TU Delft

Delft University of Technology

Document Version
Final published version

Citation (APA)
Suriyababu, V. K. (2026). *Robust and Automated Geometry Processing Workflows for Engineering Applications*. [Dissertation (TU Delft), Delft University of Technology].

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
In case the licence states “Dutch Copyright Act (Article 25fa)”, this publication was made available Green Open Access via the TU Delft Institutional Repository pursuant to Dutch Copyright Act (Article 25fa, the Taverne amendment). This provision does not affect copyright ownership.
Unless copyright is transferred by contract or statute, it remains with the copyright holder.

Sharing and reuse
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.

Robust and Automated Geometry Processing Workflows for Engineering Applications

Vijai Kumar Suriyababu

Robust and Automated Geometry Processing Workflows for Engineering Applications

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus, Prof.dr.ir. H. Bijl,
chair of the Board for Doctorates
to be defended publicly on
Wednesday, 1 July 2026 at 10:00 A.M.

by

Vijai Kumar SURIYABABU

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus	chairperson
Dr. rer. nat. M. Möller	Delft University of Technology, promotor
Prof. dr. ir. C. Vuik	Delft University of Technology, promotor

Independent members:

Prof. Dr.-Ing. habil S. Hickel	Delft University of Technology
Prof. dr. D. Moxey	King's College London, United Kingdom
Prof. dr. Z.J. Wang	Kansas University, USA
Dr. K.A. Hildebrandt	Delft University of Technology
Dr. L. Nan	Delft University of Technology
Prof. dr. ir. M.B. van Gijzen	Delft University of Technology, *reserve member*

This dissertation was partly funded from 2019 to 2022 by the Austrian Research Promotion Agency (FFG). The remaining research was self-funded.

Keywords: robust geometry processing; mesh repair; point clouds; shrink wrapping; implicit meshing; industrial pipelines

Cover by: Vijai Kumar Suriyababu

Copyright © 2026 by Vijai Kumar Suriyababu

An electronic copy of this dissertation is available at https://repository.tudelft.nl/.

Contents

List of Figures

List of Tables

List of Algorithms

List of Code Listings

Summary

Modern computer-aided engineering (CAE) depends on discrete 3D data such as meshes and point clouds, yet raw geometric models are rarely ready for direct use. Industrial CAD assemblies and scans often contain holes, topological ambiguities, irregular sampling, and feature loss that can disrupt downstream simulation and design workflows. This thesis addresses these practical gaps by developing multiple geometry-processing workflows that prioritize robustness, automation, and reproducibility.

One contribution focuses on topological queries and cleanup. A curvature-aware, low-parameter approach detects hole boundaries and supports through-hole removal when needed. The same chapter also presents extraction of negative-volume regions as a separate topology-processing task. Independently, the thesis introduces Series of Local Triangulations (SOLT), an intermediate representation for stable resampling of non-uniform point clouds into uniform, high-fidelity sets while preserving geometric detail, and explores a preliminary application to CAD-free higher-order surface reconstruction.

For simulation-ready models, the thesis proposes a shrink-wrap mesh generation workflow on octrees that combines distance fields and adaptive morphology. This framework enables reliable topology simplification, controlled genus reduction, and robust wrap mesh generation for complex real-world geometries. In a separate application-focused workflow, the thesis also presents a heat-method-based algorithm for mean camber line extraction and structured multiblock decomposition, offering a practical and implementation-friendly alternative to traditional geometric constructions.

These algorithmic contributions culminate in *tgLang*, a strongly typed domain-specific programming language for scientific computing and geometry processing. By elevating geometric entities to first-class language constructs and coupling them with deterministic execution, runtime support, and native deployment, *tgLang* transforms complex mesh workflows into concise and reproducible programs. Together, the methods and language presented in this thesis provide a flexible toolbox that guides imperfect input geometry toward reliable, application-ready computational models.

Samenvatting

Moderne computer-aided engineering (CAE) steunt op discrete 3D-data zoals meshes en puntenwolken, maar ruwe geometrische modellen zijn zelden direct bruikbaar. Industriële CAD-assemblies en scans bevatten vaak gaten, topologische ambiguiteiten, onregelmatige sampling en verlies van features, wat downstream simulatie- en ontwerpprocessen kan verstoren. Dit proefschrift pakt deze praktische knelpunten aan door meerdere geometrieverwerkingspijplijnen en workflows te ontwikkelen die robuustheid, automatisering en reproduceerbaarheid vooropstellen.

Een bijdrage richt zich op topologische analyse en opschoning. Een krommingsbewuste aanpak met weinig parameters detecteert randen van gaten en ondersteunt waar nodig de verwijdering van doorlopende gaten. Hetzelfde hoofdstuk presenteert daarnaast extractie van negatieve-volumegebieden als een aparte topologische verwerkingstaak. Onafhankelijk daarvan introduceert het proefschrift Series of Local Triangulations (SOLT), een tussenrepresentatie voor stabiele resampling van niet-uniforme puntenwolken naar uniforme, hoogwaardige puntensets met behoud van geometrisch detail, en verkent een voorlopige toepassing voor CAD-vrije hogerorde-oppervlaktereconstructie.

Voor simulatieklare modellen stelt het proefschrift een shrink-wrap meshgeneratieworkflow op octrees voor, die afstandsvelden en adaptieve morfologie combineert. Dit raamwerk maakt betrouwbare topologieversimpeling, gecontroleerde genusreductie en robuuste wrap-meshgeneratie voor complexe industriële geometrie mogelijk. In een aparte, toepassingsgerichte workflow presenteert het proefschrift daarnaast een op de heat method gebaseerd algoritme voor mean camber lineextractie en gestructureerde multiblock decompositie, als een praktische en implementatievriendelijke alternatieve route ten opzichte van traditionele geometrische constructies.

Deze algoritmische bijdragen komen samen in *tgLang*, een sterk getypeerde domeinspecifieke programmeertaal voor wetenschappelijk rekenen en geometrieverwerking. Door geometrische entiteiten als first class taalconstructies te behandelen en die te koppelen aan deterministische uitvoering en high-performance backends, maakt *tgLang* complexe mesh-workflows beknopt en reproduceerbaar. Samen bieden de methoden en taal in dit proefschrift een flexibele gereedschapskist die onvolmaakte industriële geometrie richting betrouwbare, toepassingsklare rekenmodellen leidt.

1

Introduction

"Where there is matter, there is geometry."
Johannes Kepler

"Geometry is the art of correct reasoning on incorrect figures."
George Polya

"The purpose of computing is insight, not numbers."
Richard Hamming

1.1. Motivation and Context

Over the past decades, 3D data has become central to industries ranging from computer-aided design and manufacturing to biomedical imaging and virtual reality. Modern scanners and simulation pipelines routinely generate point clouds and surface meshes comprising millions of elements. Moreover, consumer-grade devices costing less than 1,000 EUR can now capture entire commercial vehicles with reasonable accuracy. To make these datasets useful, geometry-processing algorithms are required to clean, reconstruct, analyse, recognise, and manipulate complex shapes in an automated fashion. Whether repairing holes in a scanned model for additive manufacturing, extracting cavities for computational fluid dynamics (CFD), or defining high-level, domain-specific abstractions for shape operators, geometry processing in engineering practice lies at the heart of contemporary digital design workflows.

Despite major advances in foundational algorithms, several persistent challenges remain. Raw data are often noisy, incomplete, or corrupted; the underlying topology may be highly complex or unknown; and high-order surface features can be lost during resampling or smoothing. Tailoring a mesh-processing pipeline to a specific engineering application often requires considerable engineering effort. While many algorithms are evaluated on large, crowdsourced datasets exhibiting diverse defects,

a trend often referred to as testing "in the wild", these benchmarks are not always fully representative of the specific complexities arising from real-world CAD models or engineering scans. To distinguish our focus from methods designed for general-purpose repositories, we adopt the term application-oriented geometry processing throughout this thesis.

This thesis addresses these practical challenges through a set of complementary contributions. The work does not follow one rigid end-to-end pipeline. Instead, it develops several focused workflows for topology processing, point-cloud resampling, simulation-oriented mesh generation, domain-specific feature extraction, and a dedicated programming language (*tgLang*) for reproducible geometry-processing workflows.

1.2. Core Challenges in Mesh and Surface Processing

Real-world geometry-processing tasks must contend with a variety of interrelated difficulties that motivate the work in this thesis:

1. **Topological Uncertainty and Defects:** Scanned meshes frequently contain holes, spurious handles, or inverted elements. Detecting and repairing these defects robustly, with limited parameter dependence, is important for downstream tasks, such as Boolean operations or finite-element analysis.
2. **Unstructured Data to High-Quality Meshes:** Point clouds and unorganised triangle soups lack connectivity. Converting them into manifold surfaces that accurately capture intricate geometric features, while avoiding oversmoothing or excessive mesh complexity, remains an open problem.
3. **Feature Preservation during Resampling:** Many remeshing methods, including isotropic subdivision or smoothing-based approaches, tend to obliterate sharp creases, ridges, or fine-scale details that are semantically important in engineering and cultural-heritage applications.
4. **Application-Driven Mesh Generation:** Different applications impose different surface requirements (e.g., watertightness for CFD vs. minimal element count for real-time rendering). Designing a single remeshing framework that can be tuned across these diverse domains is challenging.
5. **Bridging Algorithmic and Programming Workflows:** Even with state-of-the-art algorithms, integrating them into end-user workflows often involves error-prone scripting or low-level coding in C++ libraries. A higher-level language tailored to geometry-processing abstractions can reduce friction in development, prototyping, and deployment.

1.3. Contributions and Thesis Outline

The remainder of this thesis is organised around a set of connected but distinct workflows. Each chapter addresses one of the core challenges identified previously and presents a corresponding contribution.

Chapter 2 addresses the foundational problem of topological cleanup. In response to the challenge of topological uncertainty and defects, this chapter presents our first contribution: a fast, curvature-adaptive algorithm for detecting hole boundaries, along with related topology-processing operations such as through-hole removal and extraction of negative-volume regions.

Chapter 3 tackles the challenge of generating high-quality surface representations from unstructured data. Here, we introduce a resampling workflow that converts non-uniform point clouds into uniform ones via an intermediate mesh representation called SOLT. This approach preserves fine geometric detail and also motivates a preliminary application to CAD-free higher-order surface reconstruction.

Chapter 4 introduces a robust, application-driven method for generating simulation-ready meshes, addressing the challenges of feature preservation and application-specific requirements. This chapter details our third contribution: a shrink-wrap mesh generation workflow that combines distance fields with adaptive morphology on octree-based volumetric representations. This technique is useful for preparing complex assemblies for analysis while adapting to varying levels of topological complexity.

In a specialised application workflow, **Chapter 5** demonstrates how distance-field methods can be adapted for domain-specific feature extraction. As a fourth contribution, this chapter proposes a PDE-inspired approach to compute smooth, geometrically faithful mean camber lines from aerodynamic profiles, facilitating automated airfoil analysis and parameterisation.

Finally, **Chapter 6** addresses the challenge of bridging the gap between algorithms and practical workflows. This chapter presents the systems contribution of the thesis, *tgLang*, a domain-specific language for geometry processing. It is designed to abstract away low-level implementation details and to unify geometry-processing primitives within a concise, deterministic programming model, enabling users to prototype and deploy complex workflows more effectively.

2

Topological Hole Detection and Negative Volume Extraction in Surface Meshes

Abstract. We present a simple and fast algorithm for identifying and extracting boundary loops of holes in discrete 2-manifold surfaces embedded in 3D Euclidean space. This work addresses intentionally created "through-holes" or "tunnels" in a geometry, as opposed to incidental holes caused by missing triangles in the mesh. The algorithm detects these features by operating directly on the input geometry, without requiring any simplified proxy or volumetric representation. We use discrete Gaussian curvature to approximate the local surface behaviour, which allows for the efficient filtering of edges that are unlikely to form a hole boundary. While this chapter primarily illustrates the method on triangulated surfaces, the underlying data structures and principles are directly applicable to meshes with mixed polygonal elements. The chapter also demonstrates how to extract internal negative volumes by leveraging the shrink wrap mesh generation algorithm (detailed in later chapters), a useful step for applications like computational fluid dynamics.

This chapter is based on: V.K. Suriyababu and C. Vuik. "A Simple and Fast Hole Detection Algorithm for Triangulated Surfaces." Journal of Computing and Information Science in Engineering, 21(4):044502, 2021. doi:10.1115/1.4049030

The negative volume extraction section is based on: V.K. Suriyababu, C. Vuik, and M. Möller. "Shrink Wrap Mesh Generation Using Morphological Operators with Selected Applications." Zenodo, *2022.* https://doi.org/10.5281/zenodo.6562410

2.1. Topological Hole Detection

The problem of hole detection and filling has long been a central topic in geometry processing and mesh repair. Many numerical simulations, for instance, rely on simplified geometries that require extensive manual defeaturing. For models with hundreds or thousands of holes, this process can be a daunting manual task [3]. Automated hole detection is therefore essential for applications such as preparing models for computational fluid dynamics (CFD), where holes may define the inlets and outlets of the simulation domain. Furthermore, identifying and closing all holes is critical for volumetric mesh generation, which typically requires a closed, "watertight" input geometry to prevent leaking [4].

Several methods in topology processing share similarities with our work. Doraiswamy et al. [5] proposed computing the Reeb graph of a discrete surface by solving for a scalar field to identify critical points. While such algorithms are robust for computing the genus of a surface, they can be sensitive to small surface perturbations, potentially leading to an excessive number of topological features. They are also computationally expensive compared to the methods proposed in this chapter. Other approaches, such as the one by Borrell et al. [6], rely on a voxelized representation of the mesh to compute its genus. This makes it impossible to extract the exact geometry of the hole boundary from the original surface. Although these methods are related to topological analysis, their primary goal is not the precise geometric identification of holes.

In the domain of feature detection from point clouds, Smereka et al. [7] used a modified Hough transform to detect circular objects. This is achieved by classifying points as inliers or outliers to a parametric circle, providing an approximation of point clusters that could form circles or cylinders. To the best of our knowledge, the only works that directly attempt to solve the problem of detecting holes in discrete surfaces are those proposed by Wang et al. [8] and Lozano-Durán et al. [6]. These methods rely on triangle coplanarity to merge triangles into planes, which are then clustered to extract hole contours. This approach works well for surfaces with some inherent planarity but can generate too many planes on complex industrial models, leading the authors to suggest surface segmentation as a potential remedy.

Our work differs by relying only on two simple geometric criteria: feature angle and discrete Gaussian curvature. The algorithm is designed to avoid the removal of topological elements, making most of its stages fully parallelizable. It requires no voxelization or statistical approximation and is capable of extracting the accurate geometry of the detected holes. However, our algorithm is not a replacement for the topological approaches mentioned above; rather, it is complementary. Due to its semi-heuristic nature, our method can occasionally over- or under-predict the number of holes, as shown in the Experiments section. We see its primary appli-

cation as a fast and direct feature detector, which can be complemented by more exhaustive methods like Reeb graphs, especially when information about internal voids is also required.

The algorithm was developed using a top-down approach. We began with a brute-force method that searched the entire geometry and progressively added geometric filters to reduce the search space. The key filters are the feature angle and the discrete Gaussian curvature, making the final algorithm semi-heuristic. This chapter presents two primary algorithms: one for solids and one for sheets.

The general workflow is as follows: First, the input mesh is loaded into the half-edge data structure. Next, we identify an initial set of candidate edges by computing the feature angle for every edge in the mesh. Edges whose feature angle falls within our specified range are marked. Then, as a second filtering step, the Gaussian curvature is computed at the vertices of these candidate edges. Edges associated with purely elliptic regions are pruned from the candidate set. Finally, we search for closed loops within this greatly reduced search space.

2.1.1. Essential Definitions and Nomenclature

We begin by defining the basic elements of a discrete surface. The algorithms proposed are valid for any polygonal surface, but for simplicity, our discussion is restricted to triangulated surfaces due to their prevalence in practice. A surface is composed of faces (triangles), which are in turn defined by vertices and edges. The ordering of vertices within a face (e.g., clockwise or counter-clockwise) determines the surface orientation, and it is imperative that this orientation is consistent across the entire mesh.

All algorithms presented here are built upon the half-edge mesh data structure. Specifically, we use the implementation from OpenMesh [9], a robust open-source library. We chose OpenMesh over alternatives like CGAL [10] because CGAL enforces strict 2-manifold properties, preventing the representation of non-manifold vertices or edges. OpenMesh provides greater flexibility, allowing the developer to handle such configurations as needed. Furthermore, OpenMesh is released under a more permissive license and has no external dependencies.

2.1.1.1. Half-Edge Data Structure

The half-edge data structure represents each edge of a mesh as a pair of directed half-edges. This allows each face to own a unique set of boundary half-edges, simplifying traversal and local queries. As shown in Figure 1, each half-edge stores a reference to its origin vertex and its pair (the other half-edge making up the full edge). This structure makes it trivial to find adjacent faces. Each vertex, in turn, stores a reference to one of its outgoing half-edges. This basic structure supports iterators and circulators for tasks like visiting all faces around a vertex. Most implementations

also allow custom data properties to be stored on any mesh entity (vertices, edges, faces, or half-edges).

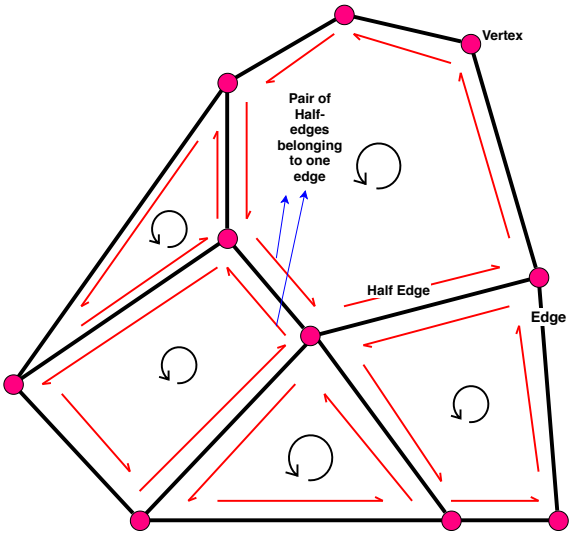


Figure 1: A schematic of the half-edge data structure on a polygonal mesh.

2.1.1.2. Discrete Gaussian Curvature

While curvature is formally defined for continuous surfaces, discrete equivalents are essential for processing CAD and graphics models [11]. The Gaussian curvature K_V at a vertex V on a discrete surface is defined by the angle defect [11]:

$$K_V = 2\pi - \sum_i \theta_i$$

where θ_i are the angles of the incident faces at vertex V. The sign of K_V classifies the local shape at the vertex:

Gaussian Curvature (K_V)	Local Surface Classification
> 0	Elliptic (sphere-like)
< 0	Hyperbolic (saddle-like)
$= 0$	Parabolic (planar or cylindrical)

Table 1: Classification of local surface geometry based on discrete Gaussian curvature.

We use the discrete Gaussian curvature as an initial filtering metric to accelerate the algorithm. In the CAD-like test geometries used to develop the method, edges forming through-hole boundaries were consistently adjacent to vertices classified as

hyperbolic or parabolic by the sign of their angle-defect curvature. This observation is empirical rather than a topological guarantee: it is used to reduce the search space before loop extraction, not to prove that all possible holes will be retained. Therefore, we filter our search space to include only edges connected to such vertices, effectively removing large planar and elliptic regions. This significantly reduces the number of candidate edges that must be processed in later stages.

2.1.1.3. Feature Angle

The primary filtering criterion for our algorithm is the feature angle. The feature angle is the dihedral angle between two adjacent polygons, measured across their shared edge. Edges that form the sharp corner of a drilled or cut-through hole typically have a feature angle of approximately 90 degrees. However, to account for variations in real-world meshes, we use a more conservative range of 30 to 120 degrees. While this range can be adjusted, we found it to be effective for all of our test cases. Widening this range increases the size of the search space and slows down the algorithm, while narrowing it may cause legitimate holes to be missed. Conversely, rounded fillets and chamfers can move the local feature angle away from the ideal sharp-edge case or introduce additional sharp loops that satisfy the same criterion. These cases are therefore treated as expected limitations of the heuristic and are revisited in the experimental discussion.

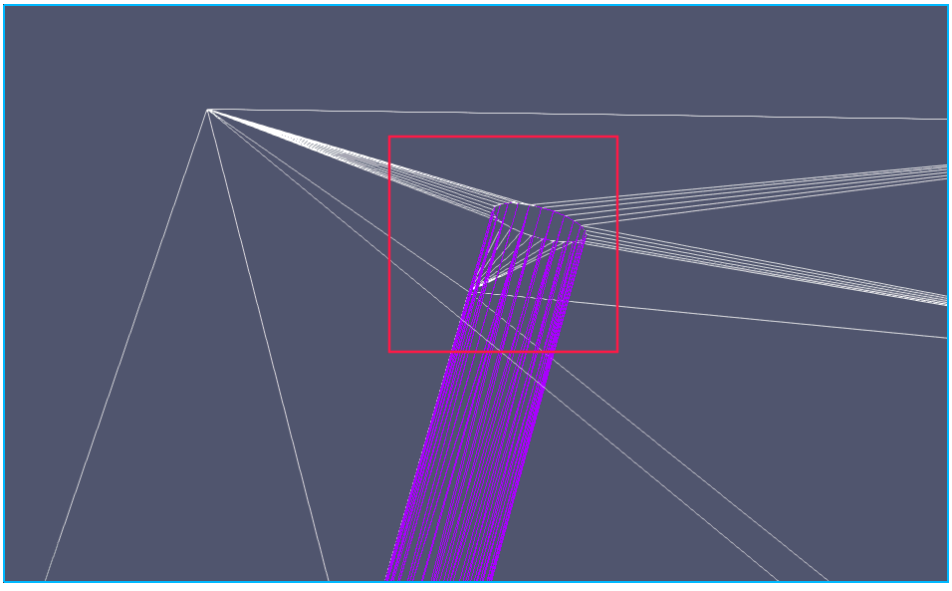


Figure 2: The feature angle is the angle between the normals of adjacent faces (e.g., the pink and white faces).

2.1.1.4. Hole

In mesh processing literature, the term "hole" typically refers to missing polygons in a discrete surface, which create a boundary where there should be none. Numerous algorithms exist for detecting and closing these types of holes [12], [13]. This chapter, however, does not focus on such defects. We address holes that are intentionally created geometric features, such as tunnels, passages, or drilled holes. Our work is therefore more closely related to the problem of detecting the genus of a surface or identifying closed-loop features.

2.1.2. Algorithm for Solids

2.1.2.1. Exhaustive Brute Force Search Algorithm

We begin with a naive brute-force approach that searches the entire geometry. This algorithm explores an exhaustive search space and represents the most straightforward method for detecting holes in discrete surfaces. Its runtime is poor compared to the improved algorithm presented next and is therefore not viable for practical CAD applications. There is no meaningful way to improve its asymptotic runtime; any tuning is limited to adjusting the feature angle threshold. The method also produces a large number of false positives, requiring multiple heuristic filters to make the output usable.

Hole Detection in Solids (Naive Exhaustive Search)

Input: Discrete surface (solid)
Result: A vector of chains (every chain is a closed loop of edges)

```
Initialize surface into a half-edge data structure
Create an empty vector possible_edges to store edges
For all faces of the surface:
    Get the three surrounding faces
    Compute the feature angle between the current face and its neighbours
    If the feature angle matches a specific feature angle criterion:
        Add the edge to possible_edges
Loop through possible_edges and form chains
Return vector of chains
```

Algorithm 1: A brute-force baseline algorithm without Gaussian curvature filtering.

2.1.2.2. Gaussian Filtered Search Algorithm

To address the performance limitations of the brute-force approach, we introduce geometric filters to reduce the search space. This is our fastest and most robust

algorithm for closed, solid geometries. It is efficient at detecting most holes but can fail on poorly triangulated models where feature angles are inconsistent. Examples of such cases are shown in the Experiments section.

Hole Detection in Solids (Gaussian Filtered Search)

Input: Discrete surface (solid)
Result: A vector of chains (every chain is a closed loop of edges)

```
Initialize surface into a half-edge data structure
Create an empty vector possible_edges to store edges
For all faces of the surface:
    Get the three surrounding faces
    Compute the feature angle between the current face and its neighbours
    If the feature angle matches a specific feature angle criterion:
        Add the edge to possible_edges
For all edges in possible_edges:
    Compute the discrete Gaussian curvature
For all edges in possible_edges:
    If the edge is not possibly part of a hyperboloid or cylinder:
        Remove the edge from possible_edges
While not all edges in possible_edges have valence equal to 2:
    Remove edges which do not have the matching valence
Loop through possible_edges and form chains
Return vector of chains
```

Algorithm 2: The primary algorithm for detecting holes in solid geometries using Gaussian curvature filtering.

2.1.3. Algorithm for Sheets

Detecting holes in sheets (open surfaces) is simpler than in solids. The problem reduces to identifying all boundary edges and grouping them into loops. In a half-edge data structure, a boundary half-edge is one whose pair points to itself. Unlike the solid case, this structural property is sufficient on its own: no geometric filtering based on feature angle or discrete Gaussian curvature is required, since boundary half-edges already unambiguously identify hole edges by construction. The algorithm collects all such half-edges and connects them to form loops.

Once the loops are formed, a pruning step can be applied to distinguish structurally meaningful hole boundaries from spurious loops caused by a few missing triangles. This is achieved by filtering loops based on their approximate perimeter or the

diameter of their minimal enclosing ball: loops whose size falls below a threshold are discarded as mesh artifacts, while larger loops are retained as genuine features.

Hole Detection in Sheets

Input: Discrete surface (sheet)
Result: A vector of edge chains representing boundary loops

```
Initialize surface into a half-edge data structure
Create an empty set boundary_edges
For all half-edges h in the mesh:
    If h is a boundary half-edge (h.pair() == h):
        Add the full edge corresponding to h to boundary_edges
Form closed loops (chains) from the edges in boundary_edges
Return vector of edge chains
```

Algorithm 3: Hole detection in sheet geometries.

2.1.4. Formation of Chains (Closed Loops)

After filtering, the remaining candidate edges are connected to form closed loops. The algorithm starts with an arbitrary, unvisited edge from the candidate set and traverses to its neighbors, collecting connected edges until it returns to the starting edge. The half-edge data structure makes this traversal trivial. If a path does not form a closed loop (i.e., it terminates at a dead end), it is discarded. The process repeats until all candidate edges have been visited and assigned to a loop.

2.1.5. Approximate Diameter and Removal

Once a hole loop has been identified, we compute its approximate diameter using a minimal enclosing ball algorithm [14]. This allows for quick matching of hole pairs (e.g., the inlet and outlet of a tunnel) based on similar radii. A detected through-hole can be removed by deleting the faces forming the tunnel between the paired loops and re-triangulating the openings.

2.1.6. Experiments and Limitations

We tested the hole detection algorithm on various sheet and solid geometries with different degrees of complexity. The implementation uses C++11, with Python for visualization. All benchmarks were run on a single core of an Intel Core i7-8700K CPU @ 3.70GHz with 64 GB of RAM. The results presented are for the fast, Gaussian-filtered search algorithm. The combined runtime for all geometries shown in Figure 3 was less than 5 seconds.

The algorithm's runtime scales linearly with the number of input triangles, as shown in Figure 4. We compared the Gaussian-filtered search algorithm against a naive brute-force counterpart, which only uses the feature angle criterion. As expected, the filtered algorithm is substantially more scalable, whereas the brute-force approach grows much more rapidly and fails on complex cases. This analysis demonstrates the practical feasibility of our filtered search approach. The runtime can be further improved via parallelization; all stages of the algorithm, except for the final loop formation, are embarrassingly parallel and could achieve significant speedup with libraries like OpenMP.

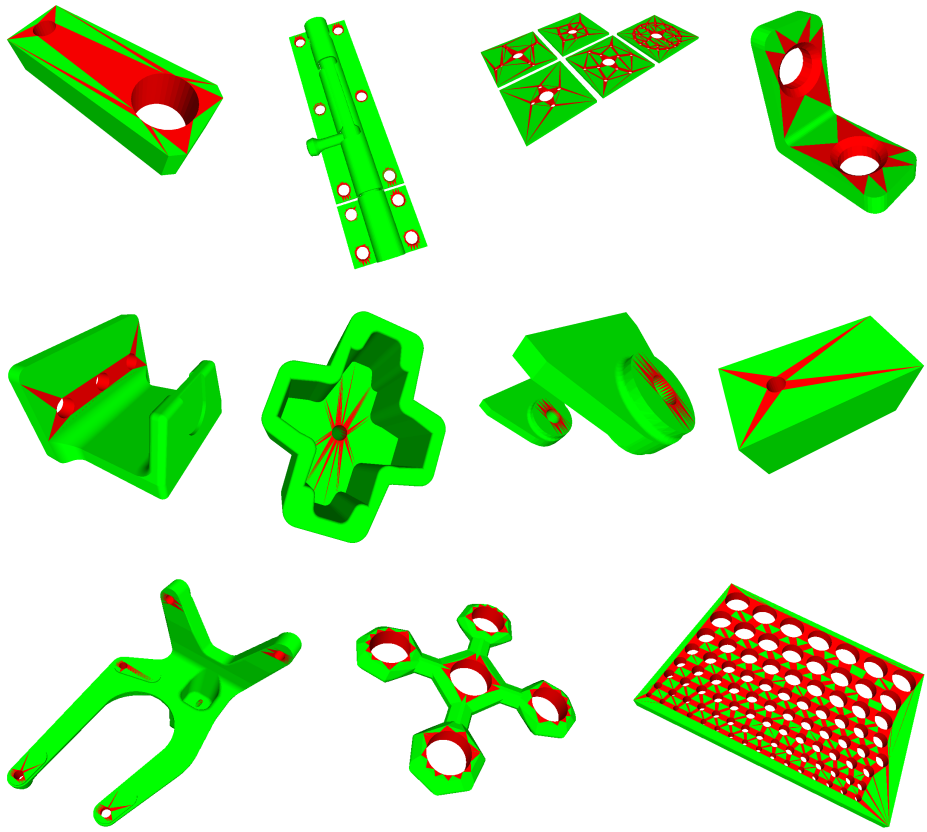


Figure 3: Holes detected by the algorithm in different geometries. Faces forming the hole boundaries are colored red.

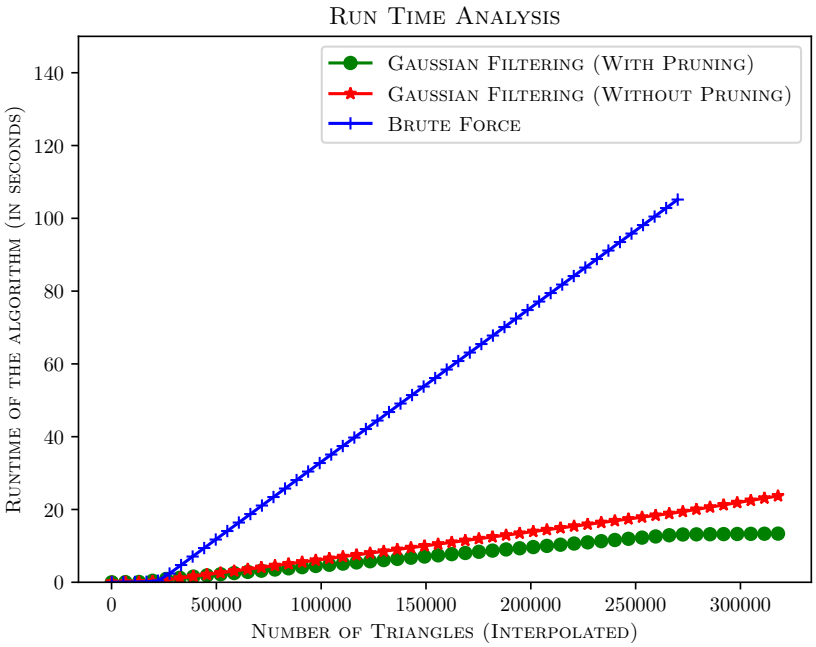


Figure 4: Runtime analysis. The brute-force algorithm fails or gives erroneous results for larger test cases.

The algorithm succeeds in detecting most holes but can sometimes over- or under-predict. The Table 2 shows the number of holes detected by the algorithm compared to the actual number for various geometries.

Geometry	Actual Holes	Detected Holes
Cuboid with holes	2	2
Door Latch	24	31
Multiple Cuboids with Holes	124	124
Cylindrical Sheet	2	2
Fidget Spinner	10	10
L Bracket	4	6
Camera mounting bracket	6	6
Geometry with cylindrical hole	2	2
Twin mounts	4	4
Cuboid with one hole	2	2
Gear with holes	3	3
Slab with multiple holes	112	112

Table 2: Comparison of actual vs. detected holes for different geometries.

The algorithm can fail in cases of poor triangulation, where feature angles become inconsistent. Scanned geometries, for example, often lack smooth curvature and sharp features. Figure 5 shows a simple case where protruding, non-planar triangles disrupt the feature angle around a hole. In such scenarios, one might expand the feature angle range, but the more robust solution is to repair the mesh to remove such artifacts. These defects are common in industrial models and often require preprocessing in a dedicated CAD cleanup tool. Over-prediction can also occur when other sharp features, like fillets or chamfers, are coincidentally within the feature angle criteria. These extra detected loops can be presented to the end-user for interactive deletion.

As a fallback, if the solid algorithm fails to detect any holes in a given geometry, we run the sheet algorithm on the same input to check for boundary loops before terminating.

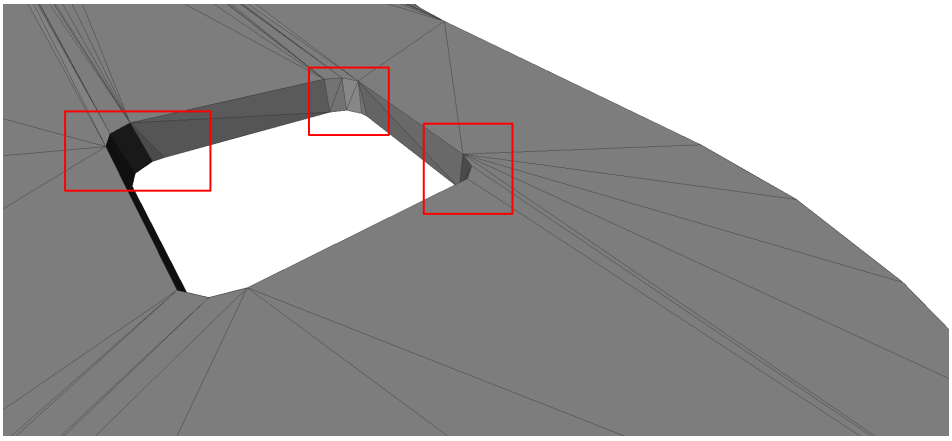


Figure 5: A hole with protruding faces that disrupt the feature angle consistency.

2.1.7. Implementation in tgLang

The original prototype for hole detection was implemented in C++ to prioritize performance. The workflow was later reimplemented in *tgLang* to make the algorithm easier to understand and reproduce, while keeping the processing pipeline concise. To keep the chapter focused on the methodology rather than on implementation details, the *tgLang* code listing is provided in Appendix A.

2.2. Negative Volume Extraction

Sometimes, the feature of interest is not the individual topological holes, but rather the entire fluid or negative volume. Estimating this volume by identifying holes one-by-one can be computationally expensive. While it may be easy to determine the inside and outside of a simple solid object, defining a negative volume is not straightforward, as all openings must be correctly identified to extract the entire volume. We have simplified this process by using our shrink-wrap mesh generation algorithm, detailed in Chapter 4. The present section should therefore be read as an application of the shrink-wrap construction rather than as an independent volume-reconstruction algorithm.

For an industrial practitioner interested in extracting the fluid volume of a car's interior, for example, the process can be fully automated. This requires a clean, genus-zero outer shell of the object, which can be obtained using our shrink-wrapping algorithm. Once this outer shell is available, the negative volume can be extracted using simple boolean operations. The workflow is as follows:

1. Subtract the genus-zero shrink-wrapped geometry from the original input geometry.
2. Split the resulting mesh based on connectivity into its constituent components.
3. The component with the largest volume is the desired internal negative volume.

For the boolean operations required in this process, we used existing CGAL functionality [15].

The robustness of this procedure depends primarily on the shrink-wrapped outer shell. If the shell is genus-zero, watertight, and fully encloses the original model, the boolean subtraction produces a connected set of candidate void regions that can be separated by connectivity. Its geometric accuracy is bounded by the shrink-wrap resolution and offset parameters discussed in Chapter 4: a coarser wrap gives a more conservative but less accurate approximation of the true fluid boundary, while a finer wrap follows the input more closely but may preserve small openings or defects. Consequently, this workflow is best suited to applications where an approximate interior volume or particle distribution is sufficient, such as preprocessing for meshless simulations, rather than as a substitute for a validated CAD-derived fluid domain.

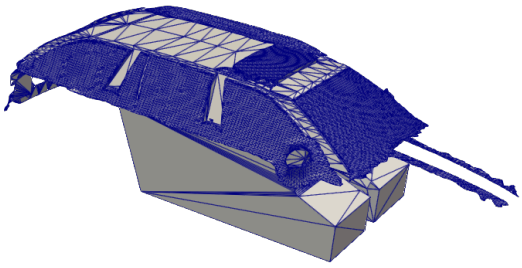


Figure 7: Wrapped Car

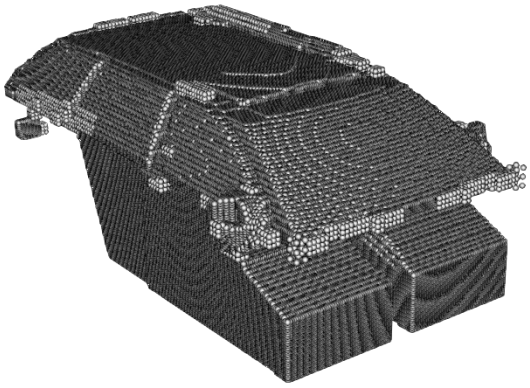


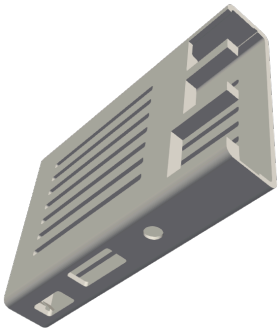
Figure 8: Car fluid volume as point cloud

Figure 6: A partial car geometry that has been shrink-wrapped and its fluid volume extracted as a volumetric point cloud. Such point clouds are used as particle distributions for meshless methods like SPH.

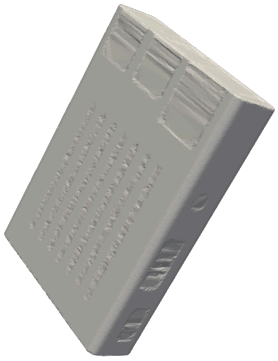
This technique is particularly useful for preparing models for meshless simulation methods like Smoothed Particle Hydrodynamics (SPH), which require a volumetric

point distribution. In Figure 6, we show a partial car geometry that has been shrink-wrapped. Its internal fluid volume is then extracted and represented as a particle distribution for an SPH simulation.

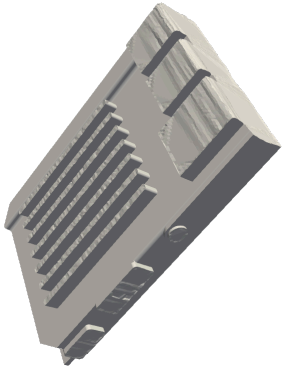
In a second example, Figure 9 shows a Raspberry Pi case, its shrink-wrapped representation, and the subsequently extracted fluid volume. This process facilitates the creation of a clean fluid volume, which can be further processed if needed.



Raspberry Pi case geometry



Wrapped Geometry



Fluid volume

,

Figure 9: A Raspberry Pi case and its extracted internal fluid volume.

3

Structure-from-Unstructured: SOLT for Point Cloud Resampling and Surface Reconstruction

Abstract. The increasing reliance on 3D scanning and meshless methods highlights the need for algorithms optimized for point cloud representations in CAE simulations. While voxel-based binning methods are simple, they often compromise geometric and topological integrity, particularly with coarse voxelizations. We propose an algorithm based on a **Series of Local Triangulations** (SOLT) as an intermediate representation for point clouds, enabling efficient upsampling and downsampling. This robust and straightforward approach preserves local geometric structure during resampling and reduces feature loss and topological distortion in the tested cases. This chapter also explores a preliminary application of the SOLT philosophy to CAD-free higher-order surface reconstruction using weighted Least Squares (LS) fittings on the intermediate structure. The proposed techniques are designed to fit into engineering preprocessing workflows by avoiding complex optimization or machine learning methods while delivering reliable results for point cloud resampling and demonstrating a possible route toward CAD-free higher-order surface reconstruction.

This chapter is based on: V.K. Suriyababu, C. Vuik, and M. Möller. "Resampling Point Clouds Using Series of Local Triangulations." Journal of Imaging, 11(2):49, 2025. *doi:10.3390/jimaging11020049*

The higher-order surface reconstruction section was presented as a research note at the International Meshing Roundtable (IMR) 2025, Texas.

3.1. Introduction

Triangular meshes have long been the standard representation for discrete surfaces in computational geometry. However, point clouds have emerged as a viable alternative, particularly in scenarios where generating a surface mesh is challenging [16]. Techniques such as smoothed particle hydrodynamics (SPH) and radial basis function-finite differences (RBF-FD) [17] use tree-based representations [18] to establish connectivity in point clouds, offering advantages over traditional mesh-based methods for specific applications. Furthermore, advances in point-based rendering, driven by neural networks [19], highlight the growing importance of point clouds. Innovations in 3D scanning and LiDAR have cemented their role as a reliable representation of complex 3D geometries.

3.1.1. Related Work: Algorithmic Approaches

Point clouds can be oriented or unoriented depending on their source. Establishing consistent orientations in unoriented point clouds often requires specialized algorithms [20]. Additionally, for effective use, point clouds must often adhere to specific distribution patterns or maintain a desired resolution. While various methods have been developed for resampling point clouds [21], [22], [23], [24], [25], most approaches focus exclusively on either upsampling or downsampling. We provide a comparison against some of these works in our numerical experiments section.

Voxelization is a common intermediate representation used in resampling workflows. For instance, Chenlei et al. [23] construct occupancy grids for point clouds and resample them by averaging local voxel neighborhoods. However, voxelization often introduces artifacts that necessitate additional projection or optimization steps. While such artifacts may be tolerable for certain applications, CAE workflows typically demand highly uniform point cloud distributions [18].

3.1.2. Related Work: Learning-Based Approaches

Learning-based methods have emerged as powerful tools for point cloud processing, leveraging deep learning architectures to tackle tasks such as denoising, resampling, and reconstruction [26]. For instance, Zhao et al. [1] proposed a multi-task learning network for LiDAR point cloud preprocessing, incorporating denoising, segmentation, and completion branches. Similarly, Zhao et al. [27] introduced the ICDDPM model for image-conditioned single-view reconstruction, achieving state-of-the-art results on datasets like ShapeNet and PASCAL3D+. Other works, such as Wu et al. [28], Chen et al. [29], and Rong et al. [30], focus on advanced resampling and upsampling techniques, demonstrating impressive performance across diverse non-CAE datasets.

Despite their effectiveness, these learning-based methods are not explicitly tailored for computer-aided engineering (CAE) applications. This does not mean that they

cannot be used in a CAE workflow; rather, their training objectives and benchmark datasets typically emphasize visual reconstruction quality, denoising, or generic upsampling. CAE preprocessing often imposes additional requirements, such as predictable spacing, feature preservation near sharp engineering edges, reproducibility without retraining, and robustness on models that differ substantially from the training distribution. For completeness, we still include comparisons with select learning-based methods in our numerical experiments section to showcase the strengths of our approach.

3.1.3. Our Approach

In contrast to the aforementioned methods, we propose a novel algorithm leveraging an intermediate representation termed the **Series of Local Triangulations** (SOLT). By constructing and refining local Delaunay triangulations, SOLT avoids the artifacts associated with voxelization while ensuring uniformity and topology preservation. Unlike deep learning approaches, SOLT does not rely on pre-trained models, making it lightweight and suitable for CAE preprocessing workflows where reproducibility and predictable point spacing are important.

3.1.4. Contributions

The key contributions of this work are as follows:

1. A novel algorithm, SOLT, enabling efficient point cloud upsampling and downsampling while reducing grid artifacts and topology loss in the tested cases.
2. An evaluation of SOLT across a variety of inputs, including mechanically sampled point clouds and real-world 3D scans.
3. A preliminary application of the SOLT philosophy to dense and coarse CAD-free higher-order surface reconstruction using local fittings.
4. An evaluation of SOLT against selected state-of-the-art methods, highlighting its accuracy, computational efficiency, and fidelity for point cloud resampling, together with an application-oriented demonstration for higher-order surface reconstruction.

3.2. Methodology for Point Cloud Resampling

Figure 10 illustrates the point resampling workflow. Steps such as denoising and smoothing are optional and depend on the characteristics of the input point cloud. The methodologies for these steps are detailed in [31]. These filters are useful for handling noise or perturbations introduced by data collection sources, such as LiDAR or 3D scanners. Notably, our approach does not modify the point cloud directly; adjustments are applied to the intermediate mesh representation described in the following sections.

3.2.1. Series of Local Triangulations (SOLT)

The **Series of Local Triangulations** (SOLT) technique represents point cloud data by constructing localized triangulations around individual points.

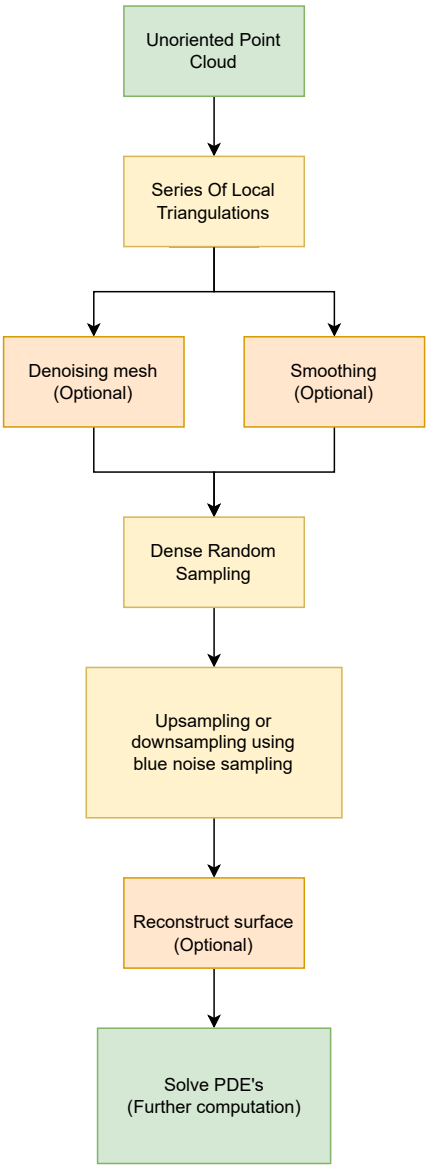


Figure 10: Workflow of the overall methodology. Optional modules are highlighted in light orange.

It uses distance and geometric parameters (tangent coordinates) to identify nearby points and computes a characteristic length scale for each local neighbourhood,

ensuring adaptability to varying point densities. Heuristics are applied to handle problematic configurations, maintaining stability and preserving geometry. Points are sorted to enable consistent local Delaunay triangulations, followed by iterative refinements. The resulting local triangulations are merged into a global triangulation by de-duplicating indices.

Series of Local Triangulations (SOLT)

Input: Unoriented point cloud
Output: Series of localized triangulations

1: For each point `p` in the point cloud:
2: Identify a local neighborhood `S(p)` of `p` using a distance threshold or tangent space coordinates
3: Determine the characteristic length scale of `S(p)` as the distance to its farthest neighbor
4: If points in `S(p)` are closer than a local tolerance, merge or perturb them to prevent degeneracies
5: Arrange points in `S(p)` in a counter-clockwise order for consistency
6: Compute the local Delaunay triangulation of `S(p)`
7: Refine the triangulation to ensure it adheres to the Delaunay criterion
8: Merge the local triangulations into a global mesh by eliminating duplicate vertices
9: Optionally, apply smoothing or denoising
10: Return series of localized triangulations

Algorithm 4: Series of Local Triangulations (SOLT) algorithm.

3.2.1.1. Nearest Point Selection for Local Triangulation

A key parameter in the SOLT algorithm is the search radius r, which defines the neighbourhood of each point. The neighborhood $S(p)$ for each point p is defined as:

$$S(p) = \{q \in P \mid |q - p| \leq r \wedge |T(q) - T(p)| \leq \varepsilon\}$$

where:

1. r: search radius (default: average point-to-point distance)
2. P: point cloud
3. $N(p_i)$: nearest neighbors of p_i
4. $T(p)$ and $T(q)$: tangent coordinate vectors at p and q
5. ε: threshold for tangent coordinate similarity

Alternatively, a fixed number of nearest points (k) can be used instead of a radius. The radius r is computed as:

$$r = \left(\frac{1}{n}\right) \sum_{i=1}^{n} |p_i - p_j|, p_j \in N(p_i)$$

In the experiments in this chapter, r is initialized from the average nearest-neighbor spacing and then enlarged locally if fewer than the minimum number of points required for triangulation are found. The tangent-coordinate tolerance ε is chosen relative to the same spacing, so that the neighborhood adapts to the scale of the point cloud rather than to an absolute unit. Points are treated as nearly coincident when their separation is below a local tolerance τ, chosen as a small fraction of the local spacing; such points are either merged or perturbed by a negligible amount before local triangulation to avoid zero-area triangles and unstable circumcircle tests. These thresholds are heuristic but scale-normalized, which makes the procedure reproducible across geometries with different units and sampling densities.

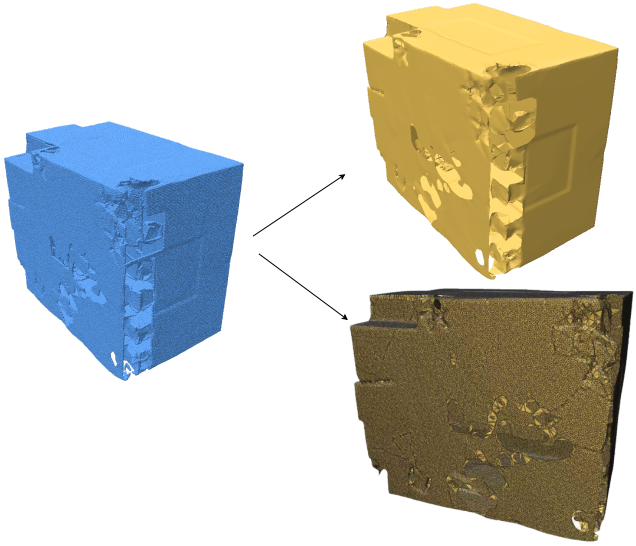


Figure 11: The SOLT intermediate representation. Local Delaunay triangulations are constructed around each point and merged into a global triangulation, serving as the basis for resampling operations.

3.2.1.2. Mesh Quality and Robustness

The SOLT mesh is an intermediate representation. Issues such as self-intersections or non-manifold edges do not affect the final resampling results. The SOLT representation supports the following operations:

1. Random sampling
2. Blue noise sampling (a refinement of random sampling)
3. Feature distance field computations

3.2.1.3. Circumcircle Criterion for Delaunay Triangulation

Delaunay triangulation ensures that no point lies inside the circumcircle of any triangle. For a triangle $\triangle(p_1, p_2, p_3)$, the circumcircle condition is:

$$|q - c| > r_c$$

for any q not in $\{p_1, p_2, p_3\}$.

where:

1. c: circumcenter of $\triangle(p_1, p_2, p_3)$
2. r_c: circumradius of $\triangle(p_1, p_2, p_3)$
3. q: any point outside the $\triangle$

Edges violating this condition are flipped to restore the Delaunay property.

3.2.2. Point Resampling

After calculating the intermediate SOLT representation, the next step involves resampling the points to meet specific application requirements. Point resampling can be tailored to generate a new point cloud with desired characteristics, such as uniform distribution, adherence to distance constraints, or alignment with specific features like sharp edges or curves.

Random Sampling on SOLT (Area-Weighted Sampling)

Input: SOLT, `numPoints` (desired number of points)
Output: Randomly sampled point cloud

1: Compute the area of each triangle T in SOLT
2: Normalize triangle areas to form a cumulative distribution function (CDF)
3: Initialize an empty set S
4: For each i from 1 to numPoints:
5: Select a triangle T randomly from the CDF, so that selection probability is proportional to triangle area
6: Generate random barycentric coordinates (u, v, w), where $u + v + w = 1$
7: Compute the sampled point p as $p = u * v_1 + v * v_2 + w * v_3$, where v_1, v_2, v_3 are the vertices of T
8: Add p to S
9: Return S

Algorithm 5: Random sampling on SOLT using area-weighted selection.

Blue Noise Sampling on SOLT

Input: SOLT, `minDistance` (minimum spacing between points)
Output: Blue noise sampled point cloud

Compute the area of each triangle T in SOLT and form a CDF
Initialize an empty set S and a candidate queue Q
Randomly select an initial triangle T_0 from the CDF
Generate a random point p_0 within T_0 and add p_0 to S and Q
While Q is not empty:
    Remove a point p from Q
    For each candidate point c generated around p:
        Select a triangle T_c containing c, weighted by area
        If c lies within T_c and satisfies the `minDistance` criterion from all points in S, add c to S and Q
Return S

Algorithm 6: Blue noise sampling on SOLT for uniform point distribution.

If random sampling is sufficient, the process can use methods that rely on random numbers and triangle areas to generate a new point cloud, as shown in Algorithm 5. However, for more structured applications, employing more sophisticated techniques like blue noise sampling is preferable (Algorithm 6).

3.2.3. Feature-Preserving Resampling via Distance Fields

Preserving features is crucial for applications requiring geometric fidelity. To address this, we estimate a distance field from feature points or curves onto the mesh and use it as a constraint. The generalized signed distance field is computed using the method described in [32].

At a high level, this method treats the detected feature set as the zero-level source and propagates distances over the surface so that each surface point receives a scalar value measuring its proximity to sharp curves or feature points. The resulting field is not used as a signed inside-outside classifier here; it is used as a surface-aware proximity measure. Small values indicate regions where resampling should retain more geometric detail, while larger values correspond to smooth regions where more uniform sampling is sufficient.

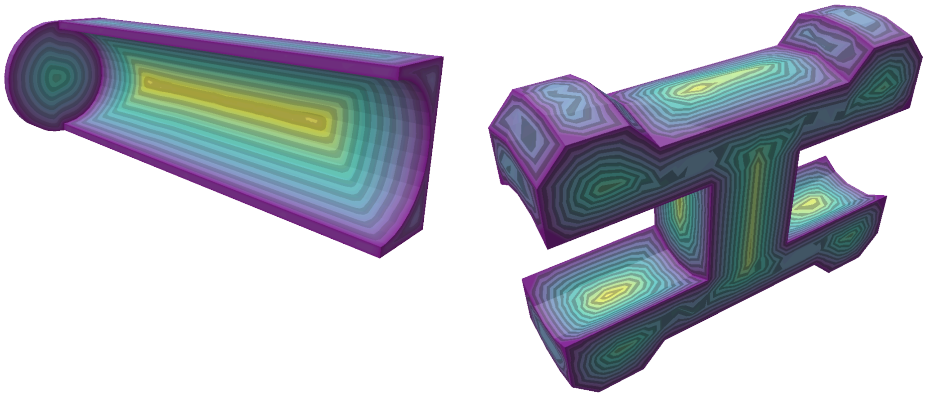


Figure 12: Feature distance field visualization on mechanical geometries. The distance field highlights sharp features and edges, guiding the resampling process to preserve geometric detail.

Feature-Preserving Resampling Using Feature Distance Field

Input: Original point cloud P, feature distance field $D(x)$
Output: Resampled point cloud P'

Estimate SOLT for P as per Algorithm 1
Compute feature distance field $D(x)$
Initialize P' = empty
Set minimum distance d
While target sampling density derived from $D(x)$ is not achieved:
    Generate candidate point p using Blue Noise Sampling
    If $D(p) <$ threshold and p satisfies distance criterion with all existing points in P':
        Add p to P'
Return P'

Algorithm 7: Feature-preserving resampling using feature distance field.

This feature distance field $D(x)$ is used to augment both resampling and local fitting processes. For resampling, the field modifies the sampling distribution, ensuring newly sampled points are sensitive to features. During Blue Noise Sampling, regions near features (where $D(x)$ is small) are sampled more densely. The workflow is described in Algorithm 7.

3.3. Numerical Experiments

The algorithm was implemented in C++ and evaluated using datasets including the Waterloo Point Cloud Database [33], [34], Thingi10k [35], and SimJEB [36]. The tests were conducted on an Intel i5-8350U laptop with 8 threads and an integrated GPU. These datasets feature diverse geometries such as everyday objects, mechanical components, and intricate high-genus structures, providing a comprehensive evaluation of the algorithm's performance in feature preservation, noise handling, and topological fidelity.

3.3.1. Quantitative Metrics

To evaluate the robustness of our algorithm, the following quantitative metrics were considered:

1. **Chamfer Distance Loss (%):** Calculates the average bidirectional distance between two point clouds as a percentage.

$$L_{c(\%)} = \left(\left(\frac{1}{|}P| \right) \sum_{p \in P} \min_{q \in Q} |p - q|^2 + \left(\frac{1}{|}Q| \right) \sum_{q \in Q} \min_{p \in P} |p - q|^2 \right) * 100$$

2. **Hausdorff Distance Loss (%):** Captures the maximum distance between the closest points of two point clouds, expressed as a percentage.

$$L_{h(\%)} = \left(\max \left(\sup_{p \in P} \inf_{q \in Q} |p - q|, \sup_{q \in Q} \inf_{p \in P} |p - q| \right) \right) * 100$$

3. **Uniformity Index (%):** Evaluates the evenness of point distribution across the resampled point cloud.

$$U(\%) = \left(1 - \left(\frac{\sigma_{\text{nn}}}{\mu_{\text{nn}}} \right) \right) * 100$$

4. **Volume Preservation Error (%):** Quantifies the percentage difference in volume between the original and resampled meshes.

$$E_{v(\%)} = \left(|V_{\text{orig}} - V_{\text{resamp}} \frac{|}{V_{\text{orig}}} \right) * 100$$

5. **Computational Time (s):** Measures the time taken by the algorithm to process and resample a point cloud.

$$T = t_{\text{end}} - t_{\text{start}}$$

6. **Compression Ratio:** Assesses the reduction in data size during resampling.

$$C = \frac{S_{\text{orig}}}{S_{\text{resamp}}}$$

3.3.2. Smooth Geometries

This experiment focuses on resampling point clouds from smooth geometries with low genus, using the Waterloo Point Cloud Dataset.

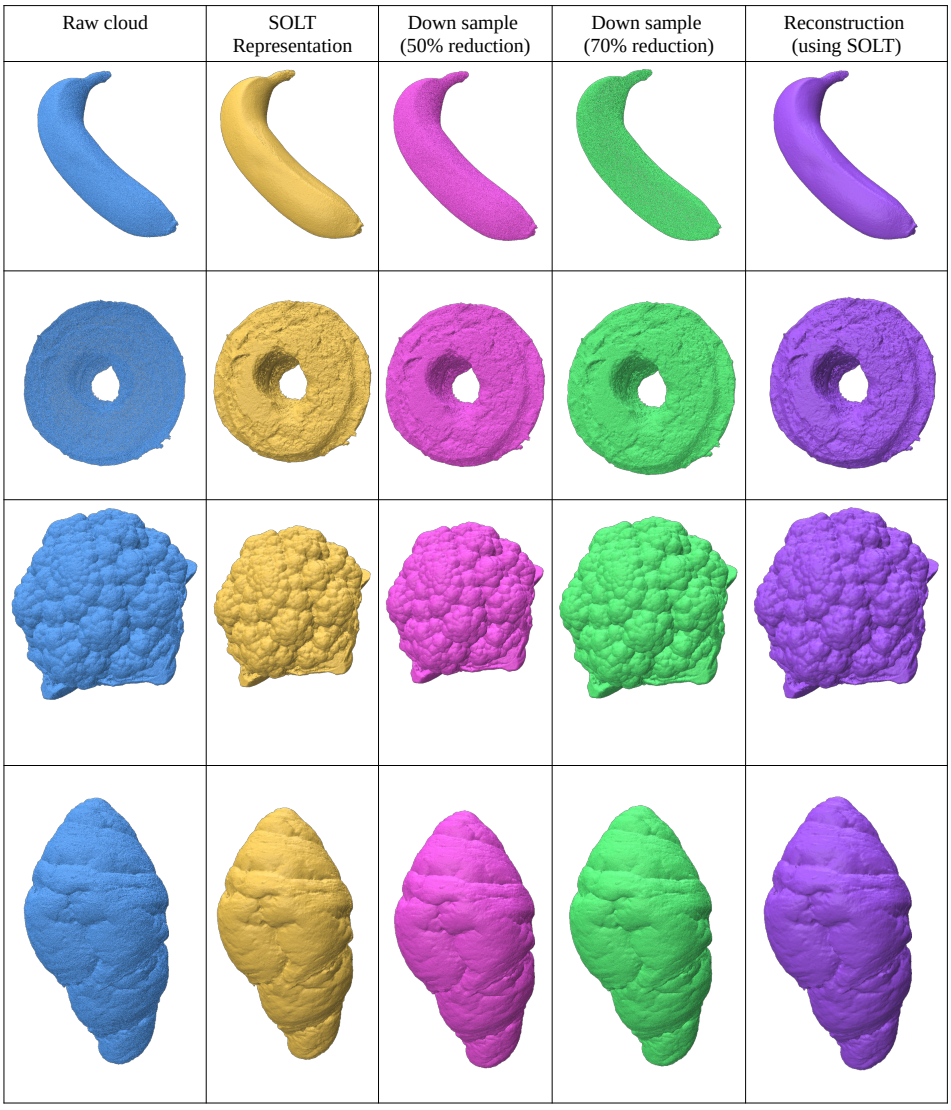


Figure 13: Point cloud (blue) meshed using SOLT (yellow), downsampled in two stages (pink and green), and reconstructed using the SOLT representation (purple).

The initial mesh is constructed using SOLT, followed by resampling based on the desired point-to-point distance criterion (Figure 13).

Metric	Range (% or Value)
Chamfer Distance Loss (%)	0.1–0.3
Hausdorff Distance Loss (%)	0.22–1.25
Uniformity Index (%)	96–99
Volume Preservation Error (%)	1.5
Computational Time (s)	5.0–30.0
Compression Ratio	2:1–3.33:1

Table 3: Quantitative metrics for smooth geometries.

We demonstrate 50% and 75% reductions in point density while preserving genus through reconstructed meshes. For refinement, Restricted Voronoi Diagram-based (RVD) re-meshing techniques [31] are suggested. Results show that the resampled point clouds maintain genus and exhibit uniform density distribution due to constrained blue noise sampling.

3.3.3. Mechanical Geometries

This experiment evaluates geometries with intricate feature curves sampled from the SimJEB dataset [36]. Dense point clouds are generated using random point cloud generation [37]. Results show effective feature preservation through explicit and implicit techniques (Figure 14).

Metric	Range (% or Value)
Chamfer Distance Loss (%)	0.3–0.6
Hausdorff Distance Loss (%)	0.8–1.3
Uniformity Index (%)	97–99
Volume Preservation Error (%)	0.5–1.0
Computational Time (s)	7.5–47.9
Compression Ratio	2:1–3.33:1

Table 4: Quantitative metrics for mechanical geometries.

3.3.4. Geometries with Intricate Features

To further evaluate the robustness of our algorithm, we tested intricate geometries sourced from the Thingi10k dataset [35]. These geometries include objects with sharp creases and mixed features, presenting significant challenges for resampling

and reconstruction techniques. The geometries were first converted into point clouds from triangular meshes to serve as input for the resampling process.

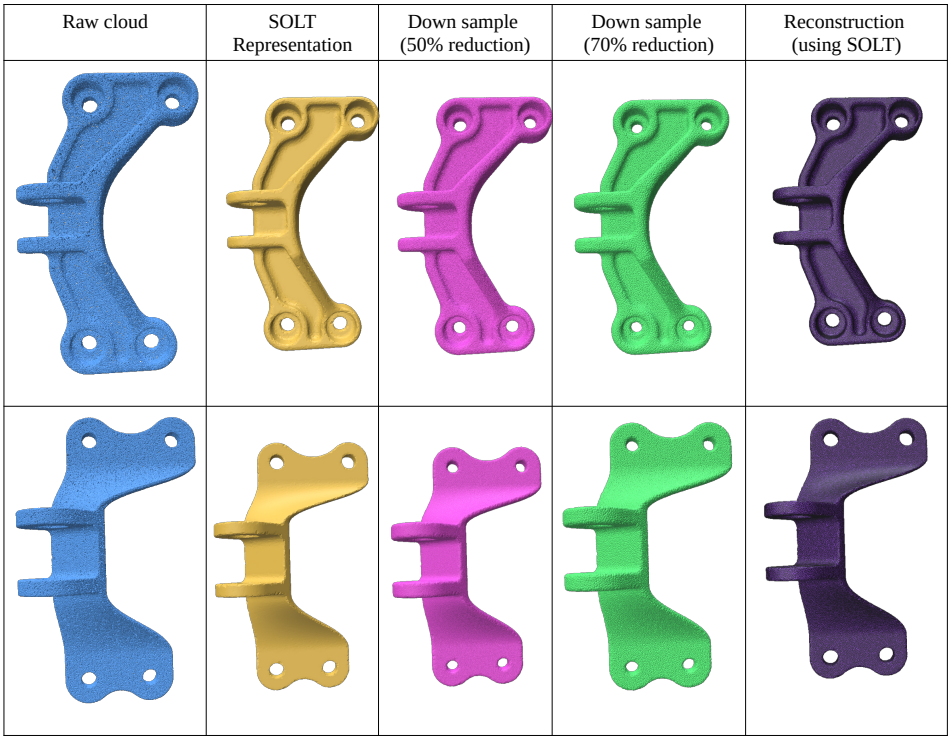


Figure 14: Point clouds synthesized from the SimJEB dataset. Point cloud (blue) meshed using SOLT (yellow), downsampled in two stages (pink and green), and reconstructed using the SOLT representation (purple).

Results, illustrated in Figure 15 and Figure 16, confirm that the algorithm accurately preserves intricate details such as sharp creases and twist-like features. These findings highlight the adaptability of the SOLT method to handle diverse geometric complexities.

The SOLT method was applied to generate an initial mesh representation of the point cloud, followed by blue noise sampling for resampling. Feature preservation was achieved using the feature distance field technique described in the Feature-Preserving Resampling section. This approach effectively retained sharp and smooth features, demonstrating the versatility and robustness of the algorithm.

Metric	Observed Range (% or Value)
Chamfer Distance Loss (%)	0.45–0.78
Hausdorff Distance Loss (%)	0.92–1.6
Uniformity Index (%)	96–97.5
Volume Preservation Error (%)	1.2–1.8
Computational Time (s)	12.0–46.2
Compression Ratio	2.5:1–3.33:1

Table 5: Quantitative metrics for intricate feature geometries.

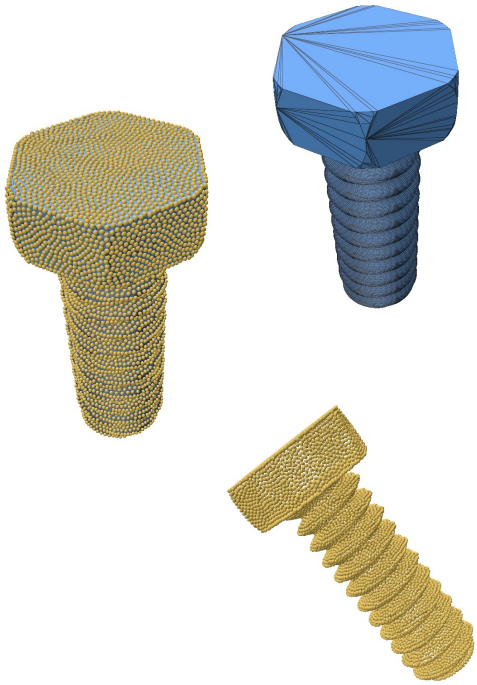


Figure 15: A screw geometry resampled using our algorithm (geometry from the Thingi10k dataset).

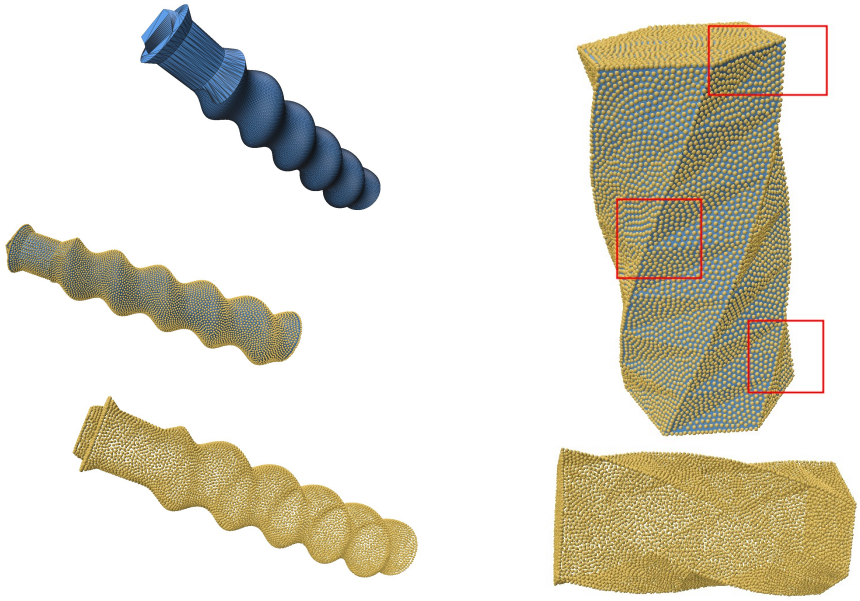


Figure 16: A mixture of smooth and sharp geometries with twist-like features (geometries from the Thingi10k dataset).

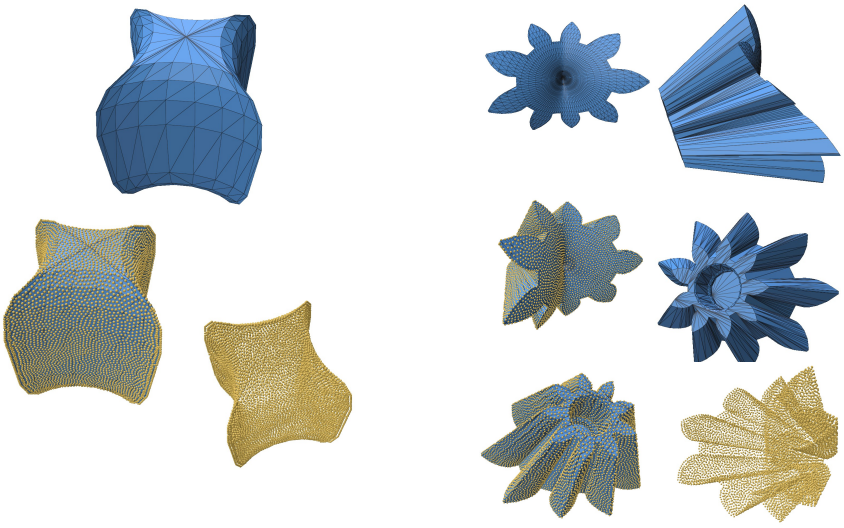


Figure 17: Mechanical components from the Thingi10k dataset. The red inset boxes highlight sharp crease and corner regions used for visual inspection of feature preservation.

3.4. Comparison Against Existing Works

In this section, we compare the resampling capabilities of our algorithm against selected existing works from the literature. A small subset of examples is chosen for this comparison, focusing on those with readily available implementations and examples. Many existing works were excluded from this comparison as they either lack support for Linux or require specific hardware, such as high-end GPUs, which limits their accessibility.

3.4.1. Traditional Methods

The method proposed in [1] introduces a two-step framework for intrinsic and isotropic resampling. It combines efficient intrinsic control using geodesic measurements with geometrically optimized resampling to address challenges such as non-uniform point density and adjacency information in point clouds. This algorithm demonstrates strong performance in applications such as point cloud simplification, mesh reconstruction, and shape registration, leveraging geometric updates for isotropic or adaptively isotropic resampling.

Despite its strengths, the efficiency of the algorithm in [1] significantly decreases when the target output exceeds 50,000 points. This is primarily due to the computational overhead of Delaunay triangulation and geodesic coordinate mapping, which impact its scalability for high-resolution point clouds. As a result, the algorithm is less practical for applications requiring large-scale resampling.

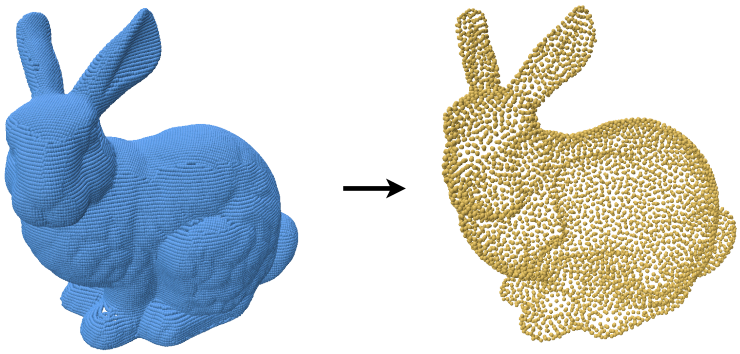


Figure 18: Input Bunny point cloud along with a 5000-point resample produced by [1]. These results were provided by the authors.

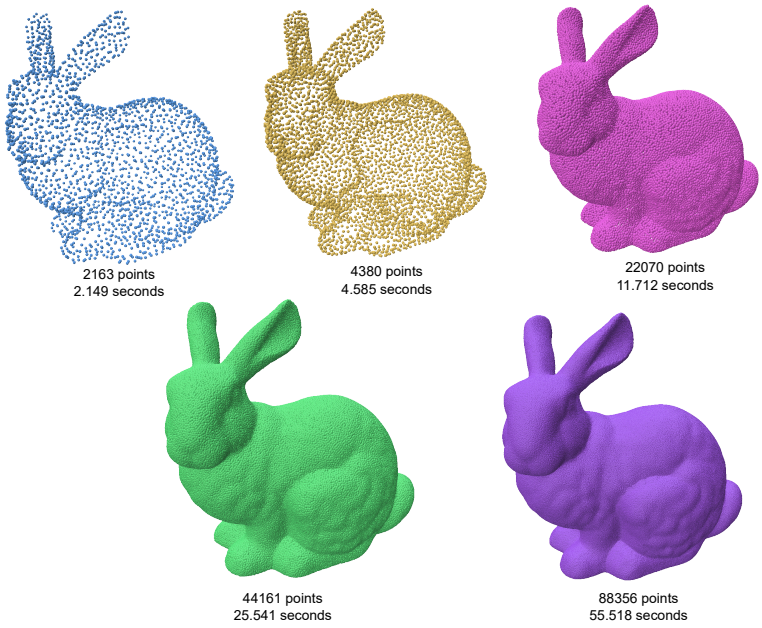


Figure 19: Bunny resampled at various sizes using SOLT, along with corresponding sampling times. The results demonstrate that SOLT maintains consistent efficiency and quality as sample size increases, comparable to the algorithms proposed in [1].

3.4.2. Learning-Based Methods

The method proposed in [30], known as RepKPU, introduces a novel approach to point cloud upsampling by leveraging kernel point representation and a Kernel-to-Displacement paradigm. This technique reformulates upsampling as the deformation of kernel points guided by learned geometric features, enabling density-sensitive and position-adaptive local geometry representations. RepKPU demonstrates superior performance across several benchmarks, including the PU-GAN and PU1K datasets, producing smoother and more uniform point clouds while maintaining computational efficiency.

For this study, we used the chair example provided by the authors of RepKPU. When tested on the same data, our method generated a hole-free reconstruction, whereas the RepKPU result supplied for comparison contains visible holes in the reconstructed surface. This comparison should be interpreted qualitatively, since the available RepKPU result does not provide the full set of benchmark quantities needed for a controlled quantitative study. It nevertheless illustrates a practical advantage of the SOLT representation on this example: local geometric consistency is preserved without relying on a learned prior.

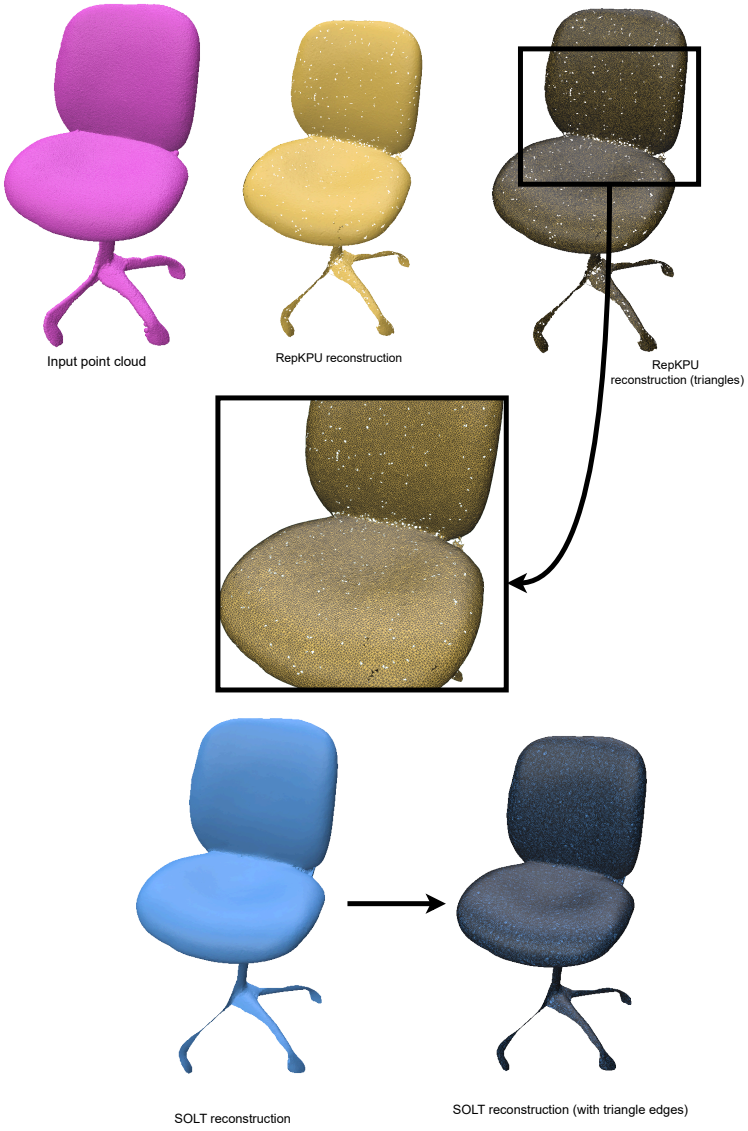


Figure 20: Chair reconstruction from input point cloud using the RepKPU workflow (results shared by the authors). The reconstruction contains multiple visible holes. For comparison, the SOLT reconstruction of the same chair geometry is shown and is hole-free in this example.

Figure 21: Chair resampled at various sizes using SOLT, along with corresponding sampling times. The results demonstrate that SOLT maintains consistent efficiency and visually coherent output as the sample size increases.

3.5. Application: CAD-Free Higher-Order Surface Reconstruction

Beyond point cloud resampling, the philosophy of using an intermediate local structure can be applied to CAD-free higher-order surface reconstruction. Local fitting is already an established route for high-order surface reconstruction, for example through WALF (Weighted Average of Local Fittings) [38], [39]. The question considered here is narrower: whether a SOLT-style intermediate support structure can make such local-fitting workflows more flexible for CAD-free, non-uniform, and feature-sensitive surface data. We use a concept similar to SOLT, which we term Series of Local Fittings (SOLF).

High-order meshes are important for high-order finite-element and finite-volume methods because curved elements can represent smooth boundaries more accurately than linear elements at comparable resolution [40]. When a reliable CAD model is available, high-order meshing can use the CAD geometry as the reference

for placing curved boundary nodes and for measuring geometric error. In many reverse-engineering, scanned-data, or repaired-mesh workflows, however, the original CAD model is unavailable, incomplete, or no longer consistent with the surface data. In these cases, a CAD-free high-order workflow must infer the curved geometry directly from a mesh or point cloud.

This CAD-free setting is substantially harder. Curving a linear mesh can introduce invalid or inverted elements; feature curves and sharp corners must be preserved; the curved boundary must remain sufficiently faithful to the measured or reconstructed surface; and the local surface fit must be stable under non-uniform sampling, noise, and missing data. Existing work addresses these issues through local polynomial reconstruction, CAD-free low-order to high-order conversion, and bijective coarse high-order mesh generation [38], [39], [41].

Higher-Order Surface Reconstruction Workflow

Input: Surface mesh
Output: Higher-order mesh with refined points

1: Calculate a feature distance field
2: Analyze surface density for uniformity
3: Build an intermediate support structure (random point sampling)
4: If mesh density is globally uniform:
5: └ Proceed with mesh stencils alone
6: Else:
7: └ Include points from intermediate support structures (for local fittings)
8: Estimate local fittings using dynamic stencils (based on the feature distance field)
9: Determine initial positions for higher-order points
10: Apply a weighted Least Squares (LS) method to refine the higher-order point positions
11: Return higher-order mesh with refined points

Algorithm 8: Higher-order surface reconstruction workflow using SOLF.

Our work builds on this local-fitting foundation but generalizes the support construction using a weighted Least Squares (LS) approach over neighborhoods derived from the intermediate structure. We argue that noted drawbacks can be addressed within a flexible framework, especially given the diverse array of LS variants [42].

3.5.1. Higher-Order Surface Reconstruction Workflow

The workflow for generating a higher-order mesh using the SOLF method is outlined in Algorithm 8. In this context, analyzing surface density means comparing the coefficient of variation of local edge lengths or nearest-neighbor distances across the mesh. If this variation is small, topological mesh stencils provide sufficient local support. If it is large, additional support points from the intermediate structure are included in the least-squares neighborhood.

3.5.1.1. Reconstruction with Least Squares

Given a point x_i and its neighboring points, the local approximation at x_i is achieved by fitting a polynomial $P(x;\beta)$ to the values y_j at neighboring points x_j. This is formalized as minimizing the weighted sum of squared differences.

The higher-order nodes are first placed at Legendre-Gauss-Lobatto (LGL) positions on each element edge or face. LGL points are standard interpolation points used in high-order finite-element and spectral-element discretizations; they include the element endpoints and provide the initial parametric locations of the curved higher-order nodes. In the workflow below, these initial positions are then projected or adjusted using the fitted local surface.

Reconstruction with Least Squares

Input: Surface mesh, feature distance field, intermediate support structure (optional), LGL points
Output: Reconstructed surface with higher-order points

1: For each mesh node x_i:
2: Identify neighboring points for x_i by combining topological stencil neighbors with Euclidean nearest neighbors
3: Compute the weight function $w(x_i, x_j) = \exp\left(-\frac{|x_i - x_j|^2}{2\sigma^2}\right)$
4: Construct a local polynomial approximation $P(x;\beta)$ over the neighborhood
Solve the weighted least squares problem:
5: $$\min_{\beta} \sum_{j=1}^{n} w(x_i, x_j)\big(y_j - P(x_j;\beta)\big)^2$$
6: Update higher-order point positions using the local polynomial fit $P(x_i)$
7: Return reconstructed surface with higher-order points

Algorithm 9: Reconstruction with weighted Least Squares fitting.

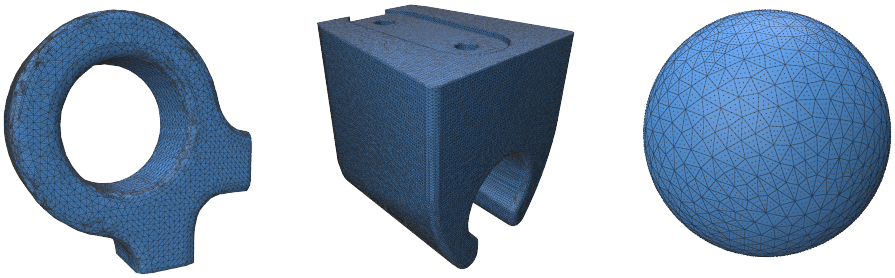


Figure 22: Dense higher-order surface reconstruction examples. Left and center: mechanical geometries with curved surfaces reconstructed using SOLF. Right: sphere mesh with fourth-order (p4) elements showing smooth curvature preservation.

In practice, the weighted least-squares system can be solved with any numerically stable dense least-squares solver, such as QR factorization or an SVD-based fallback for ill-conditioned neighborhoods.

3.5.2. Coarse Higher-Order Surface Reconstruction

For generating coarse higher-order meshes, we rely on the surface multigrid method proposed by [43].

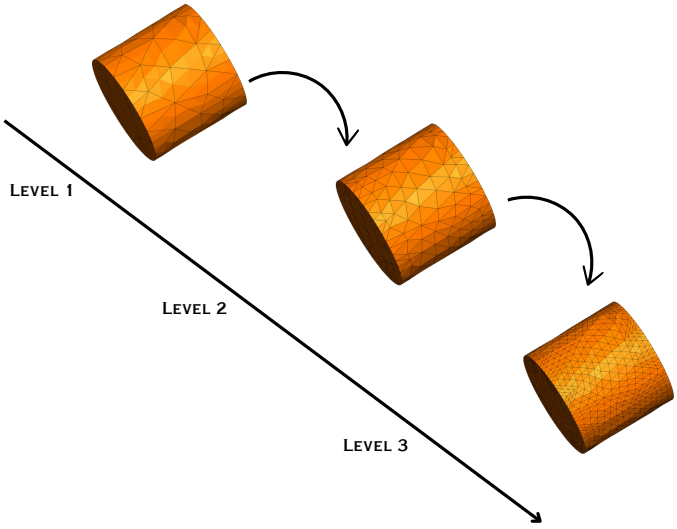


Figure 23: Coarse mesh with higher-order elements.

This method provides a way to decimate a dense mesh and create a bijective map between points in the coarse and dense meshes. The workflow is described in Algorithm 10.

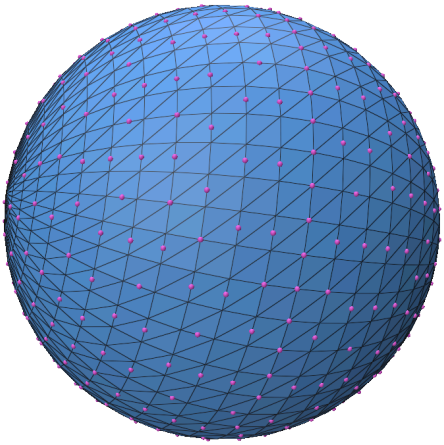


Figure 24: Coarse-to-fine mapping using the multigrid framework, preserving geometric detail from the dense mesh.

Coarse Mesh Generation Using the Multigrid Framework

Input: Dense linear surface mesh
Output: Positions of higher-order points on a coarse SOLF surface

1: Decimate the mesh using quality and feature-preservation criteria (e.g., target edge length, curvature, and boundary/feature constraints)
2: Generate a coarse-to-fine mesh mapping using the multigrid framework
3: For each point in the coarse mesh:
4: Estimate a SOLF surface for the point, using the corresponding mapped points on the dense mesh to preserve detail
5: Find nearby points from the intermediate support structure and use them to generate the local LS surface
6: Estimate the positions of the higher-order points based on the coarse SOLF surface using the LS reconstruction algorithm
7: Return positions of higher-order points on a coarse SOLF surface

Algorithm 10: Coarse mesh generation using the multigrid framework.

This application should be interpreted as a preliminary study of SOLT-style support structures in CAD-free high-order surface reconstruction. Prior work such as WALF has already shown that local fittings are a valid route for high-order surface reconstruction [38], [39] the role of SOLF is to explore whether an intermediate support structure can improve the flexibility of this route for CAD-free, non-uniform, and

feature-sensitive data. The examples show that higher-order node positions can be generated from local fittings, but they do not yet establish element validity guarantees, surface-error convergence, or solver-level accuracy. Future work should expand and validate this algorithm in a dedicated follow-up study, including a separate evaluation against established high-order meshing workflows.

3.6. Conclusion

This chapter presented the Series of Local Triangulations (SOLT) as an intermediate representation for point-cloud resampling. By constructing local triangulations and using them for area-weighted and blue-noise sampling, the method avoids the grid artifacts commonly introduced by coarse voxelization and provides a practical route for both upsampling and downsampling.

The numerical examples show that SOLT preserves point spacing, geometric features, and reconstructed topology across smooth geometries, mechanical components, and scanned or mesh-derived point clouds. The feature-distance-field extension further allows the sampling density to be biased toward sharp edges and corner regions, which is important in CAE preprocessing where local geometric detail often has direct influence on subsequent meshing or analysis steps. Comparisons with selected traditional and learning-based methods indicate that SOLT is competitive in the tested cases while remaining deterministic and independent of training data.

The chapter also outlined SOLF as a potential application of the SOLT idea for CAD-free higher-order surface reconstruction. This application builds on established local-fitting approaches and explores whether an intermediate support structure can make such workflows more flexible for non-uniform and feature-sensitive surface data.

The examples in this chapter show that SOLT can act as a practical preprocessing component for CAE-oriented point-cloud workflows, especially when predictable spacing and feature preservation are required. A complete end-to-end solver workflow using the resampled output is not demonstrated here, so integration with downstream meshing and simulation tools remains a subject for further validation. Within this scope, the method advances point cloud processing and illustrates a possible future path toward CAD-free higher-order surface reconstruction by addressing challenges in feature preservation, surface reconstruction, and geometric fidelity.

3.7. Limitations

The algorithms proposed in this chapter are designed with CAE preprocessing requirements in mind and tested mainly on engineering-style geometries. Conse-

quently, their robustness and performance in other fields may not align with the levels demonstrated in the experiments. The method also assumes that meaningful local neighborhoods can be constructed from the input point cloud. Extremely sparse data, strong noise, large missing regions, or severe non-uniform sampling can make the local triangulations less reliable and may require additional denoising, completion, or adaptive parameter selection.

Several parameters, including neighborhood radius, tangent-coordinate tolerance, near-coincident point tolerance, and blue-noise spacing, are scale-normalized but still heuristic. The guidance provided in this chapter is intended to make the experiments reproducible, but a fully automatic parameter-selection strategy remains future work. The comparison against learning-based methods is also limited by the availability of implementations and shared examples; in particular, the RepKPU comparison should be interpreted as a qualitative example rather than as a complete benchmark. Finally, the current implementation is not parallelized. Introducing parallel neighborhood construction and local triangulation would improve scalability for large point clouds and make the workflow more suitable for time-sensitive engineering applications.

4

Shrink Wrap Mesh Generation Using Morphological Methods

Abstract. Various computational fluid dynamics simulations in engineering, such as external aerodynamics, only need the silhouette of an input geometry. Often, preparing such a model is a laborious process that can take many hours of manual work. Furthermore, industrial CAD geometries are often overly complex, containing intricate features and topological holes that are irrelevant for such simulations. We present an automated way to shrink-wrap triangulated surfaces for topology and surface simplification. Building upon the concepts of mathematical morphology and recent advancements in geometry processing, we present a straightforward and robust algorithm for producing genus-simplified surfaces. Our techniques are equally applicable to general polyhedral meshes and are well-suited for handling both oriented and unoriented point clouds. We provide examples using unoriented point clouds to demonstrate the versatility of our algorithms. While designed with a wide variety of applications in mind, we specifically highlight external aerodynamic simulation as a suitable area of application and discuss surface simplification. Additionally, we emphasize the practicality and ease of implementing the proposed algorithms, and we chain additional algorithms to develop variants of our wrap algorithm.

This chapter is based on: V.K. Suriyababu, C. Vuik, and M. Möller. "Towards a High Quality Shrink Wrap Mesh Generation Algorithm Using Mathematical Morphology." Computer-Aided Design, 164:103608, 2023. doi:10.1016/j.cad.2023.103608

4.1. Introduction

The shrink-wrapping algorithm is a powerful tool for remeshing and simplifying polyhedral surfaces. Most approaches take a volumetric approach to the problem,

working by projecting a voxelized approximation of the input surface onto itself. Several papers in the literature focus on this problem, with much of the foundational work being accomplished in industry. Attene et al. [44] provide a detailed comparison of different mesh repair algorithms, offering an excellent overview of state-of-the-art methods that use a global approach. A key theme is the reliance on a volumetric intermediate representation. However, many of these algorithms are generic mesh simplification approaches for computer graphics and do not explicitly solve the problem with CAE simulations in mind. Notable contributions from Esteve et al. [45] and Nooruddin et al. [46] also employ volumetric approaches, but lack fine control over the surface genus and do not always guarantee a manifold mesh as output, making their results unsuitable for numerical simulation. Y. K. Lee et al. [47] summarize existing techniques tailored to shrink-wrapping for engineering applications, highlighting their effectiveness in closing gaps and removing interior parts of complex assemblies.

More recent works on shrink-wrapping have emerged from the computer graphics community. **Point2Mesh** [2] solves the problem with a self-prior approach that employs deep learning, tailor-made for point clouds. The recent three-dimensional alpha-wrapping algorithm [48] uses alpha shapes and is extremely robust to various input geometries. Depending on the alpha value, the geometry's genus can change, making it a powerful surface reconstruction algorithm that can handle various defects.

While the algorithms proposed in our work can be fine-tuned to function similarly, we explicitly focus on fully automated genus simplification. We propose an algorithm that can close gaps in triangulated surfaces and remove internal structures under the selected morphological scale. The significant contribution is an efficient way of computing a genus-simplified offset surface with the help of morphological operators. The algorithm is designed to produce a usable simplified output even for imperfect geometries, such as those with missing triangles, non-manifold edges, or completely separated components. We also extend the existing shrink-wrap algorithm for selective genus control, using the semi-heuristic hole detection algorithm detailed in Chapter 2 [49] to control the closing operations.

4.2. Morphological Operators

Mathematical morphology is a vibrant subject that finds typical applications in image processing [50] but has been extended to many other areas, including three-dimensional geometry processing [51]. Broadly, the subject is composed of four operators:

1. Erosion ($M \ominus \sigma$)
2. Dilation ($M \oplus \sigma$)

3. Opening $((M \ominus \sigma) \oplus \sigma)$
4. Closing $((M \oplus \sigma) \ominus \sigma)$

Erosion is an iterative process that erodes a given volume M using a structuring element σ, while dilation iteratively expands the input volume. In our work, we draw inspiration from Chen et al. [52], who employed morphological operations for computing discrete surface offsets. However, our focus is on a more specific form of mesh and topology simplification. Instead of computing exact offset surfaces, we aim to derive simplified offset surfaces that primarily serve the purpose of topological simplification. To represent all the morphological operators, we utilize an Octree data structure.

4.2.1. Mathematical Morphology of Surface Meshes

In our investigation, the morphological operators are analogous to their counterparts in image processing, with some key distinctions. In image processing, operators are typically applied to 2D grids, where a binary image provides an implicit representation of the object's boundaries. Surface meshes lack such an implicit mask. To perform morphological operations, a volumetric representation must first be calculated from the boundary representation. This establishes a comparable binary mask that aligns with the principles of morphological operators.

A key difference in our erosion operator is that we do not erode the geometry beyond its original boundary, as our focus is on preserving the internal volume. Let us consider the dilation and erosion operators on binary images and their geometric counterparts, as the opening and closing operators are combinations of these two.

4.2.2. Dilation

In the dilation process, we begin by dilating the given image using a square as the structural element. Once the image's boundary is identified, performing one step of dilation becomes a matter of finding the neighboring faces of the border. A crucial aspect of this approach is selecting the neighbors in the positive normal direction of the boundary.

In the context of image processing, the dilation operation can be described as:

```
def dilation(image):
    dilated_image = copy(image)
    for each pixel p in image:
        for each neighbor n of pixel p:
            if n is a background pixel:
                mark pixel p as a foreground pixel
                break
    return dilated_image
```

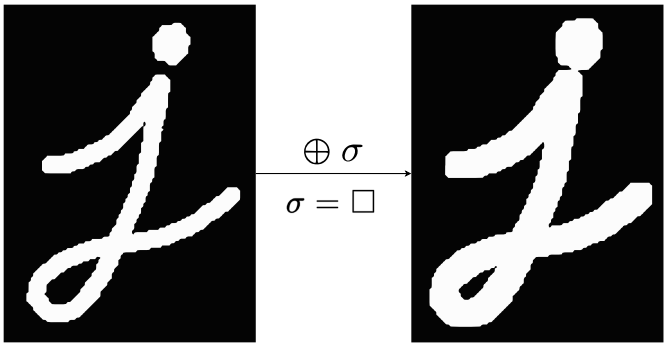


Figure 25: Dilation operation on a binary image. The input image on the left is dilated to obtain the image on the right.

Similarly, in the context of octree-based mesh processing, the dilation operation can be described as:

```
def dilation(mesh):
    dilated_mesh = copy(mesh)
    for each face f in mesh:
        for each neighboring face n of face f:
            if n is an empty face:
                mark face f as a filled face
                break
    return dilated_mesh
```

4.2.3. Erosion

The erosion operator also starts from the boundary of a given image. However, it differs from the dilation operator in that it finds the border neighbors in the negative normal direction. A pivotal contrast to the dilation approach is that an image cannot be eroded infinitely.

In the case of shrink wrapping, it is essential to prevent the erosion operation from destroying the internal volume of a surface mesh. Therefore, the erosion operation usually stops at the boundary of a surface mesh. The pseudo-code for image erosion is:

```
def erosion(image):
    eroded_image = copy(image)
    for each pixel p in image:
        for each neighbor n of pixel p:
            if n is a foreground pixel:
                mark pixel p as a background pixel
                break
    return eroded_image
```

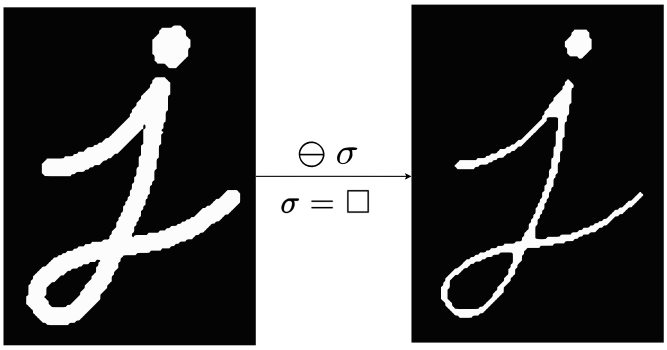


Figure 26: Erosion operation on a binary image.

For octree-based meshes, the erosion operation is:

```
def erosion(mesh):
    eroded_mesh = copy(mesh)
    for each face f in mesh:
        if f is not on the boundary of the mesh:
            for each neighboring face n of face f:
                if n is a filled face:
                    mark face f as an empty face
                    break
    return eroded_mesh
```

4.2.4. Closing and its Extension to Surfaces

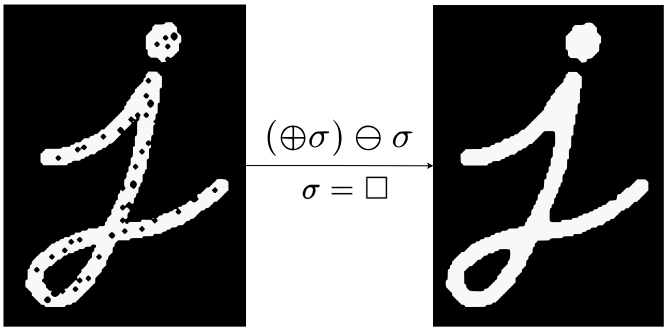


Figure 27: Closing operation on a binary image (black blobs on the left indicate missing pixels), equivalent to holes or missing triangles in a surface mesh.

The closing operator combines the dilation and erosion operations. In computer vision, this operator is used to close holes (missing pixels) in a binary image. We apply the same operation to 3D data to close holes in meshes. The results can be seen in Figure 28. There are two differences compared to the image processing approach:

1. We do not erode beyond the boundary of the input geometry to preserve its shape.
2. We project the eroded geometry onto the input geometry to maintain its original topology.

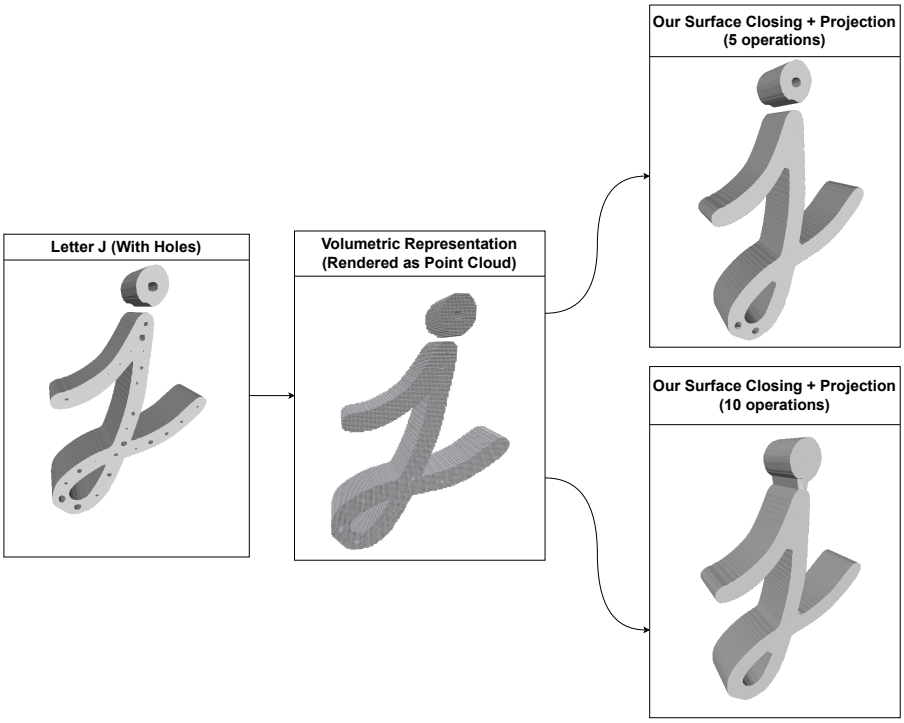


Figure 28: Closing operation on a three-dimensional surface. This geometry is equivalent to the two-dimensional binary images in Figure 27. Additional holes have been intentionally introduced in the geometry. The closing operation is performed on the leftmost geometry and then projected onto the ground truth.

4.3. Shrink-Wrap Algorithm

This work focuses on three versions of the wrapping algorithm: simple wrap, smooth wrap, and developable wrap. The simple wrap algorithm acts as a base for all variants, with additional algorithms chained to its results to yield the other versions.

4.3.1. Simple Wrap

The simple wrap algorithm acts as a common denominator for all variants and has the following steps:

1. Conversion of boundary representation (B-Rep) to a volumetric representation (V-Rep).
2. Computation of signed distance function.
3. Dilation of the input surface for a given topological sphere level.

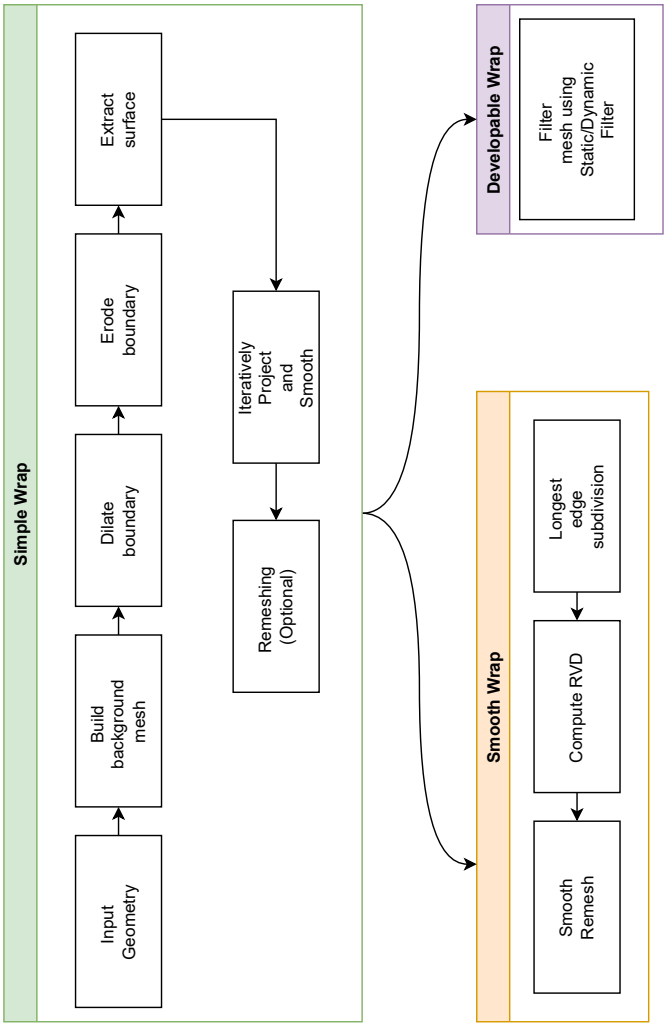


Figure 29: Algorithm workflow.

4. Erode the dilated surface to obtain a topologically hole-free offset surface.
5. Iteratively project and smooth the dilated surface.
6. Remesh the surface using existing remeshing algorithms (optional).

4.3.1.1. B-Rep to V-Rep

For efficient computation of morphological operators, geometry information needs to be encoded in a volumetric representation. We use an octree, which we consider a special kind of Cartesian mesh. To ensure morphological operators work correctly, the geometry should sit approximately in the center of the octree. First, the miniball algorithm [14] is used to obtain a tightly fitting sphere of an input geometry. Then we compute a bounding box for this sphere with a specified offset to ensure our geometry sits at the center of our voxelization. Post-refinement, the generalized winding number approach [53] helps distinguish the cells inside and outside the geometry. The full workflow is described in Algorithm 11.

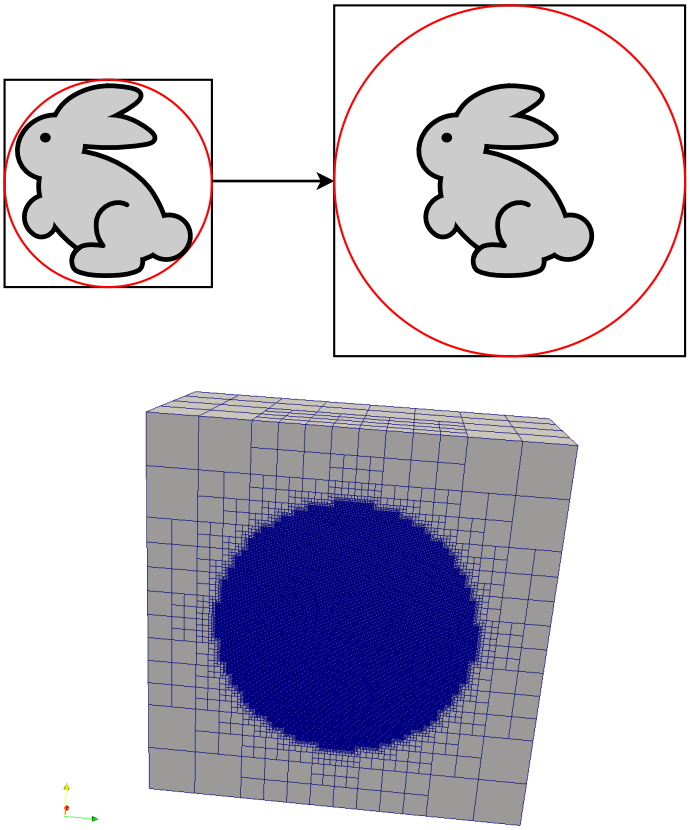


Figure 30: Left: Spherical bounding box computation ensures the geometry is centered in the octree domain. Right: Spherical refinement of the octree cells.

B-Rep to V-Rep Conversion

Input: Surface mesh (B-Rep), sphere offset, Cartesian mesh resolution
Result: Voxelized mesh where every cell has a scalar associated with it (inside / outside)

1: Initialize surface
2: Compute a tight bounding sphere using the mini ball algorithm and store its radius and center
3: Calculate the bounding box of the sphere, which is offset at a user-specified distance (2.0 in our experiments)
4: Initialize a Cartesian mesh with a specified cell size or number of cells (64 x 64 x 64 in all of our experiments)
5: For all cells of the Cartesian mesh:
6: | If cell inside bounding sphere, mark for refinement
7: | Else mark for coarsening
8: For all cells in bounding sphere:
9: | Compute the generalized winding number (this indirectly gives the solid angle for all the cells in the octree)
10: Mark cells outside bounding sphere as outside and store this in the respective cells
11: For all cells in bounding sphere:
12: | If solid angle is higher than 0.9 steradians (based on our experimental observation), mark the cell as inside and store this in the respective cell
13: | Else mark the cell as outside and store this in the respective cell
14: Return classified voxelized mesh

Algorithm 11: B-Rep to V-Rep.

4.3.1.2. Computation of Signed Distance Function

Once the cells of the octree are classified, an artificial signed distance function can be bestowed upon the voxelization. We rely on generalized winding numbers as they are swift and immune to imperfections in the input surface. We use the term "artificial" since we do not compute the exact distance, but an integer that indicates a voxel's relative position to its boundary. This is achieved by outward propagation from the zero-level set voxels.

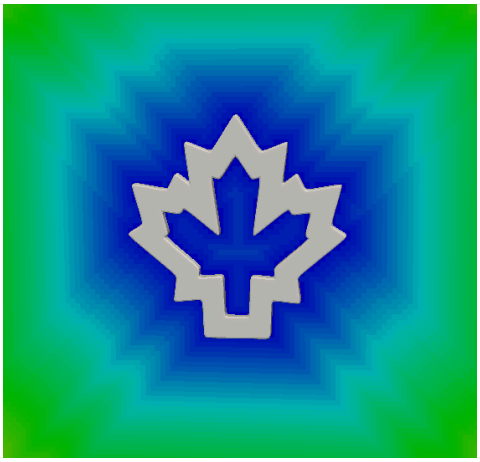


Figure 31: Artificial signed distance function of a maple leaf (computed using our approach).

4.3.1.3. Dilation driven approximated offsets

We only need to dilate the input surface until we achieve a spherical topology.

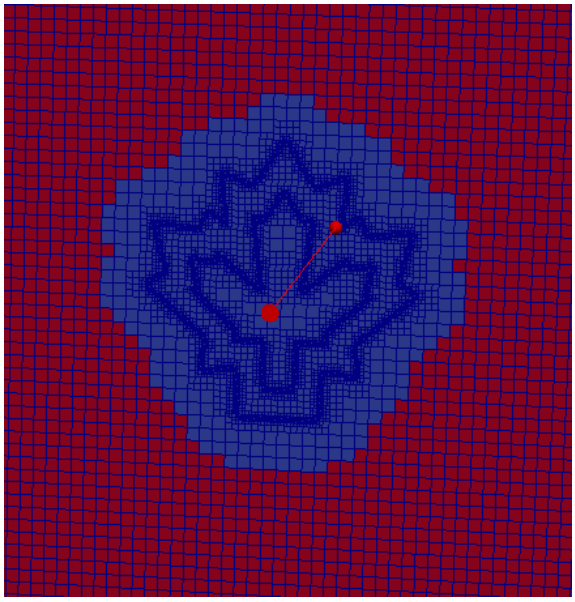


Figure 32: Dilated maple leaf geometry with a spherical topology.

Here, we use a user-specified parameter called **Topological sphere level**. This parameter dictates the number of outward propagation levels as shown in Algorithm 12. The bigger the hole in the geometry, the larger the topological sphere level.

Surface Dilation

Input: Interior cells C_i, target level L_t
Result: Dilated surface cells C_d

Initialize seed set $S = C_i$
Initialize current level $l = 0$
While $l \leq L_t$:
    Initialize next seed set $S' = \emptyset$
    For all cells $c \in S$:
        For all neighbours $n \in N(c)$:
            If n is an exterior cell: $S' = S' \cup \{n\}$
    Set $S = S'$
    Increment $l = l + 1$
    If $l = L_t$: store S as dilated cells C_d
Return C_d

Algorithm 12: Dilation of the input surface.

4.3.1.4. Erosion of dilated offset surface

Surface Erosion

Input: Dilated cells C_d, target level L_t
Result: Eroded surface cells C_e

Initialize seed set $S = C_d$
For $l = L_t$ down to 0:
    Initialize next seed set $S' = \emptyset$
    For all cells $c \in S$:
        For all neighbours $n \in N(c)$:
            If n is at level $l - 1$: $S' = S' \cup \{n\}$
    Set $S = S'$
Final set S forms the genus-simplified surface cells C_e
Return C_e

Algorithm 13: Erosion of the dilated offset surface.

The dilated surface is eroded towards the input surface. The number of levels is the same as the topological sphere level. Once eroded, this gives a hole-free approxi-

mation of the input geometry, as detailed in Algorithm 13. The geometry is never eroded beyond the boundary voxels for volume preservation. Surface extraction (Algorithm 14) becomes a simple task of identifying the boundary between the final eroded cells and the exterior.

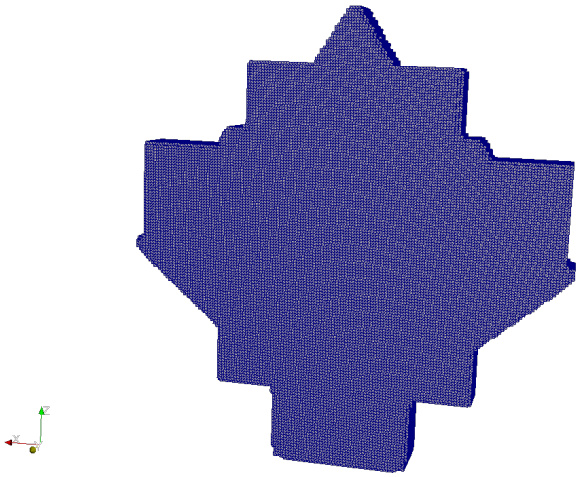


Figure 33: Eroded offset surface (unsmoothed and hole free).

Surface Extraction

Input: Eroded surface cells C_e, exterior cells C_o
Result: Genus-simplified watertight surface mesh M

1: Initialize boundary face set $F = \emptyset$
2: For all cells $c \in C_e$:
3: ⌊ For all neighbours $n \in N(c)$:
4: ⌊ ⌊ If $n \in C_o$: add shared face between c and n to F
5: Construct mesh M from boundary faces F
6: Return M

Algorithm 14: Surface extraction.

4.3.1.5. Projection and smoothing

The eroded surface needs to be projected onto the input geometry. We chose a point cloud-based approach over direct projection for robustness with imperfect meshes.

Figure 34: Projected and smoothed mesh.

We sample a uniform point cloud on the input geometry and construct a kD tree [54]. We find the nearest neighbor in this point cloud for every vertex in the eroded surface and move the vertex to this position. A few cycles of Laplacian smoothing followed by projection provide better quality results, as seen in Figure 34.

Projection and Smoothing

Result: Projected and smoothed mesh

```
Sample a uniform point cloud on the input surface
Build a kD tree on the uniformly sampled point cloud
For all vertices in the extracted surface:
    Find the nearest vertex in the kD tree and move the vertex
For all vertices in the projected mesh:
    Find one ring neighborhood
    Average the position of the current vertex with the vertices from the one ring neighborhood
Return projected and smoothed mesh
```

Algorithm 15: Projection and smoothing.

4.3.2. Smooth wrap algorithm

Our shrink wrap algorithm at its core is a topological simplification algorithm. For applications that desire a smoother version of the output, we introduce the smooth wrap. The simple wrap result is subdivided until it reaches an average edge length.

Then, a smooth surface is reconstructed using a restricted Voronoi diagram based approach [55].

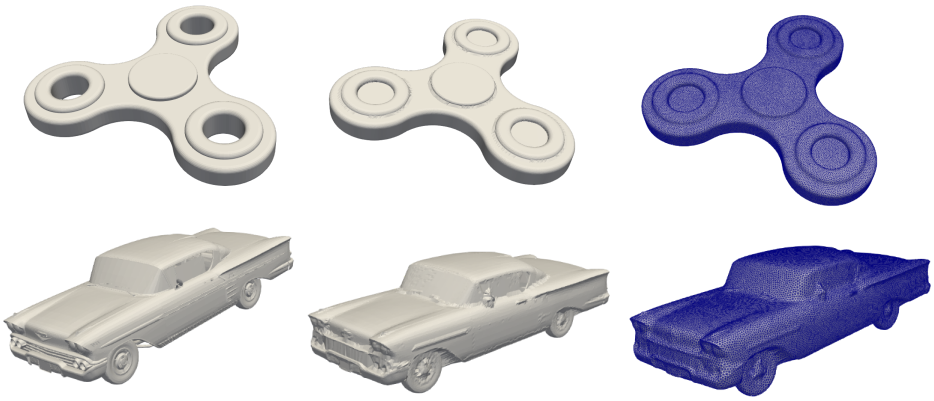


Figure 35: Input geometries along with their simple and smooth wrapped geometries.

Smooth Wrap

Result: Smoothly wrapped mesh

1: Determine median edge length of the wrapped surface mesh
2: Subdivide the simply wrapped mesh until all the edges are divided to the median edge length
3: Uniformly sample points on the surface
4: Smoothly reconstruct a surface using RVD based approach
5: Return smoothly wrapped mesh

Algorithm 16: Smooth wrap algorithm.

4.3.3. Developable wrap algorithm

Developable surfaces are valuable in architectural and manufacturing applications. To enhance the manufacturability of our wrapped geometries, we apply a static/dynamic filtering algorithm [56] as a post-processing step. This approach iteratively removes weaker features, resulting in a simplified geometry that retains only the most prominent features.

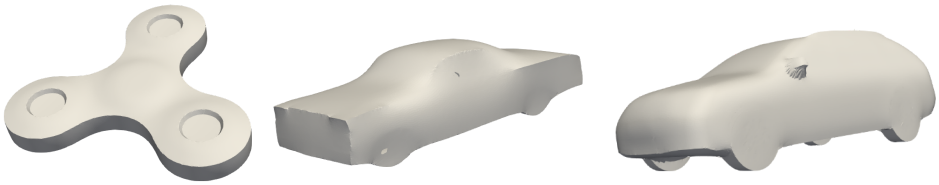


Figure 36: Various geometries filtered using static/dynamic filtering. The resulting geometries are developable.

4.4. Numerical Experiments

We perform experiments on a wide variety of input geometries that help underscore the robustness of our algorithm. Our algorithm produced a valid two-manifold surface mesh without any holes in all cases.

4.4.1. Effect of topological sphere level

The **Topological sphere level** is the only parameter in the algorithm. It can be increased or decreased depending on the size of the biggest hole in the geometry. Since the algorithm is fast, one can choose the value on a trial and error basis.

4.4.2. Effect on bad quality geometries

We show a car geometry in Figure 37 which is missing most of its bottom. Our algorithm produces a tight wrap even in this case. The skull geometry shown in Figure 38 has many non-manifold edges and disconnected components. Without any pre-processing, our algorithm produces a perfect two-manifold mesh without any leaks, as shown in Figure 39.

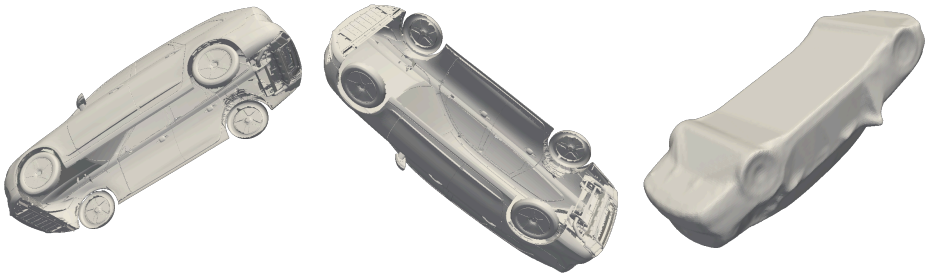


Figure 37: Bad quality geometries shrink wrapped (car with hole) - input mesh along with wrapped output mesh.

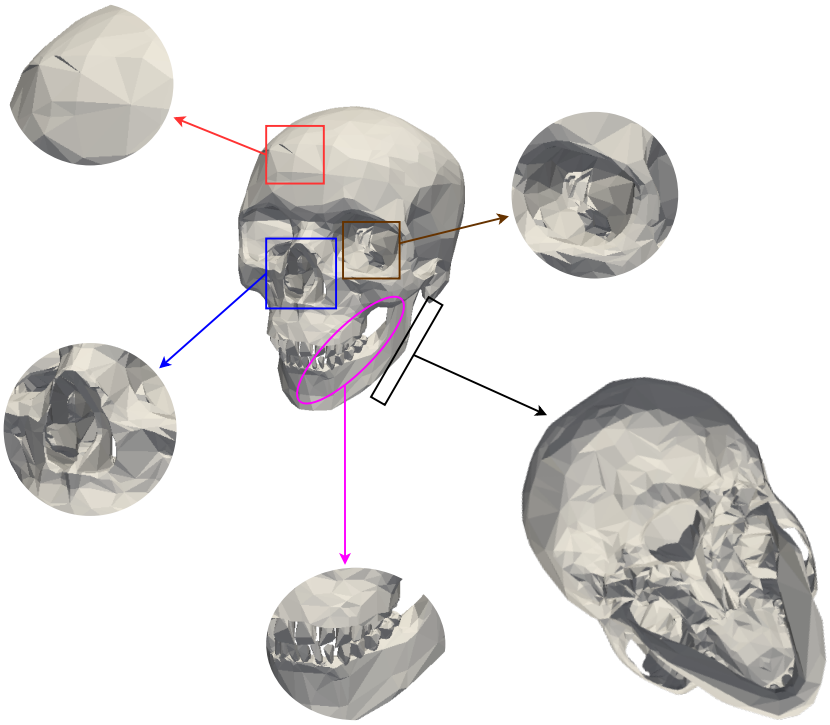
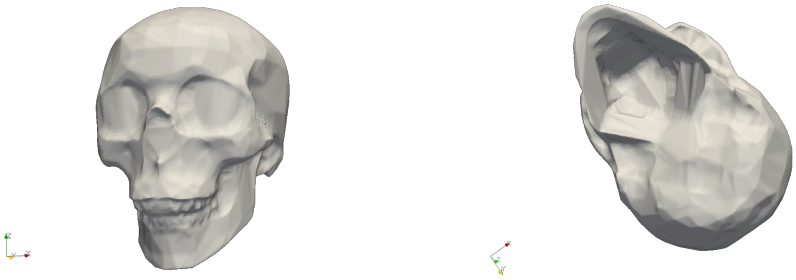


Figure 38: A human skull geometry with at least 15 non-manifold edges and many disconnected components. Various defects are highlighted.



Wrapped (Front View) Wrapped (Bottom View)

Figure 39: Wrapped skull geometry (watertight geometry with a genus zero).

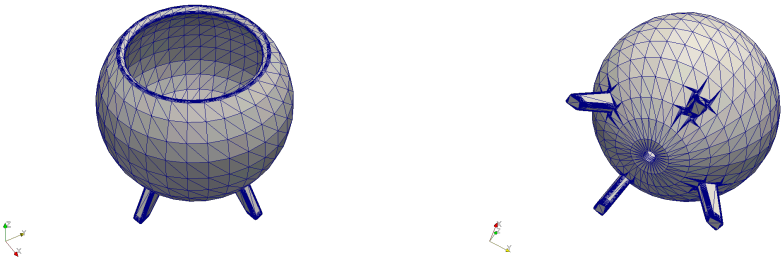
4.4.3. Selective Genus Closing

Almost all shrink wrapping algorithms are either used for remeshing (preserving genus) or surface simplification (genus zero). However, there are scenarios where

an industrial practitioner is only interested in closing selective holes. We propose two ways to do this.

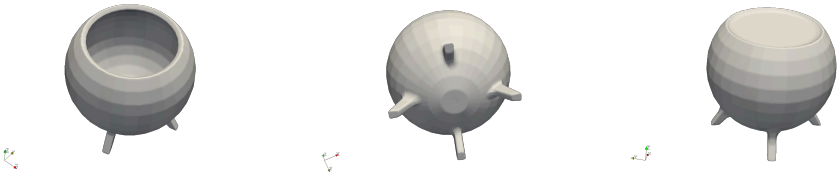
4.4.3.1. Implicit Genus control

A lower value for the topological sphere level usually leads to incomplete closing of the geometry. This can be a desirable side effect in the case of selective genus control. Figure 40 shows a geometry with a huge topological hole at the top and a smaller one at the bottom. The effects of different topological sphere levels are shown in Figure 41. A value of 50 yields a genus-zero surface; however, at level 10, only the smallest hole in the mesh is closed.



Bigger hole Smaller hole

Figure 40: An example geometry with a big hole on top and a small hole at the bottom.



Level 10 (top) Level 10 (bottom) Level 50

Figure 41: Shrink wrapped geometry for different topological sphere levels.

4.4.3.2. Explicit Genus control

Explicit genus control requires prior information about the topological holes in the mesh. Therefore, we use our topological hole detection algorithm [49] from Chapter 2. Topological hole information is extracted and used as a boundary condition in the dilation stage. The faces which are part of the desired holes are not diffused into the volume, thus preserving the structure. The dilation step in Algorithm 12 is modified as shown in Algorithm 17.

Surface Dilation with Genus Control

Input: Interior cells C_i, target level L_t, detected hole boundaries H
Result: Dilated surface cells C_d preserving holes

Detect holes using hole detection algorithm to obtain H
Compute blocked cells C_b = cells intersecting with H
Initialize seed set $S = C_i \setminus C_b$
Initialize current level $l = 0$
While $l \leq L_t$:
    Initialize next seed set $S' = \emptyset$
    For all cells $c \in S$:
        For all neighbours $n \in N(c)$:
            If n is exterior and $n \notin C_b$: $S' = S' \cup \{n\}$
    Set $S = S'$
    Increment $l = l + 1$
    If $l = L_t$: store S as dilated cells C_d
Return C_d

Algorithm 17: Dilation of the input surface (with genus control).

4.4.4. Runtime analysis

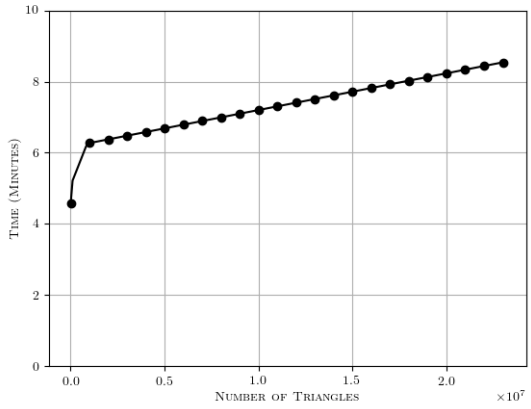


Figure 42: Run time comparison for increasing number of triangles.

The algorithm presented in this chapter is memory-bound. All results were obtained using a laptop equipped with a 12-core Intel Core i7 CPU and 64 gigabytes of RAM. Figure 42 illustrates the program runtime as a function of the number of triangles

in the input mesh. Our algorithm exhibits linear scalability. Even for the largest triangulation in our investigation, consisting of 23 million triangles, the algorithm converged to a manifold surface in under 10 minutes.

4.5. Applications and Variants

4.5.1. External aerodynamic simulation

One suitable area of application for shrink-wrapped geometries is external aerodynamic simulation. We ran RANS simulations on a generic shrink-wrapped car geometry using OpenFOAM [57]. The purpose of this example is limited: it demonstrates that the shrink-wrapped output can be accepted by a standard CFD workflow and can be used to generate a flow visualization without additional manual cleanup. It should not be read as a validation study of aerodynamic accuracy, since no validated reference geometry is used here for quantitative comparison.

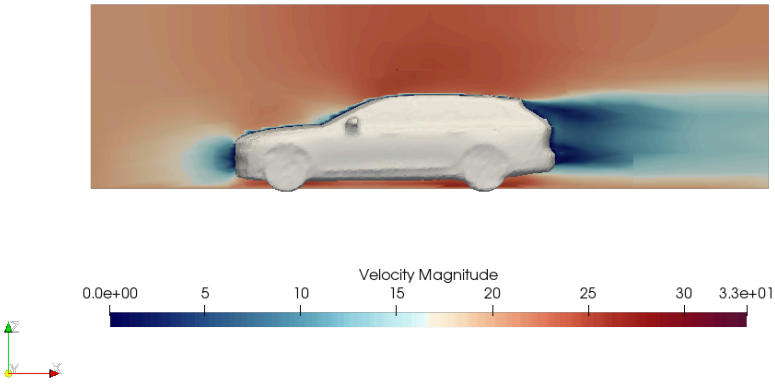
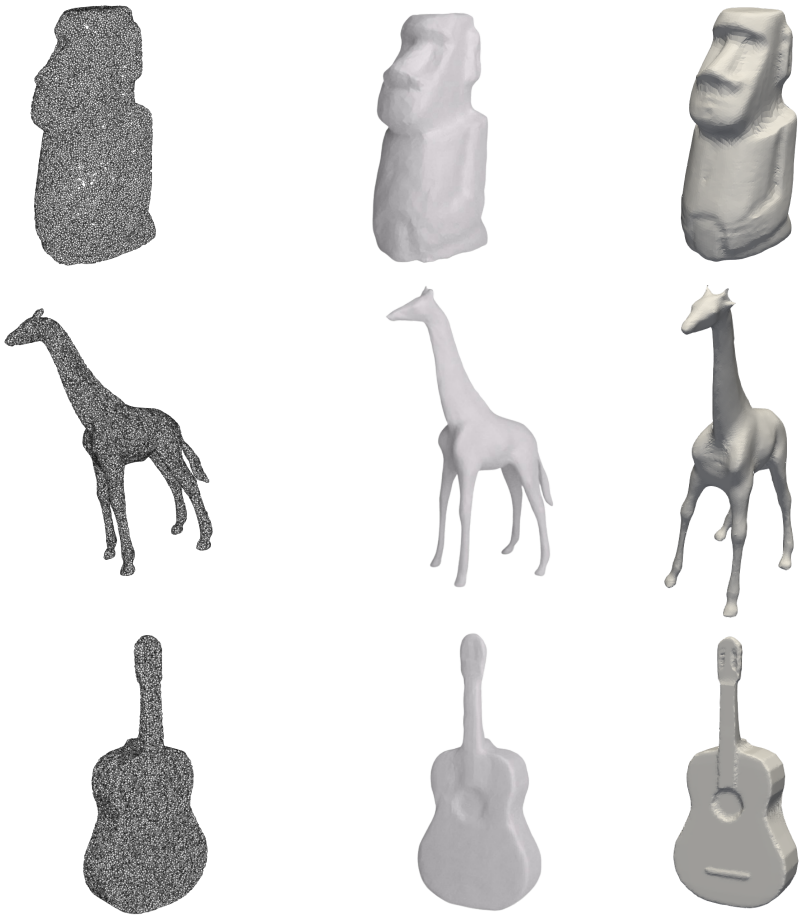


Figure 43: External aerodynamic simulation of a generic car model (shrink wrapped).

4.6. Comparison against similar approaches

4.6.1. Point2Mesh

We extended our shrink wrapping algorithm for point clouds to make a fair comparison against **Point2Mesh** [2]. Since we rely on generalized winding numbers for inside-outside segmentation, there is a straightforward extension to point clouds. We compute a series of local triangulations and consistently ensure their orientation using a greedy approach. Hence, our algorithm does not require an oriented point cloud. We found that we produce similar quality results in most cases. Our algorithm complements the authors' work nicely. If our algorithm is considered an initial **a priori**, it improves the convergence speed of the **Point2Mesh** algorithm.



Point Cloud *Point2Mesh* *Ours*

Figure 44: Few point cloud geometries from **Point2Mesh** [2] along with its output. Our wrap algorithm produces equally smooth results except for a few artifacts created as a result of morphological operators.

4.6.2. Alpha wrapping

Pierre Alliez et al. [48] recently proposed a practical shrink wrapping algorithm based on three-dimensional alpha wrapping. The term alpha controls the accuracy of the wrapping algorithm. For comparison, we show the effect of increasing the alpha value in Figure 45 for the same geometry used in Section 4.4.3.1. It can be noticed that while increasing alpha closes the bigger hole, the bottom of the geometry gets severely distorted. Our algorithm closes both the holes of the geometry, as shown in Figure 41, without heavily distorting the bottom of the geometry. While

alpha wrapping is an excellent surface reconstruction algorithm, it is not suitable for producing a hole-free geometry approximation.

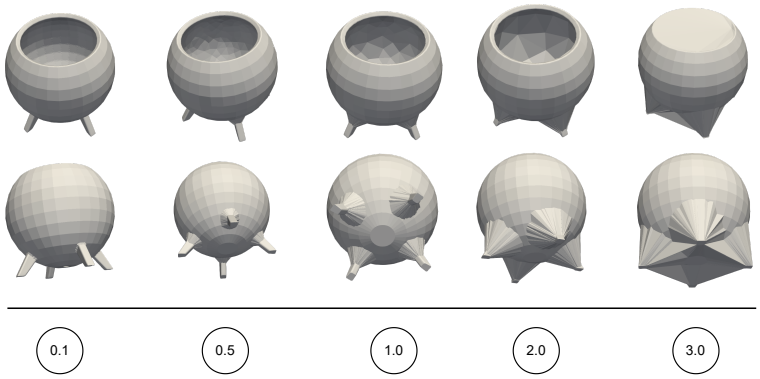


Figure 45: Influence of increasing alpha value on a geometry. The offset value is fixed at 0.000001 for all the cases.

4.7. Conclusion

This chapter presented a practical algorithm for genus-simplified shrink wrapping of polyhedral surfaces using morphological operators. The method converts an imperfect boundary representation into a volumetric representation, applies dilation and erosion to obtain a simplified offset surface, and then projects the result back toward the input geometry. The same formulation also extends naturally to point clouds through local triangulation and winding-number-based inside-outside classification.

The examples demonstrate that the method can close gaps, remove internal structures, and produce watertight manifold outputs for defective geometries while retaining explicit control over genus when hole information is available. The OpenFOAM example further illustrates a suitable area of application: shrink-wrapped outputs can be accepted by a standard CFD preprocessing workflow without additional manual cleanup. This example is intended as a workflow demonstration, not as a quantitative aerodynamic validation.

The algorithm is memory-bound and scales approximately linearly with the input size in the tested cases. These properties make morphological shrink wrapping a practical preprocessing tool for simplifying complex industrial geometries before downstream meshing, visualization, or simulation-oriented workflows.

5

Heat Method Based Mean Camber Line Extraction

Abstract. We propose an algorithm to extract the mean camber line of an airfoil using the heat method. By solving the heat equation on a Cartesian mesh inside the airfoil geometry, we obtain a smooth scalar field whose gradient reveals the mean camber line. This method avoids the need for complex geometric constructions, such as Voronoi diagrams, within the airfoil geometry. It also does not require the airfoil coordinates to be evenly distributed on the upper and lower surfaces. The heat method approach is simpler to implement than computational geometry based methods and can be easily coupled with existing computational fluid dynamics codes. We demonstrate the efficiency, robustness, and flexibility of the algorithm on a wide range of airfoil geometries.

This chapter is based on: S. Vijai Kumar and C. Vuik. "Distance Field Driven Mean Camber Line Extraction Algorithm." Reports of the Delft Institute of Applied Mathematics*, Delft University of Technology, 2020. https://research.tudelft.nl/en/publications/distance-field-driven-mean-camber-line-extraction-algorithm*

5.1. Introduction

Airfoil selection is a preliminary but critical decision in the initial design of aerodynamic machinery such as aircraft, race cars, and turbines. Traditionally, aerodynamicists design airfoil contours using techniques such as thin airfoil theory or panel methods [58], complemented by wind tunnel testing [59]. For standard airfoils, such as the NACA or Eppler series, formulae for computing parameters like the mean camber line are readily available [60], [61]. However, manually computing the mean camber line of an arbitrary airfoil typically requires constructing a Voronoi diagram to pack a series of inscribed circles, a computationally intensive process

[62]. While there have been notable improvements in computing airfoil coordinates for a given camber [63], this investigation deals with the inverse problem: computing the mean camber line for any given airfoil coordinate distribution.

Recent years have seen an increased use of indirect and stochastic approaches in aircraft design. Studies by Sekar et al. demonstrate the capability of machine learning algorithms in airfoil selection and performance characterization [64], [65], [66].

The field of computational geometry has also seen an increase in indirect approaches. A notable work relevant to our investigation is the use of the heat equation to compute geodesic distances on surfaces [67]. Our investigation focuses on computing mean camber lines in airfoils without requiring the computation of complex geometric predicates. We use distance fields, computed on a Cartesian mesh inside the airfoil geometry, and perform a simple gradient computation to extract the mean camber line. The proposed algorithm is useful for reverse engineering the mean camber lines of arbitrary airfoil contours. It imposes no specific requirement on the uniformity of the point distribution and does not require any distinction between the upper and lower surfaces of the airfoil.

5.2. Methodology

The algorithm is built on top of distance fields, which are popular in various domains of computational fluid dynamics. We make use of the Fast Marching Method [68] to compute the distance field by solving the Eikonal wave equation on an arbitrary domain using a finite difference method [69]. The distance field is treated as a general scalar field. The gradient of this scalar field yields the mean camber line of an airfoil. While the algorithm can be generalized to extract the medial axis of arbitrary domains for applications like mesh generation [69], this chapter focuses exclusively on the extraction of mean camber lines from airfoil contours.

5.2.1. Theory

The Level Set Method is an extension of curvature-shortening flow [70], where a curve is diffused down to a singularity. Its primary advantage is the ability to compute a signed distance field, where the sign distinguishes the inside from the outside of a closed curve. Sethian [68] utilized a stationary Eikonal wave equation for computing the distance field via the Fast Marching Method. The distance field `d` is computed by solving:

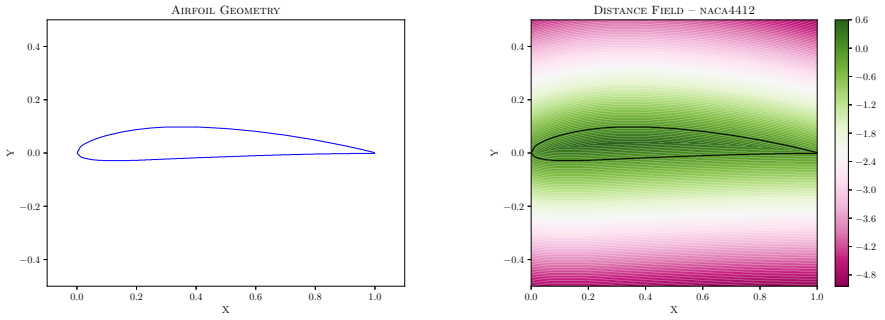
$$F\ |\nabla\varphi| = 1 + \varepsilon\nabla^2\varphi$$

where `F` is the velocity and `phi` describes the arrival time of the wavefronts. By setting `epsilon` to zero, it becomes the boundary value formulation for the front tracking problem. This equation can be solved using an upwind finite difference

scheme on a 2D Cartesian mesh. This particular distance field can also be computed with the diffusion/heat equation [67], [71].

5.2.1.1. Distance Field Computation using the Fast Marching Method

We project a closed polygon (representing the airfoil) onto a Cartesian mesh. The cells that intersect the boundary of the polygon are assigned as the zero-level set. We assign an artificial velocity to this zero-level set, and the Eikonal wave equation is solved inside the domain. The result is a distance field inside the airfoil, as shown in Figure 46.



Input airfoil geometry.

Distance field computed using the fast marching method.

Figure 46: Distance field computation steps.

5.2.1.2. Distance Field Computation using the Heat Equation

A newer approach for approximating the distance field is with the heat/diffusion equation. The problem reduces to a simple heat conduction problem inside the airfoil. The boundary nodes are assigned a high temperature, and a null value is assigned everywhere else. The heat conduction equation is then solved inside the airfoil using the finite difference method:

$$\frac{\partial T}{\partial t} = k\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right)$$

The resulting scalar field, representing the temperature, resembles the distance field produced by the Fast Marching Method. The workflow is described in Algorithm 18.

Approximated Distance Field via Heat Equation

Input: Airfoil boundary on a Cartesian mesh, thermal parameters (`k`, `sigma`)
Result: Approximated distance field

1: Set temperature value `T = 1` on the cells marked as boundary cells in the input Cartesian mesh representing the airfoil geometry
2: Set temperature `T = 0` elsewhere in the mesh
3: Since the temperature is obtained using an explicit finite difference scheme, stability needs to be enforced by choosing appropriate values for `k` and ΔT
4: Choose `k = 1` and a CFL number (or) $\sigma = 0.25$
The value of ΔT is computed using:
5:
$$\Delta T = \frac{\sigma \cdot h^2}{k}$$
6: While the temperature of every cell has not reached a value greater than or equal to 0.1:
7: For all cells in the interior of the airfoil:
Compute the temperature using the finite difference method:
8:
$$T_{i,j}^{t+1} = T_{i,j}^{t} + \Delta T k \left(\frac{T_{i,j-1}^{t} + T_{i-1,j}^{t} - 4T_{i,j}^{t} + T_{i+1,j}^{t} + T_{i,j+1}^{t}}{h^2} \right)$$
9: Return approximated distance field

Algorithm 18: Approximated distance field like scalar field using heat equation (explicit FDM).

5.2.2. Algorithm

The algorithm for extracting the mean camber line is based on the distance field generated inside the airfoil. The algorithm itself is rotationally invariant; however, the computed mean camber line can be transformed with simple affine transformations. If the airfoil is parallel to the X or Y axis, the straightforward algorithm Algorithm 19 can be followed. Once the airfoil geometry is projected to a Cartesian mesh, the distance field is computed inside. The gradient of the distance field along the chord-wise direction yields the mean camber line.

Mean Camber Line Extraction (Axis-Aligned Airfoil)

Input: Airfoil geometry (line segments or ordered point set) on a Cartesian mesh
Result: Vector of line segments

1: Read airfoil geometry as line segments or a vector of points
2: Compute the concave hull of the airfoil coordinates
3: Create a **Cartesian mesh** of a size greater than the axis-aligned bounding box of the airfoil geometry
4: For all edges of the airfoil:
5: Find intersecting cells in the Cartesian mesh and mark them as boundary cells
6: Set the intersecting cells as the zeroth level set
7: Compute the distance field inside the airfoil geometry using the fast marching or heat method
8: Compute the gradient along the chord-wise direction (X or Y)
9: The result is a mask on the Cartesian mesh where cells forming the mean camber line are marked
10: Group the marked cells into a set of connected edges to form the mean camber line
11: Return vector of line segments

Algorithm 19: Algorithm to extract mean camber lines in airfoils (airfoil parallel to a global axis).

For airfoils that are not parallel to either axis, an oriented bounding box can be used to find the angle of rotation. The airfoil can be temporarily rotated to be parallel with an axis, processed, and then rotated back, as described in Algorithm 20.

5.3. Theoretical Results

Some details of the algorithm are best explained through discrete theoretical arguments.

5.3.1. Definitions

Let $S = (p_0, p_1, \ldots, p_{n-1})$ be an ordered cyclic list of 2D points describing the airfoil boundary. Let $P(S)$ be the closed polygon obtained by connecting consecutive points $p_i \to p_{i+1}$ (indices modulo n). Assume $P(S)$ is simple (no self-intersections), and let Ω denote its interior.

Mean Camber Line Extraction (Arbitrarily Oriented Airfoil)

Input: Airfoil geometry (line segments or ordered point set)
Result: Vector of line segments

1: Read airfoil geometry as line segments or a vector of points
2: Compute the concave hull of the airfoil geometry
3: Compute the oriented bounding box of the airfoil
4: Get the orientation angle θ of the bounding box with respect to the X and Y axes
5: Rotate the airfoil by $-\theta$ to align with an axis
6: Repeat steps 3 to 9 from the previous algorithm
7: Rotate the computed mean camber line back by θ
8: Return vector of line segments

Algorithm 20: Algorithm to extract mean camber lines in arbitrarily oriented airfoils.

Let u be the unit chord-wise direction.

For a scalar field d sampled on a Cartesian grid with spacings $\left(h_x, h_y\right)$, define the chord-wise directional derivative at grid node (i, j) using a central difference stencil. If $u = \left(u_x, u_y\right)$ and $d_{i,j}$ denotes the field value at the grid node, then

$$\left(\partial_u d\right)_{i,j} = u_x \cdot \frac{d_{i+1,j} - d_{i-1,j}}{2h_x} + u_y \cdot \frac{d_{i,j+1} - d_{i,j-1}}{2h_y}.$$

In the implementation, the magnitude $|\partial_u d|$ is used for ridge-like responses.

5.3.2. Proposition 1: Uniform Point Spacing Is Not Required

Proposition. A planar airfoil geometry defined by a discrete set of boundary points does not require a uniform coordinate distribution.

Justification. Let S_1 be a fine ordered point set and let $S_2 \subset S_1$ be a sparser subset that preserves cyclic order. The polygonal chains $P(S_1)$ and $P(S_2)$ represent the same embedded boundary as long as decimation does not introduce large geometric shortcuts. Concretely, assume the two polygons are close in the Hausdorff sense,

$$d_{H(P(S_1),P(S_2))} \leq \varepsilon,$$

and that the major bending features are preserved, e.g., by ensuring that high-turning vertices in S_1 are retained in S_2 (equivalently, the turning angles at retained vertices differ by at most a small tolerance). Under these conditions, $P(S_2)$ differs from $P(S_1)$ only by a narrow boundary perturbation of thickness $O(\varepsilon)$, while the

cyclic ordering and the enclosed region are preserved. Therefore, uniform spacing is not required; what matters is geometric approximation quality and preservation of the boundary order and strong turns.

5.3.3. Proposition 2: Mean Camber Computation Does Not Require Top-Bottom Boundary Separation

Proposition. Distance-field based mean camber line computation does not require any distinction between the top and bottom of the airfoil.

Justification. The boundary is treated as a single closed polygon $P(S)$. Both distance constructions below impose a uniform boundary condition on the entire boundary, without separating it into parts.

For the Euclidean distance-to-boundary field,

$$d(x) = \min_{y \in P(S)} \|x - y\|,$$

the minimization is over the entire boundary set, so any partition of the boundary into "top" and "bottom" is irrelevant.

For PDE-based approximations, the boundary is also treated uniformly. In Fast Marching, the field is the solution of the Eikonal equation with a single boundary condition,

$$|\nabla d| = 1 \text{ in } \Omega, \quad d = 0 \text{ on } \partial\Omega.$$

In the heat method, one computes a heat diffusion step with a single boundary condition on the whole boundary, and then post-processes the result to obtain a distance-like field. Since the boundary data is applied identically everywhere on $\partial\Omega$, the resulting scalar field is invariant to any top/bottom labeling. Hence, no explicit distinction between top and bottom surfaces is required.

5.3.4. Proposition 3: Directional Derivatives of the Distance Field Recover the Mean Camber Line

Proposition. The chord-wise directional derivative of a distance field produces the mean camber line of an airfoil.

Justification. The relevant geometric structure is the center set of the airfoil interior, which is captured by the medial-axis-like locus where the distance-to-boundary field has multiple closest boundary points. Informally, for a typical airfoil and a chord-wise direction u, most chord-wise rays intersect the interior such that the closest boundary point switches once from one side of the airfoil to the other. The switch occurs near points that are approximately equidistant to the two sides, which is the defining property of a centerline.

In the Fast Marching setting, the computed field approximates the viscosity solution of the Eikonal problem, so its level sets propagate inward from the entire boundary. Where the inward fronts meet (i.e., where multiple boundary influences arrive simultaneously), the field forms a ridge-like non-smooth region corresponding to the medial structure. In the heat method setting, the post-processed distance-like field exhibits the same qualitative behavior: diffusion from the whole boundary produces a scalar field whose steepest changes align with the locus that is maximally separated from the boundary.

Now consider the discrete directional derivative $\partial_u d$ computed by central differences on the Cartesian grid. Away from the center locus, the nearest boundary feature is stable under small chord-wise motion, so d changes smoothly and $|\partial_u d|$ remains moderate. Near the center locus, the identity of the closest boundary feature changes rapidly along chord-wise motion, producing a locally maximal response in $|\partial_u d|$. Therefore, grid cells (or nodes) where $|\partial_u d|$ is locally maximal along chord-wise neighborhoods trace a connected path near this medial branch, which is interpreted as the mean camber line.

In practice, selecting a threshold on $|\partial_u d|$ and enforcing connectivity (and optionally applying a thinning step) yields a single polyline that approximates the mean camber line.

5.4. Numerical Experiments

5.4.1. Implementation

The original prototype for these experiments was written in Python to iterate quickly. We reimplemented the workflow in *tgLang* for stronger typing, clearer intent, and reproducible runs that integrate directly with the geometry and grid libraries used throughout this thesis. This code can be executed in tgLang Studio [72], a web-based integrated development environment.

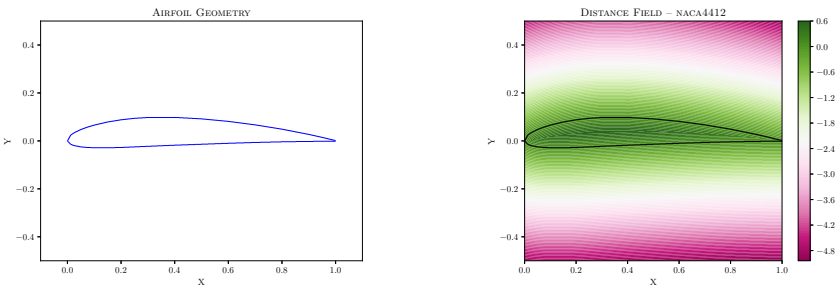
```
import "utils/os" as os;
import "math" as math;
import "shape/2d/curve" as s2c;
import "grid/fdm" as fdm;
import "geometry/curve" as curve;

// Quadratic subcell peak refinement (j-1, j, j+1 valid).
func quadPeakDelta(double fM1, double f0, double fP1) -> double {
    double denom = 2.0 * (fM1 - 2.0 * f0 + fP1);
    if (math::abs(denom) < 1e-14) { return 0.0; }
    double x = (fM1 - fP1) / denom;
    // Clamp for stability
    if (x < -0.5) { x = -0.5; }
    if (x >  0.5) { x =  0.5; }
    return x;
}

// Camber from maximal interior distance per x-column.
func extractCamber(fdmGrid<2> g,
```

```
                    array phi,
                    array<double,{2}> origin,
                    array<double,{2}> spacing) -> collection {
    array dims = g.dims();
    int nx = dims.get(0);
    int ny = dims.get(1);

    double hx = spacing[0];
    double hy = spacing[1];

    collection pts;

    for i from 0 to nx - 1 {
        int found = 0;
        int jBest = 0;
        double bestVal = 0.0; // most negative

        for j from 0 to ny - 1 {
            double v = phi[i, j];
            if (v < 0.0) {
                if (found == 0) {
                    found = 1;
                    jBest = j;
                    bestVal = v;
                } else {
                    if (v < bestVal) {
                        bestVal = v;
                        jBest = j;
                    }
                }
            }
        }

        if (found == 1) {
            double jRef = jBest;
            if (jBest > 0 && jBest < ny - 1) {
                double fM1 = phi[i, jBest - 1];
                double f0  = phi[i, jBest];
                double fP1 = phi[i, jBest + 1];
                double dj = quadPeakDelta(fM1, f0, fP1);
                jRef = jBest + dj;
            }

            double x = origin[0] + i * hx;
            double y = origin[1] + jRef * hy;
            pts.append([x, y]);
        }
    }

    return pts;
}

func main() -> int {
    curveNetwork airfoil = s2c::naca6("63A012", 400);

    // Square grid around AABB center
    collection aabb = airfoil.aabb();
    array mn = aabb.get(0);
    array mx = aabb.get(1);

    double dx = math::abs(mx[0] - mn[0]);
    double dy = math::abs(mx[1] - mn[1]);

    double maxDim = dx;
    if (dy > maxDim) { maxDim = dy; }

    double length = maxDim * 1.2;
```

```
    array<double,{2}> origin = [
        0.5 * (mn[0] + mx[0]) - 0.5 * length,
        0.5 * (mn[1] + mx[1]) - 0.5 * length
    ];

    int n = 256;
    fdmGrid<2> g = fdm::uniform2d(origin, length, n);
    array<double,{2}> spacing = g.spacing();

    g.addShape(airfoil);
    array phi = g.distanceField("phi");

    collection pts = extractCamber(g, phi, origin, spacing);
    if (pts.size() < 2) {
        print("Camber empty (no interior samples).");
        return 0;
    }

    // Smooth + densify for zoom
    pts = curve::taubinSmooth(pts, 30, 0.5, -0.53);
    pts = curve::resampleUniformCurve(pts, 4096);

    curveNetwork camber = curve::buildPolyline(pts);

    plot p("Mean Camber (max interior distance)");
    p.addHeatmap("phi", phi);
    p.addCurveNetwork("airfoil", airfoil);
    p.addCurveNetwork("camber", camber);
    p.save("meanCamber.tgvis");

    return 0;
}
```

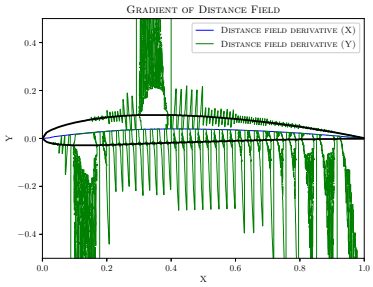
5.4.2. Analysis of Results

Figure 47 and Figure 48 show an overview of the algorithm's components. The algorithm has been tested on several airfoil geometries available in the literature, and their mean camber lines are shown in Figure 49. In most cases, the algorithm computes the mean camber line directly connecting the leading and trailing edges. The computed lines were compared with those from existing approaches [73] and show good agreement. The runtime of the algorithm is nearly instantaneous for typical airfoil geometries, and therefore a detailed runtime analysis has not been included.

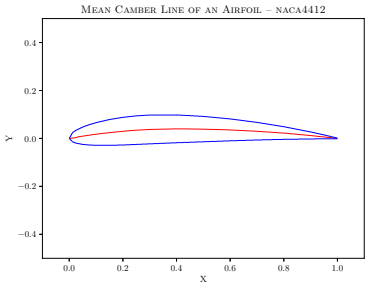


Airfoil geometry as input Distance field

Figure 47: Different steps of the algorithm (Step 1 and 2)



Derivative of distance field



Extracted mean camber line

Figure 48: Different steps of the algorithm (Step 3 and 4)

5.4.3. Extension for Mesh Generation

The algorithm can be generalized for volumetric decomposition, such as for structured multiblock grid generation. Similar investigations have been performed using the Fast Marching Method [69]. By computing the derivative of the distance field in both the X and Y directions, a closed polygon can be decomposed into smaller blocks, as shown in Figure 50.

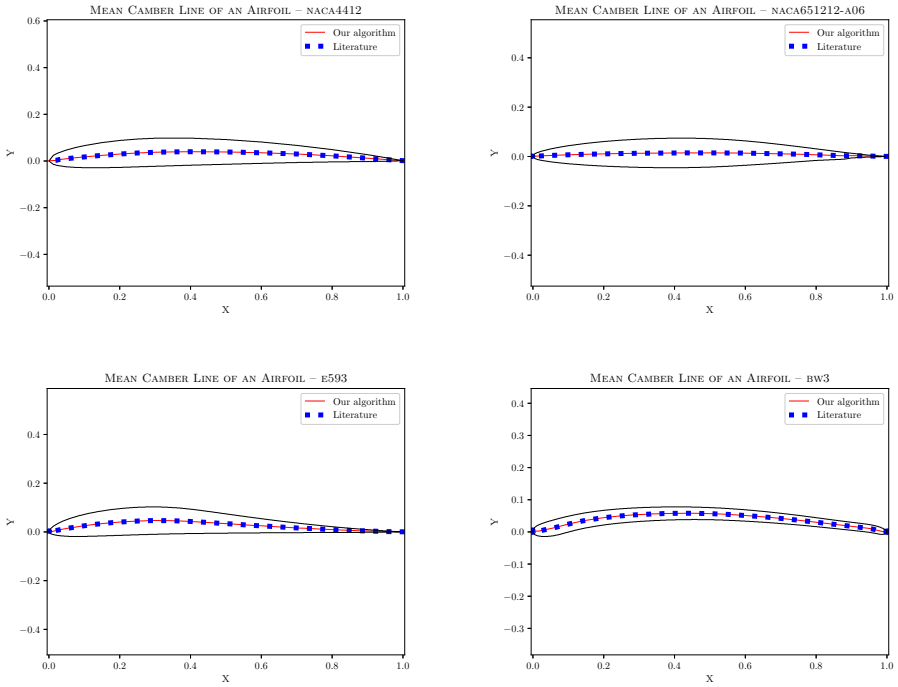


Figure 49: Mean camber line (red) computed on different airfoils (blue) and their comparison against existing approaches.

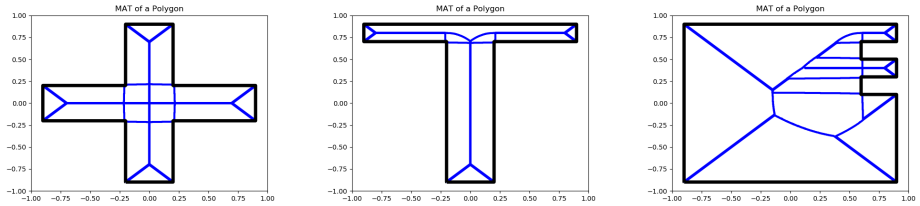


Figure 50: Different polygons and their block decompositions.

5.5. Conclusion

This chapter presented a fully automatic algorithm for computing the mean camber line of airfoils using distance fields. Since the computations are performed on a Cartesian mesh, the same discretization strategy can be integrated with CFD or FEA preprocessing workflows. The approach is useful for reverse engineering existing airfoil contours and for rapid prototyping of new ones. It does not require evenly distributed airfoil coordinates, explicit separation of upper and lower surfaces, or local curvature information.

The examples show that the distance-field formulation produces mean camber lines that agree well with existing approaches for a range of airfoil geometries. The same idea can also support related decomposition tasks, as illustrated by the block-decomposition example. Extensions to three-dimensional geometries, such as wings and turbine blades, and use in data generation for inverse-design workflows are promising directions for future work. The algorithm has been implemented in *tgLang*, making it accessible for rapid experimentation and integration into the geometry-processing workflows used in this thesis.

6

tgLang: A Programming Language for Geometry Processing

Abstract. We introduce *tgLang*, a strongly typed programming language for geometry processing that promotes rapid, reproducible experimentation while retaining a path to native deployment. By elevating core geometric entities such as `pointCloud` and `surfaceMesh` to first-class types and providing a familiar C-like syntax, *tgLang* reduces boilerplate and enables safe, concise implementations of stateful mesh algorithms. *tgLang* includes a bytecode virtual machine, a profiler, a precise generational garbage collector, and an LLVM-based native backend for distributable executables. Its compiler works on both native platforms and the web through WebAssembly. The language supports user-extensible modules and deterministic task-based parallel execution, enabling reproducible results without sacrificing performance.

6.1. Introduction

Anyone who has implemented geometry-processing algorithms has likely encountered a gap between the compactness of the mathematical description and the complexity of the executable workflow. The bottleneck is rarely the solver alone. It is often the mesh preparation, the ad hoc scripts, and the small but fragile pieces that hold the pipeline together. Real-world geometries from CAD are imperfect and require extensive manual repair and preparation, a process that is tedious and error-prone. A 2018 survey [74] confirmed this reality, showing that engineers spend a disproportionate amount of time preparing meshes rather than performing the actual analysis. Even when robust algorithms exist, their practical application typically involves labor-intensive scripting, parameter tuning, and ad hoc integration.

Powerful C++ libraries such as CGAL [15], OpenMesh [9], OpenVolumeMesh [75], geometry-central [76], and libigl [77] provide efficient data structures and a wide

range of geometry-processing algorithms. However, their template-heavy APIs, verbose iteration patterns, and low-level memory and topology management can hinder rapid prototyping and experimentation. For researchers, industrial practitioners, and educators, developing and testing new ideas often requires substantial boilerplate code and careful handling of mesh connectivity, ownership, and geometric assumptions, which slows down the research and development cycle.

To mitigate these issues, practitioners frequently turn to high-level scripting environments, for example Python-based tools such as PyMesh [78] or Open3D [79], to prototype algorithms quickly. While effective for early exploration, such approaches typically sacrifice performance, static guarantees, and deterministic behavior. In practice, this often creates two codebases: one research-oriented implementation used to explore the algorithm, and a second production-oriented implementation developed later for deployment. This separation slows down transfer from research to use, and it can also introduce subtle differences between the algorithm that was tested and the algorithm that is eventually shipped.

Domain-specific programming languages (DSLs) offer an alternative by raising the level of abstraction while preserving performance through compilation or optimized backends. Existing systems demonstrate the viability of this approach in specific contexts. Gmsh provides a mature scripting interface tailored to its meshing kernel [80]. Liszt [81] showed that high-level finite-element abstractions can be compiled to near-native code. More recent systems such as MeshTaichi [82] and I❤Mesh [83] explore compiler-based and symbolic approaches for mesh-centric computation. However, these languages primarily focus on numerical computation and traversal over fixed meshes and provide limited support for imperative, stateful operations such as topology modification, local remeshing, and morphological transformations. These operations are central to industrial geometry processing.

The algorithms developed in the earlier chapters of this thesis, including topological repair (Chapter 2), surface reconstruction from unstructured data (Chapter 3), application-driven mesh generation (Chapter 4), and distance-field-based feature extraction (Chapter 5), rely heavily on such stateful geometric operations. Their implementation exposed recurring challenges related to code complexity, reuse, and the gap between algorithmic descriptions and executable implementations.

This chapter introduces *tgLang*, a strongly typed domain-specific language designed to address these challenges directly. *tgLang* treats geometric entities such as surface meshes, volume meshes, grids, point clouds, and curve networks as first-class language constructs. By embedding these abstractions directly into the type system, the language allows geometry-processing algorithms to be expressed in a concise and natural style that closely mirrors their conceptual form, while enforcing key invariants at compile time.

tgLang combines a familiar C-like syntax with Python-like flexibility, lowering the barrier to rapid experimentation without sacrificing performance or safety. High-level programs use optimized runtime kernels for computationally intensive tasks, including linear algebra, neighborhood queries, and mesh operations, and can also be compiled to native executables through the LLVM backend. The compiler works on native machines as well as the web through WebAssembly, while execution is handled by a bytecode virtual machine with profiling support and a precise generational garbage collector. The language employs a deterministic task-based parallel execution model, producing reproducible results across runs. This property is essential for scientific experimentation, debugging, and industrial validation.

Many individual constructs in *tgLang*, such as arrays, functions, modules, and control flow, are intentionally conventional and could be implemented in a C++, Python, or Julia framework. The motivation for a dedicated DSL is not novelty in these generic constructs. Rather, the intended contribution is the combination of geometry-native types, controlled execution semantics, reproducible parallelism, interactive diagnostics, and multiple deployment backends under a single language-level model. In a framework, these concerns are typically distributed across library conventions, build systems, memory ownership rules, and external tooling. In *tgLang*, they are part of the programming model itself. This is the central advantage for geometry-processing practitioners: stateful mesh operations can be written close to their algorithmic form while the language runtime retains control over type invariants, execution order, profiling, and backend selection.

We evaluate *tgLang* with case studies in point cloud downsampling, curve-network analysis, grid operations, and surface and volume meshing, demonstrating reduced boilerplate and clearer invariant expression through static typing.

In summary, *tgLang* serves both as a standalone geometry-processing language and as a unifying framework for the algorithms developed throughout this thesis. By abstracting away incidental implementation complexity while preserving explicit control over stateful mesh operations and performance-critical kernels, *tgLang* bridges the gap between geometric algorithm design and practical, high-performance deployment.

6.2. Language Design

tgLang is a statically typed domain-specific language for geometry processing. Its design is guided by the following principles:

1. Explicit types that capture geometric intent and invariants.
2. A familiar syntax combining C-style structure with Python-inspired readability.
3. A standard library providing essential operations for all supported types.
4. Deterministic execution semantics suitable for reproducible computation.

This section introduces the core language constructs through representative examples. Additional examples covering array operations, geometry processing, and user-defined types are provided in accompanying scripts and appendices. Details of the parallel execution model are discussed separately.

6.2.1. Types

Although *tgLang* is domain-specific, it provides the standard data types expected of a general-purpose language. Primitive types such as `int`, `double`, and `bool` coexist uniformly with geometry-specific types such as `surfaceMesh`, `volumeMesh`, and `pointCloud`. This uniform treatment allows algorithms to combine numerical logic and geometric operations without artificial separation.

Unlike many DSLs that minimize explicit typing, *tgLang* enforces strict static typing. This choice reduces undefined behavior at runtime and enables geometric assumptions, such as dimensionality or valid connectivity, to be expressed directly through types.

Category	Types and Description
Primitive	`int`, `double`, `bool`, `string`, `none`
Collections	`array<dtype, rank|shape>` (resizable homogeneous arrays), `collection` (heterogeneous container), `table` (key value store with inferred types)
Geometry	`pointCloud`, `lineGrid`, `squareGrid`, `cubeGrid`, `fdmGrid<dim>`, `surfaceMesh`, `volumeMesh`, `curveNetwork`
I/O	`ifile`, `ofile`
Diagnostics	`error`
Visualization	`plot`
Extensibility	User-defined record types via `special`

Table 6: Built-in data types in *tgLang*.

The `array<dtype, rank|shape>` syntax is part of the language grammar and does not denote templates or generic programming. It serves as a compact notation for expressing dimensionality and layout. Arrays may be resized dynamically, subject to type consistency. In contrast, `collection` allows heterogeneous values and is intended for flexible aggregation rather than numerical computation.

6.2.2. Basic Constructs

Variable declaration and assignment follow familiar conventions. Types are explicit, assignments are strongly typed, and the language supports both single-line comments using `//` and multiline comments using `/* */`.

```
// Single-line comment
double a = 5.0;
int count = 3;

/*
 Multi-line comments may span
 multiple lines and are useful
 for documenting code blocks.
 */
bool flag = true;
string msg = "hello";

none result = none;
```

Tables provide convenient key value storage with inferred value types.

```
table shape = {
    "name"   : "circle",
    "radius" : 2.5,
    "center" : [1.0, 2.0]
};

double r = shape["radius"];
```

Conditionals. Conditionals use `if`, `elif`, and `else`, following conventional semantics.

```
if (mesh.faceCount() == 0) {
    print("empty mesh");
} elif (mesh.faceCount() < 100) {
    print("coarse mesh");
} else {
    print("detailed mesh");
}
```

Loops and Iteration. *tgLang* supports `for` and `while` loops. Extended iteration constructs are provided to match common geometry-processing patterns.

```
int i = 0;
while (i < mesh.edgeCount()) {
    if (mesh.isBoundaryEdge(i)) {
        break;
    }
    i = i + 1;
}
```

```
for i from 0 to mesh.faceCount() by 2 when (i != 6) {
    print("face", i);
}

for k from 0 through 3 { print("layer", k); }

array<double,{4}> v = [0.0, 1.0, 2.0, 3.0];
for (idx, val) : enumerate(v) {
    print(idx, val);
}
```

These constructs reduce index bookkeeping and align iteration closely with geometric intent.

Specials. The `special` construct defines user-defined record types with explicit fields and methods. Specials are not classes and do not support object-oriented features such as inheritance. They provide lightweight, explicitly typed abstractions with associated behavior.

```
special Rectangle {
    int width;
    int height;

    func init(int w, int h) {
        self.width = max(0, w);
        self.height = max(0, h);
    }

    func area() -> int {
        return self.width * self.height;
    }
}

Rectangle r = Rectangle(3, 4);
print(r.area());
```

6.2.3. Functions and Modules

Functions declare parameter and return types explicitly, enabling static type checking and clear interfaces. For arrays, explicit forms such as `array<double,{3}>` are the canonical type form. The shorthand `array` form in declarations or return types is syntactic sugar: the compiler still infers and enforces a concrete array element type and shape from usage.

```
func centroid(pointCloud pc) -> array<double,{3}> {
    array<double,{3}> c = [0.0, 0.0, 0.0];
    for p : pc {
        c = c + p;
    }
    return c / pc.size();
}
```

Modules are global namespaces that group related functionality. They may be extended after definition, allowing incremental growth without modifying existing code.

```
import "geometry/surface_mesh" as sm;

extendModule "geometry/surface_mesh" {
    func averageEdgeLength(surfaceMesh m) -> double {
        double s = 0.0;
        for e from 0 to m.edgeCount() - 1 {
            s = s + m.edgeLength(e);
        }
        return s / m.edgeCount();
    }
}

surfaceMesh m = sm::load("mesh.stl");
```

```
double avg = sm::averageEdgeLength(m);
```

6.2.4. Errors, Testing, and Debugging

tgLang provides a deterministic error model based on typed error objects. Errors may be thrown and handled using structured exception handling.

```
import "testing" as testing;

testing::assertTrue(1 < 2);

try {
    throw tgBoundsError("bad index");
} catch tgBoundsError {
    print("caught bounds error");
}

try {
    throw tgTypeError("invalid type");
} catch {
    print("caught generic error");
}
```

6.3. Execution Backends

Execution in *tgLang* is organized around two complementary backends. The primary semantic reference path is the *tgLang* Virtual Machine (tgVM), which executes stack-based bytecode with deterministic runtime behavior. In parallel, *tgLang* provides an LLVM-based native backend for ahead-of-time compilation to distributable executables. This section presents these backends in sequence, emphasizing their shared semantic model and their respective roles in the overall system.

tgLang initially employed a simple tree-walking interpreter to validate language semantics rapidly during early development. The system then transitioned to tgVM, where source programs are compiled into stack-based *tgLang Bytecode* (TGBC), serialized using *Concise Binary Object Representation* (CBOR) into `.tgb` modules, and executed by the VM runtime. This architecture separates front-end syntax processing from back-end execution while preserving precise source-level error mapping and deterministic parallel semantics.

6.3.1. Compilation and Execution Pipeline

The execution pipeline is lightweight and targets bytecode:

1. **Dependency Analysis.** All imports are resolved into a unified source stream, eliminating hidden compilation units.
2. **Lexing and Parsing.** An *Another Tool for Language Recognition* (ANTLR4) grammar [84], [85] drives an adaptive `LL(*)` (unbounded lookahead, left-to-right) parser, producing a concrete syntax tree that preserves full source structure.
3. **Bytecode Compilation.** The abstract syntax tree (AST) is lowered to stack-based bytecode with explicit locals, constants, and source maps.

4. **Optional Optimization.** Bytecode passes eliminate dead code and simplify constant expressions.
5. **VM Execution.** The tgVM executes bytecode and dispatches to native kernels where available.

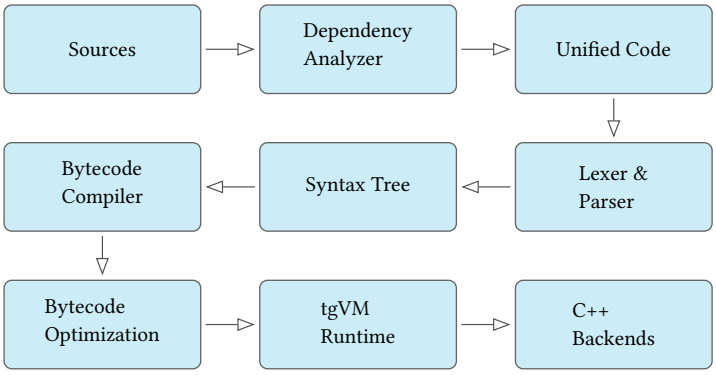


Figure 51: tgLang compilation pipeline from source input to VM execution and native backends.

6.3.2. tgVM: Execution Model and Runtime

The VM is stack-based with a fixed locals array per function. Each bytecode program contains a constant pool, a string table (identifiers), an instruction stream, and an optional source map for error reporting. Source maps attach `(file, line, col)` to instruction program-counter (PC) positions, enabling precise exception locations and disassembly. The runtime maintains modules, functions, and method registries, resolves overloaded calls by exact signature, and dispatches either to bytecode or to native kernels wrapped as bytecode stubs.

Specials (record types) are enforced at runtime: field layouts are validated, constructors run initializer bytecode, and field access is checked in the VM. Module imports and aliases are resolved into the runtime scope, keeping name resolution consistent across scripts and compiled modules.

6.3.3. Bytecode Instruction Set

The instruction set is compact and designed to keep execution inspectable. The VM includes a disassembler for `.tgb` files and records per-instruction source locations for error mapping. Table 7 lists a representative subset of instructions.

Opcode	Role
`PushConst i`	Pushes a constant from the constant pool.
`LoadLocal i`	Loads a local variable onto the operand stack.
`StoreLocal i`	Stores the stack top into a local variable.
`LoadGlobal i`	Loads a global/module value by name index.
`StoreGlobal i`	Stores a value into global/module scope.
`ImportModule i j`	Imports a module and binds an alias.
`CallFunction i n`	Calls a function by name and arity.
`CallModuleFunction m f n`	Calls a module-scoped function.
`CallMethod i n`	Invokes a type method on a receiver.
`FieldGet i`	Reads a field from a special/record object.
`FieldSet i`	Writes a field on a special/record object.
`IndexGet n`	Indexed read for arrays, collections, and tables.
`IndexSet n`	Indexed write for arrays, collections, and tables.
`Jump a`	Unconditional control-flow transfer.
`JumpIfFalse a`	Conditional branch on boolean false.
`Return`	Returns from the current function.
`Try a b`	Begins a guarded exception region.
`Throw i b`	Raises an exception category/value.
`EndTry`	Ends the guarded exception region.
`BuildVector a b`	Constructs a vector literal.
`BuildArray r n`	Constructs an array literal with rank/data.
`BinaryOp op`	Binary arithmetic/logical operation.
`UnaryOp op`	Unary arithmetic/logical operation.
`ParForRange a b c`	Deterministic parallel range loop.

Table 7: Representative tgVM bytecode instructions (TGBC).

6.3.4. Garbage Collection

tgVM includes precise, generational garbage collection (GC) for VM-managed objects. Each runtime owns a private heap to avoid cross-thread contention. Roots are enumerated precisely from the operand stack, locals, bytecode constants, module variables, and special initializers. A write barrier maintains a remembered set of old objects referencing young objects, enabling efficient minor collections.

The choice of garbage collection is a tradeoff. Geometry values such as meshes, point clouds, arrays, and tables are VM-managed objects, but their large internal buffers are held inside type-specific runtime data structures rather than being copied as ordinary scalar values. GC is used to manage the lifetime of these language-level objects, temporary values, records, closures, tables, and references that arise during scripting and interactive experimentation. During parallel execution, worker heaps are isolated and results are adopted back into the main heap by traversing the reachable object graph, avoiding deep cloning of returned structures. This design avoids manual lifetime management for the objects most likely to be created and discarded frequently, while still making the cost of managed execution explicit.

GC is opt-in via a runtime flag (exposed as `--garbageCollect` on the command-line interface), keeping overhead explicit for performance-sensitive runs. For very large batch workflows, disabling GC or using explicit object lifetimes may be preferable. The current implementation therefore treats GC as a usability and safety mechanism for exploratory programming, not as a replacement for careful memory management in memory-bound meshing kernels.

6.3.5. Deterministic Parallel Execution

Parallelism in *tgLang* is expressed through higher-order combinators and is implemented directly in the VM. Worker runtimes are cloned from the parent runtime, preserving module state and aliases. Each worker owns a private heap and call stack, so locals and temporaries remain thread-local. Work partitioning is explicit and deterministic, yielding repeatable results.

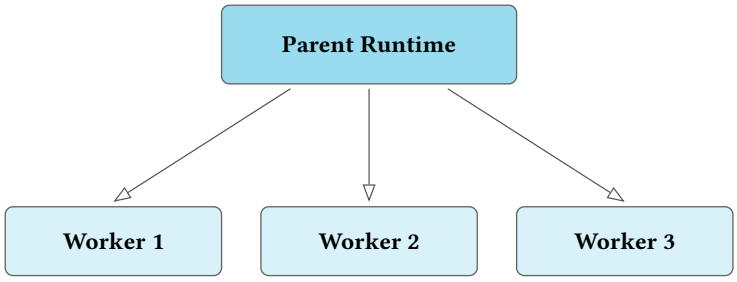


Figure 52: Parallel execution with VM worker runtimes and fixed data partitioning.

When GC is enabled, the VM enters a global no-collect scope for parallel regions. Results produced by workers are adopted into the main heap after the parallel region completes, preserving object identity without deep copying.

6.3.6. Interactive REPL

In addition to script-based execution, *tgLang* provides an interactive Read-Eval-Print Loop (REPL) for rapid exploration, debugging, and learning. The REPL supports multi-line input for complex statements and automatically prints the value of any expression not terminated by a semicolon. For introspection, it offers commands such as `:help`, `:modules`, and `:types`, allowing users to query available modules, functions, and type methods directly within the session. Integration with the underlying shell is provided through the `!` prefix.

```
~@ : tgLang

    ****************************************************
    *               >>> t g L a n g <<<                *
    * A DSL for scientific computing and mesh processing *
    ****************************************************

Interactive REPL mode. Type :help for commands, :q to exit.

>> array<double,{2,2,2}> T = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0] with(2 | 2 | 2);
>> array<double,{2,2,2}> A = T + 1.0;
>> print(T)
Syntax error at line 1, column 8: missing ';' at end of statement
print(T)
       ^
>> print(T);
array{dtype=double, shape=[2,2,2], size=8}
>> T
array{dtype=double, shape=[2,2,2], size=8}
>> A
array{dtype=double, shape=[2,2,2], size=8}
>> A[0, 0, 0]
2
>>
```

Figure 53: The *tgLang* REPL showing startup, multi-line input, and automatic printing of results.

This is useful during algorithm development because many geometry-processing workflows are built incrementally. A user can inspect available operations, test a small expression, load data, check intermediate values, and then move the working commands into a script once the workflow stabilizes. The REPL therefore supports the early exploratory phase without forcing a separate prototyping environment.

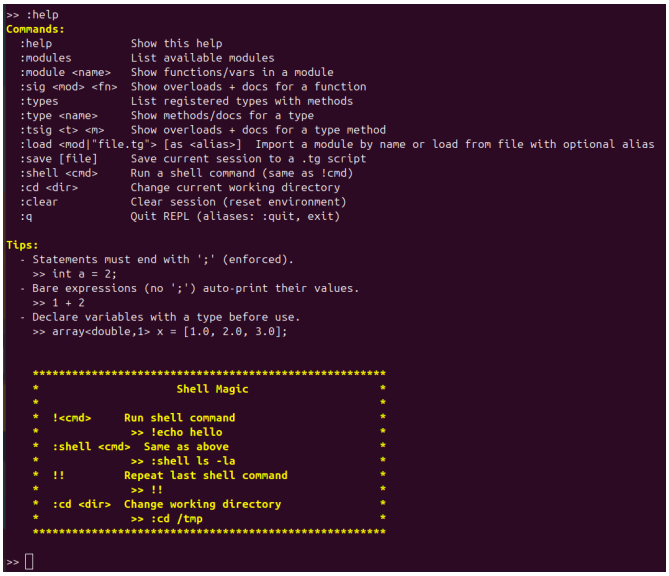


Figure 54: REPL introspection commands for listing modules and inspecting function signatures.

6.3.7. LLVM Native Backend

Beyond bytecode execution, *tgLang* provides an LLVM-based native backend that compiles programs ahead of time to distributable native executables. The native route is not a translation of tgVM bytecode. It reuses the stabilized parser and semantic analysis, but then lowers directly from the checked program representation to LLVM IR, with explicit runtime calls where language semantics are intentionally runtime-backed.

The tgLang LLVM backend is intentionally designed as a hybrid compiler rather than a full reimplementation of the VM runtime: it lowers core control flow, primitive numeric operations, typed arrays, direct calls, loops, and other stable language constructs to native machine code, while delegating dynamic language features and domain-specific objects (for example `surfaceMesh`, `pointCloud`, plotting, file abstractions, and module/type-method dispatch) to runtime entry points through a bridge layer. This boundary preserves correctness and development velocity for complex geometry-centric semantics that are impractical to model directly in LLVM Intermediate Representation (IR), while still enabling native executable generation for complete workflows. In practice, this architecture provides broad behavioral parity with the bytecode VM and a clear optimization path: keep rich object semantics in shared runtime code, and lower performance-critical regions natively where the compiler has sufficient type information.

To preserve language semantics in the native path, execution relies on a unified tagged-value application binary interface (ABI) shared between generated code and runtime services. Dynamic operations, including numeric operators, type conversions, indexed access, and predicate evaluation, are resolved through the runtime interface. In addition, module-level and type-method dispatch are mediated through a compatibility layer so that native execution remains consistent with the language-level module and type system.

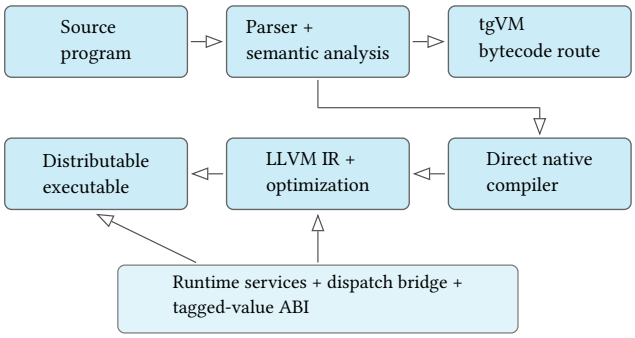


Figure 55: LLVM native backend architecture in *tgLang*, showing native lowering and runtime-dispatch integration.

As summarized in Figure 55, the native backend is evaluated as an execution path for complete programs rather than only as a collection of isolated compiler features. The validation corpus covers 180 scripts spanning scalar language features, arrays, vectors, modules, parallel constructs, PDE helpers, mesh and point-cloud operations, curve networks, grids, I/O, and volume-mesh examples. All 180 programs compile and execute through the native route, which supports the claim that *tgLang* can produce distributable executables for representative workflows.

This validation does not imply that every VM and native execution is observationally identical. In a separate parity audit, 143 of the 180 programs produced exactly matching stdout, while the remaining cases involved mainly numeric-output differences and object/string formatting differences. The current result is therefore best read as evidence of broad native execution coverage, with exact parity and further optimization remaining as engineering work.

6.3.8. Summary

The *tgLang* execution system comprises two complementary backends. The tgVM executes compiled bytecode with explicit locals, an operand stack, and a runtime registry for modules, functions, and methods. Source maps preserve precise error locations, while native kernels integrate seamlessly as bytecode entry points. Deterministic parallelism is implemented via cloned worker runtimes and fixed partitioning, with a precise, generational GC available when enabled. The LLVM

native backend provides a separate ahead-of-time compiler for distributable native executables, sharing the same semantic model through a unified tagged-value representation and runtime dispatch bridge. Together, this architecture keeps execution predictable while allowing users to move from interactive prototyping to native deployment.

6.4. Numerical Experiments

tgLang provides geometry-native types alongside numerical tensors, allowing algorithms to combine array-style computation with domain-specific types such as meshes and grids. This section demonstrates the use of the core types through representative examples.

6.4.1. Arrays

Arrays are first-class typed tensors supporting elementwise operators, broadcasting, slicing, masking, reductions, parallel maps, and solver compatibility.

Readable Array Math. High-level array expressions compile to efficient backends.

```
import "data/array" as arrays;
import "math" as math;
import "utils/time" as chrono;
func main() -> int {
    int N = 5000000;
    double dx = 1e-6;
    array<double,{N}> x = arrays::linspace(0.0, (N - 1) * dx, N);
    int t0 = chrono::nowMs();
    array<double,{N}> s = math::sin(x);
    array<double,{N}> y = (s ** 2.0) + (x * 2.0) + 1.0;
    int t1 = chrono::nowMs();
    print("readable array math ms:", t1 - t0, ", y[mid]=", y[N / 2]);
    return 0;
}
```

Parallel Array Maps. `parallel::mapArray` applies a function elementwise; `mapArrayCollect` aggregates multiple outputs.

```
import "data/array" as arrays;
import parallel;
import "testing" as testing;
func sq(double t) -> double { return t * t; }
func stats(double t) -> collection { return [t, t * t, t > 3.5]; }

test "array-parallel-examples" {
    array<double,{8}> x = arrays::arange(0.0, 8.0, 1.0);
    array<double,{8}> y = parallel::mapArray("sq", x);
    for i from 0 to 7 {
        testing::assertEqual(y[i], x[i] * x[i], 1e-12);
    }
    collection cols = parallel::mapArrayCollect("stats", x);
    array<bool,{8}> c2 = cols[2];
    testing::assertEqual(c2[3], false);
    testing::assertEqual(c2[4], true);
}
```

6.4.2. Point Clouds

Point clouds are native types for unstructured 3D points, with built-in support for neighborhood search via the nanoflann [54] library. It is straightforward to implement algorithms such as downsampling by aggregating points within voxel cells.

Voxel Grid Downsampling. This example uses a packed integer key for voxel bins and accumulates per-cell sums and counts.

```
// Pack 3 signed ints into one int key (assumes axis ranges fit after offset).
func packKey(int vx, int vy, int vz) -> int {
    int off = 524288;
    int hv = vx + off;
    int mv = vy + off;
    int lv = vz + off;
    return hv * 1048576 * 1024 + mv * 1048576 + lv;
}

func calculateVoxelKey(array point, double inv) -> int {
    int vx = (point.get(0) * inv).floor().toInt();
    int vy = (point.get(1) * inv).floor().toInt();
    int vz = (point.get(2) * inv).floor().toInt();
    return packKey(vx, vy, vz);
}

func downsamplePointCloudVoxels(pointCloud p, double voxelSize) -> pointCloud {
    pointCloud downsampled;
    table voxelTable;
    double inv = 1.0 / voxelSize;
    for point : p {
        int key = calculateVoxelKey(point, inv);
        double x = point.get(0);
        double y = point.get(1);
        double z = point.get(2);
        if (! voxelTable.contains(key)) {
            voxelTable[key] = [x, y, z, 1.0];
        } else {
            collection acc = voxelTable[key];
            voxelTable[key] = [acc[0] + x, acc[1] + y, acc[2] + z, acc[3] + 1.0];
        }
    }
    for entry : voxelTable.entries() {
        collection acc = entry[1];
        array centroid = [acc[0] / acc[3], acc[1] / acc[3], acc[2] / acc[3]];
        downsampled.addPoint(centroid);
    }
    return downsampled;
}
```

6.4.3. Surface Meshes

The `surfaceMesh` type supports polygonal geometry, connectivity queries, and mesh mutations. It also provides common discrete differential geometry operators, such as the cotangent Laplacian, as part of the type registry.

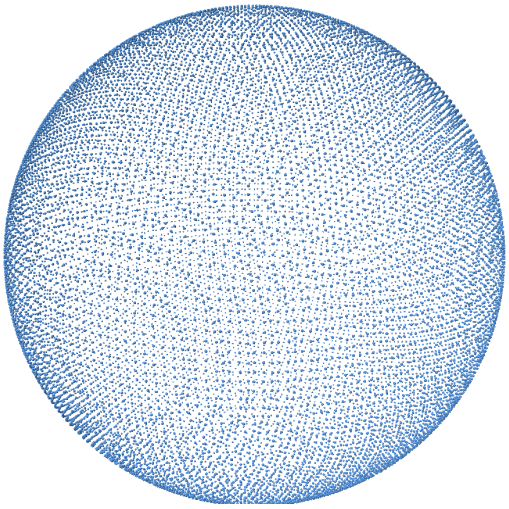


Figure 56: Voxel grid downsampling of a randomly sampled spherical point cloud

Isotropic remeshing A representative isotropic remeshing driver performs edge splits, collapses, flips, and tangential smoothing.

```
import "geometry/surface_mesh" as sm;

// The complete helper implementations are provided in the extended
// isotropic remeshing workflow later in this chapter.

func remesh(surfaceMesh m, double targetH, int iters, double featureAngleDeg) -> none {
    double splitRatio = 1.333; double collapseRatio = 0.8;
    for it from 0 to iters {
        int s = splitLongEdges(m, targetH, splitRatio);
        int c = collapseShortEdges(m, targetH, collapseRatio);
        int f = flipImprove(m, featureAngleDeg);
        tangentialSmooth(m, targetH, 0.2);
        print("iter", it, "s/c/f:", s, c, f);
    }
}
```

Taubin smoothing This example implements volume-preserving Laplacian smoothing using the cotangent Laplacian.

```
import "geometry/surface_mesh" as sm;

func applyDisplacement(surfaceMesh m, array D, double scale) -> none {
    int nV = m.vertexCount();
    for i from 0 to nV - 1 {
        if (m.isBoundaryVertex(i)) { continue; }
        array p = m.getPosition(i);
        array d = [D[i,0], D[i,1], D[i,2]];
        m.setPosition(i, p.axpy(scale, d));
    }
}

func taubinSmooth(surfaceMesh m, int iters, double lambda, double mu) -> none {
    for it from 0 to iters {
        array L1 = m.laplaceCotan(); applyDisplacement(m, L1, lambda);
        array L2 = m.laplaceCotan(); applyDisplacement(m, L2, mu);
```

```
    }
}
```

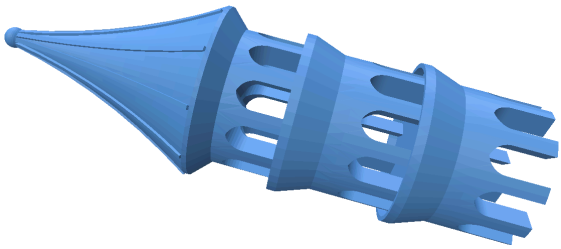


Figure 57: Taubin smoothing applied to a surface mesh, preserving volume while reducing noise.

6.4.4. Grids

Grids provide structured discretizations for 1D, 2D, and 3D domains. We provide `lineGrid`, `squareGrid`, `cubeGrid`, and `fdmGrid<dim>`. The `cubeGrid` is an adaptively refined Cartesian mesh which uses an octree for refinement and coarsening. It is powerful for solving PDEs on adaptively refined grids.

2D Heat Equation on a squareGrid. This example shows a simple heat diffusion solver on a uniform 2D grid with a central hotspot.

```
import "grid/square" as sg;
import "data/array" as arr;
import "utils/time" as chrono;
func main() -> int {
    double h = 1.0 / 32.0;
    squareGrid g = sg::uniform([0.0, 0.0], 1.0, h);
    g.clearBC();
    g.setDirichletAll(0.0);
    int N = g.cellCount();
    array T = arr::zeros("double", [N]);
    // hot spot at center cell (approximate: set first cell)
    T.set(0, 1.0);
    double alpha = 1e-3;
    double dt = 0.2 *(h * h) /(4.0 * alpha);
    int steps = 100;
    int t0 = chrono::nowMs();
    for s from 0 to steps - 1 {
        array S = g.applyFaceSum(T, h);
        array L = S *(1.0 /(h * h));
        T = T +(alpha * dt) * L;
    }
    int t1 = chrono::nowMs();
    print("squareGrid heat: ms=", (t1 - t0), " min=", T.min(), " max=", T.max());
    return 0;
}
```

3D Heat Equation on a cubeGrid. A 3D heat diffusion solver on a uniform `cubeGrid`, using an explicit Euler time-stepping scheme.

```
import "data/array" as arr;
import "utils/time" as chrono;
func main() -> int {
    double alpha = 1.0e-3;
    double L = 1.0;
    double dh = 1.0 / 16.0;
    double dt = 0.2 *(dh * dh) /(6.0 * alpha);
    int steps = 200;

    cubeGrid g;
    g.initializeUniform([0.0, 0.0, 0.0], L, dh);
    g.clearBC();
    g.setDirichletAll(0.0);
    int N = g.cellCount();
    array T = arr::zeros("double", [N]);
    int cId = g.locatePoint([L * 0.5, L * 0.5, L * 0.5]);
    if (cId >= 0 && cId < N) { T.set(cId, 1.0); }

    int t0 = chrono::nowMs();
    for s from 0 to steps - 1 {
        array S = g.applyFaceSum(T, dh);
        array Lh = S *(1.0 /(dh * dh));
        T = T +(alpha * dt) * Lh;
    }
    int t1 = chrono::nowMs();
    print("cubeGrid heat: ms=", (t1 - t0), " min=", T.min(), " max=", T.max());
    return 0;
}
```

6.4.5. Curve Networks

Curve networks represent embedded graphs or polylines, with support for construction, component analysis, and intersection splitting.

Polar Web Construction. This example procedurally builds a polar web, resolves edge intersections, and reports connectivity statistics.

```
import "data/array" as arrays;
import "math/constants" as constants;
import "math" as math;

func buildPolarWeb(int rings, int slices, double step) -> curveNetwork {
    curveNetwork net;
    int center = net.addNode(0.0, 0.0, 0.0);
    array ringIds = arrays::zeros("int", [rings, slices]);
    for r from 0 to rings {
        double radius = step * (r + 1);
        for s from 0 to slices {
            double theta = constants::TWO_PI * s / slices;
            double x = radius * math::cos(theta);
            double y = radius * math::sin(theta);
            int node = net.addNode(x, y, 0.0);
            ringIds[r, s] = node;
            if (r == 0) {
                net.addEdge(center, node);
            } else {
                int prevRadial = ringIds[r - 1, s];
                net.addEdge(prevRadial, node);
            }
            if (s > 0) {
                int prevSlice = ringIds[r, s - 1];
                net.addEdge(prevSlice, node);
            }
        }
```

```
        int first = ringIds[r, 0];
        int last = ringIds[r, slices - 1];
        net.addEdge(last, first);
    }
    return net;
}

func main() -> int {
    curveNetwork net = buildPolarWeb(4, 12, 0.35);
    int inserted = net.splitIntersections();
    net.buildConnectivity();
    array labels = net.componentLabels();
    print("intersections:", inserted, "components:", labels.max() + 1);
    return 0;
}
```

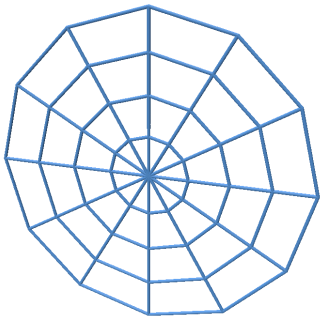


Figure 58: A procedurally generated polar web curve network after resolving self-intersections.

6.4.6. Runtime Performance

All scripts were executed on a machine with an Intel(R) Core(TM) i7-10850H processor and 96 GiB RAM. The tgVM column reports the reference execution path used for language semantics. The native column reports wall-clock runtime of the generated executable and does not include native compilation time. These numbers should be read as evidence that the LLVM route produces and runs native executables for representative workflows, not as a fully optimized backend benchmark. Some rows correspond to the examples shown above, while others are drawn from the same validation script set to cover additional language and runtime components.

Script	tgVM (ms)	Native runtime (ms)
Array math	120	210
Point cloud voxel downsampling (100k points)	2300	425
Hole detection on bracket STL (6.45 MB)	737	1270
Mean camber extraction (128 x 128 grid)	210	10
1D lineGrid heat equation (4096 cells, 2000 steps)	739	110
Cube grid sphere example	150	82
Curve network polar web	40	93

Table 8: Measured tgVM and native executable runtimes for representative scripts.

The hole-detection and mean-camber examples are included because they exercise application-level geometry workflows rather than only isolated language features. The current native route uses explicit runtime bridges for rich domain objects such as meshes, curve networks, grids, plotting, and file abstractions; extending bridge coverage and lowering more geometry-heavy operations directly are important optimization directions.

6.5. Why a Language Rather Than a Library?

A central design question throughout the development of *tgLang* was whether the system should be implemented as a standalone language or as a library embedded in an existing language such as C++, Python, or Julia. This question is important because, for many geometry-processing tasks, a library is indeed the simpler and more appropriate design. If the goal were only to expose a fixed set of geometry kernels, a library would be the natural choice.

The motivation for *tgLang* is different. It is aimed at complete geometry-processing workflows: loading data, applying geometry operations, testing intermediate assumptions, visualizing results, profiling runtime, and sharing the program with someone else. In a library-based design, these pieces usually live in different places. The host language controls the execution model, package environment, memory behavior, parallel semantics, error reporting, and deployment model. The geometry library provides useful data structures and algorithms, but the surrounding workflow still depends on external build systems, visualization tools, and user-specific conventions.

It is also important to separate the domain-specific motivation from the ordinary language machinery. Features such as user-defined `special` types, modules, containers, and familiar control-flow constructs are not, by themselves, the reason *tgLang* exists. Most individual geometry-processing algorithms could be implemented without them, and similar abstractions already exist in mature host languages. Their role in *tgLang* is more practical: they allow users to write mature programs rather than isolated scripts, and they let users extend the type system with more complex domain objects instead of depending entirely on types anticipated by the language designer.

The main advantage of the DSL is therefore workflow portability. Geometry entities such as `surfaceMesh`, `pointCloud`, `curveNetwork`, and `cubeGrid` are first-class program values with uniform diagnostics, serialization, visualization, and runtime support. The same source program can run locally, through the VM, through native compilation, or in the browser-based *tgLang Studio* environment without requiring the user to install anything.

This makes *tgLang* closer to an executable geometry-processing environment than to a conventional algorithm library. It does not limit *tgLang* to prototyping: programs can already be compiled to distributable native executables, allowing users to move from exploratory scripts toward production-ready code when required. The same compilation path can also be extended toward C ABI-compatible libraries, where a generated header and shared library (`.so`, `.dll`, or `.dylib`) could be embedded from C and C-compatible host languages. The language-level approach therefore provides a stronger abstraction boundary without giving up a route to conventional deployment: the workflow carries its types, execution model, visualization path, and deployment target with it.

6.6. Map-Reduce Point Cloud Downsampling

To demonstrate how *tgLang* expresses data-parallel geometry processing, we present a voxel-grid point cloud downsampling workflow in a Map-Reduce formulation. The objective is to replace all points inside each voxel cell by a single representative centroid while preserving the global shape. Conceptually, the map stage assigns each point to a spatial key (cell identifier), and the reduce stage aggregates per-key statistics (coordinate sums and counts), from which centroids are recovered. This example is both conceptual and executable. At the conceptual level, it instantiates a standard key-based reduction pattern over geometric data. At the practical level, it shows a complete workflow: domain construction, point-to-cell assignment, reduction by key, centroid reconstruction, and timing of each phase. The same pattern extends to many preprocessing tasks used in this thesis, including feature binning, neighborhood statistics, and coarse geometric summarization.

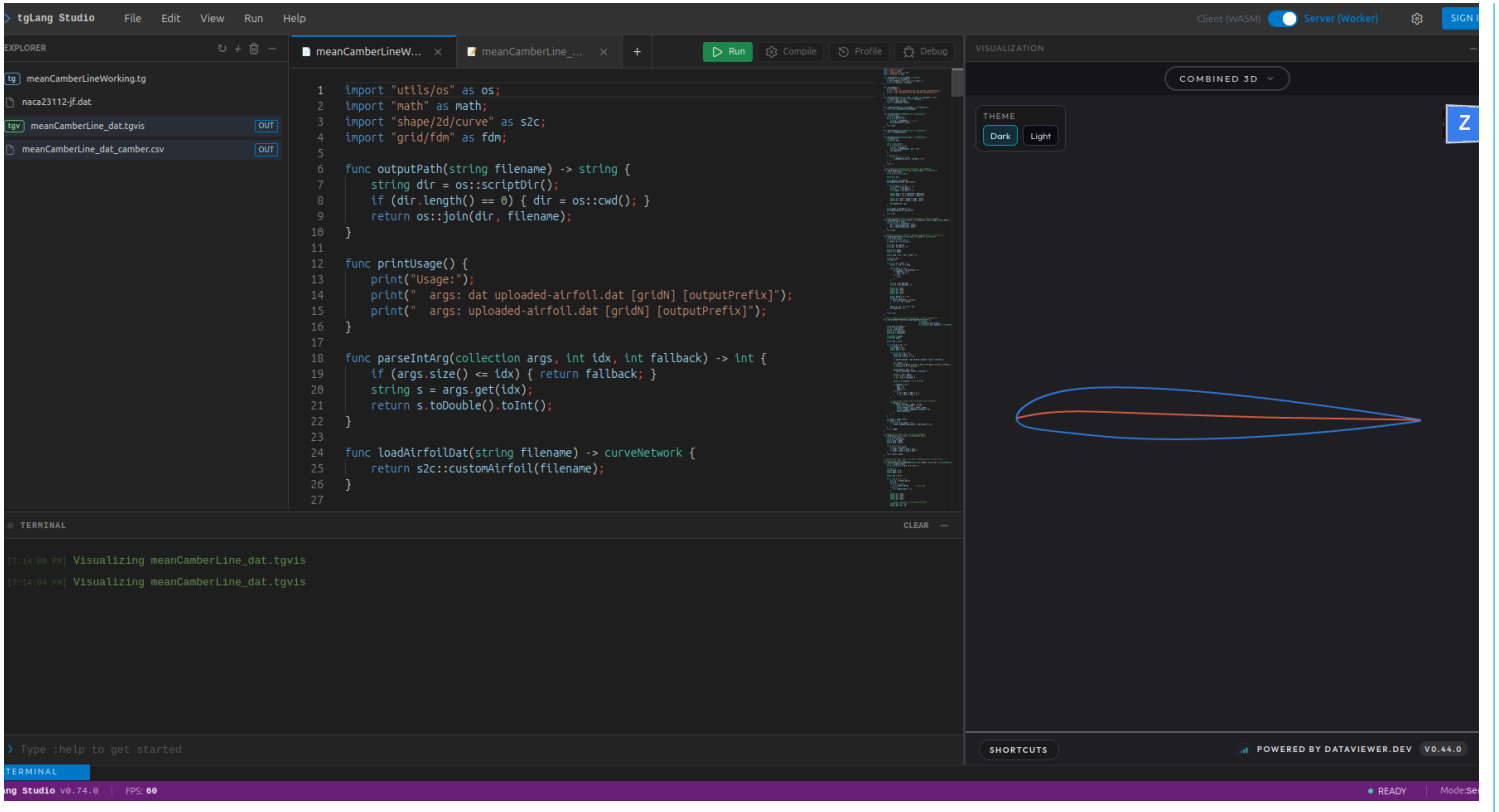


Figure 59: *tgLang Studio* running in the browser, showing source code, generated output files, terminal feedback, and an integrated geometry visualization panel.

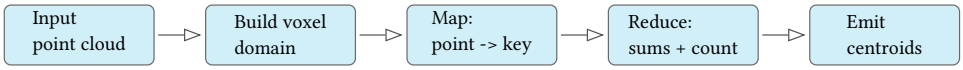


Figure 60: Map-Reduce downsampling workflow in *tgLang*: key assignment, per-key aggregation, and centroid reconstruction.

```
import "geometry/pointcloud" as pct;
import parallel as parallel;
import "utils/time" as chrono;
import "math" as math;

// Map: point -> [cellId:int, x, y, z]
func toCellRow(array p, cubeGrid g) -> collection {
    int id = g.locatePoint(p);
    collection row;
    row.append(id);
    row.append(p.get(0));
    row.append(p.get(1));
    row.append(p.get(2));
    return row;
}

// Map accumulated entry -> centroid [x,y,z]
func centroidRow(collection entry) -> array {
    collection r = entry[1];
    return [r[0] / r[3], r[1] / r[3], r[2] / r[3]];
}

func main() -> int {
    int t0 = chrono::nowMs();
    // Inputs
    pointCloud pc = pct::generateSphere(100000, 2.0, [0.0, 0.0, 0.0]);
    double voxel = 0.1;
    // --- Step 1: compute bounding box ---
    collection boundsData = pc.getBounds();
    array bbMin = boundsData[0];
    array bbMax = boundsData[1];
    double dx = bbMax[0] - bbMin[0];
    double dy = bbMax[1] - bbMin[1];
    double dz = bbMax[2] - bbMin[2];
    double length = math::max(dx, math::max(dy, dz));
    // Pad domain slightly to avoid boundary misses
    double eps = 1e-9 * length;
    array bbMinPad = [bbMin[0] - eps, bbMin[1] - eps, bbMin[2] - eps];
    double lengthPad = length + 2.0 * eps;
    // --- Step 2: uniform cubeGrid (no adaptation) ---
    cubeGrid g;
    g.initializeUniform(bbMinPad, lengthPad, voxel);
    // --- Step 3: map points to cell ids ---
    collection mapSrc;
    mapSrc.append(pc);
    mapSrc.append(g);
    collection rows = parallel::map("toCellRow", mapSrc);
    int t1 = chrono::nowMs();
    // --- Step 4: reduce by cell id to sums & counts ---
    table accum = parallel::reduceByKeySumCountInt(rows);
    int t2 = chrono::nowMs();
    // --- Step 5: centroids & emit ---
    collection centSrc;
    centSrc.append(accum.entries());
    collection centroids = parallel::map("centroidRow", centSrc);
    pointCloud out;
    for i from 0 to centroids.size() {
        out.addPoint(centroids.get(i));
    }
    int t3 = chrono::nowMs();
```

```
    print("grid cells:", g.cellCount(), ", voxels:", out.size());
    print("map(ms):", t1 - t0, ", reduce(ms):", t2 - t1, ", centroid(ms):", t3 - t2);
    return 0;
}
```

6.7. Isotropic Remeshing Workflow

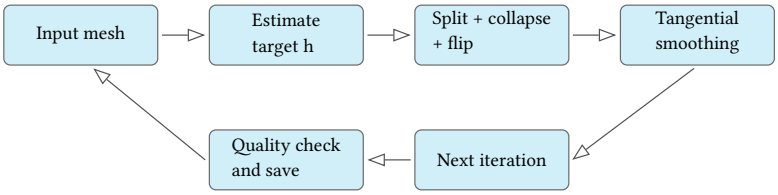


Figure 61: Isotropic remeshing workflow in *tgLang*: local topology edits, smoothing, and iterative quality-controlled updates.

Whereas the previous program emphasizes data aggregation, the next program emphasizes topology-aware surface improvement. It combines edge split, edge collapse, edge flip, and tangential smoothing into a compact isotropic remeshing loop. Together, these two programs illustrate complementary capabilities of *tgLang*: scalable data-parallel reduction on unstructured points and concise expression of iterative mesh-editing operators on surfaces.

```
import "geometry/surface_mesh" as sm;

// Basic isotropic remeshing on a triangle surfaceMesh
// Pipeline per iteration:
// 1) split long edges (> splitRatio * h)
// 2) collapse short edges (< collapseRatio * h)
// 3) flip edges to improve mesh quality
// 4) tangential smoothing (Laplacian step projected to tangent)

func estimateMeanEdgeLength(surfaceMesh m) -> double {
    int nE = m.edgeCount();
    double sum = 0.0; int cnt = 0;
    for e from 0 to nE - 1 {
        if (m.isBoundaryEdge(e)) { continue; }
        sum = sum + m.edgeLength(e);
        cnt = cnt + 1;
    }
    if (cnt == 0) { return 0.0; }
    return sum / cnt;
}

func splitLongEdges(surfaceMesh m, double h, double ratio) -> int {
    int nE = m.edgeCount(); int ops = 0;
    double thresh = ratio * h;
    for e from 0 to nE - 1 {
        if (m.isBoundaryEdge(e)) { continue; }
        double L = m.edgeLength(e);
        if (L > thresh) {
            int nv = m.edgeSplit(e);
            if (nv >= 0) { ops = ops + 1; }
        }
    }
    m.compact();
    return ops;
}
```

```
func collapseShortEdges(surfaceMesh m, double h, double ratio) -> int {
    int nE = m.edgeCount(); int ops = 0;
    double thresh = ratio * h;
    for e from 0 to nE - 1 {
        if (m.isBoundaryEdge(e)) { continue; }
        double L = m.edgeLength(e);
        if (L < thresh) {
            bool ok = m.edgeCollapse(e);
            if (ok) { ops = ops + 1; }
        }
    }
    m.compact();
    return ops;
}

func flipImprove(surfaceMesh m, double featureAngleDeg) -> int {
    int nE = m.edgeCount(); int flips = 0;
    for e from 0 to nE - 1 {
        if (m.isBoundaryEdge(e)) { continue; }
        double ang = m.edgeDihedralDeg(e);
        if (ang > featureAngleDeg) { continue; }
        bool ok = m.edgeFlip(e);
        if (ok) { flips = flips + 1; }
    }
    return flips;
}

func tangentialSmooth(surfaceMesh m, double h, double step) -> none {
    int nV = m.vertexCount();
    array L = m.laplaceCotan();       // double {nV,3}
    array N = m.vertexNormals();      // double {nV,3}
    double lam = step * h;
    for i from 0 to nV - 1 {
        if (m.isBoundaryVertex(i)) { continue; }
        double lx = L[i,0]; double ly = L[i,1]; double lz = L[i,2];
        double nx = N[i,0]; double ny = N[i,1]; double nz = N[i,2];
        double dot = lx*nx + ly*ny + lz*nz;
        double tx = lx - dot * nx;
        double ty = ly - dot * ny;
        double tz = lz - dot * nz;
        array p = m.getPosition(i);
        array q = [p[0] + lam * tx, p[1] + lam * ty, p[2] + lam * tz];
        m.setPosition(i, q);
    }
}

func remesh(surfaceMesh m, double targetH, int iters, double featureAngleDeg) -> none {
    double splitRatio = 1.3333333333;  // 4/3 h
    double collapseRatio = 0.8;        // 4/5 h
    double smoothStep = 0.2;           // relative step size

    for it from 0 to iters {
        int s = splitLongEdges(m, targetH, splitRatio);
        int c = collapseShortEdges(m, targetH, collapseRatio);
        int f = flipImprove(m, featureAngleDeg);
        tangentialSmooth(m, targetH, smoothStep);
        m.save("remeshed_" + it.toString() + ".stl");
        print("iter", it, " split:", s, " collapse:", c, " flip:", f, " V,E,F:",
        m.vertexCount(), m.edgeCount(), m.faceCount());
    }
}

func main() -> int {
    string in = "manga.stl";
    string out = "manga_remeshed.stl";

```

```
    // Load mesh
    surfaceMesh m; m = sm::load(in);
    print("loaded:", m.toString());

    if (m.faceCount() == 0 || m.vertexCount() == 0) { print("Mesh is empty or failed to
    load. Set a valid path."); return 0; }

    // Target edge length: estimate from current mesh if not set
    double h0 = estimateMeanEdgeLength(m);
    if (h0 <= 0.0) { h0 = 0.01; }
    double h = h0;
    print("estimated h:", h);

    // Remesh (set feature angle in degrees, e.g., 45.0)
    double featureAngleDeg = 45.0;
    remesh(m, h, 5, featureAngleDeg);

    // Save result
    bool ok = m.save(out);
    print("saved ->", out, " status:", ok);
    return 0;
}
```

6.8. Taubin Smoothing Workflow

Taubin smoothing is used here as a non-shrinking variant of Laplacian smoothing. A standard Laplacian update with only a positive step tends to contract the surface over repeated iterations. Taubin's idea is to apply two consecutive passes per iteration, first with a positive coefficient λ and then with a negative coefficient μ, so that high-frequency noise is attenuated while low-frequency shape drift is compensated. In operator form, each iteration applies $p \leftarrow p + \lambda L(p)$ followed by $p \leftarrow p + \mu L(p)$, where L denotes the cotangent Laplacian.

The implementation below follows this structure directly. The displacement field is computed from the current mesh, applied to all non-boundary vertices, then recomputed for the second pass after the first update. This is important because the second pass must act on the filtered geometry, not on stale Laplacian values. With typical choices such as $\lambda > 0$ and $\mu < 0$ with $|\mu|$ slightly larger than λ, the method produces smooth surfaces with substantially reduced shrinkage compared with one-pass Laplacian smoothing.

```
import "geometry/surface_mesh" as sm;

func vec3(double x, double y, double z) -> array { return [x, y, z]; }

func applyDisplacement(surfaceMesh m, array D, double scale) -> none {
    int nV = m.vertexCount();
    for i from 0 to nV - 1 {
        if (m.isBoundaryVertex(i)) { continue; }
        array p = m.getPosition(i);
        array d = vec3(D[i,0], D[i,1], D[i,2]);
        array q = p.axpy(scale, d);
        m.setPosition(i, q);
    }
}

func taubinSmooth(surfaceMesh m, int iters, double lambda, double mu) -> none {
```

```
    for it from 0 to iters {
        // First pass: p <- p + λ L(p)
        array L1 = m.laplaceCotan();
        applyDisplacement(m, L1, lambda);
        // Second pass: p <- p + μ L(p)
        array L2 = m.laplaceCotan();
        applyDisplacement(m, L2, mu);
        print("taubin it=", it, " V,E,F=", m.vertexCount(), m.edgeCount(), m.faceCount());
    }
}

func main() -> int {
    string in = "manga.stl";
    string out = "manga_taubin.stl";

    surfaceMesh m = sm::load(in);
    print("loaded:", m.toString());
    if (m.faceCount() == 0 || m.vertexCount() == 0) {
        print("Mesh is empty or failed to load. Ensure file exists at:", in);
        return 0;
    }

    // Typical parameters for non-shrinking smoothing
    int iters = 20;
    double lambda = 0.5;   // positive
    double mu = -0.53;     // negative, |mu| >= lambda

    taubinSmooth(m, iters, lambda, mu);

    bool enablePlot = false; // toggle to visualize in polyscope
    if (enablePlot) {
        plot viewer("SURFACE_MESH_TAUBIN");
        viewer.addMesh("smoothed", m);
        viewer.save("surfaceMeshTaubinSmoothing.tgvis");
    }

    bool ok = m.save(out);
    print("saved ->", out, " status:", ok);
    return 0;
}
```

6.9. Conclusion

This chapter presented *tgLang* as a domain-specific language built to close a long-standing gap in geometry processing practice. That gap lies between rapid experimentation and robust, reproducible execution. The core design decision is to treat geometry entities as first-class typed objects rather than external libraries attached to a generic scripting language. By integrating arrays, point clouds, surface meshes, curve networks, and grids into one coherent type system, *tgLang* supports concise program structure without sacrificing semantic clarity.

The implementation demonstrates that this design can be realized as a practical execution stack. tgVM provides deterministic bytecode execution with explicit runtime structure, source-level diagnostics, and predictable parallel behavior, while the LLVM backend provides ahead-of-time compilation to distributable native executables under the same semantic model. This hybrid architecture preserves rich dynamic and geometry-specific behavior through runtime services and bridge dispatch, while selectively lowering stable language constructs and performance-

critical regions to native code. The result is not only a conceptual language framework, but an operational system that supports end-to-end algorithm development and deployment.

The examples in this chapter show that *tgLang* can express both data-parallel and topology-modifying workflows in a compact, readable form, and the runtime measurements demonstrate that representative scripts can execute through both the VM and native routes. Together with the WebAssembly delivery path and tgLang Studio [72], the language is usable across local and browser environments without fragmenting semantics. In this sense, the contribution of *tgLang* is both methodological and practical. It provides a reproducible programming substrate for geometry-centric research and an engineering-ready path from prototype scripts to scalable execution.

The strongest case for *tgLang* is therefore not that it replaces mature general-purpose languages. Instead, it targets a narrower gap: executable, reproducible geometry-processing workflows in which meshes, grids, point clouds, and curve networks are first-class program values with predictable semantics. This scope is deliberately limited, but it is precisely the scope in which conventional scripts and low-level libraries often leave too much behavior implicit.

6.10. Future and Ongoing Work

Several directions naturally follow from the current *tgLang* implementation.

Native backend maturity. The LLVM backend already produces distributable executables for representative workflows, but the current design deliberately relies on explicit runtime bridges for rich domain objects such as meshes, grids, curve networks, plotting, and file abstractions. Future work should expand this bridge coverage, reduce remaining VM/native parity differences, and identify which geometry operations should remain runtime-backed and which should be lowered more directly. A useful next step is a systematic parity suite that compares not only stdout, but also generated arrays, mesh statistics, output files, and visualization metadata across tgVM and native execution.

Optimization and specialization. The current native route demonstrates executable generation rather than a fully optimized compiler pipeline. More aggressive specialization is needed for typed arrays, stencil-style grid operations, mesh traversal, and repeated method calls inside tight loops. This includes avoiding unnecessary tagged-value dispatch when static type information is available, improving array temporary elimination, and exposing more geometry kernels to the compiler as typed operations. These improvements would make the native backend more predictable for larger production workloads.

Library generation and interoperability. A natural extension of native executable generation is downloadable C ABI-compatible library generation from *tgLang* programs. In that workflow, a user could prototype an algorithm in the browser or locally, then export a generated header and shared library (`.so`, `.dll`, or `.dylib`) for use from C and C-compatible host languages. This would make *tgLang* useful not only as a scripting and experimentation environment, but also as a route for embedding validated geometry-processing routines into existing engineering software.

Language and module system. The current module system is sufficient for the examples in this thesis, but larger user projects require stronger package structure, clearer dependency boundaries, versioning, and documentation generation. The `special` mechanism also opens a path toward richer user-defined domain types. A key research question is where the boundary should lie between concepts that belong in the language model itself and concepts that should remain in libraries. This boundary matters because adding too much to the language can make it harder to learn, while adding too little leaves users dependent on ad hoc conventions.

Interactive and browser-based workflows. *tgLang Studio* already demonstrates that geometry programs can be written, executed, and visualized without installing a local toolchain. Future work should make this environment more useful for real research workflows: richer diagnostics, integrated profiling views, reproducible example bundles, downloadable native artifacts, and better support for comparing intermediate geometry states. The goal is not only convenience, but reproducibility: a reviewer, collaborator, or student should be able to run the same geometry workflow with minimal environment friction.

GPU and simulation-pipeline integration. Many geometry-processing and simulation-preprocessing tasks contain structured parallelism that could benefit from GPU execution. Future work should investigate GPU backends for array operations, grid operations, and selected point-cloud kernels, while preserving deterministic behavior where possible. In parallel, tighter integration with simulation pipelines is needed, including exporting solver-ready artifacts, recording preprocessing provenance, and validating generated geometry workflows against downstream solver behavior.

Overall, future development should keep the central design goal intact: *tgLang* should remain a focused language for executable geometry-processing workflows, not a general-purpose language with geometry libraries attached. The most important work is therefore not only adding features, but deciding which abstractions make geometry algorithms easier to write, test, share, and deploy without hiding the assumptions that matter for engineering use.

7

Conclusion

This thesis addressed practical challenges in geometry processing by developing multiple complementary workflows and a language framework for reproducible implementation. The central motivation was to reduce the gap between algorithmic formulation and robust deployment on imperfect real-world geometric data.

The first set of contributions focused on topology and representation. Chapter 2 introduced a curvature-aware, semi-heuristic strategy for detecting hole boundaries and related topological structures in discrete surfaces, including supporting operations for through-hole handling and negative-volume extraction. Chapter 3 introduced Series of Local Triangulations (SOLT) as an intermediate representation for point-cloud resampling, showing that local mesh structure can stabilize feature-preserving sampling while also motivating a preliminary application to CAD-free higher-order surface reconstruction.

The second set of contributions focused on application-oriented mesh generation and feature extraction. Chapter 4 presented a shrink-wrap workflow that combines distance fields and adaptive morphology on octree-based volumetric representations, enabling robust topology simplification and controlled genus reduction for complex engineering geometries. Chapter 5 presented a distance-field-based method for mean camber line extraction, demonstrating that the target geometric feature can be recovered without restrictive assumptions on uniform coordinate distribution or explicit upper-lower boundary separation.

The final contribution, in Chapter 6, was *tgLang*, a domain-specific programming language that unifies geometry-native abstractions with deterministic execution semantics and practical backend support. By combining concise expression with reproducible execution, *tgLang* provides a concrete path from prototype geometry algorithms to maintainable computational workflows.

Taken together, these contributions establish a coherent toolbox for robust geometry processing. The methods address different stages of the same practical pipeline:

repairing or interpreting imperfect geometry, constructing useful intermediate representations, generating simulation-oriented meshes, extracting application-specific features, and preserving the resulting workflows in an executable form. They show that reliable geometry-processing workflows emerge not only from strong individual algorithms, but also from the way those algorithms are represented, composed, and executed in practice.

From a broader perspective, the thesis argues that geometry processing should be treated as a full-stack problem. Robustness depends simultaneously on topology handling, geometric representation, numerically stable operators, and the quality of the computational interface through which algorithms are expressed. By addressing these layers together, the presented work supports both scientific reproducibility and practical deployment in engineering workflows.

The overall outcome is therefore twofold. First, the thesis contributes concrete methods for challenging geometry-processing tasks that arise in practice. Second, it contributes a programming model that helps preserve these methods as maintainable and extensible workflows over time. This combination is essential for translating geometry-processing research into durable computational tools.

7.1. Future Work

Several directions naturally follow from the work presented in this thesis. The individual chapters already discuss limitations and immediate extensions in their own contexts; this section expands on the main directions that connect those chapter-level observations across the thesis.

Topological hole detection and negative volume extraction. The most immediate extension is to reduce the reliance on hand-tuned geometric thresholds. The current method is intentionally simple and fast, but it remains semi-heuristic. Future work could combine the curvature and feature-angle filters with more global topological descriptors, such as graph-based or Reeb-based summaries, to better distinguish true through-holes from fillets, chamfers, and other sharp but non-topological features. A second direction is to make the negative-volume extraction workflow quantitatively verifiable by comparing extracted volumes against CAD-derived or benchmark fluid domains.

Series of Local Triangulations. Future work should focus on stronger reproducibility and validation. The SOLT representation is useful because it creates local surface structure without requiring a global mesh, but parameter selection, feature-distance construction, and density adaptation could be made more automatic. The preliminary extension toward CAD-free higher-order surface reconstruction should also be developed more fully: element-quality metrics, surface-error convergence,

and downstream solver behavior are necessary before the approach can be claimed as a complete high-order meshing workflow.

Shrink-wrap mesh generation. The main open challenge is application-specific accuracy. The method is robust for topology simplification and geometry cleanup, but different simulations tolerate different levels of geometric approximation. Future work should therefore connect shrink-wrap parameters directly to simulation error, for example through comparisons with reference aerodynamic or thermal benchmarks. This would make the method easier to use in engineering settings where robustness alone is insufficient and quantitative accuracy is required.

Mean camber line extraction. The distance-field formulation removes several restrictions of classical airfoil-specific constructions, but the thesis focuses on two-dimensional sections and representative decomposition examples. A natural future extension is a section-wise treatment of swept geometries: slicing a wing or blade at multiple stations, extracting mean camber lines on the sections, and reconstructing a swept surface from the resulting curves. This extension should be validated systematically by studying sensitivity to slice placement, surface noise, twist, thickness variation, and station-to-station correspondence. A second direction is to connect the extracted features directly to downstream inverse-design and mesh-blocking workflows, so that mean-line and decomposition information can be used as reusable geometric constraints rather than only as diagnostic output.

tgLang. The next step is to continue turning the language from a research prototype into a stable engineering tool. This includes expanding the standard library, strengthening native compilation and optimization, improving interoperability with existing mesh libraries, and validating complete end-to-end workflows that combine geometry repair, meshing, and simulation preprocessing. A related deployment direction is to let users prototype an algorithm in the browser and then obtain downloadable C ABI-compatible libraries from `tglang.org`, making the same workflow callable from C and C-compatible host languages. More broadly, the language raises a research question that extends beyond this thesis: which geometry-processing concepts should be part of the programming model itself, and which should remain library-level abstractions?

The long-term goal is an ecosystem in which robust algorithms, reusable implementations, and reproducible execution are developed together. Geometry processing for engineering applications will continue to involve imperfect data, application-specific tolerances, and complex workflows. Addressing these challenges requires not only better algorithms, but also better ways to express, compose, test, and deploy them.

References

[1] L. Zhao, Y. Hu, X. Yang, Z. Dou, and L. Kang, "Robust multi-task learning network for complex LiDAR point cloud data preprocessing," *Expert Systems with Applications*, vol. 237, p. 121552, 2024, doi: https://doi.org/10.1016/j.eswa.2023.121552.

[2] R. Hanocka, G. Metzer, R. Giryes, and D. Cohen-Or, "Point2Mesh: A Self-Prior for Deformable Meshes," *ACM Trans. Graph.*, vol. 39, no. 4, July 2020, doi: 10.1145/3386569.3392415.

[3] F. Danglade, "Estimation of CAD Model Simplification Impact on CFD Analysis using Machine Learning Techniques," in *CAD'15*, CAD Solutions LLC, June 2015. doi: 10.14733/cadconfp.2015.58-63.

[4] J. Huang, H. Su, and L. J. Guibas, "Robust Watertight Manifold Surface Generation Method for ShapeNet Models," *CoRR*, 2018, [Online]. Available: http://arxiv.org/abs/1802.01698

[5] H. Doraiswamy and V. Natarajan, "Efficient algorithms for computing Reeb graphs," *Computational Geometry*, vol. 42, no. 6, pp. 606–616, 2009, doi: https://doi.org/10.1016/j.comgeo.2008.12.003.

[6] A. Lozano-Durán and G. Borrell, "Algorithm 964: An Efficient Algorithm to Compute the Genus of Discrete Surfaces and Applications to Turbulent Flows," *ACM Trans. Math. Softw.*, vol. 42, no. 4, pp. 34:1–34:19, June 2016, doi: 10.1145/2845076.

[7] M. Smereka and I. Dulęba, "Circular Object Detection Using a Modified Hough Transform," *Int. J. Appl. Math. Comput. Sci.*, vol. 18, no. 1, pp. 85–91, Mar. 2008, doi: 10.2478/v10006-008-0008-9.

[8] Y. Wang, R. Liu, F. Li, S. Endo, T. Baba, and Y. Uehara, "An effective hole detection method for 3D models," in *2012 Proceedings of the 20th European Signal Processing Conference (EUSIPCO)*, Aug. 2012, pp. 1940–1944.

[9] L. Kobbelt, S. Bischoff, M. Botsch, and S. Steinberg, "OpenMesh: A Generic and Efficient Polygon Mesh Data Structure," 2002, [Online]. Available: https://pub.uni-bielefeld.de/publication/1961694

[10] The CGAL Project, *CGAL User and Reference Manual*, 4.14 ed. CGAL Editorial Board, 2019. [Online]. Available: https://doc.cgal.org/4.14/Manual/packages.html

[11] M. Meyer, M. Desbrun, P. Schröder, and A. H. Barr, "Discrete Differential-Geometry Operators for Triangulated 2-Manifolds," in *Visualization and Mathematics III*, H.-C. Hege and K. Polthier, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 35–57.

[12] J. Long, "A Hybrid Hole-filling Algorithm," Master's thesis, 2013.

[13] E. Pérez, S. Salamanca, P. Merchán, and A. Adán, "A Comparison of Hole-filling Methods in 3D," *Int. J. Appl. Math. Comput. Sci.*, vol. 26, no. 4, pp. 885–903, Dec. 2016, doi: 10.1515/amcs-2016-0063.

[14] B. Gärtner, "Fast and Robust Smallest Enclosing Balls," in *Proceedings of the 7th Annual European Symposium on Algorithms*, in ESA '99. London, UK, UK: Springer-Verlag, 1999, pp. 325–338. [Online]. Available: http://dl.acm.org/citation.cfm?id=647909.740295

[15] The CGAL Project, *CGAL User and Reference Manual*, 5.3 ed. CGAL Editorial Board, 2021. [Online]. Available: https://doc.cgal.org/5.3/Manual/packages.html

[16] T. Rios *et al.*, "Scalability of Learning Tasks on 3D CAE Models Using Point Cloud Autoencoders," 2019, pp. 1367–1374. doi: 10.1109/SSCI44817.2019.9002982.

[17] B. Fornberg and N. Flyer, *A Primer on Radial Basis Functions with Applications to the Geosciences*,. Philadelphia, PA: Society for Industrial, Applied Mathematics, 2015. doi: 10.1137/1.9781611974041.

[18] P. Suchde, T. Jacquemin, and O. Davydov, "Point Cloud Generation for Meshfree Methods: An Overview," *Archives of Computational Methods in Engineering*, vol. 30, no. 2, pp. 889–915, Mar. 2023, doi: 10.1007/s11831-022-09820-w.

[19] K.-A. Aliev, A. Sevastopolsky, M. Kolos, D. Ulyanov, and V. Lempitsky, "Neural Point-Based Graphics," 2020.

[20] G. Metzer, R. Hanocka, D. Zorin, R. Giryes, D. Panozzo, and D. Cohen-Or, "Orienting Point Clouds with Dipole Propagation," *ACM Trans. Graph.*, vol. 40, no. 4, July 2021, doi: 10.1145/3450626.3459835.

[21] Q. Deng, S. Zhang, and Z. Ding, "An Efficient Hypergraph Approach to Robust Point Cloud Resampling," *IEEE Transactions on Image Processing*, vol. 31, no. , pp. 1924–1937, 2022, doi: 10.1109/TIP.2022.3149225.

[22] S. Chen, D. Tian, C. Feng, A. Vetro, and J. Kovačević, "Fast Resampling of Three-Dimensional Point Clouds via Graphs," *IEEE Transactions on Signal Processing*, vol. 66, no. 3, pp. 666–681, 2018, doi: 10.1109/TSP.2017.2771730.

[23] C. Lv, W. Lin, and B. Zhao, "Intrinsic and Isotropic Resampling for 3D Point Clouds," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 3, pp. 3274–3291, 2023, doi: 10.1109/TPAMI.2022.3185644.

[24] Y. Xiao, T. Zhang, J. Cao, and Z. Chen, "Accelerated Lloyd's Method for Resampling 3D Point Clouds," *IEEE Transactions on Multimedia*, no. , pp. 1–14, 2024, doi: 10.1109/TMM.2024.3405664.

[25] X. Jiao *et al.*, "Weighted Poisson-disk Resampling on Large-Scale Point Clouds," *arXiv preprint arXiv:2412.09177*, 2025.

[26] B. Fei *et al.*, "Comprehensive Review of Deep Learning-Based 3D Point Cloud Completion Processing and Analysis," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 12, pp. 22862–22883, 2022, doi: 10.1109/TITS.2022.3195555.

[27] L. Zhao, Y. Hu, X. Yang, Z. Dou, and Q. Wu, "ICDDPM: Image-conditioned denoising diffusion probabilistic model for real-world complex point cloud single view reconstruction," *Expert Systems with Applications*, vol. 259, p. 125370, 2025, doi: https://doi.org/10.1016/j.eswa.2024.125370.

[28] C. Wu, J. Zheng, J. Pfrommer, and J. Beyerer, "Attention-Based Point Cloud Edge Sampling," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.

[29] H. Chen, B. Du, S. Luo, and W. Hu, "Deep Point Set Resampling via Gradient Fields," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, no. , p. 1, 2022, doi: 10.1109/TPAMI.2022.3175183.

[30] Y. Rong, H. Zhou, K. Xia, C. Mei, J. Wang, and T. Lu, "RepKPU: Point Cloud Upsampling with Kernel Point Representation and Deformation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 21050–21060.

[31] V. K. Suriyababu, C. Vuik, and M. Möller, "Towards a High Quality Shrink Wrap Mesh Generation Algorithm Using Mathematical Morphology," *Computer-Aided Design*, vol. 164, p. 103608, 2023, doi: https://doi.org/10.1016/j.cad.2023.103608.

[32] N. Feng and K. Crane, "A Heat Method for Generalized Signed Distance," *ACM Trans. Graph.*, vol. 43, no. 4, July 2024, doi: 10.1145/3658220.

[33] H. Su, Z. Duanmu, W. Liu, Q. Liu, and Z. Wang, "Perceptual quality assessment of 3D point clouds," in *2019 IEEE International Conference on Image Processing (ICIP)*, 2019, pp. 3182–3186.

[34] Q. Liu, H. Su, Z. Duanmu, W. Liu, and Z. Wang, "Perceptual Quality Assessment of Colored 3D Point Clouds," *IEEE Transactions on Visualization and Computer Graphics*, no. , p. 1, 2022, doi: 10.1109/TVCG.2022.3167151.

[35] Q. Zhou and A. Jacobson, "Thingi10K: A Dataset of 10,000 3D-Printing Models," *arXiv preprint arXiv:1605.04797*, 2016.

[36] E. Whalen, A. Beyene, and C. Mueller, "SimJEB: Simulated Jet Engine Bracket Dataset," *Computer Graphics Forum*, 2021, doi: 10.1111/cgf.14353.

[37] G. Turk, "Generating Random Points in Triangles," in *Graphics Gems*, USA: Academic Press Professional, Inc., 1990, pp. 24–28.

[38] X. Jiao and D. Wang, "Reconstructing high-order surfaces for meshing," *Engineering with Computers*, vol. 28, no. 4, pp. 361–373, Oct. 2012, doi: 10.1007/s00366-011-0244-8.

[39] J. Ims and Z. J. Wang, "Automated low-order to high-order mesh conversion," *Engineering with Computers*, vol. 35, no. 1, pp. 323–335, Jan. 2019, doi: 10.1007/s00366-018-0602-x.

[40] Z. Wang *et al.*, "High-order CFD methods: current status and perspective," *International Journal for Numerical Methods in Fluids*, vol. 72, no. 8, pp. 811–845, 2013, doi: https://doi.org/10.1002/fld.3767.

[41] Z. Jiang, Z. Zhang, Y. Hu, T. Schneider, D. Zorin, and D. Panozzo, "Bijective and coarse high-order tetrahedral meshes," *ACM Trans. Graph.*, vol. 40, no. 4, July 2021, doi: 10.1145/3450626.3459840.

[42] Z.-Q. Cheng, Y.-Z. Wang, B. Li, K. Xu, G. Dang, and S.-Y. Jin, "A survey of methods for moving least squares surfaces," in *Proceedings of the Fifth Eurographics / IEEE VGTC Conference on Point-Based Graphics*, in SPBG'08. Los Angeles, CA: Eurographics Association, 2008, pp. 9–23.

[43] H.-T. D. Liu, J. E. Zhang, M. Ben-Chen, and A. Jacobson, "Surface Multigrid via Intrinsic Prolongation," *ACM Trans. Graph.*, vol. 40, no. 4, 2021.

[44] M. Attene, M. Campen, and L. Kobbelt, "Polygon Mesh Repairing: An Application Perspective," *ACM Comput. Surv.*, vol. 45, no. 2, Mar. 2013, doi: 10.1145/2431211.2431214.

[45] J. Esteve, P. Brunet, and A. Vinacau, "Approximation of a Variable Density Cloud of Points by Shrinking a Discrete Membrane," *Computer Graphics Forum*, vol. 24, no. 4, pp. 791–807, 2005.

[46] F. Nooruddin and G. Turk, "Simplification and repair of polygonal models using volumetric techniques," *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 2, pp. 191–205, 2003, doi: 10.1109/TVCG.2003.1196006.

[47] Y. K. Lee, C. K. Lim, H. Ghazialam, H. Vardhan, and E. Eklund, "Surface Mesh Generation for Dirty Geometries by Shrink Wrapping using Cartesian Grid Approach," in *Proceedings of the 15th International Meshing Roundtable*, P. P. Pébay, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 393–410.

[48] P. Alliez, D. Cohen-Steiner, M. Hemmer, C. Portaneri, and M. Rouxel-Labbé, "3D Alpha Wrapping," *CGAL User and Reference Manual*. CGAL Editorial Board, 2022.

[49] S. Vijai Kumar and C. Vuik, "A Simple and Fast Hole Detection Algorithm for Triangulated Surfaces," *Journal of Computing and Information Science in Engineering*, vol. 21, no. 4, 2021, doi: 10.1115/1.4049030.

[50] L. Najman and H. Talbot, "Introduction to Mathematical Morphology," in *Mathematical Morphology*, John Wiley & Sons, Ltd, 2013, ch. 1, pp. 1–33. doi: https://doi.org/10.1002/9781118600788.ch1.

[51] D. Jeulin, "Analysis and Modeling of 3D Microstructures," in *Mathematical Morphology*, John Wiley & Sons, Ltd, 2013, ch. 19, pp. 421–444. doi: https://doi.org/10.1002/9781118600788.ch19.

[52] Z. Chen, D. Panozzo, and J. Dumas, "Half-Space Power Diagrams and Discrete Surface Offsets," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 10, pp. 2970–2981, 2020, doi: 10.1109/TVCG.2019.2945961.

[53] G. Barill, N. Dickson, R. Schmidt, D. I. Levin, and A. Jacobson, "Fast Winding Numbers for Soups and Clouds," *ACM Transactions on Graphics*, 2018.

[54] J. L. Blanco and P. K. Rai, "nanoflann: a C++ header-only fork of FLANN, a library for Nearest Neighbor (NN) with KD-trees." 2014.

[55] D. Boltcheva and B. Lévy, "Surface reconstruction by computing restricted Voronoi cells in parallel," *Computer-Aided Design*, vol. 90, pp. 123–134, 2017, doi: https://doi.org/10.1016/j.cad.2017.05.011.

[56] J. Zhang, B. Deng, Y. Hong, Y. Peng, W. Qin, and L. Liu, "Static/Dynamic Filtering for Mesh Geometry," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 4, pp. 1774–1787, Apr. 2019, doi: 10.1109/TVCG.2018.2816926.

[57] T. O. Foundation, "OpenFOAM v8 User Guide." [Online]. Available: https://cfd.direct/openfoam/user-guide

[58] I. Abbott and A. von Doenhoff, *Theory of Wing Sections: Including a Summary of Airfoil Data*. in Dover Books on Aeronautical Engineering. Dover Publications, 2012. [Online]. Available: https://books.google.com/books?id=lWe8AQAAQBAJ

[59] J. Barlow, W. Rae, and A. Pope, *Low-Speed Wind Tunnel Testing*. in Aerospace engineering, mechanical engineering. Wiley, 1999. [Online]. Available: https://books.google.com/books?id=LBMoAQAAMAAJ

[60] J. ANDERSON and J. Anderson, *Fundamentals of Aerodynamics*. in McGraw-Hill series in aeronautical and aerospace engineering. McGraw-Hill, 2010. [Online]. Available: https://books.google.com/books?id=7B4GQ9GljPMC

[61] *Aircraft Performance & Design*. McGraw-Hill Education (India) Pvt Limited, 2010. [Online]. Available: https://books.google.com/books?id=Ck9l1DGj5-4C

[62] H. A. L. Q.-d. A. Z. L.-p. A. Y. Z.-p. Li Wen-long AND Xie, "Section Curve Reconstruction and Mean-Camber Curve Extraction of a Point-Sampled Blade Surface," *PLOS ONE*, vol. 9, no. 12, pp. 1–30, 2015, doi: 10.1371/journal.pone.0115471.

[63] R. Carmichael, "Algorithm for calculating coordinates of cambered NACA airfoils at specified chord locations," in *1st AIAA, Aircraft, Technology Integration, and Operations Forum*, 2001. doi: 10.2514/6.2001-5235.

[64] V. Sekar, M. Zhang, C. Shu, and B. C. Khoo, "Inverse Design of Airfoil Using a Deep Convolutional Neural Network," *AIAA Journal*, vol. 57, no. 3, pp. 993–1003, 2019, doi: 10.2514/1.J057894.

[65] Y. Zhang, W. J. Sung, and D. N. Mavris, "Application of Convolutional Neural Network to Predict Airfoil Lift Coefficient," in *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, 2018. doi: 10.2514/6.2018-1903.

[66] E. Yilmaz and B. German, "A Convolutional Neural Network Approach to Training Predictors for Airfoil Performance," in *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, 2017, pp. 232–2323. doi: 10.2514/6.2017-3660.

[67] K. Crane, C. Weischedel, and M. Wardetzky, "The Heat Method for Distance Computation," *Commun. ACM*, vol. 60, no. 11, pp. 90–99, Oct. 2017, doi: 10.1145/3131280.

[68] J. A. Sethian, "A fast marching level set method for monotonically advancing fronts," *Proceedings of the National Academy of Sciences*, vol. 93, no. 4, pp. 1591–1595, 1996, doi: 10.1073/pnas.93.4.1591.

[69] H. Xia and P. G. Tucker, "Fast equal and biased distance fields for medial axis transform with meshing in mind," *Applied Mathematical Modelling*, vol. 35, no. 12, pp. 5804–5819, 2011, doi: https://doi.org/10.1016/j.apm.2011.05.001.

[70] R. Kimmel and G. Sapiro, "Shortening three-dimensional curves via two-dimensional flows," *Computers & Mathematics with Applications*, vol. 29, no. 3, pp. 49–62, 1995, doi: https://doi.org/10.1016/0898-1221(94)00228-D.

[71] H. DU, T. YOO, and H. QIN, "PDE-Based Medial Axis Extraction and Shape Manipulation of Arbitrary Meshes," *Journal of Systems Science and Complexity*, vol. 21, no. 4, p. 609, Nov. 2008, doi: 10.1007/s11424-008-9138-2.

[72] V. K. Suriyababu, "tgLang Studio: A web-based integrated development environment for geometry processing." [Online]. Available: https://tglang.org/

[73] Airfoil Tools, "Airfoil Tools." 2020.

[74] X. Song and others, "Questionnaire-Based Discussion of Finite Element Multi-Physics Simulation Software in Power Electronics," *IEEE Transactions on Power Electronics*, vol. 33, pp. 10226–10236, 2018.

[75] OpenVolumeMesh Developers, "OpenVolumeMesh." 2025.

[76] N. Sharp, K. Crane, and others, "GeometryCentral: A modern C++ library of data structures and algorithms for geometry processing," 2019.

[77] A. Jacobson, D. Panozzo, and others, "libigl: A simple C++ geometry processing library." 2018.

[78] PyMesh Developers, "PyMesh: Geometry Processing Library." 2025.

[79] Q.-Y. Zhou, J. Park, and V. Koltun, "Open3D: A Modern Library for 3D Data Processing," *arXiv:1801.09847*, 2018.

[80] C. Geuzaine and J.-F. Remacle, "Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities," *International Journal for Numerical Methods in Engineering*, vol. 79, no. 11, pp. 1309–1331, 2009, doi: 10.1002/nme.2579.

[81] Z. DeVito *et al.*, "Liszt: A domain-specific language for building portable mesh-based PDE solvers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, New York, NY, USA: ACM, 2011, pp. 9:1–9:12. doi: 10.1145/2063384.2063396.

[82] C. Yu, Y. Xu, Y. Kuang, Y. Hu, and T. Liu, "MeshTaichi: A Compiler for Efficient Mesh-Based Operations," *ACM Trans. Graph.*, vol. 41, no. 6, Nov. 2022, doi: 10.1145/3550454.3555430.

[83] Y. Li, S. Kamil, K. Crane, A. Jacobson, and Y. I. Gingold, "I❤️Mesh: A DSL for Mesh Processing," *ACM Transactions on Graphics*, vol. 43, no. 5, pp. 154:1–154:17, July 2024, doi: 10.1145/3662181.

[84] T. Parr, S. Harwell, and K. Fisher, "Adaptive LL(*) parsing: the power of dynamic analysis," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, pp. 579–598. doi: 10.1145/2660193.2660202.

[85] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013. [Online]. Available: https://books.google.at/books?id=gA9QDwAAQBAJ

Appendix

A

Hole Detection

This appendix contains the complete feature-edge hole detection workflow discussed in Chapter 2. The code is written in *tgLang*.

```
import "utils/os" as os;
import "math" as math;
import "geometry/curve" as curve;
import "data/array" as arrays;
import "geometry/surface_mesh" as smesh;

func defaultMeshPath() -> string {
    string dir = os::scriptDir();
    if (dir.length() == 0) { dir = os::cwd(); }
    return os::join(dir, "inputs/j_with_holes.stl");
}

// Select edges by feature angle + low gaussian curvature, then extract closed loops.
special holeDetector {
    surfaceMesh mesh;
    double featureMinDeg;
    double featureMaxDeg;
    bool verbose;
    array K;
    array A;
    array VN;
    double loopNormalDotMax;

    func init(string meshPath, bool verbose) {
        self.featureMinDeg = 250.0;
        self.featureMaxDeg = 290.0;
        self.verbose = verbose;
        surfaceMesh m = smesh::load(meshPath);
        if (self.verbose) {
            print("mesh:", m.toString(), " boundaryEdges:", m.boundaryEdgeCount());
        }
```

```
        self.mesh = m;
        self.K = self.mesh.gaussianCurvature();
        self.A = self.mesh.vertexAreas();
        self.VN = self.mesh.vertexNormals();
        self.loopNormalDotMax = 0.65;
    }

    func collectFeatureEdges() -> array {
        int E = self.mesh.edgeCount();
        array edges = self.mesh.edgeVerticesArray();
        for e from 0 through E - 1 {
            double dih = self.mesh.edgeDihedralDeg(e);
            double anhedral = 360.0 - dih;
            if (anhedral < self.featureMinDeg || anhedral > self.featureMaxDeg) {
                edges[e, 0] = -1;
                edges[e, 1] = -1;
            }
        }
        if (self.verbose) {
            int count = 0;
            for e from 0 through E - 1 {
                if (edges[e, 0] >= 0) { count = count + 1; }
            }
            print("feature edges:", count);
        }
        return edges;
    }

    func filterByGaussian(array edges) -> array {
        int E = self.mesh.edgeCount();
        array K = self.K;
        array A = self.A;
        double dMin = 1e30;
        double dMax = -1e30;
        int negCount = 0;
        if (self.verbose) {
            print("gaussian filter: candidates:", E);
        }
        for i from 0 through E - 1 {
            int v0 = edges[i, 0];
            int v1 = edges[i, 1];
            if (v0 < 0 || v1 < 0) { continue; }
            double d0 = K[v0] * A[v0];
            double d1 = K[v1] * A[v1];
            double edgeVal = (d0 + d1) * 0.5;
            if (edgeVal < dMin) { dMin = edgeVal; }
            if (edgeVal > dMax) { dMax = edgeVal; }
            if (edgeVal < 0.0) { negCount = negCount + 1; }
            if (edgeVal >= 0.0) {
                edges[i, 0] = -1;
                edges[i, 1] = -1;
            }
        }
        if (self.verbose) {
            print("gaussian edgeVal range:", dMin, dMax, "neg:", negCount);
            print("after gaussian filter:", negCount);
        }
        return edges;
    }

    func filterValence2(array edges) -> array {
        int V = self.mesh.vertexCount();
        int E = self.mesh.edgeCount();
        array deg = arrays::zeros("int", [V]);
        for i from 0 through E - 1 {
            int v0 = edges[i, 0];
            int v1 = edges[i, 1];
```

```
            if (v0 < 0 || v1 < 0) { continue; }
            deg[v0] = deg[v0] + 1;
            deg[v1] = deg[v1] + 1;
        }
        for i from 0 through E - 1 {
            int v0 = edges[i, 0];
            int v1 = edges[i, 1];
            if (v0 < 0 || v1 < 0) { continue; }
            if (deg[v0] != 2 || deg[v1] != 2) {
                edges[i, 0] = -1;
                edges[i, 1] = -1;
            }
        }
        return edges;
    }

    func buildLoopsFromEdges(array edges) -> collection {
        collection loops;
        int V = self.mesh.vertexCount();
        array neigh0 = arrays::zeros("int", [V]);
        array neigh1 = arrays::zeros("int", [V]);
        array deg = arrays::zeros("int", [V]);
        for v from 0 through V - 1 {
            neigh0[v] = -1;
            neigh1[v] = -1;
        }
        int E = self.mesh.edgeCount();
        for i from 0 through E - 1 {
            int v0 = edges[i, 0];
            int v1 = edges[i, 1];
            if (v0 < 0 || v1 < 0) { continue; }
            if (neigh0[v0] == -1) { neigh0[v0] = v1; }
            else { neigh1[v0] = v1; }
            if (neigh0[v1] == -1) { neigh0[v1] = v0; }
            else { neigh1[v1] = v0; }
            deg[v0] = deg[v0] + 1;
            deg[v1] = deg[v1] + 1;
        }
        array visited = arrays::zeros("int", [V]);
        array VN = self.VN;
        for v from 0 through V - 1 {
            if (deg[v] != 2 || visited[v] != 0) { continue; }
            collection loop;
            int start = v;
            int prev = -1;
            int cur = v;
            while true {
                loop.append(cur);
                visited[cur] = 1;
                int next = neigh0[cur];
                if (next == prev) { next = neigh1[cur]; }
                if (next == -1) { break; }
                prev = cur;
                cur = next;
                if (cur == start) { break; }
            }
            if (cur == start && loop.size() > 2) {
                int n = loop.size();
                array pos = arrays::zeros("double", [n, 3]);
                array ed = arrays::zeros("int", [n, 2]);
                array normals = arrays::zeros("double", [n, 3]);
                for j from 0 through n - 1 {
                    int vid = loop[j];
                    array p = self.mesh.getPosition(vid);
                    pos[j, 0] = p[0];
                    pos[j, 1] = p[1];
                    pos[j, 2] = p[2];
```

```
                    normals[j, 0] = VN[vid, 0];
                    normals[j, 1] = VN[vid, 1];
                    normals[j, 2] = VN[vid, 2];
                    ed[j, 0] = j;
                    ed[j, 1] = (j + 1) % n;
                }
                curveNetwork cn;
                cn.buildFromArrays(pos, ed);
                cn.setVertexProperty("vertexNormals", normals);
                loops.append(cn);
            }
        }
        return loops;
    }

    func loopNormalScore(curveNetwork cn) -> double {
        int n = cn.nodeCount();
        if (n < 3) { return 1.0; }
        array pos = cn.positionsArray();
        array normals = cn.getVertexProperty("vertexNormals");
        array c = arrays::zeros("double", [3]);
        array p = arrays::zeros("double", [3]);
        for i from 0 through n - 1 {
            p[0] = pos[i, 0];
            p[1] = pos[i, 1];
            p[2] = pos[i, 2];
            c.axpyInPlace(1.0, p);
        }
        c.scaleInPlace(1.0 / (n * 1.0));

        array nrm = arrays::zeros("double", [3]);
        array tmp = arrays::zeros("double", [3]);
        array a = arrays::zeros("double", [3]);
        array b = arrays::zeros("double", [3]);
        for i from 0 through n - 1 {
            a[0] = pos[i, 0];
            a[1] = pos[i, 1];
            a[2] = pos[i, 2];
            b[0] = pos[(i + 1) % n, 0];
            b[1] = pos[(i + 1) % n, 1];
            b[2] = pos[(i + 1) % n, 2];
            a.subInPlace(c);
            b.subInPlace(c);
            math::crossInPlace(tmp, a, b);
            nrm.axpyInPlace(1.0, tmp);
        }
        double nlen = nrm.norm();
        if (nlen == 0.0) { return 1.0; }
        nrm.scaleInPlace(1.0 / nlen);

        double acc = 0.0;
        for i from 0 through n - 1 {
            double vx = normals[i, 0];
            double vy = normals[i, 1];
            double vz = normals[i, 2];
            double dot = nrm[0] * vx + nrm[1] * vy + nrm[2] * vz;
            acc = acc + math::abs(dot);
        }
        return acc / (n * 1.0);
    }

    func detect() -> collection {
        array edges = self.collectFeatureEdges();
        edges = self.filterByGaussian(edges);
        edges = self.filterValence2(edges);
        collection loops = self.buildLoopsFromEdges(edges);
        if (loops.size() == 0) { return empty; }
```

```
        collection good;
        for i from 0 through loops.size() - 1 {
            curveNetwork cn = loops[i];
            double score = self.loopNormalScore(cn);
            if (self.verbose) {
                print("loop", i, "normal score:", score, "verts:", cn.nodeCount());
            }
            if (score <= self.loopNormalDotMax) {
                good.append(cn);
            }
        }
        return good;
    }
}

func main(collection args) -> int {
    string path = defaultMeshPath();
    if (args.size() > 1) {
        path = args[1];
    }

    holeDetector hd = holeDetector(path, false);
    collection loops = hd.detect();
    print("loops:", loops.size());
    for (li, c) : enumerate(loops) {
        curveNetwork cn = c;
        if (cn.edgeCount() <= 3) { continue; }
        print("loop", li, "edges:", cn.edgeCount());
    }

    plot p("Hole Detection (Feature Edge Loops)");
    p.addMesh("mesh", hd.mesh);
    for (i, c) : enumerate(loops) {
        curveNetwork cn = c;
        if (cn.edgeCount() <= 3) { continue; }
        string label = "loop_" + i.toString();
        p.addCurveNetwork(label, cn);
    }
    p.save("holeDetection.tgvis");
    return 0;
}
```

Appendix

B

tgLang Complete Grammar

This appendix contains the complete ANTLR4 grammar specification for tgLang.

```
grammar tensorGrad;

// Entry point
program
    : (importStatement
    | moduleDeclaration
    | extendModuleDeclaration
    | functionDeclaration
    | specialDeclaration
    | extendSpecialDeclaration
    | testStatement
    | statement)+ EOF
    ;


// Statements (no functionDeclaration here)
statement       : varDeclaration
                | assignment
                | printStatement
                | exprStatement
                | returnStatement
                | ifStatement
                | forStatement
                | parForStatement
                | whileStatement
                | throwStatement
                | tryCatchStatement
                | breakStatement
                ;

// Imports
importStatement
    : 'import' ID ('as' ID)? ';' # ModuleImport
```

```
    | 'import' STRING ('as' ID)? ';' # FileImport
    ;

// --------------------
// Special types (records/classes) grammar extensions
// --------------------
specialDeclaration
    : 'special' qualifiedTypeName '{' specialBodyItem* '}'
    ;

extendSpecialDeclaration
    : 'extend' 'special' qualifiedTypeName '{' specialMethodDecl+ '}'
    ;

qualifiedTypeName
    : (ID '::')* ID
    ;

specialBodyItem
    : specialFieldDecl
    | specialInitDecl
    | specialMethodDecl
    ;

specialFieldDecl
    : type ID ('=' expr)? ';'
    ;

// init cannot contain 'return' statements
specialInitDecl
    : 'func' 'init' '(' parameterList? ')' '{' initStatement* '}'
    ;

// Statements allowed inside init (same as 'statement' but without returnStatement)
initStatement
    : varDeclaration
    | assignment
    | printStatement
    | exprStatement
    | ifStatement
    | forStatement
    | whileStatement
    | throwStatement
    | tryCatchStatement
    | breakStatement
    ;

specialMethodDecl
    : 'func' ID '(' parameterList? ')' ('->' type)? '{' statement* '}'
    ;



// Modules
moduleDeclaration
    : 'module' (ID | STRING) '{' (functionDeclaration | varDeclaration |
    specialDeclaration)* '}'
    ;

// Extend module
extendModuleDeclaration
    : 'extendModule' (ID | STRING) '{' (functionDeclaration | varDeclaration |
    specialDeclaration)* '}'
    ;

// Function Declaration (restricted to top level)
functionDeclaration
    : 'func' modifier* ID '(' parameterList? ')' ('->' type)? '{' statement* '}'
```

```
    ;

// Modifiers
modifier
    : 'cached'
    ; // Add more as needed

// Parameters
parameterList   : type ID (',' type ID)*;

// If Statement
ifStatement     : 'if' expr '{' statement* '}' (elifStatement)* (elseStatement)?;
elifStatement   : 'elif' expr '{' statement* '}';
elseStatement   : 'else' '{' statement* '}';

// For Loop (disambiguated ordering)
// Order alts by earliest distinguishing token to avoid conflicts.
forStatement
    // Most-specific forms first (keywords immediately after loop var or paren tuple)
    : 'for' ID 'from' expr 'through' expr ('by' expr)? ('when' expr)? '{' statement* '}'
    # ForRangeInclusive
    | 'for' ID 'from' expr 'to'      expr ('by' expr)? ('when' expr)? '{' statement* '}'
    # ForRange
    | 'for' '(' idList ')' ':' 'zip' '(' exprList ')' ('when' expr)? '{' statement* '}'
    # ForZip
    | 'for' '(' ID ',' ID ')' ':' 'enumerate' '(' expr ')' ('when' expr)? '{' statement*
    '}'         # ForEnumerate
    | 'for' '(' idList ')' ':' 'grid' '(' exprList ')' ('when' expr)? '{' statement* '}'
    # ForGrid
    // Fallback iterable form last to avoid stealing other alts
    | 'for' ID ':' expr                          ('when' expr)? '{' statement* '}'
    # ForIter
    ;

// Parallel For (range-only for phase 1)
parForStatement
    : 'parfor' ID 'from' expr 'through' expr ('by' expr)? ('when' expr)? '{' statement* '}'
    # ParForRangeInclusive
    | 'parfor' ID 'from' expr 'to'      expr ('by' expr)? ('when' expr)? '{' statement* '}'
    # ParForRange
    | 'parfor' '(' idList ')' ':' 'zip' '(' exprList ')' ('when' expr)? '{' statement* '}'
    # ParForZip
    | 'parfor' '(' ID ',' ID ')' ':' 'enumerate' '(' expr ')' ('when' expr)? '{' statement*
    '}'     # ParForEnumerate
    | 'parfor' ID ':' expr                        ('when' expr)? '{' statement* '}'
    # ParForIter
    | 'parfor' '(' idList ')' ':' 'grid' '(' exprList ')' ('when' expr)? '{' statement* '}'
    # ParForGrid
    ;

idList      : ID (',' ID)* ;

// While Loop
whileStatement
    : 'while' expr '{' statement* '}'
    ;

// Throw / Try-Catch (named error categories)
throwStatement
    : 'throw' ID '(' expr? ')' ';'
    ;

tryCatchStatement
    : 'try' '{' statement* '}' 'catch' (ID)? ('as' ID)? '{' statement* '}'
    ;

// Zig-style test blocks (top-level only, run only in test mode)
testStatement
    : 'test' STRING '{' statement* '}'
```

```
    ;

// Iterable
iterable
    : ID // Variable referring to intVector or doubleVector
    | expr
    ;

// Expressions
exprStatement   : expr ';';
varDeclaration
    : type ID (('=' expr) | ('(' exprList? ')'))? ';'
    ;
assignment
    : ID '=' expr ';'                   # SimpleAssignment
    | ID '[' exprList ']' '=' expr ';'              # IndexedAssignment
    | expr '.' ID '=' expr ';'          # FieldAssignment
    ;

// Types
type            : 'int'
                | 'none'
                | 'double'
                | 'bool'
                | 'string'
                | arrayType
                | 'plot'
                | 'collection'
                | 'table'
                | 'ifile'
                | 'ofile'
                | 'error'
                | 'lineGrid'
                | 'squareGrid'
                | 'cubeGrid'
                | fdmGridType
                | 'surfaceMesh'
                | 'volumeMesh'
                | 'pointCloud'
                | 'curveNetwork'
                | qualifiedTypeName
                ;

// Array type with optional dtype and dimension/shape and optional filler, e.g.:
//   array                          // dynamic until constructed
//   array<double,2>                // 2D array (rank by INT)
//   array<int,DIM>                 // 2D array (rank by identifier)
//   array<double,{2,2}>            // 2x2 shape explicitly
//   array<int,{64,64,64}, "fill"] // 64x64x64 with optional filler identifier as string
arrayType       : 'array' ('<' primitiveDType (',' arrayDimOrShape (',' STRING)? )? '>')?
                ;

// Finite-difference grid with dimension parameter, e.g. fdmGrid<2>
fdmGridType     : 'fdmGrid' '<' INT '>'
                ;

// Dimension or explicit shape for array type parameter
arrayDimOrShape : INT                   # ArrayRankInt
                | ID                    # ArrayRankId
                | '{' dimList '}'     # ArrayShape
                ;

dimList         : dim (',' dim)*;
dim             : INT | ID;

primitiveDType  : 'double' | 'int' | 'bool'
                ;
```

```
224
225 expr
226         : expr '.' ID '(' exprList? ')'          # MethodCall
227         | expr '.' ID                            # FieldAccess
228         | '-' expr                              # UnaryMinus
229         | '!' expr                              # UnaryNot
230         | expr '**' expr                        # PowerOp
231         | expr op=('*' | '/' | '%') expr        # MultiplicativeOp
232         | expr op=('+' | '-') expr              # AdditiveOp
233         | expr op=('==' | '!=' | '<' | '<=' | '>' | '>=') expr # ComparisonOp
234         | expr op=('&&' | '||') expr            # LogicalOp
235         | '(' expr ')'                          # Parenthesized
236         | ID '(' exprList? ')'                  # FunctionCall
237         | ID '::' ID '(' exprList? ')'          # ModuleFunctionCall
238         | ID '::' ID                            # ModuleVariableAccess
239         | ID '[' exprList? ']'                  # IndexAccess
240         | vectorExpr 'with' shapeExpr             # ShapeAnnotatedLiteral
241         | vectorExpr                            # VectorLiteral
242         | matrixExpr                            # MatrixLiteral
243         | tableExpr                             # TableLiteral
244         | BOOL                                  # BoolLiteral
245         | NONE                                  # NoneLiteral
246         | STRING                                # StringLiteral
247         | INT                                   # IntLiteral
248         | DOUBLE                                # DoubleLiteral
249         | ID                                    # Variable
250         ;
251
252 exprList        : expr (',' expr)*;
253
254 // Unified Compound Literals
255 vectorExpr      : '[' expr (',' expr)* ']';
256
257 matrixExpr
258     : '[' exprList ( '@' exprList )* ']'
259     ;
260
261 tableExpr
262     : '{' tableEntry (',' tableEntry)* '}'
263     ;
264
265 tableEntry
266     : expr ':' expr
267     ;
268
269 shapeExpr
270     : '(' INT '|' INT ')'                     # MatrixShape
271     | '(' INT '|' INT '|' INT ')'             # TensorShape
272     ;
273
274 // Print and return statements
275 printStatement  : 'print' '(' expr (',' expr)* ')' ';';
276 returnStatement : 'return' expr? ';';
277 breakStatement  : 'break' ';';
278
279 // Literals and identifiers
280 NONE            : 'none';
281 BOOL            : 'true' | 'false';
282 STRING          : '"' .*? '"';
283 INT             : [0-9]+;
284 DOUBLE
285     : [0-9]+ ('.' [0-9]*)? ([eE] [+\-]? [0-9]+)?
286     | '.' [0-9]+ ([eE] [+\-]? [0-9]+)?
287     ;
288 ID              : [a-zA-Z_] [a-zA-Z0-9_]*;
289
290 // Skip whitespace and comments
```

```
WS                : [ \t\r\n]+ -> skip;
COMMENT           : '//' ~[\r\n]* -> channel(HIDDEN);
MULTILINE_COMMENT : '/*' .*? '*/' -> channel(HIDDEN);
```

Epilogue

This part-time PhD was carried out alongside the demands of a full-time industrial career. It began not from an abstract academic question, but from the daily experience of working with messy, imperfect, and often frustrating industrial geometries. Many of the problems addressed in this dissertation came from real bottlenecks I encountered in practice. They required solutions that were not only mathematically meaningful, but also robust enough to survive contact with engineering workflows.

At its heart, this thesis is about finding structure in unstructured data. Whether the task is identifying topological loops around holes, building a coherent mesh from a disconnected cloud of points, generating a shrink-wrapped simulation domain, or extracting a mean camber line from an airfoil, the underlying challenge is similar: make local, reliable decisions that together produce a useful global result. Over time, the idea of a series of local approximations became more than a technical device. It became a practical way for me to reason about large geometry-processing problems.

This work is not an endpoint. The algorithms for hole detection, resampling, shrink wrapping, and feature extraction are useful building blocks, but the final chapter on *tgLang* points to a broader problem. In application-oriented geometry processing, the difficulty is often not only the absence of algorithms. It is also the difficulty of composing, testing, sharing, and deploying them as reliable workflows. This realization has shifted my focus toward the structure of computation itself: toward better abstractions, languages, and compilers for geometric computation.

The path from a small town in South India to a PhD in Europe has been long and unlikely. It has involved work, uncertainty, restarts, and persistence. In some sense, it has also been a process of finding structure: in data, in work, and in life. This dissertation marks the completion of one stage of that process. The work of building, refining, and creating continues.

Acknowledgements

A part-time PhD, pursued alongside the daily demands of an engineering career, is a long and uneven undertaking. This dissertation would not have been possible without the guidance, flexibility, and support of many people and institutions.

First and foremost, I am deeply grateful to my promotoren, Prof. dr. ir. C. Vuik (Kees) and Dr. rer. nat. M. Möller (Matthias). Their flexibility was essential for making this part-time PhD possible. They gave me the freedom to explore my own ideas, while also helping me keep the work focused and scientifically grounded. Their guidance shaped not only this thesis, but also the way I think about research, implementation, and engineering problems.

I thank Engineering Software Steyr (ESS), my employer during much of this PhD, for supporting my part-time doctoral work during the first three years, from 2019 to 2022. The practical problems I encountered in the industrial environment at ESS strongly influenced the direction of this dissertation and kept the research connected to real engineering workflows. I also gratefully acknowledge the financial support from the Austrian Research Promotion Agency (FFG) during this period (2019 to 2022).

On a personal note, I thank my wife for the years we shared during this long period of my life. A PhD does not happen in isolation, and the personal circumstances around it inevitably become part of the journey.

I thank my parents and my family in India. Their love, sacrifices, and belief in education gave me the foundation from which everything else became possible. Their encouragement has stayed with me throughout this journey.

Finally, I thank my friends in Europe and back home for conversations, encouragement, and much-needed distractions along the way. I also acknowledge the discipline and persistence it took to bring this long journey to completion.

Curriculum Vitæ

Vijai Kumar SURIYABABU

20-12-1991 Born in Srivilliputtur, India.

Education

2009 to 2013 B.E. in Aerospace Engineering
Noorul Islam Center for Higher Education, India

2014 to 2016 M.S. (by Research) in Aerospace Engineering
MIT Campus, Anna University, India

2019 to 2026 Ph.D. in Applied Mathematics
Delft University of Technology, The Netherlands

Thesis: Robust and Automated Geometry Processing Workflows for Engineering Applications
Promotors: Prof. dr. ir. C. Vuik, Dr. rer. nat. M. Möller

Professional Experience

2018 to 2026 Software Developer (Geometry Processing & Mesh Generation)
(April 2018 to July 2026)
Engineering Software Steyr GmbH, Austria

2016 to 2018 Junior Software Development Engineer
(July 2016 to February 2018)
Program Development Company (GridPro), India

2016 Developer Intern
(March 2016 to June 2016)
Program Development Company (GridPro), India

Awards

2013 University Gold Medal
Noorul Islam University, India

2019 Industrial Research Grant (100k EUR), FFG
PhD funding for first three years (2019 to 2022)

List of Publications

[5.] **V.K. Suriyababu**, C. Vuik, and M. Möller. "Resampling Point Clouds Using Series of Local Triangulations." *Journal of Imaging*, 11(2):49, 2025.
https://doi.org/10.3390/jimaging11020049

[4.] **V.K. Suriyababu**, C. Vuik, and M. Möller. "Towards a High Quality Shrink Wrap Mesh Generation Algorithm Using Mathematical Morphology." *Computer-Aided Design*, 164:103608, 2023.
https://doi.org/10.1016/j.cad.2023.103608

[3.] **V.K. Suriyababu**, C. Vuik, and M. Möller. "Shrink Wrap Mesh Generation Using Morphological Operators with Selected Applications." *Zenodo*, 2022.
https://doi.org/10.5281/zenodo.6562410

[2.] **S. Vijai Kumar** and C. Vuik. "A Simple and Fast Hole Detection Algorithm for Triangulated Surfaces." *Journal of Computing and Information Science in Engineering*, 21(4):044502, 2021.
https://doi.org/10.1115/1.4049030

[1.] **S. Vijai Kumar** and C. Vuik. "Distance Field Driven Mean Camber Line Extraction Algorithm." *Reports of the Delft Institute of Applied Mathematics*, Delft University of Technology, 2020.
https://research.tudelft.nl/en/publications/distance-field-driven-mean-camber-line-extraction-algorithm

Conference Presentations

[4.] **Vijai Kumar**, Cornelis Vuik, and Matthias Möller. "Simplified Dense and Coarse Higher-Order Mesh Generation Using Moving Least Squares." *SIAM International Meshing Roundtable 2025 (Texas)* – Accepted.
https://internationalmeshingroundtable.com/assets/research-notes/imr33/2002-compressed.pdf

[3.] **Vijai Kumar**, Cornelis Vuik, and Matthias Möller. "Region Selective Subdivision of Intrinsic Meshes." *SIAM International Meshing Roundtable 2023 (Amsterdam).*
https://internationalmeshingroundtable.com/assets/research-notes/imr31/2003-comp.pdf

[2.] **Vijai Kumar**, Cornelis Vuik, and Matthias Möller. "Shrink Wrap Mesh Generation Using Morphological Operators with Selected Applications." *SIAM International Meshing Roundtable 2022 (Online).*
https://doi.org/10.5281/zenodo.6562409

[1.] **Vijai Kumar** and Cornelis Vuik. "A Robust Particle Generation Algorithm for Particle Methods." *SIAM CSE 2021.*
https://meetings.siam.org/sess/dsp_talk.cfm?p=110056

Propositions

accompanying the dissertation

Robust and Automated Geometry Processing Workflows for Engineering Applications

by

Vijai Kumar SURIYABABU

The following propositions pertain to this dissertation

1. In CAD-oriented surface meshes, through-hole boundaries can be detected using local geometric criteria based on feature angle and discrete Gaussian curvature, without requiring global topological graph construction. (Chapter 2) This proposition pertains to this dissertation.
2. Series of Local Triangulations (SOLT) provides an intermediate representation that decouples local geometry reconstruction from sampling-density control, enabling feature-preserving resampling of non-uniform point clouds. (Chapter 3) This proposition pertains to this dissertation.
3. Enclosed negative-volume extraction for computational fluid dynamics can be formulated as a two-stage workflow: first constructing a genus-zero outer shell via shrink-wrap morphology, then applying boolean subtraction. (Chapters 2 and 4) This proposition pertains to this dissertation.
4. A domain-specific programming language with geometry-native types and deterministic execution semantics can improve reproducibility and reduce implementation overhead in geometry-processing workflows. (Chapter 6) This proposition pertains to this dissertation.

The following propositions do not pertain to this dissertation

5. The reliance on perfect, watertight meshes has historically been a bottleneck; future algorithms should be designed to tolerate real-world geometric defects.
6. In industrial CAE, deep learning models still face reliability and scalability constraints; in many workflows, traditional algorithms remain preferable for robustness and interpretability.
7. The reproducibility of a computational result depends as much on the accessibility of its implementation as on the correctness of its derivation.
8. Automation without understanding only amplifies human error; real progress lies in augmenting human expertise rather than replacing it.
9. In computational engineering, benchmark culture should reward transparent failure analysis as much as reported success.
10. Long-term scientific impact depends more on maintainable tools and shared workflows than on one-off demonstrations.

These propositions are regarded as opposable and defendable, and have been approved as such by the promotoren Prof. dr. ir. C. Vuik and Dr. rer. nat. M. Möller.