Delaunay Methods for Approximating Geometric Domains

DISTRIBUTION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By
Joshua Aaron Levine, M.S.
Graduate Program in Computer Science and Engineering

The Ohio State University
2009

Dissertation Committee:
Dr. Tamal K. Dey, Advisor
Dr. Yusu Wang
Dr. Rephael Wenger
© Copyright by
Joshua Aaron Levine
2009
ABSTRACT

The Delaunay triangulation is used extensively for representing geometric domains. Since its definition in 1934, researchers have applied it and its dual structure, the Voronoi diagram, to algorithms for surface reconstruction, mesh generation, simplification, and feature computation. Its universality can be explained by the mathematical properties it naturally optimizes, the provable guarantees it lends to algorithm analysis, and the design simplicity it provides for algorithms.

We consider the use of the Delaunay triangulation to approximate three different domains. We begin by developing a technique for simplifying vector field datasets using Delaunay triangulations. Piecewise-linear interpolation over Delaunay triangulations gives good approximations for scalar fields, motivating our approach on vector-valued data. We remove vertices from the Delaunay triangulation using a local error metric which is biased to preserve critical points of the field and to prevent topological changes during simplification. Experimental results show the effectiveness of this technique on both two and three dimensional datasets.

We next present an algorithm, \texttt{Dellso}, for building Delaunay meshes to approximate smooth surfaces defined by the isosurface of a volume datasets. \texttt{Dellso} employs a two stage algorithm which discards the need to maintain the full 3D Delaunay triangulation in the second stage. Implementation results have shown that by using this optimization we can obtain a 2-3 times speedup over its one stage counterpart.
The resulting meshes are different from those produced by more common isosurfacing techniques (e.g. Marching Cubes) in that they are well graded and their topology is provably correct.

The third domain we investigate is piecewise smooth complexes (PSCs). We have designed DelPSC, an algorithm to build Delaunay meshes that approximate PSCs as well as the volumes contained within them. DelPSC was designed to be easily implementable, removing the need for many of the expensive computations that previously made Delaunay meshing for PSCs impractical. Its meshing strategy employs a protection scheme for sharp features that covers them with protecting balls whose radius mimics the feature size. Unfortunately, feature size computations can be costly, so we propose a novel protection method which first places balls along each curve and then iteratively shrinks them until they maintain a set of protection properties. We guarantee that with this set of protection properties our Delaunay refinement terminates and that by reducing a single scale parameter, the correct topology is achieved as well.

The approach used in DelPSC allows for meshing a wide variety of objects such as non-smooth CAD parts and non-manifold objects. We found with DelIso that Delaunay meshing for smooth surfaces could be adopted to mesh isosurfaces of volume datasets. Similarly, the strategy for Delaunay meshing of PSCs can be used to build meshes of the interface surfaces that multi-label volume data sets define. Our final contribution discusses the application of DelPSC to this form of data, commonly produced by segmenting MRI or CT datasets. We show the effectiveness of this technique on data from a variety of medical fields and discuss the its ability to control the quality and size of the output meshes.
ACKNOWLEDGMENTS

I would like to sincerely thank my advisor, Tamal K. Dey, for guiding me through this process. My research could not have been completed without his knowledge and experience. He made contributions to all sections of this dissertation, and therefore many of the words herein echo his own thoughts and influence. As an advisor, Tamal was firm enough to keep me working the right direction, but kind enough to listen if I disagreed. His patience with me was tremendous, and it is the one quality of his I most hope to be able to emulate.

In addition, my committee, collaborators, and coauthors deserve praise. In particular, it was a pleasure to work with Rephael Wenger, Yusu Wang, Siu-Wing Cheng, Firdaus Janoos, and Oleksiy Busaryev. Each one of them had an impact on the way I think, and consequently this work.

My family and friends have been here continuously to support me. My mother and father, Priscilla and Marc, as well as my sisters, Arielle and Elana, have encouraged me the entire time. My office mates Tathagata, Jian, Kuiyu, Issam, Brian, and Jim were always ready if I need someone to bounce an idea off of or just give me a distraction.

I also want to acknowledge financial support from NSF grants CCF-0430735 and CCF-0635008 as well as the Department of Computer Science and Engineering. These sources completely funded my Ph.D. research.
In any form of experimental work, it is often a struggle to find datasets to run experiments on. From Chapter 3, I would like to thank Han-Wei Shen for providing us with the ocean wind dataset as well as useful insights. I am also indebted to Nikos Platis and Theoharis Theoharis for providing us access to their datasets and OpenDX simplification module.

Many of the datasets from the experiments in Chapter 4 were freely obtained at sites such as http://www.volvis.org. I would also like to thank Jason Bryan for assistance using VolSuite to generate the Marching Cubes isosurfaces.

In general, many of the datasets used for Chapters 5 and 6 were found on online model databases such as the ones found at http://shapes.aim-at-shape.net/ and http://www-roc.inria.fr/gamma/gamma.php.

I finally wish to thank Thomas Kerwin, Miriah Meyer, and Ross Whitaker for aiding in obtaining datasets used in Chapter 7. The temporal bone and dog skull dataset are provided courtesy of Don Stredney at the Ohio Supercomputer Center. The tomato and orange datasets were produced at Lawrence Berkeley Laboratory by Bill Johnston and Wing Nip of the Information and Computing Sciences Division. The torso dataset is courtesy of John Triedman and Matthew Jolley at Boston Children’s Hospital.
VITA

January 27, 1981 .................................. Born: New York, NY, USA

May 2003 ............................................. B.S.E. Comp. Engineering,
         B.S. Mathematics,
         Case Western Reserve University,
         Cleveland, OH, USA

January 2004 ...................................... M.S. Computer Science,
         Case Western Reserve University,
         Cleveland, OH, USA

Autumn 2004—Summer 2005 .................. University Fellow,
         The Ohio State University

Autumn 2005—Summer 2008 .................. Graduate Teaching Associate,
         The Ohio State University

Winter 2006—Present ......................... Graduate Research Associate,
         The Ohio State University

PUBLICATIONS

Research Publications


**FIELDS OF STUDY**

Major Field: Computer Science and Engineering

Studies in Delaunay Mesh Generation: Prof. Tamal K. Dey
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Abstract</th>
<th>ii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Vita</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiii</td>
</tr>
</tbody>
</table>

Chapters:

1. Introduction ............................................. 1
   1.1 Meshes .................................................. 2
   1.2 Vector Field Approximation ......................... 3
       1.2.1 Simplification Techniques ..................... 5
   1.3 Isosurface Meshing ..................................... 6
   1.4 Mesh Generation for More Generalized Domains ...... 7
       1.4.1 Practical Meshing for PSCs .................... 9
   1.5 Meshing Multi-Label Data ............................ 10
       1.5.1 Past Approaches to Interface Surface Generation 11

2. Background .................................................. 14
   2.1 Geometric Domains ..................................... 14
       2.1.1 Implicit Domains ................................ 14
       2.1.2 Piecewise Smooth Complexes ................... 16
   2.2 Delaunay Triangulations and Voronoi Diagrams ...... 18
       2.2.1 Restricted Delaunay Triangulations ............ 20
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.2</td>
<td>Delaunay Properties</td>
<td>22</td>
</tr>
<tr>
<td>2.3</td>
<td>Delaunay Refinement for Mesh Generation</td>
<td>24</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Preserving Geometry for Polyhedral Meshing</td>
<td>25</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Preserving Topology for Smooth Surface Meshing</td>
<td>26</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Meshing Piecewise-Smooth Domains</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>A Simplification Algorithm for Vector Fields</td>
<td>28</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Simplification Algorithm</td>
<td>30</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Error Metric</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>Feature Preservation</td>
<td>34</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Bounded Error</td>
<td>34</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Topological Preservation</td>
<td>36</td>
</tr>
<tr>
<td>3.3</td>
<td>Experimental Results</td>
<td>37</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Two Dimensions</td>
<td>38</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Three Dimensions</td>
<td>43</td>
</tr>
<tr>
<td>3.4</td>
<td>Discussion</td>
<td>49</td>
</tr>
<tr>
<td>4.1</td>
<td>Isosurface Mesh Generation</td>
<td>52</td>
</tr>
<tr>
<td>4.2</td>
<td>Delaunay vs. Non-Delaunay Isosurfaces</td>
<td>53</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Avoiding the Full Delaunay Triangulation</td>
<td>55</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Intersection Search</td>
<td>58</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Timing Comparisons</td>
<td>59</td>
</tr>
<tr>
<td>4.3</td>
<td>Algorithm Details</td>
<td>60</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Isosurface Recovery (Stage 1)</td>
<td>61</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Isosurface Refinement (Stage 2)</td>
<td>64</td>
</tr>
<tr>
<td>4.4</td>
<td>Intersection Search</td>
<td>69</td>
</tr>
<tr>
<td>4.4.1</td>
<td>kd-tree Searches</td>
<td>69</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Voxel-Ray Intersection</td>
<td>71</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Improving the Speed of Intersection Searches</td>
<td>72</td>
</tr>
<tr>
<td>4.5</td>
<td>Boundary Issues</td>
<td>73</td>
</tr>
<tr>
<td>4.6</td>
<td>Discussion</td>
<td>74</td>
</tr>
<tr>
<td>5.1</td>
<td>A Practical Meshing Algorithm for PSCs</td>
<td>76</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Algorithm Overview</td>
<td>79</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Sharp Feature Preservation</td>
<td>80</td>
</tr>
<tr>
<td>5.2</td>
<td>Meshing PSCs</td>
<td>84</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Field error for simplification of the gravity field</td>
</tr>
<tr>
<td>3.2</td>
<td>Field errors for gravity field simplification, sampling in balls of radius 1.0 around all critical points</td>
</tr>
<tr>
<td>3.3</td>
<td>Field error for simplification of the Lorenz attractor</td>
</tr>
<tr>
<td>4.1</td>
<td>Volume datasets with Delaunay refinement time comparisons: using a 3D Delaunay triangulation for the entire refinement (3D) vs. our algorithm (Dellso). Times are in seconds</td>
</tr>
<tr>
<td>4.2</td>
<td>Average insertion times $t_{RC}$ and $t_{RF}$ (in seconds) for inserting $n_{RC}$ and $n_{RF}$ points in the Recover() ($RC$) and Refine() ($RF$) stages, respectively</td>
</tr>
<tr>
<td>4.3</td>
<td>Comparison of the number of trilinear computations per intersection search for the Recover() ($RC$) and Refine() ($RF$) stages. Shown are the average number of computations per intersection search ($CPI_{RC}$ and $CPI_{RF}$) and the $CPI$s during only those searches which found intersection points ($CPI_{RC}^<em>$ and $CPI_{RF}^</em>$)</td>
</tr>
<tr>
<td>5.1</td>
<td>Input datasets. Output mesh sizes and time to mesh (not including protection times) are shown. Times are in seconds</td>
</tr>
<tr>
<td>6.1</td>
<td>Protection time comparison for PSC datasets from Chapter 5. Input size is measured in number of triangles and protection times are in seconds. Fertility and Metaball are not shown because they have no curves to protect</td>
</tr>
<tr>
<td>6.2</td>
<td>Protection and meshing times for PSC datasets. Times are in seconds</td>
</tr>
</tbody>
</table>
7.1 Dataset summary. For each dataset the dimensions, source, and number of labels are given. Note there is an additional “background” label present in all datasets. 

7.2 Meshing summary. For each dataset the number of output vertices and the number of curves protected are listed. Output times (in minutes:seconds) are broken down by protection time as well and time to generate the mesh. File I/O times are excluded.

7.3 Quality and shape statistics. The average and maximum aspect ratio (radius/shortest edge length) are given for each of the datasets. Also shown are the minimum dihedral angle (in degrees) between all pairs of faces circling each non-manifold edge. This measure shows how sharply the interface surfaces are meeting at curve elements.
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Left: A clover-shaped 1-dimensional PSC. Middle: There are five elements in $D_1$, highlighted in red, orange, yellow, green, and blue. Right: And two elements in $D_0$, highlighted in cyan and magenta. Note there is no limitation on the number of elements which share adjacent boundaries.</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>Left: Voronoi diagram for the given point set in $\mathbb{R}^2$ shown in red. Middle: The Delaunay triangulation shown in blue. Right: Empty circumball property for Delaunay simplices; the magenta triangle has an empty circumcircle shown in green. The magenta edge has infinitely many empty balls with centers on the dual Voronoi edge.</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>Restricted Delaunay triangulations. Left: a sample of our clover PSC with its corresponding Voronoi diagram (in blue). Right: Within the dual Delaunay triangulation (dashed and red segments), only a subset of its edges are restricted to the clover (red segments). When a sample is chosen carefully, restricted Delaunay triangulations can approximate this shape.</td>
<td>22</td>
</tr>
<tr>
<td>2.4</td>
<td>Left: $\text{Del} S</td>
<td>_{\sigma}$. Right: $\text{Skl}^2 S</td>
</tr>
<tr>
<td>2.5</td>
<td>Delaunay Refinement Algorithm.</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>Simplification of a 2D vector field to error threshold $\varepsilon = 0.15$. The surfaces for the two component functions are preserved as well as their zero-level sets. By juxtaposing the simplified vector field (red) on top of the original (black), we see little change from the original.</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Calculation of $\varphi$ when deleting a vertex from the mesh.</td>
<td>32</td>
</tr>
</tbody>
</table>
3.3 Vector field simplification algorithm, via vertex deletion............ 33

3.4 Left: The tip of $\dot{v}'$ is in a ball of radius $\varepsilon\|\dot{v}\|$. Center: A maximal magnitude change. Right: A maximal angle change............. 35

3.5 Vector field with separatrices........................................ 39

3.6 A topological change of the 2D vector field. Left to right: Separatrices of the simplification having 540, 539, and 538 vertices............. 40

3.7 Top: Ocean wind data with nullclines. Bottom: Ocean wind data after simplifying to $\varepsilon = 0.10$........................................ 42

3.8 A topological change of the ocean wind dataset. Left to right: Nullclines and critical points of the simplification at $\varepsilon = 0.00$, 0.01, and 0.10. Critical points are highlighted with arrows...................... 43

3.9 Gravity dataset streamlines and nullclines. Left: First viewpoint. Right: Rotated to the back side of the field......................... 44

3.10 Simplified gravity field streamlines and nullclines. Left: Our Delaunay technique at $\varepsilon = 0.05$. Right: Technique of Platis and Theoharis run to a corresponding level of 6% tetrahedra............................... 46

3.11 Lorenz attractor streamlines and its three nullclines................. 48

3.12 Top row: Simplified Lorenz attractor streamlines. Left: Our Delaunay technique at $\varepsilon = 0.025$. Right: Technique of Platis and Theoharis run to a corresponding level of 2% of the original tetrahedra. The bottom row shows the nullclines at the same level of simplification............. 50

4.1 Atom (left) and Fuel (right) isosurfaces. In both images, the left surfaces are produced by Marching Cubes and the right are the output of DelIso.......................... 54

4.2 After Stage 1, we can insert points in Del $S|_{\Sigma}$ without Del $S$. Left: We refine a triangle $\sigma$ (shaded) by inserting the center of $B_{\sigma}$. Center: This point encroaches a topological disk $E$. Right: We replace $E$ with $D$ by connecting the boundary of $E$ to the point............................... 57

4.3 DelIso algorithm........................................................... 61
4.4 Recover() algorithm. .................................................. 62
4.5 Engine (left) and Tooth (right) isosurfaces. The Tooth isosurface is drawn with a clipping plane to see the inside component of the surface. In both examples the left surface is the final output of DelIso and the right is the out after Recover(). .................................................. 64
4.6 Refine() algorithm. .................................................. 65
4.7 DelIso output isosurfaces. Upper left: Isosurface for Baby1 (back) and mesh of isosurface for Baby2 (front). Upper right: External components of Chest (back), by clipping the front we see the mesh for the lungs, trachea, and bronchi (front). Lower left: Aneurism dataset, by zooming in we see the mesh adaptively samples the tubular features. Lower right: Isosurface of BluntFin and close up view of mesh on the sharp feature. .................................................. 67
4.8 Intersect() algorithm. .................................................. 69
4.9 A kd-tree facilitates ray-isosurface intersections. Left: Isosurface Σ, ray r, and the kd-tree. Center: Using only the tree and r we identify a set of candidate voxels (in green). Right: Using κ_{min} and κ_{max} we can prune the cyan cells. .................................................. 70
5.1 Meshed PSCs, Metaball (Smooth), Part (Manifold PSC), and Wedge (Non-manifold with small angles). .................................................. 77
5.2 Sharp features on Anchor are preserved in both surface (middle) and volume (right) meshing and topology is recovered. .................................................. 79
5.3 Disk condition. Left: Triangles incident to point p ∈ σ and restricted to σ do not form a disk since they form two disks pinched at p violating condition D1. Middle: The point p ∈ σ has a topological disk but some of its vertices (lightly shaded) belong to τ violating condition D4. Right: Points p and q satisfy the disk condition. Point p, an interior point in σ, lies in the interior of its disk in σ. Point q, a boundary point, has three disks for each of the three 2-faces. .................................................. 83
5.4 DelPSC1 algorithm. .................................................. 84
5.5 Pseudocode for Mesh2Complex() and Mesh3Complex().

5.6 Protection in action on Casting. We placed weighted vertices on all elements of $\mathcal{D}_1$ which protect these elements when meshed.

5.7 The two dashed circles denote $B(x_{k-1}, \frac{6}{g} r_{k-1})$ and $B(x_0, r_{x_0})$. The bold circle denotes $B_{x_k}$.

5.8 Left: Within ball $B$, $V_p$ intersects $\sigma$ and $\tau$ both of which intersect some edge of $V_p$. This is not possible according to Lemma 5.3.2. Middle: Also not possible since there is another component of $\sigma$ within $B \cap V_p$ other than $\bar{\sigma}$. Right: Within $B$, $\sigma$ intersects $V_p$ in a topological disk. It is possible that there is a different component ($\tau$) which does not intersect any Voronoi edge and hence does not contribute any dual restricted triangle incident to $p$.

5.9 Left: Adjacent points on curves in $\mathcal{D}_1$ are joined by restricted edges. Middle: A surface patch is meshed with a manifold though topology is not fully recovered. Right: Topology is fully recovered.

5.10 $F$ is a Voronoi facet. Left: $F$ intersects a 2-face in a closed topological interval (1-ball) which is $b_F$. Here $b_F$ intersects $\text{Bd} F$ at two points, a 0-sphere. Right: $F$ intersects the 1-face in a single point which is $b_F$, and for $1 \leq i \leq 3$, $F \cap \sigma_i$ are closed topological 1-balls incident to $b_F$. Here $b_F \cap \text{Bd} F = \emptyset$, a $-1$-sphere.

5.11 Output on Fertility at different levels of $\lambda$. As $\lambda$ is reduced, eventually the correct topology is achieved.

5.12 Serrated: surface (middle) and volume (right) mesh.

5.13 Hand: manifold with boundary (two levels of refinement).

5.14 Four PSCs: Pin-Head, Guide, Saturn, and Swirl. The last two are non-manifold. In each figure we show the input (left), surface mesh (middle) and volume mesh (right, where present).
6.1 Covering 1-faces. Left: A 1-face $\sigma$ between $u$ and $v$ is being protected. The ball $b$ shown with solid boundary has $\text{seg}_\sigma(b)$ as the curve segment between $x_1$ and $x_2$. The balls satisfy C1 and C2 but intersect arbitrarily. Right: Balls are refined and they start satisfying separation properties C1-C3.

6.2 Curve segment between $x$ and $y$ is being covered. Aiding balls are shown with solid boundaries. Notice how the centers of $b_1$ and $b_2$ are placed with the aiding balls. One the left, the final ball is computed by enlarging the aiding ball. On the right, we show the other case where the final ball $b_{i+1}$ is enlarged.

6.3 Pseudocode for Separate() and RefineBall().

6.4 Pseudocode for Protect().

6.5 Pseudocode for DelPSC2.

6.6 Protection: Top Left: A 2d example of curve protection. Top Right and Bottom Left: 2 different 3d examples of curve complexes where we have run Protect(). Bottom Right: The final set of protecting balls returned by DelPSC2 on the Wedge model.

6.7 DelPSC output meshes. Left to right, top to bottom: Part, Guide, Wedge, 9 Holes, Cog, Horn, Arm, Octo, Lock, Swirl, Pump, and Plate models.

7.1 MNI Brain Atlas, illustrating structural atlas of the brain with seven anatomical regions shown. We generate consistent meshes for the surfaces where these regions interface. The exploded view on the right shows the interior surfaces.

7.2 Refinement algorithm on the Spheres dataset. Top Left: a set of curves are extracted where three or more labeled regions intersect. Top Right: these curves are protected with an initial set of balls that may later get refined by the algorithm. Bottom Left: these balls are converted to weighted points and refinement fills in samples for the interface surfaces. Bottom Right: The output mesh captures all interfaces.
7.3 Soft segmentation of the brain. Grey matter is draw transparently and white matter is drawn in green. Red triangles show where the white matter protrudes through the grey. .......................... 135

7.4 CT datasets. We grouped labels of the Bulldog Skull (left) and Temporal Bone (right) datasets to form 4 and 5 labels, respectively. .... 136

7.5 Torso dataset with close ups of protected curves. Labels are four the skin (transparent), skeleton (yellow), lungs (cyan), and heart (orange). 137

7.6 Volume meshes for the sphere and brain atlas. Our algorithm generates Delaunay tetrahedral meshes which can be used for finite element simulations. .......................... 138

7.7 Orange dataset. From left to right, top to bottom we show the labels for the outer skin, membrane, pulp, and seeds. ...................... 139

7.8 Tomato dataset. From left to right, top to bottom we show the labels for the endocarp, locule, placenta, and core. ...................... 140
CHAPTER 1

INTRODUCTION

Modeling and simulation software is being used with increasing frequency to complement traditional engineering approaches. When researchers leverage computers for these types of analysis, they often require robust, efficient algorithms to first compute representations of geometric domains. This dissertation focuses primarily on the discretization techniques for these domains. Because we want accurate and correct results, we draw upon the knowledge of geometers and topologists to ensure our algorithms produce representations that meet any desired requirements. However, we also want our results to be relevant to real world problems, so we must seek a balance between what the best choices are in theory and what will address the practical needs of a problem.

The Delaunay triangulation is a shared component for all of algorithms within this dissertation, and in many of the cases we argue it allows us to achieve this balance. It has both the benefit of many well known and efficient algorithms to compute it as well as a number of useful geometric properties that lend to proving facts regarding it. In particular, we discuss its use in approximating three different domains with algorithms for vector field simplification [50], isosurface mesh generation [47], and meshing of piecewise smooth complexes [27, 28, 46, 48].
1.1 Meshes

All of the domains we discuss will be represented by meshes. Meshes give us a fundamental datastructure to represent a domain discretely. By design this allows us to manipulate meshes using computers, but the tradeoff is that it may be the case that they only approximate portions of the domain which cannot be finitely represented.

We focus primarily on using meshes to approximate \textit{geometric} domains in three dimensions, often called shapes. Meshes are commonly used for this task: computer graphics professionals draw, light, texture, and morph 3D shapes using meshes; engineers perform static and dynamic analysis using finite element methods on meshes to test and verify the fruits of their designs; cartographers model terrain using a set of height measures connected by a mesh; and biochemists deduce molecular shape and interactions using mesh representations.

With each application, researchers have designed different algorithms to generate meshes which are appropriate to the task. Often though, a single mesh is designed by hand using an enormous amount of resources to ensure the mesh models its domain correctly. One of our focuses is on automatic mesh generation techniques: we would like to build meshes which minimize the need for a domain expert to be present at every stage.

Informally, a \textit{mesh} consists of two parts: (1) a set of discrete \textit{samples} \( S \subseteq D \) of some geometric domain \( D \) and (2) a collection of connecting elements or \textit{cells} which tessellate the unsampled regions of \( D \) with neither overlaps nor gaps. For example, the set of integer pairs in the Cartesian plane form a sample of \( \mathbb{R}^2 \), and each square that can be formed by connecting the four samples \((i, j), (i + 1, j), (i + 1, j + 1), \) and \((i, j + 1)\) satisfy our loose requirements for being the cells of a mesh. There is no
requirement that cells must be squares or even uniform in appearance; we could add the diagonal to each square or use only the prime integers for coordinates.

Any algorithm to build a mesh must consequently have both a strategy for generating domain samples and a mechanism for providing connectivity between the samples. Isolated, the former is classically known as a sampling problem, i.e. what is the best set of samples which capture the domain, and the latter is a reconstruction problem, given a fixed set of samples, what is the best approximating set of cells for the domain we can construct. A mesh generation algorithm benefits from having the flexibility to adjust either piece, as we are creating our mesh we will see that we often have the option to either sample further or to change connectivity.

Since a mesh is generally used for later processing, a number of different criteria are generally imposed on the output of a meshing algorithm. They can be as simple as number of samples used, or more complicated rules such as the level of smoothness of the approximation or topological correctness. In general, proving that an output mesh satisfies these additional criteria involves significant extra machinery. The solutions we discuss herein all seek to enforce some set of requirements on the output, generally through localized conditions such as the shape and size of the cells.

1.2 Vector Field Approximation

We first look at a problem where the mesh is known, but we would like to perform some processing on it to improve its storage requirements. A variety of dynamical systems such as fluid flow, weather, and surface gradients can be modeled by a domain we call a vector field. For example, some applications require designing vector fields to perform texture synthesis and non-photorealistic rendering [59, 130]. A common
technique for managing vector data is an input point sample, where a vector value is stored at each sample. A triangulated mesh connects the sample, providing the means to interpolate the vector values in between samples.

Given this mesh, vector field values inside grid cells may then be interpolated from the data values at the cell vertices. Often the dataset is very large which leads to a major difficulty in analysis. Naturally, it is desirable to simplify vector field data while preserving its important features. Associated concerns are that of determining a good triangulation in terms of interpolation error and a simplification strategy that keeps the error low.

A popular candidate for triangulating scalar field data is the Delaunay triangulation because it possesses several optimization properties that help in good approximations. Recent sampling theory shows that in surface approximations both point-wise and normal-wise approximation errors depend on the circumradius of the triangles [45]. Since a Delaunay triangulation keeps the circumradii of triangles small, it often provides a good approximation. While there are some results showing that the best triangulation is often not Delaunay [55, 99], in general the Delaunay triangulation works well in practice and has been proven to give the optimal interpolation for certain classes (i.e. convex) of functions [25].

We address vector field approximations with Delaunay meshes in Chapter 3. There are few other works which discuss how a triangulation affects approximation of vector fields. One relevant work is the research of Scheuermann and Hagen [101]. They indicate that the topological structure of a vector field is indeed dependent on the choice of triangulation, as flipping a single edge could change the critical points and thus the topology of the vector field. Furthermore, they give an algorithm for computing
a data-dependent triangulation that maintains critical points. Recently, Mebarki et al. [78] used Delaunay triangulations to draw streamlines effectively for vector fields in two dimensions.

1.2.1 Simplification Techniques

We study Delaunay meshes for vector fields because we would ultimately like to represent them in simpler forms. One class of vector field simplification techniques are those which make visualizing the vector field simpler. A prominent method is to use the vector field’s topology [62, 67] to create a representative structure capturing the vector field. This topological visualization may still be too complex. Many have simplified it by removing [42], perturbing [118], or combining [117, 122] critical points or using saddle connectors (in three dimensions) [115]. Vector field clustering algorithms [54, 65, 112] are also often used for simplification. Here the main goal is to create some sort of hierarchy where representative vectors for the different clusters may be chosen. Also, decomposition of the vector field into simpler component fields (a curl-free and divergence-free component) has been used to improve visualizations [116].

Another approach is domain compression. Instead of creating a visualization that has less data, we would like to simplify the vector field by simplifying the underlying mesh. A coarser mesh reduces storage requirements and gives level of detail control to the user. Moreover, a sparse mesh has potential benefits in terms of streamline integration. Optimization of this integration [53, 119] makes use of barycentric coordinates to propagate the position of the streamline. Thus, reducing the number of cells intersected by a streamline reduces its construction cost.
When simplifying the triangular mesh upon which the vector field will be approximated, it is important to maintain both the topological structure as well a geometric proximity to the original vector field. An important work in this area is that of Theisel et al. [113, 114] who simplify two dimensional vector fields while preserving their topology. They give exact criteria for performing an edge collapse in the mesh without changing the topology of the field, but it is unclear how to extend this criteria to higher dimensions. Conversely, Platis and Theoharis [93] simplify tetrahedral meshes (not necessarily Delaunay) of a vector field using edge collapses guided by a collection of error metrics, but do not explicitly address topological concerns.

1.3 Isosurface Meshing

We next look at a problem commonly found in the medical and visualization communities. In these domains, it is often the case that surfaces are represented in both an explicit form and an implicit form. For computational purposes, the explicit form is often represented by a piecewise linear surface (a polygonal mesh), while the implicit form is usually represented by a (scalar) volume dataset, a collection of points in \( \mathbb{R}^3 \) each of which has the value of some scalar field \( f \) associated with it.

Moving from one form to another is often necessary: the implicit form is compact and appropriate for certain applications such as blending, but the explicit form allows other computations such as static analysis and finite element methods to be performed efficiently. The transformation from the implicit form to the explicit one is popularly known as the isosurface problem.

Many significant research results have been produced for meshing of isosurfaces. One of the earliest known approaches to isosurface polygonalization is credited to
Wyvill et al. [127] and was followed by the Marching Cubes algorithm of Lorensen and Cline [76]. More recently though, there has been a growing concern with designing algorithms that satisfy additional constraints regarding geometric closeness and topological equivalence between the output triangulation and the isosurface. Such a mesh is favored in many applications since these qualities lead to reducing discretization error and mesh size. Variants of the original Marching Cubes exist to solve the topological concern [14, 34, 85, 120], and the already efficient algorithm can be improved using octrees [125]. Geometric quality has also been addressed using advancing front techniques [103], particle systems [79], and a variant of the Marching Cubes lookup table [95].

Under some mild assumptions, the isosurface problem fits into the general category of surface mesh generation. In Chapter 4 we discuss an algorithm to generate high quality meshes for isosurfaces [47]. Our approach uses an algorithmic paradigm called Delaunay refinement. Numerous algorithms for surface meshing exist [16, 17, 31, 111] some [17, 31] of which use a Delaunay-based approach to approximate the surface in question. Our algorithm is based on the technique proposed by Cheng et al. [31] for smooth surface meshing which was later adapted in the SurfRemesh algorithm by Dey et al. [51] for meshing polygonal surfaces. This algorithm provides guarantees for the topology and geometry of the output mesh with respect to the input surface.

1.4 Mesh Generation for More Generalized Domains

Our final chapters are devoted to designing algorithms for shapes which have been considered more challenging to mesh. In general, a mesh generation algorithm strives to make guarantees about automatic termination and input shape approximation.
However, these guarantees only hold under particular assumptions on the input. For example, they may require that the input be a collection of polygons with an angle condition where two polygons meet. Or they may require a topological condition such as the input is a smooth manifold without boundary.

Unfortunately, many input shapes fail to meet these criteria. For example, the isosurface meshing algorithms described above typically assume the isosurface is a compact, smooth manifold. But under trilinear interpolation, one of the commonly used interpolation schemes for isosurfaces, the isosurface can be piecewise smooth, and not always a surface. Many algorithms tolerate small regions of non-smoothness or have extra steps they can use to fix non-manifold regions, but it is often necessary to model the shape in a more general way.

One such higher-order shape model is a *piecewise smooth complex*, or PSC, a representation that we will use to handle some of these issues in a clean manner. Here the input shape is modeled as a collection of smooth patches which may meet at arbitrarily sharp angles or in non-manifold configurations. The shape is assumed to respect the properties of a usual complex; the interiors or all patches are disjoint, and any two patches that intersect do so in collection of lower dimensional pieces. This class includes polyhedral domains, smooth and piecewise smooth surfaces, and even non-manifolds.

For PSCs, we will also look at meshing not only the complex of patches, curves, and vertices, but also the interior volume that this shape bounds. We will use the terms *surface mesh* for the former and *volume mesh* for the later. The cells in our volume meshes will be a collection of tetrahedra as opposed to a collection of triangles that are used for a surface mesh. We consider volume meshes because they are a key
component for analysis with finite element methods. Luckily, our approach naturally builds a volume mesh because we treat the domain as a shape embedded in three dimensional space. Enforcing quality criteria on the shape of tetrahedra is the main challenge that remains.

1.4.1 Practical Meshing for PSCs

It is natural to first try to adopt algorithms that work for smooth surfaces and see how well they will tolerate non-smooth shapes. Boissonnat and Oudot [18] gave a provable algorithm for a class of non-smooth surfaces called Lipschitz-surfaces. They showed that their algorithm for smooth surface meshing [17] extends to this class only if input angles are sufficiently large. Unfortunately, this approach failed to admit small input angles which limited the input class. Rineau and Yvinec [98] implemented an algorithm for meshing volumes bounded by piecewise smooth surfaces; their approach also suffers from an angle constraint. Recently, Cheng, Dey, and Ramos [29] proposed an approach that completely removed any constraint on input angles. As a result their algorithm could accommodate PSCs.

The algorithm of Cheng et al. [29] uses the idea of protecting non-smooth curves and vertices in the input complex with balls. This idea already gave good results for meshing piecewise linear shapes [33, 38, 83]. A novelty introduced by Cheng et al. is that they turn the balls into weighted points and then carry out the mesh refinement in the weighted Delaunay triangulation [11, 56]. Notwithstanding its theoretical success, practical relevance of this algorithm remains questionable. It employs some expensive numerical computations that are hard to implement. In Chapter 5 we introduce an algorithm [27] for meshing PSCs in a more practical way. We distill the
theoretical algorithm of Cheng et al. to produce a more practical one that primarily uses combinatorial primitives. The result is one of the main contributions of this dissertation: a provable, implementable algorithm for mesh generation of piecewise smooth complexes.

We have improved further [48] upon the practicality of the algorithm. While many of the numerically unstable primitives were removed, the mechanism for building the set of balls which cover the non-smooth features of a PSC still requires some expensive feature size computations. In Chapter 6 we discuss a novel way of computing these balls where the non-smoothness of shape drives the ball sizes. This approach effectively eliminates another major bottleneck and creates an algorithm which can be implemented more robustly. More importantly, the algorithm has the same set of provable guarantees which certify the output mesh approximates the input domain.

1.5 Meshing Multi-Label Data

Similarly to how we can adapt smooth surface meshing algorithms to produce meshes of isosurfaces, we can use our mesh generation algorithm for PSCs to mesh the interface surfaces of multi-label data [46]. The medical fields use microscopy, PET-CT, and MRI to produce volumetric images scanned from anatomical objects; primarily for the purposes of aiding in the diagnosis and treatment by a skilled professional. These devices capture data in the form of scalar volume datasets. However, these 3D images are often scanned from real-life entities. Thus, to further understand the object, a classification algorithm is then applied to label each grid point using the scalar data values and hints of its known structure. These labels subdivide the volume into regions representing structural elements of the scanned object such as different
materials (e.g. bone, muscle, organs, etc.) or different components (e.g. spine, jaw bones, uncus, etc.).

Standard labeling approaches use “hard” segmentations that produce a binary mask for each label. However, there is usually a substantial overlap between these segments, resulting in multiple labels at voxels in the overlap region. Alternatively, in “soft” segmentation methods, a set of real-valued probabilities for anatomical features is associated with each voxel of the image. The result is a fuzzy boundary between the segments [123]. For a more in-depth review of medical image segmentation, the reader is referred to [71].

Given a labeled volume dataset, there is a need for the reconstruction and visualization of the surfaces where different labeled regions meet. We call such surfaces interface surfaces, as they lie precisely where two or more labeled regions intersect. Interface surfaces fall into the general category of PSCs, and they are often a key component in the structure of scanned data. For example, in images obtained for medical applications it is essential to extract and visualize the surfaces of the different anatomical features for diagnosis, surgery simulation and planning, and therapy evaluation [61]. Similarly, multi-label data is generated in domains such as finite element simulations and multi-fluid hydrodynamic calculations [97]. Chapter 7 discusses the application of our PSC meshing algorithm to mesh generation for interface surfaces.

1.5.1 Past Approaches to Interface Surface Generation

For dealing with multi-labeled data on regular rectilinear grids, the concept of Marching Cubes [76, 127] has been extended to automatically create and apply lookup
tables for interface surfaces between two or more materials [12, 13, 126, 128]. These methods can suffer from problems of ambiguities in the labeling, topological defects, and an algebraic increase in the number of cases with respect to the number of materials. The result is that they can create inconsistent meshes where the boundaries of the triangle meshes do not consistently fit together. Solutions to these problems have been variously proposed, including repairing ambiguous voxels before extracting the meshes [15], domain subdivision [96], assigning probabilities at the vertices and interpolating them to extract topologically correct surface elements [66]. Algorithms based on marching tetrahedra [84] are also commonly used [52, 82]. Dual contouring methods that preserve sharp edges [70] have also been proposed for tetrahedral and hexahedral meshes [131].

In general though, grid based methods do not guarantee topological correctness, can generate poor quality triangles, and the resolution of the output mesh depends upon the grid resolution itself. The resulting meshes often require additional post-processing to make them suitable for later applications.

Alternate approaches for extracting interface surfaces use methods from computational solid geometry [15, 73]. Level set methods [72, 103] or particle-based methods [39, 80] are also commonly used for determining interface surfaces and meshing them. While these methods produce high quality and topologically accurate meshes, they can also be computationally expensive.

In computational fluid dynamics, the extraction of the interface between multiple materials from volumetric simulations has been extensively studied. The simple line interface calculation (SLIC) algorithm [86] partitions cells with axis-aligned planes, such that the total material volume in each cell is correct but results in discontinuous
interfaces and is only first-order accurate. Alternative methods including the piece-wise linear interface calculation (PLIC) algorithm [129] and its extensions [92, 102] provide second order accuracy for these data. Other approaches involve isosurfacing over the dual grid [21], grid subdivision [10], and extensions to generalize polyhedral meshes [3]. The main difference of these methods is that they make assumptions of continuity, low curvature, and smoothness that are acceptable for fluid-flow data but not necessarily for the kinds of problems addressed in Chapter 7.
CHAPTER 2

BACKGROUND

We first begin by enumerating the different geometric domains that are used throughout this dissertation. Each algorithm discussed in this dissertation make use of a special structure called the Delaunay triangulation. After describing this structure and some of its relevant properties, we will introduce a high level approach to use the Delaunay triangulation for mesh generation.

2.1 Geometric Domains

Our primary concern is with meshes on domains that are embedded in \( \mathbb{R}^2 \) or \( \mathbb{R}^3 \). A knowledge of some concepts from basic topology and geometry are assumed. For additional references, we refer the reader to [45, 56, 68, 69].

2.1.1 Implicit Domains

Let \( f : \mathbb{R}^d \to \mathbb{R} \) be a \( C^2 \)-smooth function, called a scalar field. Scalar fields are found in a number of common applications; for example, pressure or temperature within a volume can be represented by a scalar field. While it is possible to express a scalar field as a closed form equation, often they are represented by a discrete sample. Let \( V \subset \mathbb{R}^d \) be a discrete point set. We call the set of pairs \( \{(p, f(p)) \mid p \in V\} \) a volume dataset of \( f \). A volume dataset is a subset of the graph of \( f \).
The point set $V$ which samples a volume dataset often has some implied connective structure which meshes the set $V$. Most commonly the points in $V$ form a regular grid lying in $\mathbb{R}^d$, where cells connect up the adjacent points in this grid. However other, non-regular forms exist too. The cells in this mesh are used to interpolate the values of $f$ in unsampled regions. To distinguish between when we are talking about a mesh and when we are referring to the cells in a volume dataset, we often call the cells *voxels*.

We call these domains such as these *implicit* because they implicitly define a collection of surfaces. Assume that $\Sigma = f^{-1}(\kappa)$ is a compact surface. The set $\Sigma$ is the *isosurface* of value $\kappa$ for $f$ and $\kappa$ is the *isovalue* on which $\Sigma$ is defined. Note that one cannot know $\Sigma$ given only the discrete data in the volume dataset. Instead, one interpolates a function $g$ and presumes that $\Sigma = f^{-1}(\kappa) = g^{-1}(\kappa)$.

We define a *$d$-dimensional vector field*, $v : \mathbb{R}^d \to \mathbb{R}^d$, in terms of its $d$ component scalar fields, $f_j : \mathbb{R}^d \to \mathbb{R}$ for $1 \leq j \leq d$:

$$v(x) = (f_1(x), f_2(x), \ldots, f_d(x)).$$

We also find vector fields in many domains. Surface gradients, wind and fluid flow are commonly encoded as a vector field. When we build a volume dataset for a vector field, we again take a set $V \subset \mathbb{R}^d$ such that each $p \in V$ vertex has an associated vector value $v(p)$. Thus, the volume dataset encodes the set of pairs $(p, v(p))$. Since we only know the true values of $v$ at the points in the we will use interpolation (over a mesh) to approximate $v$ elsewhere. We will do some comparisons to study how the mesh and samples in $V$ affect this interpolation with respect the ground truth values of $v$. 
2.1.2 Piecewise Smooth Complexes

Assume a generic intersection property that a $k$-manifold $\sigma \subset \mathbb{R}^3$, $0 \leq k \leq 2$, and a $j$-manifold $\sigma' \subset \mathbb{R}^3$, $0 \leq j \leq 2$, intersect (if at all) in a $(k + j - 3)$-manifold if $\sigma \not\subset \sigma'$ and $\sigma' \not\subset \sigma$. We use $\text{Int} \ X$ and $\text{Bd} \ X$ to denote the interior and boundary of a topological space $X$, respectively.

A PSC is a collection of $k$-dimensional faces called vertices (0-faces), curves (1-faces), and patches (2-faces) (see Figure 2.1). Here each $k$-face is a compact subset of a $C^2$-smooth $k$-manifold, $0 \leq k \leq 2$. $k$-faces are closed and hence contains their boundaries. For a PSC $\mathcal{D}$, we use $\mathcal{D}_i$ to denote the set of $i$-faces in a PSC (called the $i$-th stratum) and $\mathcal{D}_{\leq i}$ to represent the union $\mathcal{D}_0 \cup \mathcal{D}_1 \cup \ldots \cup \mathcal{D}_i$.

Figure 2.1: Left: A clover-shaped 1-dimensional PSC. Middle: There are five elements in $\mathcal{D}_1$, highlighted in red, orange, yellow, green, and blue. Right: And two elements in $\mathcal{D}_0$, highlighted in cyan and magenta. Note there is no limitation on the number of elements which share adjacent boundaries.

For $\mathcal{D}$ to be a PSC, it must also satisfy the usual requirements for being a complex: (1) interiors of the elements are pairwise disjoint and for any $\sigma \in \mathcal{D}$, $\text{Bd} \ \sigma \subset \mathcal{D}$; (2) for any $\sigma, \sigma' \in \mathcal{D}$, either $\sigma \cap \sigma' = \emptyset$ or $\sigma \cap \sigma'$ is a union of elements in $\mathcal{D}$. We use $|\mathcal{D}|$
to denote the underlying space of $D$. For $0 \leq k \leq 2$, we also use $|D_k|$ to denote the underlying space of $D_k$.

For any point $x$ on a 2-face $\sigma$, we use $n_\sigma(x)$ to denote a unit outward normal to the surface of $\sigma$ at $x$. For any point $x$ on a 1-face $\sigma$, $n_\sigma(x)$ denotes a unit oriented tangent to the curve of $\sigma$ at $x$.

**Multi-Labeled Data**

One can define an implicit form of PSCs using labeled data stored as scalar fields. Consider a set of $N$ $C^2$-smooth indicator functions [90], $f_i : \mathbb{R}^3 \rightarrow \mathbb{R}$, $1 \leq i \leq N$. Each $f_i$ corresponds to the presence of some label $i$ throughout the input dataset.

For practical purposes, multi-label datasets come in the form of a 3D grid sampled over a discrete set $V \subset \mathbb{R}^3$ with a list of $N$ scalar values at each point $p \in V$. The scalar values correspond to the values of each of the indicator functions at $p$.

For any $x \in \mathbb{R}^3$, the indicator functions dictate the labeling at $x$. Since the values are scalar, if $f_i(x) \geq f_j(x), \forall j \neq i$, then $x$ is given the label $i$. The set $R_i = \{x \in \mathbb{R}^3 \mid f_i(x) \geq f_j(x), \forall j \neq i\}$ is the region in space given the label $i$. Note that this definition implies that a point $x$ may have multiple labels, or alternatively, the sets $R_i$ are not disjoint. We make some assumptions about how these sets can intersect. For any two labels $i$ and $j$, where $i \neq j$, the interface $R_i \cap R_j$ is assumed to be either empty or a set of smooth two-dimensional patches. Similarly, for any three labels $i$, $j$, and $k$, where $i \neq j \neq k$, the interface $R_i \cap R_j \cap R_k$ is assumed to be either empty or a set of smooth one-dimensional curve pieces.

We can define a piecewise smooth complex $D$ using the sets $R_i$. The set of 2-faces, $D_2$, is defined by the union of all pairwise interfaces between the labeled regions, that is $D_2 = \bigcup_{i \neq j} R_i \cap R_j$. Using the set of 2-faces, we can define the set of 1-faces. Each
curve in $\mathcal{D}_1$ is defined as the intersection three or more elements of $\mathcal{D}_2$. Because of our assumption on the intersections of triples of labeled regions, this means each element in $\mathcal{D}_1$ is a piece of a smooth curve. However, these curves may be on the boundary of more than three labeled regions. We include in $\mathcal{D}_1$ only those elements which are not equal to discrete points. These elements will be placed in $\mathcal{D}_0$, which we define as the set of points where three of more elements of $\mathcal{D}_1$ intersect. By definition, any two curves will always intersect in a set of points, each of which lies on the boundary of at least four labeled regions.

### 2.2 Delaunay Triangulations and Voronoi Diagrams

For a finite point set $S \subset \mathbb{R}^3$, let Del $S$ and Vor $S$ denote the Delaunay triangulation and Voronoi diagram of $S$. We will define these structures using the domain $\mathbb{R}^3$, and illustrate them in $\mathbb{R}^2$, but their definitions generalize to any metric space. These structures both partition a space in a dual sense. For each point $p \in S$, define the *Voronoi cell of $p$*, as the set $V_p = \{ x \in \mathbb{R}^3 | \forall q \in S, \|p - x\| \leq \|q - x\| \}$. Each Voronoi cell is a convex polyhedron, and for any two $p, q \in S$, if $V_p \cap V_q$ is non-empty then it is the set of points which are equidistant from $p$ and $q$. For $2 \leq j \leq 4$, the intersection of $j$ Voronoi cells is called a $(4-j)$-dimensional Voronoi face. The 0-, 1-, and 2-dimensional Voronoi faces are called *Voronoi vertices*, *edges*, and *facets*, respectively. Together, the *Voronoi diagram of $S$* is the collection of all Voronoi faces (Figure 2.2, Left).

The convex hull of $j \leq 4$ points in $S$ is a $(j-1)$-dimensional *Delaunay simplex*, $\sigma$, if the vertices of $\sigma$ define a $(4-j)$-dimensional Voronoi face, $V_\sigma$, in Vor $S$. We call $\sigma$ and $V_\sigma$ the *dual* of each other. The 1-, 2-, and 3-dimensional Delaunay simplices are
called Delaunay edges, triangles, and tetrahedra, respectively. The Delaunay simplices decompose the convex hull of \( S \) into the Delaunay triangulation of \( S \), denoted \( \text{Del} S \) (Figure 2.2, middle). One interesting property that we will often use is that each Delaunay simplex has a circumscribing ball which is empty of all other points in \( S \) (Figure 2.2, Right).

We put one condition on the point set that is required for many of our proofs. The point set \( S \) is assumed to be in general position, or that no 4 coplanar points in \( S \) lie on the same circle and no 5 points in \( S \) lie on the same sphere. This assumption removes the non-generic cases where the Delaunay triangulation would be ambiguously defined. In practice though, it is possible to encounter situations where the points are in a degenerate position. A number of approaches to handle these situations exist \([44, 57]\). However, we rely on the CGAL library \([23]\) to compute our Delaunay triangulations, and our experiments show that it robustly handles these degeneracies.

We will also use a generalization of the Delaunay triangulation, often called the weighted Delaunay triangulation \([11, 56]\). This triangulation is defined using a distance metric that is slightly different from Euclidean distance. In \( \mathbb{R}^3 \), a weighted point \( p = (x, r) \) has coordinate information \( x \in \mathbb{R}^3 \) as well as a scalar weight \( r \) associated with it. Conceptually, we treat these points as closed balls centered at \( x \) of radius \( r \), written \( B(x, r) \). The squared weighted distance of between any two weighted points \( p, q \) is \( \| x_p - x_q \|^2 - r_p^2 - r_q^2 \).

Under this distance metric, we can define weighted versions of both the Delaunay triangulation and Voronoi diagram. If every point has a weight of zero, these structures correspond to their unweighted counterparts. Consider the Voronoi face lying
between two points of zero weight, its location is on the plane equidistant from both points. If we add positive weights at one of the points, it causes the Voronoi face to be pushed away, the higher the weight, the more the “push.” Intuitively, as we add weights to different points, the normal direction of each Voronoi face does not change, it is simply pushed one way or another based on the weights. For a weighted point set $S$ we will use the same notation $\text{Del} S$ and $\text{Vor} S$ for these structures, but throughout this dissertation we will make it clear whether $S$ is weighted or unweighted.

### 2.2.1 Restricted Delaunay Triangulations

Delaunay triangulations provide a key aspect for mesh generation: the connectivity between samples. Their special properties are essential to many of the proofs used for algorithm correctness. Since the Delaunay triangulation partitions the convex hull of a point set, it often contain many extra components which we would like to filter.
out. For example, consider building a mesh which approximates a surface embedded in \( \mathbb{R}^3 \). Ideally we would like a set of two-dimensional elements, such as triangles, which matching the dimensionality of the target domain. However, the Delaunay triangulation will have three-dimensional Delaunay tetrahedra connecting samples. We describe a way to remove these elements without violating the structure of the Delaunay triangulation.

To accomplish this filtration, we rely heavily on a subcomplex of the Delaunay triangulation called the restricted Delaunay triangulation. Let \( S \) be a point set sampled from some domain \( D \subset \mathbb{R}^3 \). For any subset \( X \subseteq D \), the \emph{restricted Voronoi face}, \( V_\xi|_X \), is the intersection \( V_\xi \cap X \). The \emph{restricted Delaunay triangulation}, \( \text{Del}_S|_X \), consists of the Delaunay simplices dual to restricted Voronoi faces, i.e. \( \text{Del}_S|_X = \{ \xi \in \text{Del}_S \mid V_\xi \cap X \neq \emptyset \} \). A simplex \( \tau \in \text{Del}_S \) is in \( \text{Del}_S|_X \) if its dual Voronoi face \( V_\tau \) has a non-empty intersection with \( X \). Figure 2.3 illustrates an example restricted Delaunay triangulation in two dimensions.

When dealing with PSCs, an \( i \)-face \( \sigma \in D_i \) should be meshed with \( i \)-simplices. For any \( \sigma \in D \), let \( \text{Del}_S|_\sigma \) denotes the Delaunay subcomplex restricted to \( \sigma \). \( \text{Del}_S|_\sigma \) may have lower dimensional simplices not incident to any restricted \( i \)-simplex. Therefore, we compute special subcomplexes of restricted complexes. Define the following \( i \)-dimensional subcomplexes (see Figure 2.4):

\[
\text{Skl}^i S|_\sigma = \text{closure}\{ t \mid t \in \text{Del}_S|_\sigma \text{ is an } i\text{-simplex} \}
\]

We extend the definition to each strata \( D_i \) of \( D \):

\[
\text{Skl}^i S|_{D_i} = \bigcup_{\sigma \in D_i} \text{Skl}^i S|_\sigma.
\]
Figure 2.3: Restricted Delaunay triangulations. Left: a sample of our clover PSC with its corresponding Voronoi diagram (in blue). Right: Within the dual Delaunay triangulation (dashed and red segments), only a subset of its edges are restricted to the clover (red segments). When a sample is chosen carefully, restricted Delaunay triangulations can approximate this shape.

Notice that computation of $\text{Skl}^i \, S|_{D_1}$ is easier than $\text{Del} \, S|_{D_1}$ since the former one involves computations of intersections only between $(3 - i)$-dimensional Voronoi faces with $i$-faces in $D$. In fact, the only computation we need to determine $\text{Skl}^1 \, S|_{D_1}$ and $\text{Skl}^2 \, S|_{D_2}$ is Voronoi edge intersections with $|D|$.

2.2.2 Delaunay Properties

Voronoi diagrams partition the ambient space of the point set. Moreover, when the point set is sampled properly from a smooth surface, the Voronoi cells take on other interesting properties. In particular, the shape of the Voronoi cells can be used to approximate the normal and feature size of the surface at each point [8, 9, 45] as well as how accurate this approximation is [4]. We will use the notion of poles defined
by Amenta and Bern [8] for estimating the local feature size as was done in Dey et al. [51]. The positive pole for a point $q$ is the furthest Voronoi vertex in $V_q$ (this may be infinitely far away if $V_q$ is unbounded). The negative pole is the furthest Voronoi vertex of $V_q$ in the opposite direction. The pole height, $h_q$, for $q$ is the distance from $q$ to its negative pole.

For a smooth surface $\Sigma$, a Voronoi diagram satisfies the topological ball property if each $k$-dimensional Voronoi face either does not intersect the surface or intersects it in a $(k-1)$-ball. We know that if a point sample $S$ of a surface $\Sigma$ is sufficiently dense then $\text{Vor} \ S$ satisfies the topological ball property [8, 45]. Edelsbrunner and Shah’s theorem [58] says that if $\text{Vor} \ S$ satisfies the topological ball property then $\text{Del} \ S|_\Sigma$ is homeomorphic to $\Sigma$. Together, these facts provide the basis for the topological guarantee of many Delaunay algorithms. An extended version of this property, the extended topological ball property, will also be used to give topological guarantees on non-smooth shapes.
2.3 Delaunay Refinement for Mesh Generation

The Delaunay triangulation gives us a means for connecting samples of domains; however, we still need a tool to generate samples. We take an iterative approach that was pioneered by Chew [35] and Ruppert [100] in two dimensions and later extended to polyhedral domains [36, 81, 107]. This technique, called Delaunay refinement has been used to generate meshes for nearly two decades. The simplicity of the Delaunay refinement paradigm and some natural properties that the Delaunay triangulation optimize are one explanation for its popularity. In addition, many of these techniques benefit from provable guarantees on the output mesh. The generic structure of the algorithm is such that different constraints can be easily imposed on the output to generate meshes which satisfy different properties.

```
1 DelRefine(D)
2 Initialize a sample S from D. Compute DelS.
3
4 while (a property is not satisfied) {
5     Insert a new point c into S.
6     Update DelS.
7 } //end while
8
9 Output a mesh using DelS.
```

Figure 2.5: Delaunay Refinement Algorithm.

In its more general form, the approach can be described in a short loop shown in Figure 2.5. The technique presupposes the existence of a primitive which generates new sample points c on the domain. The algorithm constructs a mesh iteratively. At
each step of the refinement, a set of properties are checked on the mesh (i.e. triangle size, triangle shape, local topology), and if there is a violation a new point is inserted to improve the offending region. The new point inserted often matches a “furthest point criteria” used by Chew [35]. The Delaunay triangulation is updated and these properties checked until they are satisfied; explicitly forcing the output mesh to have some set of desired features. It then suffices to prove that the algorithm terminates to guarantee algorithm correctness.

To show termination, an idea known as a packing argument is often applied. A packing argument states that if points are being inserted into a compact domain, and each point inserted maintains a lower bound on the distance to all other points, then only finitely many points may be inserted.

During Delaunay refinement for surfaces and PSCs, restricted Delaunay triangulations act as a mesh of a working approximation of the input. As a subset of the Delaunay triangulation, elements in the restricted Delaunay triangulation inherit the Delaunay property as well. This fact is important for meshes with shape quality concerns, a byproduct of satisfying Delaunay constraints is that elements have a tendency toward improved shape over elements used in arbitrary triangulations over the same point set.

To better understand Delaunay refinement, we discuss some of the different properties which trigger refinement for various domains.

### 2.3.1 Preserving Geometry for Polyhedral Meshing

Mesh generation for polyhedral domains [2, 75, 81, 100, 107] enforces the property that each input polyhedral and edge appear as a collection of Delaunay simplices in
the output. As such, the algorithm triggers insertions when some face of the input does not appear in the output. Unfortunately, the first algorithms failed to handle inputs where faces meet at acute angles, because insertions near small angles could continue indefinitely.

A number of solutions were proposed to handle these situations [33, 38, 83, 108, 109] with the general theme of protecting the regions near sharp angles using some measure of the local feature size [33, 38, 83]. However, sometimes computing this information is expensive, difficult, or impossible so additional research was necessary to minimize these computations [30, 91] while guaranteeing termination on inputs of arbitrary angle.

2.3.2 Preserving Topology for Smooth Surface Meshing

Since the output mesh of a polyhedral domain conforms exactly to the input, topology preservation was automatic with these algorithms. However, in the smooth setting guaranteed topology is at the forefront of many Delaunay refinement techniques. For curves in two dimensions, topology is generally marginally less difficult [20], but in three dimensions significant research has gone into producing output meshes with correct topology. Leveraging the sampling theory of smooth surfaces [8, 9, 45] a number of Delaunay refinement algorithms have been proposed for meshing smooth domains [17, 26, 31, 63] as well as the volumes contained within them [88]. In particular, the algorithm of Cheng et al. [31] triggers topological violations by enforcing the topological ball property on all mesh vertices.
2.3.3 Meshing Piecewise-Smooth Domains

Neither meshing algorithm for polyhedral domains nor for smooth surfaces provides a satisfactory solution in the case where the input is a piecewise smooth complex. The non-smooth regions where the smooth patches meet poses an extra challenge for past researchers. Boissonnat and Oudot [18] alleviated this problem for a class of surfaces that are only mildly non-smooth as they forbid non-smooth regions subtending small angles. A more recent work of Rineau and Yvinec also suffers from restrictions on the angles that smooth patches meet [98].

In a recent work Cheng, Dey, and Ramos [29] succeeded in designing an algorithm to handle non-smoothness with arbitrarily small angles in the input. Drawing upon the idea of protecting small angle regions with balls [31], they protect the curves where different surface patches meet. A novelty of the algorithm is that these protecting balls are turned into weighted points and a Delaunay refinement is run using weighted Delaunay triangulations. The refinement is triggered by violations of the extended topological ball property [58] to ensure topology preservation for PSCs. This algorithm requires expensive tests to guarantee the topology is preserved; in later chapters we will address how to capture the topology in a more computationally practical manner.
A common problem for researchers who work with vector fields is managing the excessive amount of data they are processing. Vector fields are often represented discretely as a set of samples in the ambient space of the field with a vector associated with each sample. Since no clean sampling theory exists for vector fields, generally datasets are defined over large meshes which are oversampled to preserve features. A related question is how best to interpolate the data in unsampled regions. Both the simplification and interpolation techniques seek to define the vector field in unsampled regions in a way which preserves topological features and maintains a geometric closeness with the original field.

We study [50] both of these questions by approaching the problem using a Delaunay technique. Our approach simplifies vector field datasets by leveraging a simple paradigm, vertex deletion in Delaunay triangulations. We choose which vertex to delete by selecting the vertex in the current mesh which has the lowest value of a local field error measurement, biased to maintain regions near critical points of the field. Our work is effective because of this metric as well as our choice of the Delaunay triangulation. The Delaunay triangulation is known to give good approximations of
Figure 3.1: Simplification of a 2D vector field to error threshold $\varepsilon = 0.15$. The surfaces for the two component functions are preserved as well as their zero-level sets. By juxtaposing the simplified vector field (red) on top of the original (black), we see little change from the original.

scalar fields. Since a vector field can be regarded as a collection of component scalar fields, a Delaunay triangulation has the potential to preserve each component and thus the structure of the vector field as a whole. Figure 3.1 shows an overview of our simplification technique on a two dimensional dataset.

Simplification of triangulated scalar field data has received significant attention in the research community. Cignoni et al. [37] give an overview of simplification in three dimensions involving edge collapses as well as methods to calculate the approximation error. Garland and Zhou [60] discuss quadric-based simplification for a variety of domains including scalar fields. A vector field can be thought of as a composition of multiple scalar fields defined on a shared domain. One perspective of the simplification technique we present is that of concurrently simplifying these fields. However, special
care must be taken and different metrics need to be used to preserve features of the
vector field which result from the composite of these scalar fields.

One of the main contributions of this chapter is to show that a simple, easily
implemented Delaunay strategy works very well for simplifying vector fields in two and
three dimensions. We support this claim with several observations from experiments
run on two and three dimensional vector fields. We also compare our work to the
simplification technique of Platis and Theoharis [93] who use edge collapses in a
non-Delaunay setting.

3.1 Simplification Algorithm

Let $V$ be the volume dataset which samples a given vector field. We first construct
a Delaunay mesh $\text{Del} V$. Let $C$ be the collection of $d$-dimensional Delaunay simplices
(triangles for $d = 2$ and tetrahedra for $d = 3$). At each vertex $p$ of the mesh, we have
a fixed vector field value $v(p)$. For all other points $x$ inside a simplex $c \in C$, the vector
field is defined by linearly interpolating the values of each scalar field based on the
vertices \{ $p^c_j \in c \mid j \in [1, d]$ \} of $c$. Let \( (\beta_1^c, \beta_2^c, \ldots, \beta_d^c) \) be the barycentric coordinates
of $x$ in a simplex $c$ and \{ $v(p^c_j)$ \} be the collection of vectors defined at the vertices of
$c$. We define an $d$-dimensional piecewise linear (PL) vector field, \( \hat{v} : \mathbb{R}^d \rightarrow \mathbb{R}^d \), as

\[
\hat{v}(x) = \sum_{j=1}^{d} \beta_j^c v(p^c_j) \quad \text{where} \quad x \in c.
\]

Our simplification algorithm simplifies the underlying mesh $\text{Del} V$ by deleting
vertices in $V$. A simplified vector field is obtained by linear interpolation over this
simplified mesh.
3.1.1 Error Metric

One difference between our algorithm and the algorithms of Theisel et al. [113, 114] and Platis and Theoharis [93] is that we always maintain a Delaunay triangulation of some subset of the input point set. A second principle difference is that instead of using edge collapses to reduce our mesh, we delete individual vertices. At each iteration of the algorithm we must decide a vertex to delete. This choice is made by selecting the vertex whose deletion causes the least amount of error at the vertex. Let \( v' \) be the new vector field we create after removing a vertex \( p \) from the mesh. We choose the following error metric, \( \varphi : \mathbb{R}^d \to \mathbb{R} \), defined as

\[
\varphi(x) = \frac{\|v'(x) - v(x)\|}{\|v(x)\|}.
\]

Figure 3.2 illustrates the calculation of this error metric. The error \( \varphi(p) \) captures the relative change in the field at \( p \) after its removal from the mesh. Note first how deletion in a Delaunay triangulation only locally affects the mesh. To see this fact consider the link of \( p \) defined as follows. The \( d \)-simplices incident to \( p \) constitute its star. The lower dimensional simplices that border the star constitute the link of \( p \). In Figure 3.2, the link of \( p \) is the cycle of dotted edges. When \( p \) is removed, its star and only its star is also removed. The hole which is created by the removal of the star is retriangulated, and it is known that to maintain the Delaunay property of the triangulation the only changes to the triangulation which occur are within this hole. Therefore, any error occurs only within the region bordered by the link.

For simplicity, we choose only to look at the change in the vector field for the deleted vertex. This decision allows the equation for \( \varphi(x) \) to make a comparison with the initial vector field and the new vector field since vector values at vertices are
fixed. This calculation is memoryless; we do not track error over the entire sequence of deletions. Despite this fact, we see that experimentally this metric performs well up to certain thresholds for keeping the global error bounded. However, a more complicated scheme for sampling the maximal error could be envisioned since the changes to the field happen only in a local neighborhood of the deleted vertex.

3.1.2 Vertex Deletion Algorithm

We start our simplification by creating a priority queue of all vertices $p \in V$ based on their deletion error, $\varphi(p)$. We perform a trial deletion for each vertex in the Delaunay triangulation to compute $\varphi(p)$. Vertices with the lowest error have highest priority. We then remove the top vertex from the queue, delete it from the mesh, and retriangulate the hole so that we maintain a Delaunay triangulation. We also update the priority of the remaining vertices in the priority queue. This process is repeated until we have removed all vertices that have $\varphi(p)$ less than some user specified error.
threshold, $\varepsilon$. For summary and completeness, we present the pseudocode for the algorithm in Figure 3.3.

```plaintext
1 SimplifyVF($\varepsilon$, V, F)
2 DelV ← Construct_Delaunay(V)
3 A ← Build_Queue(DelV, F)
4 $\varphi_A ← 0$

5 while ($\varphi_A < \varepsilon$) {
6     ($p$, $\varphi_p$) ← Pop(A)
7     Q ← Adjacent_Vertices(DelV, p)
8     Delaunay_Remove(DelV, p)
9     if ($\varphi_p > \varphi_A$)
10        $\varphi_A ← \varphi_p$
11     forall $q ∈ Q$
12        Update_Priority(A, q)
13 } //end while
```

Figure 3.3: Vector field simplification algorithm, via vertex deletion.

Only a few of the algorithm’s steps require detailed explanation. Build_Queue() takes the Delaunay triangulation and vector field values as input and computes $\varphi(p)$ for all $p ∈ V$, placing each pair in the queue sorted by $\varphi(p)$. We then iteratively pop off vertices and remove them from the mesh. This operation requires a local remove and repair to maintain the Delaunay property of Del V, performed by Delaunay_Remove(). Since the mesh has changed, we update the priority of all adjacent vertices in the set $Q$, the only vertices affected by the removal of $p$. During all of this, we keep track of the maximum error seen $\varphi_A$, and we stop once the maximum error exceeds the threshold $\varepsilon$. 

33
3.2 Feature Preservation

The error metric $\varphi$ has two important qualities that help maintain the structure of the vector field: (1) the magnitude and angle changes of the vector field are bounded at each vertex removal (Theorem 3.2.1) and (2) the topological structure given by separatrices remains mostly unaffected (experimental results).

3.2.1 Bounded Error

We have already indicated that when a vertex $p$ is removed from the Delaunay triangulation we examine the error in the vector field at $p$ to guide our deletion algorithm. We show that by bounding $\varphi(p)$, we can bound the change in magnitude and angle of the vector field at $p$ after its removal.

**Theorem 3.2.1.** For a sample point $p \in \mathbb{R}^d$, if $\varphi(p) \leq \varepsilon$ then

1. $\|\hat{v}'(p)\| - \|\hat{v}(p)\| \leq \varepsilon \|\hat{v}(p)\|$ and

2. $\angle(\hat{v}'(p), \hat{v}(p)) \leq \arcsin \varepsilon$.

*Proof.* We give a geometric proof. For shorthand we let $\hat{v} = \hat{v}(p)$ and $\hat{v}' = \hat{v}'(p)$. The condition $\varphi(x) \leq \varepsilon$ implies that the tip of $\hat{v}'$ lies in a ball of radius $\varepsilon \|\hat{v}\|$ centered at the tip of $\hat{v}$ as shown in left of Figure 3.4.

The change in magnitude between $\hat{v}$ and $\hat{v}'$ in the worst case can be seen by growing or shrinking $\hat{v}$ along the direction of $\hat{v}$. This is shown in the center of Figure 3.4. Thus in the worst case the change in vector magnitude is the radius of the ball, $\varepsilon \|\hat{v}\|$.

Similarly, as shown in the right image of Figure 3.4, the worst case for the change in angle occurs in the case where $\hat{v}'$ intersects the ball at a point on the circumference.
such that the line of $\hat{v}'$ is tangential to the ball. At this point, the angle between $\hat{v}'$ and $\hat{v}$ is precisely $\arcsin \varepsilon$.

Theorem 3.2.1 indicates that at each step of the algorithm when we remove a vertex using our error metric we bound both the change in magnitude and angle of the vectors at $p$. Unfortunately, the region contained within the link of $p$ may not enjoy the same property. A sequence of vertex removals could change the error globally as each individual error compounds. However, we find that for low values of $\varepsilon$ we can simplify the vector significantly without too much global error. Our experiments show that we can greatly reduce the size of the mesh for $\varepsilon \leq 0.05$ while preserving the topology. This means less than a 5% change in the vector magnitude and a change of angle of $\approx 2.87^\circ$ or less. We discuss choosing $\varepsilon$ more thoroughly in Section 3.3 based on our different experiments.
3.2.2 Topological Preservation

The critical points of a vector field $v$ are simply those points $x \in \mathbb{R}^d$ such that $v(x) = 0$. Alternatively, one can think of them as the intersections of the zero-level sets, or nullclines, of each component function $f_i$. In classifying a vector field, if we know all of the critical points and how the flow moves between them, we capture the essence of the vector field. Critical points are commonly classified using analysis of the Jacobian of $v$ at the critical point. Informally, sinks and sources are those critical points where all eigenvalues of the Jacobian have the same sign (negative and positive respectively), and saddle points are those critical points where the eigenvalues have mixed signs. We refer the reader to more complete classifications in two [67] and three dimensions [62] as well as the pictorial study of Abraham and Shaw [1] and more thorough introductory mathematical texts [68, 69].

An integral curve, $\gamma : I \subset \mathbb{R} \to \mathbb{R}^d$, based at a point $x$, is a function with the following constraints:

$$
\gamma(0) = x.
$$

$$
(\forall t \in I) \ [ \dot{\gamma}(t) = v(\gamma(t)) ].
$$

In many contexts, integral curves (streamlines) are actually treated as the image, $\gamma(I)$, of the function $\gamma$. Separatrices in two dimensions are special integral curves that connect saddle points to other critical points. These integral curves are defined in the direction of the two eigenvectors of the Jacobian of the field at the critical point. In three dimensions, separatrices are both curves and surfaces corresponding to the three different eigenvectors of the Jacobian.
A topological structure for a vector field is obtained by taking the set of critical points and connecting them together using separatrices. These partition the domain of the vector field such that each region has a specific type of flow. This partitioning is regarded as a topological skeleton of the vector field.

We observe that the topological skeleton is preserved quite well in practice when the threshold for \( \varphi \) is kept low. This preservation is the result of the bias of the error metric. Since we normalize by \( \|v(x)\| \) when calculating \( \varphi \), we bias simplification so that we do not remove points that are close to critical points unless they only cause relatively small changes to the vector field. The result is that we can still greatly simplify a vector field while maintaining most of its topology. However, the simplification may alter the topology at places where some instabilities are present in the vector field.

In Section 3.3 we discuss two such instabilities that can cause topological changes when simplifying. One results from separatrices that run closely in parallel. The compounded local error that changes the destinations of streamlines can change the origin and destinations of these separatrices. The analytic vector field we discuss has this type of instability. The other instability results from zero level sets that intersect tangentially. They can cause critical points to be created, destroyed, or moved with small perturbations in the vector field. The ocean wind dataset that we chose has examples of this instability.

3.3 Experimental Results

We ran experiments on two and three dimensional vector field data from both analytical functions and real data. All experiments were run on a Pentium 4, 2.8Ghz
with 1 GB of RAM. For two dimensional Delaunay triangulations we used Shewchuk’s Triangle software [106] where as in three dimensions we used CGAL [23] for this purpose. Visualizations in two dimensions were created in OpenGL using a fourth-order Runge-Kutta integration for the streamlines while in three dimensions we used OpenDX [87].

### 3.3.1 Two Dimensions

Our first set of experiments were done with two dimensional vector fields for ease of viewing. We show results for an analytic vector field and a real ocean wind dataset.

#### Analytic Vector Field

We first present an analytic example, corresponding to the vector field shown in Figure 3.1. This vector field is defined by the following equations:

\[
\begin{align*}
    f_1(x) &= 50 \cos(0.06\|x\|) \\
    f_2(x) &= 50 \cos(0.001x_1x_2).
\end{align*}
\]

We show in Figure 3.5 streamlines for this vector field as well as its separatrices in cyan. We initially define the vector field on a grid of 29244 vertices over the domain \([-120, 120] \times [-120, 120]\).

We aid visualization using separatrices drawn in shades of cyan for the original vector field and magenta for the simplified vector field. We can observe that this vector field has sixteen critical points within the experimental domain. Of particular interest are the saddle/focus pairs on both the top right and bottom left of the vector field.
We find that simplifying to $\varepsilon = 0.05$ (1221 vertices) results in a good approximation of the field. We can push this bound further and still maintain geometry, as in Figure 3.1 where we have simplified to $\varepsilon = 0.15$ (428 total vertices). Here both the vectors and the nullclines stay relatively close to the original. However, the effects of unstable separatrices may alter topology. Consider Figure 3.6 where the simplified separatrices (magenta) overlay the original (cyan).

Compounded small changes in the vectors of the field cause the separatrices to change their destinations. In the circled region of Figure 3.6 we removed two vertices. The corresponding change affects the separatrices $\alpha$ and $\beta$ as labeled in the center image. Originally, $\alpha$ passes inside of $\beta$ and spirals inward towards the focus $z$, while $\beta$ begins at the point $x$ and passes around the focus, ending at the saddle point $y$. After the change, these two separatrices “swap” in the affected region. The result (right image) is that $\alpha$ (originally passing inside of $\beta$) now ends up outside of $\beta$ and
sweeps down and outside of the domain. The separatrix $\beta$ originates from a closed orbit, drawn as the black ring, and spirals outward to its destination saddle point $y$.

The corresponding error threshold to the mesh of this size was $\varepsilon = 0.116$. Completely preventing topological changes would require more sophisticated methods that preserve all separatrices along with all critical points. For simplicity, we bias to preserve areas around critical points as opposed to preserving all separatrices. This works well for the most part, but may not prevent interactions among separatrices that run nearly parallel. Interestingly, the instability of this region means that the topological change is not permanent; as we continue to remove vertices the topological structure of the mesh eventually returns to its original form. In this particular example, when we reduce the mesh size to 533 vertices, the two separatrices swap back. Despite this small topological change, as Figure 3.1 demonstrates, the resulting vector field is still geometrically close to the original at an even lower error threshold ($\varepsilon = 0.15$).
**Ocean Wind Data**

For our second example in two dimensions we use an ocean wind dataset. We show a representation of the dataset in Figure 3.7 (Top) with a set of streamlines integrated from this data as well as its nullclines. This dataset originally has 165027 vertices and 12 critical points. We have boxed a key area where topological instability occurs.

We also show the resulting vector field after simplification with $\varepsilon = 0.1$ in Figure 3.7 (Bottom). At this level of simplification the number of vertices in the mesh was reduced to 889. We note here how the simplification streamlines and the original streamlines are still very close as shown by the zoomed regions despite the huge change in the number of vertices.

We have already discussed the effect of instabilities caused by separatrices running parallel. This example shows the effect of another kind of instability caused by nullclines that run almost tangential to one another. Even at low levels of simplification, this instability can cause critical points to be created, destroyed, or moved far away since changes in the mesh cause the nullclines to interleave in different manners.

We show in Figure 3.8 what happens in the boxed region of Figure 3.7. This figure shows the nullclines for the original as well as $\varepsilon = 0.01$ (7626 vertices) and 0.10 (889 vertices). In this region there are originally five critical points (arrows on the upper half). However, the nullclines pass each other tangentially at this region, so small perturbations in the field cause the nullclines to interleave, creating new critical points (arrows on the lower half). At $\varepsilon = 0.01$, two new critical points are created whereas at $\varepsilon = 0.10$ these critical points move, and the two critical points on the right disappear.
Figure 3.7: Top: Ocean wind data with nullclines. Bottom: Ocean wind data after simplifying to $\varepsilon = 0.10$. 

42
Figure 3.8: A topological change of the ocean wind dataset. Left to right: Nullclines and critical points of the simplification at $\varepsilon = 0.00$, 0.01, and 0.10. Critical points are highlighted with arrows.

### 3.3.2 Three Dimensions

We present results for experiments run on two different three dimensional datasets. We combine conventional methods for vector field visualization (streamlines) as well as the use of nullclines to demonstrate the success of our algorithm in three dimensions. Furthermore, we compare our results with that of Platis and Theoharis [93].

**Gravity Field Dataset**

Our first three dimensional dataset is that of a gravity field used in [93]. This field represents the gravitational field of 3 spheres and has five different critical points. We use OpenDX [87] to depict this vector field in Figure 3.9. Here we draw 200 randomly placed streamlines (colored by vector magnitude) contained in the bounding volume of dimensions $[0, 29] \times [0, 29] \times [0, 29]$. We also draw the three nullcines, which are surfaces in three dimensions. The blue surface is for the $f_1$ nullcline, green for $f_2$, and gold for $f_3$. 

43
For our experiments we took the original dataset used by Platis and Theoharis and first created a Delaunay mesh of the 27000 sample points using 146334 tetrahedra. We first ran our algorithm to a variety of levels of $\varepsilon$. We then ran the OpenDX module of Platis and Theoharis using the same input Delaunay mesh and the percentage of tetrahedra corresponding to each $\varepsilon$ level as a stopping goal. Note that their algorithm uses an integer input for the target percentage of tetrahedra, so we chose the closest match for all cases. Platis and Theoharis’s algorithm requires five weights to balance their error metric; for this dataset we used the weights $w_{FA} = 0$, $w_{FL} = 10$, $w_D = 0$, $w_C = 0.5$, and $w_V = 0.5$. Obviously, the results of their algorithm are dependent on the choice of these weights; we selected those which were representative of the least error for this dataset in their paper.

To provide a quantitative comparison of the two algorithms, we use a field error calculation similar to the one used by Platis and Theoharis. This error is calculated
by sampling each simplified vector field on a dense grid over the domain. We calculate 
\[ \|v(x) - v(x)\| \] for each sample point \( x \) and scale it by dividing by the maximum vector 
magnitude in the initial field, giving a more meaningful percentage. We show \( \varepsilon_{\text{mean}} \) 
and \( \varepsilon_{\text{max}} \), the mean and maximum values of this error, for both vector fields in Table 
3.1 for each level of simplification.

<table>
<thead>
<tr>
<th>( \varepsilon )</th>
<th>( \varepsilon_{\text{mean}} )</th>
<th>( \varepsilon_{\text{max}} )</th>
<th></th>
<th>( % )</th>
<th>( \varepsilon_{\text{mean}} )</th>
<th>( \varepsilon_{\text{max}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>0.04%</td>
<td>0.48%</td>
<td>40</td>
<td>0.04%</td>
<td>1.05%</td>
<td></td>
</tr>
<tr>
<td>0.02</td>
<td>0.11%</td>
<td>0.85%</td>
<td>21</td>
<td>0.08%</td>
<td>4.32%</td>
<td></td>
</tr>
<tr>
<td>0.03</td>
<td>0.20%</td>
<td>1.53%</td>
<td>12</td>
<td>0.14%</td>
<td>5.55%</td>
<td></td>
</tr>
<tr>
<td>0.04</td>
<td>0.30%</td>
<td>2.47%</td>
<td>8</td>
<td>0.21%</td>
<td>27.80%</td>
<td></td>
</tr>
<tr>
<td>0.05</td>
<td>0.37%</td>
<td>2.92%</td>
<td>6</td>
<td>0.27%</td>
<td>27.80%</td>
<td></td>
</tr>
<tr>
<td>0.07</td>
<td>0.48%</td>
<td>4.36%</td>
<td>4</td>
<td>0.37%</td>
<td>27.80%</td>
<td></td>
</tr>
<tr>
<td>0.10</td>
<td>0.74%</td>
<td>11.65%</td>
<td>3</td>
<td>0.47%</td>
<td>27.80%</td>
<td></td>
</tr>
<tr>
<td>0.15</td>
<td>1.12%</td>
<td>22.18%</td>
<td>2</td>
<td>0.63%</td>
<td>27.80%</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Field error for simplification of the gravity field.

From this table, we see that for all levels of simplification the Delaunay technique 
had a higher \( \varepsilon_{\text{mean}} \) and a lower \( \varepsilon_{\text{max}} \). In fact for all simplification levels past 10% 
of the original number of tetrahedra there is a large spike in the maximum for the 
algorithm of Platis and Theoharis. This result most likely indicates that there is a 
region in the domain that is not preserved well.

In Figure 3.10 we show a visual comparison of this field simplified to a threshold 
of \( \varepsilon = 0.05 \) which corresponds to 6% of the original tetrahedra. We draw the original 
streamlines as white tubes and the streamlines of each simplified field as smaller, 
colored tubes. In this manner, it is easy to pick out which streamlines have changed
Figure 3.10: Simplified gravity field streamlines and nullclines. Left: Our Delaunay technique at $\varepsilon = 0.05$. Right: Technique of Platis and Theoharis run to a corresponding level of 6% tetrahedra.

since they will break through the outer white tube and bend in a different way. Visually it appears that further from the regions of critical points our Delaunay algorithm allows more error, and in regions near the critical points we remain closer to the original field. This observation is also supported by examining the field error in a local neighborhood of each of the five critical points.

In Table 3.2 we show the field error of a dense sample in a ball of radius 1.0 around each critical point. The Delaunay algorithm has dramatically lower $\varepsilon^{\text{mean}}$ and $\varepsilon^{\text{max}}$ values at all levels of simplification, demonstrating how the normalization of our error metric causes these regions to be preserved.
<table>
<thead>
<tr>
<th>ε</th>
<th>ε^{mean}</th>
<th>ε^{max}</th>
<th>%</th>
<th>ε^{mean}</th>
<th>ε^{max}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>0.00%</td>
<td>0.00%</td>
<td>40</td>
<td>0.01%</td>
<td>0.26%</td>
</tr>
<tr>
<td>0.02</td>
<td>0.00%</td>
<td>0.00%</td>
<td>21</td>
<td>0.03%</td>
<td>0.68%</td>
</tr>
<tr>
<td>0.03</td>
<td>0.00%</td>
<td>0.03%</td>
<td>12</td>
<td>0.06%</td>
<td>2.47%</td>
</tr>
<tr>
<td>0.04</td>
<td>0.00%</td>
<td>0.15%</td>
<td>8</td>
<td>2.78%</td>
<td>38.49%</td>
</tr>
<tr>
<td>0.05</td>
<td>0.00%</td>
<td>0.17%</td>
<td>6</td>
<td>2.81%</td>
<td>38.49%</td>
</tr>
<tr>
<td>0.07</td>
<td>0.00%</td>
<td>0.19%</td>
<td>4</td>
<td>2.94%</td>
<td>38.49%</td>
</tr>
<tr>
<td>0.10</td>
<td>0.05%</td>
<td>0.90%</td>
<td>3</td>
<td>3.10%</td>
<td>38.49%</td>
</tr>
<tr>
<td>0.15</td>
<td>0.11%</td>
<td>1.41%</td>
<td>2</td>
<td>3.22%</td>
<td>38.49%</td>
</tr>
</tbody>
</table>

Table 3.2: Field errors for gravity field simplification, sampling in balls of radius 1.0 around all critical points

**Lorenz Attractor**

We also study an analytic vector field, the Lorenz attractor, described by the following equations:

\[ f_1(x) = \alpha(x_2 - x_1) \]
\[ f_2(x) = \beta x_1 - x_2 - x_1x_3 \]
\[ f_3(x) = x_1x_2 - \gamma x_3 \]

where \( \alpha = 10 \), \( \beta = 28 \), and \( \gamma = \frac{8}{3} \). This form and the values for \( \alpha \), \( \beta \), and \( \gamma \) were taken from [69]. We examine the field on a grid of 35937 vertices contained within the bounding volume of \([-24, 24] \times [-24, 24] \times [-4, 44]\) (to center the strange attractor for the system). The initial Delaunay triangulation had 196608 tetrahedra. We show streamlines as well as the three component nullcline surfaces for this system in Figure 3.11.
We show the field error for both of the resultant simplified fields in Table 3.3. For all levels of simplification, our algorithm maintains both a lower $\varepsilon^{\text{mean}}$ and $\varepsilon^{\text{max}}$.

In Figure 3.12 (Top) we show streamlines for the Lorenz attractor at $\varepsilon = 0.025$ and its equivalent level of 2% simplification using Platis and Theoharis’s method. Both algorithms are able to maintain a close approximation to the original, although slight errors for the streamlines manifest as a result of compounded differences in the
<table>
<thead>
<tr>
<th>$\varepsilon$</th>
<th>$\varepsilon_{\text{mean}}$</th>
<th>$\varepsilon_{\text{max}}$</th>
<th>%</th>
<th>$\varepsilon_{\text{mean}}$</th>
<th>$\varepsilon_{\text{max}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0025</td>
<td>0.05%</td>
<td>0.80%</td>
<td>31</td>
<td>0.09%</td>
<td>1.04%</td>
</tr>
<tr>
<td>0.0050</td>
<td>0.07%</td>
<td>1.05%</td>
<td>14</td>
<td>0.16%</td>
<td>1.47%</td>
</tr>
<tr>
<td>0.0075</td>
<td>0.09%</td>
<td>1.34%</td>
<td>9</td>
<td>0.22%</td>
<td>2.25%</td>
</tr>
<tr>
<td>0.0100</td>
<td>0.11%</td>
<td>0.96%</td>
<td>6</td>
<td>0.26%</td>
<td>2.28%</td>
</tr>
<tr>
<td>0.0150</td>
<td>0.15%</td>
<td>1.23%</td>
<td>4</td>
<td>0.39%</td>
<td>2.87%</td>
</tr>
<tr>
<td>0.0200</td>
<td>0.19%</td>
<td>1.45%</td>
<td>3</td>
<td>0.53%</td>
<td>5.42%</td>
</tr>
<tr>
<td>0.0250</td>
<td>0.26%</td>
<td>2.90%</td>
<td>2</td>
<td>0.67%</td>
<td>6.67%</td>
</tr>
<tr>
<td>0.0300</td>
<td>0.47%</td>
<td>3.98%</td>
<td>1</td>
<td>0.76%</td>
<td>6.47%</td>
</tr>
</tbody>
</table>

Table 3.3: Field error for simplification of the Lorenz attractor.

simplified vectors. Since we have used longer streamlines than in previous examples to visualize this field, the errors are more apparent. Also, it appears that further from the Lorenz attractor our algorithm shows more error than that of Platis and Theoharis, whereas in the region of the attractor the Delaunay algorithm maintains a closer match.

We also show the nullcline surfaces for $f_2$ at the same level of simplification in Figure 3.12 (Bottom). Both $f_2$ and $f_3$ are saddle surfaces and the attractor itself lies where the inner parabolic lines intersect with the plane of $f_1$. At this level of simplification the surfaces are well preserved, although in the results from Platis and Theoharis the surface is perturbed slightly more. The surface of $f_3$ (not shown) had similar artifacts as well.

### 3.4 Discussion

We have presented an easy to implement algorithm for vector field simplification based on Delaunay triangulations. There have been few works that approach vector
fields from this perspective, and our results indicate that Delaunay triangulations can be leveraged in both two and three dimensions for this application. We have demonstrated that this technique can be successful at greatly reducing the size of vector field datasets while maintaining a closeness to the original. In particular, our
experiments show that by bounding the local change to low thresholds ($\varepsilon \leq 0.05$) we can retain both a nearness in geometry and avoid major topological change.

We have also presented a comparison of our work with an alternate technique of simplification by Platis and Theoharis [93]. Our results ran with a comparable execution time and an equal or better field error in the simplified output fields. An important aspect of our algorithm is that we preserve areas near critical points as a result of the normalization of our error metric. Our technique is also applicable to vector fields in any dimension. An additional advantage of our algorithm over that of Platis and Theoharis is that we require only a single parameter as opposed to six.
CHAPTER 4

ISOSURFACE MESH GENERATION

Scientists often desire techniques for visualizing three dimensional volume datasets produced by computational fluid dynamics simulation or medical technologies such as CT and MRI. One main approach to visualization involves computing an isosurface of some scalar field the dataset samples. The dataset implicitly defines such a surface for any scalar value, but in many cases having a mesh to explicitly show a single value is more useful for understanding the dataset. We present a Delaunay refinement approach to build a mesh for isosurfaces.

Our approach addresses a practical limitation with traditional Delaunay refinement. It is known that the computation of a three dimensional Delaunay triangulation and its restricted Delaunay triangulation can be quite expensive. In particular, the cumulative cost of repeated insertions in the structures may become prohibitive. Our main contribution is a technique to reduce the burden of three dimensional Delaunay triangulations. We observe that the topological ball property, a key to topological guarantee, is satisfied early in the refinement process and that the bulk of the refinement is carried out to capture the geometry. Once the density of sampled points is enough for the topology of the restricted Delaunay triangulation to be correct, we can discard the Delaunay triangulation entirely and continue refinement using a
more lightweight structure to produce an output faster. We show that our two stage
technique greatly improves the time and space required for computations.

4.1 Delaunay vs. Non-Delaunay Isosurfaces

The merits of Delaunay refinement, and the surfaces it produces, have been shown
in the context of smooth domains [17, 31, 51, 63, 88]. Sampling theory in the context
of surface reconstruction [8, 45] shows that in surface approximations both point-wise
and normal-wise approximation errors depend on the circumradius of the triangles.
Since a Delaunay triangulation keeps the circumradii of triangles small, it often pro-
vides a good approximation.

By comparison, Marching Cubes [76, 127] is a popular algorithm for extracting
isosurface meshes which are not necessarily Delaunay. This algorithm has many ad-
vantages; in particular, its speed and simplicity allow for rapid isosurface generation.
However, the following issues remain a concern:

1. Topological and geometric closeness are not guaranteed between the output
   triangulation and the isosurface $\Sigma$.

2. The triangulation of $\Sigma$ is not adaptive in the sense that the gradedness or
distribution of the surface samples is not sensitive to the features of the surface.

3. No constraints regarding triangle shape exist—this deficiency can lead to nu-
merical error and other simulation issues for finite element methods on the
output polygonal mesh.
Variants of the original Marching Cubes exist to solve the topological concern [14, 34, 120]. However by adapting the Delaunay refinement approach in [31], our algorithm, DelIso, addresses all three of these concerns simultaneously and efficiently while producing a Delaunay mesh.

Figure 4.1: Atom (left) and Fuel (right) isosurfaces. In both images, the left surfaces are produced by Marching Cubes and the right are the output of DelIso.

As motivating examples, consider the isosurfaces generated in Figure 4.1. Here we show the isosurfaces of the Atom and Fuel datasets generated with the isovaleses 20.1 and 70.1, respectively. In each figure we have the outputs of both Marching Cubes and our isosurface meshing algorithm, DelIso. While both are topologically correct for these cases, only DelIso is provably so. In particular, we can guarantee that the mesh DelIso outputs is homeomorphic to the isosurface defined by trilinear interpolation.

Figure 4.1 also shows how DelIso produces meshes that adapt to the feature size of the surface. The outer spherical regions of the isosurface are meshed with larger, flatter triangles, the center torus is meshed with smaller triangles, and the inner ellipsoid with even smaller triangles. As a result, the Marching Cubes version has
22498 vertices compared with only 2089 vertices in the DelIso version. The Fuel dataset similarly has smaller triangles around the through holes of the surface. In both datasets, there are arbitrarily skinny triangles in the Marching Cubes output, while in the output of our Delaunay refinement the aspect ratios are kept bounded.

4.2 Restricted Delaunay Refinement

Our goal is to avoid using the three dimensional Delaunay triangulation for the bulk of the insertions in a Delaunay refinement. We will see that we can maintain the restricted Delaunay triangulation without the need for the full three dimensional Delaunay triangulation. Since we are modeling our isosurface as a smooth surface, our refinement will work in the usual manner of triggering insertions until the topological ball property is satisfied. We note that when the topological ball property is satisfied, each Voronoi edge $V_\sigma$ that intersects the surface does so in a single point $x$. A ball $B_\sigma$ centered at $x$ and circumscribing $\sigma$ is called the Voronoi ball of $\sigma$. We say that a triangle $\sigma$ is encroached by a point $p$ if $B_\sigma$ contains $p$ in the interior.

4.2.1 Avoiding the Full Delaunay Triangulation

The principal computational bottleneck in working with the restricted Delaunay triangulation is computing and maintaining the three dimensional Delaunay triangulation under point insertions. Our algorithm overcomes this difficulty by splitting the Delaunay refinement algorithm into two stages, where after the first we discard the Delaunay triangulation. For this strategy to work, one needs to have inserted sufficiently many points in the first stage so that the restricted Delaunay triangulation remains homeomorphic to the surface with further insertions. Although we cannot
determine this point precisely in the algorithm, our experiments show that by controlling some user defined parameters we can insert enough points in the first stage to fulfill this condition.

Consider an insertion step in the second stage where the three dimensional Delaunay triangulation is not available. Let \( p \) be a new point to be inserted in the existing point set \( S \). By our Stage 2 assumption, we know that \( \text{Del}(S \cup \{p\})|_{\Sigma} \) remains homeomorphic to \( \Sigma \). Thus \( \text{Del}(S \cup \{p\})|_{\Sigma} \) is a piecewise linear manifold, so the union of all triangles incident to any point in \( \text{Del}(S \cup \{p\})|_{\Sigma} \) is a topological disk. Let \( D \) be the set of triangles incident on \( p \) whose underlying space is \( \cup D \). The set \( D \) consists of the triangles that are in \( \text{Del}(S \cup \{p\})|_{\Sigma} \) but not in \( \text{Del} S|_{\Sigma} \). This simple observation is crucial for determining the new triangles that are needed to update the restricted Delaunay triangulation upon inserting \( p \).

First we determine the triangles that should be deleted from \( \text{Del} S|_{\Sigma} \) as a result of inserting \( p \). Let \( E \) be this set of triangles with underlying space \( \cup E \). \( E \) is the set of triangles which are encroached by \( p \). We know the boundary of \( \cup E \) must be same as that of \( \cup D \). Since \( \cup D \) is a topological disk, its boundary must be a single cycle implying that \( \cup E \) is also a topological disk. Once we determine \( E \), computing \( D \) is trivial as its triangles are computed by connecting \( p \) to the boundary edges of \( E \). Therefore, the main task in updating \( \text{Del} S|_{\Sigma} \) reduces to computing the set \( E \). Figure 4.2 illustrates the technique we use.

Since \( \cup E \) is a topological disk, we can compute its triangles by a walk in the adjacency structure of \( \text{Del} S|_{\Sigma} \). Suppose that we have computed a connected set \( E' \subseteq E \). For each triangle \( \sigma \in \text{Del} S|_{\Sigma} \) not in \( E' \) but sharing an edge with a triangle
Figure 4.2. After Stage 1, we can insert points in \( \text{Del} S|_{\Sigma} \) without \( \text{Del} S \). Left: We refine a triangle \( \sigma \) (shaded) by inserting the center of \( B_\sigma \). Center: This point encroaches a topological disk \( E \). Right: We replace \( E \) with \( D \) by connecting the boundary of \( E \) to the point.

in \( E' \), we check if \( \sigma \) is encroached by \( p \). If so, we add \( \sigma \) to \( E' \) and then walk to its neighbors.

To check if a restricted Delaunay triangle \( \sigma \) is encroached we need to compute the Voronoi ball \( B_\sigma \) and thus the intersection point between \( \Sigma \) and the dual Voronoi edge, \( V_\sigma \), of \( \sigma \). We first compute the geometric dual of \( \sigma \), a line \( \ell_\sigma \) passing through the circumcenter of \( \sigma \) and perpendicular to the plane of \( \sigma \). Since the line \( \ell_\sigma \) contains \( V_\sigma \), we can use it to find the intersection of \( V_\sigma \) and \( \Sigma \). Let \( x \) be the closest point to the circumcenter of \( \sigma \) where \( \ell_\sigma \) intersects the surface \( \Sigma \). Since after Stage 1 the triangles in \( \text{Del} S|_{\Sigma} \) approximate the surface to a reasonable level, the intersection of \( V_\sigma \) and \( \Sigma \) lies close to \( \sigma \) meaning that \( x \) is the intersection point between \( V_\sigma \) and \( \Sigma \) as well. Therefore, the ball centered at \( x \) which circumscribes \( \sigma \) is \( B_\sigma \). If \( p \) lies in the ball \( B_\sigma \), the triangle \( \sigma \) is encroached and does not belong to \( \text{Del}(S \cup \{p\})|_{\Sigma} \). For
improving future checks to determine if \( \sigma \) is encroached, we keep the point \( x \) stored with \( \sigma \) when \( \sigma \) is created.

The final piece needed to compute \( E \) is an initial triangle in \( E' \). In general, this computation would require a point location test. Fortunately, in our case the point \( p \) to be inserted is always the center of the Voronoi ball \( B_\sigma \) of some triangle \( \sigma \in \text{Del} S|\Sigma \). The insertion of \( p \) certainly eliminates \( \sigma \) from the restricted Delaunay triangulation. Therefore, we can initialize \( E' \) with \( \sigma \) and then walk through adjacent triangles to determine \( E \).

### 4.2.2 Intersection Search

Typically at each iteration of a Delaunay refinement we insert a point to correct some violation of desired criteria. In the style of Chew’s “furthest point strategy” [35] we select points which are as far away as possible to insert. As a consequence of this property, the termination of the algorithm can be shown because we maintain a lower bound on the distance between points. The points that meet this criteria will always lie on Voronoi edges, since the edges are equidistance from some of the samples. For example, if we select a point not on a Voronoi edge and locate its nearest neighbor \( p \) (by identifying the Voronoi cell that point is in), we can easily see that the points on the boundary of that cell maximize the distance to \( p \) within the cell.

Since we desired points which lie on the isosurface for our mesh, we will insert the intersection between some Voronoi edge and the surface at each point. In three dimensions, Voronoi edges are either line segments or rays, thus one of the core computations of our algorithm is computing the intersection between an arbitrary line or ray and the surface \( \Sigma \). By partitioning the volume dataset into voxels we
accomplish this task by first collecting the voxels that intersect the line and then determining if the line intersects the surface in each of these voxels. We elaborate on the actual computation of the intersection later in Section 4.4.

For the second stage of our algorithm, we seek the closest intersection point of $\ell_\sigma$ with $\Sigma$ for a triangle $\sigma$. This intersection point is assumed to be the intersection of $V_\sigma$ with the surface. To improve this computation, we start traversing voxels at the circumcenter of $\sigma$ and step in both directions along $\ell_\sigma$. By searching in this manner we find the intersection point after traversing only a few voxels.

4.2.3 Timing Comparisons

Before describing the algorithm, we provide some examples to illustrate how DelIso improves the computational cost over the standard Delaunay refinement involving full three dimensional structures. For our experiments, we used a variety of volume datasets generated at various isovalues $\kappa$, Table 4.1 gives a list of each dataset, the isovalue at which we generated the surface, and the dimensions of the dataset.

Table 4.1 also shows a timing comparison. This table indicates that a significant amount of processing time can be saved by using our technique. The first column of timings were generated by running the first stage of DelIso to a particular threshold (described in the next section) and the second was generated by running DelIso in a two stage form to the same threshold. For all datasets except for the Chest dataset, we saw a speed up of 2 of greater. For the Chest dataset, we had to make the switch to the second stage at a later time, most likely because of the complex features within the lungs.
<table>
<thead>
<tr>
<th>Dataset</th>
<th>κ</th>
<th>Dimensions</th>
<th>3D</th>
<th>DelIso</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fuel</td>
<td>70.1</td>
<td>$64 \times 64 \times 64$</td>
<td>17.73</td>
<td>5.18</td>
<td>3.43</td>
</tr>
<tr>
<td>Atom</td>
<td>20.1</td>
<td>$128 \times 128 \times 128$</td>
<td>16.39</td>
<td>5.85</td>
<td>2.80</td>
</tr>
<tr>
<td>BluntFin</td>
<td>54.3</td>
<td>$256 \times 128 \times 64$</td>
<td>116.69</td>
<td>42.28</td>
<td>2.76</td>
</tr>
<tr>
<td>Engine</td>
<td>40.1</td>
<td>$256 \times 256 \times 256$</td>
<td>311.84</td>
<td>101.10</td>
<td>3.08</td>
</tr>
<tr>
<td>Leg</td>
<td>22.1</td>
<td>$341 \times 341 \times 93$</td>
<td>441.11</td>
<td>165.23</td>
<td>2.67</td>
</tr>
<tr>
<td>Tooth</td>
<td>146.9</td>
<td>$256 \times 256 \times 161$</td>
<td>76.90</td>
<td>34.35</td>
<td>2.24</td>
</tr>
<tr>
<td>Chest</td>
<td>82.7</td>
<td>$384 \times 384 \times 240$</td>
<td>1033.56</td>
<td>657.34</td>
<td>1.57</td>
</tr>
<tr>
<td>Baby1</td>
<td>35.8</td>
<td>$256 \times 256 \times 98$</td>
<td>485.58</td>
<td>204.44</td>
<td>2.38</td>
</tr>
<tr>
<td>Baby2</td>
<td>131.6</td>
<td>$256 \times 256 \times 98$</td>
<td>1134.87</td>
<td>468.93</td>
<td>2.42</td>
</tr>
<tr>
<td>Monkey</td>
<td>40.1</td>
<td>$256 \times 256 \times 62$</td>
<td>2570.14</td>
<td>791.46</td>
<td>3.25</td>
</tr>
<tr>
<td>Aneurism</td>
<td>124.6</td>
<td>$256 \times 256 \times 256$</td>
<td>663.65</td>
<td>262.05</td>
<td>2.53</td>
</tr>
<tr>
<td>Pig</td>
<td>120.1</td>
<td>$512 \times 512 \times 134$</td>
<td>614.19</td>
<td>288.83</td>
<td>2.13</td>
</tr>
</tbody>
</table>

Table 4.1: Volume datasets with Delaunay refinement time comparisons: using a 3D Delaunay triangulation for the entire refinement (3D) vs. our algorithm (Dellso). Times are in seconds.

### 4.3 Algorithm Details

In our two-stage algorithm, the first stage uses the full three dimensional Delaunay triangulation to create a point set whose restricted Delaunay triangulation is assumed to satisfy the topological ball property with respect to the isosurface in question. Intuitively, after the first stage we have recovered a “rough” version of the surface, but have not satisfied the geometric constraints desired. We next extract the restricted Delaunay triangulation into a polygonal mesh data structure. The second stage uses only this mesh to continue refinement of the restricted Delaunay triangulation until it is geometrically close to the isosurface.

In Figure 4.3 we give pseudocode for our two stage algorithm to mesh the isosurface $\Sigma$ at isovalue $\kappa$ defined by a volume dataset. We assume this volume dataset is accessible globally. These two stages are named Recover() and Refine() respectively.
This algorithm assumes that globally we have a primitive operation that given a volume dataset and isovalue $\kappa$ will compute the set of intersection points between a line segment $r$ and $\Sigma$.

\begin{verbatim}
1 DelIso($\kappa$)
2     DelS ← InitTriangulation()
3     DelS ← Recover(DelS)
4     $T_S$ ← Refine(DelS)
5     return $T_S$.
\end{verbatim}

Figure 4.3: DelIso algorithm.

InitTriangulation() bootstraps our algorithm by creating a small (random) sample of points $S$ on $\Sigma$ and computing Del $S$. In the Recover() stage we take Del $S$ and by repeated insertions augment the set $S$ so that the restricted Delaunay triangulation satisfies a manifold property. We also perform a small amount of geometric refinement so that in the second stage we can assume future insertions will not break the topological ball property. Next, in the Refine() stage we extract Del $S|_\Sigma$ into a polygonal mesh data structure, $T_S$, and further refine it by inserting additional points without using the full Delaunay triangulation. The result is a restricted Delaunay triangulation (stored in $T_S$) which is both topologically correct and geometrically close to $\Sigma$.

4.3.1 Isosurface Recovery (Stage 1)

In Figure 4.4 we show the pseudocode for the Recover() stage. These steps are motivated by the topological recovery phase of Dey et al. [51]. We explain each step in turn.
1  Recover(Del S)
2      Del S ← MultiIntersect(Del S)
3      Del S ← ExtractManifold(Del S)
4      Del S ← ApproximateGeom(Del S, , , r_{\text{min}})
5      Del S ← ExtractManifold(Del S)
6  return Del S.

Figure 4.4: Recover() algorithm.

MultiIntersect() takes a Delaunay triangulation of a set \( S \) and computes the restricted Voronoi face \( V_\sigma \cap \Sigma \) of each Delaunay 2-simplex \( \sigma \in \text{Del} \, S \). The set \( V_\sigma \cap \Sigma \) is simply a set of points. If this set has multiple elements, the point furthest from \( \sigma \) is inserted. This process is repeated until no 2-simplices in Del \( S \) have restricted Voronoi faces with more than one element.

ExtractManifold() takes a Delaunay triangulation of a set \( S \) and considers the set of Delaunay 2-simplices in the restricted Delaunay triangulation of \( S \) which are adjacent to each \( q \in S \). It checks that this set \( D_q = \{ \sigma \in \text{Del} \, S|_\Sigma \mid \sigma \cap q \neq \emptyset \} \) forms a topological disk. If it does not, we insert the vertex \( V_\sigma \cap \Sigma \) furthest from \( q \) where \( \sigma \in D_q \). This process is repeated until for each \( q \in S \) the set \( D_q \) forms a topological disk.

After ExtractManifold() returns, \( \text{Del} \, S|_\Sigma \) is a manifold, but the topological ball property may still not hold. We observe that only a minor amount of geometric refinement is typically needed before we can discard the Delaunay triangulation in our second stage.

This geometric refinement is performed in ApproximateGeom(). This step checks that each 2-simplex \( \sigma \in \text{Del} \, S|_\Sigma \) satisfies certain geometric constraints. Let \( r \) be the
circumradius of $\sigma$, $h$ be the distance from the circumcenter of $\sigma$ to the point in $V_{\sigma} \cap \Sigma$, and $l$ be the shortest edge length of $\sigma$. We check that:

1. $h/r > \varepsilon$,
2. $r/l > \lambda$, and
3. $r > r_{\text{min}}$.

If $r$ is not too small (condition 3) and either of conditions 1 or 2 holds, we insert the intersection point $V_{\sigma} \cap \Sigma$. This process is repeated until there is no such 2-simplex $\sigma$ with $r > r_{\text{min}}$ for which condition 1 or 2 holds. The three parameters $\varepsilon$, $\lambda$, and $r_{\text{min}}$ are user inputs. Typical values for each are $\varepsilon = 0.2$, $\lambda = 2.0$, and $r_{\text{min}} = 0.001b$ where $b$ is the smallest dimension of the bounding box. These values are scale independent and work well for all models we tested.

Each condition is motivated for different reasons. Condition 1 causes the insertion of points which create 2-simplices in $\text{Del } S|_\Sigma$ respecting the features of $\Sigma$. Dey et al. [51] originally checked the ratio $r/h_q$, the radius over the pole height (see Section 2.2.2) for a point $q$, for capturing geometric features. These two refinement criteria are similar in intent, as one causes the size of triangles to try to match the curvature of the shape while the other causes the triangles to respect the feature size of the shape. However, empirical evidence shows that $h/r$ appears more likely to miss features, but we still favor it here because it is significantly easier to compute.

Condition 2 checks the aspect ratio of each triangle, inserting points to remove triangles whose aspect ratio is greater than the set bound $\lambda$. This causes the triangles in our output mesh to have better shape. Finally, condition 3 prevents triangles from becoming too small—which can cause numeric precision errors.
Figure 4.5: Engine (left) and Tooth (right) isosurfaces. The Tooth isosurface is drawn with a clipping plane to see the inside component of the surface. In both examples the left surface is the final output of DelIso and the right is the out after Recover().

In Figure 4.5 we show the output meshes for the Engine and Tooth datasets. In each of these figures the resulting restricted Delaunay triangulation after the Recover() stage is shown on the right while the final output surface is drawn on the left. We observe that the mesh is topologically correct after Recover(), but still needs further refinement to capture the features appropriately.

4.3.2 Isosurface Refinement (Stage 2)

In the second stage of our algorithm, we extract Del$S|_{\Sigma}$ from the result of the first stage into a polygonal mesh structure which is independent from Del$S$. We continue our Delaunay refinement algorithm, but we will no longer require the three dimensional Delaunay triangulation to compute the restricted Delaunay triangulation.

For a point set $S$, define $T_S = (S, F)$ to be the triangular mesh structure where

$$F \subseteq \{\{a, b, c\} \mid a, b, c \in S\}$$
is the set of facets in $T_s$. We store adjacency information within the facets as well—each facet $f \in F$ knows the three facets which share an edge with $f$. This data structure is satisfactory for representing 2-manifolds (potentially with boundary) as triangular meshes. As discussed in Section 4.2.1, for each $f \in F$ we also store the nearest intersection point $x$ to $f$ between $\Sigma$ and the dual line of $f$ to improve future encroachment checks on point insertions.

In Figure 4.6 we show pseudocode for the second stage of our algorithm. We first compute the pole heights for each vertex in $\text{Del } S$. These pole heights are used as refinement criteria later in the Refine() step. Next, $\text{BuildTriMesh()}$ creates the polygonal mesh structure using $\text{Del } S$, the output of the first stage. Finally, by using $\text{RefineGeom()}$ we refine $T_s$.

```
1 Refine(Del S)
2    ComputePoles(Del S)
3    $T_s \leftarrow \text{BuildTriMesh(Del S)}$
4    $T_s \leftarrow \text{RefineGeom}(T_s, \varepsilon_1, \varepsilon_2, \lambda, r_{\text{min}})$
5    return $T_s$.
```

Figure 4.6: Refine() algorithm.

With the exception of $\text{BuildTriMesh()}$, the other steps of this stage require some additional clarifications:

$\text{ComputePoles()}$ takes as input the Delaunay triangulation, $\text{Del } S$, and computes the pole height, $h_q$, for each $q \in \text{Del } S$. For any $q$, we know that $V_q \cap \Sigma$ is a topological disk at this stage since $\text{ExtractManifold()}$ guarantees this property. Moreover, this disk separates $V_q$ into exactly two subsets, denoted $V_q^+$ and $V_q^-$, on either side of
the disk. Let \( q^+ \) and \( q^- \) be the points which are furthest from \( q \) in \( V_q^+ \) and \( V_q^- \), respectively. \( q^+ \) and \( q^- \) are similar to poles defined for smooth surfaces. We define 

\[
h_q = \min\{\|q - q^+\|, \|q - q^-\|\}.
\]

RefineGeom() takes as input the triangular mesh \( T_S \) output by BuildTriMesh() and generates a new mesh \( T_S \) by refining each triangle \( \sigma \). Let \( h_\sigma \) be the pole height of a triangle \( \sigma \) defined by averaging the pole heights at the vertices of \( \sigma \). We check the following criteria:

1. \( h/r > \varepsilon_1 \),
2. \( r/h_\sigma > \varepsilon_2 \),
3. \( r/l > \lambda \), and
4. \( r > r_{\text{min}} \).

Similar to ApproximateGeom(), we have three types of criteria. First, conditions 1 and 2 are used to capture the features of \( \Sigma \). Condition 3 keeps the aspect ratio of the triangles bounded, and condition 4 prevents triangles from becoming too small. If any of conditions 1, 2, or 3 holds, and condition 4 is satisfied, we insert the point \( V_\sigma \cap \Sigma \). When inserting, we first compute the disk consisting of encroached facets as described in Section 4.2.1, remove it, and then fill it by connecting its boundary to the vertex we insert. In our experiments, we use the same values for \( \lambda \) and \( r_{\text{min}} \) as before. We pick \( \varepsilon_1 = 0.1 \), a reduction from \( \varepsilon = 0.2 \) in the first stage, and \( \varepsilon_2 = 0.2 \).

In practice, we have seen that the value \( r/h_\sigma \) is more desirable than \( h/r \) as a criterion for geometric refinement. It is very easy for a large triangle to have \( h = 0 \), even if there is a high curvature region within of the piece of the surface the triangle is approximating. By comparison, pole heights give a better indication of the features
Figure 4.7: Dellso output isosurfaces. Upper left: Isosurface for Baby1 (back) and mesh of isosurface for Baby2 (front). Upper right: External components of Chest (back), by clipping the front we see the mesh for the lungs, trachea, and bronchi (front). Lower left: Aneurism dataset, by zooming in we see the mesh adaptively samples the tubular features. Lower right: Isosurface of BluntFin and close up view of mesh on the sharp feature.
of the shape itself, since we are taking a measurement that involves more than the location of one point. However, pole computations are fairly expensive and require Del$S$, so we prefer $h/r$ in the first stage. Then at the start of Refine() we compute them once before discarding Del$S$. When a new point is inserted we estimate its pole height by averaging the pole heights of its adjacent vertices. At this stage since $T_S$ is close to $\Sigma$, nearby points have similar pole heights. The averaging step is an effective approximation that allows us to use $r/h_\sigma$ for additional refinement.

Figure 4.7 shows results for additional datasets (Baby, Chest, Aneurism, and BluntFin). In these examples the adaptivity of DelIso is shown by capturing the small features densely and smoothly transitioning to larger triangles in flatter regions. The tube, mouth, and double sheet of Baby, the inner features of Chest, and the tubular structure of Aneurism are meshed densely as a result. To correctly mesh the sharp features of BluntFin, denser triangles were also required.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$t_{RC}$</th>
<th>$n_{RC}$</th>
<th>$t_{RF}$</th>
<th>$n_{RF}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fuel</td>
<td>1.54e-03</td>
<td>1345</td>
<td>1.36e-05</td>
<td>2936</td>
</tr>
<tr>
<td>Atom</td>
<td>1.36e-03</td>
<td>554</td>
<td>1.79e-05</td>
<td>1788</td>
</tr>
<tr>
<td>BluntFin</td>
<td>1.49e-03</td>
<td>8428</td>
<td>1.64e-04</td>
<td>20827</td>
</tr>
<tr>
<td>Engine</td>
<td>1.53e-03</td>
<td>23291</td>
<td>1.56e-05</td>
<td>84942</td>
</tr>
<tr>
<td>Leg</td>
<td>1.66e-03</td>
<td>38889</td>
<td>1.70e-05</td>
<td>82185</td>
</tr>
<tr>
<td>Tooth</td>
<td>1.58e-03</td>
<td>5416</td>
<td>1.54e-05</td>
<td>14242</td>
</tr>
<tr>
<td>Chest</td>
<td>1.82e-03</td>
<td>76887</td>
<td>1.78e-05</td>
<td>280953</td>
</tr>
<tr>
<td>Baby1</td>
<td>1.56e-03</td>
<td>50886</td>
<td>1.67e-05</td>
<td>217031</td>
</tr>
<tr>
<td>Baby2</td>
<td>1.71e-03</td>
<td>120637</td>
<td>1.74e-05</td>
<td>392150</td>
</tr>
<tr>
<td>Monkey</td>
<td>1.60e-03</td>
<td>155304</td>
<td>1.63e-05</td>
<td>423557</td>
</tr>
<tr>
<td>Aneurism</td>
<td>1.71e-03</td>
<td>74173</td>
<td>1.57e-05</td>
<td>140557</td>
</tr>
<tr>
<td>Pig</td>
<td>1.62e-03</td>
<td>52240</td>
<td>1.60e-05</td>
<td>314368</td>
</tr>
</tbody>
</table>

Table 4.2: Average insertion times $t_{RC}$ and $t_{RF}$ (in seconds) for inserting $n_{RC}$ and $n_{RF}$ points in the Recover() ($RC$) and Refine() ($RF$) stages, respectively.
Table 4.2 gives timing results showing the amount of time saved for insertions. For each dataset, we calculate the average insertion times during the Recover() and Refine() stages. In all instances, insertion during Refine() is approximately two orders of magnitude faster. Not shown in this table is where most of the insertion time is spent. For the Recover() stage the majority (80-85%) is spent determining which new Delaunay simplices are restricted Delaunay. The remaining portion is used to compute the new Delaunay triangulation and identify new Delaunay simplices.

4.4 Intersection Search

We have deferred discussion of how to compute the intersection points between a ray and a volume dataset to this section. Let the line segment $r = \overline{a_0a_1}$, be parameterized by $r(t) = a + bt$ for $t \in [0, 1]$ where $a = a_0$ and $b = a_1 - a_0$. Given a volume dataset, an isovalue $\kappa$, and $r$, we compute the set of all points in $r \cap \Sigma$ in two steps, shown as the Intersect() algorithm in Figure 4.8. In Intersect() we first identify which voxels of the volume dataset are potential candidates for intersection and then compute the intersection points in those cells.

```plaintext
1 Intersect(\hat{f}, \kappa, r) \{
2     V \leftarrow \text{CollectVoxels()}
3     X \leftarrow \text{FindIntersections}(V)
4     \text{return } X.
```

Figure 4.8: Intersect() algorithm.
4.4.1 *kd-tree Searches*

There are a number of different techniques for ray tracing isosurfaces based on *kd-tree* data structures. The goal of these approaches is to rapidly identify which primitive elements (in our case, voxels) are candidates for a more expensive intersection check.

We modify one approach in Havran [64] using a method similar to Wald *et al.* [121] and adapt it to our needs. First, we construct a *kd-tree* on the voxels where at each level we split the set of voxels in half along voxel boundaries in the dimension of the split. At each node we store the maximum and minimum function values ($\kappa_{\text{min}}$ and $\kappa_{\text{max}}$) defined over the portion of the volume represented by that node. These values are computed in a recursive manner.

![Diagram of a kd-tree facilitating ray-isosurface intersections.](image)

Figure 4.9: A *kd-tree* facilitates ray-isosurface intersections. Left: Isosurface $\Sigma$, ray $r$, and the *kd-tree*. Center: Using only the tree and $r$ we identify a set of candidate voxels (in green). Right: Using $\kappa_{\text{min}}$ and $\kappa_{\text{max}}$ we can prune the cyan cells.

Once we have a *kd-tree* where each node stores this maximum and minimum information, we can collect the set of candidate voxels where $r$ intersects $\Sigma$ in an efficient manner. Figure 4.9 illustrates this technique in two dimensions. The recursive
traversal of Havran begins at the root of the tree and checks if $r$ crosses the split plane, if so it traverses both sides, otherwise it checks whichever side $r$ lies on and continues into the tree. In this manner we prune away just those voxels that the ray does not intersect. However, we can prune additional voxels by checking if $\Sigma$ is within those voxels too. Using the maximum and minimum isovalues, we do an additional check to ensure that the subtree we are traversing has the isosurface in it by checking $\kappa_{\text{min}} \leq \kappa \leq \kappa_{\text{max}}$. If not, we prune that region even though the $r$ intersects it.

4.4.2 Voxel-Ray Intersection

Once we have collected the set $V$ of candidate voxels in which $\Sigma$ may intersect $r$, we compute the intersection points. One approach is to do an iterative search using Newton’s method or other variants. However, we are most interested in finding these intersections exactly, so we solve the system defined by the equations for $r$ and the trilinear surface specified by the eight sample points at the corner of each voxel. This technique is given by Parker et al. [89] and reduces to solving the roots of (at most) a cubic polynomial. To solve this polynomial, we use the algorithm of Schwarze [104].

Reviewing notation from Parker et al., we can specify this system as follows. First consider a voxel which is located on the unit cube and let $\{\rho_{ijk} \mid i, j, k \in \{0, 1\}\}$ be the set of isovalues at the corners. We can define $\rho(u, v, w)$ for $(u, v, w) \in [0, 1]^3$ by trilinear interpolation:

$$\rho(u, v, w) = \sum_{i,j,k \in \{0,1\}} u_i v_j w_k \rho_{ijk},$$

where $u_0 = u$, $u_1 = 1 - u$, $v_0 = v$, and so on. For a voxel spanned by coordinates $(x_0, y_0, z_0)$ and $(x_1, y_1, z_1)$, we can express any point $(x, y, z)$ in these $(u, v, w)$
coordinates by the transformation:

\[(u, v, w) = \left(\frac{x - x_0}{x_1 - x_0}, \frac{y - y_0}{y_1 - y_0}, \frac{z - z_0}{z_1 - z_0}\right).\]

Consequently, we need only take the set of points on the line segment \(r(t) = a + bt\) between the points \(a_0\) and \(a_1\) and determine where \(\rho(x, y, z)\) is equal to \(\kappa\). Let \((u_0^{a_0}, v_0^{a_0}, w_0^{a_0})\) and \((u_0^{a_1}, v_0^{a_1}, w_0^{a_1})\) be the coordinates of \(a_0\) and \(a_1\) in the \((u, v, w)\) space. Thus \((u_1^{a_0}, v_1^{a_0}, w_1^{a_0})\) will be defined as \((1 - u_0^{a_0}, 1 - v_0^{a_0}, 1 - w_0^{a_0})\) and similarly for \((u_1^{a_1}, v_1^{a_1}, w_1^{a_1})\). We can then define the coordinates for \(a\) and \(b\) in terms of \(a_0\) and \(a_1\).

Finally, we can rewrite the equation for \(\rho\) parameterized by \(t\) as:

\[
\rho(t) = \sum_{i, j, k \in \{0, 1\}} r_{ijk}(t) \rho_{ijk} = \sum_{i, j, k \in \{0, 1\}} (u_i^a + u_i^b t)(v_j^a + v_j^b t)(w_k^a + w_k^b t) \rho_{ijk}.
\]

### 4.4.3 Improving the Speed of Intersection Searches

In our implementation of DelIso, one important enhancement is used during the Refine() stage. As discussed in Section 4.2.2, we can improve searching for the intersection point of \(\ell_\sigma\) and \(\Sigma\) by stepping along the voxels from the circumcenter of \(\sigma\). Since during this stage we are searching for the closest intersection point to \(\sigma\), we search iteratively with a segment of increasing size along the dual line of \(\sigma\) centered at \(\sigma\)’s circumcenter. For each iteration we perform an intersection check. If we find an intersection we are done, otherwise we increase the size of our segment and repeat, skipping voxels we have already searched. In this manner, we check the closest voxels first—the result is an intersection search with fewer voxel-ray intersection computations.
In Table 4.3 we show a comparison of the number of voxel-ray intersection computations versus the number of searches at each stage of the algorithm. Since the voxel-ray intersection requires solving a potentially cubic system, this computation is expensive and one would like to minimize it. This table shows that on average we only do one per search, but for those searches where we actually do find an intersection point, we tend to do more. However, in the Refine() stage, we minimize the number of computations by searching for intersection points incrementally along the dual line of each triangle.

### 4.5 Boundary Issues

One assumption for our algorithm is that the isosurface does not have any boundary. For volume datasets, it is common to have isosurfaces with boundaries since the
surface is often truncated by the bounding volume of the dataset. For example, in our experiments only Atom, Engine, and Tooth did not have boundary curves at the iso-values chosen. Special treatment is required in the Disk() test to handle boundaries, since points on the boundary cannot satisfy this condition.

We make two modifications to adapt the original algorithm to isosurfaces with boundary. First, we precompute the boundary curves and insert them into the initial sample. Second, we modify the Disk() test to allow half disks for points on the boundary.

The insertion of the boundary points is done at the end of the InitTriangulation() step. After computing the random sample of points and inserting them, we next compute the set of boundary curves and insert those. For simplicity, this is done using Marching Squares on each of the six boundary faces. Upon insertion, we mark these points and allow the Disk() test to be satisfied for these points by either a complete topological disk or a topological half disk. In this manner, we relax the constraints of the algorithm, but only at known boundary points.

4.6 Discussion

We have presented an algorithm, DelIso, for meshing isosurfaces with Delaunay triangles. This algorithm works on the principle of Delaunay refinement with the topological ball property. Although a three dimensional Delaunay triangulation is at the core of this refinement paradigm, we show how the bulk of the computations can be carried out without maintaining the full 3D structure. This improvement eases the computational burden without sacrificing any quality guarantees.
Research is still necessary to make further enhancements to this algorithm. Clearly, the earlier we switch to Refine(), the less time will be required for meshing. However, a certain sampling density is necessary to guarantee the topological ball property at this stage. One would like to understand the precise point where the surface can be extracted with a provable guarantee that we can continue to update $\text{ Del } S|_\Sigma$ without $\text{ Del } S$. Computing ray-volume intersections is also an active field of research which could also improve the implementation of this algorithm. Finally, replacing the more complicated topological tests with one single topological disk test could both ease programmer burden as well as improve the speed of the algorithm.
CHAPTER 5

A PRACTICAL MESHING ALGORITHM FOR PSCS

As indicated in Chapter 4, mesh generation is rarely as simple as modeling the input domain as a smooth surface. Even for the case of isosurfaces, there may be curved regions of non-smoothness or non-manifoldness. These regions are generally approximated as some smooth manifold, requiring tiny, dense triangles to capture the feature. An alternate approach is to model an input as a piecewise-linear shape, where every curved region is protected (as opposed to none). Most surfaces in practice are neither perfectly smooth nor piecewise linear; a higher order model where both smoothness and non-smoothness is accounted for is necessary.

One such broad model that we will use is that of a piecewise smooth complex, or PSC. Example PSCs include smooth and non-smooth surfaces both with and without boundaries, volumes enclosed by them, and most importantly non-manifolds. Recently, Cheng et al. [29] provided a theoretical foundation for a Delaunay refinement algorithm to mesh shapes of this general class. It is the first algorithm that can compute Delaunay meshes for such a large class of domains with theoretical guarantees. However, the major shortcoming of the algorithm is that it involves expensive computations at each refinement stage making it quite hard for implementation.
We will present two versions of an algorithm, DelPSC, that draw upon the ideas in [29], but both are more practical. The first version, DelPSC1 [27, 28], is presented in this chapter and the second, DelPSC2 [48], is described in Chapter 6. This claim of practicality is justified by implementations of both algorithms with experimental results on a vast array of disparate domains. Figure 5.1 shows some examples.

Figure 5.1: Meshed PSCs, Metaball (Smooth), Part (Manifold PSC), and Wedge (Non-manifold with small angles).

The original algorithm in [29] has both a protection phase for the curved features as well as a refinement phase for surface patches. The refinement phase inserts points in the domain iteratively to compensate for four types of violations:

1. A Voronoi edge intersecting the domain multiple times,
2. Normals on the curves and surface patches varying beyond a threshold within Voronoi cells,

3. A Delaunay edge in the restricted triangulation connecting vertices across different patches, and

4. The restricted Delaunay triangles incident to points in a patch do not make a topological disk.

Our first contribution is an algorithm [27] that replaces these four tests with a single one. This single test checks for topological disk violations and inserts points when it fails. Once intersections of surface patches with Voronoi edges are determined, this test is purely combinatorial making it easily implementable.

Obviously, by only ensuring this test we cannot claim that the output has the same topology as the input shape. For example, if the input is a manifold, then by verifying that each point of the output is a manifold we will ensure our shape is a manifold too. But we may not have the same genus as the input shape. Using only our single condition, we are able to guarantee that the output restricted to each stratum of the input PSC is a manifold with all vertices contained within the strata [28].

To account for this problem, we pair our topological condition with a geometric one based on triangle size. Using these conditions together, we can argue that once the triangles are sufficiently small, the output will also be homeomorphic to the input and preserve all input features. We control triangle size with a parameter to the algorithm, and we observe that the algorithm achieves the appropriate resolution point very rapidly.
Following the algorithm of Cheng et al. [29] our algorithm first performs a protection phase where a set of weighted points is computed to protect each one dimensional feature in the later refinement stage. This step requires some expense to compute a feature-size like measure along the non-smooth edges, but we only have to perform it once before refinement begins. In general though, the properties that protecting balls require for the refinement phase can be satisfied in practice by using sufficiently small balls around the non-smooth edges.

5.1 Algorithm Overview

We show an example output of DelPSC1 in Figure 5.2 on the Anchor model. DelPSC1 protects all non-smooth regions (marked in blue in Figure 5.2). These features are preserved in the output. Topology is also recovered successfully. We can easily use DelPSC1 to mesh the volume contained with the PSC as well.

![Example Output](image)

Figure 5.2: Sharp features on Anchor are preserved in both surface (middle) and volume (right) meshing and topology is recovered.
5.1.1 Sharp Feature Preservation

The neighborhoods of the curves and vertices in $\mathcal{D}_{\leq 1}$ are regions of potential problems for Delaunay refinement. First, if the elements incident to these curves and vertices make small angles at the points of incidences, usual Delaunay refinement may not terminate. Second, these curves and vertices represent “features” in the input which should be preserved in the output for many applications. Usual Delaunay refinement may destroy these features [18, 51] or may be made to preserve them for a restricted class of inputs [109].

To overcome this difficulty, we protect all elements in $\mathcal{D}_{\leq 1}$ with balls that satisfy certain properties [29]. These balls are turned into weighted points and we use the weighted Delaunay triangulation for the next stage of the refinement. The properties of the protecting balls make sure that the curves in $\mathcal{D}_1$ remain meshed properly throughout the algorithm. In particular, adjacent points along any curve in $\mathcal{D}_1$ remain connected with restricted Delaunay edges. Specifically, the protecting balls should have the following properties:

**Protection properties:** Let $\omega \leq 0.076$ be a positive constant and $\mathcal{B}_p = B(p, r_p)$ denote the protecting ball of a point $p$.

1. Any two adjacent balls on a 1-face must overlap significantly without containing each other’s centers.

2. No three balls have a common intersection.

3. Say $\mathcal{B}_p$ has center $p \in \sigma$. Further, let $\mathcal{B}_p^* = B(p, R)$ be a different ball with radius $R$ and center $p$ where $R \leq cr_p$ for some $c \leq 8$. 
(a) For \( \tau = \sigma \) or any 2-face incident to \( \sigma \), \( \angle n_\tau(p), n_\tau(z) \leq 2 \omega \) for any \( z \in B_p^* \cap \tau \).

The same result holds for the surfaces of the 2-faces incident to \( \sigma \).

(b) \( B_p^* \) intersects \( \sigma \) in a single open curve and any 2-face incident to \( \sigma \) in a topological disk. The same result holds for the surfaces of the 2-faces incident to \( \sigma \).

After computing the protecting balls, we turn each of them into a weighted vertex that is inserted into a weighted Delaunay triangulation. That is, for each protecting ball \( B_p \), we obtain the weighted point \((p, w_p)\), where \( w_p = r_p^2 \). For technical reasons we need to ensure that each 2-face is intersected by some Voronoi edge in the Voronoi diagram Vor \( S \) of the current point set. The weighted vertices ensure it for 2-faces that have boundaries. For 2-faces without boundary, initially we place three weighted points satisfying the protection properties.

When the protection step is complete, we insert points iteratively outside the protected regions to mesh 2-faces. These points are not weighted and we never insert a new point inside any of the protected elements of \( D_{\leq 1} \). Therefore, all curves in \( D_1 \) are meshed homeomorphically with restricted Delaunay edges whose vertices lie on the curves. This preserves non-smooth features features in the output.

In summary, our algorithm always maintains a point set \( S \) with the following two properties:

1. \( S \) contains all weighted points placed in protection step, and

2. Other points in \( S \) are unweighted and they lie outside the protecting balls.

We call such a point set admissible.

The following lemma [29] is an important consequence of the protection properties.
Lemma 5.1.1. Let $S$ be an admissible point set. Let $p$ and $q$ be adjacent weighted vertices on a 1-face $\sigma$. Let $\sigma_{pq}$ denote the curve segment between $p$ and $q$. $V_{pq}$ is the only Voronoi facet in Vor $S$ that intersects $\sigma_{pq}$, and $V_{pq}$ intersects $\sigma_{pq}$ exactly once.

5.1.2 Faithful Topology Approximation

In a mesh of a 2-manifold, the triangles incident to a vertex should form a topological disk. One can turn this into a condition for sampling 2-manifolds in the input PSC. Our refinement “disk” condition applied to only a single 2-manifold is exactly this condition. However, since a PSC may have several 2-manifolds, potentially forming even non-manifolds, one needs to incorporate some more requirements into the disk condition. Let $p$ be a point on a 2-face $\sigma$. Let $\operatorname{Umb}_D(p)$ and $\operatorname{Umb}_\sigma(p)$ be the set of triangles which are incident to $p$ in $\operatorname{Skl}_2 S|_D$ and $\operatorname{Skl}_2 S|_\sigma$, respectively. The following disk condition is used for refinement. Once the restricted Delaunay triangles are collected, this check is only combinatorial.

\textbf{DiskCondition}(p):

\hspace{1em} (D1) For each $\sigma \in D_2$ containing $p$, the underlying space of $\operatorname{Umb}_\sigma(p)$ is a 2-disk,

\hspace{1em} (D2) The point $p$ is in the interior of this 2-disk if and only if $p \in \operatorname{Int} \sigma$,

\hspace{1em} (D3) Also, if $p$ is in $\operatorname{Bd} \sigma$, it is not connected to any other point on $D_1$ which is not adjacent to it, and

\hspace{1em} (D4) All vertices of $\operatorname{Umb}_\sigma(p)$ are in $\sigma$.

Figure 5.3 illustrates how this disk condition might fail as well as cases where it passes.
Figure 5.3: Disk condition. Left: Triangles incident to point \( p \in \sigma \) and restricted to \( \sigma \) do not form a disk since they form two disks pinched at \( p \) violating condition D1. Middle: The point \( p \in \sigma \) has a topological disk but some of its vertices (lightly shaded) belong to \( \tau \) violating condition D4. Right: Points \( p \) and \( q \) satisfy the disk condition. Point \( p \), an interior point in \( \sigma \), lies in the interior of its disk in \( \sigma \). Point \( q \), a boundary point, has three disks for each of the three 2-faces.

Restricted Delaunay complexes were considered in previous works because they become topologically equivalent (homeomorphic) when the sampled set is sufficiently dense. This topological argument follows by showing that once the sample is dense enough, it satisfies the topological ball property [58]. It turns out that even for PSCs, a similar result holds [29]. However, it is computationally very difficult to determine when the sample is sufficiently dense. To bypass this difficulty we sample the domain at a resolution specified by the user. Paired with the **DiskCondition** we certify that output mesh restricted to each manifold element is a manifold and when the resolution parameter is small enough, it is homeomorphic too. Empirically we observe that homeomorphism is achieved quite early in the refinement process.
5.2 Meshing PSCs

For any triangle \( t \in \text{Sk}^2 \mathcal{S}|_\sigma \), define \( \text{size}(t, \sigma) \) to be the maximum weighted distance between the vertices of \( t \) and points where dual Voronoi edge \( V_t \) intersects \( \sigma \). Notice that if all vertices of \( t \) are unweighted, the maximum weighted distance is just the maximum Euclidean distance. In addition to enforcing \textbf{DiskCondition}, we refine triangles who have \( \text{size}(t, \sigma) \) greater than user threshold \( \lambda \).

When we mesh volumes, we use the standard technique of inserting circumcenters of tetrahedra that have radius-edge ratio (denoted \( \rho() \)) greater than a threshold, \( \rho_0 \geq 1 \). If the insertion of the circumcenter threatens to delete any triangle in \( \text{Sk}^2 \mathcal{S}|_{\mathcal{D}_{\leq 1}} \), the circumcenter is not inserted. In this case we say that the triangle is \textit{encroached} by the circumcenter. Essentially, this strategy allows refining most of the tetrahedra except the ones near boundary.

5.2.1 Algorithm

\begin{verbatim}
1 DelPSC1(D, \lambda, \rho_0)
2    S ← Protection(D).
3    S ← Mesh2Complex(D, \lambda, S).
4    S ← Mesh3Complex(D, \rho_0, S).
5    return \bigcup_i \text{Sk}^i \mathcal{S}|_{\mathcal{D}_i}.
\end{verbatim}

Figure 5.4: DelPSC1 algorithm.

Figure 5.4 summarizes our algorithm. The Protection() phase generates a set of weighted points to protect \( \mathcal{D}_{\leq 1} \). Mesh2Complex() and Mesh3Complex() are described
in Figure 5.5. Note that after any point is inserted into the set $S$, Del $S$ and Vor $S$ are updated implicitly.

```
1 Mesh2Complex($D$, $\lambda$, $S$)
2     while ($|S|$ has increased) {
3         if ($\exists p \in S$ s.t. DiskCondition($p$) is violated) {
4             Let $t \in Umb_D(p)$ maximize size($t, \sigma$) over all $\sigma$ containing $p$.
5             Insert $x \in V_t|_{\sigma}$ that realizes size($t, \sigma$) into $S$.
6         } else if ($\exists (t, \sigma)$ where $t \in Skl^2 S|_{\sigma}$ s.t. size($t, \sigma$) > $\lambda$) {
7             Insert $x \in V_t|_{D}$ that realizes size($t, \sigma$) into $S$.
8         }
9     }
10    return $S$.
```

```
1 Mesh3Complex($D$, $\rho_0$, $S$)
2     while ($\exists (t, \sigma)$ where $t \in Skl^3 S|_{\sigma}$ s.t. $\rho(t) > \rho_0$ and $V_t$ does not encroach any triangle $t' \in Skl^2 S|_D$) {
3         Insert $x = V_t$ into $S$.
4     } //end while
5    return $S$.
```

Figure 5.5: Pseudocode for Mesh2Complex() and Mesh3Complex().

5.2.2 Protection Computations

To satisfy the protection properties we compute two quantities at the points where balls are centered.

First, we compute the $\omega$-deviation at a point $x \in \sigma$ defined as follows. If $\sigma \in D_{\leq 1}$, for $\omega > 0$, let $\sigma_{x,\omega} = \{ y \in \sigma : \angle n_{\sigma}(x), n_{\sigma}(y) = \omega \}$. If $\sigma \in D_2$, define $\sigma_{x,\omega}$ analogously but varying $y$ over the surface of $\sigma$. The distance between $x$ and $\sigma_{x,\omega}$ is the $\omega$-deviation radius of $x$ in $\sigma$. It is $\infty$ if $\sigma_{x,\omega} = \emptyset$. Let $d_x$ be the minimum of the $\omega$-deviation radius
of $x$ over all $\sigma$ containing $x$. By construction, \angle n_\sigma(x), n_\sigma(y) = \omega$ for some $y \in \sigma$ such that $\|x - y\| = d_x$.

Second, for any 1- or 2-face $\sigma$ containing $x$, we compute the tangential contact points between $\sigma$ and any sphere centered at $x$. Select the tangential contact point nearest to $x$ (over all 1- and 2-faces containing $x$). Let $d'_x$ be the distance between $x$ and this nearest tangential contact point.

It is not hard to prove that, for any $r < \min\{d_x, d'_x\}$, $B(x, r) \cap \sigma$ is a closed ball of dimension $\dim(\sigma)$. Also, since $r < d_x$, the normal deviation property 3(a) is satisfied. To satisfy property 1 and 2, we take a fraction of the minimum of $d_x$ and $d'_x$ to determine the size of the ball at $x$. Let $r_x = \frac{\omega}{8} \min\{d_x, d'_x\}$.

![Figure 5.6: Protection in action on Casting. We placed weighted vertices on all elements of $D_1$ which protect these elements when meshed.](image)

For each curve $\sigma$ with endpoints, say $u$ and $v$, we first compute the balls $B_u$ and $B_v$ with radii $r_u$ and $r_v$. Then, starting from, say $B_u$, we march along the curve placing the centers of the balls till we reach $B_v$. These centers can be chosen using a procedure described in [29].
We compute the intersection points $x_0 = B_v \cap \sigma$ and $x_1 = B_u \cap \sigma$. The protecting ball at $x_1$ is $B_{x_1} = B(x_1, r_{x_1})$. The protecting ball at $x_0$ is constructed last. We march from $B_{x_1}$ toward $x_0$ to construct more protecting balls. For $k \geq 2$, let $B_{x_{k-1}}$ be the last protecting ball placed and let $B_p$ be the last protecting ball placed before $B_{x_{k-1}}$. We compute the two intersection points between $\sigma$ and the boundary of $B(x_{k-1}, \frac{6}{5}r_{x_{k-1}})$. Among these two points let $x_k$ be the point such that $\angle px_{k-1}x_k > \pi/2$. One can show that $x_k$ is well-defined and $x_k$ lies between $x_{k-1}$ and $v$ along $\sigma$. Define

$$r_k = \max \left\{ \frac{1}{2} \|x_{k-1} - x_k\|, \min_{0 \leq j \leq k} r_{x_k}/8 + \|x_k - x_j\|/8 \right\}.$$

If $B(x_k, r_k) \cap B(x_0, r_{x_0}) = \emptyset$, the protecting ball at $x_k$ is

$$B_{x_k} = B(x_k, r_k).$$

Figure 5.7 shows an example of the construction of $B_{x_k}$. We force $r_k \geq \frac{1}{2} \|x_{k-1} - x_k\|$ so that $B_{x_k}$ overlaps significantly with $B_{x_{k-1}}$. This is desirable because the protecting balls are supposed to cover $\sigma$ in the end.

![Figure 5.7: The two dashed circles denote $B(x_{k-1}, \frac{6}{5}r_{k-1})$ and $B(x_0, r_{x_0})$. The bold circle denotes $B_{x_k}$.](image)

We continue to march toward $x_0$ and construct protecting balls until the candidate ball $B(x_m, r_m)$ that we want to put down overlaps with $B(x_0, r_{x_0})$. In this case, we
reject $x_m$ and $B(x_m, r_m)$ and compute the intersection points between $\sigma$ and the bisector plane of $x_{m-1}$ and $x_0$. Let $y_m$ be the intersection point that lies between $x_{m-1}$ and $x_0$ along $\sigma$. Finally, the protecting ball at $y_m$ is $B_{y_m} = B(y_m, R)$ and the protecting ball at $x_0$ is $B_{x_0} = B(x_0, R)$, where $R = \frac{2}{3} ||x_{m-1} - y_m|| = \frac{2}{3} ||y_m - x_0||$. It can be shown that the constructed balls satisfy all protection properties.

**Lemma 5.2.1.** The protecting balls computed by the above described procedure satisfy the protection properties.

### 5.3 Analysis

The analysis of DelPSC1 establishes two main facts: (1) the algorithm terminates, (2) at termination the output mesh satisfies properties T1-T3 as stated later. To show T1-T3, it will be necessary to prove that DelPSC1 maintains an admissible point set $S$ throughout its execution.

**Lemma 5.3.1.** DelPSC1 never attempts to insert a point in any protecting ball.

*Proof.* In Mesh2Complex(), points that are intersection of Voronoi edges and $|\mathcal{D}|$ are inserted. Since no three protecting balls intersect, all points on a Voronoi edge have positive distance from all vertices, weighted or not. This means no point of any Voronoi edge lies inside a protecting ball. Therefore, the inserted points in Mesh2Complex() must lie outside all protecting balls. For the same reason a circum-center inserted in Mesh3Complex() cannot lie in any of the protecting balls. \(\square\)

### 5.3.1 Termination

We apply the standard packing argument used to show termination of our Delaunay refinement. To do this, we must argue that there exists a lower bound between
the distance of each point inserted by DelPSC1. Since $\mathcal{D}$ is compact, we know that only finitely many points may be inserted. We use one technical lemma, whose proof appears in [28], to show this.

**Lemma 5.3.2.** Let $p \in S$ be a point on a 2-face $\sigma$. Let $\overline{\sigma}$ be the connected component in $V_p|\sigma$ containing $p$. There exists a constant $\lambda > 0$ so that following holds:

If some edge of $V_p$ intersects $\sigma$ and $\text{size}(t, \sigma) < \lambda$ for each triangle $t \in \text{Skl}^2 S|\mathcal{D}_2$ incident to $p$, then

1. there is no 2-face $\tau$ where $p \notin \tau$ and $\tau$ intersects a Voronoi edge in $V_p$.
2. $\overline{\sigma} = V_p \cap B \cap \sigma$ where $B = B(p, 2\lambda)$ if $p$ is unweighted and $B = B(p, 2\text{radius}(B_p) + 2\lambda)$ otherwise;
3. $\overline{\sigma}$ is a 2-disk;
4. any edge of $V_p$ intersects $\overline{\sigma}$ at most once;
5. any facet of $V_p$ intersects $\overline{\sigma}$ in an empty set or an open curve.

This result says that if dual Voronoi edges of all restricted triangles incident to a point $p$ in a 2-face $\sigma$ have nearby intersections with $\sigma$, the connected component of $\sigma$ containing $p$ within $V_p$ satisfies some nice properties. These properties allow one to argue that restricted triangles incident to $p$ will form a disk eventually. Given Lemma 5.3.2, we use the following two lemmas to argue termination.

**Lemma 5.3.3.** Mesh2Complex() terminates.

*Proof.* Mesh2Complex() inserts points for one of two reasons, either for a violation of the DiskCondition or when some triangle is larger than the sizing parameter $\lambda$. In
the latter case, this means only finitely many triangles which are larger than $\lambda$ can be used to approximate the surface patches; so eventually this second case will never occur.

Thus we need only to argue that \textbf{DiskCondition} is eventually satisfied. However, by Lemma 5.3.2 we know that there exists some $\lambda$ such that the restricted triangles incident to each point $p$ on a 2-face $\sigma$ form a topological disk (Figure 5.8). Also, this topological disk cannot include a point from a 2-face other than $\sigma$ since then the distance between $p$ and that point will be large enough to contradict the resolution level determined by $\lambda$.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.8.png}
\caption{Figure 5.8: Left: Within ball $B$, $V_p$ intersects $\sigma$ and $\tau$ both of which intersect some edge of $V_p$. This is not possible according to Lemma 5.3.2. Middle: Also not possible since there is another component of $\sigma$ within $B \cap V_p$ other than $\bar{\sigma}$. Right: Within $B$, $\sigma$ intersects $V_p$ in a topological disk. It is possible that there is a different component ($\tau$) which does not intersect any Voronoi edge and hence does not contribute any dual restricted triangle incident to $p$.}
\end{figure}

\textbf{Lemma 5.3.4.} \textit{Mesh3Complex()} terminates.
Proof. In Mesh3Complex() we refine any tetrahedron $t$ which has a radius-edge ratio greater than some constant $\rho_0 \geq 1$. Let $r_t$ and $\ell_t$ be the circumradius and shortest edge length of $t$. We insert the circumcenter of $t$, which happens to be $V_t$, to fix $t$. We know $V_t$ is a distance at least $r_t$ from all other points. Since $\rho(t) = r_t/\ell_t > \rho_0 \geq 1$ the insertion of $V_t$ introduces no edges in $\text{Del}S$ which are shorter than $\ell_t$. However, the radius of some elements in $\text{Del}S$ will shrink.

Let $\ell_0$ be the shortest edge length in $\text{Del}S$ at the start of Mesh3Complex(). By the above argument, no point will ever be inserted within a distance of $\ell_0$ of any point in $\text{Del}S$. Therefore, a packing argument shows that only finitely many points may be inserted. \qed

**Theorem 5.3.1.** DelPSC1 terminates.

**Proof.** Follows immediately from Lemmas 5.3.3 and 5.3.4. \qed

### 5.3.2 Topology Preservation

The output of DelPSC1 satisfies certain topological properties. Property T1 ensures feature preservation (Figure 5.9, Left). Property T2 ensures each manifold element is approximated with a manifold and incidence structure among them is preserved (Figure 5.9, Middle). Property T3 ensures topological equivalence between input and output when resolution parameter is sufficiently small (Figure 5.9, Right) and Figure 5.11).

(T1) For each $\sigma \in \mathcal{D}_1$, $\text{Sk}^1 S|_\sigma$ is homeomorphic to $\sigma$ and two vertices are joined by an edge in $\text{Sk}^1 S|_\sigma$ if and only if these two vertices are adjacent on $\sigma$.  

91
(T2) For $0 \leq i \leq 2$ and $\sigma \in \mathcal{D}_i$, $\text{Skl}^i S|_\sigma$ is a $i$-manifold with vertices only in $\sigma$. Further, $\text{Bd} \text{Skl}^i S|_\sigma = \text{Skl}^{i-1} S|_{\text{Bd} \sigma}$. For $i = 3$, the statement is true if the set $\text{Skl}^i S|_\sigma$ is not empty at the end of $\text{Mesh2Complex}()$.

(T3) There exists a $\lambda > 0$ so that the output mesh of $\text{DelPSC1}(\mathcal{D}, \lambda)$ is homeomorphic to $\mathcal{D}$. Further, this homeomorphism respects stratification with vertex restrictions, that is, for $0 \leq i \leq 3$, $\text{Skl}^i S|_\sigma$ is homeomorphic to $\sigma \in \mathcal{D}_i$ where $\text{Bd} \text{Skl}^i S|_\sigma = \text{Skl}^{i-1} S|_{\text{Bd} \sigma}$ and vertices of $\text{Skl}^i S|_\sigma$ lie in $\sigma$.

Figure 5.9: Left: Adjacent points on curves in $\mathcal{D}_1$ are joined by restricted edges. Middle: A surface patch is meshed with a manifold though topology is not fully recovered. Right: Topology is fully recovered.

The proof of T1 follows immediately from Lemma 5.1.1. One requires some non-trivial analysis to prove T2 which we skip here. To prove T3 we need a result of Edelsbrunner and Shah [58].

A CW-complex $\mathcal{R}$ is a collection of closed (topological) balls whose interiors are pairwise disjoint and whose boundaries are union of other closed balls in $\mathcal{R}$. A finite set $S \subset |\mathcal{D}|$ has the extended topological ball property for $\mathcal{D}$ if there is a CW-complex $\mathcal{R}$
Figure 5.10: $F$ is a Voronoi facet. Left: $F$ intersects a 2-face in a closed topological interval (1-ball) which is $b_F$. Here $b_F$ intersects $\text{Bd} F$ at two points, a 0-sphere. Right: $F$ intersects the 1-face in a single point which is $b_F$, and for $1 \leq i \leq 3$, $F \cap \sigma_i$ are closed topological 1-balls incident to $b_F$. Here $b_F \cap \text{Bd} F = \emptyset$, a $-1$-sphere.

with $|\mathcal{R}| = |\mathcal{D}|$ that satisfies the following conditions for each Voronoi face $F \in \text{Vor} S$ intersecting $|\mathcal{D}|$:

(C1) The restricted Voronoi face $F \cap |\mathcal{D}|$ is the underlying space of a CW-complex $\mathcal{R}' \subseteq \mathcal{R}$.

(C2) The closed balls in $\mathcal{R}'$ are incident to a unique closed ball $b_F \in \mathcal{R}'$.

(C3) If $b_F$ is a $j$-ball, then $b_F \cap \text{Bd} F$ is a $(j - 1)$-sphere.

(C4) Each $\ell$-ball in $\mathcal{R}'$, except $b_F$, intersects $\text{Bd} F$ in a $(\ell - 1)$-ball.

Figure 5.10 shows two examples of a Voronoi facet $F$ that satisfies the above conditions.

A result of Edelsbrunner and Shah [58] says that if $S$ has the extended topological ball property for $\mathcal{D}$, the underlying space of $\text{Del} S|_{\mathcal{D}}$ is homeomorphic to $|\mathcal{D}|$. Of course, to apply this result we would require a CW-complex with underlying space as
We see that when $\lambda$ is sufficiently small, $\text{Vor} S|\mathcal{D}$ provides such a CW-complex when our algorithm terminates. It can be shown that the following two properties P1, P2 imply Edelsbrunner-Shah conditions C1-C4. Let $F$ be a $k$-face of $\text{Vor} S$.

(P1) If $F$ intersects an element $\sigma \in \mathcal{D}_j \subseteq \mathcal{D}$, the intersection is a closed $(k+j-3)$-ball.

(P2) There is a unique lowest dimensional element $\sigma_F \in \mathcal{D}$ so that $F$ intersects $\sigma_F$ and only elements that are incident to $\sigma_F$.

One can show that when dual Voronoi edges of all restricted triangles intersect the surface patches within sufficiently small distance from their vertices (that is, $\text{size}(t, \sigma)$ is small), properties P1 and P2 hold. The refinement step for triangles in DelPSC1 achieves the required condition when $\lambda$ is sufficiently small. Also, when P1 and P2 hold, Del $S|\mathcal{D}$ equals $\bigcup_i \text{Skl}_i S|\mathcal{D}_i$, the output of DelPSC1.

**Theorem 5.3.2.** The output of DelPSC1 satisfies T1, T2, and T3.

### 5.4 Meshing Results

We have implemented DelPSC1 with the aid of the CGAL [23] library for maintaining a weighted Delaunay triangulation. With this implementation we have experimented on a variety of different shapes with varied levels of smoothness, including piecewise-linear (PLCs), piecewise-smooth (PSCs), and smooth shapes. Our examples incorporate both manifold and non-manifold shapes. Three of these examples also have sharp angles. The different datasets shown in the images are summarized in Table 5.1. We show the time to mesh each model on a Pentium 4 2.8 GHz machine with 2 GB of memory, given the input parameters $\rho_0 = 1.4$ and $\lambda$ as 10% of the minimum dimension of the bounding box.
The input to DelPSC1 is a polygonal mesh that represents a PSC. We first mark those edges which are non-manifold or have an inner dihedral angle less than a user-specified parameter and then collect them together to form the complex $D_1$. Using the marked edges in $D_1$, we group the input polygons into elements of $D_2$. An octree is built to bucket these input polygons for quick intersection checks with dual Voronoi edges. The next step is to create the protecting balls for elements of $D_{\leq 1}$ as described in Section 5.2.2. We finally pass all of this information to the Delaunay meshers described as Mesh2Complex() and Mesh3Complex().

We show a variety of the output (both surface and volume meshes) for each input models. Figures 5.12-5.14 show additional results. In each figure we show the input
model, the output surface mesh with the protected elements highlighted, and the volume mesh output (if it exists). In particular, the Fertility and Metaball models (Figures 5.11 and 5.1) are taken to be a smooth manifolds, so no input curves are protected. The Swirl and Hand models (Figure 5.14) both have no enclosed volumes, so they do not have a volume mesh associated with them.

Table 5.1: Input datasets. Output mesh sizes and time to mesh (not including protection times) are shown. Times are in seconds.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Smoothness</th>
<th>Manifold</th>
<th>Sharp</th>
<th>Time to Mesh</th>
<th># of vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serated</td>
<td>PLC</td>
<td>Yes</td>
<td>Yes</td>
<td>94.4</td>
<td>13047</td>
</tr>
<tr>
<td>Anchor</td>
<td>PSC</td>
<td>Yes</td>
<td>No</td>
<td>43.9</td>
<td>7939</td>
</tr>
<tr>
<td>Casting</td>
<td>PSC</td>
<td>Yes</td>
<td>No</td>
<td>170.9</td>
<td>19810</td>
</tr>
<tr>
<td>Guide</td>
<td>PSC</td>
<td>Yes</td>
<td>No</td>
<td>53.5</td>
<td>9492</td>
</tr>
<tr>
<td>Part</td>
<td>PSC</td>
<td>Yes</td>
<td>No</td>
<td>22.2</td>
<td>3026</td>
</tr>
<tr>
<td>Pin-Head</td>
<td>PSC</td>
<td>Yes</td>
<td>Yes</td>
<td>90.1</td>
<td>13958</td>
</tr>
<tr>
<td>Saturn</td>
<td>PSC</td>
<td>No</td>
<td>No</td>
<td>4.8</td>
<td>1340</td>
</tr>
<tr>
<td>Swirl</td>
<td>PSC</td>
<td>No</td>
<td>No</td>
<td>86.7</td>
<td>9288</td>
</tr>
<tr>
<td>Wedge</td>
<td>PSC</td>
<td>No</td>
<td>Yes</td>
<td>9.2</td>
<td>2980</td>
</tr>
<tr>
<td>Fertility</td>
<td>Smooth</td>
<td>Yes</td>
<td>No</td>
<td>57.7</td>
<td>9113</td>
</tr>
<tr>
<td>Metaball</td>
<td>Smooth</td>
<td>Yes</td>
<td>No</td>
<td>12.1</td>
<td>3288</td>
</tr>
<tr>
<td>Hand</td>
<td>Smooth</td>
<td>No</td>
<td>No</td>
<td>24.4</td>
<td>4872</td>
</tr>
</tbody>
</table>

Figure 5.12: Serated: surface (middle) and volume (right) mesh.
5.5 Discussion

We have presented a practical algorithm to mesh a wide variety of geometric domains with Delaunay refinement technique. The output mesh maintains a manifold property and captures the topology of the input with a sufficient level of refinement. An interesting aspect of the algorithm is that by using this strategy we can explicitly preserve input features in the output mesh.

A number of experimental results validate our claims. Our implementation can handle arbitrarily small input angles. When applied to volumes, the algorithm guarantees bounded radius-edge ratio for most of the tetrahedra except near boundary. It can be easily extended to guarantee bounded aspect ratio for most triangles except the ones near non-smooth elements. Furthermore, optimization based techniques can be used to improve qualities of the elements [6].

One of the main expenses of the algorithm involves computing the protecting balls in the protection step. We next look at an algorithm which eliminates the need for many of these primitives.
Figure 5.14: Four PSCs: Pin-Head, Guide, Saturn, and Swirl. The last two are non-manifold. In each figure we show the input (left), surface mesh (middle) and volume mesh (right, where present).
CHAPTER 6

ELIMINATING EXPENSIVE PREDICATES WHEN MESHING PSCS

We saw in Chapter 5 that Delaunay mesh generation of non-smooth domains is a difficult challenge. There are two problems we must overcome to mesh this domain; both are caused by the lack of global smoothness. First, the sampling theory developed for smooth surfaces breaks down for non-smooth surfaces. Secondly, small input angles possibly present at non-smooth regions pose problems for the termination of Delaunay refinement [108].

These two challenges require computational solutions that can involve expensive numeric primitives. In Chapter 5 we compensated for some of the expense by allowing the user to input a scale parameter $\lambda$ which specified the desired triangle size. However, feature size computations were still required to compute a set of protecting balls which cover the curved features in $D_1$. This chapter develops a sampling approach for curved features which is similar in spirit. Instead of computing an exact size for feature balls, we begin with an initial set of protecting balls that may be too large. The algorithm then naturally shrinks them as it meshes two dimensional features.

An important concern for our algorithm is that we would like to guarantee that even if the input scale is incorrect, the algorithm terminates and outputs a mesh
which approximates the input complex at a coarse level. We were able to prove this with a fixed set of protecting balls in DelPSC1, and we will again show it when we allow protecting balls to be resized during meshing. In addition, our theoretical framework allows us to show a similar guarantee on the topology of the output mesh. Specifically, we argue the output mesh always satisfies a manifold property and that when the input scale is small enough, the output mesh captures the input topology too.

We use the same disk condition to drive refinement, but upon a violation we respond differently since our set of protecting balls is not fixed throughout the algorithm. We saw in Chapter 5 that this condition alone allows an output PSC which can miss small features. Thus we again pair it with an input parameter $\lambda$ which now controls the maximum triangle size and protecting ball size. Our experiments will continue to indicate that the disk condition alone usually suffices to generate a mesh that is homeomorphic to the input.

To show the algorithm always terminates we first argue (with Lemma 6.1.2) that if the protecting balls are sufficiently small and sufficiently separated, then the disk condition holds when the restricted Delaunay triangles are sufficiently small. Therefore, the failure of the disk condition signals either the protecting balls do not satisfy separation properties (and hence are too large) or that the triangles are too large. Since the algorithm refines the larger of the two, we guarantee that neither a protecting ball nor a triangle gets arbitrarily small. In refining the largest first, we can ensures termination with a packing argument.

The approach in this chapter allows the refinement algorithm to determine the balls automatically instead of pre-computing them. The result is that nearly all of the
expensive feature computations are eliminated. We report experimental results for
our protection algorithm and meshing in Section 6.4. The code and a video explaining
the experimental results has also been released [49].

6.1 Protection and Refinement

Our new meshing algorithm, DelPSC2, again computes a set of balls protecting 1-
faces. Unlike in Chapter 5, these protecting balls are adjusted on the fly as refinement
proceeds. There are two main reasons for wanting to compute the protecting balls
this way. The first is to reduce the expense of computing feature sizes. Table 6.1
shows a comparison for the time to compute the initial set protecting balls using the
technique in Chapter ?? versus the one we use in this chapter.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Input size</th>
<th>DelPSC1 # of Balls</th>
<th>DelPSC2 # of Balls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anchor</td>
<td>1050</td>
<td>1.02</td>
<td>1162</td>
</tr>
<tr>
<td>Casting</td>
<td>10224</td>
<td>34.19</td>
<td>3872</td>
</tr>
<tr>
<td>Guide</td>
<td>952</td>
<td>0.92</td>
<td>1240</td>
</tr>
<tr>
<td>Hand</td>
<td>999999</td>
<td>107.75</td>
<td>970</td>
</tr>
<tr>
<td>Part</td>
<td>872</td>
<td>0.58</td>
<td>650</td>
</tr>
<tr>
<td>Pin-Head</td>
<td>74</td>
<td>0.06</td>
<td>381</td>
</tr>
<tr>
<td>Saturn</td>
<td>2100</td>
<td>0.70</td>
<td>518</td>
</tr>
<tr>
<td>Serated</td>
<td>34</td>
<td>0.23</td>
<td>1896</td>
</tr>
<tr>
<td>Swirl</td>
<td>2400</td>
<td>9.38</td>
<td>3295</td>
</tr>
<tr>
<td>Wedge</td>
<td>50</td>
<td>0.02</td>
<td>331</td>
</tr>
</tbody>
</table>

Table 6.1: Protection time comparison for PSC datasets from Chapter 5. Input size
is measured in number of triangles and protection times are in seconds. Fertility and
Metaball are not shown because they have no curves to protect.

In particular, Table 6.1 shows that Hand stands out as requiring an exorbitant
amount of time to compute a very small number of protecting balls. Estimating
feature sizes requires a global view of the input model. So for input models that have a large number of triangles, such as Hand and Casting, protection requires significantly larger time than the others. For comparison, Serated has only 34 triangles, so needed less than a second to compute 1896 protecting balls. When computing a similar number of protecting balls using the approach in DelPSC2, we generally take about the same amount of time for small inputs, but for large ones we see there a significant savings in protection time.

More importantly though, computing feature size exactly is a challenging problem. Even if these computations were not expensive, they can easily suffer from numerical precision errors. By avoiding features size computations, we build an algorithm that is ultimately more robust to different kinds of inputs. And for certain kinds of data, for example the ones discussed in Chapter 7, computing an appropriate measure of feature size may be impossible.

Our ball protection requires two subroutines, one for covering a 1-face with balls and a second for refining the ball sets when they are too large. We show these routines can build a set of balls which satisfy separation criteria (conditions C1-C3) that allow us to prove termination. Our final algorithm refines both the protecting balls and the triangulation simultaneously either to satisfy a disk condition or to achieve a refinement level dictated by an input scale parameter.

6.1.1 Covering 1-faces

Let $\sigma(x, y)$ denote the curve segment oriented from $x$ to $y$ on any 1-face $\sigma$. In this notation $\sigma = \sigma(u, v)$ where $\sigma$ is oriented from the end point $u$ to the other end point $v$. Let $b = B(c, r)$ be a ball with $c \in \sigma(x, y)$. The intersection $b \cap \sigma$ is a set of
curve segments. Among them the curve segment containing \( c \) is called the \textit{segment of b in }\sigma, \text{seg}_\sigma(b), \text{ see Figure 6.1. We call two balls } b \text{ and } b' \text{ and their corresponding weighted vertices adjacent if their centers are adjacent on a 1-face } \sigma. \text{ We use } d(x, y) \text{ to denote the Euclidean distance between two points } x, y \text{ and use } d_\sigma(x, y) \text{ to denote the length of a curve segment } \sigma(x, y). \text{ Let } b_0, b_1, \ldots, b_k \text{ be a set of balls that protect } \sigma \text{ where } b_i = B(c_i, r_i) \text{ with } c_i \in \sigma. \text{ We require that the balls satisfy the following conditions:}

1. \( b_0 \) and \( b_k \) are centered at \( u \) and \( v \) respectively. These will be called the \textit{vertex balls}.

2. \( \sigma \) is covered by the balls, that is, \( \sigma \subseteq \cup_i \text{seg}_\sigma(b_i) \) and any two adjacent balls \( b = B(c, r) \) and \( b' = B(c', r') \) intersect deeply, that is, \( d(c, c') \leq r + \frac{6r'}{7} \) where \( r' \leq r \).

3. No point in \( \text{seg}_\sigma(b_i) \) is contained in a ball non-adjacent to \( b_i \).

Notice that the choice of the constant \( \frac{6}{7} \) in C2 is a little arbitrary. We need only a factor of \( r' \) in the expression and a follow-up analysis with other constants are also possible. It is worthwhile to note that one of the consequences of conditions C1-C3 is the result of Lemma 6.1.1 that will imply R2 in Lemma 6.1.2.

Recall that we will maintain a point set \( S \subset D \) with the following properties throughout the algorithm: all points in \( S \) except those in \( D_{\leq 1} \) are unweighted and no unweighted point has a negative weighted distance to any other point. This means each unweighted point \( p \in S \) has \( V_p \) non-empty. We call such a point set \textit{admissible}.
Figure 6.1: Covering 1-faces. Left: A 1-face $\sigma$ between $u$ and $v$ is being protected. The ball $b$ shown with solid boundary has $\text{seg}_\sigma(b)$ as the curve segment between $x_1$ and $x_2$. The balls satisfy C1 and C2 but intersect arbitrarily. Right: Balls are refined and they start satisfying separation properties C1-C3.

**Lemma 6.1.1.** Let $S$ be an admissible point set satisfying conditions C1-C3. Let $p$ and $q$ be adjacent weighted vertices on a 1-face $\sigma$. $V_{pq}$ is the only Voronoi facet in $\text{Vor}S$ that intersects $\sigma(p,q)$.

*Proof.* Let $z$ be any point in $\sigma(p,q)$. Let $b_p$ and $b_q$ be the balls centered at $p$ and $q$ respectively. The point $z$ is contained in $b_p \cup b_q$. Due to property C3, $z$ being a point in $\sigma(p,q)$ cannot lie inside any ball other than $b_p$ and $b_q$. We remark that $b_p$ and $b_q$ may have a common intersection with another ball, but $z$ cannot be contained in that ball. Therefore, $z$ cannot lie on any Voronoi facet partly defined by a point other than $p$ and $q$. However, $\sigma(p,q)$ has to intersect at least one Voronoi facet since $p$ and $q$ lie in two different Voronoi cells. Therefore, the only Voronoi facet which intersects $\sigma(p,q)$ is $V_{pq}$.

Admissible point sets give us a topological condition. We also need a geometric one for our proofs. We say $S$ has the $\lambda$-weight property if each point in $S$ has a weight at most $\lambda \geq 0$. We say $S$ has the $\lambda$-size property if $\text{size}(t, \sigma) \leq \lambda$ for each triangle
$t \in \text{Sk}^2 S|_\sigma$ and $S$ has the $\lambda$-weight property.

The following lemma provides the key connection between our ball sizing algorithm and our refinement algorithm. See [48] for a proof sketch. It serves a similar purpose to Lemma 5.3.2 in Chapter 5, but is modified to handle the differences between the two algorithms.

**Lemma 6.1.2.** Let $S \subset \mathcal{D}$ be an admissible point set and $p \in S$ be a point on a 2-face $\sigma$. Let $\sigma_p \subset \sigma$ be the set of all connected components in $V_p|_\sigma$ that intersect a Voronoi edge. There exists a constant $\lambda > 0$ so that hypotheses H1-H3:

(H1) $\sigma_p$ is not empty.

(H2) $S$ satisfies $\lambda$-size property.

(H3) Weighted points in $S$ satisfy C1-C3.

imply results R1 and R2:

(R1) $\sigma_p$ is a 2-disk where any edge of $V_p$ intersects $\sigma_p$ at most once and any facet of $V_p$ intersects $\sigma_p$ in an empty set or an open curve.

(R2) If $p \in \text{Bd} \sigma$, at least two Voronoi facets of $V_p$ intersect $\text{Bd} \sigma$, each intersecting one of the curve segments between $p$ and its adjacent weighted points (possibly two) in $\text{Bd} \sigma$.

Interpreted in terms of the Delaunay triangulation, the conclusion of the above lemma implies that the triangles incident to $p$ and restricted with respect to $\sigma$ form a topological disk around $p$. This disk has $p$ at the boundary if and only if $p$ is in
Bd $\sigma$. Furthermore, if $p$ is in Bd $\sigma$, it is connected to its two adjacent weighted points in Bd $\sigma$ on this disk. Our disk condition is formulated with these properties.

We will see that H1 holds if H2 holds. Therefore, the conclusion of Lemma 6.1.2 fails only if either there is a protecting ball with radius more than $\lambda$, or there is a triangle $t \in \text{Sk}^2 S|\sigma$ for which size$(t, \sigma) > \lambda$ for some $\lambda > 0$. However, since we do not know which of the above two cases has happened, we take a conservative approach. We compute the maximum radius $r_{\text{max}}$ of all protecting balls and also compute the maximum $d_{\text{max}} = \text{size}(t, \sigma)$ over all $t$ and $\sigma$. Let $x$ be the point of intersection of a Voronoi edge with $\mathcal{D}$ which realizes $d_{\text{max}}$. If $r_{\text{max}} > d_{\text{max}}$ we refine the largest protecting ball. Otherwise, we insert $x$. In the first case we are ensured that we are refining balls of size larger than a fixed positive constant. In the second case, we are inserting a point in a compact domain with a positive lower bound on its distances to every other points. Termination by a packing argument follows.

For the above algorithm to work, it is important that the balls satisfy C1-C3 when they are sufficiently small. It turns out that it is difficult to maintain the condition C3 at early phases when the balls are relatively large. We replace C3 by the following two conditions that are maintained by the ball refinement algorithm. These conditions imply C3 when the balls are small enough. For a 1-face $\sigma(u, v)$ covered by balls $b_0, b_1, \ldots, b_k$ these conditions are:

(C3.a) Let $b = B(c, r)$ be any ball in $\{b_0, b_1, \ldots, b_k\}$. For an adjacent ball $b' = B(c', r')$, if $c'$ is contained in $\text{seg}_\sigma(b)$, then $d_\sigma(c, c') \geq \frac{8}{7}r$.

(C3.b) Any two balls centered in different 1-faces do not intersect.
Lemma 6.1.3. There exists a $\lambda > 0$ so that, if all protecting balls are smaller than $\lambda$, $C3.a$ and $C3.b$ imply $C3$.

Proof. First consider any two non-adjacent balls $b_1$ and $b_2$ with centers $c_1$ and $c_2$ respectively on a curve $\sigma$. It follows from the differentiability of $\sigma$ that there exists $\lambda > 0$ so that any ball of size smaller than $\lambda$ intersects $\sigma$ in a single segment. Assuming that each $b_i$ has a radius smaller than $\lambda$, we have $b_i \cap \sigma = \text{seg}_\sigma(b_i)$. We claim that no point in $\text{seg}_\sigma(b_2)$ can lie in $\text{seg}_\sigma(b_1)$ when $\lambda$ is sufficiently small. If there were such a point there would exist a ball $b_3$ adjacent to $b_1$ whose center $c_3$ would lie in $\text{seg}_\sigma(b_1)$. But we know that the length $d_\sigma(c_1, c_3)$ is at least $\frac{8}{7}r_1$ by property $C3.a$. Making $\lambda$ sufficiently small, $d(c_1, c_3)$ can be set arbitrarily close to $\frac{8}{7}r_1$ contradicting the fact that $c_3$ is contained in $\text{seg}_\sigma(b_1)$. Therefore, we can claim that the curve segments of two non-adjacent balls cannot intersect if they are centered on a same 1-face $\sigma$ (notice that the balls may intersect otherwise). The lemma follows since balls centered on different 1-faces are not allowed to intersect by $C3.b$. 

6.1.2 Ball Refinement

The ball refinement routine simply removes a ball and covers the curve segment between the centers of its adjacent balls with balls of smaller radii. Therefore, we encounter the generic situation where a curve segment $\sigma(x, y)$ needs to be covered by protecting balls whose radii are determined by a given parameter $\alpha > 0$. The points $x$ and $y$ are the right and left end points of some segments, say $\text{seg}_\sigma(b_0)$ and $\text{seg}_\sigma(b_k)$ respectively, see Figure 6.2. We call this routine Cover().

We proceed from $x$ toward $y$ along the curve while computing the balls that satisfy conditions C1, C2, and C3.a. Condition C3.b is taken care of by another
routine called Separate(). As we walk from $x$ to $y$, each step places a new ball of radius $\alpha$ that intersect deeply with the previous ball while covering a new piece of the curve. When we reach $y$, we place a ball that intersects deeply with both the endpoint ball and the previous one in the march.

More specifically, suppose that $b_i = B(c_i, r_i)$ is already computed. Let $\sigma(x_i, y_i) = \text{seg}_\sigma(b_i)$. We compute a small ball $\beta_{i+1} = B(y_i, \alpha/3)$ that aids the computation of $b_{i+1}$, see Figure 6.2. The aiding ball helps compute the next ball in the march so that its center is not contained in the previous ball. We use the right end point, say $z_{i+1}$, of the segment $\text{seg}_\sigma(\beta_{i+1})$ covered by the aiding ball to place the center of the next ball $b_{i+1}$.

We will eventually encounter one of two situations near the end: either $z_{i+1}$ extends past $\sigma(x, y)$ or $\text{seg}_\sigma(b_{i+1})$ contains $y$. These situations are shown in the left and right images Figure 6.2, respectively. In the first case $b_k$ may violate C3.a and in the second case $b_{i+1}$ may not intersect $b_k$ deeply violating C2. Cover() terminates differently depending on which. If $z_{i+1} \in \text{seg}_\sigma(b_k)$, we throw away $z_{i+1}$ and take $b_{i+1}$ as $\beta_{i+1}$.

![Figure 6.2: Curve segment between $x$ and $y$ is being covered. Aiding balls are shown with solid boundaries. Notice how the centers of $b_1$ and $b_2$ are placed with the aiding balls. One the left, the final ball is computed by enlarging the aiding ball. On the right, we show the other case where the final ball $b_{i+1}$ is enlarged.](image-url)
enlarged concentrically to a radius $\frac{2\alpha}{3}$. In the other case when $\text{seg}_\sigma(b_{i+1})$ contains $y$, we enlarge $b_{i+1}$ to a radius of $\frac{7\alpha}{6}$.

**Lemma 6.1.4.** Cover() maintains C1, C2, and C3.a.

**Proof.** Cover() affects only the sequence of balls $b_0, b_1, \ldots, b_k$ as described. By construction the new balls cover entire curve segment $\sigma(x, y)$ and vertex balls are not changed. So, C1 and the first part of C2 are satisfied. To prove the second part consider a ball $b_i = B(c_i, r_i)$. We have following cases for different balls in the sequence:

**Case 1 ($i = 0$):** We need to examine the effect of the new ball $b_1$ onto $b_0$. First notice that $\text{seg}_\sigma(b_0)$ cannot contain $c_1$ by construction. Therefore, C3.a holds trivially.

If $b_1$ is not the last ball created by Cover(), the radius of $b_1$ is $\alpha$ and $d(c_0, c_1) < r_0 + \alpha/3 < r_0 + r_1/3$ where $r_1 = \alpha < r_0$ by construction. This satisfies C2. If $b_1$ is the final ball, then two cases occur. If $b_1$ is the enlarged aiding ball $\beta_1$, its radius is $2\alpha/3$. Then, $d(c_0, c_1) = r_0 < r_0 + r_1/3$. This satisfies C2. In the case where $b_1$ is not the enlarged aiding ball, its radius is $\frac{7}{6}\alpha$. We can similarly argue that $d(c_0, c_1) < r_0 + \alpha/3 = r_0 + \frac{2}{3}r_1$. So, $b_0$ satisfies C2.

**Case 2 ($i = k$):** In this case the center $c_{k-1}$ of $b_{k-1}$ cannot lie in $\text{seg}_\sigma(b_k)$ by construction. So, C3.a holds trivially. The ball $b_{k-1}$ is necessarily the last ball created.

If $b_{k-1}$ is enlarged $\beta_{k-1}$, $d(c_{k-1}, c_k) < r_k + \alpha/3 = r_k + r_{k-1}/2$ satisfying C2. In the case where $b_{k-1}$ is not the enlarged aiding ball, its radius is $\frac{7}{6}\alpha$. Then, $d(c_k, c_{k-1}) < r_k + \alpha = r_k + \frac{6}{7}r_{k-1}$ satisfying C2.

**Case 3 ($i \neq 0$ and $i \neq k$):** If $b_i$ is adjacent to $b_0$ and $b_k$, it satisfies C2 by arguments in previous two cases. Also, $b_i$ is not large enough to contain $c_0$ or $c_k$. Hence C3.a also holds. Now consider the case where $b_i$ is adjacent to another ball $b$ where $b \neq b_0$ and $b \neq b_k$. If neither $b_i$ nor $b$ is created by end game, we have $d(c, c_i) \leq r_i + \alpha/3 = r_i + r_i/3$.
satisfying C2. Also, in this case \( d_x(c, c_i) \geq \frac{4}{3} \alpha = \frac{4}{3} r_i \) which satisfies C3.a. We are left with the case when either of \( b \) and \( b_i \) is final ball in the sequence. With similar arguments, one can check that C2 and C3.a hold in these cases too.

Cover() does not necessarily satisfy C3.b. We use the routine Separate() to enforce C3.b on a set of balls \( B \). This routine calls RefineBall() which removes the ball \( b \) and replaces it with smaller balls. Both routines are shown in Figure 6.3.

**Lemma 6.1.5.** RefineBall() terminates, and maintains C1, C2, C3.a, and C3.b.

**Proof.** Observe that RefineBall() makes recursive calls to itself and through Separate(). Consider the tree of ball refinements rooted at \( b \) made by these recursive calls. An internal node \( b' \) in the tree represents a ball \( b' \) that is refined into smaller balls (children).

Notice that RefineBall() may refine at most one vertex ball which is necessarily the root \( b \). Since all vertex balls are kept fixed after that, two balls centered at two different 1-faces intersect only if the larger ball has a radius more than a fixed positive constant \( \delta > 0 \). Hence an internal node is refined only if it has a radius more than \( \delta > 0 \).

The children of a node \( b' \) are created by Cover() which, by construction, creates only finitely many balls with radius at most \( (1/4 \times 7/6) = 7/24 \)th the radius of \( b' \). The height of the refinement tree is finite since any path from the root to a leaf has internal nodes with radius larger than \( \delta > 0 \) and each level decreases the radius by a factor \( 7/24 \) or less. Also, this tree has finitely many children for all nodes. Therefore, the tree is finite implying that RefineBall() terminates.
Separate($\mathcal{B}$)

while ($\exists$ intersecting balls $b, b' \in \mathcal{B}$ with $c, c'$ in different 1-faces) {
    if ($r \geq r'$) {
        $\mathcal{B} \leftarrow \mathcal{B} \setminus \{b\} \cup \text{RefineBall}(b)$.
    } else {
        $\mathcal{B} \leftarrow \mathcal{B} \setminus \{b'\} \cup \text{RefineBall}(b')$.
    }
}

//end while

return $\mathcal{B}$.

RefineBall($b$)

if ($b$ covers $\sigma \in \mathcal{D}_1$) {
    Let $r_\sigma$ be the minimum radius of balls adjacent to $b$ on $\sigma$ and $b$.
}

if ($b = B(c, r)$ is a vertex ball) {
    Shrink $b$ to $b' = B(c, r/2)$.
    for each $\sigma$ covered by $b$ {
        Let $b_\sigma$ be the adjacent ball to $b$ on $\sigma$.
        Compute $\mathcal{B}_\sigma \leftarrow \text{RefineBall}(b_\sigma)$.
    }
    return Separate($\cup_{\sigma} \mathcal{B}_\sigma$).
} else {
    Let $b_1$ and $b_2$ be adjacent to $b$.
    Let $\sigma(x, y)$ be the segment between $\text{seg}_\sigma(b_1)$ and $\text{seg}_\sigma(b_2)$.
    Remove $b$.
    return Cover($x, y, r_\sigma/4$).
}

Figure 6.3: Pseudocode for Separate() and RefineBall().

To show RefineBall() maintains C1,C2,C3.a, and C3.b, assume that C1,C2, and C3.a hold before calling RefineBall(). It creates new balls by calling Cover() which satisfies C1,C2, and C3.a (Lemma 6.1.4). If a ball does not satisfy C3.b (may happen
only after a vertex ball is refined), it refines it by calling Separate(). The claim follows. □

6.2 Meshing Algorithm

The algorithm for meshing \( \mathcal{D} \) first protects the 1-faces with Protect() where the input scale parameter \( \lambda \) is the upper limit for the radii of the protecting balls. Pseudocode for Protect() is shown in Figure 6.4.

```
Protect(\mathcal{D}, \lambda)
for each vertex \( v \in \mathcal{D}_0 \) {
    Let \( r_v \) be \( \frac{1}{3} \)rd the distance of \( v \) to any other vertex in \( \mathcal{D}_0 \).
    \( \mathcal{B} \leftarrow \{ B(v, r_v) \} \cup \mathcal{B} \).
}
for each \( \sigma \in \mathcal{D}_1 \) {
    Let \( u \) and \( v \) be the end points of \( \sigma \).
    Let \( u_x = \text{seg}_\sigma(B_u) \) and \( y_v = \text{seg}_\sigma(B_v) \).
    Let \( \alpha \leftarrow \min\{r_u, r_v\} \).
    \( \mathcal{B} \leftarrow \mathcal{B} \cup \text{Cover}(x, y, \alpha) \).
}
while (\( \exists \) \( b \in \mathcal{B} \) with radius greater than \( \lambda \)) {
    Compute \( \mathcal{B} \leftarrow \mathcal{B} \setminus \{b\} \cup \text{RefineBall}(b) \).
}
return Separate(\mathcal{B}).
```

Figure 6.4: Pseudocode for Protect().

Lemma 6.2.1. Protect() terminates with balls satisfying \( C1, C2, C3.a, \) and \( C3.b \).

Proof. Observe that at the end of the second loop, Protect() creates a set \( \mathcal{B} \) of finitely many balls (most likely quite large). After that, we represent the refinements with
a refinement tree for each ball \( b \in B \). Each node in these trees represent a call to 
\text{RefineBall}() and its children represent the finitely many balls created as output of 
the call (Lemma 6.1.5).

When we refine for size (the third loop) a ball is refined only if its radius is more 
than \( \lambda > 0 \). When we call \text{Separate()} if a vertex ball, say \( b' = B(u, r) \), is refined, 
its radius cannot be smaller than half of the distance of \( u \) from all 1-faces that do 
not contain \( u \). This is because \( b' \) has to intersect a smaller ball centered on such a 
1-face (this constraint is built into \text{Separate}()). It follows that \( r \) is larger than a fixed 
positive constant \( \lambda_1 > 0 \). If the radii of all vertex balls are larger than \( \lambda_1 \), two balls 
centered at two different 1-faces intersect only if the larger ball has a radius more 
than a fixed positive constant \( \lambda_2 > 0 \).

Thus the argument in the proof of Lemma 6.1.5 applies to argue that the tree of 
refinement for each ball \( b \in B \) is finite. Hence \text{Protect()} terminates. At termination 
it must satisfy C1,C2,C3.a, and C3.b since it refines balls with \text{Cover()} and calls 
\text{Separate()} to enforce C3.b.

After protecting \( D_1 \) with \text{Protect()}, refinement of \( D_2 \) begins. In this phase we run 
Delaunay refinement with the \textbf{DiskCondition} used in Chapter 5. See Figure 5.3 for 
more explanations. Once the restricted Delaunay triangles are collected, the above 
checks are only combinatorial. One may notice that D1 and D2 are dual to R1 and R2 
of Lemma 6.1.2. We assume that as we insert points, weighted or unweighted, Vor \( S \) 
and Del \( S \) get updated appropriately. The pseudocode for the algorithm, \text{DelPSC2}, is 
shown in Figure 6.5.

Notice that when checking the disk condition we refine either a ball or a triangle 
if D1 or D2 is violated. However, for D3 or D4 violations we only refine a triangle.

113
DelPSC2(\(D, \lambda\))

//Protection
\[ B \leftarrow \text{Protect}(D, \lambda). \text{ Construct the weighted point set } S \text{ from } B. \]

//Mesh2Complex

while (\(|S|\) has increased) {

Let \( t \in \text{Skl}^2 S_\sigma \) maximize size\((t, \sigma)\) over all \( t \) and \( \sigma \).

Let \( d_1 \) be this maximum, realized by \( x \in V_t|_\sigma \).

Let \( d_2 \) be the maximum radius of all vertex balls realized by ball \( b \).

if (\( \exists p \in S \) s.t. \( D_1 \) or \( D_2 \) of \( \text{DiskCondition}(p) \) is violated) {

if (\( d_1 \geq d_2 \)) {

Insert \( x \) into \( S \).

} else {

Compute \( B \leftarrow B \setminus \{b\} \cup \text{RefineBall}(b) \).

}

} else if (\( \exists p \in S \) s.t. \( D_3 \) or \( D_4 \) of \( \text{DiskCondition}(p) \) is violated) {

Insert \( x \) into \( S \).

} else if (\( d_1 > \lambda \)) {

Insert \( x \) into \( S \).

}

} //end while

return \( \bigcup_i \text{Skil}^i S_D \).

Figure 6.5: Pseudocode for DelPSC2.

This is important because \( D_3 \) and \( D_4 \) are not covered by Lemma 6.1.2 and may be violated no matter how small the balls are. We argue separately for \( D_3 \) and \( D_4 \) in the termination proof. If the disk condition is satisfied we refine triangles to reach the refinement level of the input scale.

An important property of DelPSC2 is that it never inserts unweighted points inside any protecting ball. If \( x \) is inserted because of a violation of \( D_1 \) or \( D_2 \) and lies in a protecting ball \( b = B(q, r) \), its weighted distance to \( q \) would be non-positive. Its
weighted distance to its nearest point in $S$ would even be smaller. Since the largest ball has a positive radius, $x$ would not be inserted (we would call RefineBall() instead). If $x$ is inserted from a D3 or D4 violation, the point $p$ is connected to a point $q$ where $q \notin \sigma$. Then, $p$ and $q$ have a positive weighted distance since no two balls from two different 2-faces intersect. Finally, since $\lambda > 0$, any point $x$ inserted because a triangle does not meet the input scale will be of positive distance from its Voronoi neighbors, and thus outside of every protecting ball. Therefore, point $x$ has a positive weighted distance from $p$ and hence cannot be inside a ball.

### 6.2.1 Termination

We need the following result which says that when a sufficiently small scale is reached, each Voronoi cell $V_p$ contains at least one Voronoi edge intersecting $\sigma$ if $p \in \sigma \in \mathcal{D}_2$.

**Lemma 6.2.2.** There exists a $\lambda > 0$ so that if $S$ satisfies the $\lambda$-size property, then $\sigma_p$ is non-empty for any $p \in \sigma$

**Proof.** Suppose that $\sigma_p$ is empty, that is, no Voronoi edge of $V_p$ intersects $\sigma$. By our assumption Bd $\sigma$ is non-empty and let $q \in \sigma$ be a weighted point in Bd $\sigma$. Because of Lemma 6.1.1 an edge of $V_q$ has to intersect $\sigma$. Consider walking on a path in $\sigma$ from $p$ to $q$. Let $p = p_0, p_1, \ldots, p_k = q$ be the sequence of vertices whose Voronoi cells are encountered along this walk. Since no edge of $V_p$ intersects $\sigma$ and some edge of $V_q$ intersects $\sigma$, there exists two consecutive vertices $p_i$ and $p_{i+1}$ in this sequence so that no edge of $V_{p_i}$ intersects $\sigma$ whereas some edge of $V_{p_{i+1}}$ does intersect $\sigma$. By Lemma 6.1.2 we can claim that $\sigma_{p_{i+1}}$ is a disk. A boundary cycle of $\sigma_{p_i}$ overlaps
with the boundary of \( \sigma_{p_i+1} \). This is impossible as the curves on the boundary of \( \sigma_{p_i+1} \) intersect Voronoi edges whereas those on the boundary of \( \sigma_{p_i} \) do not.

**Theorem 6.2.1.** DelPSC2 terminates.

**Proof.** Consider a vertex \( p \) on a 2-face \( \sigma \). The conclusion of Lemma 6.1.2 implies D1 and D2 of **DiskCondition**. Therefore, if it does not hold for \( p \), at least one of the premises of Lemma 6.1.2 does not hold.

(H1) If \( \sigma_p \) is empty, then according to Lemma 6.2.2 \( S \) does not satisfy the \( \lambda_1 \)-size property for some \( \lambda_1 > 0 \). (H2) \( S \) does not satisfy the \( \lambda_2 \)-property for some \( \lambda_2 > 0 \). (H3) Protecting balls do not satisfy C1-C3. But, when the disk condition is checked, the protecting balls satisfy condition C1,C2,C3.a, and C3.b due to Lemma 6.1.5. Therefore, if H3 has failed, at least one ball has a radius more than \( \lambda_3 \) where \( \lambda_3 \) satisfies Lemma 6.1.3.

Let \( d_1 \) be the maximum size\((t, \sigma)\) over all \( t \) and \( \sigma \) and \( x \) be the point realizing this distance. Let \( d_2 \) be the maximum radius of all protecting balls. The argument in the previous paragraph implies \( \max\{d_1, d_2\} \geq \delta = \min\{\lambda_1, \lambda_2, \lambda_3\} \). In DelPSC2 the point \( x \) is inserted if \( d_1 \geq d_2 \). Otherwise, the ball with radius \( d_2 \geq \delta \) is refined.

The entire ball refinement can be represented with a tree as in the proof of Lemma 6.2.1 where a ball is refined only if its radius is at least a fixed positive constant. The argument for Lemma 6.2.1 still holds to claim that only finitely many balls are refined altogether. Therefore, the algorithm cannot refine balls forever. This also implies that the minimum size of the balls remains larger than a fixed positive constant, say \( \xi > 0 \).
Now we argue that each point inserted by the algorithm maintains a lower bound on its distance to all other points. Then, a standard packing argument implies termination. If a point \( x \) is inserted because of violation of D1 or D2, it maintains a weighted distance at least \( \delta > 0 \) If D3 or D4 is violated, the weighted distance of \( x \) from \( p \) is at least half the weighted distance between \( p \) and a point \( q \) where either \( p \) and \( q \) are non-adjacent points in \( \mathcal{D}_1 \) or \( q \) lies on a different 2-face. Since protecting balls have a minimum size \( \xi > 0 \) and they intersect deeply C2, the weighted distance between \( p \) and \( q \) is larger than a fixed positive constant. Hence, \( x \) has a distance more than a fixed positive constant from all other points. When refining for the input scale, points are inserted only if its weighted distance is at least \( \lambda > 0 \) from all other points.

\[ \square \]

### 6.3 Proof of Guarantees

**Guarantees:** The analysis of the algorithm establishes two main facts: (1) the algorithm terminates, (2) at termination the output mesh \( M \) of DelPSC2 has guarantees G1 and G2:

(G1) For each \( \sigma \in \mathcal{D}_2 \), \( \text{Skl}^2S|_\sigma \) is a 2-manifold with vertices only in \( \sigma \). Further, \( \text{Bd Skl}^2S|_\sigma \) is homeomorphic to \( \text{Bd} \sigma \) with vertices only in \( \text{Bd} \sigma \).

(G2) There exists a \( \lambda > 0 \) so that \( M \) is homeomorphic to \( |\mathcal{D}| \). Further, this homeomorphism respects stratification with vertex restrictions, that is, for \( 0 \leq i \leq 2 \), \( \text{Skl}^i S|_\sigma \) is homeomorphic to \( \sigma \in \mathcal{D}_i \) where \( \text{Bd Skl}^i S|_\sigma = \text{Skl}^{i-1} S|_{\text{Bd} \sigma} \) and vertices of \( \text{Sk}^i S|_\sigma \) lie in \( \sigma \).

**Theorem 6.3.1.** \( M \) has guarantee G1.
Proof. At the end of the Mesh2Complex() step of DelPSC2, the disk condition ensures that Skl² S|σ is a simplicial complex where each vertex v belongs to σ and has a 2-disk as its star. It follows from a result in PL topology that Skl² S|σ is a 2-manifold when DelPSC2 terminates.

The boundary of Skl² S|σ has all weighted vertices in Bd σ. Each such point p is connected to its adjacent vertices in Bd σ by the disk condition. Therefore, the boundary of Skl² S|σ consists of edges that connect adjacent vertices in Bd σ and hence this boundary is homeomorphic to Bd σ. □

To prove G2 we use a result of Edelsbrunner and Shah [58] about the extended topological ball property (TBP). It can be shown that the following two properties P1 and P2 imply the extended TBP [29]. Therefore, according to the Edelsbrunner-Shah [58] result, the underlying space of Del S|D is homeomorphic to the |D| if P1 and P2 hold. Let F be a k-face of Vor S where S is the output vertex set.

(P1) If F intersects an element σ ∈ D_j ⊆ D, the intersection is a closed (k+j−3)-ball.

(P2) There is a unique lowest dimensional element σ_F ∈ D so that F intersects σ_F and only elements that are incident to σ_F.

Lemma 6.1.2 almost provides condition P1 except for the case that V_p may intersect a patch τ where p ∉ τ. Lemma 6.3.1 establishes that this is not possible. Lemma 6.3.2 gives P2. Proofs of both of them appear in the original paper [48].

**Lemma 6.3.1.** There exists a constant λ > 0 so that if S has the λ-size property, then for each point p, V_p|D = ∪_{σ_p} σ_p.
Lemma 6.3.2. There exists a $\lambda > 0$ so that if $S$ has the $\lambda$-size property, then the following holds. Let $F$ be a $k$-face in Vor $S$. There is an element $\sigma_F \in \mathcal{D}$ so that $F$ intersects $\sigma_F$ and only elements in $\mathcal{D}$ that have $\sigma_F$ on their boundary.

Theorem 6.3.2. $M$ satisfies $G2$.

Proof. For a sufficiently small $\lambda > 0$, the triangles in Skl$^2 S|_\mathcal{D}$ satisfy the conditions for Lemma 6.3.1 and Lemma 6.3.2. This means that properties $P1$ and $P2$ are satisfied when $\lambda$ is sufficiently small. Also when $P1$ and $P2$ are satisfied $\bigcup_i$ Skl$^i S|_\mathcal{D} = \text{Del} S|_\mathcal{D}$. It follows that the Edelsbrunner-Shah conditions are satisfied for the output $M$ of DelPSC2. Thus, $M$ has an underlying space homeomorphic to $|\mathcal{D}|$. The homeomorphism constructed by Edelsbrunner and Shah actually respects the stratification, that is, for each $\sigma \in \mathcal{D}_i$, Skl$^i S|_\sigma$ is homeomorphic to $\sigma$. Also, Skl$^1 S|_\sigma$ consists of only edges that connect adjacent vertices on $\sigma$. Furthermore, property $G1$ holds for any output of DelPSC2. This means, $\text{Bd} (\text{Skl}^2 S|_\sigma) = \text{Skl}^1 S|_{\text{Bd}\sigma}$. Because of the vertex balls, we also have $\text{Bd} (\text{Skl}^1 S|_\sigma) = \text{Skl}^0 S|_{\text{Bd}\sigma}$ trivially. Therefore, for $0 \leq i \leq 2$, $\text{Bd} (\text{Skl}^i S|_\sigma) = \text{Skl}^{i-1} S|_{\text{Bd}\sigma}$ and Skl$^i S|_\sigma$ has vertices only in $\sigma$.

6.4 Experimental Results

We have implemented DelPSC2 using the CGAL library [23]. The input to our software is a polygonal model which we assume approximates a PSC. A user specified threshold for dihedral angles is used to select edges of the input as sharp features (elements of $\mathcal{D}_1$) which we protect. Non-manifold and boundary edges are also included as elements of $\mathcal{D}_1$. In Figure 6.6 we show how our protection algorithm works on four different sets of curves.
Figure 6.6: Protection: Top Left: A 2d example of curve protection. Top Right and Bottom Left: 2 different 3d examples of curve complexes where we have run Protect(). Bottom Right: The final set of protecting balls returned by DelPSC2 on the Wedge model.

In Table 6.2 we show both the time to protect curves as well as the time to generate the surface mesh for twelve different datasets. All experiments were run on a PC with a 2.8GHz CPU and 2GB RAM. We set the input parameter $\lambda$ to 5% of the minimum dimension of the bounding box for each model. Those datasets which took no time for protection had no sharp features in their input; the PSC they approximate was
assumed to be a single smooth patch. The majority of the datasets were meshed in under one minute, only those with complicated topologies took longer.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Protection Time</th>
<th>Meshing Time</th>
<th># of vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 Holes</td>
<td>0.0</td>
<td>105.6</td>
<td>8725</td>
</tr>
<tr>
<td>Arm</td>
<td>52.6</td>
<td>339.7</td>
<td>20692</td>
</tr>
<tr>
<td>Cog</td>
<td>1.4</td>
<td>56.2</td>
<td>7697</td>
</tr>
<tr>
<td>Guide</td>
<td>2.6</td>
<td>22.9</td>
<td>4414</td>
</tr>
<tr>
<td>Horn</td>
<td>0.0</td>
<td>21.5</td>
<td>3192</td>
</tr>
<tr>
<td>Lock</td>
<td>10.6</td>
<td>26.6</td>
<td>5314</td>
</tr>
<tr>
<td>Octo</td>
<td>0.0</td>
<td>7.3</td>
<td>1410</td>
</tr>
<tr>
<td>Part</td>
<td>0.2</td>
<td>23.2</td>
<td>4261</td>
</tr>
<tr>
<td>Plate</td>
<td>17.9</td>
<td>83.9</td>
<td>9773</td>
</tr>
<tr>
<td>Pump</td>
<td>19.1</td>
<td>319.6</td>
<td>20301</td>
</tr>
<tr>
<td>Swirl</td>
<td>0.0</td>
<td>61.1</td>
<td>6880</td>
</tr>
<tr>
<td>Wedge</td>
<td>0.2</td>
<td>13.2</td>
<td>3080</td>
</tr>
</tbody>
</table>

Table 6.2: Protection and meshing times for PSC datasets. Times are in seconds.

In Figure 6.7 we show output meshes for various input shapes. These meshes include four smooth shapes (9 Holes, Horn, Octo, and Swirl), two non-manifolds (Horn and Wedge), and eight piecewise-smooth shapes (Part, Guide, Wedge, Cog, Arm, Lock, Pump, and Plate). DelPSC2 has been used to mesh dozens of other models; further experimentation was presented in a recent multimedia presentation [49].

6.5 Discussion

We have given a second, more practical algorithm to mesh a wide variety of geometric domains with the Delaunay refinement technique. Unlike previous approaches, this algorithm computes the protecting balls on the fly and thus gets rid of expensive computations to fix them a priori. We have eliminated all numerical computations
Figure 6.7: DelPSC output meshes. Left to right, top to bottom: Part, Guide, Wedge, 9 Holes, Cog, Horn, Arm, Octo, Lock, Swirl, Pump, and Plate models.
except for computing intersection points of the input curves with spheres (to determine the end points of $\text{seg}_\sigma(b)$) and computing intersection of Voronoi edges with surface patches (to determine restricted triangles and their sizes). The second computation is always necessary for the restricted Delaunay mesh generation [17, 29]. Moreover, these computations are much easier than the normal variation and gap size computations as proposed in [29].

Our algorithm maintains a manifold property that with increasing level of refinement guarantees the topology of the input is captured. Interestingly, our experiments found that even at low levels of refinement the topology is correct using only the disk condition. This allows for capturing the complex topologies regularly found in CAD inputs. Moreover, by preserving 1-faces in the PSC explicitly, our algorithm naturally preserves all of the input non-smooth features. However, one requirement is that an explicit representation of these features must be available so that we can sample them. For polygonal inputs, it is relatively easy to partition the input into a PSC using the thresholding approach described for the experiments, but for implicitly represented PSCs (Chapter 7) we use a subdivision approach.

We note that this implementation of the algorithm uses a variation on the disk condition. Condition D3 has been changed to only require that each point $p$ sampled from a curve $\sigma \in D_1$ is not connected to any non-adjacent points on $\sigma$ (as opposed to requiring global disconnection from all non-adjacent samples in $D_1$).

Our algorithm can be extended to guarantee bounded aspect ratio at all triangles except near the 1-faces, but cannot provide true adaptively sized meshes. While our experiments will create some varied sizing in the output mesh, these are artifacts of satisfying the disk condition. Many of our output meshes exhibit this property;
it is caused by $\lambda$ (which controls only the maximum triangle size) having a value larger than the triangle size needed to fulfill the disk condition. We note that since adaptive meshing ultimately requires an estimate of the input feature size, it may not be possible to produce such meshes without additional numerical computations.
CHAPTER 7

MESHING INTERFACES OF MULTI-LABEL DATA

We have seen in Chapters 5 and 6 two different algorithms to mesh piecewise smooth complexes with theoretical guarantees. Our goal in this chapter is to bring to bear the practicality of these algorithms to a similar domain encountered in the medical fields. In medical imaging, the generation of surface representations of anatomical objects, obtained by labeling images from various modalities, is a critical component for visualization. The interfaces between labeled regions can be modeled as PSCs, which allow them meet at arbitrary angles and with complex topologies. Figure 7.1 gives an example multi-label dataset which identifies the structural regions of the brain.

In general, the identification of the interface surface for multi-label data is not an easy task, confounded by overlaps of the components, fuzzy boundaries, inherent non-manifold configurations, and arbitrarily sharp angles at the intersections of surface patches. These issues often cause many traditional meshing algorithms to fail. Thus there is need for robustly generating a mesh for interface surfaces which preserves their non-manifold topology and captures their geometric features.

Our primary contribution is the application [46] of the algorithm [48] in Chapter 6 to automatically generate meshes which approximate the interface surfaces defined by
multi-label datasets. This algorithm is an ideal choice for meshing this domain. We can guarantee that it produces meshes that are topologically equivalent to interface surfaces despite their inherent non-manifold topology. Its novel protection scheme allows it to generate meshes for interface surfaces that meet at any angle. An early approach extends the labeling to the tetrahedra within Delaunay triangulation to construct the labeled regions [94]. However, it lacked any theoretical contribution to ensure that it could handle interface surfaces meeting at arbitrary angles.

A second major strength of the algorithm is its simplicity. It only requires two numeric primitives that can be implemented robustly. The first computes intersection points between arbitrary line segments and the interface surface. Second, the algorithm needs a routine for computing the curve network where three or more labeled regions meet. As a result, high quality meshes can be generated with less computational overhead.
Moreover, the algorithm can be applied to datasets from a broad range of data sources. Our experiments demonstrate the algorithm’s effectiveness at producing meshes of interface surfaces for datasets from a variety domains and modalities. These meshes may be used for surface visualization and computation, and the same algorithm can produce volume meshes which are suitable inputs for finite element simulations. The quality of output meshes can be controlled by various user parameters, allowing the generation of both low density meshes suitable for rapid prototyping needs or high density meshes when accuracy is a crucial factor.

7.1 Algorithm

The input to our algorithm is a set of $N$ smooth indicator functions $f_i : \mathbb{R}^3 \rightarrow \mathbb{R}$, $1 \leq i \leq N$. To generate indicator functions we took several segmented volume datasets and converted them to indicator functions. The output of typical hard segmentation algorithms is a set of binary masks over the space $V$. Each binary mask corresponds to one $f_i$, but using the binary values directly is not feasible as an input form. We apply imaging filters (see Section 7.2.1) that convert the representation of this data to a scalar field over $V$. Soft segmentation algorithms output a probability that a grid point is of each label $i$, and hence those can be used more readily to represent indicator functions. Our strategy for generating interface surfaces leaves the input form as flexible as possible to allow a domain expert control over labeling.

7.1.1 Refinement for Interface Surfaces

We follow the refinement approach in Chapter 6 to build our meshes. For interface surfaces, we will protect the input curve features where 3 labeled regions meet by covering them with a set of balls (see top two images in Figure 7.2). Computing
these set of balls directly on multi-label data would be challenging, so the approach from Chapter 6 is ideal. We begin by using a set which may have radii that are too large. The progression of the algorithm itself controls the need to shrink balls that are too large.

Figure 7.2: Refinement algorithm on the Spheres dataset. Top Left: a set of curves are extracted where three or more labeled regions intersect. Top Right: these curves are protected with an initial set of balls that may later get refined by the algorithm. Bottom Left: these balls are converted to weighted points and refinement fills in samples for the interface surfaces. Bottom Right: The output mesh captures all interfaces.
The balls are then turned into weighted points and inserted into a Delaunay triangulation. The restricted Delaunay triangulation is computed, and each vertex in it is checked if it satisfies the DiskCondition. We follow the same rules as before, if any vertex fails the DiskCondition, either a ball is shrunk or a new vertex is inserted which lies on the intersection between $D$ and a Voronoi edge dual to some a restricted triangle. Finally, once all vertices satisfy the DiskCondition, the restricted Delaunay triangulation is refined further to satisfy size and quality requirements.

The computation of intersection points between Voronoi edges and $D$ requires a primitive that computes a point of intersection between an arbitrary line segment and the interface surface defined by the indicator functions. We use a binary-decision walking oracle, similar to the one discussed in [17]. This oracle uses binary search to find all points on the segment where the labeling transitions. It proceeds as follows. First, the labels at each endpoint of the line segment, $l_a$ and $l_b$, are computed as well as the label at the midpoint, $l_m$. If $l_a = l_b$, but $l_m$ is different, then each half of the segment is recursively checked. If $l_a \neq l_b$ each the half between $a$ and $b$ is checked provided that the label at that half’s endpoint differs from $l_m$. Once the distance between $a$ and $b$ becomes smaller than a user specified tolerance, an intersection point (the midpoint) is returned instead of recursively checking the halves.

This approach is different from how we computed intersection points for DelISO in Chapter 4. We found with multi-label data that a binary oracle was simpler to implement and robust than solving the trilinear system defined by the indicator functions.
7.1.2 Extracting Curves

The final ingredient required to implement the algorithm is a method to extract the curve network $\mathcal{D}_{\leq 1}$ from the input dataset. An example network for the spheres dataset is shown in top left image of Figure 7.2. We use an algorithm that walks through each cube cell in the volume dataset that contains three or more labels. Our goal is to generate a set of segments whose union is the set $\mathcal{D}_{\leq 1}$.

We first use bilinear interpolation on each face of the cell to compute the intersection points with a curve (defined by three indicator functions) and the face. If a cell has only two such points, we connect them with a segment, forming a piece of some element of $\mathcal{D}_1$. Otherwise, if a cell has more than two (indicating that more than one curve passes through the cell), we subdivide the cell at its midpoint into eight pieces. Each of the 8 subcells are checked again, and we allow them to be subdivided up to three levels deep. If there are still more than two points at the finest level of subdivision, we connect all intersection points to their centroid, forming an element of $\mathcal{D}_0$.

This process again saves us from solving the roots of a cubic equation, instead favoring more robust subdivision. One caveat is that occasionally multiple elements of $\mathcal{D}_0$ may be computed in close proximity to each other. Our curve extraction algorithm performs a final merge step which combines elements of $\mathcal{D}_0$ that are nearby within a user defined tolerance. After this merging process, we group segments to form each individual curve in $\mathcal{D}_1$ by growing curves from each element of $\mathcal{D}_0$. 
7.2 Implementation Details

7.2.1 Data Preprocessing

Each of our datasets requires some preprocessing to convert them to indicator functions suitable for meshing. Some datasets require first separating the data into individual binary masks for each label. At this point, binary morphology operators such as dilation or median filtering [105] can be applied to clean the data. All datasets from hard segmentations then are converted from binary masks to a signed scalar field. For this task we apply a signed Danielsson distance filter [41] to each mask.

After producing the raw indicator functions, additional preprocessing steps were applied to control the size of features in the data. We use a mix of Gaussian smoothing [43] as well as curvature anisotropic diffusion (CAD) smoothing [124] at this step. CAD is a variation of classical non-linear diffusion, where the speed of the diffusion is proportional to the local curvature of the level-sets of the image. Areas of high curvature diffuse faster than areas of low curvature, thereby smoothing out noise while preserving large scale image features and edges. While this method does not exhibit the edge enhancing properties of classic anisotropic diffusion, it is more stable and preserves finer structures in the images.

7.2.2 Input Parameters

Our implementation accepts a parameter to scale the radii of initial protecting balls as well as parameters to control triangle shape. Our parameters include a maximum bound for triangle circumradii (thus uniformly sizing all triangles) and shape quality is controlled with a bound for the circumradius to shortest edge ratio. Mesh conformity to the interface surface is controlled by allowing the user to specify
a maximum distance from the circumcenter of the triangle to target shape. Triangles are refined to satisfy shape criteria provided their insertions do not get too close to a protecting ball.

To generate volume meshes, a similar set of parameters control tetrahedra shape. We allow the user to specify a uniform size control via a maximum bound on tetrahedra circumradii. Tetrahedra aspect ratios are controlled similar to triangles by specifying tetrahedra aspect ratio in terms of a circumradius to shortest edge threshold. This quality criterion generates tetrahedra suitable for finite element methods, but can still allow some sliver tetrahedra in the output mesh. These slivers can be eliminated as a postprocessing step or using the circumradius to inscribing radius ratio for refinement.

7.3 Experimental Results

Our experimental datasets cover a broad range of inputs from various sources. We used synthetic data and segmented datasets from both CT and MRI. Our datasets include segmentations done manually by experts as well as by using various both hard and soft classification algorithms. Table 7.1 gives an overview of the datasets, their size, number of labels, and modalities.

Our experimental PC setup consisted of an Intel Pentium4 CPU at 2.8Ghz with 2GB of memory. Our algorithm is implemented in C++ using CGAL [23] for the computation of the weighted Delaunay triangulation. We used the same machine for both preprocessing and mesh generation. Preprocessing required no more than a few minutes for each label, once the appropriate filters were selected. Mesh generation times are reported in Table 7.2.
Table 7.1: Dataset summary. For each dataset the dimensions, source, and number of labels are given. Note there is an additional “background” label present in all datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># of Labels</th>
<th>Source</th>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spheres</td>
<td>8</td>
<td>synthetic</td>
<td>128 × 128 × 128</td>
</tr>
<tr>
<td>Orange</td>
<td>4</td>
<td>MRI</td>
<td>256 × 256 × 64</td>
</tr>
<tr>
<td>Temporal Bone</td>
<td>5</td>
<td>CT</td>
<td>256 × 256 × 256</td>
</tr>
<tr>
<td>Tomato</td>
<td>4</td>
<td>MRI</td>
<td>256 × 256 × 64</td>
</tr>
<tr>
<td>MNI Brain Atlas</td>
<td>7</td>
<td>MRI</td>
<td>182 × 218 × 182</td>
</tr>
<tr>
<td>Torso</td>
<td>4</td>
<td>MRI</td>
<td>260 × 121 × 169</td>
</tr>
<tr>
<td>Bulldog Skull</td>
<td>4</td>
<td>CT</td>
<td>196 × 176 × 268</td>
</tr>
<tr>
<td>Soft Brain</td>
<td>2</td>
<td>MRI</td>
<td>181 × 217 × 181</td>
</tr>
</tbody>
</table>

Table 7.2: Meshing summary. For each dataset the number of output vertices and the number of curves protected are listed. Output times (in minutes:seconds) are broken down by protection time as well and time to generate the mesh. File I/O times are excluded.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># of Curves</th>
<th># of Vertices</th>
<th>Protection Time</th>
<th>Meshing Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spheres</td>
<td>36</td>
<td>4492</td>
<td>0:16</td>
<td>0:50</td>
</tr>
<tr>
<td>Orange</td>
<td>24</td>
<td>24543</td>
<td>0:14</td>
<td>8:46</td>
</tr>
<tr>
<td>Temporal Bone</td>
<td>16</td>
<td>24898</td>
<td>0:11</td>
<td>8:32</td>
</tr>
<tr>
<td>Tomato</td>
<td>35</td>
<td>25701</td>
<td>1:01</td>
<td>9:07</td>
</tr>
<tr>
<td>MNI Brain Atlas</td>
<td>64</td>
<td>35890</td>
<td>1:04</td>
<td>17:50</td>
</tr>
<tr>
<td>Torso</td>
<td>57</td>
<td>39826</td>
<td>1:37</td>
<td>19:27</td>
</tr>
<tr>
<td>Bulldog Skull</td>
<td>17</td>
<td>51474</td>
<td>0:10</td>
<td>32:48</td>
</tr>
<tr>
<td>Soft Brain</td>
<td>100</td>
<td>211664</td>
<td>1:57</td>
<td>244:01</td>
</tr>
</tbody>
</table>

These times are quite competitive for meshing interface surfaces in a high quality manner. In particular, the major bottleneck of the algorithm is computing the weighted Delaunay triangulation with dynamic insertions. If we specify a dense output mesh (as we did for the soft segmentation of the brain), we see the computation time grow non-linearly. However, since our meshing density does not rely on the grid
resolution, we can save ourselves computation by specifying more relaxed refinement criteria. At lower resolutions we can still capture the shapes quite well. Thus, one strength of our algorithm is we can compute meshes which capture the topology using significantly fewer vertices.

A visualization of the mesh we generated for the MNI brain atlas is shown in Figure 7.1. This brain image was obtained from the MNI-space atlases provided along with FSL [110]. A single subject’s structural image was hand segmented, and the labels were then propagated to more than 50 subjects’ structural images using non-linear registration. Each resulting labeled brain was then transformed into MNI152 space using affine registration, before averaging segmentations across subjects to produce the final probability images.

Figure 7.3 shows a mesh from a dataset generated by probabilistic segmentation of the brain’s grey and white matter. This dataset was obtained by first normalizing a T1-weighted MR image to the MNI152 reference space, followed by a probabilistic segmentation [123] using a combination of in-house and standard tools such as FSL [110]. The prior probabilities for the tissue distributions were obtained from the MNI-ICBM atlas (NIH P-20 project) [77], specified in the same reference space. The resulting probability images contain values in the range of zero to one, representing the posterior probability of a voxel being either grey matter or white matter. It is interesting to note that our mesh reveals that in this segmentation the white matter pierces through the outer grey matter in several locations.

Two datasets generated from CT sources are shown in Figure 7.4. Both datasets were hand-segmented by experts into more than 50 structural elements. We group them in a smaller number of labels to make them more manageable for our software.
Figure 7.3: Soft segmentation of the brain. Grey matter is drawn transparently and white matter is drawn in green. Red triangles show where the white matter protrudes through the grey.

to mesh. As a result, both of these datasets had far fewer curve elements than the MRI datasets.

Table 7.3 shows a summary of quality statistics for the triangles produced by the meshing algorithm. We control the aspect ratios of triangles by refining to enforce the condition that all triangles have aspect ratio below a minimum threshold. This table also shows the minimum dihedral angle between any pair of triangles circling an edge along a curve element. This value indicates how sharply the interface surfaces are meeting in the output mesh. Our implementation can tolerate surfaces meeting at arbitrary angles, as the table indicates.
Table 7.3: Quality and shape statistics. The average and maximum aspect ratio (radius/shortest edge length) are given for each of the datasets. Also shown are the minimum dihedral angle (in degrees) between all pairs of faces circling each non-manifold edge. This measure shows how sharply the interface surfaces are meeting at curve elements.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># of triangles</th>
<th>mean aspect ratio</th>
<th>max aspect ratio</th>
<th>min angle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spheres</td>
<td>9527</td>
<td>1.00</td>
<td>1.05</td>
<td>42.35°</td>
</tr>
<tr>
<td>Orange</td>
<td>49017</td>
<td>1.00</td>
<td>1.03</td>
<td>20.71°</td>
</tr>
<tr>
<td>Temporal Bone</td>
<td>49911</td>
<td>1.00</td>
<td>1.03</td>
<td>3.74°</td>
</tr>
<tr>
<td>Tomato</td>
<td>52729</td>
<td>1.00</td>
<td>1.03</td>
<td>18.39°</td>
</tr>
<tr>
<td>MNI Brain Atlas</td>
<td>72962</td>
<td>1.00</td>
<td>1.03</td>
<td>13.30°</td>
</tr>
<tr>
<td>Torso</td>
<td>82104</td>
<td>1.00</td>
<td>1.05</td>
<td>14.69°</td>
</tr>
<tr>
<td>Bulldog Skull</td>
<td>103036</td>
<td>1.00</td>
<td>1.02</td>
<td>12.10°</td>
</tr>
<tr>
<td>Soft Brain</td>
<td>424690</td>
<td>1.00</td>
<td>1.02</td>
<td>8.72°</td>
</tr>
</tbody>
</table>

Three additional datasets come from MRI sources. Figure 7.5 shows a Torso dataset with four of the major structures labeled. This dataset has a large network
of curves to protect where the skin, lungs, and skeleton meet, shown in the closeup shots. Figures 7.7 and 7.8 show the Orange and Tomato datasets, where we peel away the outer layers to see the internal structures. These two datasets labels are generally contained within each other, thus they have few curves to protect.

After satisfying the constraints for surface refinement, we can generate volume meshes as well. Figure 7.6 shows some example volume meshes for the Spheres and MNI Brain Atlas datasets. Since the 3D Delaunay triangulation already generates a tetrahedralization of the volume contained within the shape, as soon as the interface surface is captured we automatically have a volume mesh. We can refine these tetrahedra by using the standard approach of inserting the circumcenters of tetrahedra with bad radius-edge ratios. We perform this insertion for any tetrahedra provided
that the circumcenter does not cause the deletion of one of the triangles on the interface surface. This strategy lets us refine within the volume, but ensures the interface surface triangles remain in the output mesh.

7.4 Discussion

Since our algorithm extends a proven theoretical result, we can mesh interface surfaces with surface elements that meet at arbitrarily sharp angles. Our experiments show that this technique lends itself to generating both low resolution meshes fast and higher quality approximations in reasonable time. While our times for meshing are competitive, its performance bottleneck is the asymptotic speed to compute the weighted Delaunay triangulation. A faster implementation of this data structure could greatly improve the run time.
Figure 7.7: Orange dataset. From left to right, top to bottom we show the labels for the outer skin, membrane, pulp, and seeds.

Since we do not explicitly compute a measure of feature size for the surfaces, our implementation favors using a uniform size requirement to control the mesh density. It is often desirable to have meshes which adapt to the input’s features. However, for the case of interface surfaces, a true estimation of this measure may not be possible to compute without a prohibitive amount of overhead. Future research is necessary to explore how should an adaptive mesh be generated for the case of piecewise smooth shapes.
Figure 7.8: Tomato dataset. From left to right, top to bottom we show the labels for the endocarp, locule, placenta, and core.

Our implementation allows for labeled inputs generated from various imaging and segmentation sources. We use only basic filters on these datasets to minimize the amount of preprocessing time. Separating this processing step from the algorithm itself allows a domain expert control over how much the data is altered when converted to a system of indicator functions. Since our algorithm uses the indicator functions as input, it is interesting to note that we can process probabilistic segmentations in a more direct manner. Common usage thresholds and converts these soft segmentations to binary masks before processing—creating a potential loss of information.
CHAPTER 8

CONCLUSIONS

We have presented a collection of algorithms which use Delaunay triangulations to approximate geometric domains. The first is a simplification technique for vector field data in two and three dimensions using vertex deletion. We provide experimental evidence to demonstrate the effectiveness of this approach. Our second algorithm, DelIso, shows how smooth isosurfaces can be meshed using Delaunay refinement. Our approach differs from traditional Delaunay refinement in that we can discard the full three dimensional structure after a certain level of refinement has been reached. This two-staged approach provides a 2-3 times speedup. Finally, we describe two versions of DelPSC, an algorithm for meshing PSCs, both of which increase in practicality. Using this algorithm we can mesh interface surfaces of multi-label data as well.

These algorithms all make small attempts at taking theoretical contributions and making them applicable to real world needs, whether they are engineering, design, or medical imaging. Their greatest strength is in the balance they strike between offering practical benefits and provable guarantees. Since meshing algorithms are always discussed within the context of a shape design pipeline each algorithm must be robust enough to handle the application’s demands.
For piecewise smooth domains, the algorithms presented here are some of the first known algorithms for Delaunay mesh generation that are able to process such a wide, disparate class of shapes automatically. The fact they can certifiably accomplish this task is a significant boon. Moreover, the notion of protecting feature curves with a set of balls is a novel extension from the ideas of polyhedral meshing. This idea has already found uses in shape repair [22] and may continue to be a fruitful idea for other geometric problems.

In general, a 3D Delaunay triangulation requires $O(n^2)$ runtime, where $n$ is the number of vertices in the triangulation. However, it is known that for points sampled from polyhedra embedded in three dimensions that the Delaunay triangulation has $O(n)$ space complexity [7]. Indeed, by carefully choosing points to insert and using a clever spatial location structure, a sparse algorithm for meshing piecewise-linear inputs has $O(n \log n + m)$ runtime and $O(m)$ space requirements where $n$ is based on the input features of the shape and $m$ is the number of vertices in the output mesh [2]. No work has yet extended this result to smooth or piecewise smooth domains.

8.1 Future Directions

In its current form, any algorithm for meshing PSCs requires that non-smooth and non-manifold features are known preemptively so that they can be protected before the algorithm begins. However, for some class of inputs, these features are not necessarily known at the initial stages. For example, while some techniques exist for computing the sharp features of an isosurface a priori [74], it may be more efficient to avoid such a step. Even our algorithm for meshing interface surfaces of multi-label data required a subdivision approach to identify the curves. In general, when
meshing PSCs there may be some benefit to detecting non-smooth regions during refinement. One can envision an algorithm where we dynamically protect non-smooth regions when they are encountered.

Our meshing algorithms have focused on generating isotropic meshes where the triangles approach equilateral in shape. While isotropic meshes often provide good quality approximations for shapes [5], especially in a generic sense, it is well known that meshes with elongated triangles in appropriate directions can improve the numeric interpolations performed on meshes. Only recently have provably algorithms [19, 32] to generate meshes that respect an anisotropic field been discovered by the research community. Two important research goals are natural continuations of this research. The first is practical: no provable algorithm has been implemented for meshing 3-dimensional domains anisotropically. Second, the majority of the known algorithms restrict the input domain greatly, but many of the applications which require anisotropy also require the mesh to approximate a piecewise-smooth shape. An algorithm which can mesh PSCs with elements that honor anisotropic constraints would be a significant result, but it is an open question if this complex, potentially conflicting set of requirements can be satisfied simultaneously.

All of the algorithms in this dissertation have been implemented on single CPU systems. However, as is often the case in computer science, a single machine does not provide enough resources to compute solutions for larger datasets. One future research direction is developing parallel versions of the some of the algorithms. A parallel system presents a number of unique opportunities for innovation.

Finally, there is an obvious connection between surface reconstruction, sampling, and mesh generation algorithms. For the case of smooth surfaces, the theory that
connects them [8, 9, 45] is well developed, but few advances [24, 40] have been made for reconstruction of classes of shapes which are non-smooth or non-manifold. As 3D scanning technology becomes more robust, we are generating point cloud datasets of more complex, non-smooth shapes. Applying smooth surface reconstruction techniques to these datasets will not be sufficient. Since we have a proven technique for meshing PSCs, an important question remains to see if we can take aspects of our Delaunay refinement algorithm and apply it to reconstruction of PSCs from a fix point cloud data set. Any such algorithm will require machinery to reconstruct the two-dimensional smooth patches as well as a technique for identifying regions of non-smoothness and non-manifoldness; the latter of which may not be sufficiently sampled by modern scanning techniques even if the smooth patches have been sampled properly.
BIBLIOGRAPHY


149


