDAS code. This is in stark contrast with existing mesh libraries, which tend to be very invasive in the user code relying on them due to the close connection between navigation, mesh editing, and property management.

2. **Usability:** The user code sees, at all times, a valid mesh: the library simulates each operation transparently allowing the user to navigate on a valid mesh before and after every operation, dramatically simplifying the logic required to define invariants and desiderata. Properties on the mesh are also similarly abstracted, allowing to attach attributes on every simplex independently on the data structure used for implementing the specification.

3. **Efficiency:** The specification purposely requires only definitions of properties on individual elements. This feature allows runtimes for IDAS program to parallelize the computation (Section 5.4) without requiring special attention from an user. We demonstrate that automatic parallelization of mesh editing algorithms is possible on multi-core architectures.

4. **Robustness:** The IDAS specification moves the majority of the robustness issues typical of meshing algorithm on the runtime used to execute a IDAS program instead of the IDAS code itself. This simplifies the development of robust algorithms: for example, the user invariants are guaranteed to be enforced during processing, as the runtime will automatically check them on every modified element. As long as the user provides correct code for the invariant (for example to check for area positivity of an element using a predicate), then the runtime ensures that the invariants will be satisfied for all elements.

Given an algorithm in IDAS form, we design an algorithm and runtime library to realize it, with guarantees on satisfying the invariant and a best effort to maximize the desiderata. Our library exploits shared memory parallelism without any additional effort required from users in the algorithm specification.

«««< Updated upstream To demonstrate the generality and effectiveness of our approach, we provide IDAS formulations for five popular mesh editing algorithms (Section 5.4, Figure **??**): (1) shortest edge collapse [**?**] (decimation for triangle meshes), (2) QSlim [**?**], (3) isotropic triangle meshing [**?**] (remeshing for triangle meshes), (4) harmonic triangulations [**?**] (quality improvement for 3D volumes), and (5) robust tetrahedral mesh generation [**?**] (conversion of surface meshes to volumetric meshes). The IDAS formulation closely resemble the textual description of the algorithms in the corresponding papers: it is compact, readable, and easy to adapt for requirements of specific applications. As an example, we show that modifying (1) and (2) to guarantee a maximal geometric error requires is straightforward. Despite its generality, IDAS implementations executed using our library are comparable or faster than state of the art implementations in open-source

software: the overhead due to the framework generality is more than compensated by the automatic parallelization (Section 5.4).

We believe our contribution is an important step to allow researchers and practitioners to effectively develop new mesh-editing algorithms, shielding the designer of mesh editing algorithms from many of the robustness and correctness challenges plaguing previous low-level approaches, by moving these components inside the runtime environment. It will also allow mesh editing algorithms to be used more easily in larger systems, as they can be tailored to requirements of a specific application with minimal programming effort.

We provide an open-source implementation of our library and of the five mesh editing algorithms as additional material. ======= Given an algorithm in IDAS form, we propose a software library to realize it, with guarantees on satisfying the invariant and a best effort to maximize the desiderata. Our library exploits shared memory parallelism without any additional effort required from users in the algorithm specification.

To demonstrate the generality and effectiveness of our approach, we provide IDAS formulations for four mesh editing algorithms (Section 5.4): (1) shortest edge collapse (decimation for triangle meshes), (2) isotropic triangle meshing (remeshing for triangle meshes), (3) harmonic triangulations (quality improvement for 3D volumes), and (4) robust tetrahedral mesh generation (conversion of surface meshes to volumetric meshes). The IDAS formulation closely resembles the textual description of the algorithms in the corresponding papers: it is compact, readable, and easy to adapt for requirements of specific applications. As an example, we show that modifying (1) and (2) to guarantee a maximal geometric error requires is straightforward. Despite its generality, IDAS implementations executed using our library are comparable or faster than state-of-the-art implementations in open-source software: the overhead due to the framework generality is more than compensated by the automatic parallelization (Section 5.4).

We believe our contribution is an important step to allow researchers and practitioners to effectively develop new mesh-editing algorithms, protecting the designer of mesh editing algorithms from many of the robustness and correctness challenges plaguing previous low-level approaches, by moving these components inside the runtime environment. It will also allow mesh editing algorithms to be used more easily in larger simulation systems, as they can be tailored to the requirements of a specific application with minimal programming effort. »»»> Stashed changes

5.2 Related Work

5.2.1 Mesh Data Structures

Efficient data structures for representing solid geometry have been an intriguing research topic since the early days of computer graphics [Requicha 1980]. As a result, there is a large variety of mesh data structure designs, where they are each optimized for different usage scenarios. Index-array-based mesh data structure encodes each element as a list of vertex indices on its boundary. It is simple and memory efficient, but neighborhood query and local operations are not directly supported. Graph-based mesh data structures, including half-edge [**?**], winged-edge [Baumgart 1972], quad-edge [Guibas and Stolfi 1985], cell-tuple [Brisson 1989], etc., view meshes as graphs,

where each element contains links to its adjacent elements. This design allows for efficient local query and update, making it ideal for algorithms like mesh simplification [**?**]. Linear-algebra-based mesh data structures, such as [DiCarlo et al. 2014; Zayer et al. 2017; Mahmoud et al. 2021], encode adjacency information as sparse matrices. This design elegantly reduces neighborhood query and local operations to sparse matrix computations, which are highly optimized for modern parallel computing architecture. Closely related, is the concept of generalized combinatorial maps [**??**], and the CGoGN library [**?**] provide an efficient implementation which provide parallel traversal of the mesh. By design, mesh data structures provide a low-level interface to manipulate vertices, edges, faces, and tetrahedra. Different designs differ vastly in API and implementation details, making it hard to port algorithm from one data structure to another. In contrast, our framework decouples mesh data structure choice from algorithm specification, providing the flexibility of switching the underlying data structure in a seamless manner.

5.2.2 Domain specific languages in graphics

Our abstraction model of mesh processing algorithms draw inspiration from domain specific languages (DSL) in graphics. For dense regularly structured data such as images, Halide [**?**] popularized the idea of decoupling image processing operations from low level scheduling tasks. Similar abstraction that separates algorithm description from low level data structure and/or parallel architecture can also be found in other DSLs such as Simit [**?**] for simulation over triangle meshes, Taco [**?**] for dense and sparse tensor algebra, Taichi [**?**] for simulation over sparse volumetric data, and Penrose [**?**] for generating diagrams from math notation.

5.2.3 Parallel Meshing

To meet the demand of generating large meshes, a number of popular mesh generation algorithms have been redesigned to leverage modern parallel computing hardware, both in a shared memory and distributed memory setting. Typically a divide-and-conquer strategy is adopted where a mesh is partitioned to run local processing operations on each subdomain in parallel. There are two key challenges involved: (1) how to handle operations involving elements shared by multiple partitions; (2) how to ensure load stay balanced across different processors as the mesh evolves.

One way to mitigate both challenges is to ensure mesh is partitioned into similar sized patches with high area to boundary ratio. A large number of partitioning strategies are available, including clustering-based approaches [Mahmoud et al. 2021], spacial-hierarchy-based approach [**??**], space-filling-curve-based approach [**??**], and general purpose graph partitioning [**?**]. Many variations of space-filling curves have also been used to construct mesh partitions [**??**]. To handle potential conflicts that may arise at partition boundaries, various synchronization strategies have been proposed [**???**] to minimize the amount of communication.

After generating the submeshes, some methods allow each compute node to work on them independently without synchronization. Once all threads are done, the meshes are merged [**????**]. However these methods require complicated merge steps since the tetrahedra in the intermediate boundaries may not align. There are some techniques that compromise the Delaunay condition in some cases, so that the merging operation can be simpler [**?**]. To avoid the tricky merge

operations, other parallel strategies maintain a single complete Delaunay tetrahedralization and use synchronization techniques to avoid race conditions when working on a partition boundary [??]. The parallel constrained Delaunay meshing algorithm [?] cleverly defines the boundary and edge constrains to reduce the variable and unpredictable communication patterns. Some other techniques use locks for handling conflicts and data races [???].

Another set of methods use recursive divide-and-conquer techniques for parallel implementation on shared memory machines [?]. All threads independently work on the internal parts of the mesh and skip the operations at the boundary. After this phase, processing of only the boundary elements becomes the new problem. This technique is then recursively used until all the mesh elements are processed. A similar set of techniques use clever space-filling curves for re-partitioning the mesh boundaries after each recursive phase [??].

Since the submesh boundaries are the main areas of concern, some methods entirely avoid any operations on these boundaries while ensuring the correctness of the result [??]. These methods precompute the domain separators such that their facets are Delaunay admissible. This completely eliminates synchronization overheads, but only applies for Delaunay meshing.

Another conflict handling strategy is to simply reject the offending operations and try executing them later with a new domain partitioning [?]. This reject-and-repartition strategy may not guarantee algorithm termination, thus special care is needed to handle this case.

As the domain mesh evolves, keeping load balanced across processors becomes critical. Typically, this is done by periodically repartitioning the updated mesh. ? proposes a predictive load balancing method to keep partitions balanced. ? uses simple rescaling of the space-filling curve to repartition the domain.

In this work, we are targeting only shared-memory parallelism, thus making the problem of reducing communications between processors less relevant. We use a graph-based space partitioning technique [?] due to its simplicity and availability as open-source code (METIS), but we use it only to reduce the risk of conflicts. To avoid conflicts, we use a shared memory locking mechanism. This approach is only possible for shared-memory parallelism but has the major advantage of not requiring rebalancing and to respect, to a certain degree, the execution order prescribed by the user-code. This approach is possible thanks to the availability of efficient parallel atomic instructions, and parallel libraries based on them (oneTBB).

5.2.4 Scope of Mesh Editing

Mesh Generation Tetrahedral meshing algorithms heavily rely on mesh editing operations. The most common approaches are Delaunay methods [??????????????????Bishop 2016; ?; ?; ?], which strive to generate meshes satisfying the Delaunay condition, grid methods [Yerry and Shephard 1983; Bern et al. 1994; ?; ?; ?; ?; ?], which start from a regular lattice or with a hierarchical space partitioning and optionally intersect the background mesh with the input surface, and front-advancing methods [Sadek 1980; ?; ?; ?], which insert one element at a time, growing the volumetric mesh (i.e. marching in space), until the entire volume is filled .

These algorithms rely on local operations on mesh data-structures, and benefit from our framework to simplify the implementation and gain automatic parallelization. We discuss an implementation of one the more recent algorithms [??] in Section 5.4. Note that some of these

algorithms use local operation that are not implemented yet (such as 5-6 swap), but they could be added to our framework.

Constrained Meshing. Downstream applications often require meshes to satisfy either quality (avoidance of zero volume elements) or geometric (distance to the input surface) constraints. For example, **?** creates a surface approximation within a tolerance volume, the TetWild algorithms [**??**] use an envelope [**?**] to restricts the geometry of the boundary of the tetrahedral mesh, [**?**] adds constraints to local remeshing to avoid interpenetrations in simulations, and [**?**] extends mesh simplification [**??**] to ensure a non self-intersecting result.

These criteria are explicitly modeled as invariants in our framework, and they can be easily swapped in and out existing implementations, as we demonstrate in Section 5.4.

Mesh Improvement. Mesh improvements modifies an existing mesh by changing its connectivity and position of the vertices to improve the quality of its elements [Canann et al. 1996; **?**; **?**; **?**; **?**; **?**; **?**; **?**; **?**; **?**]. We show in Section 5.4 a reimplementation of [**?**] in IDAS form.

Dynamic Remeshing and Adaptive Mesh Refinement (AMR) Simulations involving large deformations are common in computer graphics, and if the surface or volume deformed is represented by a mesh, it is inevitable that after a large deformation the quality of the elements will deteriorate, and the mesh will have to be updated. Additionally, it is often required to concentrate more elements in regions of interest whose location is changing during the simulation, for example to capture a fold in a cloth simulation, or a fracture in a brittle material. These two challenges are tackled in elastoplastic and viscoplastic simulations [**????**], in fluid simulations [**?????**], in cloth simulation [**???????**], and fracture simulation [**?**]. All these algorithms could benefit from our contribution, to simplify their implementation and obtaining speedup due to the automatic parallelization offered by our approach.

A different approach is discussed in [**?**], where the refinement is performed on the basis to avoid the difficulties with explicit remeshing. However, this approach cannot coarsen a dense input, and also cannot increase the quality of elements, making it usable only for specific scenarios [**?**]. Our approach aims at lowering the barrier for integrating explicit remeshing algorithms in simulation applications, thus allowing to directly use standard simulation methods on adaptive meshes without having to pay the high implementation cost for the mesh generation.

When remeshing is paired with algorithms simulating contacts that do not tolerate interpenetrations (for example [**?**]), it is necessary to ensure that adaptive remeshing does not break this invariant. This can be achieved adding non-penetration constraints to each local mesh editing operations, as proposed in [**?**]. Our framework is ideal for developing such methods, as additional constraints can be added to existing mesh editing algorithms with minimal modifications, as we demonstrate in Section 5.4.

Parametrization Conformal mesh parametrization algorithms adapt the mesh during optimization, as a a fixed triangulation restricts the space of metrics realizable [**????????**]. Two very

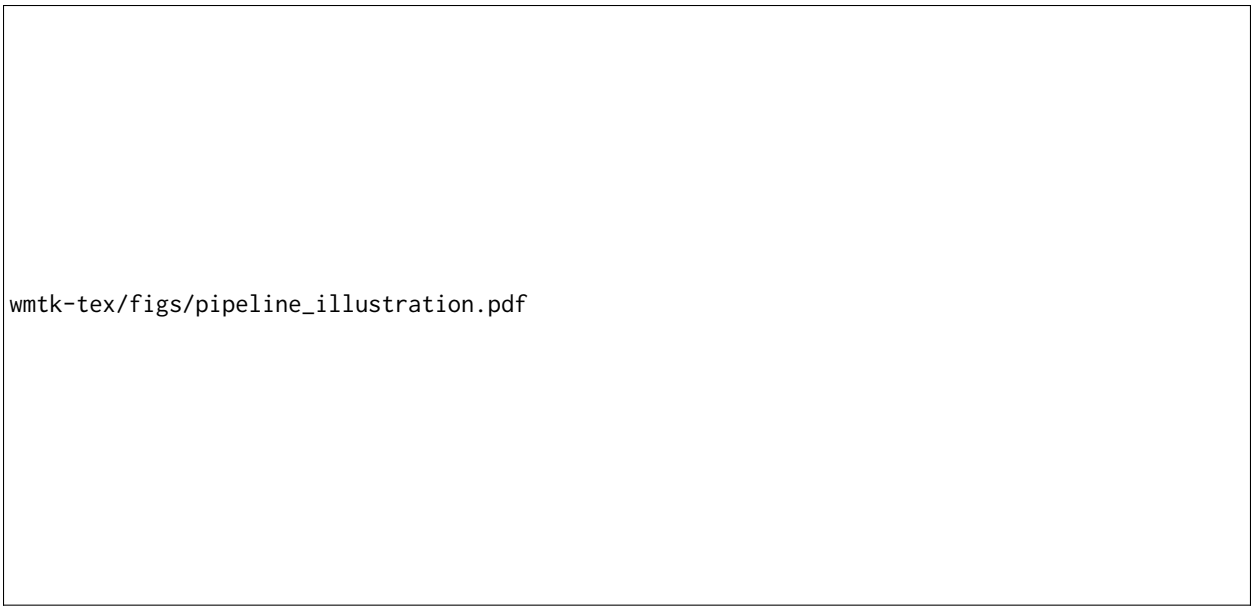


Figure 5.1: Overview of the components behind our specification. A mesh is represented trough its topology, implemented by our library, and a list of user provided attributes. Before an operation is attempted, we explicitly perform a pre-check, and, if successful, we generate a mesh (attributes and topology). At this point, we trigger the after check to validate the operation (e.g., check if the newly generate mesh has positive volume). In case the after check fails, we *automatically* rollback the operation and restore the mesh to its previous *valid* state.

recent works [??] introduce robust algorithms based on Ptolemy flips to compute conformal maps satisfying a prescribed metric.

All these methods require changing the mesh connectivity of a triangle mesh, and could thus benefit from our framework to simplify their implementation and parallelize the mesh editing operations.

Mesh Arrangements/Boolean Operations Boolean operations are basic algorithms often used in geometry processing applications. Recently, [?] proposed a robust way to compute them by constructing a space arrangement, and then filtering the result using the generalized winding number [?]. A similar approach, using an approximated meshing algorithm, has been extended in [?], using a tetrahedral mesher to create the initial arrangement. The reimplementation of TetWild introduced in this paper (Section 5.4) can be extended for a similar purpose.

5.3 Method

Our declarative specification is designed to remove the burden of low-level management of the mesh connectivity and attributes, allowing an algorithm designer to focus only on high-level requirements. The design consists of five components.

5.3.1 Mesh editing components

Operation Rollback. It is common to perform mesh editing to improve a given energy functional, such as mesh quality or element size. However, due to the discrete nature of the operations, it is not possible to use standard smooth optimization techniques, and instead the effect of the energy is evaluated before and after every operation to measure its effect on the energy. This is commonly implemented using an ad-hoc energy evaluation that "simulates" the operation only for the purpose of measuring the energy change. This simulation is complex (especially in 3D), and error-prone, as not only the connectivity changes, but the energy likely depends on properties attached to mesh elements, which needs to be updated accordingly.

We propose instead to make this process opaque to the user, providing to the user-code an explicit copy of the mesh (and up to date attributes) *before and after* the operation is performed to allow an easy and reliable energy evaluation. The correctness and efficiency of this process is handled by the runtime. This reduces the complexity of mesh editing considerably in our experience, as it makes them more similar to traditional finite difference approaches where the energy is evaluated on different points on the domain to approximate its derivative.

Explicit Invariants. It is common to have a set of desiderata on the mesh that needs to be satisfied, such as avoiding triangle insertions or self-intersections. Given the complexity of a mesh editing algorithm it is difficult to ensure that they are satisfied, as these conditions needs to be checked after every operation is applied (and they often depend on attributes too, such as vertex positions).

We propose to make these invariants explicit, and delegate to the library the task of ensuring that they are checked after every mesh modifications, and after the input is loaded. In this way, not only the code is simpler, but it is much easier to ensure correctness, as the checks are handled transparently by the library.

Explicit Attribute Update. Mesh attributes are usually handled by low-level meshing libraries, allowing to attach them to the desired mesh element (vertex, edge, face, triangle, or tetrahedra). However, the handling of attributes after a local operation is performed is usually a responsibility of the user code, as it is dependent on the application.

We propose to make this process more explicit, requiring the user to provide the rules on how to update attributes after operation in a high-level specifications, and delegating the actual update to the library. This makes the specification more direct and less error-prone, and allows users to write algorithms without having to know the low-level details on how the local mesh operations work.

Parallel Scheduling. The type and scheduling of local operations is crucial in mesh editing algorithms. It usually involves maintaining a priority queue of operations, which is updated after every local operation.

We provide a direct way of controlling the operations performed and how the queue is updated. In the library, we can then distribute the work automatically on multiple threads, hiding from the

user code the complexity of performing mesh editing operations in parallel and ensuring that race conditions are avoided.

Abstract Mesh Navigation. Both invariant and attribute updates require navigating a mesh. Instead of relying on data-structure specific navigation, we favor the use of the cell tuple abstraction [Brisson 1989]. This allows the specification to be independent of the mesh data structure used in the library. The `Tuple` stores four indices (three for surface meshes), vertex, edge, face, and tetrahedron and provides a single function per index, called `switch`, to change one index while keeping the other indices fixed. For instance `switch_vertex` changes the vertex index while keeping edge, face, and tetrahedron fixed which has the effect of selecting the opposite vertex on an edge.

5.3.2 Declarative Specification

Our API provides two abstractions: a `TetMesh` (and `TriMesh` class for 2D) (Algorithm 1), and a `Scheduler` (Algorithm 2).

Mesh Classes Both the `TetMesh` and `TriMesh` classes provide the basic local operations (e.g., edge split or collapse) and, for each operation, their corresponding *before* and *after* methods. The mesh class is responsible of implementing the operations changing the topology, and the application code must *only* override the before and after methods to update attributes. The before method has a view of the mesh before the operation, and can thus navigate it to cache local attributes, while the after method has a view after the operation is performed, and it is responsible for updating attributes. In the simple case of regular subdivision of a triangle mesh, the `split_before` caches the coordinates of the two edge endpoints, and the `split_after` computes the position of the newly inserted vertex by averaging them.

In addition, the mesh class provides a method, which can be overridden by the user code, that *automatically* verifies user-provided invariants (e.g., maintain positive elements' volume). All user-provided methods return a Boolean status to notify the mesh classes if the operation fails; in case it does, our API rolls back the operation and restores the topology to the previous valid state. As the connectivity and attributes management is handled by the class, this ensures that, in case of failure of the operation, the mesh will go back to a valid state.

Our API provides the standard local operations: edge collapsing. edge/face swapping, edge splitting, and smoothing. We also provide an additional, non standard, operation: triangle insertion. This operation is an enhanced version of splitting where multiple edges, faces, and tetrahedra are subdivided to represent an input triangle provided as input. This operation is useful to compute mesh arrangements, and it is also used in meshing algorithms [**?**].

Since the `TetMesh` class only handles topology, the operation requires the list of edges and tetrahedron the input triangle intersects. Internally it subdivides all of them, and generates a valid tetrahedral mesh using the connectivity table in [**?**]. The before operation provides the user the list of faces that will be changed by the operation, allowing the user code to explore the mesh and

```
class TetMesh
{
public:

    bool split_edge(const Tuple& t,
        std::vector<Tuple>& new_tets);
    bool collapse_edge(const Tuple& t,
        std::vector<Tuple>& new_tets);
    bool swap_edge(const Tuple& t,
        std::vector<Tuple>& new_tets);
    bool swap_face(const Tuple& t,
        std::vector<Tuple>& new_tets);
    bool smooth_vertex(const Tuple& t);

    bool insert_triangle(
        const std::vector<Tuple>& intersected_tets,
        const std::vector<Tuple>& intersected_edges);

protected:

    bool invariants(const std::vector<Tuple>& tets);

    bool split_before(const Tuple& t);
    bool split_after(const Tuple& t);

    bool collapse_before(const Tuple& t);
    bool collapse_after(const Tuple& t);

    bool swap_edge_before(const Tuple& t);
    bool swap_edge_after(const Tuple& t);

    bool swap_face_before(const Tuple& t);
    bool swap_face_after(const Tuple& t);

    bool smooth_before(const Tuple& t);
    bool smooth_after(const Tuple& t);

    bool insert_triangle_before(
        const std::vector<Tuple>& faces);
    bool insert_triangle_after(
        const std::vector<Tuple>& faces,
        const std::vector<std::vector<Tuple>>& new_f);
};
```

Algorithm 1: API of our `TetMesh` class.

```
template <class Mesh>
struct Scheduler
{
    function<double
        (const Mesh&, Op op, const Tuple&)>
        priority = ...;

    function<vector<pair<Op, Tuple>>
        (const Mesh&, Op, const vector<Tuple>&)>
        renew_neighbor_tuples = ...;

    function<vector<size_t>
        (Mesh&, const Tuple&)>
        lock_vertices = ...

    function<bool(const Mesh&)>
        stopping_criterion = ...;

    function<bool
        (const Mesh&, tuple<double, Op, Tuple>& t)>
        should_process = ...;

    size_t num_threads = ...;
    size_t max_retry_limit = ...;
    size_t stopping_criterion_checking_frequency = ...;

    bool operator()
        (Mesh& m,
        const vector<pair<Op, Tuple>>& ops);
};
```

Algorithm 2: API of our Scheduler.

cache attributes, while the after provides a mapping between the old faces and any newly inserted face in the mesh.

Scheduler The second part of our API is the `Scheduler` that is responsible to control the order of the individual operations and then execute a list of operations. The main purpose of the scheduler is to abstract the operation order and hide parallelization details from the user. Our scheduler provides customizable callbacks, including,

- *Priority* to order the local mesh edit operations.
- *Renew neighbor tuples* that is invoked after a successful operation, to add newly created tuples and operations into the queue.
- *Lock vertices* that provides information on the affected region for the operation, and avoiding conflicts.
- *Stopping criterion* that is checked periodically to terminate the program if certain criterion is met. For example, number of vertices, or quality criterion.

5.3.3 Implementation.

We implement a runtime for our specification in C++, using Intel oneTBB for parallelization.

Data Structure. We opt for an indexed data structure, where we explicitly represent the vertices and the simplex of higher dimension (triangle for 2D, tetrahedra for 3D). Each vertex explicitly stores a list of incident simplices, and each simplex stores a sorted list of its vertices. While not the most efficient option for navigation, this data structure makes the implementation of local operation much simpler.

Parallelization. To avoid conflicts between local operations working on the same part of the mesh, we introduce a synchronization mechanism using locks.

Each mesh vertex is associated with a mutex. Whenever a thread wants to access (read/write) any attribute stored in a vertex, edge, triangle, or tetrahedron it must first acquire a lock on *all* the vertices of the tetrahedron containing the element(s) storing the attribute (Figure 5.2). For example, if a thread wants to read a value on an edge of a 3D mesh, it first needs to acquire a lock on all vertices of all the tetrahedra containing that edge. This mechanism is used also for mesh navigation, and for updating the mesh connectivity.

At a first look, this mechanism might seem cumbersome and expensive. However, we rely on asynchronous, tentative lock acquisition operations. We *try* to acquire the lock, and give up if the lock is already taken by another thread. These operations are efficient on modern hardware and dramatically improve the performance, while avoiding deadlocks. A downside is that an operation might be skipped due to impossibility of acquiring a mutex. These operations are retried for several times (by default 10 times) and run serially if the still do not succeed. Before performing any local operation, we try to acquire the lock on vertices in the 1-ring or 2-ring of the vertex

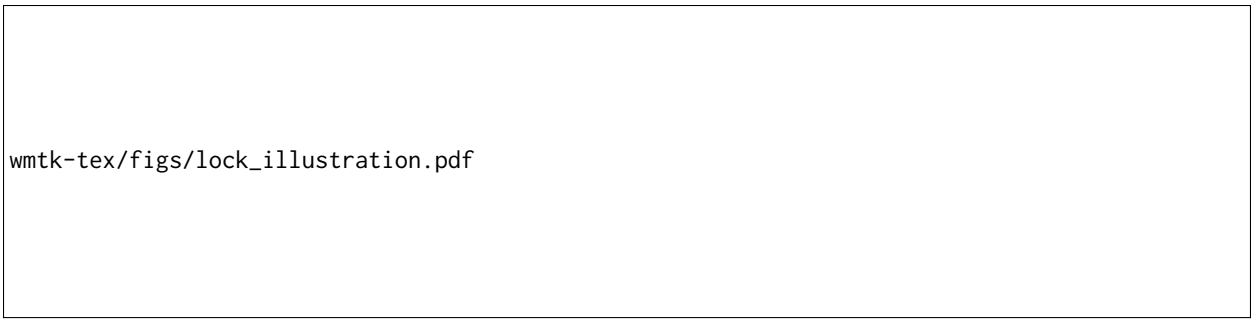


Figure 5.2: Example of the locking region for two edges. In the example the operation requires locking a two-ring neighborhood (e.g., for the edge collapse operation). If the two edges are sufficiently far (right) both operations can be safely executed in parallel. When the two edges are close (left) the operations might fail acquiring the mutexes in the shared area.

involved in the operation. For example, a vertex smoothing operation requires acquiring the 1-ring vertex neighborhood of the smoothed vertex, while an edge-collapse operation on an edge (v_1, v_2) requires acquiring the lock on the 2-ring vertex neighborhood of both v_1 and v_2 (Figure 5.2).

Finally, since we partition the input mesh using Morton Encoding [**?**], the amount of conflicts (and skipped operations) is low.

5.3.4 Example: Shortest Edge Collapse

We show how the library is used in a classical example, shortest edge collapse. In this case, we add a 3D position to every vertex as a vertex attribute (by default, there are no attributes attached to mesh elements). For every attribute and for every operation we plan to use in the scheduler, we need to provide a function that updates such attribute (Algorithm 3). In the `collapse_before` function, we cache the two vertex coordinates associated with the collapsed edge represented by `Tuple t`. In the `collapse_after` function, we generate a new vertex in the middle of the two endpoints of the collapsed edge.

Equipped with the 3D position attribute, which at this point will be automatically kept up to date by the library, we can now schedule the collapse operation (Algorithm 4). For shortest edge collapse, we want to attempt to collapse all edges, prioritizing the shortest ones, until we reach a fixed number of collapses `n_collapse`: in the code we registering the operation type (`ops`), specify how to update the queue after an operation (`renew`) by adding all the neighbouring edges, and specifying the edge length as a priority (`priority`). Note that the outdated elements in the queue that are affected by a local operation are automatically invalidated using a tagging mechanism on the tuples which is opaque to the user.

5.4 Applications

To showcase the generality and effectiveness of our approach, we implement five popular mesh editing algorithms in our framework, and compare them with reference implementations. Overall,

```
//Save two vertices attached to edge t
bool collapse_before(const Tuple& t)
{
    cache.v1p = verts[t.vid()];
    cache.v2p = verts[switch_vertex(t).vid()];
    return true;
}

//Generate a new point
bool collapse_after(const Tuple& t)
{
    verts[t.vid()] = (cache.v1p + cache.v2p) / 2.0;
    return true;
}
```

Algorithm 3: Overridden methods in `TriMesh` sub-class to implement shortest edge collapse.

the performance of our method are competitive for surface applications, but the overhead due to the approach generality is higher in 3D, leading to higher running time.

Shortest Edge Collapsing The simplest algorithm for simplifying a triangle edge is shortest edge collapse [**?**], which performs a series of collapse operations prioritizing the shorter edges. The algorithm requires only one local operation, edge collapse. A common criteria for termination is reaching a desired number of mesh elements. We compare our implementation with the "decimate" implementation in libigl [**?**]. The serial libigl implementation is comparable when running the algorithms serially, and our parallel implementation is up to 9 times faster when using 32 threads (Figure 5.3).

QSlim We use our framework to implement QSlim [**?**]. QSlim collapse edges based on the planarity of the two adjacent faces measured with an error quadric. The algorithm continues to collapse until it reaches a target number of edges. We compare our implementation with the QSlim implementation in libigl [**?**]. The serial libigl implementation is 8 times faster than our implementation, due to their direct manipulations of elements in the queue with each collapse. But our parallel implementation is twice as fast when using 16 threads (Figure 5.4).

Isotropic Remeshing We implemented the widely used algorithm for isotropic remeshing proposed in [**?**]. This algorithm alternates edge collapse, edge flips, edge splits, and tangential smoothing to obtain a mesh that is isotropic (i.e. all elements have the same size) and where all triangles are close to equilateral. The process is guided by a user-provided target edge length L, and terminates when no local operation leads to either an improvement in the desired edge lengths or an improvement in vertex valence [**?**].

```
//Collect edges attacched to tris
vector<Tuple> new_edges_after(const vector<Tuple>& tris)
{
    vector<Tuple> new_edges;
    for (auto t : tris) {
        for (auto j = 0; j < 3; j++) {
            new_edges.push_back(
                tuple_from_edge(t.fid(), j));
        }
    }
    return new_edges;
}

bool collapse_shortest(int n_collapses)
{
    //Register operations
    auto ops = vector<pair<Op, Tuple>>();
    for (auto& l : get_edges())
        ops.emplace_back("edge_collapse", l);

    //After a successful operation,
    //we append all new edges
    auto renew = [](auto& m, auto op, auto& tris) {
        auto edges = m.new_edges_after(tris);
        auto optup = vector<pair<Op, Tuple>>();
        for (auto& e : edges)
            optup.emplace_back("edge_collapse", e);
        return optup;
    };

    //priority in which we collapse
    auto priority = [](auto& m, auto op, const Tuple& e) {
        const auto v1 = m.verts[e.vid()];
        const auto v2 = m.verts[e.switch_vertex(m).vid()];
        auto len2 = (v1 - v2).squaredNorm();
        return -len2;
    };

    //Set the functions to the scheduler
    Scheduler executor;
    executor.renew_neighbor_tuples = renew;
    executor.priority = priority;
    executor.stopping_criterion_checking_frequency =
                n_collapses;
    //We stop only when we perform n collapses
    executor.stopping_criterion =
                [](auto& m) { return true; };
    //Run the executor
    executor(*this, ops);
}
```

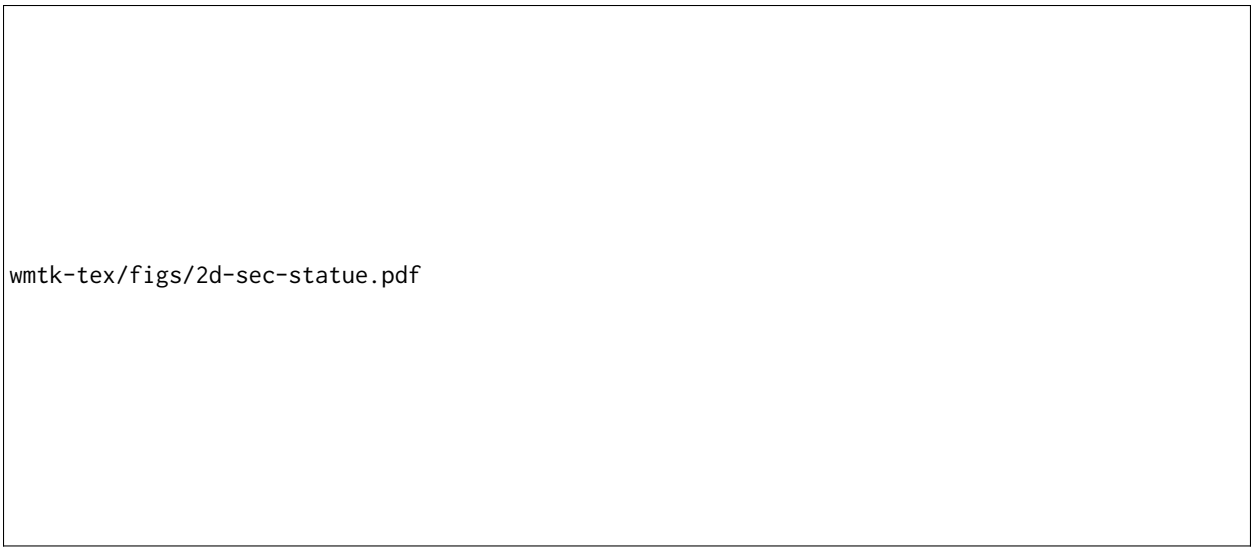


Figure 5.3: Comparison of our parallel implementation (32 threads) of shortest edge collapse (scalability plot on the right, from 1 to 32 threads) of a model with 281, 724 faces with the serial version in libigl. Both libigl and our output have 28, 168 faces and comparable edge length (1.058 for libigl versus 1.061 for ours). Our serial method runs in 4.9s (5.84s on a single thread, 0.52s with 32 thread, leading to a speedup of 11×), while libigl runs in 2.74s.

In Figure 5.5, we compare our implementation of [**?**] with the implementation in OpenFlipper [**?**]. The OpenFlipper implementation is 2.5 times faster when running on a single thread, and our implementation becomes faster after 4 threads are used (Figure 5.5).

Harmonic Triangulations The *harmonic triangulations* algorithm has been introduced as an alternative to sliver exudation in the Delaunay tetrahedralization pipeline to efficiently reduce sliver tetrahedra. The original paper [**?**] proposes to use both flip and smoothing operations.

The code provided by the authors implements a reduced version of the algorithm proposed in the paper, restricting the optimization to 3-2 edge swap operations. We thus implemented both a reduced version for a fair comparison (Figure 5.6) and a complete version. Our more generic framework is twice as slower than the hand-optimized code written by the authors when running serially, and it is 2 times faster when running on 32 threads (Figure 5.6).

Tetrahedral Meshing The TetWild algorithm is a tetrahedral meshing algorithm with minimal input requirements: given an input triangle soup, it can generate a tetrahedral mesh which approximates its volume. We take inspiration from the original algorithm introduced in [**??**] with a few modifications: (1) we use the insertion operation [**?**] (using rational coordinates) as a replacement for their BSP partitioning, as this simplifies the implementation, (2) we use the envelope proposed in [**?**] instead of sampling, and (3) we use 2-3 face swapping, 3-2 and 4-4 edge swapping operations, to simplify the implementation. We show results on two models in

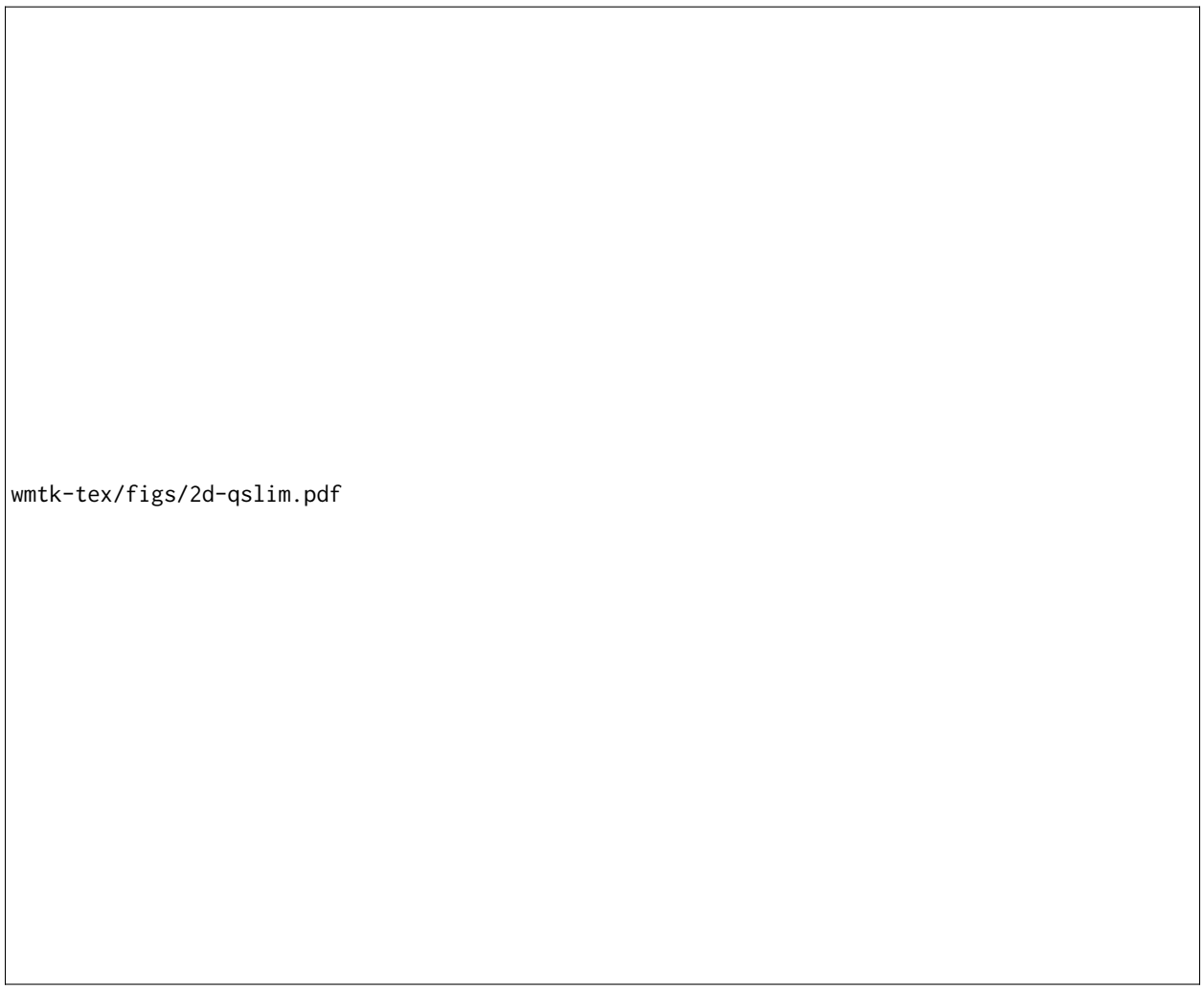


Figure 5.4: Comparison of our parallel implementation of QSlim with the serial version in libigl for a model with 1,909,755 faces. Top right, the libigl output has 17,891 faces and takes 41.26s. Bottom left, our output has 17,906 faces and runs in 306.59s. Our implementation scales well: 347.59s with one thread and 13.88s with 32 (25× speedup).

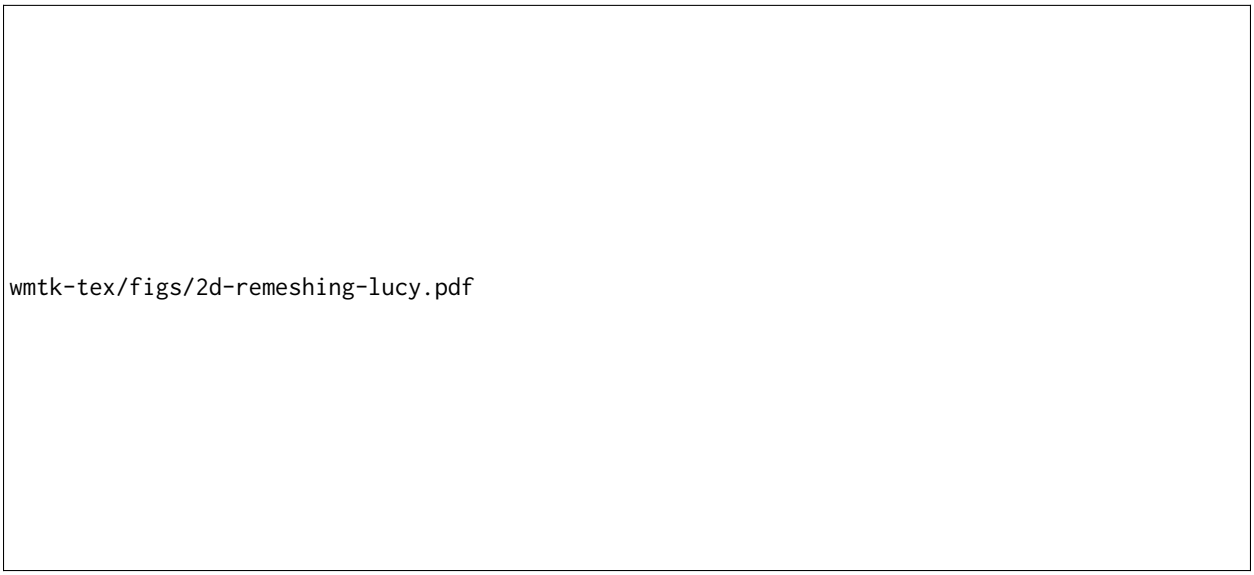


Figure 5.5: Example of uniform remeshing a model with 2, 529, 744 triangles (left) with the same target edge length. Middle, [**?**] remeshes it to 78, 322 faces in 31.96 seconds. On the right is our 32-thread implementation, which generates 71, 640 triangles in 8.2 seconds (78.34s serial, 95.0s for a single thread, leading to a speedup of 11×). The difference in density of the meshes is due to differences in the detail of the implementation, which makes the two methods reach an average vertex valence of 5.999, and a similar target edge length (differ 0.01% of the bounding box diagonal length) with a different element budget.

Figure 5.6: Example of *Harmonic Triangulations* starting with one million Gaussian distributed random points. Both our and the reference implementation reach a similar target number of tetrahedra (5.9 million for the reference and 6.1 million for ours, due to a difference in operation ordering) and a similar Mean Harmonic Index (0.547 for the reference and 0.554 for ours). Our method takes 3.82s with 32 threads (15.49s serial, 40.49s on a single thread, speedup of 11×), while the reference serial implementation takes 6.37s.

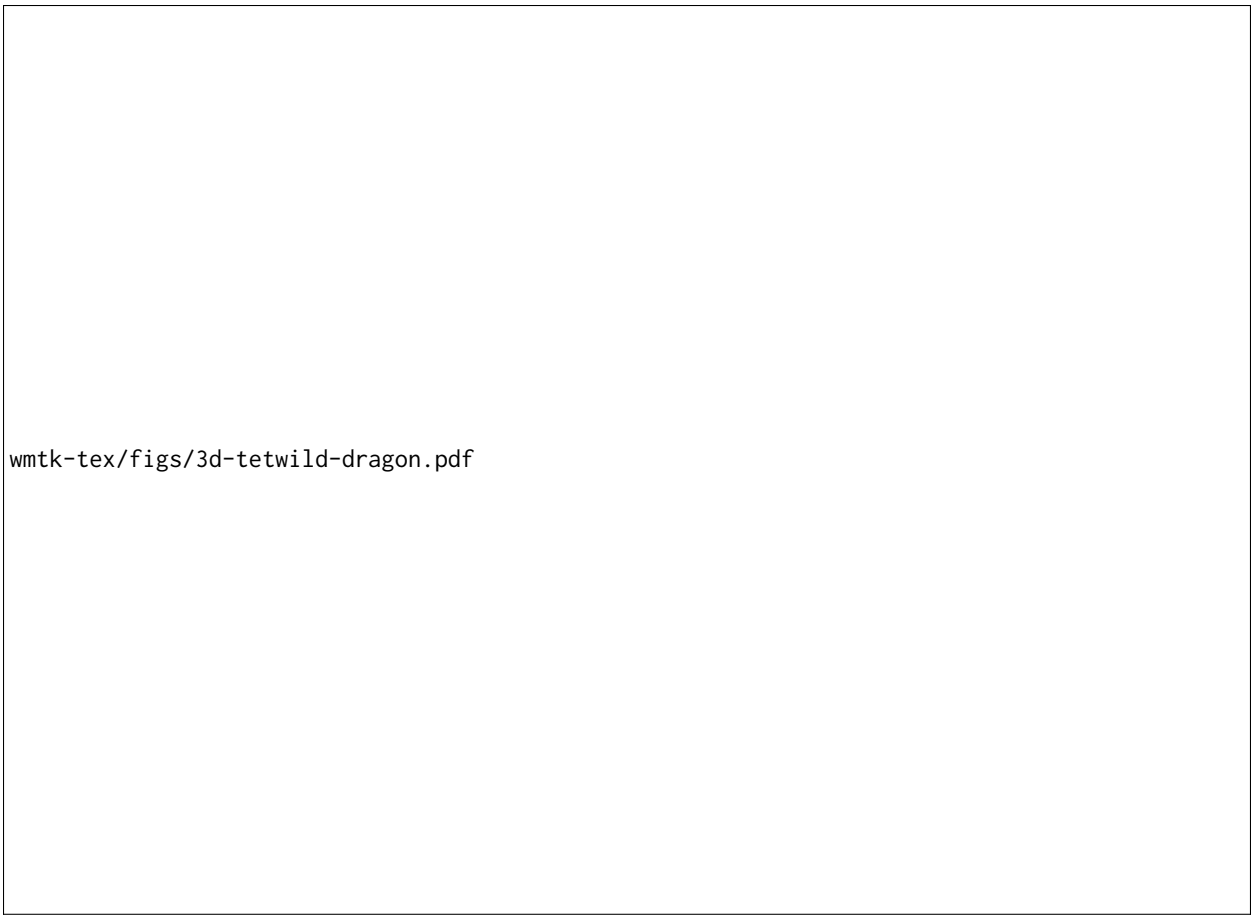


Figure 5.7: Tetrahedralize a surface with 856, 294 faces. Original TetWild (top right) generates a mesh with 56, 761 tetrahedra in 287.58s; our reimplementation (bottom left) generates a mesh with 44, 866 tetrahedra in 153.33s with 8 threads (452.42s serial, 521.47s on a single thread, speedup of 3.4×). The difference in number of tetrahedra is likely due to the different order of scheduling of operations due to the partitioning.

Figure 5.7: the results are very similar to the original implementation, and our version is 2 times faster when using 8 threads. We experimentally observe that our framework scales well up to 8 threads, after that the algorithm becomes slower. This is because, as we increase the number of threads and partitions, the frequent conflict in tetrahedral mesh edge operations affects the parallel performance. We believe that this observation might be useful for the future design of high performance concurrent mesh generation algorithms.

5.4.1 Parallelization

Enabling the parallelization mechanism introduces a minor slowdown as visible in the difference between the pure serial and one thread timings on our applications, due to the additional cost of allocating mutexes and to acquired them. The algorithm scales well in all 5 applications (figures 5.3, 5.4, 5.5, 5.6, 5.7), obtaining a scaling speedup. We would like to remark that thanks to our

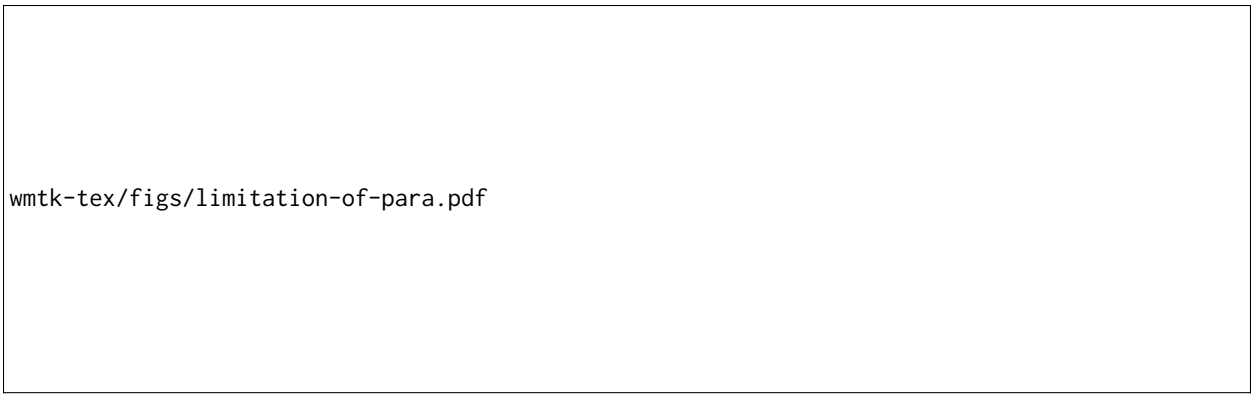


Figure 5.8: Example of splitting the longest edge on the bottom of the triangle. On the left, the edge can be split to generate the dark blue edge. In the next iteration, the left edge is split (by the light blue edge) leading to a decreasing maximum edge-length. On the left, the bottom edge is locked illustrated by a dashed line. In this case the next iteration splits the left edge (dark blue edge) and so on. As long as the dashed edge is locked it will never be split preventing the maximum edge length to decrease.

specification and our runtime, the serial and parallel implementation of the 5 algorithms above is almost identical.

An inevitable drawback of parallelization is that the algorithms cannot efficiently preserve ordering. For instance, in shortest edge collapse, every thread will try collapse edges in its own partition independently from the others. If one of these collapses append on the partition's interface, the thread will need to acquire a lock. In case of failure the collapse is postponed to a later stage thus not respecting the order (Figure 5.8). This is a rare event that is more problematic for fast operations.

5.4.2 Algorithm Modifications

A major motivation to invent and develop this declarative language is enabling easy customization of meshing algorithms. As an example, we add an additional termination criteria to the shortest edge collapse and uniform surface refinement. Integrating the envelope check is straightforward with our approach, as it only requires adding the envelope check to the invariants. We use the open-source library proposed in [**?**], which allows to directly specify the maximal allowed surface deviation. The envelope adds a noticeable computational cost, which is ameliorated by our parallel implementation (figures 5.9 and 5.10).

5.4.3 Large-scale dataset validation

To validate our framework we run our reimplementation of uniform remeshing and tetrahedral meshing on the Thingi10k dataset [**?**]. We run all experiments serially on an individual node of an HPC cluster an Intel Xeon Platinum 8268 24C 205W 2.9GHz Processors limiting the runtime to 15 hours.

Figure 5.9: Shortest edge collapse with envelope containment of a model with 857, 976 faces. Our method successfully generates a mesh with 71, 298 faces in 37.49s with 32 threads (731.32s serial, 725.34s on a single thread, speedup of 20×).

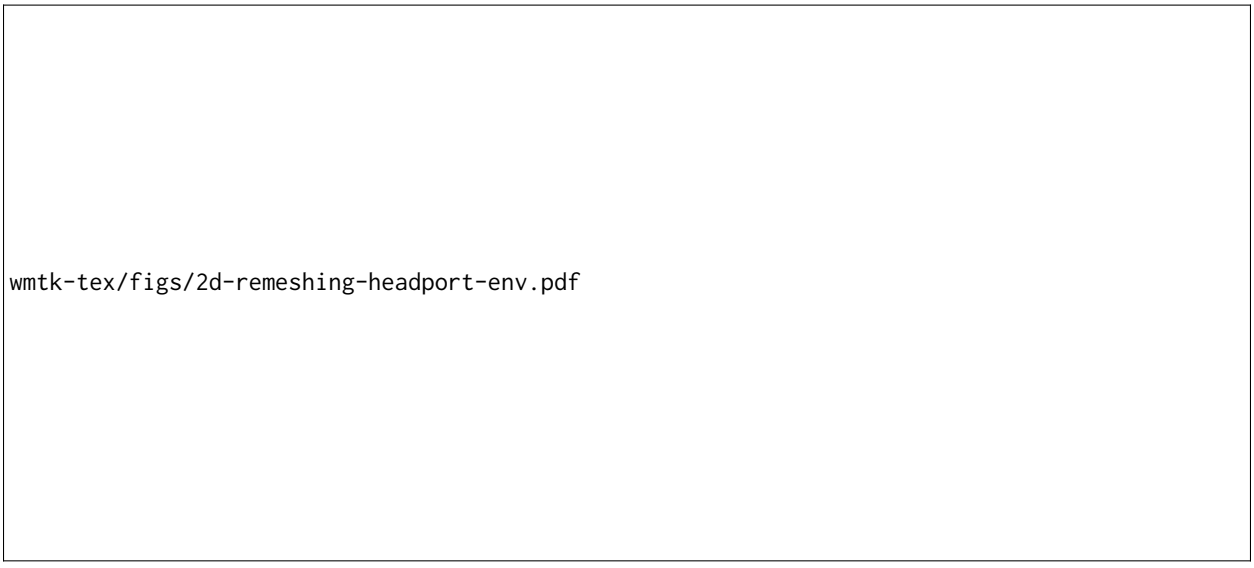


Figure 5.10: Uniform remeshing with envelope containment check of a model with 198, 918 faces. Our method produces a mesh with 68, 202 faces in 29.11s with 32 threads and 493.54s for a single thread (483.68s for the serial version) leading to a speedup of 16×.

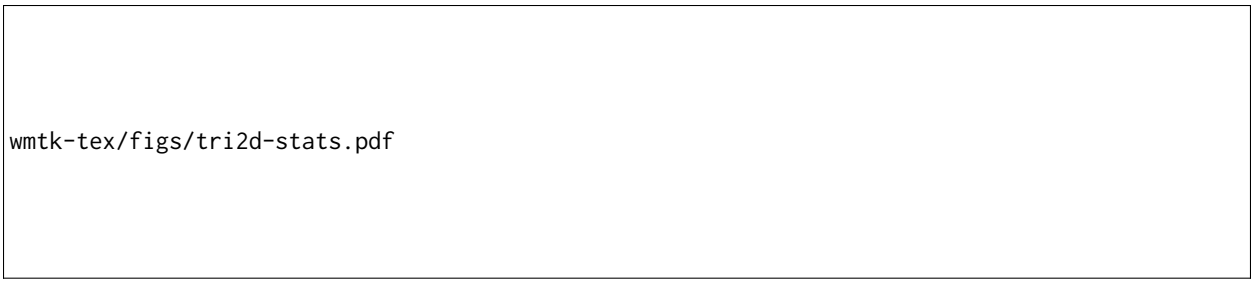


Figure 5.11: Timings, target edge length ratio, and valence for every model in the dataset. Most models finish within a minute. The target edge length ratio measure how well our algorithm simplifies the meshes to reach the desired edge length, with an optimal value of 1. Since uniform remeshing strives to generate regular meshes, for most model our algorithm is able to obtain the optimal valence of 6.

Figure 5.12: Timing, max and average AMIPS energy (capped at 20) for maximum 25 iterations of tetrahedral meshing. Most models finish withing 20 minutes with only a few taking up to a day. Even by limiting the iterations to 25, most models reach an average AMIPS energy lower than 10, with optimal value at 3.

For uniform remeshing, Figure 5.11 shows the time, average edge length normalized by the target, and average valence of isotropic remeshing on the ten thousand models. Most of our models finish within 10 seconds with only a few requiring more than a minute. For almost all meshes, the algorithm succeeds at reaching the target edge length and valence of 6.

For TetWild we limit the number of iterations to 25 (Figure 5.12). We note that within the 15 hours limit only 2.5% models did not finish, and after 25 iterations 3% of the models still have some rational coordinates. Among the successful models, most finish within 20 minutes and succeed in achieving high-quality meshes (only 8 models have average AMIPS larger than 10).

5.5 Concluding Remarks

This paper introduces a new declarative specification for mesh algorithms to allow an easier implementation, while at the same time obtaining competitive performances and exploiting parallel hardware.

Using this specification, we implement five popular mesh editing algorithms covering mesh generation and optimization on surfaces and volumes, which can be easily adapted for other use

cases: we demonstrated that integrating an envelope check requires only a few lines of code.

The library we implemented supports shared memory parallelism, which leads to a good scaling on the machines we tested it on. We believe an exciting venue for future work would be the implementation of a library for our specification targeting MPI to distribute the computation over an HPC cluster. Having access to such a library would allow our five mesh editing applications to run on a distributed environment with minimal or no changes.

6 Conclusion

A | Appendix

A.1 Proofs

A.1.1 Theorem 3.2

Consider the unique (up to symmetry) piece-wise affine map T deforming Δ into a reference prism $\hat{\Delta} = \{u, v \geq 0 | u + v \leq 1\} \times [-1, 1]$, identifying the bottom surface of $\hat{\Delta}$ as $z = -1$, middle surface as $z = 0$, and top surface as $z = 1$. Let T_T be the affine map mapping tetrahedron $\boldsymbol{T}$ to the reference prism. Since the volume of all $\boldsymbol{T}$ is positive by assumption, the map T is bijective and orientation preserving. In particular, T transforms $\mathcal{V}(p)$ to the const vector $e_z = (0, 0, 1)$ since the edges of the prism are mapped to axis aligned edges of the reference prism. The projection operator $\mathcal{P}(p)$ can thus be equivalently defined as $\mathcal{P}(p) = T^{-1}(\mathcal{P}_z(T(p)))$, where $\mathcal{P}_z$ is the projection over the z-axis in the reference prism. $\mathcal{P}_z$ is bijective if the piecewise linear mesh intersecting Δ is composed of triangles with positive area (and the boundaries are mapped to the boundaries) [**?**], after being mapped to the reference Δ and projected by $\mathcal{P}_z$. In the reference domain, having positive area after projection (with a fixed boundary) is equivalent to that the dot product between the projection direction and the normal of every face is positive. In the top half of Δ, $\mathcal{P}_z$ is bijective if for every point $T(p), p \in f\ T(n_p) \cdot [0, 0, 1] = T(n_p) \cdot T(\mathcal{V}(p)) > 0$ which holds since $n_p \cdot \mathcal{V}(p) > 0$ (from the definition of section) and the fact that T is orientation preserving and thus not changing the sign of the dot product.

A.1.2 Proposition 3.4

For simplicity, we will show that a uniform lower bound δ for all vertices exists. By consider a triangle of T, extruded over the displacement field $\mathcal{N}$, we obtain a generalized half prism with six vertices 0, e_1, e_2, δn_0, $e_1 + \delta n_1$, $e_2 + \delta\ n_2$, with δ a positive constant. The indices of the tetrahedra are:

(0 ,2, 3, 4), (0 ,3, 4, 5), (0 ,1, 5, 3), (0 ,1, 2, 3), (1 ,2, 3, 4), (0 ,1, 2, 4), (0 ,1, 5, 4), (1 ,3, 4, 5), (1 ,2, 3, 5), (0 ,2, 5, 4), (0 ,1, 2, 5), (2 ,3, 4, 5).

For a sufficiently small δ, the linear term will dominate the higher power of δ, which we omit. The dominant volume terms are listed in the following table: ($e_{ij} := e_j - e_i$)

(0 ,2, 3, 4)	$\text{vol1}(0, e_2, \delta n_0, e_1 + \delta n_1)$	$\delta\langle e_2, n_0, e_1\rangle$
(0 ,3, 4, 5)	$\text{vol2}(0, \delta n_0, e_1 + \delta n_1, e_2 + \delta n_2)$	$\delta\langle n_0, e_1, e_2\rangle$
(0 ,1, 5, 3)	$\text{vol3}(0, e_1, e_2 + \delta n_2, \delta n_0)$	$\delta\,\langle e_1, e_2, n_0\rangle$
(0 ,1, 2, 3)	$\text{vol4}(0, e_1, e_2, \delta n_0)$	$\delta\langle e_1, e_2, n_0\rangle$
(1 ,2, 3, 4)	$\text{vol5}(e_1, e_2, \delta n_0, e_1 + \delta n_1)$	$\langle e_{12}, \delta n_0 - e_1, \delta n_1\rangle$ $= \delta\langle e_2, -e_1, n_1\rangle$
(0 ,1, 2, 4)	$\text{vol6}(0, e_1, e_2, e_1 + \delta n_1)$	$\delta\langle e_1, e_2, n_1\rangle$
(0 ,1, 5, 4)	$\text{vol7}(0, e_1, e_2 + \delta n_2, e_1 + \delta n_1)$	$\langle e_1, e_2 + \delta n_2, \delta n_1\rangle$ $= \delta\langle e_1, e_2, n_1\rangle$
(1 ,3, 4, 5)	$\text{vol8}(e_1, \delta n_0, e_1 + \delta n_1, e_2 + \delta n_2)$	$\delta\langle \delta n_0 - e_1, n_1, e_{12} + \delta n_2\rangle$ $= \delta\langle -e_1, n_1, e_2\rangle$
(1 ,2, 3, 5)	$\text{vol9}(e_1, e_2, \delta n_0, e_2 + \delta n_2)$	$\langle e_{12}, \delta n_0 - e_1, e_{12} + \delta n_2\rangle$ $= \delta\langle e_{12}, \delta n_0 - e_1, n_2\rangle$ $= \delta\langle e_2, -e_1, n_2\rangle$
(0 ,2, 5, 4)	$\text{vol10}(0, e_2, e_2 + \delta n_2, e_1 + \delta n_1)$	$\delta\langle e_2, n_2, e_1\rangle$
(0 ,1, 2, 5)	$\text{vol11}(0, e_1, e_2, e_2 + \delta n_2)$	$\delta\langle e_1, e_2, n_2\rangle$
(2, 3, 4, 5)	$\text{vol12}(e_2, \delta n_0, e_1 + \delta n_1, e_2 + \delta n_2)$	$\langle \delta n_0 - e_2, \delta n_1 - e_{12}, \delta n_2\rangle$ $= \delta\langle -e_2, e_1, n_2\rangle$

For all the 12 tetrahedra the linear term is multiplied by a determinant containing the edges of the prism. We check directly that all these determinants are positive due to the assumption that $\mathbf{C}\mathcal{N}_i > 0$.

A.1.3 Theorem 3.5

Consider a ball around a point p of the middle face M of the prism. If the radius $r(p) > 0$ is sufficiently small, it contains, at most, a single vertex of M, and parts of incident faces. As M is compact, there is a minimal value of r on M, and it is positive. The line along the normal, passing through v, intersects incident faces at v so it cannot intersect them at any other point. Thus, the top and bottom prism vertices can be obtained by displacing V by r in either direction along the normal. Consider the r-neighborhood of the M, i.e. the union of all balls of radius r centered at points of M. This is a convex set, containing only M and parts of vertex-adjacent faces. All top and bottom prism vertices are in this set. Thus tetrahedra connecting these vertices are also in the set, i.e., complete prisms. The fact that tetrahedra are nondegenerate is established in Proposition 3.4.

A.1.4 Theorem 3.6

To show that $\mathcal{T}$ is a section of $\mathcal{S}$ we need to show that (1) $M_\Delta = \mathcal{T} \cap \Delta$ is a simply connected patch for any prism of $\mathcal{S}$ and (2) for each prism Δ and every point $p \in M_\Delta$ the dot product between the face normal $n(p)$ and the pillars $\mathcal{V}_i, i = 1, 2, 3$ is strictly positive.

(1) is trivial since for every prism Δ of $\mathcal{S}$, Δ contains exactly one triangle of the topological bevel refinement face of $\mathcal{T}$.

To prove (2), consider the dot products of the normals of each of the triangles in the beveled region sharing at least a vertex with M_Δ, and the vectors along the pillars of M_Δ. We use colors to refer to the triangles of beveled prisms as shown in Figure) A.1 left.

The prism corresponding to the pink triangle covers only the interior of the original triangle. It

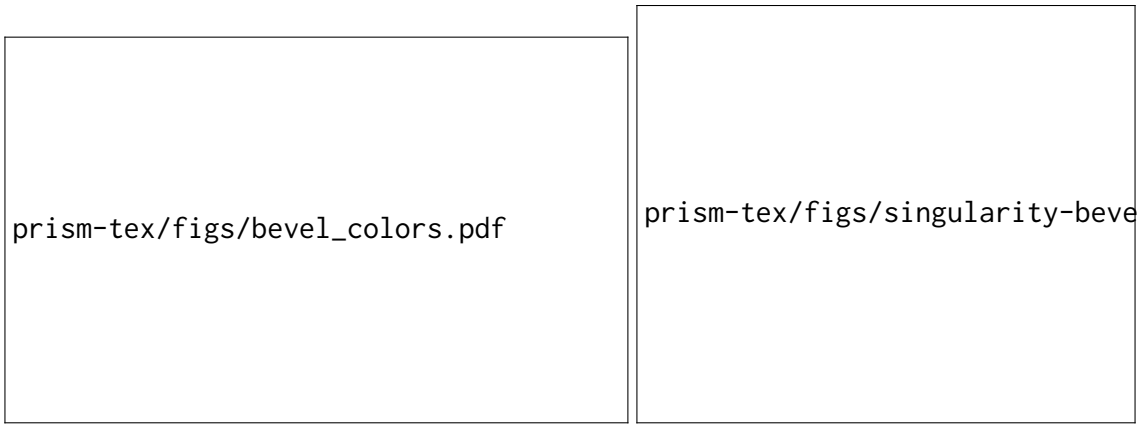


Figure A.1: Bevel patterns used in the proofs in Appendix A.1.4 and Appendix A.2

satisfies the dot product condition since its pillars are copied from the solution of Problem (3.3) at each vertex. A similar argument can be made for the green prisms: each prism gets its pillars from the edges incident at two of the vertices, which are compatible with the normal of the adjacent triangle (e.g., red and blue pillars have positive dot product with the normal of T_1). Finally, the prisms corresponding to orange triangles cover the same one ring that was used to compute the original pillars (e.g., red pillar is compatible with T_1 and T_2). Since the same pillar is used for each of the 3 vertices of the original prism, the dot product is unchanged.

A.1.5 Theorem 3.7

We use A° to denote the interior of a set A. We first assert that the topological disk patch $\mathcal{T} \cap C$ coincides with $\mathcal{T} \cap C'$. To make the notation more concise, we define $\mathcal{T}_C = \mathcal{T} \cap C$ and $\mathcal{T}_{C'} = \mathcal{T} \cap C'$.

We show that $\mathcal{T}_C \subset C'$ by contradiction: assume there is $p \in \mathcal{T}_C$ but $p \notin C'$. Choose a point $q \in \partial\mathcal{T}_C$, that belongs to single prisms in C and C' and denote the enclosing prism in C as Δ_q and the corresponding one in C' as Δ'_q. It follows from the positivity of the volumes of Δ_q and Δ'_q (Assumption 1), that there is $\varepsilon > 0$ such that the half-ball (for sufficiently small ε) $\mathrm{Ball}(q, \varepsilon) \cap \Delta_q$ coincides with $\mathrm{Ball}(q, \varepsilon) \cap \Delta'_q$. Futhermore, the validity of the initial shell guarantees that there exists an interior point $\bar{q} \in \mathrm{Ball}(q, \varepsilon) \cap \Delta_q^\circ \cap \mathcal{T}$, and $\bar{q} \in \Delta_q'^\circ \cap \mathcal{T} \subset C'^\circ \cap \mathcal{T}$.

Since neither p or $\bar{q}$ lies on the boundary side triangles of $\partial C'$ (since $p, \bar{q} \notin \partial\mathcal{T}_C$), therefore there exists an interior path P (i.e., a path both in $\mathcal{T}_C$ and in $\mathcal{T}_{C'}$) that connects p and $\bar{q}$. By continuity the path P it will cross $\partial C'$ at some point $p_i \in \mathcal{T}_C$. Since the $\partial\mathcal{T}_{C'} = \partial\mathcal{T}_C$ is a fixed boundary loop (by the definition of $\mathbb{O}$), and does not contain interior points, p_i can only cross on the top surface or bottom surface of C' which contradicts Assumption 2. The roles of C and C' can be inverted to complete the proof.

We then map M_C to a regular planar n-gon D, with vertices on ∂M_C mapped cyclically to the vertices of ∂D. It follows from [**?**, Corollary 6.2] that such a convex combination mapping $\varphi \colon M_C \to D$ is bijective. Then, similarly to the construction in Theorem 3.2, we define $\phi \colon C \to D \times [-1, 1]$ through affine mapping induced by the thetrahedralization of the prisms Δ_i. Similarly, we define $\psi \colon C' \to D \times [-1, 1]$. The mapping ψ is also bijective because of Assumption 1 and the definition of $\mathbb{O}$ it follows that the codomain of ψ is the same as ϕ. Note that in both cases, the map

transforms the vector field $\mathcal{V}(p)$ to the constant vector field $e_z = (0, 0, 2)$, and the projection is $\mathcal{P}_z$, as defined in Appendix A.1.1.

The dot product condition (Assumption 3) ensures that $\mathcal{P}_z$ defines a bijection between D and $\psi(\mathcal{T}_C)$; and further, the image satisfies $\mathcal{P}_z(\psi(\Delta_i' \cap \mathcal{T})) = \psi(M_i')$ where M_i' is the middle surface (a single triangle) of Δ_i'. Therefore, since ψ and $\mathcal{P}_z$ are both continuous and bijective, they are homeomorphisms which preserve topology, thus $\Delta_i' \cap \mathcal{T}$ is a simply connected topological disk.

A.2 Extension to meshes with Singularities

Most of the proofs and definition in Section 3.3 easily extends to meshes with singularity or pinched shells. For instance, both Theorem 3.2 and Definition 3.3 apply to pinched shells by just considering a prism made of 4 (2 for the top and 2 for the bottom slab) instead of 6. Propositions 3.4 and 3.5 both rely on per-vertex properties; by just excluding singular vertices (and neighboring prism) from the statements the proofs (and our algorithm) still holds.

Theorem 3.7 requires some minor additional considerations. As a consequence of beveling, no two singularities are adjacent. Therefore, we can always find a point $q \in \partial\mathcal{T}_C$ that belongs to a single prism (Appendix A.1.5) since every prism will have a positive volume because no singularities are adjacent.

Finally Proposition 3.6 requires a new proof since the beveling pattern used for singularities is different.

Proof. In Figure A.1 right, we highlight in red and orange the triangles not covered by the discussion in Appendix A.1.4. The prism corresponding to an orange middle triangle intersects the edge-adjacent triangle. And since the new pillars are computed as the average of the normals of the two adjacent triangles with dihedral angle strictly less than 360° (Section 3.3.5 inset), each dot product is positive. The prism for a pink middle triangle intersects only the interior of the original triangle, and the dot product between the pillars and the triangle normal are positive by construction. □

A.3 Reduction to a Standard QP

With slack variable t, Problem 3.2 can be rewritten as

$$\max_t t;\ d_v n_f \geq t, \|d_v\| = 1, t \geq \epsilon.$$

In the admissible region, substituting $t = 1/s$,

$$\min_s s;\ s d_v n_f \geq 1, \|d_v\| = 1, 0 < s \leq 1/\epsilon.$$

Substituting $x = s d_v$ (thus $\|x\| = s\|d\| = s$), we obtain the QP problem 3.3. Note that the lower bound on s is implied from $s d_v n_f \geq 1$, and the upper bound $\|x\| < 1/\epsilon$ is checked a posteriori.

A.4 Bijectivity of the Nonlinear Prismatic Transformation

In this section, we show that the isoparametric transformation of prismatic finite element [?] is also bijective if I1 holds.

We follow the notation from Appendix A.1.2. The map is defined as (for the top slab)

$$\begin{aligned} f(u,v,\eta) &= ue_1 + ve_2 + \eta n_0 + u\eta n_{01} + v\eta n_{02}, \\ \det J_f &= \langle (1-u-v)n_0 + un_1 + vn_2, e_1 + \eta n_{01}, e_2 + \eta n_{02} \rangle, \end{aligned}$$

where η is the variable corresponding to the thickness direction, and $n_{ij} = n_j - n_i$. Observe that $\det J_f(u,v,\eta_0)$ is linear in u,v for fixed η_0, the extrema will only be achieved at the corner points [?]. Therefore, it is sufficient to check the signs at three edges $(0,0,\eta)$, $(1,0,\eta)$, $(0,1,\eta)$ for $\eta \in [0,1]$.

$$\begin{aligned} &\det J_f(0,0,\eta) \\ &= \langle n_0, (1-\eta)e_1 + \eta(e_1 + n_1 - n_0), (1-\eta)e_2 + \eta(e_2 + n_2 - n_0) \rangle \\ &= (1-\eta)^2 \langle n_0, e_1, e_2 \rangle + (1-\eta)\eta \langle n_0, e_1, e_2 + n_2 \rangle \\ &+ (1-\eta)\eta \langle n_0, e_1 + n_1, e_2 \rangle + \eta^2 \langle n_0, e_1 + n_1, e_2 + n_2 \rangle \\ &= (1-\eta)^2 \mathrm{vol}_4 + (1-\eta)\eta(\mathrm{vol}_3 + \mathrm{vol}_1) + \eta^2 \mathrm{vol}_2 > 0. \end{aligned}$$

By symmetry, it is easy to verify that the following are positive,

$$\begin{aligned} \det J_f(1,0,\eta) &= (1-\eta)^2 \mathrm{vol}_6 + (1-\eta)\eta(\mathrm{vol}_7 + \mathrm{vol}_5) + \eta^2 \mathrm{vol}_8, \\ \det J_f(0,1,\eta) &= (1-\eta)^2 \mathrm{vol}_{11} + (1-\eta)\eta(\mathrm{vol}_9 + \mathrm{vol}_{10}) + \eta^2 \mathrm{vol}_{12}. \end{aligned}$$

A.5 Double Slab

In Section 3.3.1, we explain that we want to ensure that our shell validity is independent from the enumeration of the faces; for this reason, we require that the conditions I1 and I2 are satisfied for all possible decompositions. Without using the double slab, different decompositions of the prism will result in different intersections with the middle surface. In other words, the projection operator will project the input mesh to a different set of triangles on the middle surface. The double slab makes the middle surface independent of the decomposition by construction. We note that Theorem 3.6 is valid only for double-slab prisms since the statement requires that one prism contains only one triangle.

A.6 Unpublished Material

(Disclaimer: this section is unpublished material. And serves as post publish notes for future references.)

A.6.1 Pointless Variant

We extend the framework, by allowing the relaxation of definition of a section. Originally, any intersection between the triangle and the (different decomposition of the) prism is counted. Here, we explicitly forgive when the intersection is only one point on the vertex of mid surface and the triangle. This predicate can be easily implemented, making use of the bitwise equal coordinates. Therefore, the initial tracking section only follows edge-adjacency relationship, instead of the vertex adjacent one in the main text. With such adjusted condition for section, the optimization can be trivially accomodated. The only thing that is different (simpler) is the part for bevel, since we have less triangles to consider. The existing one is still valid, however, we can get away with simpler red-green intersection

A.6.2 Dynamic Intersection Check

To make easier the guarantee of a self-intersection-free shell, substitue the AABB tree collision check with a dynamic hashgrid. We initialize two hashgrids at the beginning, one for top surface and another one for bottom surface. This is followed by additional shrinking to make sure they are clear. Each local opeartion is responsible to trial and keep the validity of the surfaces. Furthermore, the top/bottom surface should not interfere with the reference input surface, this can be accelerated through the maintainance of the tracking list, and is trial checked before a local operation is performed.

A.6.3 Feature Snapping Shell

We can separate the definiition of reference surface and input surface to make a feature snapping shell.

A.6.4 Positive Detereminant for Natural Prism Map

This section follows and extends Appendix A.4 In fact, we observe that, for a quadratic bezier curve piece $at^2 + bt(1-t) + c(1-t)^2, t \in [0,1]$, the sufficient and necessary condition is $a > 0, c > 0, b > -2\sqrt{ac}$, and the last relation is equivalent to (but more clearly stated as) $b > 0 || b^2 - 4ac < 0$. The current text (all 12 positive tetra) impose an unnatural constraint on edge splits: in the case of an agressively positive prism (all decompositions are all-positive), sub-prism may not be positive in the same sense. While this will not break anything, forbidding edge split will lead to unexpected lower quality in the optimization procedure.

A.7 The geometric map is bijective

Consider a connected 3-dimensional compact manifold (curved) tetrahedral mesh $\mathcal{M} = \{\sigma_i\}, i = 1, \ldots, n$ with $\sigma_i = g_i(\hat{\tau})$, $\hat{\tau}$ a regular unit tetrahedron, $\det(J_{g_i}) > 0$ at all points including the

boundary, and σ_i and σ_j agree on a shared face. Let $\mathcal{M}_D$ be the domain obtained by copies of $\hat{\tau}$ and identifying $\hat{\tau}_i$ along common faces. We then define the map

$$\sigma \colon M_D \to \mathbb{R}^3,$$

by setting $\sigma|_{\hat{\tau}_i} = \sigma_i$.

Proposition A.1. *Suppose $\sigma|_{\partial \mathcal{M}_D}$ is injective. Then σ is injective on the whole domain $\mathcal{M}_D$.*

Proof. Our argument closely follow from [?, Appendix B] and [?, Theorem 1].

We consider point $y \in \mathbb{R}^3$ in general position, but not in the faces, edges, vertices, or any plane spanned by a linear face of $\mathcal{M}$. For each tetrahedron τ, we construct the map $\hat{\Psi}$ as a composition of $\sigma|_{\partial \hat{\tau}_i}$ (restricted to the triangular faces of the regular tetrahedron) and the projection map χ to the unit sphere centered around y.

We parametrize the image of $\sigma|_{\partial \hat{\tau}_i}$ as $x(u, v)$ on the projective plane (note that since $\det J_{g_i} > 0$, the image is a non-degenerate surface homeomorphic to a sphere). Since $x(u, v)$, and its normal field $n(u, v) = \frac{\partial}{\partial u} x \times \frac{\partial}{\partial v} x$ are polynomial functions, Consider the parametric curve $n(u, v) \cdot (x(u, v) - y) = 0$, the algebraic curve partitions the surface into finite number of patches, and on each patch the orientation of $\hat{\Psi}$ is constant. We can further triangulate the partitions to create a (abstract simplicial) complex.

Similar to [?, Appendix B], we count the number of pre-images of y, which equals to the degree in general positions,

$$\deg(\sigma)(y) = \sum_{i=1}^{n} \deg(\hat{\Psi}|_{\tau_i}) = \deg(\hat{\Psi}|_{\partial \mathcal{M}}).$$

Since $\sigma|_{\partial \mathcal{M}_D}$ is injective, and furthermore

$$\deg(\hat{\Psi}|_{\partial \mathcal{M}}) = \deg \chi|_{\partial \mathcal{M}} = \begin{cases} 1, & y \in \mathcal{M} \\ 0, & y \notin \mathcal{M}. \end{cases}$$

Thus we have shown the map is injective for the general positions for y, and it remains to be shown that the map is an open map, which follows from the same argument of [?, Lemma 2]. □

A.8 Local Operations

[?, Fig. 11] introduce a set of *valid* local operations to modify the shell, including edge split, edge collapse, edge flip and vertex smoothing. The operations are an analogue of the triangle mesh edit operations [?], by simultaneously editing the shared connectivity of bottom, middle and top surface of the shell. [?, Theorem 3.7] outlines invariant conditions, which maintains the shell projection to be bijective.

Our algorithm adopts and extends the local operations therein to the high order setting. In addition to the existing conditions, we also validate the curved volumetric mesh in the shell. The

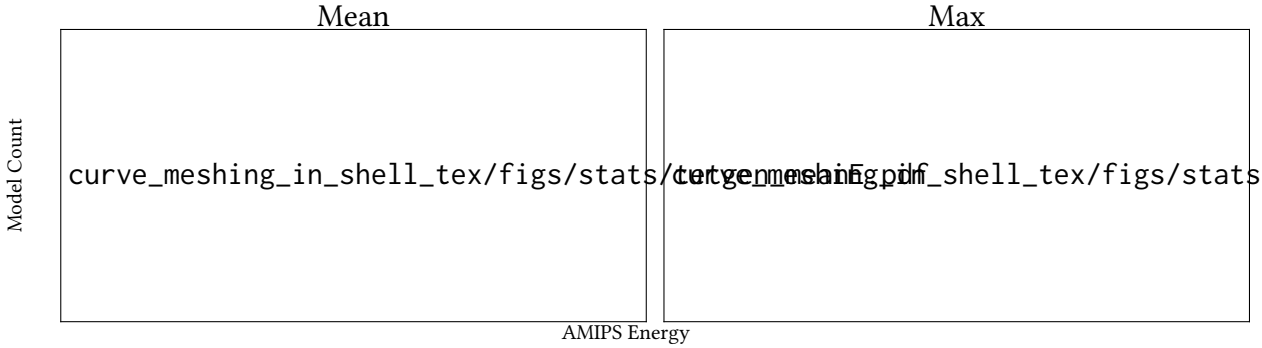


Figure A.2: Histogram of mean and maximum conformal AMIPS energy [?] of the output of our method and TetGen.

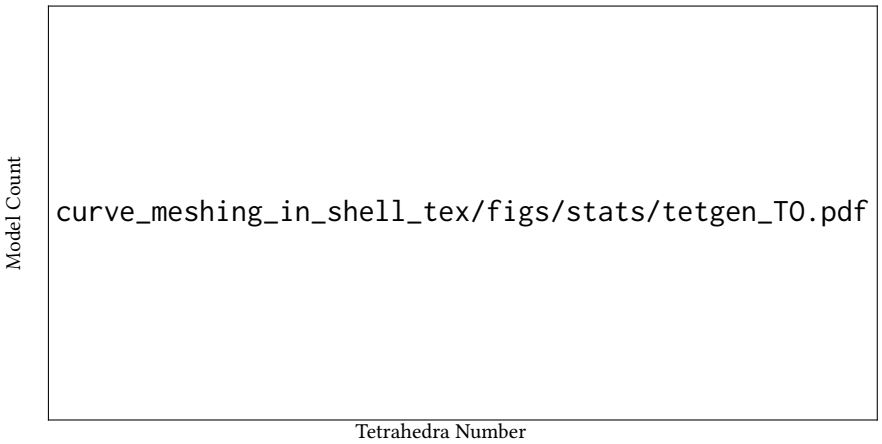


Figure A.3: Histogram of output tetrahedra number for TetGen and our method.

algorithm maintains the global intersection free bottom (top) surface with a dynamic hash grid [?]. Then for each prism, we check the positivity (defined by the determinant of Jacobian of the geometric map) of the prismatic element (each decomposed into three tetrahedra).

In the presence of feature annotation and feature straightening (Section 4.5) more care is taken to maintain the valid correspondence between the grouped feature chains and the curved edges: edge flip is disabled on the edges annotated as features; collapse is only allowed when it does not degenerate the chain, and the two endpoints of the edge is on the same chain. Since we require a map from the original input edges, we insert additional degree of freedoms in the input mesh. When performing edge split, the insertion of the new vertex is queried among the pre-image of the current edge, as an existing vertex of the input. For vertex smoothing (more specifically pan), the target location is limited to the set of input vertices.

A.9 Boundary preserving TetGen comparison

We compared our conforming tetrahedral meshing algorithm (Section 4.4.3) with TetGen on the linear shells (triangle meshes) of 3522 models from the Thingi10k dataset, giving each model sufficient computing resources (2 hours maximum running time and 32GB memory usage). Inheriting the robustness from TetWild, our method successfully processed all the inputs while preserving the triangulation, while TetGen fails on 224 models (215 models are not conforming, and 9 models have no output).

In Figure A.2, we show the average and maximum element quality of the output of our method and TetGen. Our method has a better average and maximum output quality than TetGen. Note that in the quality plot, the "tail" of TetGen's distribution is longer than ours. The maximum average energy of TetGen's output and ours are 3×10^8 and 3×10^5 respectively. The largest maximum energy of TetGen's output and ours are 3×10^{12} and 7×10^6 respectively. Our method generates denser output (Figure A.3), but our focus is on robustness instead of efficiency in this step.

A.10 TetWild Overview

Bibliography

Abgrall, R., Dobrzynski, C., and Froehly, A. (2012). A method for computing curved 2D and 3D meshes via the linear elasticity analogy: preliminary results. Research Report RR-8061.

Abgrall, R., Dobrzynski, C., and Froehly, A. (2014). A method for computing curved meshes via the linear elasticity analogy, application to fluid dynamics problems. *International Journal for Numerical Methods in Fluids*, 76(4):246–266.

Babuska, I. and Guo, B. Q. (1988). The h-p version of the finite element method for domains with curved boundaries. *SIAM Journal on Numerical Analysis*, 25(4):837–861.

Baumgart, B. G. (1972). Winged edge polyhedron representation. Technical report, Stanford, CA, USA.

Bern, M., Eppstein, D., and Gilbert, J. (1994). Provably good mesh generation. *Journal of Computer and System Sciences*, 48(3):384 – 409.

Bishop, C. J. (2016). Nonobtuse triangulations of pslgs. *Discrete & Computational Geometry*, 56(1):43–92.

Brisson, E. (1989). Representing geometric structures in d dimensions: Topology and order. In *Proceedings of the Fifth Annual Symposium on Computational Geometry*, SCG '89, page 218–227, New York, NY, USA. Association for Computing Machinery.

Canann, S. A., Muthukrishnan, S. N., and Phillips, R. K. (1996). Topological refinement procedures for triangular finite element meshes. *Engineering with Computers*, 12(3):243–255.

Cardoze, D., Cunha, A., Miller, G. L., Phillips, T., and Walkington, N. (2004). A bézier-based approach to unstructured moving meshes. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, pages 310–319, New York, NY, USA. ACM.

Cheng, X.-X., Fu, X.-M., Zhang, C., and Chai, S. (2019). Practical error-bounded remeshing by adaptive refinement. *Computers & Graphics*, 82:163 – 173.

DiCarlo, A., Paoluzzi, A., and Shapiro, V. (2014). Linear algebraic representation for topological structures. *Computer-Aided Design*, 46:269–274. 2013 SIAM Conference on Geometric and Physical Modeling.

Dobrzynski, C. and El Jannoun, G. (2017). High order mesh untangling for complex curved geometries. Research Report RR-9120.

Dougherty, R., Faber, V., and Murphy, M. (2004). Unflippable tetrahedral complexes. *Discrete & Computational Geometry*, 32(3):309–315.

George, P. and Borouchaki, H. (2012). Construction of tetrahedral meshes of degree two. *International Journal for Numerical Methods in Engineering*, 90(9):1156–1182.

Geuzaine, C., Johnen, A., Lambrechts, J., Remacle, J.-F., and Toulorge, T. (2015). *The Generation of Valid Curvilinear Meshes*, pages 15–39. Springer International Publishing, Cham.

Ghasemi, A., Taylor, L. K., and Newman, III, J. C. (2016). Massively parallel curved spectral/finite element mesh generation of industrial cad geometries in two and three dimensions. *Fluids Engineering Division Summer Meeting*, (50299).

Guibas, L. and Stolfi, J. (1985). Primitives for the manipulation of general subdivisions and the computation of voronoi. *ACM Trans. Graph.*, 4(2):74–123.

Hahmann, S. and Bonneau, G. . (2003). Polynomial surfaces interpolating arbitrary triangulations. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):99–109.

Jameson, A., Alonso, J., and McMullen, M. (2002). Application of a non-linear frequency domain solver to the euler and navier-stokes equations. In *40th AIAA Aerospace Sciences Meeting & Exhibit*.

Knupp, P. M. (2000). Achieving finite element mesh quality via optimization of the jacobian matrix norm and associated quantities. part i, a framework for surface mesh optimization. *International Journal for Numerical Methods in Engineering*, 48(3):401–420.

Mahmoud, A. H., Porumbescu, S. D., and Owens, J. D. (2021). Rxmesh: A gpu mesh data structure. *ACM Trans. Graph.*, 40(4).

Persson, P.-O. and Peraire, J. (2009). Curved mesh generation and mesh refinement using lagrangian solid mechanics. In *47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition*.

Poya, R., Sevilla, R., and Gil, A. J. (2016). A unified approach for a posteriori high-order curved mesh generation using solid mechanics. *Computational Mechanics*, 58(3):457–490.

Requicha, A. G. (1980). Representations for rigid solids: Theory, methods, and systems. *ACM Comput. Surv.*, 12(4):437–464.

Roca, X., Gargallo-Peiró, A., and Sarrate, J. (2012). Defining quality measures for high-order planar triangles and curved mesh generation. In Quadros, W. R., editor, *Proceedings of the 20th International Meshing Roundtable*, pages 365–383, Berlin, Heidelberg. Springer Berlin Heidelberg.

Sadek, E. A. (1980). A scheme for the automatic generation of triangular finite elements. *International Journal for Numerical Methods in Engineering*, 15(12):1813–1822.

Suwelack, S., Lukarski, D., Heuveline, V., Dillmann, R., and Speidel, S. (2013). Accurate surface embedding for higher order finite elements. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA'13, pages 187–192, New York, NY, USA. ACM.

Tong, W.-h. and Kim, T.-w. (2009). High-order approximation of implicit surfaces by g1 triangular spline surfaces. *Computer-Aided Design*, 41(6):441–455.

Xie, Z. Q., Sevilla, R., Hassan, O., and Morgan, K. (2013). The generation of arbitrary order curved meshes for 3d finite element analysis. *Computational Mechanics*, 51(3):361–374.

Yerry, M. A. and Shephard, M. S. (1983). A modified quadtree approach to finite element mesh generation. *IEEE Computer Graphics and Applications*, 3(1):39–46.

Zayer, R., Steinberger, M., and Seidel, H.-P. (2017). A gpu-adapted structure for unstructured grids. *Comput. Graph. Forum*, 36(2):495–507.