

A screenshot of a computer

AI-generated content may be incorrect.

Great. I’ll prepare a comprehensive technical overview of the paper *Volume-Aware Design of Composite Molds* with the following:

* Detailed explanation of all computational geometry algorithms used, including novel contributions.
* Complete list of software tools, libraries, and environments (with a focus on Python suitability).
* Stepwise breakdown of the workflow with pseudocode blocks to guide implementation from preprocessing to mold generation.
* Applications and practical use cases such as mold casting and complex geometry handling.
* A review of all stated assumptions and constraints relevant to implementation and modeling.

This overview will be tailored to help you build an educational prototype in Python. I’ll let you know when it’s ready for review.

**Volume-Aware Design of Composite Molds: Technical Overview**

**1. Algorithmic Techniques Employed**

**Volumetric Mold Analysis:** The core idea is to analyze the volume around the object to determine where to split a two-part mold. Instead of relying only on surface visibility (as in previous methods), the paper introduces an *object-volume aware* approach. They construct a tetrahedral representation of the volume between the object’s surface and an outer boundary (e.g. convex hull or offset surface) and use shortest path computations in this volume as proxies for feasible “escape paths” of mold material[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Our%20input%20is%20a%20closed,we%20locate%20the%20additional%20membranes)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=paths%20from%20the%20interior%20points,The%20mold%02able%20objects%20are). This volumetric analysis identifies internal cut surfaces (termed **membranes**) in the flexible silicone mold part such that each mold piece can be removed without collision. The approach enables casting extremely complex shapes (thin features, non-zero genus surfaces, even multiple interlocked parts) using only two mold pieces[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=simulations%2Cwas%20demonstrated%20in%20,be%20cast%20using%20previous%20techniques), whereas previous mold design techniques would fail or require many more pieces[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=paths%20from%20the%20interior%20points,The%20mold%02able%20objects%20are).

**Parting Direction Optimization:** To split the mold into two halves, the algorithm searches for two directions from which the object is mostly visible. It samples many candidate directions on the unit sphere and uses GPU-accelerated rendering to find how much of the object’s surface is visible vs. hidden for each direction[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=We%20begin%20by%20finding%20the,2). The two directions that together minimize the total hidden area are chosen as the **parting directions** (d1, d2)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf" \l ":~:text=We%20begin%20by%20finding%20the,2" \t "_blank). This ensures each mold half can cover as much of the object’s surface as possible, reducing the need for additional pieces. This is a **heuristic search** on the Gaussian sphere, and it improves on naive or opposite-direction guesses by explicitly minimizing the non-visible area[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=candidate%20direction%2C%20we%20use%20GPU,faces%20F1%20and%20F2%20of). (Notably, the chosen directions need not be opposite; any two directions that cover the surface well are acceptable[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=the%20visible%20and%20non,and%20then%20use%20a%20greedy).)

**Escape Path and Shortest-Path Analysis:** Given the parting directions, the volume between the object and outer boundary is conceptually split so that one mold piece corresponds to points escaping in direction d1 and the other to d2. To compute this formally, they find **shortest paths** from every interior point in the volume to the exterior boundary. These shortest paths are computed on the graph of the tetrahedral mesh, using a customized metric that penalizes paths running too close to the object’s surface[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=The%20placement%20of%20cutting%20membranesin,makes%20traveling%20near%20the%20object)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=membranesthattravelthevolume%20almosttangentially%20to%20theobject%20surface,right). (They multiply edge lengths by a factor $w = e^{\alpha d\_m}$, where $d\_m$ is the distance of the edge’s midpoint to the object surface, making paths near the surface effectively “longer”[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=boundary,right). This encourages cut surfaces to intersect the object at near-right angles, avoiding long thin silicone flaps[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=boundary,right)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=membranesthattravelthevolume%20almosttangentially%20to%20theobject%20surface,right).) Dijkstra’s algorithm is used to find these weighted geodesics for all interior vertices efficiently[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=to%200,of%20the%20convex%20hull%20may). Also, instead of using the raw convex hull as the escape boundary, they compute an **offset surface** slightly larger than the convex hull to better capture concave shape features (via a level-set using OpenVDB)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf" \l ":~:text=forthe%20bowl%20in%20Figure%207,showsthe%20effects%20of%20the%20two" \t "_blank). A bias is added to path lengths so that paths effectively “aim” for this offset surface[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=M%20that%20encloses%20the%20convex,5%20COMPOSITE%20MOLD%20FABRICATION), ensuring the escape routes account for the object’s overall shape (see *Fig. 7* in the paper for the effect[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=membranesthattravelthevolume%20almosttangentially%20to%20theobject%20surface,right)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=difficultto%20handle%20%28Figure%207,right)).

**Parting Surface Determination:** With shortest escape paths in hand, the algorithm identifies the primary **parting surface** (the main cut) by separating interior edges whose endpoints escape through different outer regions. First, the outer boundary (∂H) is partitioned into two regions (∂H1 and ∂H2) corresponding to the two parting directions; a greedy flood-fill on the convex hull’s faces assigns each face to the side whose normal best aligns with d1 or d2[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Given%20the%20two%20parting%20directions%2C,Then%2C%20the%20edge%20is%20traversed)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=%E2%88%82H%20whose%20normals%20best%20align,vertices%20wi%20and%20wj%20belong). Then, for each tetrahedral edge inside the volume, if one endpoint’s shortest path leads to ∂H1 and the other’s leads to ∂H2, that edge is cut by the parting surface[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=interior%20volume%20as%20follows%3A%20For,between%20%E2%88%82H1%20and%20%E2%88%82H2%20is). This effectively yields a set of “cut” edges forming a continuous dividing surface through the volume (see **Figure 5a** in the paper)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf" \l ":~:text=interior%20volume%20as%20follows%3A%20For,between%20%E2%88%82H1%20and%20%E2%88%82H2%20is" \t "_blank). The parting surface splits the silicone volume into two halves (O1 and O2) corresponding to the two mold pieces[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=%E2%88%82H2%2C%20according%20to%20the%20alignment,Wecomputetheshortest). The method does *not* require that these two pieces are already fully removable—any remaining undercuts will be handled next by additional cuts[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=the%20non,normal%20to%20d1%20and%20d2).

**Additional Internal Membranes:** After the initial two-part division, the algorithm looks for **internal features that would still prevent mold removal** within each half. The innovative criterion introduced is based on pairs of neighboring points whose escape routes go around opposite sides of an object feature[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=features%20that%20could%20prevent%20the,We%20intro%02duce%20a%20cutting). In other words, if two adjacent vertices in the volume have to “split apart” and go different ways around a protrusion of the object, that indicates a problematic undercut. The paper formalizes this by considering the loop formed by: the edge between the two interior vertices, their two shortest path routes out to the boundary, and the path along the boundary between those two exit points[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=features%20that%20could%20prevent%20the,vi%20and%20vj%20if%20a)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=introduced%20when%20the%20escape%20paths,if%20a%20discreteapproximationoftheminimalsurfaceboundedby%20the%20edge). If the minimal surface spanning that closed loop intersects the object’s surface (i.e. the object lies in between those two escape paths), then an **additional membrane** (an extra cut in the silicone) is inserted along that interior edge[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=features%20that%20could%20prevent%20the,vi%20and%20vj%20if%20a)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=introduced%20when%20the%20escape%20paths,if%20a%20discreteapproximationoftheminimalsurfaceboundedby%20the%20edge). This topological test (using a discrete minimal surface as a heuristic) finds internal “bridges” of silicone that would wrap around parts of the object. By cutting along these interior edges, the mold piece can flex or open around that feature without tearing[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=features%20that%20could%20prevent%20the,We%20intro%02duce%20a%20cutting)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=We%20can%20formalize%20the%20idea,if%20a%20discreteapproximationoftheminimalsurfaceboundedby%20the%20edge). Importantly, these additional membranes do **not** break the mold into more pieces; they start and end on existing boundaries (often running from the object’s surface out to the main parting surface or another membrane), creating thin silicone flaps that remain attached but can bend. This approach is novel compared to prior work, as it uses a volumetric/topological condition (shortest-path loops and minimal surfaces) to systematically detect undercut features that require internal cuts.

**Non-Manifold Surface Extraction:** Once all edges that need to be cut (either by the main parting surface or additional membranes) are flagged, the union of those cut edges implicitly defines a complex surface inside the volume. This cut surface can be non-manifold – multiple cut patches can meet at a line or at the object’s surface. The paper develops a custom **marching tetrahedra** algorithm to extract the triangulated surface geometry of all these membranes[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=4,or%20not)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=4,standard%20geodesics%20could%20result%20in). Marching tetrahedra is akin to marching cubes but on a tetrahedral grid; here each tetrahedron has some edges marked “cut” and others not. There are $2^6=64$ possible configurations of cut vs. uncut edges in a tetrahedron[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=4,objectsur%02face%20mesh%20M%20and%20the). They enumerate these cases (expanding on classic algorithms by Bloomenthal et al. for surfaces in volumes) to produce a consistent triangular mesh that cuts through the tetra mesh according to the flags[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=4,objectsur%02face%20mesh%20M%20and%20the). This mesh $C$ represents the entire layout of internal mold cut surfaces (both parting and additional membranes). It will generally meet the object’s surface $M$ along some curves and meet the outer boundary ∂H along a closed loop (where the parting surface hits the mold outer edge), and can have internal non-manifold junctions where membranes connect. Ensuring topological consistency here is critical – the result is essentially the blueprint of the silicone’s internal seams.

**Surface Smoothing:** Because the cut surface $C$ is initially formed on the discrete tetra grid, it may be jagged or uneven. The authors apply a **Laplacian smoothing** procedure to $C$ with constraints to preserve important boundaries[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=4,interior%20verticesto%20the%20bound%02ary%20according)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=the%20boundary%20vertices%20only,5%20Shortest%20path%20computation). In practice, they perform smoothing in two stages: first moving points along the non-manifold boundary curves (and the intersections with the object and outer surface) slightly, then re-projecting them exactly back onto $M$ or ∂H[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=the%20boundary%20vertices%20only,cutting%20membranesin%20themold%20volume%20depends). Then they smooth the interior of each membrane patch while keeping those boundary curves fixed[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=those%20vertices%20onto%20the%20original,cutting%20membranesin%20themold%20volume%20depends). A damping factor (they used 0.5) is applied in iterative smoothing to ensure convergence without distortion[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=originally,from%20interior%20vertices%20to%20the). The result is a cleaner, more gently curving set of membrane surfaces (see *Fig. 6* in the paper for a comparison[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=boundary,makes%20traveling%20near%20the%20object)). This improves the mold’s physical performance, since smoother silicone parts align better and have fewer tiny slivers or sharp corners that could tear or misalign.

**Optimal Pouring Direction via Persistent Homology:** Designing the mold geometry is only part of the challenge; one must also decide how to orient the mold during casting. If the mold is filled from a poor direction, air can become trapped in concave regions (local maxima relative to gravity) leading to bubbles and casting defects. The paper tackles this by treating each potential pouring orientation as a height function on the object’s surface and analyzing its **topological critical points**[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=5,of%20local%20maxima%20for%20a)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=direction,surface%20between%20the%20silicone%20part). They specifically use **persistence pairing** from computational topology (Edelsbrunner et al. 2000) to rate the significance of each local maximum of the height function[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Weuse%20thepairing%20mechanismfrompersistencehomology%20,and%20the%20hard%20plastic%20shell)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=given%20pouring%20direction,it%20merges%20with%20an%02other%20component). A local maximum paired with a saddle (in the Morse theoretic sense) and having a large persistence value corresponds to a deep concave pocket where air would likely get trapped[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=given%20pouring%20direction,it%20merges%20with%20an%02other%20component)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Therefore%2C%20pairs%20,the%20sorting%20criterion%2C%20to%20get) (these are the red regions in Fig. 12 left[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Fig.%2012.%20Left%3A%20maximum,Milnor)). Shallow or insignificant maxima (low persistence) are considered harmless or easily resolved by slight tilting. For a given direction, they sum up a weighted “air trap score” based on the volume/area of regions that would trap air even after tilting the mold by a small angle[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=5,to)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=amount%20of%20air%20which%20would,for%20silicone%20pouring%20and%20choose). The algorithm then searches for the orientation that minimizes this score. In practice, they evaluate many sample directions and choose the best for pouring the resin. They do this for both the silicone casting (when creating the silicone mold pieces in the metamold) and for the final resin casting inside the composite mold[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=5,to)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=score%20of%20a%20set%20of,Ta%02ble%201%20reports). Notably, the chosen silicone pour directions for the two mold halves were kept nearly aligned, and the resin pour direction was chosen near the bisector of those, to ensure the silicone has a flat level when curing[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=score%20of%20a%20set%20of,Ta%02ble%201%20reports). This use of **topological persistence** is a key algorithmic contribution: it provides a robust way to quantify and compare “trapped pocket” features of different orientations, whereas a naive geometric analysis might miss subtle traps or overestimate minor ones.

**Geometric Interface Design:** Another technical element is designing the interface between the hard shell and the silicone parts. Two strategies are discussed: (a) using an **inflated convex hull** of the object as the outer shape of the silicone, or (b) using a **custom offset surface shaped by the pouring direction’s height field**[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=We%203D%20print%20a%20metamold,to%20design%20the%20interface%20surface). The latter produces a more uniform silicone thickness (they chose a fixed ~15 mm offset) and a flatter top surface, which is easier to cast and more flexible[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=with%20the%20resin%20pouring%20direction%2C,directions%20that%20prevent%20the%20presence). They also add **alignment features** on the parting surface – e.g. applying a Perlin noise displacement so that the two silicone halves interlock with a unique fit[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Figure%209%20summarizesthe%20whole%20fabrication,base%20is%20orthogonal%20to%20the). This ensures the mold can be reassembled in exactly the right position, preventing misalignment of the cast. Additionally, small **air vent channels** are incorporated (e.g. thin tunnels in the mold) to allow air to escape while pouring material[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Figure%209%20summarizesthe%20whole%20fabrication,base%20is%20orthogonal%20to%20the)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=match%2C%20using%20Boolean%20operations%20,directions%20that%20prevent%20the%20presence). These design elements don’t change the core algorithms, but they are important for a functional prototype.

In summary, the paper’s novel algorithms include the volumetric escape path analysis for cut placement, the topological criterion for undercut-driven membrane insertion, and the use of persistent homology for optimizing mold orientation. These are built on classic computational geometry techniques (tetrahedral meshing, graph shortest paths, marching tetrahedra, etc.) combined in a new way to handle unprecedented shape complexity[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=simulations%2Cwas%20demonstrated%20in%20,be%20cast%20using%20previous%20techniques)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Our%20technique%20is%20based%20on,holes%20cannot%20be%20reproduced%20by).

**2. Software Tools, Libraries, and Python Equivalents**

The implementation in the paper uses a combination of custom code and existing geometry libraries. Below we identify the tools referenced or implied, and discuss Python-compatible alternatives:

* **Tetrahedral Meshing**: The paper used **TetWild**[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=TetWild%20,shapes%20well%20enough%20for%20all) and sometimes **TetGen**[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=the%20tetrahedralization%20software%20we%20used,geometry%20generation%20procedure%2Crequiring%20the%20computationof) to generate a tetrahedral mesh of the mold volume. TetWild (Hu et al. 2018) is a robust mesher that can handle complex geometry by trading off exact boundary conformity for element quality[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=TetWild%20,shapes%20well%20enough%20for%20all). TetGen is a well-known Delaunay-based tetrahedralization tool. In Python, one can use **PyMesh** or **meshpy** (TetGen via C++ library) to perform similar volume meshing. *PyMesh* in particular provides wrappers to TetGen, and one could integrate TetWild via its Python bindings if available (TetWild’s code might be wrapped in PyMesh or one can call it as an external process). Another alternative is **CGAL’s 3D mesh generator** (which can be accessed via CGAL’s Python bindings or third-party wrappers) to tetrahedralize the space between two surfaces. Ensure the input mesh $M$ is watertight and define an outer boundary (e.g. convex hull) to mesh the in-between volume.
* **Distance Field / Offsetting**: To compute offset surfaces (like the expanded outer boundary ∂F), the authors used **OpenVDB**[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=used%20OpenVDB%20,is%20robust%20and%20produced%20a). OpenVDB is a C++ library for volumetric grids and level sets; it can dilate (offset) a surface efficiently and generate a mesh for the offset. In Python, one could use **PyOpenVDB** (if available) or sample the object into a voxel grid using libraries like **scikit-image** or **vtk** and then perform a morphological dilation to approximate an offset. Another approach is to use **Open3D** or **trimesh** to voxelize and compute signed distance fields. The key requirement is to compute distances from the object surface to find the offset that encloses the object’s convex hull (the paper used an offset radius equal to the convex hull’s max distance to $M$[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=the%20shortest%20paths%20to%20a,showsthe%20effects%20of%20the%20two)). Python’s scipy.ndimage or skimage.morphology can perform distance transforms on volumetric data, which can then be thresholded.
* **GPU Rendering for Visibility**: The parting direction search relies on rendering the object from many directions to evaluate visible surface area[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=We%20begin%20by%20finding%20the,2). The authors mention using a GPU rasterization approach. In Python, one might use **OpenGL** (via **PyOpenGL**) or a high-level tool like **ModernGL** or **Pyrender** to render the mesh off-screen. Another approach is to use **PyTorch3D** or **Open3D**’s visualization, but obtaining exact visible area might require custom shaders or reading the depth buffer. A simpler but slower alternative is ray-casting: for each candidate direction, cast rays (or sample points) towards the object and determine hits, though this is likely too slow in pure Python for complex models. If an OpenGL context is available, one can render the mesh with a flat shader and count pixels (with known projection area per pixel) to estimate visible area. There are also Python-friendly rasterizers like **vedo** or **vtk** that could compute visible surface via camera projection. The GPU step is performance-critical; in an educational prototype, you might reduce the number of directions or resolution if not using actual GPU rendering.
* **Graph Shortest Paths**: Computing weighted shortest paths on a graph of potentially millions of nodes (tetrahedral vertices) was done via Dijkstra’s algorithm[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=to%200,of%20the%20convex%20hull%20may). In Python, one could use **networkx** for generic graphs, but that would be far too slow for millions of vertices. Instead, one should rely on the fact that this is a geometric graph. A better approach is to use **SciPy**’s sparse matrix functions or implement a custom Dijkstra using arrays (e.g. using heapq for the priority queue). For a prototype with smaller meshes, a straightforward Dijkstra is fine; for large scales, this part would ideally use a C++ extension or algorithms from libraries (CGAL has similar computations for geodesics, for example). If available, one might leverage the **GUDHI** library (which is more for topology) or simply call out to a C++ routine via Cython for efficiency.
* **Persistent Homology**: The persistent homology analysis for pouring direction used a classical algorithm to compute persistence pairs of critical points (maxima and saddles)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf" \l ":~:text=given%20pouring%20direction,point%20creating%20a%20topological%20feature" \t "_blank)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Therefore%2C%20pairs%20,the%20sorting%20criterion%2C%20to%20get). The authors cite Edelsbrunner et al. 2000 and likely implemented a form of persistence diagram for 0-dimensional homology of the superlevel sets (essentially a merge tree of the height field). In Python, the **GUDHI** library provides tools for computing persistence diagrams for scalar functions on meshes. One could also use **Ripser** or other topological libraries, but since we just need persistence of maxima, an easier route is to compute the **Morse-Smale complex** or simply simulate the union-find algorithm for sublevel sets. The height function can be sampled at vertices; then critical points and their persistence can be derived by sorting and union-find. GUDHI’s extended persistence might directly give the persistence of maxima versus 1-saddles. Another option is using **Dionysus** (a Python library for persistent homology). For an educational prototype, if coding from scratch, one can implement a simplified version: find all local maxima on the mesh, then gradually “flood” from the highest peaks downward until regions merge at saddles, measuring the height difference (this difference is the persistence of that maximum). This can be done with a priority queue (akin to Kruskal’s algorithm for minimum spanning tree but in reverse height order).
* **Mesh Processing and Booleans**: For Boolean operations (such as subtracting volumes or cutting the shell and metamold), the authors reference Zhou et al. 2016 for robust mesh booleans[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=details%20of%20the%20parting%20surface,directions%20that%20prevent%20the%20presence). In Python, robust boolean operations can be done with **CGAL (Python bindings)** or the **OpenSCAD** computational geometry kernel via wrappers. The library **trimesh** has boolean operations (internally using either CGAL or other methods if installed). Another reliable option is **Carve** or **MeshBoolean** (though those might need compilation). For creating the hard shell prism and subtracting the mold cavity, one could also use an **OpenVDB** approach: voxelize the shell and silicone volume and subtract volumes on a grid (which is robust to numerical issues), then convert back to mesh. The interface surface design (inflated convex hull or offset) can also be done by taking the object’s convex hull (via **scipy.spatial.ConvexHull**) and extruding or offsetting it; the silhouette-based prism shape can be obtained by projecting points on the hull onto a plane. In Python, generating a prism aligned to a direction could be done by taking the convex hull’s projection onto the plane perpendicular to the chosen direction (use something like shapely for 2D hull of that projection) and extruding. These steps might be simplified in an educational setting by using primitives (e.g., a big cylinder or box that encloses the object).
* **Smoothing**: Smoothing the membrane surface can be achieved with Laplacian smoothing. Libraries like **Open3D** or **trimesh** allow simple Laplacian smoothing of a mesh. Otherwise, implementing Laplacian smoothing in Python is straightforward: for each vertex, move it slightly toward the centroid of its neighbors iteratively. Just remember to constrain certain vertices (those on the object surface or outer boundary should stay put on those surfaces). If using Open3D, you can mark boundary vertices and only smooth the rest, or adjust the algorithm to re-project boundary ones after smoothing. Another approach is to treat the cut surface as an implicit function and use a smoothing filter in the volumetric domain (but that’s overkill here). Given the need to preserve exact contact with the object, a practical approach is: identify vertices on $M$ and ∂H by proximity and pin them; smooth internal vertices; then push any that drifted on boundaries back to the original boundary.
* **Miscellaneous**: The authors mention adding Perlin noise for alignment keys on the parting surface. In Python, one could use a noise library like **noise** (Perlin noise implementation) to perturb vertices on the parting surface mesh a small amount along the surface normal. For generating the metamold (the inverse mold to cast silicone), one might use the results of the boolean operations: essentially, the metamold for one half is the negative volume of the silicone (including membranes) within the hard shell piece. This likely requires computing a mesh for the silicone volume (object plus internal cut surfaces) and subtracting it from a solid block that fits into the shell half. Tools like **Blender’s Python API** or **Trimesh** can be helpful for complex boolean and inversion operations if precision is not paramount in the prototype.

**Language considerations:** Many of these libraries (TetGen, CGAL, OpenVDB) are C++ based, but Python bindings or CLI tools exist. In a Python prototype, one might combine calls to compiled binaries (e.g., run TetGen via subprocess on an .OFF file) and use file I/O to bring results back into Python. Memory and performance are concerns if we try to handle millions of tetrahedra purely in Python – therefore it is often necessary to use optimized libraries or simplify the model for demonstration. The key is to maintain the fidelity of the algorithms while possibly operating on down-sampled data. For example, one could voxel-downsample a very dense mesh for the purpose of computing cut layout, then project the cut surfaces back onto the original mesh for accuracy (though as the paper notes, transferring results between resolutions can be non-trivial[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Finally%2C%20a%20practical%20limitation%20is,approach%20significantly%20ex%02tendsthe%20range%20of)).

In summary, a Python implementation would likely use a hybrid of: **PyMesh/TetGen** (for meshing), **NumPy/SciPy** (for numeric computations like Dijkstra and distance), **PyOpenVDB or skimage** (for offsets), **GUDHI** (for persistence analysis), **trimesh/CGAL** (for booleans), and basic mesh libraries for smoothing and transformations. Each step has at least one Python-friendly solution, though integration and handling large data would be the main challenges.

**3. Step-by-Step Mold Generation Workflow**

This section breaks down the entire pipeline – from input mesh to final mold – into sequential steps, with technical details and pseudocode for key algorithms. The goal is to outline a **modular implementation** where each stage can be understood and prototyped independently.

**Step 1: Input and Volume Preprocessing**

**Input:** A closed, manifold triangle mesh $M$ representing the object to cast. It should be watertight (no holes) and ideally oriented (normals pointing outward). Any extremely thin features or small gaps should be noted, as they influence required mesh resolution.

**Outer Boundary Definition:** Compute an outer boundary surface that will serve as the exterior of the silicone mold. The simplest choice is the convex hull of $M$. Often, we then offset this surface outward by a margin to ensure a minimum mold thickness. For example, the paper inflates the convex hull or uses a custom offset to ~15mm away from the object[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=with%20the%20resin%20pouring%20direction%2C,directions%20that%20prevent%20the%20presence). We’ll call the final chosen outer surface ∂H. This surface should fully enclose $M$ and ideally be somewhat smooth (no sharp spikes).

* *Computing Convex Hull:* Use scipy.spatial.ConvexHull on $M$’s vertices (or another hull algorithm) to get a set of faces for ∂H.
* *Offsetting:* Optionally, push ∂H outward by a fixed amount. In practice, one can voxelize the space and dilate, or move each vertex of the convex hull outward along its normal. Ensure that during offset, ∂H still encloses $M$ completely.

**Tetrahedralize the Volume:** Fill the volume between $M$ and ∂H with tetrahedra. This creates a volume mesh $H$ in which $M$ is an interior boundary and ∂H is the outer boundary. Each tetra’s vertices carry a label (inside the mold volume). Tools like TetWild can produce a high-quality mesh even if $M$ has many details[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=TetWild%20,shapes%20well%20enough%20for%20all). The output is a set of vertices $V$ and tetrahedron cells connecting them. Mark all faces of $H$ that correspond to $M$ as “object boundary” and those on ∂H as “exterior boundary.”

**Data Structures:** We will need to traverse adjacency in this mesh. Build from $H$:

* A graph of vertices $V$ with edges $E$ (we consider each tetra edge as an edge in the graph).
* For each vertex, we’ll store a placeholder for shortest path distance and path endpoint, etc.
* For boundary faces, we know which side (object or outer) they belong to.

**Pseudo-code (Volume Prep & Meshing):**

pseudo

CopyEdit

mesh = load\_triangle\_mesh("object.obj")

outer\_surface = compute\_convex\_hull(mesh.vertices)

outer\_surface = offset\_surface(outer\_surface, distance=15mm)

H = tetrahedralize(volume\_between=mesh and outer\_surface)

for each vertex v in H.vertices:

if v lies on object surface M:

v.type = "object\_boundary"

else if v lies on outer surface ∂H:

v.type = "exterior\_boundary"

# Determine which side of ∂H (to be assigned later to ∂H1 or ∂H2)

else:

v.type = "interior"

*Comment:* In practice, tetrahedralization will return not only vertices but also information about which facets are which. We flag vertices on $M$ or ∂H by checking if they belong to those facets.

**Step 2: Parting Direction Selection (Visibility Analysis)**

The goal of this step is to find two directions for which the object’s surface can be divided such that each mold half sees as much of the surface as possible. We call these parting directions $d\_1$ and $d\_2$, and they will correspond to how the two mold pieces separate.

**Surface Visibility Computation:** We sample $k$ directions evenly on the unit sphere (e.g., via a Fibonacci spiral or simply a dense regular sampling). For each direction $d$, we compute the total area of the object’s surface $M$ that is **visible** when looking from that direction (i.e., facing the object along $d$). More importantly, we note the area that is **not visible** (the hidden area on the far side or in undercuts relative to $d$).

This can be done by rendering the object from direction $d$ (orthographic projection) and marking which triangles are hit by the view. Alternatively, perform a raycast from each triangle’s outward normal direction dot $d$: if the dot product is below 0 (triangle faces away) or if another triangle obstructs it, mark it invisible. A robust approach is GPU rendering: render depth from that direction and mark triangles whose fragments appear. The paper used GPU rendering to accelerate this[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=k%20candidate%20parting%20directions%20on,2).

**Selecting Best Directions:** We need two directions. One brute-force approach:

* Compute for each direction $d\_i$ a *visibility mask* of which surface triangles are visible.
* Then for any pair $(d\_i, d\_j)$, combine their masks to see which triangles are visible from at least one of the two. Compute the total non-visible area under that pair.
* Choose the pair with minimum combined hidden area.

However, testing all pairs of $k$ directions is $O(k^2)$. The paper took a simpler approach: they picked the top two individual directions that minimized hidden area[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=k%20candidate%20parting%20directions%20on,2). In many cases, one direction covers one side, the second naturally should cover the opposite side where the first had most hidden area. They also note that they do *not* require these directions to guarantee full mold removability on their own[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=the%20visible%20and%20non,and%20then%20use%20a%20greedy), since additional membranes will handle leftover undercuts.

For an implementation, you might do a greedy approach:

1. Pick $d\_1$ as the direction with minimum hidden area (max visible area).
2. Pick $d\_2$ as the direction (not too close to $d\_1$ ideally) that maximally improves coverage of what $d\_1$ missed.

**Pseudo-code (Parting Direction Search):**

pseudo

CopyEdit

directions = sample\_sphere(density=N) # e.g., N ~ 100 or more

visibility = [] # list to store hidden area for each direction

visible\_faces\_list = []

for d in directions:

visible\_faces = render\_visible\_triangles(M, view\_dir=d)

visible\_faces\_list.append(visible\_faces)

hidden\_area = total\_area(M) - area(visible\_faces)

visibility.append(hidden\_area)

# Find best pair

best\_pair = None

min\_hidden\_pair\_area = inf

for i in range(len(directions)):

for j in range(i+1, len(directions)):

# combine sets of visible faces

combined\_visible = visible\_faces\_list[i] ∪ visible\_faces\_list[j]

hidden\_area\_pair = total\_area(M) - area(combined\_visible)

if hidden\_area\_pair < min\_hidden\_pair\_area:

min\_hidden\_pair\_area = hidden\_area\_pair

best\_pair = (i, j)

d1 = directions[best\_pair[0]]

d2 = directions[best\_pair[1]]

*Optimization:* Instead of a double loop, one could note that if $d\_1$ is best single, $d\_2$ likely covers the complement. But the above brute force ensures optimal pair at cost of O($k^2$). For moderate $k$ (hundreds), this is manageable.

At the end of this step, we have $d\_1$ and $d\_2$ defined. These will be used to guide how we label the outer boundary ∂H and ultimately how the mold splits.

**Step 3: Partitioning the Mold Volume (Main Parting Surface)**

With two parting directions, we divide the outer boundary ∂H into two regions ∂H1 and ∂H2, then use shortest path analysis to determine the parting surface through the volume.

**Partition Outer Surface:** The convex hull (or outer surface ∂H) is segmented into two contiguous patches: one roughly facing $d\_1$ and the other facing $d\_2$. The paper chooses a starting face $F\_1$ on ∂H whose normal is most aligned with $d\_1$, and $F\_2$ whose normal aligns with $d\_2$[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Given%20the%20two%20parting%20directions%2C,Then%2C%20the%20edge%20is%20traversed). Then a greedy region-growing is done: expand from $F\_1$ adding neighboring faces whose normals favor $d\_1$ until a boundary is reached, and similarly for $F\_2$[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Given%20the%20two%20parting%20directions%2C,Then%2C%20the%20edge%20is%20traversed). This yields ∂H1 and ∂H2 (meeting along a dividing curve on ∂H). We mark each *exterior boundary* vertex as belonging to ∂H1 or ∂H2 accordingly.

* Implementation: Do a DFS/BFS on the dual graph of ∂H (graph of faces). Use a priority or condition: at the frontier, compare alignment of candidate face’s normal with $d\_1$ vs $d\_2$ to decide which side to assign it. (The original method just grew from both simultaneously; any face not clearly aligned might be absorbed by one side arbitrarily, except those very near the boundary line, which they leave as a small buffer[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=correspondtothetwosiliconemoldpieces,the%20convex%20hull%20bounding%20boxdiagonal) to avoid noise.)

**Compute Weighted Shortest Paths:** Now, for each interior vertex $v \in V$, we want the shortest path from $v$ to *any* vertex on the outer boundary ∂H (exterior) **in terms of the weighted metric**. This gives us two pieces of information: the distance and the identity of the boundary vertex (or face) where the path ends. We can do this by multi-source Dijkstra: initialize all exterior boundary vertices with distance 0, put them in a min-heap, and propagate inward through the tetra graph. The weight of an edge between vertices $u$ and $v$ is defined as the Euclidean length $\ell(u,v)$ times a factor that depends on distance from the object. The paper’s weighting: $w(u,v) = \ell(u,v) \cdot \exp(\alpha \cdot \max(d\_M(u), d\_M(v)))$ where $d\_M(x)$ is the distance from point $x$ to the object surface $M$[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=membranesthattravelthevolume%20almosttangentially%20to%20theobject%20surface,right)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=difficultto%20handle%20%28Figure%207,right). They set $\alpha=0.25$ in experiments[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=to%200,of%20the%20convex%20hull%20may). This makes paths that hug the object surface effectively longer.

Additionally, to simulate the offset boundary ∂F, we add a constant bias to paths that exit at certain parts of ∂H: if a boundary vertex lies on ∂H, we can precompute its distance to the offset surface and treat that as an extra cost to add when that vertex is reached[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=M%20that%20encloses%20the%20convex,5%20COMPOSITE%20MOLD%20FABRICATION). A simpler educational approach: include that offset layer in the tetra mesh itself (if not too costly), or ignore this nuance if focusing on main ideas.

After running Dijkstra, each interior vertex $v$ will have:

* v.dist: the distance of the shortest weighted path to the exterior.
* v.exit: a pointer or id of the exterior vertex where that path hits ∂H.

Because we started at all boundary nodes, each interior vertex automatically got the path to the *closest* boundary (in weighted sense). Now, using the labels from ∂H1/∂H2:

* If v.exit is in ∂H1, we consider $v$ as belonging to mold piece 1; if in ∂H2, then piece 2.

**Identify Parting Surface Edges:** We iterate over every edge (or every pair of adjacent interior vertices) in the tetra mesh. For an edge connecting vertices $v\_i$ and $v\_j$:

* If one has exit in ∂H1 and the other in ∂H2, then this edge is cut by the parting surface[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=interior%20volume%20as%20follows%3A%20For,between%20%E2%88%82H1%20and%20%E2%88%82H2%20is). Mark it as a **parting cut**.
* If both share the same exit side, the parting surface does not cut this edge (they are in the same mold half for now).

Optionally, the paper ignores edges whose endpoints’ exit points lie very close to the dividing line on ∂H (within 1.5% of bounding box diagonal)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf" \l ":~:text=correspondtothetwosiliconemoldpieces,the%20convex%20hull%20bounding%20boxdiagonal" \t "_blank). Those could be numerical noise cases. We can similarly skip edges if the two exit points are actually neighbors on the boundary split line.

At this point, all edges that straddle the two regions define an initial cut surface splitting the volume into two connected sets of tetra (each set corresponding to one silicone piece O1 or O2)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf" \l ":~:text=interior%20volume%20as%20follows%3A%20For,between%20%E2%88%82H1%20and%20%E2%88%82H2%20is" \t "_blank).

**Pseudo-code (Parting Surface via Shortest Paths):**

pseudo

CopyEdit

# Prepare multi-source for Dijkstra:

dist = dict() # shortest distance to boundary

exitID = dict() # which boundary vertex it reaches

pq = MinHeap()

for each vertex v in H.vertices:

if v.type == "exterior\_boundary":

dist[v] = 0

exitID[v] = v # it exits at itself

pq.push((0, v))

else:

dist[v] = infinity

exitID[v] = None

# Dijkstra (weighted by distance from object):

while pq not empty:

(d, u) = pq.pop()

if d > dist[u]: continue # skip stale entry

for each neighbor w of u: # w connected by an edge

# weight = Euclidean length \* exp(alpha \* avg(dist\_to\_object(u), dist\_to\_object(w)))

base\_len = length(u,w)

dm = max(distance\_to\_M[u], distance\_to\_M[w])

wgt = base\_len \* exp(alpha \* dm)

# If u or w is on convex hull, add offset bias for leaving at u.exit:

# (we could add after the fact based on exit point, as in the paper)

if d + wgt < dist[w]:

dist[w] = d + wgt

exitID[w] = exitID[u] # it will exit where u exits (multi-source means boundary have themselves)

pq.push((dist[w], w))

# Now classify edges for parting

parting\_cut\_edges = []

for each edge (i, j) in H.edges:

if exitID[i] != None and exitID[j] != None:

side\_i = (exitID[i] in exterior\_region1) ? 1 : 2

side\_j = (exitID[j] in exterior\_region1) ? 1 : 2

if side\_i != side\_j:

if not near\_boundary\_line(exitID[i], exitID[j]):

parting\_cut\_edges.append((i,j))

This produces the set of edges cut by the main parting membrane.

**Step 4: Detecting Additional Undercut Membranes**

Even after the main split, each mold half may contain internal undercut features – parts of the object that cause the mold material to hook around and prevent a clean removal. To address this, we add *additional internal membranes* (cuts) within each half.

The rule from the paper: if two adjacent interior vertices $v\_i, v\_j$ (connected by an edge in the tetra mesh **and both belonging to the same mold half**) have escape paths that go around different sides of some object geometry, we insert a membrane along that edge[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=features%20that%20could%20prevent%20the,We%20intro%02duce%20a%20cutting).

In practice:

* For each **interior edge** $(v\_i, v\_j)$ that was *not* already cut by the parting surface, consider the loop formed by:
  + the edge $v\_i v\_j$,
  + the shortest path from $v\_i$ to its boundary exit $w\_i = \text{exitID}[v\_i]$,
  + the shortest path from $v\_j$ to its exit $w\_j$,
  + and the path along ∂H between $w\_i$ and $w\_j$ on the outer surface.
* Determine if the object’s surface $M$ intersects the interior of this loop (specifically, if the minimal surface spanning this loop intersects $M$)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf" \l ":~:text=features%20that%20could%20prevent%20the,vi%20and%20vj%20if%20a" \t "_blank)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=%3D%20,if%20a%20discreteapproximationoftheminimalsurfaceboundedby%20the%20edge).

Equivalently, we can interpret this as: do $v\_i$ and $v\_j$ “go around” different sides of the object? If yes, an additional cut is needed.

A simpler (but not perfectly equivalent) heuristic: if $v\_i$ and $v\_j$ belong to the same mold half (same exit side), but **their exit vertices on ∂H are on nearly opposite sides of the object’s silhouette as seen from that mold half**, then the object is between their paths. In graph terms, if the exit points $w\_i$ and $w\_j$ are far apart on ∂H and the straight-line segment between them (through the interior) intersects $M$, that indicates trouble.

To implement robustly:  
For each interior edge $(v\_i, v\_j)$ not already cut:

* Let $w\_i = \text{exitID}[v\_i]$, $w\_j = \text{exitID}[v\_j]$ (both lie on the *same* ∂H region now).
* We can find a path on ∂H between $w\_i$ and $w\_j$ (e.g., along the triangle adjacency on ∂H).
* We now have a closed polyline loop: $v\_i \to ... \to w\_i$ (shortest path), $w\_i \to ... \to w\_j$ (along ∂H), $w\_j \to ... \to v\_j$ (shortest path), and $(v\_j \to v\_i)$ the interior edge.
* We can approximate the minimal surface of this loop by, say, triangulating this loop (it’s like a Jordan curve that likely goes around some object parts).
* Check intersection: one way is to take each triangle of this loop and test if any triangle of the object $M$ intersects it. Or, trace a ray from a point on the edge $v\_i v\_j$ in some direction and see if it hits $M$ an odd number of times (winding test).

A simpler criterion used by the authors: if the *span* of the two escape paths encloses part of the object, that edge should be cut[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=features%20that%20could%20prevent%20the,vi%20and%20vj%20if%20a). We can check this by sampling a point halfway along the edge $v\_i v\_j$ or on the minimal surface and see if that point is inside the object’s volume (for which we can use a point-in-mesh test, since $M$ is closed). If yes, then the object is in between.

**Mark Additional Cuts:** Any edge that meets the criterion gets marked as an additional membrane cut. These edges, together with the parting edges, now form the full set of cut edges.

**Pseudo-code (Additional Membrane Criterion):**

pseudo

CopyEdit

additional\_cut\_edges = []

for each edge (i, j) in H.edges:

if (i,j) not in parting\_cut\_edges:

if exitID[i] and exitID[j] are both not None and exitID[i] == exitID[j]:

# They escape to the same region (same mold half)

# Compute or retrieve the exit vertices w\_i, w\_j

w\_i = exitID[i]

w\_j = exitID[j]

# Find path on outer surface between w\_i and w\_j (e.g., a BFS on ∂H graph)

outer\_path = shortest\_path\_on\_surface(∂H\_graph, w\_i, w\_j)

# Form loop and test intersection

loop = combine\_paths([path\_from(i to w\_i), path\_from(j to w\_j), outer\_path, (j->i edge)])

if object\_intersects(loop):

additional\_cut\_edges.append((i,j))

Here, path\_from(i to w\_i) is simply the sequence of edges from $v\_i$ to its boundary exit (which we could store during the Dijkstra propagation via parent pointers). Similarly for $v\_j$. outer\_path can be found by a simple graph search on the outer boundary mesh (since ∂H is not huge, this is fine). The object\_intersects(loop) function could do a point-in-solid test: e.g., take the centroid of the loop and test if it lies inside the object (perhaps by casting a ray to see if it intersects an odd number of triangles of $M$). If the centroid of that loop is inside $M$, likely the object is encircled by the loop, indicating a needed cut. This is a heuristic; the actual minimal surface intersection is a bit more exact, but this should catch the major cases.

Now we have two lists: parting\_cut\_edges and additional\_cut\_edges. Together, these are all edges in the tetra mesh that will be cut by some membrane.

**Step 5: Constructing the Cut Surface Mesh**

Using the flagged edges, we build the actual geometry of the cut surfaces (membranes). This involves creating triangular facets within each tetrahedron that has some of its edges cut. Essentially, we are reconstructing the implicit surface where an “inside mold” region meets an “outside mold” region or splits internally.

**Extended Marching Tetrahedra:** The classical marching tetrahedra algorithm assumes a scalar field and outputs an iso-surface. Here, we don’t exactly have a scalar field, but we have a binary label on each edge (cut or uncut). We can treat it like an implicit function that is 1 on cut edges and 0 on uncut edges and then extract the 0.5 level surface. The paper explicitly handled all 64 cases of edge cut configurations to produce manifold or non-manifold patches as needed[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=4,objectsur%02face%20mesh%20M%20and%20the). Essentially:

* If an edge is cut, that means the surface passes through that edge (somewhere between the two vertices).
* If an edge is uncut, the surface does not intersect that edge.

Within one tetrahedron (4 vertices, 6 edges): look at which edges are cut. For example:

* If exactly 3 edges forming a closed loop are cut, that indicates a patch that likely connects those cuts.
* If edges on the tetra’s faces are cut in a certain pattern, the surface could pass through connecting them.

The implementation can leverage known patterns: e.g. Bloomenthal’s polygonization of implicit surfaces, or treat it as if each vertex had a binary value (inside/outside of something) and you’re extracting the interface. But here “inside/outside” is not globally defined because the surface is non-manifold (some tetra might have multiple patches).

A simpler approach is the **dual approach**: each cut edge in the mesh could be thought of as generating a face in the dual mesh. The dual of a tetra mesh is a cell complex where each tetra yields a dual vertex (or node). But since the surfaces are non-manifold, using duals (like SurfaceNets) is tricky[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=4,objectsur%02face%20mesh%20M%20and%20the). So we stick with direct construction.

We iterate through each tetrahedron:

* Identify its configuration (which edges are cut).
* If no or all edges are cut, skip (no interface or fully cut — fully cut likely won’t happen because that would separate something completely).
* If some edges are cut, create new vertices at the midpoints of those edges (or at a proportional location if needed; since we don’t have an actual scalar field, midpoints are fine as initial positions).
* Connect these new vertices to form one or more triangles that partition the tetra accordingly. (The paper ensures these patches align across tetra boundaries.)

We must also ensure that where a cut surface touches the object surface $M$ or the outer surface ∂H, the extracted mesh has boundary edges exactly along those contact curves. In practice, if a tetra has a vertex on the object surface and an adjacent interior edge is cut, the cut surface will intersect the object at that vertex. We should include that vertex of $M$ as a corner of the membrane patch. Similar logic for outer boundary.

After processing all tetra, we collect all triangles generated. This yields a mesh $C$ (with possibly non-manifold edges).

**Merge and Clean:** Many tetra share edges, so the generated facets will naturally join up. We must be careful to fuse coincident vertices (e.g., two tetra might cut the same edge, generating the same midpoint — ensure only one vertex exists in the mesh at that location). A union-find or dictionary of edge->newVertex can ensure we reuse the same vertex on a cut edge.

**Pseudo-code (Surface Extraction Skeleton):**

pseudo

CopyEdit

cut\_surface\_mesh = Mesh()

edge\_vertex\_map = {} # maps a cut edge (i,j) to the index of the new vertex on that edge

for each tetrahedron T in H:

# Identify cut edges in this tetra

cut\_edges = []

for each edge e = (a,b) of T:

if e in parting\_cut\_edges or e in additional\_cut\_edges:

cut\_edges.append(e)

if e not in edge\_vertex\_map:

# Create a new vertex at midpoint of e (could weight by distance if needed)

pos = 0.5 \* (vertex\_positions[a] + vertex\_positions[b])

vid = cut\_surface\_mesh.add\_vertex(pos)

edge\_vertex\_map[e] = vid

if cut\_edges == [] or len(cut\_edges) == 6:

continue # no cut or tetra fully separated (the latter likely doesn't happen in two-piece context)

# Determine facet topology based on cut\_edges:

facets = triangulate\_cut\_in\_tetra(T, cut\_edges, edge\_vertex\_map)

for f in facets:

cut\_surface\_mesh.add\_face(f)

The function triangulate\_cut\_in\_tetra would implement the case analysis for how to connect the new vertices within the tetra. This can be complex; one could refer to existing tables from implicit surface polygonization literature:

* If cut edges form a single closed loop (like 3 or 4 edges in a cycle), you create a patch bounded by that loop.
* If cut edges form two disjoint segments, you might get two separate patches in that tetra.

Given the complexity, an easier educational strategy is: treat the indicator of one mold piece vs the other as a scalar field. For instance, label each tetra vertex with a value: 0 if it belongs to mold piece 1, 1 if it belongs to mold piece 2 (after parting cut). Additional cuts complicate this because within one part there can be an internal cut that doesn’t correspond to a global binary classification. So this might require multi-label (not just two, since additional membranes divide regions in one mold piece). The non-manifold algorithm is indeed required to capture that.

For clarity, after this step we have a raw cut surface mesh $C$ representing all the internal membranes.

**Step 6: Smoothing the Membrane Surfaces**

We now improve the quality of the cut surface mesh $C$. The main objective is to remove staircase artifacts and tiny facets from the discrete grid while **preserving the boundary constraints**:

* Edges of $C$ that lie along the object surface $M$ should remain exactly on $M$ (so that the silicone will fit snugly to the object there).
* Edges of $C$ that lie on the outer surface ∂H likewise should stay on ∂H (so that the silicone fits against the hard shell there).
* Edges where multiple membrane patches meet (non-manifold edges within $C$) should probably remain at their intersection (we can allow them to move if it doesn’t break topology, but better keep them as is to maintain the partition connectivity).

The paper’s approach: perform Laplacian smoothing in two phases[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=the%20boundary%20vertices%20only,cutting%20membranesin%20themold%20volume%20depends):

1. Smooth the boundary curves on $C$ that lie on $M$ or ∂H *tangentially* and then snap them back to the surface. For example, if an edge of $C$ is on $M$, we can move its interior vertices slightly along $M$ (without leaving $M$) to even out spacing or curvature. Then project those points back onto $M$ exactly[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=the%20boundary%20vertices%20only,5%20Shortest%20path%20computation). They specifically mention re-projecting after smoothing boundary vertices[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=the%20boundary%20vertices%20only,5%20Shortest%20path%20computation).
2. Smooth the interior points of $C$ (vertices not on any constrained boundary) with a standard Laplacian or HC-smoothing, while keeping the boundary points fixed[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=those%20vertices%20onto%20the%20original,cutting%20membranesin%20themold%20volume%20depends). This will relax the mesh facets to be more planar and less jagged.

We apply a damping factor (like 0.5) to avoid overshooting in each iteration[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=originally,from%20interior%20vertices%20to%20the). Iterate until convergence or a set number of steps (the paper likely did enough iterations to remove visible noise, perhaps 20 iterations or until change is below threshold).

**Pseudo-code (Laplacian Smooth with Constraints):**

pseudo

CopyEdit

# Identify constraint sets:

on\_object = {v in C.vertices | v coincident with some point on M}

on\_outer = {v in C.vertices | v coincident with some point on ∂H}

on\_boundary\_curve = on\_object ∪ on\_outer # all vertices on either physical boundary

internal\_vertices = C.vertices - on\_boundary\_curve

# Optionally smooth boundary curves (keeping them on surfaces):

for v in on\_object ∪ on\_outer:

# Compute centroid of neighbors (standard Laplacian)

nbrs = C.neighbors(v)

centroid = average(nbrs.positions)

new\_pos = v.position \* 0.5 + centroid \* 0.5 # damping 0.5

# Project back to surface:

if v ∈ on\_object:

v.position = project\_onto\_surface(new\_pos, M)

else if v ∈ on\_outer:

v.position = project\_onto\_surface(new\_pos, ∂H)

# Smooth internal vertices:

for v in internal\_vertices:

nbrs = C.neighbors(v)

centroid = average(nbrs.positions)

v.position = v.position \* 0.5 + centroid \* 0.5

Repeat the above smoothing a few times. Ensure that after internal smoothing, if any vertices were supposed to remain on a non-manifold intersection, we might treat those as constrained too (though the paper suggests the non-manifold junctions are bounded by $M$ or ∂H anyway[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=4,objectsur%02face%20mesh%20M%20and%20the), so likely all non-manifold points are on those boundaries or we consider them separately).

Now the cut surface $C$ is finalized, defining where the silicone mold will split. We have effectively designed the flexible mold insert geometry.

**Step 7: Designing and Generating the Hard Shell & Metamolds (Final Mold Components)**

With the silicone cut layout known, we proceed to create the actual physical mold parts:

* Two silicone pieces (each corresponding to O1 and O2 volumes).
* Two hard plastic shell pieces that support the silicone.
* Two “metamold” pieces used to cast the silicone parts (these are temporary molds for making the silicone inserts).

**Silicone Volume and Mold Halves:** The volume of each silicone piece is essentially the region between the object $M$ and the outer interface ∂H, minus the volume removed by the internal membranes (the membranes are cuts, not voids – the silicone still occupies those thin regions, just that they mark where the mold splits). For each half, we can produce a triangle mesh representing the silicone piece by taking the object mesh $M$, the outer interface (clipped to that half), and the cut surface patches that belong to that half, and combining them as boundary of a volume. In other words, mold piece 1’s silicone is bounded by: a portion of $M$, a portion of ∂H, and all relevant membrane patches (parting and additional) that enclose piece 1. The assembly of those surfaces forms a closed volume which can be meshed or used as a solid.

However, we might not need an explicit mesh of the silicone volume for our purposes. Instead, to fabricate:

* **Hard Shell**: They choose a cylindrical or prismatic shape that encloses the silicone. The shell’s interior will be shaped exactly like the silicone’s outer surface, and its outside is just a simple shape (for mechanical stability). The paper specifically used a prism aligned with the resin pouring direction, with a flat base, and whose cross-section matches the silhouette of the convex hull[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=details%20of%20the%20parting%20surface,directions%20that%20prevent%20the%20presence). For example, if pouring direction is “up”, the shell could be a tall cylinder or box around the object, with the top open. We then use a boolean subtraction to carve out the inside: subtract the silicone shape (plus some clearance) from this prism to form the shell’s cavity[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=details%20of%20the%20parting%20surface,directions%20that%20prevent%20the%20presence). They also integrated pour holes and air vents into the shell (small cylinders connecting the cavity to the outside)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf" \l ":~:text=Figure%209%20summarizesthe%20whole%20fabrication,base%20is%20orthogonal%20to%20the" \t "_blank)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=match%2C%20using%20Boolean%20operations%20,directions%20that%20prevent%20the%20presence). In design terms, one can start with a big block, cut out the silicone volume, cut out channels for pouring and air, and that yields the shell half.
* **Metamold**: This is used to mold the silicone pieces. Each metamold is essentially the *inverse* of a silicone piece, including the internal membranes as protruding features. In practice, for each silicone half, the metamold can be made by taking a copy of that silicone volume mesh and treating it as negative: you embed the silicone shape inside a solid and then subtract it to leave a cavity of that shape. However, the metamold does not need to cover the entire silicone; recall how they fabricated: they assembled one shell piece with one metamold piece to create a closed container for casting silicone[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=the%20hard%20plastic%20shell%20,f). That implies each metamold is like a “lid” that completes the container. Likely the metamold includes the parting surface geometry and additional membranes as **solid ridges** (so that those ridges create the corresponding thin gaps in the cast silicone)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf" \l ":~:text=hard%20plastic%20shell,%28see%20Figure%2010" \t "_blank). Essentially, everything that is a cut in the silicone appears as a thin protruding wall in the metamold.

To design the metamold:

* Take the cut surface mesh $C$ for one half (e.g., all parting and additional membranes on piece 1). Thicken those slightly (to a wall thickness that can be 3D printed but as thin as possible, say 1-2mm). These will become the “knife” that carves the membranes.
* The metamold also needs to form the object’s shape on the silicone’s inner side. Possibly they 3D printed a dummy object or integrated the object’s negative into the metamold so that the silicone forms correctly. However, since the silicone directly contacts the actual object in use, one might instead rely on casting the object’s shape by pouring silicone around the actual object or a 3D print of the object. The authors likely 3D printed the object shape as part of the metamold as well (so the silicone is cast with the correct cavity to fit the object)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf" \l ":~:text=first%20time%2C%20the%20proposed%20approach,volumetric%20approach%20overcomes%20previous%20limitations" \t "_blank) (they mention extending mold design to what artisans did – likely making a clay or printed master of the object for the mold).

In a computational design sense, one can create a solid model of the metamold by:

* Taking the hard shell piece and intersecting it with the region of the silicone piece (so basically splitting at the parting surface).
* Then subtracting the silicone volume from it, *but leaving the membranes as negative (void)* or equivalently adding solid where membranes are in the silicone.
* This is tricky to visualize, but essentially: start with a copy of the shell half, carve out everything that should be filled by silicone *except* leave thin separators where you want gaps in the silicone (membrane positions).
* Another way: for each membrane patch in $C$, extend it slightly into the interior as a thin volume (this will be a solid in the metamold that occupies where silicone should not go). Also include a solid representation of the parting surface (maybe the entire open side covered).
* Subtract these from a block that covers the open side of the shell.

Because of the complexity, let's outline a simpler approach:

* Represent the silicone half volume as a closed mesh (we could get this by performing a boolean between the convex hull half and the object plus membranes).
* Generate the shell half by subtracting silicone-half from a prism.
* Generate the metamold half by subtracting the silicone-half from a block that covers the parting opening. Then *invert the membranes* by adding them back in as solids: meaning add thin walls along the cut surfaces in that cavity.

This yields:

* Shell half: rigid frame with correct cavity shape (including space for membranes).
* Metamold half: rigid piece that when assembled with shell half leaves the cavity for silicone.

**Assembly for silicone casting:** Put shell half and metamold half together, pour silicone, let cure, then remove metamold leaving silicone in shell.

Repeat for other side. Then you have two silicone pieces in two shell halves.

Finally, **resin casting:** put the two halves (silicone+shell) together (the parting surfaces of silicone meet, aligned by the Perlin keys), secure them, and pour resin (or other casting material) in from the top (through an opening in the shell)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf" \l ":~:text=additional%20membranes%20%28Figure%203,f" \t "_blank)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=with%20the%20resin%20pouring%20direction%2C,directions%20that%20prevent%20the%20presence). Air escapes through vents, etc. After curing, open the shell, flex the silicone halves off the cast object, and you have your replica.

**Pseudo-code (Designing Shell & Metamold geometry):**  
*(This is more conceptual; in practice, use CAD boolean operations.)*

pseudo

CopyEdit

silicone\_half\_mesh = combine\_surfaces(M\_half, outer\_half, C\_half) # surfaces bounding silicone piece

# Hard Shell half:

prism = make\_prism(shape=silhouette\_of(outer\_half, direction=pour\_dir\_resin), height=h)

shell\_half = boolean\_subtract(prism, silicone\_half\_mesh)

add\_pour\_hole(shell\_half, position=top\_center, radius=small)

add\_air\_vents(shell\_half, positions=some\_high\_corners)

# Metamold half:

cover = make\_block\_cover(outer\_half.opening) # block that covers the parting opening of shell

metamold\_half = boolean\_subtract(cover, silicone\_half\_mesh)

# Now metamold\_half has a cavity of exactly the silicone shape.

# We need to add solid ridges where membranes are:

for each triangle t in C\_half (the cut surface patches):

extrude\_inward = offset\_triangle(t, offset=-ε along its normal into metamold material)

metamold\_half = boolean\_union(metamold\_half, extrude\_inward) # add a thin wall

# (Actually we invert add vs subtract: since metamold currently has a cavity, adding solid in it is a subtraction from silicone volume)

The above is conceptual. In practice, it's easier to build metamold by directly constructing the negative volume of silicone plus membranes.

After this step, we have:

* shell\_half\_1.stl, shell\_half\_2.stl – models for 3D printing rigid shell parts.
* metamold\_half\_1.stl, metamold\_half\_2.stl – models for 3D printing the metamolds (likely in a dissolvable or rigid material).
* We also output perhaps silicone\_half\_1.obj (for simulation or verification) which represents the silicone part in that half.

**Step 8: Selecting Casting Orientation (Pouring Direction)**

We already determined the pouring direction in Step 1 for the resin implicitly (parting direction alignment). However, a more thorough approach is to run the persistent homology analysis to refine the pouring angle, especially if the top of the mold (the pour opening) can be tilted or chosen differently than exactly aligning with one of the parting directions.

**Persistent Homology for Orientation:** We consider the final assembled mold and the casting material being poured in. For each candidate orientation (which can be a small perturbation around the chosen one, or a different angle altogether), define a height function $f(p) = \langle p, \mathbf{g} \rangle$ (dot product with gravity direction) on the *object’s surface*. Compute its local maxima and their persistence as described earlier. Essentially:

* Build the upper envelope of the surface as we lower a horizontal plane. Each time a new component appears (at a local maximum), note at what height it eventually connects into a bigger component (at a saddle).
* The difference in height is the persistence of that maximum[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Therefore%2C%20pairs%20,the%20sorting%20criterion%2C%20to%20get).
* Weight each such feature by the volume of the “bowl” it represents (or area of the region, etc., to estimate trapped air).
* Sum the contributions of all significant maxima for an overall “trap score” for that orientation[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=amount%20of%20air%20which%20would,for%20silicone%20pouring%20and%20choose).

We do this for many orientations (the paper sampled a cone around the bisector of the two parting directions for resin[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=from%20those%20scoring%20lowest%20a,Ta%02ble%201%20reports), and for silicone they chose two nearly aligned directions[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=score%20of%20a%20set%20of,Ta%02ble%201%20reports)). Then choose the orientation with minimum trap score. That’s our pouring direction.

For Python prototyping, one might not implement full persistence diagrams. Instead, a heuristic:

* For each direction, simulate pouring by checking for each concave region facing upward if it has an outlet. For example, do a flood fill of the object’s surface from the lowest point upward; any “basin” that isn’t open on top is a trap. Compute its depth and area.
* This is essentially computing the basins of the height function. Persistence gives a rigorous way to pair each basin’s deepest point (max) with the spillover saddle.

If using GUDHI or similar, we can compute persistence pairs (max-saddle) for the sublevel sets of $-f$ (or superlevel of $f$) to get exactly these features.

Finally, after picking the orientation, ensure the mold design (the shell base, etc.) is oriented accordingly. The shell’s base was made orthogonal to the resin pour direction in our design, so that likely is consistent. If a different direction came out of persistence analysis, one would adjust the shell design to that new direction.

**Finalize:** Output the orientation instructions: e.g., “Mold should be oriented such that direction $(\theta, \phi)$ is vertical during pouring.” Also, any tilt angle allowances – the paper mentions slight tilting can help let bubbles escape from smaller pockets[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=5,to). They assumed the ability to tilt by a small angle $\alpha$ (like 5° or so) during the pour to clear some bubbles[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Fig.%2012.%20Left%3A%20maximum,Milnor). Their scoring took that into account by discounting traps that could be resolved by a small tilt.

**4. Applications and Use Cases**

The **Volume-Aware Composite Molding** technique has broad applications in areas where highly complex shapes need to be reproduced by casting:

* **Art and Cultural Heritage Reproduction:** As noted by the authors, this approach enables casting objects that previously only skilled artisans could handle with multi-part molds[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=We%20have%20introduced%20a%20novel,should%20be%20cut%20and%20opened). Intricate sculptures, museum artifacts, or archaeological finds with complex topology can be duplicated in resin or plaster using a two-piece mold designed by this system. For example, the paper demonstrated casting a knotted sculpture and entangled geometries that would normally defy molding[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=and%20surface%20topology%2C%20we%20define,be%20cast%20using%20previous%20techniques). This could aid restoration (creating replicas of fragile originals) or mass production of art pieces.
* **Industrial Manufacturing of Complex Parts:** In manufacturing, molds (often injection molds or casting molds) typically avoid undercuts, but with this technique, one could create molds for parts with complex internal features or unusual shapes. For instance, a part with internal voids or overlapping elements could be cast in silicone molds produced by this method. While industrial molds are often multi-part metal tooling, this method using 3D printing and silicone is well-suited for **rapid prototyping** or low-volume production of complex parts (for example, figurines, biomedical models, or aerospace components with convoluted shapes).
* **Customization and Design Prototyping:** Designers who create elaborate 3D models (like jewelry with knots, or fashion accessories, or detailed miniatures) can use this to make molds for casting their designs in various materials (resins, wax (for lost-wax casting), etc.). The automation of cut design means one doesn’t need expert knowledge of mold making to get a workable mold. The composite mold (rigid shell + flexible insert) is especially useful for one-off or custom molds because it can be fabricated on consumer 3D printers.
* **Educational and Hobbyist Use:** Hobbyists in the maker community could cast objects that were previously mold-making nightmares. For example, complex tabletop gaming miniatures or cosplay props with intricate geometry could be molded with fewer pieces. Because the technique uses 3D printed parts and silicone, it’s accessible without industrial equipment. It essentially broadens the range of what shapes a hobbyist can mold and duplicate.
* **Robotics and Path Planning Analogy:** While not an application of casting per se, the concept of analyzing volumes for escape paths has parallels in robotic path planning and vision. A robot that needs to figure out how to extract an object from a constraining volume (or how to grasp it from certain directions) could use similar volumetric analysis. The “escape path” concept is akin to finding collision-free paths. For instance, a robot planning to disassemble a tightly interlocked object might compute where to “cut” or separate components in a way analogous to these mold membranes. Additionally, the visibility computations relate to robot vision in determining viewpoints that see the whole object. Thus, the algorithms might inform automated fixturing, assembly/disassembly planning, or even surgical planning (where to cut to extract a complex-shaped tumor, metaphorically).
* **Metal Casting (Foundry) Applications:** The authors suggest potential extension to metal casting[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Besidesitslimitations%2Cwebelieveour%20approach%20significantly%20ex%02tendsthe%20range,very%20hard%20constraints%2C%20and%20further). Casting metals typically uses sand molds or investment casting, but those have limitations for complex shapes. A silicone mold wouldn’t handle molten metal directly, but the concept could be used to 3D print wax models for lost-wax casting of complex metal parts. They even experimented with a pewter-based alloy[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Besidesitslimitations%2Cwebelieveour%20approach%20significantly%20ex%02tendsthe%20range,very%20hard%20constraints%2C%20and%20further). In art foundries, creating wax molds of intricate shapes (to then make ceramic shells) could benefit from this approach.

In essence, any scenario requiring duplication of complex 3D geometry via molding could benefit. The composite mold approach particularly shines when the shape has **internal voids, undercuts, or entangled components**. Traditional mold techniques would avoid such shapes or require many more mold sections. Here we only have two rigid pieces to handle (like a normal two-part mold), and the flexibility of the silicone and strategic internal cuts handle the complexity.

**5. Assumptions, Constraints, and Implementation Challenges**

Despite its power, the method comes with several assumptions and practical constraints that a developer should keep in mind:

* **Input Geometry Requirements:** The input mesh must be a closed, manifold surface (no holes, no non-manifold edges)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf" \l ":~:text=Our%20input%20is%20a%20closed,we%20locate%20the%20additional%20membranes" \t "_blank). The algorithm assumes a clear inside vs. outside. Normals should ideally point outward (for distance calculations and visibility to make sense). If the mesh has multiple connected components, the method can handle it (even entangled components[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=with%20the%20molds%20here%20being,be%20cast%20using%20previous%20techniques)), but all components together still define a single mold cavity. Extremely thin features or very fine details will require a sufficiently fine tetrahedral mesh to capture them; otherwise, the shortest path analysis might “skip over” them. In practice, this means memory and time can blow up for highly detailed models, so some decimation or feature simplification might be needed (the paper notes difficulty with highly detailed boundaries and that multi-resolution approaches are non-trivial[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Finally%2C%20a%20practical%20limitation%20is,approach%20significantly%20ex%02tendsthe%20range%20of)).
* **Physical Mold Constraints:** The object must be moldable at all in a two-piece flexible mold. Certain shapes are impossible to cast even with this method:
  + Objects that have a **fully enclosed void with no opening** (like a bottle with a long neck but a big sealed chamber) cannot be cast without a core, as noted[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=The%20main%20limitations%20of%20the,object%20into%20a%20few%20separateparts). The algorithm might design a mold, but when pouring, the material would trap air or the part would be locked in.
  + Extremely **long, thin appendages** that are attached by a small area (e.g., a long spear in a statue’s hand) can be technically molded, but the resulting silicone would have to encompass that long thin cavity, making the mold huge or fragile[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=The%20main%20limitations%20of%20the,object%20into%20a%20few%20separateparts)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=simple%20casting,Furthermore%2C%20even%20if%20our). The authors mention the mold might become unnecessarily large or complex and suggest in such cases even their approach might yield an impractical result (the user might consider splitting the object itself into pieces to mold separately)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf" \l ":~:text=thin%20parts%20that%20are%20almost,Furthermore%2C%20even%20if%20our" \t "_blank).
  + If the algorithm produces a **very thin silicone membrane** (thin strand of silicone) as part of the mold, that piece could deform or tear during casting[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=this%20kind%20of%20decomposition%20in,is%20an%20interesting%20research%20challenge). For example, a spherical object with a very thin tunnel would cause a thin silicone plug to form; it would satisfy geometric moldability, but physically it might flop or block resin flow[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=formulation%20produced%20good%20membranes%20for,many%20robust%20tech%02niques%20that%20claim). The current approach doesn’t explicitly check for silicone piece robustness. In a prototype, you might need to put a lower bound on membrane size or issue a warning for extremely thin features in the mold design.
* **Two-Piece Limitation:** The method is intentionally constrained to produce exactly two mold pieces (each being a silicone insert + shell). It will add internal cuts (membranes) but not create additional free pieces. This simplifies assembly but means some situations that would ideally use three or more pieces are forced into two with complicated internal cuts. If an object truly requires three separate mold pieces (e.g., a shape like a closed loop might traditionally use a three-part mold), this algorithm will instead create a continuous two-part mold with membranes, which could be harder to use. The benefit is fewer pieces, but the trade-off is a more complex silicone shape.
* **Computational Complexity:** The pipeline is computationally heavy:
  + Tetrahedralizing millions of elements can take minutes (TetWild or TetGen often 1–10 minutes for complex models)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf" \l ":~:text=the%20tetrahedralization%20software%20we%20used,geometry%20generation%20procedure%2Crequiring%20the%20computationof" \t "_blank).
  + Computing shortest paths on a graph with ~millions of vertices is also intensive (the paper reports tens of minutes for the escape path computation on a desktop)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf" \l ":~:text=match%20at%20L694%20equipped%20with,Tetgen%29%2058m%205s%2027m" \t "_blank). This step is $O(V \log V + E)$ with $V$ vertices and $E$ edges, which for a fine mesh can be huge. In Python, this would be a bottleneck; a C++ backend or reducing mesh size is advised.
  + The additional membrane detection is essentially checking a criterion for many edges (also millions) – another heavy loop. They optimized it with geometric tests, but it’s still costly.
  + The marching tetrahedra to extract the surface is not too bad (linear in number of tetra), but smoothing involves iterative relaxation, which is usually fast (a few seconds).
  + Persistent homology for each tested orientation can be heavy if done naively. The paper likely precomputed a structure to quickly evaluate different orientations. For an educational prototype, one might restrict the search space (e.g., test 10 candidate directions rather than hundreds).

Memory is also a concern – storing a 17 million tetrahedral mesh and associated data can use several GB of RAM. In a Python environment, careful memory management or using memory-mapped arrays might be necessary if approaching that scale. One way to mitigate is to use a coarser volume mesh for computing membranes (since slight inaccuracies in membrane placement might be tolerable) to save time, as long as the essential topology is captured.

* **Mesh Quality and Numerical Robustness:** The algorithms assume the tetra mesh is a reasonable approximation of the shape. If the mesh is too coarse, a thin protrusion of the object might not be sampled at all by the graph, and the algorithm could miss a needed cut. On the other hand, if the mesh is extremely fine, you get better accuracy at the cost of performance. There is a balancing act or need for adaptive meshing (TetWild inherently is adaptive to some degree by approximating surface within a tolerance). Robustness issues can arise:
  + Dijkstra might have numerical ties; small differences might route paths differently. The threshold near the boundary split line helps avoid random assignment causing jagged parting surfaces[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=correspondtothetwosiliconemoldpieces,the%20convex%20hull%20bounding%20boxdiagonal).
  + Boolean operations on complex meshes (like subtracting the silicone shape from the shell) can fail or produce errors if not using a robust library. Non-manifold edges or coincident surfaces (e.g., the parting surface lying flush with a shell cut) require high precision. This is why a library like CGAL or the method by Zhou et al. is needed[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=details%20of%20the%20parting%20surface,directions%20that%20prevent%20the%20presence).
  + Intersections: when generating the membrane surfaces, one must ensure they properly intersect $M$ and ∂H exactly. The marching algorithm ensures the cut mesh $C$ shares border with $M$ and ∂H where intended. Any gap would mean an open mold or a leak. Checking water-tightness of each silicone half’s surface is important.
  + The use of floats vs. doubles: With many calculations (distance, exp weights), rounding could affect results. It’s advisable to use double precision for distance and path computations to avoid artifacts in membrane placement.
* **Physical Assembly and Usage:** Some assumptions in design for successful casting:
  + The silicone’s flexibility is counted on to remove the mold. This means silicone hardness (e.g., using a relatively soft silicone like shore A ~20-30) is assumed. A very rigid “silicone” would not deform and the method might fail. Conversely, if too soft, the mold might sag. In a prototype, one might simulate or at least consider the silicone material properties when deciding membrane thickness and shape.
  + The alignment features (like Perlin noise bumps) assume the user can assemble the mold accurately. If those features are too fine or the 3D printer resolution is low, they might not fit well, causing misalignment. One might need to adjust them or include registration keys (like slots or pins in the shell).
  + Pouring and vent holes must be properly placed at high points so that air indeed escapes and resin fills all cavities. The algorithm chooses them based on the persistence analysis, but a user should verify no air pockets in a virtual fill simulation or by scrutiny of the pouring direction result (the red-green regions from Fig.12 analysis[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Fig.%2012.%20Left%3A%20maximum,Milnor)).
  + The size of the mold: The outer shell was chosen to be not much larger than the object (15mm clearance)[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf" \l ":~:text=with%20the%20resin%20pouring%20direction%2C,directions%20that%20prevent%20the%20presence" \t "_blank). If the object is large (say 30 cm), the mold is ~30 cm + 3 cm padding. This is fine for many cases, but extremely large objects would result in very large molds (and lots of silicone). So there is a practical size limitation unless using industrial equipment. The authors built objects up to ~30cm I believe[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=with%20the%20resin%20pouring%20direction%2C,directions%20that%20prevent%20the%20presence). Also, small objects with very fine detail might push the limit of printer and silicone detail.
* **Open Challenges:** The authors note that if an object is really complex, sometimes even robust tetra meshing struggles[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=Finally%2C%20a%20practical%20limitation%20is,approach%20significantly%20ex%02tendsthe%20range%20of). They had to use TetGen for one case where TetWild's approximation was not sufficient[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=used%20OpenVDB%20,times%20the%20bounding%20box%20diagonal). Also, transferring the cut design to a different mesh resolution is hard[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=to%20work%20for%20highly%20detailed,between%20models%20of%20different%20resolutions) – meaning if you compute on a coarse mesh and try to refine it, you might introduce slight mismatches. This is an area for future research. Additionally, fully automating the metamold generation and making it foolproof is non-trivial; minor human intervention might be needed if booleans fail.
* **Python Prototype Constraints:** For an educational prototype in Python, the main challenges are performance and precision:
  + **Performance:** Likely you’ll use smaller examples (maybe voxelize a 50x50x50 grid instead of millions of tetrahedra) to demonstrate the concept. The algorithms scale up but not in pure Python.
  + **Precision:** Python’s float is double precision, which is fine, but using libraries (like CGAL or others) might require conversion or careful handling of large/small values. Keep units consistent (e.g., working in mm).
  + **Library Integration:** Installing and interfacing with libraries like TetGen, CGAL, GUDHI, etc., can be a hurdle depending on the environment. One may prepare a conda environment with those pre-installed for ease.

In conclusion, *Volume-Aware Design of Composite Molds* makes some assumptions (closed input, two-piece mold) and navigates various geometric challenges through a clever combination of algorithms. While implementing it fully in Python is ambitious, breaking it down into modules as above can make it tractable. By understanding these constraints and making conservative choices (like slightly over-sizing membranes for strength, verifying each step’s output), a developer can create a working prototype that illustrates the power of volumetric mold design for extremely complex objects. This approach significantly extends the range of models that can be cast[opus.lib.uts.edu.au](https://opus.lib.uts.edu.au/bitstream/10453/137699/4/composite_compressed.pdf#:~:text=robustly%20transfer%20the%20generated%20geometry%2C,Also), bringing previously “impossible” designs into the realm of manufacturability with accessible tools.

**TO TRY:**

1. **offset\_weight:** try the weighted distance (distance from convex hull point to offset surface) instead of smallest distance to the convex hull

2. **region\_growing:** region growing from the faces that the draw direction intersects (idk how much this would help, as convex hull is being split really well)

3. **dist\_dir:** Split the mesh on 2 criteria: alignment of face normal with draw direction, and smallest distance to the convex hull (similar to point 1, but the quantities being weighed together are different)

Create diff branches for points 1, 2, 3