Topological Methods for 3D Point Cloud Processing

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

William Joseph Beksi

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Nikolaos Papanikolopoulos, Adviser

August, 2018

Acknowledgements

First and foremost, I thank my adviser Nikolaos Papanikolopoulos. Nikos was the first person in my career to see my potential and give me the freedom to pursue my own research interests. My success is a direct result of his advice, encouragement, and support over the years. I am beholden to my labmates at the Center for Distributed Robotics for their questions, comments, and healthy criticism of my work. I'm especially grateful to Duc Fehr for letting me get involved in his research as a new member of the lab. I also thank all of my committee members for their feedback through out this entire process.

The inspiration for this research is due to the Institute for Mathematics and its Applications (IMA) located at the University of Minnesota. Each year the IMA explores a theme in applied mathematics through a series of tutorials, workshops, and seminars. It was during the Scientific and Engineering Applications of Algebraic Topology (2013-2014) that I began to conceive of the ideas for this thesis.

This work would not be possible without the computational, storage, and software resources provided by the Minnesota Supercomputing Institute (MSI) and its excellent staff. I especially thank Jeffrey McDonald for his friendship and support.

I acknowledge and thank the National Science Foundation (NSF) for supporting this research and the University of Minnesota Informatics Institute (UMII) for funding my fellowship.

Last but not least, I am thankful for the unwavering faith of my family, especially from my wife Han Li.

Dedication

To my old man, scientia potentia est.

Abstract

3D point cloud datasets are becoming more common due to the availability of low-cost sensors. Light detection and ranging (LIDAR), stereo, structured light, and time-of-flight (ToF) are examples of sensors that capture a 3D representation of the environment. These sensors are increasingly found in mobile devices and machines such as smartphones, tablets, robots, and autonomous vehicles. As hardware technology advances, algorithms and data structures are needed to process the data generated by these sensors in innovative and meaningful ways.

This dissertation develops and applies algebraic topological methods for processing 3D point cloud datasets. The area of topological data analysis (TDA) has matured in recent years allowing researchers to analyze point cloud datasets using techniques that take into account the 'shape' of the data. This includes topological features such as connected components, holes, voids, and higher dimensional analogs. These ideas have been successfully applied to datasets which are naturally embedded in a metric space (such as Euclidean space) where distances between points can be used to form a parameterized sequence of spaces. By studying the changing topology of this sequence we gain information about the underlying data.

In the first part of the thesis, we present a fast approach to build a 3D Vietoris-Rips complex which allows us to approximate the topology of a point cloud. The construction of the complex is done in three parallelized phases: nearest neighbors search, edge list generation, and triangle list generation. The edge and triangle lists can then be used for persistent homology computations.

In the second part of the thesis, we present approaches to segment 3D point cloud data using ideas from persistent homology theory. The proposed algorithms first generate a simplicial complex representation of the point cloud dataset. Then, the zeroth homology group of the complex which corresponds to the number of connected components is computed. Finally, we extract the clusters of each connected component in the dataset. We show that these methods provide a stable segmentation of point cloud data under the presence of noise and poor sampling conditions, thus providing advantages over contemporary segmentation procedures.

In the third part of the thesis, we address an open problem in computational topology by introducing a nearly linear time algorithm for incrementally computing topologically persistent 1-cycles. Further, we develop a second algorithm that utilizes the output of the first to generate a spanning tree upon which non-bounding minimal 1-cycles can be computed. These non-bounding minimal 1-cycles are then used to locate and fill holes in a dataset. Experimental results show the efficacy of our algorithms for reconstructing the surface of 3D point clouds produced by noisy sensor data.

In the fourth part of the thesis, we develop a global feature descriptor termed Signature of Topologically Persistent Points (STPP) that encodes topological invariants (zeroth and first homology groups) of 3D point cloud data. STPP is a competitive 3D point cloud descriptor when compared to the state of art and is resilient to noisy sensor data. We demonstrate experimentally that STPP can be used as a distinctive signature, thus allowing for 3D point cloud processing tasks such as object detection and classification.

This dissertation makes progress towards effective, efficient, and scalable topological methods for 3D point cloud processing along two directions. We present algorithms with an analysis of their theoretical performance and proof of correctness. We also demonstrate the feasibility and applicability of our results with experiments using publicly available datasets.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Contemporary times have witnessed a dramatic increase in advanced technologies such as smartphones, robots, and autonomous/connected vehicles. These mobile devices and machines are fueling the growth of an interconnected data sharing ecosystem known as the Internet of Things (IoT) [4]. For each of these technologies, the number of sensors deployed is rapidly progressing. Moreover, these sensors have the capacity to continuously generate vast amounts of data. To add value to raw sensor data, we must be able to understand it. Consequently, this requires effective, efficient, and scalable algorithms and data structures.

Among the various sensor data modalities such as audio, image, text, etc., 3D data (in the form of point clouds) is beginning to constitute a growing portion of the spectrum. A 3D point cloud is composed of a set of points where each point has a Cartesian coordinate value in Euclidean space, and optionally an RGB color value. With today's hardware, the typical size of a 3D point cloud can range from thousands to hundreds of thousands of points. They can also be stitched together in a process know as *registration*, a procedure for finding a spatial transformation that aligns two point clouds, to produce datasets with billions of points.

Robotics is at the forefront of technologies that are making use of 3D point cloud data. Senors such as light detection and ranging (LIDAR), stereo, structured light, and time-of-flight (ToF) can enhance the vision capabilities of a robot, Figure 1.1. The 3D representation of the environment provided by these sensors can facilitate robotic tasks such as object detection, pose estimation, motion planning, and grasping. The DARPA

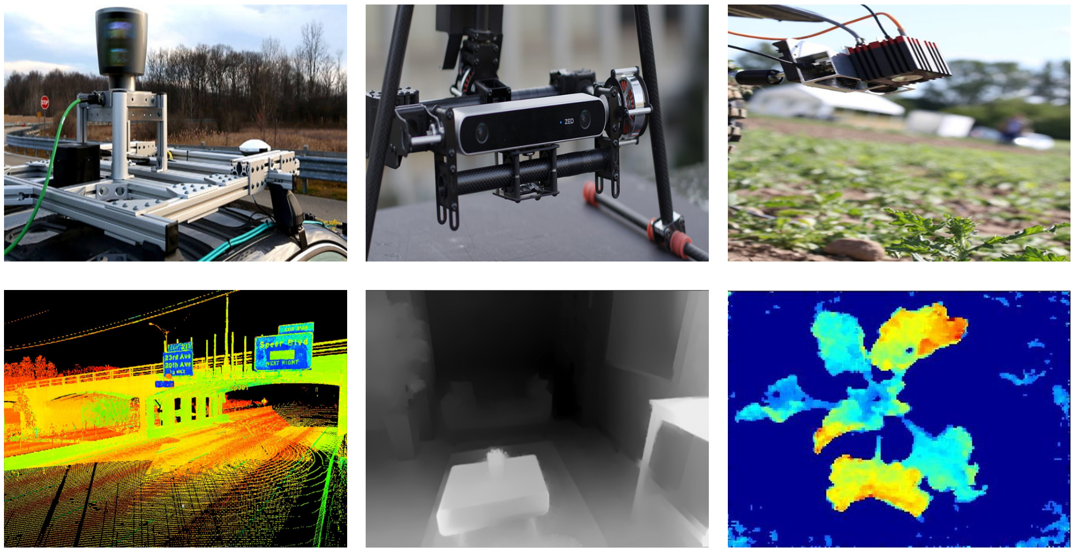


Figure 1.1: Examples of a LIDAR system, stereo camera, ToF camera (top row) and their respective datasets (bottom row).

Grand, Urban, and Robotic Challenges and the Amazon Picking Challenge have shown the effectiveness of incorporating 3D sensors into robotic platforms. These challenges have propelled advances in 3D perception for self-driving cars, humanoid robotics, and warehouse automation.

1.1 3D Point Cloud Processing Challenges

Contrary to other data modalities, processing 3D point clouds poses several significant challenges. Point cloud data generated by low-cost sensors suffers from the presence of artifacts, non-uniform noise, and variation in density. The established 2D image processing techniques are not directly applicable to 3D point clouds. The main reasons for this are:

- Difference in the data representation: An image is organized as a matrix while a 3D point cloud is an unorganized and irregularly distributed set of points.
- Difference in the information presentation: An image contains ambiguous spatial information and abundant spectral information while a 3D point cloud contains

explicit spatial information and possibly no spectral information.

- Difference in the spatial neighborhood: An image is arranged in a grid-like pattern thus allowing the neighborhood of a pixel to be easily determined while the neighborhood of 3D point cloud must be determined by a nearest neighbors search.

Motivated by these observations, this thesis explores new algorithms and data structures for 3D point cloud processing. The fundamental viewpoint of this work is that the incorporation of topological features can yield new insight into the structure of 3D point cloud data which is not obtainable from other methods. To this end, we make use of persistent homology, the main tool of topological data analysis (TDA).

1.2 Persistent Homology and its Applications

Datasets are often contained in a metric space, such as Euclidean space, with an inherited distance function. In many cases, we are not interested in the exact geometry of these spaces, but instead we seek to understand basic topological characteristics. Algebraic topology obtains these characteristics through homology. Homology associates algebraic structures to a space. These algebraic structures are robust in the sense that they do not change when the underlying space is transformed by deformations.

Persistent homology is an algebraic method for measuring the topological features of a dataset. The key insight of persistent homology is the following: one considers a series of scale values, to extract qualitative information from data, since a priori there is no clear choice for what the value of scale parameter should be. Persistent homology thus captures how the homology of the algebraic structures change as the scale parameter increases. In other words, it detects which features ‘persist’ across changes in resolution.

The initial motivation of persistent homology was to denoise datasets where small size features are classified as noise. However, one’s definition noise is application dependent. Features come in multiple scales and can be nested in complicated relationships. Persistent homology addresses these factors.

The notion of persistence emanated independently in the work of Frosini, Ferri, and collaborators in Bologna, Italy, in the doctoral work of Robins at the University of Colorado, Boulder, and within the BioGeometry project of Edelsbrunner et al. at

Duke University, North Carolina [5]. These developments happened at the turn of the last century with relevant discoveries occurring over a period of fifteen years. Frosini and Ferri refer to persistent homology in degree 0 as size theory which is motivated by the study of the natural pseudo-distance between two functions on homeomorphic topological spaces [6–8]. Persistence is defined in shape theoretic terms by Robins who uses the idea in the study of fractal sets with alpha shapes [9]. Alpha shapes are also at the foundation of the developments at Duke University [10].

Simplicial filtrations and the differentiation between positive and negative simplices, two of the essential algebraic components of persistence, can be traced back to the implementation of three-dimensional alpha shapes by Mücke [11] and the incremental Betti number algorithm of Delfinado and Edelsbrunner [12]. Another pivotal insight is the existence of a unique pairing where positive simplices mark the appearance (birth) of topological features while negative simplices mark their disappearance (death). This pairing of positive and negative simplices, and an algorithm to compute it, has been crucial in connecting the mathematical ideas to practical problems.

Early applications of persistent homology can be found in the two-dimensional coverage of static sensor networks. Given a set of sensors with minimal observational capabilities, the question is to determine if the sensors cover a given area. Based on weak assumptions regarding the location of the sensors in the network, DeSilva and Ghrist use the homology of simplicial complexes to decide this question [13–15]. They show that by using persistence these characterizations can be made robust to variations in the distribution of sensors and gaps in the coverage. More recently, Gamble et al. have developed methods that make use of zigzag persistent homology to analyze coverage properties in dynamic sensor networks [16].

The situation in which objects are embedded in three dimensions is significant and its applications are numerous. Given an object in $\mathbb{R}^3$, a typical topological question is how many connected components, holes, and voids exist. These features (components, holes, voids) have concrete interpretations. For example, the number of celestial bodies in a galaxy, the number of independent closed routes that go around an obstacle, and the number of sections of a cell that are occupied by fluid [17].

1.3 Contributions

Towards developing topological methods for 3D point cloud processing this thesis makes the following contributions. In the first part, we introduce algorithms and data structures to speedup the simplicial complex construction process which is a prerequisite to computing persistent homology. The second part presents 3D segmentation algorithms based on computing the zeroth homology group (number of connected components) in a point cloud. In the third part, we introduce a new algorithm for quickly computing the first homology group (number of holes) and show how it can be used for determining the boundary points of holes in a 3D point cloud. In the last part, we propose a global 3D point cloud descriptor based on computing topological persistence. A brief overview of each problem along with our contributions is given next.

1.3.1 Fast Construction of the Vietoris-Rips Complex in $\mathbb{R}^3$

A simplicial complex, formally defined in Chapter 2, is a space built from the union of points, edges, triangles, tetrahedra, and higher-dimensional polytopes. Approximating a space as a simplicial complex allows us to compute homology. The Vietoris-Rips complex, one of many different types of simplicial complexes used in TDA, is particularly appealing because it can be computed efficiently. Our contribution is a set of algorithms and data structures that not only allow the Vietoris-Rips complex to be computed efficiently, but also show how it lends itself to be constructed in parallel. This fast construction of the Vietoris-Rips complex is then used in the 3D point cloud processing applications that follow.

1.3.2 Point Cloud Segmentation

To correctly interpret 3D point cloud data, we need to partition the dataset into clusters that correspond to objects or regions. This process, known as data segmentation, is an imperative filtering step for high-level functions such as machine learning. Using ideas from persistent homology theory, we address this problem as follows. First, our algorithms grow a simplicial complex representation of the dataset. Then, at each step in the growth process we compute the zeroth homology group of the complex. For region-based segmentation, we combine global (topological) and local (color, surface normal) information to segment colored 3D point clouds. Lastly, we extract out the segmented objects/regions of the dataset. We show that these methods provide a stable segmentation of point cloud data in the presence of noise and poorly sampled data, thus providing advantages over contemporary segmentation techniques.

1.3.3 Point Cloud Hole Boundary Detection

The absence of connectivity information simplifies the definition and implementation of many tasks in the area of point cloud processing. However the detection of holes, straightforward in the case of mesh-based datasets, becomes an ill-defined problem. Knowing about the existence and location of holes in the data is necessary for many point cloud applications. These applications can range from surface reconstruction to determining where to place a sensor in order to gain more information. To address this problem, we first establish a fast algorithm for computing 1-cycles (holes) in a 3D point cloud. We then propose a topological approach to finding the boundary points of holes in a dataset that is qualitatively competitive with and quantitatively exceeds the current dominant method in this area.

1.3.4 Point Cloud Signatures

Within a 3D point cloud, objects can be recognized by either local or global means. Local description of an object is carried out by encoding geometric information within the neighborhood of a point. A global description can be realized by encoding the entire object geometry. While the use of discriminative geometric information is typically used in composing point cloud descriptors, we look to the topological structure of the data as a complementary source of information. In doing so, we present the Signature of Topologically Persistent Points (STPP), a global descriptor that encodes topological invariants of 3D point cloud data. These topological invariants include the zeroth and first homology groups and are computed using persistent homology. We show that STPP is a competitive 3D point cloud descriptor when compared to the state of art and is resilient to noisy sensor data. We also demonstrate experimentally on a publicly available RGB-D dataset that STPP can be used as a distinctive signature, thus allowing for 3D point cloud processing tasks such as object detection and classification.

1.4 Thesis Organization

This remainder of this thesis is organized as follows. In Chapter 2, we present the essential mathematical background necessary for this dissertation. This background starts with defining simplices and ends with computing persistent homology. We then explore parallel methods for constructing a three dimensional Vietoris-Rips complex in Chapter 3. In Chapter 4, the first application of our work is presented: object and region segmentation of 3D point clouds. This application is followed by the introduction of a fast incremental algorithm for computing persistence and a technique for hole boundary point detection in Chapter 5. Our final application, in Chapter 6, shows how to generate a topologically persistent signature that can be used for object detection and classification. We conclude the thesis with an overview of our contributions and highlight avenues of future research.

Chapter 2

Mathematical Background

This thesis makes use of ideas stemming from algebraic topology. In the following sections we provide a brief index of key definitions and constructs necessary to compute topological persistence. We start with simplices, simplicial complexes, and abstract simplicial complexes. Then, we proceed with chains, cycles, boundaries, and homology groups for $\mathbb{Z}_2$ coefficients. Finally, we define filters, filtrations, persistent homology, and show how to compute persistent homology.

2.1 Simplices

Definition 1. *A k-simplex σ is the convex hull of $k+1$ affinely independent points $p_0, \ldots, p_k \in \mathbb{R}^n$. We denote $\sigma = conv\{p_0, \ldots, p_k\}$ where the dimension of σ is k.*

For any of the points, p_i, the k vectors $p_j - p_i$, $j \neq i$, are linearly independent. In other words, given a set of $k+1$ points, a simplex is the set of points each of which is a linear combination of these points with nonnegative coefficients summing to 1. The convex hull is simply the solid polyhedron determined by the $k+1$ points. Examples of simplices can be seen in Figure 2.1.

Definition 2. *A face of σ is $convS$ where $S \subset \{v_0, \ldots, v_k\}$ is a subset of the $k+1$ points.*

For example, the four faces of a tetrahedron correspond to the four subsets of S obtained by removing points one at a time from σ. These four triangle faces are themselves

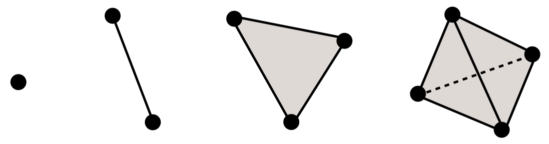


Figure 2.1: A point, edge, triangle, and tetrahedron are simplices of dimension 0, 1, 2, and 3 respectively.

2-simplices. There also exists six edge faces and four point faces.

2.2 Simplicial Complexes

Topological spaces are composed of simplicial complexes, Figure 2.2.

Definition 3. *A simplicial complex K is a finite collection of simplices such that if $\sigma \in K$ and τ is a face of σ, then $\tau \in K$ and if $\sigma, \sigma' \in K$ then $\sigma \cap \sigma'$ is either empty or a face of both σ and σ'.*

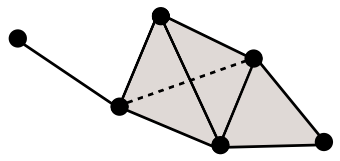


Figure 2.2: A simplicial complex is constructed by gluing together simplices.

An *abstract simplicial complex*, based on a set of points $P = \{p_0, \ldots, p_k\}$, is a collection K of simplices closed under the operation of taking subsets: if $\sigma \subset P$ is a simplex ($\sigma \in K$) and τ is a face of σ ($\tau \subset \sigma \subset P$), then $\tau \in K$ as well.

2.2.1 Nerve Complex

The Nerve complex is an abstract simplicial complex constructed from an open covering of a topological space.

Definition 4. *Let $X = \{x_0, \ldots, x_{m-1}\} \in \mathbb{R}^n$ be a set of points in Euclidean n-space and let $U = \{U_i\}$ be a finite collection of sets. The nerve of U is an abstract simplicial* *complex whose k-simplices are unordered collections of $k+1$ elements of U having non-empty intersection,*

$$N(U) = \{X \subseteq U \mid \bigcap X \neq \emptyset\}.$$

According to the Nerve theorem [18], when U consists of convex sets in Euclidean space, $N(U)$ is homotopic (topologically equivalent) to the union $\cup_i U_i$.

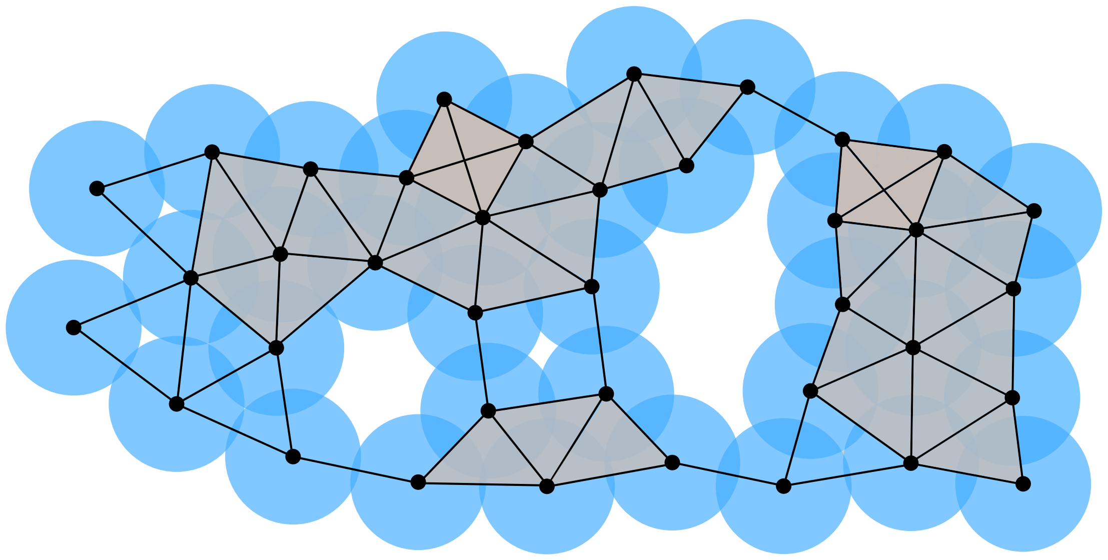


Figure 2.3: A Čech complex.

2.2.2 Čech Complex

The Čech complex is an abstract simplicial complex that models of the topology of a space, Figure 2.3.

Definition 5. *Let $X = \{x_0, \ldots, x_{m-1}\} \in \mathbb{R}^n$ be a set of points in Euclidean n-space and let r be a fixed radius. The Čech complex is an abstract simplicial complex whose k-simplices are determined by unordered $(k+1)$-tuples of points in X whose closed balls of radius r centered at x, $B(x,r)$, have a point of common intersection:*

$$C_r(X) = \{\sigma \subseteq X\} \mid \bigcap_{x \in \sigma} B(x,r) \neq \emptyset\}.$$

The Čech complex is the nerve of the union of balls of radius r. If the cover sets, and all nonempty finite intersections of the cover sets are contractible, then the Čech complex faithfully captures the topology of the cover [19].

2.2.3 Vietoris-Rips Complex

The Vietoris-Rips complex is an abstract simplicial complex that approximates the topology of a space, Figure 2.4.

Definition 6. *Let $X = \{x_0, \ldots, x_{m-1}\} \in \mathbb{R}^n$ be a set of points in Euclidean n-space and let r be a fixed radius. The Vietoris-Rips complex of X is an abstract simplicial complex whose k-simplices correspond to unordered $(k+1)$-tuples of points in X that are pairwise within r distance of each other,*

$$VR_r(X) = \{\sigma \subseteq X \mid d(x_i, x_j) \leq r, \forall x_i \neq x_j \in \sigma\},$$

where d is the Euclidean distance metric.

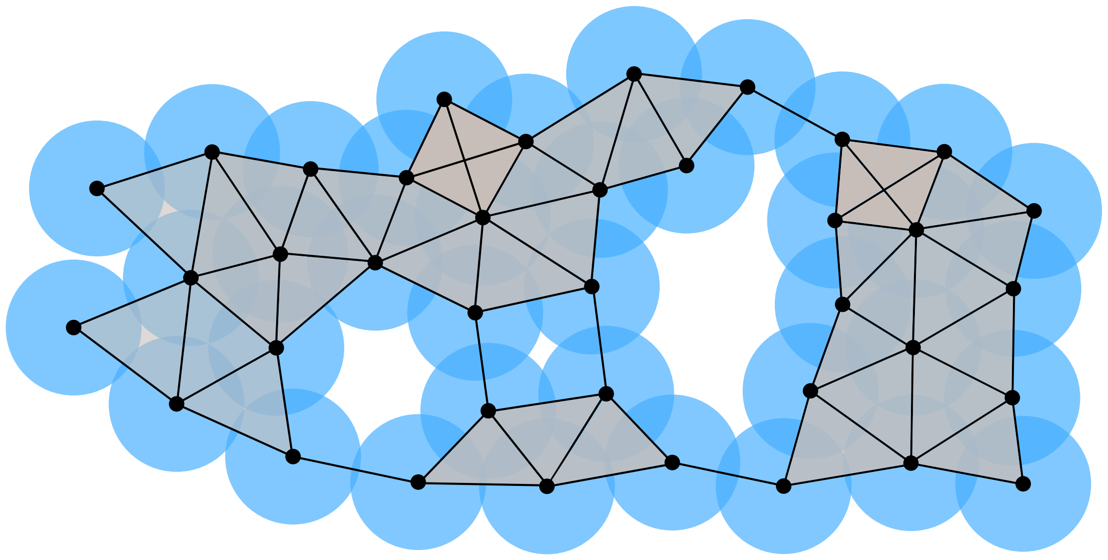


Figure 2.4: A Vietoris-Rips complex.

An explicit example showing the difference between a Čech and Vietoris-Rips complex can be seen in Figure 2.5. Although the Vietoris-Rips complex has more simplices, it is less expensive to compute when compared to the corresponding Čech complex. The Nerve theorem does not hold for Vietoris-Rips complexes, however for all $r > 0$ the following inclusions do hold: $\mathrm{C}_r \subset \mathrm{VR}_r \subset \mathrm{C}_{2r}$ (Theorem 2.5 [14]). Therefore, if the Čech complexes for both r and $2r$ are good approximations of the underlying space then it follows that the Vietoris-Rips complex is too.

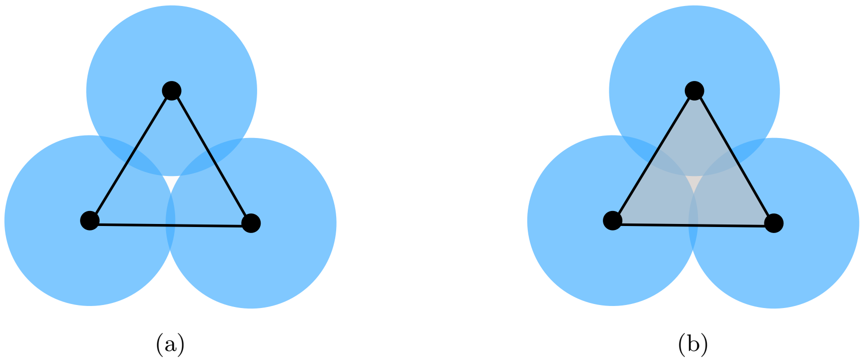


Figure 2.5: In the Čech complex (a), all balls of radius r must have a common point of intersection to form a 2-simplex while in the Vietoris-Rips complex (b) only pairwise intersections among the balls are needed.

2.3 Chains, Boundaries, Cycles

Let K be a simplicial complex in $\mathbb{R}^3$. A *k-chain* is a subset of k-simplices in K. Addition of chains is defined via integer coefficients modulo 2, i.e. the sum of two k-chains c and d is the symmetric difference of the two sets,

$$c + d = (c \cup d) - (c \cap d). \tag{2.1}$$

In addition, taking the symmetric difference is a commutative operation. We denote C_k as the set of all k-chains together with the addition operator. The zero element of C_k forms the empty set. For every integer k there is a chain group, however for a complex in $\mathbb{R}^3$ only the chain groups for $0 \leq k \leq 3$ may be non-trivial.

The *boundary* $\partial_k(\sigma)$ of a k-simplex σ is defined as the set of its $(k-1)$-dimensional faces which also forms a $(k-1)$-chain. To find the boundary of a k-chain we sum

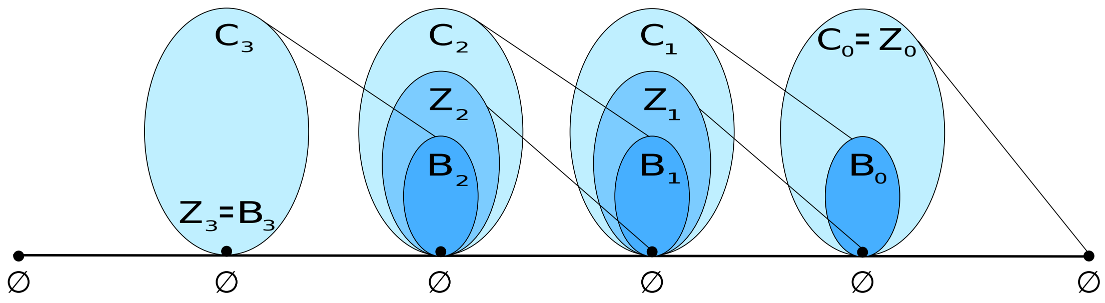


Figure 2.6: The chain, cycle, and boundary groups along with their images under the boundary operators.

of the boundaries of its simplices, $\partial_k(c) = \sum_{\sigma \in c} \partial_k(c)$. Each boundary operator is a homomorphism $\partial_k : C_k \to C_{k-1}$ defined as follows.

Definition 7. *A boundary homomorphism is the linear mapping $\partial_k : C_k(K) \to C_{k-1}(K)$ produced by associating each basis element of $C_k(K)$ to the formal sum of its oriented faces of dimension $k-1$.*

The collection of boundary operators on the chain groups form a chain complex,

$$\cdots \to \emptyset \to C_3 \overset{\partial_3}{\to} C_2 \overset{\partial_2}{\to} C_1 \overset{\partial_1}{\to} C_0 \to \emptyset \cdots .$$

The *kernel* of ∂_k is defined as the set of k-chains with empty boundary and the *image* of ∂_k is defined as the set of $(k-1)$-chains that are boundaries of k-chains:

$$\text{ker } \partial_k = \{c \in C_k \mid \partial_k(c) = \emptyset\}$$
$$\text{img } \partial_k = \{d \in C_{k-1} \mid \exists c \in C_k : d = \partial_k(c)\}.$$

A k-*cycle* is a k-chain in the kernel of ∂_k while a k-boundary is a k-chain in the image of ∂_{k+1}. The sets Z_k of k-cycles and B_k of k-boundaries, with addition, form subgroups of C_k. An important property of the boundary operators is that the boundary of every boundary is empty, $\partial_{k1} \circ \partial_k(c) = \emptyset$. Thus, the groups are nested, $B_k \subset Z_k \subset C_k$, as shown in Figure 2.6. Taking the boundary of a point yields the empty set. This implies that every 0-chain is also a 0-cycle, $Z_0 = C_0$. Since K is a complex in $\mathbb{R}^3$, there exist no non-empty 3-cycles or 3-boundaries, $Z_3 = B_3 = \{\emptyset\}$.

2.4 Homology Groups

Homology is an algebraic means to identify the holes in a topological space. It is based on the concept of a boundary homomorphism where simplicial homology encodes how simplices are attached to their lower dimensional faces.

The kth *homology group* is defined as the kth cycle group factored by the kth boundary group: $H_k = Z_k/B_k$. Its elements consist of the homology classes $c + B_k = \{c + b \mid b \in B_k\}$, for all $c \in Z_k$. The sum of two classes is $(c + B_k) + (d + B_k) = (c + d) + B_k$ and the zero element is $\emptyset + B_k = B_k$.

$H_k(K)$ is the quotient vector space whose generators are k-cycles (which correspond to k-dimensional boundaryless subcomplexes surrounding a hole) modulo the equivalence relation which states that two such k-cycles are the same (homologous) whenever they are the oriented boundary of a $(k + 1)$-dimensional subcomplex. Additionally, a main property of $H_k(K)$ is that its dimension (number of generators) corresponds to the number of k-dimensional holes in the simplicial complex.

The homology groups are vector spaces where a subset generates a vector space if every element is the sum of elements in the subset, and a basis is a minimal generating set. Although there are no canonical bases, all bases have the same size which corresponds to the rank of the group. Since taking symmetric differences is equivalent to adding modulo 2, the size of a group is equal to 2 raised to the power of its rank. Figure 2.7 shows an example of adding two 1-cycles.

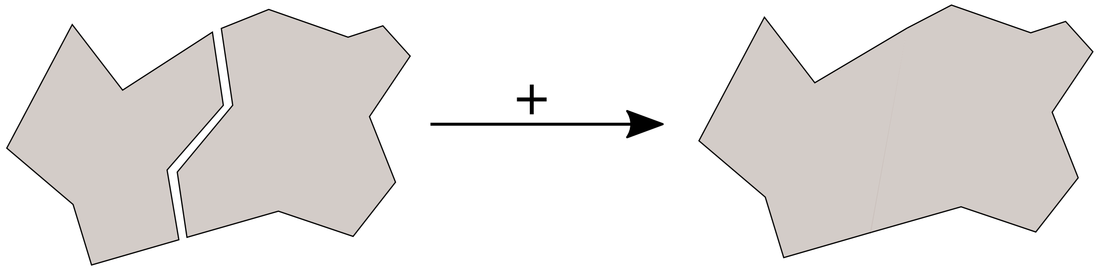


Figure 2.7: The symmetric difference between two 1-cycles produces a new 1-cycle.

The kth Betti number of K is defined as the rank of the kth homology group, $\beta_k = \text{rank } H_k$ where

$$\text{rank } H_k = \text{rank } Z_k - \text{rank } B_k. \tag{2.2}$$

For complexes in $\mathbb{R}^3$, only the Betti numbers for $0 \leq k \leq 2$ can be non-zero. A non-bounding 0-cycle represents a set of components of K where there is one basis element per component. Therefore, β_0 is the number of components that comprise K. A set of holes formed by K is represented by a non-bounding 1-cycle. Each hole can be expressed as a sum of holes in a basis and β_1 is the size of the basis. A non-bounding 2-cycle represents a set of voids in K which are components of $\mathbb{R}^3 - K$. Finally, because K is a complex in $\mathbb{R}^3$ there are no 3-cycles.

2.5 Computing Homology Groups

To compute the homology groups of a simplicial complex K we first encode the simplices in a *boundary matrix.* After reducing the boundary matrices we then compute the Betti numbers of K using Equation 2.2. The details of the boundary matrix creation and an example of computing homology follow.

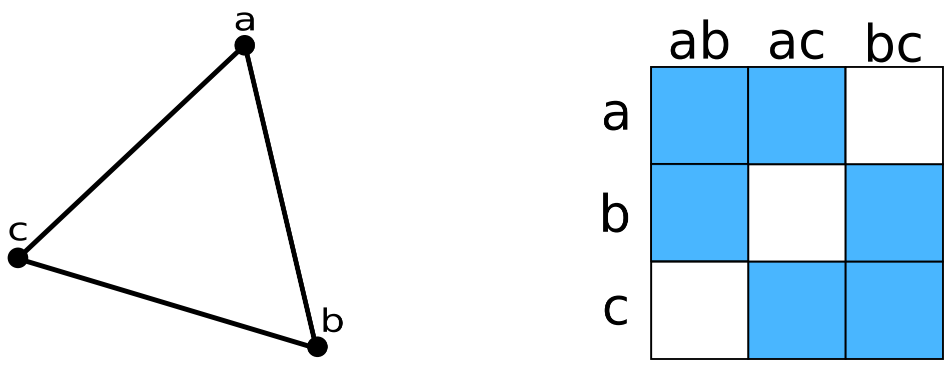
2.5.1 Boundary Matrix

To compute homology we represent the relationship between a simplex and its face using a boundary matrix, Figure 2.8. For each entry of a boundary matrix we store a 1 if simplex τ is a face of simplex σ, otherwise we store a 0. We then reduce the boundary matrix into Smith normal form to compute the Betti numbers. In normal form, the number of zero columns is the rank of the cycle group and the number of ones on the diagonal is the rank of the boundary group, Figure 2.9. The Betti number is the difference between the rank of the cycle and boundary groups, $\beta_k = \text{rank } Z_k - \text{rank } B_k$.

2.5.2 Example

Given the simplicial complex in Figure 2.8 (a), the boundary matrices are

$$\begin{aligned} \partial_0 &= \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}, \\ \partial_1 &= \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}. \end{aligned}$$



(a) (b)

Figure 2.8: A simplicial complex (a) and its boundary matrix representation (b).

After reduction, we have the following matrices

$$\bar{\partial}_0 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix},$$

$$\bar{\partial}_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

The Betti numbers are

$$\begin{aligned} \beta_0 &= 3 - 2 = 1, \\ \beta_1 &= 1 - 0 = 1, \end{aligned}$$

indicating that the complex contains one connected component and one hole.

2.6 Filters and Filtrations

An ordering of the simplices, where each prefix of the ordering contains the simplices of a subcomplex, is called a *filter*. A sequence of subcomplexes defined by taking successively larger prefixes is called a *filtration*. Figure 2.10 illustrates an example of a filtration consisting of 7 simplices. A filtration can be thought of as the evolution of a simplicial complex in which the sole element of change is growth.

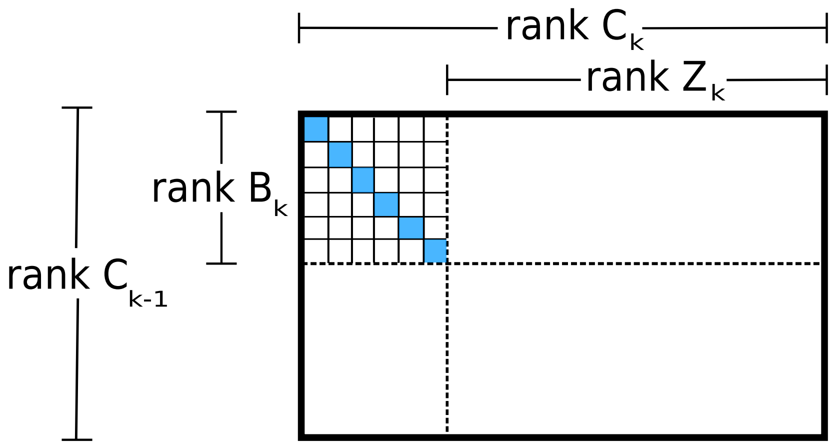


Figure 2.9: In the reduced boundary matrix, the number of zero columns is the rank of the cycle group, Z_k, and the number of ones on the diagonal is the rank of the boundary group, B_k. The Betti number is the difference between the rank of the cycle group and the rank of boundary group, $\beta_k = \text{rank } Z_k - \text{rank } B_k$.

Given a simplicial complex K, let $f : K \to R$ be a non-decreasing function. Here non-decreasing means that if τ is a face of σ, then $f(\tau) \leq f(\sigma)$. The level subcomplexes are then defined to be $K(\alpha) = f^{-1}(-\infty, \alpha]$. Let α_i denote the values of f on the simplices of K in increasing order, thus the level subcomplexes define a filtration of K. Furthermore, if $K_L = K(\alpha_L)$ where L is the index of the largest value α_L then

$$\emptyset \subset K_0 \subset K_1 \subset \ldots \subset K_L = K.$$

As the scale parameter increases, new simplices are added until the complete complex $K(\infty) = K$ is obtained. For simplicial complexes described in Section 2.2, we generate a filter and the resulting filtration by growing epsilon balls about each point.

2.7 Persistent Homology

Persistence is a way to measure topological attributes by their lifetime in a filtration. More precisely, let Z_k^l and B_k^l be the kth cycle group and kth boundary group, respectively, of the lth complex K^l in a filtration. To obtain the persistent cycles in K^l, its kth cycle group is factored by the kth boundary group of K^{l+p}, p complexes later in

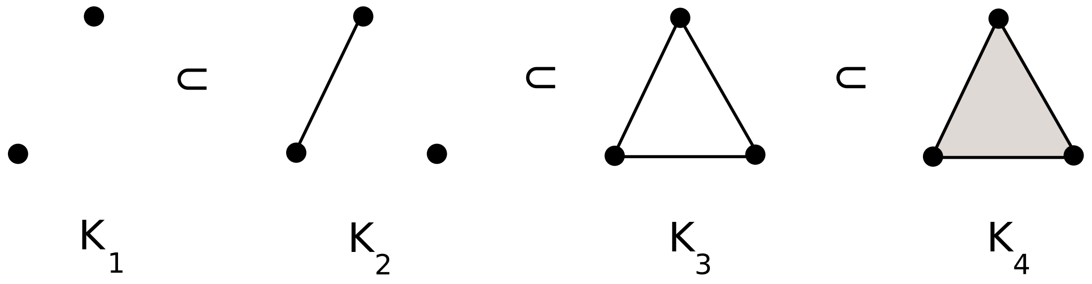


Figure 2.10: A filtration is made up of a sequence of subcomplexes.

the filtration. The *p-persistent* kth homology group of K^l is defined as

$$H_k^{l,p} = Z_k^l/(B_k^{l+p} \cap Z_k^l),$$

and the *p-persistent* kth Betti number $\beta_k^{l,p}$ of K^l is the rank of $H_k^{l,p}$. As p increases negative simplices cancel positive simplices earlier in the filtration, i.e. increasing p by one will shorten the persistence of all non-bounding cycles by one.

2.8 Computing Persistent Homology

To compute the persistent homology of a filtered simplicial complex K we first encode the complex in a *boundary matrix*. We then proceed to reduce the matrix using the *standard algorithm*. The details of these operations along with an example are given next.

2.8.1 Standard Algorithm

Similar to computing homology groups we use a boundary matrix to store the information regarding the faces of every simplex. We place a total ordering on the simplices of the complex in the filtration such that the following conditions hold:

- The face of a simplex precedes the simplex.
- The simplex in the ith complex K_i precedes simplices in K_j, $j > i$.

Let n be the total number of simplices in the complex and let $\sigma_1, \dots, \sigma_n$ be the simplices with respect to this ordering. We build a square matrix δ of size $n \times n$ by storing a 1 in

$\delta(i,j)$ if σ_i is a face of σ_j, otherwise we store a 0 in $\delta(i,j)$. Once the boundary matrix is constructed, we can then proceed to compute persistent homology.

Persistent homology can be computed by reducing the boundary matrix using Gaussian elimination. For every $j \in \{1,\ldots,n\}$ we define low(j) to be the largest index value i such that $\delta(i,j) = 1$. If column j consists of all 0 entries then low(j) is undefined. This process for reducing the boundary matrix is illustrated in Algorithm 2.1. In the worst case, the standard algorithm has a runtime complexity that is cubic in the number of simplices.

Algorithm 2.1 Standard Algorithm

```
Input: Boundary matrix $B$
Output: Reduced boundary matrix $\bar{B}$
1: for $i = 1$ to $n$ do
2:     while $\exists\ i < j$ and low($i$) = low($j$) do
3:         Add column $i$ to column $j$
4:     end while
5: end for
```

After the boundary matrix is reduced, we can find the persistence pairs (barcodes) as follows:

- If low(j) $= i$ then σ_i is paired with σ_j. The entrance of σ_i in the filtration corresponds to the birth of a feature that dies with the entrance of σ_j.
- If low(j) is undefined then the entrance of σ_j in the filtration causes the birth of a feature. If there exists a k such that low(k) $= j$ then σ_j is paired with σ_k. The entrance of σ_k causes the death of σ_j. If no such k exists then σ_j is unpaired.

The pair (σ_i, σ_j) gives the half-open interval $[i,j)$ and an unpaired σ_k gives the infinite interval $[k,\infty)$.

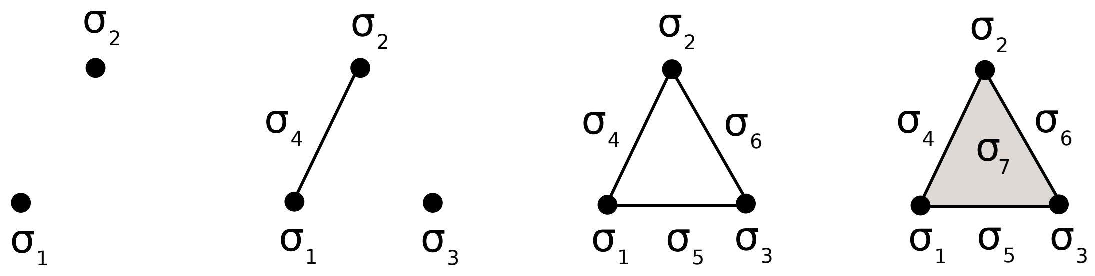


Figure 2.11: An ordering of the simplices that is compatible with the filtration in Figure 2.10.

2.8.2 Example

Working with the filtration in Figure 2.10 and its simplicial ordering in Figure 2.11, the boundary matrix is

$$B = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

The reduced boundary matrix is:

$$\bar{B} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

After reduction, we have the following intervals from $\bar{B}$:

- σ_1 is positive and unpaired. This gives the interval $[1, \infty)$ in H_0.

- σ_2 is positive and paired with σ_4. Since σ_2 and σ_4 enter at the same time in the filtration there is no interval.
- σ_3 is positive and paired with σ_5. This gives the interval $[2, 3)$ in H_0.
- σ_6 is positive and paired with σ_7. This gives the interval $[3, 4)$ in H_1.

Chapter 3

Fast Construction of the Vietoris-Rips Complex in $\mathbb{R}^3$

The topology of a point cloud can be described using a Čech or Vietoris-Rips (VR) complex. It is nontrivial to compute a Čech complex since we need very precise data on the distances between points and we need to check for a large number of intersections. Consequently, in many applications the Čech complex is unattainable, thus we work with the VR complex which only measures pairwise distances between points. Despite being simple to compute, constructing the VR complex is the primary bottleneck in a TDA pipeline due to the growth in the number of simplices as the complex is built at different scales. In this chapter, we focus on speeding up the construction of three dimensional VR complexes which have many practical applications.

3.1 Related Work

The VR complex is often computed using provisional algorithms due to its simplicity. These implementations may be sufficient for building low-dimensional complexes on small datasets consisting of a few hundred points in size. Nevertheless, even for low-dimensional complexes the computational cost quickly rises when the size of the dataset and the scale parameter increase.

Previous work by Zomorodian developed fast algorithms for creating complexes in arbitrary dimensions [20]. Zomorodian's approach separates the construction of the filtered VR complex into two phases. In the first phase, a neighborhood graph on the input points based on the ambient metric is computed. In the second phase, the graph is expanded to include higher-dimensional simplices. In contrast to this serial approach for any dimension, we optimize the construction of VR complexes in $\mathbb{R}^3$.

Open source libraries such as Dionysus, DIPHA, GUDHI, JavaPlex, and Perseus contain software for computing the VR complex [21]. Other recent software packages for computing VR complexes include Eirene [22] and Ripser [23]. We compare our work to Ripser, a lean implementation for computing VR persistence barcodes, which is considered the state of the art.

3.2 Problem Formulation

Suppose that $S \subseteq \mathbb{R}^3$ is a finite set of three dimensional points. The Vietoris-Rips complex of S at radius r is

$$\mathrm{VR}_r(S) = \{\sigma \subseteq S \mid \mathrm{d}(u,v) \leq r, \forall u \neq v \in \sigma\}, \tag{3.1}$$

where d is the Euclidean metric. Our approach is to separate the construction of the filtered VR complex into three parallelized phases. In the first phase, we perform a nearest neighbors search about each point. In the second phase, we use the results of the nearest neighbors search to create an edge list. In the final phase, we generate a triangle list based on the edge relationships among the points. The generated edge and triangle lists can then be used for persistent homology computations.

3.3 Nearest Neighbors Search

Searching for nearest neighbors is a classic computing problem. The idea is to preprocess S such that for a given query q the closest $p \in S$ can be quickly found Figure 3.1. Given a 3D point cloud, we begin by sorting the points by their xyz-coordinates. Next, we initialize and utilize a kd-tree to perform a nearest neighbors search about each point, Algorithm 3.2. Among the set of nearest neighbors returned from a radius search, we save the indices of the points whose position in the sort comes after the current point. These points within r distance of each other form the *r-neighborhood graph* of S whose

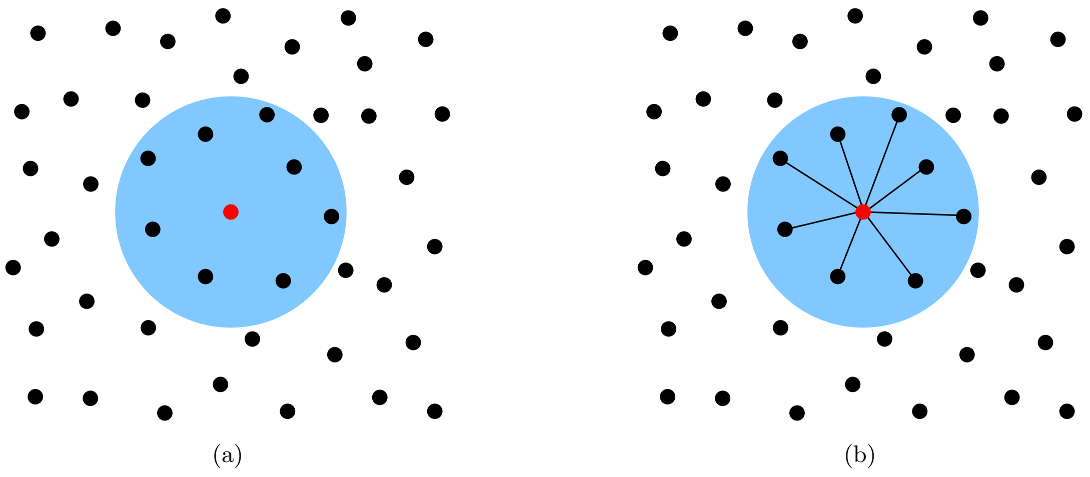


Figure 3.1: In the first phase of constructing the VR complex the neighborhood of each point (a) is searched for neighboring points (b).

edges in the VR complex are

$$\{(u, v) \in S \times S \mid u \neq v \text{ and } \mathrm{d}(u, v) \leq r\}.$$

3.4 Edge List Construction

The product of computing nearest neighbors is an array where the indices correspond the positions of the sorted points and the entries contain the number of nearest neighbors (edges) per point. We proceed to find the total number of edges by performing a reduction on the array. To find the offset of each set of edges, for each point, an all-prefix-sums operation is performed [24]. From the prefix sums array a variable length edge list is created, Figure 3.2. The list is then populated with edge information in parallel, Algorithm 3.3.

3.5 Triangle List Construction

Points within distance r of each other create edges and these edges constitute the r-neighborhood graph of S. We identify triangles by finding all three cliques in the

Algorithm 3.2 Nearest Neighbors

```
    Input: Set of points $S$
    Output: $\{(u, v) \in S \times S \mid u \neq v$ and $d(u, v) \leq r\}$
 1: sort(cloud.points)
 2: kd-tree.buildIndex(cloud.points)
 3: for $i \in [1, |\text{cloud.points}|]$ do
 4:     query ← [cloud.points[i].x, cloud.points[i].y, cloud.points[i].z]
 5:     nearest_neighbors ← radiusSearch(query, radius, kd-tree)
 6:     for $j \in [1, |\text{nearest\_neighbors}|]$ do
 7:         if $i <$ nearest_neighbors[j] then
 8:             cloud.points[i].nearest_neighbors ← j
 9:             cloud.points[i].n_edges ← cloud.points[i].n_edges + 1
10:         end if
11:     end for
12: end for
```

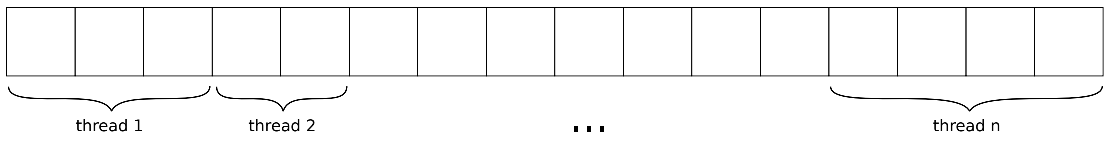


Figure 3.2: A prefix sum of offsets allows for the creation of a variable length list that can be written in parallel.

Algorithm 3.3 Edge List

```
    Input: $\{(u, v) \in S \times S \mid u \neq v$ and $\mathrm{d}(u, v) \leq r\}$
    Output: Populated variable length edge list
 1: for $i \in [1, |\text{cloud.points}|]$ do
 2:     offset ← cloud.points[i].edge_offset
 3:     for $j \in [1, |\text{cloud.points[i].nearest\_neighbors}|]$ do
 4:         cloud.edges[offset].u ← i
 5:         cloud.edges[offset].v ← cloud.points[i].nearest_neighbors[j]
 6:         cloud.edges[offset].negative ← 0
 7:         cloud.edges[offset].death_index ← 0
 8:         offset ← offset + 1
 9:     end for
10: end for
```

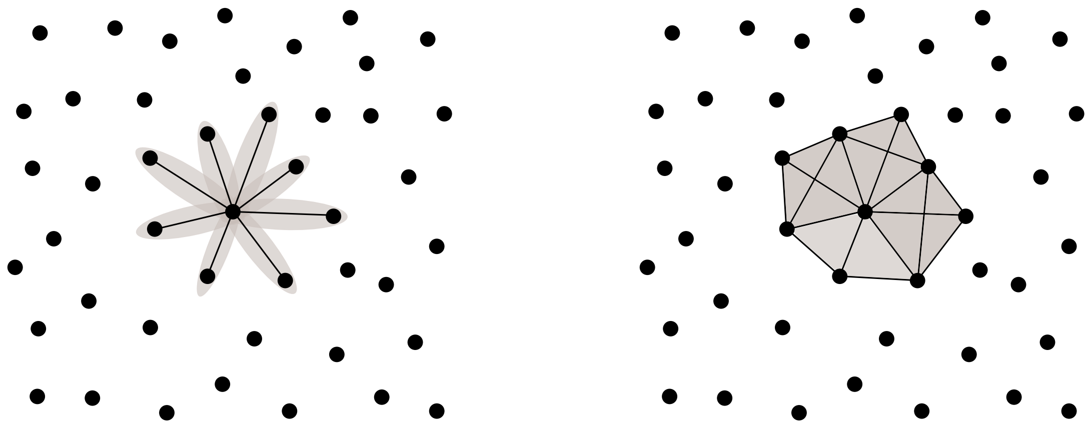


Figure 3.3: In the second and third phases of constructing the VR complex the edge (a) and triangle (b) cliques, highlighted by the colored regions, are determined.

r-neighborhood-graph as follows. Suppose $u, v, w \in S$. If edges (u, v) and (u, w) exist then we have a triangle if and only if v and w are neighbors (i.e. (v, w) exists), Algorithm 3.4. The total number of triangles and their offsets are computed using a reduction and all-prefix-sums operations, respectively. After the resulting operations, a variable length triangle list is constructed (Figure 3.2) and initialized in parallel using Algorithm 3.5.

Algorithm 3.4 Compute Triangles

```
Input: $\{(u, v) \in S \times S \mid u \neq v$ and $\mathrm{d}(u, v) \leq r\}$
Output: List of edge offsets comprising each triangle
for $i \in [1, |\text{cloud.points}|]$ do
    for $j \in [|\text{cloud.points[i].nearest\_neighbors}|]$ do
        v ← cloud.points[i].nearest_neighbors[j]
        for $k \in [j + 1, |\text{cloud.points[i].nearest\_neighbors}|]$ do
            w ← cloud.points.[i].nearest_neighbors[k]
            for $l \in [1, |\text{cloud.points[v].nearest\_neighbors}|]$ do
                if w = cloud.points[v].nearest_neighbors[l] then
                    cloud.points[i].uv_offset ← cloud.points[i].edge_offset + j
                    cloud.points[i].uw_offset ← cloud.points[i].edge_offset + k
                    cloud.points[i].vw_offset ← cloud.points[v].edge_offset + l
                    cloud.points[i].n_triangles ← cloud.points[i].n_triangles + 1
                end if
            end for
        end for
    end for
end for
```

3.6 Analysis

Although we only need to check pairwise distances the VR complex has the same worst-case complexity as the Čech complex. In the worst-case, the VR complex can have up to $2^{|S|} - 1$ simplices and dimension $|S| - 1$. However, in applications we usually compute the VR complex up to dimension $k \ll |S| - 1$ where $k = 1$ or $k = 2$.

Algorithm 3.5 Triangle List

```
Input: List of edge offsets comprising each triangle
Output: Populated variable length triangle list
1: for $i \in [1, |\text{cloud.points}|]$ do
2:     offset ← cloud.points[i].triangle_offset
3:     for $j \in [1, |\text{cloud.points[i].v}|]$ do
4:         cloud.triangles[offset].uv_offset ← cloud.points[i].uv_offset[j]
5:         cloud.triangles[offset].uw_offset ← cloud.points[i].uw_offset[j]
6:         cloud.triangles[offset].vw_offset ← cloud.points[i].vw_offset[j]
7:         cloud.triangles[offset].negative ← 0
8:         offset ← offset + 1
9:     end for
10: end for
```

3.7 Experiments

In this section we evaluate our parallel VR (PVR) construction, implemented in C++, against Ripser[1] a publicly available C++ software program for the computation of VR persistence barcodes.

3.7.1 Experimental Setup

Searching for nearest neighbors is done using nanoflann [25], a fast library for building kd-trees of datasets. We also make use of OpenMP, a set of compiler directives and callable runtime library routines for invoking parallelism and the CUDA parallel computing platform. All experiments were performed using a single node 64-bit GNU/Linux machine with 24 CPU cores and an NVIDIA Tesla K40 GPU.

3.7.2 Experimental Results

The performance of the PVR complex construction compared to Ripser on point clouds ranging from 15k to 100k points is shown in Figure 3.3. For each plot, the size of the complex (number of simplices) as a function of the radius and the construction time

[1] https://github.com/Ripser/

	15k Point Cloud	30k Point Cloud	60k Point Cloud	100k Point Cloud
PVR	0.401 ±0.26	3.215 ±4.87	3.32 ±4.26	9.018 ±12.82
Ripser	8.728 ±0.37	45.098 ±1.82	141.37 ±2.73	474.24 ±4.74

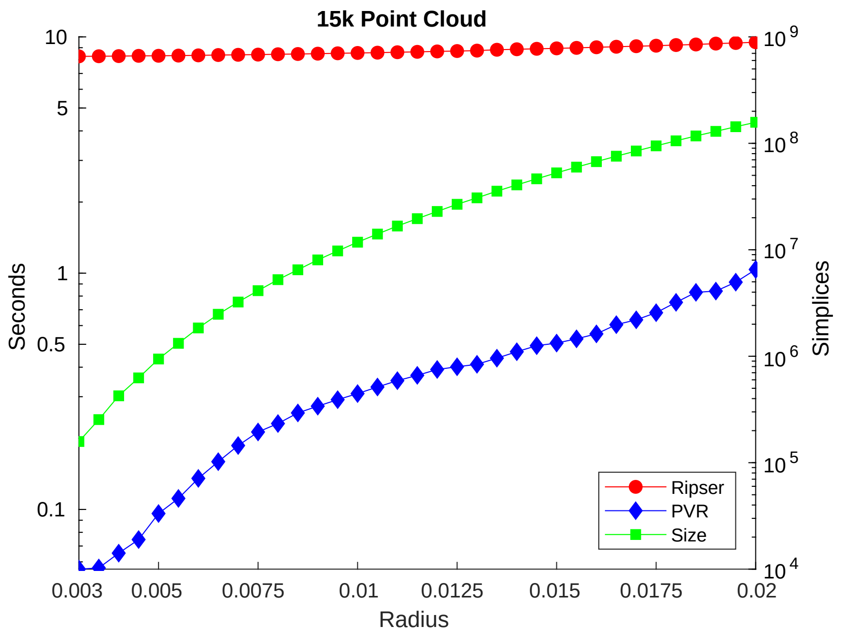
Table 3.1: The mean runtime results (seconds) of PVR versus Ripser on various sized 3D point clouds.

(seconds) is shown for both PVR and Ripser. Table 3.1 shows the overall mean runtime between PVR and Ripser for each point cloud.

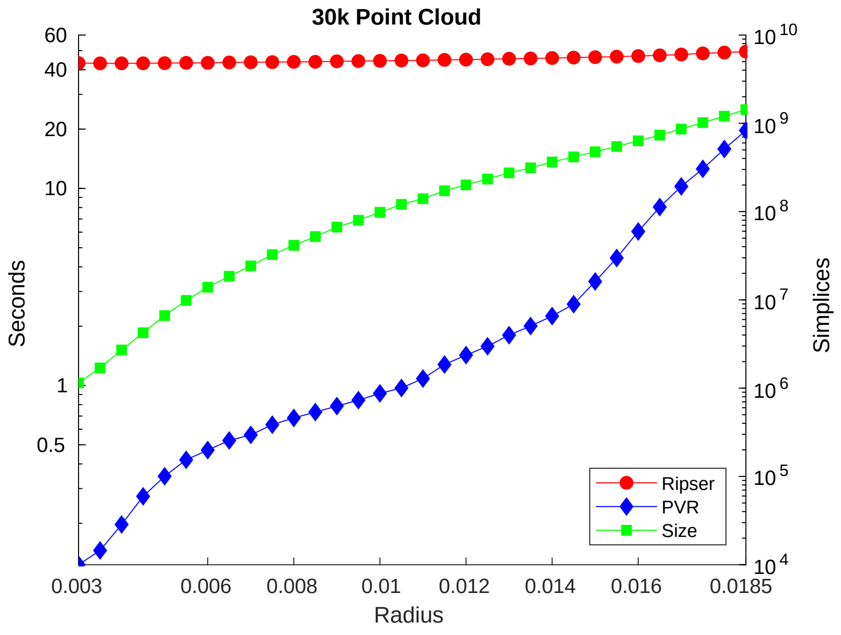
From the plots in Figure 3.3 we can see that the runtime of PVR, through the range of filtration values, remains below the growth of the complex as it approaches a billion simplices. In comparison to Ripser, PVR is significantly faster on various sized point clouds. However, we note that the design goals between these two implementations of the VR complex differ. Ripser was designed for computing VR complexes, in any dimension, with memory efficiency in mind. On the other hand, PVR was conceived for constructing VR complexes as quickly as possible for real-time 3D point cloud processing applications.

3.8 Conclusion

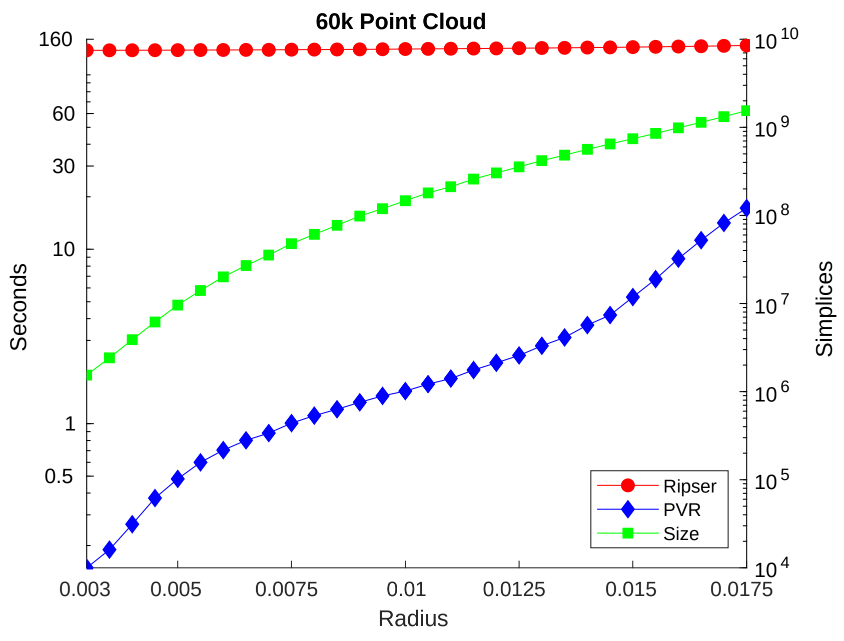
This chapter developed algorithms and data structures for the parallel assembly of a VR complex in $\mathbb{R}^3$. Fast construction of the VR complex, having many important applications, is a major hurdle in TDA due to its high computational costs. We've shown the increased performance of constructing a 3D VR complex, with a speedup over a state of the art implementation, for practical point cloud processing tasks.


15k Point Cloud
Seconds
10
5
1
0.5
0.1
Simplices
10^9
10^8
10^7
10^6
10^5
10^4
0.003
0.005
0.0075
0.01
0.0125
0.015
0.0175
0.02
Radius
Ripser
PVR
Size

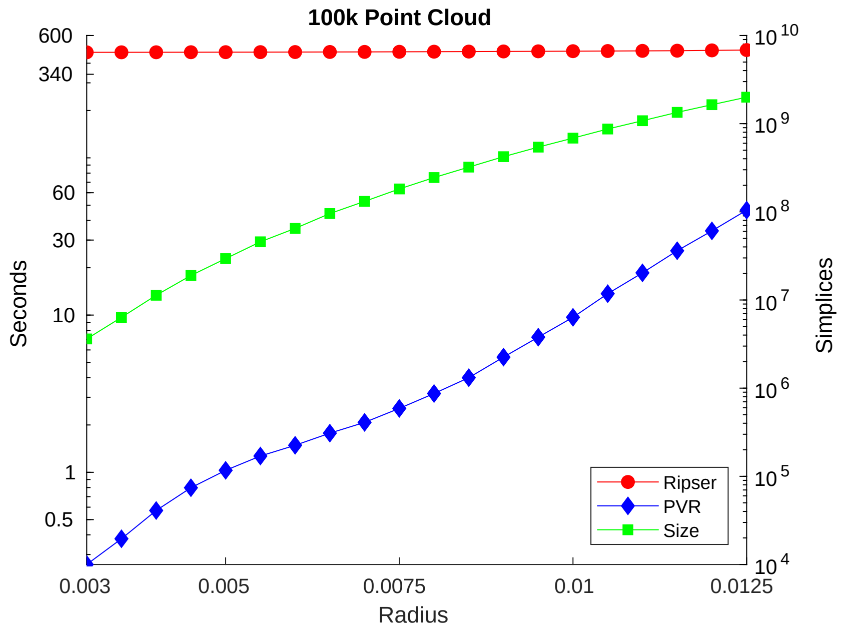
(a)


30k Point Cloud
Seconds
60
40
20
10
1
0.5
Simplices
10^{10}
10^9
10^8
10^7
10^6
10^5
10^4
0.003
0.006
0.008
0.01
0.012
0.014
0.016
0.0185
Radius
Ripser
PVR
Size

(b)



(c)



(d)

Figure 3.3: The construction time of Parallel Vietoris-Rips (PVR) and Ripser along with the size of the complex for a given radius using a set of non-uniformly sampled points in $\mathbb{R}^3$. The results of each plot are averaged over 10 independent runs per plot point.

Chapter 4

Point Cloud Segmentation

3D point clouds can be filtered, segmented, compressed, etc., to accomplish the job at hand. Segmentation algorithms aim to divide a point cloud into constituent objects or regions that are perceptually meaningful. The act of segmentation is a vital preprocessing step in computer and robotic vision systems. The performance of high level tasks such as object localization, feature extraction, and classification are dependent upon the quality of the segmented data. This chapter presents techniques for performing object and region segmentation of 3D points clouds utilizing topological persistence.

4.1 Object Segmentation

Segmenting objects in point clouds is a challenging problem. 3D point clouds are often incomplete, sparse, unorganized, lack connection information, and have an uneven dispersion. In addition, object clusters can be highly entangled and the topological features of an object's surface can be arbitrary with no statistical distribution pattern in the data. Real-world sensor data is noisy. The physical limitations of sensors, boundaries between features, multiple areas of reflectance, occlusions, etc., lead to the creation of outliers that make the process of segmentation difficult.

In this section, we implement and evaluate algorithms for segmentation based on computing the topological persistence of a point cloud dataset at different spatial resolutions. Our key contributions are:

- The introduction of persistent homology to the area of 3D point cloud processing.

- A novel approach for segmenting 3D point clouds based on topological persistence.

4.1.1 Related Work

Data segmentation has been a heavily worked on research problem for decades with a rich history of literature. Many different approaches have been taken in segmenting both 2D and 3D data. In the following paragraphs we highlight several areas of research that are relevant to our work.

A variety of algorithms have been proposed in computer graphics on the segmentation of 3D models of single objects where the objects are typically represented by meshes [26]. These algorithms include watersheds [27], k-means [28], hierarchical [29], and spectral clustering [30]. The objective of these methods is to decompose an object into meaningful parts. Typically a graph is constructed from an input mesh. Clustering is then performed on the graph to produce a segmentation using graph cuts.

Prior to the appearance of inexpensive RGB-D sensors, datasets produced by laser scanning and stereo vision technologies fueled segmentation-based research. Golovinskiy and Funkhouser segment foreground objects from background clutter in outdoor scenes using a min-cut method on a nearest neighbors graph [31]. A multi-class point cloud segmentation technique, developed by Johnson-Roberson et al., proposes multiple seeding methods and a min-cut framework [32]. In [33], Douillard et al. demonstrate a set of segmentation methods for 3D point clouds of varying densities.

With the introduction of RGB-D sensors, interest in segmenting point clouds has continued to increase. Pre-segmentation based on surface normals followed by surface patch estimation and model selection to find a learned representation for the given data is presented by Richtsfeld et al. [34]. A segmentation strategy by Mishra et al. extracts objects, defined as compact regions enclosed by the depth and contact boundary in a scene, as single regions using color, texture, and 3D information [35]. Efficient planar segmentation of organized point clouds based on connected components is proposed by Trevor et al. [36].

The idea of topological persistence has been used in computer vision for performing image segmentation. Hierarchical segmentation using the mean shift method is discussed by Paris and Durand [37]. The authors apply these concepts to a density function in a 5-dimensional space that combines the x- and y-coordinates of a pixel along with its

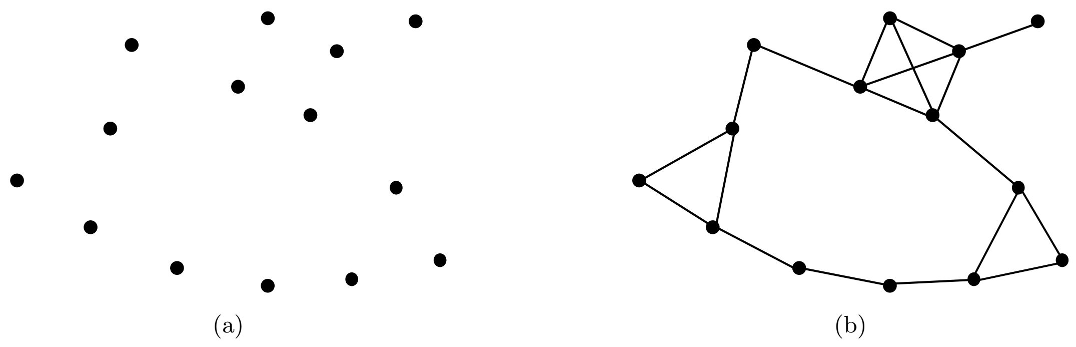


Figure 4.1: A point cloud (a) and its approximation as the 1-skeleton of a VR complex (b).

RGB color components. A way to integrate knowledge about topological properties into a random field segmentation model is introduced by Chen et al. [38]. In contrast to these works on 2D images, we utilize topological persistence to support 3D point cloud processing tasks.

4.1.2 Problem Formulation

Let $X = \{x_0, \ldots, x_{m-1}\} \in \mathbb{R}^3$ be a topological space and let $x_0, \ldots, x_{m-1}$ denote the points in a point cloud captured by a depth sensor. To segment out the objects of X we have the following objectives. First, we represent the topology of the space as a simplicial complex. Next, we compute the zero-dimensional homology group of X using a filtration. Lastly, the connected component clusters are extracted from the data.

To achieve these objectives, we make the following assumptions:

Assumption 1. *The input space is voxelized: points in the space are approximated (downsampled) with their centroid.*

Assumption 2. *The input space is locally connected: each point consists of a neighborhood containing open and connected sets. This constraint is enforced by using only the neighboring points during the segmentation process.*

4.1.3 Vietoris-Rips Complex Construction

Given an input point cloud our goal is to form the Vietoris-Rips (VR) complex, $\mathrm{VR}_r(X) = \{\sigma \subseteq X \mid \mathrm{d}(x_i, x_j) \leq r, \forall x_i \neq x_j \in \sigma\}$, where d is the Euclidean metric and the vertices of σ are pairwise within distance r of each other. We observe that the 1-skeleton of the VR complex is sufficient to compute the zeroth homology group of the space at each step in the filtration, Figure 4.1. Thus, to generate $\mathrm{VR}_r(X)$ we construct a set of edges as outlined in Algorithm 4.6. The input to the algorithm is an ordered set of points. A kd-tree is instantiated and used to carry out range searches for finding the nearest neighbors of each point at a given radius.

Algorithm 4.6 Compute Connected Components

```
  Input: Ordered set of points $X$
  Output: One or more sets of points each representing a connected component
 1: kd-tree.buildIndex(cloud.points)
 2: while radius $\leq$ max_distance do
 3:     for $i \in [1, |\text{cloud.points}|]$ do
 4:         query ← [cloud.points[i].x, cloud.points[i].y, cloud.points[i].z]
 5:         nearest_neighbors ← radiusSearch(query, radius, kd-tree)
 6:         min_nn ← min(nearest_neighbors)
 7:         for $j \in [1, |\text{nearest\_neighbors}|]$ do
 8:             nn ← nearest_neighbors[j]
 9:             if cloud.points[nn].parent > cloud.points[min_nn].id then
10:                 root ← find(cloud.points[nn])
11:                 if cloud.points[root].id = cloud.points[nn].id then
12:                     cloud.points[nn].death ← radius
13:                 end if
14:                 union(cloud.points[nn], cloud.points[min_nn])
15:             end if
16:         end for
17:     end for
18:     radius ← radius + step_size
19: end while
```

4.1.4 Persistent Homology Computation

In Section 2.8, we saw how the standard algorithm can be used for computing persistent homology. Alternatively, to compute the zero-dimensional homology of a complex we can use a modified form of a disjoint-set data structure [39] as follows. Let $G = (V, E)$ be a graph with a function f and an ordering on the points $u, v \in V$ such that $f(u) < f(v)$. The original version of the disjoint-set data structure tracks the components of the graph. In our modified version, we care about the order in which the points are added and the function value on the points which causes the components to merge.

To populate the data structure we add points, one at a time, in the order of their function values. For each point u, the data structure stores a pointer to its parent, denoted u.parent. The data structure is initialized by making each point the root of its own tree, u.parent $= u$. We then use the following functions during the filtration process:

- **find**(u): Returns the root of the tree containing u by recursively following u.parent until it finds the point who's parent is itself.
- **union**(u, v): Merges the trees containing u and v into one or does nothing if u and v already belong to the same tree. This operation is performed by calling find(u) and find(v) to determine the roots of the respective trees. If the roots of the two points are equal, then the components are contained in the same tree and nothing is done. Conversely, if the roots differ then the function sets one to be the parent of the other.

Initially, all points are born at time zero and die when merged in the data structure. The death of a point is tracked as follows. If the call to union(u, v) determines that the points belong to different trees, we set a.parent $= b$ where a and b are the roots of the respective trees and a.birth $>$ b.birth. The death time of the younger component, a, is set to be the function value of b.

4.1.5 Connected Component Extraction

At the end of the filtration, we find the connected components based on the sets of points that are joined to 0-dimensional simplices of infinite persistence. Within the data structure, a simplex of infinite persistence is a point with no recorded death time and represents the root of a tree. Each extracted connected component is equivalent to a segmented object in the point cloud.

4.2 Region Segmentation

Within a 3D point cloud, a region can be defined as a group of connected points with similar properties. The concept of a region is essential in interpreting point cloud data since regions may correspond to objects or parts of an object in a scene. Segmentation of regions is a crucial preprocessing step towards pattern recognition and scene understanding.

Region segmentation algorithms are based on the expansion of a region whenever its interior is homogeneous according to certain features such as intensity, color, or texture. Unlike edge-based segmentation, which returns boundaries between regions, region-based segmentation is a method that allows the determination of regions directly.

Region growing is one of the simplest and most popular algorithms for region-based segmentation. Traditional implementations start by choosing a starting point called a seed. Then, the region is grown by adding similar neighboring points according to a homogeneity criterion, thus increasing the size of the region step by step. The homogeneity criterion has the function of determining whether a point belongs to the growing region or not. The decision of merging is generally taken based only on the difference between the evaluated point and the region. However, it is not easy to decide when this variance is small (or large) enough to make a decision.

Defining a predicate for a homogeneity criterion which controls the region growing process involves the interplay between local and global considerations. Neighbors are joined together by a local or regional decision process, however what is desired is a satisfactory global result when the algorithm terminates. In general, this dilemma can be resolved by using as much global information, as is available, to make local decisions.

In this section, we present a new 3D region segmentation technique utilizing persistent homology that builds upon ideas in Section 4.1. Our main contribution is:

- The novel combination of global (topological) and local (color, surface normal) information, to produce a stable region growing segmentation of a 3D point cloud.

This approach is fully automated and lacks the requirement of an initial seeding. Furthermore, the final segmentation does not depend on the order in which the regions are grown or joined.

4.2.1 Related Work

Region-based segmentation is a classic technique in computer vision and image processing with over forty years of history [40, 41]. 'Region growing' and 'split and merge' are the two most common region-based segmentation algorithms. In the following paragraphs we summarize several domains of region segmentation research that are relevant to our work.

The notion of seeded region growing was introduced by Adams and Bischof [42]. Later work by Lin et al. furnishes algorithms that do not require an initial seeding of the data [43]. A region growing sequence built by incrementing a maximal homogeneity threshold from a small to large value is described by Revol et al. [44]. Their work uses an assessment function to determine the optimal homogeneity criterion.

Segmentation of regions is often used in the processing of medical images. A framework for segmenting 3D imaging data volumes, developed by Justice et al., uses interactively guided initial seed points [45]. Interactive region segmentation with graph cuts is proposed by Boykov and Jolly [46]. The approach requires that a user impose constraints for segmentation by indicating which seeds must be part of an object and which seeds must be part of the background. The rest of the image is segmented automatically by computing a global optimum among all segmentations satisfying the constraints. In a work by Pohle and Toennies, a region growing algorithm is introduced based on a model that describes homogeneity and simple shape properties [47].

Tremeau and Borel present a color segmentation algorithm that combines region growing and merging processes [48]. The algorithm generates a segmentation of an image into spatially disconnected but similar in color regions. A method that simultaneously segments and models a point cloud through a minimum spanning tree ellipsoidal region growing process is presented by Pauling et al. [49].

A statistically robust segmentation algorithm for planar and non-planar surfaces extracted from laser scanning data is proposed by Nurunnabi et al. [50]. The algorithm uses principle components analysis (PCA) to make saliency features used for region growing more resistant to outliers. Holz and Behnke present a fast segmentation method for range images and organized point clouds [51]. The main idea, geared towards planes, is to approximately reconstruct the surface by means of polygonal meshing and then segment the resulting mesh by growing regions about local neighborhoods.

A hybrid split-and-merge segmentation method using persistent homology is described by Letscher and Fritts [52]. The algorithm first performs edge detection on the input image. Then, the image is split into regions using edge-directed topology where regions with similar features are selected and merged in order of topological persistence.

4.2.2 Problem Formulation

Given a topological space $X = \{x_0, \ldots, x_{m-1}\} \in \mathbb{R}^3$ where $x_0, \ldots, x_{m-1}$ are the points in a point cloud captured by an RGB-D sensor, our goal is to partition X into disjoint subsets $X_1, X_2, \ldots, X_N$ such that

$$\bigcup_{i=1}^{N} X_i = X \tag{4.1}$$

$$X_i, \quad i = 1, 2, \ldots, N \text{ is connected} \tag{4.2}$$

$$P(X_i) = \text{ TRUE for } i = 1, 2, \ldots, N \tag{4.3}$$

$$P(X_i \cup X_j) = \text{ FALSE for } i \neq j, X_i, X_j \text{ adjacent}, \tag{4.4}$$

where P is a logical predicate defined on a set of proximate points.

The predicate P associated with the third and fourth conditions determines what type of attributes the segmented regions must have to satisfy the homogeneity criterion. The second condition implies that regions must be connected, i.e. constructed with contiguous points. This requirement affects the central structure of the segmentation algorithm, especially the region growing process, by enforcing the constraint that points are processed according to neighbor relationships. The first condition simply states that the space is composed of the disjoint regions.

4.2.3 Homogeneity Criterion

Our criterion of homogeneity employs the analysis of the color characteristics of the nearest neighbors. The CIELAB color space, defined by the International Commission on Illumination and chosen for its perceptually uniform color distances, is used to compare the chromaticity between two neighboring points

$$\Delta E = \sqrt{(l_2 - l_1)^2 + (a_2 - a_1)^2 + (b_2 - b_1)^2}, \tag{4.5}$$

where l corresponds to the luminance channel, a and b are the color channels, and ΔE is the Euclidean distance.

In addition, a similarity measure using the surface normal of a point is used. We estimate the coordinates of the surface normal by performing an eigendecomposition of the covariance matrix created from the nearest neighbors. The angle between a surface normal u and its neighbor v is computed as

$$\theta = \arccos \frac{u \cdot v}{\|u\| \|v\|}. \tag{4.6}$$

4.2.4 Region Growing Algorithm

The homogeneity criterion formulated in the previous subsection is used in conjunction with topological persistence to perform region growing. Regions are grown by constructing the 1-skeleton of the VR complex as described in 4.1.3. As regions are grown based on local similarities between nearest neighbors, the global connectedness of the region is preserved using topology. To enforce the constraint that each disjoint region is a connected component we compute the zeroth homology group of the complex at each step of the filtration, Algorithm 4.7.

Similar to Algorithm 4.6, the region growing algorithm takes as input an ordered set of points, i.e. $u, v \in V$ and $f(u) < f(v)$. Range queries for finding the nearest neighbors of each point at a given radius are done with a kd-tree. For all nearest neighbors we compute the similarity measures based on Equations 4.5 and 4.6. The indices of the nearest neighbors that satisfy the homogeneity criterion are saved.

Among the indices of similar nearest neighbors we compare the parent function value of the current nearest neighbor to the nearest neighbor with the minimum function value. We proceed to find the root of the similar nearest neighbor point if the parent function value is greater (younger). When a point has not been previously joined to any other point, we record the radius (death) at which the points are to be connected. In the last step, we take the union of the pair by setting the parent of the point with the larger (younger) function value to be a sibling of the point with the smaller function value.

Algorithm 4.7 Grow Homogeneous Regions

```
Input: Ordered set of points $X$
Output: One or more sets of points each representing a homogeneous region
kd-tree.buildIndex(cloud.points)
while radius $\leq$ max_distance do
    for $i \in [1, |\text{cloud.points}|]$ do
        query $\leftarrow$ [cloud.points[i].x, cloud.points[i].y, cloud.points[i].z]
        nearest_neighbors $\leftarrow$ radiusSearch(query, radius, kd-tree)
        min_nn $\leftarrow$ min(nearest_neighbors)
        for $j \in [1, |\text{nearest\_neighbors}|]$ do
            $\theta \leftarrow$ computeNormalAngle()
            $\Delta E \leftarrow$ computeColorDistance()
            if $\theta \in (\theta_{min}, \theta_{max})$ and $\Delta E \in (\Delta E_{min}, \Delta E_{max})$ then
                similar_nearest_neighbors $\leftarrow$ nearest_neighbors[j]
            end if
        end for
        min_nn $\leftarrow$ min(similar_nearest_neighbors)
        for $j \in [1, |\text{similar\_nearest\_neighbors}|]$ do
            similar_nn $\leftarrow$ similar_nearest_neighbors[j]
            if cloud.points[similar_nn].parent $>$ cloud.points[min_nn].id then
                root $\leftarrow$ find(point(similar_nn))
                if cloud.points[root].id $=$ cloud.points[similar_nn].id then
                    cloud.points[similar_nn].death $\leftarrow$ radius
                end if
                union(cloud.points[similar_nn], cloud.points[min_nn])
            end if
        end for
    end for
    radius $\leftarrow$ radius $+$ step_size
end while
```

4.2.5 Region Growing Post-processing

At the end of region growing, we may have regions that are smaller than a predefined region size. To satisfy property (4.1) we post-process the remaining regions as follows. For each region representative, a nearest neighbors query is run. The point from the nearest neighboring region that best satisfies the homogeneity criterion is selected. A union operation is then performed thus joining the two regions. In practice, sufficiently small regions can be treated as noise and removed resulting in a cleaner segmentation of the data.

4.2.6 Region Extraction

The outer loop of Algorithm 4.7 is run for each step in the filtration. At each step the 0-dimensional homology group of the complex, which corresponds to the number of disjoint regions, is computed. Upon completion of the filtration we extract the segmented regions based on the sets of points that are joined to 0-dimensional simplices of infinite persistence, i.e. points with no recorded death time.

4.3 Analysis

Our disjoint-set data structure for computing and storing 0-cycles requires $\mathcal{O}(n)$ operations where n is the number of 0-simplices. However, by making use of weighted merging for *union* and path compression for *find*, the disjoint-set data structure can operate in nearly constant time. More precisely, the amortized time per operation is $\mathcal{O}(\alpha^{-1}(n))$ where $\alpha^{-1}(n)$ is the extremely slow growing inverse of the Ackermann function. Therefore, computing 0-dimensional homology can be done in $\mathcal{O}(n \cdot \alpha^{-1}(n))$. In addition, we can significantly speedup up each filtration step by precomputing the nearest neighbors and similarity measures for each point in parallel.

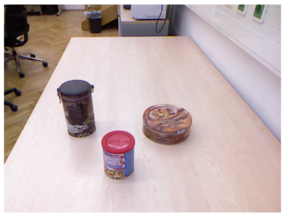
4.4 Object Segmentation Experiments

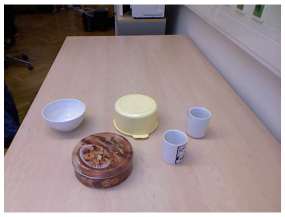
In this section, we perform an evaluation of the object segmentation method outlined in Section 4.1. The experiments are conducted using the Object Segmentation Database (OSD) [1]. The OSD provides RGB-D data in several subcategories to enable evaluation

(a) Scene 1

(b) Scene 2

(c) Scene 3


(d) Scene 4


(e) Scene 5

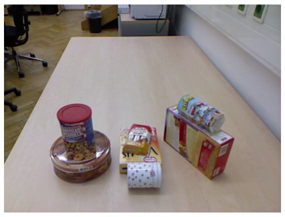

(f) Scene 6

Figure 4.2: The RGB images of selected scenes from the Object Segmentation Database [1] used for the object segmentation experiments.

of object segmentation approaches. The six scenes selected from the database for the experiments are shown in Figure 4.2. These scenes, labeled 1-6, contain a mixture of occluded and stacked objects.

4.4.1 Experimental Setup

All experimental runs were done using MATLAB on a 64-bit GNU/Linux machine with a single CPU core. Before segmentation occurs the data is preprocessed as follows. First, we remove outlying points (background clutter) that are beyond range of the objects of interest. Next, we find and remove all points that correspond to a planar surface upon which the objects are resting. We then downsample the point cloud with a voxelgrid filter using 1 cm^3 voxels. Finally, we construct the VR complex, compute homology, and extract the connected components.

4.4.2 Experimental Results

The results of segmenting the six scenes can be seen in Figure 4.2. For each scene, we show the filtered representation of the point cloud prior to segmentation followed by the color coded clusters extracted after segmentation. We set a threshold of 32 points as the minimum number of points to be considered a cluster. The filtration is performed for 10 steps up to the maximum distance value set for the complex. Note that under the existence of noise and occlusions in the dataset, the topological filtration process cleanly segments each object cluster. However, this method fails to segment the individually stacked objects in scene 6. Therefore, the segmentation procedure may benefit from the incorporation of additional features such color and surface normals when processing complex scenes.

To aid our understanding of the results we present the barcode diagrams for each scene in Figs. 4.3a - 4.3f. In these diagrams, the x-axis represents the filtration values and the y-axis represents the 0-dimensional generators of homology. The length of the blue lines corresponds to the lifespan of the generators of homology. Shorter lines are points that die early in the filtration while longer lines are points that persist for a greater number of steps. A line with a red triangle equates to a point with infinite persistence; the corresponding 0-dimensional generating cycle persisted past the end of the filtration and did not get filled in as a boundary after all simplices of the underlying complex had been added.

Table 4.1 displays the following information for each scene: the number of points in the point cloud after filtering, the maximum distance value chosen for the VR complex construction, the size of the complex in the number of simplices, and the total CPU execution time.

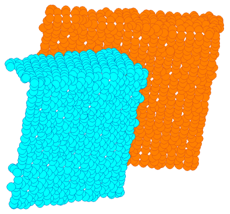
4.5 Region Segmentation Experiments

In this section, we evaluate the 3D region segmentation algorithm described in Section 4.2. The following eight objects were used in the experiments: plant, water heater, paper roll, robot, box, shoe, shuttle, coffee can. The experimental runs were carried out using MATLAB on a 64-bit GNU/Linux machine with a single CPU core.

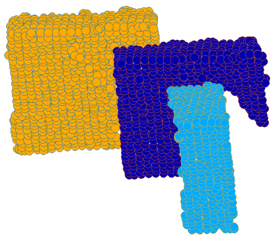
4.5.1 Experimental Setup

Prior to executing region segmentation we preprocess the data using the following steps. First, background subtraction is done to remove outlying points that are beyond range

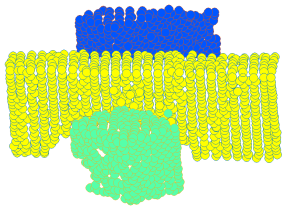
(a) Scene 1 filtered.

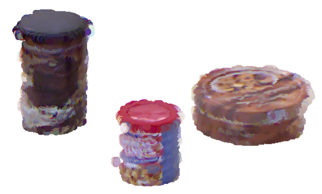

(b) Scene 1 segmented.

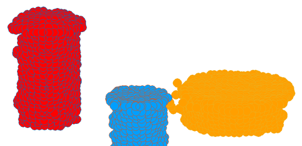
(c) Scene 2 filtered.

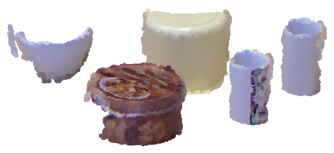

(d) Scene 2 segmented.

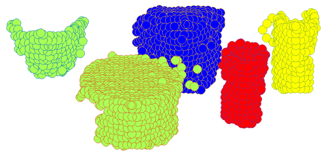
(e) Scene 3 filtered.

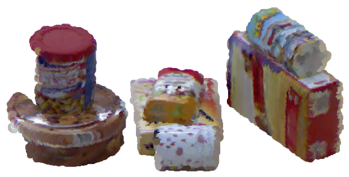

(f) Scene 3 segmented.


(g) Scene 4 filtered.


(h) Scene 4 segmented.


(i) Scene 5 filtered.


(j) Scene 5 segmented.


(k) Scene 6 filtered.

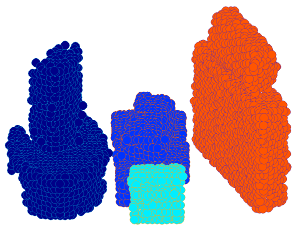
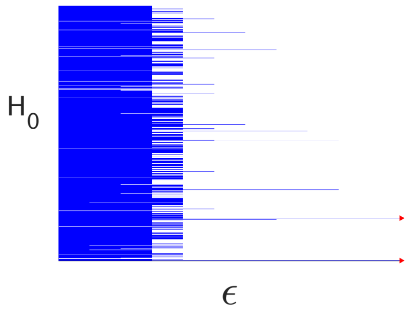
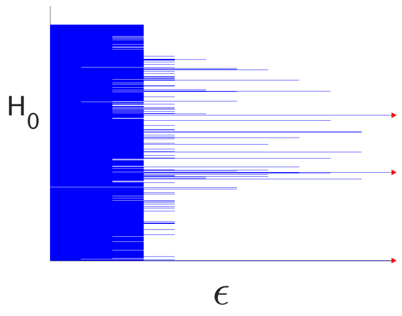

(l) Scene 6 segmented.

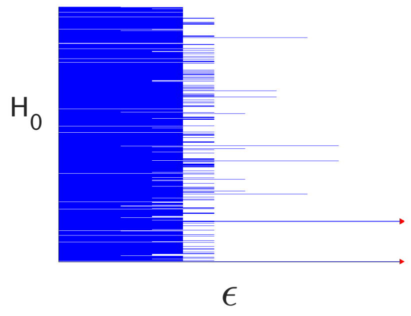
Figure 4.2: The results of 3D object segmentation using topological persistence.



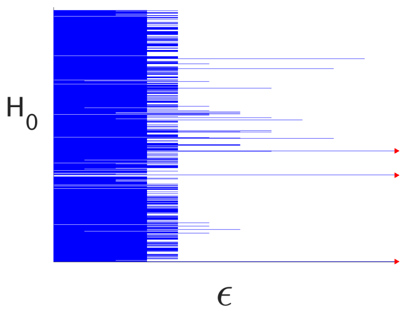
(a) Scene 1 barcode: 2 clusters.



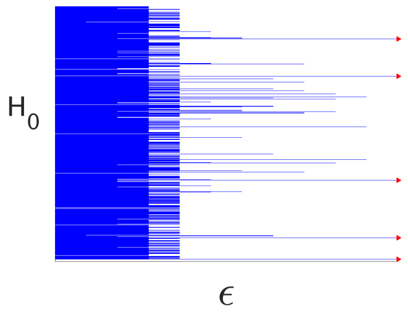
(b) Scene 2 barcode: 3 clusters.



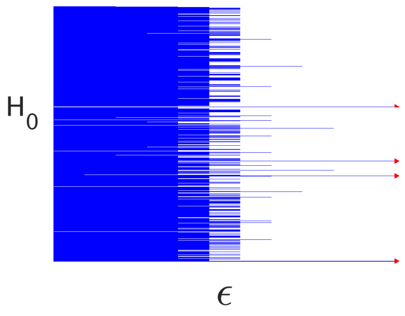
(c) Scene 3 barcode: 3 clusters.



(d) Scene 4 barcode: 3 clusters.



(e) Scene 5 barcode: 5 clusters.



(f) Scene 6 barcode: 4 clusters.

Figure 4.3: The barcode diagrams for the scenes in Figure 4.2. Lines with red triangles show points of infinite persistence.

Scene	Points	Distance	Size	Time
1	1289	0.025	4,813	2.211
2	1924	0.030	12,264	4.025
3	1410	0.020	3,744	2.344
4	1524	0.025	5,903	2.826
5	1581	0.025	5,994	3.179
6	2444	0.016	5,365	4.866

Table 4.1: The experimental results of object segmentation using topological persistence. The column categories are as follows: Points - Number of points in the point cloud, Distance - Maximum value for the complex construction, Size - Number of simplices in the complex, Time - CPU time in seconds.

of the object to be segmented. Next, the surface is removed upon which the object is resting. To do this, we find all points that correspond to a planar model and discard them.

After preprocessing the point cloud, we run a filtration using topological persistence in combination with the homogeneity criterion to grow regions and then extract the segmented regions. The results of segmenting the eight objects can be seen in Figure 4.2. For each object, we show the preprocessed representation of the point cloud before region segmentation proceeded by the randomly colored regions extracted after the segmentation has finished.

All filtrations are performed for ten steps up to the maximum distance value set for the complex. The threshold for the minimum number of points to be considered a region ranges from 8 to 128. For the homogeneity criterion, the color difference ΔE ranges from 3 to 10, and the angle difference θ between surface normals is set to $\pm 5°$.

4.5.2 Experimental Results

Under the existence of noise and the inherent limitations of the RGB-D sensor resolution, our region segmentation process is able to extract the major regions of each object. In the plant point cloud, the regions corresponding to the pot, soil, trunk, and leaves are captured. Note the existence of noise around each plant leaf in the original point cloud is

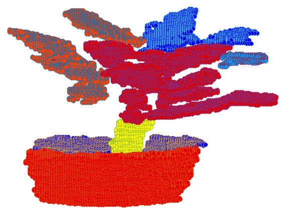
Object	Points	Distance	Features	Time
1	12,729	0.03	color+normals	1.508
2	10,395	0.035	color+normals	3.314
3	7,178	0.03	color+normals	1.884
4	25,790	0.02	color+normals	6.620
5	10,511	0.03	color	5.432
6	9,268	0.03	color	3.594
7	12,524	0.02	color	1.404
8	8,376	0.028	color	1.541

Table 4.2: The results of region segmentation using topological persistence. The column categories are as follows: Points - Number of points in the point cloud, Distance - Maximum value for the complex construction, Features - Type of local features used, Time - CPU time in seconds to compute topological persistence.

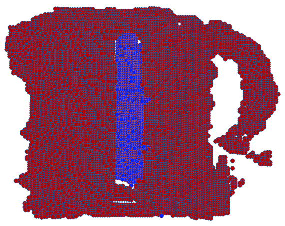
removed in the segmented version resulting in a clean delineation of the leaves. Regions of uniform color distribution and/or surface normals such as the water heater, paper roll, and shoe are easily segmented. The robot, box, shuttle, and coffee can render much more challenging scenarios. These objects have regions that are very small, contain subtle color changes, and are disjoint when in fact they should not be. For example, in the segmented coffee can 112 regions are found. Some of these regions make up the product label, many are noise. Therefore, a trade-off in region segmentation must be made between granularity and noise.

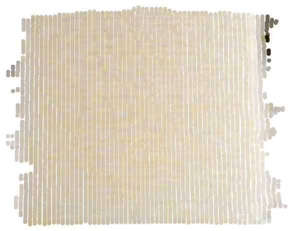
Similar to the object segmentation experiments in Section 4.4, we present the barcode diagrams for each point cloud in Figures 4.3a - 4.2h. In these diagrams the filtration values are represented on the x-axis, and the y-axis shows the 0-dimensional generators of homology. Lines with blue triangles indicate points with infinite persistence and correspond to the set representatives of the segmented regions. Table 4.2 displays the following information for each object (plant, water heater, paper roll, robot, box, shoe, shuttle, coffee can): the number of points in the point cloud after filtering, the maximum distance value chosen for the VR complex construction, the type of local features used, and the execution time for computing topological persistence.

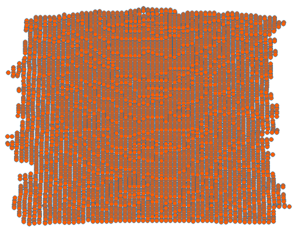
(a) Plant filtered.

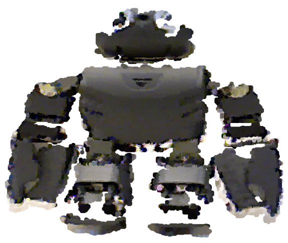

(b) Plant region segmented.

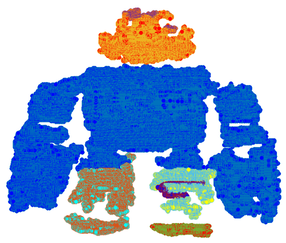
(c) Water heater filtered.

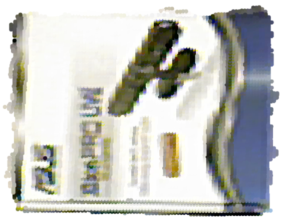

(d) Water heater region segmented.

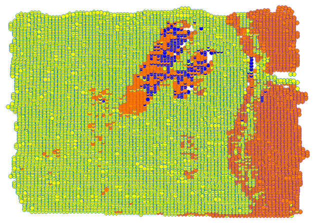

(e) Paper roll filtered.

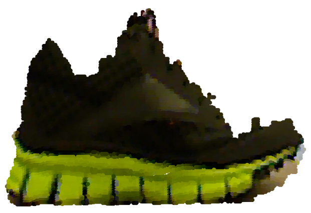

(f) Paper roll region segmented.

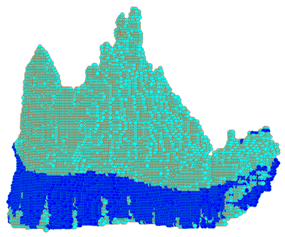

(g) Robot filtered.


(h) Robot region segmented.

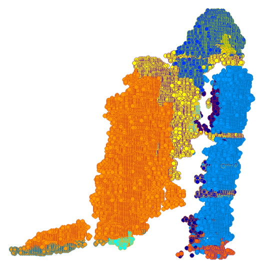

(i) Box filtered.

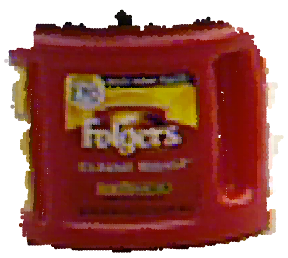

(j) Box region segmented.


(k) Shoe filtered.


(l) Shoe region segmented.

(m) Shuttle filtered.


(n) Shuttle region segmented.



(o) Coffee can filtered.

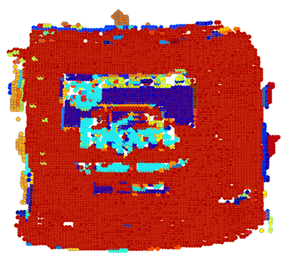
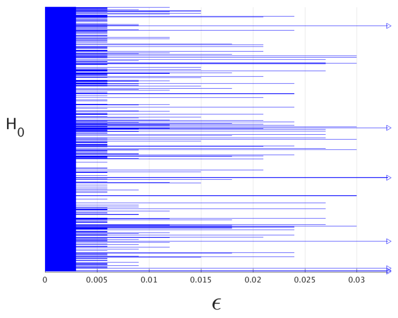
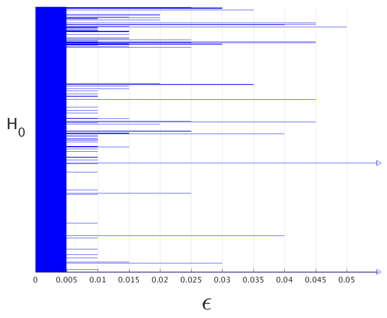

(p) Coffee can region segmented.

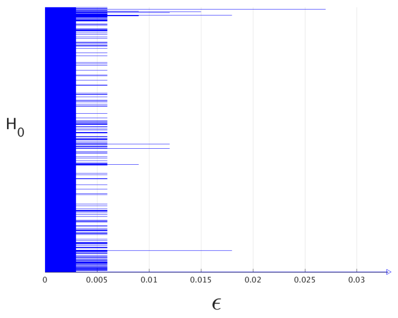
Figure 4.2: The results of 3D region segmentation using topological persistence.



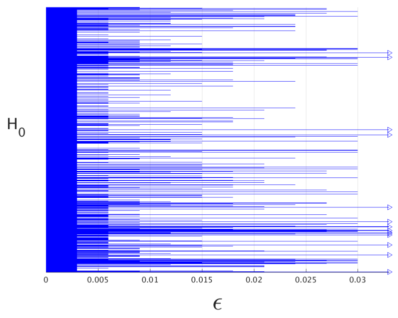
(a) Plant barcode: 7 regions.



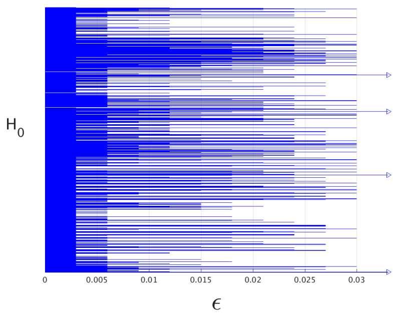
(b) Water heater barcode: 2 regions.



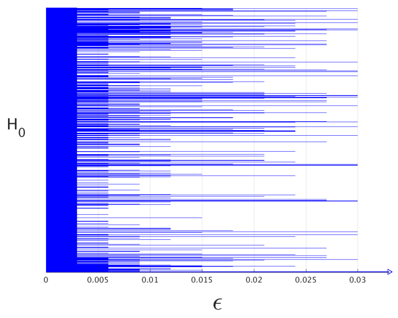
(c) Paper roll barcode: 1 region.



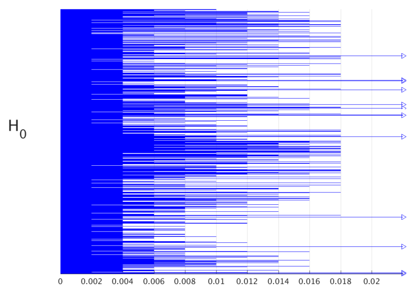
(d) Robot barcode: 14 regions.



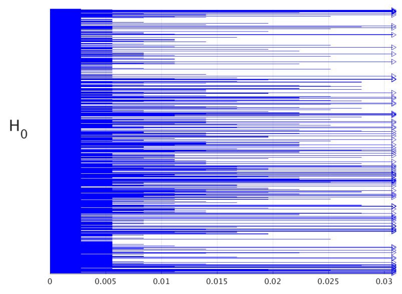
(e) Box barcode: 4 regions.



(f) Shoe barcode: 2 regions.



(g) Shuttle barcode: 12 regions.



(h) Coffee can barcode: 112 regions.

Figure 4.2: The barcode diagrams for the objects in Figure 4.2. Lines with blue triangles indicate points of infinite persistence.

4.6 Conclusion

The segmentation of noisy point cloud data is problematic due to the occurrence of sensor artifacts, outliers, and variation in the density of the data. Towards making progress in addressing these problems, this chapter presented and evaluated algorithms and data structures for segmenting 3D point clouds by applying topological persistence. The purpose of this work is to not only establish a new way of performing segmentation, but also to introduce the notion of persistent homology into the area of 3D point cloud processing.

Our experimental results show that point cloud object segmentation using topological persistence is an effective and robust method for finding and extracting scene objects in the presence of noise. We've seen that regions in point cloud data often match fundamental parts of an object or entirely complete objects. Moreover, the extraction of regions is an imperative action in high-level computer and robotic vision tasks such as scene understanding. Experimentally, our results show a region growing approach using persistent homology that combines global and local knowledge is a capable method for identifying regions in noisy 3D point cloud data.

Chapter 5

Point Cloud Hole Boundary Detection

The nature of point cloud datasets requires fundamentally different processing paradigms for extracting information. Compared to mesh-based datasets, the lack of explicit connectivity information simplifies the definition and implementation of many tasks encountered in point cloud processing. On the contrary, the detection of holes in the surface, trivial in the case of meshes, becomes an ill-defined problem, Figure 5.1.

The knowledge of holes in the data is vital for many applications dealing with 3D point clouds. For example, it can be used to reconstruct surfaces with boundaries or to direct a further step for the next best view. Furthermore, identification of points on the boundary of a hole is required before any attempt to algorithmically fill in the hole is made.

In this chapter, we establish fast algorithms for computing 1-cycles and locating non-bounding minimal 1-cycles. These non-bounding minimal 1-cycles tell us precisely the boundary points of the holes in the dataset. The main contributions of our work are:

- A nearly linear time algorithm for incrementally calculating topologically persistent 1-cycles.
- An algorithm for computing non-bounding minimal 1-cycles based on the output of the topologically persistent 1-cycles algorithm.

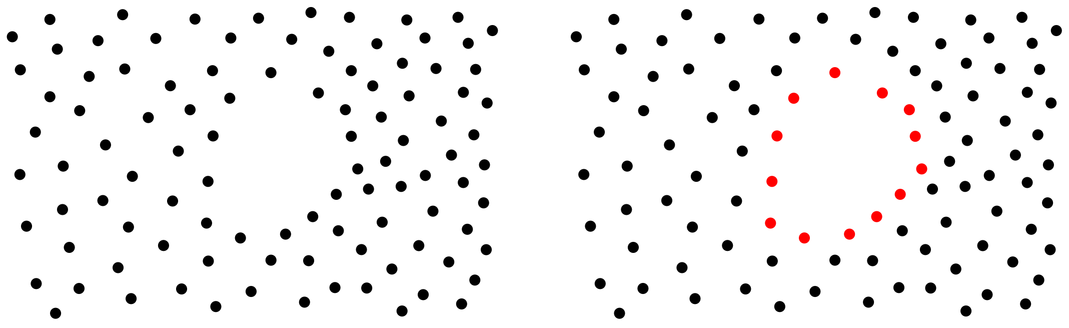


Figure 5.1: A point cloud (a) and the points on the boundary of a hole highlighted in red (b). One of challenges in detecting the boundary points of a hole is that the density of the point cloud can vary. For example, in this point cloud the left side is more sparse than the right side.

- A topological approach for 3D point cloud hole boundary point detection utilizing these algorithms.

The content of this chapter is organized as follows. First, we take note of previous work in the area of computing 1-cycles and the state of the art in 3D point cloud hole boundary point detection. Next, we provide the algorithmic background for an incremental way to compute persistent homology. We then present our approach to quickly compute persistent 1-cycles incrementally. Finally, we devise a new algorithm for hole boundary point detection based on incrementally computing 1-cycles.

5.1 Related Work

The fast incremental algorithm we develop for computing the topological persistence of 1-cycles is based on the algorithm proposed by Edelsbrunner et al. [53]. In their work, it was noted that the addition of 1-cycles is the slowest part of the algorithm. They asked for a different and more efficient algorithm, if one exists. Our work addresses this problem and provides a comparison with the state of the art [54]. Additionally, we both compute and store generating 1-cycles which can have useful applications [55].

Within the computer graphics and computer vision communities, there has been a substantial amount of research done in the area of hole filling for the purpose of 3D surface reconstruction [56]. Hole filling methods can be classified into two types: point cloud-based methods and mesh-based methods. Each hole filling type can be further divided into volume-based methods and surface-based methods. Popular hole filling methods utilize moving least squares [57], radial basis functions [58], a spatial Poisson formulation [59], and template-based data-driven approaches [60].

Detecting holes in a triangle mesh is a straight forward task. A triangle mesh is composed of a set of vertices, edges, and triangles. Two triangles that share an edge are called adjacent triangles of that edge. A boundary edge is defined as an edge that is only adjacent to a single triangle. Correspondingly, a boundary edge loop is the representation of a closed hole which can be extracted from the input mesh by tracing its adjacent boundary edges.

The detection of holes in a point cloud is much more difficult due to the absence of connectivity information. Bendels et al. [61] propose a method to detect the boundary of missing regions in a point cloud. Their work computes the probability of a given point being a boundary point of a hole according to three criteria: angle criterion, half-disc criterion, and shape criterion. These criteria are then combined into a weighted sum with multiple tuning parameters. This state of the art method is currently used in recent data-driven deep learning approaches to hole filling [62]. In contrast to the work of Bendels et al., we propose a hole detection method that does not require heuristics and can be executed in parallel.

5.2 Preliminaries

In the following subsections we present the background behind computing persistent homology incrementally, an alternative method to the standard algorithm (Section 2.8) when simplicial complexes exist in 3-dimensions or less.

5.2.1 Betti Numbers Algorithm

A procedure to compute the Betti numbers of all complexes in a filtration is shown in Algorithm 5.8. It is based on the ordering of simplices in a filter as follows. Let the sequence σ^i be a filter and let the sequence $K^i = \{\sigma^j \mid 0 \leq j \leq i\}$, $0 \leq i < m$, be the corresponding filtration. At each step, the algorithm must decide if a $(k+1)$-simplex σ^i belongs to a $(k+1)$-cycle in K^i. When $k+1=0$, this is trivial since every vertex belongs to a 0-cycle. For edges, the algorithm maintains the connected components of the complex each represented by its vertex set. An edge belongs to a 1-cycle if and only if its two endpoints belong to the same connected component. Triangles and tetrahedra are treated similarly by the algorithm using symmetry as outlined by [12].

After establishing the presence of cycles the algorithm labels each simplex. A $(k+1)$-simplex σ^i is *positive* if it belongs to a $(k+1)$-cycle, otherwise it's labeled *negative*. The correctness of the algorithm implies the following. Let $\beta_k = \beta_k^l$ be the kth Betti number of K^l. In addition, let $\text{pos}_k = \text{pos}_k^l$ and $\text{neg}_k = \text{neg}_k^l$ be the number of positive and negative k-simplices in K^l. Then,

$$\beta_k = \text{pos}_k - \text{neg}_{k+1}, \tag{5.1}$$

for $0 \leq k \leq 2$. The Betti number, β_k, is the number of k-simplices that create k-cycles, minus the number of $(k+1)$-simplices that destroy k-cycles by creating k-boundaries. Note that Equation (5.1) is just a different way to express Equation (2.2). Furthermore, Betti numbers are non-negative hence $\text{pos}_k \geq \text{neg}_{k+1}$ for all l.

Algorithm 5.8 Betti Numbers

```
Input: Filtration $K$
Output: Betti numbers $\beta_0, \beta_1, \beta_2$
1: for $i \in [0, m-1]$ do
2:     $k \leftarrow \dim \sigma^i - 1$
3:     if $\sigma^i$ belongs to a $(k+1)$-cycle in $K^i$ then
4:         $\beta_{k+1} \leftarrow \beta_{k+1} + 1$
5:     else
6:         $\beta_k \leftarrow \beta_k - 1$
7:     end if
8: end for
9: return $(\beta_0, \beta_1, \beta_2)$
```

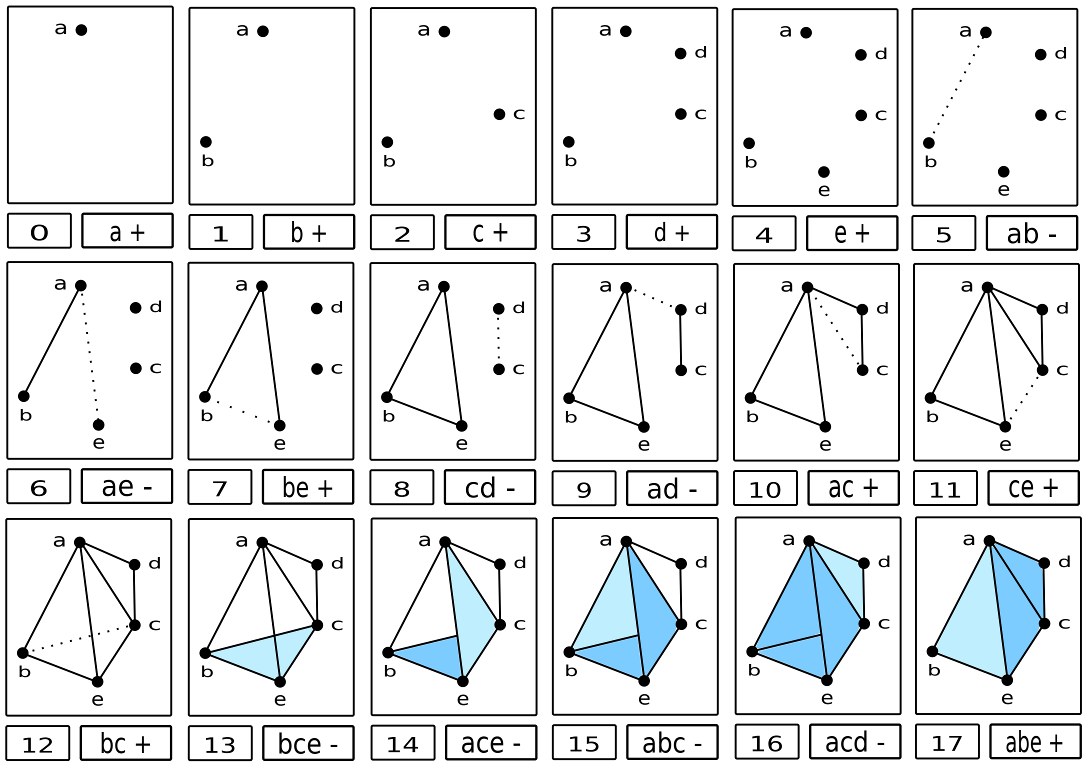


Figure 5.2: An example of a filter consisting of a sequence of simplices that constitute a subcomplex.

5.2.2 Pairing Simplices Algorithm

The pairing simplices algorithm matches the creation of a non-bounding cycle with its conversion to a boundary. It measures the lifetime of non-bounding cycles by determining when a cycle's homology class is created and when its class merges with a boundary group. Positive simplices create classes while negative simplices merge classes with the boundary groups. These events are detected by maintaining a basis for H_k implicitly through simplex representatives.

The algorithm starts with an initially empty basis for H_k. For each positive k-simplex σ^i, a non-bounding k-cycle c^i that contains σ^i but no other positive k-simplices is first identified. Upon finding c^i, the homology class of c^i is added as a new element to the basis of H_k. In this way, the class $c^i + B_k$ is represented by c^i. Likewise, c^i is represented by σ^i. For each negative $(k+1)$-simplex σ^j, its corresponding positive k-simplex σ^i is found. Next, the homology class of σ^i is removed from the basis. A general homology class of K^i can be expressed as a sum of basis classes,

$$\begin{aligned} d + B_k &= \sum (c^g + B_k) \\ &= B_k + \sum c^g. \end{aligned}$$

The chains d and $\sum c^g$ are homologous, i.e. they belong to the same homology class. For each c^g, there is a positive k-simplex σ^g, $g < j$, not yet paired by the algorithm. The collection of positive k-simplices, $\Gamma = \Gamma(d)$, is determined uniquely by d. The youngest simplex in Γ is the simplex with the largest index and is denoted as $y(d)$.

As seen in Algorithm 5.9, the procedure determines if σ^j turns the k-cycle created by σ^i into a k-boundary. If so, (σ^i, σ^j) are appended to the list L_k. The difference between the indices minus one $(j - i - 1)$ is the persistence of that k-cycle.

5.2.3 Computing Persistent Homology Incrementally

The persistence of a k-cycle created by σ^i and destroyed by σ^j is one short of the difference between the indices, $j - i - 1$. We now explain how to determine the index i of the youngest positive k-simplex in $\Gamma(d)$, where $d = \partial_{k+1}(\sigma^j)$ (line 6, Algorithm 5.9). First, we describe the data structure and cycle search procedure for finding σ^j. Then, we show a comprehensive algorithm that combines the elements of preceding algorithms

Algorithm 5.9 Pair Simplices

```
    Input: Filtration $K$
    Output: Simplicial pair lists $L_0, L_1, L_2$
 1: $L_0 \leftarrow L_1 \leftarrow L_2 \leftarrow \emptyset$
 2: for $j \in [0, m-1]$ do
 3:     $k \leftarrow \dim \sigma^j - 1$
 4:     if $\sigma^j$ is negative then
 5:         $d \leftarrow \partial_{k+1}(\sigma^j)$
 6:         $i \leftarrow \mathrm{y}(d)$
 7:         $L_k \leftarrow L_k \cup \{(\sigma^i, \sigma^j)\}$
 8:     end if
 9: end for
10: return $(L_0, L_1, L_2)$
```

for calculating both Betti numbers and persistence along with an example of computing the persistence of a 1-cycle.

Data Structure

The data structure consists of a hash table, $T[0, \ldots, m-1]$, which is initially empty. A slot $T[i]$ stores set of positive simplices Λ^i defining a cycle created by σ^i and terminated by σ^j. This cycle contains the youngest simplex, σ^i, in $\Gamma(d)$. The simplices in the set Λ^i are not necessarily the same as the ones in $\Gamma(d)$. However, it's guaranteed that d is homologous to the sum of cycles represented by the simplices in the set and that the set contains the youngest simplex σ^i [53].

An illustration of the data structure can be seen in Figure 5.3. Each simplex in the filter has a slot in the hash table, but data is only stored in the slots of the positive simplices. This information consists of the index j of the matching negative simplex and a set of positive simplices defining a cycle. Cycles that exist beyond the end of the filter are indicated by ∞.

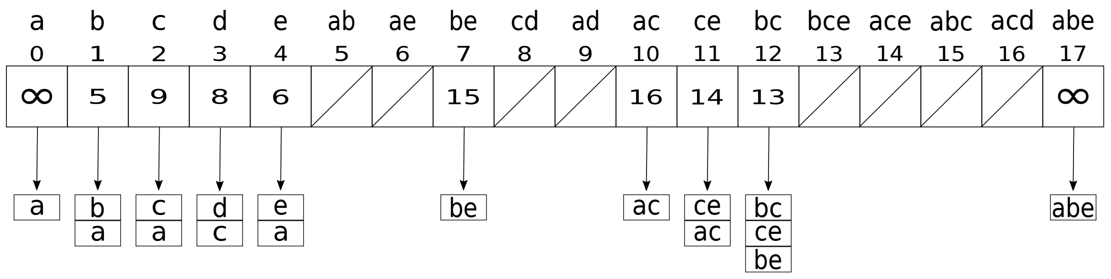


Figure 5.3: The state of the hash table after running Algorithm 5.9 on the filter in Figure 5.2.

Cycle Search

Let σ^j be a negative k-simplex and suppose that the algorithm arrives at index j in the hash table T. Recall that the homology class of $d = \partial\sigma^j$ in H_k^{j-1} is represented by $\Gamma(d)$, the set of positive k-simplices. The goal is to find the correct slot in T that corresponds to the youngest k-simplex in $\Gamma(d)$.

To begin, the set Λ is created to contain the positive k-simplices in d and $i = \max(\Lambda)$ is initialized to be the index of the youngest member of Λ. If $T[i]$ is unoccupied, then $i = y(d)$ and the search concludes. The set Λ and index j are stored in $T[i]$.

On the contrary, if $T[i]$ is occupied then it contains a set Λ^i that represents a permanently stored k-cycle. This k-cycle is already a k-boundary. Therefore, Λ and Λ^i are added to produce a new Λ that represents a k-cycle homologous to the old one (and also homologous to d). The cycle search procedure for σ^j shows how to implement lines 5 and 6 of Algorithm 5.9 and is shown in Algorithm 5.10.

Example

We combine Algorithms 5.8 - 5.10 into an all-inclusive method for computing both Betti numbers and persistence incrementally as shown in Algorithm 5.11. In the event of a collision in the hash table, the sets Λ and Λ^i are added by taking the symmetric difference between the two (Equation (2.1)). Consider the first negative triangle bce in the filter of Figure 5.2. We have $\Lambda = \{bc, ce, be\}$, $i = 12$, and $j = 13$. The $12th$ slot of T is unoccupied, therefore we store Λ and j in $T[12]$. Processing the proceeding negative triangle, ace, we get $\Lambda = \{ce, ac\}$, $i = 11$, and $j = 14$. Since $T[11]$ is unoccupied, Λ and

Algorithm 5.10 Youngest

```
    Input: Chain $d \leftarrow \partial_{k+1}(\sigma^j)$
    Output: Index $i$
 1: $\Lambda \leftarrow \{\sigma \in \partial_{k+1}(\sigma^j) \mid \sigma$ positive$\}$
 2: while 1 do
 3:     $i \leftarrow \max(\Lambda)$
 4:     if $T[i]$ is unoccupied then
 5:         $T[i] \leftarrow \Lambda$ and $j$
 6:         break
 7:     end if
 8:     $\Lambda \leftarrow \Lambda + \Lambda^i$
 9: end while
10: return $(i)$
```

j are stored in the $11th$ slot of T.

The next negative triangle, abc, creates a collision. Initially, we have $\Lambda = \{bc, ac\}$, $\Lambda^i = \{bc, ce, be\}$, $i = 12$, and $j = 15$. The sum of the two 1-cycles is $\Lambda + \Lambda^i = \{ce, ac, be\}$, which is the new set Λ and $i = 11$. We have another collision at $T[11]$, thus we again take the sum of the two 1-cycles: $\Lambda + \Lambda^i = \{be\}$. This time we find that $T[i]$ is unoccupied for $i = 7$, therefore $T[7]$ stores $\Lambda = \{be\}$ and $j = 15$. The remaining two triangles are handled as follows. For triangle acd we store $\Lambda = \{ac\}$ and $j = 16$ in $T[10]$. Triangle abe gives $\Lambda + \Lambda^i = \{\emptyset\}$. Finally, the number of positive edges minus the number of negative triangles yields the Betti number: $\beta_1 = 4 - 4 = 0$.

5.3 Fast Incremental Persistence for 1-Cycles

The running time of cycle search can be improved to almost constant time for $k = 0$ and $k = 2$ using a disjoint-set data structure and its supporting operations [39]. However, searching for 1-cycles has required the slower algorithm described above as there can be multiple unpaired edges at any time [53]. In this section, we introduce a fast algorithm for incrementally computing the persistence of 1-cycles based on the pairing of simplices. We first explain the data structure followed by the cycle storage

Algorithm 5.11 Incremental Persistence

```
Input: Chain $d \leftarrow \partial_{k+1}(\sigma^j)$
Output: Hash table $T$
 1: for all $\sigma^j$ do
 2:     $\Lambda \leftarrow \{\sigma \in \partial_{k+1}(\sigma^j) \mid \sigma$ positive$\}$
 3:     $i \leftarrow \max(\Lambda)$
 4:     if $T[i]$ is unoccupied then
 5:         $T[i]$.cycle $\leftarrow \Lambda$
 6:         $T[i]$.death $\leftarrow j$
 7:         $\sigma^j \leftarrow$ negative
 8:     else
 9:         while 1 do
10:             $\Lambda \leftarrow \Lambda + T[i]$.cycle
11:             if $\Lambda$ is empty then break end if
12:             $i \leftarrow \max(\Lambda)$
13:             if $T[i]$ is unoccupied then
14:                 $T[i]$.cycle $\leftarrow \Lambda$
15:                 $T[i]$.death $\leftarrow j$
16:                 $\sigma^j \leftarrow$ negative
17:                 break
18:             end if
19:         end while
20:     end if
21: end for
```

and cycle update procedures. Then, we provide an example of the algorithm in action, show its correctness, and analyze its running time.

Algorithm 5.12 Fast Incremental Persistence

```
    **Input:** Chain $d \leftarrow \partial_{k+1}(\sigma^j)$
    **Output:** Hash table $T$
1: for all $\sigma^j$ do
2:     $\Lambda \leftarrow \{\sigma \in \partial_{k+1}(\sigma^j) \mid \sigma$ positive$\}$
3:     storeCycles($\Lambda$)
4: end for
```

5.3.1 Data Structure

The data structure we use is made up of a hash table T, initially empty, where a slot $T[i]$ stores set of positive k-simplices Λ^i defining a cycle created by σ^i and annihilated by σ^j. The youngest simplex, σ^i in $\Gamma(d)$, is contained in this cycle. Although the simplices in Λ^i are not necessarily the same as the ones in $\Gamma(d)$, d is homologous to the sum of cycles represented by the simplices in the set and the set contains σ^i, the youngest simplex.

Each k-simplex in the filter has a slot in the hash table with data stored only for positive simplices. This information consists of the index j of the matching negative $(k+1)$-simplex and a balanced binary search tree (BST) of positive k-simplices defining a cycle. Additionally, we keep a list of positive k-simplices to be updated after a cycle is stored as described in the next subsection.

5.3.2 Cycle Storage Algorithm

Suppose that the algorithm reaches index j in the hash table T and let σ^j be a negative $k+1$-simplex. Recall that $\Gamma(d)$, the set of positive k-simplices, is represented by the homology class $d = \partial\sigma^j$ in H_k^{j-1}. Our aim is to find the proper slot in T that corresponds to the youngest k-simplex in $\Gamma(d)$. We start with a set Λ that contains the positive k-simplices in d, and set $i = \max(\Lambda)$ to be the index of the youngest member of Λ. There are two cases to consider.

Algorithm 5.13 Store Cycles

```
$\Lambda \leftarrow \text{sort}(\Lambda)$, $i \leftarrow \max(\Lambda)$, $\Lambda^c \leftarrow \{\emptyset\}$
**if** $T[i]$ is unoccupied **then**
    **for** $n \in [0, |\Lambda| - 2]$ **do**
        **if** $T[\Lambda(n)]$ is unoccupied **then**
            $\Lambda^c \leftarrow \Lambda^c + \Lambda(n)$
        **else if** $T[\Lambda(n)]$.cycle is nonempty **then**
            $\Lambda^c \leftarrow \Lambda^c + T[\Lambda(n)]$.cycle
        **else**
            continue
        **end if**
        $T[\Lambda(n)]$.update $\leftarrow i$
    **end for**
    $T[i]$.cycle $\leftarrow \Lambda^c$
    $T[i]$.death $\leftarrow j$
    $\sigma^j \leftarrow$ negative
    updateCycles($i, 0, T[i]$.update)
**else**
    **for** $n \in [0, |\Lambda| - 2]$ **do**
        **if** $T[\Lambda(n)]$ is unoccupied **then**
            $\Lambda^c \leftarrow \Lambda^c + \Lambda(n)$
        **else if** $T[\Lambda(n)]$.cycle is nonempty **then**
            $\Lambda^c \leftarrow \Lambda^c + T[\Lambda(n)]$.cycle
        **end if**
    **end for**
    $\Lambda \leftarrow \Lambda^c + T[i]$.cycle
    **if** $\Lambda$ is nonempty **then**
        $i \leftarrow \max(\Lambda)$
        **if** $T[i]$ is unoccupied **then**
            **for** $n \in [0, |\Lambda| - 2]$ **do**
                $T[\Lambda(n)]$.update $\leftarrow i$
            **end for**
            $T[i]$.cycle $\leftarrow \Lambda - i$
            $T[i]$.death $\leftarrow j$
            $\sigma^j \leftarrow$ negative
            updateCycles($i, 0, T[i]$.update)
        **end if**
    **end if**
```

Case 1

In the first case, the slot $T[i]$ in the hash table is unoccupied and we proceed to form a new cycle set Λ^c by the following procedure. For each element in Λ, excluding the youngest, we replace it with its cycle if it has one. To do this, we add Λ^c with the cycle stored at that element by taking the symmetric difference of the two sets. After constructing Λ^c we store it in the ith position of T.

Case 2

In the second case, the slot $T[i]$ in the hash table is already occupied. We need to find the correct slot to store the cycle. Since $T[i]$ is nonempty it contains a set Λ^i representing a stored k-cycle which is also a k-boundary. Similar to the previous case, we create a new cycle set Λ^c by replacing each element in Λ, exclusive of the youngest, with its cycle if it has one. Then we add Λ^c and Λ^i to get a new set Λ. This Λ is either empty or it represents a k-cycle homologous to the old one and therefore also homologous to d.

Algorithm 5.14 Update Cycles

```
1: if update is nonempty then
2:     for $n \in [0, |\text{update} - 1]$ do
3:         updateCycles($i$, update($n$), $T$[update($n$)].update)
4:     end for
5: end if
6: $T$[update($n$)].cycle $\leftarrow$ $T$[update($n$)].cycle $+$ $T[i]$.cycle $+ i$
```

5.3.3 Cycle Update Algorithm

For each entry in the hash table T that belongs to a cycle, a list of positive k-simplices to be updated is maintained. When storing a cycle at $T[i]$ we check if the list of elements to be updated is empty or not. If the list is nonempty, then for each simplex we remove the ith element from its cycle and add the cycle elements stored in $T[i]$. The cycle update procedure is recursively performed for each simplex in the list.

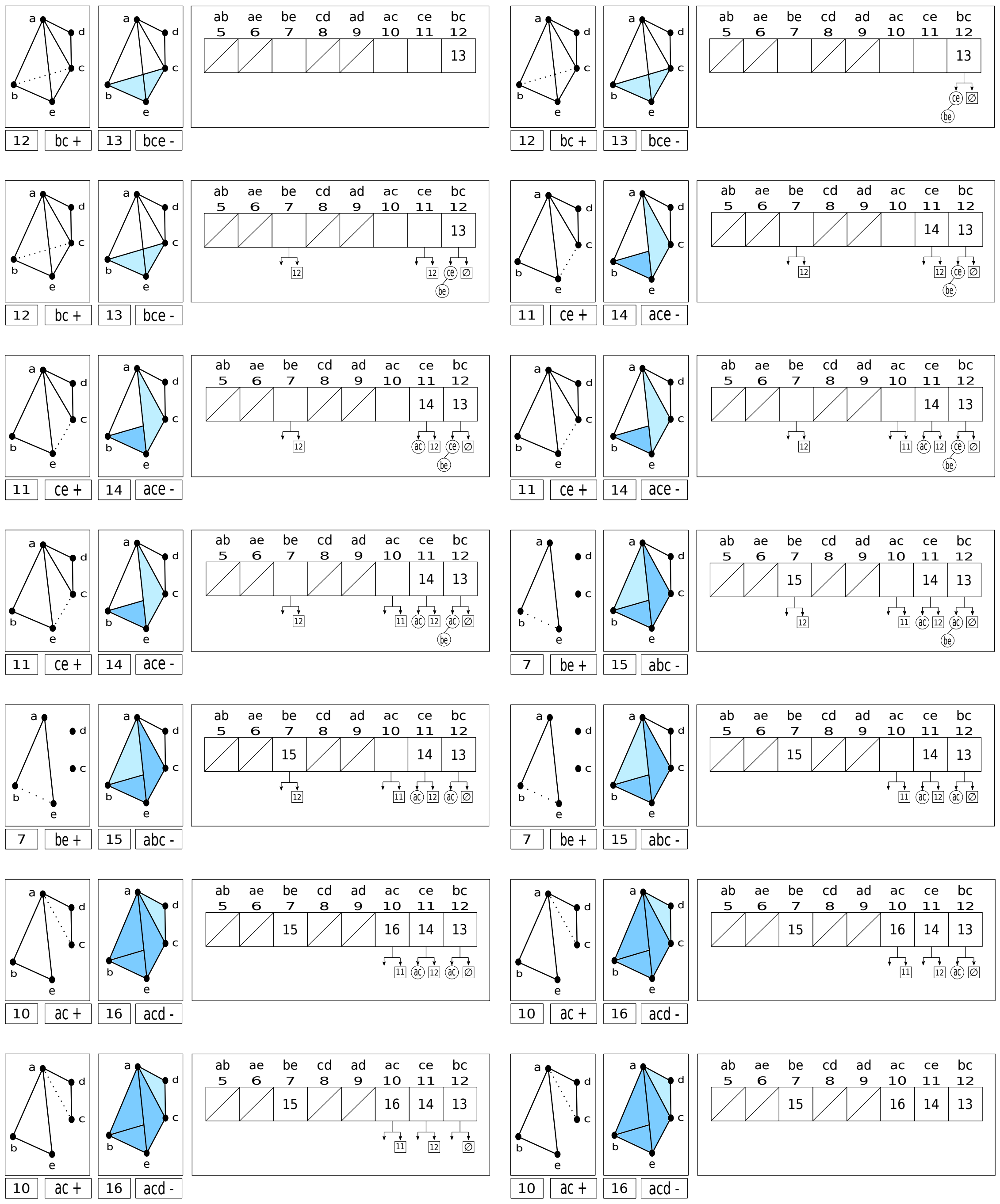


Figure 5.4: The hash table states (left to right, top to bottom) while processing 1-cycles for the filter shown in Figure 5.2.

5.3.4 Example

In this subsection we show how to compute the 1-cycles for the filter in Figure 5.2 using the fast incremental persistence method, Algorithm 5.12. The hash table states are depicted in Figure 5.4. The details of the cycle storage and update procedures are given in Algorithms 5.13 and 5.14.

For the first negative triangle, bce, we have $\Lambda = \{be, ce, bc\}$, $i = 12$, and $j = 13$. Since $T[12]$ is empty we create and store $\Lambda^c = \{be, ce\}$ along with j in the slot. In addition, for edges be and ce we add i to the list of simplices to be updated. Proceeding with the next negative triangle, ace, we have $\Lambda = \{ac, ce\}$, $i = 11$, and $j = 14$. Slot $T[11]$ is empty, therefore $\Lambda^c = \{ac\}$ and j are stored in the $11th$ position of T. We also notice that $T[11]$ has a nonempty update list indicating that a cycle update needs to be performed. To perform the update, we remove the simplex corresponding to $i = 11$ (ce) in $T[12]$ and replace it with the cycle stored at $T[11]$ (ac).

The remaining two negative triangles are processed as follows. Triangle abc brings about a collision with $\Lambda = \{ac, bc\}$, $\Lambda^i = \{be, ac\}$, $i = 12$, and $j = 15$. We construct $\Lambda^c = \{ac\}$ and sum the two 1-cycles, $\Lambda^c + \Lambda^i = \{be\}$, which is the new set Λ with $i = 7$. We store $j = 15$ in slot 7 and execute a cycle update. The cycle update mechanism removes the $7th$ element (be) from $T[12]$. The last negative triangle, acd, gives $\Lambda = \{ac\}$, $i = 10$, and $j = 16$ with $T[10]$ available. The cycle update process for slot 10 recursively traverses slots 11 and 12. Upon finding the simplex list in slot 12 empty, element 10 (ac) is removed from the stored cycle. Backtracking to slot 11, element 10 is removed thus completing the cycle update procedure.

There are two key observations this example. First, when processing triangle abc only one sum is needed versus two sums in the original algorithm. Second, cycle updates for single element 1-cycles reduce the size of the stored cycles.

5.3.5 Correctness

Consider the case were $T[i]$ is occupied. The construction of the set Λ^c replaces each element in Λ, excluding the youngest, with its stored cycle if it has one. Thus, Λ^c consists of positive k-simplices all older than σ^i. Furthermore, the cycle set Λ^i stored at $T[i]$ also contains other positive k-simplices all older than σ^i. Adding Λ^c and Λ^i

produces a new set Λ where all of the simplices in Λ are older than σ^i. Therefore, the new index i is less than the previous one which implies that the search for an open slot in T moves strictly from right to left. The search halts at the unoccupied slot $T[g]$ of the hash table where $g = y(d)$ and $d = \partial_{k+1}(\sigma^j)$ is the boundary of the negative $(k+1)$-simplex that initiated the search, all other possibilities lead to contradictions.

To show that $T[g]$ is the correct slot consider the following. Let e be the cycle created by Λ^g. We know that e and d are homologous in K^{j-1} because e is obtained from d by adding bounding cycles. When the youngest positive simplex corresponds to an unoccupied slot in T we have a collision-free cycle. Cycle storage computes a collision-free cycle, e.g. e is collision-free since its youngest positive simplex is σ^g and $T[g]$ is unoccupied before e arrives. After a cycle is placed in slot $T[g]$, existing cycles that store g are updated. The simplices that store a cycle containing g are younger than g, therefore the update procedure moves strictly from left to right in T. Each update replaces g with the cycle e modulo g. The update ends at the youngest simplex with a cycle that includes g.

Cycle Storage-Update Lemma. *Let $\Lambda = \Lambda^c + \Lambda^i$ where Λ is nonempty, $i = max(\Lambda)$, and $T[i]$ is occupied. $T[i]$ represents a single element 1-cycle.*

Proof. Suppose that the cycle stored at $T[i]$ consists of more than one element. It takes at most two additions to compute the new set Λ: one addition to compute Λ^c and one addition to sum Λ^c and Λ^i. Therefore, it must be the case that either Λ^c or Λ^i contains a cycle that has not been updated thus requiring more than two additions and contradicting the correctness of the algorithm. $\square$

5.3.6 Analysis

In this subsection we analyze the running time of computing 1-cycles. Let $\Lambda = \{\sigma^o, \sigma^m, \sigma^y\}$ where σ^o, σ^m, and σ^y respectively denote the oldest, middle, and youngest elements of Λ. Furthermore, let t_o, t_m, and t_y represent the cycles stored as balanced BSTs in the hash table slots $T[\sigma^o]$, $T[\sigma^m]$, and $T[\sigma^y]$ respectively. We consider each case of cycle storage separately.

In the first case of cycle storage the slot $T[\sigma^y]$ is unoccupied. Assume that the cycle trees t_o and t_m stored in $T[\sigma^o]$ and $T[\sigma^m]$ have i and j elements respectively. We construct Λ^c as follows. Among t_o and t_m we choose the largest tree, say t_o, and create a copy called t_c. We then perform a tree traversal on t_m. For each element in t_m we check to see if it's in t_c. If so, we delete that element in t_c, otherwise we insert it. Therefore, it takes $j \cdot \log(i)$ operations to construct t_c which represents Λ^c and is stored in $T[\sigma^y]$. Since i and j can be at most the number of positive edges minus 1, the worst case runtime is $\mathcal{O}\big((e_p - 1) \cdot \log(e_p - 1)\big)$ where e_p is the number of positive edges.

In the second case of cycle storage the slot $T[\sigma^y]$ is occupied. Assume that the cycle trees t_o, t_m, and t_y stored in $T[\sigma^o]$, $T[\sigma^m]$, and $T[\sigma^y]$ have i, j, and k elements respectively. Λ^c is constructed by starting with a copy of the largest tree, say t_o. Next, tree traversals are done on both t_m and t_y. If an element from either t_m or t_y appears in t_c we delete it, if not we insert it into t_c. In total we have $(j + k) \cdot \log(i)$ operations to build t_c and a worst case runtime of $\mathcal{O}\big((2e_p - 2) \cdot \log(e_p - 1)\big)$.

Each time an unoccupied slot in T is found we check to see if a cycle update needs to be performed. The update list for a stored cycle can be no longer than $e_p - 1$ elements. For each cycle in the list, we need to perform at most 1 delete operation and $(e_p - 1)$ insert operations. Thus, the runtime for cycle update is $\mathcal{O}\big((e_p - 1) \cdot (\log(e_p - 1) + (e_p - 1) \cdot \log(e_p - 1))\big)$. A parallel implementation of cycle update could reduce the runtime to $\mathcal{O}\big(\log(e_p - 1) + (e_p - 1) \cdot \log(e_p - 1)\big)$.

As the slots in T fill up the runtime is dominated by the second case of cycle storage due to the additional tree traversal. Therefore, the fast incremental persistence algorithm runs in time at most $\mathcal{O}\big(c \cdot n\big)$ where n is the number of triangles and the constant $c = (2e_p - 2) \cdot \log(e_p - 1)$ is the amount of work done per triangle. In practice, the number of elements stored per cycle is far less the number of positive edges. Moreover, single element 1-cycles that trigger a cycle update reduce the size of stored cycles.

5.4 Minimal 1-Cycles

The fast incremental persistence algorithm marks simplices as either positive or negative. Specifically, a simplex of dimension k in a filter is positive if it creates a k-cycle or negative if it destroys a $(k - 1)$-cycle by turning it into a boundary. In a filter, each negative edge connects two components and the set of all negative edges forms a spanning tree of the subcomplex. We use this spanning tree to find the *minimal* 1-cycles, i.e. the smallest sets of non-bounding edges that envelop the holes in the subcomplex.

5.4.1 Minimal 1-Cycles Algorithm

When Algorithm 5.12 terminates each unpaired positive edge corresponds to a non-bounding edge of a 1-cycle, its slot in the hash table T is empty otherwise the edge would bound. If we add this unpaired positive edge to the spanning tree of negative edges a cycle is created. However, this cycle is not necessarily the smallest cycle bounding the hole, it's a homologous cycle among possibly other cycles surrounding the hole.

To find the minimal 1-cycle we check if there are any edges in the set of paired positive edges that connect to vertices on the cycle. If such connections exist we augment the cycle with those edges. Then, we compute the shortest path between the endpoints of the original unpaired positive edge modulo their incident edge.

Concretely, computing minimal 1-cycles begins by constructing a graph, $G = (V, E)$, where the vertices are the set of 0-simplices and the edges are the set of negative 1-simplices. The graph G is the spanning tree of the subcomplex. The minimal 1-cycles can then be found in two steps.

Step 1

For each unpaired positive edge we compute and store the shortest path from one endpoint of the edge to the other in G. This path can be computed efficiently using breadth-first search (BFS) [39]. Next, we determine if this is indeed the shortest path between the endpoints in the subcomplex.

Step 2

Given an initial shortest path computed in the previous step, we proceed to verify whether the path constitutes a minimal 1-cycle. First, we project the vertices onto a plane and find those that are contained in the polygon formed by the path. To confirm if a given vertex lies inside the polygon we use the classic point-in-polygon algorithm from computational geometry. The algorithm counts the number of times a ray through each vertex intersects the edges of the polygon. A vertex is inside the polygon if the number of intersections is odd, otherwise the vertex is outside.

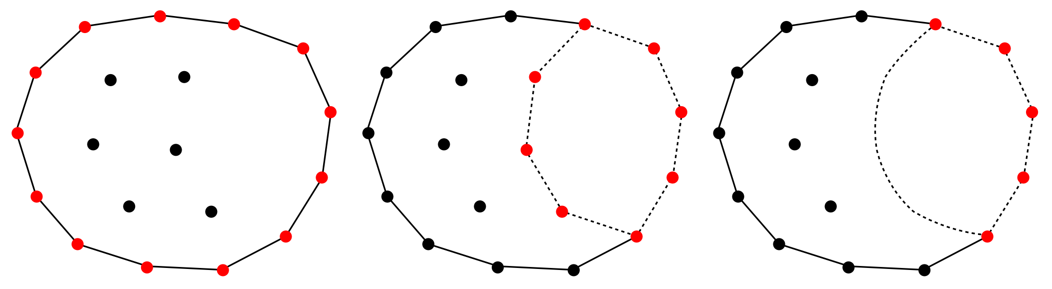


Figure 5.5: The red vertices constitute an initial followed by a final minimal 1-cycle. Edges inside the graph are akin to a chord from one vertex to another on the final path.

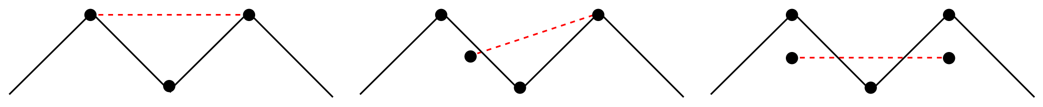

(a) Boundary to boundary.
(b) Interior to boundary.
(c) Interior to interior.

Figure 5.6: The three corner cases to handle when checking for a paired positive edge from one vertex to another.

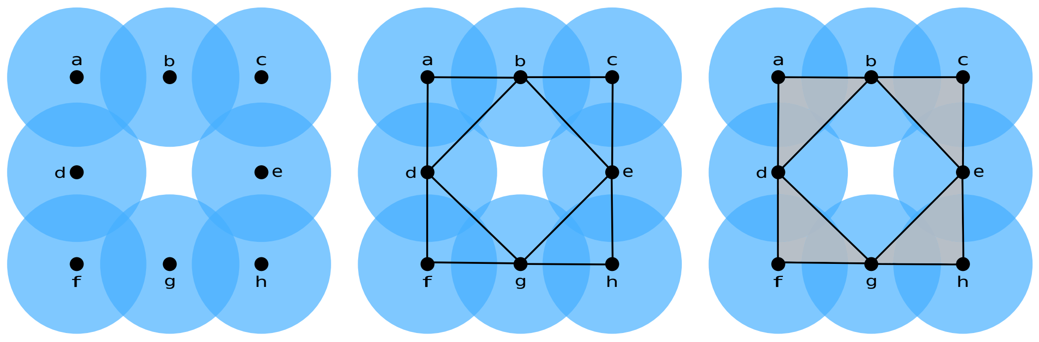
After determining if there are vertices inside the polygon, we then check for the existence of a paired positive edge from one vertex to another. If such edges exist, we add them to the set of edges that compose the initial shortest path. This action is analogous to adding a chord to the cycle created by the unpaired positive edge in G, Figure 5.5. For non-convex polygons there are three corner cases to evaluate: the edge between boundary vertices, the edge between interior and boundary vertices, and the edge between interior vertices must all lie inside the polygon, Figure 5.6. We then run BFS again on the updated path. The end result is the shortest path between the two endpoints of the unpaired positive edge in the subcomplex bounding the hole.

5.4.2 Example

A procedure for computing minimal 1-cycles is given in Algorithm 5.15. Consider the point cloud shown in Figure 5.7a. The connectivity graph and simplicial complex of the point cloud are shown in Figures 5.7b and 5.7c, respectively. This complex consists of 8

Algorithm 5.15 Minimal 1-Cycles

```
Input: Hash table $T$
Output: Set of minimal 1-cycles
1: $\sigma^+ \leftarrow \{\sigma \in \partial_2(\sigma^{1\ldots j}) \mid \sigma$ positive$\}$
2: $\sigma^- \leftarrow \{\sigma \in \partial_2(\sigma^{1\ldots j}) \mid \sigma$ negative$\}$
3: $G \leftarrow$ spanning tree of $\sigma^-$
4: for all $\sigma \in \sigma^+$ do
5:     if $T[\sigma]$ is empty then
6:         $T[\sigma]$.path $\leftarrow$ BFS$(G, \sigma.u, \sigma.v)$
7:         if $T[\sigma]$.path has cords then
8:             $T[\sigma]$.path $\leftarrow$ add cords
9:             $T[\sigma]$.path $\leftarrow$ BFS$(T[\sigma]$.path$, \sigma.u, \sigma.v)$
10:        end if
11:    end if
12: end for
```



(a) Point cloud. (b) Connectivity graph. (c) Simplicial complex.

Figure 5.7: The example dataset for computing minimal 1-cycles in Section 5.4.2.

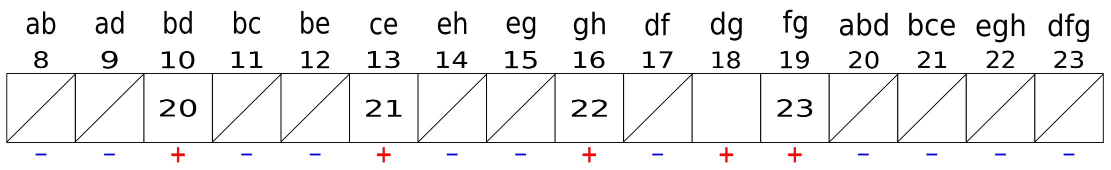
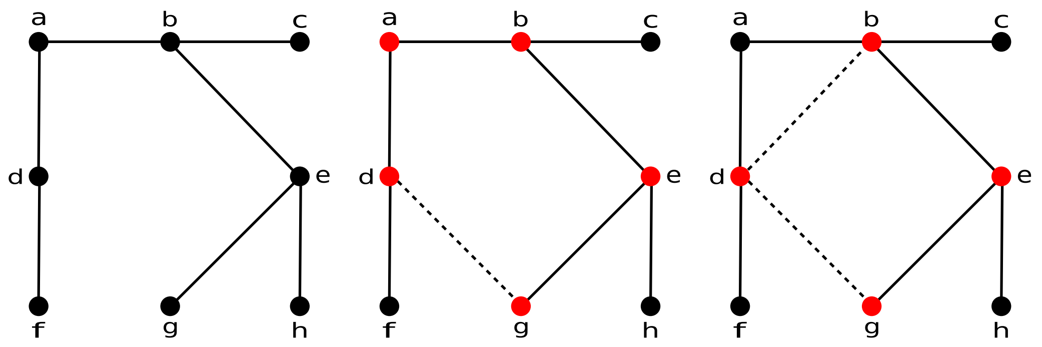


Figure 5.8: The hash table state after determining persistent 1-cycles on the dataset shown in Figure 5.7.



(a) Spanning tree. (b) Initial 1-cycle. (c) Final 1-cycle.

Figure 5.9: The process of finding minimal 1-cycles on the dataset shown in Figure 5.7.

points, 12 edges, and 4 triangles with $\beta_0 = \beta_1 = 1$. We find the minimal 1-cycles using the output of the fast incremental persistence algorithm (Figure 5.8) as follows.

We begin by initializing a set of positive edges, a set of negative edges, and generate a spanning tree using the negative edge set. The spanning tree is shown in Figure 5.9a where $\sigma^+ = \{bd, ce, gh, dg, fg\}$ and $\sigma^- = \{ab, ad, bc, be, eh, eg, df\}$. Next, we iterate over the elements of the positive edge set. Upon finding the only unpaired positive edge, dg, we perform the following actions.

First, we compute the shortest path from d to g in the spanning tree which gives us an initial 1-cycle consisting of the vertex set $\{d, a, b, e, g\}$, Figure 5.9b. Next, we determine if there are any projected vertices inside the polygon formed by the initial shortest path. In this case there are none. Then, we check if there exists any paired positive edges that serve as chords in the 1-cycle. In this example, we find that bd can be added as a chord. Finally, we recompute the shortest path from d to g thus giving the minimal 1-cycle that consists of the vertex set $\{d, b, e, g\}$, Figure 5.9c.

5.4.3 Correctness

The set of all positive edges form the fundamental cycle basis of the spanning tree G. Furthermore, the cycle space of G is a collection of Eulerian subgraphs. Recall that a subgraph of G is said to be Eulerian if each of its vertices has an even number of incident edges. Eulerian subgraphs form a vector space over $\mathbb{Z}_2$. The vector addition operation is the symmetric difference between subgraphs, the identity operation is multiplication by the scalar 1, and multiplication by the scalar 0 takes every element to the empty graph thus forming the additive identity element for the cycle space.

The algorithm reduces the size of the Eulerian subgraphs until a smallest non-bounding subgraph is found. In the first step, we add an unpaired positive edge to G hence creating a cycle and the initial Eulerian subgraph. In the second step, we potentially add paired positive edges if they act as chords in the cycle therefore creating smaller Eulerian subgraphs. Lastly, we compute the smallest non-bounding Eulerian subgraph that contains the unpaired positive edge.

Minimal 1-Cycle Lemma. *If c is a minimal 1-cycle, then it is a smallest non-bounding Eulerian subgraph.*

Proof. Let c be a minimal 1-cycle and suppose that c is not a smallest non-bounding Eulerian subgraph. Then there exists a paired positive edge e that can be added as a chord to c. We can reduce the size c by taking the symmetric difference of c and the cycle created by adding e thus obtaining a 1-cycle with less edges than c, a contradiction. □

5.4.4 Analysis

The running time of the algorithm is dominated by the search for the shortest path using BFS. BFS runs in time proportional to $O(|V| + |E|)$ where $|V|$ is the number of 0-simplices and $|E|$ is the number of negative 1-simplices. Observe that the problem of finding a minimal 1-cycle for one hole is independent of finding a minimal 1-cycle for another hole in the subcomplex. Therefore, the workload for finding minimal 1-cycles can be distributed and all minimal 1-cycles of a subcomplex can be computed in parallel.

5.5 1-Cycle Computation Experiments

In this section we compare our Fast Incremental Persistence (FIP) algorithm for computing 1-cycles, explained in Section 5.3, against version 1.5 of the Persistent Homology Algorithm Toolbox (PHAT) [54]. PHAT is a state of the art software library providing methods for computing the persistence pairs of a filtered cell complex represented by an ordered boundary matrix with $\mathbb{Z}_2$ coefficients.

5.5.1 Experimental Setup

Our implementation of FIP, written in C++, makes use of a single CPU core. FIP is compared against several algorithmic variants provided by the PHAT C++ library interface. These variants include the 'standard' [63], 'twist' [64], 'chunk' [65], and 'spectral sequence' [63] algorithms using the bit tree pivot column representation. In PHAT, all but the standard algorithm exploit the special structure of the boundary matrix to take shortcuts in the computation.

In addition, we've compiled PHAT with OpenMP support to allow the chunk and the spectral sequence algorithms to make use of multiple CPU cores. The ordered boundary matrices, used as input by both FIP and PHAT, were generated using our parallel implementation of constructing the Vietoris-Rips complex (PVR) as explained in Chapter 3. All experiments were performed using a single node 64-bit GNU/Linux machine with up to 24 available CPU cores.

5.5.2 Experimental Results

The runtime performance of FIP versus PHAT on a point cloud of size 15k points is shown in Figure 5.10. Within the plot, the size of the complex (number of simplices) as a function of the radius and the runtime (seconds) for computing 1-cycles is shown for all the algorithms. From the results we see that FIP is more than an order of magnitude faster than PHAT as the simplicial complex grows in size. It is also worth noting that the speed of FIP at the lower range of the radius value make it ideal for computing persistent 1-cycles in real-time on 3D point cloud datasets.

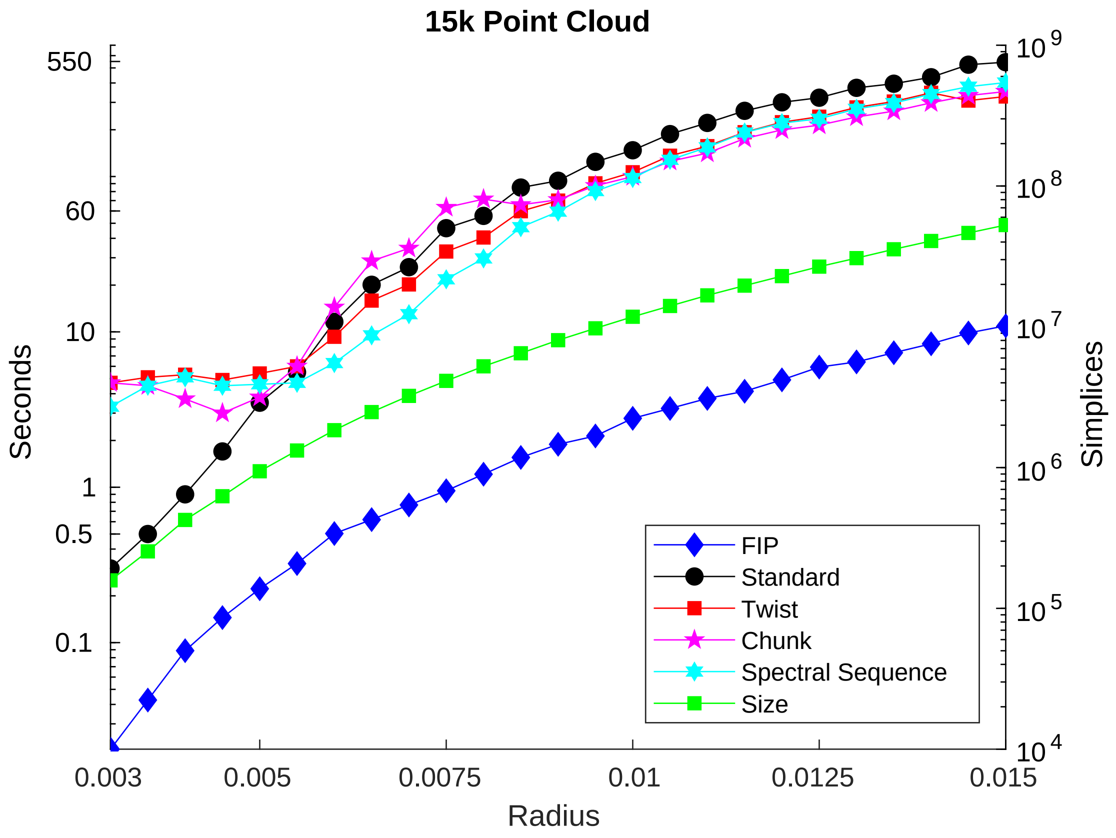
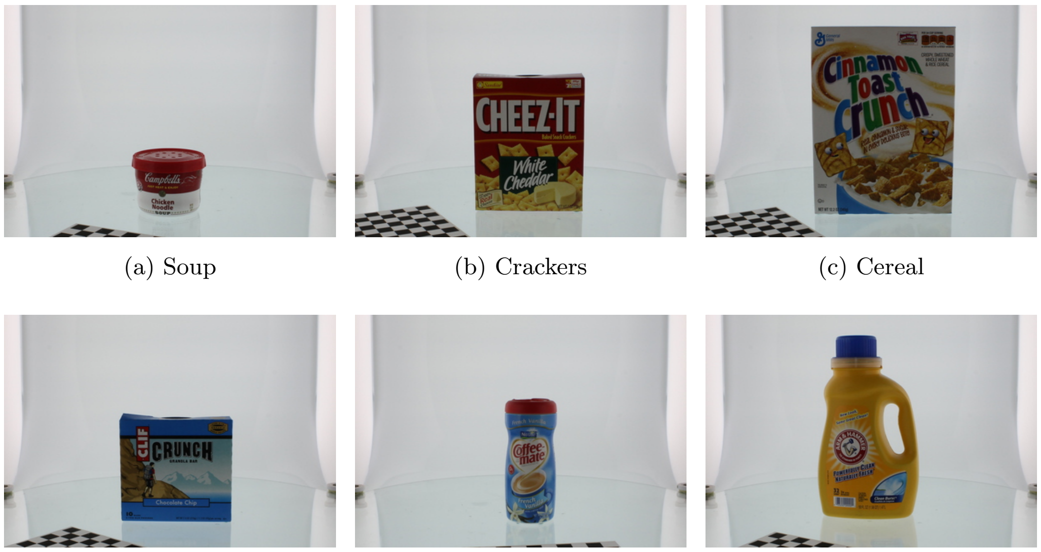


Figure 5.10: The runtime of Fast Incremental Persistence (FIP) and several variants of the Persistent Homology Algorithm Toolbox (PHAT) along with the size of the complex for a given radius using a set of non-uniformly sampled points in $\mathbb{R}^3$. The results of each plot are averaged over 10 independent runs per plot point.



(a) Soup (b) Crackers (c) Cereal

(d) Snack Bars (e) Coffee Creamer (f) Detergent

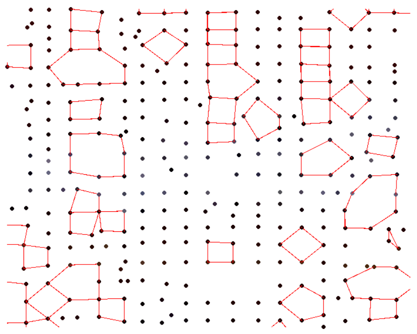
Figure 5.11: The RGB images of the objects from the BigBIRD dataset [2] used in the experiments.

5.6 Hole Boundary Detection Experiments

This section provides an experimental evaluation of the minimal 1-cycles algorithm presented in Section 5.4. The objective of the experiments is to show the applicability of finding hole boundary points (minimal 1-cycles) for aiding surface reconstruction of 3D point clouds (Algorithm 5.15).

5.6.1 Experimental Setup

The experiments were performed using the BigBIRD dataset [2] on the following six objects: soup, crackers, cereal, snack bars, coffee creamer, detergent. Figure 5.11 shows the RGB images of the objects. The algorithms were implemented and tested using MATLAB on a 64-bit GNU/Linux machine with a single CPU core. The point clouds were first downsampled using a 2 mm voxel grid. Then, we induced additional noise by randomly removing 10, 20, and 30 percent of the points. For each point cloud, the maximum distance value of the Vietoris-Rips (VR) complex was set to 3 mm.


(a) Minimal 1-cycles.

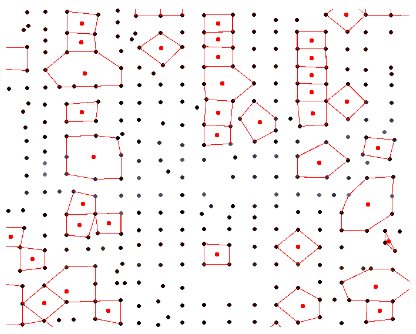

(b) Minimal 1-cycles and centroids.

Figure 5.12: An example of locating hole boundary points and filling the holes with their centroids.

5.6.2 Experimental Results

Holes on the surface of a 3D point cloud are found and filled in the following way. The points on the path of a minimal 1-cycle projected onto a plane form a polygon. We compute the centroid of this polygon and add it to the reconstructed point cloud, Figure 5.12. Minimal 1-cycles composed of a maximum of 8, 16, and 32 points are computed for point clouds that have been reduced by 10, 20, and 30 percent, respectively. Figure 5.12 shows the reduced point clouds while Figure 5.12 shows the reconstructed point clouds with the computed centroids. From the results, we can see that the centroid is a good approximation for reconstructing small holes formed by convex polygons. As the point clouds are further reduced, the holes become larger and the number of non-convex polygons increases. Thus, for bigger complicated holes a more sophisticated hole filling scheme is required.

A comparison of the runtimes between Algorithms 5.11 and 5.12 is shown in Tables 5.1 - 5.4. For each table the column categories are as follows: Points - number of points processed, 2-simplices - number of 2-simplices processed, Betti 1 - number of holes detected in the point cloud, Time - CPU time in seconds to compute persistent 1-cycles. In Tables 5.1 - 5.3 we can see that our fast incremental persistence algorithm outperforms the original incremental persistence algorithm for point clouds of various sizes. Runtimes on the raw point clouds (i.e. no downsampling) are reported in Table 5.4. Runs that did not complete were not reported. Table 5.4 highlights the scalability in the number of 2-simplices of our fast approach for computing persistence incrementally.

5.7 Conclusion

Data produced by depth sensors is corrupted to a varying extent by noise. This noise can manifest itself as missing points on the surfaces of objects. Preprocessing of noisy sensor data is an essential step in the pipeline of many computer vision and robotic applications. In this chapter, we developed algorithms and data structures for incrementally computing topologically persistent 1-cycles with application to hole boundary point detection. In summary:

- We provided a fast incremental persistence algorithm for calculating 1-cycles.
- We developed an algorithm for computing non-bounding minimal 1-cycles, i.e. the smallest set of points on the boundary of a hole, based on the spanning tree output of the fast incremental persistence algorithm.
- We showed experimentally the application these algorithms to 3D point cloud surface reconstruction.

(a) 10% point reduction.

(b) 20% point reduction.

(c) 30% point reduction.

(d) 10% point reduction.

(e) 20% point reduction.

(f) 30% point reduction.

Cinnamon Toast Crunch
(g) 10% point reduction.

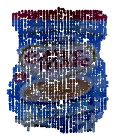
Cinnamon Toast Crunch
(h) 20% point reduction.

Cinnamon Toast Crunch
(i) 30% point reduction.

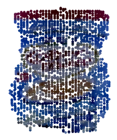
(j) 10% point reduction.

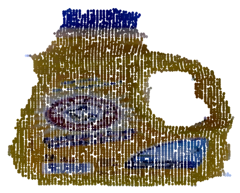
(k) 20% point reduction.

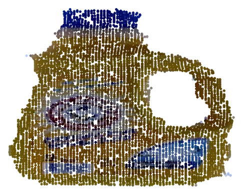
(l) 30% point reduction.


(m) 10% point reduction.

(n) 20% point reduction.


(o) 30% point reduction.


(p) 10% point reduction.


(q) 20% point reduction.

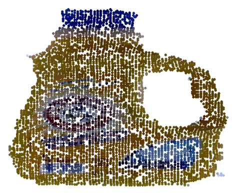
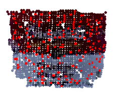
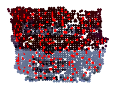
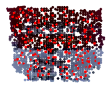
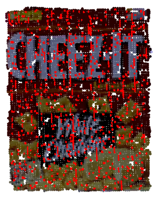

(r) 30% point reduction.

Figure 5.12: The reduced point clouds with 10, 20, and 30 percent of the points randomly removed.

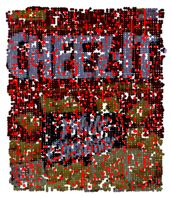

(a) 8 point 1-cycles.


(b) 16 point 1-cycles.

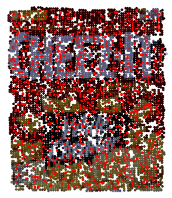

(c) 32 point 1-cycles.

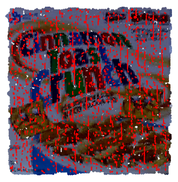


(d) 8 point 1-cycles.

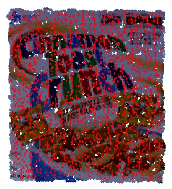


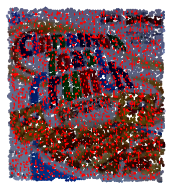
(e) 16 point 1-cycles.

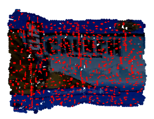

(f) 32 point 1-cycles.

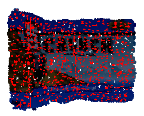


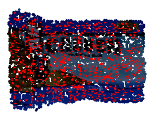
(g) 8 point 1-cycles.

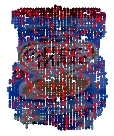

(h) 16 point 1-cycles.

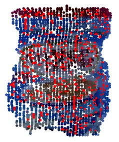

(i) 32 point 1-cycles.

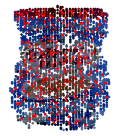

(j) 8 point 1-cycles.

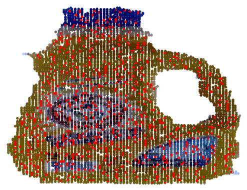

(k) 16 point 1-cycles.

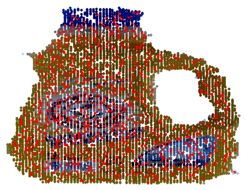

(l) 32 point 1-cycles.


(m) 8 point 1-cycles.


(n) 16 point 1-cycles.


(o) 32 point 1-cycles.


(p) 8 point 1-cycles.


(q) 16 point 1-cycles.

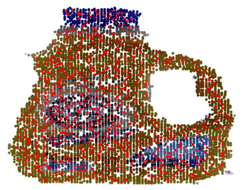

(r) 32 point 1-cycles.

Figure 5.12: The reconstructed point clouds with centroids (red) computed using minimal 1-cycles of maximum length of 8, 16, and 32 points.

Object	Points	2-simplices	Betti 1	Time (Alg. 5.11)	Time (Alg. 5.12)
Soup	1,533	4,451	200	1.744	0.353
Crackers	6,571	19,724	994	5.884	1.614
Cereal	11,279	31,589	2031	8.841	2.533
Snack Bars	4,902	12,513	951	2.862	0.961
Coffee Creamer	2,643	7,669	394	2.744	0.536
Detergent	6,585	18,976	868	5.673	1.340

Table 5.1: The results of computing persistent 1-cycles for the objects in Figure 5.11 with the point clouds reduced by 10%.

Object	Points	2-simplices	Betti 1	Time (Alg. 5.11)	Time (Alg. 5.12)
Soup	1,363	3,079	209	0.922	0.212
Crackers	5,841	13,932	987	3.706	0.954
Cereal	10,026	22,102	2,006	5.442	1.533
Snack Bars	4,357	8,750	935	1.851	0.585
Coffee Creamer	2,349	5,364	394	1.541	0.339
Detergent	5,853	13,383	908	3.551	0.813

Table 5.2: The results of computing persistent 1-cycles for the objects in Figure 5.11 with the point clouds reduced by 20%.

Object	Points	2-simplices	Betti 1	Time (Alg. 5.11)	Time (Alg. 5.12)
Soup	1,193	2,073	195	0.605	0.130
Crackers	5,111	9,220	929	2.049	0.542
Cereal	8,773	14,787	1720	3.020	0.815
Snack Bars	3,813	6,035	733	1.153	0.310
Coffee Creamer	2,056	3,711	310	0.961	0.194
Detergent	5,122	8,995	830	1.961	0.477

Table 5.3: The results of computing persistent 1-cycles for the objects in Figure 5.11 with the point clouds reduced by 30%.

Object	Points	2-simplices	Betti 1	Time (Alg. 5.11)	Time (Alg. 5.12)
Soup	3,726	92,661	0	869.694	2.018
Crackers	17,586	536,008	0	8592.778	12.067
Cereal	32,148	1,083,191	0	-	24.917
Snack Bars	13,622	456,727	0	7802.860	10.827
Coffee Creamer	6,425	155,849	1	1129.257	3.734
Detergent	15,706	287,429	4	2435.037	6.444

Table 5.4: The results of computing persistent 1-cycles for the objects in Figure 5.11 on the raw point clouds.

Chapter 6

Point Cloud Signatures

Shape analysis is essential for computer vision applications such as matching, retrieval, reconstruction, detection, and classification. Automatically recognizing shapes in 3D datasets continues to attract interest in the research community. In recent times, this research activity has been driven by the development of economical 3D sensor (RGB-D, stereo, time-of-flight, etc.) technologies.

Shape recognition is performed by either a local or global approach. Local descriptors rely on keypoints extracted from surfaces. The aim of descriptors using local methods is to single out points that are distinctive in order to allow for effective description and matching. Within the local neighborhood of each keypoint, geometric information is encoded to obtain a compact representation of the input data invariant up to a predefined transformation (translation, rotation, scaling, point density variations, etc.). Local descriptors are ideal for performing operations in cluttered scenes containing occluded objects. Global descriptors encode object geometry and are not computed for individual points, but for a whole cluster of points that represents an object. Although global descriptors require a clean segmentation of an object they are useful for many operations such as shape-based recognition, retrieval, clustering, and classification.

In this chapter, we introduce a new global shape descriptor STPP (Signature of Topologically Persistent Points) which is based on computing the persistence of the zeroth and first homology groups of a 3D point cloud. This signature is used to create a feature vector where the birth and death of the generators of homology correspond to the evolution of the number of connected components and the number of holes in the dataset. The advantages of STPP are the following:

- It can be computed quickly and efficiently on point cloud data.
- No preprocessing (sampling, hole filling, surface normal calculation, etc.) is required.
- A small number of features are used.
- Only a single tuning parameter is needed.

In addition, STPP can cope with noisy datasets without compromising on performance.

6.1 Related Work

The area of shape analysis has generated a considerable amount of research. Shape representation and description is a field of shape analysis with many important applications. In this section, we review relevant global descriptors for describing shapes composed of 3D point cloud data followed by developments in shape analysis using topological persistence that has led to this work.

Rusu et al. generalize the fast point feature histograms (FPFH) idea to create a descriptor that captures the relationship of local geometric parts in whole objects [66]. This descriptor, termed the global fast point feature histogram (GFPFH), is used for scene interpretation in robotic manipulation scenarios. Their work is followed by a 3D point cloud descriptor, called the viewpoint feature histogram (VFH) descriptor, which incorporates both geometry and viewpoint [67].

A multimodal perception system consisting of hierarchical object geometric categorization and appearance classification for personal robots introduces the global radius-based surface descriptor (GRSD) [68]. This descriptor developed by Marton et al. is derived from the radius-based surface descriptor [69], and can generalize over objects with similar geometry thus limiting the possibilities of corresponding an object instance to its 3D point cloud cluster.

The clustered viewpoint feature histogram (CVFH) descriptor, based on the VFH descriptor, is described by Aldoma et al. [70]. The main idea behind the semi-global

CVFH descriptor is to take advantage of object parts obtained by a depth sensor and use them to build a coordinate system similar to the VFH descriptor. It improves upon issues that make VFH sensitive to missing point cloud data due to partial occlusions, segmentation, or sensor artifacts.

A global object recognition pipeline utilizing a 3D semi-global describer called oriented, unique, and repeatable CVFH (OUR-CVFH) [71] is proposed by the creators of CVFH. OUR-CVFH employs a method to estimate semi-global unique reference frames (SGURF) computed on the surface of an object as seen from a single viewpoint. By exploiting the orientation provided by these reference frames, OUR-CVFH efficiently encodes the geometrical properties of an object surface.

Wohlkinger and Vincze introduce an ensemble of shape functions (ESF) descriptor [72]. This global shape descriptor is built on three distinct shape functions that describe distance, angle, and area distributions on the surface of a partial point cloud. ESF allows for real-time classification of objects sensed with an RGB-D sensor based on learning from synthetic CAD models.

Kanezaki et al. present the concatenated voxelized shape and color histograms descriptor, (Con)VOSCH [73]. The descriptor combines the GRSD and the circular color cubic higher-order local auto correlation descriptors (C^3-HLAC). It is designed to facilitate object classification by considering geometric and visual features in a unified manner.

Fehr et al. put forward a global covariance-based point cloud descriptor for object detection and recognition [74]. The descriptor is constructed by forming the covariance matrix from an RGB-D feature vector. The authors show that covariance descriptors are computationally fast and provide a compact (low dimensionality) representation of a 3D point cloud object.

In contrast to the aforementioned global descriptors, STPP encodes the *topological* information of a 3D point cloud. Our work is inspired by an early study of shape description and classification via persistent homology [75]. Additional inspiration comes from the results of Li et al. [76] where persistence diagrams built from functions defined on objects serve as compact and informative descriptors for images and shapes.

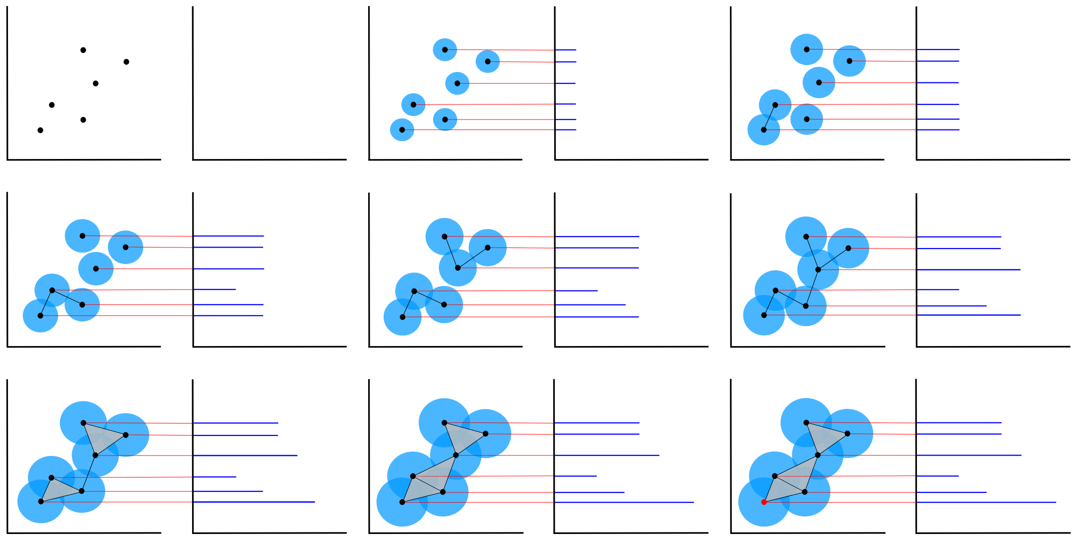


Figure 6.1: The evolution of a scale parameter defining the neighborhood radius about each ordered point and the corresponding barcode diagram. When the neighborhoods of two points overlap, the younger point dies while the elder point survives. This produces a topologically persistent signature of the point cloud.

6.2 Problem Formulation

Given a topological space $X = \{x_0, \ldots, x_{m-1}\} \in \mathbb{R}^3$ where $x_0, \ldots, x_{m-1}$ are the points in a point cloud acquired by a 3D sensor, our goal is to compute a topologically persistent description of the point cloud. We formulate a solution to the problem by creating a framework that:

- Models the input space as a Vietoris-Rips (VR) complex.
- Uses a fast incremental persistence algorithm to compute the zeroth and first generators of homology through a filtration, Figure 6.1.
- Forms a feature vector comprised of the birth-death pairing of the homology generators.

The end result is the capability to distinguish between noisy 3D point cloud datasets using only topological features.

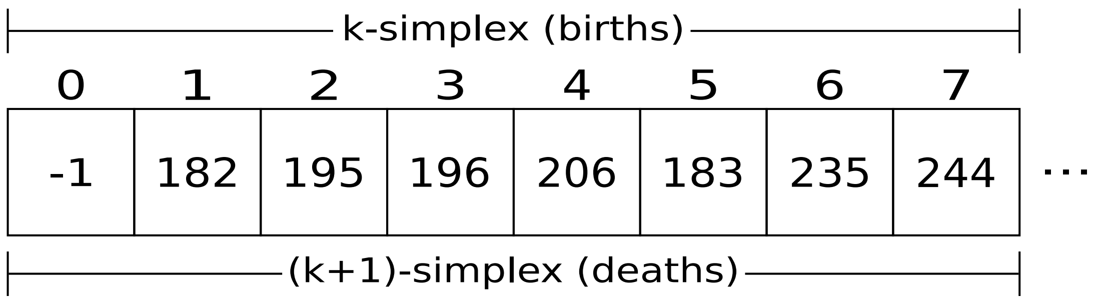


Figure 6.2: The birth-death pairings of the homology generators are encoded in a feature vector where the indices of the vector correspond to the birth of a k-simplex and the entries of the vector correspond to the index of a destroying $(k+1)$-simplex. Unpaired simplices, denoted by -1 entries, represent k-dimensional holes.

6.3 Topologically Persistent Feature Vector

In this section, we give an overview of barcode/persistence diagrams and the role they play in comparing datasets. We then describe our approach to computing a topologically persistent signature of a 3D point cloud. This approach is based on our algorithms for constructing the VR complex, computing 0-cycles, and computing 1-cycles.

6.3.1 Overview

A barcode or persistence diagram encapsulates a concise description of the topological changes that occur during a filtration. Intuitively, a k-dimensional hole born at time b and filled at time d gives rise to a point (b, d) in the kth persistence diagram or an interval in the kth barcode diagram. Therefore, a persistence diagram is a multiset of points in $\mathbb{R}^2$ while a barcode diagram is an equivalent multiset of intervals in $\mathbb{R}$.

The use of distances between barcode/persistence diagrams has received attention in applications [77–79] where they serve as topological proxies for the input data. Distances between the diagrams serve as measures of the similarity between datasets. These distances can be expressed as a bottleneck or Wasserstein distance between two planar point sets using the L_∞ metric.

6.3.2 Construction

Although the distance between barcode/persistence diagrams has been shown to measure the similarity between some datasets, we observe that a single scalar value is not enough to discriminate between massive 3D point clouds. In contrast to the work mentioned in the previous subsection we construct a vector of topologically persistent features as follows.

The output of Algorithm 5.12 is a sorted birth and death pairing between a k-simplex and $(k+1)$-simplex, respectively. We encode this information in a feature vector where the indices of the vector correspond to the birth of a k-simplex and the entries of the vector correspond to the index of a destroying $(k+1)$-simplex, Figure 6.2. This pairing is unique, e.g. a 0-simplex is terminated by exactly one 1-simplex and a 1-simplex is terminated by exactly one 2-simplex. An unpaired simplex is a simplex of infinite persistence and represents a k-dimensional hole.

6.4 Analysis

The VR complex can be computed quickly and efficiently in parallel (Chapter 3). With a disjoint-set data structure employing weighted merging for union and path compression for find, the amortized time per operation for computing 0-cycles is $\mathcal{O}(\alpha^{-1}(n))$, where n is the number of 0-simplices and $\alpha^{-1}(n)$ is the very slowly growing inverse of the Ackermann function (Chapter 4). For computing 1-cycles our fast incremental persistence algorithm operates in nearly linear time, $\mathcal{O}\big(c \cdot n\big)$, where n is the number of 2-simplices and the constant c is the amount of work done per 2-simplex thus making the computation of persistent 1-cycles on large 3D datasets feasible (Chapter 5).

6.5 Experiments

In this section, we set up and evaluate a processing pipeline for computing topological signatures as described in Section 6.3. The experiments were performed using the RGB-D Object Dataset [3] where we focus on the task of object category classification. Construction of the simplicial complexes and computing topological persistence were done on a multi-core 64-bit GNU/Linux machine.

Figure 6.3: A subset of objects from the RGB-D Object Dataset [3] used for the experiments.

6.5.1 Experimental Setup

The RGB-D Object Dataset consists of 300 objects divided into 51 categories and provides roughly 250,000 point clouds. Following the experimental procedure in [3] we subsample the dataset by taking every fifth point cloud. This gives us approximately 45,000 point clouds upon which we run classification experiments. To perform category recognition, we randomly leave one object out from each category for testing and train the classifiers on the point clouds of the remaining objects. Classification is performed using an SVM classifier and RBF kernel [80]. The accuracies are averaged over 10 trials.

6.5.2 Experiment Results

Experiment 1

In this experiment, we compare STPP against five different global 3D point cloud descriptors: VFH, GRSD, CVFH, OUR-CVFH, and ESF. Implementations of each of these descriptors are publicly available in the Point Cloud Library (PCL).[1] We compute STPP features using a single step filtration as follows.

The VR complex representation of the data is computed by first sorting the points by their Cartesian coordinates. A nearest neighbors search about each point is then

[1] http://www.pointclouds.org

Descriptor	Category Accuracy	Number of Features
GRSD	8.89 ±0.55	4
STPP	23.32 ±1.58	2
VFH	23.61 ±1.92	4
OUR-CVFH	26.91 ±1.64	6
CVFH	29.39 ±1.35	5
ESF	39.60 ±1.01	6

Table 6.1: The accuracy results of object category classification.

carried out up to a radius of 3.5 mm using a kd-tree. Next, we create an edge list where an edge exists between two neighboring points. We then proceed to create a triangle list by finding all three cliques in the edge graph. Once we have the edge and triangle lists we can compute the signature of the point cloud using Algorithm 5.12.

The last stage of the pipeline uses the output of the incremental persistence computation to form a feature vector. The indices of the vector range over the sorted 0-simplices and 1-simplices while the entries consist of the indices of the destroying 1-simplices and 2-simplices. We also construct feature vectors for each of the five global descriptors on the subsampled data. These feature vectors are then used to train separate classifiers for comparison as shown in Table 6.1.

Experiment 2

To gain an insight into the effectiveness of the STPP features for performing category classification, we create separate feature vectors for the generators of homology: 0-cycles (Betti 0), 1-cycles (Betti 1), and the combination of 0-cycles and 1-cycles (Betti 0+1). We then use these feature vectors to compute the classification accuracy on category subsamples. The results of this experiment are presented in Figure 6.4.

Experiment 3

This experiment considers the affect of exposing the STPP descriptor to different noise levels. To compare the effect of noise on the descriptor we randomly perturb all the points by $\delta \in [-0.0005, 0.0005]$, $\delta \in [-0.001, 0.001]$, and $\delta \in [-0.003, 0.003]$. This perturbation essentially deforms the surface of the point cloud. We then recompute the signatures and rerun the category classification experiments as previously described. The average accuracy of STPP was found to be 23.38% ±1.72 for $\delta \in [-0.0005, 0.0005]$, 22.35% ±2.63 for $\delta \in [-0.001, 0.001]$, and 16.10% ±1.04 for $\delta \in [-0.003, 0.003]$.

Experiment 4

In this experiment, we explore the use of STPP as a feature in a covariance descriptor. A covariance descriptor condenses features (position, color, normals, etc.) over the entire point cloud of an object. Concretely, let $f_i \in \mathbb{R}^p$, for $i = 1, 2, \ldots, n$, be a feature vector consisting of the n points of an object. The covariance descriptor of the object is then defined as

$$C = \frac{1}{n-1}\sum_{i=1}^{n}(f_i - \mu_f)(f_i - \mu_f)^T,$$

where μ_f is the mean feature vector and $C \in \mathcal{S}^p_{++}$ is the space of $p \times p$ Symmetric Positive Definite (SPD) matrices. Distances between covariance descriptors are approximated using the log-Euclidean metric. We use the Betti 0 information, defined for every point, and a three step filtration with maximum scale values of 3, 4, and 5 mm. First, we obtain a baseline accuracy of 67.20% ±3.00 using the Cartesian coordinate, color, and surface normal coordinate of each point as features. Then, we add in the Betti 0 values for each filtration step as features. This improves the overall accuracy to 68.29% ±3.01.

Discussion

In these experiments we compare the performance of several state of the art global geometric descriptors with a STPP, global topological descriptor. The classification accuracy of STPP on the RGB-D Object Dataset is competitive with other global descriptors. Furthermore, we see that combining the birth-death pairing of the homology generators (Betti 0+1) increases the overall classification accuracy of the descriptor. STPP shows robustness to noisy datasets with a moderate degradation in accuracy. We also see that encoding both geometrical (surface normal) and topological (Betti 0) features in a covariance descriptor helps improve the classification accuracy.

In terms of storage and computational performance, STPP is compact and fast to

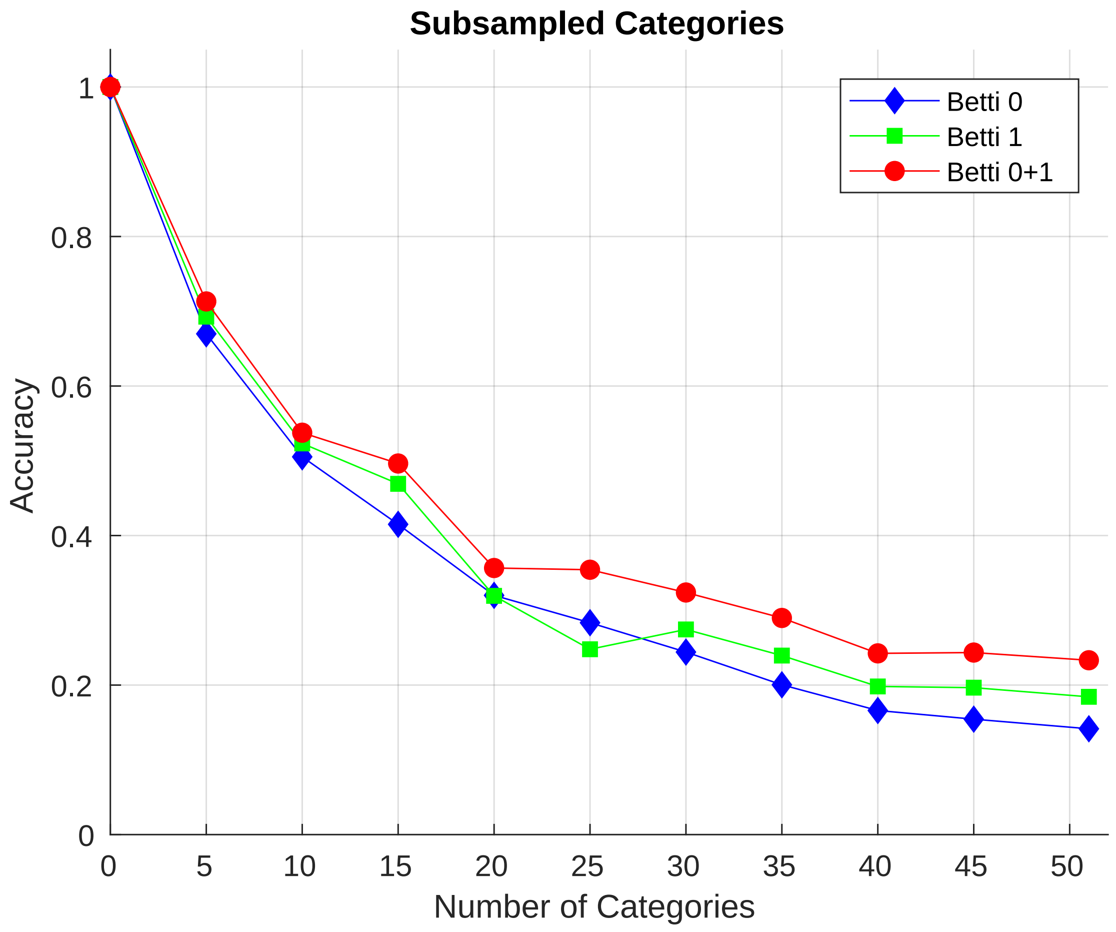


Figure 6.4: The classification accuracy results on the subsampled categories (1-51) using Betti 0 (0-cycles), Betti 1 (1-cycles), and Betti 0+1 (0-cycles and 1-cycles) as features.

compute when compared to other 3D point cloud descriptors. Moreover, no preprocessing of the point cloud is required to support the descriptor. Using only two features, the STPP feature vector stores a double precision value for each vertex (0-simplex) and edge (1-simplex) in a point cloud. For example, given a point cloud consisting of 10k points and 70k edges, STPP stores $10,000 \cdot 8 + 70,000 \cdot 8 = 640,000$ bytes. Computing STPP on the same point cloud takes 0.040 seconds for 0-cycles and 0.200 seconds for 1-cycles giving a total of 0.240 seconds.

6.6 Conclusion

In this chapter we presented STPP, a novel 3D point cloud descriptor that uses persistent homology to compute a topological signature based on the birth and death of 0-cycles and 1-cycles. Persistent homology allows us to track topological features, such as the number of connected components and holes in a dataset, at varying spatial scales. 3D point cloud description is a necessary prerequisite for high-level computer vision tasks including object detection and classification.

STPP encodes the basic topological structure of 3D point cloud data. To show the feasibility of STPP, we implemented a pipeline that computes and compares the topological signature of 3D point cloud objects against geometrical-based descriptors. We showed that the classification performance of STPP is competitive, it inherently deals with noisy datasets, and is theoretically well-founded. We also demonstrated that for certain combinations, topological features provide complementary information, which in turn improves the performance of covariance-based descriptors.

Chapter 7

Conclusion and Discussion

In this thesis, we studied how topological features can aid us in the processing of 3D point cloud datasets. We did this using the primary tool of TDA, persistent homology, which allows us to compute homology groups (e.g. connected components, holes, voids, and higher dimensional analogs), at multiple resolutions. Not only did we develop efficient algorithms for computing persistent homology in low dimensions, but we also showed novel applications of these algorithms to relevant problems in computer and robotic vision systems. In the following we summarize our contributions, discuss open problems, and point out future research directions. We then conclude with a broader discussion on the future of 3D point cloud processing.

7.1 Contributions and Open Problems

To be a useful for TDA, a simplicial complex must satisfy certain theoretical properties. Specifically, if we build a simplicial complex on a set of points sampled from a space then the homology of the complex has to approximate the homology of the space. In Chapter 3, we studied the problem of building a Vietoris-Rips (VR) complex over a filtration of various sized point clouds. We showed how the VR complex can be quickly computed in $\mathbb{R}^3$ and that its construction is suitable for parallelization. As a result, our construction of the VR complex is ideal for real-time applications on 3D point cloud datasets. The development of new simplicial complexes is an active area of research. An immediate research question is whether we can obtain a better simplicial complex representation in terms of computational performance and memory efficiency.

Data segmentation is a necessary preprocessing step in computer and robotic vision systems when handling 3D point clouds. The stability of higher level tasks is contingent upon the condition of the segmented data. In Chapter 4, we explored methods for object and region segmentation using topological persistence. By combining both local and global properties of the data we showed how homogeneous areas can be identified and extracted under the presence of noise. From this work the following question arises: can we utilize topological persistence to find regions that conform to object boundaries better than existing methods.

The knowledge of the existence and location of holes in a dataset is essential in 3D point cloud processing. This information can be applied towards the preprocessing of noisy sensor data. In Chapter 5, we presented algorithms and data structures for determining both the number of holes (1-cycles) and the boundary points that comprise the holes. The challenges that lie ahead involve the optimization and parallelization of these methods for real-time 3D point cloud processing applications.

The analyses of shapes in point cloud data is an important component of many 3D computer vision systems. With the availability of low-cost 3D sensors, detection and classification is a major application in this field of research. In Chapter 6, we introduced a global point cloud descriptor (STPP) that uses a feature vector based on the birth-death pairing of the homology generators. From this work, we seek to determine if the combination of both geometrical and topological features can provide additional insight for describing 3D point cloud datasets.

7.2 Future Research Directions

These are remarkable times for the intersection of data science and applications in computer and robotic vision. Technology has matured to the point where it is economically viable to equip smartphones, robots, autonomous/connected vehicles, and other mobile devices and machines with an array of sensors. The rich datasets produced by these sensors are allowing devices and machines to perform a variety of impressive and challenging tasks, especially in the field of robotics. This trend will continue as interest in industry and government for this technology grows stronger. In the following, we list possible directions to pursue future research with the goal of developing practical and effective 3D point cloud processing methods.

7.2.1 Simplicial Complex Construction

Many of the recent simplicial complexes proposed for TDA are based on variations of the Delaunay complex. The Delaunay complex and its dual, the Voronoi diagram, have many useful properties in computational geometry. The complexity of Delaunay complex is $\mathcal{O}(N \log N)$ for dimension $d = 2$ and $\mathcal{O}(N^{\lceil d/2 \rceil})$ for $d \geq 3$ where N is the number of simplices. Therefore, the construction of the Delaunay complex is prohibitive in high dimensions. In low dimensions, developing capable algorithms for computing the complex for large numbers of simplices is a potential subject of our research.

Another avenue of research in simplicial complex construction is the use of reduction techniques. In this approach, heuristics are used to reduce the size of filtered complex while leaving the homology unchanged. One way of performing this reduction to filtrations of simplicial complexes is based on discrete Morse theory [81]. The algorithm developed in [82] makes use of this idea by computing a partial matching of simplices in a filtered simplicial complex such that (i) only simplices that enter the filtration at the same time are paired, (ii) the homology is determined by unpaired simplices, and (iii) paired simplices can be removed from the filtered complex without affecting the overall persistent homology. Unfortunately, heuristics must be relied upon to find partial matchings that reduce the complex size since finding optimal partial matchings was shown to be NP-hard [83].

Another heuristic, the tidy-set method, reduces the size of clique complexes (e.g. the VR complex) [84]. The tidy-set method skips the construction of the clique complex by extracting a minimal representation of the graph using maximal cliques thus requiring less memory. The method cannot be extended to filtered complexes, however it can be used for computing zigzag persistent homology.

7.2.2 Supervoxel Segmentation

Supervoxel segmentation is the unsupervised over-segmentation of a point cloud into regions of perceptually similar voxels. Its counterpart, superpixel segmentation, is a widely used preprocessing step in 2D image segmentation algorithms. Supervoxels can reduce the number of regions that must be considered later by more computationally expensive algorithms with a minimal loss of information. We are working on extending our topologically persistent region segmentation method to this more restricted form of segmentation.

7.2.3 Morphological Operations

Having established a method for hole boundary point detection, we can now begin to expand into the area of 3D morphological operations. Mathematical morphology is a theory and technique used for describing and manipulating shapes using set theory. The basic morphological operations include dilation, erosion, opening, closing, thinning, thickening, and more. In 2D image processing, morphological operations are used for filtering and noise removal. In this area, we are working on the creation of 3D morphological operations that incorporate a multiresolution approach using persistent homology.

7.2.4 Topological Descriptors

We've seen how topological features can provide additional information over geometrical-based descriptors. Going forward, our goal is to combine the differentiating power of geometry with the classifying power of topology. We are exploring the combination of geometrical and topological features at the local level for improved predicative performance on classification tasks.

7.3 Concluding Remarks

We started this thesis by noting the rapid increase in 3D sensor datasets brought about by new technologies. We end with the optimism that this new data modality will enable these technologies to make greater progress in solving real-world problems. In this dissertation, we explored the potential of algebraic topological methods for 3D point cloud processing. It is our hope that engineers and researchers will continue this exploration in developing and applying topological data analysis to extract meaningful information from point cloud data.

References

[1] Andreas Richtsfeld. The object segmentation database (OSD), 2012.

[2] Arjun Singh, James Sha, Karthik S Narayan, Tudor Achim, and Pieter Abbeel. Bigbird: A large-scale 3D database of object instances. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 509–516, 2014.

[3] Kevin Lai, Liefeng Bo, Xiaofeng Ren, and Dieter Fox. A large-scale hierarchical multi-view rgb-d object dataset. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1817–1824. IEEE, 2011.

[4] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[5] Herbert Edelsbrunner and John Harer. Persistent homology-a survey. *Contemporary mathematics*, 453:257–282, 2008.

[6] Michele d'Amico, Patrizio Frosini, and Claudia Landi. Optimal matching between reduced size functions. *DISMI, Univ. di Modena e Reggio Emilia, Italy, Technical report 35*, 2003.

[7] Francesca Cagliari, Massimo Ferri, and Paola Pozzi. Size functions from a categorical viewpoint. *Acta Applicandae Mathematica*, 67(3):225–235, 2001.

[8] Patrizio Frosini and Claudia Landi. Size theory as a topological tool for computer vision. *Pattern Recognition and Image Analysis*, 9(4):596–603, 1999.

[9] Vanessa Robins. Towards computing homology from finite approximations. In *Topology proceedings*, volume 24, pages 503–532, 1999.

[10] Herbert Edelsbrunner, David Kirkpatrick, and Raimund Seidel. On the shape of a set of points in the plane. *IEEE Transactions on information theory*, 29(4):551–559, 1983.

[11] Herbert Edelsbrunner and Ernst P Mücke. Three-dimensional alpha shapes. *ACM Transactions on Graphics (TOG)*, 13(1):43–72, 1994.

[12] Cecil Jose A Delfinado and Herbert Edelsbrunner. An incremental algorithm for betti numbers of simplicial complexes on the 3-sphere. *Computer Aided Geometric Design*, 12(7):771–784, 1995.

[13] Vin De Silva and Robert Ghrist. Coordinate-free coverage in sensor networks with controlled boundaries via homology. *The International Journal of Robotics Research*, 25(12):1205–1222, 2006.

[14] Vin De Silva and Robert Ghrist. Coverage in sensor networks via persistent homology. *Algebraic & Geometric Topology*, 7(1):339–358, 2007.

[15] Vin De Silva and Robert Ghrist. Homological sensor networks. *Notices of the American mathematical society*, 54(1), 2007.

[16] Jennifer Gamble, Harish Chintakunta, and Hamid Krim. Coordinate-free quantification of coverage in dynamic sensor networks. *Signal Processing*, 114:1–18, 2015.

[17] Afra J Zomorodian. *Topology for computing*, volume 16. Cambridge university press, 2005.

[18] Anders Björner. Topological methods. *Handbook of combinatorics*, 2:1819–1872, 1995.

[19] Allen Hatcher. *Algebraic topology.* Cambridge University Press, 2002.

[20] Afra Zomorodian. Fast construction of the Vietoris-Rips complex. *Computers & Graphics*, 34(3):263–271, 2010.

[21] Nina Otter, Mason A Porter, Ulrike Tillmann, Peter Grindrod, and Heather A Harrington. A roadmap for the computation of persistent homology. *EPJ Data Science*, 6(1):17, 2017.

[22] Gregory Henselman and Robert Ghrist. Matroid filtrations and computational persistent homology. *arXiv preprint arXiv:1606.00199*, 2016.

[23] Ulrich Bauer. Ripser: a lean C++ code for the computation of vietorisrips persistence barcodes, 2016.

[24] Guy E Blelloch. Prefix sums and their applications. 1990.

[25] Jose Luis Blanco and Pranjal Kumar Rai. nanoflann: a C++ header-only fork of FLANN, a library for nearest neighbor (NN) with kd-trees, 2014.

[26] Ariel Shamir. Segmentation and shape extraction of 3D boundary meshes. *Proceedings Eurographics State-of-the-Art Report*, 2006:137–49, 2006.

[27] Alan P Mangan and Ross T Whitaker. Partitioning 3D surface meshes using watershed segmentation. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):308–321, 1999.

[28] Shymon Shlafman, Ayellet Tal, and Sagi Katz. Metamorphosis of polyhedral surfaces using decomposition. In *Computer Graphics Forum*, volume 21, pages 219–228. Wiley Online Library, 2002.

[29] Michael Garland, Andrew Willmott, and Paul S Heckbert. Hierarchical face clustering on polygonal surfaces. In *Proceedings of the 2001 Symposium on Interactive 3D graphics*, pages 49–58. ACM, 2001.

[30] Rong Liu and Hao Zhang. Segmentation of 3D meshes through spectral clustering. In *Proceedings of the 12th Pacific Conference on Computer Graphics and Applications*, pages 298–305, 2004.

[31] Aleksey Golovinskiy and Thomas Funkhouser. Min-cut based segmentation of point clouds. In *IEEE 12th International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 39–46, 2009.

[32] Matthew Johnson-Roberson, Jeannette Bohg, Mårten Björkman, and Danica Kragic. Attention-based active 3D point cloud segmentation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1165–1170, 2010.

[33] Bertrand Douillard, James Underwood, Noah Kuntz, Vsevolod Vlaskine, Alastair Quadros, Peter Morton, and Alon Frenkel. On the segmentation of 3D lidar point clouds. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2798–2805, 2011.

[34] Andreas Richtsfeld, Thomas Mörwald, Johann Prankl, Michael Zillich, and Markus Vincze. Segmentation of unknown objects in indoor environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4791–4796, 2012.

[35] Ajay K Mishra, Ashish Shrivastava, and Yiannis Aloimonos. Segmenting "simple" objects using RGB-D. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4406–4413, 2012.

[36] Alexander JB Trevor, Suat Gedikli, Radu B Rusu, and Henrik I Christensen. Efficient organized point cloud segmentation with connected components. *Semantic Perception Mapping and Exploration (SPME)*, 2013.

[37] Sylvain Paris and Frédo Durand. A topological approach to hierarchical segmentation using mean shift. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–8, 2007.

[38] Chao Chen, Daniel Freedman, and Christoph H Lampert. Enforcing topological constraints in random field image segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2089–2096, 2011.

[39] Robert Sedgewick and Kevin Wayne. *Algorithms, fourth edition.* Pearson Education, Inc., 2011.

[40] Steven W Zucker. Region growing: Childhood and adolescence. *Computer graphics and image processing*, 5(3):382–399, 1976.

[41] Jordi Freixenet, Xavier Muñoz, David Raba, Joan Martí, and Xavier Cufí. Yet another survey on image segmentation: Region and boundary information integration. In *European Conference on Computer Vision (ECCV)*, pages 408–422. Springer, 2002.

[42] Rolf Adams and Leanne Bischof. Seeded region growing. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 16(6):641–647, 1994.

[43] Zheng Lin, Jesse Jin, and Hugues Talbot. Unseeded region growing for 3D image segmentation. In *Selected papers from the Pan-Sydney workshop on Visualisation-Volume 2*, pages 31–37. Australian Computer Society, Inc., 2000.

[44] Chantal Revol-Muller, Francoise Peyrin, Yannick Carrillon, and Christophe Odet. Automated 3D region growing algorithm based on an assessment function. *Pattern Recognition Letters*, 23(1):137–150, 2002.

[45] R Kyle Justice, Ernest M Stokely, John S Strobel, Raymond E Ideker, and William M Smith. Medical image segmentation using 3D seeded region growing. In *Medical Imaging 1997*, pages 900–910. International Society for Optics and Photonics, 1997.

[46] Yuri Y Boykov and Marie-Pierre Jolly. Interactive graph cuts for optimal boundary & region segmentation of objects in nd images. In *IEEE International Conference on Computer Vision (ICCV)*, volume 1, pages 105–112, 2001.

[47] Regina Pohle and Klaus D Toennies. Segmentation of medical images using adaptive region growing. In *Medical Imaging 2001*, pages 1337–1346. International Society for Optics and Photonics, 2001.

[48] Alain Tremeau and Nathalie Borel. A region growing and merging algorithm to color segmentation. *Pattern recognition*, 30(7):1191–1203, 1997.

[49] Frederick Pauling, Mike Bosse, and Robert Zlot. Automatic segmentation of 3D laser point clouds by ellipsoidal region growing. In *Australasian Conference on Robotics and Automation (ACRA)*, 2009.

[50] Abdul Nurunnabi, David Belton, and Geoff West. Robust segmentation in laser scanning 3D point cloud data. In *International Conference on Digital Image Computing Techniques and Applications (DICTA)*, pages 1–8, 2012.

[51] Dirk Holz and Sven Behnke. Fast range image segmentation and smoothing using approximate surface reconstruction and region growing. In *Intelligent Autonomous Systems 12*, pages 61–73. Springer, 2013.

[52] David Letscher and Jason Fritts. Image segmentation using topological persistence. In *Computer Analysis of Images and Patterns*, pages 587–595. Springer, 2007.

[53] Herbert Edelsbrunner, David Letscher, and Afra Zomorodian. Topological persistence and simplification. *Discrete and Computational Geometry*, 28(4):511–533, 2002.

[54] Ulrich Bauer, Michael Kerber, Jan Reininghaus, and Hubert Wagner. Phat–persistent homology algorithms toolbox. *Journal of Symbolic Computation*, 78:76–90, 2017.

[55] Oleksiy Busaryev, Tamal K Dey, and Yusu Wang. Tracking a generator by persistence. *Discrete Mathematics, Algorithms and Applications*, 2(04):539–552, 2010.

[56] Xiaoyuan Guo, Jun Xiao, and Ying Wang. A survey on algorithms of hole filling in 3d surface reconstruction. *The Visual Computer*, pages 1–11, 2016.

[57] Jianning Wang and Manuel M Oliveira. A hole-filling strategy for reconstruction of smooth surfaces in range images. In *Computer Graphics and Image Processing, 2003. SIBGRAPI 2003. XVI Brazilian Symposium on*, pages 11–18. IEEE, 2003.

[58] John Branch, Flavio Prieto, and Pierre Boulanger. Automatic hole-filling of triangular meshes using local radial basis function. In *3D Data Processing, Visualization, and Transmission, Third International Symposium on*, pages 727–734. IEEE, 2006.

[59] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Symposium on Geometry Processing*. The Eurographics Association, 2006.

[60] Minhyuk Sung, Vladimir G Kim, Roland Angst, and Leonidas Guibas. Data-driven structural priors for shape completion. *ACM Transactions on Graphics (TOG)*, 34(6):175, 2015.

[61] Gerhard H Bendels, Ruwen Schnabel, and Rheinhard Klein. Detecting holes in point set surfaces. *Journal of WSCG*, 14, 2006.

[62] Xiaoguang Han, Zhen Li, Haibin Huang, Evangelos Kalogerakis, and Yizhou Yu. High-resolution shape completion using deep neural networks for global structure and local geometry inference. In *IEEE International Conference on Computer Vision (ICCV)*, 2017.

[63] Herbert Edelsbrunner and John Harer. *Computational topology: An introduction.* American Mathematical Society, 2010.

[64] Chao Chen and Michael Kerber. Persistent homology computation with a twist. In *Proceedings 27th European Workshop on Computational Geometry*, volume 11, 2011.

[65] Ulrich Bauer, Michael Kerber, and Jan Reininghaus. Clear and compress: Computing persistent homology in chunks. In *Topological methods in data analysis and visualization III*, pages 103–117. Springer, 2014.

[66] Radu Bogdan Rusu, Andreas Holzbach, Michael Beetz, and Gary Bradski. Detecting and segmenting objects for mobile manipulation. In *IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 47–54, 2009.

[67] Radu Bogdan Rusu, Gary Bradski, Romain Thibaux, and John Hsu. Fast 3D recognition and pose using the viewpoint feature histogram. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2155–2162, 2010.

[68] Zoltan-Csaba Marton, Dejan Pangercic, Radu Bogdan Rusu, Andreas Holzbach, and Michael Beetz. Hierarchical object geometric categorization and appearance classification for mobile manipulation. In *IEEE-RAS International Conference on Humanoid Robots*, pages 365–370, 2010.

[69] Zoltan-Csaba Marton, Dejan Pangercic, Nico Blodow, Jonathan Kleinehellefort, and Michael Beetz. General 3D modelling of novel objects from a single view. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3700–3705, 2010.

[70] Aitor Aldoma, Markus Vincze, Nico Blodow, David Gossow, Suat Gedikli, Radu Bogdan Rusu, and Gary Bradski. CAD-model recognition and 6DOF pose estimation using 3D cues. In *IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 585–592, 2011.

[71] Aitor Aldoma, Federico Tombari, Radu Bogdan Rusu, and Markus Vincze. OUR-CVFH–oriented, unique and repeatable clustered viewpoint feature histogram for object recognition and 6DOF pose estimation. In *Joint DAGM (German Association for Pattern Recognition) and OAGM Symposium*, pages 113–122. Springer, 2012.

[72] Walter Wohlkinger and Markus Vincze. Ensemble of shape functions for 3D object classification. In *IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 2987–2992, 2011.

[73] Asako Kanezaki, Zoltan-Csaba Marton, Dejan Pangercic, Tatsuya Harada, Yasuo Kuniyoshi, and Michael Beetz. Voxelized shape and color histograms for RGB-D. In *IROS Workshop on Active Semantic Perception*, 2011.

[74] Duc Fehr, William J Beksi, Dimitris Zermas, and Nikolaos Papanikolopoulos. Covariance based point cloud descriptors for object detection and recognition. *Computer Vision and Image Understanding*, 142:80–93, 2016.

[75] Gunnar Carlsson, Afra Zomorodian, Anne Collins, and Leonidas J Guibas. Persistence barcodes for shapes. *International Journal of Shape Modeling*, 11(02):149–187, 2005.

[76] Chunyuan Li, Maks Ovsjanikov, and Frederic Chazal. Persistence-based structural recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1995–2002, 2014.

[77] Jennifer Gamble and Giseon Heo. Exploring uses of persistent homology for statistical analysis of landmark-based shape data. *Journal of Multivariate Analysis*, 101(9):2184–2199, 2010.

[78] Aaron Adcock, Daniel Rubin, and Gunnar Carlsson. Classification of hepatic lesions using the matching metric. *Computer vision and image understanding*, 121:36–42, 2014.

[79] Chen Gu, Leonidas Guibas, and Michael Kerber. Topology-driven trajectory synthesis with an example on retinal cell motions. In *International Workshop on Algorithms in Bioinformatics*, pages 326–339. Springer, 2014.

[80] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27, 2011.

[81] Robin Forman. Morse theory for cell complexes. *Topology*, 37(5):945–979, 1998.

[82] Konstantin Mischaikow and Vidit Nanda. Morse theory for filtrations and efficient computation of persistent homology. *Discrete & Computational Geometry*, 50(2):330–353, 2013.

[83] Michael Joswig and Marc E Pfetsch. Computing optimal morse matchings. *SIAM Journal on Discrete Mathematics*, 20(1):11–25, 2006.

[84] Afra Zomorodian. The tidy set: a minimal simplicial set for computing homology of clique complexes. In *Proceedings of the twenty-sixth annual symposium on Computational geometry*, pages 257–266. ACM, 2010.